Dynamic Arrays

A dynamic array automatically doubles its size when you try to make an insertion and there is no more space left. It's a great data structure for implementing a stack, so let's do that. Let's say we have a dynamic array that has these functions:

- `pop()`: return the last item in the dynamic array, and decrement the `last` counter to point to the previous index.
- `push(item)`: increment the `last` index, and put the `item` there. If there's no more space in our array, allocate a new array with double the size and copy over all of our elements.

What is the time cost of m calls to `push()` and `pop()` on our stack?

For **standard worst-case asymptotic analysis**, we would think of it this way: In the worst case we do as much doubling as possible, so all m operations are pushes. What's the time cost of each push? In the worst case for a push we have a doubling. The insertion itself takes constant time, and the doubling costs n time, where n is the number of items in the array. The only bound we can put on the number of items in the array is that it's as many as m (you may be noticing we're overshooting things here—we'll fix that when we use amortized analysis). So we have $O(m)$ time *per insertion*, for $O(m^2)$ time for all of our m operations. Not good.

We can use amortized analysis to show that the runtime is in fact smaller. There are two main methods for amortized analysis: **aggregate analysis** and **the banking method**.

Aggregate analysis involves simply *looking at the total cost of all of the calls to the function in the worst case*, rather than looking at the worst case cost of a given function call, and multiplying by the number of calls.

Here, instead of looking at the worst case for each of the m operations and multiplying by m, let's look at the worst case for all m operations overall.

In the worst case, all of our m operations are pushes. We'll do m work for the actual pushes as well as some additional work for the dynamic array doubling. How much does the doubling cost in total?

Well the first doubling costs 1. The second costs 2. The 3rd costs 4. The 4th costs 8.

"Here comes the log thing!" You may be thinking. Not quite. That's for:

$$1 * 2 * 2 * 2 ...$$

Here we're actually looking at:

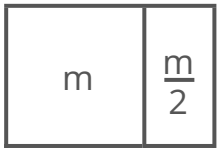
$$1 + 2 + 4 + 8 + ... + n/2 + n$$

Or equivalently:

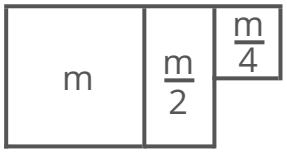
$$n + n/2 + n/4 + ... + 4 + 2 + 1$$

Again we can see what this comes to when we draw it out. If this is m:

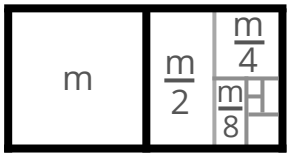
$\frac{m}{2}$ is half the size:



$\frac{m}{4}$ is half the size of that:



And so on:



We see that the whole right-hand side ends up being another square of size m. So our sum is $2m$.

So we could say that the amortized cost of doubling is $2m$, which makes $3m$ when added to the cost of the insertions, which is $O(m)$. So our worst-case amortized cost of m operations is m. Much better than our original estimate of $O(m^2)$

Now let's try **the banking method**. The banking method involves imagining that each computational step has a dollar "cost" and figuring out how much money we need to give each "item" that passes through our system to "pay its way" for the work that's done with it.

Here we would start by saying that each element stored **after the midpoint** holds $2.

This way, when we do a doubling, the work is fully paid for—each element after the midpoint can pay for copying 1) itself and 2) one element before the midpoint.

So in order to maintain this $2 per item after the midpoint, we need to give each item $2 when we push it, as well as paying $1 to put it in. So we have our amortized cost of $3 per push. That's a constant cost per push, giving us $O(m)$ time for our m total operations.

So we can do amortized analysis using either the banking method or aggregate analysis. For this problem we used both approaches to get an $O(m)$ amortized time cost, while classic worst-case asymptotic analysis gave us $O(m^2)$.