

CS624 - Analysis of Algorithms

Spring 2018 – Amortized analysis

October 26, 2018

Amortized Analysis

- Amortized analysis is a technique for analyzing the efficiency of an algorithm.
- 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.
- We should be able to show that there really aren't very many expensive operations in any legal sequence.

Incrementing a Binary Counter

- This is a very simple example – there is only one legal sequence.
- Suppose we have a binary counter – an array of k bits, where each bit can be flipped independently of the others.
- 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?

Incrementing a Binary Counter

counter value	bit index								incremental cost	total cost
	7	6	5	4	3	2	1	0		
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

How expensive is this?

Aggregate Method

- It's clear that the cost of any one step can be large.
- For instance, the cost of the 16th step is 5, and in fact the cost of step number 2^n is going to be $n + 1$.
- 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 – 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.
- Therefore 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 etc.

- The total number of bit flips in the first n steps is therefore

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

- No matter how far out we go, the average cost of a step is bounded above by 2.
- Note that the result has nothing to do with the value of k .
- We say that the *amortized cost* of each step is ≤ 2 .

The Accounting Method

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

The Accounting Method

- 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.
- For the 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.

The Accounting Method

- At each step in our process we walk 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 (for its reset to 0).
- 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.
- Again we see that the amortized cost of each operation is 2 (rather, ≤ 2).

The Potential Method

- Quite similar to the accounting method, but 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.
- 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 Potential Method

- We assume that 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 is that D_0 is always the initial state of the data structure.
- The amortized cost \hat{c}_i of the i^{th} operation is
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
- Therefore we get

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

The Potential Method

- To make this work we must have $\Phi(D_n) \geq \Phi(D_0)$ for all n .
- The trick is to come up with a potential function Φ that satisfies that constraint.
- In fact, usually, we just set $\Phi(D_0) = 0$.
- For the binary counter problem, D_i is simply the state of the counter after the i^{th} operation.
- Suppose we set

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

- Thus $\Phi(D_0) = 0$, and clearly $\Phi(D_n) \geq \Phi(D_0)$ for all $n \geq 0$. This is a legal potential function.

The Potential Method

- Let us set $t_i =$ the number of 1's that are reset during the i^{th} operation. Now the actual cost c_i of the i^{th} operation is $t_i + 1$ (because t_i bits are reset to 0, and one bit is set to 1). Also –
- If $b_i = 0$, then the i^{th} operation must have reset all the bits (there are k of them), and so $b_{i-1} = t_i = k$.
- If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$
- So in either case $b_i \leq b_{i-1} - t_i + 1$

The Potential Method

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

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

- The amortized cost is
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$$
- We see in yet a third way that the amortized cost of each operation is ≤ 2 .

Example – Dynamic Tables

- We need to store objects in a table of strings, for instance.
- We preallocate a table of a fixed size, but we don't know how many elements we will need to store in it – so the table may become full.
- We then create a larger table and copy the contents of the original one into the new one, and then deallocate the original table.
- Assume for simplicity that 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.
- When the table is full, a new table of twice the size is allocated and the processing happens as described above.

Dynamic Tables

- What is the cost of managing a table like this?
- Assuming that the cost of a single insertion into the table is 1 and the costs of allocating a new table and deallocating an old one are small and fixed.
- There is the m cost of copying m elements from the old table to the new one.
- Most of the time the cost of an insertion is very small but occasionally it is much bigger.
- For simplicity, we assume that the table is initially empty.

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 Insert operations is

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

- So the amortized cost of each operation is at most 3.

The Accounting Method

- The amortized cost 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 the size of the table is m immediately after expansion – 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 .

The Potential Method

- Quite similar to the accounting method
- A potential function tells us how much money we saved up
- This function is initially 0, and also to be 0 just after each table doubling.
- 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 $\Phi(T) = 2 \cdot \text{num}[T] - \text{size}[T]$
- At the beginning, and also immediately after each expansion, $\Phi(T) = 0$. Just before an expansion, we have $\text{num}[T] = \text{size}[T]$, so $\Phi(T) = \text{num}[T]$, which is exactly what costs to move every element in T into the new table.
- Clearly $\Phi(T) \geq 0 = \Phi(\text{the empty table})$, so Φ is a legal potential function.

The Potential Method

- If 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} TableInsert operation does not trigger an expansion, then we have $size_i = size_{i-1}$, and the amortized cost of the operation, \hat{c}_i is:

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

The Potential Method

- If the i^{th} TableInsert 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 $size_i = 2 \cdot (num_i - 1)$. Then \hat{c}_i is

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

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