CS 624
Lecture 10: Amortized Analysis

Amortized analysis is a technique for analyzing the efficiency of an algorithm. It's very powerful, and it comes in three flavors. We'll only be able to give an introduction to it here.

An *amortized analysis* is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive. The idea is that we should be able to show that there really aren't very many expensive operations in any legal sequence.

Amortized analysis gives us a worst-case bound on the cost of an algorithm. But it is a very sophisticated kind of worst-case bound, because it often allows us to show that even though there may be a certain number of very expensive steps in the algorithm, there aren't really that many compared to the other steps and so the average cost per step (even in the worst case) is better than you might think.

We'll give two very simple examples of this kind of analysis here, and a more substantial—and very remarkable—one in the next lecture.

1 Incrementing a binary counter

Our first example is very simple—maybe even too simple, because there is only one legal sequence. But it is easy to understand, and it's a good introduction.

Suppose we have a binary counter. That is, we have an array of k bits, where each bit can be flipped independently of the others. And suppose the cost of flipping a bit is 1, and that there are no other costs we need to consider. What happens when we start from 0 (i.e., all the bits are 0) and start counting? Figure 1 shows what happens.

How expensive is this?

1.1 Aggregate analysis

It's clear that the cost of any one step can be large. For instance, the cost of the 16^{th} step is 5, and in fact the cost of step number 2^n is going to be $n + 1$. So we can't bound the cost of the steps.

On the other hand, most of the steps have a very low cost, and it does appear that the average cost is low. For instance, we see that the total cost of the first 16 steps is 31, so the average cost of each step (at least out to 16 steps) is less than 2.

counter value	bit index 7	6	5	4	3	2	1	0	incremental cost	total cost
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	1
2	0	0	0	0	0	0	1	0	2	3
3	0	0	0	0	0	0	1	1	1	4
4	0	0	0	0	0	1	0	0	3	7
5	0	0	0	0	0	1	0	1	1	8
6	0	0	0	0	0	1	1	0	2	10
7	0	0	0	0	0	1	1	1	1	11
8	0	0	0	0	1	0	0	0	4	15
9	0	0	0	0	1	0	0	1	1	16
10	0	0	0	0	1	0	1	0	2	18
11	0	0	0	0	1	0	1	1	1	19
12	0	0	0	0	1	1	0	0	3	22
13	0	0	0	0	1	1	0	1	1	23
14	0	0	0	0	1	1	1	0	2	25
15	0	0	0	0	1	1	1	1	1	26
16	0	0	0	1	0	0	0	0	5	31

Figure 1: Incrementing a binary counter.

And in fact, this continues to be true. We can count how many bit flips there are in the first n steps:

- Every step flips bit 0, so that contributes n bit flips.
- Every other step flips bit 1, so that contributes $\lfloor \frac{n}{2} \rfloor$ bit flips.
- Every fourth step flips bit 2, so that contributes $\lfloor \frac{n}{4} \rfloor$ bit flips.

And so on. The total number of bit flips in the first n steps is therefore

$$\sum_{i=0}^{k} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

So no matter how far out we go, the average cost of a step is bounded above by 2. (And also note that the result has nothing at all to do with the value of k. I think this is actually kind of surprising.)

We say that the *amortized cost* of each step is ≤ 2.

1.2 The accounting method

This is another method of amortized analysis. We assign differing charges to different operations, with some operations charged more or less than what they actually cost. The amount we charge an operation is called its *amortized cost*. If the amortized cost is greater than the actual cost, then we put the "extra money" aside as a *credit* to be used later. We just have to make sure that at each step the total charges that we have collected so far suffice to pay for the operations we have performed. That is, if c_i is the cost of step i, and $\hat{c}_i$ is the amortized cost (i.e., the charge we actually impose, which might be more or less than c_i), then for each n, we have

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

Thus, at step 1 we certainly must have $\hat{c}_1 \geq c_1$. If $\hat{c}_1 > c_1$, then we have some "money left over". That means that $\hat{c}_2$ could actually be $< c_2$, provided that if we add in the "money left over" (i.e., the credit) from step 1, that we get a value $\geq c_2$. And so on.

Here's how we can use the accounting method to analyze our binary counter. We will charge an amortized cost of 2 dollars to set a bit to 1. When we set a bit to 1, one of those two dollars pays for the cost of setting the bit. The other dollar is money left over. We leave it with that bit, to be used later when the bit needs to be reset to 0.

Thus at every step, each bit that is set to 1 has a dollar sitting on it.

At each step in our process we walk to to the left until we find the first 0 bit. We set that to 1 and reset all the bits to its right to 0. We charge 2 to set the new bit to 1. One of the dollars pays for the cost of setting the bit to 1, and the other dollar is left on the bit for later. All the bits to the right already have dollars left with them, so it doesn't cost anything additional to reset them to 0—that cost has already been paid for.

Thus each step in the process is charged exactly 2 dollars and this suffices to pay for every operation, even though some operations set many bits. And so again we see that the amortized cost of each operation is 2 (or strictly speaking, ≤ 2).

1.3 The potential method

This is the third method of amortized analysis. It's actually similar to the accounting method[1], except that instead of associating the extra credits with individual objects in the data structure, the total amount of extra credits accumulated at any point is thought of as a "potential" (like potential energy in physics) and is associated with the data structure as a whole. The key thing here, though is that the potential is actually a function of the data structure itself, not how it was produced. So in other words, if there were two different sequences of operations that could lead to the same state, the potential of that state would still be the same.

The idea is this: Let's start with the accounting method. At each step we allocate $\hat{c}_i$ and spend c_i. So the "extra" amount we have saved is $\hat{c}_i - c_i$. (And it's important to remember that this could

[1] In fact, it's really equivalent to it, as we'll see.

actually be less than 0.) But in any case, the total extra amount we have after n steps is

$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$$

and we have already assumed that we've been able to set things up so that this is always ≥ 0. This value is our potential: if we say that we are in state D_n after n steps, then by definition,

$$\Phi(D_n) = \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$$

Actually, we also introduce an "initial potential" $\Phi(D_0)$ which is the potential at "state 0"—i.e., at the beginning, before anything has happened, and we then define

$$\Phi(D_n) - \Phi(D_0) = \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \tag{1}$$

We can pick $\Phi(D_0)$ however we want, but almost always it will be 0. So in any case, we see that at every step n, $\Phi(D_n) - \Phi(D_0) \geq 0$. And that's the only real criterion we need: we must have

$$\Phi(D_n) - \Phi(D_0) \geq 0 \tag{2}$$

for all n.

The book actually does the same thing, but presents it slightly differently. Here's how the book does it:
We assume that somehow we have a potential function Φ on states of a data structure that takes a particular state D_i and produces a real number $\Phi(D_i)$. Our convention will be that D_0 is always the initial state of the data structure. The amortized cost $\hat{c}_i$ of the i^{th} operation is just

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Adding things up, we get

$$\begin{aligned} \sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} \big(c_i + \Phi(D_i) - \Phi(D_{i-1})\big) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \end{aligned}$$

and to make this work, just as in the accounting method, we must have $\Phi(D_n) \geq \Phi(D_0)$ for all n. So the trick is to come up with a potential function Φ that satisfies that constraint. In fact, usually, we just set $\Phi(D_0) = 0$.

The only real difference between the potential function method and the accounting method is that in the potential function method we actually start with the potential function, and (if we need to) we derive the accounting costs $\hat{c}_i$ from it.

For the binary counter problem, D_i is simply the state of the counter after the i^{th} operation. Suppose we set

$$\Phi(D_i) = b_i = \text{the number of 1's in the counter } \textit{after} \text{ the } i^{\text{th}} \text{ operation}$$

(If you remember, this is exactly the value of $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$. But we're not going to use this fact. We'll pretend we're starting all over "from scratch".)

Thus $\Phi(D_0) = 0$, and clearly $\Phi(D_n) \geq \Phi(D_0)$ for all $n \geq 0$. So we certainly have a legal potential function. Now let's see what this says about the amortized cost of an operation.

Let us also set

$$t_i = \text{the number of 1's that are reset during the } i^{\text{th}} \text{ operation}$$

There are two cases we need to consider:

$b_i = 0$. In this case, all the bits just previous to the i^{th} operation must have been 1 (that is $b_{i-1} = k$, and the i^{th} operation must have reset all those bits to 0 (that is $t_i = k$). And further, no bit was set to 1 in this operation[2]. So we have

$$b_{i-1} = t_i = k$$
$$c_i = t_i$$

$b_i > 0$. This is the much more common case. In this case t_i bits are reset to 0, and one bit is set to 1, and so we have

$$b_i = b_{i-1} - t_i + 1$$
$$c_i = t_i + 1$$

Thus in either case, we see that

$$b_i \leq b_{i-1} - t_i + 1$$
$$c_i \leq t_i + 1$$

Therefore the potential difference as we change from state D_{i-1} to state D_i is

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i \end{aligned}$$

and so the amortized cost is

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2 \end{aligned}$$

and so we see in yet a third way that the amortized cost of each operation is ≤ 2.

[2]This is like an odometer that "rolls around to 0".

2 Dynamic Tables

Here we'll consider another problem, also very simple to analyze. We have a need to store objects in a table—a table of strings, for instance. We preallocate a table of a fixed size, but we don't really know how many elements we will need to store in it. So the table may become full. At this point, we have to create a new, larger table and copy the contents of the original one into the new one, and then deallocate the original table. And this might happen any number of times.

Here we will only consider the simplest case—items are only inserted into the table, but never deleted. Such tables actually do occur in compilers, so this is not just a toy problem.

A widely used way of managing these tables is what is called "table doubling": when the table is full, a new table of twice the size is allocated and the processing happens as described above.

Now what is the cost of managing a table like this? We can assume that the cost of a single insertion into the table is 1. That's what happens most of the time. We can assume that the costs of allocating a new table and deallocating an old one are small and fixed. But there is the additional cost of copying m elements from the old table to the new one, and we assume that that cost is m. So most of the time the cost of an insertion is very small—1, in fact. But occasionally it is much bigger. We want to find the amortized cost. For simplicity, we assume that the table is initially empty. (Of course this would never be the case in practice, but we really want to know what happens in the long run, and this just makes the algebra come out easier without really affecting the final result.)

2.1 The aggregate method

The cost of the i^{th} operation is

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

Thus, the total cost of n TABLE-INSERT operations is

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n \end{aligned}$$

and so the amortized cost of each operation is at most 3.

2.2 The accounting method

We'll say that the amortized cost $\hat{c}_i$ for the i^{th} insertion is 3 dollars, and this works out as follows, intuitively at least:

- One dollar pays for inserting the element itself.
- One dollar is stored to move the element later when the table is doubled.

- One dollar is stored to move an element in the table that was already moved from a previous table.

Suppose for instance, the size of the table is m immediately after expansion. So the number of elements in the table is $m/2$. If we charge 3 dollars for each insertion, then by the time the table is filled up again, we will have $2(m/2)$ extra dollars, which pays for moving all the elements in the table to the new table.

So again we see that the amortized cost per operation is ≤ 3.

2.3 The potential method

Essentially this is the same as the accounting method. We want a potential function that tells us how much money we have saved up. We want this function to be initially 0, and also to be 0 just after each table doubling. And we want it to grow as elements are inserted so that when we need to double the table we have enough saved up to do it. The obvious function then is

$$\Phi(T) = 2 \cdot num[T] - size[T]$$

At the beginning, and also immediately after each expansion, $\Phi(T) = 0$. Just before an expansion, we have $num[T] = size[T]$, so $\Phi(T) = num[T]$, which is exactly what costs to move every element in T into the new table.

And clearly $\Phi(T) \geq 0 = \Phi(\text{the empty table})$, so Φ is a a legal potential function.

Let's see how this works out for each operation: let's say that num_i is the number of elements in the table *after* the i^{th} operation, that $size_i$ is the size of the table *after* the i^{th} operation, and that Φ_i is the potential *after* the i^{th} operation.

- If the i^{th} TABLE-INSERT operation does not trigger an expansion, then we have $size_i = size_{i-1}$, and the amortized cost of the operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\ &= 3\end{aligned}$$

- If the i^{th} TABLE-INSERT operation *does* trigger an expansion, then we have $size_i = 2 \cdot size_{i-1}$ and $size_{i-1} = num_{i-1} = num_i - 1$, so $size_i = 2 \cdot (num_i - 1)$. And so we have

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= num_i + \big(2 \cdot num_i - 2 \cdot (num_i - 1)\big) - \big(2(num_i - 1) - (num_i - 1)\big) \\ &= num_i + 2 - (num_i - 1) \\ &= 3\end{aligned}$$

so again we see that the amortized cost of each insertion is ≤ 3.