CODE OPTIMIZATION IN `C++11`

Contents

Chapter 1

Raison d'être[1]

1.1 Code as recipes, code as inspiration, and the joy of puzzles as a defense against creeping nihilism

The exponential rise of computing power has been unprecedented in human history: there has been no other advancement so significant and brief, whether in food production, gasoline efficiency of cars, or survival rates from medical procedures. This tremendous shift has left programming as the *lingua franca*[2] of our age, the universal language that joins all fields of study in our era. Whether one works in social sciences, statistics, or literature, the word "algorithm" is increasingly prevalent (*e.g.*, in analyzing human behavior through computer-monitored reaction experiments, estimating posterior distributions through MCMC, or analyzing word usage patterns to determine authorship, respectively).

However, in spite of this shift toward computer science, the increasing interest in our computational world is increasingly focused on a superficial flavor of software engineering, *i.e.*, grabbing up existing packages and code snippets from online search engines and applying them to data. The ease with which software is copied and transferred through the internet has greatly facilitated cross-pollination of ideas, but it has also yielded an apparent game-

[1]French: "reason to be", literally the reason for existing

[2]A Latin expression referring to French as the common language of the era, here applied to `C` and `C++`; in other words, clear as mud.

theoretic advantage to this superficial approach to computation: if only one person in the world need solve something for the rest of us to benefit, then we might all decide to sit back and wait for someone else to solve whatever puzzle we're working on[3].

Thus, although the the democratization of computing has certainly produced greater exposure to computers, it could be argued that in many ways it has actually led to a more shallow appreciation for the depth and beauty of the magic tricks that make faster, bigger, and better computing possible; the underlying algorithms and techniques are seen as commodities, things to be invented once and then shared, and even scientists increasingly see themselves as consumers of tools rather than creators. This has corresponded with a rise in fatalism in computing, an idea that not only will someone else deliver us, but that it is even *pointless* for us to try, that of course there is something better out there already or that even if we created something new, that nobody would care or else someone else would surely unseat us before long. This creeping nihilism[4] has penetrated deep into the collective unconscious of modern computer science, permeating even the elite scientists in industry and academics: even highly qualified experts on online question-and-answer platforms routinely answer with deflections, asking why you might want to do something challenging or suggesting an existing software package[5].

While caution does represent good engineering practice (invent an well-tested, interchangeable widget once, and then use it again and again), it does nothing to help us invent something the first time[6]. When we think of

[3]In economics, this paradox is called "the tragedy of the commons": If each family is able to graze one sheep per day on the "commons" (the grassy area shared by those in the village), the grass will not be trampled into mud and every family will feed enough livestock to survive. However, as a rational individual, you may feel convinced that, because all other families follow the one sheep per day limit, you alone could graze *two* sheep per day. Adding in one more sheep per day would likely not damage the grass but would be enough to substantially improve your family's production. But every family may observe this symmetrically, and so each will bring two sheep per day, which will trample the grass, leaving the entire village, you included, without feed for your livestock.

[4]From the Latin "nihil" (nothing), so nihilism is literally "nothingism", it is essentially an attitude that nothing has meaning and that everything is valueless.

[5]"Have you ever heard of `boost`? Why don't you just use `boost`? `boost boost`, `boost boost boost`..."

[6]Tools can be quite useful, but do not simply copy everything without ever bothering to learn or try yourself. Shameless copying is how karaoke tragedies like Star Wars Episode 7 get made[7].

[7]"Did I ever tell you of the tragedy of Episode 7?"

inventing something for the first time, we often think of the things that have already been done, about classic algorithms and their creators, some quaint event from a sepia-drenched time before. But we go through the same process of *ex nihilo*[8] creation whenever we decide to try to go further than what is known today[9].

Learning to invent something the first time is a tricky business; invention is not about memorization, it is about creativity, about immersing ourselves in a problem and seeing what we can come up with. It is disciplined but playful. It is also clear why even experts can be wary of *terra incognita*[10]: all the accumulated knowledge of an expert may prove useless against a new puzzle[11]. Furthermore, if you are primarily motivated by external validation, why work hard to diminish your admiring audience as as you climb to more and more rarefied heights? The answer is clear: because everything worthwhile has always happened in some frontier or other.

This book can be seen as a collection of recipes for improving code performance. But I would prefer it be seen as something more, as a collection of examples with which we can hone our adventurous, frontier spirits. The examples are also not meant to be the final word on the best or fastest ways to solve each puzzle presented; instead, they only pull us down the rabbit hole and give us a brief look around. It is not the recipes here that matter, it is the rabbit hole itself. It is not about a particular solution as much as it's about the joy of the game, the fun in wandering around that precious frontier that goes on forever.

You cannot look up tomorrow's revolutionary ideas on Wikipedia. Someone needs to create them. Maybe you.

[8] "From nothing", *i.e.*, working from scratch

[9] Legend holds that the pillars at the Straight of Gibraltar were once inscribed with "NON PLUS ULTRA", meaning "nothing further beyond", a landmark indicating the boundary of the world, beyond which was assumed to only be dangerous ocean and the monsters beneath. The Holy Roman Emperor and King of Spain Charles V, during whose rule colonies in the Americas were brought under unified control, adopted the opposing motto "Plus Ultra".

[10] Latin: "unknown lands"

[11] As science fiction writer Arthur C. Clarke once put it: "If an elderly but distinguished scientist says that something is possible, he is almost certainly right; but if he says that it is impossible, he is very probably wrong." For most of human history, a sub-5-minute mile was unthinkable, and now it is routine for elite athletes.

1.2 The scientist as an alpinist

Ask yourself this: Why do people climb mountains? Why do they leave their safe, warm beds and deliberately seek out places that are cold, dangerous, remote, and without help? There are few extremely rich mountain climbers, so it cannot simply be for pursuit of material wealth. And even though climbers may be recorded in the history books, they are less famous than politicians, musicians, entrepreneurs, or football stars.

Instead, alpinists are motivated by the challenge itself, by the fact that no other person has stood on a particular peak or climbed a particular wall. They are focused on the struggle trying to reach it, by the fact that, in spite of what we may think, not everything has been done; there are frontiers everywhere if we look closely and, if we are lucky, we might be the first and last to touch a particular frontier[12]. Like the great mountaineers, explorers, artists, and others who have come before us, we are motivated by curiosity and a thirst for adventure and novelty[13].

The motivation driving scientists is essentially the same. We aren't driven solely by pursuit of money[14], degrees, titles, and advancement, but by the thrill of finding or building something new– however small– that has never been seen by any person in the history of human existence, to stand on that mountain for the first time. That joy of discovery is one of the very few things that cannot be bought and sold, and once you have a piece of it, it is yours forever[16]. You do not have to be seen climbing a mountain or creating a beautiful piece of code to be able to appreciate the view.

[12]By definition, once it has been reached, it is no longer pure frontier.

[13]Van Gogh was a commercial failure as an artist. Mozart died poor and was buried in a mass grave. Grigori Perelman ultimately rejected the adulation of his peers and did not go collect his $1 million prize.

[14]This is not to say that you should take your finances for granted nor deliberately starve yourself (a well-fed computer scientist is a productive computer scientist!). But this does not mean everything needs to be monetized[15]. In our modern material society, simply take some time to reflect on what Socrates said about virtue and money: "Virtue does not come from wealth, but wealth, and every other good thing which men have comes from virtue."

[15]Khaaaan [Academyyyyy]!

[16]And even if you are not the first up a particular mountain, admiring the view and breathing the thin air is still an amazing feeling.

1.3 Top-down learning

This book is constructed in a "top-down" style[17]: we will start with a problem, construct the simplest naive solution, and then iteratively cut away at it until we arrive at a very efficient implementation.

There are many reasons for this approach: 1, it is more motivating to start by knowing why you would like to solve a problem. 2, this book exists to help you learn how to solve problems, how to climb mountains, rather showing you a photograph of the view from the top. And 3, it is designed so that everyone can participate. It is withholding to simply show someone the mountain top without any discussion of how to climb (this is reminiscent of scientists discussing in jargon rather than starting in intuitive terms– it discourages those who are not already familiar with the subject at hand). This is, unfortunately, ubiquitous in modern science[18]. This is an important lesson for you once you develop your talents and become stronger scientists. There is a nice challenge in itself to empathizing with those who do not yet understand and guiding them up the mountain[19]. We will climb together as a team, and we will be able to climb higher because we do.

Questions

1. [**Level 1**] How are scientists like adventurers?

2. [**Level 2**] How does the tragedy of the commons contribute to nihilistic culture in computer science?

3. [**Level 3**] Read (`http://www.newyorker.com/archive/2006/04/10/060410fi_fiction`) or listen to (`http://downloads.newyorker.com/mp3/fiction/101217_fiction_ozick.mp3`) "In the Reign of Harad

[17]This resembles the computer science notion of "lazy" computation.

[18]The average scientific publication is opaque and can be difficult to understand even by experts in a nearby field.

[19]There was a study done where they showed a young child a box of pencils, emptied out the pencils, and then filled the box with candy and re-closed. Then another child is invited into the room, and they asked the first first child what the other child thinks is in the box. Even though the box now has candy and *not* pencils in it, it is the cleverest children who answer, "Pencils" because they know enough be able to empathize with the ignorance of the other child.

IV" by Steven Millhauser. How does the story relate to the struggles and rewards of scientific research? Who are some scientists whose lives parallel that of the maker of miniatures?

Chapter 2

Why efficiency matters

2.1 Do we live in a post-optimization age?

Clearly we are interested in the challenge of solving problems faster by using more clever tricks (why else would we be writing / reading this book?); however, let's start by challenging that worldview. Perhaps everything is already "fast enough". It is certainly not an unpopular idea that things are now fast enough, that we have good enough compilers, that we have fast hardware, *etc.* `Python`, `R`, and other high-level languages are often fast enough and are quite easy to use, and that obsessing over efficiency in languages like `C` is an anachronism, something to be relegated to the fluorescent lighting of a basement that smells of pizza boxes and vaporized solder. So let us begin by seriously considering that argument: do we really live in a post-optimization age[1]?

First, let's consider arguments as to why optimization is no longer relevant. One reasonable point in this vein is that hardware, specifically CPUs, continue to get faster and faster. Decades of "Moore's law"[2] have left us spoiled, insouciantly anticipating further exponential growth. Faster hardware, it could be argued, would render obsessively optimized code unnecessary; rather than obsessing about performance now in order to reduce runtime, one could simply upgrade to better hardware and achieve the same

[1]People increasingly chatter about living in a post-[anything] age, so I suppose it was only a matter of time before that "anything" grew to include optimization as well.

[2]The roughly true conjecture that the number of transistors embedded on a CPU will grow exponentially with time

decrease in runtime[3].

There are several holes in the argument that faster hardware makes code optimization obsolete. A first counterpoint is that humans are naturally competitive[4], and so the availability of higher-performance hardware will not confer an advantage to us alone, but also to those competing with us. A second counterpoint is the fact that Moore's law has been slowing. As engineers have crammed more and more transistors onto a CPU, the physical dimensions of the processor have themselves become an issue. On one hand, keeping the processor size the same and using smaller transistors shrinks the size of insulators, permitting greater parasitic capacitance and even quantum tunneling between disconnected wires. On the other hand, physically larger processors experience delays while transmitting information because modern clock speeds (*e.g.*, in the GHz range) can oscillate many times before the speed of light is able to propagate a short distance. While people have been predicting the end of Moore's law for a few years now, the advances that have extended its life come at greater complexity and greater cost, and so it is not likely to continue in the same manner. This can already seen with chip makers moving to multicore models rather than focusing on improved clock speed. Multicore processors simplify some issues of the propagation delay compared to those that would attempt use a single core and a faster clock speed.

Another reasonable point as to why optimization would no longer be relevant is because we already solve most useful tasks fast enough[5].

For instance, the world would keep spinning if a web page loads in 0.5 seconds rather than 0.1 seconds, and when we use a spreadsheet, perhaps we won't really notice the difference between a computation taking 0.1 seconds or taking 0.01 seconds (even though a $10\times$ speedup due to our software imple-

[3]This is called "throwing hardware at the problem".

[4]In that we often compete for one another over resources that are finite or appear to be finite

[5]There is a famous quote from computer scientist Donald Knuth, which says "Premature optimization is the root of all evil." There is some wisdom in this– *e.g.*, do not try a fancy method if a simple method has not yet been tested– but the pregnant word "premature" is the distinguishing feature. In fact, Knuth's quote is in line with this book's top-down ideal, which focuses first on the problem and its simple solutions before progressing to more sophisticated solutions.

mentation would likely represent a quite significant achievement)[6]; however, our views of which tasks are obscure and which tasks are useful are themselves subconsciously limited by the tools that we have available. Tasks that would once be labeled obscure are now thought of as quotidian[8]. Through that lens, we can see that our demands as users tend to expand like a gas[9]. Users are constantly more demanding when it comes to speed, power efficiency (*i.e.*, battery life), and ability to perform more complex tasks effectively[11].

[6]Henry David Thoreau once said that "We are in a great haste to construct a magnetic telegraph from Maine to Texas; but Main and Texas, it may be, have nothing important to communicate." To non-scientists, an overzealous interest in speedups may feel something like this.[7]

[7]You will, on occasion, even meet academics who, between pint glasses of what is no doubt boxed wine, enjoy explaining that none of this *really* matters, that hardware design, programming language choice, software optimization, and plenty of other things don't matter, and yadda yadda yadda that humanity would do perfectly fine if we established a barter economy and went back to living in straw huts– and have you been to burning man by the way? because at *burning man*, they have. . . ; however, when you ask, "Hey, can I borrow your SR-71 supersonic jet, not the titanium-aluminum-vanadium alloy one, but the one made out of wood and straw, can I borrow it?" they will without question try to say that they left it in the pocket of another pair of pants and then try to quickly change the subject without you noticing. Surprising no one.

[8]*E.g.*, encryption is used in online shopping, real-time compression is used for video chat, and voice recognition is used to track our identity when calling customer service.

[9]There is a great albeit apocryphal story about Bill Gates claiming that 650K of usable RAM "ought to be enough for anybody"; however, Gates has since denied that this ever happened. And would a billionaire philanthropist who also won the 1973 Wimbledon tournament[10] *really* need to lie to impress us?

[10]Not really

[11]For a large online retailer, even small delays in their webpage may actually translate to millions of dollars in lost sales.[12]

[12]Once when I did some office work at a large investment bank during my early teenage years, I expressed interest in the portfolio management algorithms, and was dismissed "There are a bunch of egg heads who worry about that stuff". I was advised to focus my attention on dressing well, and I stopped going shortly after. During the 2008 financial contraction, that investment bank disappeared in a matter of days. If you don't obsess about your job, someone else will. As William S. Burroughs put it, "This is a war universe. War all the time. That is its nature. There may be other universes based on all sorts of principles, but ours seems to be based on war and games."

2.2 When does efficiency matter?

There are several reasons to avidly pursue faster code, cases where our appetite for performance is infinite and where payoffs for even moderate improvements are real:

> **Scale advantage:** Some problems cannot be done halfway. For example, internet search cannot be adequately implemented until you cross a threshold of storing data from a large percentage of the internet (ideally, all of it). A substantial speedup or memory savings can easily mean the difference between being able to solve the problem and not[13].
>
> **Speed advantage:** In settings like high-throughput trading[14] or game engine design, a small speedup can mean a large, very valuable difference between you and a competitor.
>
> **Complexity advantage:** Faster runtimes not only translate to better automated trading speeds in investing or better frame rates in video games; instead, it is often advantageous to exchange that runtime improvement for a greater model complexity (*e.g.*, better prediction algorithms in trading or a higher-quality, more life-like world in gaming). Achieving a higher-quality result with the same runtime as a competitor can offer a large advantage. For example, in cryptanalysis, the difference in cracking an encrypted file in 1s (seconds) second rather than in 10s may not be very useful, but the ability to crack the file rather than not crack the file is everything[15].

These are only a few categories and a few examples of where the quest for efficiency is not only still relevant, but especially important today. As

[13]Implementing search on only a minuscule fraction of the internet or implementing a social network that supports only a handful of users are both generally as useless as having a fraction of the ingredients necessary to bake a cake.

[14]High-throughput trading is where investors (usually large firms) try to use a very small edge in market information to buy or sell shares shortly before others. For example, confirmed news stories about the death of a world leader can invite uncertainty and quickly depress markets, and the first actor to sell shares and then re-buy them after the price drops could earn a large profit while still owning all of their original shares... assuming they correctly predicted the market.

[15]For tasks like cryptography and cryptanalysis, complexity advantage is closely intertwined with scale advantage, demonstrating the fuzzy boundary between the categories itemized here.

robotics and automation continue to proliferate in factories, warfare, commerce, *etc.*, those who can most efficiently instruct the automated creations will own the future.

But this quest for performance is not only competitive. The methods used to speed up code are often pure puzzles that are interesting in their own right. It is interesting to dig things open, to see how they work, and to think about how we would design them differently (and the pros and cons of doing so). These puzzles would fit just as well in a competitive military scenario (*e.g.*, trying to break an opponents encryption before they break yours) as they would fit a pure scientist working alone for education or enjoyment[16]. It is an art.

2.3 All languages are not created equal[17]

Programming languages each have their advantages and disadvantages relative to others. Because they are very accessible, languages like `Python` are often the most comfortable for new programmers[18], and deviations into `C/C++` may be seen as excessive or even pedantic. For this reason, we should genuinely ask ourselves, "Is it really worth using `C/C++` anymore?" We will directly test this in our first benchmark.

2.4 Measuring performance

There are many ways to measure performance: the big-oh[20] runtime of an algorithm; total number of operations required by an implementation of an algorithm; the total number of multiply operations required by an algorithm (in contrast with operations that have historically been cheaper, like addition, bitwise and `&` and bitwise or `|`); the FLOPS (floating point operations per second); *etc.*. Each of these measurement criteria has its pros and cons,

[16]Like a Swiss watchmaker working in a small alpine city and honing their craft

[17]"Some languages are more equal than others…"

[18]Or perhaps their first programming language, also known as their "mother tongue" or "Muttersprache" in German[19]

[19]Deutsch ist nicht meine Muttersprache.

[20]Also known as Landau notation.[21]

[21]"Landau system?!" "Landau isn't a system. He's a man."

but for our purposes, one measure reigns supreme: the real runtime of the implementation[22].

The reason why is simple: empirical runtime directly answers the question "How fast can you do this?"[24]

There are different measures of empirical runtime, most notably "real" runtime (also called "stopwatch" time or "wall clock" time) and "user" runtime. Real runtime may include things like operating system overhead, while user time measures the actual time the executable is occupying the CPU. While this precision makes user time appealing, there are also two problems this: The first problem with user time is the notion that because operations like frequently allocating large blocks of memory (which can be costly for the operating system if a contiguous block of memory is not available) may be costly in practice, then that should be included in our evaluation (even if some of the delay is caused by the operating system). The second problem with user time is its response to parallelism: if you use multithreading or

[22]We will focus only on what we can directly measure. "If it is 'truth', rather than measurable 'fact', that you're interested in, Dr. Tyree's Philosophy class is right down the hall."[23]

[23]Do you understand this reference? Well then you could explain it to the other students... *If only you spoke Hovitos...*

[24]Does this mean that things like big-oh runtime are useless? Not at all. But it does cast some light on the distance between a good big-oh runtime and fast performance in practice. For small problems, an algorithm $\in O(n^2)$ with a low runtime constant may outperform an $O(n \log(n))$ algorithm with a moderate runtime constant, showing the advantage of the empirical performance over the big-oh runtime: "constants can kill you". But the distance between the big-oh runtime and the empirical real execution time also has benefits for the big-oh runtime: real execution time can be highly dependent on the implementation, the compiler, and the architecture that its run on. Operations that are fast on one CPU may be slow on another. This is one of the great advantages of big-oh runtime: it is much more agnostic as to which architecture you are running on[25]. Big-oh runtime (on the more abstract extreme) and real execution time (on the practical extreme) offer complementary types of information: if I were at the planning phase of a project, I would generally start with algorithms that have a good big-oh runtime. But at the end of the project, real execution time is the thing that will really impress other people, especially those who are not computer scientists.

[25]When you allow for very different architectures (*i.e.*, not only Motorolla 68000 vs. Intel x86-64, but also different computation models, *e.g.*, the "random access computation model", the "uniform decision tree computation model", or the non-deterministic Turing machine), solving the same problem on those architectures can change even the big-oh computational complexities dramatically. Most modern computers closely resemble the random access computation model, where memory is like a big array that can be accessed by its integer indices.

the GPU, then the algorithm can be distributed over multiple cores, and in some cases, can receive a considerable speedup in the real runtime; however, the user time will count the total time that the process receives attention, meaning that two cores running perfectly in parallel (with no overhead) will result in a real time of x but a user time of $2x$.

Although UNIX and Linux have built-in commands for measuring the runtime of an executable, we would prefer to precisely target the code that we're interested in. For this reason, we will create a `Clock` class in `C++`, which we can use like a stopwatch to measure elapsed time (Listing 2.1).

Using this class, we will compare the runtime of a simple task in `Python` and `C++`. For this first benchmark, we will use a basic `for` loop to sum the integers $0 + 1 + 2 + \cdots + 2^{24} - 1$. In `Python` we will use the `time.time()` function (Listing 2.2), and in `C++` we will use the `Clock` class to measure elapsed time (Listing 2.3).

We will start by turning off other programs (*e.g.*, the music player, the video player, and the web browser[26]) and running each piece of benchmark code and recording its runtime. It is important to run only on benchmark at a time; running them simultaneously will force them to compete for resources and will have undesirable effects on memory and cache performance.

When we run our python code with `python [filename.py]`. It took roughly 2.151s (we will round to four significant figures). The `C++`, compiled with `g++ -std=c++11 [filename.cpp]`[27] takes 0.06690s. This is a stark difference in the runtimes! By typing `echo "2.151 / 0.0669" | bc -l`, we can see that this is a $> 30\times$ speedup.

However, the `C++` can be made even faster still by using compiler optimizations. We will compile again, this time using `g++ -std=c++11 -O3 [filename.cpp]`. The `-O3` means we are using the third-level optimizations; `-O`, `-O2` would perform similar optimizations, but a subset of those we will have available from `-O3`. Compiler optimizations are of critical importance, and will be discussed in greater detail later.

[26]Because we now live in a time when it is common to stream music and videos 24/7, closing all other programs while benchmarking can feel like quite a heavy cost. This is one reason to have multiple computers, at least one for benchmarking (*e.g.*, a simple Linux server that you `ssh` into) and one for receiving your daily "infotainment" while your benchmarks are running.

[27]`-std=c++11` allows us to make use of the `C++11` language extensions. Some of these will be crucial to achieving better performance later on in this book.

Listing 2.1: Clock.hpp. The `Clock` class can be used to measure elapsed time. The member function `tick()` starts the timer, the function `tock()` returns the current elapsed time in seconds, and the function `ptock()` prints the current elapsed time in seconds. On construction, the timer is started. Because the start time is marked by simply writing to a `float`, the impact of this measurement on the actual runtime should be minor.

```
#ifndef _CLOCK_HPP
#define _CLOCK_HPP

#include <iostream>
#include <iomanip>
#include <chrono>

class Clock {
protected:
  std::chrono::steady_clock::time_point start_time;
public:
  Clock() {
    tick();
  }
  void tick() {
    start_time = std::chrono::steady_clock::now();
  }
  float tock() {
    std::chrono::steady_clock::time_point end_time =
        std::chrono::steady_clock::now();
    return
        float(std::chrono::duration_cast<std::chrono::microseconds>(end_time
        - start_time).count()) / 1e6f;
  }
  // Print elapsed time with newline
  void ptock() {
    float elapsed = tock();
    std::cout << "Took " << elapsed << " seconds" << std::endl;
  }
};

#endif
```

Listing 2.2: A `for` loop in `Python`. The loop sums the first 2^{24} integers.

```python
from time import time

N=2**24

t1=time()
tot = 0.0
for i in xrange(N):
  tot += i

t2=time()
print tot
print "Took", t2-t1
```

Listing 2.3: A `for` loop in `C++`. The loop sums the first 2^{24} integers.

```cpp
#include "../Clock.hpp"

const unsigned int N=1<<24;

int main() {
  Clock c;
  double tot=0.0;
  for (unsigned int i=0; i<N; ++i)
    tot += i;
  c.ptock();

  return 0;
}
```

2.5 Beware of dead code

When we rerun the `C++` benchmark with `-O3` optimizations, something curious happens: the program takes 0s to run! Now on one hand, this could be a good sign; it could mean that the `C++` code can simply be optimized to be so fast that our clock, which uses `float` time units, is unable to measure it without underflow[28]. But it could also be a sign that something is wrong[29]. Whenever our results look too good, we should be skeptical[30]. The `Python` program printed 1.40737479967e+14 as the result of the sum, so let's check and make sure that our `C++` program produces the same result (Listing 2.4). Note that our modified `C++` program prints the result `tot` *after* the runtime has been recorded. This is important, because printing to the screen can be surprisingly costly[31]. When we time Listing 2.4 again with `g++ -std=c++11 -O3`, we get a mysteriously different result: it prints the same numerical result as the `Python` code (roughly 1.40737e+14 with the default `std::cout` precision), but it now takes 0.02205s instead of the 0s we saw before; however, the 0s result was flawed (it wasn't actually computing the sum), and so this is our best result. It is a 3× speedup over the `g++` without `-O3` and a 97× speedup over `Python`.

The reason behind this discrepancy is not only interesting, it is sobering. One of the many optimizations performed by the `g++` compiler is "dead code elimination". Dead code elimination automatically detects code where the results are never used. For this reason, dead code can be safely trimmed away without changing any measurable results of the program. This can, as in this example, result in a benchmark that essentially does nothing (because everything has been marked as dead code by the optimizing compiler and removed). For this reason, we should be wary any time we see a runtime drop to 0s[32], because it may simply be the result of dead code elimination. Compilers are always improving, but we should be wary of anything that

[28] "Underflow" is the term for a type of numerical error where floating point values (*e.g.*, either `float` or `double`) are so close to zero that they can no longer store such a small value and so they simply become zero instead.

[29] Have we "delved too greedily and too deep" with our optimizations?

[30] Our minds are trained to see what we want to see, so we should do our best to continually swim against that current. This is true in both science and life.

[31] Printing to the screen can be quite useful, but printing too often is frequently derided by some purists as "operator entertainment" because it may clutter the program and be detrimental to runtime.

[32] As Bram Stoker, the author of Dracula, once wrote, "The dead travel fast."

Listing 2.4: Updated version of Listing 2.3. This updated version uses the variable `tot` after computing the sum; therefore, the computation of the sum will no longer be silenced by dead code elimination.

```
#include "../Clock.hpp"

const unsigned int N=1<<24;

int main() {
  Clock c;
  double tot=0.0;
  for (unsigned int i=0; i<N; ++i)
    tot += i;
  c.ptock();

  std::cout << tot << std::endl;

  return 0;
}
```

looks too good to be true.

2.6 Why Python is slower than C++

Python is a very flexible language (*e.g.*, without any special preparations, a function be called one time with a list as the argument, a second time with a class object as the argument, and a third time with a class type as the argument). But as we've observed, this flexibility comes at a price: less dynamic, compiled languages like `Fortran`, `C`, and `C++` can exploit their lack of flexibility to rewrite the code in an equivalent but faster form. The fact that these compiled languages are so intertwined with the hardware is precisely what makes them so useful for high-performance tasks; the `C` programming language has been referred to as "portable assembly"[33], and we can use this to write `C` and `C++` code that anticipate what kind of assembly will be created in order to improve performance. This is one of the reasons why `C++` was chosen for this book[34].

[33] *I.e.*, architecture agnostic assembly

[34] A professor, a smart and witty guy who was adjunct from IBM and taught for fun, once said that when you have one hand on the hardware and one hand on the software,

Within each iteration of the `for` loop, `Python` may need to look up the variable `tot` again (possibly even using the string "tot" to look it up)[35], while in `C++`, it will refer to a specific address in RAM, and in optimized `C++` code, we can count on the compiler to optimize it to refer to a single register on the computer[36]. For this reason, languages like `Python` can be slow on loops with many iterations and with simple tasks inside the loop; the language overhead starts to dwarf the cost of the actual work being done.

This is not just said to disparage `Python`; in fact, it is a great practice to implement anything complicated in organized `Python` to ensure it works before implementing in a language like `C++`[37]. Instead, looking into this performance difference between `Python` and `C++` sheds a light on the pros and cons of each language. Understanding the things that make `Python` slower than `C++` can help us to understand how to make faster `C++` code, whether through compiler optimizations or through our own design.

And consider: even if the `Python` code were just as fast, it would be going through an automated version of the same thought processes we just employed. If there ever is such a high-performance `Python` interpreter, we could simply use it without interest in how it works, but this would be unwise as well as boring. When possible, we should seek to be the one who knows how this better `Python` interpreter was created, and how we can write a better one. This is far more interesting than being someone who simply *uses* it, a means to be an artist rather than just a collector of finery[38].

---

you can get a better grip and make it harder for someone from the business end of things to pull you out of your job.

[35]Details like this will depend on the language used and the quality of the interpreter.

[36]By using two registers, one for the loop counter `i` and one for the total `tot`, it is possible to run the entire loop without ever reading from or writing to RAM. Reading from and writing to RAM would be considerably slower than registers to access.

[37]See Donald Knuth, we heard you!

[38]A student once told me that he didn't want to use `C++` because the proprietary Visual Studio compilers, on which he wrote `.NET` code, were superior because they were "managed code". "Managed code" was essentially a clever marketing term for proprietary, closed-source code without which the student could no longer work. If Microsoft had decided to charge $1,000 / year to use the `.NET` framework, this student would have been trapped. It is shocking when this kind of dependency so successfully infiltrates the hacker / DIY culture of computer science. As Chuck Palahniuk wrote in Fight Club, "Lord knows what they charged. It was beautiful. We were selling... their own fat @$&es back to them."

Questions

1. [**Level 1**] Come up with 3 applied settings not listed here why a speed-up matters.

2. [**Level 2**] Repeat 32 repetitions of the benchmarks in Listing 2.2 and 2.4 on your computer (using `-O3` to compile the `C++` code). Record the average runtimes. Now repeat these benchmarks again, but while streaming some videos (on several multiple tabs) on the Tor browser. How do the average runtimes change?

3. [**Level 3**] What happens to the `Python` runtime when you change the line `tot=0.0` to `tot=0`? Why? What happens to the `C++` runtime (optimized with `-O3`) when you change the line `double tot=0.0` to `unsigned long tot=0`? Can you think of any reasons why this would be the case?

Chapter 3

Benchmarking

3.1 Treating runtime as a stochastic process

As you have probably already observed, the runtimes you measure will vary slightly as you repeatedly benchmark your code. This is perfectly natural: other programs will be running (some more extraneous[1] and some more basic, like the operating system doing chores to the mouse and GUI running[2]), and they may stochastically make more or less demand on the hardware (*e.g.*, a garbage collector may stay silent for a long time and then suddenly require a lot of CPU time and memory in a short burst; a program benchmarked at the same time may look significantly slower).

For this reason, we usually treat benchmarks as a stochastic process[3]; therefore, it is customary to take repeated measurements of performance and compare the averages between two implementations. By taking averages from a larger and larger number of experimental replicates, more and more subtle differences can be detected. In this book, we will assume that 1024 replicates are sufficient.

You will also find that, if you have been running many memory-intensive programs that caused the operating system to use the swap space[4], even

[1]Did you remember to close your web browser before benchmarking?

[2]For real perfectionists, the best way to benchmark is to drop out of the `X` server and run in "single user" mode. This can actually increase performance, decrease variability between repeated benchmarks, and help you detect more subtle differences in performance.

[3]*i.e.*, a collection of measurements taken in the presence of random variability

[4]Disk space masquerading as slow RAM when the user tries to use more RAM than is available

Listing 3.1: Incorrect swapping of the values `x` and `y`. The value `x` is erased irrecoverably, and therefore is no longer available to set the new value of `y`.

```
void swap(int & x, int & y) {
  x = y;
  y = x; // Wrong: x was already set to y, so both
         // values will have the old value of y
}
```

after you have stopped these programs, your benchmarks may run slower. The same is sometimes true for CPU-intensive programs, some of which lazily fetch libraries and spawn processes when launched, but not terminate those processes after the program is closed[5]. For this reason, it can be good practice to perform a "cold"[6] boot before benchmarking.

3.2 The XOR swap trick

A classic riddle from computer science goes like this: you have binary values (*e.g.*, integers), which you would like to swap. That is, after swapping, the new value of `x` should be `y` and the new value of `y` should be `x`. A naive programmer may write the simple, incorrect code in Listing 3.1. An alternative would be to use temporary variables named `old_x` and `old_y` (Listing 3.2); however, this requires two new variables (which could require two registers on the processor, meaning that if other variables were already stored in registers, they may need to be written back to RAM so that two registers are now free). Also, the code in Listing 3.2 requires four copy operations. Now the question is, can we do better? And how few registers and operations do we need?

One way forward is clear: we can get away with using only a single temporary variable. Consider the incorrect swapping code from Listing 3.1; the value of `x` was irrecoverably lost, but not the value of `y`. So let's simply use a temporary value to save the value of `x`. Listing 3.3 does this, using only one temporary value and only three copy operations.

[5]One reason for doing this is that it can make it faster to launch the program the second time.

[6]*i.e.*, fresh

Listing 3.2: Correct swapping of the values `x` and `y` via two temporary variables.

```
void swap(int & x, int & y) {
  int old_x = x;
  int old_y = y;
  x = old_y;
  y = old_x;
}
```

Listing 3.3: Correct swapping of the values `x` and `y` via one temporary variable.

```
void swap(int & x, int & y) {
  int old_x = x;
  x = y;
  y = old_x;
}
```

Now we ask ourselves if we can do any better. If you're a waiter carrying two plates of food, linguine in your left hand and ravioli in your right, it feels intuitive that to swap them (so the ravioli is now in your left hand and the linguine is now in your right), you would need to sit one plate down temporarily (this is equivalent to the single temporary store we used in Listing 3.3). But with binary values, it turns out we can do better.

Consider two single-bit values, `a` and `b`. If someone told you only the value of `a` but not the value of `b`, it would not be possible to infer the value of `b`; it could freely be either value $\in \{0, 1\}$. But if someone also told you whether both `a`,`b` are true, both `a`,`b` are false, or whether exactly one of `a`,`b` is true, then you could figure out the value of `b`. For example, if you know `a=0` and you know that exactly one of `a`,`b` is true, then you know that `b=1` (*i.e.*, `b` is true).

The operation that tells you about both `a`,`b` in this way is called "XOR", an abbreviation of "exclusive or". In `C++` it is written as `a^b` and is equivalent to `(a || b) && !(a && b)`, meaning either `a` or `b` should be true, but not both. Amazingly, XOR does not only allow us to recover the value of `b` using the value of `a` and `a^b`; the method for recovering `b` is itself the XOR operation: `b == (a^b)^a`. Also, note that XOR is a symmetric operator, so

Listing 3.4: XOR swapping of two boolean values `x` and `y` with no temporary variables.

```
void swap(bool & x, bool & y) {
  x = x^y;
  y = y^x; // y has now been XORed with x twice; y=old_x
  x = x^y; // x has now been XORed with y twice; x=old_y
  // The values have been swapped
}
```

Listing 3.5: XOR swapping of two boolean values `x` and `y` with no temporary variables.

```
void swap(int & x, int & y) {
  x = x^y;
  y = y^x;
  x = x^y;
}
```

it does not matter whether we perform `a^b` or `b^a`.

What does this have to do with swapping values? Well, we can swap two `int` values `a` and `b` without any temporary storage (Listing 3.4). Consider that an integer is simply a block of boolean values. We can therefore do the same thing with a full integer, swapping all bitwise values simultaneously using the bitwise XOR operator (again, `^`), as shown in Listing 3.5[7].

Now we would like to ask ourselves, is this actually faster? XOR operations are cheap to implement in circuitry (more efficient than operations like `+`, which needs to consider carry operations between bits). Is three XOR operations better than three copies using one temporary value?

Let's test it. One swap operation would be far too fast to time accurately (and, in some circumstances, could even be optimized out by a clever compiler by simply exchanging the variable names at compile time). For this reason, we will perform many swap operations. Listing 3.6 performs these by copying and using a temporary value, whereas Listing 3.7 does this using the XOR swapping trick and uses no temporary values. There are arguments to be made on either side: the fact that the XOR method does not need a

[7]I do not know how the waiter would XOR the linguine and ravioli together, but if you ever see that happen, I'd like to know.

Listing 3.6: Copy swap benchmark. Several swaps are performed by copying to a temporary value.

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  unsigned long N = 1<<24;

  unsigned long*x = new unsigned long[N];
  unsigned long*y = new unsigned long[N];

  for (unsigned long i=0; i<N; ++i) {
    x[i] = i;
    y[i] = N-i;
  }

  Clock c;
  unsigned long temp;
  for (unsigned long i=0; i<N; ++i) {
    temp = x[i];
    x[i] = y[i];
    y[i] = temp;
  }
  c.ptock();

  return 0;
}
```

temporary value could be useful, but it does so by slightly increasing the complexity of the operations performed.

To take into account stochasticity, we will script with `bash` and `awk` to will compute the average runtime over 1024 replicate trials (Listing 3.8). We can run the same script again using for the XOR swap benchmark by making a few changes[8].

When we benchmark both the copy swap and the XOR swap in this manner, we see our results (Table 3.1). From the results, the copy-based

[8]Whenever you copy code in this way, make sure that you did not leave a little of the old code behind. For example, perhaps you replaced most instances of “copy” with “xor”, but forgot one. This is a very common source of problems, and so it is advisable that you create your own general script for benchmarking a generic `.cpp` file argument.

Listing 3.7: XOR swap benchmark. Several swaps are performed using XOR without any temporary values.

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  const unsigned long N = 1<<24;

  unsigned long*x = new unsigned long[N];
  unsigned long*y = new unsigned long[N];

  for (unsigned long i=0; i<N; ++i) {
    x[i] = i;
    y[i] = N-i;
  }

  Clock c;
  for (unsigned long i=0; i<N; ++i) {
    x[i] ^= y[i];
    y[i] ^= x[i];
    x[i] ^= y[i];
  }
  c.ptock();

  return 0;
}
```

Listing 3.8: Benchmarking Listing 3.6 over 1024 replicates. This `bash` script takes care to save the runtimes before computing the average so that `awk` is not running at the same time as the benchmarks.

```
# compile:
g++ -std=c++11 copy-swap.cpp -o copy

# save 1024 replicate runtimes:
for ((i=0; i<1024; ++i))
do
./xor
done > /tmp/copy

# Compute average runtime:
cat /tmp/copy | awk 'BEGIN {tot=0;} // {tot += $2} END {print tot/NR;}'
```

swap appears much faster[9]. It can also be useful to look at the minimum and maximum runtimes (or the 95 percentile runtimes) in order to add error bars to our measurements, or to plot and overlay histograms of the runtimes of the different methods. However, with 1024 replicates (and therefore, an expectation that the variability of the average will not be very large), we will trust that a nearly 2× speedup is not simply the result of random variability. Thus we declare the copy swap the winner of this benchmark; in this case, it appears the simplest methods are the best[11].

What happens when we turn on compiler optimizations? Let's run it again and find out. Here we will keep everything in our benchmark script the same with the exception of adding the `-O3` flag when running `g++`. Interestingly, in this case, both methods perform nearly identically well (Table 3.2).

[9]Note that we have not proven, strictly speaking, that it is *significantly* faster. To do that, we would need an estimate of the variability as well. But we do know that the variability of an average of n independent and identically distributed replicate trials should shrink as n grows large[10].

[10]This is basically the law of large numbers, but it can also be seen in a general form using the central limit theorem.

[11]If we were in Brooklyn or Seattle, we would then immediately start re-branding the basic copy swap as "old world style", "handmade", and, of course, "artisinal", and attach a slogan like, "Sometimes the old ways [pauses to turn and slowly smile at camera] are the best."[12].

[12]With the exception of bubble sort... and thalidomide... Basically, be wary of anything from before 1960. But yeah, otherwise, the old ways are the best.

	Copy swap	XOR swap
Average runtime (seconds)	0.07542	0.1268

Table 3.1: Swap runtimes without compiler optimizations. Runtimes are reported to four significant figures. The copy-based method appears faster on average than the XOR-based method.

	Copy swap	XOR swap
Average runtime (seconds)	0.03020	0.03034

Table 3.2: Swap runtimes with compiler optimizations. Runtimes are reported to four significant figures. Both methods appear to perform roughly the same.

So while the XOR method does not seem to be worth the effort in this case, it is promising for cases where we would run out of registers otherwise.

3.3 Looking into the assembly code

Perhaps the compiler is so smart that it detects that we're swapping with XOR and therefore the `-O3` optimizations simply replace our XOR code with code that uses a temporary value to perform the copy swap? This is difficult to know because compilers are still somewhat of a black box to us, and they are updated all the time.

But we can actually test this by recompiling to assembly code rather than compiling to an executable: `g++ -std=c++11 xor-swap.cpp -O3 -S`. This command produces a new file, `xor-swap.s`, and when we read it, we can find some XOR operations, whereas we see no such operations when we generate the corresponding file `copy-swap.s`. Interestingly, what we see is actually the opposite: the compiler is actually converting the copy-based swap to an xor-based swap (we see no XOR operations in the assembly file `copy-swap.s` after running `g++ -std=c++11 copy-swap.cpp -S` but do see them when in the assembly file after running `g++ -std=c++11 copy-swap.cpp -O3 -S`. In fact, the assembly files produced by the copy and XOR methods with `-O3` are very similar.

3.4 Why does the compiler not remove our benchmarks as dead code?

In previous benchmarks, we saw examples where a result was computed but where the result was never used, and that this sometimes yielded a runtime of 0s. This was because the compiler eliminated basically the entire benchmark as "dead code". So this naturally gives rise to a question: why are our benchmarks here not labeled dead code? After all, we never print the results using `std::cout`.

This subtlety is because we are modifying pointers (this is how the arrays are stored), and the compiler is not yet reliable to identify which pointers are modifying something temporary (these modifications can be marked as dead code if that temporary value has no subsequent dependencies) as opposed to the pointers modifying something of great import (*e.g.*, modifying global variables or even more hacky practices[13]). Of course, in these benchmarks we do no such thing, but the compiler is not smart enough to know for sure. There are even cases where we do not modify any non-temporary values, but where the compiler cannot know.

Generally this has to do with memory allocation and aliasing, which will be discussed in greater detail in Chapter 11. But for now, we can safely assume dead code elimination is not ruining our benchmark results by the fact that these very elementary benchmarks have nonzero runtimes.

3.5 Quadratic vs. subquadratic sorting algorithms

Here we will again test a simpler method against a more sophisticated counterpart. In this case we will use a simple quadratic selection sort (Listing 3.9, improved in Listing 3.10) and compare it against a subquadratic (*i.e.*, $O(n \log(n))$) merge sort implementation (Listing 3.11). Like with the swapping question, we can imagine pros and cons to the various methods *a priori*[14]: Selection sort will be quite simple to implement well in place[15]. But

[13] *E.g.*, it may not be good practice, but one could directly modify the return value of `int main()` by changing modifying its address via a pointer

[14] "From earlier", referring here to our belief before we see evidence.

[15] An "in-place" method does not make allocations on the fly and only uses $O(1)$ temporary storage; it simply operates directly on the memory provided and modifies it to produce

on the other hand, merge sort is an $O(n \log(n))$ algorithm, whereas selection sort is $\in O(n^2)$. Of course, the answer here will be determined by the n, the length of the list, being sorted; any constant advantage of the selection sort implementation can be overcome by using a large enough n, because n^2 will eventually grow to overwhelm the constant.

Here we will make a two minor changes to our benchmarking methods: we will use `-O3` by default, and we will accept command line argument for the length of the list so that we can appreciate the influence the size of the list makes.

Listing 3.9: Selection sort implementation and benchmark. The array is sorted by finding the smallest item in the full list, then finding the smallest item in the remaining $n-1$ elements, then the smallest item in the remaining $n-2$ elements and so forth, for an overall runtime in $O(\sum_{i=1}^{n} i) = O(n^2)$. Here, elements are swapped using `std::swap`, an implementation of the `swap` algorithm used above, which is templated to accept generic types. This selection sort implementation runs in place, meaning it directly modifies our array rather than making a copy.

```
#include "../Clock.hpp"
#include <iostream>

void selection_sort(unsigned long*source, unsigned long n) {
  for (unsigned long i=0; i<n; ++i)
    for (unsigned long j=i+1; j<n; ++j)
      if (source[j] < source[i])
        std::swap(source[i], source[j]);
}

int main(int argc, char**argv) {
  if (argc == 2) {
    const unsigned long N = atoi(argv[1]);

    unsigned long*x = new unsigned long[N];

    for (unsigned long i=0; i<N; ++i)
      x[i] = rand()%10000;

    Clock c;
    selection_sort(x, N);
```

the desired result. You will better understand the significance of in-place implementations after Chapter 9, which discusses cache.

```
    c.ptock();
  }
  else
    std::cerr << "Usage: sort <n>" << std::endl;
  return 0;
}
```

Listing 3.10: Improved selection sort. By modifying a local variable, aliasing concerns (discussed in Chapters 10 and 11) are avoided, enabling better compiler optimization.

```
#include "../Clock.hpp"
#include <iostream>

void selection_sort(unsigned long*source, unsigned long n) {
  for (unsigned long i=0; i<n; ++i) {
    unsigned long min_ind=i;
    unsigned long min_val=source[i];
    for (unsigned long j=i+1; j<n; ++j) {
      const unsigned long source_j = source[j];

      if (source_j < min_val) {
        min_ind = j;
        min_val = source_j;
      }
    }
    std::swap(source[i], source[min_ind]);
  }
}

int main(int argc, char**argv) {
  if (argc == 2) {
    const unsigned long N = atoi(argv[1]);

    unsigned long*x = new unsigned long[N];

    for (unsigned long i=0; i<N; ++i)
      x[i] = rand()%10000;

    Clock c;
    selection_sort(x, N);
    c.ptock();
  }
  else
```

```
    std::cerr << "Usage: sort <n>" << std::endl;
  return 0;
}
```

Listing 3.11: Merge sort implementation and benchmark. The array is sorted by splitting it in half, sorting the left and right halves, and then merging the sorted results in $O(n)$, creating a divide-and-conquer algorithm that runs in $O(n \log(n))$. When a base case list length $n = 1$ is detected, the list is deemed already sorted (terminating further recursion). This implementation does not sort in place, and thus needs to copy the sorted result back after finishing.

```
#include "../Clock.hpp"
#include <iostream>

void merge_sort(unsigned long*source, unsigned long n) {
  if (n == 1)
    return;

  unsigned long*source_2 = source + n/2;
  merge_sort(source, n/2);
  merge_sort(source_2, n-n/2);

  unsigned long*buffer = new unsigned long[n];
  // Merge sorted halves into buffer:
  unsigned long i=0, j=0, buffer_ind=0;
  while (i < n/2 && j < (n-n/2)) {
    if (source[i] < source_2[j]) {
      buffer[buffer_ind] = source[i];
      ++i;
    }
    else {
      // In case of equality, order doesn't matter, so use this case:
      buffer[buffer_ind] = source_2[j];
      ++j;
    }

    ++buffer_ind;
  }
  // Copy remaining values:
  for (; i<n/2; ++i, ++buffer_ind)
    buffer[buffer_ind] = source[i];
  for (; j<n-n/2; ++j, ++buffer_ind)
    buffer[buffer_ind] = source_2[j];
  // Copy back sorted list from buffer:
  for (i=0; i<n; ++i)
    source[i] = buffer[i];

  delete[] buffer;
}

int main(int argc, char**argv) {
  if (argc == 2) {
    const unsigned long N = atoi(argv[1]);

    unsigned long*x = new unsigned long[N];

    for (unsigned long i=0; i<N; ++i)
      x[i] = rand()%10000;

    Clock c;
    merge_sort(x, N);
    c.ptock();
  }
  else
    std::cerr << "Usage: sort <n>" << std::endl;
  return 0;
}
```

	Selection sort	Selection sort (2)	Merge sort
$n = 16$	4.375e-07	1.08301e-06	7.406e-06
$n = 1024$	0.0009152	0.0008235	0.0001358
$n = 65536$	2.063	1.609	0.008960

Table 3.3: Runtimes (in seconds) of selection sort (Listing 3.9), the modified selection sort (Listing 3.10), and merge sort (Listing 3.11). Runtimes are performed on arrays of different lengths n and reported to four significant figures. For small arrays, both methods are efficient, but selection sort is more than $10\times$ faster (meaning it would could offer substantial savings if many small lists were being sorted). For larger arrays, the advantage of merge sort becomes apparent.

The merge sort implementation from Listing 3.11 does indeed perform substantially better than the selection sort implementation on large problems, but is more than $10\times$ slower on small problems (Table 3.3). A natural question arises: how can we achieve the qualities of selection sort for small problems and the qualities of merge sort for larger problems? One way to accomplish this is to modify our selection sort algorithm so that the base case is no longer $n = 1$ (which we can ignore, as a one-element list has only one possible ordering and is thus already sorted), and instead use a threshold $n \leq K$, at which point selection sort will be used to finish (Listing 3.12). We will arbitrarily use $K = 32$, because that will guarantee good performance on lists of the size where our merge sort implementation has been shown to struggle compared to selection sort.

Listing 3.12: Modified merge sort implementation and benchmark. The base case now calls selection sort when $n \leq 32$.

```
#include "../Clock.hpp"
#include <iostream>

void selection_sort(unsigned long*source, unsigned long n) {
  for (unsigned long i=0; i<n; ++i) {
    unsigned long min_ind=i;
    unsigned long min_val=source[i];
    for (unsigned long j=i+1; j<n; ++j)
      if (source[j] < min_val) {
        min_ind = j;
        min_val = source[j];
      }

    std::swap(source[i], source[min_ind]);
  }
}

void merge_sort(unsigned long*source, const unsigned long n) {
  if (n <= 32) {
    selection_sort(source, n);
    return;
  }

  unsigned long*source_2 = source + n/2;
  merge_sort(source, n/2);
  merge_sort(source_2, n-n/2);

  unsigned long*buffer = new unsigned long[n];
  // Merge sorted halves into buffer:
  unsigned long i=0, j=0, buffer_ind=0;
  while (i < n/2 && j < (n-n/2)) {
    if (source[i] < source_2[j]) {
      buffer[buffer_ind] = source[i];
      ++i;
    }
    else {
      // In case of equality, order doesn't matter, so use this case:
      buffer[buffer_ind] = source_2[j];
      ++j;
    }

    ++buffer_ind;
  }
  // Copy remaining values:
  for (; i<n/2; ++i, ++buffer_ind)
    buffer[buffer_ind] = source[i];
  for (; j<n-n/2; ++j, ++buffer_ind)
    buffer[buffer_ind] = source_2[j];

  // Copy back sorted list from buffer:
  for (i=0; i<n; ++i)
    source[i] = buffer[i];

  delete[] buffer;
}

int main(int argc, char**argv) {
  if (argc == 2) {
    const unsigned long N = atoi(argv[1]);

    unsigned long*x = new unsigned long[N];

    for (unsigned long i=0; i<N; ++i)
      x[i] = rand()%10000;

    Clock c;
    merge_sort(x, N);
    c.ptock();
  }
  else
    std::cerr << "Usage: sort <n>" << std::endl;
  return 0;
}
```

Table 3.4 appends the runtimes of the modified merge sort implementation to the results already shown in Table 3.3. Not only does the modified merge sort perform well on small arrays (*e.g.*, $n = 16$), it also performs substantially better than the original merge sort on large arrays. Herein lies a crucial difference between modifying the merge sort recursion itself instead of simply using a single `if` statement to estimate whether it's more efficient to call selection sort or merge sort on a list of the given size. Where the latter would not improve the runtime of the modified merge sort on large arrays, the former (which is how we implemented it in Listing 3.12) will also improve performance on large arrays by enabling a more shallow call tree from its recursions. The method demonstrated here is quite useful for creating efficient implementations of divide-and-conquer algorithms (*e.g.*, sorting, Strassen matrix multiplication, Karatsuba's integer multiplication, FFT, *etc.*). Not only is a faster algorithm used on trivially small problems (on which the divide-and-conquer will likely perform worse than a naive method), the cost of the recursion itself may even be amortized out, because the computational cost at a single "leaf" in the call tree[16] may dwarf the overhead from a recursive implementation in many cases.

	Selection sort	Selection sort (2)	Merge sort	Merge sort (2)
$n = 16$	4.375e-07	1.083e-06	7.406e-06	4.688e-07
$n = 1024$	0.0009152	0.0008235	0.0001358	7.966e-05
$n = 65536$	2.063	1.609	0.008960	0.004901

Table 3.4: Runtimes (in seconds) of selection sort (Listing 3.9), modified selection sort (Listing 3.9), merge sort (Listing 3.11), and the modified merge sort (Listing 3.12). Runtimes are performed on arrays of different lengths n and reported to four significant figures. The modified selection sort implementation is not only competitive with selection sort on small arrays, it's also substantially faster than the original merge sort on large arrays.

3.6 Profiling

We can profile our code (*i.e.*, run the code and measure the specific amounts of CPU time taken by each line or function) with `valgrind`. To profile the call `./a.out 1024`, we simply run `valgrind --tool=callgrind ./a.out 1024`; this will output a `callgrind.out...` file, which we can visualize by

[16]The tree visualizing the recursive calls

calling `kcachegrind callgrind.out...`, where `callgrind.out...` is the file output by `valgrind`.

Questions

1. [**Level 1**] Using $n = 65536$, repeat the benchmarks of the modified merge sort implementation with different values of K and plot the performance as a function of K. What is the best K on your system?

2. [**Level 2**] Implement merge sort without allocating memory inside the function (instead, accept two parameters, `unsigned long*buffer` and `unsigned long*source`, both of which will be allocated once in `main`, before calling the sorting function). Time replicate trials with $n = 65536$. What is the best K for the merge sort implementation that does not allocate memory (previous question)? Using the best K value for each implementation, is there any speedup when not allocating memory inside the function? If so, how large is the speedup?

3. [**Level 3**] Implement a recursive, in-place quicksort (using a random pivot element), which does not allocate any memory inside the function and which uses no buffer. How does the performance compare with merge sort and the sans-allocation merge sort? Does this surprise you? Why do you think the results would be what they are?

Projects

1. [**Level 2**] Make the fastest sorting implementation you can (do not bother with radixing or other complicated algorithms: only use the $<$ operator).

Chapter 4

Conway's Game of Life, Buffering, and the Surprising Lethargy of `if` Statements

4.1 Cellular automata and Conway's Game of Life

Cellular automata are built by using simple rules, which are used to update the state of a "cell" (usually with boolean possible states, which indicate living or dead). These cells are updated as a function of the states of the adjacent cells. This state-dependent determinism, by which the future state is a deterministic function of the previous state makes this an "automata". In the general case, adjacency could be defined using an arbitrary graph, but in practice, adjacency is usually defined as neighboring pixels on a grid or lattice.[1]

One of the earliest and most famous cellular automata was proposed by John Conway in 1970. Conway's cells are simply square pixels on a rectangular grid, and the rules for cells coming alive, staying alive, and dying are as follows:

> **A cell comes alive** if it touches exactly 3 living neighboring pixels in the previous iteration.

[1]For more information on various types of cellular automata, see Stephen Wolfram's book *A New Kind of Science*.

A cell stays alive if it touches 2 or 3 living neighboring pixels in the previous iteration.

A cell dies if it touches fewer than 2 (so called death from "loneliness") or more than 3 living neighboring pixels (so called death from "overcrowding") in the previous iteration.

In this way, Conway's model can be thought of as a very simple model of cells growing on a petri dish: too many living neighbors and there is death from overconsumption of resources; too few living neighbors, and the cell colony may die out because it does not reproduce fast enough to reliably overcome spontaneous processes that kill the cells. Note that Conway included diagonal pixels as "adjacent" (so the minimum number of living adjacent cells is 0 and the maximum is 8).

Interestingly, Conway's "game of life" is a universal Turing machine, meaning that any Turing machine can be encoded as an initial pixel "board" for the game of life (and vice versa: clearly a large enough computer, which can be emulated by a Turing machine, can be used to implement Conway's game of life).[2]

In this chapter, we will implement Conway's game of life in `C++`.

4.2 A wrong, first implementation via `std::vector`

We will begin with the simplest implementation (an implementation that will actually have a crucial mistake). In this case, we will simply use `std::vector<std::vector<bool> >` to store the board matrix, then advance each cell in turn using the rules described above (Listing 4.1). This initial implementation has a flaw in that it: it modifes the cells before they are used by their neighbors, which will change the outcome as to whether

[2]This Turing complete claim is the kind of thing that sounds quite interesting when someone mentions it at a conference, but the 500th time you hear it, and like always the person doesn't know any more information, you may begin to wonder if it was just a rumor started years ago by Conway to see whether anyone bothers to understand things anymore.[3] They key is to use "gliders" (patterns in the game of life that reproduce themselves translated by a pixel or so, and thus appear to move as generations pass) to pass information between components and to construct logic gates using other patterns.

[3]By the third day of a conference, it is normal to begin to die from overcrowding.

these neighbors will live or die in the next iteration. We will not bother timing this version, since we already know that it is flawed.

Listing 4.1: Simple, erroroneous game of life implementation. This implementation modifies cells before they are used by their neighbors to the right and down.

```
#include "../Clock.hpp"
#include <iostream>
#include <vector>

bool advance_single(const std::vector<std::vector<bool> > board, const
    unsigned int R, const unsigned int C, int r, int c) {
  unsigned int living_neighbors = 0;
  for (int i=std::max(0,r-1); i<=std::min(int(R)-1,r+1); ++i) {
    for (int j=std::max(0,c-1); j<=std::min(int(C)-1,c+1); ++j) {
      // Do not count this cell:
      if (i != r || j != c)
        if (board[i][j])
          ++living_neighbors;
    }
  }

  return living_neighbors == 3 || (board[r][c] && living_neighbors >=2 &&
      living_neighbors <= 3);
}

void advance(std::vector<std::vector<bool> > & board, const unsigned int
    R, const unsigned int C) {
  // Note:
  // R == board.size()
  // C == board[0].size()

  std::vector<std::vector<bool> > result(R, std::vector<bool>(C, false));

  for (unsigned int i=0; i<R; ++i)
    for (unsigned int j=0; j<C; ++j)
      // Error: we're modifying the board, and thus changing the
      // number of living neighbors for other cells:
      board[i][j] = advance_single(board, R, C, i, j);
}

void print_board(const std::vector<std::vector<bool> > board, unsigned
    int R, unsigned int C) {
  for (unsigned int i=0; i<R; ++i) {
    for (unsigned int j=0; j<C; ++j)
      std::cout << int(board[i][j]);
    std::cout << std::endl;
  }
}

int main(int argc, char**argv) {
  if (argc == 3) {
    const unsigned int R = 1<<10;
    const unsigned int C = 1<<11;

    unsigned int NUM_REPS = atoi(argv[1]);
    bool print = bool( atoi(argv[2]) );

    srand(0);
    std::vector<std::vector<bool> > cur(R, std::vector<bool>(C));
    for (unsigned int i=0; i<R; ++i)
      for (unsigned int j=0; j<C; ++j)
        cur[i][j] = (rand() % 2 == 0);

    if (print)
      print_board(cur, R, C);

    Clock c;

    for (unsigned int rep=0; rep<NUM_REPS; ++rep)
      advance(cur, R, C);

    c.ptock();

    if (print)
      print_board(cur, R, C);
  }
  else
    std::cerr << "usage game-of-life <generations> <0:don't print,
        1:print>" << std::endl;
  return 0;
}
```

4.3 Corrected implementation with `std::vector`

Our second attempt will use the same basic approach, but will return a new board instead of modifying the current board. By using 2× the memory in this manner, all of the new cell states can be computed using the existing board's states without modifying them.

Listing 4.2: A corrected version of Listing 4.1. Rather than modify the board, a new board is returned.

```
#include "../Clock.hpp"
#include <iostream>
#include <vector>

bool advance_single(const std::vector<std::vector<bool> > board, const
    unsigned int R, const unsigned int C, int r, int c) {
  unsigned int living_neighbors = 0;
  for (int i=std::max(0,r-1); i<=std::min(int(R)-1,r+1); ++i) {
    for (int j=std::max(0,c-1); j<=std::min(int(C)-1,c+1); ++j) {
      // Do not count this cell:
      if (i != r || j != c)
        if (board[i][j])
          ++living_neighbors;
    }
  }

  return living_neighbors == 3 || (board[r][c] && living_neighbors >=2 &&
      living_neighbors <= 3);
}

std::vector<std::vector<bool> > advance(std::vector<std::vector<bool> >
    board, const unsigned int R, const unsigned int C) {
  // Note:
  // R == board.size()
  // C == board[0].size()

  std::vector<std::vector<bool> > result(R, std::vector<bool>(C, false));

  for (unsigned int i=0; i<R; ++i)
    for (unsigned int j=0; j<C; ++j)
      result[i][j] = advance_single(board, R, C, i, j);

  return result;
}

void print_board(const std::vector<std::vector<bool> > board, unsigned
    int R, unsigned int C) {
  for (unsigned int i=0; i<R; ++i) {
    for (unsigned int j=0; j<C; ++j)
      std::cout << int(board[i][j]);
    std::cout << std::endl;
  }
}

int main(int argc, char**argv) {
  if (argc == 3) {
    const unsigned int R = 1<<10;
    const unsigned int C = 1<<11;

    unsigned int NUM_REPS = atoi(argv[1]);
    bool print = bool( atoi(argv[2]) );

    srand(0);
    std::vector<std::vector<bool> > cur(R, std::vector<bool>(C));
    for (unsigned int i=0; i<R; ++i)
      for (unsigned int j=0; j<C; ++j)
        cur[i][j] = (rand() % 2 == 0);

    if (print)
      print_board(cur, R, C);

    Clock c;

    for (unsigned int rep=0; rep<NUM_REPS; ++rep)
      cur = advance(cur, R, C);

    c.ptock();

    if (print)
      print_board(cur, R, C);
  }
  else
    std::cerr << "usage game-of-life <generations> <0:don't print,
        1:print>" << std::endl;
  return 0;
}
```

While the modified implementation in Listing 4.2 is correct, it is still incredibly slow. In contrast with Listing 4.1, we now make two copies of the 1024×2048 board (storing one board takes 2MB) each time we advance. But even more importantly, by looking carefully we see that our function `advance_single`, which counts the neighboring living cells in the previous iteration and returns a `bool` indicating whether the cell will be alive in the next generation, *also* copies the board, and so the board will be copied 1024×2048 times to advance the board one generation. Even with `-O3` optimizations (note that `-O3` optimizations will be used throughout this chapter even when not explicitly mentioned), the code from Listing 4.2 takes over an hour to process 1000 generations.

4.4 The importance of passing by reference

Our third attempt will pass by reference wherever possible (Listing 4.3). By passing the board by reference (via the `&` symbol) whenever possible, many fewer copies are made. Simply by passing by reference, we will now see a massive speedup in our program.[4]

However, we still copy the new state of the board (*i.e.*, the return value of `advance`). Note that each return value allocates a new board, and we are not guaranteed that this allocation will allocate the same block of memory used by the previous board. This means that the board may move through memory as the generations of the game advance, and so the board may not stay cached[6]. The code in Listing 4.3 takes 51.44s to run 1000 generations.

Listing 4.3: A modified version of Listing 4.3. The large `std::vector<std::vector<bool> >` object is passed by reference wherever possible.

```
#include "../Clock.hpp"
#include <iostream>
#include <vector>
```

[4]I once had a roommate who was tasked with writing some fast computational physics code during the summer. After a lot of work, it was quite slow, and when we looked at the code together, I saw that he was passing his matrices by value, and was thus making local copies every time. After adding in a few humble ampersands, his code purred like a walrus. He later went to the NSA.[5]

[5]I know this because he needed– you guessed it– a reference (a personal reference in this case).

[6]Cache will be discussed more thoroughly in Chapter 9.

```
bool advance_single(const std::vector<std::vector<bool> > & board, const
    unsigned int R, const unsigned int C, int r, int c) {
  unsigned int living_neighbors = 0;
  for (int i=std::max(0,r-1); i<=std::min(int(R)-1,r+1); ++i) {
    for (int j=std::max(0,c-1); j<=std::min(int(C)-1,c+1); ++j) {
      // Do not count this cell:
      if (i != r || j != c)
        if (board[i][j])
          ++living_neighbors;
    }
  }

  return living_neighbors == 3 || (board[r][c] && living_neighbors >=2 &&
      living_neighbors <= 3);
}

std::vector<std::vector<bool> > advance(const
    std::vector<std::vector<bool> > & board, const unsigned int R, const
    unsigned int C) {
  // Note:
  // R == board.size()
  // C == board[0].size()

  std::vector<std::vector<bool> > result(R, std::vector<bool>(C, false));

  for (unsigned int i=0; i<R; ++i)
    for (unsigned int j=0; j<C; ++j)
      result[i][j] = advance_single(board, R, C, i, j);

  return result;
}

void print_board(const std::vector<std::vector<bool> > & board, unsigned
    int R, unsigned int C) {
  for (unsigned int i=0; i<R; ++i) {
    for (unsigned int j=0; j<C; ++j)
      std::cout << int(board[i][j]);
    std::cout << std::endl;
  }
}

int main(int argc, char**argv) {
  if (argc == 3) {
    const unsigned int R = 1<<10;
    const unsigned int C = 1<<11;

    unsigned int NUM_REPS = atoi(argv[1]);
    bool print = bool( atoi(argv[2]) );

    srand(0);
    std::vector<std::vector<bool> > cur(R, std::vector<bool>(C));
    for (unsigned int i=0; i<R; ++i)
      for (unsigned int j=0; j<C; ++j)
        cur[i][j] = (rand() % 2 == 0);

    if (print)
      print_board(cur, R, C);

    Clock c;

    for (unsigned int rep=0; rep<NUM_REPS; ++rep)
      cur = advance(cur, R, C);

    c.ptock();

    if (print)
      print_board(cur, R, C);
  }
  else
    std::cerr << "usage game-of-life <generations> <0:don't print,
        1:print>" << std::endl;
  return 0;
}
```

4.5 Buffering using `bool*`

There is a way to ensure that we use the same memory for the current board and for the next board. We will allocate two boards, one for our current state and one for the next state, at the beginning of our `main` function, and then we can use the current board to initialize the next board. This method is known as "buffering", and is quite commonly used for graphics rendering (*e.g.*, drawing the next state of the screen in a video game engine).

After we initialize the next board, it will become next iteration's current board. There we have two options: One option is to copy the memory from the next board to the current board (*e.g.*, using `memcpy`). The second option is more lazy, and is therefore faster[7]: if we use pointers for the arrays, we can simply swap the pointers (`bool*` instead of `std::vector<bool>`) for the next and current boards instead of swapping the array contents (Listing 4.4)[9]

Note that without optimizations on, Listing 4.4 will run substantially faster than Listing 4.3, but with optimizations turned on it is only be moderately faster: it runs 1000 generations in 43.41s.

Listing 4.4: A buffered game of life. Two matrices are allocated at the start of the program. Then one is initialized as the current board, and one is used to construct the next board. In order to subsequently make the next board into the current board and repeat the process, the pointers are swapped rather than copying the data itself.

```
#include "../Clock.hpp"
#include <iostream>

bool advance_single(const bool*prev, const unsigned int R, const unsigned
    int C, int r, int c) {
  unsigned int living_neighbors = 0;
  for (int i=std::max(0,r-1); i<=std::min(int(R)-1,r+1); ++i) {
    for (int j=std::max(0,c-1); j<=std::min(int(C)-1,c+1); ++j) {
      // Do not count this cell:
      if (i != r || j != c)
        if (prev[i*C+j])
          ++living_neighbors;
    }
  }
  return living_neighbors == 3 || (prev[r*C+c] && living_neighbors >=2 &&
      living_neighbors <= 3);
}

void advance(bool*cur, const bool*prev, const unsigned int R, const
    unsigned int C) {
  for (unsigned int i=0; i<R; ++i)
    for (unsigned int j=0; j<C; ++j)
      cur[i*C+j] = advance_single(prev, R, C, i, j);
}
```

[7] "The best mathematician is a lazy mathematician."[8]

[8] You know who said that? Me neither, I was going to look it up, but I was hanging out, doing some math, taking it easy and, long story short, this one got away from me...

[9] Allowing such lazy assignment or swap operations to be performed in an object-oriented context (for clarity, we are doing this manually here, not using a object oriented code) is a large advancement made in `C++11`, using "rvalue references".

```
void print_board(const bool*board, const unsigned int R, const unsigned
    int C) {
  for (unsigned int i=0; i<R; ++i) {
    for (unsigned int j=0; j<C; ++j)
      std::cout << int(board[i*C+j]);
    std::cout << std::endl;
  }
}

int main(int argc, char**argv) {
  if (argc == 3) {
    const unsigned int R = 1<<10;
    const unsigned int C = 1<<11;

    unsigned int NUM_REPS = atoi(argv[1]);
    bool print = bool( atoi(argv[2]) );

    srand(0);
    bool*prev = (bool*)calloc(R*C,sizeof(bool));
    bool*cur = (bool*)calloc(R*C,sizeof(bool));

    for (unsigned int i=0; i<R; ++i)
      for (unsigned int j=0; j<C; ++j)
        cur[i*C+j] = (rand() % 2 == 0);

    if (print)
      print_board(cur, R, C);

    Clock c;

    for (unsigned int rep=0; rep<NUM_REPS; ++rep) {
      std::swap(cur, prev);
      advance(cur, prev, R, C);
    }

    c.ptock();

    if (print)
      print_board(cur, R, C);
  }
  else
    std::cerr << "usage game-of-life <generations> <0:don't print,
        1:print>" << std::endl;
  return 0;
```

Listing 4.5: Testing Listing 4.4 against Listing 4.3. This `bash` script runs both methods for 100 generations, saves the output (by setting the second command line parameter to `1`), uses `grep` to ignore the potentially different runtimes, and then uses `diff` to compare the results character by character. No output means the test succeeds, while any other output means it fails.

```
g++ -std=c++11 return-copy.cpp
#run 100 generations and print initial board and final board:
./a.out 100 1 > /tmp/a
g++ -std=c++11 unbuffered.cpp
#run 100 generations and print initial board and final board:
./a.out 100 1 > /tmp/b
diff <(grep -v Took /tmp/a) <(grep -v Took /tmp/b)
```

```
}
```

Note that our iterative approach to successively honing our code from a naive version to a fast version allows us to use the simpler versions to test the more advanced versions and to verify that they are correct. For example, this can be done by using the second command line argument `1`, which will print the initial board and final board (Listing 4.5).

4.6 Decreasing reliance on `if` statements

`if` statements are generally much slower than other types of statements. This is partly because of pipelining, where the processor is built to partition tasks into separate sub-tasks. This enables one circuit to solve one sub-task while the circuit for a different sub-task is being applied to separate data[10].

[10]This is like doing laundry: if the washing and drying step were performed together and each take roughly an hour, then a load of laundry takes 2 hours and 10 loads of laundry take 20 hours. But with separate washing and drying steps, while the first load of laundry is drying, the second load of laundry can be put in the washer. With the exception of the first load (there is nothing being dried while the first load is in the washer) and the last load (there is nothing new being washed while the final load is being dried), 2 jobs are always running simultaneously. Thus, if you did many loads of laundry (enough that the first and last loads didn't contribute much), you would get a roughly $2\times$ speedup by having separate washing and drying apparatuses[11].

[11]Or is it apparati?

These kinds of tricks rely heavily on knowing what task will come next; `if` statements make it difficult to know the next task until they are evaluated. In contrast, a loop of the form `for (int i=0; i<10; ++i) array[i] += 1;` may sometimes be able to process the `i=1` iteration of the loop simultaneous to the `i=0` iteration. `for (int i=0; i<10; ++i) if (array[i] == 0) break; array[i] += 1;` ruins that predictive ability. This becomes more important with loop unrolling optimizations mentioned in Chapter 10. In fact, so great is the speed-up from hardware-level parallelizations of which modern processors are capable that the best strategy is often to use "branch prediction"[12] on the hardware to predict whether the `if` statement will have a `true` or `false` argument, and thus whether the code inside the `if` statement will be executed; this sometimes allows the code inside the `if` statement to be run simultaneously with the previous few and subsequent few assembly instructions.

Of course, if the branch prediction proves to be wrong, the downside is that the state of the processor before the incorrect branch prediction needs to be restored and all of the code run since the incorrect branch prediction needs to be re-run with the correct boolean argument. As you may guess, this inflicts a fairly large runtime penalty. From that, you may find yourself asking, "Why is branch prediction even useful, given that penalty for being wrong is so high?" One reason that many `branch` statments can be correctly predicted is because of loops: the loop `while (a > 0)` will have a `branch` statement at the end of the loop that jumps back to the top of the loop when `a>0`. Such a high percentage of backwards branches are taken because of loops, and so using fairly primitive means, it is possible to perform branch prediction accurately enough that the benefits of frequently predicting correctly outweigh the high cost of predicting incorrectly. It's reminiscent of coming to a fork in the road when driving, and your GPS is slow to tell you which way to go; you could wait (no branch prediction) or you could take your best guess and then, if you were wrong, you would need to drive back to the fork in the road and take the correct route. Being wrong will take

[12]`if` statements translate to `branch` statements in assembly code. `branch` statements either jump to a specified line (like conditional versions of `goto` statements in `C`) if certain conditions are met. For example, the lines `if ( x == 7 ) x +=1; x *= 2;` may translate to assembly like `LINE1: SUB x,x,7; LINE2: BRnp LINE4; LINE3: ADD x,x,1; LINE4: MUL x,x,2;`, where `BRANCHnp` means "branch if the previous line had a result value that was (n)egative or (p)ositive". In this way, it branches *around* the line `x += 1` when the argument inside the `if` statement is `false`.

more time than waiting for the correct GPS directions, but if you're right often enough, it's still beneficial on average to guess rather than wait for your GPS.

Several of the `if` statements in our game of life program will result in `branch` statements that are are not quite as conducive to branch prediction. For example, `for` loops in `advance_single` have this form: `for (int i=std::max(0,r-1); i<=std::min(int(R)-1,r+1); ++i)`. Every time a cell is advanced with `advance_single`, the `std::min` and `std::max` functions will use `branch` statements[13]. These `min` and `max` statements are used so that we do not try to look for neighboring cells in row -1 or row `r` (the valid range is 0, 1, . . . `r-1`). One workaround for this is to simply pad our matrix with a row filled with dead cells on the top, a row filled with dead cells on the bottom, a column filled with dead cells on the left, and a column filled with dead cells on the right. Our $(R+2)\times(C+2)$ matrix will be larger, but not significantly so for large problems, and it will allow us to avoid a few `if` statements for each call to `advance_single` (Listing 4.6). It runs in 17.47s, under half the runtime of the buffered version without a border.

Listing 4.6: A buffered game of life with a border of dead pixels. By embedding the matrix inside a border of dead pixels, we no longer need special cases to avoid going off the edge of the matrix, and thus fewer `if` statements are used.

```
#include "../Clock.hpp"
#include <iostream>

bool advance_single(const bool*prev, const unsigned int R, const unsigned
    int C, int r, int c) {
  unsigned int living_neighbors = 0;
  for (int i=-1; i<=1; ++i) {
    for (int j=-1; j<=1; ++j) {
      // Do not count this cell:
      if (i != 0 || j != 0)
        if (prev[(r+i)*(C+2)+(c+j)])
          ++living_neighbors;
    }
  }
  return living_neighbors == 3 || (prev[r*(C+2)+c] && living_neighbors
      >=2 && living_neighbors <= 3);
```

[13]Consider how you would implement a `min` function: T min(const T & a, const T & b) { if (a < b) return a; return b; }

```
}

void advance(bool*cur, const bool*prev, const unsigned int R, const
    unsigned int C) {
  for (unsigned int i=1; i<R+1; ++i)
    for (unsigned int j=1; j<C+1; ++j)
      cur[i*(C+2)+j] = advance_single(prev, R, C, i, j);
}

void print_board(const bool*board, const unsigned int R, const unsigned
    int C) {
  for (unsigned int i=1; i<R+1; ++i) {
    for (unsigned int j=1; j<C+1; ++j)
      std::cout << int(board[i*(C+2)+j]);
    std::cout << std::endl;
  }
}

int main(int argc, char**argv) {
  if (argc == 3) {
    const unsigned int R = 1<<10;
    const unsigned int C = 1<<11;

    unsigned int NUM_REPS = atoi(argv[1]);
    bool print = bool( atoi(argv[2]) );

    srand(0);
    bool*prev = (bool*)calloc((R+2)*(C+2),sizeof(bool));
    bool*cur = (bool*)calloc((R+2)*(C+2),sizeof(bool));

    for (unsigned int i=1; i<R+1; ++i)
      for (unsigned int j=1; j<C+1; ++j)
        cur[i*(C+2)+j] = (rand() % 2 == 0);

    if (print)
      print_board(cur, R, C);

    Clock c;

    for (unsigned int rep=0; rep<NUM_REPS; ++rep) {
      std::swap(cur, prev);
      advance(cur, prev, R, C);
    }

    c.ptock();
    if (print)
      print_board(cur, R, C);
  }
  else
    std::cerr << "usage game-of-life <generations> <0:don't print,
        1:print>" << std::endl;
  return 0;
}
```

Listing 4.6 still features two `if` statements inside the loops of `advance_single`[14]. Fortunately, it is possible to remove both of those `if` statements. The first `if` statement ensures we do not count a cell as its own neighbor: `if (i != 0 || j != 0)`. This can be omitted as long as we remember to subtract its contribution out after the loops are finished.[15] The inner `if` statement checks whether the neighboring cell is living or dead and increments `living_neighbors` if the neighboring cell was living: `if (prev[(r+i)*(C+2)+(c+j)]) ++living_neighbors;`. Once again, this can be simplified in terms of arithmetic: `living_neighbors += prev[(r+i)*(C+2)+(c+j)];`. Here we use the fact that `bool` types are actually integers where `false` is equivalent to 0 and `true` is equivalent to 1. The result, Listing 4.7, runs 1000 generations in 7.289s, less than half the runtime of Listing 4.6.

Listing 4.7: A buffered game of life with still fewer `if` statements. `if` statements are replaced with equivalent arithmetic.

```
#include "../Clock.hpp"
#include <iostream>

bool advance_single(const bool*prev, const unsigned int R, const unsigned
    int C, int r, int c) {
  unsigned int living_neighbors = 0;
  // These loops will be unrolled beautifully:
  for (int i=-1; i<=1; ++i)
    for (int j=-1; j<=1; ++j)
```

[14]We should be much more wary of `if` statements that will be called many times, such as those inside loops.

[15]It may seem strange that a subtraction is actually cheaper than an `if` statement, but it's true; not only does the `if` statement make optimizations much more difficult, it's also called 9 times inside this loop, whereas the subtraction operation would simply be performed 1 time outside the loop.

```
      living_neighbors += prev[(r+i)*(C+2)+(c+j)];

  living_neighbors -= prev[r*(C+2)+c];
  return living_neighbors == 3 || (prev[r*(C+2)+c] && living_neighbors
      >=2 && living_neighbors <= 3);
}

void advance(bool*cur, const bool*prev, const unsigned int R, const
    unsigned int C) {
  for (unsigned int i=1; i<R+1; ++i)
    for (unsigned int j=1; j<C+1; ++j)
      cur[i*(C+2)+j] = advance_single(prev, R, C, i, j);
}

void print_board(const bool*board, const unsigned int R, const unsigned
    int C) {
  for (unsigned int i=1; i<R+1; ++i) {
    for (unsigned int j=1; j<C+1; ++j)
      std::cout << int(board[i*(C+2)+j]);
    std::cout << std::endl;
  }
}

int main(int argc, char**argv) {
  if (argc == 3) {
    const unsigned int R = 1<<10;
    const unsigned int C = 1<<11;

    unsigned int NUM_REPS = atoi(argv[1]);
    bool print = bool( atoi(argv[2]) );

    srand(0);
    bool*prev = (bool*)calloc((R+2)*(C+2),sizeof(bool));
    bool*cur = (bool*)calloc((R+2)*(C+2),sizeof(bool));

    for (unsigned int i=1; i<R+1; ++i)
      for (unsigned int j=1; j<C+1; ++j)
        cur[i*(C+2)+j] = (rand() % 2 == 0);

    if (print)
      print_board(cur, R, C);

    Clock c;

    for (unsigned int rep=0; rep<NUM_REPS; ++rep) {
      std::swap(cur, prev);
      advance(cur, prev, R, C);
    }

    c.ptock();

    if (print)
      print_board(cur, R, C);
  }
  else
    std::cerr << "usage game-of-life <generations> <0:don't print,
        1:print>" << std::endl;
  return 0;
}
```

4.7 Replacing logical operations with a table

Logical operations on booleans (*e.g.*, `&&`, `||`, *etc.*) are quite fast; however, the line `return living_neighbors == 3 || (prev[r*(C+2)+c] && living_neighbors >=2 && living_neighbors <= 3);` will often need to wait on the previous operation to be completed before subsequent operations can be performed. For example, `living_neighbors == 3` means that the remainder of the logical operation need not be computed, whereas `living_neighbors != 3` will defer to the remainder of the `||`. We can simplify this further by memorizing the possible cases in a table before compilation. Clearly, there are two cases: when the cell is alive, the requirements to stay alive are distinct from the requirements to bring a dead cell to life. Within each of those cases, whether the cell will be living or dead in the next iteration will be a function of `living_neighbors`. For this reason, we will make a two-dimensional table: the first index is whether the cell is currently living or dead, and the second index will be the number of living neighbors. The boolean value in the table cell will be the new state of the cell (`false` for dead, `true` for living). By using this table, we no longer need to compute logical operations at runtime; instead, we will simply `return prev_and_alive_neighbors_to_next [is_this_cell_living][living_neighbors];`. The number of rows in our table will be 2 (to include both booleans, which are 0 and 1 as integers). Likewise, the number of columns in our table will be 9 (there are 9 cells in a

3×3 grid) and we exclude the cell on which we focus (it is not included in its own neighbors), so the valid columns will be 0, 1, ... 8, which consists of 9 possible column values.

The resulting implementation (Listing 4.8) shaves off still more runtime to reach 1000 generations in 5.786s. While the implementation from Listing 4.2, which did not include references, could be regarded as poorly written code, every subsequent listing was well-written `C++` code[16], and yet we've still managed to reach a speedup of over $8.8\times$ over what a good compiler could produce on our first reasonable implementation.

Listing 4.8: Even fewer `if` statements via table lookup. `if` statements are replaced with a table lookup and arithmetic.

```
#include "../Clock.hpp"
#include <iostream>

const bool prev_and_alive_neighbors_to_next[2][9] = {
  // previously dead:
  {false, false, false, true, false, false, false, false, false},
  // previously alive:
  {false, false, true, true, false, false, false, false, false}
};

bool advance_single(const bool*prev, const unsigned int R, const unsigned
    int C, int r, int c) {
  unsigned int living_neighbors = 0;
  // These loops will be unrolled beautifully:
  for (int i=-1; i<=1; ++i)
    for (int j=-1; j<=1; ++j)
      living_neighbors += prev[(r+i)*(C+2)+(c+j)];

  bool is_this_cell_living = prev[r*(C+2)+c];
  living_neighbors -= is_this_cell_living;
  return
      prev_and_alive_neighbors_to_next[is_this_cell_living][living_neighbors];
}

void advance(bool*cur, const bool*prev, const unsigned int R, const
```

[16]The fact that variables are passed by value by default in `C++` is often confusing to those more familiar with `Java` and `Python`, where pointers to objects are always passed[17].

[17]This is essentially how a reference is implemented by the compiler: a reference is a pointer that can only be initialized once and which does not need to be dereferenced every time you retrieve its value or modify it.

```
    unsigned int C) {
  for (unsigned int i=1; i<R+1; ++i)
    for (unsigned int j=1; j<C+1; ++j)
      cur[i*(C+2)+j] = advance_single(prev, R, C, i, j);
}

void print_board(const bool*board, const unsigned int R, const unsigned
    int C) {
  for (unsigned int i=1; i<R+1; ++i) {
    for (unsigned int j=1; j<C+1; ++j)
      std::cout << int(board[i*(C+2)+j]);
    std::cout << std::endl;
  }
}

int main(int argc, char**argv) {
  if (argc == 3) {
    const unsigned int R = 1<<10;
    const unsigned int C = 1<<11;

    unsigned int NUM_REPS = atoi(argv[1]);
    bool print = bool( atoi(argv[2]) );

    srand(0);
    bool*prev = (bool*)calloc((R+2)*(C+2),sizeof(bool));
    bool*cur = (bool*)calloc((R+2)*(C+2),sizeof(bool));

    for (unsigned int i=1; i<R+1; ++i)
      for (unsigned int j=1; j<C+1; ++j)
        cur[i*(C+2)+j] = (rand() % 2 == 0);

    if (print)
      print_board(cur, R, C);

    Clock c;

    for (unsigned int rep=0; rep<NUM_REPS; ++rep) {
      std::swap(cur, prev);
      advance(cur, prev, R, C);
    }

    c.ptock();

    if (print)
      print_board(cur, R, C);
  }
  else
    std::cerr << "usage game-of-life <generations> <0:don't print,
        1:print>" << std::endl;
  return 0;
}
```

4.8 Cutting the memory footprint in half

Note that all of the correct approaches used here use some sort of second matrix, and even if it is temporary, that means we still need enough RAM to allocate both matrices simultaneously. Just in case we are in a "memory-bound" scenario[18], we should consider the important unanswered question: can we somehow use a single matrix and eschew the use of a buffer? On the surface, it seems difficult: if we are to modify a cell's state, how will we be able to retrieve its old state when we subsequently modify its neighbors? Here we can exploit the fact that, as stated above, a `bool` really is treated like a small integer (usually 8 bits, which is the same size as an ascii `char`[19] on most architectures).[21].

Since we're already wasting some bits by using an 8-bit `bool`, we might as well use an `char`. We will simply treat this as an 8-bit integer[22]

Now that we have access to 8 bits, we can twiddle them so that we use the least-significant bit as the previous state of that cell and the second-least-significant bit as the new state of the cell. For example, the value 2 (`0b10` in

[18]A situation where the amount of available memory is the greatest limitation on performance or usability

[19]Unicode characters have a greater alphabet, and so use 16 bits or more.[20]

[20]Here the number of available emojis becomes very important.

[21]The reason for this has to do with the circuitry in a modern computer: it is simply not efficient to be able to access an individual bit at a time, and so minimal chunks of bits (usually bytes or more) are required. We can still modify individual bits by using "bit twiddling". This will be discussed in further detail in Chapter 5.

[22]Purists will caution that basic types like `char`, `int`, `long`, *etc.*, do not have strictly standard sizes; however, although this criticism would be technically correct, very little has changed between the time when 64-bit operating systems became ubiquitous and the present day when I'm writing this. If these values change, adjust accordingly to use the correct 8-bit type. As standardization improves, it will be reasonable to expect most compilers to implement type names like `int8` and `float32` in the future.

binary) would mean the cell was previously dead, but is now alive (meaning it must have had exactly 3 living neighbors in the previous iteration).

After advancing the cells one iteration, a final step can be added to make the new state into the current state: we simply divide by 2 (and round down, taking the floor). For example, `0b10` would become `0b1` when we divide by 2, `0b11` would become `0b1`, and `0b01` would become `0b0`. We can divide by 2 and floor (*i.e.*, perform integer division by 2) easily by simply bit shifting to the right by 1 bit: `val = val >> 1;` or the equivalent `val >>= 1;`.

When we sum `living_neighbors`, we will also only want to look at the least-significant bit; we will simply bitwise and it with `1`, `board[index] & 1`, and it will turn into either a `0` or a `1`, the same as it was before when we were treating a `bool` as an integer. This is why we assigned the least-significant bit as the current state (otherwise, we would have needed to shift every result inside the loop of neighbors).

The other difference is that we will directly modify the board in `advance_single`, rather than return a `bool` value; however, we want to take care to only modify the second most-significant bit. So if a cell that was dead is coming to life, we want to replace its value `0b0` with `0b10`. This can be accomplished by using the bitwise `|` operator[23] and the left bit shift operator `<<`. In this manner, we do not destroy the current state before all cells have been visited. After every cell has been visited, we will bit shift them all to the right so that their new states become current. The resulting code, shown in Listing 4.9, takes 9.644s to run 1000 iterations. Although this is slower than the fastest buffered method we tried, it uses half the memory.

Listing 4.9: An unbuffered game of life using bit twiddling. The least-significant bit is used for the current state of the cell and the second least-significant bit is used for the next state of the cell.

```
#include "../Clock.hpp"
#include <iostream>

// Code:
// 00 -> dead now, dead prev
// 01 -> dead now, alive prev
```

[23]Note that a single `&` and `|` perform bitwise operations on all bits, whereas the double `&&` and `||` perform the operations as if the arguments are `bool` types. *E.g.*, `2&1` will return `0`, whereas `2&&1` will return `1`, *i.e.*, `true`. Likewise `if (6&2) f();` will call function `f()` because the argument to the `if` statement is nonzero, even though it does not set the least-significant bit.

```
// 10 -> alive now, dead prev
// 11 -> alive now, alive prev
void advance_single(char*board, const unsigned int R, const unsigned int
    C, int r, int c) {
  // living_neighbors can never be >9, so a char is enough:
  unsigned char living_neighbors = 0;
  for (int i=-1; i<=1; ++i)
    for (int j=-1; j<=1; ++j)
      // Use &1 to look only at previous value (%2 would also work,
      // but may be slower):
      living_neighbors += (board[(r+i)*(C+2)+(c+j)] & 1);

  living_neighbors -= (board[r*(C+2)+c] & 1);

  board[r*(C+2)+c] = board[r*(C+2)+c] | ((living_neighbors == 3 || (
      (board[r*(C+2)+c] & 1) && living_neighbors >=2 && living_neighbors
      <= 3)) << 1);
}

void shift_current_to_prev(char*board, const unsigned int R, const
    unsigned int C) {
  for (unsigned int i=1; i<R+1; ++i)
    for (unsigned int j=1; j<C+1; ++j)
      board[i*(C+2)+j] >>= 1;
}

void advance(char*board, const unsigned int R, const unsigned int C) {
  for (unsigned int i=1; i<R+1; ++i)
    for (unsigned int j=1; j<C+1; ++j)
      advance_single(board, R, C, i, j);

  shift_current_to_prev(board, R, C);
}

void print_board(const char*board, const unsigned int R, const unsigned
    int C) {
  for (unsigned int i=1; i<R+1; ++i) {
    for (unsigned int j=1; j<C+1; ++j)
      std::cout << int(board[i*(C+2)+j]);
    std::cout << std::endl;
  }
}

int main(int argc, char**argv) {
  if (argc == 3) {
    const unsigned int R = 1<<10;
    const unsigned int C = 1<<11;

    unsigned int NUM_REPS = atoi(argv[1]);
    bool print = bool( atoi(argv[2]) );

    srand(0);
    char*board = (char*)calloc((R+2)*(C+2),sizeof(char));

    for (unsigned int i=1; i<R+1; ++i)
      for (unsigned int j=1; j<C+1; ++j)
        board[i*(C+2)+j] = (rand() % 2 == 0);

    if (print)
      print_board(board, R, C);

    Clock c;

    for (unsigned int rep=0; rep<NUM_REPS; ++rep)
      advance(board, R, C);

    c.ptock();

    if (print)
      print_board(board, R, C);
  }
  else
    std::cerr << "usage game-of-life <generations> <0:don't print,
        1:print>" << std::endl;
  return 0;
}
```

Of course, further speedups are possible: Whe we sum the living neighbors, we are summing over a 3×3 grid. When we sum the living neighbors of the next cell, we are summing over a 3×3 grid, with 6 of those cells overlapping the previous sum. It follows that we could sum the living neighbors first for all cells (reusing the compution in the overlapping 3×3 windows) and then use those living neighbors as the buffer[24]. Likewise, we've constructed the table to use the sum of 8 cells, which is performed by summing the 3×3 grid and subtracting out the center cell. This table could be modified to use the full sum of the 3×3 grid and thereby avoid the subtraction operation.

[24]Once we know the number of living neighbors for each cell, we can erase the boolean cell states without consequence, and so we no longer need an additional buffer.

And there are certainly other hacks in addition to these, which can be used to save time or memory.

Questions

1. [**Level 1**] Does the unbuffered game of life implementation from Listing 4.9 use memory as efficiently as possible (you may exclude the dead cell borders in this question)? What can be done to make it more efficient? What challenges would arise?

2. [**Level 2**] Modify Listing 4.9 so that it uses a table to update the `char` values (instead of using bitwise logic). Benchmark it. What influence does this have on the performance?

3. [**Level 3**] Propose a new technique for speeding up a game of life implementation even faster than done here. Is it buffered or unbuffered? Be specific about details of how the method works, and sketch out enough so that another student could implement your method from your description.

Projects

1. [**Level 2**] Write the fastest game of life implementation that you can. Test it with `diff` against the implementations in this chapter to make sure its output matches the correct result.

Chapter 5

Bit-Packed Strings and Hardware Parallelism

5.1 Nucleotide strings

DNA molecules are composed of chemical chains of G (guanine), A (adenine), T (thymine), and C (cytosine). DNA sequences (*i.e.*, the strings with characters in {'G', 'A', 'T', 'C'}) are studied because of their causal relationship in forming proteins[1]. Genomes[2] are studied because of their importance to biochemical discoveries[3], evolution[4], and medical research[5].

[1]Essentially, the most numerous of the functional biochemical units of biology, although other important functional elements exist.

[2]A genome is the sequence of all DNA in an organism. You can think of it like a series of encyclopedias, a collection of separate books, where each book is contiguous. In this book we will only focus on situations where there is one chromosome in the genome (*i.e.*, where there is only one book in the set of encyclopedias). Thus, it is sufficient here to assume that a genome can be stored as a single string rather than requiring a collection of strings (this is the case for some bacterial genomes, but it would not be correct for the human genome).

[3]*E.g.*, green fluorescent protein (GFP), the means by which we can visually observe components of a living cell

[4]*E.g.*, building phylogenies, the evolutionary trees showing the evolutionary distance between species

[5]*E.g.*, defects in "splicing" RNA, an intermediate product between DNA and proteins, which can lead to changes in some cells and therefore lead to severe disorders such as spinal muscular atrophy

5.2 Loading a genome from a text file

A genome is usually stored in a "fasta"[6] file; however, for simplicity, we will work with flat text files filled only with characters in the alphabet {'G', 'A', 'T', 'C'} (where case matters: 'g' will not be a valid character for our purposes).

First, we will write a short piece of code to load the DNA from a text file (Listing 5.1). On a sample genome of 4.4 million base pairs (*i.e.*, a string of length 4.4 million nucleotide characters), executing this code takes 0.06738s on average (estimated by repeating over 1024 replicates).

Listing 5.1: Loading a DNA string from file. Characters are simply appended to a string.

```cpp
#include "../Clock.hpp"
#include <fstream>
#include <string>

int main() {
  // A file of 4E6 G A T and C characters (the contents are
  // unimportant):
  std::ifstream fin("bigger.txt");
  std::string genome;

  Clock c;

  char base;
  while (fin >> base)
    genome += base;

  c.ptock();

  return 0;
}
```

[6]Originally pronounced "Fast A", but mentioned so frequently in computational biology that it has been replaced by the quicker-to-say "fasteh"[7].

[7]Pronounce "faster" with a Boston accent. Now how you like them apples?

5.3 Bit packing

Our DNA string uses only 4 values (G, A, T, and C); however, ASCII characters can store 256 possible values. 4 possible values can be written with 2 bits ($4 = 2^{2 \text{ bits}}$)[8], while `char` has 8 bits. This means for each DNA character, we could be storing 4 DNA characters if we used bits as efficiently as possible. *I.e.*, for each byte we actually use, a naive string will waste 3 bytes. As we will demonstrate in this chapter, bit packing saves substantial space, but it can be used to achieve large speedups as well.

It turns out, using bits efficiently is more complicated than it might seem. For instance, we cannot directly modify one bit of RAM. Instead, we need to load a block of bits from RAM, and then retrieve the particular bit in which we're interested.[9] As described in Chapter 4, a `bool` is actually an 8-bit integer rather than a single bit. So in order to pack 4 nucleotide characters into an 8-bit byte, we need directly manipulate the bits of the 8-bit `char`. This direct bit manipulation is called "bit twiddling".

We will focus on a few key operators: `&` performs bitwise and (`0b1101 & 0b1000` returns `0b1000`), `|` performs bitwise or (`0b1101 | 0b1000` returns `0b1101`), `<<` performs bit shifting to the left (`0b1101 << 2` returns `0b110100`), and `>>` performs bit shifting to the right (`0b1101 >> 1` returns `0b110`).[10] From these basic ingredients, we can perform very sophisticated operations.

[8]Half of a byte is 4 bits and is called a "nibble", a pun on a diminutive bite. As a student, I started calling the 2 bits necessary for a nucleotide a "nibblet" as a convenient shorthand.

[9]This is partly because of the lack of modularity and the many wires we would need in order to multiplex a particular bit– for this, we would need every single bit in RAM to be wired to the processor! The number of wires would physically limit the size of the chip and the parasitic capacitance from these many wires would require lower clock speeds. For this reason (as well as modularity), we build computers in a more hierarchical manner, multiplexing blocks of bits from RAM to the bus, which can then be accessed by the processor.

[10]Note that each of these operations also has a corresponding modifier operation: `x = x << 1` is equivalent to `x <<= 1`. Likewise, `x = x | 7` is equivalent to `x |= 7`.[11]

[11]Why do we use these shorter modifier forms? There are two reasons: it is less typing, and it can produce faster assembly code (we are specifically helping the compiler to notice that the destination of the operation is the same as one of the sources). This can help the compiler to use only two registers and also to hold the result in a register a while longer rather than writing it back to RAM.

5.4 An initial (and suboptimal) code

Before we begin, we will establish a code for the nucleotides: each nibblet should map bijectively to a nucleotide. Although this code is arbitrary (*i.g.*, G could be represented by `0b00` and T could be represented by `0b01` or vice versa). Later we will see that some codes have attractive properties, but for now we will begin with the code G=`0b00`, C=`0b01`, T=`0b10`, and A=`0b11`.

As explained above, by "bit packing" the values (*i.e.*, by bit twiddling carefully to be as efficient as possible with our available bits), we can store a string of 4 nucleotides in a single byte. As an example, we will bit pack the string "ACTC" using the code above.

Let us begin by initializing variables for our code: `g_code`, `c_code`, `t_code`, and `a_code`. Then we can use the `<<=` and `|=` operators to insert the characters "ACTC" (Listing 5.2). Note that when we print `x` to the screen, it will be interpreted as a `char`, and so it will print a single character. This basic approach can be optimized slightly (Listing 5.3); however, while Listing 5.3 will avoids temporary result variables, it must proceed sequentially and cannot apply all 4 operations simultaneously, even if the compiler and hardware would support it[12].

Although we could use `+=` instead of `|=`, `+=` will be slightly slower than `|=`. Consider: `|=` is a purely bitwise operation, meaning there is no cross-talk between different bits (*e.g.*, the most-significant bit and least-significant bits of the arguments can be operated on independently); however, addition with `+=` *does* allow information to be transmitted between bits (this occurs during "carry" operations). For this reason, circuitry for bitwise or is simpler and usually faster than circuitry for integer addition. By using `|=`, we are exploiting the fact that we know that the arguments have mutually exclusive values at each bit. This means that a carry operation is impossible, and thus we know in advance that `+=` will behave like `|=`. When these operations are performed very frequently inside a loop, operations like `|=` can be significantly faster than using `+=`. The same goes for the bit shift operators: `x*2` can be replaced with `x+x`, but even better, it can be replaced with `x<<1`[13]. For this reason, and for the sake of elegance, `int(pow(2.0, n))` (where `n` is an

[12]This is because the variable `x` is modified by each line, and its state (used by future lines) will not be known until the current line terminates. In compiler jargon, this is known as a "read after write" (RAW) dependency, and it may inhibit automated parallelism.

[13]In a base-10 system, multiplying by 10 shifts the digits left and appends a zero as the least-significant digit. In binary, multiplying by 2 has the same effect.

`int`) is often written with the more pleasing `1<<n`. Likewise, integer division `n/2` can be performed as `n>>1`. Compilers are currently smart enough to figure out some of these tricks, but they are not smart enough to figure out everything[14].

Listing 5.2: Packing 4 nucleotides into a byte and printing. The nucleotide string "ACTC" is packed into a single 8-bit `char` and then printed in binary using `std::bitset`.

```cpp
#include <iostream>
#include <bitset>

int main() {
  char g_code=0b00;
  char c_code=0b01;
  char t_code=0b10;
  char a_code=0b11;

  char actc = 0;
  actc |= (a_code << 6); // 0b11000000
  actc |= (c_code << 4); // 0b11010000
  actc |= (t_code << 2); // 0b11011000
  actc |= c_code;      // 0b11011001

  // not helpful: prints as single character:
  std::cout << actc << std::endl;

  // slightly helpful: prints as single int value:
  std::cout << int(actc) << std::endl;

  // useful for debugging: prints as binary:
  std::cout << std::bitset<8>(actc) << std::endl;

  return 0;
}
```

Listing 5.3: Optimized version of Listing 5.2. Some temporary values (the results of the `(a_code << 6)` operations) can be avoided. This is reminiscent of Horner's rule, by which a degree-n polynomial is evaluated in $O(n)$ rather than $O(n^2)$.

[14]And besides, as long as it doesn't detract from the readability and cleanliness of our code, we might as well make these insights explicit.

```
#include <iostream>
#include <bitset>

int main() {
  char g_code=0b00;
  char c_code=0b01;
  char t_code=0b10;
  char a_code=0b11;

  char actc = 0;
  actc |= a_code; // 0b00000011
  actc <<= 2;    // 0b00001100
  actc |= c_code; // 0b00001101
  actc <<= 2;    // 0b00110100
  actc |= t_code; // 0b00110110
  actc <<= 2;    // 0b11011000
  actc |= c_code; // 0b11011001

  // not helpful: prints as single character:
  std::cout << actc << std::endl;

  // slightly helpful: prints as single int value:
  std::cout << int(actc) << std::endl;

  // useful for debugging: prints as binary:
  std::cout << std::bitset<8>(actc) << std::endl;

  return 0;
}
```

5.5 Bit packing a genome

We can now put together what we've just performed on a single byte and perform it on an entire genome (Listing 5.4). Note that this early version does not worry about subtleties where the genome string is not divisible by the block size[15]. Instead, this early version will cut off the final characters if the total number of characters is not divisible by the block size. Here we change from using blocks of type `char` to blocks of type `unsigned long`. This means that we now pack 32 nucleotides in a block (an `unsigned long`

[15]See the Donald Knuth quote in Chapter 2.

has 64 bits, which holds 32 nibblets) rather than 4 nucleotides in a block.[16] Later in this chapter we will see the benefit of packing into type `unsigned long` rather than `char`.

Listing 5.4: Bit packing a genome (truncated and with dead code). The outer loop with variable `block` iterates over all blocks, and the inner loop over variable `j` enumerates over each nucleotide packed in a block, using a method similar to that of Listing 5.3. Note that this implementation oversimplifies the problem by assuming the genome divides evenly by the block size. The implementation also suffers from unwanted dead code elimination, making benchmarking results inaccurate.

```
#include "../Clock.hpp"
#include <fstream>
#include <string>
#include <bitset>

constexpr unsigned int NUCLEOTIDES_PER_BLOCK = sizeof(unsigned long) * 8
    / 2;

unsigned char nucleotide_code(char nuc) {
  if (nuc == 'G')
    return 0;
  if (nuc == 'C')
    return 1;
  if (nuc == 'T')
    return 2;
  if (nuc == 'A')
    return 3;
  return -1;
}

unsigned long * bit_packed_assume_perfectly_divisible_into_blocks(const
    std::string & genome) {
  unsigned long num_blocks = genome.size() / NUCLEOTIDES_PER_BLOCK;
  // Assumes genome is evenly divisible into blocks (if not, will cut
  // off the trailing genome.size() % NUCLEOTIDES_PER_BLOCK nucleotides).

  unsigned long * result = (unsigned long*)calloc(num_blocks,
      sizeof(unsigned long));
```

[16]Note that this will be significantly less space efficient when we need to store many strings that contain 4 nucleotides or fewer; however, our use-case is for large genomes.

```
  unsigned long i=0;
  for (unsigned long block=0; block<num_blocks; ++block) {
    unsigned long next_block = 0;

    for (unsigned char j=0; j<NUCLEOTIDES_PER_BLOCK; ++j, ++i) {
      next_block <<= 2;

      next_block |= nucleotide_code(genome[i]);
    }
    result[block] = next_block;
  }
  return result;
}

int main() {
  // A file of 4E6 G A T and C characters (the contents are
  // unimportant):
  std::ifstream fin("bigger.txt");
  std::string genome;

  char base;
  while (fin >> base)
    genome += base;

  Clock c;

  unsigned long*packed =
      bit_packed_assume_perfectly_divisible_into_blocks(genome);

  c.ptock();

  // The variable packed is never used; this code will appear to run
      instantly!

  return 0;
}
```

Unfortunately, a quick benchmark of the implementation on a genome of our choice reports a runtime of 0s; it is the result of unwanted dead code elimination, which essentially erases our benchmark. Listing 5.5 fixes this dead code issue in the usual style by printing the result (here it is sufficient to simply print the first few characters of the bit-packed result[17]).

[17]The compiler is not yet smart enough to notice that we only use those first few

Listing 5.5: Bit packing a genome (improved version of Listing 5.4). By printing the first few blocks of the result, unwanted dead code elimination is avoided. Now that the implementation works for benchmarking, we also improve the bit packing routine so that it properly handles cases where the genome size is not evenly divisible by the block size.

```
#include "../Clock.hpp"
#include <fstream>
#include <string>
#include <bitset>

// 8 bits in a byte (sizeof returns number of bytes) and we need 2
// bits per nucleotide (there are 4, and 2^(2 bits) = 4):
constexpr unsigned int NUCLEOTIDES_PER_BLOCK = sizeof(unsigned long) * 8
    / 2;

unsigned char nucleotide_code(char nuc) {
  if (nuc == 'G')
    return 0;
  if (nuc == 'C')
    return 1;
  if (nuc == 'T')
    return 2;
  if (nuc == 'A')
    return 3;
  return -1;
}

unsigned long * bit_packed(const std::string & genome) {
  unsigned long num_blocks = genome.size() / NUCLEOTIDES_PER_BLOCK;
  // If it doesn't divide evenly (* should be faster than %), add one
  // more block:
  if ( num_blocks * NUCLEOTIDES_PER_BLOCK != genome.size() )
    ++num_blocks;

  unsigned long * result = (unsigned long*)calloc(num_blocks,
      sizeof(unsigned long));

  unsigned long i=0;
  for (unsigned long block=0; block<num_blocks; ++block) {
    unsigned long next_block = 0;
```

characters and thus *only* bit pack those. If we weren't using dynamic arrays allocated at runtime using pointers, it may be feasible that the compiler could realize this in the near future.

```
    for (unsigned char j=0; j<NUCLEOTIDES_PER_BLOCK; ++j, ++i) {
      next_block <<= 2;

      if (i < genome.size())
        next_block |= nucleotide_code(genome[i]);
    }
    result[block] = next_block;
  }
  return result;
}

int main() {
  // A file of 4E6 G A T and C characters (the contents are
  // unimportant):
  std::ifstream fin("bigger.txt");
  std::string genome;

  char base;
  while (fin >> base)
    genome += base;

  Clock c;

  unsigned long*packed = bit_packed(genome);

  c.ptock();

  for (unsigned int i=0; i<2*NUCLEOTIDES_PER_BLOCK; ++i)
    std::cout << genome[i] << " ";
  std::cout << "..." << std::endl;

  for (unsigned int i=0; i<2; ++i)
    std::cout << std::bitset<sizeof(unsigned long)*8>(packed[i]);
  std::cout << "..." << std::endl;

  return 0;
}
```

Benchmarking 1024 replicates of our bit packing routine takes on average 0.03639s. Note that we must modify our benchmarking script to ignore the lines that include the results on the first few characters (*i.e.*, the code we used to prevent unwanted dead code elimination). This can be accomplished by running `./a.out | head -1` instead of `./a.out`.

5.6 Eliminating `if` statements

As we saw in Chapter 4, eliminating `if` statements, particularly those inside loops, is good approach for improving performance. Here we see another opportunity to do this: First, we can replace the function `nucleotide_code` with a table of 256 `char` values. Instead of `nucleotide_code('A')` returning 3, we simply initialize our table so that `nuc_to_code['A'] = 3`. Note that with this table, we are using the fact that `char` types are really 8-bit integers underneath, and so `nuc_to_code['A']` is the same as `nuc_to_code[65]`. There are 256 possible ASCII characters, and so we must fill our table at every index; however, we know in advance that the only valid queries to this table should be `nuc_to_code['G']`, `nuc_to_code['C']`, `nuc_to_code['T']`, and `nuc_to_code['A']`. Thus, we will initialize the indices corresponding to all non-nucleotide characters with an invalid value (*i.e.*, a value not in our nucleotide code, so that whenever we see this value, we know something is wrong), 255.

When constructing this table, we have multiple options; we could allocate a table `char*nuc_to_code = new char[256];` and then initialize `nuc_to_code['A'] = 3;`, *etc.* at the beginning of `main`. However, this approach has some undesirable (albeit not dire) consequences: First, this requires us to initialize this array manually before it is used. This can result in easy-to-make errors, *e.g.*, if we forget to initialize the table or if we accidentally initialize the table after it's used. Second, this initialization code must run every time our program starts. Although it is not so slow, it may nonetheless increase the overall runtime of our program.

An alternative would be to initialize the array at compile time `const unsigned char nuc_to_code[] = { ...};`. The downside is that we must type in each of the 256 values. Fortunately, we can write a very short program to write this table code for us (Listing 5.6). This approach also has an additional advantage: because we've declared our array to be `const`, the compiler knows that it should never change, and so it may be possible to perform optimizations in some cases, *e.g.*, directly replacing `nuc_to_code['A']` with `3` at compile time and eschewing the array lookup operation at runtime.

Listing 5.6: Generating a table mapping uppercase nucleotide characters to their integer codes. The output of this program can be included in another `C++` program. It only needs to be run once, pasting the output into the top of another `.cpp` file.

```
#include <iostream>

int nucleotide_code(char nuc) {
  if (nuc == 'G')
    return 0;
  if (nuc == 'C')
    return 1;
  if (nuc == 'T')
    return 2;
  if (nuc == 'A')
    return 3;
  return -1;
}

int main() {
  std::cout << "const unsigned char nuc_to_code[] = {";
  for (unsigned int i=0; i<256; ++i) {
    std::cout << int((unsigned char)(nucleotide_code(char(i))));
    if (i != 255)
      std::cout << ", ";
  }
  std::cout << "};" << std::endl;

  return 0;
}
```

By pasting the output of Listing 5.6 into a `.cpp` file, we can create a new implementation that uses a table instead of the `if` statement-based `nuc_to_code` function (Listing 5.7).

Listing 5.7: Improved version of Listing 5.5. The `nuc_to_code` function is replaced by a table. This table is produced by running the code in Listing 5.6.

```
#include "../Clock.hpp"
#include <fstream>
#include <string>
#include <bitset>

// G -> 0
// C -> 1
// T -> 2
// A -> 3
// -1uc (i.e., 255) otherwise

// This table is generated automatically as the output of
```

```
    generate_table.cpp:
const unsigned char nuc_to_code[] = {255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 3, 255, 1, 255, 255, 255, 0, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 2, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255};

constexpr unsigned int NUCLEOTIDES_PER_BLOCK = sizeof(unsigned long) * 8
    / 2;

unsigned long * bit_packed(const std::string & genome) {
  unsigned long num_blocks = genome.size() / NUCLEOTIDES_PER_BLOCK;
  // If it doesn't divide evenly (* should be faster than %), add one
  // more block:
  if ( num_blocks * NUCLEOTIDES_PER_BLOCK != genome.size() )
    ++num_blocks;

  unsigned long * result = (unsigned long*)calloc(num_blocks,
      sizeof(unsigned long));

  unsigned long i=0;
  for (unsigned long block=0; block<num_blocks; ++block) {
    unsigned long next_block = 0;

    for (unsigned char j=0; j<NUCLEOTIDES_PER_BLOCK; ++j, ++i) {
      next_block <<= 2;

      if (i < genome.size())
        next_block |= nuc_to_code[ genome[i] ];
    }
    result[block] = next_block;
  }
  return result;
}

int main() {
  // A file of 4E6 G A T and C characters (the contents are
  // unimportant):
  std::ifstream fin("bigger.txt");
  std::string genome;

  char base;
  while (fin >> base)
    genome += base;

  Clock c;

  unsigned long*packed = bit_packed(genome);

  c.ptock();

  for (unsigned int i=0; i<2*NUCLEOTIDES_PER_BLOCK; ++i)
    std::cout << genome[i] << " ";
  std::cout << "..." << std::endl;

  for (unsigned int i=0; i<2; ++i)
    std::cout << std::bitset<sizeof(unsigned long)*8>(packed[i]);
  std::cout << "..." << std::endl;

  return 0;
}
```

Benchmarking 1024 repetitions of Listing 5.5 takes 0.005386s on average, a $> 6\times$ speedup over the code in Listing 5.5. However, there is still an opportunity to remove a `if` statement within a loop: the line `if (i < genome.size())` is executed thousands of times, even though it will only be false a very small number of times, and even then, only in the final block. Here we will manually "peel" the final iteration off of the outer `for` loop and manually perform that final iteration after the loop. This enables us to ignore the case where `i >= genome.size()` everywhere inside the loop (Listing 5.8).

Listing 5.8: Improved version of Listing 5.7. The final iteration of the loop `for (unsigned long block=0; block<num_blocks; ++block)` has been

peeled off and manually performed after the loop. The allows us to ignore the case where `i >= genome.size()` throughout that loop.

```
#include "../Clock.hpp"
#include <fstream>
#include <string>
#include <bitset>

// G -> 0
// C -> 1
// T -> 2
// A -> 3
// -1uc (i.e., 255) otherwise
const unsigned char nuc_to_code[] = {255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 3, 255, 1, 255, 255, 255, 0, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 2, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255};

constexpr unsigned int NUCLEOTIDES_PER_BLOCK = sizeof(unsigned long) * 8
    / 2;

unsigned long * bit_packed(const std::string & genome) {
  unsigned long num_blocks = genome.size() / NUCLEOTIDES_PER_BLOCK;
  // If it doesn't divide evenly (* should be faster than %), add one
  // more block:
  if ( num_blocks * NUCLEOTIDES_PER_BLOCK != genome.size() )
    ++num_blocks;

  unsigned long * result = (unsigned long*)calloc(num_blocks,
      sizeof(unsigned long));
  unsigned long i=0;
  // Go through all but the final block:
  for (unsigned long block=0; block<num_blocks-1; ++block) {
    unsigned long next_block = 0;

    for (unsigned char j=0; j<NUCLEOTIDES_PER_BLOCK; ++j, ++i) {
      next_block <<= 2;

      next_block |= nuc_to_code[ genome[i] ];
    }
    result[block] = next_block;
  }

  // Perform final block separately, and only check boundary of genome
  // in final block:
  unsigned long next_block = 0;
  for (unsigned char j=0; j<NUCLEOTIDES_PER_BLOCK; ++j, ++i) {
    next_block <<= 2;

    // This is the final block: check to make sure the boundaries are met:
    if (i < genome.size())
      next_block |= nuc_to_code[ genome[i] ];
  }
  result[num_blocks-1] = next_block;

  return result;
}

int main() {
  // A file of 4E6 G A T and C characters (the contents are
  // unimportant):
  std::ifstream fin("bigger.txt");
  std::string genome;

  char base;
  while (fin >> base)
    genome += base;

  Clock c;

  unsigned long*packed = bit_packed(genome);

  c.ptock();

  for (unsigned int i=0; i<2*NUCLEOTIDES_PER_BLOCK; ++i)
```

```
    std::cout << genome[i] << " ";
  std::cout << "..." << std::endl;

  for (unsigned int i=0; i<2; ++i)
    std::cout << std::bitset<sizeof(unsigned long)*8>(packed[i]);
  std::cout << "..." << std::endl;

  return 0;
}
```

In 1024 replicate trials, the average runtime for Listing 5.8 is 0.003620s, a further $> 1.48\times$ speedup over Listing 5.7.

5.7 Nucleotide complements

A sequence most efficiently binds its complementary DNA sequence. 'A' binds with 'T' (and vice versa) and 'C' binds with 'G' (and vice versa).[18]

Listing 5.9 computes the nucleotide complement of a genome. Over 1024 replicate trials, the average runtime (of complementation only, not of loading the genome) is 0.04262s.

Listing 5.9: Naive complement of a genome. Each nucleotide is replaced with its complementary character.

```
#include "../Clock.hpp"
#include <fstream>
#include <string>

constexpr unsigned int NUCLEOTIDES_PER_BLOCK = sizeof(unsigned long) * 8
    / 2;

char nucleotide_complement(char nuc) {
  if (nuc == 'G')
    return 'C';
  if (nuc == 'C')
    return 'G';
```

[18]In the cell, DNA is usually double stranded, meaning you almost never see a single DNA molecule without also seeing it bonded to its complementary molecule.[19]

[19]If one strand starts with a 'T' and the other starts with an 'A', which do we use for the genome then? It's arbitrary; one strand becomes known as the "sense" strand, while the other is the "antisense" strand. If you know one strand's sequence, you should easily be able to find the sequence of the complementary strand.

```
  if (nuc == 'A')
    return 'T';
  if (nuc == 'T')
    return 'A';
  return -1;
}

std::string complement(const std::string & genome) {
  std::string result;

  for (char c : genome)
    result += nucleotide_complement(c);

  return result;
}

int main() {
  // A file of 4E6 G A T and C characters (the contents are
  // unimportant):
  std::ifstream fin("bigger.txt");
  std::string genome;

  Clock c;

  char base;
  while (fin >> base)
    genome += base;

  std::string comp = complement(genome);

  c.ptock();

  for (unsigned int i=0; i<2*NUCLEOTIDES_PER_BLOCK; ++i)
    std::cout << genome[i] << " ";
  std::cout << "..." << std::endl;
  for (unsigned int i=0; i<2*NUCLEOTIDES_PER_BLOCK; ++i)
    std::cout << comp[i] << " ";
  std::cout << "..." << std::endl;

  return 0;
}
```

To speed up Listing 5.9, we could attempt to remove some `if` statements from the `nucleotide_complement` function; however, we can do far better by using the bit-packed form.

5.8 Hardware parallelism[20]

Let us first consider the possibility of performing complementation as a bitwise operation. Each nibblet can be complemented separately from every other nibblet, and so bitwise complementation certainly feels like an option; however, the code that we chose in Section 5.4 does not easily lend itself to complementation. But that first code was arbitrarily chosen, and we are free to choose any code we like. For this reason, we can choose a code where bitwise not (*i.e.*, ∼ operator) can be used to perform complementation. Let G=`0b00`. Then C=∼`0b00`=`0b11`. Likewise, let A=`0b01`. Then T=∼`0b01`=`0b10`.

First, we will remake our table generation code (Listing 5.10). Second, we will use that code to perform the bit-packed complement by simply performing bitwise not ∼ on every block. Here we see the reason for using blocks of type `unsigned long` rather than of type `char`: using the ∼ operator on a `char` computes a 4-nucleotide complement, whereas using the ∼ operator on an `unsigned long` type computes a 32 nucleotide complement. Since bitwise not is inexpensive (like `&` and `|`, it is a true bitwise operation and carries no information between bits in different positions), this can result in a large speedup. Listing 5.11 performs the bit-packed complement in this manner.

Listing 5.10: Generating a table using an improved nucleotide code (revised version of Listing 5.6). The output of this program is used in another `.cpp` file.

```
#include <iostream>

// Alternate ordering makes it so that we can complement with the ~
    operator:
int nucleotide_code(char nuc) {
  if (nuc == 'G')
    return 0;
  if (nuc == 'A')
    return 1;
  if (nuc == 'T')
    return 2;
  if (nuc == 'C')
    return 3;
  return -1;
```

[20] "Why... soo... *serial*?"[21]

[21] "Did I ever tell you... how I got these fast runtimes?"

```
}

int main() {
  std::cout << "const unsigned char nuc_to_code[] = {";
  for (unsigned int i=0; i<256; ++i) {
    std::cout << int((unsigned char)(nucleotide_code(char(i))));
    if (i != 255)
      std::cout << ", ";
  }
  std::cout << "};" << std::endl;

  return 0;
}
```

Listing 5.11: Bit-packed complement. The method simply traverses the bit-packed array of `unsigned long` types and makes a copy that has been bitwise notted using the $\sim$ operator.

```
#include "../Clock.hpp"
#include <fstream>
#include <string>
#include <bitset>

// Alternate ordering makes it so that we can complement with the ~
    operator:
// G -> 0 = 00
// A -> 1 = 01
// T -> 2 = 10
// C -> 3 = 11
// -1uc (i.e., 255) otherwise
const unsigned char nuc_to_code[] = {255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 1, 255, 3, 255, 255, 255, 0, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 2, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255};

constexpr unsigned int NUCLEOTIDES_PER_BLOCK = sizeof(unsigned long) * 8
    / 2;

// Rewritten to modify reference variables so that both result and
// num_blocks are provided (a return value could only return 1 unless
// we wrap it in std::pair<>):
void bit_pack(unsigned long * & result, unsigned long & num_blocks, const
    std::string & genome) {
  num_blocks = genome.size() / NUCLEOTIDES_PER_BLOCK;
  // If it doesn't divide easily (* should be faster than %), add one
  // more block:
  if ( num_blocks * NUCLEOTIDES_PER_BLOCK != genome.size() )
    ++num_blocks;

  result = (unsigned long*)calloc(num_blocks, sizeof(unsigned long));

  unsigned long i=0;
  // Go through all but the final block:
  for (unsigned long block=0; block<num_blocks-1; ++block) {
    unsigned long next_block = 0;

    for (unsigned char j=0; j<NUCLEOTIDES_PER_BLOCK; ++j, ++i) {
      next_block <<= 2;

      next_block |= nuc_to_code[ genome[i] ];
    }
    result[block] = next_block;
  }

  // Perform final block separately, and only check boundary of genome
  // in final block:
  unsigned long next_block = 0;
  for (unsigned char j=0; j<NUCLEOTIDES_PER_BLOCK; ++j, ++i) {
    next_block <<= 2;

    // This is the final block: check to make sure the boundaries are met:
    if (i < genome.size())
      next_block |= nuc_to_code[ genome[i] ];
  }
  result[num_blocks-1] = next_block;
}

unsigned long* packed_complement(const unsigned long*packed_genome,
    unsigned long num_blocks) {
  unsigned long * result = (unsigned long*)calloc(num_blocks,
      sizeof(unsigned long));
  for (unsigned long block=0; block<num_blocks; ++block)
    // With our new ordering of the nucleotides, bit-wise complement
    // will perform nucleotide complement!
    result[block] = ~packed_genome[block];

  // The above will also flip any unused bits in the last block, but
  // those are to be ignored anyway.
  return result;
}

int main() {
  // A file of 4E6 G A T and C characters (the contents are
  // unimportant):
  std::ifstream fin("bigger.txt");
  std::string genome;

  char base;
  while (fin >> base)
    genome += base;

  unsigned long*packed;
  unsigned long num_blocks;
  bit_pack(packed, num_blocks, genome);

  Clock c;

  unsigned long*comp = packed_complement(packed, num_blocks);

  c.ptock();

  for (unsigned int i=0; i<2*NUCLEOTIDES_PER_BLOCK; ++i)
    std::cout << genome[i] << " ";
  std::cout << "..." << std::endl;

  for (unsigned int i=0; i<2; ++i)
    std::cout << std::bitset<8*sizeof(unsigned long)>(packed[i]);
  std::cout << "..." << std::endl;
  for (unsigned int i=0; i<2; ++i)
    std::cout << std::bitset<8*sizeof(unsigned long)>(comp[i]);
  std::cout << "..." << std::endl;

  char ordered_nucleoties[] = {'G','A','T','C'}; // Order matches code
      above
  for (unsigned int i=0; i<2; ++i) {
    unsigned long mask = -1ul & ~((-1ul)>>2); // 11000....
    for (unsigned char j=0; j<NUCLEOTIDES_PER_BLOCK; ++j) {
      unsigned long shifted_bit_pair = comp[i] & mask;
      std::cout << ordered_nucleoties[ shifted_bit_pair >>
          2*(NUCLEOTIDES_PER_BLOCK-j-1) ] << " ";
      mask >>= 2;
    }
  }
  std::cout << "..." << std::endl;

  return 0;
}
```

In 1024 replicate trials, the bit-packed complement in Listing 5.11 takes only 0.0003611s. This is a $> 118\times$ speedup over Listing 5.9.

5.9 Vectorization

Variants of this kind of hardware-level parallelism can be leveraged for many types of tasks to achieve a substantial speedup. For instance, if a CPU can process `long` type without emulation (via 64-bit registers that can be operated on by the ALU[22] in one assembly operation), then it may be possible to pack 8 8-bit `char` types into a single 64-bit register. If the CPU supports it, an addition operation on the 64-bit register could, for example, suppress the carry operations on every eight bit (*i.e.*, the 7 least-significant bits 0 through 6 can carry out when adding, but bit 7 cannot carry to bit 8 as usual). If the chip supports this operation[23], then it can be used to perform 8 `char` additions simultaneously. This is the idea behind vectorization, a kind

[22]The "arithmetic and logic unit", the part of the CPU that does the mathematical operations like adding `int` types, multiplying `double` types, *etc.*

[23]Such operations do add complexity to the chip, but they are not inherently difficult: suppressing carry operations should make addition more simple, as discussed above regarding why `|=` is easier than `+=`.

of parallelization that is based on packing multiple operands into a register and operating on them simultaneously[24].

Vectorizing in this manner is sometimes referred to as "single instruction multiple data" (SIMD), meaning that a single type of operation (*e.g.*, addition) is specified, but is applied to several separate independent arguments, which have been bit packed into a large register. These operations are commonly referred to by the popular proprietary implementations on Intel processors: SSE and the more modern AVX.

Hardware limitations are the only real limit for this approach: if you have a machine with a true 1024-bit ALU and support for the corresponding vectorizing operations, then you can achieve a large speedup[25]. Current consumer hardware is on the cusp of supporting AVX512[26], meaning 8 independent operations on 64-bit `double` types could be performed simultaneously. Hardware can be very important to practical performance in these cases: A 128-bit register can only hold 2 `double` types, and so vectorization might not even be beneficial for `double` (because there may be some small overhead cost[27] that will outweigh the more muted benefit); vectorization with a 128-bit register may still prove beneficial for 32-bit `float` types or for 16-bit `short` integer types.

5.10 Loading a genome from a text file (revisited)

It is not uncommon that optimized implementations of methods like bit-packed nucleotide complement will have their performance limited by the simple but deceptively slow step of loading the data from a file. While reading from disk isn't particularly fast, sequential disk access can be quite

[24]This is in contrast with thread-based parallelization featured in multicore chips and GPUs, where, for example, many operations may be performed by separate ALUs at the same time

[25]Assuming the addition of all that hardware didn't trash the chip's clock speed or otherwise impair its performance

[26]Hardware evolves quickly, and no doubt, if you are reading this in the future, AVX65536 may be commonplace; if so, don't be a hater: close the SnapGhost app on your flying, 8G hover scooter for a moment and take some time to reflect that it was once the past.

[27]*E.g.*, copying into special-purpose vectorized registers

efficient[28]; instead, it is because of how we modify the string: frequently appending to a string can be slow, because it will force the string to resize (which may cost a reallocation and copying). Appending to arrays will be discussed in greater detail in Chapter 6.

Even more efficient would be to store the file in its bit-packed form (writing the `unsigned long` variables in binary format so that they can be serialized in directly as an array rather than in the character-by-character manner used to read ASCII with the `std::ifstream >> char` operator; however, in this case, we would need to start with a non-human readable file, and this will be most useful for very large data sets and data centers where economies of scale incentivize even minor decreases in disk usage. We will discuss direct memory access and serialization of such binary files in Chapter 14.

Questions

1. [**Level 1**] What are some advantages and disadvantages of the kind of hardware parallelism discussed in this chapter (compared to, for example, writing code to use multiple threads on a multicore processor)?

2. [**Level 2**] Re-implement Listing 5.11 using blocks of `char` instead of blocks of `unsigned long`. Does the performance change? Re-implement again using unsigned long long instead of unsigned long. Does performance change? Explain why you think this would happen.

Projects

1. [**Level 2**] Write a program that loads a genome from a file (the first command line argument is the genome file name) and the file name of a file full of short DNA reads (second command line argument). The file of short DNA reads will have one DNA sequence per line, and each short read will have length ≤ 32. The program will process each nucleotide string in order and will print all indices where that short string occurs in the genome. Do not use multithreading and do not use

[28]This is largely the result of caching. Caching in RAM will be discussed in greater detail in Chapter 9. Disk caching is similar; just as we benefit from accessing RAM in a sequential manner, so too do we benefit from accessing disk in a sequential manner.

any regular expression libraries or anything libraries not `#include`d in this chapter. How fast can you make your code?

2. [**Level 3**] The same as Level 2, but you must search using only bit-packed strings (the files should still be loaded from standard ASCII format and then converted to bit-packed strings).

Chapter 6

Lists, Vectors, and Memory

6.1 The case for linked lists

While arrays use a single block of memory, linked lists use separate blocks of memory, which each include a pointer to the next block. Arrays have efficient random access[1], but linked lists, not using a single block of memory, do not have efficient random access; however, because of they are composed of separate blocks of memory, it is possible to insert data into the middle or onto the end of a linked list in $O(1)$. In contrast, once an array is allocated, it isn't possible to directly resize it (the memory past the end of the array may be used by something else).

On paper, when data is inserted into the middle or appended, linked lists are the logical choice (as long as frequent random accesses are not performed). But in practice, arrays are very efficient. One reason for this is that arrays are contiguous in memory, and so they cache very efficiently (cache will be discussed in greater depth in Chapter 9). Another reason for this is the minimalism of arrays: Storing 1024 8-bit `char` types in an array costs 1KB (1024 bytes). Meanwhile, every element in a linked list needs to store a 64-bit pointer[2] to the next element. Thus storing 1024 8-bit `char` types in a singly linked list costs $1024 \times (8+64)$ bits $= 1024 \times (1+8)$ bytes = 9KB (this is a $9\times$ the memory used by the array). A doubly linked list (where each block has a pointer to both the previous and the next element)[3] would require 1024

[1]The ability to jump to the element at an arbitrary index in $O(1)$

[2]In nearly all modern 64-bit operating systems, a pointer uses 64 bits

[3]This is the most general and flexible type of linked list. Without a pointer to the

$\times$ (1+16) bytes = 17KB, or 17$\times$ the memory used by the array.

6.2 Vectors: resizable arrays

Inserting data into the middle of an array requires shifting all of the displaced data to the right in order to make room[4]. But if there were a way to more efficiently resize an array, appending to the end of the array would be possible, and would allow us to have the efficiency and cache performance of an array but with the flexibility of a linked list. Resizable arrays of this form are exactly what vectors are, *e.g.*, the `std::vector` we occasionally used in previous chapters.

How do vectors work?[5] First let's assume that there is no way to "grow" the allocation of the array to a new size in $O(1)$ time (because, as said above, the memory after the end of the array may be occupied). We will revisit this assumption later. With this limitation, resizing means something clear: allocating a new block of memory to use as a new, larger array; however, this new block of memory will be empty, and so the existing elements will need to be copied into it before the new element is appended.

This leads to a problem. The first append operation will cost $\propto$ 1 step (allocate a new array of size 1, copy 0 existing elements into it and then append the new element). The second append operation will cost $\propto$ 2 steps: allocate a new array of size 2, copy in the existing 1 element, append the new element). The third append operation will cost $\propto$ 3 steps and so on, so that the cost of appending n successive elements (*e.g.*, when reading data from a file and appending it onto a vector), will be $\in O(1+2+\cdots+n) = O(n^2)$. In

previous element, insertion into the middle of the list becomes more complicated, because you would need to know the element *immediately before* the insertion point rather than the insertion point itself. The same goes for removing elements from the middle of a linked list: it's easier when the list is doubly linked, although some use cases may still be $O(1)$ with a singly linked list.

[4]For this reason, inserting data into the middle of an array *is right out*. If you want to insert into the middle of something easily and you don't want to use a linked list, then go climb a tree.

[5]Right now you're probably thinking, "It's not exactly rocket surgery, is it? Aren't vectors the sort of code that's sure to already be implemented well and turned into a cheap commodity, the kind of code that no one in their right mind would implement themselves?"[6]

[6]Well lucky for you, smart people are often a little crazy, so let's do this!

contrast, a linked list could append n successive times in $O(n)$, and so this means of implementing a vector would be unacceptable.

If we grow the vector by a constant k elements each time we run out of space, then we can guarantee fewer resize operations, but we will end up with the same problem, and the runtime will still be $\in O(n^2)$:

$$\begin{aligned}
O(k + \underbrace{1 + 1 + \cdots + 1}_{k \text{ operations}} + 2k + \underbrace{1 + 1 + \cdots + 1}_{2k \text{ operations}} + \cdots) = \\
O(k + 2k + 3k + \cdots + n) = \\
O\left(\sum_{i=1}^{\frac{n}{k}} ik\right) = \\
O\left(k\frac{n^2}{k^2}\right) = \\
O\left(\frac{n^2}{k}\right),
\end{aligned}$$

which will be $\in O(n^2)$ whenever k is a constant. When $k \in \theta(n)$, then this does achieve a total runtime in $O(n)$; however, this is essentially the same as knowing the size of the array *a priori*, which is a way of essentially using a fixed-sized array rather than a vector. So far, we have found no legitimate resizing scheme that will perform n append operations in $O(n)$.

6.3 Vector resizing schemes

However, consider the case where the i^{th} resize operation grows by some amount a_i and occurs when the current size is s_i[7]. We can show that in order to have a total cost of $O(n)$ for n append operations, then there are constraints that bind the growth amount a_i in terms of the size before resizing s_i.

Consider the worst-case instance for overhead where you resize, but where there are no further append operations. Clearly, if you resize to a very large

[7]Note that i is the number of resize operations so far, *not* the number of append operations performed so far.

array and then no more append operations follow, this can be very inefficient: the overhead from the resizing would be much larger than the actual amount of useful work being done. When s_i is the size of the array before the final resize operation, then the worst-case scenario is where $n = s_i + 1$, meaning there is only one append operation that forces the resize. The cost of the final resize operation is $\in O(s_i + a_i)$, because the new array will be allocated with size $s_{i+1} = s_i + a_i$ and s_i values need to be copied from the previous array, and $s_i + a_i + s_i = 2s_i + a_i \leq 2(s_i + a_i) \in O(s_i + a_i)$. If the total cost of all append operations is in $O(n)$ (this is our goal), then the cost of the last resize operation must be in $O(n)$, and so $s_i + a_i \in O(n)$. $O(n) = O(s_i)$ because $n = s_i + 1$, and so $s_i + a_i \in O(s_i)$, and by subtracting across, we find that $a_i \in O(s_i)$.

Note that this does not yet prove that a_i *must* be proportional to s_i; instead, it shows that a_i *cannot* be larger than some constant multiple of s_i. Intuitively, this makes sense: if we resize with $a_i \gg s_i$, but no further append operations are performed, then there is at least $a_i - s_i$ work wasted; growing the vector too aggressively would be a problem if we suddenly stopped performing further append operations.

Let us first investigate what will happen by growing by $a_i = s_i$. This is a doubling scheme, where every time you use up the capacity of the current array allocation, then you allocate and copy to a new array with double the capacity.

The total cost for n append operations will be $\propto 1 + 2 + 4 + 8 + \cdots s_i$, where the final capacity $s_i \geq n$, because it must contain n items, and where $s_i \leq 2n$, because no further resize occurred. So the total cost will be

$$\begin{aligned}
&\leq \propto 1 + 2 + 4 + 8 + \cdots 2n \\
&= \sum_{i=0}^{\log(2n)} 2^i \\
&= 2^{\log(2n)+1} - 1 \\
&= 2(2n) - 1 \\
&\in O(n).
\end{aligned}$$

Thus using the doubling scheme $a_i = s_i$ would satisfy our need.

Note that this doubling scheme will waste $n - 2$ slots of memory in the worst case (if the n^{th} and final append operation must grow from $n - 1$ to $2(n - 1)$). We now want to ascertain whether it is possible for us to grow by a slower value (so that we do not waste as much RAM), or if exponential growth is the only scheme that would suffice (this second case is equivalent to saying $a_i \in \Omega(s_i)$).

We know that the total cost of allocation operations $\sum_{i=0}^{k+1} s_i$ must be $\geq n$ (necessary to hold n elements), and we want the total allocation cost to be $\sum_{i=0}^{k+1} s_i < cn$, because the cost of copy operations should be $\in O(n)$ so that the cost of all operations will be $\in O(n)$. As above, the final capacity $n \leq s_{k+1} \leq 2n$. From these we have

$$n \quad \leq \sum_{i=0}^{k+1} s_i \quad < cn \leq cs_{k+1}.$$

Denoting partial sums as $T_k = \sum_{i=0}^{k}$, we divide by the nonnegative size s_{k+1}:

$$\begin{aligned}
1 < \frac{\sum_{i=0}^{k+1} s_i}{s_{k+1}} = \frac{T_{k+1}}{T_{k+1} - T_k} &\quad < \quad c \\
1 > \frac{T_{k+1} - T_k}{T_{k+1}} &\quad > \quad \frac{1}{c} > 0 \\
1 > 1 - \frac{T_k}{T_{k+1}} &\quad > \quad \frac{1}{c} > 0 \\
0 > -\frac{T_k}{T_{k+1}} &\quad > \quad \frac{1}{c} - 1 > -1 \\
0 < \frac{T_k}{T_{k+1}} &\quad < \quad 1 - \frac{1}{c} < 1 \\
0 > \frac{T_{k+1}}{T_k} &\quad > \quad \frac{1}{1 - \frac{1}{c}} > 1
\end{aligned}$$

Thus, $\frac{T_{k+1}}{T_k} > 1$, *i.e.*, T_k grows at least exponentially. The finite difference (*i.e.*, $T_{k+1} - T_k$) of any exponential (or faster growing) sequence is itself at least exponential, and thus we see that s_k must at least grow exponentially. This implies $\frac{s_{k+1}}{s_k} \geq d > 1$, equivalent to $\frac{s_k + a_k}{s_k} = 1 + \frac{a_k}{s_k} \geq d > 1$, and thus $a_k \geq d - 1 s_k$ where $d - 1 > 0$; at every resize, we must at least grow by some factor times the current size (or perhaps grow faster).

Above, we have already proven that $a_k \in O(s_k)$ (if we grew too quickly, there was a risk that the final append operation results in an expensive resize that is never utilized). And now, we've just proven that $a_k \in \Omega(s_k)$ (*i.e.*, we must grow at least exponentially); therefore, $a_k \in \theta(s_k)$. Thus the amount by which we grow the array must be proportional to the size of the array itself.

6.4 std::list vs std::vector

Here we will compare the performance of `std::list` (Listing 6.1) to `std::vector` (Listing 6.2). We also compare these to benchmarks using a pre-allocated vector (Listing 6.3) and an `C`-style array (Listing 6.4).

Listing 6.1: Benchmark appending values to a linked list (via `std::list`). Values are appended using the $O(1)$ `push_back` function.

```
#include "../Clock.hpp"
#include <iostream>
#include <list>

int main() {
  const unsigned long N = 128000;
  const unsigned long REPS = 128;

  Clock c;
  for (unsigned long r=0; r<REPS; ++r) {
    std::list<long> long_list;
    for (unsigned long i=0; i<N; ++i)
      long_list.push_back(i);
  }
  c.ptock();

  return 0;
}
```

Listing 6.2: Benchmark appending values to a vector (via `std::vector`). Values are appended using the the amortized $\widetilde{O}(1)$ `push_back` function.

```
#include "../Clock.hpp"
#include <iostream>
#include <vector>

int main() {
  const unsigned long N = 128000;
  const unsigned long REPS = 128;

  Clock c;
  for (unsigned long r=0; r<REPS; ++r) {
    std::vector<long> long_vector;
    for (unsigned long i=0; i<N; ++i)
      long_vector.push_back(i);
  }
  c.ptock();

  return 0;
}
```

Listing 6.3: Benchmark filling a pre-allocated vector (via `std::vector`). The vector is allocated first and then filled using the `[]` operator (*not* using the `push_back` function)).

```
#include "../Clock.hpp"
#include <iostream>
#include <vector>

int main() {
  const unsigned long N = 128000;
  const unsigned long REPS = 128;

  Clock c;
  for (unsigned long r=0; r<REPS; ++r) {
    std::vector<long> long_vector(N);
    for (unsigned long i=0; i<N; ++i)
      long_vector[i] = i;
  }
  c.ptock();

  return 0;
}
```

Listing 6.4: Benchmark filling an array. The array is allocated with `malloc` and filled using the `[]` operator.

```
#include "../Clock.hpp"
#include <iostream>
#include <vector>

int main() {
  const unsigned long N = 128000;
  const unsigned long REPS = 128;

  Clock c;
  for (unsigned long r=0; r<REPS; ++r) {
    long*long_array = (long*)malloc(N*sizeof(long));
    for (unsigned long i=0; i<N; ++i)
```

	Average runtime (seconds)
`std::list`	0.6458
`std::vector`	0.08937
`std::vector` (pre-allocated)	0.01705
`unsigned long*`	0.01032

Table 6.1: Runtimes to fill a linked list (Listing 6.1), vector (Listing 6.2), pre-allocated vector (Listing 6.3), and array (Listing 6.4).

```
      long_array[i] = i;
    free(long_array);
  }
  c.ptock();

  return 0;
}
```

The average runtimes over 1024 timing replicates are given in Table 6.1. `std::list` is far slower than `std::vector`. The vector data structure offers a nice balance of speed and flexibility (*i.e.*, it's size does not need to be known during construction); however, a pre-allocated vector prevents resize operations from being necessary (it is essentially used as an array[8]). A simple array is fastest of all; it is surprisingly faster than the pre-allocated vector, due to the overhead of the `std::vector` data structure.

6.5 Creating our own vector implementation

Here we will use what we've learned about vectors to construct our own vector implementation. To start with, we will create a class that simply uses `new[]` (and `delete[]` in the destructor) and which resizes by adding one and doubling ($a_k = 1 + s_k$) when we call `push_back` (Listing 6.5). Pay no attention to the `__restrict` keyword; it will be explained in Chapter 11.[9]

[8]Perhaps you're wondering whether `push_back` is even useful? One of the all-time great responses I've heard: "So if someone shoves you, you gonna just stand there like a punk?" But in a serious answer to the question, append operations are very important in cases where you don't know the size in advance.

[9]"Pay no attention to the `__restrict` behind the `T*`!"

Listing 6.5: An in-house vector implementation and benchmark. The array is resized by doubling the current size and adding 1. The initial capacity for a vector is set at 4 elements (this prevents some early resizing). The meaning of __restrict will be revealed in Chapter 11.

```
#include "../Clock.hpp"
#include <iostream>

template <typename T>
class Vector {
private:
  unsigned long _size;
  unsigned long _capacity;
  T*__restrict _data;
public:
  Vector():
    _size(0),
    _capacity(4),
    _data( new T[_capacity] )
  { }

  Vector(unsigned long sz):
    _size(sz),
    _capacity(sz),
    _data( new T[_capacity] )
  { }

  ~Vector() {
    delete[] _data;
  }

  void push_back(const T & element) {
    if (_capacity == _size) {
      _capacity = (unsigned long)(_capacity*2)+1;
      T*__restrict new_data = new T[_capacity];

      for (unsigned long i=0; i<_size; ++i)
        new_data[i] = _data[i];

      delete[] _data;
      _data = new_data;
    }

    _data[_size] = element;
    ++_size;
  }

  const T & operator [](unsigned long i) const {
    return _data[i];
  }

  T & operator [](unsigned long i) {
    return _data[i];
  }
};

int main() {
  const unsigned long N = 128000;
  const unsigned long REPS = 128;

  Clock c;

  for (unsigned int r=0; r<REPS; ++r) {
    Vector<int> vec;
    for (unsigned int i=0; i<N; ++i)
      vec.push_back(i);
  }
  c.ptock();

  return 0;
}
```

On the same benchmark running `push_back`, this simple in-house vector implementation takes 0.05547s on average (in comparison, `std::vector` used $> 1.6\times$ the runtime). Likewise, using a pre-allocated version (similar to Listing 6.3), the runtime drops to 0.005567s (in comparison, the pre-allocated `std::vector` used $> 16\times$ the runtime). We can improve our design further so that it is both faster and uses less memory.

6.6 Memory allocation

The available memory to our program is essentially a large, untouched array of blocks. When we request a block of memory (*e.g.*, with `new[]` or `malloc`), the first available contiguous block that is large enough will be reserved. This block is reserved until it is freed (with `delete[]` or `free`). Let's consider the impact of our vector doubling scheme on these allocations.

A doubling scheme grows exponentially, so the sizes before reallocation will be $s_{i+1} = 2s_i$ (we add 1 more to this, but for this analysis the extra 1 will be unimportant). Let's use T_k again, the sum of all the allocations made thus far during the cumulative resize procedures: $T_k = \sum_{i=0}^{k} s_i = \sum_{i=0}^{k} 2^i$ (here we will assume we start with capacity 1 instead of 4 to again simplify our analysis). $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$, and so we can see that $s_{k+1} = 2^{k+1} > T_k$, meaning that allocating the new, double-length vector will require more memory than the cumulative amount allocated thus far. Because of this, Figure 6.1 visualizes the fact that the next allocation can never be stored in the memory previously used (even if we did not need the previous vector to copy from), and so the allocations will continue to creep forward in memory. This is bad for both memory usage and performance: constantly moving to novel memory addresses prevents the memory accesses from being cached effectively (cache performance will be discussed in further detail in Chapter 9).

6.7 Memory and an improved vector

We have proven that the vector needs to grow exponentially in order to achieve a total runtime of $O(n)$, but we have not considered different exponential growth constants. For example, instead of doubling the vector during ever each resize operation, we could triple the vector ($s_{k+1} = 3s_k$). Or we could grow by 1.1: $s_{k+1} = 1.1s_k$. All of these will satisfy our total runtime of $O(n)$, but there are pros and cons to each when it comes to practical performance: Let us denote the growth constant as $\lambda = \frac{s_{k+1}}{s_k}$. Higher constants will have the benefit of performing fewer overall resize operations (*i.e.*, they will grow more quickly); however, as we've already seen with $\lambda = 2$, this comes at the cost of memory usage and cache performance due to moving further into memory. On the other hand, lower growth constants will need to resize more times, but may possibly reuse some of the previous allocations.

To formalize this, we want $s_{k+1} \leq \sum_{i=0}^{k} s_i$. But even this is not strict enough: Remember that we need to temporarily have both the old allocation s_k and the new allocation s_{k+1} simultaneously in memory so that we can copy the data from the old array to the new array. So we require $s_{k+1} \leq \sum_{i=0}^{k-1} s_i$. Consider an exponential series with growth constant λ: $\sum_{i=0}^{k-1} s_i = \sum_{i=0}^{k-1} \lambda^i$. Without even using calculus, we can solve this series by exploiting a clever symmetry:

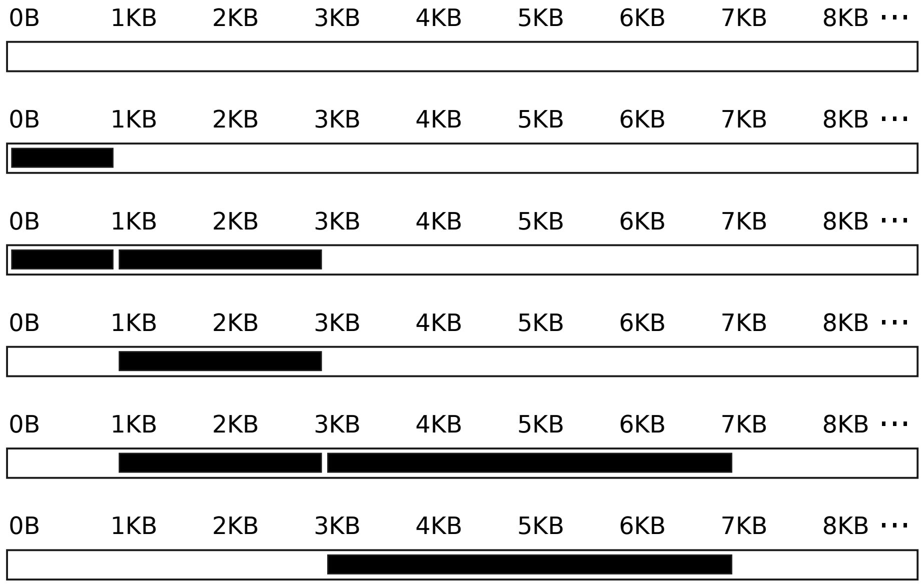

Figure 6.1: Allocation by doubling. An initially empty block of memory (first row) is replaced by an vector that initially allocates 1KB (second row). When that memory is exhausted, it allocates a new 2KB block and copies the existing 1KB of data into the 2KB block (third row). Then initial 1KB block can be freed (third row). This process continues as the vector is resized as the result of several append operations (*i.e.*, `push_back`). Because the series $\sum_{i=0}^{k} 2^i$ never exceeds the next term in the sequence, 2^{k+1}, the next allocation will always exceed the sum of all previous allocations; therefore, previous allocations can never be reused by this vector instance. As a result, the allocations continue to creep forward in memory.

$$\begin{aligned}
T_{k-1} = \sum_{i=0}^{k-1} \lambda^i &= 1 + \lambda + \lambda^2 + \lambda^3 + \cdots + \lambda^{k-1} \\
T_{k-1}\lambda &= \lambda + \lambda^2 + \lambda^3 + \lambda^4 + \cdots + \lambda^k \\
T_{k-1}\lambda - \lambda^k &= \lambda + \lambda^2 + \lambda^3 + \lambda^4 + \cdots + \lambda^{k-1} \\
T_{k-1}\lambda - \lambda^k + 1 &= 1 + \lambda + \lambda^2 + \lambda^3 + \lambda^4 + \cdots + \lambda^{k-1} \\
&= T_{k-1} \\
T_{k-1}(\lambda - 1) - \lambda^k + 1 &= 0 \\
T_{k-1}(\lambda - 1) &= \lambda^k - 1 \\
T_{k-1} &= \frac{\lambda^k - 1}{\lambda - 1}.
\end{aligned}$$

Thus we know we want $s_{k+1} = \lambda^{k+1} \le T_{k-1} = \frac{\lambda^k - 1}{\lambda - 1}$.[10]

$$\begin{aligned}
\lambda^{k+1}(\lambda - 1) &\le \lambda^k - 1 \\
\lambda^k(\lambda^2 - \lambda) &\le \lambda^k - 1 \\
\lambda^k(\lambda^2 - \lambda) - \lambda^k + 1 &\le 0 \\
\lambda^k(\lambda^2 - \lambda - 1) + 1 &\le 0.
\end{aligned}$$

This does not have an easy solution; however, when $k \gg 1$, then λ^k will become very large, and so a negative value for $\lambda^2 - \lambda - 1$ will ensure $\lambda^k(\lambda^2 - \lambda - 1) + 1 \le 0$. Solving $\lambda^2 - \lambda - 1 = 0$ for λ gives us the boundary condition of the region we want for λ. The solution for this can be found with the quadratic formula. We can solve this numerically using `numpy`: `import numpy as np; x = np.matrix([[1.0, 1.0],[1.0, 0]]); print np.linalg.eigvals(x);`.[11] Eigendecomposition yields two

[10]Because we are only interested in $\lambda > 1$ (*i.e.*, the vector grows rather than shrinks), so we can multiply by $\lambda - 1$ and preserve the direction of the inequality).

[11]Here we're using "eigendecomposition"[12] on the characteristic matrix for the recurrence relation, which gives the exponential bases for the closed form solutions of the recurrence.

[12]From the German word "Eigen" meaning self, these are the vectors where sending them as the input to a square matrix will produce an output that stretches or compresses, but does not rotate. It's like finding the direction along which an arrow shot into a windstorm would accelerate, decelerate, reverse, or stop, but would not turn. Anyway,

roots: we are only interested in the nonnegative of these, which is actually the golden ratio[13]! This is the largest resize constant that will asymptotically permit reuse of previously allocated memory. A vector class and benchmark implemented with $\lambda = 1.6$ is shown in Listing 6.6.

Listing 6.6: Revised in-house vector using $\lambda = 1.6$..

```
#include "../Clock.hpp"
#include <iostream>
#include <assert.h>

template <typename T>
class Vector {
private:
  unsigned long _size;
  unsigned long _capacity;
  T*__restrict _data;
public:
  Vector():
    _size(0),
    _capacity(4),
    _data( new T[_capacity] )
  { }

  Vector(unsigned long sz):
    _size(sz),
    _capacity(sz),
    _data( new T[_capacity] )
  { }

  ~Vector() {
    delete[] _data;
  }

  void push_back(const T & element) {
    if (_capacity == _size) {
      _capacity = (unsigned long) (_capacity * 1.6 + 1);
      T*__restrict new_data = new T[_capacity];
```

eigenvectors are also known (at least among other vectors) for being quite selfish. If an eigenvector had 3 chocolate bars and you had -1 chocolate bars (because you owed some people some chocolate, you know how it is), they wouldn't even throw a couple of bars your way so that you'd each have some. But yeah, eigenvectors are just, you know, one of those "the rich get richer" sort. . . So much for equality (er, well, unless the eigenvalue is 1).

[13]This is $\lambda < \phi \approx 1.618$

```
      for (unsigned long i=0; i<_size; ++i)
        new_data[i] = _data[i];

      delete[] _data;
      _data = new_data;
    }

    _data[_size] = element;
    ++_size;
  }

  const T & operator [](unsigned long i) const {
    return _data[i];
  }

  T & operator [](unsigned long i) {
    return _data[i];
  }
};

int main() {
  const unsigned long N = 128000;
  const unsigned long REPS = 128;

  Clock c;

  for (unsigned int r=0; r<REPS; ++r) {
    Vector<int> vec;
    for (unsigned int i=0; i<N; ++i)
      vec.push_back(i);

    /* Test: */
    //    for (unsigned long i=0; i<N; ++i)
    //      assert(vec[i] == i);
  }
  c.ptock();

  return 0;
}
```

However, there is still more to consider. Note that the above proof considers the case where k is large. On the other extreme, when k is small, we are not guaranteed that $\lambda < 1.618$ will reuse previous allocations. On the other extreme, we can try to force reuse of previously allocated memory when k is small. We cannot ever reuse the first or second allocations: Trying to reuse the first block alone yields $\lambda^2 \leq 1$, which contradicts our above assertion that we need $\lambda > 1$; however, let's consider $k = 3$: $\lambda^3 \leq 1 + \lambda$, where the fourth allocation can be built from the first two allocations. The solution to this $k = 3$ equation yields $\lambda <\approx 1.32$. This is benchmarked in Listing 6.7.

Listing 6.7: Revised in-house vector using $\lambda = 1.3$..

```
#include "../Clock.hpp"
#include <iostream>
#include <assert.h>

template <typename T>
class Vector {
private:
  unsigned long _size;
  unsigned long _capacity;
  T*__restrict _data;
public:
  Vector():
    _size(0),
    _capacity(4),
    _data( new T[_capacity] )
  { }

  Vector(unsigned long sz):
    _size(sz),
    _capacity(sz),
    _data( new T[_capacity] )
  { }

  ~Vector() {
    delete[] _data;
  }

  void push_back(const T & element) {
    if (_capacity == _size) {
      _capacity = (unsigned long)(_capacity*1.3 + 1);
      T*__restrict new_data = new T[_capacity];

      for (unsigned long i=0; i<_size; ++i)
        new_data[i] = _data[i];

      delete[] _data;
      _data = new_data;
    }

    _data[_size] = element;
    ++_size;
  }

  const T & operator [](unsigned long i) const {
    return _data[i];
  }

  T & operator [](unsigned long i) {
    return _data[i];
  }
};

int main() {
  const unsigned long N = 128000;
  const unsigned long REPS = 128;

  Clock c;

  for (unsigned int r=0; r<REPS; ++r) {
    Vector<int> vec;
    for (unsigned int i=0; i<N; ++i)
      vec.push_back(i);

    /* Test: */
    //    for (unsigned long i=0; i<N; ++i)
    //      assert(vec[i] == i);
  }
  c.ptock();

  return 0;
}
```

$\lambda = 1.3$ ensures that memory can be reused as early as possible (as shown above, reusing during the third allocation is not possible), but it will grow vectors less aggressively and therefore need to copy more data. Clearly these opposing forces have a balance: for this reason we will also investigate $\lambda = 1.5$. Why 1.5? Because it is between 1.32 and 1.618, but also because we can use bit twiddling to avoid floating point math and thus avoid an integer to float (or integer to double) conversion: Recall that `x/2` is equivalent to `x>>1`. Thus, we can do `x=(unsigned long)(x*1.5);` more efficiently by running `x = x + x>>1;`, or more efficiently, `x += x>>1;`. This is performed in Listing 6.8 (with the small modification that we will always add one element to guarantee the vector grows if the initial capacity were ≤ 1).

Listing 6.8: Revised in-house vector using $\lambda = 1.5$..

```
#include "../Clock.hpp"
#include <iostream>
#include <assert.h>

template <typename T>
class Vector {
private:
  unsigned long _size;
  unsigned long _capacity;
  T*__restrict _data;
public:
  Vector():
    _size(0),
    _capacity(4),
    _data( new T[_capacity] )
  { }

  Vector(unsigned long sz):
    _size(sz),
    _capacity(sz),
    _data( new T[_capacity] )
  { }

  ~Vector() {
    delete[] _data;
  }

  void push_back(const T & element) {
    if (_capacity == _size) {
      _capacity += (_capacity>>1) + 1;
      T*__restrict new_data = new T[_capacity];

      for (unsigned long i=0; i<_size; ++i)
        new_data[i] = _data[i];

      delete[] _data;
      _data = new_data;
    }
    _data[_size] = element;
    ++_size;
  }

  const T & operator [](unsigned long i) const {
    return _data[i];
  }

  T & operator [](unsigned long i) {
    return _data[i];
  }
};

int main() {
  const unsigned long N = 128000;
  const unsigned long REPS = 128;

  Clock c;

  for (unsigned int r=0; r<REPS; ++r) {
    Vector<int> vec;
    for (unsigned int i=0; i<N; ++i)
      vec.push_back(i);

    /* Test: */
    //   for (unsigned long i=0; i<N; ++i)
    //     assert(vec[i] == i);
  }
  c.ptock();

  return 0;
}
```

Table 6.2 lists the runtimes for these in-house vector implementations, alongside the `std::list` and `std::vector` runtimes. Using $\lambda = 1.5$ offers the best performance, with a speedup of $> 2\times$ compared to $\lambda = 2$ and a speedup of $> 3\times$ compared to `std::vector`.

6.8 Using `realloc`

The vector approaches thus far have all relied on allocating a new, larger block with `new[]`, copying data to the new block from the existing array,

	Average runtime (seconds)
`std::list`	0.6458
`std::vector`	0.08937
$\lambda = 2$	0.05547
$\lambda = 1.6$	0.06651
$\lambda = 1.3$	0.05626
$\lambda = 1.5$	0.02685

Table 6.2: Runtimes to fill a `std::list` (Listing 6.1), `std::vector` (Listing 6.2), vector with $\lambda = 2$ (Listing 6.5), vector with $\lambda = 1.6$ (Listing 6.6), vector with $\lambda = 1.3$ (Listing 6.7), and vector with $\lambda = 1.5$ (Listing 6.8).

and then deleting the old array with `delete[]`. There is an alternative approach that directly asks the operating system to simply resize the block of memory we've already allocated. This method is called `realloc`, and it is compatible with memory allocated with `malloc`[14] and freed with `free`; it cannot be used with `new[]` and `delete[]`.

`malloc` is a function from `C`, which behaves similarly to `C++`'s `new[]` but with two very important differences: First, `malloc` is allocated by requesting a number of bytes (*not* the number of elements you want). For this reason, allocating an array of `N` `double` types would be done with `new[]` by calling `new double[N]` and would be done with `malloc` by calling `(double*) malloc(N*sizeof(double))`. Calling malloc without multiplying the number of elements by `sizeof` is a common error. The second difference between `malloc` and `new[]` is that `malloc` simply allocates a block of memory; if we've built an array of of primitives (*e.g.*, each element in our vector is an `int`, `float`, `char`, ...), this doesn't matter. But if we've allocated an array of objects (*e.g.*, each element in our vector is a `std::list<int>`), `malloc` does not call constructors on the elements (while `new[]` would call the constructor). Likewise, when `free` is called on a block of memory, no destructors will be called (the destructors would automatically be called for each element if we were using `delete[]` to free memory allocated by `new[]`).

[14] And `calloc`. `calloc` is similar to `malloc`, except it initializes all memory cells to 0.[15]

[15] Calling `malloc` and then `memset` to zero out the allocated memory is not strictly the same as calling `calloc`. This is because `calloc` arrays are allocated in a lazy manner and are not actually allocated until the array has been used. In contrast `malloc` (and `new[]`) will allocate immediately. This can make `calloc` efficient, but it also makes it slightly dangerous for dead code elimination when benchmarking.

A more subtle but nonetheless important difference is that `malloc` will return `NULL` when the allocation fails (*e.g.*, if you ask for more memory than your system can serve), whereas `new[]` will throw an exception. For this reason, it is good practice to check the return value from `malloc` to ensure it is not `NULL`, especially if your code is memory intensive and will run on foreign systems rather than your own (or where the size of the data on which your software will be applied are unknown to you).

Listing 6.9: Revised in-house vector using `realloc` with $\lambda = 1.5$..

```
#include "../Clock.hpp"
#include <iostream>
#include <cstring>

template <typename T>
class Vector {
private:
  unsigned long _size;
  unsigned long _capacity;
  T*__restrict _data;
public:
  Vector():
    _size(0),
    _capacity(4),
    _data( (T*) malloc(_capacity*sizeof(T)) )
  {
    // fixme: check return value of malloc
  }

  Vector(unsigned long sz):
    _size(sz),
    _capacity(sz),
    _data( (T*) malloc(_capacity*sizeof(T)) )
  {
    // fixme: check return value of malloc
  }

  ~Vector() {
    free( _data );
  }

  void push_back(const T & element) {
    if (_capacity == _size) {
      // Need to resize:
      _capacity += (_capacity>>1) + 1;
      _data = (T*)realloc(_data, _capacity*sizeof(T));
      // fixme: check return value of realloc
    }

    _data[_size] = element;
    ++_size;
  }

  const T & operator [](unsigned long i) const {
    return _data[i];
  }

  T & operator [](unsigned long i) {
    return _data[i];
  }
};

int main() {
  const unsigned long N = 128000;
  const unsigned long REPS = 128;

  Clock c;

  for (unsigned int r=0; r<REPS; ++r) {
    Vector<int> vec;
    for (unsigned int i=0; i<N; ++i)
      vec.push_back(i);
  }
  c.ptock();

  return 0;
}
```

A more obscure function is `realloc`, which simply requests that an allocated block be reallocated to now own more or less memory. This can be very efficient if the memory after the current allocation is already free. For example, the allocations performed in Figure 6.1 could simply be extended in $O(1)$ (not including the time to verify that the following memory is available) by simply rewriting the ledger so that the memory now owns the subsequent block as well. Although `realloc` is not well suited to apply to vectors of objects, it is excellent for implementing vectors of primitive types. Listing 6.9 implements a vector that is resized using `realloc`. Its runtime is 0.01511s, a $> 5\times$ speedup over `std::vector` and a $> 1.7\times$ speedup over our in-house implementation using `new[]` and $\lambda = 1.5$. For such a simple, solid, and ubiquitous piece of software like `std::vector`, these speedups are quite impressive.

Questions

1. **[Level 1]** Vectors provably perform $\widetilde{O}(1)$ amortized copy operations per appended value. Come up with an instance where a linked list (which does not perform any additional copying as it grows) would still be advantageous to a vector. How can a vector still be used to get the best of both worlds?

2. **[Level 2]** At a scientific conference, you hear a presentation about a vector resizing method that grows with $a_i = \frac{s_i}{\log(1+s_i)}$. This method does not grow the vector quite as aggressively, and thus helps to reuse memory from previous allocations more effectively. What is your opinion of using this method when appending very large numbers of values to a vector? Come up with mathematical and empirical evidence (*i.e.*, a plot of runtimes) to justify your answer.

3. **[Level 3]** Using a vector implemented with `new[]`, use different values of $\lambda \in [1.3, 1.61]$ and plot two series: First, the total number of copy operations performed to append 128000 values (as a function of λ). Second, the largest difference in any two memory addresses used during the entire process of pushing 128000 values (as a function of λ).

Chapter 7

Maps, Hash Tables, and Sorted Vectors

7.1 Maps

Maps are data structures for associating key-value pairs. They're somtimes called "dictionaries" because of their functional resemblance to the books of alphabetically sorted words with entries holding the definition (and etymology and pronunciation) alongside the word. "Map" is the mathematical term for a function that translates a "key" from an input set (in the case of an English dictionary, a word) into the corresponding "value" from a output set (in the case of an English dictionary, the definition). These data structures are quite important for easy manipulation of data, and are thus used quite frequently.

Trivial implementations would simply store a list or vector of n key-value pairs in arbitrary order and then search them in n key comparisons[1]; however, using a sorted vector like an English dictionary will allow lookup in $O(\log(n))$ steps rather than n steps[2].

[1]To be perfectly precise about the runtime, we need to consider that each of these calls to the key `== operator` may cost more than $O(1)$. For example, checking whether two strings of length k are equal costs $O(k)$ in the worst case, because we must go through every character when they are equal in order to verify; when they are *not* equal, then calling `==` can be much faster (*i.e.*, we can `return false` on the first non-equal characters), and so the average-case runtime will depend on the length of the strings as well as the degree of similarity between them.

[2]A sorted linked list is not of much use, because we cannot jump halfway through the

There is a limitation to storing maps via arrays or via vectors: as we saw in Chapter 6, appending data to the end of a vector can be made efficient (amortized $\widetilde{O}(1)$ using exponential growth), but inserting data into the middle of any kind of array (vectors included) will always require shifting all following elements to the next index, and thus cost $O(n)$ in the worst case. For this reason, neither vectors nor linked lists will perform well in practice when we need to dynamically add key-value pairs to the map.

7.2 Balanced binary trees

Balanced binary trees offer a means by which we can guarantee that each insertion will cost $\in O(\log(n))$ in the worst case and inductively preserve our sorted order while inserting. Thus, we are able to use the sorted order to lookup key-value pairs in $O(\log(n))$ steps. Binary trees are like linked lists in that they function using pointers[3] These pointers can make a fairly high overhead when the key and value types are both quite small.

Compared to storing data in a vector and performing each lookup by scanning over all n items, balanced binary trees are quite fast.[4] Balanced binary trees are the data structures used by the `std::map` class.

Listing 7.1 generates several random strings, inserts them as keys into maps of `std::string` to `unsigned long`, and then measures the time to traverse both maps and modify the `unsigned long` value associated with each key. Running the benchmark with the balanced binary tree used by `std::map` takes 2.493s.

Listing 7.1: Benchmark measuring the time to modify values associated with keys in two `std::map` types. Values are modified using the $\Theta(\log(n))$ `[]` operator.

```
#include "../Clock.hpp"
#include <iostream>
#include <string>
#include <vector>
#include <map>
```

remaining list in $O(1)$ steps as we can with an array or a vector.

[3]Each node has two node pointers (to the left and right children) or three node pointers (to the left and right children and the parent).

[4]But as we will see, this is like saying, "That's the fanciest restaurant between the check cashing place and the cash4gold place": overall, it isn't often the best you can get.

```
std::string random_string() {
  std::string res = "";

  const unsigned int len=32;
  for (unsigned char i=0; i<len; ++i)
    // Append a random capital letter:
    res += 'A' + (rand() % 26);

  return res;
}

int main() {
  const unsigned long N = 100000;
  const unsigned long REPS = 16;

  std::vector<std::string> all_strs;
  std::map<std::string, unsigned long> str_to_int_a;
  std::map<std::string, unsigned long> str_to_int_b;

  for (unsigned long i=0; i<N; ++i) {
    std::string name = random_string();
    all_strs.push_back(name);
    str_to_int_a[name] = i;
    str_to_int_b[name] = N-i;
  }

  Clock c;

  for (unsigned long r=0; r<REPS; ++r)
    for (const std::string & name : all_strs) {
      ++str_to_int_b[name];
      str_to_int_a[name] -= str_to_int_b[name];
    }
  c.ptock();

  return 0;
}
```

7.3 Hash tables

Balanced binary trees like those used by `std::map` hold the data in a sorted order; however, this is only one means by which we can implement a map.

Another approach is to convert each key into an integer using a deterministic "hash" function h, and then use that integer as an index in an array (a "hash table"). This approach does not necessarily keep the keys in sorted order, and so, unlike balanced binary trees, there is no guarantee in general that key insertion and lookup will cost $\in O(\log(n))$. Hash tables are inefficient (*e.g.*, $O(n^2)$) when many distinct objects $x, y, \ldots$ produce similar hashes $h(x) = h(y)$; thus the performance of hash tables are highly dependent upon the hash function and the specific keys on which it will be applied.

From this, the question naturally arises: "Why bother using hash tables when balanced binary trees have guaranteed performance on arbitrary data?"[5] For one thing, hash tables can be implemented in a very lightweight manner (*e.g.*, not using as many pointers per key-value pair inserted). But additionally, hash functions can also exploit the parallel nature of hardware, in which several bits in the hash can be computed simultaneously[6]. The specifics of engineering a good hash function will be described in greater detail in Chapter 8.

Listing 7.2 repeats the same task as in Listing 7.1, but using `std::unordered_map`[7] instead of `std::map` (and using the default hash function for `std::string`). This small change significantly improves performance: the benchmark now runs in 0.6739s. This is a $> 3\times$ speedup over the balanced binary tree-based implementation via `std::map`.

Listing 7.2: Benchmark measuring the time to modify values associated with keys in two `std::map` types. Values are modified using the `[]` operator, which has a runtime dependent on the hash function and data used.

```
#include "../Clock.hpp"
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>

std::string random_string() {
  std::string res = "";
```

[5]This is often voiced by the people who refused to lend you their wood and straw SR-71 in a previous chapter. In case hashes are ever discussed, such people often carry a business card, which reads, "Give up your inquiries, which are completely useless."

[6]Just as `x+y`, `arr[i]`, *etc.*, can be computed via hardware in $O(1)$ for fixed-length integers.

[7]So called because it does not store data in a sorted order as do the balanced binary trees implementing `std::map`.

```
  const unsigned int len=32;
  for (unsigned char i=0; i<len; ++i)
    // Append a random capital letter:
    res += 'A' + (rand() % 26);

  return res;
}

int main() {
  const unsigned long N = 100000;
  const unsigned long REPS = 16;

  std::vector<std::string> all_strs;
  std::unordered_map<std::string, unsigned long> str_to_int_a;
  std::unordered_map<std::string, unsigned long> str_to_int_b;

  for (unsigned long i=0; i<N; ++i) {
    std::string name = random_string();
    all_strs.push_back(name);
    str_to_int_a[name] = i;
    str_to_int_b[name] = N-i;
  }

  Clock c;

  for (unsigned long r=0; r<REPS; ++r)
    for (const std::string & name : all_strs) {
      ++str_to_int_b[name];
      str_to_int_a[name] -= str_to_int_b[name];
    }
  c.ptock();

  return 0;
}
```

7.4 Associated vectors

Thus far we have focused on maps that can grow dynamically at the same time we query them by their keys. In that case, balanced binary trees and hash tables are often the best practical performance we can achieve; however, in many cases, we will insert all key-value pairs into the map and *then* do work that needs to look up values from keys. An example of this more constrained use-case would be an English dictionary: words are inserted offline, and then the dictionary is finalized. Only after finalizing the set of keys (*i.e.*, the set of words contained) do we need to put the dictionary into a bound book in an order that facilitates quick retrieval. This may sound like a small benefit, but it is important because it permits us to insert the key-value pairs in a lazy manner, by simply appending them to a vector. Once all insertions are finished, this vector of key-value pairs can be sorted by key in $O(n \log(n))$ or better[8].

Assuming the sort costs $O(n \log(n))$, this doesn't achieve any theoretical speedup over a balanced binary tree: the balanced binary tree requires $O(\log(n))$ steps to lookup each of the n values inserted, also resulting in a runtime in $O(n \log(n))$. But in practice, sorting is a very lightweight operation compared to constructing a balanced binary tree. Furthermore, looking up a key in a sorted array can be performed in $O(\log(n))$, but with a faster runtime constant than that available to a pointer-based implementation of a balanced binary tree.[9]

But there is an even more important potential speedup: when processing two maps with identical collections of keys, the sorted arrays of key-value pairs from each map will not require lookup at all: if key `x` occupies index `i` in the sorted key-value array in map `m_a`, then the key `x` must also occupy index `i` in the sorted key-value array in map `m_b` (when both maps use identical collections of keys). This means that processing can proceed without any lookup steps at all, and by using integer array indices rather than lookup by key.

When two maps do not use identical key sets, a similar speedup can be employed, which will traverse both arrays in $O(n)$ and skip the keys found in one map but not found in the other. This process is highly reminiscent of the "merge" step in merge sort.

Thus, when key-value pairs are rarely inserted, but are frequently accessed, using sorted vectors or sorted arrays can pose a significant speedup over both balanced binary trees and hash tables.

Listing 7.3 uses a remarkable 0.01229s, a $> 54\times$ speedup over the hash

[8]Keys for which we only have a `<` operator cannot be sorted faster than $O(n \log(n))$, but other types of keys, *e.g.*, strings and fixed-precision integers, can often be sorted more quickly.

[9]This is partly because of cache performance, which is quite good when processing contiguous data structures like arrays.

table-based implementation and $> 202\times$ speedup over the balanced binary tree-based implementation.

Listing 7.3: Benchmark measuring the time to modify values associated with keys via sorting. Vectors of key-value pairs are constructed and sorted. Because the key collections are known to be identical in advance, the vectors are now aligned by keys, and so the same keys can be retrieved by simply visiting the same integer indices.

```
#include "../Clock.hpp"
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

std::string random_string() {
  std::string res = "";

  const unsigned int len=32;
  for (unsigned char i=0; i<len; ++i)
    // Append a random capital letter:
    res += 'A' + (rand() % 26);

  return res;
}

int main() {
  const unsigned long N = 100000;
  const unsigned long REPS = 16;

  std::vector<std::pair<std::string, unsigned long> > str_and_int_a;
  std::vector<std::pair<std::string, unsigned long> > str_and_int_b;
  for (unsigned long i=0; i<N; ++i) {
    std::string name = random_string();
    str_and_int_a.push_back( std::make_pair(name, i) );
    str_and_int_b.push_back( std::make_pair(name, N-i) );
  }
  std::sort(str_and_int_a.begin(), str_and_int_a.end());
  std::sort(str_and_int_b.begin(), str_and_int_b.end());

  Clock c;

  // Now the keys are already aligned; now we can simply index them
  // with integers:
  for (unsigned long r=0; r<REPS; ++r)
```

```
    for (unsigned long i=0; i<N; ++i) {
      ++str_and_int_b[i].second;
      str_and_int_a[i].second -= str_and_int_b[i].second;
    }
  c.ptock();

  // Can also easily look up val from name in log(n) time; in contrast
  // to a tree (map) or hash table (unordered_map), a sorted vector is
  // more lightweight.

  return 0;
}
```

7.5 Member variables

One common use-case for maps is to associate objects with values after the fact. For example, when constructing a graph, the designer of the graph package may not have included the ability to store data for the color of each `Node` type; however, using the `Node` (or, `Node*`, a pointers to a node) as keys, one could easily construct a map that allows us to retrieve or modify the color of a particular node. Importantly, it is possible to do this without modifying any of the source code from the graph package[10].

This flexibility comes at a price. A faster approach would be to simply encode an integer `int color` attribute via a member variable in the `Node` class. Pairing these at compile time essentially uses the address of each `Node` in memory to construct a perfect hash table (*i.e.*, a hash table guaranteed not to have any collisions). Using a member variable in this way can achieve significantly higher performance, but does not easily allow us to use the mapping from `Node` to `color` attribute at runtime. For example, a balanced binary tree or hash table-based map would more easily address the use-case in which we want to allow the user to input the key at runtime and lookup up or modify the color associated with it.

Listing 7.4 performs a second benchmark, which is easily suited for a member variable rather than a map. In the first implementation of this benchmark, strings are used to map to integers. It runs in 0.9016s. Listing 7.5

[10]This becomes more important when writing object-oriented code, where we create one working widget and hopefully it works so well that we need never revisit that task.

repeats this with a member variable rather than a map, and runs in 0.004165s (a $> 216\times$ speedup over the `std::map`-based version).

Listing 7.4: A benchmark performing association via `std::map`.

```
#include "../Clock.hpp"
#include <iostream>
#include <string>
#include <vector>
#include <map>

std::string random_string() {
  std::string res = "";

  const unsigned int len=32;
  for (unsigned char i=0; i<len; ++i)
    // Apped a random capital letter:
    res += 'A' + (rand() % 26);

  return res;
}

int main() {
  const unsigned long N = 100000;
  const unsigned long REPS = 16;

  std::vector<std::string> all_strs;
  std::map<std::string, int> str_to_int;

  for (unsigned long i=0; i<N; ++i) {
    std::string name = random_string();
    all_strs.push_back(name);
    str_to_int[name] = i;
  }

  Clock c;

  for (unsigned long r=0; r<REPS; ++r)
    for (const std::string & name : all_strs)
      ++str_to_int[name];

  c.ptock();

  return 0;
}
```

Listing 7.5: A reimplementation of Listing 7.5 via a member variable rather than association via `std::map`.

```
#include "../Clock.hpp"
#include <iostream>
#include <string>
#include <vector>
#include <map>

struct StrAndInt {
  std::string name;
  long val;
};

std::string random_string() {
  std::string res = "";

  const unsigned int len=32;
  for (unsigned char i=0; i<len; ++i)
    // Apped a random capital letter:
    res += 'A' + (rand() % 26);

  return res;
}

int main() {
  const unsigned long N = 100000;
  const unsigned long REPS = 16;

  std::vector<StrAndInt> all_objs;

  for (unsigned long i=0; i<N; ++i) {
    std::string name = random_string();
    all_objs.push_back({name,long(i)});
  }

  Clock c;

  for (unsigned long r=0; r<REPS; ++r)
    for (StrAndInt & obj : all_objs)
      ++obj.val;

  c.ptock();

  return 0;
}
```

7.6 A reminder against premature optimization

In many cases, maps (whether through balanced binary trees or hash tables) are great tools for quickly hacking out an idea, and so they are inespensible in scripting langauges like `Python`. Do not let a focus on performance be a distraction where performance is not a limiting factor.[11]

Questions

1. [**Level 1**] Write a simple program to insert many 32-character strings of capital letters (as keys) with random integer values. Iterate through all keys with first character 'A' and sum up the corresponding values. How simply can you implement this with `std::map`? And how long does this take to run?

2. [**Level 2**] Of the methods covered in this chapter, which is the highest performance method applicable to the method from the question above? How much faster is the superior approach? Assume that all values must be inserted into your map-like data structure (even those that do not start with 'A'), but where only those that begin with 'A' will have their value summed.

3. [**Level 3**] Assume your CPU natively supports b-bit fixed-precision integers, and that these are used for hashing. What is the largest array length n on which we can use one of these b-bit integers as an index? If we have a perfect hash ("perfect hashing" denotes hashing that provably produces 0 collisions) and the hash function itself uses a constant number of native integer operations, what is the cost of

[11]Sometimes, when reading in a strange file format, `C++`'s `std::map` and `std::unordered_map` or `Python`'s `dict` are very helpful, especially for making a first working version of the code, and starting with a map-free implementation can sometimes drive you crazy.[12]

[12]"I saw the most mediocre minds of my generation destroyed by maplessness..."

inserting into the hash table? What is the cost of looking up an item in the hash table? What would be the cost for these operations when using a balanced binary tree containing n objects? What is the speedup of perfect hashing relative to the balanced binary tree (in terms of n)? Why would this be?

Chapter 8

Creating Fast Hashes

Chapter 7 discusses maps and highlights the frequent practical advantage of hash tables over balanced binary trees; however, this advantage came with a caveat: hash tables will only be efficient when we can create hash functions that produce few collissions between distinct objects[1]. This chapter explores the practical task of constructing a hash on a `std::set` of objects.

8.1 Initial implementation via `std::set`

Our baseline approach is to avoid hashes altogether: in this case, we store our set of sets by simply using `std::set<std::set<T> >` (Listing 8.1). This implementation will use the `<` operator when inserting elements into each set and when inserting the sets into the set of sets. When each set is filled with several random values $\in \{0, 1, 2, \ldots 255\}$ and when we build a set of 2^{18} such sets, the runtime is 11.18s.

Listing 8.1: An ordered set of ordered sets, implemented via `std::set<std::set<T> >`.

```
#include "../Clock.hpp"
#include <set>

int main() {
  const unsigned long NUMBER_SETS=1<<18;
  const unsigned long SET_INSERTIONS=1<<8;
```

[1]Note that we often say "distinct" objects, because hashes on the same object necessarily give the same result (because the hash function is deterministic).

```
  const unsigned long MAX_ELEMENT_PLUS_ONE=1<<8;

  srand(0);

  Clock c;

  std::set<std::set<unsigned int> > collection_of_sets;
  for (unsigned int i=0; i<NUMBER_SETS; ++i) {
    std::set<unsigned int> st;
    for (unsigned int j=0; j<SET_INSERTIONS; ++j)
      st.insert( rand() % MAX_ELEMENT_PLUS_ONE );
    collection_of_sets.insert(st);
  }

  c.ptock();

  return 0;
}
```

8.2 Properties for a hash on `std::set<T>`

Before we write the hash function h for a `std::set<T>`, we need to consider a few properties that we want. One such property is commutativity: we need $h(\{x, y, \ldots\} = h(\{y, x, \ldots\})$. Also, we need our hash function to be deterministic, so that we ensure feeding in identical sets will produce an identical hash. Also, we would like the runtime of calling h to be at most linear in the number of elements in the set. Lastly, our hash function should be flexible so that it adapts easily to different types `T` contained within the set. This last requirement naturally leads us to consider using the standard hash (if one is available) on each element of type `T`.

8.3 Hashing via XOR

One simple approach (which is commutative, efficient, and can easily use `std::hash<T>` on each element in the set) is to simply chain together the XOR of the hashes of all elements in the set (Listing 8.2). When run on the same benchmark (generating 2^{18} sets, each of which contains several values $\in \{0, 1, 2, \ldots 255\}$), the runtime is 25.68s, which is significantly slower than simply using `std::set<std::set<T> >`.

Listing 8.2: A hash set of ordered sets, implemented via `std::unordered_set<std::set<T> >` and an XOR hash.

```cpp
#include "../Clock.hpp"
#include <functional>
#include <set>
#include <unordered_set>

template <typename T>
struct SetHash {
  std::size_t operator() (const std::set<T> & s) const {
    std::hash<T> single_hash;
    std::size_t combined_hash_value = 0;
    for (const T & obj : s) {
      unsigned long single = single_hash(obj);
      combined_hash_value ^= single;
    }
    return combined_hash_value;
  }
};

int main() {
  const unsigned long NUMBER_SETS=1<<18;
  const unsigned long SET_INSERTIONS=1<<8;
  const unsigned long MAX_ELEMENT_PLUS_ONE=1<<8;

  srand(0);
  Clock c;

  std::unordered_set<std::set<unsigned int>, SetHash<unsigned int> >
      collection_of_sets;
  for (unsigned int i=0; i<NUMBER_SETS; ++i) {
    std::set<unsigned int> st;
    for (unsigned int j=0; j<SET_INSERTIONS; ++j)
      st.insert( rand() % MAX_ELEMENT_PLUS_ONE );
    collection_of_sets.insert(st);
  }

  c.ptock();

  return 0;
}
```

8.4 Hashing via sum

Let's dissect why the XOR implementation performs poorly: Consider that `std::hash<int>(i)` returns `i`, and so the element hashes will only generate 8 bits of information. By only XORing them together, we never leave the 8 least-significant bits; therefore, the XOR set hash can only distinguish 256 possible sets, and so multiple non-equivalent sets will produce the same hash. This is a collision; when looking up an element with a non-unique hash, the table will be forced to lookup *all* items with that hash, and so the runtime of looking up all items becomes slow: the total of pairs of items sharing the same hash is $|\ \{x : h(x) = h(y) \wedge x \neq y\}\ |$, which grows quadratically with the number of items sharing the same hash: $\binom{|\ \{h(x)=i\}\ |}{2}$.[2] Thus the XOR set hash performs slowly because many distinct sets share the same hash, and thus there is a non-trivial quadratic term in our runtime.

Because we are using 64-bit `unsigned long` types for our hash, collisions may be avoided by using the higher-significance bits (rather than limiting ourselves to the least-significant 8 bits). But the question remains how best to do this. One option that bears a resemblance to the XOR approach is to simply sum the hashes of the elements in the set (Listing 8.3). By summing the hashes, we maintain commutativity (*i.e.*, $h(\{x,y\}) = h(\{y,x\})$) but where XOR will never use the higher-order bits, the sum has a possibility of using them[3], even though the hashes of each element is still in $\{0,1,2,\ldots 255\}$. By using this strategy, the performance improves, reaching 11.66s (a $> 2\times$ speedup over the XOR approach, although still only roughly equal to the `std::set<std::set<T> >` approach).

Listing 8.3: A hash set of ordered sets, implemented via `std::unordered_set<std::set<T> >` and a sum hash.

```
#include "../Clock.hpp"
#include <functional>
#include <set>
#include <unordered_set>

template <typename T>
struct SumSetHash {
  std::size_t operator() (const std::set<T> & s) const {
    std::hash<T> single_hash;
```

[2] `>mfw 2many collisions :_(` $\times\infty$

[3] Because of carry operations

```
    std::size_t combined_hash_value = 0;
    for (const T & obj : s) {
      unsigned long single = single_hash(obj);
      combined_hash_value += single;
    }
    return combined_hash_value;
  }
};

int main() {
  const unsigned long NUMBER_SETS=1<<18;
  const unsigned long SET_INSERTIONS=1<<8;
  const unsigned long MAX_ELEMENT_PLUS_ONE=1<<8;

  srand(0);

  Clock c;

  std::unordered_set<std::set<unsigned int>, SumSetHash<unsigned int> >
      collection_of_sets;
  for (unsigned int i=0; i<NUMBER_SETS; ++i) {
    std::set<unsigned int> st;
    for (unsigned int j=0; j<SET_INSERTIONS; ++j)
      st.insert( rand() % MAX_ELEMENT_PLUS_ONE );
    collection_of_sets.insert(st);
  }

  c.ptock();

  return 0;
}
```

8.5 Scaling by a prime

Summing the hash values will use the higher-significance bits more effectively, but for such small element values, it will not reach the most significant bits in a 64-bit `unsigned long`. For this reason, it may be helpful to scale the hashes of the elements in the set when we sum them. However, some scaling factors will not "scatter" the information into several bits as we hope: for example, scaling by 2^k will simply shift the hash result left by k bits (thus moving the bits we have, but keeping the number of bits in use constant[4]). The same phenomenon will occur when we scale by a constant that *contains* 2 in its prime factors, even if it is not a power of 2.

An intuitive choice is to simply avoid scalars that contain any non-trivial factors[5], would be to scale the values by a prime number, preferably a fairly large value[6], which will be capable of moving the information from the hashes of the elements into the higher-significance bits. A downside of multiplying the hashes by a constant is that, in general, elements with small hashes (like those considered) will scale to large values and thus may lose variability in the lower-significance bits. For this reason, we can XOR in the element hash again. The result is a version of the sum set hash that has been modified (Listing 8.4).

Listing 8.4: A hash set of ordered sets, implemented via `std::unordered_set<std::set<T> >` and a scaled sum hash.

```
#include "../Clock.hpp"
#include <functional>
#include <set>
#include <unordered_set>

template <typename T>
struct SumPrimeSetHash {
  std::size_t operator() (const std::set<T> & s) const {
    std::hash<T> single_hash;
    std::size_t combined_hash_value = 0;
    for (const T & obj : s) {
      unsigned long single = single_hash(obj);
      combined_hash_value += 2147483647ul*single ^ single;
    }
    return combined_hash_value;
  }
};

int main() {
  const unsigned long NUMBER_SETS=1<<18;
  const unsigned long SET_INSERTIONS=1<<8;
  const unsigned long MAX_ELEMENT_PLUS_ONE=1<<8;

  srand(0);
```

[4]Or, if we shift too far, actually *decreasing* the number of bits in use.
[5]*I.e.*, we exclude the number itself and 1, both of which must divide evenly.
[6]Thus not a power of 2, since 2 is the only even prime.

```
  Clock c;

  std::unordered_set<std::set<unsigned int>, SumPrimeSetHash<unsigned
      int> > collection_of_sets;
  for (unsigned int i=0; i<NUMBER_SETS; ++i) {
    std::set<unsigned int> st;
    for (unsigned int j=0; j<SET_INSERTIONS; ++j)
      st.insert( rand() % MAX_ELEMENT_PLUS_ONE );
    collection_of_sets.insert(st);
  }

  c.ptock();

  return 0;
}
```

8.6 “Universal hashing”

Thus far we’ve approached the choice of hash fairly heuristically[7]; let’s now attack this problem more theoretically. Consider a hash that scales the first element in the set by one constant a_1, scales the second element in the set by another constant a_2, *etc.* The raw result will be of the form $h(x) = \sum_i a_i \cdot x_i$ (where x is a set).[8]

A collision occurs when two distinct sets $x \neq y$ produce the same hash:

$$h(x) = \sum_i a_i \cdot x_i = h(y) = \sum_i a_i \cdot y_i.$$

We have not yet developed a strategy for choosing $a_1, a_2, \ldots$, but hopefully the goal of avoiding collisions will help us. We could try to approach the

[7][Reaches for bottle of hand sanitizer]

[8]Note that these operations are not necessarily commutative if two sets with identical contents iterate through those contents in a different order[9]; however, `std::set` works by keeping the values in a sorted order, just as `std::map` does. Thus, this approach is safe for hasing `std::set` objects, but may need to be refined for hashing `std::unordered_set` objects.

[9]A sorted order is not necessary, but a consistent order on sets with the same contents is.

problem by choosing values $a_1, a_2, \ldots$ for which collisions are rare[10]. This means that we would like there to be few instances where

$$\begin{aligned}\sum_i a_i \cdot x_i &= \sum_i a_i \cdot y_i\\ \sum_i a_i \cdot (x_i - y_i) &= 0\\ a_1 \cdot (x_1 - y_1) + \sum_{i>1} a_i \cdot (x_i - y_i) &= 0\\ a_1 \cdot (x_1 - y_1) &= \sum_{i>1} a_i \cdot (y_i - x_i).\end{aligned}$$

We are interested in the case where $x \neq y$, and so we know that there must be at least some elements differing between x and y; thus, without loss of generality[11], assume that $x_1 \neq y_1$ and so $x_1 - y_1 \neq 0$. We would like to prove that, given $a_2, a_3, \ldots$, there are few values of a_1 for which this equation would hold.

The means to do this is not so clear. But, from number theory (see Appendix A[12]), we do know that nonzero integers have unique inverses modulo any prime. Thus, given $x_1 - y_1 \neq 0$, there is a unique number $a_1 \in \{1, 2, \ldots p-1\}$ satisfying

$$a_1 \cdot (x_1 - y_1) \equiv \sum_{i>1} a_i \cdot (y_i - x_i) \text{ (mod p)}^{13}$$

for some prime p.

This means that if we choose random values for $a_1, a_2, a_3, \ldots$, each $\in \{1, 2, \ldots p-1\}$, there is only one possible a_1 that would lead to a collision between unequal objects, or equivalently $\Pr(h(x) = h(y)) = \frac{1}{p-1}$ (because there are $p-1$ possible choices for a_1, but only one will produce $h(x) = h(y)$). Thus, if we choose a large prime number for p, we should be able to dramatically decrease the number of collisions. This strategy is called

[10]Importantly, we don't know the x and y in advance, so we would need to show that collisions are rare in general, not only rare for particular sets x and y.

[11]"w.l.o.g."

[12]Power level hidden in appendix

[13]Using $u \equiv v$ (mod$p$) is the same as saying $u \bmod p = v \bmod p$ or `u % p == v % p` in `C++`. The $\equiv$ symbol simply applies the modulo to both sides.

“universal” because it can be used to guarantee that the probability of a collision is $\leq\approx \frac{1}{m}$ where m is the size of the hash table.

Listing 8.5 uses $p = 18446744073709551557$, which was found by simply starting at $2^{64} - 1$ and counting down to the first prime[14] The listing also introduces a small random number generator because the benchmark uses `rand()` to insert random elements into the sets, and calling `srand` to seed our random a_i (and thus make the hash deterministic) at the start of each set hash call will alter the values inserted into the sets.

Listing 8.5: A hash set of ordered sets, implemented via `std::unordered_set<std::set<T> >` and a prime-based universal hash function.

```
#include "../Clock.hpp"
#include <functional>
#include <set>
#include <unordered_set>

// Some arbitrary random number generator from StackOverflow:
class SimpleRandomGenerator {
private:
  unsigned long _random_value;

  static unsigned long ROL(unsigned long v, unsigned char shift) {
    return ((((v) >> ((sizeof(v) * 8) - (shift))) | ((v) << (shift))));
  }
public:
  SimpleRandomGenerator(unsigned long seed):
    _random_value(seed)
  { }

  unsigned long next() {
    _random_value = _random_value * 214013L + 2531011L;
    _random_value = ROL(_random_value, 16);
    return _random_value;
  }
};

template <typename T>
class UniversalSetHash {
public:
  static const unsigned long biggest_64_bit_prime =
```

[14] *I.e.*, it is the largest prime number that fits in a 64-bit `unsigned long`.

```
      18446744073709551557ul;

  std::size_t operator() (const std::set<T> & s) const {
    SimpleRandomGenerator srg(0);

    std::hash<T> single_hash;
    std::size_t combined_hash_value = 0;
    for (const T & obj : s) {
      unsigned long single = ( single_hash(obj) * srg.next() ) %
          biggest_64_bit_prime;
      combined_hash_value += single;
    }
    combined_hash_value += ( s.size() * srg.next() ) %
        biggest_64_bit_prime;
    return combined_hash_value;
  }
};

int main() {
  const unsigned long NUMBER_SETS=1<<18;
  const unsigned long SET_INSERTIONS=1<<8;
  const unsigned long MAX_ELEMENT_PLUS_ONE=1<<8;

  srand(0);

  Clock c;

  std::unordered_set<std::set<unsigned int>, UniversalSetHash<unsigned
      int> > collection_of_sets;
  for (unsigned int i=0; i<NUMBER_SETS; ++i) {
    std::set<unsigned int> st;
    for (unsigned int j=0; j<SET_INSERTIONS; ++j)
      st.insert( rand() % MAX_ELEMENT_PLUS_ONE );
    collection_of_sets.insert(st);
  }

  c.ptock();

  return 0;
}
```

On the benchmark, the runtime is 9.894s, over 1 second faster than the `std::set<std::set<T> >` approach.

8.7 Investigating the number of collisions

Let's now consider the number of insertions into the `std::unordered_set<std::set<T> >` where the hashes of two sets produce the same value.[15].

Listing 8.6 counts the number of insertions using a hash value that has already occurred and the number of "collisions"[18] The results are shown in Table 8.1. Our universal hash achieved no collisions[19].

Listing 8.6: Counting the collisions, implemented with various hash algorithms on a set.

```
#include <iostream>
#include <functional>
#include <set>
#include <map>
#include <random>

constexpr unsigned long BIG_PRIME_NEAR_POWER_OF_TWO = 2147483647ul;
// A prime with fairly good scattering of 0 and 1 bits:
constexpr unsigned long BIG_PRIME_WITH_GOOD_BINARY_SCATTERING =
    3644798167ul;

template <typename T>
struct XORSetHash {
  std::size_t operator() (const std::set<T> & s) const {
    std::hash<T> single_hash;
    std::size_t combined_hash_value = 0;

    for (const T & obj : s) {
      unsigned long single = single_hash(obj);
      combined_hash_value ^= single;
    }
    return combined_hash_value;
```

[15]These are meant to measure the number of collisions, but to be technically precise, we don't make sure the sets aren't identical to one another here, so to be truly pedantic, we haven't proven all of these to be collisions. Any deterministic hash function[16] will produce $h(x) = h(y)$ when $x = y$[17].

[16]As long as it is also "pure", *i.e.*, it doesn't modify global variables

[17]"Cutting-edge wildlife research at the U of MT confirms that bears really do use the toilet in the woods."

[18]Again, to be pedantic: we haven't proven them precisly to be collisions, but it's close enough for this discussion.

[19]"New York Matinee called it 'a playful but mysterious little dish'."

```
  }
};

template <typename T>
struct SumSetHash {
  std::size_t operator() (const std::set<T> & s) const {
    std::hash<T> single_hash;
    std::size_t combined_hash_value = 0;

    for (const T & obj : s) {
      unsigned long single = single_hash(obj);
      combined_hash_value += single;
    }
    return combined_hash_value;
  }
};

template <typename T>
struct SumPrimeSetHash {
  std::size_t operator() (const std::set<T> & s) const {
    std::hash<T> single_hash;
    std::size_t combined_hash_value = 0;
    for (const T & obj : s) {
      unsigned long single = single_hash(obj);
      combined_hash_value += BIG_PRIME_WITH_GOOD_BINARY_SCATTERING*single
          ^ single;
    }
    return combined_hash_value;
  }
};

template <typename T, unsigned long PRIME>
struct XORPrimeSetHash {
  std::size_t operator() (const std::set<T> & s) const {
    std::hash<T> single_hash;
    std::size_t combined_hash_value = 0;
    for (const T & obj : s) {
      unsigned long single = single_hash(obj);
      // (s1*p) + (s2*p) + ... == (s1+s2+...)*p
      // is bijective to s1+s2+..., and so will not resist collision
          better than s1+s2+...
      // however,
      // (s1*p) ^ (s2*p) ^ ... != (s1^s2^...)*p
      // this can be more collision resistant than s1^s2^....
      // If PRIME is near a power of 2, then
      // PRIME=11111111...1x where x is short and has both 0 and 1 bits.
      // In that case, single*PRIME = sum_i 2^i*single + x*single.
      // Aside from x*single, it behaves much like a reduce built of ^
          operations, because
      // (exclusing x), everything is simply bit-shifted at all shifts,
      // those shifts are summed. This often leads to products of the
          form a11111...1111b.
      // Each of those results is XORed, so it will behave much like a
          reduce built of
      // ^ operations.

      // So, ideally choose a PRIME not near a power of 2.
      combined_hash_value ^= single*PRIME;
    }
    return combined_hash_value;
  }
};

// Some arbitrary random number generator from StackOverflow:
class SimpleRandomGenerator {
private:
  unsigned long _random_value;

  static unsigned long ROL(unsigned long v, unsigned char shift) {
    return ((((v) >> ((sizeof(v) * 8) - (shift))) | ((v) << (shift))));
  }
public:
  SimpleRandomGenerator(unsigned long seed):
    _random_value(seed)
  { }

  unsigned long next() {
    _random_value = _random_value * 214013L + 2531011L;
    _random_value = ROL(_random_value, 16);
    return _random_value;
  }
};

template <typename T>
class UniversalSetHash {
private:
  // Some arbitrary random number generator from StackOverflow:
public:
  std::size_t operator() (const std::set<T> & s) const {
    const unsigned long biggest_64_bit_prime = 18446744073709551557ul;

    SimpleRandomGenerator srg(0);

    std::hash<T> single_hash;
    std::size_t combined_hash_value = 0;
    for (const T & obj : s) {
      unsigned long single = ( single_hash(obj) * srg.next() ) %
          biggest_64_bit_prime;
      combined_hash_value += single;
    }
    combined_hash_value += ( s.size() * srg.next() ) %
        biggest_64_bit_prime;
    return combined_hash_value;
  }
};

template <typename SETHASH>
void print_collisions() {
  const unsigned long NUMBER_SETS=1<<18;
  const unsigned long SET_INSERTIONS=1<<8;
  const unsigned long MAX_ELEMENT_PLUS_ONE=1<<8;

  srand(0);

  SETHASH sh;
  std::map<unsigned long, unsigned int> hash_to_object_count;
  for (unsigned int i=0; i<NUMBER_SETS; ++i) {
    std::set<unsigned int> st;
    for (unsigned int j=0; j<SET_INSERTIONS; ++j)
      st.insert( rand() % MAX_ELEMENT_PLUS_ONE );

    ++hash_to_object_count[ sh(st) ];
  }

  unsigned int items_in_non_unique_buckets = 0;
  unsigned long collisions = 0;
  for (const auto & pair : hash_to_object_count) {
    unsigned int items_in_bin = pair.second;

    items_in_non_unique_buckets += (items_in_bin-1);
    collisions += items_in_bin*(items_in_bin-1)/2; // items in bin choose
        2
  }
  std::cout << typeid(SETHASH).name() << " items added to non-unique
```

```
      buckets:" << items_in_non_unique_buckets << " collisions:" <<
      collisions << std::endl;
}

int main() {
  print_collisions<XORSetHash<unsigned int> >();
  print_collisions<SumSetHash<unsigned int> >();
  print_collisions<SumPrimeSetHash<unsigned int> >();
  print_collisions<UniversalSetHash<unsigned int> >();
  print_collisions<XORPrimeSetHash<unsigned int,
      BIG_PRIME_NEAR_POWER_OF_TWO> >();
  print_collisions<XORPrimeSetHash<unsigned int,
      BIG_PRIME_WITH_GOOD_BINARY_SCATTERING> >();

  return 0;
}
```

Hash	Occupied insertions	Collisions
XOR	261888	134226345
Sum	256562	11252516
Prime-scaled sum	60047	75174
Universal	0	0

Table 8.1: Collisions from various hash methods, found by running Listing 8.6. Occupied insertions refer to the sum of all insertions into a bucket with an occupied hash, while collisions refer to $\sum_i \binom{k_i}{2}$, where k_i is the number of objects with hash value i.

8.8 Hash caching

As you can see, evaluating our universal hash is more computationally expensive than simpler approaches (*e.g.*, XORing the element hashes together). Thus in a case where the XOR approach does *not* produce many collisions, it is preferred; however, the cost of re-computing the hash value can be improved by caching it on `std::set` construction and then simply returning the hash value when our hash function is called. Obviously, we can't easily mess with the `std::set` code directly, and so a viable approach to this would be to declare our own `HashSet` class that contains a `std::set` object (which will be immutable after construction[20]). A `HashSet<T>` object could be constructed from a `std::set<T>` object and would cache its hash value on construction. In practice, caching these kinds of pure functions on immutable objects[21] can substantially improve efficiency.

Questions

1. [**Level 2**] Write a simple alternative to the `std::set<T>` class, which will be immutable and which will cache the hash value on construction. How do the runtimes in your answer above change by switching to your alternative class?

2. [**Level 3**] "Perfect hashing" denotes hashing that provably produces 0 collisions. Assume you have a perfect hash function h defined to take `std::set` arguments and returns an `unsigned long`. What is the greatest number of objects that you can hash without collisions if the output of your hash is an `unsigned long`? You may assume that `sizeof(unsigned long)` returns 8.

3. [**Level 3**] Following from the question above, consider hashing several sets: You hash all possible sets with nonnegative integers $< t$ (*i.e.*, the "power-set" of $\{0, 1, 2, \ldots t-1\}$) and have no collisions. How many sets are possible if their contents are all nonnegative integers $< t$? Combining this with your answer above, what is the maximum possible value of t for which we can guarantee perfect hashing using h?

[20]Immutability is essential when caching the hash value; regardless, immutability important when using that object in a set (or unordered set) or as the key in a map (or unordered map). Consider: do you really want to go reorganize that data structure every time the key changes?!

[21]*i.e.*, "memoization", and no, I amb nod tybing "memorization" wid food in my moud

Chapter 9

Cache and Transposition

9.1 SRAM and DRAM

Computer store information in digital circuitry using voltage: high voltage can indicate a 1 state and low voltage can indicate a 0 state[1]. In the circuit construction, there are multiple options to keep a bit at a stable value.

SRAM ("static RAM") is built using inverters. Basically, a transistor is an analog faucet where you can apply a signal to tell how much to open the pipe, and electricity will either flow through or will be stopped up and accumulate. A transistor can be used to make a circuit where a low-voltage signal to the control results in a low-voltage output and where a high-voltage control signal results in a high-voltage output (by using the output to measure what flows through). But alternatively, we can use a transistor to build an inverter: a high-voltage input will allow electricity to flow through the transistor, but it will drain electricity from the source (*i.e.*, the wire where the electricity is coming from). An inverter turns a 0 (a low-voltage signal) into a 1 (a high-voltage signal) or turns a 1 into a 0. If we chain two inverters together in a loop, then consider what happens: A low-voltage input is turned into a high-voltage output by the first inverter, and that is sent as input into the second inverter and turned into a low-voltage output, which is then sent back into the first inverter. In this way, it is self-reinforcing: a low-voltage input to the first inverter will loop through the circuit and become lower and lower. Likewise, a high-voltage input to the first inverter will loop through

[1]There are also reverse schemes; this is related to whether a design is "active high" or "active low".

the circuit and become higher and higher. Thus an analog circuit becomes digital, and thus we have a the start of a 1-bit memory cell.

A downside of that inverter design is that electricity is always flowing through the wires, which is very inefficient. We can improve this using a CMOS[2] design. We can control each inverter using two different transistors: One transistor may allow electricity to flow into the output, and the other transistor may allow electricity to drain from the output. In this manner, we can turn the first control on and the second control off, which will allow electricity to flow into the output and turn off draining from the output, resulting in a very high-voltage output. Likewise, if we turn the first control off and the second control on, then no electricity is allowed to flow into the output and also electrical charge will be drained from the output (making a very low-voltage output). Thus each inverter costs 2 transistors, and thus 1-bit of SRAM memory costs 2 inverters or 4 transistors. We will also need 1 or 2 transistors to access 1 bit of SRAM: these transistors decide whether we are connected to the output or not, so that we can connect to only one particular bit of SRAM.

SRAM is called static because its state doesn't change unless we manually force it to do so (*e.g.*, when we set a single bit of RAM); however, this required 4 transistors. On the other hand, we could implement 1 bit of RAM using a capacitor[3]: the capacitor stores the voltage, and then later we can read its voltage back; however, capacitors have some limitations: First, they will slowly leak their voltage, and so they will need to be periodically refreshed. Second, when we read from the capacitor, we will actually modify its state by lowering its charge. Thus, when we read from the capacitor, we will need to refresh it as well. We will also need to use a transistor to guard the capacitor so that it is not constantly leaking its charge into the connected wires. The more analog, constantly changing nature of the charge in the capacitor (as opposed to looped inverters) makes it more "dynamic", and thus a 1-bit RAM cell of this type is called DRAM ("dynamic RAM").

SRAM is used sparingly because it requires more transistors to implement. But SRAM is also fast: the feedback of the inverters in SRAM means that they quickly reach either a 0 or a 1, and so we can reliably modify or read

[2]Complementary metal-oxide semiconductor

[3]The thing that makes that really satisfying high-pitched sound when your camera flash is charging (or when Iron Man's repulsors are charging)

SRAM very quickly. DRAM, on the other hand, is cheaper[4] and denser[5], but it is slower to read from[7] and requires periodic refreshing. Because of these pros and cons, both types of RAM are useful, but in a complementary way:

SRAM is used on the processor for things like registers. It is fast, but expensive, and so we have do not have much. Also, if you are tempted to build the entire computer using SRAM, consider physical space: even if we had the money, it would make the circuit so much larger that it would take more time for the signals to propagate, and thus limit our clock speed[8]. Meanwhile, DRAM is used for the bulk of main memory (we generally just call it "RAM")[9]. Because writing to or reading from a single cell of DRAM is not as fast because we are charging or discharging a capacitor[10] and because it is not physically located on the chip[11].

Clearly a program that only uses 1 `int` variable (which fits in a single register on the chip) can be very fast, because it can be optimized by the compiler to use only registers using the on-chip SRAM. Likewise, a program that uses lots of memory will inevitably need to use RAM (*i.e.*, DRAM) and will be slower by comparison. The question is this: is there any sort of middle ground?[12]

There is indeed a middle ground: cache.

[4]Fewer transistors = less $ilicon

[5]Less silicon = more memory in a small space[6]

[6]Not using a larger space means we can still use a high clock speed and it will have time to propagate through the circuit fully.

[7]Capacitors charge and discharge in a decaying exponential, and so writing and reading from DRAM takes time.

[8]*i.e.*, "What is the deal with the black box– if the black box is the only part that survives a crash, then why don't they just make the whole plane out of the black box?"

[9]There is an old story about a German engineer after World War II bemoaning Germany's loss despite their engineering prowess of that era. "One of our tanks was worth four of theirs!" When asked what happened, the man sighed and replied, "They always had five..." Some battles are won with mass production; DRAM is proof of that.

[10]Citation: Iron Man 2 when I– I mean *Iron Man* was fighting that guy with electric whips at the Grand Prix. I– that is, Iron Man kept getting hit while waiting for the capacitors to charge.

[11]Even at the speed of light, larger distances make things slower. If we run out of RAM, we can use disk, which is further and slower still.

[12]Not that I'm Iron Man– I'm asking for a friend.[13]

[13]But not to say that I'm *not* Iron Man either. Or Banksy. You know what, Mr. Nakamoto was my father, please call me Satoshi...

9.2 Memory hierarchy & cache

Unlike registers, we do not interface with the cache manually; rather than explicitly saying "I want to load the contents of address `0xF97A3450` into register `R1`"[14] as we would in assembly code, we interface with the cache by simply reading from or writing to RAM. Fetching the value from an address from RAM will fetch not only that value, but values in the neighboring block of addresses. Thus, dereferencing address `0xF97A3450` may also fetch addresses `0xF97A3449`, `0xF97A3451`, *etc.* Where do we put these other fetched values that we did not explicitly request? In an array known as the "cache".

Cache can be made of SRAM, but placed physically further away form the heart of the CPU than registers are. Thus, reading and writing from registers is still significantly faster. But if our assembly code had asked for the contents of address `0xF97A3450` to be loaded into register `R1` (which went to RAM for the request), and then the next line of assembly code asks for the contents of address `0xF97A3451` to be loaded into register `R2`, then rather than go all the way to RAM, that second load may be found in the cache. Reading from and writing to the cache is significantly faster than reading from and writing to RAM.

Because the cache is automatically updated as we read from and write to RAM, the pattern with which we access data is very important. Reading from address `0x00000000` then address `0x00000001` then address `0x00000002`, *etc.* will need to go to RAM for the first fetch (this is called a "cache miss", because it will check the cache before going to RAM and find that the address is, unfortunately, not currently in the cache), followed by several operations where the addresses *are* in the cache, and thus the computer will not go out to RAM (these are known as "cache hits"). While cache misses need to go out to RAM, cache hits do not, and so they can be significantly faster. Note that the choice of what register to use is determined by the programmer while coding in assembly code and by the compiler during compilation in `C/C++` code, but that all cache is managed at runtime on the circuit and has nothing to do with the compiler.

Thus far, we have discussed "the" cache as if it is a unique entity; however, most modern high-performance CPUs have multiple hierarchical levels of cache: Registers are the fastest and most scarcely available, and they must be invoked explicitly. Caches are automatically used when we request data

[14] *BEEP BOOP. DID I HEAR A PLEASE?*

from RAM. Of these, the L1 cache is the fastest and smallest, L2 is larger but slower, L3 is larger still and slower still, and so on. These caches are searched hierarchically as you would expect: If request data from RAM, the computer will check the L1 cache first. If it is found, it will return it to us, but if it is not found, it will try the L2 cache, and so on until it either finds the address in a cache or goes out to RAM. Each access of a particular address also caches that address. For example, if an address was not found in the L1 cache, and so the computer checks the L2 cache and finds it, then the address and its data will automatically be imported into the L1 cache. Likewise, if the computer does need to go all the way out to RAM to access an address, then in doing so it will populate nearby addresses into the L3 cache, a smaller block of those addresses into the L2 cache, and a still smaller block of those addresses into the L1 cache. Each of these blocks is known as a "cache line", and each cache will have cache lines of different size, with smaller caches like the L1 cache using smaller cache lines. This is illustrated in Figure 9.1.[15]

The differences in costs for memory accesses from different places in the hierarchy are quite stark (Table 9.1). It is also worth noting that many modern computers have two distinct L1 caches: L1d is used for data, while L1i is used for the instructions in our program (in languages like `C++` we don't often think of it this way, but source code is really just a type of data, an array of bytes like any other array). Using a dedicated cache for code ensures that it is less likely to compete for cache space with data that we load. Thus we see that *ceteris paribus*[16], smaller code is more efficient. This is one reason why optimizations like loop unrolling (which will be discussed in Chapter 10) are not always productive when taken to their logical conclusion[17].

9.3 Circuit implementations of caches

Caches work by partitioning the address into several bits. An address will be partitioned into tag bits, cache set bits, and offset bits. In the case of

[15]Note that in this chapter we focus on the inclusive cache model: when an address is cached in L1, then it should also be cached in L2 and L3, *etc.* This uses memory less efficiently by storing multiple copies, but simplifies the circuitry.

[16]Latin: "all other things being equal"

[17]*i.e.*, when applied *ad extremum*[18]

[18]*i.e.*, *ad nauseam*, *i.e.*, *ad*– you get the point.

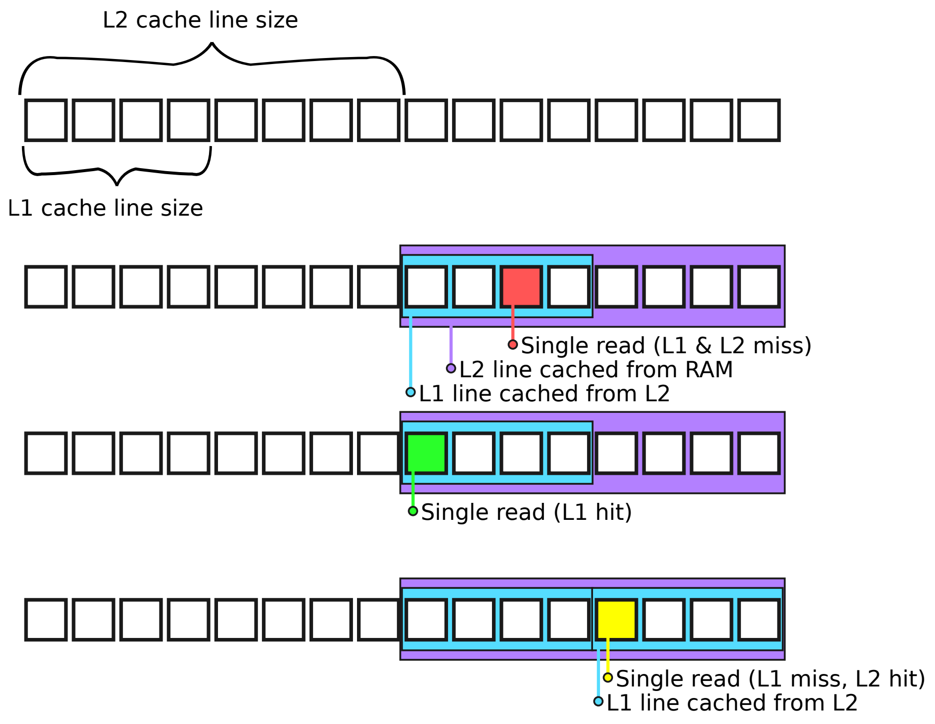

Figure 9.1: Illustration of cache access. Data in RAM are drawn, with white blocks indicating uncached memory, light blue background indicating a cache line in L1, and a purple background indicating a cache line that is cached in L2. The top row shows no data cached. In the second row, a single read is performed. This address (colored red) indicates an L1 cache miss followed by an L2 cache miss, meaning the data must be fetched from RAM, which is slow. Following the cache misses, the purple cache line being cached into L2 and the smaller light blue cache line being cached into L1. In the third row, another read is performed. Reading this address (colored green) results in an L1 cache hit, resulting in a fast load time. In the fourth row, another read is performed. This read (colored yellow) results in an L1 cache miss, but an L2 cache hit, and will result in a moderate load time.

Name	Cache hit latency	Cache size
L1	4 cycles	32KB
L2	10 cycles	256KB
L3	40 cycles	8MB
RAM	120 cycles $\approx$ 30ns	16GB

Table 9.1: Approximate latencies on Xeon 5500 series CPU from `https://stackoverflow.com/questions/10274355/cycles-cost-for-l1-cache-hit-vs-register-on-x86#10274402`.

a cache hit, offset bits are used to decide which bytes in the cache line are those we wanted to load. In the third line of Figure 9.1, a read triggers an L1 cache hit. Because we only want to read one illustrated square of memory and each cache line of the L1 cache contains 4 such squares, we know that we need 2 bits to index them. In general the cache line size is 2^{offset} where $offset$ is the number of bits of the address used as offset bits (they will be the least-significant bits). The set bits (the next most significant) are used as the bits to grab a particular cache line from the cache.

For example, if such a cache holds 256KB and has cache lines of size 256B, then it holds 1024 of these lines. 256B cache lines will need 8 offset bits to access and storing 1024 cache lines means that we would use 10 set bits. From this it is natural to question why we need the tag bits. The tag bits comprise the remainder of the address, and are used to verify that the address in the cache is indeed the correct address (and not simply an address with an identical suffix). The cache stores the current tags in an array whose size is the number of cache lines. Thus, when we query the cache for an address (the address will be partitioned into the tag bits, the cache set bits, and the offset bits), the cache set bits will be used as the index to retrieve a particular cache line. The data for that cache line will be retrieved in one array in the circuitry, while the tag will be retrieved from another array of the circuitry. After the tag is retrieved from the cache, the desired tag from our address is compared to it, and if they match we have a cache hit (if they don't match we have a cache miss). We do not necessarily need to send in the offset bits; instead, we retrieve the full cache line (which is itself an array of data), and then look in the correct index in that cache line using the offset bits.

The above scheme describes a simple and effective type of cache: the direct-mapped cache. But there is a problem with this design, which is clear when we access evenly spaced addresses that are too far apart to fit in the same cache line. These addresses can result in many different addresses that produce identical set bits, which mean they will fight over the same cache line. This can induce behavior where each cache miss is not followed by many subsequent cache hits; instead, we may have several cache misses in sequence, which is very bad for performance.

Another cache scheme is the fully associative cache. In this case, we don't use any cache set bits: instead, we divide the address into the tag (using the most-significant bits) and the offset (using the least-significant bits). In this case, we can simply compare all currently cached tags in parallel, and if one matches, we return the data in the same index (*e.g.*, if the third tag in the cache matches, we return the data from the third cache line). This design has the advantage that it is essentially impossible to trick by spacing our addresses in a particular manner. But even though we can perform all of these tag comparisons in parallel, it requires many comparators, which means we need a lot of circuitry. Thus, fully associative caches may not be efficient designs for building large caches.

A happy medium is the set-associative cache. In a set associative cache, for each string of set bits, we retrieve not one but but a few tags. These tags correspond to a few cache lines. Thus for a particular string of set bits, we don't retrieve a particular cache line, but instead retrieve a small collection of cache lines and their corresponding tags. These tags are then compared in parallel as one would do with a fully-associative cache. Since there are only a few tags to compare, the circuitry is practical to build (whereas it may not be for a fully associative cache).

Set-associative caches are tolerant when we load multiple addresses that have the same set bits. Where direct-mapped caches would need to replace the data any time two addresses contain the same string of cache set bits, an 8-way set-associative cache would permit simultaneously storing 8 distinct cache lines, each sharing identical strings of cache set bits and distinct tag bits (meaning they come from distinct addresses).

Caches also use a "dirty" bit (also denoted using its opposite, which is called a "valid" bit), which is used to determine whether or not the cache line has actually been populated with current data. Importantly, when our program writes to RAM and an L1 cache hit occurs, then in the inclusive cache model discussed here, the data must also be in the L2 cache[19]. In this

[19]Even though we do not necessarily need to check because it would be slow

case, the chip designers have two options: they can either write through to the L2, L3, ... caches so that all caches are up to date or they can wait until the data is evicted from the L1 cache and only write the updated data to the L2 cache when that happens. The first case is called a "write-through" cache[20], while the second option is a "write-back" cache. There is a trade-off where write-through caches require less sophisticated circuitry, but where write-back caches are more performant when writing to memory. Very fine tuning your program will require investigating the details of the types of cache in your CPU.

9.4 Optimized access patterns

As is clear in Figure 9.1, we can achieve high performance by accessing memory in a contiguous, sequential fashion. This means that the cost of each cache miss (for any caches, *e.g.*, L1, L2, ...) is amortized out because it must be followed by several cache hits, and so the average time will be quite fast. For this reason, arrays, especially those accessed in a sequential manner, are generally the best for cache performance. This means that array-based algorithms will generally have a far superior runtime constant compared to non-contiguous data structures.[21]

If a program first iterates through all even indices in an array and then subsequently iterates through all odd indices[23], then when the array is very long, the code will be less efficient than iterating over all indices sequentially (if possible). A large performance difference only occurs when the array is long because we know that by the end of iterating through all even indices, the cache should be completely full[24]. Thus, the iteration through the odd indices will likely not benefit at all from any caching that occurred when iterating through the odd indices. Only the end of the array will be cache when we begin iterating through the odd indices, because the beginning of the array will be evicted from the cache if the entire array cannot be stored.

[20]Surprising no one

[21]Do you remember how in Chapter 6, we said that vectors were advantageous over linked lists because they store memory in contiguous blocks?[22]

[22]Pepperidge Farm remembers.

[23]We will see this precise behavior in the case of the FFT implementation in Chapter 14.

[24]No cache schematic can save us if we've already packed 1024KB of data into a cache that holds 1024KB of data.

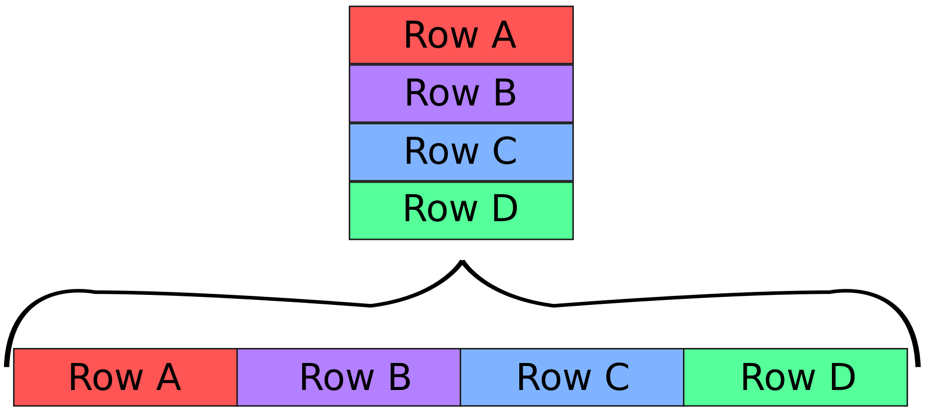

Figure 9.2: Row-order storage of a matrix. When the matrix (top) is allocated as a single block (*i.e.*, embedded in a 1D array, bottom), the rows form contiguous blocks with the second row immediately following the first row and so on. In this scheme, rows are stored in contiguous blocks but columns are not.

9.5 Cache-optimized code

Even when we are not performing very fine tuning to customize a piece of code for a particular cache, we can change our access patterns in order to achieve large speedups.

Consider a program where we sum the contents of a matrix. We will do this using two nested `for` loops, one iterating over all rows and one iterating over all columns. Listing 9.1 iterates over the matrix in column-major order (*i.e.*, with the column index in the outer loop). It runs in 0.08390s.

Let's consider the access pattern in Listing 9.1. First consider how a contiguous matrix is stored in memory in `C` (Figure 9.2).[25]

Figure 9.3 shows the effect of iterating over a matrix in column-major order. If the matrix has many columns, entire rows will not fit in a single cache line. Accessing the first element in each row (*i.e.*, visiting the first column) will result in a cache miss (red), but will cache the neighboring cache block (blue); however, these cached values will come from neighboring columns rather than the same column (assuming the matrix is wider than one cache

[25]Note that in languages like `Fortran`, the matrices are stored in transposed order, which will be different than shown here.

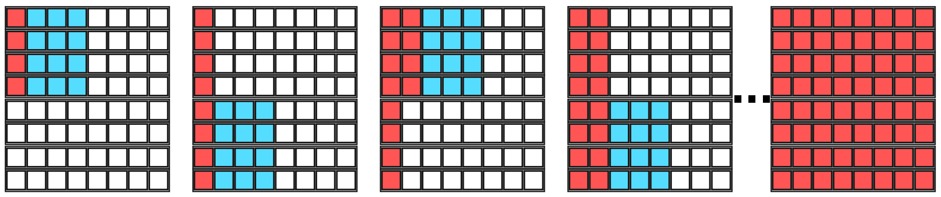

Figure 9.3: Traversing the matrix in column-major order. A single L1 cache and no L2 or L3 cache (*i.e.*, no hierarchical caching) is assumed. The panels show the results after loading the next four elements. Cache misses are labeled in red and currently cached values that have not yet been read are labeled in blue. For a large matrix, column-major traversal does not benefit from the cache and will produce cache misses at every element.

line). If the matrix is large, by the time the next column is being processed, the neighboring columns will already have been evicted from the cache, and so essentially all reads will result in cache misses.[26] This is reminiscent of "thrashing" when a computer runs out of RAM and two programs keep taking data from the disk swap and relegating the other program's data to the disk swap.[27]

Listing 9.2 iterates over the same matrix in row-major order (*i.e.*, with the column index in the inner loop). Simply swapping the order of the two `for` loops produces code with algorithmically indistinguishable runtime, but one that is practically superior (0.02013s, a $> 4\times$ speedup over the column-major version).[28]

Listing 9.1: Summing a matrix in column-major order.

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  unsigned long R = 1<<12;
  unsigned long C = 1<<12;
  double * matrix = new double[R*C];
  for (unsigned long i=0; i<R*C; ++i)
```

[26]No_no_no_nope_please_no.jpg

[27]If no computer scientists have yet started a metal band called "Cache Thrash", we should totally do it! No, you go ahead first. . . I'll be right behind you. . . soon. . . maybe. . .

[28]Two words to describe cache-optimized code: really, really, really fast.

```
    matrix[i] = i;

  Clock c;
  double result = 0.0;
  for (unsigned long c=0; c<C; ++c)
    for (unsigned long r=0; r<R; ++r)
      result += matrix[r*C+c];
  c.ptock();

  std::cout << "Sum was " << result << std::endl;

  return 0;
}
```

Listing 9.2: Summing a matrix in row-major order.

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  unsigned long R = 1<<12;
  unsigned long C = 1<<12;
  double * matrix = new double[R*C];
  for (unsigned long i=0; i<R*C; ++i)
    matrix[i] = i;

  Clock c;
  double result = 0.0;
  for (unsigned long r=0; r<R; ++r)
    for (unsigned long c=0; c<C; ++c)
      result += matrix[r*C+c];
  c.ptock();

  std::cout << "Sum was " << result << std::endl;

  return 0;
}
```

9.6 Transposition

Since we would definitely prefer to traverse a matrix in row-major order, it is difficult to conceive of a good strategy in the case of matrix transposition.

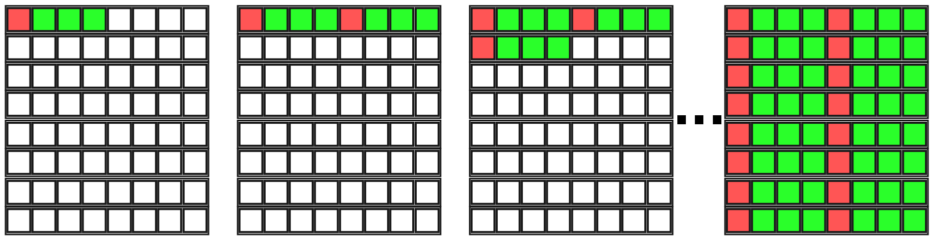

Figure 9.4: Traversing the matrix in row-major order. A single L1 cache and no L2 or L3 cache (*i.e.*, no hierarchical caching) is assumed. The panels show the results after loading the next four elements. Cache misses are labeled in red, and cache hits are green. By traversing in row-major order, each cache miss is followed by a few cache hits.

When transposing a matrix[29], we do $dest_{i,j} \leftarrow source_{j,i}$. If we iterate through the destination matrix in row-major order (which is good for cache performance), we will necessarily process the source matrix in column-major order (poor for cache performance). Likewise, if we iterate through the source matrix in row-major order (which is good for cache performance), we will necessarily process the destination matrix in column-major order (poor for cache performance). In this case, both approaches will be similar, and it may feel like there is no good solution. Listing 9.3 demonstrates naive matrix transposition, and runs in 0.03955s.

[29]Here we will focus on out-of-place transposition from a source matrix to a separate destination matrix; in-place transposition is easy when the matrix is square, but becomes mathematically difficult otherwise, because items cannot always simply be swapped. For example, consider a 5×4 matrix stored in a contiguous array of length 30. Index $(3, 2)$ corresponds to the flat index $3 \times 4 + 2 = 14$ in the array. We will want to write this into index $(2, 3)$ in the transposed array, which in the transposed array (which has shape 5×4) would be at flat index $2 \times 5 + 3 = 13$. Index 13 in the current array corresponds to index $(4, 1)$, because $i \times 4 + j = 13$, and so $i = \frac{13}{4} = 3$ and $j = 13 \bmod 5 = 1$ in the current array. If we copy the value from flat index 14 into its new home in index 13, we will need to save the value from flat index 13 so that it is not overwritten. If we save many of these, we need a buffer (and so the method is not truly in-place), and if not, we need to first copy the data from flat index 13 to its new home. This continues until the indices create a full loop. While this requires number theory for non-square matrices, for square matrices in-place transposition is simplified because we know that we can simply swap the contents of indices (i, j) and (j, i).

Listing 9.3: Naive matrix transposition.

```
#include "../Clock.hpp"
#include <iostream>

__attribute__ ((noinline))
void transpose(double*__restrict dest, double*__restrict source, const
    unsigned long r, const unsigned long c) {
  for (unsigned long i=0; i<r; ++i)
    for (unsigned long j=0; j<c; ++j)
      // dest[j,i] = source[i,j];

      // Note: this code can be rewritten to avoid using the *
      // operator, and using += instead.
      dest[j*r + i] = source[i*c + j];
}

int main() {
  unsigned long R = 1<<10;
  unsigned long C = 1<<12;
  double * matrix = new double[R*C];
  for (unsigned long i=0; i<R*C; ++i)
    matrix[i] = i;

  Clock c;
  double * result = new double[R*C];
  transpose(result, matrix, R, C);
  c.ptock();

  return 0;
}
```

9.7 Block-wise matrix transposition

The good solution is to traverse both matrices in blocks. Figure 9.5 shows how a matrix can be cached effectively by reading blocks that are no wider than the size of a cache line. In this manner, whether we iterate by rows or columns, we achieve cache hits within this square block.[30] For this reason, both the source and destination matrices in the transposition will achieve

[30]For simplicity, we will assume that the cache is "tall", *i.e.*, that the number of cache lines is larger than the size of a cache line.

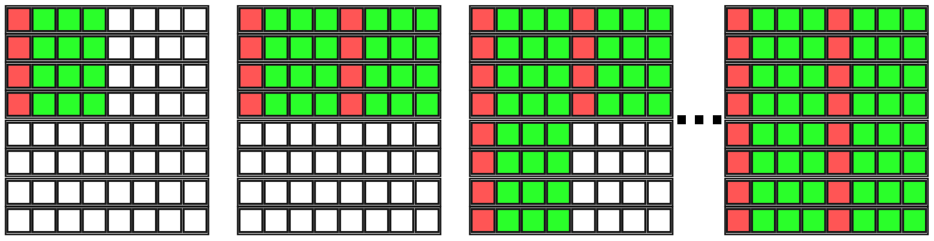

Figure 9.5: Traversing the matrix in blocks. A single L1 cache and no L2 or L3 cache (*i.e.*, no hierarchical caching) is assumed. The panels show the results after loading the next 16 elements. Cache misses are labeled in red and cache hits in green. The same pattern is produced regardless of whether the *block* is traversed by rows or by columns; thus, this method can be used effectively for transposition, because blocks of the source matrix can be traversed by row and blocks of the destination matrix can be traversed by column (or vice versa).

cache hits. Listing 9.4 runs in 0.01895s (a $> 2\times$ speedup over the naive approach).

Listing 9.4: Block matrix transposition. Better performance is achieved via a more cache-friendly access pattern.

```
#include "../Clock.hpp"
#include <iostream>
#include <assert.h>

const unsigned int BLOCK_WIDTH = 4;

__attribute__ ((noinline))
void block_transpose(double*__restrict dest, const double*__restrict
    source, const unsigned long r, const unsigned long c) {
  assert(r % BLOCK_WIDTH == 0 && c % BLOCK_WIDTH == 0);

  for (unsigned long i=0; i<r/BLOCK_WIDTH; ++i) {
    for (unsigned long j=0; j<c/BLOCK_WIDTH; ++j) {

      // Transpose an BLOCK_WIDTH x BLOCK_WIDTH block:
      for (unsigned int a=0; a<BLOCK_WIDTH; ++a) {
        for (unsigned int b=0; b<BLOCK_WIDTH; ++b) {

          // dest[j+b,i+a] = source[i+a,j+b];
          dest[(j*BLOCK_WIDTH+b)*r + i*BLOCK_WIDTH+a] =
              source[(i*BLOCK_WIDTH+a)*c + j*BLOCK_WIDTH+b];
        }
      }
    }
  }
}

int main() {
  unsigned long R = 1<<10;
  unsigned long C = 1<<12;
  double * matrix = new double[R*C];
  for (unsigned long i=0; i<R*C; ++i)
    matrix[i] = i;

  Clock c;
  double * result = new double[R*C];
  block_transpose(result, matrix, R, C);
  c.ptock();

  return 0;
}
```

9.8 Cache-oblivious matrix transposition

What is the best block size to use?[31] The largest one that will fit our cache is a good choice. We could find this by investigating our particular hardware schematics, but we could also find this empirically by just trying different block sizes and using the one that works best. However, both of these options may be unavailable if we are writing code for an unknown computer[32]. Likewise, our block matrix transposition can only be optimized for one cache size, but not for the hierarchical L1, L2, . . . cache setup in nearly all modern high-performance computers.

The solution here is "cache-oblivious"[33] matrix transposition. In cache-

[31] "I'll tell you. . . I don't know."

[32] *e.g.*, a computer owned by a customer or even on a computer that has not yet been invented

[33] This means code that isn't tuned for a particular collection of cache sizes and their setup, and it sounds bad, but in this case, it's actually a good thing: if we have a high-performance cache-oblivious method, then it should perform well on any computer.

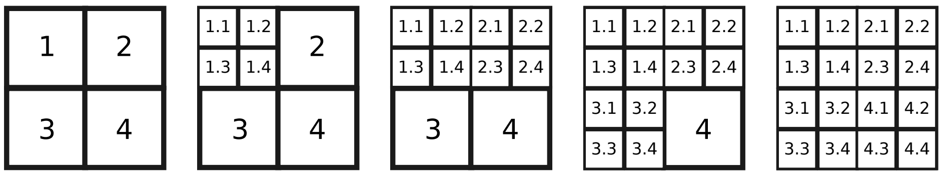

Figure 9.6: Order of source elements visited by cache-oblivious matrix transposition. By recursing as described in Figure 9.7, the top-left block is visited first, the top-right block second, the bottom-left block third, and the bottom right block fourth. This pattern continues in each visited block, so the first block is divided into sub-blocks 1.1, 1.2, 1.3, and 1.4. The full matrix will be visited in the order 1.1, 1.2, 1.3, 1.4, 2.1, 2.2, 2.3, 2.4, 3.1, 3.2, 3.3, 3.4, 4.1, 4.2, 4.3, 4.4. In this manner, square blocks of various magnitudes are processed, achieving good performance regardless of the respective L1, L2, L3 cache sizes on a particular CPU (*i.e.*, a cache-oblivious method). Each sub-block will be visited in a "Z" pattern.

oblivious matrix transposition, we simply visit the matrix in blocks, and then subdivide each of those blocks into sub-blocks. This ensures that even if the initial block was too large to fit in the cache, it will be divided into sub-blocks that can fit, and so we are guaranteed that the first block that fits in a cache will make use of no less than half of the cache line. The process by which we iteratively subdivide into smaller blocks is shown in Figure 9.6.

This can be written in a straightforward manner by simply recursively dividing the larger remaining axis of the matrix (if the matrix is tall, cutting it into two matrices stacked vertically and if the matrix is wide, cutting it into two matrices stacked horizontally). This is illustrated in Figure 9.7.

Listing 9.5 demonstrates cache-oblivious matrix transposition. When benchmarked[34], this code runs in 0.009606s, even faster than the block-wise approach in spite of the recursive implementation used here. A well-tuned[35] block transposition may be faster, but this is quite good. In theory the cost of the recursions will be amortized out if we choose a non-trivial block size for termination (thus ensuring the cost of the recursions is dwarfed by the cost of doing actual work); however, in practice, a non-recursive version may be much more optimized by the compiler, and so would still be beneficial.

[34] "Now we find out if that code is worth the [recursive] price we paid..."

[35] By adjusting the block size

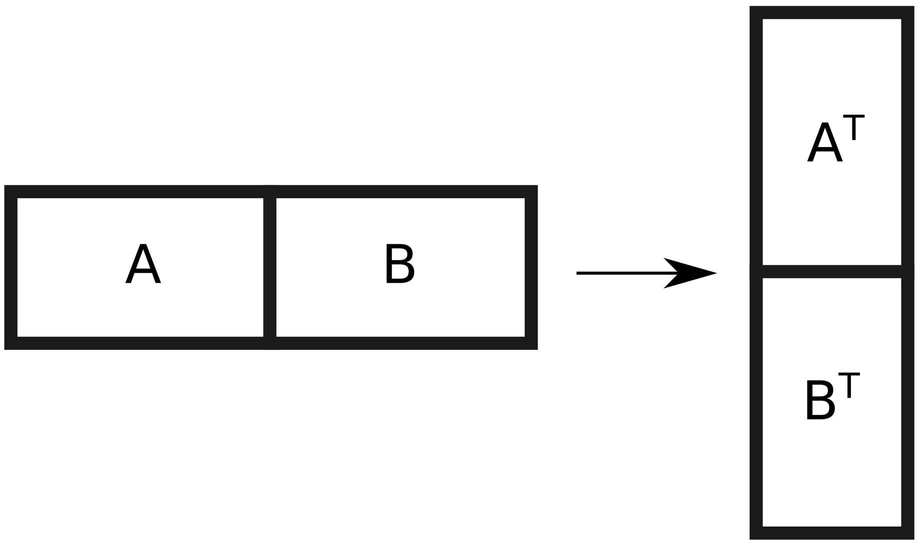

Figure 9.7: Cache-oblivious out-of-place matrix transposition. The longer axis (either rows or columns) of the source matrix is divided in half. Then, the submatrices A and B in the source matrix will produce A^T and B^T in the destination matrix. The process of converting A to A^T and of converting B to B^T are themselves transpositions and would result in recursion of the same method.

This could be implemented with a stack just as a non-recursive quicksort implementation is implemented.

Listing 9.5: Recursive cache-oblivious matrix transposition.

```
#include "../Clock.hpp"
#include <iostream>

// Chosen to be large enough that the cost of the work on a BLOCK_SIZE
// x BLOCK_SIZE matrix transposition amortizes out the cost of the
// recursions needed to reach it (but small enough that the block
// matrix fits comfortably in the cache). Note that this is not as
// challenging as choosing the size for a block matrix transpose,
// because this BLOCK_SIZE parameter needs only be large enough to
// amortize out the recursions.
constexpr unsigned int BLOCK_SIZE = 4;

// Buffered (out of place):
static void buffered_helper(double*__restrict const dest, const
    double*__restrict const source, const unsigned long R, const unsigned
    long C, const unsigned long r_start, const unsigned long r_end, const
    unsigned long c_start, const unsigned long c_end) {
  unsigned long r_span = r_end-r_start;
  unsigned long c_span = c_end-c_start;
  if ( r_span <= BLOCK_SIZE && c_span <= BLOCK_SIZE ) {
    // Small matrix fits in cache, proceed in naive manner.
    for (unsigned long r=r_start; r<r_end; ++r)
      for (unsigned long c=c_start; c<c_end; ++c)
        // dest[c,r] = source[r,c];
        dest[c*R + r] = source[r*C + c];
  }
  else {
    if (r_span > c_span) {
      buffered_helper(dest, source, R, C, r_start,r_start+r_span/2,
          c_start,c_end);
      buffered_helper(dest, source, R, C, r_start+r_span/2,r_end,
          c_start,c_end);
    }
    else {
      buffered_helper(dest, source, R, C, r_start,r_end,
          c_start,c_start+c_span/2);
      buffered_helper(dest, source, R, C, r_start,r_end,
          c_start+c_span/2,c_end);
    }
  }
}
```

```
__attribute__ ((noinline))
void recursive_transpose(double*__restrict dest, double*__restrict
    source, const unsigned long r, const unsigned long c) {
  buffered_helper(dest, source, r, c, 0, r, 0, c);
}

int main() {
  unsigned long R = 1<<10;
  unsigned long C = 1<<12;
  double * matrix = new double[R*C];
  for (unsigned long i=0; i<R*C; ++i)
    matrix[i] = i;

  Clock c;
  double * result = new double[C*R];
  recursive_transpose(result, matrix, R, C);
  c.ptock();

  return 0;
}
```

Cache performance is one of the crucial features that separates algorithmic performance from real performance of code in the wild, and because of this, programmers who have some intuition about how to exploit cache may be much more effective. Significant linear speedups from cache performance can often tip the scales when choosing between two algorithms. For example, an $O(n)$ algorithm that stores data in discontiguous chunks and accesses memory in a non-sequential manner may well be much slower than an $O(n\log(n))$ algorithm with good characteristics. Of course, if n were very large, the algorithmic complexity would overwhelm the constant speedup, but then it may not be possible to store such large data in RAM (and so we may go to disk, where RAM now feels like a cache). Algorithms that are optimized for large amounts of disk access are known as "out-of-core" algorithms, and are important for big data[36].

[36] *I.e.*, they've always been important, like when "big data" meant 1KB. Also, they will likely continue to be important, as our hunger for computational resources continues to grow.

9.9 Multicore considerations

Multicore processors often have dedicated L1 caches for each core, but when it comes to larger caches like L3, they may be shared. This can lead to challenges where the latest version of a variable is modified by one core, but has not yet been propagated to another core (because the caches are no longer strictly hierarchical). For reasons like this, multithreaded code is often more challenging to optimize for cache performance when applied to large data (small data can fit everything in the L1 caches, and so may experience fewer struggles of this nature). Good chip design is essential to producing processors that do not simply sound impressive, but which are truly fast.

Questions

1. [**Level 0**] Linked lists in high-performance code make me feel (circle one):

   ☺

   ☹

2. [**Level 1**] A program currently allocates two blocks of memory, one for the data and one for a buffer (as performed by the merge sort in Chapter 3 Question 2). These are allocated `unsigned long*source = new unsigned long[n]; unsigned long*buffer = new unsigned long[n];`. Where in memory will these allocations come from? Are they guaranteed to be adjacent? How large would they be together if they were adjacent? What could be done to guarantee that they will be adjacent? You may assume that both the source and buffer values will only be used in conjunction (*i.e.*, we never want one without the other).

3. [**Level 1**] Draw a "connect-the-dots" access pattern of calling our cache-oblivious matrix transposition on a 2×2 matrix. Do this by drawing the square matrix, numbering the indices by the order with which they are accessed, and then connect the ascending numbers without lifting your pen (use a base case block width of 1, so that we never

switch to the naive approach). Repeat on a 4×4 matrix. Repeat on an 8×8 matrix.

4. [**Level 2**] Plot runtimes per element for transposition vs. total size of RAM used for a naive out-of-place matrix transition (out-of-place transposition of an $n \times n$ matrix of `double` types uses $2n^2$ `double` values, each using 8 bytes). This will report the total elapsed time divided by n^2. On the same figure, also plot the runtime per element of out-of-place cache-oblivious matrix transposition. Now annotate the sizes of the L1, L2, and L3 caches on your system. Can you explain where the runtime per element increases?

5. [**Level 3**] Write a function to multiply 2 matrices using the standard $O(n^3)$ method: $C = A \cdot B$ means that $C_{i,j} \leftarrow \sum_k A_{i,k} \cdot B_{k,j}$. Benchmark it on the product of two $2^{10} \times 2^{10}$ matrices. Now write a second version that iterates over both matrices in a cache-friendly manner (*i.e.*, try to avoid iterating over columns in a matrix). Does transposition help? What sort of speedup do you see?

Chapter 10

Compiler Optimizations

10.1 The compiler: a hunting dog and a friend

Thus far, we have casually discussed the compiler through anthropomorphisms: "The compiler is smart enough to figure out [something]"[1]. Even though a compiler will not possess the intelligence to replace a capable programmer[2], it can be an indespensible ally, a trusted friend who removes some of the more tedious[3] aspects of programming so that we can think at the high level rather than at the low level. In this respect, the compiler resembles a trusty hunting dog, a companion who complements our intelligence with a certain brute force and speed. We track our quarry using our extensive human intelligence, but once it is within sight, we release our hound, who is much better able to close the final meters on foot.

[1]It's fun watching this cargo cult behavior develop in seasoned programmers: "Hmm. . . the compiler wants us to avoid pointers here," or "Hmm. . . the compiler isn't going to like it if we divide there," or even "Can't you see?! I *want* to marry you. . . It's the *compiler* that doesn't want me to!"

[2]If the compiler could adequately replace skill in programming, then why would the things in this book actually produce faster code?

[3]Of course, there is not really anything tedious about any task if you zoom in far enough (as Richard Feynman put it, "Nearly everything is really interesting if you go into it deeply enough."); however, some tasks may not be of utmost interest at any particular moment: register allocation is certainly an intriguing problem, but it may not be our first priority when we are working a task like optimizing a sorting algorithm[4].

[4]Or graduating

In many cases, well-optimized code generated by the compiler may actually outperform hand-edited assembly code while requiring a fraction of the development time, being much more human readable, and being easier to debug. As is the case in the hunting analogy, a human with a compiler is a far better hunter than either just the human alone or just the computer alone.[5] By better understanding how the compiler works, by understanding its virtues and its current limitations, we can successfully produce very fast code.

10.2 How compilers work: front end and back end

Compilation begins on the front end. The front end is where `C++` code is considered on a high level, where the first type of optimizations can be performed. Compilers first digest code using a parser[6], producing an intermediate data structure, the "control flow graph" (CFG). The CFG is a type of directed graph indicating the dependencies of the code. For example, consider the three `C++` statements `w = x+y; x = y+w; z-= x;`. The first of these statements, `w = x+y;` needs to finish before the second statement can begin. This is because it produces two types of dependencies: First, there is a a "read-after-write" (RAW) dependency because the first statement modifies `w` and the second statement relies on the updated value of `w`. Second, there is a "write-after-read" (WAR) dependency because the second statment modifies `x`, and so the first statement must be run before this modification takes place, so that it can use the pre-modified value of `x`[7]. Dependencies are marked by directed edges in the CFG for this block of code (Figure 10.1). The CFG also reveals the fact that the the third statment depnds on the second statment. Because of this, even when a processor is built to support parallelism (such as with SIMD, mentioned in Chapter 5), the three statements cannot be adequately parallelized in their current form, because if they were to

[5] "The deadliest weapon in the world is a marine and his rifle."

[6] Such as `lex`, the free `GNU` parser used for writing compilers.

[7] There is a last kind of dependency: "write-after-write" (WAW) dependencies occur when two statements both modify the same variable. For the code to function as intended, the last modification should be used for any subsequent read operations and the first modification should be used for any previous read operations occurring between the two modifications.

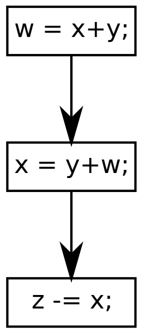

Figure 10.1: A basic CFG. Dependencies are shown for the block of code `w=x+y; x=y+w; z-=x;`.

run simultanously, the dependencies would not be respected. The CFG for an alternate block of code, `a=b+c; b=a+x; c=a+y; t=z-7; a=13;`, reveals the opportunity for parallelism: the first and fourth statements can be run in parallel. Then, after the first statement is finished, the second and third statements could be run in parallel. The fifth statement can only be run once the second and third statements complete (because of a WAR dependency). Finding a good order in which to run the statements (*e.g.*, an order that exploits parallelism effectively) is known as "scheduling", and is a crucial back-end optimization.

The front-end compiler translates the code into the CFG, transforms it with front-end optimizations, and then translates the CFG into assembly code. This is convenient because many processors (*e.g.*, the `x86-64` family) have overlapping assembly languages[8], and so compilers can be simplified by exploiting this modularity.

Back-end optimization occurs on the assembly code itself. In this stage, the code is optimized on the assembly level and is "assembled" to machine code. Machine code is a binary data consiting of blocks of data where each word contains an opcode, and register or immediate arguments. The opcode tells the chip which operation is being performed: *e.g.*, `ADD` could be represented as `1010`, which would be a binary tag to activate the add operation within the ALU[9]. The other parts of the binary word indicate what should

[8]Note that they do not share identical instruction sets; therefore chip-specific optimizations are still highly important. These are mentioned later in this chapter.

[9]ALU = Arithmetic and logic unit. It's the hardware that actually performs additions, multiplications, bitwise and/or operations, *etc.*

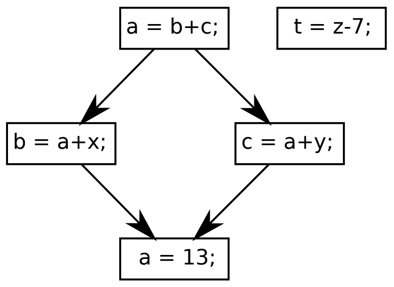

Figure 10.2: A basic CFG depicting opportunities for parallelism. Dependencies are shown for the block of code `a=b+c; b=a+x; c=a+y; t=z-7; a=13;`. The first and fourth statements can be executed in parallel, and then the second and third statements can be executed in parallel once the first statement completes. Once the second and third statements have both completed, it is safe to run the fifth statement (modifying `a`). These opportunities for parallelization can only be exploited when the hardware is capable.

be added: the word 1010 001 010 011 (spaces added only for emphasis) could indicate "add $R_1 = R_2 + R_3$". Bits in the word could also be set to indicate that the arguments are "immediate". For example, performing `x += 10` could be written by storing the value `10` somewhere in RAM and then loading both that value and `x` into registers, adding them, and then storing the result to the address of `x`; however that is quite verbose and inefficient, and so. it is preferred to enable CPUs to directly accept small numerical arguments instead of a register number. An immediate `ADD` could match opcode `1011`[10] and so the word `1011 001 001010` (space added for emphasis) would indicate: "add $R_1 = R_1 + 10$". In this mode, a chip could use those latter bits to indicate the value `10`, thereby avoiding the need to load the value `10` from RAM before the add can be begun[11]. Machine code is even more opaque than assembly code, and so developers use it even more sparingly than assembly code[12]; however, it is still useful to understand how it works. For example, it can help us understand why `x += 10` may be faster than `x += 104928` on some particular chip, because the second argument cannot

[10]Alternatively, an immediate `ADD` operation could be specified by "flag" bits not included in the opcode and arguments

[11]Consider the nice benefits to cache performance when one uses immediate arguments.

[12]I have had occassion to directly write machine code only a few times in my life. Why? Because I was writing a compiler.

be fit into the immediate arguments and so must be performed as multiple immediate additions or by loading the argument `104928` from RAM[13]. Back-end optimizations are performed at this assembly stage. In this stage, the compiler tries to find the best way to write the low-level code into machine code.[14]

Front-end optimizations are usually more algebraic, such as reordering loops (described below). Back-end optimizations consider more details about assembly code and utilizing the hardware as well as possible. There is a fuzzy line between front-end and back-end optimizations, and a sophisticated compiler could even alternate between the two.

In the most formal form, compilation is a kind of optimization puzzle on a particular algebra defined by the programming language: compilation is the task of producing an equivalent piece of code that will be as fast as possible. What does the function look like that we're optimizing? This is tricky; it is difficult to predict precisely how modifications to code will affect the runtime, because changes will influence the ability of the compiler to parallelize them, the cache performance, and more. The search space of all possible pieces of code that would produce the same output is essentially impossible because even determining which pieces of code will terminate is undecidable[15]. For this reason and for practical reasons involving the massive search space, compilers struggle with performing very large transformations to provably equivalent pieces of code. Instead, heuristics are used to distinguish beneficial modifications from harmful modifications. In the simple cases, this is easy: a transformation that produces the same result in one fewer additions is good. But a transformation that produces the same result with one fewer multiplication but two more additions is much more difficult to evaluate. This is reminiscent of heursitics used in computer chess: "Is it worth losing a bishop to take out a knight if the resulting board position is

[13]This precise example is likely not to be a large issue in practice on modern 64-bit CPUs. But you still may encounter things like when you use large integers on your CPU or if you use moderately sized values on embedded systems. Or you might encounter precisely this example if your laptop is so old that whenever you turn on text to speech, it keeps telling stories about growing up during the Civil War (which it refers to only as "The Waawah of Nawthahn Aggreshawn") and making. . . other comments to which some readers may object.

[14][Badge achieved]: Congratulations, you are now an assembly programmer!

[15]This is Turing's halting problem: assuming someone supposedly had a method to decide if a piece of code would terminate, we could then use their method inside our code, and make it do the opposite of whatever it predicted.

so-and-so? Hmm…" Cutting-edge profiling compilers are actually able to iteratively run different transformations of the code and observe how they get better or worse.[16]

By learning the techniques available to modern compilers, it is possible to anticipate where they will succeed in optimizing, and thus lead the compiler toward fast code.

10.3 Tasks where compilers excel

10.3.1 Basic mathematical equivilences on constants

One of best places where compilers excel is operating on `constexpr` integers (whose values are fixed at compile time). For instance, the code `constexpr long x=176; long y=x+8;` can be replaced by performing the operations on `constexpr` arguments at compile time: `long y=184;`. By moving the computations to compile time, the runtime can improve. These kinds of operations are called "constant propagation" or "constant folding".

In the same vein, consider the block of code constexpr int x=2; y/=x;, where we divide an integer `y` by 2. Division is relatively expensive compared to addition or bitwise operations (*e.g.*, bitwise and with `&`, bitwise or with `|` and bit shifting with `<<` and `>>`). We can exploit the fact that division by 2 is the same as shifting right by 1 bit; because the denominator is a `constexpr`, the compiler can observe this at compile time, and our code will be replaced with the faster line `y>>=1;`.

Likewise, checking if an integer is divisible by 2 can be called without the use of the modulo operator[17]: In the naive case we would write `if (y%2 == 0)` to test if `y` is even. The compiler can replace that with `if (y&1 == 0)`, thus reducing the burden from a modulo operaton to a bitwise and operation. When performing modulo operations with a `constexpr` right-hand side, the compiler can find this speedup automatically via constant propagation. `constexpr int x=2; if (y%x == 0)` will be replaced with the faster `if (y&1 == 0)`.[18]

[16]This exemplifies our credo from Chapter 2 about focusing on what we can empirically measure.

[17]Recall that modulo can be performed by a division, a multiplication, and a subtraction, making it even slower than division unless there is a fast single-purpose circuit specifically for modulo operations on your CPU.

[18]Already, we run into the difficult question of how explict we should be when optimizing

Constant propagation can have powerful implications on `if` statements and loops: if both `x` and `y` are `constexpr`, then the statement `if (x<y)` will be known at compile time, thereby avoiding the loop. Similarly, expressions like `x+0`, `b &&false`, and `1+y+2` can be simplified at compile time.

Passing boolean arguments as templates[19] can force the compiler to recognize the `constexpr` argument and create two versions of the called function: one when the argument was `true` and one when the argument was `false`. In each of these functions, any `if` statements about that `bool` parameter can be eliminated by simply removing the corresponding block of code inside the `if` in the case where the template parameter was `false`. More sophisticated versions of these approaches will be discussed in Chapter 12.

10.3.2 Register allocation

A classic problem for compilers is register allocation. That is, when we perform a series of operations such as `++w; y=w*x; x=y+3; z=w+x+y;`, if our CPU only has 2 registers[20], then we cannot dedicate a register to each of our 4 variables. Register allocation is the task of mapping variables and temporary values to registers so that we minimize the times that we need to read from or write to RAM[21].

Register allocation is NP-hard, but it is actually one of the areas where compilers are very reliable. First-generation `C` compilers were not as good as their present-day counterparts, and so this is why `C` contains options to perform many operations in one line: *e.g.*, the line `vec[++index]=c;` will

our code. On a `constexpr` integer `N`, it is generally better practice to write `N/2` rather than `N>>1`, because it will be more readable and both will perform the same operations (*i.e.*, the division will be performed at compile time). On the other hand, in the general case where `N` is not `constexpr`, it's generally better to write in the most efficient form to be explicit about the optimizations we're pursuing. This makes those optimizations apparent to another human reading our code (*e.g.*, they know that we do not plan to actually perform the more expensive integer divide operation), and our mothering may help the compiler not to miss the speedup.

[19]We generally think of using templates to pass typenames, but we can also pass `bool` and `integer` types. This can be beneficial, because template types must be `constexpr`, which will be recognized by the compiler.

[20]This is too small to be realistic on a modern desktop CPU, and is only used for illustration.

[21]We will write and read the variables to RAM, but if we are lucky, we may use the cache instead; however, this is not something that we perform manually: to us, we are still reading from and writing to RAM.

Listing 10.1: A simple recursive factorial. Tail recursion optimizations will convert this to the more efficient loop-based code.

```
unsigned long factorial(unsigned i) {
  if (i <= 0)
    return 1ul;
  // Since i>0, using (unsigned int)(i-1) is safely going to decrease:
  return i * factorial(i-1);
}
```

increment `index`, then perform `vec[index]=c;`. Expressions like the one written can help the compiler see that a variable (such as `index`) should be kept in a register during the increment operation and then written used to inex `vec` before writing `index` to RAM. `C` even included a `register` keyword, which hinted to the compiler that a highly used variable should be completely kept in a register for its lifespan if possible.

These ornate expressions today are more anachronistic than useful, and they often decrease readability of code without actually increasing its performance; however, when a portion of code is taking a substantial percent of the runtime, then it is still advisable to test expressions like this and then carefully benchmark to see if there is any change in performance. Generally, there will not be one: automatic register allocation is usually just as good as hand-edited assembly code.[22]

10.3.3 Tail recursion

Another classic optimization is "tail recursion", in which recursive calls of certain functions can be replaced with their iterative loop-based forms. For example, if we considered the recursive factorial function in Listing 10.1, then this code will be performed recursively as it is written. But with optimizations enabled, the compiler can observe that the recurrence is always called last, and so any modifications of global variables it performs cannot influence the control flow of the calling function. As a result, this recursion can be replaced by an equivalent non-recursie, loop-based form.

Because recursion necessarily grows the stack, it is generally less cache optimized. There is also overhead for copying parameter values to the stack

[22]Or better.

and reading return values from the stack. Furthermore, loops in the same scope allow the compiler to see the function as its own autonomous unit; this modularity can enable the compiler to restructure the loops or to find reusable code. When recurring expressions are computed in each recursion, it is more difficult for the compiler to identify them, because they do not exist in the same scope.

10.3.4 Named return value optimization (NRVO)

When a function returns a local object (such as our first valid game of life in Chapter 4, which returned a `std::vector<std::vector<bool> >`), this will need to be copied. Consider calling `std::vector<std::vector<bool> > board = advance(current_board,R,C);`. The literal meaning of those lines is that the `advance` function will construct a temporary local object of type `std::vector<std::vector<bool> >` for the return value, and then copy it into `board` using the `=` operator for `std::vector`. With named return value optimization (NRVO), we can count on the compiler to eliminate this copying step and effectively just opoerate directly on `board`. Preventing this copying can improve performance significantly when these objects have data allocated on the free store (with `new` or `malloc`).

10.3.5 Basic loop reordering

In Chapter 9, we summed a matrix by looping over rows and by looping over columns. In that case, the compiler did not automatically detect that the cache-unfriendly column-row loops could be rewritten as the more efficient nested row-column loops; however, sometimes the compiler can notice such things and reorder the loops accordingly. Swapping the inner and outer loops in this way is called "loop reordering", and it can be applied when the loops do not depend on one another and when the code inside both loops will not change as the result of reordering. Formalizing this last criteria is tricky, but special cases like summing are more easy. Generally, where state is updated and where the result of this iteration may be used in the result of some subsequent iteration, reordering is not safe.

The reason that the compiler doesn't automatically reorder our cache-unfriendly column-row sum from the previous chapter is because when we add `double` types, order of operations matters (this is described more thoroughly below).

10.3.6 Loop unrolling and peeling

One of the most beneficial speedups in practice is "loop unrolling". We tend to think of loops as having no overhead, ignoring the cost of the increment operation and of the test and backward branch to stay in the loop (instead of leaving the loop); however, when the loop itself is doing a simple task such as in the case `int tot=0; for (int k=0; k<N; ++k) tot+=x;`, then the cost of the overhead (which consists of an addition and a comparison for each iteration) becomes similar to or even more than the cost of the actual work (an addition). For this reason, it can be advantageous to perform a fraction of the iterations of the loop, but in each iteration, to do multiple steps: `for (int k=0; k<N/2; ++k) { tot+=x; tot+=x;}`. Now for each iteration's overhead, we do twice the actual work. As a result, this can produce a large speedup in practice. This can be done manually[23], but it can also be performed by the compile.

There are considerations that need to be taken when unrolling loops. We must ensure that the optimized code produces identical results, and in this case it will not when `N` is not divisible by 2. When `N % 2 == 1`, we should execute a single `tot+=x;` operation after the loop. Moving a few loop iterations from the start or end of the loop (and thereby changing the bounds of the loop) is called "loop peeling", and is the method by which we "peeled" off the final iteration when `N` is odd, thus ensuring we could safely unroll our loopy by a factor of 2. Of course, we could also insert if statements inside the loop to check whether it's time to terminate; however, as we saw in Chapter 5 in our nucleotide string bit-packing code, peeling is far better than inserting if statements into the loop. It sometimes helps to perform loop peeling manually, but the compiler is often quite good at loop unrolling.

This is yet another reason why `constexpr` values can be so helpful: if `N` is `constexpr`, then we will know at compile time whether the `N/2` operation will round down and require us to execute one more `tot+=x;` operation. Furthermore, if `N` is a small `constexpr` value such as `constexpr int N=10`, the loop could be unrolled fully to 10 separate lines of the form `tot+=x;`, and so there would be no overhead from the loop.

There is a downside to loop unrolling and peeling: they produce larger executables, which stresses the instruction cache. If we manually unroll, it also can hurt the compiler's ability to see our code as a whole, which can prevent it from recognizing the pattern in the many statements and thus

[23]This is referred to as "Duff's device".

obscure our code from optimizations such as loop fission and fusion.

10.3.7 Loop fission & fusion

Loop fission refers to the process by which a loop of the form `for (int k=0; k<N; ++k) { tot_a+=vec_a[k]; tot_b+=vec_b[k]; }` can be split into two loops `for (int k=0; k<N; ++k) tot_a+=vec_a[k]; for (int k=0; k<N; ++k) tot_b+=vec_b[k]; }`. Loop fusion refers to the complementary process by which two loops can be fused into a single loop.

Loop fission is beneficial when the two (or more) inner tasks are difficult enough that the compiler and CPU benefit from focusing on one task at a time (*e.g.*, focusing on one task can benefit register allocation, ensuring there are plenty of registers for every variable being used). Loop fission can also be beneficial for cache performance when large arrays are being accessed[24]. But this fission occurs at the cost of performing all of the loop overhead multiple times. Loop fusion has complementary pros and cons: loops with simple tasks inside can often benefit from fusion, which eliminates the need to pass through a second time (including executing the loop overhead twice instead of once).

The compiler benefits from our guidance in this regard, but overall it performs loop fission and fusion fairly well. Note that fusion cannot be safely performed if the modifications in one loop can influence the modifications performed inside the other. The ways in which this will limit us are larger than they may seem, and relate to the discussion of aliasing below.

10.3.8 Loop-invariant code

Another substantial optimization is the factoring out of "loop-invariant code". This is code of the form `for (int k=0; k<N; ++k) tot+=k-z*y/2;`. The expression `z*y/2` is loop-invariant code, in that it depends neither on the loop variable `k` nor on any modifications performed in previous iterations of the loop. This code could therefore be rewritten to avoid the repeated computation of `z*y/2`: `const double t=z*y/2; for (int k=0; k<N; ++k) tot+=k-t;`.

[24]If the cache is fully associative (not likely because of the difficulty in creating the circuitry) or set-associative (more likely), then this is not as important. But if the values in the array are being accessed multiple times, focusing on one array at a time can be a big benefit.

loop-invariant code can be obscured by numeric challenges mentioned below, where the order of operations matters. For example, `for (int k=0; k<N; ++k) tot+=k*x+k*y;` will likely not discover and factor out to a loop-invariant version `const double t=x+y; for (int k=0; k<N; ++k) tot+=k*t;`. Also, like many optimizations mentioned in this chapter, the ability of the compiler to find loop-invariant code is also hindered by aliasing.

10.3.9 Basic code reuse

Code reuse occurs when the same expression is computed multiple times. This is quite similar to identifying loop-invariant code, except with loop-invariant code, we have the loop structure to give an indication that something is computed again and again. In contrast, code reuse needs to find these expressions occurring in multiple different places, which is much more difficult.

For example, the code `a=w-(x*y+z); b=w+(x*y+z);` could compute the expression `x*y+z` a single time, and then reuse that result both times it is used. Even more sophisticated code reuse on `a=w-x*y; b=w-x*y+z;` would replace the second line with `b=a+z`. Code reuse is fairly difficult, but compilers do succeed in doing it sometimes. One of the more difficult concerns are numeric, where following the precise order of operations given by the programmer may obscure an algebraically identical expression[25].

Code reuse is one additional reason why it's beneficial to write organized, straightforward functions for recurring tasks, especially when they don't modify class members or global variables[26]: if we call those functions repeatedly in the same scope, there is a good chance the compiler will identify the reused expression. If, on the other hand, we were to write the recurring expression each time it is used instead of calling a function, the compiler then must first identify the recurring expression.

The challenges in code reuse also reveal yet another reason why `constexpr` code is so powerful. This becomes very important to why template recursion can perform so well (discussed in Chapter 12). Even if code

[25]They may be algebraically identical on theoretical numbers, but not so on `float` and `double` types, as described below.

[26]For member functions such as accessors, it also helps the compiler if we label the functions as `const`.

reuse fails, integer operations on constants (and `constexpr` function calls) can be simplified.

10.3.10 Basic SIMD

Basic SIMD (briefly mentioned in Chapter 5) can often help perform multiple parallelizable[27] operations simultaneously. For example, if we declare four 32-bit floats, two sums could be added simultaneously using a 64-bit SIMD register if it's available on our CPU. On the code `float w=1.5; float x=2.3; float y=-0.9; float z=0.5; w+=y; x+=z;`, these additions could be automatically performed in parallel by the compiler.

10.3.11 Basic dead code elimination

Dead code elimination is straightforward, and thus compilers excel at it. Essentially, all that needs to be done is to find computations in the CFG where the result is never used.

10.3.12 Copy propagation

"Copy propagation" is the process by which low-level copying operations on primitives may be factored out if a temporary variable is not needed. For example `int y=z; int x=y;` can be simplified to `int x=z;` if the variable `y` is not subsequently used. This optimization is a cousin of dead code elimination.

10.3.13 Chip-specific optimization

Once the code is being optimized as machine code, the precise details of our machine can matter substantially (*e.g.*, whether we have 128-bit SIMD registers or only 64-bit SIMD registers). Compiling with the `-march=native` can be very useful; this optimization tells the compiler that our executable does not need to run on the entire family of processors, but that we only want to run it on our processor. This optimization would not be suitable if we wanted to ship binaries of our software to people with potentially different

[27] According to the CFG.

CPUs, but it is great if we intend to run high-performance software on our own CPU. In practice, this flag can achieve a large benefit on high-end CPUs.

Where `-march=native` enables greater back-end optimizations, the corresponding front-end optimization flag `-mtune=native` enables our code to be "tuned" for the local CPU. This can influence the decisions such as reordering and unrolling loops. By using our own cache specifications, the compiler could potentially convert two nested `for` loops into four nested block for loops (as we did manually for the block matrix transposition in Chapter 9), and choose the sizes of the inner block `for` loops so that they access the largest loop ranges that will fit in our L1 cache. In practice, this compilation flag does not achieve as high of a speedup as `-march=native`.

10.4 Tasks where compilers still struggle

10.4.1 Aggressive inlining

Function inlining is slightly similar to loop unrolling: calling a function incurs overhead, and simple, small functions can be simply copied and pasted everywhere they are called. This is especially helpful on very short, simple functions, where the overhead of pushing parameters to the stack and reading the results off of the stack is as expensive as the actual work done by the function. This frequently occurs in simple mathematical functions, such as the kind that take one numerical argument and return a single numeric expression of that argument (*e.g.*, squaring).

The compiler tries to automatically identify good candidates for inlining, but where we know it should be performed, we can mark the function for forced inlining. There is an `inline` keyword available, but it's a common misconception that it forces inlining[28]. Even in the original C standard, it was merely a hint to the compiler. Today it is widely regarded as doing nothing at all.[29] To force inlining on newer versions of `clang` and `gcc`, write `__attribute__((always_inline)) inline` immediately before the function declaration[30]. This will force the compiler to inline the code.

[28]If there was ever an understandable misconception to have about `C++`, this would be a pretty good one to choose: "Hey, let's make a new keyword: `inline`." "Great, does it inline the code?" "No. It does nothing."

[29]In this case, "hint" has come to mean a suggestion box directly over a paper shredder.

[30]Or prototype

Sometimes, it is even benefial to inline larger functions. One reason this can be useful is when one function calls another and they both compute the same expression, which could be reused, but it won't be reused unless it occurs in the same scope. Likewise, loop fission and fusion must occur in the same scope.[31] Inlining a function gives the compiler a great chance to see the code as a whole and to perform more front-end optimizations.

You can check whether a function is inlined by looking at the `.s` file of assembly code file produced by the `-S` compilation flag: if the function has its own declaration in the assembly code, it was not inlined.

For simple expressions, `C`-style macros are still a great option, which will automatically be inlined. Note that the macro `#define square(x) x*x` will produce the wrong result for `square(5+4)`, which it will define as `5+4*5+4`. For this reason, liberal use of parentheses are necessary to protect macro arguments[33]: `#define square(x) (x)*(x)`.

If we are too aggressive with our inlining, it will force larger compile times and larger executables (which stress the instruction cache more). It may also prevent the compiler from identifying reusable code (because function calls have essentially been find-replaced with their return values, and so the fact that we call the same function in several places may need to be uncovered again). For these delicate reasons, inlining is more challenging than loop unrolling, and benefits from a hands-on approach.

10.4.2 Advanced SIMD

SIMD support is constantly improving, but it is still not reliably as good as a good assembly programmer who knows the SIMD instructions inside and out[35]. Sometimes SIMD operations will be blocked because our data is not ordered in a manner that can be seen as contiguous. For example, if we declare four `float` types, but they are broken up

[31]It's difficult[32] to split or fuse loops that occupy different scopes and that are nowhere near one another in the code.

[32]And by that, I mean impossible.

[33]There are plenty of extra parentheses wandering lonely in the dark reaches of YouTube comments these days, so just use some of those. Wait, can everyone else see those? Or am I having some sort of seizure?[34]

[34]"Are macros */ourfunctions/*?" Nope: Technically, macros aren't functions. Nice try, anon. Stay in school.

[35]This is more difficult than it sounds with CPUs improving as rapidly as they are.

by `char` types, it could hinder the ability of the compiler to automatically use SIMD: `float w=1.5; char a='a'; float y=2.3; char b='b'; float x=-0.9; char c='c'; float z=0.5;` will either obscure the SIMD operations in `w+=y; x+=z;` or, alternatively, it may require an extra copy operation to put the `float` types into the SIMD register in a contiguous fashion.

We can help the compiler substantially by arranging SIMD-capable arguments in an order where the left-hand and right-hand arguments to the operation are contiguous. Even without interrupting the `float` types with `char` types, if we had declared the variables in a different order, `float w=1.5; float y=2.3; float x=-0.9; float z=0.5;`, then the arguments (as seen packed into 64-bit registers) would no longer be contiguous. Also important for SIMD is the alignment of data: if we declare a single 8-bit `char` before the four `float` types, then the addresses may not fall into blocks that are still contiguous, but which are shifted 8 bits from the 64-bit alignment helpful to our SIMD computations. For this reason, it can be useful to keep differently sized primitives segregated into their own regions of memory and, if we must place a `char` upstream of the `float` types, to pad it with 3 more `char` types so that the alignment is not altered.

For example, our matrix transposition could use SIMD to perform 4×4 transpositions in the base case (if we can fit four of our data type in a SIMD register). This is not yet something that you can reliably count on the compiler doing for you.

As an alternative to trusting the compiler with SIMD and an alternative to using assembly, SIMD "intrinsics" provide a `C` interface to the SIMD hardware. This can be used to write semi-portable[36] high-performance code.

10.4.3 Custom hardware

Compilers are not yet smart enough to detect the intent of our code and to replace it with equivalent code that runs on other hardware. An obvious example of this is the GPU: the compiler simply will not detect highly parallelizable code and send it to the GPU. Likewise, when loading a large file, instead of looping and reading it one byte at a time, it is sometimes possible to dump it directly into RAM with help from the hardware and the operating

[36]That is, it's portable when the processor has the desired SIMD hardware. For use on generic case where the target processor is unknown, judicious `#ifdef` statements essentially write the SIMD intrinsics for each level of SIMD capability.

system. This "direct memory access" (DMA) will be discussed in Chapter 17.

For these reasons, it is sometimes advantageous to use standard functions like `memcpy`, rather than write our own. This is because the standard functions communicate our high-level intent to the compiler, and so the compiler can easily interface with a library optimized for our hardware (if it is available).

10.4.4 Numeric challenges

Numeric challenges are one of the largest reasons why our guidance is still essential to helping the compiler optimize our code. Consider this operation: `w=x<<(y-z);`, which performs a subtract and a bit shift on some integers. Will the compiler be smart enough to rewrite this operation as `x<<y>>z`? It will not, but not necessarily because it doesn't understand the qualitative similarity. It's because the two are not always numerically equivalent. If `y` is larger than the number of bits in our `int` type, for instance `y=1024`, then `x<<y` will shift every bit off the edge and yield `0`, which will then be shifted right by `z` and produce a final result of `0`. But if `z` is close to `y`, *e.g.*, `z=1020`, then the statement `w=x<<(y-z);` will perform `w=x<<4;`, which will not necessarily be `0`.

In this manner, the finite precision available destroys some algebraic properties that we know[37]. And with floating point values, this becomes much more difficult than the integer case we just demonstrated. With integer values, we can reorder our operations safely (as long as we don't experience overflow as demonstrated above): `x+y+z` should be equivalent to `x+z+y`. But with floating point values, which do not take on the precise values but merely an approximation of it, this can actually change the result. `x+y-x` should yield `y` in the integer case (again, provided there is no overflow). But with floating point values, underflow can also occur: `1 + 1e-16 - 1` yields `0.0`, not `1e-16`. Likewise, Listing 10.2 demonstrates that `7*x` is not guaranteed to precisely equal `x+x+x+x+x+x+x` when we work with floating point math. If we don't mind which of these is used, we can use the flag `-Ofast` instead of `-O3`; but this requires more caution than it would seem: not only does numeric error accumulate[38], `-Ofast` improves performance by messing up the

[37]Know *and* love

[38]*i.e.*, "the death of a thousand cuts"[39]

[39]Not to be confused with "death with a thousand *cats*", which apparently is what the

Listing 10.2: Demonstration of numeric error in floating point types. The output is `4.8999999999999994671 4.9000000000000003553`.

```
#include <iostream>

int main() {
  std::cout.precision(20);
  std::cout << 7*0.7 << " " << 0.7+0.7+0.7+0.7+0.7+0.7+0.7 << std::endl;
  return 0;
}
```

mathematical results, *e.g.*, disabling `inf` and `nan` from working properly.

For this reason, the compiler is doing its duty not to reorder floating point expressions, because doing so may slightly alter the result. And so, when we deem the change acceptable (often it is), we must do this manually. This has implications that limit our ability to exploit `constexpr` floating point types as effectively as we can exploit `constexpr` integer types. For this reason, `C++` forbids floating point template arguments; permitting them would not only be a challenging engineering task, it would require a universal standard for how we approach these numerical discrepancies, and we would risk `SomeClass<7*0.7>` not matching its appropriate declaration `SomeClass<0.7+0.7+0.7+0.7+0.7+0.7+0.7>`.

Likewise, advanced mathematical equivalences are generally not discovered by the compiler; just because you provide a recurrence that numerically converges to the golden ratio[40], the compiler will not figure this out and simply replace that code with the golden ratio[41].

Because of these many reasons, we see yet another reason why $\lambda = 1.5$ as the growth constant for vectors from Chapter 6 was beneficial in yet another way: we could multiply the length by 1.5 exactly by performing `length+=(length>>1);`, thus eschewing any floating point math that would give the compiler pause.

compiler is planning for the person at the beginning of the chapter.

[40]This could occur if we're numerically computing the ratios of the n^{th} Fibonacci number to the $(n-1)^{\text{th}}$. When n is even remotely large, we will get an approximation of the golden ratio.

[41]That would be amazing.

10.4.5 Object sovereignty

One of few reasons why truly object-oriented `C++` code may be slower than `C`-style code is that modifying one object (*e.g.*, discarding a member variable that went unused in that instance or discarding a member variable that is `constexpr`) may not be permitted because doing so would only be possible if all other instances of that type could be modified. For example, if we have `struct Point { int x; int y; };`, and we declare `Point p; p.x=7;` but never initialize or use `p.y`, it is not guaranteed that `y` will disappear, because that may only be possible if *all* `Point` objects ignore `y`. If some `Point` objects use `y`, then it cannot be discarded from them, and so discarding it from `p` would result in `Point` objects with different sizes.

The autonomy of objects is a great benefit when it comes to software engineering, but for optimization, that sovereignty of an object can obscure useful optimizations. This is particularly the case for member variables initialized with `constexpr` values and never changed; as long as some objects of that type are not initialized with a `constexpr` value, the compiler may not propagate this as fully. Note that once everything is in assembly or machine code, the back-end compiler may still exploit some of these situations. But at the front end, when we are seeing the code as a more algebraic structure, the rigidity of objects is an obstacle.[42]

10.4.6 Advanced code reuse and automated design of algorithms

No compiler is yet smart enough to automatically change from one non-trivial algorithm to another. For instance, when we wrote selection sort code, the compiler did not automatically detect that it was sorting and then replace our method with the fastest known comparison-based sorting algorithm. Such a feature may sound cool in theory, but it would also be unwieldy: remember that in Chapter 3 we exploited the fact that selection sort was better for small problems.[43]

[42] Yet another reason why writing correct, fast, and readable code is really a great quest.

[43] Imagine that paperclip from Microsoft Office jumping into your beautiful code and shouting, “Hey! It looks like you’re writing a sorting algorithm… wouldn’t you like to use bubble sort?”[44]

[44] This terrifying thought experiment shows why a good programmer should control the memes of production– otherwise, those memes might end up controlling *you*…

These advanced tasks are more of an AI problem than classic compilation.

10.4.7 Memory management

Consider the buffered mergesort implementation from Chapter 3 Question 2: It performed better than the version with on-the-fly memory allocation. But the compiler was not able to automatically convert one into the other. Memory allocations and deallocations essentially modify global variables, and so they introduce a lot of complexity into the CFG (when you think of the unmentioned global variables that each n̂ew call modifies, then the order of allocations matters because they form RAW and WAW dependencies). But even if the compiler could convert this for us, there are reasons we might not want it to. One reason is that buffered implementations sometimes achieve greater efficiency[45] by allocating many things at once. This may prove faster in some cases, but it may also require a larger up-front space requirement. Trading space for time is an art, and compilers are not yet sophisticated enough for us to trust them with this[46].

10.4.8 Aliasing

One of the largest and most ubiquitous obstacles to optimization is aliasing. This is where the compiler does not know at compile time to which addresses pointers refer, and so it is forced to assume that writing to the pointer's data (*e.g.*, `*ptr=7;`) could modify nearly any non-`const` value of the same type. Aliasing is such a large concern that we devote the entirety of Chapter 11 to it.

10.5 The future

The future of optimizing compilers is bright: as more and more of our lives is compterized, code matters more and more, and so producing faster, more reliable executables from the same code is of great importance. The future of compiler optimizations lies in finding deeper equivalences between different pieces of code. Imagine compilers that can identify an algorithm at a high

[45] *E.g.*, as the result of better cache performance

[46] If the compiler produces code that it thinks is fast, but which requires 64GB of RAM and you only have 32GB, what can you do?

level and replace it with a faster version, or a compiler that can automatically visit a matrix in cache-optimized order.

Compilation is a mathematical optimization problem: we want to make the fastest code such that no differences– on the inputs we've given– are produced in the output. When seen this way, the future of compiler design closely resembles computer algebra. Such futuristic methods are sometimes referred to as "term rewriting systems".

Some compilers and languages are built for even better exploiting the `constexpr` speedups mentioned here: code in languages like `javascript` can rewrite themselves and optimize as they go. At any given instant, all of the values currently in variables are known, and so they could temporarily be seen as `constexpr`. In this manner, "just in time" (JIT) compilers continue to perform optimizations at the last second. A static compiler like the current version of `gcc` cannot see ways to speed up `x/y` if `y` is not known at compile time; however, even if `y` is chosen randomly at runtime, a JIT compiler can re-optimize after a particular value of `y` is chosen. The JIT compiler can test whether `y==2` once during runtime after `y` is chosen. In doing so it incurs a penalty that a static compiler like `gcc` does not. But in the case where `y==2`, sometimes a large amount of code might be rewritten by the JIT compiler. If that re-optimized code consumes a large amount of processing time, pausing to re-optimize after `y` is chosen may be well worth the penalty of pausing to re-optimize.

Just like hardware, compilers are always being improved, and so when we report a runtime, it is really a runtime of (1) the code with (2) the compiler on (3) our hardware. This is important to remember when you're trying unsuccessfully to recreate a speedup that you once remember achieving.[47]

Questions

1. [**Level 1**] From the perspective of performance, name one pro and one con of manually inlining short functions rather than calling them.

[47]More often than not, these "cold fusion" speedups will first be observed late at night when you're sleep deprived, and when you're trying to reproduce the speedup, it's easy to forget that you were using a different compiler. . . [48]

[48]"Tomorrow when you wake up with dreams of applied math still on your pilow, you'll wonder if it wasn't all a dream. But you'll know where you were when you get your tuition bill!"

Focus on performance and not software engineering.

2. [**Level 1**] Can the compiler find loop-invariant code when dividing a long vector of `double` types by a `double` as performed here: `for (int i=0; i<N; ++i) vec[i]/=1.7;`? Why or why not?

3. [**Level 2**] Draw the CFG for the following block of code for several `int` types:
`a=b+c; b=c+d; c=d+e; a=b+c; b=c+d; c=d+e; z=a+b+c+d+e;`
`a-=z; b-=z; c-=z; d-=z; e-=z;`
Describe opportunities to exploit parallelism.

4. [**Level 2**] Draw the CFG for the following block of code for several `int*` types:
`*a=*b+*c; *b=*c+*d; *c=*d+*e; *a=*b+*c; *b=*c+*d;`
`*c=*d+*e; *z=*a+*b+*c+*d+*e; *a-=*z; *b-=*z; *c-=*z;`
`*d-=*z; *e-=*z;`
Describe opportunities to exploit parallelism.

5. [**Level 2**] Consider, Listing 10.3, the code for Floyd-Warshall[49], an $O(n^3)$ algorithm for finding the shortest paths in a graph.

   On the section of code labeled `// 2. Compute shortest paths with dynamic programming:`, could loop reordering be performed safely? Try initializing a simple distance matrix and comparing the computed result with the original loop order against a version where the loop on variable `k` is swapped with the loop on the `i`. Are the results the same?

6. [**Level 3**] Assume that the loops in the previous question could be reordered safely. Would exchanging the loops yield better or worse performance?

7. [**Level 3**] Gauss multiplication is a scheme for multiplying two complex numbers $(a+bi)\cdot(c+di)$ via only 3 multiplications. Create a benchmark where you allocate two arrays, `x` and `y`, both of length `N=1<<20` filled

[49] "I'm in Tampa Bay, Memorial Day weekend '86. Floyd-Warshall walks into the McDonalds. The man ordered ordered 8,000 chicken nuggets. Now *that's* a champion! Now that's a champ!"

Listing 10.3: Floyd-Warshall.

```
void floyd_warshall(double*result, const double*distance_mat, const
    unsigned int n) {
  // 1. Copy in direct distances:
  for (unsigned int i=0; i<n; ++i)
    for (unsigned int j=0; j<n; ++j)
      result[i*n+j] = distance_mat[i*n+j];

  // 2. Compute shortest paths with dynamic programming:
  for (unsigned int k=0; k<n; ++k)
    for (unsigned int i=0; i<n; ++i)
      for (unsigned int j=0; j<n; ++j)
        // Allow the path to pass through vertex k if it is a shortcut
        // from i to j:

        // result[i,j] = min(result[i,j], result[i,k] + result[k,j])
        result[i*n+j] = std::min(result[i*n+j], result[i*n+k] +
            result[k*n+j]);
}
```

with complex numbers, and where you multiply each element-wise so that a new array `z[i]=x[i]*y[i]`. Write your own struct for complex numbers, where the real and imaginary parts are `double` types. Implement a version with and a version without without Gauss multiplication. Does Gauss multiplication help? Would the compiler be able to figure out Gauss multiplication on its own? Why or why not? Compare or contrast to the case where the expression `x<<(y-z)` vs. `x<<y>>z` is computed.

Chapter 11

Aliasing and `restrict` Pointers

11.1 Aliasing[1]: a confusing plot twist

There is an old jazz song that goes, "Everyone loves my baby, but my baby just loves me..." Now, to the untrained ear, this might sound like a perfectly ordinary expression of affection: the singer is celebrating the fidelity of their relationship, which is even more impressive considering the desirability of their partner. But if we delve deeper, there is more going on here. If *everyone* loves [my] baby, then [my] baby *also* loves [my] baby. Which is fine and well, but when you also combine this knowledge with the fact that [my] baby just loves [me], then the only logical conclusion is that the singer is their own baby.[3]

Because our prior biases are against this sort of coincidence, these make for good twists in mystery novels and films. In computer science, this is called "aliasing": when two different variables actually refer to the same memory underneath. Unlike in books and movies, where these coincidences are celebrated, in programming aliasing can be hazardous and inefficient.

[1]In signal processing, "aliasing" refers to the phenomenon where sampling a signal below the Nyquist frequency may add artifacts into the sampled result, making it impossible to guarantee that the original signal can be perfectly recovered from the samples. Although this is distinct from our use of aliasing here, they both share a common root: multiple data looking or being the same. In UNIX, `alias` is a command that lets us rename a command, and in spycraft, an alias is another name for the same person.[2]

[2]Are you getting confused by the fact that aliasing can refer to multiple different things? Well, buckle up: this chapter will be full of that kind of stuff.

[3]In GDR, *baby* is *you*![4]

[4]But in all seriousness, in East Berlin, exception throws you.

When you declare two integers `int x,y;`, then the compiler allocates space on the stack for each[5]; `x` is distinct from `y`, and so aliasing between them is simply impossible. After all, `x` and `y` store their data in distinct addresses, locations in memory that are known at compile time.

However, when we use pointers, aliasing is a frequent concern. For instance, when we are making a deep copy of an object that uses dynamically allocated memory, *e.g.*, writing the `operator =` for our own vector class. Usually, when we run `vec_a=vec_b`, the steps are as follows: First, free the memory allocated by `vec_a`. Second, allocate a new vector for `vec_a` with enough space for all elements in `vec_b`. Third, copy every element from `vec_b` into `vec_a`. These steps work fine when `vec_a` and `vec_b` refer to distinct objects. But if they were the same object, then there would be a large problem: the first step, which frees the memory allocated to `vec_a` would simultaneously free `vec_b`, and in doing so, all of the data would be irretrievably lost. Of course, when the objects are allocated on the stack and the arrays that they store are non-overlapping, then this potential pitfall will not occur. But what if we have pointers to vectors? When executing `*vec_ptr_a=*vec_ptr_b`, it may not be possible to know at compile time whether `vec_ptr_a` and `vec_ptr_b` refer to the same vector object. For this reason, it is common to put a check in the `operator =` of classes that use dynamic memory: if the two objects are the same, then the assignment operation is abandoned[6].

A large effect of aliasing that we saw in Chapter 10 is that it often prevents successful compiler optimizations. Consider a function that accepts two pointer arguments `int*x,int*y`. If the function executes `*x=*y; ++*y;`, these operations will not safely run in parallel. This is because of the case where the two pointers refer to the same memory, *i.e.*, where `x=y`. In that case, modifying `*x` also modifies `*y`, and so the first line must terminate before `++*y` can safely be executed.[7]

[5]Or, possibly, a dedicated register for each if there are enough registers available to never need to store them on the stack.

[6]After all, if they are the same object, then there is no point copying anything over.

[7]This is onathem *Fight Club* situations where you later learn `*a` and `*b` referred to the same address in RAM all along.[8]

[8]Donald Kaufman: "I'm putting in a chase sequence. So the killer flees on horseback with the girl, the cop's after them on a motorcycle and it's like a battle between motors and horses. . . like technology vs. horse." Charlie Kaufman: "And they're still all one person, right?" . . . Charlie Kaufman: "How could you have somebody held prisoner in a basement and. . . and working at a police station at the same time?" Donald Kaufman:

11.2 `C` vs. `Fortran`

Aliasing is commonly referred to as the single reason why `Fortran`, which has no pointers, was favored over `C` vector and matrix operations. Because it has no pointers, `Fortran` arrays cannot be aliased. On the downside, this means that it is impossible to point to a "slice"[9] of an array or a matrix. Because of the influence on compiler optimizations, `Fortran`'s more limited vocabulary for data structures allowed compilers to optimize better: aliasing never need be considered; however, this was also a reason that `C` became so popular: the ability to use pointers made it much easier for implementing complex data structures.[10] With the `restrict` keyword, it should always be possible to write `C` code that is as fast as a good `Fortran` implementation.

11.3 Aliasing of arrays

It may seem like the compiler could just check for aliasing. But this would require comparing all pointer pairs at runtime[11], and the address referred to by some of these might not be known at compile time, because of the stochastic nature of `malloc` and `new` operations. But aliasing is not restricted to the case where the pointers themselves have the same addresses; aliasing can also occur when two arrays share some addresses. For example, consider the following code: `int*xy=new int[128]; int*x=xy; int*y=xy+64; int*mix=x+32;`. The code will use three arrays, `x`, `y`, and `mix`, each of which we will treat as being 64 el-

[pause] "Trick photography."

[9]A slice of a matrix can be implemented by storing the pointer to the start of the memory used, the new number of rows, the new number of columns, and the old number of columns (so that we can jump 1 row ahead in the matrix). Slices are useful because they don't need to make local copies, and instead look at a "window" of the existing memory. This can be better than copying for preserving cache locality, but slices also introduce more aliasing concerns.

[10]In `Fortran`, it would be possible to make an ersatz pointer into an array by simply storing an integer index that you'd like to refer to in the array; however, it is not possible to specify *which* array is being indexed (doing so would require a pointer to an array), and so we're back where we started. For this reason, implementing a linked list would be simpler in `C` or `C++` than in `Fortran`, and implementing a fancy data structure like a Fibonacci heap would likely be quite frustrating in `Fortran`.

[11]Compiler: "WHAT TIMELINE IS THIS?!"

ements in length[12]. If we write to `x[32]`, it will also modify `mix[0]`. So if we pass these three arrays as pointers to some function `void modify_arrays_64(int*a, int*b, int*c);`, which modifies its three parameters, then that function must be cautious and assume that its three parameters may use overlapping memory addresses.

Because of this, essentially `any` function that modifies pointer arguments must be compiled as valid in the case where aliasing among those pointer arguments occurs. This may sound trivial, but it can have serious implications. Consider the CFG for the following statements: `int k=rand()%64; a[k]=8; b[0]=a[7]; c[64-k]=a[k];`. Without considering aliasing, the third statement `c[64-k]=a[k]` could use the fact that we've just set `a[k]=8`, and so the third statement could be simplified to `c[64-k]=8`. But we are unsure of whether `a[k]` and `b[0]` refer to the same memory, and so `a[k]` *must* be re-read in case it was changed by the second line. Aliasing can create a significant hindrance on compiler optimization: values that could be stored in a register and reused sometimes must be fetched from RAM[13].

Listing 11.1 demonstrates a routine that uses two arrays to modify a third array. This code runs in 0.01443 seconds. The reason the performance isn't better is because the compiler is unsure whether `source`, `dest`, and `eight` refer to the same memory, and so some values may need to be re-read from RAM (or at least not saved in registers in an elegant manner). It is interesting to investigate whether specifying `source` and `eight` as `const`, meaning their values cannot be changed by the function, will improve performance (Listing 11.2). Unfortunately, this `const` version has essentially the same average runtime: 0.01443 seconds.

Listing 11.1: Operations on arrays with potential aliasing.

```
#include "../Clock.hpp"
#include <iostream>
```

[12]Note that this length of 64 is not specified anywhere in our code; we could very well treat `x`, `y`, and `mix` as arrays 32 elements in length, and even though we wouldn't be using memory as efficiently as possible, it would nonetheless be valid. Remember that these pointers are simply memory addresses.

[13]As always, when we request an address from RAM, we may get the data from the cache, but this is generally unknown, and so we should be prepared that it will come from RAM. Unless we've *really* tuned our code for a particular chip, it's better to acknowledge that the cache vs. RAM distinction is not really in our hands.

```
// If this function is inlined, the compiler may be smart enough to
// tell that the memory footprints of dest, source, and eight do not
// overlap. In a real use-case (rather than a simple benchmark like
// this one), the compiler will rarely be able to tell whether the
// memory is really non-overlapping.
__attribute__ ((noinline))
void apply(float* dest, float* source, float* eight, const unsigned long
    n) {
  // n/8:
  for (unsigned long i=0; i<(n>>3); ++i) {
    for (unsigned long j=0; j<8; ++j)
      dest[(i>>3)+j] = source[(i>>3)+j]*eight[j];
  }
}

int main() {
  const unsigned long N = 100000;
  const unsigned long REPS = 256;

  Clock c;

  float *x = new float[N];
  for (unsigned long i=0; i<N; ++i)
    x[i] = i;
  float *y = new float[N];
  for (unsigned long i=0; i<N; ++i)
    y[i] = x[i];
  float z[] = {7,1,4,5,2,3,6,8};

  c.tick();
  for (unsigned int r=0; r<REPS; ++r) {
    apply(y, x, z, N);
  }
  c.ptock();

  return 0;
}
```

Listing 11.2: Operations on arrays with potential aliasing. Unlike Listing 11.1, here `source` and `eight` are declared `const`.

```
#include "../Clock.hpp"
#include <iostream>

// If this function is inlined, the compiler may be smart enough to
// tell that the memory footprints of dest, source, and eight do not
// overlap. In a real use-case (rather than a simple benchmark like
// this one), the compiler will rarely be able to tell whether the
// memory is really non-overlapping.
__attribute__ ((noinline))
void apply(float* dest, const float* source, const float* eight, const
    unsigned long n) {
  // n/8:
  for (unsigned long i=0; i<(n>>3); ++i) {
    for (unsigned long j=0; j<8; ++j)
      dest[(i>>3)+j] = source[(i>>3)+j]*eight[j];
  }
}

int main() {
  const unsigned long N = 100000;
  const unsigned long REPS = 256;

  Clock c;

  float *x = new float[N];
  for (unsigned long i=0; i<N; ++i)
    x[i] = i;
  float *y = new float[N];
  for (unsigned long i=0; i<N; ++i)
    y[i] = x[i];
  float z[] = {7,1,4,5,2,3,6,8};

  c.tick();
  for (unsigned int r=0; r<REPS; ++r) {
    apply(y, x, z, N);
  }
  c.ptock();

  return 0;
}
```

11.4 The restrict keyword

Because aliasing is such an issue for performance, the C language offers a slightly obscure keyword to help preserve performance when using pointers: restrict. Declaring a pointer as restrict means that we are guaranteeing the compiler that aliasing with this address (or array) does not need to be considered. We are guaranteeing that the block of memory to which this pointer refers will be modified only through the pointer itself. The `restrict` keyword is used in `C` in this manner: `int*restrict x=malloc(10*sizeof(int))`. Listing 11.3 rewrites Listing 11.1 so that `restrict` pointer parameters are used. Just by adding the `restrict` keyword[14], the performance improves substantially to 0.006501 seconds.

Listing 11.3: Operations on arrays where aliasing is forbidden by `restrict`. Unlike Listing 11.1, here `source` and `eight` are declared `restrict`, explicitly forbidding aliasing.

```
#include "../Clock.hpp"
#include <iostream>

// If this function is inlined, the compiler may be smart enough to
// either 1) eliminate everything as dead code or 2) tell that the
// memory footprints of dest, source, and eight do not overlap (note:
// in a real use-case (rather than a simple benchmark like this one),
// the compiler will rarely be able to tell whether the memory is
// really non-overlapping). Also, depending on how the compiler works,
// if the variables for dest and source are not declared as restrict
// in the function that calls apply, the compiler inlining this code
// could even make the code slower.
__attribute__ ((noinline))
void apply(float*__restrict dest, float*__restrict source,
    float*__restrict eight, const unsigned long n) {
  // n/8:
  for (unsigned long i=0; i<(n>>3); ++i) {
    for (unsigned long j=0; j<8; ++j)
      dest[(i>>3)+j] = source[(i>>3)+j]*eight[j];
  }
}

int main() {
  const unsigned long N = 100000;
  const unsigned long REPS = 256;
```

[14]This makes it sound slightly easier than it is in general: in general, we can only apply the restrict keyword when we are sure the pointers refer to distinct memory, and this is not always possible.[15]

[15]If it were always possible, then the creators of the `C` language would simply have made *every* pointer `restrict`, *i.e.*, it would mean that `C` were like `Fortran` in that aliasing would not be possible.

```
  Clock c;

  float * x = new float[N];
  for (unsigned long i=0; i<N; ++i)
    x[i] = i;
  float * y = new float[N];
  for (unsigned long i=0; i<N; ++i)
    y[i] = x[i];
  float z[] = {7,1,4,5,2,3,6,8};

  c.tick();
  for (unsigned int r=0; r<REPS; ++r) {
    apply(y, x, z, N);
  }
  c.ptock();

  return 0;
}
```

`C++` is not a proper superset of the `C` language, and unfortunately does not include `restrict` in the standard; however, `restrict` is so useful, that many `C++` compilers still support it, and it is generally regarded as a kind of pseudo-standard in `C++`. Both `gcc` and `clang` offer a `__restrict` keyword instead.

Aliasing can be more costly than it might seem: it can make loop-invariant code (mentioned in Chapter 10) look as if it might not be loop invariant. To a person, a loop such as `for (int k=0; k<N; ++k) x[k]+=y[7]*z[7];` may seem to feature the loop-invariant expression `y[7]*z[7]`, which could be computed once before the loop and then used in ever iteration[16]. But when we consider aliasing, the compiler is correct to be cautious and not assume that `y[7]` and `z[7]` are constant during the loop; the may be aliased to some indices of `x`, and so may unexpected change during the loop.[17]

Although it's slightly obscure, `restrict` is of utmost import in high-performance applications like graphics and scientific computing; proper use can improve performance, even when the code is written in an optimized `C`-style that does not allocate or deallocate objects on the fly. This is why we

[16]In this manner, the loop now computes `N` additions instead of `N` additions and `N` multiplies.

[17]Human: "Compiler, it's OK, this block of data is probably *my* baby." Compiler: "Maybe I know more than you think I do, Mark..."

declared the pointer in our in-house vector class in Chapter 6 to be `restrict`, and may help us improve performance over `std::vector`; however, there is a caveat: if we start declaring pointers such as `int*part=&vec[i]`, and then modifying them together with our `restrict`-using vector, *e.g.* `for (int k=0; k<N; ++k) vec[k]+=part[k];`, then we break our promise to the compiler that we've guaranteed there will be no aliasing of `vec`. This error would likely be difficult to find, because it will not be known during compile time. It will likely not even crash our program[18]; instead, our program will just give numerically wrong results. So even though `restrict` is quite useful for performance, do not use it haphazardly.[19]

Questions

1. [**Level 1**] Rewrite Listings 11.1 and 11.3 to build aliased array arguments and pass them into the `apply` functions (note that this will be invalid in the case of Listing 11.3). Compare the results element by element. Do they match? Repeat this until you get a case where invalid use of the `restrict` keyword in your modified version of Listing 11.3 produces an incorrect result.

2. [**Level 2**] Listing 11.2 demonstrates that aliasing can still affect `const` pointer types. Can aliasing affect `constexpr` types? Create an example to justify your answer.

3. [**Level 2**] A function accepts an argument `int*arr`. Inside the function, a local variable `int t=7;` is declared, and then subsequently, `arr[k]=3;` is performed for some integer `k` (the value of `k` is unknown at compile time). After that line runs, is `t==7`? What would need to happen so that `t==3`?

4. [**Level 3**] Adapt your matrix multiplication code (both naive and transposed) from Chapter 9 Question 5 to use `restrict` matrix parameters. Does the assembly code change at all? Is there an effect on performance?

[18]As it might if there were a memory error

[19]I wouldn't give this aliased `restrict` code to my worst enemy. . . just in case my worst enemy happened to be *me*.

Chapter 12

Metaprogramming and Template Recursion

12.1 The evil of recursion

Recursive functions often offer a simple means with which we can turn ideas into code, such as the simple recursive factorial implementation from Chapter 10 or the simple recursive Fibonacci implementation in Listing 12.1; however, recursive functions have significant downsides: They are constantly pushing and popping arguments and return values on and off of the stack. This pushing and popping may not only invade larger swaths of memory (poor for cache performance), it also prevents functions from computing in registers only (and avoiding RAM altogether). This is because all functions, even those taking no parameters and offering no return value[1] will nonetheless need to push its "instruction pointer"[2].

[1]This is, of course, an illustration. Without modifying global variables, it's quite difficult to discern what such a function would even be used for.

[2]The current address of code being executed before the function is called. It is necessary to store the instruction pointer before jumping to another region of the code to make a function call, because when that function call is finished, we need to remember the next line of machine code that was going to run.

Listing 12.1: A simple recursive Fibonacci implementation.

```
unsigned long fibonacci(unsigned i) {
  if (i <= 1)
    // i=0 and i=1 return 1:
    return 1ul;

  // Since i>1, using (unsigned int)(i-1) and (unsigned int)(i-2) is
      safely going to decrease:
  return fibonacci(i-1) + fibonacci(i-2);
}
```

12.2 The challenge of iterative implementations

Converting an elegant recursive function[3] to an iterative, loop-driven implementation can be quite tricky. Furthermore, doing so may require pointers (which may result in aliasing) or may introduce more complicated control flow[4], which make compiler optimizations more difficult. For this reason, we may find ourselves stuck between two extremes: on one hand, we have the overhead and lack of optimizability of using recursion and on the other hand we have the challenge of overcoming the obstacles inherent to an iterative implementation.

12.3 Integer templates

A clever approach to this problem is template recursion[5]. In `C++`, templates allow us to use generics: If we implement a class to store a vector of `int` types, then creating a vector of `double` types should have a lot of reusable code. `C++` enables this reusable code through templates, meaning we write

[3]Such as fast Fourier transform (FFT), Strassen matrix multiplication, Karatsuba multiplication, quicksort, *etc.*

[4]*E.g.*, more loops and if statements

[5]"How did this tradition first get started? I'll tell you…I don't know." But I will say this: Good artists borrow, great artists steal. Do you know who said that? It was me.[6]

[6]And if you doubt that, then consider: was I the first one to say " 'Good artists borrow, great artists steal.' Do you know who said that? It was actually me." If we continue inductively, at some point it *must* become true…Wait, is that right?

the class for a compile-time argument `typename T`, and then our vector will work with any type `T`.[7] In the implementation of templates, the compiler essentially performs a find-replace for each unique `T` used with that template class. This enables us to write the template class one time, but then to reuse it with various `T`. Because of the find-replace semantics used for template arguments, all template arguments must be `constexpr`.

There is an obscure feature of `C++`: we can also accept integer-like[8] template arguments. Thus, instead of declaring a template class as `template <typename T> class C { ...};`, we could declare it as `template <unsigned I> class C { ...};`. Where the former template class will compile to a class for each unique type `T` used, the latter will compile to a class for each unique `unsigned int` used. This can be quite useful when implementing non-trivial[10] but constant-sized data types. For example, `std::bitset` takes an integer template argument (the number of bits needed). Likewise, if we were implementing a 1024-bit integer and 4096-bit integer classes, we could do both simultaneously by creating a template class with an integer template argument.

Of course, many applications where we use integer template arguments could be performed using `C`-style integer arguments; however, as we learned in Chapter 10, `constexpr` types can result in many beneficial optimizations. By compiling to a different class for each unique integer template argument, each of these classes can be optimized completely around its particular arguments. For instance, a loop `for (int i=0; i<N; ++i) f(i);` with a `constexpr int N` can be peeled and unrolled perfectly at compile time (without any ambiguity as to whether `N` is evenly divisible by the amount by which we unroll the loop, *etc.*). When we specify multiple classes with different template `N` arguments (*e.g.*, the 1024-bit and 4096-bit integers mentioned above), each of these cases will be optimized for their particular choice of `N`. This can yield

[7]Beware: some of the vector classes that we implemented in Chapter 6 use `malloc`, `realloc`, and `free`. They are still compatible with templating, but they are only valid with primitive types of `T`, *e.g.*, `int`, `double`, `char`, ..., because `malloc` and `free` do not call constructors and destructors (respectively).

[8]*I.e.*, `int`, `short`, `long`, `char`, `unsigned`, `bool`, ...[9]

[9]You may wonder, "Why don't we just allow arbitrary template arguments? Why constrain ourselves to the integers?" Well, consider the inexact mathematics of floating point arithmetic mentioned in Chapter 10: these numeric inconsistencies, however slight, could result in `C<7*0.7>` not to match `C<0.7+0.7+0.7+0.7+0.7+0.7+0.7>`, as mentioned briefly in Chapter 10.

[10]*i.e.*, not small enough to fit inside a primitive

a substantial speedup.

12.4 Recursing using template arguments

Template arguments can also be included for function definitions, such as with `template <bool PRINT_SORTED> void f() {...}`. Given this fact, it feels that we would do better to use template functions to somehow help us implement recursive functions. But there is a problem with this. Consider our attempt at implementing a template-recursive factorial function, *i.e.*, a recursive factorial function where we recurse using the template arguments (Listing 12.2). Essentially, the problem is the definition of the recursive base case: templates do not use runtime information (`if` statements are only called at runtime), and so what we think of as the base case, `factorial<0>` makes mention of `factorial<-1>`. Since we've defined the template argument to be `unsigned`, then `-1` will mean `-1u = 65535`. Without calling the `if` statement at compile time, the compiler does not see that `factorial<-1>` is not needed. The result is not precisely inifinite recursion (because our finite precision loops back to the beginning eventually), but it is so large that it might be seen as infinite.[11]

The solution to this problem is to embed our recursive functions into template classes instead of template functions. Unlike template functions, template classes permit specialization, the process by which we can specify a custom implementation for some template arguments. This specialization can be made for cases like `std::vector<bool>`, which uses a bit-packed implementation, but classes can also be specialized for particular integer template arguments, such as a base case `Factorial<0>`. By doing so, we can implement a working template-recursive factorial (Listing 12.3). It is certainly more verbose than the standard recursive version (we must invoke our wrapper class each time we want to call the function)[13], but it has an added

[11]Infinite recursion is like babies having babies having babies[11], which can be a pretty hard life (especially in this economy). Hey, have you seen *Dark Water* (the Japanese version, of course[12])?

[12]With the exception of *The Ring*, the Japanese originals [flicks bangs out of eyes], we can agree, are generally better than the Hollywood remakes.

[13]I bet you're thinking, "Come on, this is a pain, no one's ever going to want to use this!" Well, beware, that's also what they said to Mr. Za: They said, "Listen, Pete, nobody cares about your garlic-tomato-cheese on bread. Spaghetti's what's in right now, it's what's cool, it's what's hip, and it's gonna stay hip. Please show Mr. Za the door,

Listing 12.2: A broken attempt to perform recursion using function template arguments. This recursive factorial function produces an error saying that the maximum template recursion depth has been exceeded. This is because the function `factorial<I>` requires a new function `factorial<I-1>`, which inductively continues until every possible `unsigned int` has been exhausted: Subtracting eventually produces `-1u`, which is the largest `unsigned int` possible. The compiler understandably balks at us asking to create 65536 separate functions, and will only do so if we force it to by compiling with the flag `-ftemplate-depth=N` for some large value `N>65536`.

```
template <unsigned I>
unsigned long factorial() {
  if (I <= 0)
    return 1ul;
  // Since I>0, using (unsigned int)(I-1) is safely going to decrease:
  return I * factorial<I-1>();
}
```

benefit: all of the recursions are unraveled at compile time, and because our function simply returns operations of the template integer argument (which is necessarily `constexpr`), we end up with a runtime of 0s; all of the multiplications are performed at compile time. This is not the same as the previous cases where we found a runtime of 0s due to unwanted dead code elimination: in this case, we correctly print out `factorial(12)`, but with no actual recursive function calls.

12.5 Template-recursive Fibonacci

We can implement a template-recursive Fibonacci benchmark (Listing 12.4) in a manner reminiscent of our template-recursive factorial (Listing 12.5). Where the simple recursive version runs in 0.01642s, the template-recursive version produces the same result in 0s. Template-recursive calls are unraveled at compile time, and each of those results is computed as a `constexpr` at

won't you?" And God knows how wrong they were...

Listing 12.3: A working template recursive factorial. A template specialization is used for the recursive base case.

```
#include "../Clock.hpp"
#include <iostream>

template <unsigned i>
class Factorial {
public:
  static unsigned long evaluate() {
    return i*Factorial<i-1>::evaluate();
  }
};

template <>
class Factorial<0u> {
public:
  static unsigned long evaluate() {
    return 1ul;
  }
};

int main() {
  const unsigned long REPS = 12800000;

  unsigned long res;
  Clock c;
  for (unsigned r=0; r<REPS; ++r) {
    res=Factorial<12>::evaluate();
  }
  c.ptock();
  std::cout << res << std::endl;

  return 0;
}
```

Listing 12.4: A benchmark working with a recursive Fibonacci implementation.

```
#include "../Clock.hpp"
#include <iostream>

unsigned long fibonacci(unsigned i) {
  if (i <= 1)
    return 1;
  return fibonacci(i-1) + fibonacci(i-2);
}

int main() {
  const unsigned long REPS = 1<<10;

  srand(0);

  Clock c;

  unsigned long tot=0;
  for (unsigned long r=0; r<REPS; ++r)
    tot += fibonacci(20);

  c.ptock();
  std::cout << tot << std::endl;

  return 0;
}
```

compile time (possible because the results are integer types[14]).

Of course, the same procedure can be used in a more general case: We need not constrain ourselves to simple recursive functions like factorial or Fibonacci. We can easily apply the same techniques to a whole host of

[14]If we directly enable `-ffast-math` or use `-Ofast` instead of `-O3`, then the compiler has the option of performing more extensive `constexpr` merging of `float` and `double` types at compile time, but at the expense of possibly losing precision as described in Chapter 10[15].

[15]Sometimes this is not as bad as it may sound: both `0.7*N` and `0.7+0.7+`$\cdots$`+0.7` may be slightly imprecise, and so if we are not intentionally choosing one of these, then it is not so harmful to let the compiler make the arbitrary decision in a way that benefits the runtime.

Listing 12.5: A template-recursive Fibonacci equivalent to Listing 12.4.

```
#include "../Clock.hpp"
#include <iostream>

template <unsigned i>
class Fibonacci {
public:
  static unsigned long evaluate() {
    return Fibonacci<i-1>::evaluate() + Fibonacci<i-2>::evaluate();
  }
};

template <>
class Fibonacci<1> {
public:
  static unsigned long evaluate() {
    return 1u;
  }
};

template <>
class Fibonacci<0> {
public:
  static unsigned long evaluate() {
    return 1u;
  }
};

int main() {
  const unsigned long REPS = 1<<10;

  Clock c;

  unsigned long tot=0;
  for (unsigned r=0; r<REPS; ++r) {
    tot+=Fibonacci<20>::evaluate();
  }

  c.ptock();
  std::cout << tot << std::endl;

  return 0;
}
```

recursive functions, including those that modify arrays (*e.g.*, merge sort). This opens up vast possibilities for divide and conquer algorithms[16]. Where simple recursive implementations generally produce a smaller executable[17], template-recursive versions can optimize each recursive call separately: a template recursive implementation of Karatsuba's integer multiplication algorithm will produce distinct pieces of code depending on whether two 128-bit numbers are being multiplied or whether two 256-bit numbers are being multiplied. For many optimizations, including loop unrolling and SIMD vectorization, the best approach will depend heavily on the problem size. Having distinct pieces of code for each integer argument means that the compiler can optimize each separately, in a manner customized for its size.

When unraveling recursive calls to separate functions, a common concern is producing very large executables[18]; however, in many cases, this concern is not as significant as we might first believe. Many divide and conquer algorithms divide in half, and so the number of functions produced by the compiler is logarithmic in the problem size[19].

12.6 Using non-`constexpr` arguments

Another common concern is how we go about applying our template-recursive methods when we do not know the size of the problem at compile time. First of all, for a divide and conquer algorithm like merge sort, which halves the problem size in each recursive call, the number of function calls will grow logarithmically as mentioned above. Computing an upper bound on reasonable problem sizes will therefore produce only a moderate constant number of functions needed. For instance, ensuring that we can sort lists of size $n < 2^{32}$, we would only need to produce 32 separate functions.

Nonetheless, note that some problems, such as sorting or matrix multiplication, are not constrained to powers of 2, which will make implementation more challenging. One approach to take is to embed a problem in another

[16]Myrnyy's template-recursive FFT implementation, GFFT, is a seminal work in this arena.

[17]After all, they only produce one function definition.

[18]As we saw in Chapter 10, this can decrease performance by burdening the instruction cache with too much code.

[19]*E.g.*, using a template integer argument of `1048576` will produce only 20 unique functions.[20]

[20]"Rumors of my recursive depth have been greatly exaggerated."

problem, which is the size of the next power of 2 that would contain it. In this manner, sorting a list of 61 values would be treated as sorting a list of length 64, enabling us to implement only a logarithmic number of functions even when we consider inputs that are not powers of 2.

Calling the appropriate one of these functions at runtime is tricky: if we have a non-`constexpr` value `int n`, we cannot execute `Fibonacci<n>::evaluate()`. Thus, on the pre-computed possible values (as listed above), we need to map the runtime value `int n` to the appropriate `constexpr` value. There are two principal ways of accomplishing this: The first uses inheritance. All classes for a particular problem, regardless of the template argument, will inherit from a common base class, and the function `evaluate` will be made `virtual` instead of `static`. In this manner, we can construct an array `Base**arr`, where each element is a `Base*`. Index 0 will point to a `Derived<0>` type, index 1 will point to a `Derived<1>` type, *etc.* This strategy has a quality that we can map a runtime value `int n` to its corresponding template class `Derived<N>` (where `N=n`) at runtime; but it also has a disadvantage: using virtual functions occludes the compiler's ability to determine which function is being called until runtime, meaning the template-recursive calls may be less efficient[21] Therefore, an alternate approach is to simply try different `constexpr` values until the one matching `N` is found. For example, if `int n=1024`, meaning we're calling merge sort on a list of length 1024, then we could test `n==1`, then if that was false, test `n==2`, then if that was false, test `n==4`, and so on. In this manner, we will only need to execute $\log(n)$ branch statements before we can match `int n` with an equivalent `constexpr int N`, and then use `N` as the template argument. Although this takes $O(log(n))$ steps instead of $O(1)$ steps (as the array approach above would), it uses no virtual functions and so the compiler can optimize the code much more effectively.

12.7 Metaprogramming

The above problem, where we want to test `if (n==1)` and then `else if (n==2)` and so on, is itself a problem that we would rather solve with code instead of manually writing out each `if` statement. This is possible with "metaprogramming", the art of creating code that creates other code. This itself could be accomplished with template recursion: in each

[21]For example, they may not be inlined as effectively.

template class with template parameter `N`, we would perform a comparison, and if that comparison succeeded (*i.e.*, `if (n==N)`, then we would call `Function<N>::evaluate()`, knowing that it was the valid equivalent to `Function<n>::evaluate()`.[22]

Rather than write the `C++` code ourselves, we write *meta* code (*e.g.*, via further template recursion) that writes highly optimized `C++` code for us. When paired with `C`-style high-performance implemenations[23], template metaprogramming can produce some of the fastest code.

One example of this is forced loop unrolling. Given the iterative code in Listing 12.6, we can force the compiler to unroll it by using template recursion (Listing 12.7). Where the iterative version from Listing 12.6 runs in 0.000067s, the version with the fully unrolled loop from Listing 12.7 runs in 0s and produces an identical result. This is possible because, once the loop is fully unrolled, the program consists of a composition of `constexpr` integer values, which can all be evaluated at compile time[24]. This faster runtime performance comes at a cost: where the iterative version took 0.217s to compile, forced loop unrolling took 11.309s to compile[25]. In this manner, we exchange a longer compile time for a shorter runtime (just as we do when we enable compiler optimizations).

Template-recursive code is highly dependent on the compiler to produce good optimizations after the code has been unraveled. For this reason, it can be useful to test it with multiple compilers (*e.g.*, `gcc`, `clang`, and `icc`).

Questions

1. [**Level 1**] How can template recursion be used on problems whose size is unknown at compile time? What conditions are necessary?

Projects

1. [**Level 2**] Create the fastest template-recursive implementation of

[22]Because `n` is not `constexpr`, it is not a valid template argument.

[23]*e.g.*, using `double*__restrict` instead of `std::vector<double>`

[24]This can be verified by looking into the assembly code (when compiling with `-O3` as usual): there is no loop and the final result, 1465881288704, appears as a magic number there.

[25]"Boy. . . That escalated quickly."

Listing 12.6: A simple iterative program.

```
#include "../Clock.hpp"

int main() {
  const unsigned int N=1<<14;

  Clock c;

  unsigned long tot=0;
  for (int i=0; i<N; ++i)
    tot += i*i;

  c.ptock();
  std::cout << tot << std::endl;

  return 0;
}
```

Strassen matrix multiplication you can. What performance benefit do you observe compared to a version that uses standard recursion?

Listing 12.7: Using template metaprogramming to force full loop unrolling. This program must be compiled with `-ftemplate-depth=` flag larger than the size of the loop; `-ftemplate-depth=17000` suffices.

```
#include "../Clock.hpp"

template <unsigned int I, unsigned int N>
class SumSquaresScaled {
public:
  __attribute__ ((always_inline))
  static constexpr unsigned long value() {
    return I*I + SumSquaresScaled<I-1,N>::value();
  }
};

template <unsigned int N>
class SumSquaresScaled<0,N> {
public:
  static constexpr unsigned long value() {
    return 0;
  }
};

int main() {
  const unsigned int N=1<<14;

  Clock c;

  unsigned long tot=SumSquaresScaled<N-1,N>::value();

  c.ptock();
  std::cout << tot << std::endl;

  return 0;
}
```

Chapter 13

Multithreading with OpenMP

13.1 Multithreading

Multithreading is a standard way to improve performance, and it has continued to grow in importance as CPUs continue to feature more and more cores.[1]

Multhreading allows multiple independent processes to run simultaneously[2]. Multithreading has many hazards, such as the possibility of "race conditions", where two threads are simultaneously reading from and writing to the same data. In general, it is difficult to share non-`const` data effectively between multiple threads, and so this is generally verboten. In short, writing multithreaded code is an art, and it is generally difficult to do very well.

[1]There are a few reasons why it's become popular to feature many cores instead of a higher clockspeed. Two key reasons are the fact that the power consumption grows rapidly with the clockspeed and the fact that even the speed of light takes some time to propagate across the chip, and so if the clockspeed is too high, there will not be sufficient time for the clock signal to fully propagate across the chip before the next clock signal begins (which can cause some circuits to no longer be synchronized). Multicore chips also easily achieve large speedups on some highly important, absurdly parallelizable problems, such as numeric linear algebra or testing primality. This is the basis of high-performance GPU computing.

[2]Often, each core will have a dedicated L1 and L2 cache, but L3 caches are sometimes shared. This can add challenges as described briefly in Chapter 9.

13.2 Basic OpenMP: parallelizing `for` loops

In practice, multithreading can be performed by using calls specific to your operating system. Because this severely limits the portability of code, multiple operating systems, including Windows, Linux, and UNIX distributions, are "POSIX" compliant, meaning that among other things, they have shared functionality for handling multithreading. When we `#include <pthread.h>`, we have access to a multithreading specification that works across multiple operating systems. Unfortunately, the API with which we interface with the `pthread` library still sometimes *does* depend on the operating system.

OpenMP offers a much simpler means by which we can interface with multithreading libraries[3]. OpenMP consists of different `#pragma` commands and functions, which can be used to decorate our code, and which will be seen by the compiler to automatically build multithreaded code when we add the compilation flag `-fopenmp`. OpenMP allows us to more easily write multithreaded code, regardless of our operating system; however, as we will see, it is not completely trivial to accelerate our code with OpenMP, even on parallelizable problems.

Listing 13.1 shows a first example, where we want to perform `x[i]+=7` on every element in an array `x`. On one hand, this code is trivially parallelizable, and so it feels like a good candidate to speed up with multithreading. By simply adding `#pragma omp parallel for` above the `for` loop over `i` (see Listing 13.2) and adding `-fopenmp` to our `g++` compilation flags, we can use multiple cores[4].

Unfortunately, Listing 13.1 runs in 0.0005921, while the OpenMP adapted code from Listing 13.2 runs in 0.002180s. The reason for this is the overhead for multithreading: the cost of starting a new thread is not large, but compared to a single addition, that overhead *can* be large. Here, we should be wondering precisely how many threads were automatically started by OpenMP and whether this number of threads was large enough to incur more harm than good (which it clearly did). OpenMP *can* be used more effectively on this problem, but we would need to seize more control.

However, on an alternate problem where the cost per iteration is more

[3]But OpenMP may not be as efficient as hand-made code directly made for your operating system.

[4]This can be verified by opening your computer's System Monitor program and observing multiple cores engaged while the program runs.

Listing 13.1: A parallelizable loop with a low cost per iteration.

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  const unsigned long N = 600000;

  double * x = new double[N];
  for (unsigned long i=0; i<N; ++i)
    x[i] = i;

  Clock c;
  for (unsigned long i=0; i<N; ++i)
    x[i] += 7;
  c.ptock();

  // To prevent dead code elimination:
  std::cout << x[0] << std::endl;

  return 0;
}
```

Listing 13.2: The loop from Listing 13.2, parallelized via OpenMP.

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  const unsigned long N = 600000;

  double * x = new double[N];
  for (unsigned long i=0; i<N; ++i)
    x[i] = i;

  Clock c;
  // Compile this with -fopenmp to enable multithreading:
  #pragma omp parallel for
  for (unsigned long i=0; i<N; ++i)
    x[i] += 7;
  c.ptock();

  // To prevent dead code elimination:
  std::cout << x[0] << std::endl;

  return 0;
}
```

Listing 13.3: A parallelizable loop with a higher cost per iteration.

```
#include "../Clock.hpp"
#include <iostream>
#include <cmath>

int main() {
  const unsigned long N = 600000;

  double * x = new double[N];
  for (unsigned long i=0; i<N; ++i)
    x[i] = i;

  Clock c;
  for (unsigned long i=0; i<N; ++i)
    x[i] = sin(x[i]);
  c.ptock();

  // To prevent dead code elimination:
  std::cout << x[0] << std::endl;

  return 0;
}
```

expensive, that simple `#pragma omp parallel for` *does* yield an improvement: Listing 13.3 shows a similar absurdly parallelizable problem, but where the more expensive trigonometric function `sin` is used. By adding that single `#pragma` (Listing 13.4), the runtime improves from 0.02121s to 0.005914s[5].

13.3 Using OpenMP with sections

Listing 13.6 shows a serial implementation of a simple loop that sums an array, which has an impressive runtime of 3.906e-09s; however, this fast runtime turns out to be an artifact of dead code elimination. Listing 13.5 fixes this by simply printing the result of the sum, making it relevant; the correct runtime of this serial implementation is 0.0002309s. Listing 13.7 shows a corresponding version with attempted use of OpenMP. Summing *can* be

[5]That's a pretty decent speedup for just adding one line of code and a single compilation flag.

Listing 13.4: The loop from Listing 13.3 parallelized via OpenMP.

```
#include "../Clock.hpp"
#include <iostream>
#include <cmath>

int main() {
  const unsigned long N = 600000;

  double * x = new double[N];
  for (unsigned long i=0; i<N; ++i)
    x[i] = i;

  Clock c;
  // Compile this with -fopenmp to enable multithreading:
  #pragma omp parallel for
  for (unsigned long i=0; i<N; ++i)
    x[i] = sin(x[i]);
  c.ptock();

  // To prevent dead code elimination:
  std::cout << x[0] << std::endl;

  return 0;
}
```

Listing 13.5: Computing the sum of an array.

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  const unsigned long N = 100000;

  double * x = new double[N];
  for (unsigned long i=0; i<N; ++i)
    x[i] = i;

  double tot = 0.0;

  Clock c;
  for (unsigned long i=0; i<N; ++i)
    tot += x[i];
  c.ptock();

  std::cout << tot << std::endl;

  return 0;
}
```

done in parallel, but unfortunately in this case, Listing 13.7 does not implement parallelism correctly: the variable `tot` is read and written to by each thread, making it unstable. Bugs like these are stochastic; not only do they change when the program is run repeatedly, they also *respond* to changes in our code (*e.g.*, when we insert `std::cout` statements for debugging, it may change the result, potentially even making the bug seem to disappear), and so they are especially pernicious. For this reason, such stochastic bugs are sometimes called "Heissenbugs".[6]

What we *can* do is split our summation into two sections: we compute the sum of the first half of the array, compute the sum of the second half of the array, and then sum these half-sums (Listing 13.8). Not only does this implementation produce the correct result, it does so in 0.001797s, a speedup over the non-paralellized version.

[6]Also, if we only run our multithreaded code one time look and thus never test its consistency, could we consider our code to simultaneously be correct *and* incorrect?[7]

[7]Yes. But it is traditional to put the computer in a box.

Listing 13.6: Computing the sum of an array, but with a benchmarking error due to dead code. Because the result of `tot` is never used, the program looks artificially fast.

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  const unsigned long N = 600000;

  double * x = new double[N];
  for (unsigned long i=0; i<N; ++i)
    x[i] = i;

  double tot = 0.0;

  Clock c;
  for (unsigned long i=0; i<N; ++i)
    tot += x[i];
  c.ptock();

  return 0;
}
```

Listing 13.7: The loop from Listing 13.5 parallelized via OpenMP. The results are not only unstable and incorrect, they are stochastic.

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  const unsigned long N = 100000;

  double * x = new double[N];
  for (unsigned long i=0; i<N; ++i)
    x[i] = i;

  double tot = 0.0;

  Clock c;
  #pragma omp parallel
  for (unsigned long i=0; i<N; ++i)
    // Error! tot will be modified simultaneously by multiple threads
    tot += x[i];
  c.ptock();

  std::cout << tot << std::endl;

  return 0;
}
```

Listing 13.8: Computing the sum of an array with OpenMP sections. This code will use 2 threads and fixes the error caused by the shared variable in Listing 13.7

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  const unsigned long N = 100000;

  double * x = new double[N];
  for (unsigned long i=0; i<N; ++i)
    x[i] = i;

  double tot = 0.0;

  Clock c;

  double tot_first_half = 0.0;
  double tot_second_half = 0.0;

  // Parallel things coming up:
  #pragma omp parallel
  {
    // Parallel sections to be described here:
    #pragma omp sections nowait
    {
      // This is one parallel section:
      #pragma omp section
      {
        for (unsigned long i=0; i<N/2; ++i)
          tot_first_half += x[i];
      }

      // This is a second parallel section:
      #pragma omp section
      {
        for (unsigned long i=N/2; i<N; ++i)
          tot_second_half += x[i];
      }
    }
  }
  tot = tot_first_half + tot_second_half;
  c.ptock();

  std::cout << tot << std::endl;

  return 0;
}
```

13.4 Computing a marginal

We can use parallelism to accelerate the computation of a marginal, that is, computing the sum of every row in a matrix. A serial version (Listing 13.9) runs in 0.3340s, but the parallelized version (Listing 13.10) runs in 0.1242s.

13.5 Limits of multithreading

Parallelism can achieve large speedups on certain problems, but it is no replacement for strong algorithms, cache localization, SIMD usage, compiler optimizations, *etc.* In short, there are limits to parallelism[8]. Even so, more and more powerful GPUs and paradigms like map-reduce (which can easily be run across clusters) mean that the future is likely to see an even strong case for parallel methods. This is partly true simply because we can easily solve parallelizable problems, and so we gravitate towards those problems that we can easily solve.[9]

Questions

1. [**Level 2**] After compiling Listing 13.10 with the OpenMP flag, run the executable using UNIX system time: which of these three reported times matches most closely with the wall clock time from our `Clock` class? Why? Which of these types of times would you most want to report when presenting the success or failure of a speedup to a CEO at a company where you work?

2. [**Level 3**] Update your matrix multiplication methods from Chapter 11 Question 4 (both the naive and transposed versions) to use OpenMP. Which performs better, adding `#pragma omp parallel for` above the outer-most loop, adding the `#pragma` around the inner-most loop instead, or adding the `#pragma` around both loops? Record the speedup each produces (compared to the non-OpenMP version)? Which version

[8] "Even Dr. Manhattan can't be everywhere."

[9] A friend's wife from Microsoft said that developers are told to optimize their code under the assumption that it will have infinite threads available; this is a strong endorsement of the prediction that parallelism in hardware will continue to improve faster than clock speed.

Listing 13.9: Computing the marginal of a matrix. The sum of each row is computed.

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  const unsigned long N = 1<<14;

  double * x = new double[N*N];
  for (unsigned long i=0; i<N*N; ++i)
    x[i] = i;

  double * row_sums = new double[N];

  Clock c;

  for (unsigned int i=0; i<N; ++i) {
    double tot = 0.0;
    for (unsigned int j=0; j<N; ++j)
      tot += x[i*N+j];

    row_sums[i] = tot;
  }
  c.ptock();

  for (unsigned int i=0; i<10; ++i)
    std::cout << row_sums[i] << " ";
  std::cout << "..." << std::endl;

  return 0;
}
```

Listing 13.10: Parallelized version of Listing 13.9.

```
#include "../Clock.hpp"
#include <iostream>

int main() {
  const unsigned long N = 1<<14;

  double * x = new double[N*N];
  for (unsigned long i=0; i<N*N; ++i)
    x[i] = i;

  double * row_sums = new double[N];

  Clock c;

  #pragma omp parallel for
  for (unsigned int i=0; i<N; ++i) {
    double tot = 0.0;
    for (unsigned int j=0; j<N; ++j)
      tot += x[i*N+j];

    row_sums[i] = tot;
  }
  c.ptock();

  for (unsigned int i=0; i<10; ++i)
    std::cout << row_sums[i] << " ";
  std::cout << "..." << std::endl;

  return 0;
}
```

(the naive or transposed) achieves a larger improvement from parallelism? Why?

Chapter 14

Direct Memory Access

We noted at the end of Chapter 5 that bit-packed string operations can be so fast that the performance bottleneck will come from loading the ASCII file itself. Listing 14.1 demonstrates the method we'd used to load a large file consisting of `G`, `A`, `T`, and `C` characters. That version loaded via the `>>` operator via `std::ifstream`, and then used `std::string::operator +=` to append the character to the string. When loading a genome with 4434395 ASCII characters, it runs in 0.04744s.

Listing 14.1: Loading an ASCII file via the `>>` operator of `std::ifstream` and the `std::string::operator +=`.

```cpp
#include "../Clock.hpp"
#include <fstream>
#include <string>

int main() {
  Clock c;

  std::ifstream fin("../nucleotide-string/bigger.txt");
  std::string genome;

  char base;
  while (fin >> base)
    genome += base;
  c.ptock();

  std::cout << "Read " << genome.size() << " characters: " << genome[0]
      << genome[1] << genome[2] << genome[3] << "..." << std::endl;
  return 0;
}
```

14.1 Avoiding `std::string::operator +=`

As we observed in Chapter 6, appending to `std::vector` via `push_back` was far slower than allocating the proper size in advance and using the `[]` operator to insert the next value. This was because the `std::vector<T>::push_back` function would sometimes resize the vector, and even using an amortized $\widetilde{O}(1)$ scheme via exponential growth, the runtime constant from resizing was significant. The `std::string::operator +=` behaves similarly to `std::vector<T>::push_back`, and thus it is far better to load the file by allocating the correct number of characters in an array and then inserting their data via the `[]` operator (Listing 14.2). On the same 4434395-character ASCII file, this runs in 0.04022s; however, this moderate speedup comes at a cost: this new version currently uses a hard-coded value for the number of characters in the file. This is poor design (it is brittle to changes in the file size).

Listing 14.2: Loading an ASCII file without using the `std::string::operator +=`.

```
#include "../Clock.hpp"
#include <fstream>
#include <string>

int main() {
  // Warning: hard-coded value should match the number of characters
  // in the file:
  constexpr unsigned long N=4434395;

  Clock c;

  std::ifstream fin("../nucleotide-string/bigger.txt");
  char*genome = new char[N+1]; // Leave one extra character for the null
      char
  genome[N] = '\0';

  char base;
  for (unsigned long i=0; i<N; ++i) {
    fin >> base;
    genome[i] = base;
  }
  c.ptock();

  std::cout << "Read " << N << " characters: " << genome[0] << genome[1]
      << genome[2] << genome[3] << "..." << std::endl;

  return 0;
}
```

14.2 Direct memory access

"Direct memory access" (DMA) is a technique for fetching large amounts of contiguous data from/to disk to/from RAM. DMA can be substantially faster when reading and writing large files.[1]

There are multiple reasons why DMA can be faster than using **`std::ifstream::operator >>`**, but they can be concisely summarized by two related cases: First, we are telling the program what we want at a high level, and thus may benefit from an optimized implementation.[3] Second, there is the possibility that those software implementations of DMA may interface with custom hardware, a sort of wide pipe going directly from disk to RAM and bypassing the CPU.

Listing 14.3 demonstrates how we would load the ASCII text file via DMA. Note that unlike our previous approach (Listing 14.2), we do not need to hard code the size of the file; instead, we compute this at runtime by computing the difference between the end and start of the file. By using DMA, we decrease the runtime to 0.002495s, a $> 16\times$ speedup.

[1]You contrarians out there might reply, "You wouldn't really need to load files so large that they would be an issue, and even if you did, you would probably only load the file once. So this really isn't *that* useful." Well, if using DMA is that useless, then I challenge someone from the future to travel in time to this moment and murder me in order to save the human race from this unnecessary diversion. [pause] I guess no one's coming... What does that tell you?[2]

[2]Wait, who's that guy in the metalic spandex jumpsuit? Is that a laser pointer? Ow, it burns! Just kidding, no one from the future came to kill me. I guess I was right all along.

[3]This is reminiscent of our comments in Chapter 10 about the advantages of using the `memcpy` function.

Listing 14.3: Loading an ASCII file via DMA.

```
#include "../Clock.hpp"
#include <fstream>

int main() {
  Clock c;

  const char*fname = "../nucleotide-string/bigger.txt";

  // Get file size:
  std::ifstream fin(fname, std::ifstream::ate | std::ifstream::binary);
  unsigned long len = fin.tellg();

  char*genome = new char[len+1]; // +1 for terminating NULL
  genome[len] = 0;

  // Move back to the start of the file:
  fin.clear();
  fin.seekg(0, std::ios::beg);

  //  fin.read is equivalent to the following:
  /*
    for (unsigned long i=0; i<len; ++i)
      fin >> genome[i];
  */
  fin.read(genome, len);

  c.ptock();

  std::cout << "Read " << len << " characters: " << genome[0] <<
      genome[1] << genome[2] << genome[3] << "..." << std::endl;

  return 0;
}
```

14.3 Binary files

Our DMA implementation treats our ASCII file as a binary file; this is an important distinction. Consider what happens each time we call `std::ifstream::operator >>`: Loading the `float` value 7.6354 will load the character '`7`', followed by the character '`.`', followed by the characters '`6`', '`3`', '`5`', and '`4`'. Excluding the null-terminating character, the ASCII string `‘‘7.6354’’` contains 6 8-bit `char` types (*i.e.*, 48 bits). The raw data of a `float` type contains only 4 bytes (*i.e.*, 32 bits). For this reason, ASCII is often an inferior file format. What we want is to store the raw binary data[4]. Not only might an ASCII representation of a `float` be less space efficient, it will also take far longer to read: The `>>` operator needs to take different actions depending on the next character. A numeric character is another digit, a decimal point indicates where the decimal point lies (and is valid only when no previous decimal point has been found in this `float`), and whitespace indicates the end of the `float`. Furthermore, a `float` may be read using scientific notation, *e.g.*, `1.69e-17`, and so the finite state machine loading the float needs to be able to take different actions based on whether an `‘e’` character is loaded. Together, these different options mean that an ASCII representation of a `float` may be both time and space inefficient.

The downside of binary files is that they a no longer guaranteed to be human readable; opening a `.jpg` file in a text editor easily demonstrates the challenge of reading a file containing binary data[5]. The upside is that they pack data very efficiently.

In the case of the genome file used here, the ASCII file *can* be treated as a binary file without any conversion[6]. This is because the 8-bit `char ‘G’` is the same in both ASCII and binary representation[7]; in contrast, a `float` type may have disparate ASCII and binary forms. Listing 14.4 demonstrates how we would write an array of float types in ASCII format, while Listing 14.5 demonstrates how we would write an array of float types in binary format. Writing the ASCII file takes 0.5662s, while writing the binary file takes 0.008769s[8]; using a binary format introduces a $> 64\times$ speedup.

Listing 14.4: Creating an ASCII file from an array of `float` types.

```
#include "../Clock.hpp"
#include <fstream>
```

[4]Note that this is different from storing an *ASCII* text file of `0` and `1` characters, each of which require 8 bits despite only using 2 possible states; that would be the worst of both worlds.

[5]*I.e.*, many special characters may be used, and the file may look like a mess.

[6]Note that importantly, we store the genome in a standard string format, not in the bit-packed format used in Chapter 5.

[7]*i.e.*, it is an 8-bit binary integer in either case.

[8]Furthermore, the ASCII file may round some values to a lower number of digits; this represents information loss and also makes the current ASCII implementation in Listing 14.4 look more efficient than a lossless version.

```
int main() {
  srand(0);

  const char*fname = "ascii_file.txt";

  unsigned int N=1u<<20;
  float*arr = new float[N];
  for (unsigned int i=0; i<N; ++i)
    arr[i] = rand() / 99.0;

  Clock c;

  std::ofstream fout(fname);
  fout.precision(30);
  for (unsigned int i=0; i<N; ++i)
    fout << arr[i] << " ";

  c.ptock();

  return 0;
}
```

Listing 14.5: Creating an binary file from an array of `float` types.

```
#include "../Clock.hpp"
#include <fstream>

int main() {
  srand(0);

  const char*fname = "binary_file.txt";

  unsigned int N=1u<<20;
  float*arr = new float[N];
  for (unsigned int i=0; i<N; ++i)
    arr[i] = rand() / 99.0;

  Clock c;

  std::ofstream fout(fname, std::ios::binary);
  fout.write((char*)arr, N*sizeof(unsigned int));

  c.ptock();
  return 0;
}
```

We can recover the array of `float` types in a separate program by loading it from the file. Listing 14.6 loads the array of `float` types from ASCII, while Listing 14.7 loads the array of `float` types from a binary file. Loading from the ASCII file takes 0.2662s, while loading from the binary file takes 0.001290s; the binary version yields a $> 206\times$ speedup.

Listing 14.6: Reading an array of `float` types from an ASCII file. The array of `float` types written to `ascii_file.txt` by Listing 14.4 is recovered (except for any loss of precision when writing the `float` types).

```
#include "../Clock.hpp"
#include <fstream>

int main() {
  const char*fname = "ascii_file.txt";

  Clock c;

  // For simplicity, assume we know the number of integers in the file
  // (the number of bytes of data will not determine the number of
  // integers in ASCII format, since integers 0 or 2 or 1 use less
  // data than integer 202339):
  unsigned int N=1u<<20;

  float*arr = new float[N];

  std::ifstream fin(fname);
  for (unsigned int i=0; i<N; ++i)
    fin >> arr[i];

  c.ptock();

  std::cout << "Read " << arr[0] << " " << arr[1] << " " << arr[2] <<
      "..." << std::endl;

  return 0;
}
```

Listing 14.7: Reading an array of `float` types from a binary file. The array of `float` types written to `binary_file.txt` by Listing 14.5 is recovered.

```
#include "../Clock.hpp"
#include <fstream>

int main() {
  const char*fname = "binary_file.txt";

  Clock c;

  // Get file size:
  std::ifstream fin(fname, std::ifstream::ate | std::ios::binary);
  unsigned long N = fin.tellg();

  float*arr = new float[N];

  // Move back to the start of the file:
  fin.clear();
  fin.seekg(0, std::ios::beg);
  fin.read((char*)arr, N*sizeof(unsigned int));

  c.ptock();

  std::cout << "Read " << arr[0] << " " << arr[1] << " " << arr[2] <<
      "..." << std::endl;

  return 0;
}
```

Questions

1. [**Level 2**] Benchmark Listing 14.4 and Listing 14.5 on your machine. Repeat the benchmark after modifying the ASCII version (Listing 14.4) to write the float data in a lossless manner (*i.e.*, with the precision set to the maximum number of digits).

2. [**Level 3**] Consider the bit-packed nucleotide strings in Chapter 5. Create functions to read/write a bit-packed string from/to a binary file. Now that the bit-packed genome can be loaded directly and complemented using the bitwise $\sim$ operator (as before, assuming the superior bit-packed nucleotide code is used), what is the speedup of a bit-packed program that loads and complements a genome compared to the naive, ASCII `std::string`-based version first introduced in Chapter 5?

Chapter 15

FFT

The fast Fourier fransform (FFT) is widely regarded as one of the most important algorithms of the 20th century. One interpretation of it is the decomposition of a length-n vector "signal" into its corresponding frequency components in $O(n \log(n))$. Another equivalent formulation is the evaluation of a polynomial at n unique complex x values, where the coefficients of the polynomial are given by the n-vector. This latter interpretation is one of the keys to understanding the significance of FFT; where decomposition into frequency components sounds important for processing audio or video files, fast polynomial evaluation can be used to perform polynomial multiplication (*i.e.*, convolution of the coefficient vectors) in $O(n \log(n))$.

15.1 Simple, recursive `Python` implementation

Listing 15.1 constructs a simple, recursive implementation of the Cooley-Tukey FFT in `Python`. Performing an FFT of length 2^{15} takes 0.8428s; in comparison, the `numpy` FFT implementation takes only 0.004118s[1]. Our in-`Python` version is dramatically slower.

Several factors can contribute to the poor performance of our simple recursive FFT in `Python`: First, it is written natively in an interpreted language (as described in Chapter 2). Second, it makes local allocations in the

[1]`numpy` is written in `C` and `Fortran` and simply linked by `Python`); this permits users access to benefits of both interpreted and compiled languages.

variables `packed_evens` and `packed_odds` and in the intermediate recursive results `fft_evens` and `fft_odds`; these allocations prevent compiler optimizations (compared to a buffered implementation as described in Chapter 3) and are detrimental to cache performance (as described in Chapter 9). Third, this recursive `Python` FFT uses the `numpy.exp` function with complex arguments, which is computed via `sin` and `cos` functions; these functions were shown to be quite slow in Chapter 13. Fourth, it is a recursive implementation (overhead from standard recursion described in Chapters 10 & 12).[2].

Listing 15.1: Recursive implementation of FFT in `Python`.

```
import numpy
from time import time

# this is an r(n) = 2*r(n/2) + \Theta(n) \in \Theta(n log(n)) algorithm:
def fft(vec):
  n=len(vec)

  if n==1:
    return vec

  result = numpy.array(numpy.zeros(n), numpy.complex128)

  # packed coefficients eliminate zeros. e.g., f(x)=1+2x+3x**2+...,
  # then e(x)=1+3x**2+... = 1+0x+3x**2+0x**3+... = (1+3y+...),y=x**2.
  packed_evens = vec[::2]
  packed_odds = vec[1::2]

  # packed_evens(x**2) and packed_odds(x**2) for the first half of x
  # points. The other half of the points are the negatives of the
  # first half (used below).
  fft_evens = fft(packed_evens)
  fft_odds = fft(packed_odds)

  # Butterfly:
```

[2]Fifth, from the perspective of slightly obscure but still quite useful mathematics, we are also performing an FFT of a real-valued sequence; this can be implemented in roughly half the runtime by "packing" the real vector `[1,2,3,4,...]` into a complex vector `[1+2j,3+4j,...]`. The `numpy` library includes an implementation that uses this trick for performing real FFTs, and in some `numpy` functions, the input vector is tested to see if it contains complex values, and if not, the FFT is computed using the packed form instead. Real FFTs will not be discussed here, but can be found in most signal processing textbooks; a portable $2\times$ speedup in one of the most important algorithms ever created is certainly worthwhile!

```
  for i in xrange(n/2):
    # result = evens(x) + x*odds(x), where x is a complex root of unity
    #        = packed_evens(x**2) + x*packed_odds(x**2)
    x = numpy.exp(-2*numpy.pi*i*1j/n)
    result[i] = fft_evens[i] + x * fft_odds[i]

  for i in xrange(n/2,n):
    # result = evens(x) + x*odds(x), where x is a complex root of unity
    #        = packed_evens(x**2) + x*packed_odds(x**2)
    x=numpy.exp(-2*numpy.pi*i*1j/n)
    # first half of points are negative of second half.
    # x_i = -x_{i+n/2}, x_i**2 = x_{i+n/2}**2; therefore
    # packed_evens(x_i**2) = packed_evens(x_{i+n/2}**2) and
    # packed_odds(x_i**2) = packed_odds(x_{i+n/2}**2)
    result[i] = fft_evens[i - n/2] + x * fft_odds[i - n/2]

  return result

if __name__=='__main__':
  N=2**15
  x=numpy.array(numpy.arange(N),float)

  t1=time()
  numpy_result = numpy.fft.fft(x)
  t2=time()
  print 'numpy fft:', numpy_result
  print 'took', t2-t1, 'seconds'

  print
  t1=time()
  recursive_result = fft(x)
  t2=time()
  print 'fast ft:', recursive_result
  print 'took', t2-t1, 'seconds'

  print
  print 'Largest error', max(numpy.abs(numpy_result - recursive_result))
  print 'Recursive python FFT took', t2-t1, 'seconds'
```

As a result of the above shortcomings, "fast" algorithms based on this first FFT implementation will only be sped up significantly when we use *much* bigger arrays[3] and depending on the runtime constant of the algo-

[3]As we saw with sorting in Chapter 3, naive $O(n^2)$ algorithms often have faster runtime

rithm that calls our FFT (*e.g.*, perhaps some $O(n \log(n))$ algorithm needs to call our FFT 100 times), the "fast" algorithm may only be superior to the naive approach on an array so large we cannot store it in RAM on a current computer[4].

15.2 Recursive `C++` implementation

First, let us address the first source of poor performance, the fact that our `Python`, which uses loops in native `Python` code (*i.e.*, it does not defer the loop to a faster `numpy`-like library). This is essentially a symptom of writing `C++`-like code in `Python` and expecting it to perform well.[6]

Our recursive `C++` implementation (Listing 15.2) is much faster than our recursive FFT in `Python`. The `C++` version runs in 0.07754s; however, that is still more than 10× slower than the `numpy` implementation. As we saw with other implementations in this book, the implementation matters very much; we are essentially using a great method poorly.[8]

Listing 15.2: Recursive implementation of FFT in `C++`.

```
#include "../Clock.hpp"
#include <iostream>
```

constants.

[4]This is a frequent consideration with subcubic "fast" matrix multiplication algorithms, which are notoriously difficult to make substantially faster than naive for problems that would fit in RAM on any plausible computer.[5]

[5]These are matrices so big that just taking a picture of one and then hanging it on a wall would make the entire building crumble. *Huge* matrices.

[6]In fact, the statement `c++` is not even valid `Python` code; in `Python`, one must perform `c+=1` instead. On occasion, after spending a long time writing `C++` code, you may be tempted to write `c++` in `Python` instead of `c+=1`. DON'T DO IT! It is the equivalent of wandering around Berlin and asking, "Sprechen Sie Swiss?"[7]

[7]There are not many Swiss people in Berlin; however, there are *many* Swiss people in Switzerland. Why? Because Swiss people are *smart*. They know where the action is! From the mountains to the ocean, you might say that Switzerland has it all. The same could be said (literally this time) about France. Coincidentally, France has many French people in it, who we can presume are also very smart. Are you smart? Then there is a chance that you may be Swiss. Or French.

[8]One might say that *we are standing on the shoulders of giants and then jumping up and down as hard as we can. Wearing cleats.*[9]

[9]This is also known as jawing def– no, wait, *snatching* defeat from the *jaws* of victory. Whew, that was close.

```
#include <complex>

std::complex<double> complex_exp(unsigned int i, unsigned int n) {
  return std::complex<double>{cos(-2*M_PI * i / n), sin(-2*M_PI * i / n)};
}

std::complex<double>* fft(const std::complex<double>*vec, const unsigned
    int n) {
  std::complex<double>*result = new std::complex<double>[n];
  if (n == 1)
    result[0] = vec[0];
  else {
    std::complex<double>*evens = new std::complex<double>[n/2];
    std::complex<double>*odds = new std::complex<double>[n/2];
    for (unsigned int i=0; i<n/2; ++i) {
      evens[i] = vec[2*i];
      odds[i] = vec[2*i+1];
    }
    std::complex<double>*fft_evens = fft(evens, n/2);
    std::complex<double>*fft_odds = fft(odds, n/2);
    delete[] evens;
    delete[] odds;

    for (unsigned int i=0; i<n/2; ++i)
      result[i] = fft_evens[i] + complex_exp(i,n)*fft_odds[i];
    for (unsigned int i=n/2; i<n; ++i)
      result[i] = fft_evens[i - n/2] + complex_exp(i,n)*fft_odds[i - n/2];

    delete[] fft_evens;
    delete[] fft_odds;
  }
  return result;
}

int main() {
  const unsigned int N=1<<15;

  std::complex<double>*x = new std::complex<double>[N];
  for (unsigned int i=0; i<N; ++i)
    x[i] = std::complex<double>{double(i),0.0};

  Clock c;
  std::complex<double>*fft_x = fft(x, N);
  c.ptock();
  for (unsigned int i=0; i<std::min(N,5u); ++i)
    std::cout << fft_x[i] << " ";
  std::cout << "..." << std::endl;

  std::complex<double>*y = new std::complex<double>[N];
  for (unsigned int i=0; i<N; ++i)
    y[i] = std::complex<double>{double(i),0.0};

  return 0;
}
```

15.3 An in-place C++ implementation

Like our `Python` implementation, the `C++` implementation in Listing 15.2 allocates memory on the fly. This on-the-fly memory allocation could be removed by using a buffer (as discussed in Chapter 3), but alternatively, we could simply make a truly in-place implementation, which uses no buffers whatsoever.

This is much more challenging than it would seem, because each extraction of even and odd indices (without a buffer) requires us to perform an in-place matrix transposition of an $\frac{n}{2} \times 2$ matrix. As discussed in Chapter 9, in-place transpositions of non-square matrices are non-trivial; however, that in-place transposition of a non-square matrix is but one of many operations performed. Let us zoom out and consider the problem as a whole. But we could do this transposition efficiently in an in-place manner, then we could try to implement the FFT so that it modifies the vector passed to it, overwriting it with the FFT.

Consider a length $n = 8$ FFT. The indices will be the following binary integer values: `0b000`, `0b001`, `0b010`, `0b011`, `0b100`, `0b101`, `0b110`, `0b111`. If we were able to put the even-indexed values into the first 4 indices and put the odd-indexed values into the last 4 indices[10], then the permutation would place the original indices in this new order: `0b000`, `0b010`, `0b100`, `0b110`, `0b001`, `0b011`, `0b101`, `0b111`.

Our FFT would then recursively be called on the first half of the permuted vector and on the second half of the permuted vector. If possible, these

[10]Again, this corresponds to performing an $\frac{n}{2} \times 2$ matrix transposition in place, but just disregard this for now.

Index	First recursion	Second recursion	Third recursion
0b000	0b000	0b000	
0b001	0b010	0b100	
		———	
0b010	0b100	0b010	
0b011	0b110	0b110	
	———	———	[NO CHANGE]
0b100	0b001	0b001	
0b101	0b011	0b101	
		———	
0b110	0b101	0b011	
0b111	0b111	0b111	

Table 15.1: Permutation of a vector by recursive FFT calls with $n = 8$. The final index corresponds to the bit-reversed value of the original index in the same location.

permutations on even and odd indices at ever level of the recursion would permute the original indices as shown in Table 15.1. At each recursion, the least-significant bit of the index in the sub-problem is used to determine whether the value will be moved into the first or second half of the array (for that sub-problem size). As a result, the least-significant bit is essentially reinterpreted as the most-significant bit. Thus, unrolling the permutations performed by all recursions will swap each index with its bit-reversed value.

Because FFT first performs the recursive calls and then performs post-processing, an equivalent algorithm would apply these permutations in a bottom-up order (smallest FFT first)[11]. As a result, we could perform this full permutation in advance by simply swapping indices with their bit-reversed indices.[12]

[11]This is true for a "decimation in time" (DIT) FFT described here; the "decimation in frequency" (DIF) FFT is equivalent, but performs the recursive calls after pre-processing with complex arithmetic.

[12]A small subtlety: since the reverse of the reverse of an index is the original index, then we only want to visit each index once (whether as the index or the reverse index); otherwise, we will simply swap every index and then swap it back. One solution to this is to simply swap when the index is less[13] than the reversed index. This will ensure each pair will be visited only once.

[13]Alternatively, we could use greater than, and as long as we are consistent, the method will work for the same reason.

Listing 15.3 performs a recursive FFT in place by performing a bit-reversed permutation in advance and then assuming the values from even indices at each recursion already reside in the first half of the array (while the values from the odd indices at that recursion will be in the second half of the array). Furthermore, we have fused the two post-processing loops into a single loop and exploited the fact that `complex_exp(i+n/2,n) == -complex_exp(i,n)`, and thus halved the number of trigonometric calls performed.[14] This approach improves runtime to 0.03185s, less than half the runtime of our initial `C++` implementation.

Listing 15.3: Recursive in-place implementation of FFT in `C++`.

```
#include "../Clock.hpp"
#include <iostream>
#include <complex>

std::complex<double> complex_exp(unsigned int i, unsigned int n) {
  return std::complex<double>{cos(-2*M_PI * i / n), sin(-2*M_PI * i / n)};
}

unsigned int bit_reversed(unsigned int i, unsigned int num_bits) {
  unsigned int result = 0;
  for (unsigned char b=0; b<num_bits; ++b)
    result |= (((i & (1<<b)) >> b) << (num_bits-1)) >> b;
  return result;
}

void shuffle(std::complex<double>*vec, const unsigned int log_n) {
  for (unsigned int i=0; i<(1<<log_n); ++i) {
    unsigned int rev_i = bit_reversed(i, log_n);
    if (i < rev_i)
      std::swap(vec[i], vec[rev_i]);
  }
}

void apply_fft_helper(std::complex<double>*shuffled_vec, const unsigned
    int log_n) {
```

[14]Mathematically, this is true; however, they may be slightly numerically different. Since the algorithm is derived to satisfy mathematical principles, either of these slightly off numeric values is acceptable, and so we aren't too bothered that this will not be true numerically.[15]

[15]In short, it's true theoretically, and if it isn't true in practice, it was due to numerical error anyway.

```
  const unsigned int n=1<<log_n;

  if (log_n == 0) // n == 1, FFT has no effect
    return;

  apply_fft_helper(shuffled_vec, log_n-1);
  apply_fft_helper(shuffled_vec + n/2, log_n-1);

  for (unsigned int i=0; i<n/2; ++i) {
    std::complex<double> twiddle = complex_exp(i,n);

    std::complex<double> result_at_i = shuffled_vec[i] +
        twiddle*shuffled_vec[i + n/2];
    // complex_exp(i+n/2,n) == -complex_exp(i,n):
    std::complex<double> result_at_i_plus_n_over_2 = shuffled_vec[i] -
        twiddle*shuffled_vec[i + n/2];

    shuffled_vec[i] = result_at_i;
    shuffled_vec[i + n/2] = result_at_i_plus_n_over_2;
  }
}

void apply_fft(std::complex<double>*dest, unsigned int log_n) {
  shuffle(dest, log_n);
  apply_fft_helper(dest, log_n);
}

int main() {
  const unsigned int LOG_N = 15;
  const unsigned int N=1<<LOG_N;

  std::complex<double>*x = new std::complex<double>[N];
  for (unsigned int i=0; i<N; ++i)
    x[i] = std::complex<double>{double(i),0.0};

  Clock c;
  apply_fft(x, LOG_N);
  c.ptock();

  for (unsigned int i=0; i<std::min(N,5u); ++i)
    std::cout << x[i] << " ";
  std::cout << "..." << std::endl;

  return 0;
}
```

15.4 Eliminating trigonometric functions

Once we optimize other obstacles to performance (*e.g.*, allocations, cache misses, *etc.*), the runtime of the trigonometric functions will start to dominate[16]. One solution would be to simply compute those trigonometric values in a table in advace. Furthermore, FFT draws complex values from the unit circle[18], and so half of these values computed in an FFT of size n will be repeated in the recursions to the FFT of size $\frac{n}{2}$. For example $sin(x + \pi) = -sin(x)$. Likewise, $sin(x + \frac{\pi}{2}) = cos(x)$, meaning we only need to compute a table using `cos` rather than tables of both `sin` and `cos`. Listing 15.4 implements this, computing a large trigonometric table in an earlier offline step. It runs in 0.003732s, $\approx 10\times$ faster than our in-place implementation from Listing 15.3.[19]

Listing 15.4: Recursive in-place implementation of FFT in `C++` using a lookup table of trigonometric calls.

```cpp
#include "../Clock.hpp"
#include <iostream>
#include <complex>
#include <assert.h>

// Note: Could also use symmetry to cut cache size in half or quarter
double*COS_TABLE;
const unsigned int COS_TABLE_N = 1<<20;

std::complex<double> complex_exp(unsigned int i, unsigned int n) {
  // Uses fact that sin(theta) = cos(theta - pi/2)
  return std::complex<double>{ COS_TABLE[i * COS_TABLE_N / n],
      COS_TABLE[(i * COS_TABLE_N / n + COS_TABLE_N/4) % COS_TABLE_N]};
}

unsigned int bit_reversed(unsigned int i, unsigned int num_bits) {
  unsigned int result = 0;
  for (unsigned char b=0; b<num_bits; ++b)
    result |= (((i & (1<<b)) >> b) << (num_bits-1)) >> b;
  return result;
}
```

[16]This is because everything else is so efficient. What will happen if we remove the trigonometric calls?[17]

[17]Bane voice: "It would be extremely efficient. . . for you."

[18]These are "complex roots of unity" you sometimes hear described.

[19]"Let he who is without `sin` be very efficient."

```
void shuffle(std::complex<double>*vec, const unsigned int log_n) {
  for (unsigned int i=0; i<(1<<log_n); ++i) {
    unsigned int rev_i = bit_reversed(i, log_n);
    if (i < rev_i)
      std::swap(vec[i], vec[rev_i]);
  }
}

void apply_fft_helper(std::complex<double>*shuffled_vec, const unsigned
    int log_n) {
  const unsigned int n=1<<log_n;

  if (log_n == 0) // n == 1, FFT has no effect
    return;

  apply_fft_helper(shuffled_vec, log_n-1);
  apply_fft_helper(shuffled_vec + n/2, log_n-1);

  for (unsigned int i=0; i<n/2; ++i) {
    std::complex<double> twiddle = complex_exp(i,n);

    // Compute complex_exp once and reuse:
    std::complex<double> result_at_i = shuffled_vec[i] +
        twiddle*shuffled_vec[i + n/2];
    // complex_exp(i+n/2,n) == -complex_exp(i,n):
    std::complex<double> result_at_i_plus_n_over_2 = shuffled_vec[i] -
        twiddle*shuffled_vec[i + n/2];

    shuffled_vec[i] = result_at_i;
    shuffled_vec[i + n/2] = result_at_i_plus_n_over_2;
  }
}

void fill_cos_table() {
  COS_TABLE = new double[COS_TABLE_N];
  for (unsigned int i=0; i<COS_TABLE_N; ++i)
    COS_TABLE[i] = cos(-2*M_PI * i / COS_TABLE_N);
}

void apply_fft(std::complex<double>*dest, const unsigned int log_n) {
  assert(1<<log_n <= COS_TABLE_N);

  shuffle(dest, log_n);
  apply_fft_helper(dest, log_n);
}

int main() {
  const unsigned int LOG_N = 15;
  const unsigned int N=1<<LOG_N;

  std::complex<double>*x = new std::complex<double>[N];
  for (unsigned int i=0; i<N; ++i)
    x[i] = std::complex<double>{double(i),0.0};

  fill_cos_table();

  Clock c;
  apply_fft(x, LOG_N);
  c.ptock();

  for (unsigned int i=0; i<std::min(N,5u); ++i)
    std::cout << x[i] << " ";
  std::cout << "..." << std::endl;

  return 0;
}
```

There are drawbacks to this “lookup table” approach. For one thing, to perform FFT on a vector with a large n, the size of this lookup table will grow in $\Theta(n)$; therefore, we cannot perform an FFT in the case where a vector just barely fits in memory[20]. Likewise, it isn't perfectly fair to compare this lookup table implementation to the others because the time for computing the trigonometry is not included; even if that is performed offline, there is a use-case where we will only perform a single FFT, and then never use the table again. In that case, the trigonometric calls are still reduced (because they are reused in the recursions), but whenever the trigonometric functions are called, their cost should be included in the runtime. Moreover, for very large problems, the cache impact of that table could be very significant for large problems. *E.g.*, if the source vector being FFTed barely fits in the L2 cache, the trigonometric lookup table could dramatically increase the cost by producing many avoidable L2 cache misses. Because of these drawbacks, the lookup table approach is not satisfying. We will derive an alternate approach below.

[20]Unless we also utilize disk space.

15.5 Eliminating recursive calls

We can eliminate recursive calls by using template recursion instead[21]. A template-recursive implementation that also uses a trigonometric lookup table (Listing 15.5) runs in 0.002682s. This improvement comes not only from the elimination of recursive overhead (*e.g.*, passing parameters and retrieving return values through the stack), but also from an implementation that is better suited to compiler optimizations.

Listing 15.5: Template-recursive in-place implementation of FFT in C++ with a lookup table.

```cpp
#include "../Clock.hpp"
#include <iostream>
#include <complex>
#include <assert.h>

// Note: Could also use symmetry to cut cache size in half or quarter
double*COS_TABLE;
const unsigned int COS_TABLE_N = 1<<20;

std::complex<double> complex_exp(unsigned int i, unsigned int n) {
  // Uses fact that sin(theta) = cos(theta - pi/2)
  return std::complex<double>{ COS_TABLE[i * COS_TABLE_N / n],
      COS_TABLE[(i * COS_TABLE_N / n + COS_TABLE_N/4) % COS_TABLE_N]};
}

unsigned int bit_reversed(unsigned int i, unsigned int num_bits) {
  unsigned int result = 0;
  for (unsigned char b=0; b<num_bits; ++b)
    result |= (((i & (1<<b)) >> b) << (num_bits-1)) >> b;
  return result;
}

void shuffle(std::complex<double>*vec, const unsigned int log_n) {
  for (unsigned int i=0; i<(1<<log_n); ++i) {
    unsigned int rev_i = bit_reversed(i, log_n);
    if (i < rev_i)
      std::swap(vec[i], vec[rev_i]);
  }
}
```

[21]It may sound tricky to you, and you may find yourself asking, “What will a good implementation of this actually look like?” I don’t know– what does a gray wolf trotting through a snowy mountain pass at sunset look like? Majestic.

```
template <unsigned int LOG_N>
class FFTHelper {
public:
  static void apply(std::complex<double>*shuffled_vec) {
    constexpr unsigned int n=1<<LOG_N;

    FFTHelper<LOG_N-1>::apply(shuffled_vec);
    FFTHelper<LOG_N-1>::apply(shuffled_vec + n/2);

    for (unsigned int i=0; i<n/2; ++i) {
      std::complex<double> twiddle = complex_exp(i,n);

      // Compute complex_exp once and reuse:
      std::complex<double> result_at_i = shuffled_vec[i] +
          twiddle*shuffled_vec[i + n/2];
      // complex_exp(i+n/2,n) == -complex_exp(i,n):
      std::complex<double> result_at_i_plus_n_over_2 = shuffled_vec[i] -
          twiddle*shuffled_vec[i + n/2];

      shuffled_vec[i] = result_at_i;
      shuffled_vec[i + n/2] = result_at_i_plus_n_over_2;
    }
  }
};

template <>
class FFTHelper<0> {
public:
  static void apply(std::complex<double>*shuffled_vec) {
    // Do nothing
  }
};

void fill_cos_table() {
  COS_TABLE = new double[COS_TABLE_N];
  for (unsigned int i=0; i<COS_TABLE_N; ++i)
    COS_TABLE[i] = cos(-2*M_PI * i / COS_TABLE_N);
}

template <unsigned int LOG_N>
class FFT {
public:
  static void apply(std::complex<double>*dest) {
    assert(1<<LOG_N <= COS_TABLE_N);
    shuffle(dest, LOG_N);
    FFTHelper<LOG_N>::apply(dest);
  }
};

int main() {
  const unsigned int LOG_N = 15;
  const unsigned int N=1<<LOG_N;

  std::complex<double>*x = new std::complex<double>[N];
  for (unsigned int i=0; i<N; ++i)
    x[i] = std::complex<double>{double(i),0.0};

  fill_cos_table();

  Clock c;
  FFT<LOG_N>::apply(x);
  c.ptock();

  for (unsigned int i=0; i<std::min(N,5u); ++i)
    std::cout << x[i] << " ";
  std::cout << "..." << std::endl;

  return 0;
}
```

15.6 Removing the trigonometric lookup table

It may feel impossible to remove the lookup table without reverting to a version that computes every `sin` and `cos`; however, this can be done by using mathematical properties of complex roots of unity. If $e^{\theta j}$ is the first twiddle used (where $j = \sqrt{-1}$), then the next twiddle will have angle 2θ and will be written in complex polar form as $e^{2\theta j}$. This can be written as the first twiddle squared: $\left(e^{\theta j}\right)^2$. Note that we do not need to square in complex polar form; we can simply square the complex cartesian value, and we will have rotated an additional θ radians. Thus, by computing the first twiddle in the outer-most recursive call, we can compute every other twiddle: the next twiddle will always be the product of the current twiddle and the first twiddle in the current recursion[22]. This is shown in Listing 15.6.

The resulting runtime is 0.002917s, only slightly slower than the lookup table, and significantly faster than the highly optimized `FFTPACK` implementation employed by `numpy`. This could be sped up further by specializing our FFT template class to perform optimized FFTs on small problems without the use of complex arithmetic. Likewise, the complex arithmetic could be performed in a more efficient, `constexpr` manner. Some of these speedups will be realized automatically by the compiler if we enable the use of fast math: compiling with `-Ofast` instead of `-O3` further decreases our runtime to 0.001101s. Our initial recursive `Python` implementation, being $> 765\times$ slower, pales in comparison[23].

Listing 15.6: Template-recursive in-place implementation of FFT in `C++` using a trigonometric recurrence.

```
#include "../Clock.hpp"
#include <iostream>
#include <complex>

unsigned int bit_reversed(unsigned int i, unsigned int num_bits) {
  unsigned int result = 0;
  for (unsigned char b=0; b<num_bits; ++b)
    result |= (((i & (1<<b)) >> b) << (num_bits-1)) >> b;
  return result;
}

void shuffle(std::complex<double>*vec, const unsigned int log_n) {
  for (unsigned int i=0; i<(1<<log_n); ++i) {
    unsigned int rev_i = bit_reversed(i, log_n);
    if (i < rev_i)
      std::swap(vec[i], vec[rev_i]);
  }
}

template <unsigned int LOG_N>
class FFTHelper {
public:
  static void apply(std::complex<double>*shuffled_vec) {
    constexpr unsigned long N=1ul<<LOG_N;
```

[22]Furthermore, the first twiddles between recursions are related to one another; however, we do not yet use this.

[23]*I.e.*, it looks like a steaming pile of garbage: `https://en.wikipedia.org/wiki/List_of_burn_centers_in_the_United_States` .

```
    FFTHelper<LOG_N-1>::apply(shuffled_vec);
    FFTHelper<LOG_N-1>::apply(shuffled_vec + N/2);

    constexpr double theta = -2*M_PI / (1<<LOG_N);
    double cos_val = cos(theta);
    double sin_val = sin(theta);

    std::complex<double> first_twiddle{cos_val, sin_val};
    std::complex<double> twiddle{1.0, 0.0};

    for (unsigned int i=0; i<N/2; ++i) {
      // Compute complex_exp once and reuse:
      std::complex<double> result_at_i = shuffled_vec[i] +
          twiddle*shuffled_vec[i + N/2];
      // complex_exp(i+N/2,N) == -complex_exp(i,N):
      std::complex<double> result_at_i_plus_n_over_2 = shuffled_vec[i] -
          twiddle*shuffled_vec[i + N/2];

      shuffled_vec[i] = result_at_i;
      shuffled_vec[i + N/2] = result_at_i_plus_n_over_2;

      // Update twiddle:
      twiddle *= first_twiddle;
    }
  }
};

template <>
class FFTHelper<0u> {
public:
  static void apply(std::complex<double>*shuffled_vec) {
    // Do nothing
  }
};

template <unsigned int LOG_N>
class FFT {
public:
  static void apply(std::complex<double>*dest) {
    shuffle(dest, LOG_N);
    FFTHelper<LOG_N>::apply(dest);
  }
};
```

```
int main() {
  const unsigned int LOG_N = 15;
  const unsigned int N=1<<LOG_N;

  std::complex<double>*x = new std::complex<double>[N];
  for (unsigned int i=0; i<N; ++i)
    x[i] = std::complex<double>{double(i),0.0};

  Clock c;
  FFT<LOG_N>::apply(x);
  c.ptock();

  for (unsigned int i=0; i<std::min(N,5u); ++i)
    std::cout << x[i] << " ";
  std::cout << "..." << std::endl;

  return 0;
}
```

15.7 Improving numeric stability

Listing 15.6 does have a drawback: the recurrence is seeded by the computation of `cos(theta)` and `sin(theta)` where `theta` is inversely proportional to `N`, the problem size. On large problems, `theta` $\approx$ `0`. In such a case, `sin(theta)` $\approx$ `0`, but `cos(theta)` $\approx$ `1`. Floating point arithmetic (used by `float` and `double`) is very good at representing values close to 0, but not at representing values close to 1. For example, `double` can easily store `1e-20`, but will approximate `1 - 1e-20` $\approx$ `1`. As a result, the `cos` computation above will be seeded improperly for large problems, and may introduce substantial numerical error into each twiddle factor used.

We can improve this by once again thinking mathematically: we will try to do the work in terms of $cos(\theta) - 1$ rather than $cos(\theta)$, because $cos(\theta) - 1 \approx 0$ when $\theta \approx 0$. The value $cos(\theta) - 1 + sin(\theta) \cdot j$ is the first twiddle minus 1. We already established above that the next twiddle is the product of the current twiddle and the first twiddle (by thinking in complex polar terms). `twiddle*first_twiddle_minus_1 = twiddle*first_twiddle - twiddle = next_twiddle - twiddle`. This unwanted subtaction needs to be added back in[24]. Thus, instead of updating by applying `twiddle *= first_twiddle`, we would update by applying `twiddle = twiddle*first_twiddle_minus_1 + twiddle` or equivalently, `twiddle += twiddle*first_twiddle_minus_1`.

Computing `first_twiddle_minus_1` requires us to compute $cos(\theta) - 1$; doing this in the naive manner will still introduce the numerical error from computing $cos(\theta)$, and thus will introduce no improvement. Once again, we can use the recurrence that squaring a complex polar is the same as doubling its angle: $e^{\theta j} = \left(e^{\frac{\theta}{2}j}\right)^2 = \left(cos\left(\frac{\theta}{2}\right) + sin(\frac{\theta}{2})j\right)^2$. By expanding the square, we see that $cos(\theta) = cos\left(\frac{\theta}{2}\right)^2 - sin\left(\frac{\theta}{2}\right)^2$. Furthermore, for any x, $sin(x)^2 + cos(x)^2 = 1$. Thus,

$$\begin{aligned} cos(\theta) - 1 &= cos\left(\frac{\theta}{2}\right)^2 - sin\left(\frac{\theta}{2}\right)^2 - 1 \\ &= cos\left(\frac{\theta}{2}\right)^2 - sin\left(\frac{\theta}{2}\right)^2 - (sin(x)^2 + cos(x)^2), \end{aligned}$$

for any x. Using $x = \frac{\theta}{2}$ allows us to strategically eliminate all cosines from the function:

$$\begin{aligned} cos(\theta) - 1 &= cos\left(\frac{\theta}{2}\right)^2 - sin\left(\frac{\theta}{2}\right)^2 - (sin(x)^2 + cos(x)^2) \\ &= cos\left(\frac{\theta}{2}\right)^2 - sin\left(\frac{\theta}{2}\right)^2 - \left(sin\left(\frac{\theta}{2}\right)^2 + cos\left(\frac{\theta}{2}\right)^2\right) \\ &= -2sin\left(\frac{\theta}{2}\right)^2. \end{aligned}$$

Listing 15.7: Template-recursive in-place implementation of FFT in `C++` using a numerically stable trigonometric recurrence.

```
#include "../Clock.hpp"
```

[24] Added back in *to what*? This is a worrying sign we're getting lost in this sea of math; addition, subtraction, *etc.* should *always* take two arguments and return a value. This is why the headline "Swedes are more equal" is completely meaningless (equal *to what*?).[25]

[25] "In high school, I was voted 'most likely to'. At university, I raised awareness (primarily, about making things more equal). But now, I literally can't even." Don't be that person.

```
#include <iostream>
#include <complex>

unsigned int bit_reversed(unsigned int i, unsigned int num_bits) {
  unsigned int result = 0;
  for (unsigned char b=0; b<num_bits; ++b)
    result |= (((i & (1<<b)) >> b) << (num_bits-1)) >> b;
  return result;
}

void shuffle(std::complex<double>*vec, const unsigned int log_n) {
  for (unsigned int i=0; i<(1<<log_n); ++i) {
    unsigned int rev_i = bit_reversed(i, log_n);
    if (i < rev_i)
      std::swap(vec[i], vec[rev_i]);
  }
}

template <unsigned int LOG_N>
class FFTHelper {
public:
  static void apply(std::complex<double>*shuffled_vec) {
    constexpr unsigned long N=1ul<<LOG_N;

    FFTHelper<LOG_N-1>::apply(shuffled_vec);
    FFTHelper<LOG_N-1>::apply(shuffled_vec + N/2);

    constexpr double theta = -2*M_PI / (1<<LOG_N);
    double sin_val = sin(theta);
    // Using the numerically unstable recurrence, we see that
    // ( cos(theta/2) + sin(theta/2)j ) **2 -->
    // ( cos(theta) + sin(theta)j ).

    // Thus cos(theta) = cos(theta/2)**2 - sin(theta/2)**2.
    // Also, cos(theta/2)**2 + sin(theta/2)**2 = 1.
    // subtracting the 1 is the same as subtracting
    // cos(theta/2)**2 + sin(theta/2)**2,
    // and so from the cos(theta)-1 =
    // cos(theta/2)**2 - sin(theta/2)**2 - (cos(theta/2)**2 +
        sin(theta/2)**2)
    // = -2*sin(theta/2)**2.
    double sin_half_angle = sin(theta/2.0);
    double cos_val_minus_1 = -2*sin_half_angle*sin_half_angle;

    std::complex<double> first_twiddle_minus_1{cos_val_minus_1, sin_val};
    std::complex<double> twiddle{1.0, 0.0};

    for (unsigned int i=0; i<N/2; ++i) {
      // Compute complex_exp once and reuse:
      std::complex<double> result_at_i = shuffled_vec[i] +
          twiddle*shuffled_vec[i + N/2];
      // complex_exp(i+N/2,N) == -complex_exp(i,N):
      std::complex<double> result_at_i_plus_n_over_2 = shuffled_vec[i] -
          twiddle*shuffled_vec[i + N/2];

      shuffled_vec[i] = result_at_i;
      shuffled_vec[i + N/2] = result_at_i_plus_n_over_2;

      // Update twiddle:

      // twiddle*first_twiddle_minus_1 = twiddle*first_twiddle -
      // twiddle. Therefore, add twiddle in to correct for this:
      // twiddle = twiddle*first_twiddle_minus_1 + twiddle:
      twiddle += twiddle*first_twiddle_minus_1;
    }
  }
};

template <>
class FFTHelper<0u> {
public:
  static void apply(std::complex<double>*shuffled_vec) {
    // Do nothing
  }
};

template <unsigned int LOG_N>
class FFT {
public:
  static void apply(std::complex<double>*dest) {
    shuffle(dest, LOG_N);
    FFTHelper<LOG_N>::apply(dest);
  }
};

int main() {
  const unsigned int LOG_N = 15;
  const unsigned int N=1<<LOG_N;

  std::complex<double>*x = new std::complex<double>[N];
  for (unsigned int i=0; i<N; ++i)
    x[i] = std::complex<double>{double(i),0.0};

  Clock c;
  FFT<LOG_N>::apply(x);
  c.ptock();

  for (unsigned int i=0; i<std::min(N,5u); ++i)
    std::cout << x[i] << " ";
  std::cout << "..." << std::endl;

  return 0;
}
```

Thus, we can initialize `first_twiddle_minus_1` without cosine computation, and can update the twiddle to the next twiddle in a stable manner (Listing 15.7). The runtime of this method is essentially unchanged (using only two more addition operations– which can likely be performed simultaneously via SIMD– per iteration).

Questions

1. [**Level 2**] Is it possible to eliminate all trigonometric function calls made at runtime and also completely avoid any lookup table with size $\in \Theta(n)$? How?

2. [**Level 3**] Write a non-recursive FFT (using `for` loops instead of recursion and instead of template recursion). How does the runtime compare to the template-recursive FFT from Listing 15.5?

Projects

1. [**Level 2**] Make the most efficient FFT you can (using multithreading, SIMD intrinsics, DMA (*e.g.*, to quickly load a lookup table of trigonometric constants on the fly), and any other trick mentioned in this textbook).

Appendix A

Proof of Unique Inverse Modulo a Prime

A.1 Solving $ab \equiv c \pmod{m}$

Given a and m (both in $\mathbb{Z}$), to solve $ab \equiv c \pmod{m}$ for $x \in \mathbb{Z}$, it is sufficient to find a^{-1} such that $a^{-1}a \equiv 1 \pmod{m}$, and then multiply:

$$\begin{aligned} ab &\equiv c \pmod{m} \\ a^{-1}ab &\equiv a^{-1}c \pmod{m} \\ b &\equiv a^{-1}c \pmod{m} \end{aligned}$$

Note that this solution produces a congruence (*i.e.*, $\equiv$) rather than equality (*i.e.*, $=$), and so given one value of b that satisfies $ab \equiv c \pmod{m}$, then $b' = b + m$ will also satisfy (and inductively, $b' = b + qm$ where $q \in \mathbb{Z}$ also satisfies).

To find a^{-1} such that $a^{-1}a \equiv 1 \pmod{m}$ (note that this is *not* equivalent to $\frac{1}{a}$, because that value is not necessarily in $\mathbb{Z}$), we consider the remainder: if $a^{-1}a \equiv 1 \pmod{m}$, then we can write $a^{-1}a$ (which is the product of two integers, and therefore an integer) as $a^{-1}a = qm + 1$, meaning that when we divide $a^{-1}a$ by m, the result is a whole part q plus the remainder $\frac{1}{m}$.

A.2 Bézout identity

We will first introduce the Bézout Identity and then later use that to prove the existence of a^{-1} under certain conditions.

Bézout Identity: Given a and b (both $\in \mathbb{Z}$), then there exist x and y (both $\in \mathbb{Z}$) such that $ax + by = d$, where $d = gcd(a, b)$.

First, consider the set of all positive integer combinations: $S = \{as + bt : as + bt > 0,\ \forall s, t \in \mathbb{Z}\}$. Let $d = \min(S)$; this minimum will exist because S is a nonempty integer set (even when a and b are not guaranteed to be nonnegative, we can simply choose s and t to match their respective signs, ensuring a positive result), and is bounded below by 0, and therefore it has a minumum.

Now we can show that d divides a:

Clearly there exist q and r such that $a = qd + r$ (the remainder form of a when dividing by d). Note that since r is a remainder, it must be in $\{0, 1, \ldots d - 1\}$ (*i.e.*, when dividing by 5, the remainder must be in $\{0, 1, 2, 3, 4\}$).

First we observe that r is of the form $ax + by$: $r = a - qd$, which we see equals $a(1 - s) + bt$ by expanding $d = as + bt$.

Now will now try to show that $r = 0$ (equivalent to showing that d divides a, because we prove that dividing produces a remainder equal to zero). Suppose $r \neq 0$ (and because $r \in \{0, 1, \ldots d - 1\}$, r must be > 0). If $r > 0$ and r is of the form $r = ax + by$, then $r \in S$. But since $r \in \{0, 1, \ldots d-1\}$, then $r < d$, so we have proven that there is a value $r < d$ in S, which contradicts our definition of $d = \min(S)$. This contradiction negates our supposition that $r \neq 0$; now we know that $r = 0$. So d divides a (this is true w.l.o.g., so the same can be shown for b). Therefore, d is a common divisor of both a and b.

Now we will show that it is the greatest common divisor $d = gcd(a, b)$. Suppose any other value c also divides a and b ($\frac{a}{c} \in \mathbb{Z}$ and $\frac{b}{c} \in \mathbb{Z}$). Since d is of the form $d = ax + by$ (for some unknown integers x and y), then c must divide d: $\frac{d}{c} = \frac{a}{c}x + \frac{b}{c}y \in \mathbb{Z}$. Thus we can see that d includes all common factors of a and b, and therefore is the greatest common divisor.

Therefore, when we have integers a and b, and their greatest common divisor $d = gcd(a, b)$, then there exist some integers x and y that satisfy $ax + by = d$.

A.3 Applying Bézout Identity to Prove Existence of a^{-1}

Clearly $a^{-1}a = qm+1$ can be rewritten as $a^{-1}a - qm = 1$, which is equivalent to $a^{-1}a + q'm = 1$ where $q' = -q$. Using the Bézout Identity, we can guarantee that integers a^{-1} and q' exist for which $a^{-1}a + q'm = 1$ whenever $1 = gcd(a, m)$. If we choose a prime value of m, we are guaranteed that the greatest common divisor $gcd(a, m)$ will be one whenever $a \in \{0, 1, \ldots m - 1\}$ (this is the definition of prime: the only way to have a $gcd(a, m) \neq 1$ when m is prime would be to let $a = qm$ for some integer q). Therefore, a prime table size m will allow us to invoke the Bézout Identity and prove that an integer a^{-1} exists.[1]

A.4 Proving uniqueness of a^{-1}

Here we prove the uniqueness of $a^{-1} \pmod{m}$ whenever m is prime using a contradiction argument. Suppose there exist two integers x and y such that $x \not\equiv y \pmod{m}$ but where $ax \equiv ay \pmod{m}$. In other words, if x was a valid inverse of a using modulo m), then y would likewise be a valid inverse, proving the inverse $x = a^{-1}$ is not unique. We can subtract across to see that $ax - ay \equiv 0 \pmod{m}$ and thus $a(x - y) \equiv 0 \pmod{m}$, meaning that together a and $x - y$ compose the multi-set all of the prime factors of m (the multi-set of prime factors is of course m, since m is prime). If, as above, $a \in \{0, 1, \ldots m - 1\}$, then it cannot share any common factors with m (again, the definition of primality means not sharing any factors with any natural numbers other than 1). Thus $x - y$ must contain the prime factors of m, meaning $x - y = qm$ for some integer q. This indicates that $x - y \pmod{m} = 0$, or equivalently that $x \equiv y \pmod{m}$ (contradicting our definition above). Therefore, the inverse a^{-1} has no equivalents modulo m.

A.5 Summary

We see that when we consider a prime modulo m and values $a \in \{0, 1, \ldots m - 1\}$, then $a^{-1} \pmod{m}$ exists and is unique. This means that we can perform a division-like operation on congruences modulo a prime m. Note that this

[1] "Aghhgh! This man's appendix is about to burst!" Just hold on a little further…

description does not show you *how* to find the inverse $a^{-1} \pmod{m}$, rather it shows that it exists and is unique, which is enough to prove that $a \cdot b \equiv c \pmod{p}$ has a unique solution for $b \in \{1, 2, \ldots p-1\}$ whenever $a \neq 0$. Thus the strategy in Chapter 8 is a valid universal hash.[2]

[2] You did it! `The_Throne_Room_and_End_Title.mp4`