* Amortized Analysis

In AA, we average the time required to perform a sequence of data structures operation.

We show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within a sequence might be expensive.

Amortized analysis differs from average case in the sense that it does not involve probability. It guarantees the average performance of each operation in worst case.

We determine upper bound $T(n)$ on the total cost of a sequence of 'n' operations, Average cost per operation is then $T(n)/n$. This is considered as amortized cost for each operation.

Example:- Shop

Item 1 — 500 — 10

Item 2 — 5 — 300

'A'

300×505

1,51,500

Worst case Analysis

(Correct but not tight)

Asymptotic Analysis

'B'

$500 \times 10 + 5 \times 30$

5150

Tight bound.

Amortized Analysis.

1) Aggregate Analysis
2) Accounting method
3) Potential method.

1) Aggregate Analysis -

It assigns same amortized cost to all the operations in the sequence.

$T(n)/n$ → amortized cost for each operation

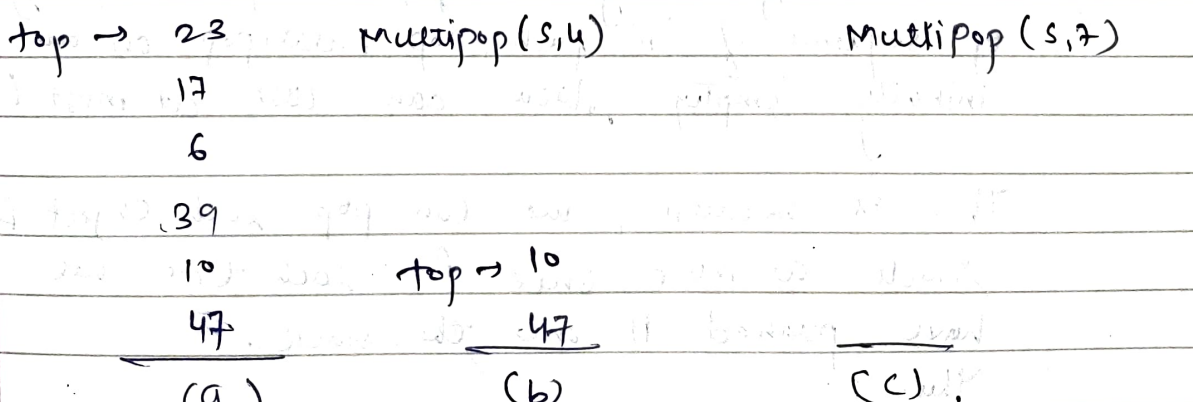
Augumented stack

PUSH (S, x) - pushes object x onto stack S.

POP (S) - pop top of 's', calling pop on empty stack generates error.

Since each of these operation runs in $O(1)$ the total cost of a sequence of n push and pop operation is therefore $O(n)$.

Consider - Multipop (S, k) - Removes top 'k' objects of stack 's'. It pops entire stack if it contains less than 'k' objects. 'k' is positive



```
Multipop (S, k)
1  while not Stack-Empty (S) and k > 0
2      pop (S)
3.     k = k-1
```

Stack-Empty returns true if there are no objects in stack, and false otherwise.

Worst Complexity of Multipop (S, k) is O(n)

∴ 'n' Push, pop, Multipop operations on empty stack.

Worst case time of any stack operation is O(n) and hence a sequence of 'n' operations costs O(n²). — as we may have O(n) Multipop operation costing O(n) each.

Although this is correct, but not tight.

Using aggregate analysis:

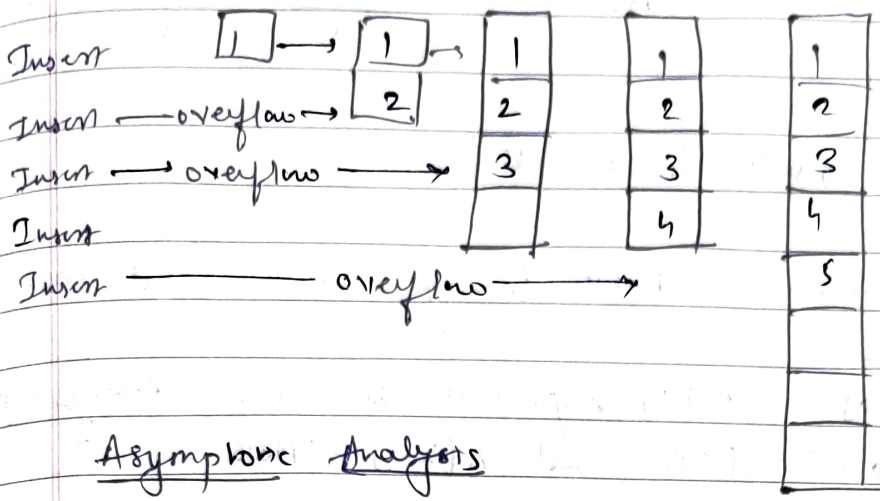
Although single Multipop is expensive, any sequence of 'n' push, pop multipop on an initially empty stack can cost at most O(n).

This is because, we can pop each object from stack at most once for each time we have pushed it onto the stack.

Thus

Amortized cost:- $\frac{O(n)}{n} = O(1)$.

Dynamic Tables



Asymptotic Analysis

Worst case for overflow - $O(n)$ to copy n elements.
for insertion of n elements - $n \cdot O(n) = O(n^2)$.

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

cost of i^{th} operation $= i$ if $i-1$ is exact power of 2
$= 1$ otherwise.

cost of n insert $= \sum_{i=1}^{n} \text{cost}_i$

$= \sum_{i=0}^{\log(n-1)} 2^i + n$.

$2^{\log(n-1)+\log 2}$
$= 2^{\log 2(n-1)}$
$= 2(n-1)$

$= \left[1 + 2 + 4 + \cdots + 2^{\log(n-1)}\right] + n$

$= 1 \cdot \frac{\left(2^{\log(n-1)+1} - 1\right)}{2-1} + n$,

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

$= O(n)$

Amortized cost of each operation $= \frac{T(n)}{n} = \frac{O(n)}{n} = O(1)$

* Binary counter increment

Consider a k-bit binary counter that counts from 0.

Array A[0] --- A[k-1] represents bits where A.length = k.

A[0] is LSB and A[k-1] = MSB.

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

Initially, x=0 so A[i]=0 for i = 0 -, k-1

```
INCREMENT (A)
1   i=0
2   while i < A.length and A[i]==1
3       A[i]=0
4       i=i+1
5   if i < A.length
6       A[i]=1
```

Thus, in worst case, increment takes time O(k) where all elements of A are 1.

Thus, a sequence of 'n' Increment on an initially zero counter takes -

O(nk) in worst case.

This is correct but not tight.

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

Amortized Analysis -

As seen in above table -

A[0] flips each time Increment is called.

A[1] flips every other time i.e.

A[1] flips $\lfloor n/2 \rfloor$ times.

Similarly

A[2] flips every fourth time - $\lfloor n/4 \rfloor$ ...

Thus -

for $i=0 \rightarrow k-1$, $A[i]$ flips $\lfloor n/2^i \rfloor$ times.

for $i \geq k$, $A[i]$ does not exist so it cannot flip.

Total no. of flips $= \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor$

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

$= \underline{2n}$

Thus worst case for a sequence of 'n' Increment operations on an initially zero counter is therefore $O(n)$

Amortized cost of each increment $= \frac{O(n)}{n} = O(1)$.

15/1/2023

* Accounting Method.

Here, we assign different charges to different operations, with some operations charged more or less than its actually costs.

We call this amount as amortized cost

When an operation's amortized cost exceeds its actual cost, we assign this difference as Credit.

This credit can be used for later operations whose amortized cost is less than their actual cost.

The amortized cost should be chosen carefully If we want to show that in the worst case, the average cost of operation is small using amortized costs, we must ensure -

"Total amortized cost of a sequence of operations must provide upper bound on total actual cost of sequence

Let us denote the actual cost of the i^{th} operation by C_i and the amortized cost of the i^{th} operation by $\hat{C}_i$, Then -

$$\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i$$ for all sequence of 'n' operations.

Total credit stored will be -

$$\text{Credit} = \sum_{i=1}^{n} \hat{C}_i - \sum_{i=1}^{n} C_i$$

we must take care that the total credit in the data structure never becomes negative.

* Stack operations

Recall, the actual costs of operations are -

PUSH — 1
POP — 1
MULTIPOP — min(k, s)

where k = no. of objects to be popped
s = size of stack when called
(ie, no. of elements in stack)

Let us assign following amortized cost -

PUSH — 2
POP — 0
MULTIPOP — 0

And analyse a sequence of ~~70~~ 8 PUSH, POP and MULTIPOP operation on initially empty stack.

Sequence - push, push, ~~pop~~ push, ~~push~~, pop, pop, push, multipop(s, 1), ~~pop~~, pop.

* Some assumptions - (Rasonale)

- Since we start with empty stack, pushes must be done first and thus build the credits
- All pops are charged against credit,
- There can never be more pops (of either type) than pushes.

operations	Amortized cost	Actual cost	Credit
push	2	1	1
push	2	1	2
pop	0	1	1
pop	0	1	0
push	2	1	1
multipop(S,1)	0	1	0
push	2	1	1
pop	0	1	0
Total Amortized cost	8		

Total amortized cost = (2+2+0+0+2+0+2+0)

$T(n) = 8$

$= 2 \times 4$ (for 4 push operations)

$= 2 \times n$ (for 'n' push operations)

$T(n) = 2n$

$T_n = O(n)$

* Incrementing Binary Counter

Here, we take following Amortized cost -

For setting a bit to 1 → 2

For setting a bit to 0 → 0

Actual cost = No. of bits flipped.

Counter value	A[2]	A[1]	A[0]	Amortized cost	Actual cost	Credits
0	0	0	0	0	0	0
1	0	0	1	2	1	1
2	0	1	0	2+0	2	1
3	0	1	1	2	1	2
4	1	0	0	0+0+2	3	1
5	1	0	1	2	1	2
6	1	1	0	0+2	2	2
7	1	1	1	2	1	3
0	0	0	0	0	3	0

Total Amortized cost = 14.

$$T(n) = \sum_{i=1}^{n} \hat{C}_i \quad \text{for } n \text{ increments}$$

$= (0+2+2+2+2+2+2+2)$ for 7 increments

$= 14$

$= 2 \times 7$ for 7 increments

$T(n) = 2n$ — for 'n' increments

$$\boxed{T(n) = O(n)}$$

* Dynamic Tables

Consider following amortized cost - ($\hat{c_i}$)
for inserting each element in table — 3
for copying each element in new table - 0

Actual cost = n if (n-1) is exact power of 2
(c_i) = 1 otherwise.

Insert element	Amortized cost	Actual cost	Credit
1	3	1	2
2	0+3	2	3
3	0+0+3	3	3
4	3	1	5
5	0+0+0+0+3	5	3
6	3	1	5
7	3	1	7
8	3	1	9
9	0+0+0+0+0+0+0+0+3	9	3.
10	3	1	5

$$T(n) = \sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{10} \hat{c_i} = 3+3+3+3+3+3+3+3+3+3$$

$= 30$ — for 10 insertions.

$T(n) = 3 \times 10$ — for 10 insertions
$= 3n$ — for n insertions

$$\boxed{T(n) = O(n)}$$

Potential Method

Potential method represents prepaid work as "potential energy" or just "potential".

Potential works as follows-

We perform 'n' operations — starting from data structure D_0.

for each $i = 1, 2, 3, \ldots, n$

Let c_i = actual cost of i^{th} operation

D_i = data structure that results after applying i^{th} operation to DS. D_{i-1}.

Potential fuction ϕ maps each data structure D_i to a real number. $\phi(D_i)$ which is potential associated with data structure D_i.

Amortized cost $\hat{c}_i$ for i^{th} operation w.r.t function ϕ is defined as.-

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

Thus, amortized cost of each operation is thus its actual cost plus change in potential due to operation.

Thus, $$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} [c_i + \phi(D_i) - \phi(D_{i-1})].$$

$$= \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0).$$

For the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ to give an upperbound on total actual cost $\sum_{i=1}^{n} c_i$, we need to define potential ϕ such that $\phi(D_n) \geq \phi(D_0)$.

To make sure that we pay in advance, we require that $\phi(D_i) \geq \phi(D_0)$ for all i.

We usually define $\phi(D_0) = 0$ and then show $\phi(D_i) \geq 0$ for all i.

If $\phi(D_i) - \phi(D_{i-1})$ is positive $\Rightarrow$ $\hat{c}_i$ represents an overcharge.

If $\phi(D_i) - \phi(D_{i-1})$ is negative $\Rightarrow$ $\hat{c}_i$ represents undercharge.

Amortized cost depend on the choice of potential fuction ϕ. Different fuction may yeild different amortized cost yet be upper bound on the actual cost.

* Stack operations

We define –

Φ ⇒ no. of objects in the stack.

Thus for an empty stack D_0, we have $\Phi(D_0) = 0$ Since the no. of elements in stack is never negative, stack D_i that results after i^{th} operation has non-negative potential.

$$\Phi(D_i) \geq 0$$
$$= \Phi(D_0).$$

Thus, we have an upperbound on actual cost by total amortized cost using Φ.

Amortized cost of various operations –

PUSH –

Let stack contains 's' objects. If i^{th} operation is PUSH, then potential difference would be –

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

Thus –

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

Multipop -

Let stack contains 's' objects. If i^{th} operation is MULTIPOP(S, k) then min (k, s) objects would be popped off the stack.

So let $k' = \min(s, k)$.

$$\begin{aligned}\phi(D_i) - \phi(D_{i-1}) &= \\ &= (s - k') - s. \\ &= -k'\end{aligned}$$

Amortized cost -

$$\begin{aligned}\hat{C}_i &= C_i + \phi(D_i) - \phi(D_{i-1}) \\ &= k' - k' \\ &= 0.\end{aligned}$$

POP -

Similar argument as above -

If i^{th} operation is pop -

$$\begin{aligned}\phi(D_i) - \phi(D_{i-1}) &= (s-1) - s \\ &= -1\end{aligned}$$

$$\begin{aligned}\hat{C}_i &= C_i + \phi(D_i) - \phi(D_{i-1}) \\ &= 1 - 1 \\ &= 0\end{aligned}$$

Thus, amortized cost of each of the three operation is O(1). Thus, total amortized cost of a sequence of 'n' operation is O(n).

* Incrementing Binary Counter.

Define ϕ = b_i after i^{th} operation
= no. of 1s in counter after i^{th} operation.

Amortized cost of an increment operation -

i^{th} increment resets t_i bits. ($1 \to 0$)

C_i = Actual cost = $t_i + 1$ (at most) set at most 1 bit.

If $b_i = 0$ then i^{th} operation reset all k-bits.
so $b_{i-1} = t_i = k$.

If $b_i > 0$ then

$$b_i = b_{i-1} - t_i + 1$$

In either case - $b_i \leq b_{i-1} - t_i + 1$

$$\phi(D_i) - \phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1}$$
$$= 1 - t_i$$

Amortized cost -

$$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1})$$
$$= (t_i + 1) + (1 - t_i)$$
$$= 2$$

* **Dynamic Table -**

Define $\phi = 2n(T) - s(T)$

$\phi(D_i) = 2n(D_i) - s(D_i)$

where $n(D_i)$ = no. of elements present after i^{th} operation

$s(D_i)$ = size of data structure after i^{th} operation

Initially, when table is empty -

$n(D_0) = 0 \quad s(D_0) = 0$

$\therefore \phi(D_0) = 0$

Let 'm' be size of table at any point of time. Here $m/2$ elements will be copied from previous table. Thus when we perform i^{th} insertion total elements $= m/2 + 1$ & size $= m$.

$$\phi(D_i) = 2\left(\frac{m}{2}+1\right) - m = m + 2 - m = 2 > 0$$

When i^{th} element to be inserted is $< m/2$ then table size would be at most $m/2$

Thus $\phi(D_i) = 2\left(\frac{m}{4}+1\right) - \frac{m}{2} = \frac{m}{2} + 2 - \frac{m}{2} = 2 > 0$

Thus $\phi(D_i) \geq \phi(D_0)$ for all i.

If $n(D_i) = s(D_i)$ then $\phi(D_i) = n(D_i)$

Also $n(D_i) = s(D_i)/2$ then $\phi(D_i) = 0$.

Actual cost

$c_i = i$ if $(i-1)$ is exact power of 2

$= 1$ otherwise.

Case 2 :- Let i^{th} insertion does not trigger expansion (means it is not worst case)

Here - $s(D_i) = s(D_{i-1})$ and
$n(D_i) = n(D_{i-1}) + 1$

$c_i = 1$.

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$
$$= 1 + [2n(D_i) - s(D_i)] - [2n(D_{i-1}) - s(D_{i-1})]$$
$$= 1 + 2[n(D_{i-1}) + 1] - s(D_i) - 2n(D_{i-1}) - s(D_i)$$
$$= 1 + 2$$
$$= 3.$$

Case 1 When i^{th} operation triggers expansion (ie. we copy $(i-1)$ elements to new table and then insert i^{th} element)

Here - $s(D_{i-1}) = s(D_i)/2$
$n(D_{i-1}) = n(D_i) - 1$

$c_i = n(D_i)$

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$
$$= c_i + [2n(D_i) - s(D_i)] - [2n(D_{i-1}) - s(D_{i-1})]$$
$$= n(D_i) + 2n(D_i) - s(D_i) - 2(n(D_i) - 1) + \frac{s(D_i)}{2}$$
$$= n(D_i) + 2 - s(D_i)/2$$

But $n(D_i) = s(D_i)/2 + 1$

$$\hat{c}_i = 1 + 2 + s(D_i)/2 - s(D_i)/2$$
$$\hat{c}_i = 3$$

Thus, amortized cost for each insertion operation is 3 (in both cases)
So, for 'n' insertions we have $3n = O(n)$

Alternative Potential fuction.

$$\phi(D_i) = 2i - 2^{\lceil \log i \rceil}$$

Case 1 when $i-1$ is exact power of 2.

$$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1})$$
$$= i + 2i - 2^{\lceil \log i \rceil} - [2(i-1) - 2^{\lceil \log (i-1) \rceil}]$$
$$= i + 2 - 2^{\lceil \log i \rceil} + 2^{\lceil \log (i-1) \rceil}$$

For ex

Let $i = 9$ as $i-1 = 8$ is exact power of 2

$$2^{\lceil \log i \rceil} = 2^{\lceil \log 9 \rceil} = 2^4 = 16 = 2(i-1)$$

$$2^{\lceil \log (i-1) \rceil} = 2^{\lceil \log 8 \rceil} = 2^3 = 8 = (i-1)$$

Thus $\hat{C}_i = i + 2 - 2(i-1) + i - 1$
$= 4 - 1$
$= 3$

Case 2 $\hat{C}_i = 1 + 2i - 2^{\lceil \log i \rceil} - [2(i-1) - 2^{\lceil \log (i-1) \rceil}]$
$= 1 + 2 - 2^{\lceil \log i \rceil} + 2^{\lceil \log (i-1) \rceil}$

when $(i-1)$ is not exact power of $2 \Rightarrow 2^{\lceil \log i \rceil} = 2^{\lceil \log (i-1) \rceil}$

$\hat{C}_i = 1 + 2 = 3$