ICS 621: Analysis of Algorithms Fall 2021

Lecture 1

Prof. Nodari Sitchinava *Scribe: In Woo Park, Darlene Agbayani, Michael Rogers*

1 Overview

In this lecture, we discuss three methods of performing ***amortized analysis***: aggregate method, accounting method, and potential method. We look at these methods in the context of two problems: binary counting and dynamic arrays.

2 Amortized Analysis

Amortized analysis is a method for analyzing algorithms in terms of the worst-case upper bound for performing a set of operations. The total cost of any given algorithm is the sum of the individual costs c_i of each operation:

$$C = \sum_{i=1}^{n} c_i$$

Typically, we want to find the upper bound on this overall cost. For example, a simple way to estimate the runtime of the following algorithm might be to say the two inner for-loops each run in linear time while lines 1 and 4 are constant operations, therefore the algorithm runs in $O(n^2)$. This may be a valid method of analyzing the runtime, but it is not a precise upper bound. In this lecture we will see how we can use amortized analysis to achieve better results, i.e., a tighter upper bound.

```
TripleLoop(A, n)
1  for k = 1 to 10                ▷ c1 = O(1)
2      for i = 1 to n − 1         ▷ c2 = O(n)
3          for j = n downto i     ▷ c3 = O(n)
4              A[i] = A[j] + 1    ▷ c4 = O(1)
```

$$\begin{aligned}\sum_{k=1}^{10}\sum_{i=1}^{n-1}\sum_{j=i}^{n} c_4 &= 10 \cdot c_4 \cdot \frac{(n+2)(n-1)}{2} \\ &= 5 \cdot c_4 \cdot (n+2)(n-1) \\ &= O(n^2)\end{aligned}$$

In amortized analysis we estimate the cost of an individual operation and calculate the worst-case runtime across all operations. Intuitively, it may appear as an average cost, but in Computer Science, a key distinction is made between average cost and amortized cost. The term "average" is reserved to describe the expected runtime of a randomized algorithm, whereas the arithmetic average of the amortized cost of n operations is used to determine the worst-case upper bound (Figure 1).

For example, let $\hat{c}_i = \log(n)$, then the worst-case runtime is $C \leq \sum_{i=1}^{n} \hat{c} = n \cdot \log n$. Recall, the expected cost of the QuickSort algorithm is $\Theta(n \log n)$. While the probability of the worst-case scenario is very low ($\frac{2}{n!}$), it is, in fact, still possible to have a runtime of $C = O(n^2)$.

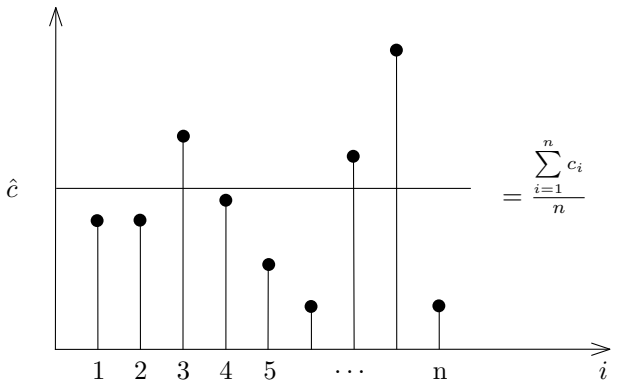


Figure 1: Amortized cost $\hat{c}_i$ over n operations.

Amortized analysis is useful 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 only using worst-case analysis. Amortized analysis provides a tighter upper bound on the overall algorithm runtime $T(n)$ in the worst case for these operations.

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 ($\hat{c}_i$) 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 the *Binary Counter* problem which is defined as follows:

Algorithm 1 Binary Counter

Require: B, k ▷ where B is an array of k bits

INCREMENT(B, k)
1 $i = 0$
2 **while** ($i < k$ and $B[i] == 1$)
3 $\quad B[i] = 0$
4 $\quad i = i + 1$
5 **if** $i < k$
6 $\quad B[i] = 1$

$\Big\}$ $\leq k$

For the runtime of n increments, we have: $C = \sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} k = k \cdot n \leq n \log n$. However, it does not take this long. A tighter analysis will show the amortized cost of each increment operation is constant.

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

Solution:
Using the worst-case analysis of this problem, we see that each call to INCREMENT(B, k) will flip, at most, $\log n$ bits, so our total algorithm complexity after n calls would be $O(n \log n)$. However, observe that many of the n calls to INCREMENT(B, k) require very little work. To begin, let's look at the first few calls to INCREMENT(B, k) and evaluate the total number of operations required as illustrated in Figure 2.

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 2: The process for Binary Counter operation.

Notice that the total number of flips at the steps where we have the most flips (highlighted in blue) 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 3 illustrates how this approach lets us count the total operations needed to perform n calls of INCREMENT(B, k).

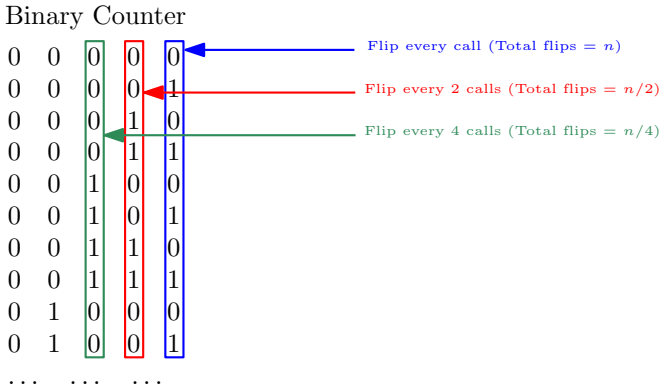


Figure 3: Total cost by column in aggregated method.

According to Figure 3, 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:

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

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 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 4 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 associate this extra coin with 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 4 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 4: Using Accounting method to perform amortized analysis on binary counter problem

Student Question: Why did we choose \$2 as our arbitrary dollar amount?

Solution: We chose \$2 as a guess. We could've used other positive integers to satisfy our two conditions but that only loosens our upper bound. Two turns out to be the lowest cost we can use without our bank going below 0, and it is the smallest upper bound.

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.

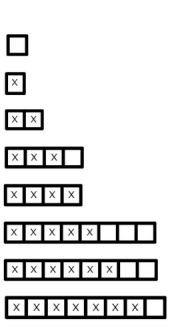
Algorithm 2 Append to an array

```
APPEND(A, x)
1  Let i be the number of items in A
2  if |A| == i
3        B = new array of size 2i
4        Copy A into B
5        A = B
6  A[i] = x
7  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 5.



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

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

As illustrated in Figure 5, 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 n 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 the 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 will have half of the elements in the array that have never been copied, so they will have $\$(d-1)$ with them. The 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 unoccupied 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.

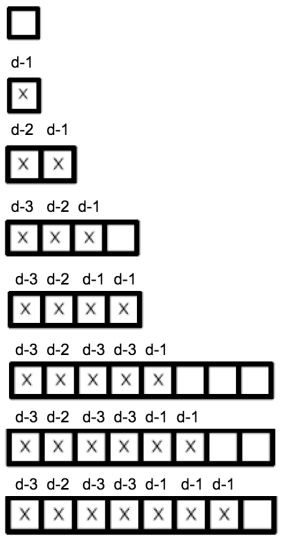


Figure 6: The coins accumulation account for the dynamic array insertion problem.

2.4 Potential Method

$$\underbrace{C = \sum_{i=1}^{n} c_i}_{\text{actual}} \leq \underbrace{\hat{C} = \sum_{i=1}^{n} \hat{c}_i}_{\text{amortized}}$$

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.

$$\Phi(D_i) : D_i \to \mathbb{R}$$

where D_i is the data structure after the ith operation

Φ_i is potential of the data structure after the ith operation.

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

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

Now, we can find total amortized cost for n operations using the following formula:

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} \Delta\Phi_i \\ &= \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \cancel{\Phi(D_1)} - \Phi(D_0) + \cdots + \Phi(D_n) - \cancel{\Phi(D_{n-1})} \qquad \text{// telescoping series} \\ &= \sum_{i=1}^{n} c_i + \underbrace{\Phi(D_n)}_{\text{final potential}} - \underbrace{\Phi(D_0)}_{\text{initial potential}}\end{aligned}$$

Therefore, if we show that our choice of the potential function Φ results in a non-negative difference $\Phi(D_n) - \Phi(D_0)$ for every n, then we can ensure that the total cost $\sum_{i=1}^{n} c_i$ is no more than the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$. One way to achieve this is by choosing the potential function, so the initial potential is zero ($\Phi(D_0) = 0$), and showing that $\Phi(D_i)$ is non-negative for every i:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \underbrace{(\Phi(D_n) - \Phi(D_0))}_{\text{if} \geq 0 \text{ (for any } n)} \leq \sum_{i=1}^{n} \hat{c}_i \Rightarrow \Phi(D_0) = 0 \text{ and } \overbrace{\Phi(D_n) \geq 0}^{\text{for all } n}$$

Thus, with two conditions for picking the potential function, along with the definition of amortized cost, $\hat{c}_i = c_i + \Delta\Phi_i$, where $\Phi : D \to \mathbb{R}^+ \cup \{0\}$, we can use this method to find a tight upper bound.

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, $\Phi(D_0) = \text{number of 1's in } D_0 = 0$ 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.

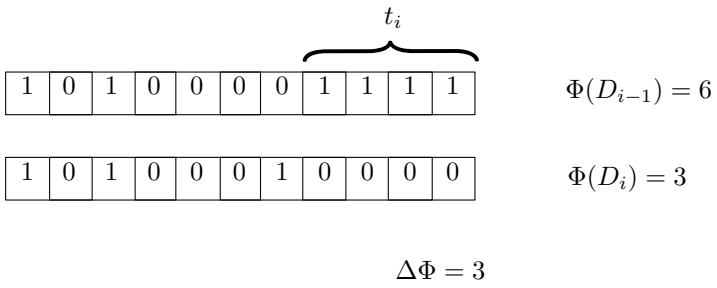


Figure 7: Example of the Φ potential value for the binary counter problem

Thus, the actual cost of the i-th call is $c_i = t_i + 1$ (+1 for flipping 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 $\hat{c}_i = 2 = O(1)$.

Dynamic Array Problem: Analyze the Dynamic Array problem using the potential method.

Recall that to avoid cost of resizing many times, dynamic arrays resize by a large amount. In this case, **Algorithm 2** doubles the array's capacity during resizing. If the algorithm resized the array at $(i+1)$ or an increased size of 1:

$$C_i \;=\; \textstyle\sum_{i=0}^{n-1}(i+1) \;=\; \Theta\left(n^2\right)$$

No amortized analysis will make this runtime better. The only way to make this quadratic runtime better is to increase the resizing of the array (i.e. capacity $\cdot$ 2). Doubling the size of the array lowers the amount of copying needed. The insert operation still runs at constant time. Therefore, we will show that the amortized cost of doing a doubling of the array is: $\hat{c}_i = 3$. Meaning that after n operations, the amortized cost is: $\sum_{i=0}^{n-1}(\hat{c}_i) \;=\; 3n$.

We define our potential as:

$$\Phi(D_i) = (2 \cdot \text{number of entries}) \text{ - } (\text{capacity of A})$$

Let's make sure this satisfies our conditions by starting with an empty array:

$$\Phi(D_0) = (2 \cdot 0) \text{ - } (0) = 0 \;\checkmark$$

We should also check if this satisfies our condition by checking if after the ith operation, our potential is greater than or equal to 0:

$$\Phi(D_i) \geq 0?$$

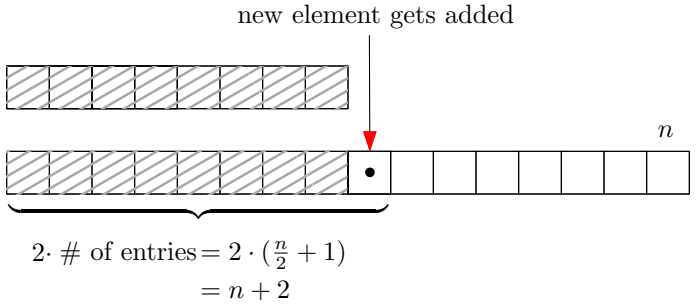


Figure 8: Dynamic array insert after a doubling resize

The dynamic array insert after a doubling resize has 2 $\cdot$ (number of entries) or: $2\left(\frac{n}{2}+\;1\right)$

$$2\left(\frac{n}{2}+\;1\right) = n+2$$

Therefore: $\Phi(D_i) \geq 0$; for any sequence of operations, including or excluding array expansion.

$$\begin{aligned} \Phi(D_i) &\geq 2\left(\frac{n}{2} + 1\right) - n \\ &\geq n + 2 - n \\ &\geq 2 \qquad\qquad\qquad \geq 0 \end{aligned} \tag{1}$$

Now the question is, what is the amortized cost?

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

Case 1: No Array Expansion

$$\begin{aligned} &\hat{c}_i = 1 + (2 \cdot size_i - capacity_i) - (2 \cdot size_{i-1} - capacity_{i-1}) \\ &\hat{c}_i = 1 + 2 - 0 = 3 \end{aligned} \tag{2}$$

Case 2: Triggers Expansion

Actual cost

$$\begin{aligned} c_i &= 1 + capacity(D_{i-1}) \\ \Phi(D_i) - \Phi(D_{i-1}) &= (2 \cdot (D_i) - capacity(D_i)) - (2 \cdot (D_{i-1} - capacity(D_{i-1})) \\ &= 2 - capacity(D_i) + capacity(D_{i-1}) \\ &= 2 - capacity(D_{i-1}) \ // \text{ result of telescoping} \end{aligned} \tag{3}$$

Ammortized cost

$$\begin{aligned} \hat{c} &= c_i + (\Phi(D_i) - \Phi(D_{i-1})) \\ &= 1 + capacity(D_{i-1}) + (2 - capacity(D_{i-1})) \\ &= 1 + 2 - 0 = 3 \end{aligned}$$

The Amortized cost of append in a dynamic array is 3. In other words, n appends cost $3n$ time.

Student Question: What if we squared the array instead? Or perhaps increased the resizing to be greater than double?
Solution: If we resized the array more than double, for example the $size^2$, you will have a large amount of entries that are empty. Of course you will have to copy less as the array grows, but take for instance, size = 2^2, 4^2, 16^2, 256^2, 65536^2, etc. You are wasting space.