(DAA) Design And Analysis Of Algorithms.

unit-I: Introduction to Algorithms.

fundamentals of algorithmic problem solving, analysis of framework, Performance Analysis:- Space complexity, Time complexity, Growth of functions:- Asympiotic notation, Big oh notation, Omega notation, Theta notation, Little oh.

unit-II: Divide And Conquer.

Divide And Conquer: General method, applications - Binary Sear Quick Sort, Merge Sort, finding the maximum and minimu

unit-III: Greedy Method.

The General Method, Knapsack Problem, Job Sequence with Deadlines, Minimum and Spanning Trees, Prim's Algorithm, Kruskal's Algorithm, Optimal Merge Patterns, Single Source shortest Paths.

unit-IV: Dynamic Programming.

General method, applications- Matrix chain multiplication, Optimal binary search trees, 0/1 knapsack Problem, All pairs shortest path problem, Travelling sales person problem, Reliability design.

unit-V: Back tracking

General method, applications- n-queen problem, sum of subsets problem, graph coloring, Hamiltoniancycles.

Branch & Bound: General method, applications - Travelling sales person problem, 0/1 Knapsack Problem, - LC Branch & Bound solution, FIFO Branch & Bound Solutions.

Text Books:

fundamental of Computer algorithms, Ellis Horowitz, Satraj Sahni and Rajasekaran, University Press.

Reference Books:

1. Introduction to the design and analysis of Algorithm - 3rd Editi Anany Levitin, Pearson Education, 2017.
2. Introduction to Algorithms - 2nd edition, T.H Cormen, C.E. Leisers RL. Rivest, CS. Stein, PHI, Pvt. Ltd/ Pearson Education.
3. Design and Analysis of Algorithms - Aho, Ullman and Hopcroft, Pearson Education.
4. Algorithms - Richard Johnson Baugh & Marcus Schedules, Pearson Education.

Unit-I: Introduction to Algorithms.

Algorithm: An algorithm is a sequence of computational steps to solve a problem. to complete a task.

(or)

An algorithm is a set of instructions that are carried out in a specific order to perform a specific task.

Pseudo Code: Pseudo code is an artificial language and informal that helps programmers to develop an algorithm.

Eg: Addition of two numbers.

1. start
2. Read a, b
3. add a, b
4. store result in c
5. print c
6. stop.

Eg: Average of 3 numbers.

1. start
2. Read a, b, c
3. add a, b, c
4. divide sum by no. of variab
5. store the result in d
6. stop.

characteristics (or) Properties of an algorithm:

1. **Input / output:**
   Each algorithm must take 0, 1 (or) more quantities as i/p and should produce atleast one o/p.
2. **fitness:** An algorithm should terminate in a finite no. of steps.
3. **Definiteness (or) unambiguous:** Each step of algorithm must be clear (i.e, it should not contain ambiguity)
4. **Effectiveness:**
   Algorithm should contain only necessary steps it should not contain any unnecessary statements.
5. **Generality:** Algorithm should run for any type of i/p (or) data.

Conventions of Algorithms:

1. An algorithm is a procedure, it has two parts first part is head and second part is body.
2. The head section consists of keyword algorithm and name of the algorithm with parameter list.

   Eg: Algorithm name ($P_1, P_2, P_3, --, P_5$)

   The head section also has the following

   // Problem Description

   // Input

   // Output
3. In the body of an ~~instruction~~ algorithm various programming constructs like if, for, while and some statements like assignments are used.
4. The compound statements may be enclosed with { and } brackets if, for, while can be open and closed by {, } respectively, proper indentation is must for block.
5. Comments are written using // at the begining.
6. The identifier should begin by a letter and not by digit It contains alphanumeric letters after first letter no need to mention data types.
7. The left arrow ":=" used as assignment operator.

   Eg: -v := 10
8. Boolean operators (True/False), logical operators (AND, OR, NO and relational operators (<, <=, >, >=, ==, +, <>) are also used

4. Input and output can be done using read and write.

5. Array [], if then also contain branch and loop can be also used in algorithm.

Eg: Algorithm for fibanocci series using recursion.

Constrast b/w algorithm and pseudocode.

Algorithm	Pseudo Code
1- It is a step by step description of the solution.	1. It is an easy way for writing algorithm.
2- It is always a real algorithm and doesnot use fake codes	2. These are fake codes.
3- They are a sequence of solutions to a problem.	3. They are representation of algorithms.
4- It is a systematically written code.	4. These are simpler ways of writing a code
5- They are an unambiguous way of writing code	5- They are a method of describing codes written in an algorithm.
6- They can be considered as pseudo code	6- They can't be considered as pseudo code.
7- There are no rules to writing algorithms.	7. Contains rules to write pseudo code are there.

Difference b/w Algorithm and Program.

Algorithm	Program
1. It is a step by step process for solving computational problem.	1. A program is nothing but set of instructions (or) executable code
2. An algorithm is designed by domain knowledge (designer)	2. The program can be implemented by specific programmer
3. An algorithm is done at design time	3. Program is done at compile time.
4. Any language. Eg: mathematical solution, general english statements.	4. Any Programming language Eg: C, C++, Java, etc...
5. After compilation of your algorithm you have to analyze based on some criterias, such as time complexity & space complexity.	5. After completion of writing your program you have to do test your program.

Algorithm for matrix addition:

```
{
 for i = 1 to m
 {
  for j = 1 to n
  {
   c[i,j] = a[i,j] + b[i,j]
  }
 }
}
```

time complexity = $2mn + 2m + 1$

Performance and Analysis of Algorithms:

An algorithm is said to be efficient and fast, if it take less time to execute and compare less memory space.

In order to analyze the performance of an algorith we use (i) Time complexity.
(ii) space complexity.

(i) Time complexity:

The amount of time that the algorithm requir for its execution is known as time complexity.

1. Best case time complexity:

If an algorithm requires minimum amount o time for its execution then it is known as best case time complexity.

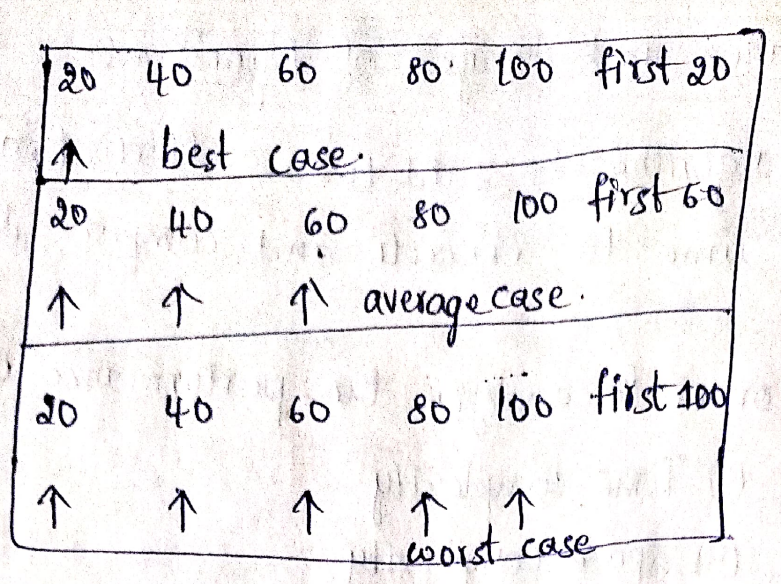
2. Worst case time complexity:

If an algorithm requires maximum amount of tim for its execution, then it is known as worst case time complexity.

3. Average case time complexity:

If an algorithm requires average amount of tim for its execution then it is known as averag case time complexity.

Eg: linear search: It is searching element one by one until key element is found.



Pseudo code Approach:

```
Algorithm for even (or) odd
{
 if (number % 2 == 0) then
     print "even";
 else
     print "odd";
}
```

Eg 2: Algorithm for sum of elements of an array.

```
Algorithm sum of array (a, n)
{
 s = 0, i ———————— 1
 for i = 1 to n to ———————— 1+n
 {
  s = s + a[i]; ———————— n
 }
  return s; ———————— 1
}        time complexity: 2n + 3.
```

There are two approaches of time complexity.

(i) frequency count (or) step count

(ii) Asemptotic notations.

Components of time complexity:

1. Capacity of system: if speed of computer is fast then o/p will be generated quickly otherwise slowly.

2. computer contains single processor (or) multiple processor:

if computer contains single processor then o/p will be generated slowly.

if computer contains multiple processor then o/p will be generated quickly.

Space Complexity: The amount of space that an algorithm requires for its execution is known as space complexity

Three types:

1. Best case space complexity: If an algorithm requires minimum amount of space then it is bestcase.
2. Worst case: If an algorithm requires maximum amount of space.
3. Average case: Requires average amount of space

Components of space complexity:

1. Instruction space: Space Required in the computer inorder to store instruction is known as instruction space.
2. Environmental stack: Amount of space required to store partially executed function.
3. Data space: space required inorder to store variables and constants.

Space complexity of an algorithm is calculated by using two parts.

1. fixed part - variables that are independent characteristics i.e, constants.
2. variable part - variables that have dependent characteristics i.e, variables.

$S(P) = C + Sp$

- $S(P)$ — space complexity of a program
- C — constant (or) fixed part
- Sp — variable part (or) instance characteristics

Eg 1:
```
abc(x, y, z)
return x+y*z+(x-y)
```
$S(P) = C + S.P$
$= 3 + 0$
$= 3$

Eg 2:
```
int square(int a)
{
return a*a;
}
```
$S(P) = C + S.P$
$= 1 + 0 = 1$

Eg 3:
```
sum(a, n)
{
total = 0;
for i=1 to n do
total += a[i]
}
```
$S(P) = C + S.P$
$= 3 + n$

2. Asymptotic Notations: By using Asymptotic notations we can calculate time complexity of an algorithm. By using asymptotic notations we can calculate Best case time complexity, worst case time complexity, average case time complexity of an algorithm.

five types

1. Big oh notation (O)
2. Big-omega notation (Ω)
3. Theta notation (Θ)
4. Little oh notation (o)
5. Little omega notation (ω)

f_0, f_1, f_2

$f_2 = f_0 + f_1$

$f_0 = f_1$

$f_n = f_{(n-1)} + f_{(n-2)} \quad \forall n=2$

1. Big oh notation (O):

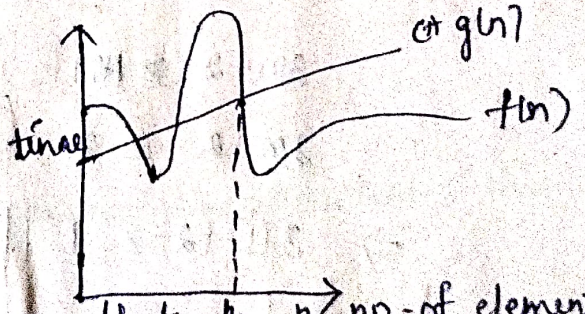
It mainly represents upper bound of an algorithm.

runTime.

→ By using Big oh notation, we can calculate maximum amount of time that an algorithm.

Def: let $f(n), g(n)$ be two non-ve functions, then $f(n) = O(g(n))$ iff there exist two +ve constants c, no such that

$f(n) \leq c*g(n) \quad \forall n \geq n_0$



Eg: $f(n) = 3n+2$; $g(n) = n$ prove that $f(n) = O(g(n))$

$f(n) = O(g(n))$

Sol: Inorder to prove $f(n) = O(g(n))$ we need to satisfy condition

$f(n) \leq c*g(n) \quad \forall n \geq n_0$

$3n+2 \leq c*n$

$3n+2 \leq 4*n$ [take $c=4$]

$3n+2 \leq 4*n$ (let $n_0 = 1$)

Let $n_0 = 1 \Rightarrow 5 \leq 4$ (X) false

Let $n_0 = 2 \Rightarrow 3(2)+2 \leq 8 \Rightarrow 8 \leq 8$ (✓) true.

for $c=3$ the condition does not satisfy.

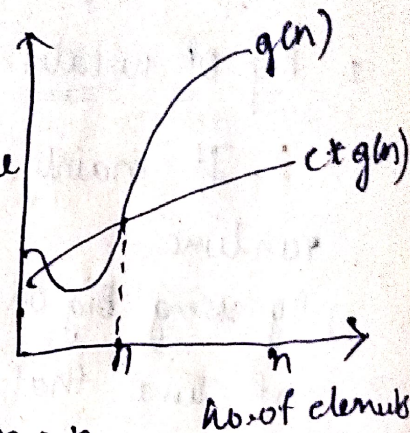
14/11/2023

② Big Omega notation (Ω)

→ It mainly represents lower bound of algorithm runtime.

→ By using Big omega notations, we can calculate minimum amount of time that an algorithm.

Def: let $f(n)$, $g(n)$ be two non negative functions then $f(n) = \Omega g(n)$ iff there exists two +ve constants c, n_0 such that $f(n) = c * g(n)$ $\forall n \geq n_0$



Eg: $f(n) = 3n+2$; $g(n)$ prove that $f(n) = \Omega g(n)$

Sol: In order to prove $f(n) = \Omega(g(n))$, we need to satisfy conditions $f(n) \geq c * g(n)$ $\forall n \geq n_0$

$3n+2 \geq c * n$ (take $c=1$)

$3n+2 \geq n$

$3n+2 \geq n$ (take $n \to n_0 = 1$)

$\Rightarrow 3(1)+2 \geq 1 \;||\; 5 \geq 1$ (true).

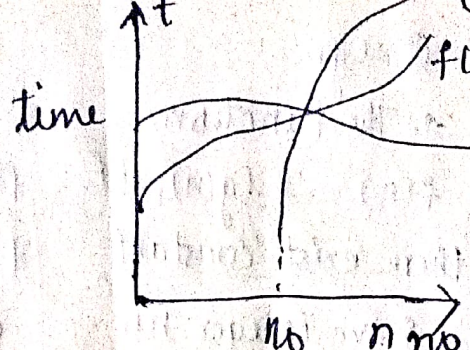
$f(n) \geq c * g(n)$ $\forall n \geq 1$

③ Theta notation (Θ)

→ It represents average bound of algorithm at runtime.

→ By using theta notation we can calculate average amount of time that algorithm requires for its execution i.e, average case time complexity of an algorithm.

Def: let $f(n)$, $g(n)$ be two non negative functions then $f(n) = \Theta(g(n))$ if there exist three constants c_1, c_2, n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ $\forall n \geq n_0$



Eg: $f(n) = 3n+2$; $g(n) = n$ PT $f(n) = O(g(n))$

Sol: Inorder to prove $f(n) = O(g(n))$, we need to satisfy

$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \quad \forall \; n \geq n_0$

$c_1 * n \leq 3n+2 \leq c_2 * n$

Let $c_1 = 1$, $c_2 = 4$

then $n \leq 3n+2 \leq 4n$.

for $n_0 = 1$ $\quad 1*1 \leq 3(1)+2 \leq 4*1$

$1 \leq 5 \leq 4$ (false)

for $n_0 = 2$ $\quad 1*2 \leq 3(2)+2 \leq 4*2$

$2 \leq 8 \leq 8$ (true).

4) Little Oh notation:- (o)

Let $f(n)$, $g(n)$ be two non negative functions then $f(n) = o(g(n))$ such that

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

5. Little omega notation:- (ω)

Let $f(n)$, $g(n)$ be two non-ve functions then $f(n) = \omega(g(n))$ such that

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0.$$

Big Oh

1. The function $f(n) = O(g(n))$ iff there exist constant c and +ve integer no such that $f(n) \leq c \cdot (g(n))$ $\forall n \geq n_0$.
2. It is indicated by 'O'
3. Represents worst case time complexity.
4. It takes maximum amount of time.
5. It is an upper bound.

Omega

1. The function $f(n) = \Omega(g(n))$ iff there exist constant c & +ve integer no such that $f(n) \geq c(g(n))$ $\forall n \geq n_0$.
2. It is indicated by 'Ω'
3. Represents best case time complexity.
4. minimum amount of time.
5. It is an lower bound.

Theta.

1. The function $f(n) = \Theta(g(n))$ iff there exists constant c_1, c_2, n

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$.

2. Represented by Θ
3. average case Time
4. average amount of time.
5. It is an average bound.

16/11/2023

Order of Growth: Measuring the performance of algorithm with i/p size n in order of growth

n	$\log_2 n$	$n \log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65576
32	5	160	1024	4294 967 296.

from the above, it is clear that $\log_2 n$ is slowest growth function. The 2^n (exponential) function is fastest growth function.

Practical Complexity

O(1)

Big Oh	Omega	Theta.
1. The function $f(n) = O(g(n))$ iff there exist constant c and +ve integer no such that $f(n) \leq c*(g(n))$ $\forall n \geq n_0$.	1. The function $f(n) = \Omega(g(n))$ iff there exist constant c & +ve integer no such that $f(n) \geq c(g(n))$ $\forall n \geq n_0$.	1. The function $f(n) = \Theta(g(n))$ iff there exists constant c_1, c_2, n $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$.
2. It is indicated by 'O'	2. It is indicated by Ω	2. Represented by Θ
3. Represents worst case time complexity.	3. Represents best case time complexity.	3. average case time
4. It takes maximum amount of time.	4. minimum amount of time.	4. average amount of time.
5. It is an upper bound.	5. It is an lower bound.	5. It is an average bound.

16/11/2023

Order of Growth: Measuring the performance of algorithm with i/p size n in order of growth

n	$\log_2 n$	$n \log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65576
32	6	160	1024	4294 967 296.

from the above, it is clear that $\log_2 n$ is slowest growth function. The 2^n (exponential) function is fastest growth function.

Practical Complexity

O(1)

Practical Complexity

Biggest

$O(1)$	⟶	constant time
$O(\log n)$	⟶	log time
$O(n)$	⟶	linear time
$O(n \log n)$	⟶	log linear time
$O(n^2)$	⟶	Quadratic time
$O(n^3)$	⟶	cubic time
$O(n^x)$	⟶	polynomial time
$O(2^n)$	⟶	exponential time
$O(n!)$	⟶	factorial time.

Lowest

frequency count (or) step count

In order to calculate time, we use frequency method. It specifies the number of times a statement to be executed.

Rules: 1. for Elements, declarations step count is '0'. (zero)

2. for return and assignment statements step count is '1'.

3. Ignore lower order exponents when higher order exponents are presents.

4. Ignore constants.

Eg: Algorithm for sum of elements in array.

5	3	4	7	2
0	1	2	3	4

$n = 5$

$i = 0$

$i = 1$

$i = 2$

$i = 3$

$i = 4$

$i = 5$ X

```
Algorithm sum (A,n)
{
    s=0 ------------ 1
    for(i=0; i<n; i++) ------ n+1
    {
        s=s+A[i]; ------ n
    }
    return s; ------------ 1
}
```

time complexity: $1+(n+1)+n+1$

$= 2n+3$

Hence ~~space~~ time complexity as $O(n)$

space complexity :-

A	n
n	1
s	1
i	1
$s(n)$ =	$n+3$

Hence space complexity $\longrightarrow O(n)$

Eg: Algorithm for sum of two matrices.

```
Algorithm Add (A, B, n)
{
  for(i=0; i<n; i++)          ---- n+1
  {
    for(j=0; j<n; j++)        ---- n(n+1)
    {
      c[i,j] = A[i,j] + B[i,j]  ---- n*n
    }
  }
}
```

time complexity $= n+1+n^2+n+n^2$

$= 2n^2+2n+1$

time complexity is $O(n^2)$

space complexity:

A	n^2
B	n^2
C	n^2
n	1
i	1
j	1
$s(n)$ =	$3n^2+3$

Eg: Algorithm for matrix multiplication:

```
Algorithm multiply (A, B, n)
{
for (i=0; i<n; i++)
 {
  for (j=0; j<n; j++)
   {
    C[i,j] = C[i,j] + A[i,k] * B[k,j]
   }
 }
}
```

∴ Time complexity = $2n^3 + 3n^2 + 2n + 1$

$T(n) = O(n^3)$

Space complexity:

A —— n^2

B —— n^2

C —— n^2

n —— 1

i —— 1

j —— 1

k —— 1

$3n^2 + 4$

Recurrence Relation of Recursive algorithms

A recurrence relation is an equation that recursively defines a sequence. The recurrences can be solved by "substitution method".

```
Int factorial(int n)
{
if(n==1)                    ——— ①
{                                  } Base
 return 1;                  ——— ①
}
else                                    T(n) = K+T(n-K)
{
 return n * factorial(n-1); } recurrence
}
```

$$T(n) = \begin{cases} T(n-1)+2 & \text{if } n>1 \\ 1 & \text{if } n=1 \end{cases}$$

So we have $T(n) = T(n-1)+1 \rightarrow ①$

$T(n-1) = T(n-2)+1 \rightarrow ②$

$T(n-2) = T(n-3)+1 \rightarrow ③$

Now ① $\Rightarrow T(n) = T(n-2)+1+1$

$T(n) = T(n-3)+1+1+1$

$T(n) = 3 + T(n-3)$

$$T(n) = k + T(n-k)$$

If $n-k=1$ then algorithm will stop

$\Rightarrow k = n-1$

$$\therefore T(n) = n-1 + T(n-(n-1))$$
$$= n-1 + T(n-n+1)$$
$$= n-1+T(1)$$
$$= n-1+1$$
$$= n$$

Hence the efficiency of the recursive function is "n".

(2) Solve the recurrence relation and find the time complexity $T(n) = 2T(n/2) + n$ and $T(1) = 2$

Sol: we have $T(n) = 2T(n/2) + n \rightarrow (1)$; $T(1) = 2$

from (1) we have $T(n/2) = 2(n/4) + n/2 \rightarrow (2)$

$T(n/4) = 2(n/8) + n/4 \rightarrow (3)$

from 2 then (1) $\rightarrow T(n) = 4 + (n/4) + 2n \rightarrow (4)$

xfe the recurrence relation and find the time complexity.

$T(n) = 2T(n/2) + n \; ; \; T(1) = 2 \rightarrow (1)$

from (1) we have;

$T(n) = 2^k + \left(\frac{n}{2^k}\right) + kn$

$\Rightarrow \frac{n}{2^k} = 1 \Rightarrow n = 2^k$

$\Rightarrow \log x = k$

$T(n) = n \; T(1) + \log_2 n$

$T(n) = n \cdot 2 + n \log_2$

$T(n) = 2n + n \log_2$

ne complexity $O(n \log_2)$

olve the recurrence relation $T(n) = T(n-1) + n$
$) = 1$ and find the time complexity.

Sol: we have $T(n) = T(n-1) + n$ —— (1) and $T(1) = 1$

Replace n by n-1 in 1.

$\Rightarrow T(n-1) = T(n-2) + n-1$ —— (2)

Again Replace n by n-2 in equ 1

$T(n-2) = T(n-3) + n-2$ —— (4)

Substitute equ (2) in equ (1)

$T(n) = T(n-2) + n-1 + n$ —— (3)

putting equ ④ in equ ③ we got

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

The General equation is

$$T(n) = n + (n-1) + (n-2) + \cdots + (n-k) + (n-(k+1))$$

If $n-(k+1) = 1$

$$n - k = 2$$

$$\therefore T(n) = n + (n-1) + (n-2) + \cdots + 2 + T(1)$$

$$= 1 + 2 + 3 + \cdots + n$$

$$T(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}.$$

Time complexity $= O(n^2)$

Amortized Analysis:

Finding average Running time for operation over the worst case sequence of operations.

1. Amortized Analysis doesn't allows Random numbers.
2. Amortized analysis and average case analysis are different
3. Incase of average analysis we can find averaging all the inputs, but whereas amortized analysis we are averaging over a sequence of analysis.

To compute amortized analysis we can follow the following techniques.

(i) Aggregate method
(ii) Acounting method
(iii) Potential method

(i) Aggregate method.

1. It is used to find the cost total per "n" no. of worst case sequence of operation.

2. Aggregate method = $\frac{T(n)}{n}$

where $T(n)$ = Total cost

n = no. of operations.

(ii) Acounting method:

→ By using acounting method we can find individual cost for an operation.

→ finally we found total cost is $\frac{T(n)}{n}$

(iii) Potential methode

→ It is like a acounting method but operations depend upon its lower cases and upper cases.

Note: upper bound, lower bound and aggregate average bound can be represented by following statements

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < x^3 \cdots < x^n$

Explain about analysis framework briefly?

→ It is a systamatic approach applied for analysing any given algorithm

- Measuring time complexity
- Measuring space complexity.
- Computing Order of growth of algorithms
- measuring input size.
- measuring Running time
- Computing Best case, worst case and average case efficiencies.

④ Measuring Input size:

The Efficiency of an algorithm can be computed as a function. Input size is passed as a parameter. It can be (i) Exact Value

(ii) approximate value.

Eg: Spell - checking algorithm

→ numbers of characters

→ number of words - Input size.

⑤ Measuring Run time:

from terms in algorithm

(i) identify basic operations

(ii) understand the concept of basic operations

(iii) compute the total number of time taken by basic operations.

formula used is $t(n) = C_{op} \cdot C(n)$

where $t(n)$ - Running time of basic operation

C_{op} - time taken by basic operation to execute

$C(n)$ - no. of times the operation need to be executed.

⑥ Completing Best case, worst and average case Efficiencies:

Linear Search: If the searching element is at starting index It is best case. Its time complexity is $O(1)$

→ If the search element is at last index (n). It's Time complexity is $O(n)$ (worst case)

Average Case time Complexity = $\frac{1+2+3+\cdots+n}{n}$

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

$= \frac{n+1}{2}$

Time Complexity is $O(n)$.

—— ○ ——