ICS 621: Analysis of Algorithms Fall 2019

Lecture 1

Prof. Nodari Sitchinava *Scribe: Mojtaba Abolfazli, Sushil Shrestha*

1 Overview

In this lecture, we explore amortized analysis methods including: *aggregate method*, *accounting method*, and *potential method* in details. We look at these methods in the context of two problems: binary counting and dynamic array.

2 Amortized Analysis

Amortized analysis is a method for analyzing algorithms in terms of required average time for performing a set of operations, which results in the upper bound on the complexity of an algorithm. This is a useful tool, especially when analyzing operations on data structures with slow, but rarely occurring operations, and fast, but more common operations. With this disparity between each operation's complexity, it is difficult to get a tight bound on the overall complexity by using worst-case analysis. Amortized analysis provides a way of averaging the slow and fast operations together to obtain an upper bound on the overall algorithm runtime $T(n)$ in the worst case.

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. The amortized cost would be $T(n)/n$ where $T(n)$ denotes the total cost of a sequence of n operations.

To demonstrate how this method works, we discuss *Binary Counter* problem which is defined as follows:

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

Example 2.1. Let $n < 2^k$ and array A be initialized to all 0's. Perform an analysis that calls the function Increment(A, k), n times and calculate amortized cost.

Solution:
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 after n calls would be $O(n \log n)$. However,

Algorithm 1 INCREMENT(A, k)

```
1: $i = 0$
2: while $i < k$ & $A[i] = 1$ do
3:     $A[i] = 0$
4:     $i = i + 1$
5: end while
6: if $i < k$ then
7:     $A[i] = 1$
8: end if
```

observe that many of the n calls to INCREMENT(A, k) 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(A, k) and evaluate the total number of operations required as illustrated in Figure 1.

Binary Counter					Flips	Total
0	0	0	0	0		
0	0	0	0	1	1	1
0	0	0	1	0	2	3
0	0	0	1	1	1	4
0	0	1	0	0	3	7
0	0	1	0	1	1	8
0	0	1	1	0	2	10
0	0	1	1	1	1	11
0	1	0	0	0	4	15
0	1	0	0	1	1	16
...		...	...		...	...

Figure 1: The process for Binary Counter operation. The highlighted rows are those with the most flipped bits. Total operations up to ith flipped bit is $2^i - 1$.

As it can be seen, the total number of flips at steps where we have the most flips is equal to $2^i - 1$. This gives some insight into the pattern, but it is still difficult to determine the amortized cost of the entire problem.

An alternative 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 in that column. Figure 2 illustrates how this approach lets us count the total operations needed to perform n calls of INCREMENT(A, k).

According to the figure, the first column is flipped at *every* call, the second column is flipped every 2 calls, the third column is flipped every 4 calls, and so on. This gives us the total number of flips after n calls:

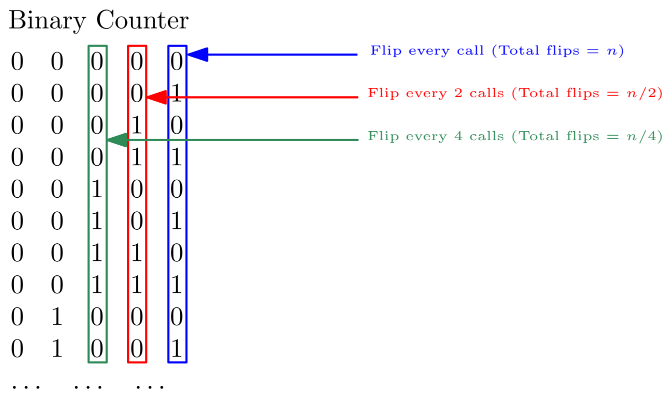

Binary Counter
0 0 0 0 0
0 0 0 0 1
0 0 0 1 0
0 0 0 1 1
0 0 1 0 0
0 0 1 0 1
0 0 1 1 0
0 0 1 1 1
0 1 0 0 0
0 1 0 0 1
... ...
Flip every call (Total flips = n)
Flip every 2 calls (Total flips = $n/2$)
Flip every 4 calls (Total flips = $n/4$)

Figure 2: Total cost by column in aggregated method.

$$\begin{aligned}
\text{Total \# of flips} &= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + \frac{n}{2^{\log n}} = \sum_{i=0}^{\log n} \frac{n}{2^i} \\
&\leq n \sum_{i=0}^{\infty} \frac{1}{2^i} \\
&= 2n
\end{aligned}$$

Therefore, the amortized cost $\hat{c}$ is:

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

Total amortized cost $\hat{c} = \sum_{i=1}^{n} \hat{c}_i$ ($\hat{c}_i$ is the amortized cost of the ith operation) provides us with an upper bound on the total runtime of an algorithm. Therefore, for any n, total amortized cost must not be less than the total actual cost:

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

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

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} 2 = 2n$$

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 in the future. The steps of the accounting method are as follows:

- Charge $\hat{c}_i = d$ dollar for the ith operation,
- Subtract actual cost c_i of the operation from d to pay for the operation,
- Put the remaining $(d - c_i)$ dollar in the bank as credit to pay for future 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:

$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (\hat{c}_i - c_i) \geq 0, \quad \forall n \in \mathbb{N}.$$

In other words, the amount in the account must never drop below 0, during any step of the algorithm. As long as this condition holds, we know that our amortized cost of $\hat{c}_i$ per operation is a valid amortized cost.

Let's look at the example of the binary counter problem. We first set our cost estimate $\hat{c}_i = d = 2$ \$. 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 \$, but we allocated 2 dollar coins which means we save extra dollar as credit in the account. We place this extra coin on the bit that has just been flipped from '0' to '1'. 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. Figure 3 shows how our saving is spent during the steps where we face multiple flips.

Binary Counter					Step Cost (\$)	Available Credit (\$)
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 Accounting method to perform amortized analysis on binary counter problem.

2.3 Dynamic Array Problem

Dynamic arrays are just like normal arrays but the capacity of the array is doubled every time array hits the full capacity. The cost of Append operation in dynamic array is 1 when array has empty spaces but insertion operation includes cost of copying old array plus cost of inserting new item to the array when spaces are fully occupied. Dynamic arrays are a common data structure available in many programming languages.

Algorithm 2 Append(x, A)

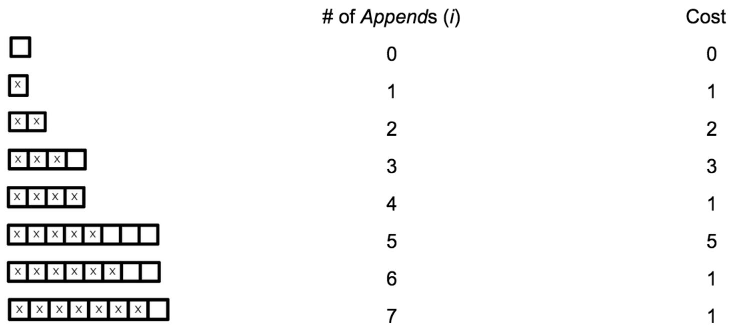
```
1: Let $i =$ number of items in $A$
2: if $|A| = i$ then
3:     $B =$ new Array of size $2 \cdot i$
4:     Copy$(A, B)$
5:     $A \leftarrow B$
6: end if
7: $A[i] = x$
8: $i = i + 1$
```

The pseudocode for inserting a value into a dynamic array is shown in Algorithm: 2. In the worst case, appending x to a dynamic array using above algorithm requires copying n elements. If we repeat the same operation n times, it will cost us $O(n^2)$ operations. But many of Append function calls won't have to copy the elements. Most of the operations will only take 1 operation(just append). Hence we can use amortized analysis to show that the amortized cost of a single append is $O(1)$.

Example 2.2. Find the amortized cost of appending a value to a dynamic array and use the result to find appropriate cost for accounting method.

Solution:

Aggregate Analysis: Using aggregate analysis, we can look at the cost of each step in Figure 4.



# of *Appends* (*i*)	Cost
0	0
1	1
2	2
3	3
4	1
5	5
6	1
7	1

Figure 4: The status of the dynamic array as we insert elements into it.

As illustrated in Figure 4, the cost of inserting ith element into the dynamic array is:

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

Now, to find the overall cost of the operation over nth iterations, we can sum these two terms into separate summations.

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

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

Accounting Method: Let us assume, we will charge d dollars for each append to the dynamic array. We only need \$1 for append, so we can save \$$(d-1)$ to pay for extending and copying the value in future. We will keep track of our saved money by associating it with the newly inserted element. Each newly inserted element(which has never been copied) will have \$$(d-1)$ with them.

By the time we have to resize the array of size k, we'll have half of the elements in the array that have never been copied, so they will have \$$(d-1)$ with them. Remaining half of the elements were copied earlier i.e. they have exhausted their savings so we will use the \$$(d-1)$ (saving from new uncopied ones) to copy remaining half of the elements(those were copied in past). So we will need:

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

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

2.4 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 $\Phi(D_i)$ as a function that maps a data structure D_i (after the ith call) to a real number.

$$\begin{aligned} &\Phi(D_i) : D_i \to \mathbb{R} \\ &\qquad \text{where } D_i \text{ is the data structure after the } i\text{th operation} \\ &\qquad\qquad \Phi_i \text{ is potential of the data structure after the } i\text{th operation.} \end{aligned}$$

Using this potential function, Φ_i, we define the amortized cost as:

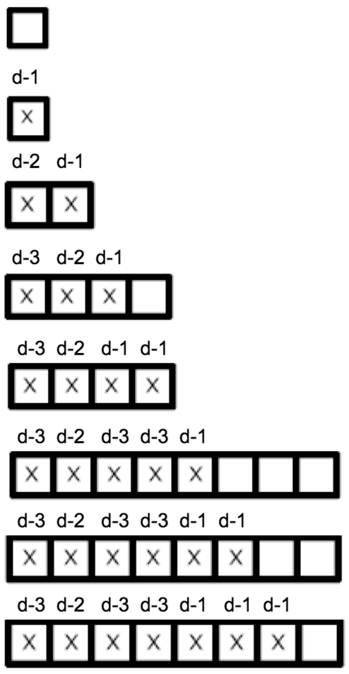


Figure 5: The coins accumulation account for the dynamic array insertion problem.

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

Now, we can find total amortized cost for n operations as:

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

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.

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

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

That means, $\Phi(D_0) =$ number of 1's in $D_0 = 0$ and $\Phi(D_i) \geq 0$ for any i.

Now, we have
$\hat{c}_i = c_i + \Delta\Phi_i$

Let's define t_i be the total 1's that were flipped to 0's during the ith call to INCREMENT. We see that t_i is the number of consecutive least significant bits that were 1's and were flipped to 0's.

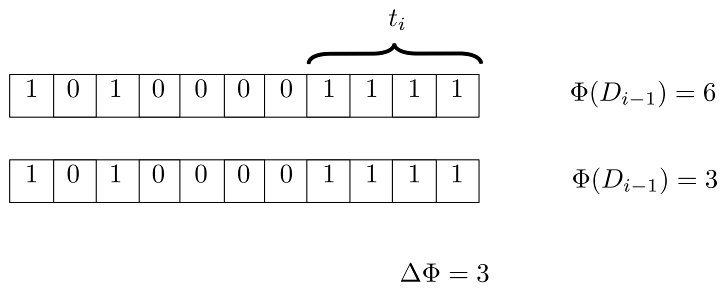


Figure 6: Example of the Φ potential value for the binary counter problem

Thus, the actual cost of the i-th call is $c_i = t_i + 1$ (+1 for fliping 0 to 1).

Also total number of 1's in the binary counter after ith increment is given by:

$$\begin{aligned}\Phi(D_i) &= \text{total number of 1's in } D_{i-1} - \text{flipped bits} \\ &= (\Phi(D_{i-1}) - t_i + 1)\end{aligned}$$

The change in the potential is:

$$\Delta\Phi_i = (\Phi(D_{i-1}) - t_i + 1) - \Phi(D_{i-1}) = 1 - t_i$$

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

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

Thus the amortized cost for increment is $\hat{c}_i = 2 = O(1)$.