CS-GY 6043 Lecture 01 Amortized Analysis

BY DAN MAO

9/7/2023

1 Definition of Amortization

(CLRS P451) Amortized analysis is used to evaluate the performance of an algorithm by averaging the time of a sequence of operations. It provides a average worst-case cost for each operation over the sequence, even if individual operations may be expensive.

2 Three Common Techniques Used in Amortized Analysis

(P451) In class we explore the three primary techniques in amortized analysis: aggregate analysis, the accounting method, and the potential method.

- **Aggregate Analysis**: Calculates an upper bound on the total cost for a sequence of n operations. The amortized cost per operation is found by averaging this total cost.
- **Accounting Method**: Assigns an individual amortized cost to each type of operation. Overcharges some operations early on, storing this as "prepaid credit" on specific objects within the data structure to offset future costs.
- **Potential Method**: Similar to the accounting method but maintains the overcharge as "potential energy" for the entire data structure rather than associating it with individual objects. Given the complexity of the potential method compared to the other two methods, we will delve into its mathematical proofs and detailed explanations in the following section.

2.1 Math Proof of Potential Methods

A. How It Works:

1. We will perform n operations, starting with an initial data structure D_0.
2. For each operation i, the actual cost is c_i and the resulting data structure is D_i.
3. A potential function ϕ assigns a "potential" value to D_i.
4. The amortized cost $\hat{c_i}$ of the i^{th} operation is defined as the sum of real cost and the change in the potential, noted as $\hat{c_i} = c_i + (\phi(D_i) - \phi(D_{i-1}))$

B. **Mathematical Relations:**

We aim to find an accurate upper bound for a sequence of operations. Specifically, we want to know if the sum of amortized cost $\sum_{i=1}^{n} \hat{c_i}$, could serve as an upper bound for the sum of actual cost $\sum_{i=1}^{n} c_i$. So we compute the total amortized cost as the sum of each operation's actual cost and its change in potential, mathematically given by:

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

This relationship impliesd the following:

- If we can define a potential function ϕ so that $\phi(D_n) \geqslant \phi(D_0)$, then the total amortized cost provides an upper bound on the total actual cost.

- In practice, it's uncertain how many operations will be performed. To ensure we pay for computational cost in advance, we usually set the initial potential $\phi(D_0)$ to be 0 and demonstrate that $\phi(D_i)$ is non-negative for all i. This approach guarantees that the cost of each operation is accounted for in advance.

3 Four Examples of Using Amortized Analysis

In class, we discussed four key examples covering various methods:

1. Accounting Method applied to Stack with Multipop
2. Aggregate Method applied to Binary Counting
3. Potential Method applied to Array Doubling
4. Aggregate Method applied to Array Doubling.

3.1 Analyzing the Cost of Slack Operations with Multipop Through the Accounting Method

A. Stack Operations: Overview

We have three primary stack operations:

- `PUSH(S, x)`: Adds an object `x` to stack `S`.
- `POP(S)`: Removes and returns the top object from stack `S`.
- `MULTIPOP(S, k)`: Eliminates the top `k` objects from stack `S`. We assume `k` will not exceed `S.size()`.

B. Traditional Analysis

When estimating the computational cost using traditional methods:

Real Cost of Operations:

- `PUSH`: 1
- `POP`: 1
- `MULTIPOP`: k

Using this conventional approach, the worst-case time complexity of a sequences of n operations appears to be $O(n^2)$. This result is derived from multiplying n (the total number of elements) with the worst-case time complexity of each individual operation. However, this doesn't accurately represent the actual time complexity.

C. Amortized Analysis

To gain a more precise understanding of the time complexity, we turn to amortized analysis. Here's how we defined the amortized cost of each operation:

1. Each element in the stack is operated on a maximum of two times:
    - Once when it is pushed (`PUSH` operation).
    - Once when it is removed, either by a `POP` or `MULTIPOP` operation.

2. Given this observation, we can distribute the cost across operations:

    - Assign a cost of 2 to the `PUSH` operation: one for the actual `PUSH`, and another saved for the eventual removal (either `POP` or `MULTIPOP`).

Amortized Cost of Operations:

- `PUSH`: 2
- `POP`: 0
- `MULTIPOP`: 0

D. Conclusion

Using amortized analysis, we find that the time complexity of the operations sequence is $O(n)$, contrary to the $O(n^2)$ derived from the traditional method.

3.2 Analyze a Binary Counter Using Aggregate Method

A. Binary Counter: Overview

We're tasked with creating a k-bit binary counter that starts at 0 and increments. We use an array $A[0, ..., k-1]$to represent the counter, where each element $A[i]$ holds a bit of the binary number x.

The function `INCREMENT(A)` loops through A, flipping bits from 1 to 0 and stopping to flip a 0 to 1, thus effectively adding 1 to x. Here, Let's review an example for an 8-bit binary counter from textbook (p455).

Example 1. An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

We observe specific patterns in bit-flipping during `INCREMENT` operations: the 0th bit changes with every operation, the 1st bit every second operation, and so on. Specifically, in n increments, each bit $A[i]$ flips $\lfloor \frac{n}{2^i} \rfloor$ times. Knowing exactly how these operations affect the complexity, we can determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The average amortized cost per operation is then $T(n)/n$.

B. Traditional Analysis:

An initial estimate suggests $O(n \cdot k)$ time for n `INCREMENT` operations, based on a worst-case time of $O(k)$ for a single operation.

C. Amortized Time Complexity:

However, not all bits flip in each `INCREMENT` operation. Specifically, in n increments, each bit $A[i]$ flips $\lfloor \frac{n}{2^i} \rfloor$ times. Summing across all bits, the total flips is less or equal to $2n$, yielding a worst-case time complexity of $O(n)$ for n increments.

D. Conclusion:

The amortized cost per `INCREMENT` operation is 2, making it more accurate and efficient than the $O(k)$ initially estimated.

3.3 Analyzing Array Doubling Using the Potential Method

A. Overview

We examine two operations:

- `Add()`: Consumes 1 unit of cost.
- `AddandCopy()`: Consumes 1 unit to copy and, when reaching the current memory allocation limit, triggers a function to copy all elements, costing n units, where n is the number of elements in the array.

B. Traditional Time Complexity Analysis

In a standard analysis, the worst-case time complexity for the `Add` operation is $O(n)$. Repeating this n times results in $O(n^2)$ complexity.

C. Amortized Time Complexity

Based on the mathematical proof outlined in Section 2.1, we intend to identify a potential function ϕ that meets the following criteria:

- $\phi_0 = 0$ initially,
- $\phi_i \geq 0$ for all $i > 0$.

For this problem, we define the potential function as:

$\phi_i = 2 \cdot (\text{number of items in the array}) - (\text{size of the array})$

This function satisfies our requirements, as following:

Case 1: Add Operation

- Actual cost $c_i = 1$,
- Amortized cost $\hat{c_i} = 1 + (\phi_i - \phi_{i-1}) = 1 + (2i - s) - (2(i+1) - s) = 3$.

Case 2: Add and Double Memory

- Actual cost $c_i = i$,

- Amortized cost $\hat{c_i} = i + (2i - s_i) - (2i - 2 - s_{i-1}) = i + 2 - s_{i-1}$, where $s_{i-1} = i - 1$ as it triggers the doubling,
- Therefore, $\hat{c_i} = 3$.

D. Conclusion

The amortized cost per operation is 3 units, yielding an overall time complexity of $O(n)$.

3.4 Analyze of Array Doubling Using the Aggregate Method

A. Overview

The algorithmic operations remain the same as described in Section 3.3.A, except that we analyze them using the Aggregate Method this time.

B. Traditional Complexity Analysis

The worst-case time complexity remain the same as described in Section 3.3.B.

C. Aggregate Analysis

Upon closely examining the process, we observe that creating an array of size i incurs a cost of 1 unit for adding an element. Additionally, doubling the memory requires $2^{\lfloor \log_2(n) \rfloor}$ units of time to copy the remaining elements.

Therefore, the total cost of a sequence of n operations can be represented as:

$$\text{Total Cost} = n + \sum_{i=0}^{\lfloor \log_2(n) \rfloor} 2^i$$

Notice that $\lfloor \log_2(n) \rfloor \leq \log_2(n)$. This allows us to conclude that:

$$\sum_{i=0}^{\lfloor \log_2(n) \rfloor} 2^i \leqslant \sum_{i=0}^{\log_2(n)} 2^i = 2n$$

D. Conclusion

Using the Aggregate Method, we find that the amortized cost per `Add()` operation is $O(1)$. This is a more accurate and efficient representation compared to the initially estimated $O(n)$ time complexity.