# CMSC 420: Amortized Analysis

Justin Wyss-Gallifent

February 7, 2023

<table>
<thead>
<tr>
<th>1 Introduction</th>
<th>2 Typically Worst-Case</th>
<th>3 Aggregate Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>3.2 Allocation for a Stack</td>
<td></td>
</tr>
<tr>
<td>4 Token Method</td>
<td>4.1 Introduction</td>
<td>4.2 An Inequality Note</td>
</tr>
<tr>
<td>4.3 Allocation for a Stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 Further Examples</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
1 Introduction

Imagine a situation in which we perform $n$ operations. Each time there is some cost involved and that cost can vary. We may ask what the average cost per operation is. Here the cost could be any resource such as time or memory.

Definition 1.0.1. The amortized cost of an operation performed $n$ times is simply the average cost.

Example 1.1. Suppose we do $n$ operations and each costs 5. The total cost is then $C(n) = 5n$ and the average cost per operation, the amortized cost, is:

$$AC(n) = \frac{1}{n}(5n) = 5 = \Theta(1)$$

That was a particularly silly example, so here is another:

Example 1.2. Suppose we do $n$ operations. We find that each operation has a base cost of 1 except operations 1, 2, 4, 8, 16, ... have supplemental costs of 1, 2, 4, 8, 16, ... respectively.

Thus the total cost of $n$ operations is:

$$1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + ...$$

This can be handily split up into the basic costs and the supplemental costs:

$$\frac{1 + 1 + ... + 1 + 1 + 2 + 4 + ...}{n}$$

It’s not immediately obvious how many terms are in the second expression above but it’s not hard to work out. The second expression gains a term each time that $n$ encounters a power of 2, thus since for any $n$ the highest power of 2 we will have passed is $\lfloor \log_2 n \rfloor$ we know that there are $\lfloor \log_2 n \rfloor$ terms in the second expression.

Thus the total cost is:

$$C(n) = n + 1 + 2 + 4 + ... + 2^{\lfloor \log_2 n \rfloor}$$

Thus the total cost of doing $n$ pushes and pops will be:

$$C(n) = n + 1 + 2 + 4 + ... + 2^{\lfloor \log_2 n \rfloor} = n + 2^{\lfloor \log_2 n \rfloor + 1} - 1$$

And the amortized cost will be:
\[ AC(n) = \frac{1}{n} \left( n + 2^{1+\lfloor \log n \rfloor} - 1 \right) \]

We can take this a little further in this case because we can discover a constant bound which holds for all \( n \). Observe that for all \( n \) we have:

\[ AC(n) \leq \frac{1}{n} (n + 2^{1+\lfloor \log n \rfloor} - 1) = \frac{1}{n} (n + 2n - 1) = 3 - \frac{1}{n} < 3 \]

And so in fact the amortized cost is always less than 3, for every \( n \), and hence we can also say that \( AC(n) = O(1) \).

2 Typically Worst-Case

Generally we will be analyzing situations in which the operations can vary. For example if we have the dictionary operations on a data structure - search (S), insert (I), and delete (D), then \( n \) operations can arise a number of ways such as:

\[ \begin{array}{c}
\text{III...} \\
\text{n operations} \\
\text{or} \\
\text{IDIDID...} \\
\text{n operations} \\
\text{or} \\
\text{IIISIIIS...} \\
\text{n operations}
\end{array} \]

Typically when we do amortized analysis we will be looking for the worst-case scenario. So in the above situation we might ask which sequence of \( n \) operations (each S, I, or D) will lead to the highest cost and hence the highest amortized cost.

There are several different approaches to amortized analysis, we will discuss two.
3 Aggregate Method

3.1 Introduction

The aggregate method of amortized analysis involves explicitly calculating the total worst-case cost of \( n \) operations and then simply dividing by \( n \).

3.2 Allocation for a Stack

Consider the problem of space allocation for a stack stored as a list. Imagine the list is a certain size, and is not necessarily full. When we pop elements off the stack we remove them from the list but the list remains the same length. When we push elements onto the stack, provided the list has space, we simply put them in the correct place in the list. However it’s possible (likely, eventually!) that at some point we overflow the list and need to reallocate space and copy over the elements before the guilty push.

Pop is easy:

```
\begin{array}{c}
5 \ 0 \ 3 \ 7 \\
\end{array}
\Rightarrow
\begin{array}{c}
5 \ 0 \ 3 \\
\end{array}
```

Push is usually easy:

```
\begin{array}{c}
5 \ 0 \ 3 \ 7 \\
\end{array}
\Rightarrow
\begin{array}{c}
5 \ 0 \ 3 \ 7 \ 6 \\
\end{array}
```

But maybe not:

```
\begin{array}{c}
5 \ 0 \ 3 \ 7 \ 6 \ 2 \ 1 \\
\end{array}
\Rightarrow
\begin{array}{c}
5 \ 0 \ 3 \ 7 \ 6 \ 2 \ 1 \ 6 \\
\end{array}
```

Suppose that:

- It costs 1 to pop an element off the stack.
- It costs 1 to push an element on the stack provided list reallocation is not necessary.
- If reallocation is necessary it costs \( k \) to reallocate a (new) list of length \( k \) including copying over the elements (but not including the guilty push).

In the worst-case we keep pushing and allocating with no popping at all. The reason this is the worst-case is that we are attempting to reallocate as often as we can to increase the cost, therefore avoiding pops. Thus the worst-case is the successive pushing of \( n \) elements.

**Example 3.1.** Suppose that when we need to reallocate we simply add 1 to the length of the list.

Let’s stick with the same example as earlier but instead of doubling the list’s length when reallocation is needed we simply increase the list’s length by 1. Here is the aggregate method.

We start with a list of length 0. We want to push but we need to reallocate,
so we reallocate to length 1, which costs 1, and we push, which costs 1, for a total cost of 1 + 1. Now we want to push again but we need to reallocate again, so we reallocate to length 2 (which also copies), which costs 2, and we push, which costs 1, for a total cost of 2 + 1. Now we want to push again but we need to reallocate again, so we reallocate to length 3 (which also copies), which costs 3, and we push, which costs 1, for a total cost of 3 + 1. And so on.

The total cost of pushing $n$ elements will be:

$$(1+1)+(2+1)+(3+1)+\ldots+(n+1) = \sum_{i=2}^{n+1} i = \frac{(n+1)(n+2)}{2} - 1 = \frac{1}{2}n^2 + \frac{3}{2}n$$

Thus the total cost of doing $n$ pushes and will satisfy:

$$C(n) = \frac{1}{2}n^2 + \frac{3}{2}n$$

The amortized cost then satisfies:

$$AC(n) = \frac{\frac{1}{2}n^2 + \frac{3}{2}n}{n} = \frac{1}{2}n + \frac{3}{2}$$

Observe that $AC(n) = \mathcal{O}(n)$, which isn’t great.

Let’s try something else!

**Example 3.2.** Suppose when we need to reallocate we do so by extending the list length by exactly double, with the exception of a list of length 0 which will be extended to a list of length 1. In a worst-case what would this cost per operation on average?

We start with a list of length 0. We want to push but we need to reallocate, so we reallocate to length 1, which costs 1, and we push, which costs 1, for a total cost of 1 + 1.

Zero Length List $\Rightarrow_{\text{Push}} X$

Now we want to push again but we need to reallocate again, so we reallocate to length 2 (which also copies), which costs 2, and we push, which costs 1, for a total cost of 2 + 1.

$$X \Rightarrow_{\text{Push}} X X$$

Now we want to push again but we need to reallocate again, so we reallocate to length 4 (which also copies), which costs 4, and we push, which costs 1, for a total cost of 4 + 1.
Now we want to push again and we don’t need to reallocate, say the reallocation cost is 0, as we have space, thus the total cost is just $0 + 1$.

Now we want to push again but we need to reallocate again, so we reallocate to length 8 (which also copies), which costs 8, and we push, which costs 1, for a total cost of $8 + 1$.

Now we get three cheap pushes, and so on.

In brief we have:

- Push 1: Cost = $1 + 1$
- Push 2: Cost = $2 + 1$
- Push 3: Cost = $4 + 1$
- Push 4: Cost = $1$
- Push 5: Cost = $8 + 1$
- Push 6: Cost = $1$
- Push 7: Cost = $1$
- Push 8: Cost = $1$
- Push 9: Cost = $16 + 1$
- Etc.

As we can see, after a reallocation from length $k$ to length $2k$ and pushing, we have $k + 1$ elements on the stack and hence have $2k - (k + 1) = k - 1$ pushes which don’t require reallocation. Thus the total cost of pushing $n$ elements follows the pattern:

$$C(n) = (1+1)+(2+1)+(4+1)+(0+1)+(8+1)+(0+1)+(0+1)+(0+1)+(16+1)+...+???
$$

This can be handily split up into push costs plus reallocation costs:

$$C(n) = \underbrace{1+1+...+1}_{\text{Push } n \text{ Times}} + \underbrace{1+2+4+...}_{\text{Reallocate}}$$

This reminds us of an earlier example but it’s a bit different. Here are two ways of figuring out how far the allocation sum goes.
1. Complicated Way:

Reallocations happen at pushes:

$$1, 2, 3, 5, 9, 17, ..., ?$$

Other than push 1 (which is special since it’s not a doubling push) the rest are pushes of the form $$2^k + 1$$ for $$k = 0, 1, 2, 3, ..., ?$$ and so if we push $$n$$ times the reallocations happen for all pushes with: $$2^k + 1 \leq n$$ for integers $$k$$. Since the $$k$$ are integers this corresponds to $$k \leq \lfloor \lg(n-1) \rfloor$$ and so we get reallocations at pushes:

$$1, 2^0 + 1, 2^1 + 1, 2^2 + 1, 2^3 + 1, 2^4 + 1, ..., 2^{|\lg(n-1)|} + 1$$

The reallocation cost of push 1 is 1 and then other than push 1 the reallocation cost of push $$2^k + 1$$ is $$2^{1+k}$$.

Thus the total cost of the reallocations is:

$$1 + 2 + 4 + 8 + ... + 2^{1+|\lg(n-1)|}$$

Thus the total cost is:

$$C(n) = n + 1 + 2 + 4 + 8 + ... + 2^{1+|\lg(n-1)|}$$

$$= n + \sum_{i=0}^{1+|\lg(n-1)|} 2^i$$

$$= n + 2^{2+|\lg(n-1)|} - 1$$

This can be bounded:

$$C(n) \leq n + 2^{2+|\lg(n-1)|} - 1 = n + 4(n - 1) - 1 = 5n - 5$$

Thus the amortized cost satisfies:

$$AC(n) \leq \frac{5n - 5}{n} = \frac{5}{b} < 5$$

And so $$AC(n) = \mathcal{O}(1)$$.

2. Easy Way: In this case if we push a total of $$n$$ elements then the final reallocation must be to a list of length $$2^k$$ with $$2^k \geq n$$, that is $$k \geq \lg n$$,
and in fact will be the smallest such reallocation, meaning \( k = \lceil \lg n \rceil \).

Thus the total cost is:

\[
C(n) = n + 1 + 2 + 4 + \ldots + 2^{\lceil \lg n \rceil}
\]

\[
= n + \sum_{i=0}^{\lceil \lg n \rceil} 2^i
\]

\[
= n + 2^{\lceil \lg n \rceil} - 1
\]

This can be bounded:

\[
C(n) \leq n + 2^{2 + \lg n} - 1 = n + 4n - 1 = 5n - 1
\]

Thus the amortized cost satisfies:

\[
AC(n) \leq \frac{5n - 1}{n} = 5 - \frac{1}{n} < 5
\]

And so \( AC(n) = \mathcal{O}(1) \).

## 4 Token Method

### 4.1 Introduction

To motivate the token method, suppose we perform \( n \) operations with costs \( x_1, x_2, \ldots, x_n \) respectively. The aggregate method simply calculates:

\[
AC(n) = \frac{x_1 + x_2 + \ldots + x_n}{n}
\]

Let’s define \( \beta = AC(n) \) and observe that this can be rewritten:

\[
\beta = \frac{x_1 + x_2 + \ldots + x_n}{n}
\]

\[
\beta n = x_1 + x_2 + \ldots + x_n
\]

\[
(\beta - x_1) + (\beta - x_2) + \ldots + (\beta - x_n) = 0
\]

It follows that finding the amortized cost \( \beta \) is equivalent to solving the equation:

\[
(\beta - x_1) + (\beta - x_2) + \ldots + (\beta - x_n) = 0
\]

Suppose we think of \( \beta \) being a number of tokens put into a bank account and the \( x_i \) being tokens spent for the operations. This equation then means that
cheap operations having $\beta > x_i$ yield a surplus and that surplus can be used for expensive operations having $\beta < x_i$.

The token method of amortized analysis basically works by solving for $\beta$ so that the surpluses yielded by the cheap operations are sufficient to cover the expensive operations.

### 4.2 An Inequality Note

In practice both the aggregate and token analyses usually yield inequalities.

In the aggregate method this is because we often have things like floors and ceilings which result in us not knowing $AC(n)$ exactly but having:

$$AC(n) = \frac{x_1 + x_2 + \ldots + x_n}{n} \geq ??$$

In the token method this is because we usually end up stating that $\beta$ must be greater than or equal to some value or expression in order for the surplus to cover the expensive operations.

### 4.3 Allocation for a Stack

Looking at the same example from earlier, suppose we deposit $\beta$ into the account with each operation.

**Example 4.1.** As before, first suppose that when we need to reallocate we simply add 1 to the length of the list. We claim that for each $1 \leq k \leq n$ we have deposited enough in the account for push number $k$.

Observe that for each $k$ we will have deposited a total of $\beta + \beta + \ldots + \beta = k\beta$ tokens. Pushes $1, 2, \ldots, k - 1$ will have cost us (including allocation) a total of:

$$(1 + 1) + (2 + 1) + (3 + 1) + \ldots + (k - 1 + 1) = \frac{k(k + 1)}{2} - 1$$

Thus our balance will be:

$$k\beta - \left(\frac{k(k + 1)}{2} - 1\right)$$

This needs to cover the cost of push $k$, which is $k + 1$, and so we need:
\[
k\beta - \left( \frac{k(k+1)}{2} - 1 \right) \geq k + 1
\]
\[
k\beta - \frac{1}{2}k^2 - \frac{1}{2}k + 1 \geq k + 1
\]
\[
k\beta \geq \frac{1}{2}k^2 + \frac{3}{2}k
\]
\[
\beta \geq \frac{1}{2}k + \frac{3}{2}
\]

Since this must hold for all \(1 \leq k \leq n\) we must have:

\[
\beta \geq \frac{1}{2}n + \frac{3}{2}
\]

Now let’s look at the doubling option.

**Example 4.2.** We divide the process into runs. Each run starts right after a reallocation plus push and ends right after the next reallocation plus push. Consequently each run consists of cheap pushes and ends with an expensive reallocation. Our goal is to show that we collect enough tokens during the cheap pushes to cover the reallocation.

Note: The first and last runs are special and we’ll ignore those for now.

Suppose we have just reallocated from length \(k\) to length \(2k\) and done a push, meaning there are \(k + 1\) elements on the stack. The next reallocation will occur at length \(2k\), when we reallocate to \(4k\) and do a push to \(2k + 1\) elements. Before that happens we will get at least \(k - 1\) cheap pushes and then one final push over the edge, which is expensive.

Each cheap push collects \(\beta\) tokens and spends 1 for a net gain of \(\beta - 1\) tokens. Thus at the end, right before the reallocation plus push we have at least \((\beta - 1)(k - 1)\) tokens in the bank.

The final push, not counting the reallocation, collects \(\beta\) tokens as well and spends 1 on pushing, meaning we have at \((\beta - 1)(k - 1) + (\beta - 1) = (\beta - 1)k\) tokens in the bank.

However the reallocation to length \(4k\) costs \(4k\) and so we must have:

\[
(\beta - 1)k \geq 4k
\]
\[
\beta - 1 \geq 4
\]
\[
\beta \geq 5
\]
5 Further Examples

Here is a different but classic example:

**Example 5.1.** Suppose we have a list $A$ which represents a binary string, so $A[0]$ represents the $2^0 = 1$ s digit, $A[1]$ represents the $2^1 = 2$ s digit, $A[2]$ represents the $2^2 = 4$ s digit, and so on.

Suppose $A$ starts at all 0s and we then implement $n$ increment operations. Every time a bit flips the cost is 1. Let’s calculate the corresponding amortized cost using the aggregate method.


This means that if we implement $n$ increments $A[0]$ will flip $n$ times, $A[1]$ will flip $\lfloor n/2 \rfloor$ times, $A[2]$ will flip $\lfloor n/4 \rfloor$ times, and so on, for a total cost of:

$$C(n) = n + \lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \ldots = \sum_{k=0}^{\infty} \lfloor n/2^k \rfloor$$

This sum is not infinite because a term is nonzero when $n/2^k \geq 1$ and this happens for $k \leq \lg n$, which for integers is $k \leq \lfloor \lg n \rfloor$, thus the total cost is really:

$$C(n) = \sum_{k=0}^{\lfloor \lg n \rfloor} \lfloor n/2^k \rfloor$$

This is a bit of an awkward sum because of the inner floor function so we’ll bound it:

$$C(n) \leq \sum_{k=0}^{\lg n} n/2^k = n \sum_{k=0}^{\lg n} \left(\frac{1}{2}\right)^k = n \left(1 - \frac{(1/2)^{1+\lg n}}{1-1/2}\right) = 2n \left(1 - \left(\frac{1}{2}\right)^{1+\lg n}\right)$$

The amortized cost then satisfies:

$$AC(n) \leq 2 \left(1 - \left(\frac{1}{2}\right)^{1+\lg n}\right)$$

We observe that we can bound this for any $n$, since for any $n$ we have $AC(n) < 2$ and so in fact $AC(n) = \mathcal{O}(1)$. 

11