Chapter 1

Basic Vocabulary

Copyright Oliver Serang, 2018
University of Montana Department of Computer Science

1.1 What is an algorithm?

An algorithm is a “recipe for doing stuff”\footnote{Travis Wheeler, personal communication} Some very algorithmic tasks include

- Cooking
- Knitting / weaving
- Sorting a deck of playing cards
- Shuffling a deck of playing cards
- Playing a song from sheet music / guitar tab
- Searching an $n \times n$ pixel image for a smaller $k \times k$ image (e.g., of a face)
1.2 Classifications of algorithms

Since the notion of an algorithm is so general, it can be useful to group algorithms into classes.

1.2.1 Online vs. offline

Online algorithms are suitable for dynamically changing data, while offline algorithms are only suitable for data that is static and known in advance.

For example, some text editors can only perform “spell check” in an offline fashion; they wait until you request a spelling check and then process the entire file while you wait. Some more advanced text editors can perform spell check online. This means that as you type, spell check will be performed in realtime, e.g., underlining a mispelled word like “asdfk” with a little red squiggle the moment you finish typing it. Checking the entire file every time you type a keystroke will likely be too inefficient in practice, and so an online implementation has to be more clever than the offline implementation.

Alternatively, consider sorting a list. An offline sorting algorithm will simply re-sort the entire list from scratch, while an online algorithm may keep the entire list sorted (in algorithms terminology, the sorted order of the list is an “invariant”, meaning we will never allow that to change), and would insert all new elements into the sorted order (inserting an item into a sorted list is substantially easier than re-sorting the entire list).

Online algorithms are more difficult to construct, but are suitable for a larger variety of problems. There are also some circumstances where online algorithms are not currently possible. For example, in internet search, it would be very difficult to instantly update the database of webpages containing a keyword “asdfk” instantly upon someone typing the last keystroke of “asdfk” on their webpage.

1.2.2 Deterministic vs. random

Deterministic algorithms will perform identical steps each time they are run on the same inputs. Randomized algorithms on the other hand, will not necessarily do so.

Interestingly, randomized algorithms can actually be constructed to always produce identical results to a deterministic algorithm. For example, if you randomly shuffle a deck of cards until they are sorted, the final result will
be the same as directly sorting the cards using a deterministic method, but the steps performed to get there will be different. (Also, we expect that the randomized algorithm would be much more inefficient for large problems.)

Randomized algorithms are sometimes quite efficient and can sometimes be made to be robust against a malicious user who would want to provide the inputs in a manner that will produce poor performance. Because the precise steps that will be performed by randomized algorithms are more difficult to anticipate, then constructing such a malicious input is more difficult (and hence we can expect that malicious input may only some fraction of the time based on randomness).

For example, if you have a sorting algorithm that is usually fast, but is slow if the input list is given in reverse-sorted order, then a randomized algorithm would first shuffle the input list to protect against the possibility that a malicious user had given us the list in reverse-sorted order. The total number of ways to shuffle $n$ unique elements is $n!$, and only one of them would produce poor results, and so by randomizing our initial algorithm, we achieve a probability of $\frac{1}{n!}$ that the performance will be poor (regardless of whether or not our algorithm is tested by a malicious user). Since $n!$ grows so quickly with $n$, this means a poor outcome would be quite improbable on a large problem.

1.2.3 Exact vs. approximate

Exact algorithms produce the precise solution, guaranteed. Approximate algorithms on the other hand, are proven only to get close to the exact solution. An $\epsilon$-approximation of some algorithm will not be guaranteed to produce an exact solution, but it is guaranteed to get within a factor of $\epsilon$ of the precise solution. For example, if the precise solution had value $x$, then an $\epsilon$-approximation algorithm is guaranteed to return a value $\in \left[\frac{x}{\epsilon}, \epsilon x\right]$. Sometimes approximations use a hard-coded $\epsilon$, e.g. a 2-approximation, but other times $\epsilon$-approximations include $\epsilon$ in the runtime function, revealing how the runtime would change in response to a higher-quality or lower-quality

\footnote{Depending on the type of problem, this bound may be one-sided. For example, a 2-approximation of the traveling salesman problem should return a result $\in [x, 2x]$ where $x$ is the optimal solution. This is because $x$ is defined as the shortest possible tour length, and so in this case an estimate that falls below the best possible would likely be seen as unacceptable. In other contexts, a two-sided approximation would be fine; in those cases a 2-approximation would return a result $\left[\frac{x}{2}, 2x\right]$, where $x$ is the correct solution.}
approximation.

1.2.4 Greedy

Greedy algorithms operate by taking the largest “step” toward the solution possible in the short-term. For example, if you are a cashier giving someone change at a café, and the amount to give back is $6.83, then a greedy approach would be to find the largest note or coin in American currency not larger than the remaining change:

1. $5 is the largest denomination ≤ $6.83; $6.83 - $5 = $1.83 remaining.
2. $1 is the largest denomination ≤ $1.83; $1.83 - $1 = $0.83 remaining.
3. 50 is the largest denomination ≤ $0.83; $0.83 - $0.5 = $0.33 remaining.
4. 25 is the largest denomination ≤ $0.33; $0.33 - $0.25 = $0.08 remaining.
5. 5 is the largest denomination ≤ $0.08; $0.08 - $0.05 = $0.03 remaining.
6. 1 is the largest denomination ≤ $0.03; $0.03 - $0.01 = $0.02 remaining.
7. 1 is the largest denomination ≤ $0.02; $0.02 - $0.01 = $0.01 remaining.
8. 1 is the largest denomination ≤ $0.01; $0.01 - $0.01 = $0.00 remaining.

Thus you give the change using a single $5 note, a single $1 note, a 50 cent piece, a quarter, a nickel, and three pennies.

Some problems will be solved optimally by a greedy algorithm, while others will not be. In American currency, greedy “change making” as above is known to give the proper change in the fewest notes and coins possible; however, in other currency systems, this is not necessarily the case. For example, if a currency system had notes of value 1, 8, and 14, then reaching value 16 would be chosen by a greedy algorithm as $16 = 1 \times 14 + 2 \times 1$, using 3 notes; however $16 = 2 \times 8$ would achieve this using only 2 notes.

If you’ve ever not crossed at a crosswalk in the direction you want to go during a green light because you know that this small instant gratification gain will slow you down at subsequent crosswalks (e.g., crosswalks that do not have a stoplight, making you wait for traffic), you understand the hazard of greedy algorithms.
1.2. CLASSIFICATIONS OF ALGORITHMS

Listing 1.1: Recursive factorial.

def factorial(n):
    if n <= 1:
        # Note that factorial is only defined for non-negative values of
        # n
        return 1
    # factorial(n) is decomposed into factorial(n-1)
    return n*factorial(n-1)

1.2.5 Optimal vs. heuristic

An optimal algorithm is guaranteed to produce the optimal solution. For example, if you are searching for the best way to arrange \( n \) friends in a row of seats at the cinema (where each friend has a ranking of people they would most like to sit next to), then an optimal algorithm will guarantee that it returns the best arrangement (or one of the best arrangements in the case that multiple arrangements tie one another).

A heuristic algorithm on the other hand, is something that is constructed as “good in practice” but is not proven to be optimal. Heuristics are generally not favored in pure algorithms studies, but they are often quite useful in practice and are used in situations where practical performance on small or moderately sized problems matters more than theoretical performance on very large problems.

1.2.6 Recursive

Recursive algorithms decompose a problem into subproblems, some of which are problems of the same type. For example, a common recursive way to implement factorial in Python is shown in Listing 1.1.

Recursive methods must implement a base case, a non-recursive solution (usually applied to small problem sizes) so that the recursion doesn’t become infinite.

1.2.7 Divide-and-conquer

Divide-and-conquer algorithms are types of recursive algorithms that decompose a problem into multiple smaller problems of the same type. They are
heavily favored in historically important algorithmic advances. Prominent examples of divide-and-conquer algorithms include merge sort, Strassen matrix multiplication, and fast Fourier transform (FFT).

1.2.8 Brute force

Brute force is a staple approach for solving problems: it simply solves a problem by trying all possible solutions. For example, you can use brute force to find all possible ways to make change of a $0.33 (Listing 1.2). The output of all_ways_to_make_change(0.33) is

solution:
3 x 0.01
1 x 0.05
1 x 0.25

solution:
8 x 0.01
1 x 0.25

solution:
8 x 0.01
5 x 0.05

solution:
13 x 0.01
4 x 0.05

solution:
18 x 0.01
3 x 0.05

solution:
23 x 0.01
2 x 0.05

solution:
28 x 0.01
1 x 0.05
Listing 1.2: Brute force for enumerating all possible ways to make change.

```python
import numpy as np
import itertools

def all_ways_to_make_change(amount):
    # American currency:
    all_denominations = [0.01, 0.05, 0.25, 0.5, 1, 2, 5, 10, 20, 50, 100]
    # E.g., the maximum possible number of nickels will be amount / 0.05:
    possible_counts_for_each_denomination = [
        np.arange(int(np.ceil(amount/coin_or_note)))
        for coin_or_note in all_denominations]
    for config in itertools.product(*possible_counts_for_each_denomination):
        total = sum([num*val for num, val in zip(config, all_denominations)])
        # if the coins actually add up to the goal amount:
        if total == amount:
            print 'solution:'
            for num, val in zip(config, all_denominations):
                # only print when actually using this coin or note:
                if num > 0:
                    print num, 'x', val
            print
```

1.2.9 Branch-and-bound

Closely related to brute force algorithms, branch-and-bound is a method for avoiding checking solutions that can be proven to be sub-optimal. Rather than try all permutations as performed in Listing 1.2, you could instead code a recursive algorithm that would abort investigating a potential solution once the subtotal exceeded the goal amount. For example, Listing 1.2 tries multiple solutions that simultaneously include both 33 pennies and 3 nickels (e.g., 33 pennies, 3 nickels, 1 dime or 33 pennies, 3 nickels, 2 dimes, . . . ); all of these configurations can be aborted because they already exceed the goal amount (and because American currency does not feature negative amounts).

“Branch and bound” refers to a combination of brute force (“branch”ing in a recursive tree of all possible solutions) and aborting subtrees where all included solutions must be invalid or suboptimal (“bound”ing by cutting the recursive tree at such proveably poor solutions).