5-12-'17

INTRODUCTION TO ALGORITHMS (UNIT-I)

→ **Algorithm**

It is a finite set of instructions to solve a particular task. Each algorithm must follow these constraints:

Input: i) Every algorithm takes 'O' or more input parameters

Output: ii) Every algorithm must produce atleast one output value

Finiteness: iii) Algorithm must contain finite number of instructi

Definiteness: iv) In our algorithm, each and every instructio must be unambiguous (clear meaning) : Definetn

Efficient: v) Tracing our algorithm using paper and pen [It consists of basic operations, operations that are not divisible]

→ **SDLC - Software Development Life Cycle**

i) ~~Design~~ Problem Definition

ii) Requirement Specifications

iii) Design a solution for our problem (DAA)

iv) Validation (if the solution works or not)

v) Implementation

vi) Analysis (DAA)

vii) Testing → a.) Debugging b.) Profiling

viii) Deployment

Algorithm Specification (How to write an algorithm)

1. comment lines to describe algorithm i/p and o/p before writing our algorithm
2. Block of statements / set of instructions - compound stmts
3. No need to declare data types for normal variables. If any data structures / records are used, then we must declare data type
   eg record: stack
   { datatype: data 1
   ⋮
   }
4. a ← expression // a = b
   Assignment operator Comparing
5. i) for i ← 1 to n step step do
   { // }
   step → keyword no step, automatically incremented by 1
   For loop
   ii) while < condition > do
   { // }
   While loop
   iii) do
   { // }
   while < condition >
   Do-While loop
   iv) repeat
   { // }
   until condition
   Repeat - Until statements
6. read "Enter values" → to take i/p
   (write output) → to give o/p
7. Algorithm name (i/p parameters)
   Header

Eg. Write an algorithm to find maximum element in given array of elements.

NOTE:
1-D array: A[i] 2-D array: A[i,j]

→ // Algorithm to find maximum element in given ar
max := A[0]
for i := 0 to n-1 step step do
{
  if < A[i] > max >
  max = A[i];

ht Code

```
Algorithm Max (a, n)
{
  m := a[0];
  for i := 1 to n-1 step step do
  {
    if < A[i] > max >
      max = A[i];
  }
  return max;
}
```

Eg. Binary Search Algorithm

```
Algorithm BinarySearch (a, ?, k)
{
  l := 0;
  r := n-1;
  m := < (l+r)/2 >;
  while < l <= r > do
  { min
    if < n < m >
    { r = m
    if < k = a[m] >
    { return m; }
```

```
    else if < n < a[m] >
    { r := m - 1; }
    else
    { l := m+1; }
  }
}
```

Performance of Algorithms

i) Space Complexity

↳ a) Fixed Path ↳ b) Variable Path

$S(P) = C + S_p$ (Instances)

Algorithm → Space of Algorithm; Constant → Fixed/Constant Path; space of references → Variable Path

Space required for program 'p'

Space for code and any i/p, o/p parameters

Space for arrays, pointers, structures, etc

ii) Time Complexity

↳ Both Compile Time (depends on compilers and interpreters) and Run Time

We consider only Compile Run Time

→ We must combine the time taken by all the basic operations

Performance Analysis

* We are using performance analysis to check which solution is the best for a problem 'P' out of 'n' no. of solutions ($S_1, S_2, S_3, \ldots\ldots, S_n$)
* Performance Analysis is done in two ways
  ↳ Space Complexity ↳ Time Complexity
* Space Complexity - It is the amount of memory required to execute a program or complete a task. It depends on two components:
  i) Fixed Path ii) Variable Path

i) Fixed Path - Contains independent characteristics of inputs and outputs for a program space for code (no. of ip, o/p = size of i/p, o/p)

Eg: Algorithm ex ($a_1, a_2, \ldots, a_n$ // i/p parameters
{
≡ // Space for code
return — // o/p parameters
}

ii) Variable Path - is dependent on input instances of a variable. Includes all references, stack space instance - i/p provided at the time of program execution eg: no of elements in array, array elements in Binary Search.

Space Complexity = Fixed Path + Variable Path

$S(P) = C + Sp$ (Instance Characteristics)

Eg.) Algorithm abc (a, b, c)
{
return a+b*c+(a+b-c)/(a+b)+4.0;
}

→ NO variable path, only fixed path
→ S = 1 word (for a, b, c)

Eg.) Algorithm sum (a, n)
{
s := 0.0;
for i := 1 to n do
s := s + a[i];
return s;
}

→ S=
To store n: no of elements — n words — n
For referring array a — 1 word — a
For returning s value — 1 word — 1
For index variable — 1 word — i
(sum (input variables) ≥ (n+3)

• For exact space complexity:
i) Priory Analysis - Before program execution
ii) Posterior Analysis - After program execution must be performed

Eg) Algorithm Rsum (a,n)
{
if (n ≤ 0) then return 0.0;
else return Rsum (a , n-1) + a[n];
}

→ S =

For referring array a = 1 word
To store value n = 1 word
Return value = 1 word
Each iteration - 3 words
For n+1 iterations
S ≥ 3(n+1)

Time Complexity - It is the amount of time required to execute a program or complete a task.
It depends on two components:
i) Compile Time ii) Run Time
Time Complexity for a program 'P'
t(P) = Compile time + Run Time
We omit compile time since it depends on the type of the compiler.
We estimate only run time.
We count the number of steps and assume that each step (basic operation) takes one unit of time.
We have two methods to calculate time complexity
i) Introducing Count Variable ii) Table Method

i) Introducing Count Variable - We declare a global count variable that is initialized to zero. Before each basic instruction, we increment count variable by 1. At the end of the program, we get number of basic steps.

Eg. Algorithm abc (a, b, c)
{
int cnt := 0;
cnt := cnt + 1;
return a+b*c+(a+b-c)/(a+b)+4.0;
}
t(P) = 1

Eg. Algorithm sum (a,n)
{
cnt := 0;
cnt := cnt + 1; // 1
s := 0;
cnt := cnt + 1;
for i := 1 to n do // n
cnt := cnt + 1; // n
s := s + a[i];
cnt := cnt + 1; // 1 - last for
cnt := cnt + 1; // 1
return s;
}
t(P) = 1 + n + n + 1 + 1 = 2n + 3

Eg. Algorithm Rsum (a,n)

{

cnt := 0;

cnt := cnt + 1;

1 // if (n ≤ 0) then cnt := cnt + 1;

return 0.0;

cnt := cnt + 1;

1 // else cnt := cnt + 1;

tRsum(n-1) // return Rsum (a, n-1) + a[n];

}

n=0, $t(P) \geq 2 + t(Rsum(n-1))$

n = 0 = 2

n > 0 = 2 + tRsum (n-1)

tRsum (n) = 2 + tRsum (n-1)

= 2 + [2 + tRsum (n-2)]

= 2n + 2

Eg. Algorithm for matrix addition

Algorithm Madd (a, b, m, n)

{

[m][n];

for i := 0 to m-1 // m+1

begin do

for j := 0 to n-1 // n+1

begin do

s[i][j] := a[i][j] + b[i][j] // m×(n+1)

end

end

Eg. Algorithm for matrix addition

Algorithm madd (a, b, m, n)

{

for i := 1 to m do // m+1

for j := 1 to n do // m(n+1)

c[i,j] = a[i,j] + b[i,j]; // mn

}

t = (m+1) + m(n+1) + (mn)

For Table Method:

Step count = 1

Frequency = (m+1)

Eg. Algorithm to print nth Fibonacci Series

Algorithm fseries (n)

{ c = a+b;

for (i: 3; i<=n; i++)

cout << a << b. c = a+b.

a = b. b = c

~~c = a+b~~

Table Method

S/E (Steps required for execution)	Frequency	Total no. of steps
1	$(m+1)$	$(m+1)$
1	$m(n+1)$	$mn+m$
1	mn	mn
		$2mn+2m+1$

If $m=n \Rightarrow 2m^2+2m+1$

= order of m^2

→ omit constant terms

P.T.O. →

19-12-17

Assymptotic Notation

i) Big-Oh (O)

Upper Bound, Worst Case

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

$f(n) \leq c * g(n)$, c is constant, $n \geq n_0$, n_0 is con

Eg. $f(n) = 2n+3 \leq 3n$; $c = 3$, $n_0 = 3$ [Substitute $n = 1, 2, 3 \cdots$ in emp. unbl condition

$f(n) = O(n)$; $c = 3$, $n_0 = 3$

ii) Omega (Ω)

Lower Bound, Best Case

$f(n) = \Omega(g(n))$

$f(n) \geq c * g(n)$; c is constant, $n \geq n_0$, n_0 is const

iii) Theta (Θ)

Average Case

$f(n) = \Theta(g(n))$

$c1 * g(n) \leq f(n) \leq c_2 * g(n)$; c1, c2 are constants, $n \geq n_0$, n_0 con

→

Recurrence Relations

$$T(n) = aT(n/b) + f(n)$$

↳ Divide and conquer principle (merge sort and quick sort)

↳ Problem 'n' is divided into 'a' sub problems, each of size 'n/b', takes f(n) to time to merge after sorting

↳ Merge sort recurrence relation : Each problem is divided into '2' sub problems, each of size 'n/2' and takes time 'n' to merge after solving

$T(n) = 2T(n/2) + n \Rightarrow n \log n$

Methods to solve Recurrence Relations

↳ Substitution Method

↳ Iteration Method

↳ Master Method

i) Substitution Method

- Firstly, we must guess the solutions
- Then, we must find the constants by using mathematical inductions (find boundary conditions c and n_0)
- If method is not very easy, go to iteration method

ii) Iteration Method

- Construct Recurrence Tree
- Find solution at each leves
- Involves mathematical manipulations

iii) Master Method

- Best Method
- Uses Master Theorem
- Involves three cases, check our case and find the answer

i) Substitution Method

Eg. $T(n) = 2T(n/2) + n$

$f(n) \leq c \cdot g(n)$

$T(n) = c \cdot n \log n$ → Guessing the solution

$T(n) \leq c \cdot n \log n$ → For big Oh Notation

Substitute $n = n/2$

$T(n/2) \leq c \cdot \frac{n}{2} \log (n/2)$

Now, $T(n) = 2T(n/2) + n$

$\Rightarrow T(n) \leq 2c \cdot \frac{n}{2} \log\left(\frac{n}{2}\right) + n$

$\Rightarrow T(n) \leq cn \log (n/2) + n$

$\Rightarrow T(n) = cn \log n - cn + n$ [$\because \log 2 = 1$]

$\Rightarrow T(n) \leq cn \log n$

$T(n) = 2 \cdot T(n/2) + n$ $\quad T(n) = cn \log n$

$n = 1 \Rightarrow T(1) = 2 \cdot T(1/2) + 1 = 2 \cdot 0 + 1 = 1$ $\quad n = 1 \Rightarrow T(1) = c \cdot 1 \cdot \log 1 = c \cdot 0 = 0$

Problem size ↓ Problem of size 1 cannot be divided further

$n = 2 \Rightarrow T(2)$ $\quad T(1) \leq T(1) \rightarrow$ False

$n = 2 \Rightarrow T(2) = 2 \cdot T(2/2) + 2 = 2 \cdot 1 + 2 = 4$ $\quad n = 2 \Rightarrow T(2) = c \cdot 2 \log 2 = c \cdot 2 \cdot 1$

Let $c = 2 \Rightarrow T(2) = 4$

$T(2) = T(2)$

$\therefore c \geq 2$

$n_0 \geq 2$

$\therefore f(n) \leq 2 \cdot g(n)$

$f(n) = O(\log n)$; $c \geq 2$; $n_0 \geq 2$

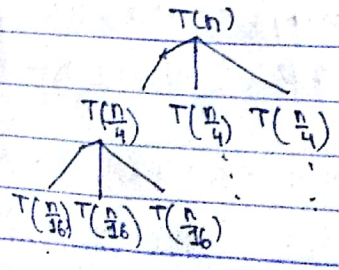
ii) Iteration Method

Construct Recursion Tree

Eg. $T(n) = 3T(n/4) + cn^2$

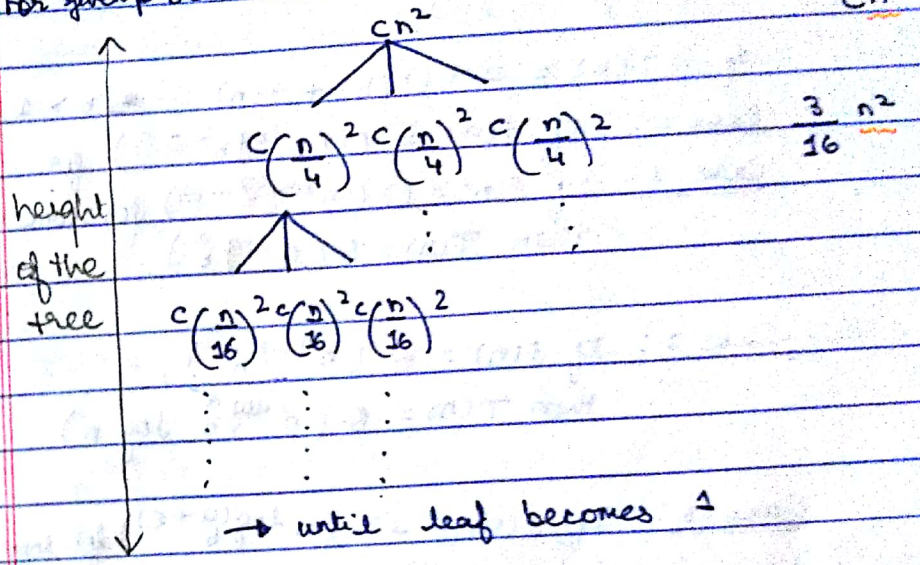
Problem divided into problems $n/4$ · 3 sub problems of size $n/4$ · cn^2 Root node and takes time cn^2 to merge

In General:



For given problem:

(Add terms at all levels)



→ until leaf becomes 1

$\left(\frac{n}{4^i}\right) = 1 \Rightarrow i = \log_4 n$

$O(n^2)$

$\therefore f(n) = O(n^2)$

[NOTE: Values of 'c' and 'n_0' not necessary]

iii) Master Method

Eg. $T(n) = aT(n/b) + f(n)$; $b > 1$, $a \geq 1$
$f(n)$ = +ve fn

Case 1: If $f(n) = O(n^{\log_b(a-\epsilon)})$ for some constant $\epsilon > 0$
then $T(n) = \theta(n^{\log_b a})$

Case 2: If $f(n) = \theta(n^{\log_b a})$,
then $T(n) = \theta(n^{\log_b a} \log n)$

Case 3: If $f(n) = \Omega(n^{\log_b(a+\epsilon)})$ for some constant $\epsilon > 0$
and if $a f(n/b) \leq c f(n)$ for some constant c
and all sufficiently large 'n', $c \leq 1$,
then $T(n) = \theta(f(n))$

NOTE:

For given function, calculate $n^{\log_b a}$
If $n^{\log_b a} > f(n)$, apply case 1
If $n^{\log_b a} < f(n)$, apply case 3
If $n^{\log_b a} = f(n)$, apply case 2

[Refer Coreman Textbook for this concept - It is not available in prescribed text book]

20-12-17

How to apply master theorem for given recurrence relation

i) Calculate $n^{\log_b a}$ = exp

ii) Compare it with $f(n)$
if $f(n) <$ exp → go to case 1
if $f(n) >$ exp → go to case 3
if $f(n) =$ exp → go to case 2

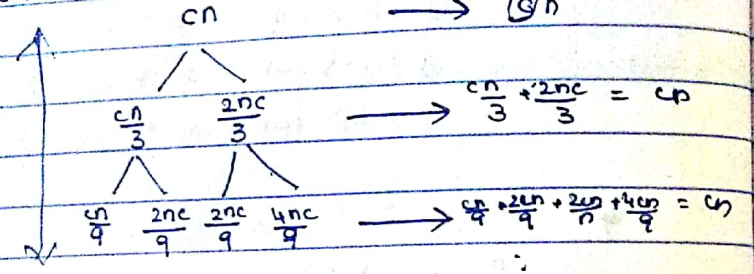
Eg. $T(n) = 2T(\frac{n}{2}) + n$
$a = 2, b = 2$
$n^{\log_2 2} = n^1 = n$
$f(n) = n$
case (ii) $T(n) = \theta(n^{\log_2 2} \log n) = \theta(n \log n)$
$\therefore T(n) = \theta(n \log n)$

Eg. $T(n) = 9T(\frac{n}{3}) + 1)$
$a = 9, b = 3$
$n^{\log_3 9} = n^{\log_3 3^2} = n^{2\log_3 3} = n^{2 \cdot 1} = n^2$
$f(n) = n$
$n^2 > n$ → case (i)
$T(n) = \theta(n^{\log_3 9}) = \theta(n^{2\log_3 3}) = \theta(n^{2\log_3 3})$
$= \theta(n^{2 \cdot 1}) = \theta(n^2)$
$\therefore T(n) = \theta(n^2)$

Eg. $T(n) = T(n/3) + T(2n/3) + O(n)$

Iteration Method

$O(n) \Rightarrow cn$



$cn \rightarrow \frac{2cn}{3} \rightarrow \frac{4cn}{3} \rightarrow \cdots\cdots$

$\left(\frac{2}{3}\right)^k n = 1$ [∵ height k = longest path, consider 2/3, not 1/3]

$k = \log_{3/2} n$ = height of tree (longest path)

$= \log_2 n$

$T(n) = cn \cdot \log_2 n$ [c is constant]

$= n \log n$

$\therefore T(n) = O(n \log n)$

Eg. Algorithm for Fibonacci Series, Calculate Step Count & Find the time complexity.

→ Algorithm fibSeries(n)

```
{
    a := 0 ; b := 0 ;
    b := 1  print (a, b);
    for i := 1 to n do
        {c := a + b ;
         print (c) ;
         a := b ;
         b := c ; }
}
```

OR

```
Algorithm fibSeries(n)
{
  if((n==0) || (n==1))
    return 1 ;
  else
    return fibSeries(n-1) + fibSeries(n-2) ;
}
```

Algorithm FibSeries(n)	S/E	Frequency	Total no. of Steps
{			
if(n<=1)	1	1	2
write(n)	1		
else			
{			
f1:= 0;	1	1	1
f2:= 1;	1	1	1
for i:=2 to n do	1	(n-1)	n-1
fn = f1+f2;	1	(n-2)	n-2
f1:= f2;	1	(n-2)	n-2
f2:= fn	1	(n-2)	n-2
write(fn);	1	(n-2)	n-2
}			
}			

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

$= 5n-5$

$= 5(n-1)$: Step Count

$= O(n)$

$T(n) = O(n)$ for fibonacci series

$O(n)$, $O(n^2)$, $O(n^3)$, $O(\log n)$, $O(2^n)$

linear, quadratic, cubic, logarithmic, exponential

Eg. Write Asymptotic notations for given polynomial expressions, find constants $c, n_0, c_1, n_0, c_1, n_0, c_2$ for O, Θ, Ω.

(i) $3n+2$ (ii) $3n+3$ (iii) $100n+6$ (iv) $10n^2+4n+2$

→ (i) $3n+2$

$f(n) = 3n+2$

a.) O

$f(n) = O(g(n))$ $fn \leq c \cdot g(n)$

$3n+2 \leq 4n$; $n_0 = 2$, $c = 4$.

$f(n) = O(n)$; $n_0 = 2$, $c = 4$

b.) Ω

$f(n) = \Omega(g(n))$ $fn \geq c \cdot g(n)$

$3n+2 \geq 3n$; $n_0 = 1$, $c = 3$

$f(n) = \Omega(n)$; $n_0 = 1$, $c = 3$

c.) Θ

$f(n) = \Theta(g(n))$ $c_1 g(n) \leq f(n) \leq c_2 g(n)$

$3n \leq 3n+2 \leq 4n$; $n_0 = 2$, $c_1 = 3$, $c_2 = 4$

$f(n) = \Theta(g(n))$; $n_0 = 2$, $c_1 = 3$, $c_2 = 4$

(ii) $3n+3$

a.) O

$3n+3 \leq 4n$ $n_0 = 1$, $c = 4$

b.) Ω

$3n+3 \geq 3n$ $n_0 = 1$, $c = 3$

c.) Θ $\therefore 3n \leq 3n+3 \leq 4n$, $n_0 = 1$, $c_1 = 3$, $c_2 = 4$

$\therefore f(n) = O(n)$, $n_0 = 1$, $c = 4$ | $f(n) = \Omega(n)$; $n = 2$, $c = 3$

$f(n) = \Theta(g(n))$, $n_0 = 1$, $c_1 = 3$, $c_2 = 4$

(iii) $100n + 6$

a.) O

$f(n) = O(g(n))$ $\quad f(n) \leq c \cdot g(n)$

$100n + 6 \leq 110n$; $n_0 = 1$, $c = 110$

$f(n) = O(n)$; $n_0 = 1$, $c = 110$

b.) Ω

$f(n) = \Omega(g(n))$ $\quad f(n) \geq c \cdot g(n)$

$100n + 6 \geq 100n$; $n_0 = 1$, $c = 100$

$f(n) = \Omega(n)$; $n_0 = 1, c = 100$

c.) Θ

$f(n) = \Theta(g(n))$ $\quad c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$

$100n \leq 100n + 6 \leq 110n$; $n_0 = 1$, $c_1 = 100$, $c_2 = 110$

$f(n) = \Theta(n)$; $n_0 = 1$, $c_1 = 100$, $c_2 = 110$

(iv) $10n^2 + 4n + 2$

a.) O

$f(n) = O(g(n))$ $\quad f(n) \leq c \cdot g(n)$

$10n^2 + 4n + 2 \leq 15n^2$; $n_0 = 1$, $c = 15$

$f(n) = O(n^2)$; $n_0 = 1$, $c = 15$

b.) Ω

$f(n) = \Omega(g(n))$ $\quad c \cdot g(n) \leq f(n)$

$10n^2 + 4n + 2 \geq 10n^2$; $n_0 = 1$, $c = 10$

$f(n) = \Omega(n^2)$; $n_0 = 1$, $c = 10$

c.) Θ

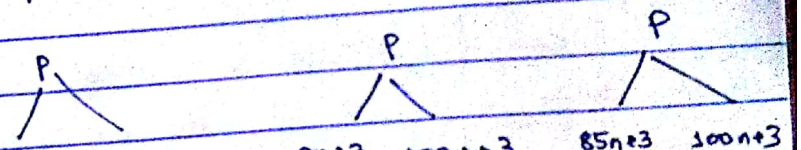
$f(n) = \Theta(g(n))$ $\quad c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

$10n^2 \leq 10n^2 + 4n + 2 \leq 15n^2$; $c_1 = 10$, $c_2 = 15$, $n_0 = 1$

21-12-'22

Practical Complexities

Eg.



cn $\quad dn^2$

$cn \leq dn^2$ $\quad$ Large difference $\quad$ Not much difference. Both similar

$n \geq \frac{c}{d}$

$O(n)$ → used for sufficiently large values of n. when $n \geq c/d$

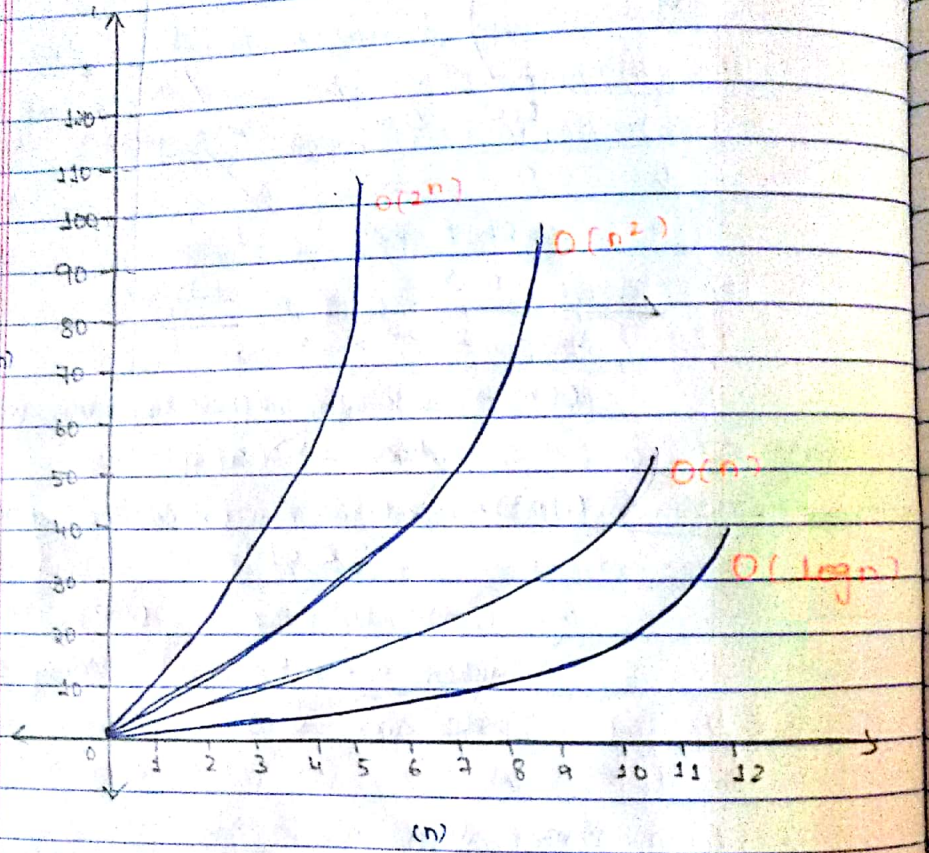
$O(n^2)$ → used when n values are small, when $n < c/d$

Eg. $O(n) = 10^6 n$ ms $\quad O(n^2) = n^2$ ms.

when $n < 10^6$, use $O(n^2)$ Eg. 10^5

else use $O(n)$

n	log n	n^2	2^n
1	0	1	2
2	1	4	4
4	2	16	16
8	3	64	256
16	4	256	65536
32	5	1024	4294967296



↑ Actual Behaviour of Algorithms in Asymptotic Notation

Performance measurement

→ Introduce gettime statement at the beginning and also at the end of the program or algorithm
→ Find the difference of the end time and the beginning time to get time taken
→ Do it for different values of 'n', i.e. different input sizes
→ By plotting a graph of f(n) vs n, find the point where performance changes drastically

Eg. 2^n → n = 40 10 - 15 days } drastic change
n = 50 300 years }

→ // Algorithm Beginning
t1 := Gettime();

// Body of Algorithm

t2 := Gettime();
t := t2 - t1;
// Algorithm end
t is time taken to perform algorithm

→

Random Algorithms

→ They generate random values.
→ There are two types of Random Algorithms:
i) Las Vegas Algorithm ii) Monte Carlo Algorithm

Las Vegas Algorithm	Monte Carlo Algorithm
Always gives correct O/P if correct i/P is given	It gives different types of O/P based on i/P
Eg: Quick Sort	Eg. Bank OTP generation
↳ pivot value may by different, but O/P is sorted algorithm	

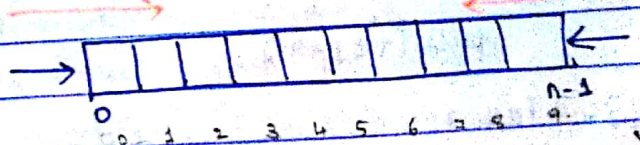
→ Sample Space ⇒ All possible outcomes
→ Monte Carlo Algorithms
Two types of outputs:
i) YES ii) NO
Eg: Generate a random no, check if it is a prime number or not., YES or NO
Used in generation of random numbers in Kryptography.

DAP Q's: Solving Recurrence Relations, Finding Time Complexity For a Given Algorithm, Performance Analysis, Elementary Data Structures Algorithms

Elementary Data structures

Data structures are divided into two categories:
1.) Linear
i) Stack (LIFO / FILO) (push, pop, top) (creation: tos = -1)
ii) Queue (FIFO) (enqueue, dequeue)
iii) Linked List
2.) Non-Linear
i) Trees
ii) Graphs

Eg. Implement two stacks in single array.

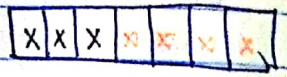


n = 10 n/2 = 5 : -1 n = 9 n/2 = 4.5 : 5 : 4

Stack 1: 0 to 4
tos1 = -1
push
if (tos1 != (n/2)-1)
else full
pop
if (tos1 != -1)
else empty

Stack 2: 5 to 9
tos2 = n
push
if (tos2 != (n-1))
else full
pop
if (tos1 !=

tos1 = -1 tos2 = n



Overflow : tos2 - tos1 = 1

Underflow 1 : tos1 > n-1 Underflow 1 : tos1 < 0

Underflow 2 : tos2 < 0 Underflow 2 : tos2 > n

Algorithm :

```
Algorithm Stack2 (n)
{
   tos1 := -1, tos2 := n;
   // Push 1
   if((tos2-tos1)=1 || (tos1 > n-1))
      write (overflow)
   else
      push [tos1++];
} // Push 2
   if ((tos2-tos1)= 1) || (tos2 < 0))
      write (overflow)
   else
      push [++tos2]
   // Pop 1
   if ((tos1 < 0))
      write (underflow)
   else
      pop [tos1--]
```

```
// Pop 2
   if ((tos2 > n))
      write (underflow)
   else
      pop [tos2--]
}
```

Eg. Implement 'm' stacks in an array of size 'n'
(Eg m = 5, n = 15)

⇒ Stack

- top = -1 ⇒ stack is created
- Insertion, Deletion from same end
- LIFO, Ordered list
- Push Algorithm (stack element)

```
Algorithm PUSH (S, x)
{
   if (top ≥ n-1) then
   {write ("Stack is full");
    return false;}
   else
   { top:=top+1;
     S[top]:= x;
     return true; }
}
```

- Pop Algorithm (return)

```
Algorithm POP (S)
{
   if (top == -1) then
   { write ("Stack is empty");
     return false; }
   else
   { x:= S[top];
     top:= top - 1;
     return true; }
}
```

⇒ Queue

- Ordered list
- Performs insertion from one end, deletion from other end
- Gen FIFO
- Circular Queue
  → Front Pointer, → Rear Pointer
  → C Queue full (r + 1 = f

- Insertion

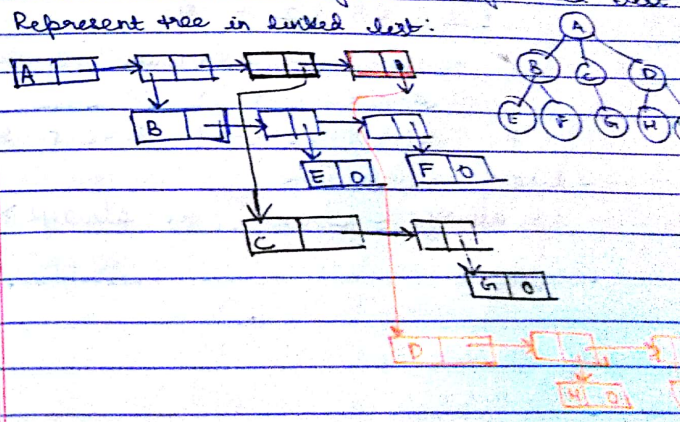
Algorithm Insert (x)
{
rear := (rear + 1) mod n; // f = rear = 1
if (front = rear) then
{ write ("Queue is full");
if (front = 0) then
rear := n - 1;
else
rear := rear - 1;
return false; }
else
q[rear] := x;
return true; }

- Deletion

Algorithm Delete ()
{
if (front = rear) then
(Full)
else
if (r ≥ f)
{
r = r - 1;
}

⇒ Trees

- Unordered list
- Contains one designated node called root node
- Contains many partitions, elements partitioned into sets ($t_1, t_2, t_3, \ldots, t_n$; $n >= 0$), these partitions are also tree
- Nodes with same parent - siblings
  longest path from leaf node to root node = height
  longest path from root node to leaf node = depth
- Number of children a node has = degree of node
- To store a tree in system: array, linked list
- Represent tree in linked list:



Binary Tree

↳ Tree with at most two children, left and right

↳ Complete Binary Tree: Each node has either 0 or two children, inserting elements from left to right

Representing complete binary tree using array

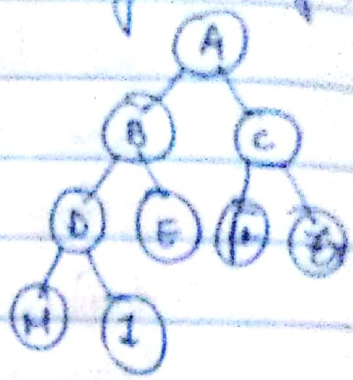
↳ For CBT, for any node 'i',

parent node at [i/2]th location

left child at [2i]th position

right child at [2i+1]th position,

when array index starts at '1'



from '0' → [lc = 2i+1] rc = [2i+2] p = [i/2]

A	B	C	D	E	F	G	H	I
1	2	3	4	5	6	7	8	9

F = 5 5/2 = 2.5 = 2 (lower bound) = B = parent

lc = x rc = x

c = 3 3/2 = 1.5 = 1 = A = p lc = 6 = F rc = 7 = G

↳ Binary Search Tree

lc always < parent rc always > parent

Dictionaries

Dictionary is an abstract data type that supports insert, delete and search operations.

Eg. Binary Search Tree (BST)

Binary Search Tree

i) Every element is a key, it is a binary tree and
. No two elements are have same key.

ii) All elements in left sub tree are less than ~~parent~~ root node

iii) All elements in right sub tree are greater than root node

iv) Left and right sub trees are also BSTs

Operations on BST:

i) Search

(a) Search k^{th} minimum element

(b) Iterative Search

(c) Recursive Search

ii) Insert

iii) Delete

(a) Node has no children

(b) Node has single child

(c) Node has two childre

a) Algorithm for recursively searching an element in BST

→ Algorithm Search (t, x)

```
{
  if (T = 0) then return 0;
  else if (x = T→ data) then return T;
  else if (x < T→ data) then Search(T→lchild, x);
  else
      return Search (T→rchild, x);
}
Record = node
{
  node *lchild;
  node *rchild;
  type data;
}
```

b) Algorithm for iteratively searching an element in BST

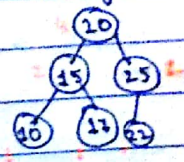
~~Algorithm~~ →

```
Record = node
{
  node *lchild;
  node *rchild;
  type data;
```

```
Algorithm Search (t, x)
{
  if (T=0) then return 0;
  while (T!=0) then
  {
    if (x = T→ data) then return T;
    else if (x < T→ data) then
        T := T→ lchild;
    else
        T := T→ rchild;
  }
```

c) Algorithm to find kth minimum element from BST
(min ele - leftmost child, max ele - rightmost child)

→ Eg.



① inOrder: 10, 15, 17, 20, 22, 25
LRN
→ linear search to find kth min ele (index k)

② Introduce another field in node (LL) left size. Left Size = 1 + no. of n in left sub tree
If k value = left size value, then kth min element, if k < ls, go t
if k value > ls, go to rst