Page No :
Date :

Asymptotic Notation

→ It refers to defining the mathematical boundation/framing of its run time performance.

→ Asymptotic analysis is input bound i.e, if there is no input to the algorithm, it is concluded to work in a constant time.

→ It refers to Computing the running time of any operation in mathematical units of computation.

Eg: the ~~com~~ running time of one operation is computed as $f(n)$ & may be for another Operation is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in n & the running time of the Second operation will increase exponentially when n increases. Significantly, the running time of both Operations will be nearly the same if n is significantly small.

Usually the time required by an algorithm falls under three types:

- Best Case - Minimum time Required
- Average Case - Average time Required
- Worst Case - Maximum time required.

Page No :
Date :

Asymptotic Notation

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

1) Big-Oh Notation (O)

O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time of an algorithm can possibly take to complete.

2) Omega Notation (Ω)

Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time of an algorithm can possibly take to complete.

3) Theta Notation (Θ)

Θ(n) is the formal way to express both the

O(1) ⟶ Constant
O(n) ⟶ linear

Page No :
Date :

Lower & upper bound of an algorithm's running.

14/12

Amortised Analysis

- Amortised analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In amortised analysis, we analyze a sequence of operation & guarantee a worst case average time which is lower than the worst case time of a particular expensive Operation.
- With amortised analysis, the average cost of an operation is small, if we average over a sequence of operations.
- An amortised analysis guarantees the average performance of each operation in the worst case.

Let us consider an example of a simple hash Table insertions. There is a trade off b/w Space & time, if we make hash table size big, search time becomes fast, but Space required becomes high.

The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to

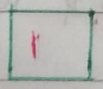
Page No:
Date:

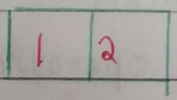
increase size of table whenever it becomes full. Following are the steps to follow when tables become full.

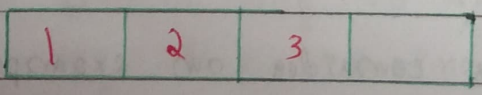
* Allocate memory for a larger table of size, typically twice the old table.
* Copy the contents of old table to new table.
* Free the old table.

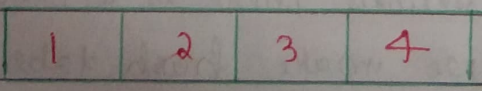
If the table has space available, we simply insert new item in available space.

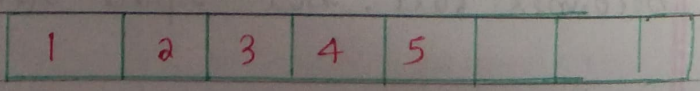
Initially table is empty & size is 0.

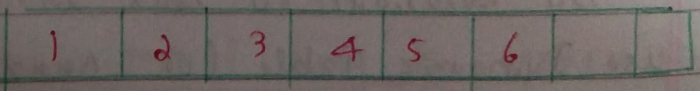
Insert Item 1 

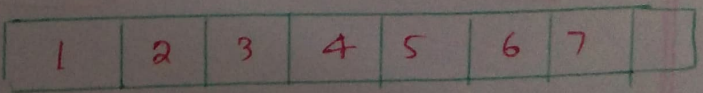
Insert Item 2 

Insert Item 3 

Insert Item 4 

Insert Item 5 

Insert Item 6 

Insert Item 7 

If we use simple analysis, the worst ~~case~~ case cost of an insertion is $O(n)$. ∴ The worst case cost of n inserts is $n * O(n)$ which is $O(n^2)$. This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take (n) time.

Item No :	1	2	3	4	5	6	7	8	9	10	. .
Table Size :	1	2	4	4	8	8	8	8	16	16	.
Cost :	1	2	3	1	5	1	1	1	9	1	

$$\text{Amortised Cost} = \frac{(1+2+3+1+5+1+1+1+9+1)}{n}$$

$$= \frac{[(1+1+1+1)\cdots+(1+2+4+\cdots)}{n}$$

$$= \frac{[n+2n]}{n} = 3 = O(1)$$ //

* Amortised Analysis is applied to algorithms where an occasional operation is very slow, but most of the other operations are faster.

* In Amortised Analysis, we analyse a sequence of operations & guarantee a worst case average time which is lower than the worst case time of a particular Expensive Operation.

Page No :
Date :

* It is upper bound : It is the average performance of each Operation is the worst case. It is concerned with the over all cost of a Sequence of Operations. It does not say anything about the cost of a Specific operation in that Sequence.

* It can be understood to take advantage of the fact that some expensive operations may pay for future operations by somehow limiting the number or cost of expensive operations that can happen in the near future.

* It may consist of a collection of cheap, less expensive & expensive operations, however, amortised analysis will show that avg cost of an Operation is cheap.

* This is diff. from average case analysis wherein averaging argument is given over all Inputs for a Specific Operation. Inputs are modeled using a suitable probability distribution.

1) Aggregate
2) Accounting
3) Potential Method

Page No :
Date :

• Aggregate Method

→ The aggregate method is used to find the total cost.

→ ~~used~~ where the total running time for a sequence of operations is analyzed

→ It computes the worst case time. T(n) for a sequence of n operations.

→ It gives an average performance of each operation in the worst case.

→ This method is less precise than other methods as all operations are assigned the same cost.

Application : Stack Operation

In the following pseudocode, the operation STACK-EMPTY returns TRUE if there are no object currently on the stack & FALSE otherwise.

```
MULTIPOP(S, K)
    while (.NOT.STACK-EMPTY(S) and K ≠ 0)
        do pop(S)
           K = K - 1
```

i) Worse-case cost for MULTIPOP is O(n). There are n successive calls to MULTIPOP would cost

$O(n^2)$. We get unfair cost $O(n^2)$ because each item can be poped only once for each item it is pushed.

ii) In a sequence of n mixed operations, the most times multipop can be called n/2. Since the cost of push & pop is O(1), the cost of n stack operation is O(n). ∴, amortised cost of an operation is the average: $O(n)/n = O(1)$

15/12/2020

Application 2 - Incrementing a binary counter

We use an array A[0..k-1] of bits, where length[A] = k, as the counter that counts upward from 0. A binary number x that is stored in the counter has its lowest-order bit in A[0] & its highest-order bit is A[k-1], so that $\sum_{i=0}^{k-1} 2^i A[i]$.

Initially, x=0 & thus A[i] = 0 for i = 0, 1 .. k-1

To add 1 (modulus 2^k) to the value in the counter,

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

A Single Execution of INCREMENT takes O(k) in worst case when Array A contains all 1's. Thus, a Sequence of n INCREMENT operation on an initially zero counter takes O(nk) in the worst case. This bound is correct but not tight.

Rowno	3	2	1	0
1	0	0	0	0
2	0	0	1	0
3	0	1	0	0
4	0	1	1	0
5	1	0	0	0
6	1	1		
7				
8				
9				
10				

Page No.:
Date:

A[i] ... A[2] A[1] A[0]

Row	3	2	1	0
1	0	0	0	0
2	0	0	0	1
3	0	0	1	0
4	0	0	1	1
5	0	1	0	0
6	0	1	0	1
7	0	1	1	0
8	0	1	1	1
9	1	0	0	0
10	1	0	0	1
11	1	0	1	0
12	1	0	1	1
13	1	1	0	0
14	1	1	0	1
15	1	1	1	0
16	1	1	1	1

Amortized Analysis

Observations: Not all bits flips each time interval is called.

A[0] flips each time interval is called.

A[1] flips every other time $n/2$

A[2] flips $n/4$ times

A[i] flips n/i times

In general, for $i = 0, 1 \ldots \log n$, bit $A[i]$ flips $n/2^i$ times in a sequence of n INCREMENT operations on an initially zero counter.

For $i > \lg(n)$, bit A_i never flips at all. The total no. of flips in a sequence is

$$\sum_{i=0}^{\lceil \log n \rceil} \left\lfloor \frac{n}{2^i} \right\rfloor < n \times \sum_{i=0}^{\infty} \frac{1}{2^i} < n \times \frac{1}{1 - 1/2} < 2n$$

$$2n = O(n)$$

Therefore, the worst case time for a sequence of n INCREMENT Operation on an initially zero counter is therefore $O(n)$, so the amortised cost of each operation is $O(n)/n = O(1)$.

17/12

• Accounting Method

credit - Difference b/w Amortised cost & actual cost stored is specific objects

- Here we expose an extra charge on inexpensive Operations & use it to pay for expensive Operations later on.
- also known as Banker's Method.
- Different charges to different operations are assigned. Some Operations are charged more or less than they actually cost.

- When an operation's amortised cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as credit.

- Credit can be used ~~for~~ later on to help pay for operations whose amortised cost is less than their actual cost.

- One must choose the amortised costs of operations carefully. The total credit in the D.S. should never become negative, Otherwise the total amortised cost would not be an upper bound on the total actual cost.

Application 1 : Stack Operation

The actual costs of the Operations were

Push 1 Pop 1

Multipop $\min(k, s)$

where k is the argument supplied to Multipop & s is the stack size when it is called.

We assign the amortised costs :

Push 2 Pop 0

Multipop 0

Page No :
Date :

Here all three amortized costs are O(1), although in general the amortised costs of the operations under consideration may differ asymptotically.

* For push operation, we pay the actual cost of the push & token & are left with a credit of 1 token out of 2 tokens charged, which we put on top of the plate.

* When we execute a pop operation, we charge the operation nothing & pay its actual cost using the credit stored in the stack. Thus by charging the push operation a little bit more, we needn't charge the pop operation anything.

* We needn't change the multipop operation anything either. We have always charged at least enough up front to pay for the multipop operations.

* Thus, for any sequence of n Push, pop & Multi pop operations, the total amortised cost is an upper bound on the total actual cost. Since the total amortised cost is O(n), so is the total actual cost.

0 ⟶ 1 : $2

1 ⟶ 0 : $0

Page No :

Date :

Example 2: Incrementing a binary Counter

When a bit is set, we use 1 dollar to pay for the actual setting of the bit & place the other dollar on the bit as credit to be used later when we flip the bit back to 0.

Every "1" has a dollar of credit on it, thus we need not charge anything to reset a bit to 0.

Credit never becomes negative.

In each operation, at least one bit is set: cost 2 dollars. Total cost for n operations : O(n).