Chapter 3
Amortized Analysis

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

In some cases, the worst-case of an individual operation may be quite expensive, but it can be proven that this worst-case cannot occur frequently. For example, consider a stack data structure that supports operations: push, pop, and pop_all, where push and pop behave in the standard manner for a stack and where pop_all iteratively pops every value off of the stack using the pop operation (Listing 3.1). The worst-case runtimes of push and pop will be constant, while pop_all will be linear in the current size of the stack (Table 3.1).

If someone asked us what was the worst-case runtime of any operation on our stack, the answer would be $O(n)$; however, if they asked us what would be the worst-case runtime of several operations on an initially empty stack, the answer is more nuanced: although one individual operation performed may prove to be expensive, we can prove that this cannot happen often. This is the basis of “amortized analysis”.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst-case runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>1</td>
</tr>
<tr>
<td>pop</td>
<td>1</td>
</tr>
<tr>
<td>pop_all</td>
<td>$n$</td>
</tr>
</tbody>
</table>

Table 3.1: Worst-case runtimes of operations on a stack currently holding $n$ items.
Listing 3.1: Simple stack.

```python
class Stack:
    def __init__(self):
        self._data = []

    def push(self, item):
        # assume O(1) append (this can be guaranteed using a linked list implementation):
        self._data.append(item)

    def pop(self):
        item = self._data[-1]
        # assume O(1) list slicing (this can be guaranteed using a linked list implementation):
        self._data = self._data[:-1]
        return item

    def pop_all(self):
        while self.size() > 0:
            self.pop()

    def size(self):
        return len(self._data)
```

# push runs in O(1) under the assumptions listed:

# pop runs in O(1) under the assumptions listed:

# push runs in worst-case O(n) where n=len(self._data);
# however, the worst-case amortized runtime is in O(1)

### 3.1 Accounting method

When operating on an initially empty stack, each `pop` step performed by `pop_all` *must* have been proceeded by a completed `push` operation; after all, if we’re popping values from the stack, they must have been pushed at some point prior. Thus, we can use the “accounting method” to pay for that work in advance. Thus, we will adjust that a single `push` operation takes 1 step plus our advance payment of the 1 step of work necessary if we ever call `pop_all`. Because `pop_all` has been paid for in advance, it is essentially free. Likewise, `pop` operations are free for the same reason (because we have pre-allocated the cost in advance by paying during the `push` operations). Thus we can see that the cost of `push` is $2 \in O(1)$, and the cost of `pop` and `pop_all` are both $0 \in O(1)$. This leads to inexpensive “amortized” costs (Table 3.2).

Low amortized costs do not guarantee anything about an individual operation; rather, they guarantee bounds on the average runtime in any long sequence of operations.

### 3.2 Potential method

The “potential method” is an alternative, more complex approach to the accounting method for deriving amortized bounds. In the potential method, a “potential” function $\Phi$ is used. This potential function computes a numeric value from the current state of our stack.

At iteration $i$, let the state of our stack be noted $S_i$. $\Phi(S_i)$ is a numeric value computing the potential of $S_i$. Think of the potential function as stored-up work as in the accounting method, but where it can behave in a more complex manner than in the accounting method. Let the cost of the operation performed at iteration $i$ be denoted $c_i$. If we construct our potential function so that the potential of our initial data structure is greater than or equal to the potential after running $n$ iterations, (i.e., $\Phi(S_n) \geq \Phi(S_0)$), then the
total runtime of executing $n$ sequential operations, $\sum_{i=1}^{n} c_i$, can be bounded above:

$$\sum_{i=1}^{n} c_i + \Phi(S_i) - \Phi(S_{i-1}) = \left( \sum_{i=1}^{n} c_i \right) + S_n - S_0$$

$$\geq \sum_{i=1}^{n} c_i.$$ 

Thus, we can use $\hat{c}_i = c_i + \Phi(S_i) - \Phi(S_{i-1})$ as a surrogate cost and guarantee that we will still achieve an upper bound on the total cost.

We will need to choose our potential function strategically. In the case of our stack, we can choose our potential function $\Phi(S_i) = S_i$.size(), which holds our necessary condition that $\Phi(S_n) \geq \Phi(S_0)$ (because we start with an empty stack).

The upper bound of the amortized cost of a push operation will be

$$\hat{c}_i = c_i + S_i.size() - S_{i-1}.size() = 1 + S_i.size() - S_{i-1}.size() = 1 + 1 = 2 \in O(1),$$

because a push operation increases the stack size by 1. Likewise, an upper bound on the amortized cost of a pop operation will be

$$\hat{c}_i = c_i + S_i.size() - S_{i-1}.size() = 1 + S_i.size() - S_{i-1}.size() = 1 + 0 = 1 \in O(1),$$

because a pop operation will decrease the stack size by 1. An upper bound on the amortized cost of a pop.all operation will be

$$\hat{c}_i = c_i + S_i.size() - S_{i-1}.size() = S_{i-1}.size() + S_i.size() - S_{i-1}.size() = S_i.size() = 0 \in O(1),$$

because the size after running multi_pop will be $S_i = 0$. Thus we verify our result with the accounting method and demonstrate that every operation is $\in \tilde{O}(1)$.

The potential method is more flexible than the accounting method, because the “work” stored up can be modified freely (using any symbolic formula) at runtime; in contrast, the simpler accounting method accumulates the stored-up work in a static manner in advance.

¹This is called a “telescoping sum” because the terms collapse down as sequential terms cancel, just like collapsing a telescope.
3.3 VECTORS: RESIZABLE ARRAYS

3.3 Vectors: resizable arrays

Consider the vector, a data structure that behaves like a contiguous array, but which allows us to dynamically append new elements. Every time an array is resized, a new array must be allocated, the existing data must be copied into the new array, and the old array must be freed. Thus if you implement an append function by growing by only 1 item each time append is called, then the cost of \( n \) successive append operations will be

\[
1 + 2 + 3 + \cdots + n - 1 + n \in \Theta(n^2).
\]

This would be inferior to simply using a linked list, which would support \( O(1) \) append operations.

However, if each time an append operation is called, we grow the vector by more than we need each, then one expensive resize operation will guarantee that the following append operations will be inexpensive. If we grow the vector exponentially, then we can improve the amortized cost of \( n \) append operations

If we grow by 1 during each append operation, the runtime of performing \( n \) operations is \( \Theta(n^2) \), and thus the amortized cost of each operation is \( \in \tilde{O}(\frac{n^2}{n}) = \tilde{O}(n) \) per operation.

On the other hand, if we grow by doubling, we can see that a resize operation that resizes from capacity \( s \) to \( 2s \) will cost \( O(s) \) steps, and that this resize operation will guarantee that the subsequent \( s - 1 \) append operations each cost \( O(1) \). Consider the cost of all resize operations: to insert \( n \) items, the final capacity will be \( \leq 2n \) (because we have an invariant that the size is never less than half the capacity). Thus, the cost of all resize operations will be

\[
\begin{align*}
&\leq 2n + n + \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 + 1 \\
&< 2n \left( 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots \right) \\
&= 4n \\
&\in O(n).
\end{align*}
\]

\footnote{For simplicity, ignore the existence of the realloc function in C here.}

\footnote{For a complete derivation of why exponential growth is necessary and practical performance considerations, see Chapter 6 of “Code Optimization in C++11” (Serang 2018).}
The total cost will be the cost of actually inserting the \( n \) items plus the cost of all resize operations, which will be \( O(n) + O(n) = O(n) \). Since all \( n \) append operations can be run in \( O(n) \) total, then the amortized cost per each append will be \( \tilde{O}(\frac{n}{n}) = \tilde{O}(1) \).

### 3.3.1 Demonstration of our \( \tilde{O}(1) \) vector doubling scheme

In order to grow our vector by doubling, at every append operation we will check whether the vector is full (i.e., whether the size that we currently use in our vector has reached the full capacity we have allocated thus far). If it is full, we will resize to double the capacity. On the other hand, if our vector is not yet full, we will simply insert the new element. In Listing 3.2, the class `GrowByDoublingVector` implements this doubling strategy, while `GrowBy1Vector` uses a naive approach, growing by 1 each time (which will take \( O(n^2) \) time). The output of the program shows that appending 10000 elements by growing by 1 takes 2.144 seconds, while growing by doubling takes 0.005739s.

Listing 3.2: Two vector implementations. `GrowBy1Vector` grows by 1 during each `append` operation, while `GrowByDoublingVector` grows by doubling its current capacity.

```python
from time import time

class Vector:
    def __init__(self):
        self._size=0

    def size(self):
        return self._size

class GrowBy1Vector(Vector):
    def __init__(self):
        Vector.__init__(self)
        self._data = []

    def append(self, item):
        new_capacity = self.size()+1
        new_data = [None]*new_capacity
        # copy in old data:
```
3.3. VECTORS: RESIZABLE ARRAYS

```python
for i in xrange(self._size):
    new_data[i] = self._data[i]

self._data = new_data

self._data[self.size()] = item
self._size += 1

def get_data(self):
    return self._data

class GrowByDoublingVector(Vector):
    def __init__(self):
        Vector.__init__(self)
        # start with a capacity of 1 element:
        self._data = [None]

    def append(self, item):
        if self.size() == self.capacity():
            # make sure we grow initially if the capacity is 0 (which
            # would double to 0) by taking the max with 1:
            new_capacity = 2*self.capacity()
            new_data = [None]*new_capacity
            # copy in old data:
            for i in xrange(self._size):
                new_data[i] = self._data[i]

            self._data = new_data

        self._data[self.size()] = item
        self._size += 1

    def capacity(self):
        return len(self._data)

    def get_data(self):
        return self._data[:v.size()]

N=10000

t1=time()
v = GrowByDoublingVector()
for i in xrange(N):
```

N=10000

t1=time()
v = GrowByDoublingVector()
for i in xrange(N):
v.append(i)
# print v.get_data()
t2=time()
print 'Growing by 1 took', t2-t1, 'seconds'

t1=time()
v = GrowByDoublingVector()
for i in xrange(N):
    v.append(i)
# print v.get_data()
t2=time()
print 'Growing by doubling took', t2-t1, 'seconds'