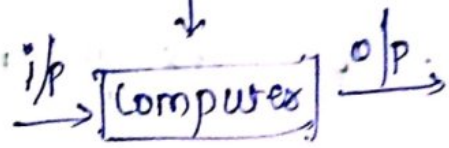
Introduction to Algorithms

→ method to solve a problem → Algorithm.

(OR).

→ An unambiguous set of instruction to solve a problem for obtaining a required output for any legitimate input in a finite amount of time.

→ Algorithm.



Properties of an (Good) algorithm

1) Finiteness.
2) Definiteness
3) effectiveness.
4) Input
5) output.

~~Eg:~~

To find Gcd of two numbers.

Scanned by CamScanner

Algorithm: Euclids. (m, n) old fashion

$GCD(m, n) = Gcd(n, m \bmod n)$

1. if $n = 0$ return gcd = m else goto step2.
2. Divide m by n and assign the remainder to r
3. assign n to m and r to n. go back to step1.

Gcd(10, 6) $m = 10$ $n = 6$.

m	n	r
10	6	4
6.	4	2.
4.	2.	0
gcd. ← 2.	0	

Pseudocode

Algorithm: Euclids(m, n)

// To find gcd of numbers using Euclids algorithm.

// Input: Two non negative numbers.

// output: gcd of two numbers.

```
while n ≠ 0 do
    r ← m % n
    m ← n
    n ← r.
return m
```


Scanned by CamScanner

Algorithm: consecutive integer checking method.

// To find gcd

//

step 1: assign $t = \min(m, n)$

step 2: perform the operation m mod t, if remainder is zero go to step 3 else go to step 4.

step 3: perform the operation n % t; if remainder is zero gcd = t else go to step 4.

step 4: decrement the value of t by 1 and go back to step 2.

m	n	t = min(m, n)	m % t	n % t
10	6	6	4	—
10	6	5	0	1
10	6	4	2	—
10	6	3	1	—
10	6	2 ← gcd	0	0

```
step1: t= min(m,n)
   2:  m%t
   3:  n%t
   4:  t=t-1
```


Scanned by CamScanner

Note:

→ Euclids algorithm is much faster when compared to the consecutive integer checking method when the input numbers are large

→ In worst case the C.I.C.M will have $\frac{\min(m,n)}{2}$ iterations.

→ consecutive Integer checking method fails if any of the two numbers is zero.

Algorithm : middle school method

step 1: Find prime factors of m

step 2: Find prime factors of n

step 3: The common prime factors is the gcd

Drawback:

→ The time to find the prime factors is not defined properly.

Scanned by CamScanner

i,e; prime factorisation set is not unambiguous.

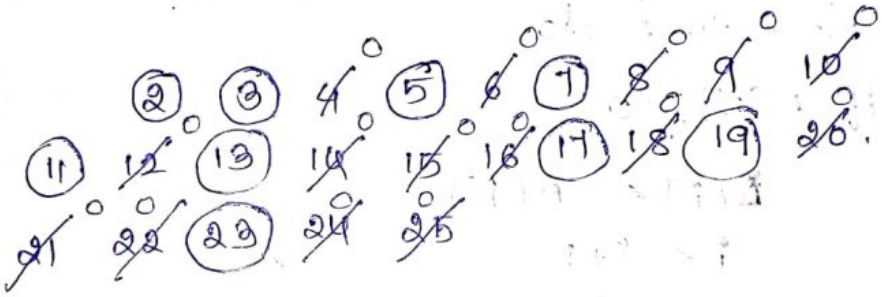
→ if any of the number is 1. it fails.

Algorithm to find generate prime number untill given number n

Algorithm:

Sieve of Eratosthenes

Eg:



n=25 ∴ iterations from 2 to $\sqrt{n}$

2 to $\sqrt{25}$

2 to 5

Algorithm : Sieve (n)

// to generate prime numbers.

// input : A positive number

// output : list of prime number stored in vector b

Scanned by CamScanner

```
for  p ← 2 to n      A[p] ← p

for  p ← 2 to √n do
     if A[p] ≠ 0
        j = p * p.
        while j ≤ n do
              A[j] ← 0
              j ← j + p

i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
          A[i] ← A[p]
          i ← i + 1

return L
```

Algorithmic design process.

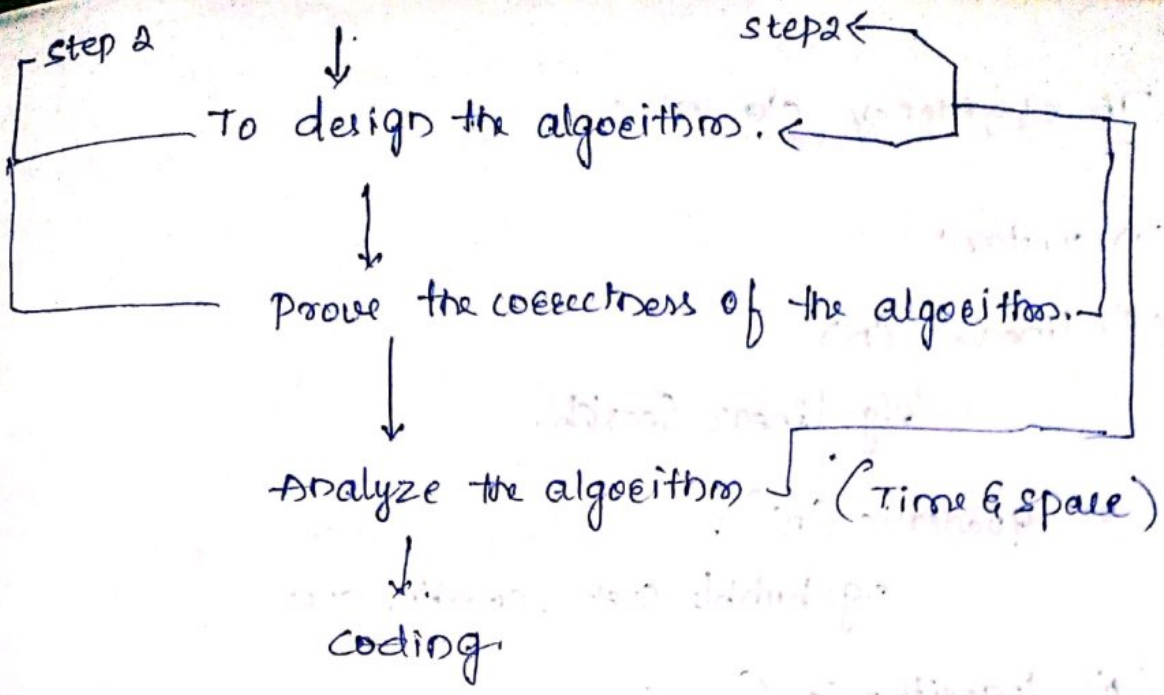
understand the problem statement

↓

Decide on: (i) computational means.
(ii) whether it gives exact or approximate solution
(iii) what data structure to use.
(iv) choose proper Algorithmic design technique.

which systems / what type of system.

Scanned by CamScanner



~~Time complexity~~

Time efficiency

$T(n) = C_{op} * C(n)$

C_{op} → Basic Operation.

$C(n)$ → frequency of basic oper.

Time efficiency of algorithm depends on the basic operation.

The basic operation is that operation which contributes the most of the total running time.

The number of basic steps defines the time - efficiency.

Scanned by CamScanner

The efficiency classes :

(1) constant

(2) linear. (n)

e.g: linear Search.

(3). quadratic (n^2).

e.g: bubble sort, selection sort.

(4) logorithmic. ($\log n$)

e.g: binary Search.

(5) $n \log n$.

e.g: mergesort, heapsort, quicksort.

(6) n^3

e.g: floyds algorithm

(7) 2^n

e.g: n-queens

Sum of Subset.

(8) $n!$

constant. $< \log n < n < n \log n < n^2 < n^3 < 2^n < n!$

Scanned by CamScanner

Problems.

(1) arrange the following efficiency classes in descending order.

$\sqrt{n}$, 3^n, 2^{2n}, $(n-2)!$, $0.0001n^3$, $2n^4+1$,

$\log n$, $5\log_{10}(n+10)^{10}$

Sol:-

$(n-2)!$, 2^{2n}, 3^n, $2n^4+1$, $0.0001n^3$, $\sqrt{n}$,

$5\log_{10}(n+100)^{10}$, $\log n$.

(2) Algorithm : abc(n)

Input: A non-negative integer n

```
s ← 0
for i ← 1 to n do
    s ← s + i
return s.
```

1) what does the algorithm compute?

→ Sum of 'n' natural numbers.

uicksoet.

$2^n < n!$

Scanned by CamScanner

2) identify the basic operation.

Addition

3) how many times the basic operation be executed

n.

4) what is the efficiency class of algorithm

linear.

(8) Compute the efficiency class of following code

```
for (i=1 ; i<=n ; i=i*2)
```

logn.

```
for (i=1 ; i*i<=n ; i++)
```

$\sqrt{n}$.

```
for (i=1 ; i<=n; i++)
    for( j=1 ; j<=i ; j++ )
        for(k=1; k<=50; k++)
```

i=1	2	n
j=1	2	n
k=50.	2×50	n×50

Scanned by CamScanner

$50 + 2 \times 50 + \cdots + n(50)$

$50\left(\frac{n(n+1)}{2}\right) = n^2$.

```
for (i=1; i<=n; i++)
    for(j=1; j<= i^2; j++)
        for (k=1; k <= n/2; k++)
```

$i=1$	$i=2$	3	$\cdots$	n
$j=1$	4	9	$\cdots$	n^2
$\frac{n}{2}$	$4 \times \frac{n}{2}$	$9 \times \frac{n}{2}$	$\cdots$	$n^2 \cdot \left(\frac{n}{2}\right)$

$\frac{n}{2} \cdot (1 + 4 + 9 + \cdots + n^2)$

$= \frac{n}{2} \cdot ($

Scanned by CamScanner

Asymptotic notation

To compare and rank the order of growth of a function (The function which expresses the time efficiency) the following asymptotic notations are used

(1) Big oh(O) — Specifies the upperbound
(2) Big omega(Ω) → specifies the lower bound
(3) Big theta(Θ) — Specifies the range of upperbound & lowerbound.

Big oh (O) notation

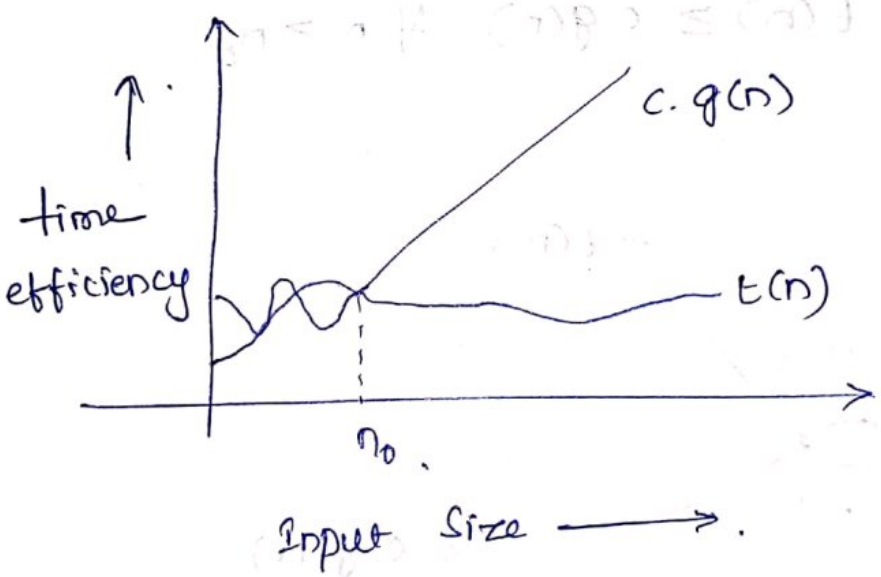
let t(n) and g(n) be the non-negative function where t(n) represents the actual time taken by the algorithm and g(n) is a Sample function Considerd for the Comparison

Scanned by CamScanner

A function $t(n)$ is said to be $O(g(n))$ denoted by $t(n) \in O(g(n))$, if $t(n)$ is bounded above some constant multiples of $g(n)$ for all larger values of n, if

There exist a positive constant C and a positive integer n_0 satisfying the statement

$$t(n) \leq C\, g(n) \quad \forall\, n \geq n_0.$$



ex:

$t(n) = n$

$g(n) = n^2$

$n \in O(n^2)$

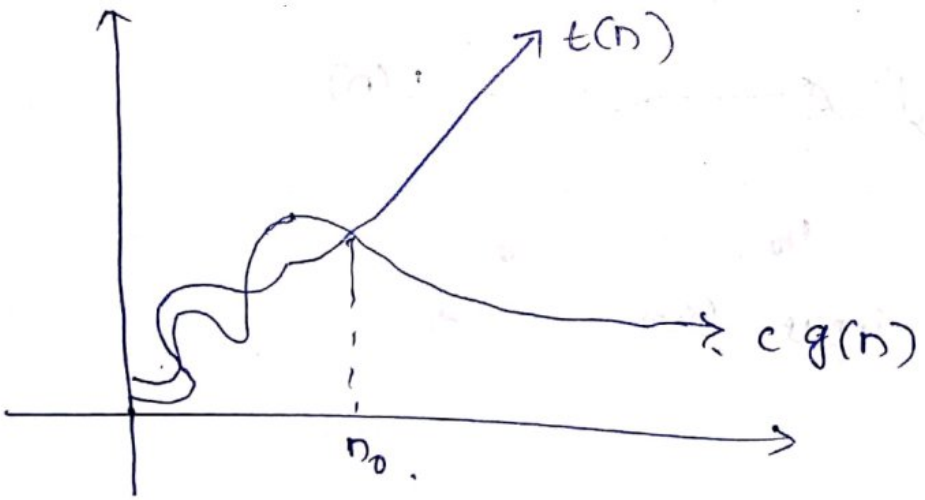
but. $n^3 \notin O(n^2)$.

Scanned by CamScanner

Big omega (Ω)

The function $t(n)$ is said to be $\Omega(g(n))$ denoted by. $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below some positive constant multiples of $g(n)$, for all larger values of n, if there exist some positive constant c & some nonnegative integer n_0 such that

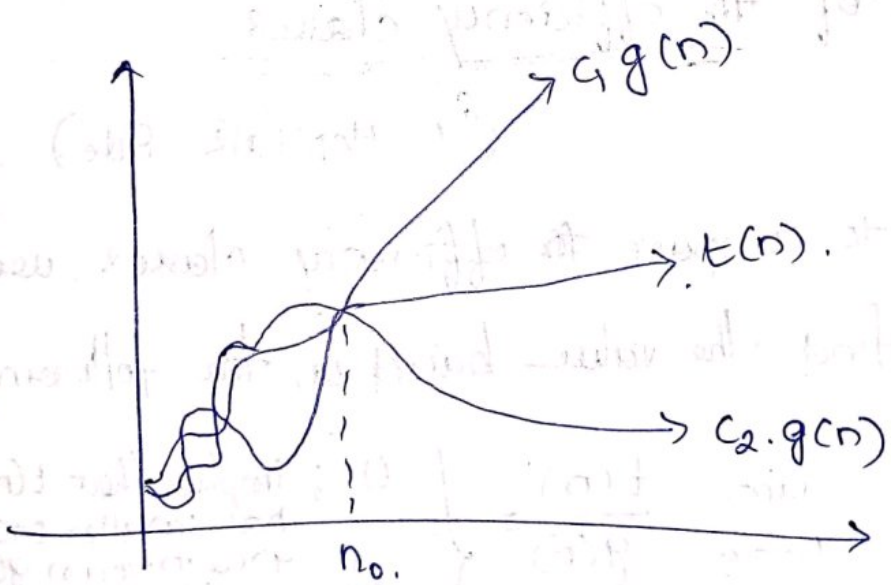
$$t(n) \geq c\, g(n) \quad \forall\, n \geq n_0.$$



Scanned by CamScanner

Big theta (Θ)

The function $t(n)$ is said to be $\Theta(g(n))$ denoted by $t(n) \in \Theta(g(n))$ if $t(n)$ is bounded both above and below some positive constant multiples of $g(n)$, for all larger value of n, if there exist some positive constant c_1 and c_2 and some non-negative integer n_0, such that $c_2 \cdot g(n) \leq t(n) \leq c_1 \cdot g(n)$ $\forall n \geq n_0$.



Scanned by CamScanner

Problems

(1) given the following statements and a function $f(n) = \frac{n(n+1)}{2}$ state whether they are true or false

(i) $f(n) \in O(n^3)$ (T)

(ii) $f(n) \in \Omega(n^2)$ (T)

(iii) $f(n) \in \Omega(n^3)$ (F)

(iv) $f(n) \in O(n)$ (F)

Comparison of the efficiency classes:

("L. Hopital's rule")

In order to compare the efficiency classes we need to find the value based on the following equations.

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & ; \text{ implies that } t(n) \text{ has smaller order growth than } g(n). \\ c & ; \text{ implies that } t(n) \text{ has the same order growth compared to } g(n) \\ \infty & ; \text{ implies } t(n) \text{ has greater order growth than } g(n) \end{cases}$$

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & ; \ O. \\ c > 0 & ; \ \Theta \\ \infty & ; \ \Omega \end{cases}$$

Scanned by CamScanner

Problems.

(1) Compare Order of growth of $\frac{1}{2} n(n-1)$ and n^2.

$\frac{1}{2} n(n-1) \rightarrow t(n)$.

$n^2 \rightarrow g(n)$

Ans:

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \lim_{n \to \infty} \frac{\frac{1}{2} n(n-1)}{n^2}$$

$$= \lim_{n \to \infty} \frac{\frac{1}{2}\left(1 - \frac{1}{n}\right)}{1} = \frac{1}{2}.$$

on applying L-Hopitals Rule as we got Constant

$\therefore \frac{1}{2} n(n-1) \in \Theta(n^2)$

(2) Compare $n!$ and 2^n.

Ans:

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \lim_{n \to \infty} \frac{n!}{2^n}$$

by sterling formula $n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$.

$$\lim_{n \to \infty} \frac{\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n}{2^n} = \lim_{n \to \infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n.$$

$= \infty$.

Scanned by CamScanner

$n! \in \Omega(2^n)$

General plan to analyze the efficiency of non-recursive algorithm.

(mathematical analysis of non recursive algorithm)

Step-1: Decide on the parameter or parameters indicating the input size

Step.2 Identify the algorithms basic operation. (located in the inner most loop).

Step 3 Check whether the number of times the basic operation executed depends only on the size of input. if it also depends on some additional property then the best, worst and average case is necessary.

Step 4: Setup a Summation rule expressing the number of times the basic operation gets executed.

Scanned by CamScanner

step 5. Using standard formula and rules of manipulation either find a closed form for the count, or atleast establish the order of growth.

Algorithm to find the maximum element in an array.

```
Algorithm MaxElement(A[0---n-1])
// To find maximum element in array
// Input: Array.
// output: maximum element.

    maxvalue ← A[0]
    for i ← 1 to n-1
        if A[i] > maxvalue
            maxvalue ← A[i].

    return maxvalue.
```


Scanned by CamScanner

Basic operation is comparison.

let $C(n)$ denotes the number of times the basic operation gets executed and can be denoted as $C(n) = \sum_{i=1}^{n-1} 1$

$$C(n) = n-1-1+1 = n-1$$

$$= \Theta(n).$$

Algorithm to find the uniqueness of an element in an array.

Algorithm Element-Uniqueness $(A[0 \cdots n-1])$

// Determining the uniqueness of list of elements

// input : array.

// output : Returns true if elements are not repeated
else returns false

```
for i ← 0 to n-2 do.
    for j ← i+1 to n-1 do.
        if A[i] == A[j]
            return False.
return True.
```


Scanned by CamScanner

Basic operation : Comparison.

The number of times the basic operation gets executed depends on type of input they we are supplying

in worst case the comparison statement will be executed maximum number of times if all elements are unique OR last and second last elements are same

let C(n) denotes the number of time the basic operation gets executed and it can be represented as. $C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1.$

$$= \sum_{i=0}^{n-2} n-1-i-1+1$$

$$= \sum_{i=0}^{n-2} n-1-i$$

$$= O(n^2).$$

best Case : 1.

Scanned by CamScanner

Lab. Programs.

Lab-1 merge Sort.

```
#include <stdio.h>
#include <stdlib.h>
# define SIZE 1000
int count;

void merge(int A[], int low, int high, int mid)
{
int i, j, k, B[SIZE];
 i = low. ;
 j = mid+1 ;
 K = low;
 while (i <= mid && j <= high)
 {
  if (A[i] < A[j])
        B[k++] = A[i++];
  else
        B[k++] = A[j++];
 }
 while (i <= mid)
        B[k++] = A[i++];
 while (j <= high)
        B[k++] = A[j++];
```


Scanned by CamScanner

```
    for(i= low ; i<= high ; i++).
            A[i] = B[i].
}.

void mergesort(int A[SIZE], int low, int high).
{
    int mid
    if (low << high).
    {
        mid = (low+high)/2;
        mergesort(A, low, mid);
        mergesort(A, mid+1, high);
        merge (A, low, high, mid);
    }
}.

int main()
{.
int  A[SIZE], B[SIZE], C[SIZE], n, i, j, c1, c2,
  cg;
Printf(" Enter number of elements \n");
Scanf ("%d", &n);.

Printf(" read elements - - - ");
for(i=0 ; i<n ; i++)
        Scanf ("%d", &A[i]);
```


Scanned by CamScanner

```
mergesort(A, 0, n-1)
Printf("\n sorted elements are \n");
for(i=0; i<n; i++)
{
  Printf("%d\t", A[i]);
}
Printf("The basic operation is executed
        %d times\n", count);

for(i=16; i<1000; i=i*2)
{
   for(j=1; j<=i; j++)
   {
      A[j] = j;
      B[j] = i-j+1;
      C[j] = (rand() % i)+1;
   }
   count=0;
   mergesort(A, 1, i);
    C1 = count.
   count = 0
   mergesort(B, 1, i);
    C2 = count;
   count=0
   mergesort(C, 1, i);
    C3 = count;
Printf("%d \t%d\t %d\t %d\n", i, C1, C2, C3);
}
```


Scanned by CamScanner

return 0;

}.

Mathematical Analysis for Recursive algorithm.

step 1. Decide on parameter / parameters indicating the input size 'n'.

step 2 : identify the basic operation of the algorithm.

step 3: check whether the number of times the basic operation is executed can vary on different inputs of the same size n. If so we need to find best, worst & average case efficiency.

step 4
Setup a reccurance relation with an appropriate base case for the number of times the basic operation gets executed

step 5 Solve the reccurance relation with appropriate methods to find the time complexity.

Scanned by CamScanner

Algorithm: factorial of a number.

Algorithm: Fact(n)

// To find factorial of number

// ~~A post~~ Input: positive integer n

// output: factorial of n.

```
if n==0  return 1
    return n * fact(n-1)
```

Basic operation: multiplication

let M(n) denotes the number of times the multiplication operation is executed by the algorithm

for n=0 M(0)=0.

$$M(n) = M(n-1) + 1 \quad \forall n > 0.$$

$$= M(n-2) + 1 + 1 \quad \text{by substitution method}$$

$$= M(n-3) + 3.$$

$$= M(n-k) + k.$$

Scanned by CamScanner

as $M(0) = 0$.

$\Rightarrow n - k = 0 \Rightarrow n = k$.

$\therefore M(n) = 0 + k = k = n$

$\therefore O(n)$

Algorithm to solve tower of hanoi problem.

Algorithm: Tower of hanoi(n, S, D, T)

//

// input:

// output:

if $n = 1$.

move n^{th} disc from s to D

else

Tower of hanoi ($n-1$, S, T, D)

move n^{th} disc from S to D.

Tower of hanoi ($n-1$, T, D, S)

Scanned by CamScanner

Basic operation: movement of disc

let $M(n)$ denotes the total number of moment of disc

Base case: $M(1) = 1$.

$$M(n) = 2M(n-1)+1 \quad \forall n>1$$

$$= 2(2M(n-2)+1)+1$$

$$= 2^2 M(n-2) + 2 + 1$$

$$= 2^2 \cdot (2 \cdot M(n-3) + 1) + 2 + 1$$

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

$$= 2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$$

$$M(k) = 2^k M(n-k) + 2^{k-1} + 2^{k-2} + 2^{k-3} + \ldots + 2 + 1$$

Substitution method

Scanned by CamScanner

$M(1) = 1$

$n - K = 1$

$K = n - 1$.

$= 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 1$.

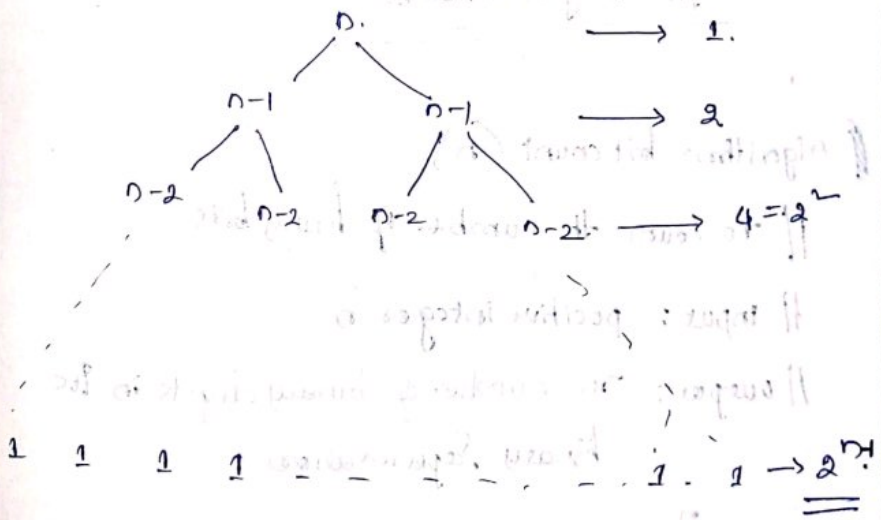
$M(n) = 2^n - 1$.

$\therefore O(2^n)$

using recursive tree method.

$M(n) = 2M(n-1) + 1$

let n denotes number of discs.



Scanned by CamScanner

let L denotes the level of the tree then the efficiency of algorithm can be calculated by counting number of nodes in the tree.

$$M(n) = \sum_{i=0}^{l} 2^i.$$

By solving this $O(2^n)$

Algorithm: To count the number of binary digits for a given number.

// Algorithm: bitcount(n)
// To count the number of binary bits
// input: positive integer n
// output: The number of binary digits in its binary representation

if n = 1 return 1
return (L n/2 J) + 1

Scanned by CamScanner

Basic Operation: Addition.

· let A[n] denoted the number of times the addition is performed.

base case: A[1] = 0.

$$A[n] = A\left[n/2\right] + 1$$

$$= A\left[\frac{n}{4}\right] + 1 + 1$$

$$= A\left[\frac{n}{8}\right] + 1 + 1 + 1$$

$$= A\left[\frac{n}{16}\right] + 3 + 1$$

$$A[k] = A\left[\frac{n}{2^k}\right] + k.$$

A[1] = 0

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

$$n = 2^k.$$

$$k = \log_2 n$$

Scanned by CamScanner

Quicksoet

```
void QuickSoet(int a[SIZE], int low, int high)
{
int i;
if (low < high)
{
   i = partition(a, low, high);
   QuickSoet(a, low, i-1);
   QuickSoet(a, i+1, high);
}
}

int partition(int a[SIZE], int low, int high)
{
  int i, j, pivot;
  Pivot = a[low];
   i = low+1;
   j = high;
 while(1)
{
```


Scanned by CamScanner

```
while (pivot ≥ A[i] && i <= high)
    i++;
while (pivot <= A[j])
    j--;
if (i<j)
    swap(a[i], a[j]);
else
    swap(a[low], a[j]);
return j;
}
}
```

Algorithm to find nth fibinoccie series number

Algorithm: fib(n) Fibinocci(n)
computes nth fib number.
// To find nth fib number
// input: A positive integer n
// output: positive integer (nth fib number).

Scanned by CamScanner

```
if (n<=1) return n
return fib(n-1) + fib(n-2)
```

Basic operation : Addition

Base case : $A(0) = 0$

$A(1) = 0$

let $A(n)$ denotes the number of addition done :

The recurrance relation to count number of addition can be expressed as

$$A(n) = A(n-1) + A(n-2) + 1$$

↓

This is second order equation

Can be expressed as

$$r^2 - r - 1 = 0$$

on solving $r = \left(\frac{1+\sqrt{5}}{2}\right)^n \alpha + \beta \left(\frac{1-\sqrt{5}}{2}\right)^n$

∴ efficiency is exponential.

Scanned by CamScanner

master theorem method.

The master theorem method applies to the reaccurance of the type

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

$a \geq 1$, $b > 1$ and $f(n)$ is asymptically positive

case 1 $f(n) < n^{\log_b^a}$ $\Rightarrow T(n) = \theta(n^{\log_b^a})$

(i) $T(n) = 4T(n/2) + n$.

$$n^{\log_b^a} = n^{\log_2^4} = n^2$$

here n^2 is greater than $f(n)$ i.e; n

$\therefore$ $T(n) = \theta(n^2)$

case 2 $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

here $n^{\log_b^a} = n^{\log_2^4} = n^2$

$f(n) = n^{\log_b^a}$

$$T(n) = \theta(n^{\log_b^a} (\log n)^{k+1})$$

$$T(n) = \theta(n^2 \cdot (\log n)^{0+1}) = \theta(n^2 \cdot \log n)$$

Scanned by CamScanner

Case 3

$$T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

$$a = 4 \quad b = 2$$

$$n^{\log_b^a} = n^2$$

$$f(n) \geq n^{\log_b^a}$$

$$n^3 > n^2$$

$$\therefore T(n) = \theta(f(n)) = \underline{\theta(n^3)}$$

e.g. (1) $T(n) = T\left(\frac{2n}{3}\right) + 1$.

Sol:-

$$a = 1 \qquad b = 3/2$$

$$n^{\log_b^a} = n^{\log_{3/2}^1} = n^0 = 1$$

$$\therefore \underline{T(n) = \theta(\log n)}$$

Scanned by CamScanner

The 3 cases of master theorem are

1) if $f(n) = O\left(n^{\log_b^a - \epsilon}\right)$, $\epsilon > 0$

$f(n)$ grows polynomially slower than $n^{\log_b^a}$ then the solution is

$$T(n) = \Theta\left(n^{\log_b^a}\right)$$

2) $f(n) = \Theta\left(n^{\log_b^a} (\log n)^k\right)$ $k \geq 0$.

and $n^{\log_b^a}$ grows at a similar rate then the solution is $T(n) = \Theta\left(n^{\log_b^a} \cdot (\log n)^{k+1}\right)$

3) $f(n) = \Omega\left(n^{\log_b^a + \epsilon}\right)$ for some $\epsilon > 0$.

and $f(n)$ grows polynomially faster than $n^{\log_b^a}$ then $T(n) = \Theta(f(n))$

Scanned by CamScanner

Emperical Analysis of an algorithm

step 1. Understand the experiment purpose

step 2. decide on the efficiency matric m to be measured and the measurement unit (Operation Count)

step 3: decide on the chareeterstics of the input Sample

step 4: prepare a progeam implementing the algorithm foe the experimentation

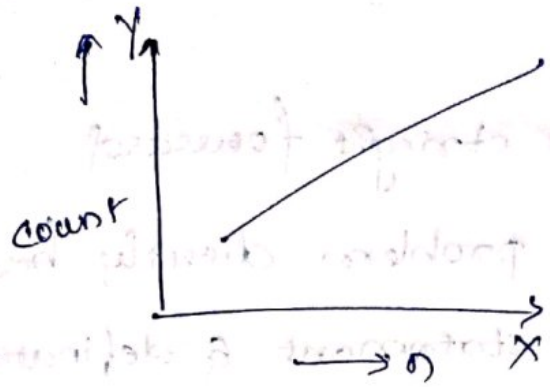
step 5: generate a Sample of inputs.

Step 6: Run the algorithm on the Sample inputs and record the data observed

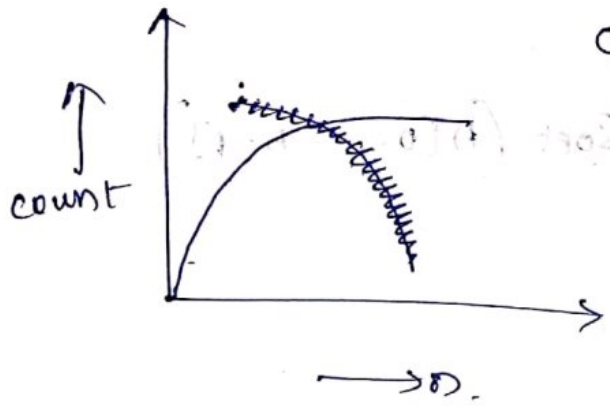
step 7: Analyse the data obtained.

By plotting a graph, by taking x as input size Y axis denoting the count (number of times basic operation executed)

Scanned by CamScanner

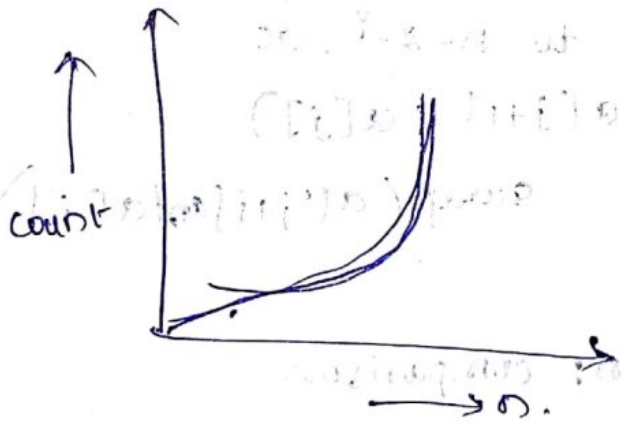


efficiency = linear.



convex concave

efficiency = logarithmic



Concave

convex

efficiency = $n \log n$ (or) n^2 (or) exponential.

Scanned by CamScanner

Brute force

Brute force is a straight forward method to solve a problem directly based on the problem statement & definations of the concept involved.

Bubble sort

Algorithm : BubbleSort (A[0 - - - n-1])

//

//

//

for i ← 0 to n-2 do

for j ← 0 to n-2-i do

if (a[j+1] < a[j])

swap (a[j+1] and a[j])

Basic operation: comparison

let C(n) denotes the number of times the basic operation gets executed

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

Scanned by CamScanner

$$= \sum_{i=0}^{n-2} n-i-1$$

$$= (n-1) + (n-2) + \cdots + 1.$$

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

$$\underline{\underline{\Theta(n^2)}}$$

trace bubble sort for the following input.

89 45 90 29 17.

45 89 90 29 17

89 9

	1st iteration	2nd iteration	3rd	4th
89	45	45	29	17
45	89	29	17	29
90	29	17	45	45
29	17	89	89	89
17	90.	90	90	90

Scanned by CamScanner

Algorithm: Selection sort

//

//

//

for i ← 0 to n-2 do

min ← i

for j ← i+1 to n-1 do

if a[j] < a[min] min ← j

swap (a[i] and a[min])

	1st iteration	2nd it.	3rd it.	4th iteration
89	17	17	17	17
45	45	29	29	29
90	90	90	45	45
29	29	45	90	89
17	89	89	89	90

Scanned by CamScanner

Basic operation : comparison

C(n) denotes total number of comparison

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

$$= \sum_{i=0}^{n-2} n-1-i-1+1$$

$$= \sum_{i=0}^{n-2} n-1-i = \frac{n(n-1)}{2}$$

$\theta(n^2)$

module - II

Divide and Conquere

Scanned by CamScanner

$$= \sum_{i=0}^{n-2} n-1-i-1+1$$

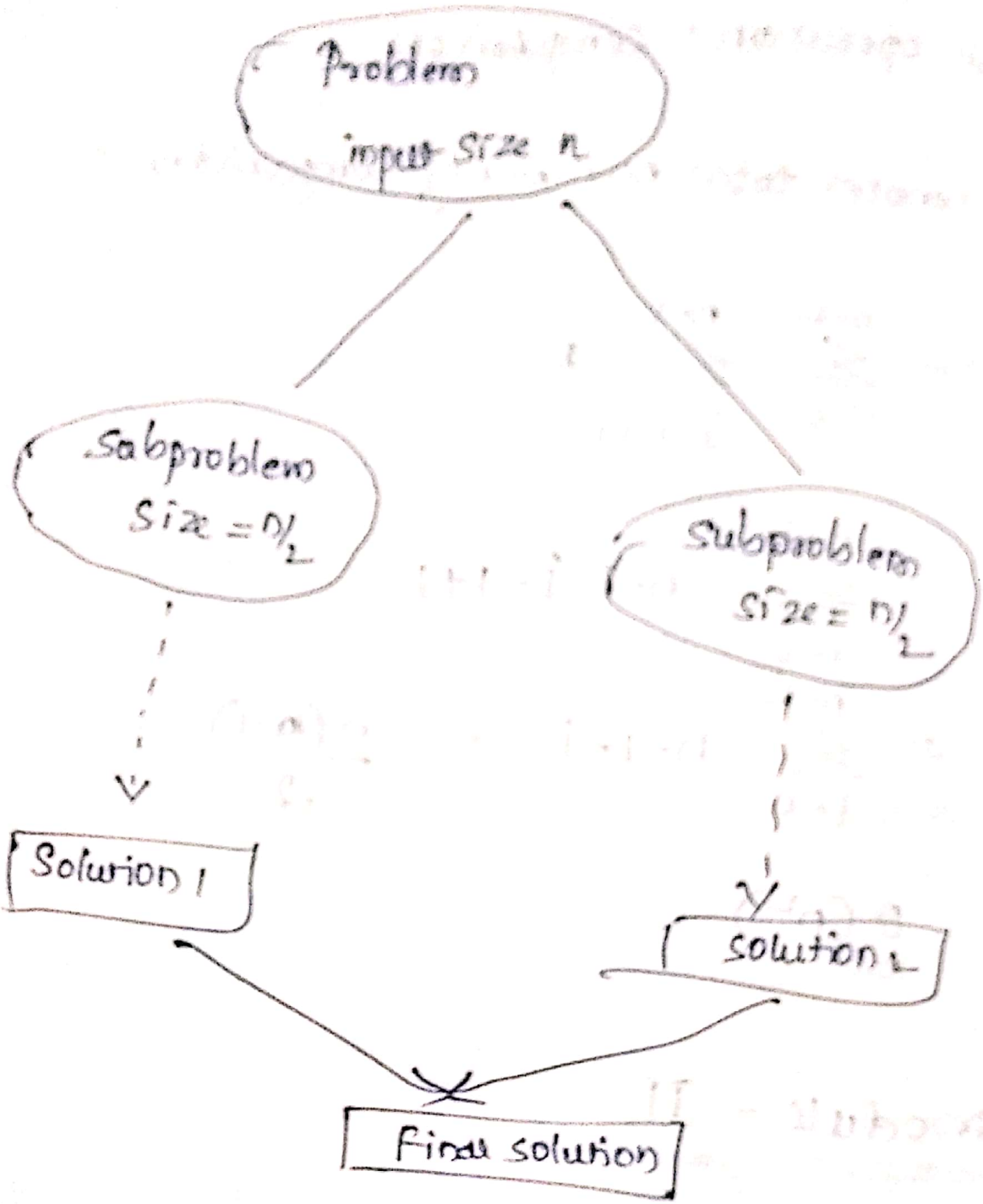
$$= \sum_{i=0}^{n-2} n-1-i = \frac{n(n-1)}{2}$$

$\theta(n^2)$

module - II

1) Divide and Conquere

Divide and Conquere is the most common method of problem solving. Here the given problem is divided into two equal parts which are recursively solved and merged together to get the final solution.



mergesort

Algorithm: mergesort(A, l, r)

// recursively sorts the elements using mergesort.

// input: array A indexed with l and r

// output: Sorted array A.

```
if l < r
    m ← (l+r)/2
    mergesort(A, l, m)
    mergesort(A, m+1, r)
    merge(A, l, m, r)
```

Algorithm: merge(A, l, m, r)

// merges two sorted arrays

// input: two sorted arrays i.e A[l...m] and A[m+1...r]

//output: Single sorted array.

```
i ← l
j ← m+1
k ← l
while i <= m and j <= r
    if A[i] < A[j]
        B[k] ← A[i] ; update i & K.
    else
        B[k] ← A[j] ; update j & k.
while i <= m
    B[k] ← A[i] ; update i & k
while j <= r
    B[k] ← A[j] ; update j & k.
```

Analysis:

Basic operation: comparison

If $n = 1$; no comparison

let $C(n)$ denote the total number of comparison done

base case $C(1) = 0$.

$C(n) = 2\,C(n/2) +$ time taken to merge.

$C(n) = 2\,C(n/2) + C_{merge}(n)$

$C(n) = 2\,C(n/2) + (n-1)$

Applying master theorem

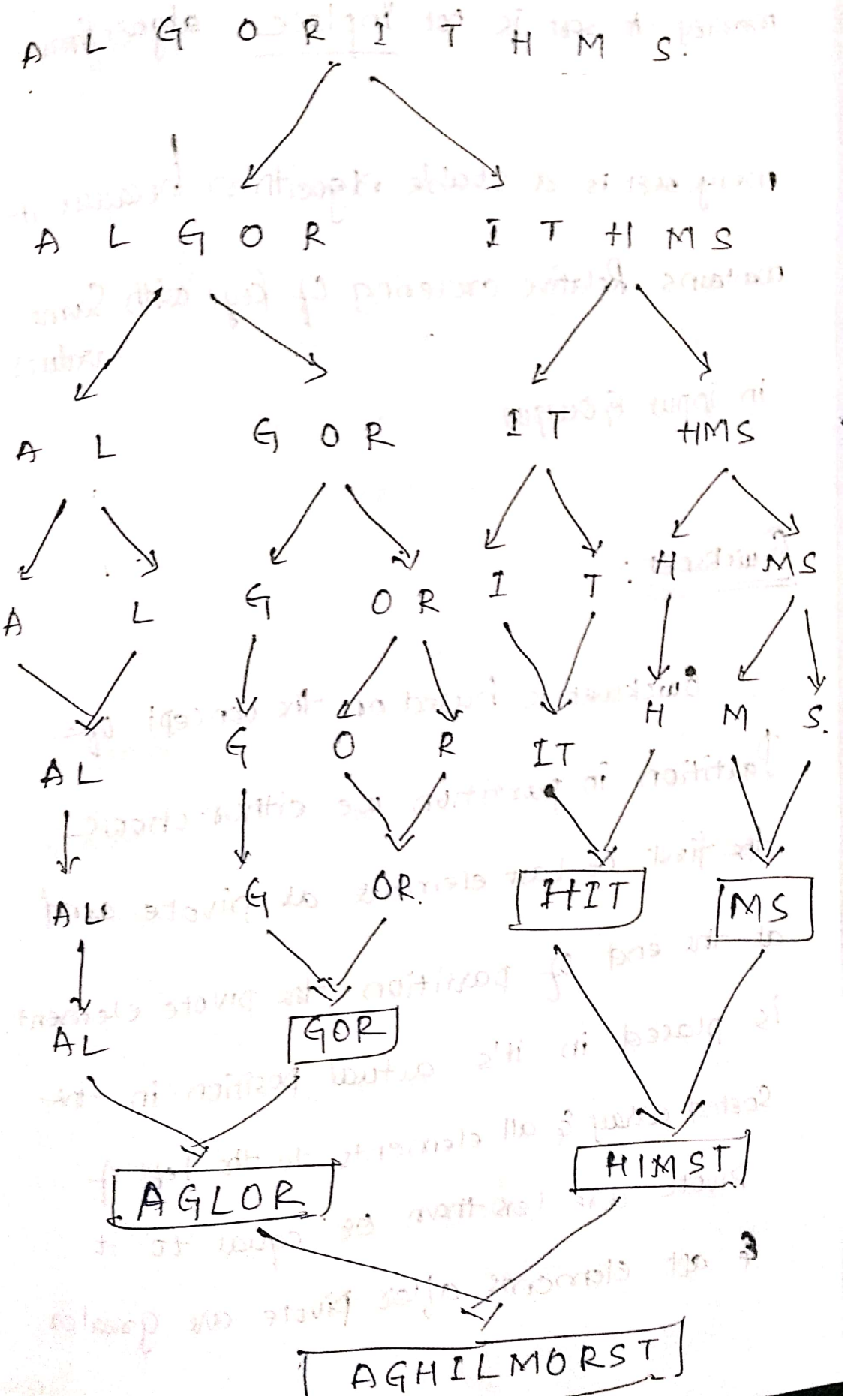
$C(n) = \Theta(n \log n)$

The actual number of comparison is less when compared to $n \log n$ which is $C(n) = n \log n - (n-1)$

$C(n) = n \log n - (n-1)$

Apply mergesort on the input "algorithms"

Characters of the word



Note:

Any sorting algorithm which uses additional memory to sort is not inplace algorithm.

mergesort is a stable algorithm because it maintains Relative ordering of keys with same value in input & output.

Quicksort:

Quicksort is based on the concept of Partition in partition we either choose the first or last element as pivote and at the end of partition the pivote element is placed in it's actual position in the sorted array & all elements to the left of Pivote are less than or equal to it. & all elements after pivote are greater.

Algorithm: QuickSort(A, l, r)

// recursively sorts the elements using Q.S

// input :

// output :

```
if l < r
    s ← partition(A, l, r);
    Quicksort(A, l, s-1);
    QuickSort(A, s+1, r);
```

Algorithm: partition (A, l, r)

// Partitions a subarray A indexed from l to r

// input : Array A

// output : it places the pivot element in its actual position in the sorted array

```
Pivot ← A[l]
i ← l+1
j ← r
while True
    while pivot ≥ A[i] and i<=r
        increment i
```

```
while pivot < A[j]
        decrement j
if i < j
        swap (A[i] and A[j])
else
        swap A[j] and A[low]
        return j
```

Analysis:

Basic operation: comparison

in each partition if pivot element divides the array into two equal parts then the efficiency is $n \log n$ which is the best case.

$$C_{best}(n) = \Theta(n \log n)$$

wurst case

If the pivot element after calling partition if it's place remains uncharged / its place is at the other end. then the recurrance can be

$C_{worst}(n) = c(n-1) + (no.\ of\ comparisons)$

$C(n) = c(n-1) + (n+1)$.

Substitution method.

$C(n) = \frac{(n+1)(n+2)}{2} - 3.$

$\simeq n^2$

The average case lies in between the best and the worst case which is equivalent to $\underline{1.38\ n \log n}$.

Apply Quicksort for the input sheets of the word "mergesort"

M E R G E S O R T

↑ i ↑ j

multiplication of large integers. (Divide and Conquer)

In brute force the time complexity to multiply a digit of size 'n' is n^2. Using divide and conquer strategy we try to reduce the number of multiplication operation.

let a & b two numbers of size 'n' whose product is denoted with $c = a * b$

let $a = a_1 a_0$ $b = b_1 b_0$

Size of $a_0 a_1$ & $b_0 b_1$ is $n/2$.

To find c we need to find the following

$c_0 = a_0 \times b_0$

$c_2 = a_1 \times b_1$

$c_1 = (a_0 + a_1) \times (b_0 + b_1) - (c_0 + c_2)$

$$C = C_2\ 10^n + C_1\ 10^{n/2} + C_0$$

E.g:

$a = 48 \qquad b = 12.$

$a_1 = 4 \quad a_0 = 8 \qquad b_1 = 1 \quad b_0 = 2$

$C_0 = a_0 \times b_0 = 8 \times 2 = 16$

$C_2 = a_1 \times b_1 = 4 \times 1 = 4$

$C_1 = (a_0 + a_1) \times (b_0 + b_1) - (C_0 + C_2)$

$= 12 \times 3 - 20 = 36 - 20 = 16$

$$C = C_2 10^n + C_1 10^{n/2} + C_0$$

$$C = 4\ 10^2 + 16 \cdot 10^{2/2} + 16$$

$$C = \underline{576}$$

(2) Find the product of 4231 and 223

Sol:

$a = 4231$ $\qquad$ $b = 223$

$\qquad\qquad\qquad\quad 0223$

$a_1 = 42$ $\qquad$ $b_1 = 02$

$a_0 = 31$ $\qquad$ $b_0 = 23$

$c_0 = a_0 \times b_0 = 31 \times 23 = 713$

$c_2 = a_1 \times b_1 = 42 \times 02 = 84$

$c_1 = (a_0 + a_1) \times (b_0 + b_1) - (c_0 + c_2)$

$= 73 \times 25 - 797$

$= 1028$

$c = c_2 10^n + c_1 10^{n/2} + c_0$

$c = 84\ 10^4 + 1028\ 10^2 + 713$

$= 943513$

(2) Find the product of 4231 and

Sol:

$a = 4231$ $\qquad$ $b = 223$

$\qquad\qquad\qquad\quad 0223$

$a_1 = 42$ $\qquad$ $b_1 = 02$

$a_0 = 31$ $\qquad$ $b_0 = 23$

$c_0 = a_0 \times b_0 = 31 \times 23 = 713$

$c_2 = a_1 \times b_1 = 42 \times 02 = 84$

$c_1 = (a_0 + a_1) \times (b_0 + b_1) - (c_0 + c_2)$

$= 73 \times 25 - 797$

$= 1028$

$c = c_2 \, 10^n + c_1 \, 10^{n/2} + c_0$

$c = 84 \, 10^4 + 1028 \, 10^2 + 713$

$= 943513$

let

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \qquad B = \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix}$$

$$C = \begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix}$$

$C_1 = E + I + J - G$

$C_2 = D + G$

$C_3 = E + F$

$C_4 = D + H + J - F$

$D = A1 \cdot (B_2 - B_4)$

$E = A_4 \cdot (B_3 - B_1)$

$F = (A_3 + A_4) \cdot B_1$

$G = (A_1 + A_2) \cdot B_4$

$H = (A_3 - A_1) \cdot (B_1 + B_2)$

$I = (A_2 - A_4) \cdot (B_3 + B_4)$

$J = (A_1 + A_4) \cdot (B_1 + B_4)$

e.g:

(1) Apply divide and Conquer to find the product of

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Sol:-

$A_1 = 1$ $\quad B_1 = 5$

$A_2 = 2$ $\quad B_2 = 6$

$A_3 = 3$ $\quad B_3 = 7$

$A_4 = 4$ $\quad B_4 = 8$.

$D = A_1 \cdot (B_2 - B_4) = 1 (6-8) = -2$

$E = A_4 \cdot (B_3 - B_1) = 4 (7-5) = 8$

$F = (A_3 + A_4) \cdot B_1 = 7 \times 5 = 35$

$G = (B_1 * A_2) \cdot B_4 = 3 \times 8 = 24$

$H = (A_3 - A_1) \cdot (B_1 + B_2) = 2 \times 11 = 22$

$I = (-2) \cdot (15) = -30$

$J = (5)(13) = 65$

$C_1 = 8 + (-30) + 65 - 24 = 19$

$C_2 = 22$

$C_3 = 43$

$C_4 = 50$

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Analysis

The recurrance relation can be denoted as $M(n) = 7\,M\left(\frac{n}{2}\right) + Cn$

base case $M(1) = 1$

On Applying master theorem

$$M(n) = \Theta\left(n^{\log_b^a}\right) = \Theta\left(n^{\log_2^7}\right)$$

$$= \Theta\left(n^{2.81}\right)$$

Decrease and Conquer

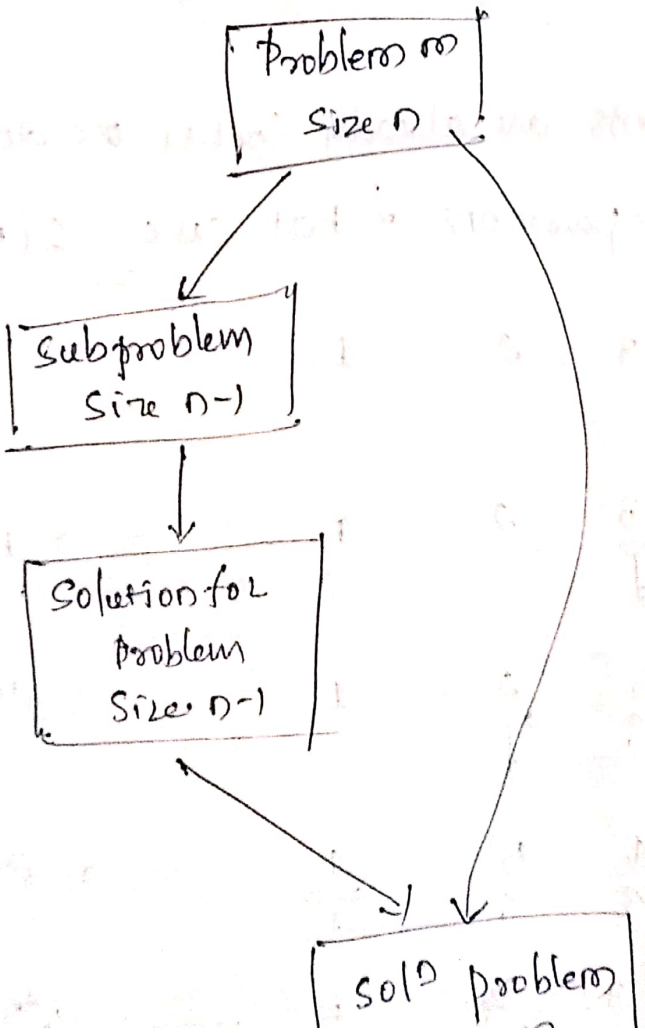
There are 3 variants of Decrease & Conquer

(1) Decrease by a constant
(Decrease by one)

(2) Decrease by constant factor (Fake coin problem, Josephers problem)

(3) variable size decrease. To find median (partition)

Decrease by one (constant)



e.g: $a^n = a^{n-1} \cdot a$

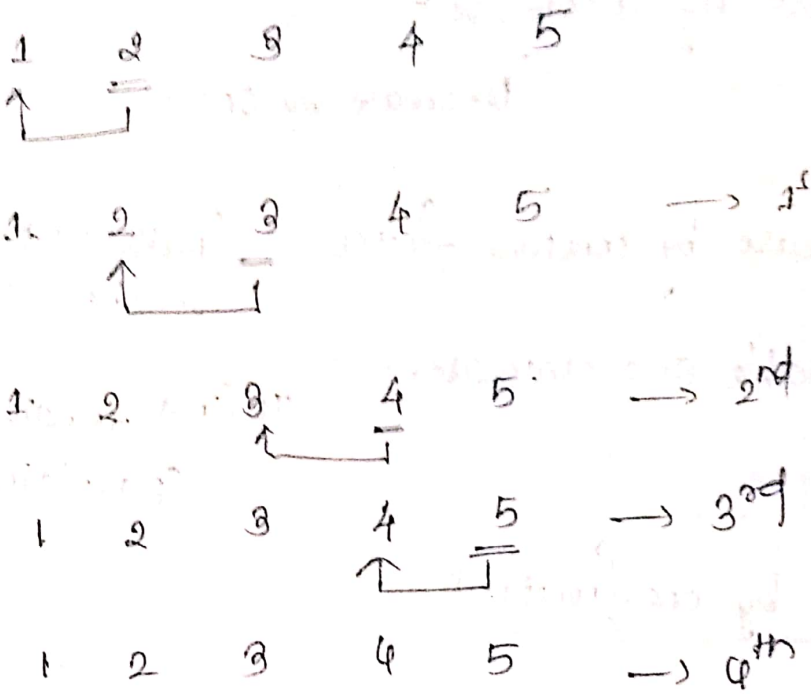
Insertion sort

Graph traversal
↳ DFS
BFS.

Insertion Sort

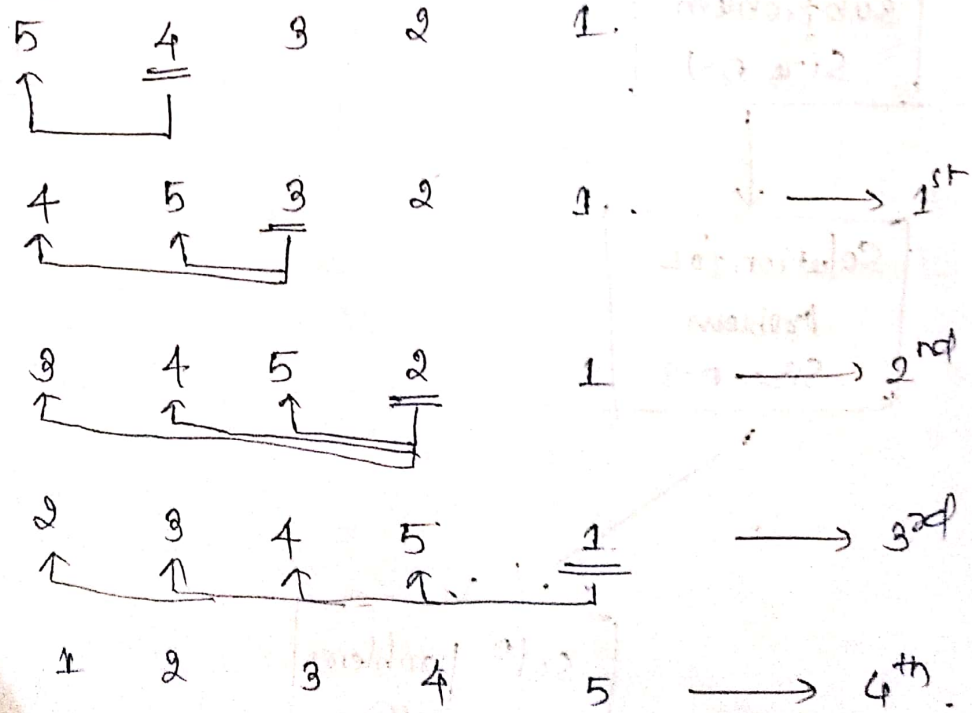
Case-I

$n = 5$



When elements are already sorted we do only $(n-1)$ comparisons. & Best case $O(n)$

Case-II



total comparisons $\frac{n(n-1)}{2} \approx O(n^2)$

worst case $O(n^2)$

Sort the characters of the word ALGORITHM using insertion sort & find the numbers of comparisons we do.

Sol:

A L G O R I T H M.

A L G O R I T H M → 1st

A G L O R I T H M → 2nd

A G L O R I T H M → 3rd

A G L O R I T H M → 4th

A G I L O R T H M → 5th

A G I L O R T H M → 6th

A G H I L O R T M → 7th

A G H I L M O R T → 8th

Algorithm : InsertionSort (A[0 ... n-1])

// sorts the elements using insertionsort

// input : Array

// output : Sorted Array

```
for i ← 1 to n-1 do
    item ← A[i]
    j ← i - 1
    while (j >= 0) and A[j] > item
        A[j+1] ← A[j]
        j ← j - 1
    A[j+1] ← item
```

Best case : if all the elements are already sorted

we do only one comparison in each iteration.

$\therefore$ efficiency is $\sum_{i=1}^{n-1} 1 = n-1-1+1 = n-1$

$\simeq O(n)$

worst case: If the elements are in decreasing order then in each iteration we do max no. of comparisons. which can be represented by

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i-1+1$$

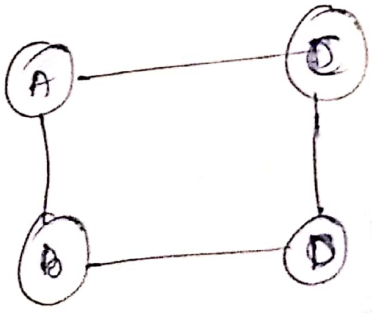
$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$$\simeq O(n^2)$$

Graph traversal :=

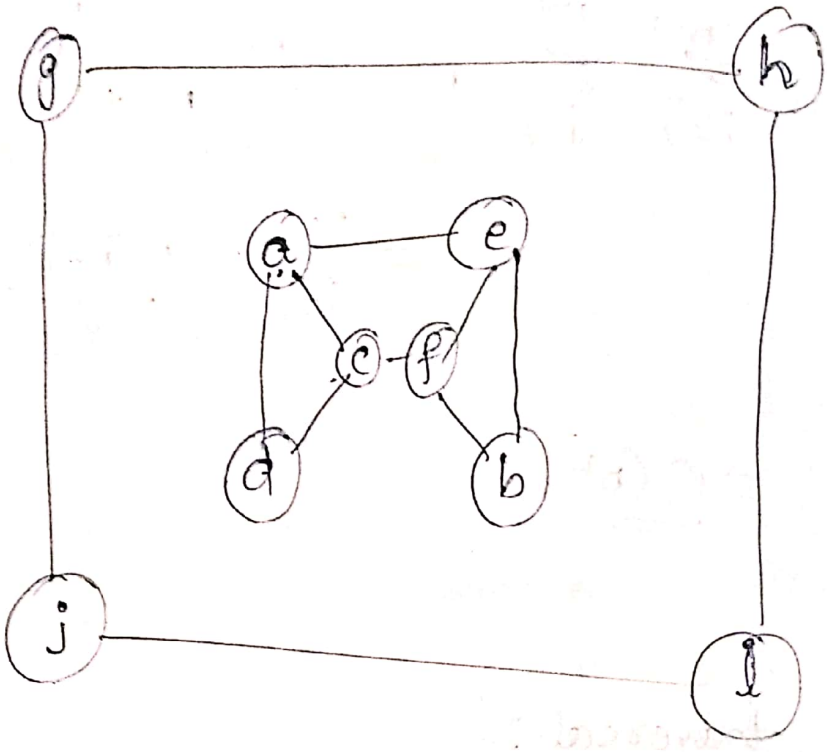
* Difference b/n tree traversal & graph traversal.

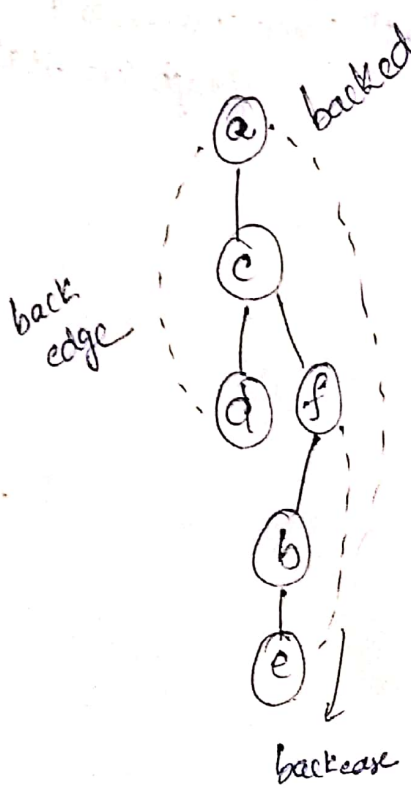
DFS



A 1, 4
B 2, 3
D 3, 2
C 4, 1

Perfoem DFS traversal on the input





$a_{1,6}$

$c_{2,5}$

$d_{3,1}$

$f_{4,4}$

$b_{5,3}$

$e_{6,2}$

$g_{7,10}$

$h_{8,9}$

$i_{9,8}$

Heapsort

is a binary tree having parent dominance & structural property (Almost Complete binary tree).

2 type of heaps

(1) max heap (2) min heap

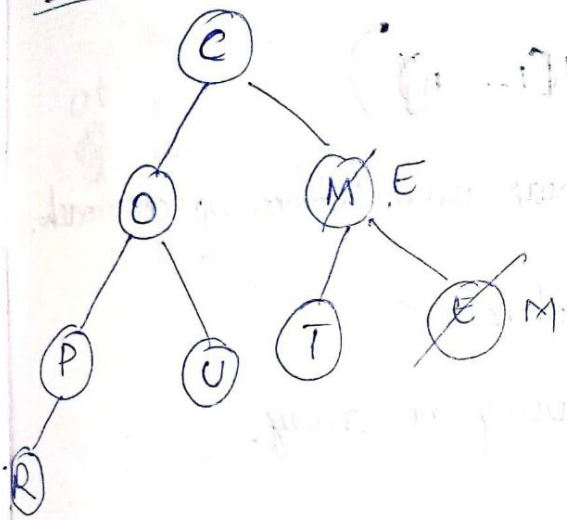
2 methods of Constructing

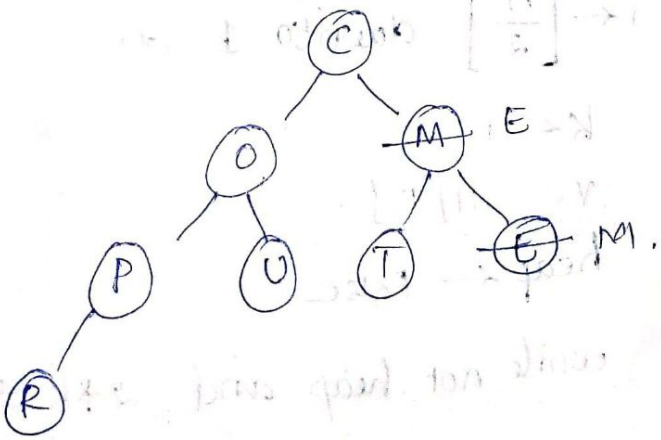
(1) Top down (2) bottom up.

e.g: Construct a min heap for word COMPUTER using both heap.

min

use bottom up if not mentioned.





complete binary

our of n nodes in tree

$\lceil \frac{n}{2} \rceil$ = leaf node

ceil

$\lfloor \frac{n}{2} \rfloor$ = non-leaf node

Algorithm to construct heap

Algorithm: Heapify(H[1...n])

// constructs a max heap using bottom up approach.

// input: an array of n elements

// output: heap represented using an array.

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i
    v ← H[k]
    heap ← False
    while not heap and 2*k ≤ n do.
        j ← 2*k
        if (j < n)
            if ( H[j] < H[j+1])
                j ← j+1
```

if $v \geq H[j]$

heap ← True

else

$H[k] \leftarrow H[j]$

$k \leftarrow j$

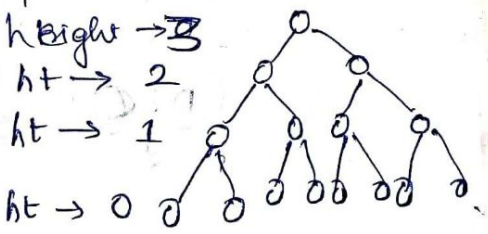
$H[k] \leftarrow v$

Efficiency of constructing a heap.

The total no. of node in Complete binary tree at height h is given by $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

where

n = no. of nodes in Complete binary tree



at height 3.

$\left\lceil \frac{15}{2^4} \right\rceil = 1.$ $\quad ht\ 0 = \left\lceil \frac{15}{2} \right\rceil = 8.$

$ht\ 2 = \left\lceil \frac{15}{2^3} \right\rceil = 2$

$ht\ 1 = \left\lceil \frac{15}{2^2} \right\rceil = 4$

while constructing a heap we do one comparison for a node at height one.

we do two comparison for a node at height 2.

∴ The number of comparison to convert a tree of height 'h' into a heap is denoted by

$$\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

$$\sum_{h=0}^{\log n} \frac{n}{2^h \cdot 2} \; ch.$$

$$\frac{(n)(c)}{2} \sum_{h=0}^{\log n} \frac{h}{2^h}$$

↳ is a gp.

$$\frac{nc}{2}(2) = n \cdot c$$

$$= O(n)$$

Comparison

node

nvert a

noted

Algoeithm foe heapsoet :

Algoeithm : heapsoet (H[1 - - - n])

//

//

//

Heapify (H[1 - - - n])

foe i ← n downto 1 do

swap (H[1] and H[i])

Heapify (H[1 - - - (n-1)])

efficiency of heapsoet.

Efficiency depends on time taken to Construct heap and time taken to construct heap foe n-1 elements aftersswapping first & last element.

efficiency = heapify (n) + heapif(n-1) after Swapping 1st & last element

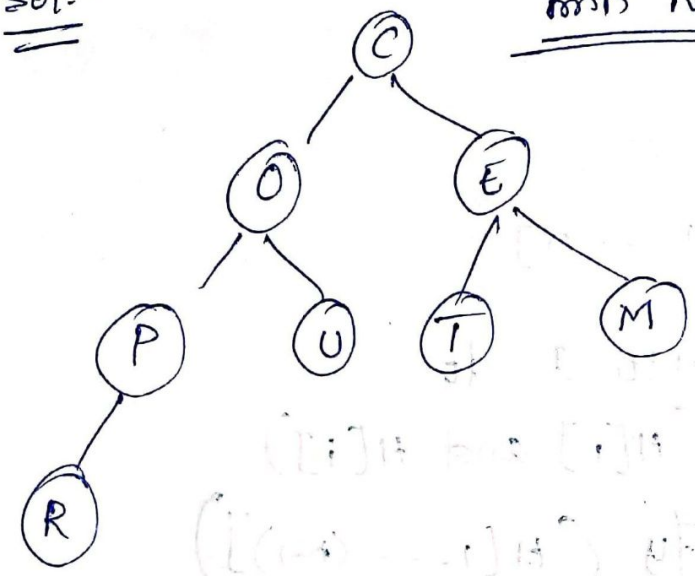
$= O(n) + (n-1) \log n.$

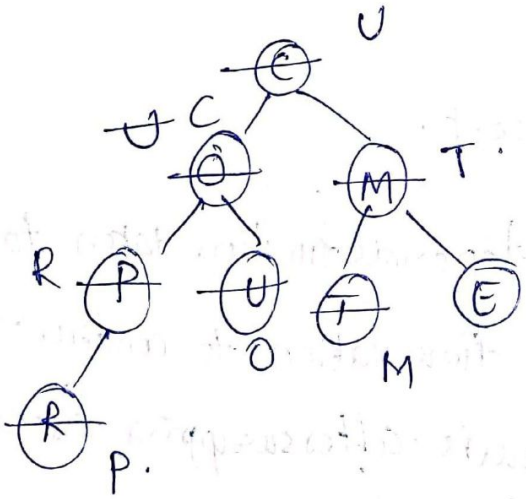
$= O(n \log n)$

Perform heapsort for the characters of the word C O M P U T E R.

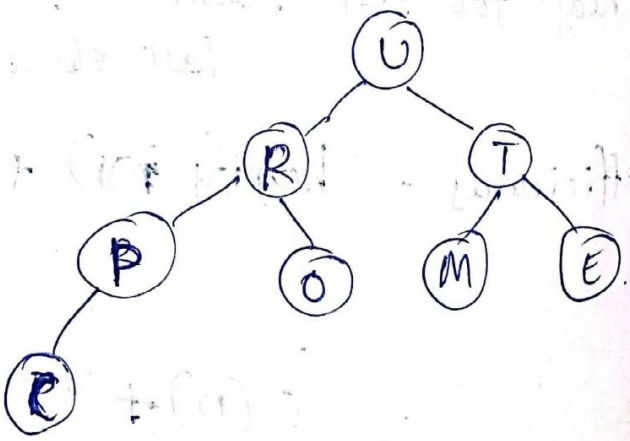
Sol:-

min heap





max



COMPUTER.

URTPOMEC.

CRTPOME|U.

TRMPOCE

ER.MPOC.|TU..

R.PMEO..C|TU.

CPMEO|RTU.

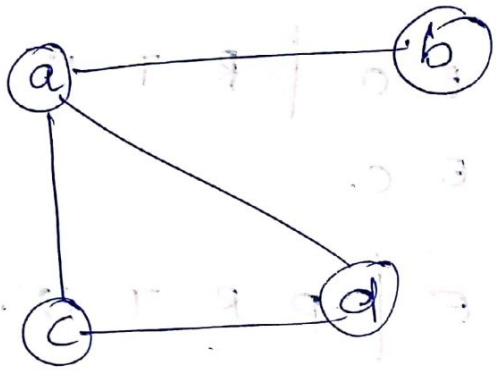
POMEC.

COME|PRTU.

Problem Reduction

Problem 1 → solution of problem 2 → solution to problem 2.

counting number of Paths in a graph:



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

The above matrix gives the info about how many edges are required to reach one vertex from another vertex.

$$A^2 = A \cdot A = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix}$$

Input Enhancement :

String matching

In the string matching problem, the main string is generally referred as text denoted with T and it's size is n

The substring to be searched is called as pattern denoted with 'p' and it's size is m.

Naive string matching algorithm.

Algorithm : NaiveString Matching(T[0..n-1], P[0...m-1])

```
//
//
//.
for i ← 0 to n-m do
    j ← 0
    while j < m and P[j] = T[i+j]
        j ← j + 1
    if (j = m)
        return 1.
return 0.
```

The basic operation is Comparison & the number of times this Comparison gets executed can be denoted with

$$\sum_{i=0}^{n-m} \sum_{j=1}^{m} 1.$$

in each iteration if we do m number of comparison → worst case

∴ Worst case efficiency

$$= \sum_{i=0}^{n-m} m - 1 + 1 = \sum_{i=0}^{n-m} m.$$

$$= m(n-m+1)$$

$$= nm - m^2 + m \longrightarrow (*).$$

$$= O(nm)$$

Best case for pattern not there in text or at the end can be denoted

$$\sum_{i=0}^{n-m} 1 = n-m+1$$

The best case efficiency of naive string matching $O(m)$.

(1) In a text of thousand Zero's find the number of Comparison made to find the pattern 001

Sol:- $n = 1000$

$m = 3$.

$$m(n-m+1) = 3(998) = 2994$$

for the same text, i.e; thousand Zero's.

and pattern 100.

the number of Comparison made is

$$n-m+1 = 998$$

Note:

The drawback of Naive string matching Algorithm is that the shift size is limited to one. to overcome this drawback we can do some preprocessing to pattern

Horspools string matching algorithm:

lets assume that the pattern to be Searched is B A R B E R

case I :

T : S0 S1 S2 - - - ~~Sn-1~~ X - - - - Sn

P : B A R B E R

The charecter in the text Compared with R i,e; X is not a part of the pattern

∴ The shift size should be m (where m is length of pattern).

shift size = 6

case II :

T : S0 S1 S2 - - - - B - - - - Sn-1

P : B A R B E R

The charecter in the text doesnot match with the last charecter of pattern.

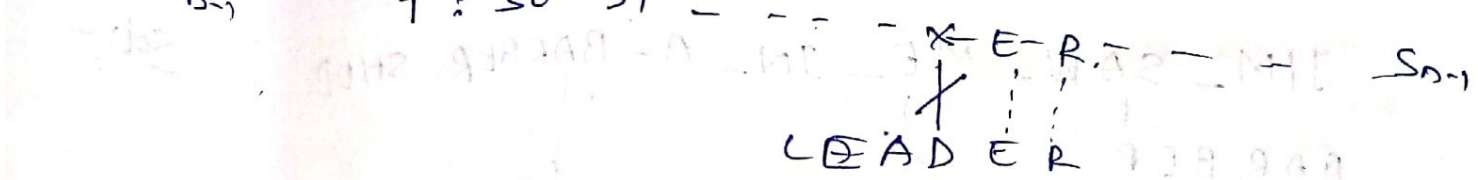
i,e; text charecter is B

but B is there in remaining m-1 character

∴ shift size should be the right most occurance of B in pattern.

shift size = 2.

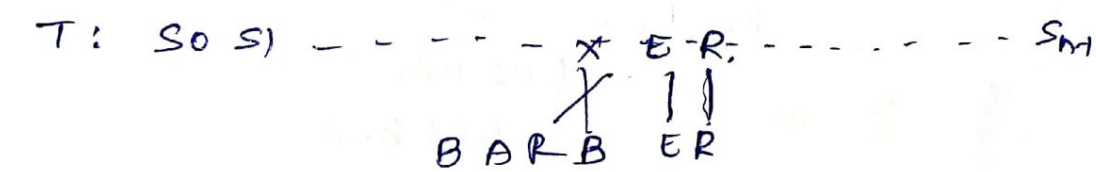
case (iii).



The last few characters matches with the text last character which is matched is 'R' but R is not there in remaining m-1 characters

∴ shift size = 6.

Case (iv)



shift value = 3.

Last few characters matches with the few char of text.

but R is also part of pattern.

∴ shift value = 3.

Construct a shift table for pattern barber.

B	A	R	B	E	R
2	4	3	2	1	

C = 6
D = 6
F = 6
_ = 6

JIM_SAW_ME_IN_A-BARBER_SHOP

BARBER
BARBER
BARBER
BARBER
BARBER
BARBER

ex. 2)

Text =

BESS_KNEW_ABOUT_BAOBAB

2 1 3 2 1

BAOBAB
BAOBAB
BAOBAB
BAOBAB
BAOBAB

Find the number of character comparison in finding the pattern EARN in the text

EARN
3 2 1

~~FAIL~~ _ ~~NEVER~~ _

Text:

FAIL_MEANS_FIRST_ATTEMPT_IN_LEARN.

EARN

EARN

EARN

EARN

EARN

EARN EARN EARN EARN

EARN.

Total 10 Comparison

Find no. of comparison to search a pattern

10000 in 100 Zeros.

4 1 1 1.

Sol:-

0000 - - - - - - - - - - - - 0 (100th)

10000.

number of iterations = n−m+1

= 100−5+1

= 96

no. of comparison in each iteration

96×5 = 480

Algorithm to construct shift table

```
shiftTable( P[0---m-1])
// constructs a shift table used in horspool/Boyer
                    moore algorithm.
// input: A pattern of length m
// output: A shift table -Table with shift
        -values

    initialize all the elements of table with m

    for i←0 to m-2 do.
        Table[P[i]] = m-1-i
    return Table.
```

Algorithm: Horspool(T[0---n-1], P[0---m-1])

```
//
//
// output: the first occurance position of pinT
shiftTable(P)
i ← m-1
while (i<=n-1)
{
        K←0. // no of match characters.
        while (K<=m-1 and P[m-1-K]=T[i-
                K←K+1;
```

```
if (k==m) return i-m+1;
else i ← i + Table[T[i]]
```

Boyermoore Algoeithm.

In Boyermoore Algoeithm, the number of charaters match is taken into consideration.

in boyermoore we construct two table mainly

(i) Bad shift table (d_1)

(ii) Good suffix table (d_2)

The bad shift table d_1 can be calculated as

d_1 = shift (non matching character in the text) — no.of.charaters matched.

The Good suffix table is calculated as

e.g: Good suffix table for ABCBAB

K.	pattern.	d_2
1.	ABCBA<u>B</u>.	2.
2.	ABCB<u>AB</u>	4
3	ABC<u>BAB</u>	4
4	AB<u>CBAB</u>	4

e.g.1

BESS_KNEW_ABOUT_BAOBAB.

BAOBAB

BAOBAB

$d_1 = 6 - 2 = 4$

$d_2(k=2) = 5$

BAOBAB

$d_1 = 6 - 1 = 5$

$d_2(k=1) = 2$

BAOBAB.

match.

eg 2 Text:

WHICH_FINALLY_HALTS.2_AT_THAT

Pattern : AT_THAT.

K	AT_THAT	d_2.
1	AT_THAT	3
2.	AT-THAT	5.
3.	AT-THAT	5.

WHICH_FINALLY_HALTS.2_AT_THAT

AT_THAT

AT_THAT

AT_THAT

$d_1 = 7 - 1 = 6$

$d_2(K=1) = 3.$

AT_THAT

$d_1 = 4 - 2 = 2$

$d_2(K=2) = 5$

AT_THAT

match.

construct a good suffix table for the pattern

0 1 0 0 1

Sol:-

K.	pattern	d_2
1.	0 1 0 0 1	~~8~~. 5 (size of pattern)
2.	0 1 0 0 1	3
3	0 1 0 0 1	3.

Procedure for Boyermoore String matching Algorithm

step 1

For the given pattern & the Alphabet used in both pattern and text, construct a bad symbol table d_1 (After calculating d_1, if d_1 is negative then take the shift size as 1)

i.e; d_1 = max { shift(x) - no. of matching char , 1 }

step 2

construct the good suffix table d_2 using the pattern

step 3

Allign the pattern against the beginning of the text.

step 4

Repeat this step until a matching Substring is found OR reach end of the text.

starting with last character pattern, compare the corresponding characters of the pattern & text

let 'k' denotes the number of matching char.

if character doesnot match after some comparison the shift size (d) is calculated as

$$d = \begin{cases} d_1 & \text{if } k=0 \\ \max(d_1, d_2) & \text{if } k>0. \end{cases}$$

Comparison count sort.

e.g: $A[0 \dots n-1] = \{32, 10, 6, 40\}$.

$count[0 \dots n-1] = \{2, 1, 0, 3\}$

A[0]	A[1]	A[2]	A[3].
6	10	32.	40.

Algorithm: ComparisonCountSort ($A[0 \dots n-1]$)

// Sort an array by

// Unsorted array A indexed from $0 \dots n-1$

// Sorted array S " " $0 \dots n-1$

```
for i ← 0 to n-1 do
    count[i] ← 0
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] > A[j]
            count[i]++
        else
            count[j]++
for i ← 0 to n-1 do.
    S[count[i]] ← A[i]
```

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

Distribution Countsort :

consider an Array A which contains the following elements as

13 11 12 13 12 13

whose values are ranged between 2 integers say x & y (x=11 & y=13 in above eg')

First we have to generate the frequency and the distribution value for the array elements

Array	11	12	13
Frequency	1	2	3
Distribution value	1	3	6

add → add →

step1 The distribution value indicates the proper position for the last occurance in the final sorted array

step2 it is better to process the element from right to left.

Algorithm: DistributionCounting (A[0---n-1])

//

// Array A indexed from 0 to n-1 between the range L and U

```
for i ← 0 to u-l do
    D[i] ← 0
for i ← 0 to n-1 do
    D[A[i]-l] ← D[A[i]-l] + 1
for j ← 1 to u-l do
    D[j] ← D[j-1] + D[j]
for i ← n-1 down to 0 do
    j ← A[i]-l
    S[D[j]-1] ← A[i]; D[j] ← D[j]-1
return S
```

Scanned by CamScanner

Dynamic programming.

Divide & conquere & Dynamic programing are similar in the sence the solution to the problem is expressed w.r.t the solution of the smaller instance of the same problem.

Difference is, in divided & Conquere the overlaping Subproblems are solved again & again but whereas in dynamic programming the overlapping Subproblem is solved only once & are stored in vector.

i) Binomial coefficient

Objective of Binomial coefficient is to find the number of combinations of 'k' elements in a set of 'n' elements. which can be denoted as n_{C_K} or $c(n, k)$.

The recursive difinition to find binomi Coefficient is $c(n, k)$ as

$$c(n,k) = \begin{cases} 1 & \text{if } k=0 \text{ or } k=n \\ 0 & \text{if } k>n \\ c(n-1,k-1)+c(n-1,k) & \end{cases}$$

eg: $^4C_3$

$n=4$
$k=3$.

n \ k →	0	1	2	3
0	1			
1	1	1.		
2	1	2	1.	
3	1	3	3	1.
4	1	4	6	(4.)

e.g: $^6C_4$ $n=6$ $k=4$

	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1.
5	1	5	10	10	5.
6	1	6	15	20	15

Algorithm : BinomialCoeff(n, k)

// computes c(n,k) using DP

// input :

// output : value of c(n,k)

```
for i ← 0 to n do.
    for j ← 0 to min(i, k)
        if j == 0 or j = i
            c(i, j) = 1.
        else
            c(i, j) = c(i-1, j) + c(i-1, j-1)
return c(n, k)
```

efficiency $\Theta(nk)$

(2) ~~All pair~~ shortest path algorithm

Algorithm : floyds

Algorithm: Floyd's (W[1---n, 1---n]

// Finds all pair shortest path using Dynamic programming

// Input: weighted matrix of graph G.

// Output: matrix which holds shortest path of all vertices.

D ← W.

for K ← 1 to n do.

for i ← 1 to n do

for j ← 1 to n do

D[i, j) ← min(D(i,j) ,

D(i, K) + D(K, j))

return D

Efficiency : $O(n^3)$

Prob 1: Trace floyd for the following input.

$$\begin{bmatrix} 0 & 2 & \infty & 1 & \infty \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & \infty & 4 & 0 \end{bmatrix}$$

	0
0	0
6	6
∞	∞
∞	∞
3	3

	∞
5	0
3	6
0	∞
2	∞
8	3

4
5
7
3
0

K

boolean

	0	2	∞	1	∞
0	0	2	∞	1	∞
6	6	0	3	2	∞
∞	∞	∞	0	4	∞
∞	∞	∞	2	0	3
3	3	5	∞	4	0

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

	∞	∞	0	4	∞
5	0	2	5	1	∞
9	6	0	3	2	∞
0	∞	∞	0	4	∞
2	∞	∞	2	0	3
8	3	5	8	4	0

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

	3	5	6	4	0
4	0	2	3	1	4
5	6	0	3	2	5
7	10	12	0	4	7
3	6	8	2	0	3
0	3	5	6	4	0

Warshall's algorithm.

Objective: to find transitive closureness

The transitive closureness of directed graph with n vertices can be defined as an n×n boolean matrix $T = \{t_{ij}\}$ in which the element in the ith row & jth column is 1 if there exist a path from i to j. Otherwise it is zero.

Algorithm : Warshall(A[1 - - - - n, 1 - - - n])

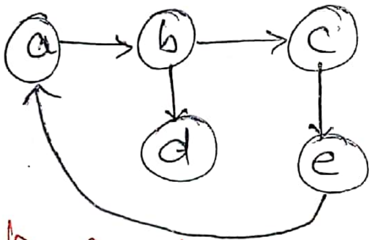
//
//
// .

$R^0 \leftarrow A$.

for $k \leftarrow 1$ to n do
    for $i \leftarrow 1$ to n do
        for $j \leftarrow 1$ to n do
            $R^k(i,j) \leftarrow R^{k-1}(i,j)$ or $R^{k-1}(i,k)$ and $R^{k-1}(k,j)$

return R^n.

Apply warshall for the input



	a	b	c	d	e
a	0	1.	0	0	0.
b	0	0	1.	1	0
c	0	0	0	0	1
d	0	0	0	0	0.
e	1	0	0	0	0.

R^1

	0	1	0	0	0
0	0	1	0	0	0
0	0	0	1	1	0
0	0	0	0	0	1
0	0	0	0	0	0.
1.	1	1.	0	0	0

R^2

	0	0	1	1	0
1	0	1	1	1.	0
0	0	0	1	1	0
0	0	0	0	0	1
0	0	0	0	0	0
1	1	1	1	1.	0.

R^3

	0	0	0	0	1.
1	0	1	1	1	1.
1	0	0	1	1	1.
0	0	0	0	0	1
0	0	0	0	0	0.
1.	1	1	1	1	1.

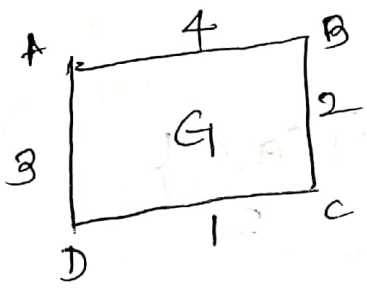
R4

	0	0	0	0	0
1	0	1	1	1	1
1	0	0	1	1	1
0	0	0	0	0	1
0	0	0	0	0	0
1	0	1	1	1	1
	1				

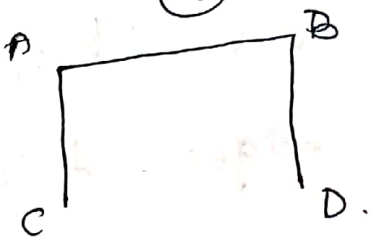
R5

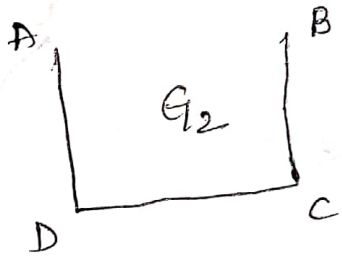
	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
0	0	0	0	0	0
1	1	1	1	1	1

Spanning tree



(9)





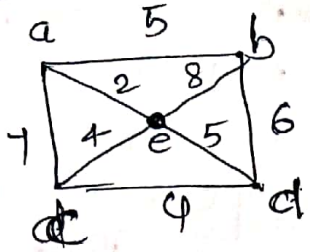
(6)

sub
acyclic connected graph which contain all the vertices

minimum cost spanning tree

↳ spanning tree with minimum weight

Prims Algo



$V = \{a, b, c, d, e\}$

$V_T = \{a\}$ // source

if n is size of graph there are n-1 iterations.

1st iteration

$V - V_T = \{b, c, d, e\}$

edges = $\{(a,b)\ (a,c)\ (a,e)\}$
5 7 2

$E_T = \{(a,e)\}$.

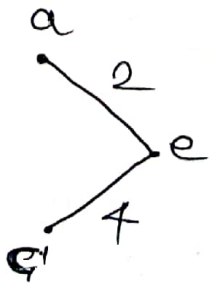
2nd iteration

$V_T = \{a, e\}$

$V - V_T = \{b, c, d\}$.

edges = $\{(a,b)\ (a,c)\ \cancel{(a,d)}\ (e,b)\ (e,c)\ (e,d)\}$
5 7 8 4 5
=

$E_T = \{(a,e)\ (e,c)\}$.



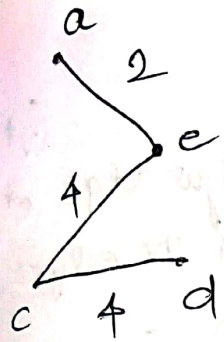
3rd iteration

$V_T = \{a, e, c\}$

$V - V_T = \{b, d\}$

edges = { (a,b) (e,b) (e,d) (c,d) }
5 8 5 4

$E_T = \{(a,e)\ (e,c)\ (c,d)$



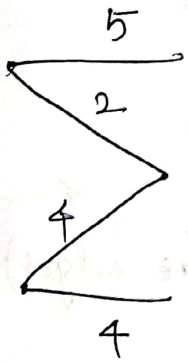
4th. iteration

$V_T = \{a, e, c, d\}$

$V - V_T = \{b\}$.

edges = { (a,b) (e,b) (d,b) }
5 8 6.

$E_T = \{(a,e)\ (e,c)\ (c,d)\ (a,b)\}$



(15) ← minimum cost.

Algorithm: prims(G)

// a connected weighted graph.

//

$V_T = \{v_0\}$

$E_T = \phi$

for $i \leftarrow 1$ to $|V|-1$ $(v^* u^*)$

find a minimum weight edge e^* among all the edges.

such that v is in V_T & u is in $V-V_T$

$V_T \leftarrow V_T \cup u^*$

$E_T \leftarrow E_T \cup e^*$

return E_T

efficiency

if input represented in the form of matrix

efficiency $|V|^2$

Adjacency list $|E| \log V$

(if min heap constructed for edges)

0/1 Knapsack

objective: to obtain max profit.

input: n objects.

for each of these object

$w_1 \longrightarrow P_1$

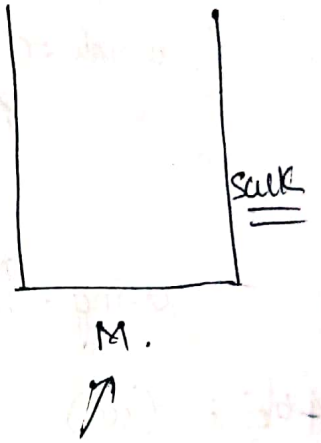
$w_2 \longrightarrow P_2$

$w_3 \longrightarrow P_3$ → profit.

⋮

$w_n \longrightarrow P_n$

→ weights



max capacity

for e.g:

if i choose 3 object

with weights w_1, w_2, w_3

then $w_1 + w_2 + w_3 \leq M$

↓ max capacity.

e.g. 1)

consider 4 objects with

	obj 1	obj 2	obj 3	obj 4
w =	2	1	3	2
P =	8	6	16	11

knapsack capacity 5.

Brute force

⇒

1.	2.	3	4	← Object Combination
(8)	(6)	(16)	(11)	← Profit

1,2	1,3	1,4	2,3	2,4	3,4
(14)	(24)	(19)	(22)	(17)	(27)

↑ most optimal.

1,2,3	1,2,4	2,3,4
X (∵ weight = 6 > 5) ∉ 5.	(25)	X

using - Dynamic programming.

4 objects. (wo) 2. 1 3 2

(P) 8 6 16 11

M = 5.

M+1 columns.

	0	1	2	3	4	5
0	0	0	0	0	0	0 ← directly write non zero value
1	0	0	8	8	8	8
2	0	6	8	14	14	14
3	0	6	8	16	22	24
4	0	6	11	17	22	27

(n+1) rows

if.

$V[i,j] = V[i-1,j]$ $(j - w_i < 0)$

else $= \max\{ V[i-1,j], V[i-1, j-w_i] + P_i \}$

i = rows.

j = columns.

$V(1,1) = V[1-1, 1] = V[0,1]$

$= \underline{0}$

$i=1 \quad j=1$

$j - w_i < 0$

$1 - 2 = -1 < 0$ ✓

$V(1,2) = \max\{ V[1-1, 2], V[1-1, 2-2] + P_1 \}$

$= \max\{ 0, 0+8 \} = \underline{8}$.

$i=1 \quad j=2.$

$j - w_i = 2-2 = 0 < 0$ ✗

$V(1,3) = \max\{ V[0,3], V[0,1] + 8 \}$

$= \underline{8}$.

$i=1$

$j=3$

$3 - 2 = 1 < 0$ ✗.

$V(2,1) = \max\{ V[1,1], V[1,0] + 6 \}$

$= \underline{6}$

$i=2$

$j=1$

$1-1 = 0 < 0$ ✗

$V(4,5) \neq V(3,5)$

$24 \neq 24$.

Therefore 4th Obj should be Considered.

$V(3, 5-2) \Rightarrow V(3,3) \neq V(2,3)$

$16 \neq 14$.

4th object weight.

$\therefore$ 3rd Object to be considered (included)

$V(2, 3-3) = V(2,0)$

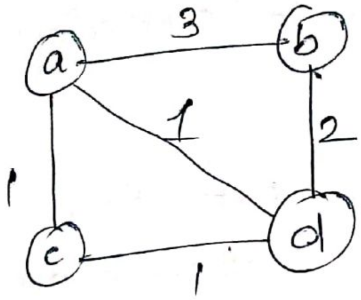
weight of 3rd Obj

$j = 0 \Rightarrow$ knapsack don't have space / extra

$\therefore$ objects are 3rd & 4th.

Kruskal's Algo

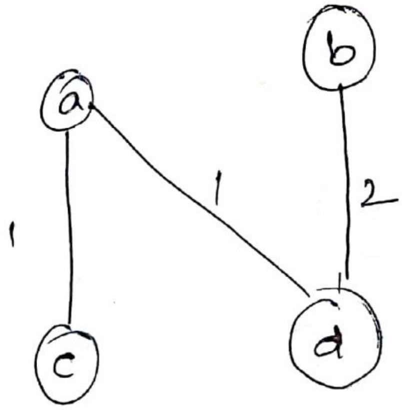
↳ objective is to find minimum spanning tree



Sort all edges.

ac – 1 ✓
ad – 1 ✓
cd – 1 ✗
bd – 2
ab – 3

sidered
ded)



= (4)

don't

Algorithm : Kruskals(G) $G = \{V, E\}$

//
//
//

sort E in the increasing order

$E_T = \phi$ //initially

ecounter = 0 // no. of iterations.

K=0 // no. of edges processed.

while ecounter < |V|–1 do

$K \leftarrow K+1$

if $E_T \cup \{e_{ik}\}$ is acyclic

$E_T \leftarrow E_T \cup \{e_{ij}\}$

return   ecounter ← ecounter +1.

Complexity / Efficiency

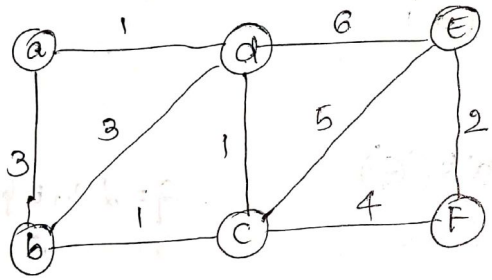
↳ depends on Sorting technique used to sort edges.

efficiency depends on = Sort E + Extracting those edges which are part of S.T

$= E \log E + E$

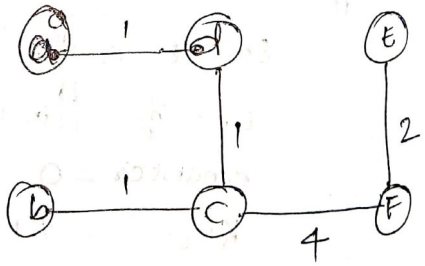
$= O(E \log E)$

e.g. (1)



sol:-

ad – 1
bc – 1
cd – 1
EF – 2
ab – 3
bd – 3
CF – 4
CE – 5
dE – 6.



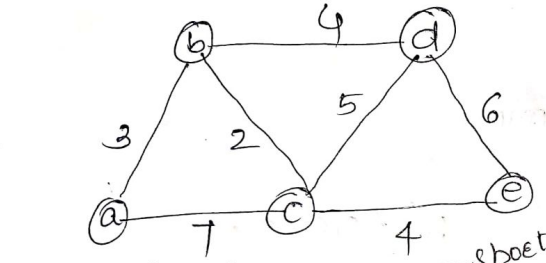
(9)

Feasible, locally optimal & irrevocable

The choices made in each step in any greed method should have these properties.

Single source shortest paths:

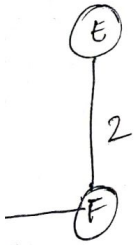
↳ Dijkstra's algo



initialization step

vertex	shortest distance d[]	parent P[]
a	0	NIL
b	∞	NIL
c	∞	NIL
d	∞	NIL
e	∞	NIL

source = a



W = { a, b, c, d, e } ← it Contains all the vertices.

↓ extract min.

a.

Sol^n = ϕ ← initially.

Sol^n = {a}.

after 1st iteration

vertex	d[]	p[]
a	0	NIL
b	3	a
c	7	a
d	∞	NIL
e	∞	NIL

W = {b, c, d, e}.

↓ extract min.

b.

Sol^n = {a, b}. W = {c, d, e}.

iter^n - 2

vertex	d[]	p[]
a	0	a
b	3	a
c	~~7~~ 5	b
d	7	b
e	∞	NIL

Item - 3

$W = \{c, d, e\}$

↓ extract min

c

$soln = \{a, b, c\}$

vertex	d[]	p[]
a	0	a
b	3	a
c	5	b
d	7	b
e	9	c

4th

$W = \{d, e\}$

↓ extract min.

d.

$soln = \{a, b, c, d\}$

or.

value of d[] & p[] are same as 3rd iteration

Algorithm: Dijkstra (G, s)

```
for each v in V do
        d[v] ← ∞.
        P[v] ← NIL
d[s] ← 0
soln ← ϕ
W ← V
```

} initialization step

while $W \neq \phi$

$u \leftarrow$ extractmin(W)

$Sol \leftarrow Sol \cup \{u\}$

$W \leftarrow W - \{u\}$

for each vertex v in V adjacent to u

relaxation ←

if $d[v] > d[u] + c(u,v)$

$d[v] = d[u] + c(u,v)$

$p[v] = u$

return d

Sol:- initialize

$W = \{a, b, c, \ldots$

↓ extr

a.

Dis-advantage

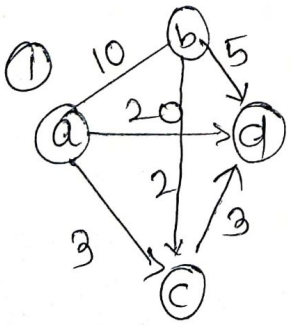
Dijkstra's algo fails whenever there is negative cycle (any of edge with -ve value)

efficiency.

have. input → adjacency list & heap constructed

$|E| \log V$. for all the edges.

e.g



source = a

to u

iv)

is

value)

ap

cted

ne

Sol:- initialize

vertex.	d[ ]	p[ ].
a	0	NIL
b	∞	NIL
c	∞	NIL
d.	∞.	NIL

$w = \{a, b, c, d\}$

↓ extractmin

a.

Huffman coding (tree)

↳ objective — used in data compression
or
data encoding
↓
("integeity")

when frequency of every char same

Fixed length

A B C D E F.
↓
A - 000
B - 001
C - 010
D - 011
E - 100
F - 111

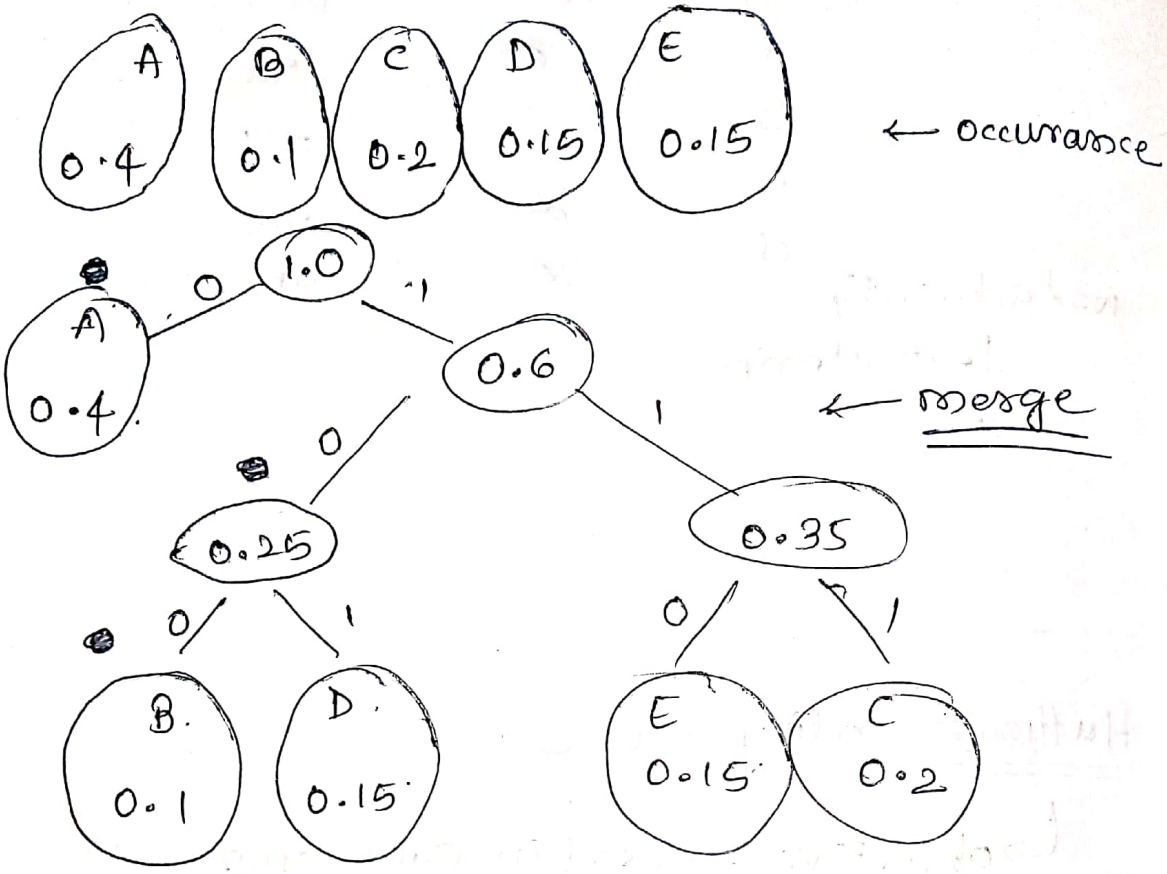
} 3 bits.

variable length

A B B C A A A D C

4 A's — 0 freq ↑
2 B's — 10
2 C's — 11
1 D — 100 freq ↓.

eg: character to be transmitted



A — 0

B — 1 0 0

C — 1 1 1

D — 1 0 1

E — 1 1 0,

encoding

A B A C A B A D

0 1 00 0 1 1 1 0 1 0 0 0.

← occurance

— merge

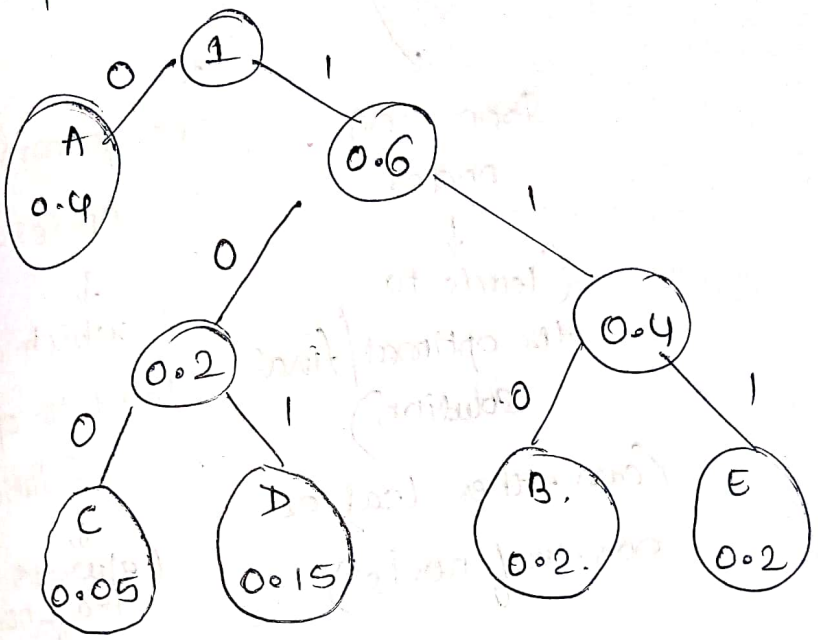
C
0.2

A D

construct huffman tree

eg.2)

A	B	C	D	E
0.4	0.2	0.05	0.15	0.20



A — 0
B — 110
C — 100
D — 101
E — 111

unit-5 < Backtracking
Branch & bound.

Backtracking

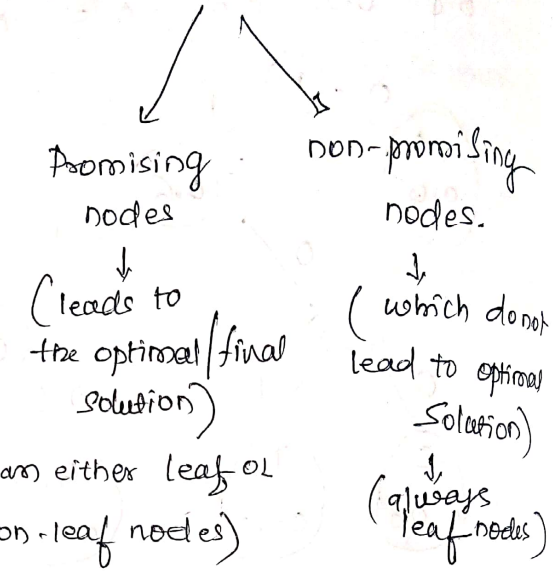
Sum of Subset's problem. → objective → To find the subsets whose summation is equal to an integer d.

$S = \{1, 2, 3, 4, 5\}$

$d = 7$.

$S_1 = \{1, 2, 4\}$
$S_2 = \{3, 4\}$
$S_3 = \{2, 5\}$.

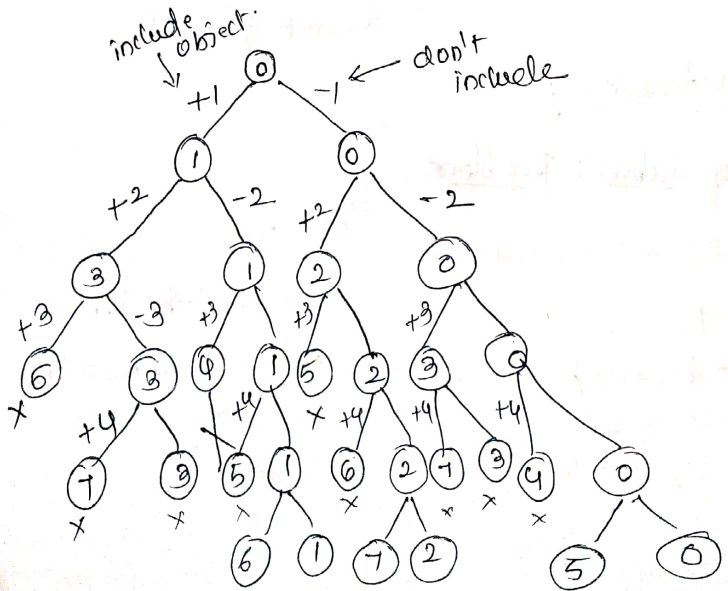
while solving Back-tracking we construct a binary tree. → state space tree



$S = \{4, 1, 2, 6, 9\}$.

$d = 7$ ↓ sort S.

$S = \{1, 2, 3, 4, 5\}$

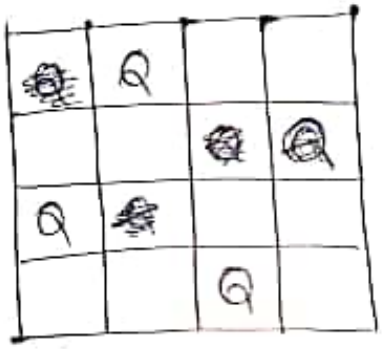


n Queen's problem (using backtracking)

↓ objective is to

.. place n Queen's on a n×n chess board in so that no two queens attack each other

(i) if they are in same row
(ii) if " " in " column
(iii) " " " diagonal to each other



n = 4

$Q_1 = (1,2)$
$Q_2 = (2,4)$
$Q_3 = (3,1)$
$Q_4 = (4,2)$

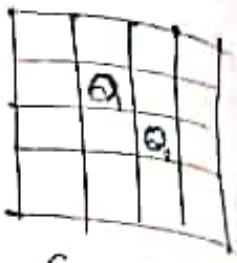
$x[1,2,3,4] = \{2,4,1,3\}$

$= \{3,1,4,2\}$.

x[1] = 2
⇒ placing 1st queen at (1,2)

To check whether they are diagonal to each other

$Q_1 = (i, j)$ $\quad Q_2 = (k, l)$

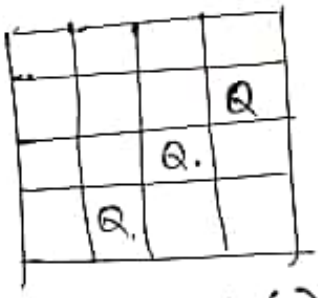


$Q_1 = (2, 2)$

$Q_2 = (3, 3)$

if $i - j = k - l$ → diag

$\Rightarrow \boxed{j - l = i - k}$ — ①



(2, 4) (3, 3) (4, 2)

$i + j = 6 \quad k + l = 6$

$\boxed{i + j = k + l} \Rightarrow \boxed{j - l = k - i}$ — ②

Combining ① & ②

$\boxed{Abs(j - l) = Abs(i - k)}$

Algorithm: nqueens(k, n)

k → queen number

n → number of queens

```
for i ← 1 to n do
    if place(k, i)
        x[k] ← i.
        if (k = n)
            print x[1...n]
        else
            nqueens(k+1, n)
```

Algorithm: place(k, i)

it returns

// True — if Queen can be placed at k^{th} row, i^{th} column

// False — if it can't be placed.

```
for j ← 1 to (k-1)
    if (x[j]=i) or
        Abs(x[j]-i) = Abs(j-k)
    return false.

return True
```