1 Introduction

Amortized analysis is a way of analyzing how much of a certain resource an algorithm uses. That resource could be time or memory, for example. This is especially useful for algorithms which involve many operations, some of which are more resource-intensive than others. The idea is to figure out, in the worst case, what the average per-operation cost is. This is called the amortized cost.

As an overly simple example suppose we did five operations. The first four cost 1 each and the fifth costs 8. The amortized cost would be:

\[
\frac{1 + 1 + 1 + 1 + 8}{5} = \frac{12}{5} = 2.4
\]

We will revisit amortized analysis when we look at scapegoat trees and hash tables.

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

2 Aggregate Method

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

3 Token Method

The token method of amortized analysis is a little sneakier. We conjecture an average per-operation cost and imagine a bank account. For every operation we put that amount into the account and make a withdrawal as needed. For cheaper operations our account will be growing and for expensive operations we need to ensure that we have enough tokens in the account to cover the operations. This will become more clear with an example.

If that conjectured amount is a variable then we can use this to solve for that variable, as we shall see.

4 An Example

4.1 The Cost Model

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 array 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:

\[
\begin{array}{cccccc}
5 & 0 & 3 & 7 \Rightarrow & 5 & 0 & 3
\end{array}
\]

Push is usually easy:

\[
\begin{array}{ccccccc}
5 & 0 & 3 & 7 \Rightarrow & 5 & 0 & 3 & 7 & 6
\end{array}
\]

But maybe not:

\[
\begin{array}{ccccccccc}
5 & 0 & 3 & 7 & 6 & 2 & 1 \Rightarrow & 5 & 0 & 3 & 7 & 6 & 2 & 1 & 6
\end{array}
\]

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. Thus the worst-case is the successive push of \( n \) elements.

Here are two scenarios:

### 4.2 Increase by Doubling

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.

1. **Aggregate Analysis:**

   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 \( 1 + 1 \).

   \[
   \text{Empty List} \xrightarrow{\text{Push}} \begin{array}{c}
   X
   \end{array}
   \]

   Now we want to push again but we need to reallocate again, so we real-
   locate 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 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.

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:

\[
(1+1)+(2+1)+(4+1)+(0+1)+(8+1)+(0+1)+(0+1)+(0+1)+(16+1)+...+???
\]

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

\[
\underbrace{1 + 1 + ... + 1}_{\text{Push n Times}} + \underbrace{1 + 2 + 4 + ...}_{\text{Reallocate}}
\]

But when does it end? 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 \( k \geq \lceil \lg n \rceil \).

Thus the total cost is:

\[
n + 1 + 2 + 4 + ... + 2^{\lceil \lg n \rceil}
\]

Thus the total cost of doing \( n \) pushes and pops will satisfy:
\[ C \leq n + 1 + 2 + 4 + \ldots + 2^{\lceil \lg n \rceil} \]
\[ \leq n + \left( 2^{1 + \lceil \lg n \rceil} - 1 \right) \]
\[ \leq n + (2^{2 + \lg n} - 1) \]
\[ \leq n + 4 \cdot 2^{\lg n} - 1 \]
\[ \leq n + 4n - 1 \]
\[ \leq 5n - 1 \]

The amortized cost then satisfies:

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

2. Token Analysis:

Suppose that we deposit 5 tokens into an account with each operation. We claim that when we need to reallocate, we will have enough saved up.

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 pops and ends with an expensive reallocation. Our goal is to show that we collect enough tokens during the cheap pushes and pops 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 \( 2^k \) and done a push, meaning there are \( k + 1 \) elements on the stack. The next reallocation will occur at length \( 2^k \), when we reallocate to \( 4^k \) and do a push to \( 2^k + 1 \) elements. Before that happens we will get at least \( k - 1 \) cheap pushes and pops and then one final push over the edge, which is expensive.

The cheap pushes and pops cover their own costs since we collect 5 tokens each but only spend 1, meaning at the end, right before the reallocation plus push we have at least \( 4(k - 1) \) tokens in the bank.

The reallocation plus push collects 5 tokens as well and spends 1 on pushing, meaning we have \( 4k \) tokens in the bank. However the reallocation to \( 4k \) costs exactly this amount, and we have it covered!

The first run can be checked as a special case and the last run, if it’s not of the above type, is simply a list of pushes and pops, which certainly pays for itself. Details are left for the reader.

3. Token Analysis Comment:
We might be upset that the token analysis technique presumed 5 tokens but it doesn’t have to. We could as easily have said that we collect $\beta$ tokens per operation and so at the end of the run we have collected $\beta k$ tokens, spent $k$ tokens, leaving us $\beta k - k$ tokens.

We need to cover the reallocation, so we need:

$$\beta k - k \geq 4k$$

$$\beta k \geq 5k$$

$$\beta \geq 5$$

4.3 Increase by 1

Suppose when we need to reallocate we do so by extending the list length by exactly 1.

1. Aggregate Analysis:

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 pops will satisfy:

$$C \leq \frac{1}{2}n^2 + \frac{3}{2}n$$

The amortized cost then satisfies:

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

2. Token Analysis:

This is a little trickier because we can’t divide the process into runs because a reallocation is happening at every step.
Suppose we deposit $\frac{1}{2} n + \frac{3}{2}$ tokens into an account with each operation. Notice that (especially for large $n$) we are putting a lot of tokens in at the start and not using them, since reallocations are not that expensive early on.

We prove using induction on the push number $1 \leq k \leq n$ that this is sufficient. Note that $n \geq 1$ is assumed to be fixed here.

For $k = 1$ we push $\frac{1}{2} n + \frac{3}{2}$ and the cost is 2. Since $n \geq 1$ we have $\frac{1}{2} n + \frac{3}{2} \geq 2$.

Assume that pushing $\frac{1}{2} n + \frac{3}{2}$ per operation is sufficient for pushes $1, ..., k$.

We claim it is sufficient for push $k + 1$.

Pushes 1 through $k$ will cost us $k$ all together and the associated allocations will cost us:

$$2 + 3 + ... + k + 1$$

Adding these together (a calculation we have seen) results in a total cost of $\frac{1}{2} k^2 + \frac{3}{2} k$ tokens.

We will have collected $k \left(\frac{1}{2} n + \frac{3}{2}\right)$ tokens and so we will have a balance of:

$$k \left(\frac{1}{2} n + \frac{3}{2}\right) - \left(\frac{1}{2} k^2 + \frac{3}{2} k\right) = \frac{1}{2} k(n - k)$$

For push $k + 1$ we collect another $\frac{1}{2} n + \frac{3}{2}$ tokens for a total of:

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

The final push costs $1 + k + 1 = k + 2$.

Observe that since $n \geq k + 1$ then $n - k \geq 1$ and so:

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

And we are done.

4.4 Commentary

The increase by 1 scenario is largely useless because we need to know $n$ beforehand in order to answer the question. This is rarely the case in the real world.

In the real world languages have some pre-defined reallocation method and in all (almost all?) cases they grow the list by some scaling factor, not by a constant, for exactly this reason.
4.5 Increase by Other Amounts

These are just two examples out of many. We could increase by tripling the length when necessary, or by adding a constant amount other than 1, or by squaring the length, and so on.