CS-GY 6043 Lecture 1 - Amortization

Junior Francisco Garcia (jfg388@nyu.edu)

September 13, 2023

1 Amortization Overview

Amortized analysis can be thought of as a way of providing a guarantee of the average cost required to perform a sequence of operations **TOGETHER**. We started our exploration of this concept by looking at a series of operations, each with a different cost.

- Operation 1: Cost of 3
- Operation 2: Cost of 10
- Operation 5: Cost of 7
- ...

If we are lazy with our analysis and go with a traditional worst-case analysis of the cost, we arrive at the following formula:

$$\text{worst case} = (\#\text{ of ops}) \cdot (\text{worst cost of an op})$$

Although this provides an upper-bound to our cost that is indeed correct, amortized analysis allows us to arrive at the average cost for each operation when considering the entire sequence of operations. This is particularly fitting in situations where the worst possible cost occurs rarely.

In class, we looked at three methods to carry out amortized analysis:

- Accounting method: Overcharges some operations early to compensate for more expensive operations later.
- Aggregate Method: Considers the total cost of a sequence of operations and calculates the average operation cost.
- Potential Method: Uses a potential function to measure the "energy" or "potential" of the data structure, and the change in this potential is used to determine the amortized cost. Derives inspiration from physics and it is the most complicated analysis method.

These methods were explored using three examples:

- Stack
- Bit-flip
- Array Doubling/Dynamic Tables

In this document, I provide my notes on what we covered in class. When applicable, I provide additional resources that helped me with my understanding. Please let me know if my notes have any mistakes that require correction.

2 Example 1: Stack

You have a stack with n elements and you are allowed to perform the following operations on it:

- **push(** x **)**: Inserts an element onto the stack. This costs 1 unit of work.
- **pop()**: Removes an element from the top of the stack. This costs 1 unit of work. We assume that the stack is non-empty.
- **multipop(** k **)**: Removes k top elements from the stack. It follows that $k \leq n$.

When considering a sequence of n push, pop (which can be thought of as **multipop(1)**), and multipop operations, we arrive at the following worst-case cost of the sequence:

$$\begin{aligned} \text{worst case} &= (\#\text{ of ops}) \times (\text{worst cost of an op}) \\ &= n \times n \\ &= n^2 \end{aligned}$$

This estimation is pessimistic since not every operation incurs the maximum cost. Also, some operations could potentially reduce the size of the stack, decreasing the cost for subsequent operations. We assert that a more refined analysis using amortization can yield a tighter bound of $\Theta(n)$ for the average cost, and we use mainly the accounting method to achieve this. We briefly touched on achieving the same result using the aggregate method.

2.1 Accounting Method

We begin our analysis by assigning both a real cost, c_i, and an amortized cost, $\hat{c}_i$, to each operation. The primary goal of our amortized analysis is to ensure that the cumulative real cost across all operations does not surpass the total amortized cost. That is,

$$\sum c_i \leq \sum \hat{c}_i$$

In the accounting method, certain operations are intentionally overcharged to accumulate "credit", which can be used to offset the costs of future operations.

- **push()**: The real cost of the push operation is 1 unit of work. However, it is charged 2 units, so we can use the remaining one later:

$$\begin{aligned} \hat{c}_i &= c_i + 1 \text{ credit (for later)} \\ \hat{c}_i &= 1 + 1 = 2 \end{aligned}$$

- **pop()**: The real cost of the pop operation is 1 unit. Using a credit from a previous push, this cost is offset:

$$\begin{aligned} \hat{c}_i &= c_i - 1 \text{ credit (used up)} \\ \hat{c}_i &= 1 - 1 = 0 \end{aligned}$$

- **multipop(k)**: Multipop is essentially k pop operations. Thus:

$$\begin{aligned} \hat{c}_i &= k \times c_i - k \times 1 \text{ credit (used up)} \\ \hat{c}_i &= k - k = 0 \end{aligned}$$

Given the above, for any sequence of n operations (comprising pushes and pops), the total amortized cost will be at most $2n$ as each operation has an amortized cost of at most 2.

Averaging over n operations, the cost per operation is:

$$\begin{aligned}\text{Average cost per operation} &= \frac{2n}{n} \\ &= 2\end{aligned}$$

which is $O(1)$. Hence, for a sequence of n operations, the average cost per operation remains constant, irrespective of the mix in the sequence. Consequently, the total cost for n operations is $\Theta(n)$, which is the goal we sought to achieve at the beginnning.

2.2 Aggregate Method

Using the aggregate method, we can derive a conclusion similar to what was achieved with the accounting method. Taking into account the actual costs of the operations we discussed:

- **push()**: Requires 1 unit of work.
- **pop()**: Requires 1 unit of work.
- **multipop(k)**: Requires k units of work, with $k \leq n$.

For a sequence of n operations, the worst-case scenario consists of n pushes onto the stack followed by n pops (or multipops) off the stack. Thus, the total cost is:

$$\text{Total cost} \leq n + n = 2n$$

Averaging this cost over all n operations[1], we get:

$$\begin{aligned}\text{Average cost per operation} &= \frac{2n}{n} \\ &= 2\end{aligned}$$

This results in an average cost of $O(1)$, implying that each operation, on average, takes constant time. For the entire sequence of n operations, the total cost is linear, $\Theta(n)$, aligning with the conclusion derived using the accounting method.

3 Bit flip

Consider a binary counter with k bits. In this scenario, we have a single operation available:

- **increment**: This operation increases the counter by 1. The cost of the operation is determined by the number of bits flipped.

For example, if we have a binary counter of 4 bits and we increment the counter, the bit positions that change will depend on the current count:

[1] I equate this to the idea in the accounting method where more expensive operations cover for cheaper ones.

$$
\begin{aligned}
0000 &\to 0001 \\
0001 &\to 0010 \\
0010 &\to 0011 \\
0011 &\to 0100 \\
&\vdots
\end{aligned}
$$

The cost of the increment operation, in the worst case, is k. However, this worst-case scenario occurs rarely. Most of the time, only a single bit is flipped. Using this information, we can derive a conservative estimate for the worst-case cost:

$$
\begin{aligned}
\text{Worst case cost} &= n \times \text{max number of bits flipped} \\
\text{Worst case cost} &= n \times k
\end{aligned}
$$

3.1 Aggregate Analysis

Instead of counting each individual operation, it is beneficial to adopt a bird's-eye view and assess how many times each individual position (or index) of the counter changes.

$$
\begin{aligned}
\text{Bit index } 0 &: \text{flips } n \text{ times} \\
\text{Bit index } 1 &: \text{flips } \frac{n}{2} \text{ times} \\
\text{Bit index } 2 &: \text{flips } \frac{n}{4} \text{ times} \\
&\vdots \\
\text{Bit index } i &: \text{flips } \frac{n}{2^i} \text{ times}
\end{aligned}
$$

This can be represented as the sum:

$$\sum_{i=0}^{k} \frac{n}{2^i}$$

Where k is the number of bits. This sum is less than or equal to:

$$\sum_{i=0}^{\infty} \frac{n}{2^i}$$

Which is equivalent to:

$$n \times (1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots) = 2n$$

Hence, we've amortized our cost from nk to $2n$.

4 Example 3: Array Doubling

In this example, we have an array and we want to store elements in it. We don't know in advance how many elements we will add, so we don't know the array's final size.

We have that:

- A as the array of size S.
- n as the current number of elements.

We aim for S to be as small as possible. This array will only support the following operation:

```
Insert(x):
    if n < S:
        add x to A
    else:
        allocate a new array of size 2S
        copy all data from A into the new array
        let A be the new array
    n = n + 1
```

Most times, the cost of a single 'Insert' operation is 1. However, in cases where we need to increase the size of the array, the cost is $n + 1$ due to copying all existing elements into a new array.

Considering the described operation, the worst-case cost for a single 'Insert' operation is:

$$\text{Worst case cost} = n + 1$$

4.1 Potential Method

We took the challenge in class of arriving at a better-amortized cost using the potential method. To do so, we first must arrive at the different definitions needed to use this method.

For this method, we associate to the data structure the potential we call phi, ϕ. The following are some definitions associated with it:

1. The potential is always non-negative: $\Phi \geq 0$.

2. Φ_0 represents the potential before the first operation, Φ_1 is the potential after the first operation, Φ_2 after the second operation, and so on. Hence, the following is the sequence of potential values after each of the n operations:

$$\Phi_0, \Phi_1, \Phi_2, \ldots, \Phi_n$$

3. We bring back the terms of real cost and amortized cost we introduced in Section 2.1 (c_i and $\hat{c}_i$).

4. The amortized cost of operation i is given by:

$$\hat{c}_i = c_i + (\Phi_i - \Phi_{i-1})$$

where $\Phi_i - \Phi_{i-1}$ represents the change in potential due to the i^{th} operation.

We then summed up over all operations from 1 to n:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi_i - \Phi_{i-1})$$

which simplifies to[2]:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + (\Phi_n - \Phi_0)$$

[2]This occurs because the terms expand to $\Phi_1 - \Phi_0 + \Phi_2 - \Phi_1 + \Phi_3 - \Phi_2 + \ldots + \Phi_n - \Phi_{n-1}$, which is an example of a telescoping series. I had to look this up as it was not straightforward to me [5].

Given that the potential Φ is always non-negative and assuming the initial potential Φ_0 is zero (or any fixed value), the inequality:

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

follows naturally.

4.2 Back to Array Doubling with the Potential Method

Let:

- n = number of elements currently in the array
- s = size of the array

In the potential method, we have to remember that when we do something cheap, we build potential that we can later use in an expensive operation. This is analogous to some operations stacking up credit for other operations in the accounting method.

We were lucky that our professor gave us the potential function for this example. However, I found some interesting CMU notes where they employed a trial-and-error approach to arrive at this. These notes also helped me when scribing the derivations in this document, the potential method explanation in it in particular. [2].

The potential function given was:

$$\Phi = 2n - s$$

For the good case where we don't need to resize the array:

$$\begin{aligned} \text{Real cost, } c_i &= 1 \\ \text{Amortized cost, } \hat{c}_i &= c_i + (\Phi_{i+1} - \Phi_i) \\ &= 1 + (2(n+1) - s) - (2n - s) = 3 \end{aligned}$$

For the bad case where we do need to resize the array:

$$\begin{aligned} \text{Real cost, } c_i &= 1 + n \\ \text{Amortized cost, } \hat{c}_i &= c_i + (\Phi_{i+1} - \Phi_i) \\ &= 1 + n + (2(n+1) - 2s) - (2n - s) = 3 \end{aligned}$$

This method now helped go from a worst-case linear cost per operation to a constant average cost per operation.

4.3 Array Doubling with Aggregate Method

We can also arrive at this result using the aggregate method.

- The cost for each insertion operation is 1.
- Additionally, the array doubling operations require copying existing elements.
- The total number of times we perform the doubling and copying can be broken down as:

  $$1 + 2 + 4 + \ldots + 2^k$$

  because at every doubling operation, we copy 1 element, then 2, then 4, all the way to 2^k. This is represented by the geometric series: $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$.

Given that the following inequality represents the state of affairs before doubling

$$2^k < n < 2^{k+1},$$

we know that, after doubling, we have:

$$2^{k+1} < 2n.$$

. Plugging in definition of the geometric series we arrived at before, we know that

$$2^{k+1} - 1 < 2n - 1$$

This means that the doubling operations is bounded by $2n - 1$. Therefore, the total cost of n operations (both the insertion and the doubling/copying) is:

$$n + (2n - 1) = 3n - 1.$$

When we divide by n to get the average cost per operation, the result is slightly less than 3, confirming that the amortized cost is $O(1)$.

5 Additional Resources

Although this document was generated by mostly copying my hand-written notes into this document, I relied on many resources useful to me when I reverse-engineered the mess that was my notes, some of which were already cited above.

- Jeff Eriksen's chapter on amortization in his book [4].
- Amortization chapter on the fat algorithms book [3].
- Professor Greg Aloupis' notes on algorithms [1].

References

[1] Greg Aloupis. 2023. Algorithms Course Resources. https://research.engineering.nyu.edu/ greg/algorithms/2413/resources.html Course notes and resources for algorithms.

[2] Daniel Anderson and David Woodruff. 2022. 15-451/651: Design Analysis of Algorithms. Lecture 3: Amortized Analysis. https://www.cs.cmu.edu/ 15451-f22/lectures/lec03-amortized.pdf

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3 ed.). MIT Press. Paperback international version ISBN 9780262533058.

[4] Jeff Erickson. 2019. *Algorithms*. Self-published. http://jeffe.cs.illinois.edu/teaching/algorithms/

[5] Wikipedia. 2023. Telescoping series. https://en.wikipedia.org/wiki/Telescoping$_s eries$