AMORTIZED ANALYSIS

Definition:

- An alternative way to evaluate an algorithm, which often use the worst-case scenario analysis, which can be overly pessimistic.
- The core idea is to "average out" the total costs over a sequence of operations
- The difference between amortized analysis and average-case analysis is that amortized analysis doesn't involve probability.
  An amortized analysis guarantees the average performance of each operation in the worst case.
- There are 3 most common techniques used in amortized analysis: aggregate analysis, the accounting method, the potential method

Aggregate Analysis

Aggregate analysis says that for a sequence of n operations that takes $T(n)$ worst-case in total. The worst-case average cost, or amortized cost, for each operation is $T(n)/n$.
This applies to all operations, even if there are different types.

Example 1: Stack operations (Aggregate Analysis)

Consider a augmented stack with 3 possible operations: PUSH, POP, MULTIPOP (k)

- We know that PUSH and POP takes $O(1)$ each. And if we think of MULTIPOP(k) as many POP operations, then the time complexity of MULTIPOP(k) is $O(\min(s,k))$, where s is the number of items on the stack.
- Worst-case analysis:
  If we follow the traditional worst-case analysis. The worst-case time for each operation is $O(n)$, which is the cost of MULTIPOP. And we have n operations, therefore the total cost is $O(n^2)$.
  ⇒ Although this is true, it is not right
- Amortized analysis:
  If we look abit deeper into the relationship between operations, we can see that for an item to be popped, it need to

be pushed into the stack first, so:

$$\text{PUSH\_count} \geq \text{POP\_count} \quad (\text{includes MULTIPOP})$$

Therefore, regardless of how many times you perform MULTIPOP, it will never exceed the number of pushes, which is $O(n)$

$\Rightarrow$ The total cost is now $O(n)$, and therefore the average cost, or amortized cost is $O(n)/n = O(1)$

Conclusions: Even though some operations are costly, if those operations happens infrequently or depends on some conditions, when viewed over a series of operations, they balance out to an average cost that is much lower

Example 2: Increment a binary counter (Aggregate Analysis)

Consider k-bit binary counter that counts from 0. A array $A[0:k-1]$ represents the bits in the counter, where $A[0]$ is the first bit and $A[k-1]$ is the k^{th} bit, such that a binary number x:

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

The pseudo code for INCREMENT is as follows

```
INCREMENT ( A, k )
    i = 0
    while i < k and i == 1:
        A[i] = 0
        i = i + 1
    if i < k:
        A[i] = 1
```

- Worst-case analysis:

  The loop in INCREMENT suggests that the operation worst-case time is $O(k)$, and if we perform this over a sequence of n operations, then the cost is $O(n \cdot k)$

  $\Rightarrow$ This is true, but not tight enough

- Amortized analysis:

  If we look closely into how frequently each bit flips, we can see that:

Counter value (n)	Bits number	Total cost (# flips)
0	0	0
1	1	1
2	1 0	3
3	1 1	4
4	1 0 0	7
5	1 0 1	8
6	1 1 0	10
7	1 1 1	11
8	1 0 0 0	15

We can see a pattern where the first bit A[0] will flips n times for n sequence, the second bit A[1] will flips $n/2$ times, the third bit A[2] will flips $n/4$ times. We can write this as:

$$\text{cost (of n sequence)} = \sum_{i=0}^{K-1} \frac{n}{2^i}$$

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

$$= n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

$$= n \left(\frac{1 - (1/2)^{\infty}}{1 - 1/2} \right) \quad \text{< geometric series ratio} = 1/2 >$$

$$= 2n$$

$\Rightarrow$ The total is therefore $O(2n) \sim O(n)$, which means each operation cost $O(n)/n = O(1)$

Steps to perform Aggregate Analysis

Step 1: Determine actual costs for each operations

Step 2: Using worst-case analysis, calculate the upper-bound, based on the total actual costs over n sequence of operations

Step 3: Using reasoning, show that there is a tighter bound

Step 4: Determine the amortized cost by $T(n)/n$

The accounting method

- You assign differing charges for different operations, they can be more or less than the actual cost of the operations. When the operation is performed, if the amortized cost is larger than the actual cost, the difference is saved in specific objects in the data structure as **credit**, which will be used for operations whose amortized cost is less than the actual cost
- The **credit needs to be non negative** at all time in n operations

$$\text{Total credit} = \underbrace{\sum_{i=1}^{n} \hat{c}_i}_{\text{amortized costs}} - \underbrace{\sum_{i=1}^{n} \hat{c}_i}_{\text{actual costs}}$$

Example 1: Stack operation (the accounting method)

Consider the previous stack operation example, we know the actual cost for the 3 types of operations are:

PUSH = 1

POP = 1

MULTIPOP = min{ S, k }

Let **assume the amortized costs**:

PUSH = 2

POP = 0

MULTIPOP = 0

Now if we pay for a operation using amortized cost:

- PUSH an item on the stack would cost 2, where 1 is the actual cost, and the other 1 is "**saved on top of the item**" as credit.
- When POP/MULTIPOP are performed, the credit on top of the items are used, hence occurs no extra cost.

  ⇒ Since the credit on top of each item never becomes negative as you cannot POP before PUSH, **the amount of credit is always nonnegative**.

Conclusion: We have shown that with the predetermined amortized cost, the amount of credit is always nonnegative ⇒ The total amortized costs upper-bound the actual cost. And total amortized cost ~ O(n)

Example 2: Incrementing binary counter (the accounting method)

Consider the previous example, we know the actual cost of each flip is $O(1)$. Let assume the amortized cost as:

- flip from $0 \rightarrow 1$: \$2
- flip from $1 \rightarrow 0$: \$0

Now lets consider how each operation's amortized cost:

- When a bit is flip from $0 \rightarrow 1$, \$2 is paid, where \$1 is the actual cost, and the other \$1 is "saved on the bit as credit".
- When a bit is flip from $1 \rightarrow 0$, the \$1 saved as credit is used to pay for the operation

$\Rightarrow$ Since we start with all bits being 0, no flip from $1 \rightarrow 0$ occured before flipping $0 \rightarrow 1$. Hence, the credit on top is always nonnegative

Conclusions: For n sequence of operations, the total amortized cost is $O(n)$, since we have shown above that each operation has amortized cost of $O(1)$.

Steps to perform The accounting method

Step 1: Determine actual costs for each operation

Step 2: Assign amortized costs for each operation

Step 3: Show that with the assigned amortized costs, the "credit" is always nonnegative

Step 4: Conclusion on the Total Amortized Cost

The potential method

- Given an intial data structure D_0, define D_i as the resulting data structure after applying the i^{th} operation on D_{i-1}.
- Define a function $\phi(D_i): D_i \rightarrow \mathbb{R}$ as a potential function that maps each data structure D_i to a real number.
- Define the amortized cost as:

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

$\hat{c}_i$: amortized cost; c_i: actual cost; $\phi(D_i) - \phi(D_{i-1})$: change in potential energy

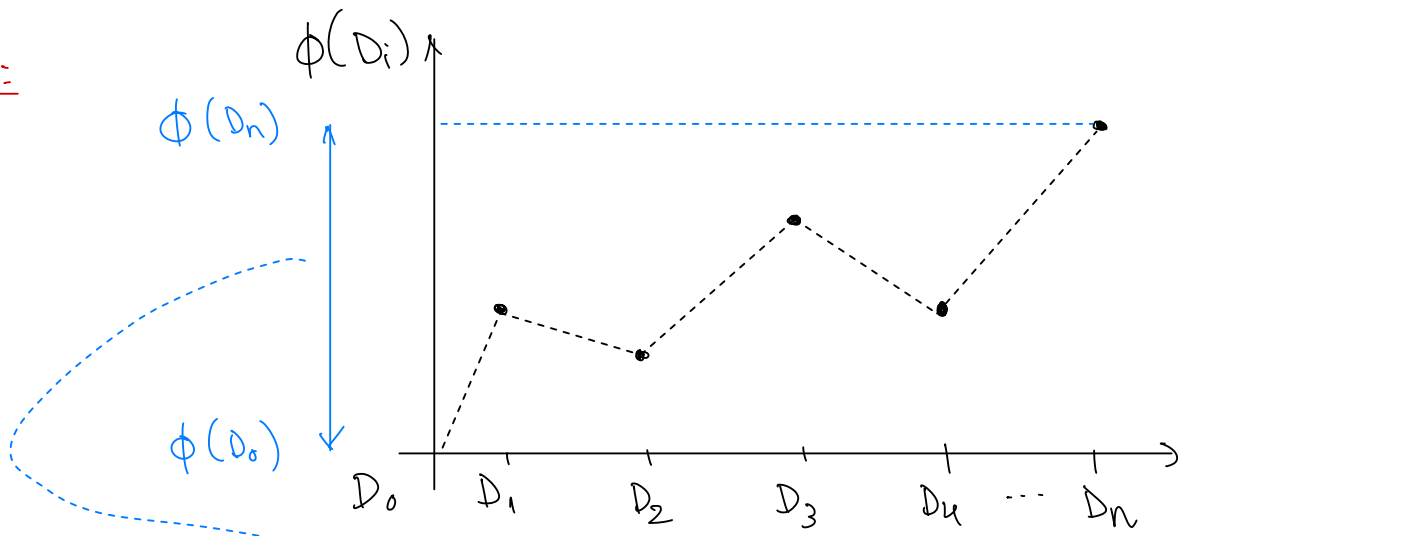
From above definition, the total amortized cost is:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(\hat{c}_i + \phi(D_i) - \phi(D_{i-1}) \right)$$

$$= \sum_{i=1}^{n} \hat{c}_i + \phi(D_n) - \phi(D_0)$$

- The goal is to find the function ϕ such that $\phi(D_n) > \phi(D_0)$, since that would effectively upper-bound $\sum_{i=1}^{n} \hat{c}_i$ (total amortized cost) over $\sum_{i=1}^{n} c_i$ (total actual cost)

Visually:



Total amortized cost = Total actual cost + potential energy

Goal: Find function $\phi(D_i)$ such that potential energy generated is nonnegative

Example 1: Stack operations (the potential method)

Step 1: Define function ϕ, show $\phi(D_i)$ is nonnegative

- Let $\phi(D_i)$ denotes the s number of items on the stack after performing operation i^{th}. For example, $\phi(D_i) = s$
- Since the items on the stack is nonnegative, $\phi(D_i)$ is also nonnegative $\Rightarrow$ Total amortized cost w.r.t ϕ is the upper bound of total actual cost

Step2: Calculate actual cost

We know the actual cost for 3 operations are:

- PUSH = 1
- POP = 1
- MULTIPOP(k) = $\min\{s, k\}$

Step 3: Determine Potential Difference

- PUSH potential difference:

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - (s)$$
$$= 1$$

- POP potential difference:

$$\Phi(D_i) - \Phi(D_{i-1}) = (s-1) - (s)$$
$$= -1$$

- MULTIPOP potential difference:

$$\Phi(D_i) - \Phi(D_{i-1}) = (s-k') - (s), \text{ where } k' = \min\{s, k\}$$
$$= -k'$$

actual cost

Step 4: Calculate amortized cost

- PUSH amortized cost

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= 1 + 1$$
$$= 2$$

- POP amortized cost:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= 1 + (-1)$$
$$= 0$$

- MULTIPOP amortized cost:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= k' - k'$$
$$= 0 \quad \text{where } k' = \min\{s, k\}$$

Step 5: Conclusion on the total amortized cost

Since each operation has amortized cost of either 2 (PUSH) or 0 (POP/MULTIPOP), the total amortized cost over n sequence of operations is $O(n)$.

Example 2: Incrementing the binary counter (the potential method)

Step 1: Determine function ϕ, show that $\phi(D_i)$ is nonnegative

- Let $\phi(D_i)$ be the number of 1-bits after performing the i^{th} operation, denotes b_i
- Since we start at 0, $\phi(D_0) = 0$. After i^{th} operations, the total number of 1-bits in the binary number is always nonnegative, so $\phi(D_i) \geq \phi(D_0) = 0$.

Step 2: Calculate actual cost for each operations

- Flip $0 \rightarrow 1$: $c_i = 1$
- Flip $1 \rightarrow 0$: $c_i = 1$

Step 3: Determine potential difference

- Flip $0 \rightarrow 1$:

$$\phi(D_i) - \phi(D_{i-1}) = (b_i + 1) - b_i = 1$$

- Flip $1 \rightarrow 0$:

$$\phi(D_i) - \phi(D_{i-1}) = (b_i - 1) - b_i = -1$$

Step 4: Calculate amortized cost

- Flip $0 \rightarrow 1$:

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + 1 = 2$$

- Flip $1 \rightarrow 0$:

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + (-1) = 0$$

Step 5: Conclusion on the total amortized cost

Since each operation has amortized cost of either 2 or 0, the total amortized cost is $O(n)$.

Steps to perform The Potential Method

- Step 1: Define ϕ function, show $\phi(D_i)$ is nonnegative.
- Step 2: Determine actual cost of each operation.
- Step 3: Determine potential difference of each operation.
- Step 4: Determine amortized cost of each operation.
- Step 5: Conclude on the total amortized cost

Picking the right potential function ϕ

A good potential function ϕ should satisfy these criteria:

- Capture the cost of the data structure up until that point in time
- Be nonnegative at all time
- Allow for saving energy (potential) during cheaper operations, which can then be utilized to offset the costs of more expensive operations

Fist thing to do: list out all possible operations, and categorize them as cheap / expensive