ICS 691: Advanced Data Structures Spring 2016

Lecture 1

Prof. Nodari Sitchinava *Scribe: Ben Karsin*

1 Overview

In the last lecture we reviewed the take-home exam and discussed the problems therein. It was the first lecture so we did not visit any new material.

In this lecture we will look at amortized analysis, which is explained in chapter 17 of [CLRS09]. We will discuss 3 methods of performing amortized analysis: *aggregate method*, *accounting method*, and *potential method*. We will look at these 3 methods in the context of two problems: the binary counting problem from the take-home exam and the problem of dynamic array.

2 Amortized Analysis

Amortized analysis is a method of analyzing algorithms that can help us determine an upper bound on the complexity of an algorithm. This is particular useful when analyzing operations on data structures, when they they involve slow, rarely occurring operations and fast, more common operations. With this disparity between each operations' complexity, it is difficult to get a tight bound on the overall complexity of a sequence of operations using worst-case analysis. Amortized analysis provides us with a way of averaging the slow and fast operations together to obtain a tight upper bound on the overall algorithm runtime.

2.1 Aggregate Method

The first method of amortized analysis is called the *aggregate method* and involves counting out the complexity of each operation. By expanding each case, one can try to determine a pattern and come up with an overall upper bound for the algorithm complexity. Intuitively, we can think of the aggregate method as being performed by *counting up* the complexity of each operation and using the sum to determine the total algorithm complexity.

To demonstrate how this method works, we will be working on exercise 3 from the take-home exam, Binary counter:

An array $A[0 \cdots k-1]$ of bits (each array element is 0 or 1) stores a binary number $x = \sum_{i=0}^{k-1} A[i]\dot{2}^i$. To add 1 (modulo 2^k) to x, we can use the following procedure:

```
Increment(A,k)
```

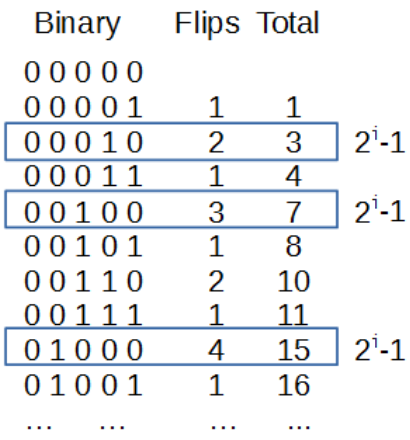

Figure 1: Illustration of the steps of the Binary counter operation. The highlighted rows are those where the most bits are flipped, where bit i is flipped, the total operations up to that point is $2^i - 1$.

```
i=0
while (i<k and A[i] = 1) do
  A[i]=0
  i=i+1
if i<k
  then A[i] = 1
```

Let $n < 2^k$ and assume the array A has been initialized to all 0's. Perform an analysis that calls $Increment(A, k)$ n times.

Using worst-case analysis of this problem, we see that each call to $Increment(A, k)$ will flip, at most, $\log n$ bits, so our total algorithm complexity is $O(n \log n)$. However, observe that many of the n calls to $Increment$ require very little work, and amortized analysis allows us to come up with a tighter upper bound. To begin, let's look at the first few calls to $Increment$ and evaluate the total work required (see Figure 1).

We see in Figure 1 that the total flips at any step is, at most, $2^i - 1$ of the total number of bits flipped that step. This gives us some insight into the pattern, but it is still difficult to determine the amortized cost of the entire problem.

An alternate approach, that gives us a simpler way to count the amortized cost, is to look at each *column* of bits and count the total flips that that column sees. Figure 2 illustrates how this approach lets us count the total work needed to perform n calls $Increment$.

With this approach, we can see that the first column is flipped *every* step, the 2nd column is flipped every 2 steps, the 3rd column is flipped every 4 steps, and so on. This gives us the total number of flips for n calls to $Increment$ as:

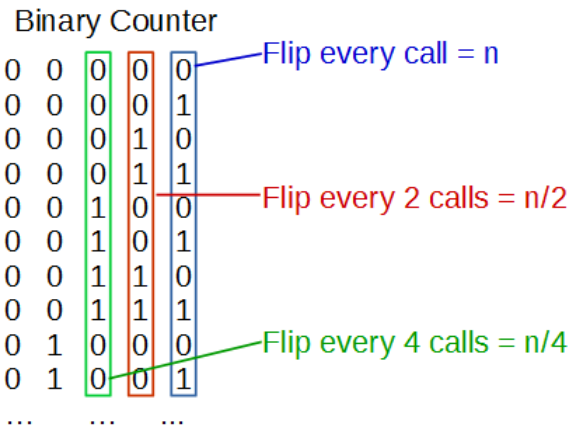

Figure 2: By looking at the *columns* of the binary counter, we can more easily total up the amortized cost with the aggregate method.

$$\begin{aligned} flips &= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + \frac{n}{2^{\log n}} \\ &\leq 2n = O(n) \end{aligned}$$

The amortized cost of call i of $Increment$, therefore, is:

$$\begin{aligned} \hat{c_i} &= \frac{\text{total \# of bit flips}}{\text{\# of calls}} \\ &= \frac{2n}{n} = 2 \end{aligned}$$

Total amortized cost, $\sum_{i=1}^{n} \hat{c_i}$, provides us with an *upper bound* on the total work of an algorithm. Therefore, for any n, our total amortized cost must be at least the total actual cost:

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

Where $\hat{c_i}$ is the amortized cost, and c_i is the actual cost, of the ith call. Thus, the total cost of n calls to $Increment(A, k)$ is:

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} 2 \\ &= 2n \end{aligned}$$

Binary Counter	Step Cost	Bank Total
0 0 0 0 0		
0 0 0 0 1	\$1	\$1
0 0 0 1 0	\$2	\$1
0 0 0 1 1	\$1	\$2
0 0 1 0 0	\$3	\$1
0 0 1 0 1	\$1	\$2
0 0 1 1 0	\$2	\$2
0 0 1 1 1	\$1	\$3
0 1 0 0 0	\$4	\$1
0 1 0 0 1	\$1	\$2
… … …		

Figure 3: Using the accounting method to perform amortized analysis of the binary counter problem.

2.2 Accounting Method

Intuitively, we can consider the accounting method as "saving for a rainy day." The idea is to allocate a fixed cost d for each step of the algorithm. Low-cost calls will accrue "money" to be able to pay for more expensive calls. The steps of the accounting method are as follows:

- Charge $\hat{c}_i = d$ dollar coins for the i-th operation,
- Subtract actual cost c_i of the operation from d to pay for the operation,
- Put the remaining $(d - c_i)$ dollar coins in the bank to pay for future (expensive) operations.

For amortized cost to be a valid upper bound on the actual cost for any number of operations, we must ensure that $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for any n. Re-ordering terms leads to:

$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (\hat{c}_i - c_i) \geq 0 \text{ for every } n.$$

In other words, the amount in "the bank" must never drop below 0, during any step of the algorithm. As long as this holds, we know that our amortized cost of $\hat{c}_i$ per operation is a valid amortized cost of the operation. To show that this is the case, we typically put the extra coins on specific objects within the data structure.

Let's look at the example of the binary counter problem. We first set our cost estimate $\hat{c}_i = d = 2$ dollar coins. Figure 3 illustrates how we "save" on low-cost operations and spend the savings on more costly ones. In this example, we see that every time we flip a bit from '0' to '1', the actual cost is 1 dollar coin, but we allocated 2 dollar coins, saving 1 dollar coin in the "bank". We place this extra coin on the '1' bit that has just been flipped from '0'. In the future, we will use the coins saved on '1' bits, to flip them to '0' and newly charged coins to flip '0' bits to '1'. Thus, we will never spend more coins than our savings.

2.3 Potential Method

The potential method is similar to a physics outlook and looks at the "potential" of the entire data structure as a single value. We define Φ as a function that maps a data structure D_i (after the ith call) to a real number. Using this potential function, Φ, we define the amortized cost as:

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

Observe that:

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

Therefore, as long as we show that our choice of the potenital function Φ results in non-negative $\Phi(D_n) - \Phi(D_0)$ for every n, $\hat{c_i}$ is a valid amortized cost of i-th operation. Furthermore, if we choose the potential function Φ such that the initial potential $\Phi(D_0) = 0$, we only need to show that $\Phi(D_i)$ is non-negative, for every i.

To analyze the binary counter problem using the potential method, we define Φ as:

$$\Phi(D_i) = \#\text{ of 1's in the binary counter}$$

Clearly $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for any i. To compute $\hat{c_i} = c_i + \Delta\Phi_i$, let's define b_i as the total number of 1's in D_{i-1}, i.e., right before calling i-th *Increment* operation, and t_i as the total 1's that were flipped to 0's. Figure 4 illustrates the value of Φ as we perform the *Increment* operation on the data structure D_i. We see that t_i is the number of consecutive least significant bits that were 1's and were flipped to 0's. Thus, the actual cost of the i-th call is $c_i = t_i + 1$ (+1 for fliping 0 to 1). The change in the potential is:

$$\Delta\Phi_i = (b_i - t_i + 1) - b_i = 1 - t_i$$

Then the amortized cost of the i-th operation is:

$$\hat{c_i} \quad = c_i + \Delta\Phi_i = (t_i + 1) + (1 - t_i) = 2$$

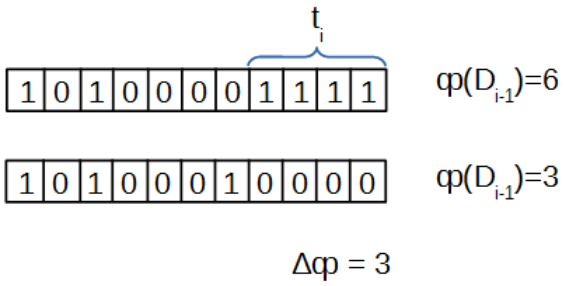

Figure 4: Example of the Φ potential value for the binary counter problem

3 Dynamic Arrays

Dynamic arrays are a common data structure available in many programming languages. In [CLRS09], the process of inserting into a dynamic array is given as follows:

```
Starting with |A|=1
Insert(x,A)
  if A not full
    insert x into A
  else
    allocate A' of size 2|A|
    copy A to A'
    set A <- A'
    insert x into A
```

Using worst-case analysis, we see that inserting x into A requires copying n elements, in the worst case. Performing n insert operations, therefore, is, in the worst case, $O(n^2)$. However, since many of calls to *Insert* take significantly less work, using amortized analysis, we can show that the amortized cost of a single insert is $O(1)$, giving us $O(n)$ amortized cost of inserting n items into a dynamic array.

3.1 Aggregate Analysis

Using aggregate analysis, we can look at the cost of each step in Figure 5. As illustrated in Figure 5, the cost of inserting element i into the dynamic array is:

$$c_i = \begin{cases} i, & \text{if } i-1 = 2^k. \\ 1, & \text{otherwise.} \end{cases}$$

Separating these two terms into separate summations, we can determine it as:

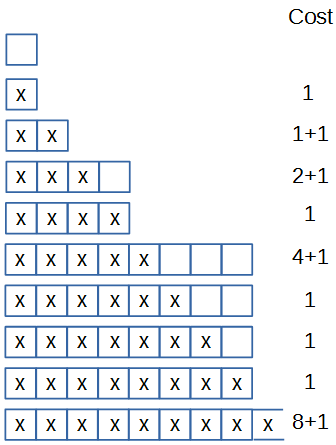

Figure 5: The status of the dynamic array as we insert elements into it.

$$\begin{aligned}\sum_{i=1}^{n} c_i &= \sum_{i=0}^{\log n} 2^k + \sum_{i=1}^{n} 1 \\ &\leq 2^{\log n+1} + n = 2n + n \\ &= 3n\end{aligned}$$

This gives us an amortized cost of each call of $\frac{3n}{n} = 3 = O(1)$

3.2 Accounting Method

Let us charge $\$d$ coins for each insertion into the dynamic array. The actual cost is \$1 if we don't need to resize the array, so we can save $\$d - 1$ to pay for copying the array in the future. We associate these $d-1$ dollar coins with the newly inserted element. Thus, we always have $d-1$ coins accumulated per each element that has never been copied before. Right before resizing an array of size k, half of the elements in the array have never been copied before. Thus, to be able to pay for moving elements that had been copied before (the other half), we need:

$$\begin{aligned}(d-1) \times \frac{k}{2} &\geq k \\ d &\geq 3\end{aligned}$$

Thus, we we charge $\hat{c}_i = 3 = O(1)$ coins for each insertion, we will never spend more than our savings during expansion of the array.

3.3 Potential Method

Let x_i be the number of items in the second half of the array and let s_i be the total capacity of the array after the i-th insertion. Then we can define the potential function Φ as $\Phi(D_i) = 2x_i$.

Note that $i = s_i/2 + x_i$. Hence, $\Phi(D_i) = 2x_i = 2i - s_i$.

Clearly, $\Phi(D_0) = 0$. and $D_i \geq 0$ for all $i \geq 0$.

Then we can compute $\Delta\Phi_i$ as:

$$\begin{aligned}\Delta\Phi_i &= \Phi(D_i) - \Phi(D_{i-1}) \\ &= (2i - s_i) - (2(i-1) - s_{i-1}) \\ &= 2 + s_{i-1} - s_i\end{aligned}$$

Note if the i-th insertion caused an expansion of the array, then $s_i = 2s_{i-1}$ and otherwise $s_i = s_{i-1}$. Therefore,

$$\begin{aligned}\Delta\Phi_i &= \begin{cases}2 - s_{i-1}, & \text{if } i - 1 = 2^k \\ 2, & \text{otherwise}\end{cases} \\ &= \begin{cases}2 - (i-1), & \text{if } i - 1 = 2^k \\ 2, & \text{otherwise}\end{cases}\end{aligned}$$

Remember from the previous section that the actual cost of insertion, c_i, is:

$$c_i = \begin{cases}i, & \text{if } (i-1) = 2^k \\ 1, & \text{otherwise}\end{cases}$$

Thus, we can calculate $\hat{c}_i$:

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi_i = \begin{cases}i + (2 - (i-1)), & \text{if } (i-1) = 2^k \\ 1 + 2, & \text{otherwise}\end{cases} \\ &= 3\end{aligned}$$

References

[CLRS09] T. Cormen, C. Lieserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, third edition, 2009.