15-451/651: Design & Analysis of Algorithms January 24, 2023
Lecture #3: Amortized Analysis last changed: January 27, 2023

1 Introduction

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 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.

Informally *amortized cost of an operation* in a sequence of operations is the total cost of all of them divided by the number of operations (i.e. the average cost of an operation). We begin by presenting two examples: the binary counter, and growing a table. In these examples the operations have large worst-case cost, but constant amortized cost. We then introduce potential functions and show how these same two examples can be analyzed using potentials. Finally we will apply the potential function method to analyze the problem of a table that both grows and shrinks.

2 First Example: 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 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 is a proof:

Proof: 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 at most $n/2$, the total cost flipping A[2] is at most $n/4$, etc. Summing these up, the total cost spent flipping all the positions in our n increments is at most $2n$. So if we distribute this cost evenly over all n operations we see that the average cost is at most 2 per operation.

2.1 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 $2n$ (actually $2n-1$), but the claim is the amortized cost is only $O(\log n)$ per increment. We can prove this by extending the idea 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 at most $n(\lg n + 1)$, which is $O(\log n)$ amortized per increment.

2.2 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 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 achieves part of what the tree methods do. 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 $11 = 1 + 2 + 8$), then the arrays of size 1, 2, and 8 would be full and the rest empty, and 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 $O(\log(n) + \log(n/2) + \log(n/4) + \ldots + 1) = O(\log^2 n)$.

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 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. □

3 Second Example: Growing a Table

A common problem in data structure design arises if you're using an array (which we'll call a table) to store something (a stack, a hash table, a `vector` in C++, etc.), and you find out that you need more space. The rule of thumb in these cases is to double the size of the table. This is expensive because you have to allocate a lot of space, and move all the data over to the new area. Despite this, we'll see that the amortized cost is $O(1)$.

We can set up a framework for analyzing this problem as follows. At any point in time the size of the table is denoted by n, and the number of elements used in the table is denoted by s, with $s \leq n$. The following API defines the way the client will use the table.

`initialize():`	create a new table of size 1 with nothing in it. ($n = 1$ and $s = 0$)
`insert():`	add a new element to the table. (increment s)

It will be useful to define the following operation:

`grow():` Double the size of the table from n to $2n$. The cost of this operation is $2n$, the new size of the table.[1] (This cost pays for allocating the new table and moving all of the data from the old table to the new one.)

Now `insert()` will be implemented as follows:

`insert():` If $s = n$ then `grow()`. Now put the new element into the table at a cost of 1. (increment s)

After an `initialize()`, what is the cost of a sequence of m inserts into the table? Let's let N denote the size of the table at the end of this process. The total work of all the `insert()` operations, not counting the `grow()` costs is m. The total work of all the `grow()` operations is $2 + 4 + 8 + \cdots + N < 2N$. But $N/2 < m$, because otherwise the table would never have grown to size N. So the total cost is at most $m + 4m = 5m$. So we can say that the amortized cost of an operation is at most 5.

4 Potentials

So far so good, but we're going to need a better way to keep track of things for more complex problems. And the way to do that is with *potential functions*, as we show in this section. But first, here's what you might call the Banker's Proof of the amortized bound of 2 on the cost of incrementing a binary number.

[1] These costs, and the ones that follow in the rest of these notes, are specified here for concreteness and clarity of analysis. All of this is somewhat arbitrary. Changes in cost by a constant factor will only effect the final result by a constant factor.

Banker's Proof of Binary Counter: 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. $\square$

Banker's proofs, like this one, involve placing the money *at a specific location in the data structure.* So when a change occurs at that location, there is money right there to use to pay for the work.

Equivalently, what we are doing in this proof is using a "potential function" that equals 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.

This technique can be applied in a much more general way. The idea is to make a rule that says how much money must be kept in the bank as a function of the state of the data structure. Then a bound is obtained on how much money is required to pay for an operation and maintain the appropriate amount of money in the bank.

The *physicist's view* of amortization uses different terminology to describe the same idea. A *potential function* $\Phi(s)$ is a mapping from data-structure states to the reals. This takes the place of the bank account in the banker's view. Both methods are useful, and will be used in this course. Sometimes one method affords more intuition than the other.

Consider a sequence of n operations $\sigma_1, \sigma_2, \ldots, \sigma_n$ on the data structure. Let the sequence of states through which the data structure passes be $s_0, s_1, \ldots, s_n$. Notice that operation σ_i changes the state from s_{i-1} to s_i. Let the cost of operation σ_i be c_i. Define the amortized cost ac_i of operation σ_i by the following formula:

$$ac_i \ = \ c_i + \Phi(s_i) - \Phi(s_{i-1}), \tag{1}$$

or

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

If we sum both sides of this equation over all the operations, we obtain the following formula:

$$\sum_i ac_i \ = \ \sum_i (c_i + \Phi(s_i) - \Phi(s_{i-1})) \ = \ \Phi(s_n) - \Phi(s_0) + \sum_i c_i.$$

Rearranging we get

$$\sum_i c_i \ = \ \left(\sum_i ac_i\right) + \Phi(s_0) - \Phi(s_n). \tag{2}$$

If $\Phi(s_0) \le \Phi(s_n)$ (as will frequently be the case) we get

$$\sum_i c_i \le \sum_i ac_i. \tag{3}$$

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 bound we obtained for the amortized cost applies to the actual cost.

Potential Function for Binary Counter: We can now apply this technique to the problem of computing the cost of binary counting. Let the potential Φ be the number of 1's in the current number. Our first goal is to show that with this potential the amortized cost of an increment operation is 2.

Consider the ith increment operation that changes the number from $i-1$ to i. Let k be the number of carries that occur as a result of the increment. The cost of the operation is $k+1$. The change in potential caused by the operation is $-k+1$. (The number of bits that change from 1 to 0 is k and one bit changes from 0 to 1.) Therefore the amortized cost of the operation is

$$ac_i = k + 1 + (-k+1) = 2.$$

Since the final potential is more than the initial potential, we can apply inequality (3) to obtain:

$$\sum_i c_i \leq \sum_i ac_i = 2n.$$

Notice that the definition of the amortized cost of an operation depends on the choice of the potential function Φ. Any choice of potential function whatsoever defines an amortized cost of each operation. However, these amortized bounds will not be useful unless $\Phi(s_0) - \Phi(s_n)$ is also bounded appropriately.

We have given two different definitions of amortized cost, the informal one in Section 1, and another in equation (1). In the context of a potential function the amortized cost is that defined by equation (1). Otherwise, it's the informal intrpretaton of Section 1.

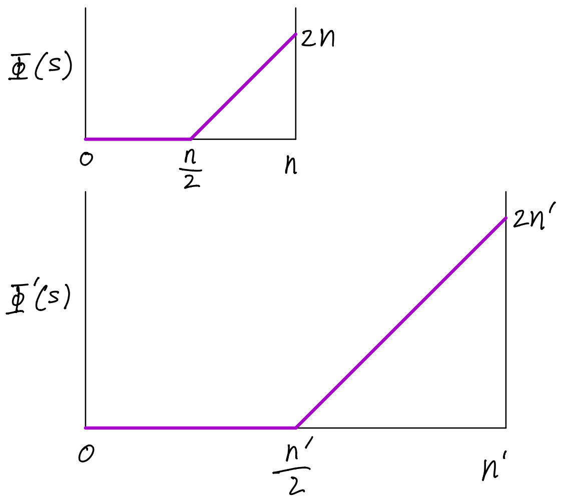
Most of the art of doing an amortized analysis is in choosing the right potential function. Once a potential function is chosen we must do two things:

1. Prove that with the chosen potential function, the amortized costs of the operations satisfy the desired bounds.
2. Bound the quantity $\Phi(s_0) - \Phi(s_n)$ appropriately.

5 Growing a Table Revisited

Let's do the analysis of a growing table using potentials. Here's the potential function:

$$\Phi(n,s) = \begin{cases} 0 & \text{if } s \leq \frac{n}{2} \\ 4(s - \frac{n}{2}) & \text{otherwise} \end{cases}$$



The first graph above shows $\Phi()$ as a function of s (n is fixed). The second graph shows what happens after n doubles. This happens when $s = n$, so you can see that the value of the function goes from $2n$ to 0 as a result.

Lemma 1

The amortized cost of an `insert()` into a table using the above potential function is at most 5. Also, the total cost of a sequence of m `insert()` operations starting from $n = 1, s = 0$ is at most $5m$.

Proof. An increment is comprised of two parts. The first part is the conditional `grow()` operation that might be done. The second part is putting the new element in, and increasing s. We will analyze these separately.

First consider `grow()`. A `grow()` happens when $s = n$, so the value of the potential is $2n$. After the grow, n has been doubled, so $s = n/2$ and the new potential is 0. So $\Delta\Phi = -2n$. The actual cost of `grow()` is $2n$. Adding these together shows that the amortized cost of `grow()` is 0.

The remaining part of the `insert()` costs 1, and causes s to increase by 1. The change in potential is 4. Thus, the amortized cost is 5.

The initial potential is 0, and the final potential is ≥ 0 therefore using equation (3) above we get:

$$\text{total cost} \leq \text{total amortized cost} = 5m.$$

□

Exercise 1

Give a proof for this lemma using the banker's method. Where would you put the banker's tokens in the data structure?

6 Growing and Shrinking a Table

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 shrinking a table is useful. As before, at any given time the size of the current array will be denoted by n and the number of things in the table will be denoted by s. The interface has the following operations.

`initialize()`:	create a new table of size 2 with nothing in it. ($n = 2$ and $s = 0$)
`insert()`:	add a new element to the table. (increment s)
`delete()`:	delete the given element from the table. (decrement s)

(Other operations, e.g., `lookup()`, are immaterial for our purposes today.) To implement the above operations, we'll need two other primitives to deal with the array:

`grow()`:	Change the size of the table from n to $2n$. The cost of this operation is $2n$, the new size of the table.
`shrink()`:	Change the size of the table from n to $n/2$. The cost of the operation is n.

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

`initialize()`:	create a new table of size 2 with nothing in it. ($n = 2$ and $s = 0$)
`insert()`:	if $s = n$ then `grow()`. Now insert the element into the table. (Cost of this part is 1.)
`delete()`:	if $s = n/4$ and $n \geq 4$ then `shrink()`. Now delete the element from the table. (Cost of this part is 1.)

There is a little bit of subtlety in this design. The situation immediately after a `grow()` or a `shrink()` is that $s = n/2$. The key thing is that right after one of these expensive operations, the system is very far from

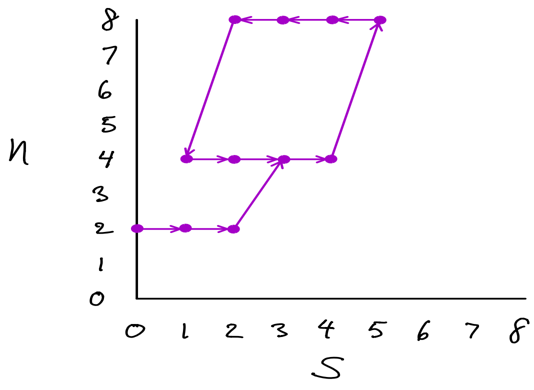
from having to do another expensive operation. This allows it time to build up its piggy bank to pay for the next expensive operation.

Exercise 2

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

It also has what is known as *hysteresis* in physics. This is because the value of n is not purely a function of s. The value of n depends on the *history* of the values of s over time. The following figure shows what happens as s goes from 0 to 5 then back down to 1, then up to 3. The points it goes through (in order) are:

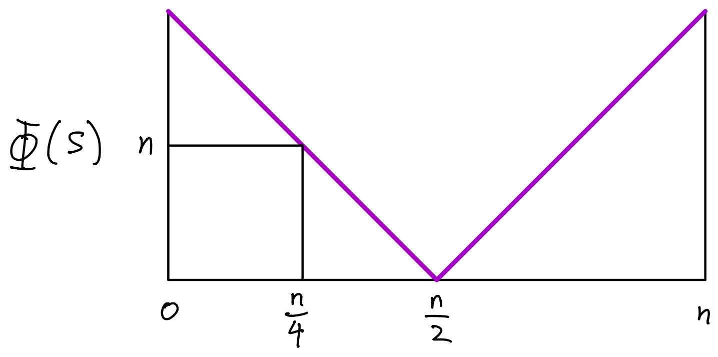
$$(0,2),(1,2),(2,2),(3,4),(4,4),(5,8),(4,8),(3,8),(2,8),(1,4),(2,4),(3,4)$$



Lemma 2

Using $\Phi(n,s) := 4|s - \frac{n}{2}|$, the amortized costs of `insert()` and `delete()` are 5.

Proof. To get a sense of the potential function, consider the figure below. It shows the value of $\Phi()$ as a function of s (n is fixed).



What is the amortized cost of an insertion? It's the actual cost plus the change in potential. A `grow()` may or may not happen as a result of an insertion. If a `grow()` occurs, what is the amortized cost of it? Before the `grow()` the potential is $4|n - \frac{n}{2}| = 2n$. After the `grow()` the potential is 0. The actual cost of the grow is $2n$. Thus the amortized cost of the grow is 0.

What about the rest of the insert?

actual cost of insert $= 1$

change in potential ≤ 4

$\Rightarrow$ amortized cost of insert ≤ 5

What about delete? If a `shrink()` happens, then the potential decreases by n, and the cost is n, so the amortized cost of `shrink()` is 0. What about the rest of the delete:

actual cost of delete $= 1$

change in potential ≤ 4.

$\Rightarrow$ amortized cost of delete ≤ 5.

This completes the proof. □

Theorem 1

The total cost of a sequence of N insertions and deletions is at most $5N + 4$.

Proof. The amortized and real costs are related as follows:

$$\sum \text{actual costs} \leq \left(\sum \text{amortized costs}\right) + \text{initial potential} - \text{final potential}.$$

The initial potential is 4, and the final potential is non-negative. The amortized costs sum to at most $5N$. □

Actually, it's easy to replace the "+4" part with 0 in the theorem. All we have to do is change the potential for $n = 2$ to the following:

$$\Phi(2, s) = \begin{cases} 0 & \text{if } s < 2 \\ 4(s-1) & \text{otherwise} \end{cases}$$

Thus, we've zeroed the potential for the case when the initial array of size 2 is not full. The proof still goes through, because we never shrink an array of size 2, and this part of the potential is not important. This new potential has an initial value of 0, completing the proof.

Exercise 3

Give a proof for this theorem using the banker's method. Where would you put the banker's tokens in the data structure?