Amortized Analysis

Often a data structure has one particularly costly operation but it doesn't get performed very frequently. In this scenario the entire mechanism should not be labelled as a costly structure just because that one operation, that is seldomly performed, is costly.

"Amortized analysis" is a method of analyzing the costs associated with a data structure that averages the worst operations out over time.

Essentially amortized analysis is "fair" compared to the other form of analysis we studied earlier. In Asymptotic analysis one bad operation ruin the entire mechanism and termed it as costly even though that operation is performed occasionaly. Therefore in Amortized analysis We want to understand how data structures actually performed in practice, Amortized analysis help us to do that by giving us an accurate description of the data structure over time. Simply looking at the worst case performance per operation can be too pessimistic, & Amortized analysis gives us a clear picture of what's going on.

An Example:

Lets say you want to ~~b~~make a cake, it involes two steps:

1) Mix batter. (fast operation)
2) Bake in an oven (slower operation).

In algorithm analysis approaches what we have studied earlier this cake making process will be termed as a slow operation. ~~But~~ Now, consider you want to bake 100 cake then it involves 100 slow operation & 100 fast operation. So if we take average then it will be the average of the step i & step 2. & Amortized analysis will tell us that this is a "medium" process.

Now if you say that what if we mix batter for 100 cake first & then bake them.

The answer is that, the cake bakeing process is a medium process because mixing cake batter & baking the cake have a logical ordering that cannot be reversed.

Aggregate Analysis :

In aggregate analysis, there are two steps. First, we must show that a sequence of n operation takes $T(n)$ time in worst case. Then we show that each operation takes $T(n)/n$ time, on average. Therefore, in aggregate analysis each operation has the same cost.

A common example of aggregate analysis is inserting an elemt in the hash table (implemented in an array).. Here as we Know array involves static memory allocation, So whenever there is a overflow we will increase that link to double (This problem is also Known as Dynamic Array problem). i.e if the array is full we will double its size.

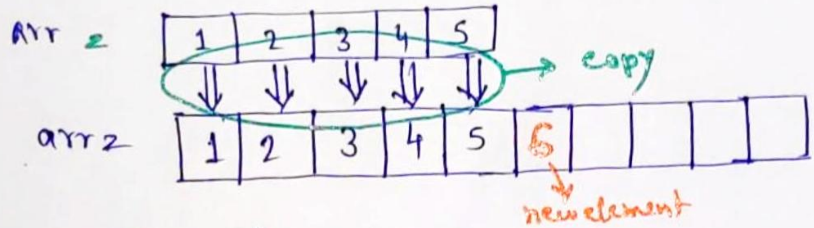
Lets consider that we have an array of size 5, & we inserted 5 element in it

	1	2	3	4	5	
arr =	1					cost = 1
arr =	1	2				Cost = 1
arr =	1	2	3			cost = 1
arr =	1	2	3	4		cost = 1
arr =	1	2	3	4	5	cost = 1

Now there is no room for the 6'th element.
– So we double the size of it.

	1	2	3	4	5	6	7	8	9	10
arr =	1									

Again we can't just double the size, at runtime as it uses complete time memory allocation So we have to create a new array of size 10, then copy all the ~~previous~~ 5 elements into it & then we can insert the newly arrived numbers.



Cost = 10 + 5 + 1
(1) (2) (3)

In case of overflow:-

1) create a new array of double size
2) copy all the element.
3) Insert new element.

Now, if we observe closely then there are two different kind of operation involve in it with different cost:

Cost → 1 (Normal insertion).

Cost → 2 * Size. (we are doubling the size)
+ Size (cost of copying).
3 * Size

In normal worst case analysis, lets say we are inserting N elements so the cost will be - N * 3N (worst case).
= $3N^2$
or $O(N^2)$.

If we analyze it over the time as suggested by Amortized Analysis then, -

(1) [1] cost = 1

(2) [1 | 2] cost = 2 + 1 + 1

(3) [1 | 2 | 3 |] cost = 4 + 2 + 1

(4) [1 | 2 | 3 | 4] cost = 1.

5

1	2	3	4	5			

cost = 8 + 4 + 1

6

1	2	3	4	5	6		

cost = 1

7

1	2	3	4	5	6	7	

cost = 1

8

1	2	3	4	5	6	7	8

cost = 1.

9

1	2	3	4	5	6	7	8	9							

cost = 16 + 8 + 1

10

1	2	3	4	5	6	7	8	9	10						

cost = 1

⋮

16

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

cost = 1.

from the above example we can see that we are performing cheaper operation for more frequent then the ~~ex~~ expansive one.

Now how many time we are doing the expansive operation:

$$3*(1 + 4 + 8 + 16 + \cdots 2N) \quad \ldots\ldots$$

2N is the final size of the array when we finish inserting N elements.

Now, if we look at the above equation we can see that, its a geometric progression & all the elements are power of 2.

So ~~for~~ the last element say $2^l = 2N$
$l = \lg N + 1$

So the total cost =

$$3\,(1 + 4 + 8 + 16 + \cdots + 2N) + \underbrace{(1 + 1 + \cdots\ )}_{\lg N + 1}$$

we can ignore the $\lg N$ part as it is very small ~~compared~~ to the first part.

$= 3*(1+4+8+16+\dots+2N)$

$= 3*(2N*2-1)$

$= 3(4N-1) \simeq 12N.$

$\therefore$ Complexity $= O(N)$.

Therefore using Amortized analysis we can see that the complexity $= O(N)$ not $O(N^2)$ [we computed earlier].

Accounting Method:

This method takes slightly different approach to the problem, Essentially, instead of looking at things in general, we want to jump into look at every single operation. In fact we assign every single operation a cost. Normally to go along with the accounting theme, we'll actually express the cost in dollars.

If we have any surplus it will goes into a bank, the reason why we are doing this is that,

- we need to overcharge for the simpler operations.
- Build up enough savings to afford a more expansive operation later.

Again to lineup ourselves with the accounting theme, the bank balance must always be '0' or greater. We dont want to be in debt, thats the whole idea.

Now lets go back to our idea of dynamic array, the solution will charge 3 dollars for each insertion.

1.

2	2
1	2

Savings = 4\$ [earnings = \$6 / exp = \$4]

2. Now we don't have any space, so we ~~will~~ have to expand it further, & copy the elements into the newly created array of double size.

1	1		
1	2		

Savings = 2\$ [earnings = \$0 / exp = \$2 prev savings = \$4.]

if we insert two elements then, -

1	1	2	2
1	2	3	4

Savings = \$6 [earnings = \$6 / exp = \$2 prev. savings = \$2].

3. Again we need to increase the size of the array.

1	1	0	0				
1	2	3	4				

Savings = \$2 [earnings = \$0 / exp = \$4 prev = \$6].

4. inserting next 4 element -

1	1	0	0	2	2	2	2
1	2	3	4	5	6	7	8

Savings = \$10 [earnings = \$12 exp = 4, prev = \$2]

5. if we want to insert another element.

1	1	0	0	0	0	0	0								
1	2	3	4	5	6	7	8								

Savings = \$2 [earnings = \$0 exp = \$8, prev = \$2]

Therefore, the savings never drops below \$2 no matter how many operation we perform, as long as we charge \$3 for each operation.

Since we know that every operation has a constant cost (3), under Big O notation:

Amortized Time $= O(3) \simeq O(1)$.

The potential Method:

The potential method is similar to the accounting method. However, instead of thinking about the analysis in terms of cost & credit the potential method thinks of work already done as potential energy that can pay for later operations. This is similar to how rolling a rock up a hill creates potential energy that can bring it back down the hill with no effort. Unlike the potential method, however, potential energy is associated with the data structure as a whole, not with the individual operations.

Key difference betn Accounting & potential:

Bank balance of a particular state is dependendent on the the previous state.

The potential Method involves a potential function

- can be used to independenty derive the potential at any state
- can also be used to compute a potential difference which shows the change in cost between two operations.
- The potential is represented by the function $\Phi(h)$

$\Phi(h) \rightarrow$ potential at state h.

In this method finding a potential function is a challenging task. For the dynamic array:

$$\Phi(h) = 2n - \text{Size}$$

(assuming we start from size 1 ~~[illegible]~~).

Lets walk along the method for the same dynamic array problem:

1. when there is 'one' item we have to expand it to 2.

1	

$\therefore \phi(h) = 2n - \text{Size} = 2*1 - 2 = 0$

2. when we have '2' items the array will be resized to '4'

1	2		

$\therefore \phi(h) = 2n - \text{Size} = 2*2 - 4 = 0$

3. when we have '3' items we do actually have a bit of spare potential.

1	2	3	

$\phi(h) = 2n - \text{Size} = 2*3 - 4 = 2$

4. However when we add another item it goes to zero again.

1	2	3	4				

$\phi(h) = 2n - \text{Size} = 2*4 - 8 = 0$

Therefore we alway get a potential energy ≥ 0.

As mentioned earlier, that we can check the potential at any point.

...

382	383	384	385			

...

For example if we at item no. 384 we can simply insert that into the given function to calculate the potential

$$\phi(h) = 2n - \text{Size}.$$
$$= 768 - 512.$$
$$= 256.$$

[512 → closest larger power of 2 is 512]

In order to calculate the amortized time of the i'th operation h_i:

$$c_i + \underbrace{\phi(h_i) - \phi(h_{i-1})}_{\text{Potential.}}$$

$\phi(h_i)$ = potential of the current state
$\phi(h_{i-1})$ = potential of the previous state.
C_i = cost of the operation.

For the dynamic array, we know our potential function

$$\phi(h) = 2n - \text{Size}.$$

To calculate the general case amortized time, we need to consider how the potential function behaves

- We have two cases:
  → The Normal Case
  → The array Expansion case.

① Normal Case:

$$C_i + \phi(h_i) - \phi(h_{i-1})$$
$$= C_i + (2i - \text{Size}) - (2(i-1) - \text{Size})$$
$$= C_i + \cancel{2i} - \cancel{\text{Size}} - \cancel{2i} + 2 + \cancel{\text{Size}}$$
$$= C_i + 2$$

Now since this is a normal case the cost of the operation = 1.

$$= 1 + 2 = 3. = O(3) \approx O(1)$$

This gives us the exact same result when we used the accounting method.

② Expansion Case

$C_i = i + 1$
$i \to$ moving everything.
$1 \to$ Insertion.

$$(i+1) + \phi(h_i) - \phi(h_{i-1})$$

$$= (i+1) + (2i - 2i) - (2(i-1) - i)$$

① array expansion
② after the expansion the size of the array

① array expansion
② size of the array before expansion.

$$= i+1 + 2i - 2i - 2i + 2 + i$$

$$= 1 + 2 = 3. \approx O(1) \approx O(1).$$

There fore whether we expand or not both gives us 3 operations in amortized time, well it's $O(1)$

∴ Accordiy to the potential method, a dynamic array incurs $O(1)$ time (amortized) per operation.

Stack Operation.

Another common example of aggregate analysis is a modified Stack. Stacks are a linear data Structure that have two constant operation.

- Push (e) ⇒ Puts an element on top of Stack
- Pop (e) ⇒ takes top element of the stack and returns it.

both of these operations are constant time so a total of n operation (in any order) will result in $O(n)$ total time.

Aggregate Analysis:

Now, a new operation is added to the Stack. multipop(K) will either pop the top K elements in the stack, or if it runs out of elements before that it will pop all the elements in the Stack & Stop. The pseudocode for multipop(K) would look like:

```
multipop (k){
    while stack not empty & K>0 {
        K = K-1
        stack.pop();
    }.
}.
```

Looking at the pseudocode it's easy to see that this is not a constant time operation. multipop can run for at most n times where n is the size of the Stack so the worst case runtime for multipop is $O(n)$. So, in atypical analysis, that mean that n multipop operations take $O(n^2)$ time.

However, that's not actually the case. Think about multipop & whats it is doing. Multipop can't function until there's been a push to the stack because it would have nothing to pop off.

In fact any sequense of n operations of multipop, pop & push can take at most $O(n)$ time.

multipop the only non-constant time operation in this stack can only take $O(n)$ time if there have also been n constant time push operation on the stack.

In the very wrost case, there are n constant time operation & just one operation taking $O(n)$ time.

For any value n, any sequense of multipop, pop & push takes $O(n)$ time. So using aggrigate analysis,

$$\frac{T(n)}{n} = \frac{O(n)}{n} = O(1).$$

So the stack has amortized cost of $O(1)$ per operation.

Accounting Method:

If we Assign the following cost of each operation, then

$$\left.\begin{array}{l} \text{Push : } 1 \\ \text{pop : } 1 \\ \text{Multipop: } \min(\text{Stack.size}, k) \end{array}\right\}$$

Multipop's cost will either be k if k is less than the number of elements in the stack, or it will be the size of the stack. Assigning amortized costs to those functions we get

$$\left.\begin{array}{l} \text{Push: } 2 \\ \text{pop : } 0 \\ \text{Multipop: } 0. \end{array}\right\} \text{ we will equate 1 cost to \$1.}$$

If we think of the stack as an actual stack of plates this become more clear. pushing a plate onto the stack is the act of placing that plate on top of the stack. & poping is the act of taking top plate off.

So when a plate is pushed onto the stack in this example we pay \$1, for the actual cost of the operation. & we are left with \$1 of the credit. This is because we take amortized cost for push (\$2), subtract the actual cost (\$1) & are left with \$1. we will place that \$1 on the plate we just pushed. So at any point of time, every plate in the stack has \$1 of credit on it.

The \$1 on the plate will act as a money needed to pop the plate off.

Multipop uses pop as subroutine. calling multipop on the stack costs no money, but the pop subroutine within multipop will use the \$1 on top of each plate to remove it. Because there is always \$1 on every plate in the stack, thus credit never will be negative.

The potential Method:

As we've seen earlier, if $\phi(h)$ is the potential function at state h, then, amortized cost of the i operation is defined by,

$$a_i = c_i + \phi(h_i) - \phi(h_{i-1})$$

Here the potential function is size of the stack

$$\phi(h) = \text{Stack.size}.$$

Now,

$$\begin{aligned} a_i &= c_i + \phi(h_i) - \phi(h_{i-1}) \\ &= 1 + (\text{Stack.size} + 1) - \text{Stack.size} \\ &= 1 + 1 = 2 \end{aligned}$$

So the amortized cost of push operation = 2.

Binary Counter

In digital logic & computing, a counter is a device which stores (& sometime displays) the number of times a particular event or process has occured, often in relationship to a clock.

To implement the binary counter we will consider the following pseudocode:

```
Increment () {
    i = 0
    while i < A.length & A[i] == 1 {
        A[i] = 0;
        i = i+1
    }
    if i < A.length
        A[i] = 1.
    }
}
```

Aggregate Analysis:

The following Table describes A after increment has been called a few times.

count	A [4]	A [3]	A [2]	A [1]	A [0]	cost	total cost
0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1
2	0	0	0	1	0	2	3
3	0	0	0	1	1	1	4
4	0	0	1	0	0	3	7
5	0	0	1	0	1	1	8
6	0	0	1	1	0	2	10
7	0	0	1	1	1	1	11
8	0	1	0	0	0	4	15
9	0	1	0	0	1	1	16
10	0	1	0	1	0	2	18
11	0	1	0	1	1	1	19
12	0	1	1	0	0	3	22

If we look at the table then for the first element we changed the bit at A[0] position. Whereas for the second element we changed bit values at A[0] & A[1], Again for the third element only the bits at A[0] is changed. Therefore the cost associated with these numbers are 1, 2 & 1 respectively. Similarly the binary representation & their corresponding cost can be obtained from the above table.

Now, if we use the worst case analysis technique, then the max cost occured for number 8.

- The cost is 4.
- we have 12 number

$\therefore$ worst case complexity can be $O(4*12) = 48$.

~~if we use the steps, and look at each steps~~ if we look at the complexity of individual steps then, the complexity is 22. (as shown in the previous table).

Again, if we take a closer look then in A[0] coloumn the value flips at every time & it ~~follows~~ in the following pattern;

A[0] flips every time $\Rightarrow$ Cost $= n = n$

A[1] flips every 2'nd time $\Rightarrow$ cost $= n/2 = n/2^1$

A[2] flips every 4'th time $\Rightarrow$ cost $= n/4 = n/2^2$

A[3] flips every 8'th time $\Rightarrow$ cost $= n/8 = n/2^3$

A[i] flips every 2^ith time $\Rightarrow$ cost $= n/2^i$

$$\therefore \text{total cost} = n + n/2 + n/4 + n/8 + \dots + n/2^i$$

$$= \sum_{i=1}^{n} n/2^i = 2n = O(n).$$

Accounting Method:

In accounting Methode we will consider the changing of bits from 0 - 1 costs 2 ~~bits~~ & 1 - 0 costs 0. therefore if we go through the table again using accounting method:

count	A[4]	A[3]	A[2]	A[1]	A[0]	cost	total cost
0	0	0	0	0	0	0	0
1	0	0	0	0	1	2	2
2	0	0	0	1	0	2	4
3	0	0	0	1	1	2	6
4	0	0	1	0	0	2	8
5	0	0	1	0	1	2	10
6	0	0	1	1	0	2	12
7	0	0	1	1	1	2	14
8	0	1	0	0	0	2	16
9	0	1	0	0	1	2	18
10	0	1	0	1	0	2	20
11	0	1	0	1	1	2	22
12	0	1	1	0	0	2	24

Here, every time only one bit goes ~~to~~ 0 to 1 therefore ~~if there~~ we are counting for n times the complexity will be,-

$$1 + 1 + 1 + \cdots n = n \approx O(n).$$

Potential Method:

Potential function $\phi(h) =$ no. of 1's in the ~~after~~ at ~~the state~~ in the counter state h.

if i'th operation have t_i bits

then Actual cost $= c_i = t_i + 1$

further $\phi(h_i) = \phi(h_{i-1}) - t_i + 1$

$$\therefore \phi(h_i) - \phi(h_{i-1}) = -t_i + 1.$$

Amortized cost $= c_i + \phi(h_i) + \phi(h_{i-1})$

$$= \cancel{t_i} + 1 - \cancel{t_i} + 1 = 2.$$