Lecture 7

Amortized Analysis

7.1 Overview

This lecture discusses 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 introduce the notion 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 previous lecture, in this lecture we will avoid use of asymptotic notation as much as possible, and focus instead on concrete cost models and bounds.

7.2 Introduction

So far we have been looking at static problems where you are given an input (like an array of n objects) and the goal is to produce an output with some desired property (e.g., the same objects, but sorted). For next few lectures, we're going to turn to problems where we have a *series* of operations, and goal is to analyze the time taken per operation. For example, rather than being given a set of n items up front, we might have a series of n insert, lookup, and remove requests to some database, and we want these operations to be efficient.

Today, we'll talk about a useful kind of analysis, called *amortized analysis* for problems of this sort. The definition of amortized cost is actually quite simple:

Definition 7.1 *The* **amortized cost** *of n operations is the total cost of the operations divided by n.*

Analyzing the amortized cost, however, will often require some thought if we want to do it well.

We will illustrate how this can be done through several examples.

7.3 Example #1: implementing a stack as an array

Say we want to use an array to implement a stack. We have an array `A`, with a variable `top` that points to the top of the stack (so `A[top]` is the next free cell). This is pretty easy:

- To implement `push(x)`, we just need to perform:

  ```
  A[top] = x;

  top++;
  ```

- To implement `x=pop()`, we just need to perform:

  ```
  top--;

  x = A[top];
  ```

  (first checking to see if `top==0` of course...)

However, what if the array is full and we need to push a new element on? In that case we can allocate a new larger array, copy the old one over, and then go on from there. This is going to be an expensive operation, so a push that requires us to do this is going to cost a lot. But maybe we can "amortize" the cost over the previous cheap operations that got us to this point. So, on average over the sequence of operations, we're not paying too much. To be specific, let us define the following cost model.

Cost model: Say that inserting into the array costs 1, taking an element out of the array costs 1, and the cost of resizing the array is the number of elements moved. (Say that all other operations, like incrementing or decrementing "top", are free.)

Question 1: What if when we resize we just increase the size by 1. Is that a good idea?

Answer 1: Not really. If our n operations consist of n pushes then even just considering the array-resizing cost we will incur a total cost of at least $1 + 2 + 3 + 4 + \ldots + (n-1) = n(n-1)/2$. That's an amortized cost of $(n-1)/2$ per operation just in resizing.

Question 2: What if instead we decide to double the size of the array when we resize?

Answer 2: This is much better. Now, in any sequence of n operations, the total cost for resizing is $1 + 2 + 4 + 8 + \ldots + 2^i$ for some $2^i < n$ (if all operations are pushes then 2^i will be the largest power of 2 less than n). This sum is at most $2n - 1$. Adding in the additional cost of n for inserting/removing, we get a total cost $< 3n$, and so our amortized cost per operation is < 3.

7.4 Piggy banks and potential functions

Here is another way to analyze the process of doubling the array in the above example. Say that every time we perform a push operation, we pay \$1 to perform it, and we put \$2 into a piggy bank. So, our out-of-pocket cost per push is \$3. Any time we need to double the array, from size L to $2L$, we pay for it using money in the bank. How do we know there will be enough money (\$$L$) in the bank to pay for it? Because after the last resizing, there were only $L/2$ elements in the array and so there must have been *at least* $L/2$ new pushes since then contributing \$2 each. So, we can pay for everything by using an out-of-pocket cost of at most \$3 per operation. Putting it another way, by spending \$3 per operation, we were able to pay for all the operations plus possibly still have money left over in the bank. This means our amortized cost is at most 3.[1]

This "piggy bank" method is often very useful for performing amortized analysis. The piggy bank is also called a *potential function*, since it is like potential energy that you can use later. The potential function is the amount of money in the bank. In the case above, the potential is 2 times the number of elements in the array after the midpoint. *Note that it is very important in this analysis to prove that the bank account doesn't go negative.* Otherwise, if the bank account can slowly drift off to negative infinity, the whole proof breaks down.

Definition 7.2 *A* **potential function** *is a function of the state of a system, that generally should be non-negative and start at 0, and is used to smooth out analysis of some algorithm or process.*

Observation: If the potential is non-negative and starts at 0, and at each step the actual cost of our algorithm plus the change in potential is at most c, then after n steps our total cost is at most cn. That is just the same thing we were saying about the piggy bank: our total cost for the n operations is just our total out of pocket cost minus the amount in the bank at the end.

Sometimes one may need in an analysis to "seed" the bank account with some initial positive amount for everything to go through. In that case, the kind of statement one would show is that the total cost for n operations is at most cn plus the initial seed amount.

Recap: The motivation for amortized analysis is that a worst-case-per-operation analysis can give overly pessimistic bound if the only way of having an expensive operation is to have a lot of cheap ones before it. Note that this is *different* from our usual notion of "average case analysis": we are not making any assumptions about the inputs being chosen at random, we are just averaging over time.

7.5 Example #2: a binary counter

Imagine we want to store a big binary counter in an array A. All the entries start at 0 and at each step we will be simply incrementing the counter. Let's say our cost model is: whenever we increment the counter, we pay \$1 for every bit we need to flip. (So, think of the counter as an

[1] In fact, if you think about it, we can pay for pop operations using money from the bank too, and even have \$1 left over. So as a more refined analysis, our amortized cost is \$3 per push and \$$-1$ per successful pop (a pop from a nonempty stack).

array of heavy stone tablets, each with a “0” on one side and a “1” on the other.) For instance, here is a trace of the first few operations and their cost:

`A[m]`	`A[m-1]`	...	`A[3]`	`A[2]`	`A[1]`	`A[0]`	**cost**
0	0	...	0	0	0	0	
							$1
0	0	...	0	0	0	1	
							$2
0	0	...	0	0	1	0	
							$1
0	0	...	0	0	1	1	
							$3
0	0	...	0	1	0	0	
							$1
0	0	...	0	1	0	1	
							$2

In a sequence of n increments, the worst-case cost per increment is $O(\log n)$, since at worst we flip $\lg(n)+1$ bits. But, what is our *amortized* cost per increment? The answer is it is at most 2. Here are two proofs.

Proof 1: Every time you flip $0 \to 1$, pay the actual cost of \$1, plus put \$1 into a piggy bank. So the total amount spent is \$2. In fact, think of each bit as having its own bank (so when you turn the stone tablet from 0 to 1, you put a \$1 coin on top of it). Now, every time you flip a $1 \to 0$, use the money in the bank (or on top of the tablet) to pay for the flip. Clearly, by design, our bank account cannot go negative. The key point now is that even though different increments can have different numbers of $1 \to 0$ flips, each increment has exactly one $0 \to 1$ flip. So, we just pay \$2 (amortized) per increment.

Equivalently, what we are doing in this proof is using a potential function that is equal to the number of 1-bits in the current count. Notice how the bank-account/potential-function allows us to smooth out our payments, making the cost easier to analyze.

Proof 2: Here is another way to analyze the amortized cost. First, how often do we flip `A[0]`? Answer: every time. How often do we flip `A[1]`? Answer: every other time. How often do we flip `A[2]`? Answer: every 4th time, and so on. So, the total cost spent on flipping `A[0]` is n, the total cost spent flipping `A[1]` is $n/2$, the total cost flipping `A[2]` is $n/4$, etc. Summing these up, the total cost spent flipping all the positions in our n increments is at most $2n$.

7.6 Example #3: What if it costs us 2^k to flip the kth bit?

Imagine a version of the counter we just discussed in which it costs 2^k to flip the bit `A[k]`. (Suspend disbelief for now — we’ll see shortly why this is interesting to consider). Now, in a sequence of n increments, a single increment could cost as much as n, but the claim is the amortized cost is only $O(\log n)$ per increment. This is probably easiest to see by the method of “Proof 2” above: `A[0]` gets flipped every time for cost of \$1 each (a total of \$$n$). `A[1]` gets flipped every other time for cost of \$2 each (a total of at most \$$n$). `A[2]` gets flipped every 4th time for cost of \$4 each (again, a total of at most \$$n$), and so on up to `A[`$\lfloor \lg n \rfloor$`]` which gets flipped once for a cost at most \$$n$. So, the total cost is $O(n \log n)$, which is $O(\log n)$ amortized per increment.

7.7 Example #4: A simple amortized dictionary data structure

One of the most common classes of data structures are the "dictionary" data structures that support fast insert and lookup operations into a set of items. In the next lecture we will look at balanced-tree data structures for this problem in which both inserts and lookups can be done with cost only $O(\log n)$ each. Note that a sorted array is good for lookups (binary search takes time only $O(\log n)$) but bad for inserts (they can take linear time), and a linked list is good for inserts (can do them in constant time) but bad for lookups (they can take linear time). Here is a method that is very simple and *almost* as good as the ones in the next lecture. This method has $O(\log^2 n)$ search time and $O(\log n)$ amortized cost per insert.

The idea of this data structure is as follows. We will have a collection of arrays, where array i has size 2^i. Each array is either empty or full, and each is in sorted order. However, there will be no relationship between the items in different arrays. The issue of which arrays are full and which are empty is based on the binary representation of the number of items we are storing. For example, if we had 11 items (where $11 = 1 + 2 + 8$), the data structure might look like this:

```
A0:  [5]
A1:  [4,8]
A2:  empty
A3:  [2, 6, 9, 12, 13, 16, 20, 25]
```

To perform a lookup, we just do binary search in each occupied array. In the worst case, this takes time $O(\log(n/2) + \log(n/4) + \log(n/8) + \ldots + 1) = O(\log^2 n)$.

What about inserts? We'll do this like mergesort. To insert the number 10, we first create an array of size 1 that just has this single number in it. We now look to see if `A0` is empty. If so we make this be `A0`. If not (like in the above example) we merge our array with `A0` to create a new array (which in the above case would now be `[5, 10]`) and look to see if `A1` is empty. If `A1` is empty, we make this be `A1`. If not (like in the above example) we merge this with `A1` to create a new array and check to see if `A2` is empty, and so on. So, inserting 10 in the example above, we now have:

```
A0:  empty
A1:  empty
A2:  [4, 5, 8, 10]
A3:  [2, 6, 9, 12, 13, 16, 20, 25]
```

Cost model: To be clear about costs, let's say that creating the initial array of size 1 costs 1, and merging two arrays of size m costs $2m$ (remember, merging sorted arrays can be done in linear time). So, the above insert had cost $1 + 2 + 4$.

For instance, if we insert again, we just put the new item into `A0` at cost 1. If we insert again, we merge the new array with `A0` and put the result into `A1` at a cost of $1 + 2$.

Claim 7.1 *The above data structure has amortized cost $O(\log n)$ per insert.*

Proof: With the cost model defined above, it's exactly the same as the binary counter with cost 2^k for counter k. ■