CMSC 420: Amortized Analysis

Justin Wyss-Gallifent

September 9, 2024

1 Introduction

Imagine a situation in which a 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 term *amortized analysis* refers to analysis of a data structure in which the the costs associated with the most costly operations are averaged out over time.

Definition 1.0.2. The term *amortized cost* is used to refer to the average per-operation cost which arises in this situation.

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:

$$\underbrace{\text{IIIII....}}_{n\text{ operations}}$$

$$\text{or}$$

$$\underbrace{\text{IDIDID....}}_{n\text{ operations}}$$

$$\text{or}$$

$$\underbrace{\text{IIISIIIS....}}_{n\text{ operations}}$$

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 Elementary Abstract Example

Here is a simple warm-up example.

> **Example 3.1.** Suppose we perform n operations. Each costs 2 (call this the base cost) and every fifth operation costs an additional 10 (call this the surplus cost). What can we say about the average per-cost operation?
>
> The base cost will of course total $2n$ but the surplus cost is a tiny bit trickier. After n iterations we will have encountered $\lfloor n/5 \rfloor$ surplus cost events and so we will have a total of $10 \lfloor n/5 \rfloor$ surplus cost.
>
> Thus the overall cost is $C(n) = 2n + 10 \lfloor n/5 \rfloor$ and the average cost would be:
>
> $$AC(n) = \frac{2n + 10 \lfloor n/5 \rfloor}{n} \leq \frac{2n + 10(n/5)}{n} = 4$$
>
> Thus the amortized cost satisfies $AC(n) = \mathcal{O}(1)$.

3.3 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:

$$\begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 0 & 3 & 7 & & & \\ \hline \end{array} \underset{\text{Pop 7}}{\Longrightarrow} \begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 0 & 3 & & & & \\ \hline \end{array}$$

Push is usually easy:

$$\begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 0 & 3 & 7 & & & \\ \hline \end{array} \underset{\text{Push 6}}{\Longrightarrow} \begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 0 & 3 & 7 & 6 & & \\ \hline \end{array}$$

But maybe not:

$$\begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 0 & 3 & 7 & 6 & 2 & 1 \\ \hline \end{array} \underset{\text{Push 6}}{\Longrightarrow} \begin{array}{|c|c|c|c|c|c|c|}\hline 5 & 0 & 3 & 7 & 6 & 2 & 1 \\ \hline \end{array} \;\; 6$$

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).

Note 3.3.1. These are hypothetical criteria we are introducing for our examples and do not necessarily represent what happens in reality and may be tweaked for other problems!

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 3.2. Suppose that when we need to reallocate we simply add 1 to the length of the list.

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:

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

Thus the total cost of doing n pushes 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 3.3. 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 $1 + 1$.

$$\text{Zero Length List} \underset{\text{Push}}{\Longrightarrow} \begin{array}{|c|}\hline \text{X} \\ \hline\end{array}$$

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 $2 + 1$.

$$\begin{array}{|c|}\hline \text{X} \\ \hline\end{array} \underset{\text{Push}}{\Longrightarrow} \begin{array}{|c|c|}\hline \text{X} & \text{X} \\ \hline\end{array}$$

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 $4 + 1$.

$$\begin{array}{|c|c|}\hline \text{X} & \text{X} \\ \hline\end{array} \underset{\text{Push}}{\Longrightarrow} \begin{array}{|c|c|c|c|}\hline \text{X} & \text{X} & \text{X} & \phantom{\text{X}} \\ \hline\end{array}$$

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 $0 + 1$.

$$\begin{array}{|c|c|c|c|}\hline \text{X} & \text{X} & \text{X} & \phantom{\text{X}} \\ \hline\end{array} \underset{\text{Push}}{\Longrightarrow} \begin{array}{|c|c|c|c|}\hline \text{X} & \text{X} & \text{X} & \text{X} \\ \hline\end{array}$$

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 $8 + 1$.

$$\begin{array}{|c|c|c|c|}\hline \text{X} & \text{X} & \text{X} & \text{X} \\ \hline\end{array} \underset{\text{Push}}{\Longrightarrow} \begin{array}{|c|c|c|c|c|c|c|c|}\hline \text{X} & \text{X} & \text{X} & \text{X} & \text{X} & \phantom{\text{X}} & \phantom{\text{X}} & \phantom{\text{X}} \\ \hline\end{array}$$

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:

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

To figure out how far to take the reallocation sum, observe that 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:

$$\begin{aligned} C(n) &= n + 1 + 2 + 4 + ... + 2^{\lceil \lg n \rceil} \\ &= n + \sum_{i=0}^{\lceil \lg n \rceil} 2^i \\ &= n + 2^{1+\lceil \lg n \rceil} - 1 \end{aligned}$$

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, ..., x_n respectively. The aggregate method simply calculates:

$$AC(n) = \frac{x_1 + x_2 + ... + x_n}{n}$$

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

$$\beta = \frac{x_1 + x_2 + ... + x_n}{n}$$
$$\beta n = x_1 + x_2 + ... + x_n$$
$$(\beta - x_1) + (\beta - x_2) + ... + (\beta - x_n) = 0$$

It follows that finding the amortized cost β is equivalent to solving the equation:

$$(\beta - x_1) + (\beta - x_2) + ... + (\beta - x_n) = 0$$

Here's how to think about this equation:

For each operation we put β tokens into a bank account and operation i costs x_i tokens which we must remove from the bank account. Cheaper operations have $x_i < \beta$ while expensive operations have $x_i > \beta$.

The idea is then that the cheap operations give rise to a surplus in our bank account which can then be used to pay the rare but expensive operations.

4.2 Elementary Abstract Example

Let's return to our simple warm-up example.

Example 4.1. Suppose we perform n operations. Each costs 2 (call this the base cost) and every fifth operation costs an additional 10 (call this the surplus cost). What can we say about the average per-cost operation?

Suppose we deposit β into our account with each operation. Think of our operations in runs of five, each run ending with a surplus cost operation.

During each run we have four operations in which we deposit β and spend 2, yielding a net surplus of $4(\beta - 2)$ at the end of those four operations. For the fifth operation we deposit β again and thus have a surplus of $4(\beta-2)+\beta$ but now we need to spend $2 + 10 = 12$. Thus we have to ensure that:

$$\begin{aligned} 4(\beta - 2) + \beta &\geq 12 \\ 5\beta - 8 &\geq 12 \\ 5\beta &\geq 20 \\ \beta &\geq 4 \end{aligned}$$

Thus provided we deposit at least 4 tokens per operation we are safe, and so $AC(n) = 4 = \mathcal{O}(1)$, the same result as earlier.

4.3 Allocation for a Stack

Let's re-examine the stack allocation example from earlier when we double the allocation each time. Recall we are pushing n items onto the stack.

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. Otherwise for a run we have, for some k:

- Run Starts: We overflowed $2k$ and reallocated to $4k$. There are $2k+1$ items on the stack.
- Run Ends: We overflowed $4k$ and reallocated to $8k$. There are $4k+1$ items on the stack.

Thus we have pushed $4k+1-(2k+1) = 2k$ items and we reallocated to $8k$ for a total cost of:

$$2k + 8k = 10k$$

Thus we need ideally the smallest β with:

$$\begin{aligned} \beta(2k) &\geq 10k \\ \beta &\geq 5 \end{aligned}$$

It doesn't matter what k is here and so we can easily see that the smallest such β is $\beta = 5$ so $AC(n) = 5 = \mathcal{O}(1)$.

Here's the situtation when we allocate to the next highest perfect square each time: Recall we are pushing n items onto the stack.

Example 4.3. For a run we have, for some k:

- Run Starts: We overflowed k^2 and reallocated to $(k+1)^2$. There are k^2+1 items on the stack.
- Run Ends: We overflowed $(k+1)^2$ and reallocated to $(k+2)^2$. There are $(k+1)^2+1$ items on the stack.

Thus we have pushed $(k+1)^2+1-(k^2+1) = 2k+1$ items and we reallocated to $(k+2)^2$ for a total cost of:

$$(k+2)^2 + 2k + 1 = k^2 + 6k + 5$$

Thus we need ideally the smallest β with:

$$\begin{aligned} \beta(2k+1) &\geq k^2 + 6k + 5 \\ \beta &\geq \frac{k^2+6k+5}{2k+1} \\ \beta &\geq \frac{1}{2}k + \frac{11}{4} + \frac{9/4}{2k+1} \end{aligned}$$

Next, for a run to make sense we must have $k \geq 1$ and in addition since we're pushing n items we must have $(k+1)^2 + 1 \leq n$ which is $k \leq \sqrt{n-1} - 1$. Thus together we have:

$$1 \leq k \leq \sqrt{n-1} - 1$$

The polynomial part is maximum when $k = \sqrt{n-1} - 1$ and the fractional part is maximum when $k = 1$. Consequently it's certainly true that if the following is true:

$$\beta = \frac{1}{2}\left(\sqrt{n-1} - 1\right) + \frac{11}{4} + \frac{9/4}{2(1)+1} = \frac{1}{2}\left(\sqrt{n-1} - 1\right) + \frac{7}{2}$$

Then the inequality for β is true for all such k. Thus:

$$AC(n) \leq \beta = \frac{1}{2}\left(\sqrt{n-1} - 1\right) + \frac{7}{2} = \mathcal{O}(\sqrt{n})$$

5 Futher Examples

Here is a different but classic example:

Example 5.1. Suppose we have an 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.

Observe that every increment flips $A[0]$, every second increment flips $A[1]$ as well, every fourth increment flips $A[2]$ as well, and so on.

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:

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

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:

$$\begin{aligned}
C(n) &\leq \sum_{k=0}^{\lfloor \lg n \rfloor} n/2^k \\
&\leq n \sum_{k=0}^{\lfloor \lg n \rfloor} \left(\frac{1}{2}\right)^k \\
&\leq n \left(\frac{1 - (1/2)^{1+\lfloor \lg n \rfloor}}{1 - 1/2} \right) \\
&\leq 2n \left(1 - \left(\frac{1}{2}\right)^{1+\lg n} \right)
\end{aligned}$$

The amortized cost then satisfies:

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

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