Design and Analysis of Algorithms II: Lecture 1 Amortization

Nick Ferrara

September 9th, 2023

1 Introduction

Purpose: Bound the cost of a (long) sequence of operations

As opposed to just calculating the worst-case cost of a sequence of operations as follows:

Worst-case Cost = # of operations $\times$ worst-case cost of each operation

We can obtain a much better upper bound that isn't overly conservative by utilizing amortized analysis.

Types of Amortized Analysis:

- Accounting: This method evaluates the cost of each operation in the context of its downstream effects on subsequent operations. It essentially treats surplus computational costs as a "savings account," depositing extra "funds" when an operation is less demanding. These saved "funds" can then be "withdrawn" to offset the cost of more resource-intensive operations that come later.
- Aggregate: This method pools together the total computational cost for a series of operations and then divides it by the number of operations performed, resulting in an average cost per operation and effectively distributes the computational costs across all the operations as an amortized cost.
- Potential Function: This method models the "potential" of the data structure being used where this potential is accounted for throughout the sequence of operations, allowing for the calculation of an average cost per operation distinct from the real cost.

2 Example: Stack

$\square \rightarrow \square \rightarrow \square \rightarrow \square$

Operations:

- Push(x) $\rightarrow$ 1 unit of work pushing x onto the stack
- Pop() $\rightarrow$ 1 unit of work removing the first element in the stack when stack is non-empty
- MultiPop(k) $\rightarrow$ k units of work removing k elements from the stack where $k \leq$ # of elements in stack

We can calculate a conservative worst-case cost for a sequence of n operations using the method shown above:

Worst-case Cost = # of operations $\times$ worst-case cost of each operation = $n \times n \ = n^2$

Argument: You cannot pop what you have not pushed.

Claim: Worst-case Cost = $\theta(n)$

Accounting Method:

We can save units of work for future operation costs to account for our argument.

Push: \$1 Real Cost + \$1 Deposited Cost = \$2 Amortized Cost
Pop: \$1 Real Cost − \$1 Withdrawn Cost = \$0 Amortized Cost
MultiPop: \$k Real Cost − k × \$1 = \$0 Amortized Cost

Now we can compute an upper bound for the worst-case real cost using the fact that worst-case amortized cost accounts for the actual cost of each operation based on the mechanics of the stack. It's important to note that the amortized cost $\hat{c_i}$ is designed to be an upper estimate of the real cost c_i. This ensures that any "savings" accrued from operations that cost less than their amortized cost will be sufficient to cover operations that may exceed their real cost, thereby preventing an overdraft. We calculate the upper bound using the following inequality where c_i represents the real cost of the ith operation and $\hat{c_i}$ represents the amortized cost of the ith operation:

$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c_i} \leq \sum_{i=1}^{n}$Worst-case Amortized Cost $= n \times 2 = 2n$

Worst-case Cost = $\theta(n)$

3 Example: Binary Counter

We have a k-bit binary number and we would like to calculate the worst-case cost of n increment operations on the binary number where the cost is defined as the number of bit flips that occur.

$\overbrace{0\ 0\ 0\ 0\ 0\ 0 \ldots 0\ 0\ 0}^{k \text{ bits}}$
$\downarrow$ increment cost = 1
0 0 0 0 0 0 ... 0 0 1
$\downarrow$ increment cost = 2
0 0 0 0 0 0 ... 0 1 0
$\downarrow$ increment cost = 1
0 0 0 0 0 0 ... 0 1 1
$\downarrow$ increment cost = 3
0 0 0 0 0 0 ... 1 0 0

Worst-case Increment Cost = k

Again, we can calculate the total cost with the conservative approach:

Worst-case Cost = # of operations $\times$ worst-case cost operation = $n \times k = O(nk)$

Argument: The cost of an increment is based solely on the number of consecutive 1's from the least significant bit.

Claim: Worst-case Cost = $\theta(n)$

Aggregate Method:

We can better understand the frequency of bit flips by considering the base-2 nature of binary numbers. Specifically, each bit position i flips half as often as the one immediately to its right (i-1). This results in a pattern where the rightmost bit (Bit 0) flips every time, the next bit (Bit 1) flips every 2nd time, and so on. This predictable pattern allows us to easily calculate the total number of bit flips.

Bit 0: Flips n times
Bit 1: Flips $\frac{n}{2}$ times
Bit 2: Flips $\frac{n}{4}$ times
$\vdots$
Bit i: Flips $\frac{n}{2^i}$ times

Total Flips = $\sum_{i=0}^{n} \frac{n}{2^i} < n \times \sum_{i=0}^{\infty} \frac{1}{2^i} = n \times \frac{1}{1-\frac{1}{2}} = 2n$

Worst-case Cost = Total Flips = $\Theta(n)$

4 Example: Array Doubling Dynamic Tables

We want to store things in an array, but don't know how many elements are in the array in advance. To address this, it's crucial to manage the array's capacity efficiently so that it doesn't consume significantly more space than the number of elements n it currently holds. We will denote the number of elements in the array as n and the size or capacity in the array as S.

`insert(x)` $\rightarrow$ inserts x into the array with cost $= 1$ when $n < S$ and cost $= n + 1$ when $n \geq S$ since we need to copy the whole array into a new array of size $= 2S$.

Again, we can find a conservative (and frankly, a bad) estimate for the total cost of n insert operations as follows:

Worst-case Cost = # of operations $\times$ worst-case cost operation $= n \times (n+1) = n^2 + n = O(n^2)$

Argument: The cost of an insert operation is not always n+1, as one might think when considering array doubling. The cost spikes only when the array is full and needs to be doubled, which is infrequent. Most insert operations have a constant cost of 1, and the cost of doubling the array is effectively "amortized" over many insert operations.

Claim: Cost $= \Theta(n)$

Aggregate Method:

Total cost = 1 per insert + copying cost if doubling $= 1 \times n$ inserts $+ \sum_{i=0}^{k} 2^i = n + 2^{k+1} - 1 < n + 2n = 3n = \Theta(n)$

Explanation: The reason $2n$ is larger than 2^{k+1} is because 2^k is the number of elements copied on the k-th doubling, but since n insertions caused this final doubling, n must be greater than the previous number of elements in the array $(n-1)$, so if $n > 2^k$, then $2n > 2^{k+1} > 2^{k+1} - 1$.

Worst-case Cost $= \Theta(n)$

Potential Method:

We associate with the data structure a quantity we call the potential function ϕ.

Real Cost: c_i
Amortized Cost: $\hat{c}_i = c_i + \Delta\phi_i = c_i + (\phi_i - \phi_{i-1})$

where ϕ_0 represents the value of ϕ before operation 1 and ϕ_i represents the value of ϕ after operation i.
The next inequality holds true due to the imposed conditions $\phi_0 = 0$ and $\phi \geq 0$, and the telescoping nature of the sum of $\Delta\phi$ values:

$\phi_1 - \phi_0$
$+$
$\phi_2 - \phi_1$
$+$
$\phi_3 - \phi_2$
$+$
$\vdots$
$+$
$\phi_{n-1} - \phi_{n-2}$
$+$
$\phi_n - \phi_{n-1}$

Thus, we are left with $\phi_n - \phi_0 \geq 0$ by previous assumption. Now, we can show that the sum of amortized costs is greater than or equal to the sum of real costs of the n operations:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} [c_i + (\phi_i - \phi_{i-1})] = \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\phi_i - \phi_{i-1}) = \phi_n - \phi_0 + \sum_{i=1}^{n} c_i \geq \sum_{i=1}^{n} c_i$$

Now, let's apply this to the array-doubling example by defining a potential function that works:

$\phi = 2n - S$, where $n = \#$ of elements and $S =$ size of array. Since the size of the array doubles only when n exceeds S, n will always be at least $S/2$, ensuring $\phi \geq 0$.

Good Case: $c = 1$, $\Delta\phi = 2$, $\hat{c} = c + \Delta\phi = 1 + 2 = 3$
Bad Case: $c = n + 1$, $\Delta\phi = 2 - S$, $\hat{c} = c + \Delta\phi = n + 1 + 2 - n = 3$ since S = n on array doubling

Since both cases lead to an amortized cost $\hat{c} = 3$, and the conditions are met, we can utilize the inequality to compute an upper bound on the worst-case real cost:

$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} 3 = 3n$

Worst-case Cost $= \Theta(n)$