Amortized Analysis

In this lecture we discuss a useful form of analysis, called *amortized analysis*, for problems in which one must perform a series of operations, and our goal is to analyze the time per operation. The motivation for amortized analysis is that looking at the worst-case time per operation can be too pessimistic if the only way to produce an expensive operation is to "set it up" with a large number of cheap operations beforehand.

We also discuss the use of a *potential function* which can be a useful aid to performing this type of analysis. A potential function is much like a bank account: if we can take our cheap operations (those whose cost is less than our bound) and put our savings from them in a bank account, use our savings to pay for expensive operations (those whose cost is greater than our bound), and somehow guarantee that our account will never go negative, then we will have proven an *amortized* bound for our procedure.

As in the earlier lectures, in this lecture we will avoid use of asymptotic notation as much as possible, and focus instead on concrete cost models and bounds.

Objectives of this lecture

In this lecture, we want to:

- Understand amortized analysis, how it differs from worst- and average-case analysis.
- See different methods for amortized analysis (aggregate, bankers, potential)
- Practice using the method of *potential functions* for amortized analysis
- See some examples of data structures and their performance using amortized analysis

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 17, Amortized Analysis

1 The ubiquitous example: Dynamic arrays (lists)

Every (good) programming language has an *array* type. An array is usually defined to be a *fixed-size* contiguous sequence of elements. However, it is extremely common for programmers to not know exactly how large an array needs to be up front. Instead, they need something more

general than this, an array where you can increase the size and add new elements over time. Luckily, most programming languages also supply such a container in their standard libraries!

Interface: List

The List API consists of:

- `initialize`(): Create an empty list
- `append`(x): Insert x at the end of the list
- `get`(i): Read and return the i^{th} element of the list

`C++` provides this with `vector`, Java has `ArrayList`, and Python has `list` (the type you get when you write something between square brackets, `[...]`). In this lecture, to avoid ambiguity between this data structure, and a fixed-size array, we'll borrow from Python and refer to this data structure as a *list*. When we say array, we will mean a fixed-size array. So, how would we implement a list type from scratch if all we have access to are regular arrays? One option would be to represent a list of size n as an array of size n, then allocate a new array of size $n+1$ and move over all of the existing elements whenever we want to append a new element, but this would make append cost $O(n)$ time! We'd like to be much much faster than this.

The standard solution is to use *doubling* resizing. We maintain an array of some *capacity* $c \geq 1$ in which the first n of the slots are the actual list elements, and the remaining slots are free space for future elements, so $c \geq n$. When we want to append to the list, we check whether $c > n$, and if so, we can just insert the new element in the next available position, increment n, and we are done. If $c = n$, i.e., the current array is full, we allocate a new array of size $2c$, i.e., double the size! We can then move all of the existing elements over to the new array, throw away the old one, and insert the new element. To make this concrete, lets add another method[1] to our data structure:

- `grow`(): Double the current capacity of the array, moving the elements from the old array into the new array. This costs n since we move n elements.

So, now we can concretely write down our algorithm in terms of this method.

Algorithm 1: Doubling List

We implement the API of List as follows:

- `initialize`(): Create an array with capacity 1. ($n = 0, c = 1$)
- `append`(x): If $c = n$ then `grow`(). Write x at position n and increment n.
- `get`(i): Return the i^{th} element of the array
- `grow`(): Double c and create a new array with capacity c, move every element from the old into the new array.

[1] If you like thinking in terms of Object-Oriented Programming, this is a *private* method, its not available to the user, but we will use it internally and it will help our analysis to separate it out from the rest of the operations.

How fast is this solution? Well, in the best case when $c > n$, we can just insert the new element and be done in $O(1)$ operations. That's pretty great. But in the worst case, we have $c = n$, and we have to spend $O(n)$ time moving all of the n elements over to the new array, which costs $O(n)$. So, in the worst case, this new algorithm still takes $O(n)$ time. Does that mean this is really a bad algorithm, though? Maybe in this case it is not our algorithm that is bad, but our analysis. By considering the worst-case performance for every append, we are really being too pessimistic, because it is impossible for every append to trigger the worst case. After triggering the worst case, it is impossible to trigger another one until we have performed $c/2$ more appends, so we can make our analysis better by considering *an entire sequence* of operations, rather than just thinking about one operation at a time. This is the key idea behind amortized analysis.

Key Idea: Amortized analysis

The key idea of amortized analysis is to consider the worst-case cost of a **sequence of operations** on a data structure, rather than the worst-case individual cost of any particular operation.

This means that amortized analysis is not applicable an isolated run of a single algorithm, e.g., it wouldn't make logical sense to ask about the amortized cost of Quicksort. Rather, we always talk about the amortized cost of a sequence of operations on a data structure.

So now that we have the idea of amortized analysis, how do we actually do it? It turns out that there are several ways, and we will see many of them in this lecture. You may have already seen some of them in your previous classes. The first method that we will analyze is probably the simplest, but the least general. It is called the *aggregate method*, or *total cost method*.

Definition: The aggregate method for amortized analysis

In this method, we will simply add up the **total cost** of performing a sequence of m operations on the data structure, then divide this by the number of operations m, yielding an average cost per operation, which we will define as our amortized cost!

This is the simplest and least general method of analysis because we will end up assigning each operation the same amortized cost. If there are multiple kinds of operations, you could choose to split the cost unevenly, giving different amortized costs to each operation. It is extremely important to not confuse this kind of analysis with *average-case* analysis, even though both involve averages!

Okay, we have to do one last thing before we analyze our algorithm, which is to decide on a cost model. When we were looking at sorting and selection, we used the comparison model which just counts the number of comparisons performed by the algorithms. This new algorithm performs no comparisons, so that would be a rather bad choice for this problem! We will stress that the cost model is always going to be somewhat arbitrary and there is no one correct choice. A good cost model should try to capture the costs of the most important/expensive steps of the algorithm so that it gives as realistic of a prediction of its performance as possible. It should also ideally be as simple as possible so that we don't make our analysis more difficult

than it needs to be. Lets go with the following simple cost model:

- Writing a value to any location in an array costs 1
- Moving a value from one array to another costs 1
- All other operations are free

Now we are ready! Lets use this model and the aggregate method to analyze the algorithm.

Lemma

The cost of a sequence of m append operations using array-doubling is at most $3m$.

Proof. After performing m appends, the number of elements in the list is $n = m$. First, lets count the cost of all of the append operations not including the growing operations (we will account for those separately). The cost of inserting elements is just 1, so in total this costs m.

Now we consider the cost of growing. After m appends, the capacity of the array will be $c = \lceil\lceil m \rceil\rceil$ (this notation means the smallest power of two that is at least m, also called the "super ceiling of m"). Also, notice that $\lceil\lceil m \rceil\rceil \leq 2m$ for any value of m. Now, since the array is capacity c, the most recent grow must have incurred a cost of $c/2$ since the previous capacity was $c/2$ and that many elements were moved from the old array to the new array. Furthermore, the grow operation before that must have cost $c/4$, and so on. So, the costs of the grow operations is

$$1+2+4+\ldots+\frac{\lceil\lceil m \rceil\rceil}{2} \leq 1+2+4+\cdots+m \leq 2m.$$

Therefore in total, the cost of a sequence of m append operations is at most $m+2m=3m$. □

The aggregate method for amortized analysis therefore gives us the following result.

Theorem: The amortized cost of array-doubling lists

The amortized cost (using the aggregate method) of `append` using the array-doubling algorithm is 3

Proof. The lemma tells us that a sequence of m append operations costs at most $3m$, hence each append in the sequence costs at most 3 on average, which is the amortized cost according to the aggregate method. □

2 The Bankers Method

So far so good, but we're going to need a better way to keep track of things for more complex problems. You may have previously heard of the *bankers method* (also known as the *accounting method*, or the *charging method*) for amortized analysis. This is a more general and powerful method of amortized analysis than the aggregate method since it allows us to assign different costs to different operations, but in exchange, it requires more creativity to use.

Definition: The bankers method of amortized analysis

In the bankers method, each operation has an actual cost as specified by the cost model, and additionally may choose to pay an extra cost (a **credit**) to store in the data structure for later use. A future operation may use this credit to offset the cost of an expensive operation. The amortized cost of an operation is its actual cost in the cost model, plus any credit it pays for, minus the value of any previous credit that it consumes.

Now lets see an example of using the bankers method to analyze our list.

Theorem: The amortized cost of array-doubling lists using the bankers method

The amortized cost (using the bankers method) of `append` using the array-doubling algorithm is 3

Proof. We will use the following charging scheme: Whenever we insert a new element into the list, we will pay a credit of 2 and leave it on the list element. Therefore, the cost of an append, not counting the grow operation is 3.

Now consider what happens when a grow operation is triggered. The array must be full ($c = n$), and the final half of the elements ($n/2$ of them) will each possess an unused credit of 2. Therefore, there is a credit of n in the data structure. We consume this credit to pay for the cost of moving the n elements to the freshly allocated array (which costs exactly n), so the amortized cost of the resizing procedure is zero. Therefore, the amortized cost of any append operation, whether it triggers a resizing or not is always 3. □

Thankfully, we get the same amortized cost that we got from the aggregate method, which suggests that the method also makes sense. An important thing that we have to be careful of and keep in mind when using the bankers method is that we must never spend credit that doesn't exist. We must be able to argue that the necessary amount of credit has been placed in data structure by previous operations, and has not already been consumed by a previous operation. In the argument above, we note that we only consume credit when growing occurs, and hence when a grow triggers, the final $n/2$ element must each possess 2 unused credits.

3 The Potential Method

The bankers method is a nice improvement over the aggregate method since it can be used to analyze trickier data structures, or data structures with different costs per operation. We can do even better with an even more general method called the *potential method*. The potential method is the most general, but therefore most difficult-to-use method that we will see for amortized analysis, so we will spend the rest of this lecture working with it.

In the potential method, we define a *potential function* Φ, which maps a *data structure state* S to a real number $\Phi(S)$. We will often use the notation S_i to denote the state of the data structure after applying the i^{th} operation, and S_0 to denote the initial state of the data structure. The

potential function can be thought of as a generalization of the credit in the data structure in the bankers method.

Definition: The potential method for amortized analysis

Consider a sequence of m operations $\sigma_1, \sigma_2, \ldots, \sigma_m$ on the data structure. Let the sequence of states through which the data structure passes be $S_0, S_1, \ldots, S_m$. Notice that operation σ_i changes the state from S_{i-1} to S_i. Let the actual cost of operation σ_i in the cost model be c_i. Given a potential function Φ, we then define the amortized cost $a\,c_i$ of operation σ_i by the following formula:

$$a\,c_i \;=\; c_i + \Phi(S_i) - \Phi(S_{i-1}),$$

In plain English, we can describe the amortized cost as

$$\text{(amortized cost)} \;=\; \text{(actual cost)} + \text{(change in potential)}.$$

Lets compare this to the bankers method for a moment. If an operation pays a credit of p, we can consider that as equivalent to increasing the potential by p. If an operation consumes p credits, we can consider that as the same as decreasing the potential by p. With this correspondence, we can observe that the amortized cost defined by the potential method is the same as the amortized cost defined by the bankers method. This shows that the potential method is indeed a generalization of the bankers method.

Returning to the general potential method, if we sum up the amortized costs of a sequence of operations, we obtain the following formula:

$$\sum_i a\,c_i \;=\; \sum_i (c_i + \Phi(S_i) - \Phi(S_{i-1})) \;=\; \Phi(S_m) - \Phi(s_0) + \sum_i c_i.$$

Note that what happened here is that the terms *telescoped*. The $\Phi(S_i)$ terms all appeared once as a positive and once as a negative, so they all canceled out, except for the first and last terms which each only appeared once. Rearranging we get

$$\sum_i c_i \;=\; \left(\sum_i a\,c_i\right) + \Phi(S_0) - \Phi(S_m).$$

If $\Phi(S_0) \leq \Phi(S_m)$ (as will frequently be the case) we get

$$\sum_i c_i \leq \sum_i a\,c_i.$$

Thus, if we can bound the amortized cost of each of the operations, and the final potential is at least as large as the initial potential, then the total amortized cost is indeed an upper bound on the total actual cost, or, the average actual cost is at most the average amortized cost. It is very common (but not required) to define our potential functions such that $\Phi(S_0) = 0$ and $\Phi(S_i) \geq 0$ for all i. Doing so guarantees that the bound above holds and removes the need for us to do an additional proof that $\Phi(S_m) \geq \Phi(S_0)$ to show that our amortized cost is actually valid.

Most of the art of doing an amortized analysis is in choosing the right potential function. This is typically the hardest part. Once a potential function is chosen we must do two things:

1. Prove that with the chosen potential function, the amortized costs of the operations satisfy the desired bounds.
2. Bound the quantity $\Phi(S_0) - \Phi(S_m)$ appropriately.

4 Lists Revisited

Let's do the analysis of the array-doubling list using the potential method. The hardest part is coming up with a good potential function. This often involves making educated guesses and then iterating with trial and error. There's no real way to guarantee that you'll pick the right one the first time.

Some key ideas to keep in mind are these:

- The goal is to make operations that have an expensive actual cost have a much cheaper amortized cost. So we want to rig the potential so that it *goes down* a lot when an expensive operation occurs, so that the change in potential is negative, making the amortized cost less than the actual cost.
- Operations that were previously cheap will have to get more expensive to compensate. Much like the bankers method, we want each cheap operation to result in the potential *increasing*, so that their amortized cost will be more than their actual cost.

In this case, the expensive operation that we are trying to cheapen is the resizing. If we can identify a property of the data structure that changes drastically when a grow occurs, then we want the potential function to go down according to this property. Since the actual values of the list do not affect the cost, the only properties of the list that matter are the capacity c and the current number of elements n. Here's an observation that we might exploit. As we append more elements, the value of n gets closer to the value of c. When a grow occurs, c gets larger again and this gap widens. So the gap between n and c looks to be just the quantity we want to base our potential on!

Let's start our trial-and-error loop.

- (**First guess**) Our first natural guess for the potential will be

$$\Phi(n, c) = n - c,$$

  since this has the properties discussed above. Unfortunately it has the undesirable property of never being positive. We'd like to stick to our plan of having our potential function never be negative, so we need something different.

- **(Second guess)** We'd still like to have the property that Φ increases when we append a new element, and decreases when we grow, but we want it to be non-negative too. Lets use the fact that since the capacity doubles when $n = c$, we have $n \geq \frac{c}{2}$, and change our potential to

$$\Phi(n, c) = n - \frac{c}{2}.$$

This looks promising. It has the properties we want and appears to always be non-negative[2]. Lets do the analysis and see if it works. Consider an append operation, and as usual, forgo analyzing the cost of a grow initially. The actual cost $ac = 1$ and we increase n by 1, so the potential increases by 1. The amortized cost so far is therefore 2.

Now we consider the cost of a grow. The actual cost $ac = n$, but what about the potential? Well, since we just triggered a grow, it must be true that $n = c$, so $\Phi(S_{i-1})$, i.e., the initial potential, must be $n - \frac{n}{2} = \frac{n}{2}$. After performing the grow, it will now be the case that $c = 2n$, so $\Phi(S_i) = n - \frac{2n}{2} = 0$. Therefore, the potential decreased by $\frac{n}{2}$. The amortized cost of a grow is therefore

$$n + \left(0 - \frac{n}{2}\right) = \frac{n}{2}.$$

Hmmm, so that didn't quite work. It **almost** worked, though. The potential went down by $\frac{n}{2}$, which is proportional to the actual cost of n that we were trying to cancel. Really we just needed the potential to change by twice as much. So, lets correct our potential by making it twice as large!

- (**Third time's a charm**) Using our latest observation, lets define our new potential function to be

$$\Phi(n, c) = 2\left(n - \frac{c}{2}\right).$$

Now we do the analysis again and see what happens. Consider an append operation separate from the cost of any grow. We pay an actual cost of $ac = 1$, and we increase n by 1 which increases the potential by 2. The amortized cost so far is 3.

Now we analyze the grow method. The actual cost is $ac = n$, and since $n = c$, the original potential is $\Phi(S_{i-1}) = 2\left(n - \frac{n}{2}\right) = n$, and the new potential (now that $c = 2n$) is $2\left(n - \frac{2n}{2}\right) = 0$. So the difference in potential is $-n$, and hence the amortized cost of a grow is

$$n + \left(0 - 2\left(\frac{n}{2} - 0\right)\right) = 0.$$

Hooray! Since the amortized cost of a grow is zero, the amortized cost of any append is 3.

Are we done? Not 100%. We still have to check our initial condition on the potential. Remember that we want $\Phi(S_m) \geq \Phi(S_0)$. It turns out that we were wrong about our Φ never being negative, because when we first initialize the data structure, we have $n = 0, c = 1$, which means that $\Phi(0, 1) = -\frac{1}{2}$. Oops. Luckily this doesn't matter at all because it is still true that $\Phi(S_m) \geq \Phi(S_0)$. So we can conclude the following.

Theorem: The amortized cost of array-doubling lists using the potential method

The amortized cost (using the potential method) of `append` using the array-doubling algorithm is 3.

Proof. Choose the potential function $\Phi(n, c) = 2\left(n - \frac{c}{2}\right)$, and observe that, as above, the amortized cost of append is 3, and that $\Phi(S_m) \geq \Phi(S_0)$. □

[2] Except for a little corner case that we'll fix momentarily

5 An Even-more-dynamic Array

A data structure that supports deletes can both grow and shrink in size. It would be nice if the size that it occupies is not *too much* bigger than necessary. This is where a list that shrinks in size is useful. Lets try to design and analyze a data structure that supports the following operations.

- `initialize`(): create an empty list
- `append`(x): insert x at the end of the list
- `get`(i): Read and return the i^{th} element of the list
- `pop`(): remove the last element of the list

As before, we will denote the capacity of the current array by c and the number of actual list elements by n. To support shrinking the list, we will replace our grow operation from earlier with two operations: grow and shrink. `grow`() increases the capacity of the array from c to $2c$, and `shrink`() decreases the capacity of the array from c to $c/2$.

Since our original cost model didn't specify deletions, lets add that in. Our new cost model is now

- Writing a value to any location in an array costs 1
- Moving a value from one array to another costs 1
- Erasing a value from an array costs 1
- All other operations are free

Using these primitives, here's how we implement the interface.

- `initialize`(): create an empty list with capacity 2. ($c = 2$ and $n = 0$)
- `append`(): if $c = n$ then `grow`(). Now insert the element into the table.
- `pop`(): if $n = c/4$ and $c \geq 4$ then `shrink`(). Now erase the element from the table.

There is a little bit of subtlety in this design. The situation immediately after a `grow()` or a `shrink()` is that $n = c/2$. The key thing is that right after one of these expensive operations, the system is very far from from having to do another expensive operation. This allows it time to build up its piggy bank to pay for the next expensive operation.

Lets now try to analyze our growing-and-shrinking array using a potential function. Since the algorithm is quite similar, we would expect a similar potential function to do the trick. As mentioned earlier, one nice observation is that $c/2$ is still the "midpoint" in some sense of the capacity, since we will always have $n = c/2$ immediately following a grow *or* a shrink. So, it would be nice if the potential function still had the property that it was equal to zero when $n = c/2$.

When we perform an append operation, we have the same desire as before. We would like to charge 2 to the potential so that after performing $c/2$ of them, we have saved up c potential

which is enough to pay for the grow. This suggests that as a starting point, our potential should still be $2\left(n-\frac{c}{2}\right)$ whenever $n \geq c/2$, the same as before!

How should we handle pops? They are a little different. If we start at half capacity right after a grow or shrink ($n = c/2$), it only takes $c/4$ pops to trigger a shrink, not $c/2$ like was the case for append. However, note that the shrink operation only has to move $c/4$ elements to the newly shrunk array (because $n = c/4$ when a shrink is triggered). So unlike grow, which requires each of its appends to pay for 2 moves, shrink only needs a 1 to 1 charge for each pop. This suggests that we do not need a constant of 2 for pop/shrink! This suggests the following potential function.

$$\Phi(n,c) = \begin{cases} 2\left(n-\frac{c}{2}\right), & \text{if } n \geq \frac{c}{2}, \\ \frac{c}{2}-n & \text{if } n < \frac{c}{2}. \end{cases}$$

The first case handles the situation where the array is in a "growing state", it is currently larger than it was after the most recent resize, and it is heading towards needing a grow operation. The second case handles the situation where it is in a "shrinking state", where it is smaller than it was after the last resize, and it is heading towards a shrink operation.

Theorem 1: The amortized cost of the growing-shrinking list

Using the doubling-halving array data structure, the amortized cost of `append` is at most 3, the amortized cost of `pop` is at most 2, and the amortized cost of `initialize` is at most 1.

Proof. As usual, lets first consider the cost of an append operation without accounting for a grow (yet). The actual cost is 1, but what about the change in potential? Well now our potential function has two cases so we have to consider two cases. If $n \geq c/2$, then the math is the same as before. n increases by 1, so the potential increases by 2. If $n < c/2$, then the potential actually decreases by 1. Therefore the worst-case potential increase is 2, and hence the amortized cost of append (not including grow) is at most 3.

Now we account for grow. This is again the same as the doubling array. Before growing, we have $n = c$ and hence the potential is n. After growing the potential is zero, so the potential has dropped by n, which is exactly the cost of moving the n elements to the new array, and hence the amortized cost of grow is zero.

Now lets look at pop and shrink. First, consider the cost of pop separately to the cost of shrink. Erasing the element from the array costs 1, and n decreases by 1. How does this affect the potential? If $n \geq c/2$, then the potential goes down by 2. If $n < c/2$, then the potential increases by 1. So, in the worst-case, the potential increase is 1, and hence the amortized cost of pop (not including shrink) is at most 2.

Now we account for shrink. Before a shrink occurs, $c = 4n$, and hence the initial potential is n. After a shrink occurs, the potential is zero, so the potential has decreased by n, which is exactly the cost of moving the n elements to the new array. Therefore, the amortized cost of shrink is exactly zero.

Lastly, we should consider the final and initial potential. $\Phi(S_0) = 1$ and $\Phi(S_m) \geq 0$, so $\Phi(S_0) - \Phi(S_m) \leq 1$. We could modify our potential function to fix this, but I'm just going to charge that 1 to be the cost of initialize for simplicity. □

Exercises: Amortized Analysis

Problem 1. Describe the difference between amortized analysis using the aggregate method, and average-case analysis. How are they different? What about *expected* cost analysis? Where does this fit in and how is it different from the first two? Make sure you understand the differences!

Problem 2. Give a proof for Theorem 1 using the banker's method. Where would you put the banker's tokens in the data structure?

Problem 3. Suppose we change `pop()` to shrink when $s = n/2$, show a sequence of operations that incur large amortized cost.

Problem 4. Modify the potential function used in the proof of Theorem 1 so that we do not need to charge 1 to the cost of initialization (in other words, change the potential so that the initial potential is zero but the proof still works).