7.1 Introduction

So far we have mostly been considering static problems, where we are given an input and need to design an algorithm to compute some output. But many problems, particularly in data structures are more dynamic – we want to consider what happens over a sequence of operations. Consider the difference, for example, between being given n items up front and asked to compute on them (sort, find median, etc.) and being given n operations on a dictionary (e.g. insert, lookup, or remove). When we analyze data structures we'll want to make operations such as those efficient.

Today we're going to talk about a type of analysis, *amortized analysis*, which is particularly well-suited to these kinds of problems. The basic definition is actually quite simple:

Definition 7.1.1 *The* amortized cost *of a sequence of n operations is the total cost divided by n.*

In other words, it's the *average cost per operation* of a sequence of operations. But don't be deceived by the word "average" – there is no probability here. When we design algorithms and bound the amortized cost, we will consider the worst-case sequence of n operations.

For example, suppose that we have 100 operations of cost 1 and then 1 operation of cost 100. The normal worst-case analysis would say that the final operation costs 100, but this obscures the fact that the average cost of the sequence is only $200/101 \approx 2$. And if in fact the *only* way that the expensive operation could happen is for it to follow a lot of cheap operations, then in some cases it doesn't make sense to look at the single most expensive operation. That is overly pessimistic if what we care about is the total time spent performing operations. Amortized analysis is an attempt to fix this.

If we can prove that the amortized cost of *any* sequence is small, then we say that our algorithm has bounded amortized complexity.

Definition 7.1.2 *If the amortized cost of any sequence of operations is $f(n)$, then the* amortized cost *or* amortized complexity *of the algorithm is $f(n)$.*

We will usually (but not always) use amortized analysis with respect to time, in which case we will talk about the *amortized time complexity* or the *amortized time*.

While the definition of amortized analysis is pretty straightforward, the analysis can end up being quite complex. We'll do a few examples of various levels of complexity. The book goes into much more detail, as does the homework.

7.2 Example: Using an array for a stack

This is probably the most straightforward example, so is a classic. Suppose that we want a stack, but we want to implement it using an array. Recall that in a stack, we have the ability to push

new elements into the stack, and the ability to pop off the most recently added element from the stack (last in first out). Note that normally doing this with an array would be crazy (you should use a linked list), but sometimes it's nice to have the extra random access ability of an array. So we'll have an array `A`, and we'll keep track of an index `top` into the array. Initially `top` is 0. When we push onto the stack we add the element to `A[top]` and then increment `top`, and when we pop we decrement `top` and then remove the element at `A[top]`.

This works out well to start with, but arrays have a fixed length. What happens if we try to push more elements onto it than the length of the array? Well, clearly we need to increase the size of the array. But this requires initializing a new array and copying over every single element! So if the current length of the array is n, it takes time n to create a new, larger array (let's say that initializing is free because we don't need to initialize entries to 0, each copy/move costs 1, and each access to `top` costs 1).

So what happens if we do the obvious thing, and create a new array of size $n+1$? Well, if we start with a small array (say size 1, but it doesn't really matter) and then do n push operations, the ith push takes time i. So the total time is

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

So the worst single operation takes time n, and the amortized cost is $\Theta(n)$. This isn't very good.

What about a different strategy: when the array fills up, we double its size. With this strategy, a single operation can still take time n, since if we double from n to $2n$ that takes time n. But what's the amortized time? If we start with an array of size 1, then we have to double when the input gets to size $2, 4, 8, 16, \ldots, \lfloor \log n \rfloor$. Since the cost of doubling is the current size, the total amount of time spent doubling is only at most

$$\sum_{i=1}^{\lfloor \log n \rfloor} 2^i \leq 2n = O(n).$$

(Note that if some of the n operations are pops then our cost might be significantly less).

On any push or pop where we do not double the size the time is 1. Hence the total time on operations which do not require doubling is only $O(n)$. Thus the total time for *all* n operations is only $O(n)$, since we spend $O(n)$ time doubling and $O(n)$ time doing everything else. Now by definition, this means that the amortized cost of push/pop is $O(1)$, despite the fact that any *single* operation might still take time $\Omega(n)$.

7.3 More complicated analysis: potential functions and piggy banks

For the stack it was easy to just analyze the total cost and divide by the number of operations. Sometimes this is not easy. A more complicated way of doing this, but which sometimes turns out to be easier to think about, is to have a piggy bank to help with the accounting.

The basic idea is to have a bank of tokens – let's call this bank L. When we have inexpensive operations, we'll pay not only the cost/time for them, but also add some extra tokens to the bank. Then for expensive operations, we can pull some tokens out of the bank to help offset the cost of the operation. Informally, we're trying to "smooth out" the cost – we charge the cheap operations more than they really cost (by requiring them to deposit in the bank), but then charge expensive operations less than they cost (by pulling tokens out of the bank). As long as there are always enough tokens in the bank, this works great.

Let's see this idea a little more formally. Suppose that we initially set our bank $L = 0$, and when we do operation i we might also take some tokens from or deposit some tokens into the bank. Let c_i be the cost of the ith operation, and let L_{i-1} be the amount in the bank before the ith operation and let L_i be the amount after the ith operation. We will define new amortized costs

$$c'_i = c_i + \Delta L = c_i + L_i - L_{i-1}$$

Then the total cost of n operations is

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \left(c'_i + L_{i-1} - L_i\right) = \sum_{i=1}^{n} c'_i + \sum_{i=1}^{n} \left(L_{i-1} - L_i\right) = \sum_{i=1}^{n} c'_i + L_0 - L_n.$$

So if we set $L_0 = 0$ and make sure the bank never goes negative (or even just make sure that L_n is nonnegative), we get that $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} c'_i$. So if we can upper bound the c'_i's, we will have a valid upper bound for the amortized cost. This will let us "smooth out" the analysis – sometimes it is hard to analyze the c_i's directly, since sequences of operations can be tricky, but if we can show that $c'_i \leq f(n)$ for all c'_i then this will imply that the amortized cost is at most $f(n)$.

A few notes. First, this is sometimes called the "potential method" – instead of thinking of L as a piggy bank, we think of it as potential energy. Then some operations increase the potential, some decrease it, but as long as it stays nonnegative we're good. And sometimes some people find it easier to think about a potential function which is simply a function of the state of the system, rather than explicitly arguing about which operations deposit or take which amounts of money. It's all the same in the end, though.

Similarly, sometimes it's easier to think of having multiple piggy banks. If we're doing operations on m elements, sometimes it's useful to think of each element as having a bank, and operations take or leave tokens in the banks of the elements they touch. But this is also the same in the end, since we can just define a global bank to be the sum of the individual banks.

7.4 Example 2: Binary counter

Now let's do an example where we can see this in action. Suppose we want a binary counter, which we store in some big array A. For now let's only worry about incrementing. So each element of A starts out as 0 and is always either 0 or 1. We'll use the convention that $A[0]$ is the least significant bit, then $A[1]$, etc. Let's not worry about the length of the array – it's large enough to hold any number that we're going to care about.

Suppose that it costs 1 every time that we flip a bit. If we do n increments, then some increment might have to flip $\log n$ bits. So the worst-case cost of an increment is $\Theta(\log n)$. But what about the amortized cost?

Let's think of each bit (i.e. each element of A) as having its own bank. They all start out at 0. When we do an increment that flips $A[i]$ from 0 to 1, we put an extra token in the the bank for bit i. When we flip a bit from 1 to 0, we use the token that's on the bit to pay for it. Clearly no bank is ever negative – whenever we want to use a token to pay for a flip from 1 to 0, that token is there. Now note that even though each increment might have to flip a different number of bits, each increment only flips a single bit from 0 to 1. So the only thing we're charged for when we do an increment is flipping a single bit to 1, and we charge ourselves 2 for that. Thus the amortized cost of an increment is only 2, rather than $\log n$.

Alternatively, we can think of a global bank L rather than a local one. Then we deposit an extra token into this bank whenever we flip a 0 to a 1, and take a token out whenever we flip a 1 to a 0. Just as with local banks, this never goes negative and it is always the case that $c'_i = c_i + L_i - L_{i-1} = 2$. Thus the amortized cost is only 2.

A third way of analyzing this is with a potential function (traditionally denoted Φ). We let Φ be the number of 1's in the counter. Then $\Phi_0 = 0$, and clearly it never goes negative. And just as before $c'_i = c_i + \Phi_i - \Phi_{i-1} = 2$. So again we get an amortized cost of 2.

Finally, we could analyze it directly. $A[0]$ is flipped every time, $A[1]$ is flipped every 2nd time, $A[2]$ is flipped every $2^2 = 4$th time, etc. Clearly $A[i]$ is flipped every 2^i times, so with n increments it is flipped $n/2^i$ times. Thus the total number of bit flips (i.e. the total cost) is at most $\sum_{i=0}^{\log n} \frac{n}{2^i} \leq 2n$, so the amortized cost is 2.

7.5 Example 3: Simple dictionary

Our last example will be a simple dictionary data structure, where we want to be able to insert and lookup a set of items. This is the same setting as last lecture, but now let's try to do something without trees. As we discussed, keeping a sorted array is great for lookups (just need to do a binary search) but terrible for inserts, since each insert might require moving almost all of the elements down the array. So the ith insert might cost i, and so for n operations the total cost could be $\Omega(n^2)$ and thus the amortized cost would be $\Omega(n)$.

As with our previous example, we'll have an array A. But rather than $A[i]$ being a bit or a single item, it is now going to be either empty or an array of exactly 2^i items. Each nonempty array is sorted, but there is no relation between the arrays. For example, if we've inserted the numbers 1 to 11 we might have something like

$$A[0] = [5]$$
$$A[1] = [2, 8]$$
$$A[2] = \emptyset$$
$$A[3] = [1, 3, 4, 6, 7, 9, 10, 11]$$

Note that after n inserts, there are at most $\log n$ arrays. Lookups are easy, since each array is

sorted – we simply do a binary search in each nonempty array. This takes time at most $\log n + \log(n/2) + \log(n/4) + \cdots + 1 = \Theta(\log^2 n)$.

But what about inserts? Consider the following algorithm for inserting an item x. We first create an array B of size 1 containing x. If $A[0]$ is empty then we set it equal to B, and are done. Otherwise, we merge $A[0]$ and $[x]$ to get a new array of size 2, set B equal to this array, and set $A[0]$ to empty. We now recurse to the next entry in A: if $A[1]$ is empty we set it equal to B and are done, otherwise we merge it with B to get an array of size 4, set $A[1]$ to empty, and recurse on $A[2]$. This continues until at some point we have a list of length 2^i and $A[i]$ is empty.

As an example, let's take the previous example and try to insert 12. We first create an array $B = [12]$. Since $A[0]$ is not empty we do a merge, and get that $B = [5, 12]$. Now $A[1]$ is nonempty so we do another merge, to get $[2, 5, 8, 12]$. Now $A[2]$ is empty, so we can put the array there. So in the end we end up with

$$\begin{aligned} A[0] &= \emptyset \\ A[1] &= \emptyset \\ A[2] &= [2, 5, 8, 12] \\ A[3] &= [1, 3, 4, 6, 7, 9, 10, 11] \end{aligned}$$

To be clear on the costs, let's say that merging two arrays of length m costs $2m$ (as discussed in the homework it really takes $2m - 1$, but since we're only looking for an upper bound we'll ignore the -1, and it's a lower order term anyway), and creating the array of size 1 costs 1. Then a single insert might be very expensive, since all entries of A might be full so we need to do lots of merges. In particular, consider the nth insert. The cost might be as large as $\sum_{i=0}^{\log n} 2 \cdot 2^i = \Theta(n)$, so it doesn't seem like we've gained much compared to using a single array. But what about the amortized cost?

We can analyze this directly – it's actually just like the binary counter, but where the cost of flipping the ith bit is 2^i instead of one! We do a merge of arrays of length 2^i one out of every 2^i steps (think of this as flipping the ith bit). So in n operations, we will have merged lists of length 1 at most n times, we will have merged lists of length 2 at most $n/2$ times, merged lists of length 4 at most $n/4$ times, etc. So the total cost is at most

$$\sum_{i=0}^{\log n} \frac{n}{2^i} \cdot 2^{i+1} = \Theta(n \log n)$$

Thus the amortized cost of an insert is only $O(\log n)$.

7.6 Multiple Operations

We'll talk more about this in future lecture, but it's important to be a little careful when applying amortized analysis to data structures that support multiple operations (which is what we'll usually be doing!). We said that fundamentally, the amortized cost of a series of m operations is the total cost of all of them divided by m. But what does this means when we have different types of

operations? For example, suppose that we have k different types of operations on the same data structure. If we say that "operations of type i have amortized cost α_i", what this means is that the total cost of any sequence which performs m_i operations of type i is $\sum_{i=1}^{k} m_i \alpha_i$. This is the obvious generalization of our definition for one operation, where we said that the amortized cost was the total cost divided by m. These kinds of guarantees make it much harder to prove anything directly about the total cost of the sequence, so we will almost always use accounting or potential arguments and argue that the "amortized cost" defined to be the true cost plus the change in potential/bank is small.

A couple quick notes about this:

- When analyzing multiple operations with accounting or potential arguments, you must use the same scheme for all operations! You cannot define a different potential function for each operation — you have to be consistent.
- Amortized time bounds are not necessarily unique when data structures support multiple operations. Different amortization schemes can assign different costs to exactly the same algorithms. For example, consider a generic data structure that can be modified by three algorithms: `Fold`, `Spindle`, and `Mutilate`. One amortization scheme might imply that `Fold` and `Spindle` each run in $O(\log n)$ amortized time, while `Mutilate` runs in $O(n)$ amortized time. Another scheme might imply that `Fold` runs in $O(\sqrt{n})$ amortized time, while `Spindle` and `Mutilate` each run in $O(1)$ amortized time. These two results are not necessarily inconsistent! Moreover, there is no general reason to prefer one of these sets of amortized time bounds over the other; our preference may depend on the context in which the data structure is used.[1]

[1] This discussion is essentially verbatim from Jeff Erickson's lecture notes