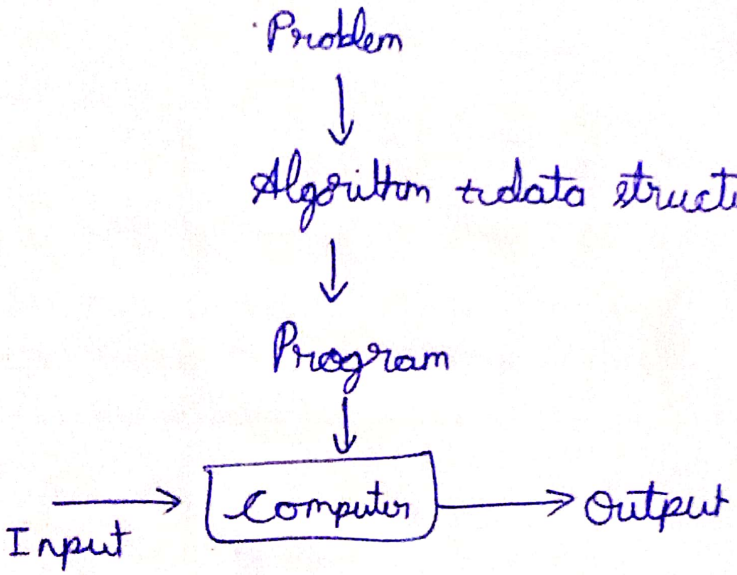
UNIT-1

Algorithm:

- A method to solve a problem
- "A set of unambiguous instruction to solve a problem."
- A set of instructions which transforms an input to an output.

Notion of an algorithm:



Properties:

* Input - zero or more
* Output - one or more
* Definitness [should terminate]
* Finiteness [clarity]
* Effectiveness

Algorithms to find GCD of Two numbers

1) Euclid's algorithm:

$GCD(a, b) = GCD(b, a \bmod b)$

Input : 2 non-negative numbers a and b
Output: Greatest common divisor of both a and b

Step 1 : If b is 0, GCD is a, if not go to step 2
Step 2 : Find remainder r after dividing a and b
Step 3 : Exchange a by b, b by r

Pseudocode is a method of expressing the solution to a problem written with the help of programming construct with english statement.

```
Algorithm GCD(a, b)
//Input : 2 non-negative no's a & b
//Output : GCD of a & b
// Finds GCD using Euclid's algorithm

while b≠0 do
        r ← a mod b
        a ← b
        b ← r
return a
```

2) Consecutive integer checking method:

In:
Out:

Step 1 : design $t = min(a, b)$

Step 2 : perform a mod t, if remainder is 0 goto step 3. else goto step 4.

Step 3 : perform b mod t, if remainder is 0 gcd is t. else goto step 4.

Step 4 : Decrement value of t by 1.

3) middle school method:

Step 1 : Find prime factor of a

Step 2 : Find prime factor of b

Step 3 : Find the common highest factor.

Sieve of Eratosthenes :

To generate prime numbers from 1 to n

// Input : An integer n

// Output : Prime numbers from 1 to n

Step 1 : Store the numbers from 2 to n in an array

or

Step 2 : Discard all multiple of 2 (excluding 2)

Step 3 : Discard all multiples of 3 (excluding 3) (only)

⋮

Step n : Discard all multiples of $\sqrt{n}$

last step : Remaining numbers are prime numbers.

Algorithm Sieve: Eratosthenes(n)

//Input : an integer $n>2$

//output : Array of all prime numbers $\leq n$

//Description : Generates prime numbers from 1 to n

```
for p ← 2 to n do A[p] → p

for p ← 2 to √n
    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
        L[i] ← A[p]
        i ← i+1

return L
```

3)

* *

Asymptotic Notations:

To rank and compare the order of growth of a function, we use asymptotic notation.

There are 3 types

* Big oh Notation (O)
* Big omega Notation (Ω)
* Theta notation (Θ)

Let $t(n)$ and $g(n)$ be two non negative functions which denotes the actual time taken by the algorithm and $g(n)$ is sample function to be compared

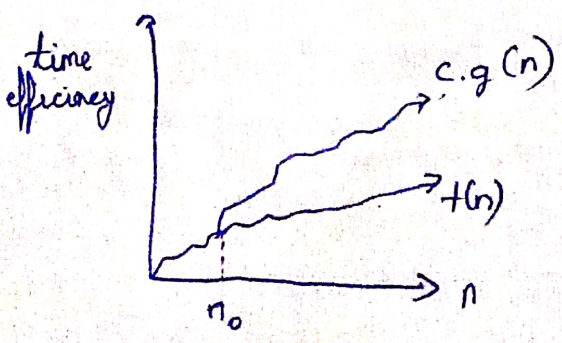
$t(n)$ → actual time

$g(n)$ → sample function

Big oh Notation (O)

The function $t(n)$ is said to be Big oh $g(n)$ d $[O(g(n))]$ denoted by $t(n) \in O(g(n))$ if $t(n)$ is bounded above some positive constant multiple of $g(n)$ for all larger values of n, if there exist a positive integer constant C, and positive integer n_0 satisfying this statement.

$$t(n) \leq c.g(n) \quad \forall \quad n \geq n_0, c \geq 0, n_0 \geq 1$$



Exc:-

$n^2 \in O(n^3)$

$\log(n) \in O(n)$

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

$t(n) = 3n+2$

$g(n) = n$

$3n+2 \leq c \cdot n$

for $c \geq 4$

$3n+2 \leq 4n$

for $n = 2$ ↑ satisfies

$n_0 = 2$

Big oh ~~refo~~ represents the upper bound.

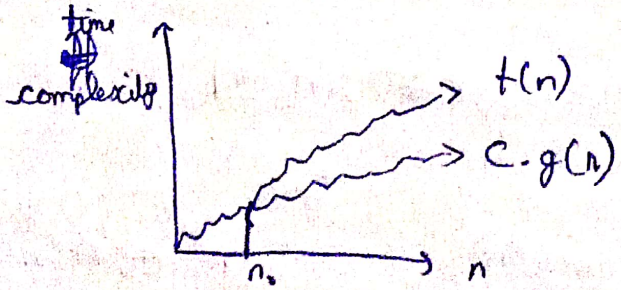
---

2) Big Omega Notation (Ω)

The function $t(n)$ is said to be omega of $g(n)$ [$\Omega(g(n))$] denoted by $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below some positive constants of multiple of $g(n)$ for all larger value of n there exist a positive integer constant c, and a non-ve integer n_0 satisfying

$t(n) \geq c \cdot g(n) \quad \forall \quad n \geq n_0, \; c > 0, \; n_0 \geq 1$

Ex: $n! \in \Omega(n)$



Ex:

$f(n) = 3n+2$ $\quad 3n+2 \in \Omega(n)$

$g(n) = n$

$3n+2 \geq c.n$

for $c=1$ $\quad 3n+2 \geq 1n$ — check until condition fails. Take last value

$c=2$ $\quad 3n+2 \geq 2n$

$c=3$ $\quad 3n+2 \geq 3n$

$c=4$ $\quad 3n+2 \geq 4n \rightarrow$ fails

$\boxed{c=3}$

$3n+2 \geq 3n$

$n=1, \quad 5 \geq 3$

$\boxed{n_0 = 1}$

Ω represents the lower bound

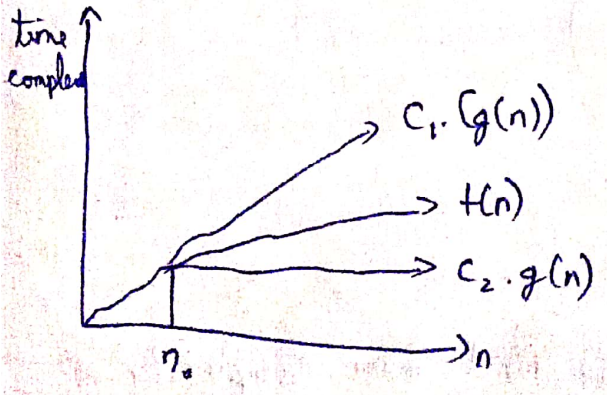
---

*

3) Theta Notation (Θ)

The function $f(n)$ is said to be $\Theta(g(n))$ denoted by $f(n) \in \Theta(g(n))$, if $f(n)$ is bounded both above and below by some positive constant multiple of $g(n)$ for all larger value of n, there exist a positive integers c_1 and c_2 and a non-negative integer n satisfying.

$$c_2 g(n) \leq f(n) \leq c_1 g(n), \forall \quad n \geq n_0, c > 0, n_0 \geq 1$$



Ex:

$f(n) = 3n+2$

$g(n) = n$

$3n+2 \in \Theta(n)$

$c_1 = 4 \quad n_0 = 2$

$c_2 = 3 \quad n_0 = 1$

$\Big\} n_0 = 2$

L'Hopital Rule:

To compare two efficiency class we need to find the value based on following equation

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & ; \ O \text{ notation} \quad t(n) \leq \\ \text{constant} & ; \ \Theta \text{ notation} \quad t(n) \text{ is same} \\ \infty & ; \ \Omega \text{ notation} \quad t(n) \geq \end{cases}$$

Example (1):

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

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

$$= \frac{1}{2}\left(1 - \frac{1}{\infty}\right)$$

$$= \frac{1}{2}(1-0)$$

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

2) $n!$ and 2^n

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

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

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

$$= \infty$$

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

General plan or procedure to find time complexity of non-recursive algorithm

Step 1: Decide on the parameter / parameters indicating the input size.

Step 2: Identify the algorithms basic step or the basic operation (located in the inner most loop)

Step 3: Check whether the number of time the basic operation execution depends only on the size of the input. If it depends on additional property then the worst, best and average case needs to be investigated

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

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

Example 1: Algorithm to find the max element in an array

```
Algorithm Maxelement (A[0.....n-1])
// Finds the max element in an array
// Input : An array A of n elements
// Output : Max value stored in array.
    maxvalue ← A[0]
    for i ← 1 to n-1
        if A[i] > maxvalue do
            maxvalue ← A[i]
    return maxvalue
```

Time complexity :

Input size = n

Basic operation = Comparision

Let C(n) denotes number of times the basic operation gets exceeded which can be expressed as C(n)

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

$$= O(n)$$

Example 2 : Algorithm to find uniqueness of an element in an array

Algorithm ElementUbiqueness (A[0, 1, 2 ..... n-1])

//Description : Returns True if all elements are unique

// Input : Array a of n elements.

//Output : Returns True if elements are unique.

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

Time complexity :

Input size = n

Basic operation = Comparision

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

In worst case we do maximum number of comparision which can be denoted by C(n)

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

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

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

$$= n-1 + n-2 + \ldots\ldots + 1$$

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

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

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

General plan to find time efficiency of recursive algorithm

~~Algorithm recursive~~

Step 1 : Decide on parameter indicating input size

Step 2 : Identify algorithm basic operation.

Step 3 : Check whether number of times of basic operation execution depends only on size of input

Step 4 : Setup a recurrence relation with appropriate base case for the number of time the basic operation gets executed.

Step 5 : Solve recurrence with appropriate methods to find the time complexity.

- Substitution
- Master theorem
- Recursive tree

Example 1 : Algo to find factorial of a number.

Algorithm fact(n)
// Description : Function computes factorial of n
// Input : — number greater than zero
// Output : Returns factorial

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

Time complexity:

Basic operation = Multiplication.

Let M(n) denotes the number of times the basic operation gets exceeded.

For n=0;

No multiplication is done ;i.e, M(0) = 0
↑
base case

For n ≥ 1

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

$= M(0) + n$

$M(n) = n$

~~$= O(n)$~~

T.C = n

Example : Towers of hanoi

Algorithm ToH(n, s, d, t)
// Solves tower of hanoi problem
// input is number of disks.
// Output disks from source to destination.

if n=1 do
    Move nth disc from source to destination.
else
    ToH(n-1, s, t, d)

Mov nth disc from source to destination

Toh (n-1, t, d, s)

Time complexity :

Basic operation = Movement of disc.

Let m(n) denote the number of ~~ter~~ movement done while moving n discs.

For n = 1 ;

$m(1) = 1$

For $n > 1$

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

$= 2\, m(n-1) + 1$

$= 2\,[2m(n-2) + 1] + 1$

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

$= 2^3 m(n-3) + 2 \times 2 + 1$

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

$= 2^k m(n-k) + 2^{k-1} + \ldots\ldots 2^1 + 2^0$

$= 2^{n-1} m(1) + 2^{n-2} + \ldots\ldots + 2^0$

$= 2^{n-1} + 2^{n-1} - 1$

$m(n) = 2^n - 1$

T.C $= 2^n - 1$

$O(2^n)$

Algorithm to count the number of bits in a binary representation

Algorithm BitCount(n)
// To count number of binary digits for a given number
// Input: an integer n
// Output: Bitlength
If n=1 return 1
return BitCount(⌊n/2⌋) + 1

Time complexity:

Basic operation is addition:

Let A(n) denotes number of addition.

If n = 1, A(1) = 0

for n > 1

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

$A(n) = A(n/4) + 2$

$= A\left(\frac{n}{8}\right) + 3$

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

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

$k = \log_2 n$

$A(n) = A(1) + \log_2 n$

$A(n) = \log_2 n$

$O(\log_2 n)$

Consider the following algorithm f(n)

```
Algorithm f(n)
// Input : An integer n
count ← 1
while n>1 do
    count ← count + 1
    n ← n/2
return count.
```

1) What is the objective of algorithm
* It counts the number of binary digits of an number N
2) Identify the basic operation.
~~Division~~ Comparision.
3) Efficiency class of the algorithm
logarithmic class $\log_2 n$

---

master - theorem

It can be applied to any recurrence of the type $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1, b \geq 1$ and $f(n)$ is asymptotically positive

Case 1:

$T(n) = \Theta\left(n^{\log_b a}\right)$; if $n^{\log_b a}$ is greater than $f(n)$

Case 2:

$T(n) = \Theta\left(f(n)\right)$; if $n^{\log_b a}$ is lesser than $f(n)$

Case 3:

$T(n) = \Theta\left(n^{\log_b a}(\log n)^{k+1}\right)$; if $n^{\log_b a} = f(n)$, $k \geq 0$

k is power of $\log n$ in $f(n)$

Apply master theorem for following recurrences.

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

$a = 4$

$b = 2$

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

Case 1 : $T(n) = \Theta(n^2)$

2) $T(n) = 9T\left(\frac{n}{3}\right) + n^2$

$a = 9 \quad b = 3$

$n^{\log_3 9} = n^2$

Case 3 : $T(n)\ (n^2 (\log n))$

$k = 0$

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

$n^{\log_2 2} = n$

$f(n) = n$

Case 3, $k = 0$: $T(n) = \Theta(n \log n)$

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

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

Case 2 : $\Theta(n^3)$

If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then $t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n)))$

By definition of big oh (O)

$t_1(n) \leq c_1 g(n) \longrightarrow ①$ $\forall n \geq n_1$

III ly

$t_2(n) \leq c_2 g_2(n) \longrightarrow ②$ $\forall n \geq n_2$

Let $c_3 = \max(c_1, c_2)$ and consider $n \geq \max(n_1, n_2)$

then $t_1(n) + t_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$

$t_1(n) + t_2(n) \leq c_3 g_1(n) + c_3 g_2(n)$

$t_1(n) + t_2(n) \leq c_3 (g_1(n) + g_2(n))$

$t_1(n) + t_2(n) \leq c_3 2 \max(g_1(n) + g_2(n))$

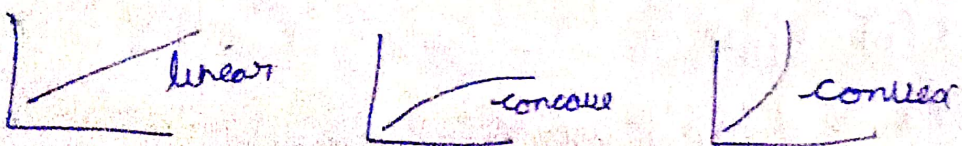
~~$t_1(n) + t_2(n)$~~

$\leftarrow O \max(g_1(n), g_2(n))$

Emperical analyses.

General plan for performing emperical analysis

* Under the experiment ~~purp~~ purpose.
* Decide on the efficiency matrix (M) to be measured (basic operation)
* Decide on characteristics of input sample size.
* Prepare a program implementing the algorithm used in experimentation
* Generate a sample of inputs
* Run the program on sample inputs and record the data observed.
* Analyse the obtained data. To analyse the output a graph needs to be plotted where x axis denoted input count and y axis denoted output count M



Proof:

By definition of big oh (O)

$t_1(n) \leq c_1 g_1(n) \longrightarrow ① \quad \forall n \geq n_1$

III ly

$t_2(n) \leq c_2 g_2(n) \longrightarrow ② \quad \forall n \geq n_2$

Let $c_3 = \max(c_1, c_2)$ and consider $n \geq \max(n_1, n_2)$

then $t_1(n) + t_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$

$t_1(n) + t_2(n) \leq c_3 g_1(n) + c_3 g_2(n)$

$t_1(n) + t_2(n) \leq c_3 (g_1(n) + g_2(n))$

$t_1(n) + t_2(n) \leq c_3 2 \max(g_1(n) + g_2(n))$

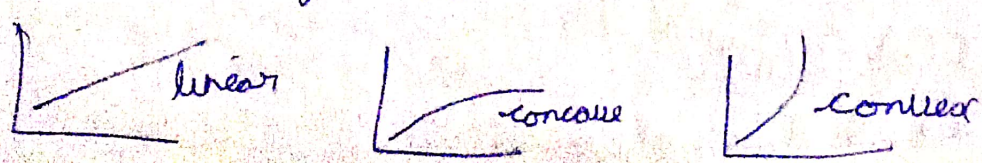
~~$t_1(n) + t_2(n)$~~

$\in O \max(g_1(n), g_2(n))$

Emperical analysis.

General plan for performing emperical analysis.

* Under the experiment ~~purp~~ purpose.
* Decide on the efficiency matrix (m) to be measured (basic operation)
* Decide on characteristics of input sample size.
* Prepare a program implementing the algorithm used in esperimentation
* Generate a sample of inputs
* Run the program on sample inputs and record the data observed.
* Analyse the obtained data. To analyse the output a graph needs to be plotted where x axis denoted input count and y axis denoted output count m



Brute Force

In this method, we try all possibilities to get satisfactory solutions to a problem.

Bubble Sort :

Algorithm Bubblesort (A[0, ..., n-1])
// Sorts the elements of an array A using bubble sort.
// Input : Array of n integers
// Output : Sorted set of elements in ascending.

```
For i ← 0 to n-2
    for j ← 0 to n-i-2
        if A[j] > A[j+1]
            Swap A[j] and A[j+1]
```

Efficiency :

The basic operation is comparision. let C(n) denotes total number of comparisions done

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

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

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

$$= (n-1) + (n-2) + (n-3) + \ldots\ldots 1$$

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

$$= \frac{n^2-n}{2}$$

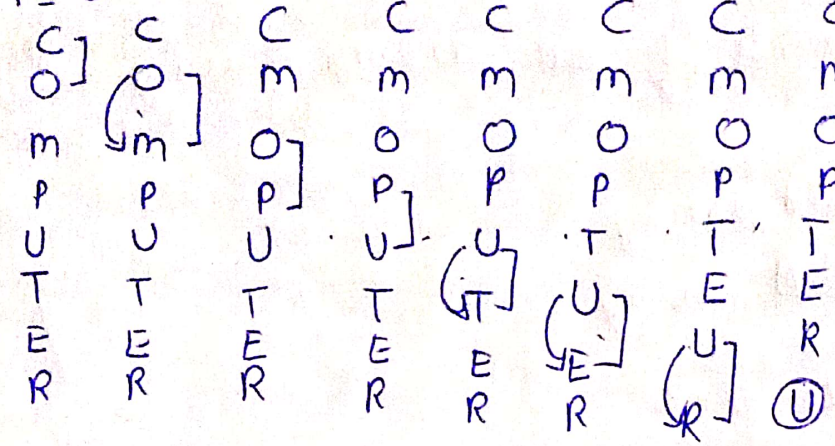
$$C(n) = \Theta(n^2)$$

Stable : It is a stable.

Inplace → doesn't uses extra memory

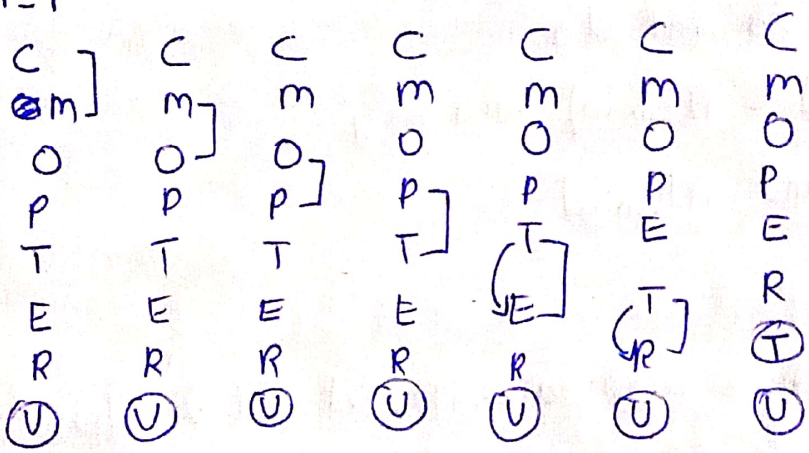
Find the number of swaps to sort a word named COMPUTER in ascending order using bubble sort.

i = 0



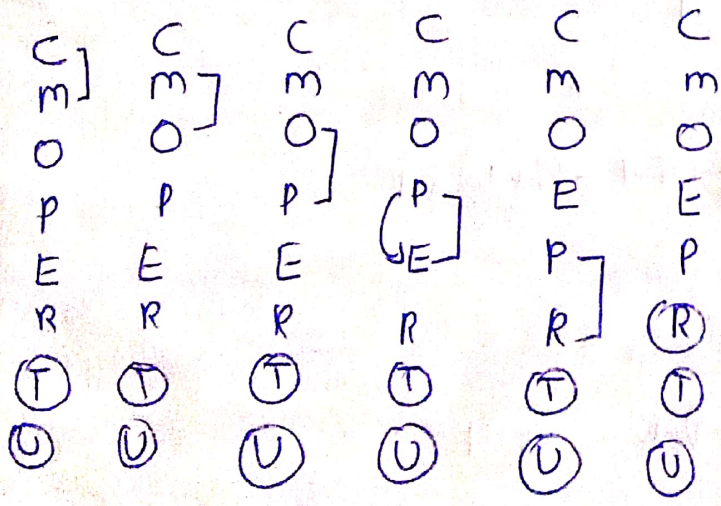
Comparision = 7
Swap = 4

i = 1



Comparison = 6
Swap = 2

i = 2



Comparision = 5
Swap = 1

Selection Sort

Algorithm selectionSort(A[0,....,n-1])

// sorts the elements of an array A using selection sort

// Input : Unsorted array

// Output : Sorted array

```
for i ← 0 to n-2
    min ← i
    for j ← i+1 to n-1 do
        if A[j] < A[min]  min ← j
    Swap A[i] and A[min]
```

Efficiency :

Basic operation is comparision, let C(n) denotes the total number of comparisions.

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

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

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

$$= (n-1)+(n-2)+\ldots+1$$

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

$$= \frac{n^2-n}{2}$$

$$= O(n^2)$$

Sort the elements using selection sort: EXAMPLE

AXEMPLE

AEXMPLE

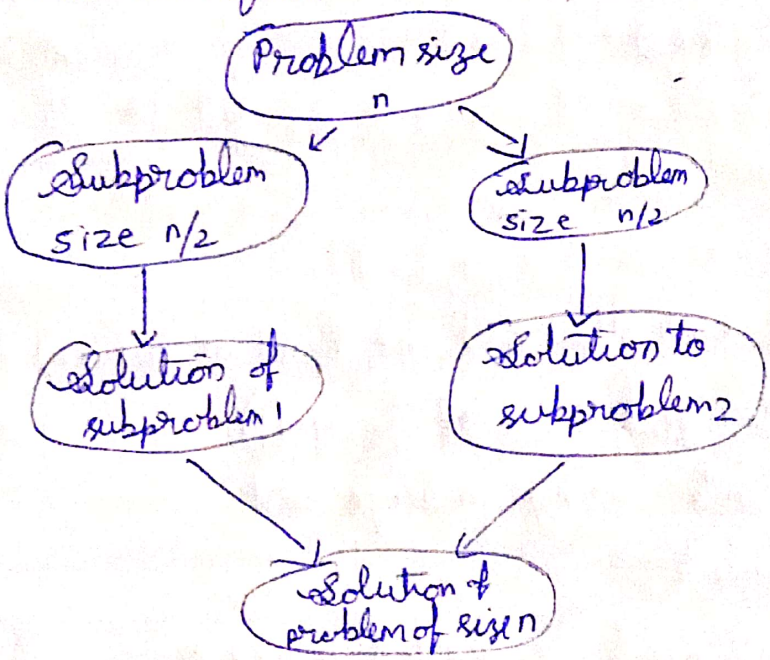
AEEMPLX

AEELPMX 6 swap

AEELMPX

AEELMPX

Divide and conquer



1) Divide the problem of size n into equal subproblems
2) Recursively solve the subproblems.
3) Combine solutions of subproblems to get the solution for the problem

23 14 10 5 35 70 4 16

(23 14 10 5)

(23 14) (10, 5)

(23) (14) (10) (5)

(14, 23) (5, 10)

(5 10 14 23)

(35 70 4 16)

(35 70) (4, 16)

(35) (70) (4) (16)

(35, 70) (4, 16)

(4 16 35 70)

Algorithm mergesort (A[0.... n-1])
// Sorts elements using merge sort
// set of elements
// A set of elements in sorted manner.
if l<r
    m ← (l+r)/2
    Mergesert (A [l.... m])
    Mergesert (A [m+1.... r])
    Merge (A, l, m, r)

Algorithm Merge (A [l.... m], A [m+1.... r])
// Merges two sorted arrays
// Input: two sorted array
// Output: to a single sorted array

```
i ← l
j ← m+1
k ← l
while (i≤mid && j≤r)
    if A[i] < A[j]
        B[k++] ← A[i++]
    else
        B[k++] ← A[j++]
while i<=m
    B[k++] ← A[i++]
while j<=r
    B[k++] ← A[j++]
~~while~~
Copy elements of B to A
```

Efficiency:

Basic operation is comparision. Let C(n) denotes the total number of comparisions done to sort n elements which can be represented as

$$C(n) = 2C\left(\frac{n}{2}\right) + C_{merge}(n)$$

In worst case to merge two sorted arrays, we do max of (n-1) comparisions

$$C(n) = 2C\left(\frac{n}{2}\right) + (n-1)$$

By applying master theorem $a=2, b=2$

$$n^{\log_b a} = n^{\log_2 2} = 1$$

Case 3 of master theorem

$$f(n) = n^{\log_a b}$$

Hence $C(n) = \theta(n \cdot \log n)$

The actual number of comparision in mergesort : $(n \log n - n + 1)$

Merge sort is not inplace but it is stable.

Program 1:

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

void merge (int A[SIZE], int l, int m, int r) {
int i,j,k, B[SIZE];
i = l;
j = m+1;
k = l;

while (i<=m && j<=r) {
      if (A[i] < A[j]) {
          B[k++] = A[i++];
          count++;
        }
      else {
          B[k++] = A[j++];
          count++;
        }
}
```

```
while (i<=m)
        B[k++] = A[i++]
while (j<=n)
        B[k++] = A[j++]

~~for(int m=0;~~
 for(i=l; i<k; i++)
        A[i--] = B[i--]
}
void mergesort(int A[SIZE], int l, int r) {
int m;
   if (l<r) {
        m = (l+r)/2
        mergesort(a, l, m);
        mergesort(a, m+1, r);
        merge(a, l, m, r);
      }
}

int main() {
  int A[SIZE], X[SIZE], Y[SIZE], Z[SIZE];
  int n, i, c1, c2, c3, j;
  printf("Enter the size of array\n");
  scanf("%d", &n);
  printf("Read elements \n");
  for(i=0; i<n; i++)
      scanf("%d", A[i]);
  mergesort(a, 0, n-1);
  printf("The sorted elements are\n");
```

```
for(i=0;i<n;i++)
    printf("%d\t",A[i]);
printf("\n The basic operation executes %d times\n",
    count);

printf("\n SIZE \t ASC \t DESC \t \t RANDOM\n");
for (i=16; i<SIZE; i=i*2) {
    for (j=0; j<i; j++) {
        X[j] = j;
        Y[j] = i-j-1;
        Z[j] = rand() % i }
        count=0;
        Mergesort(X,0,i-1);
        C1 = count;
        count=0;
        mergesort(Y,0,i-1);
        C2 = count;
        count=0;
        mergesort(Z,0,i-1);
        C3 = count;
        printf("%d\t %d\t %d\t %d\n",i,C1,C2,C3);
    }
}
return 0;
}
```

Find the position of max element in an array using divide and conquer

***

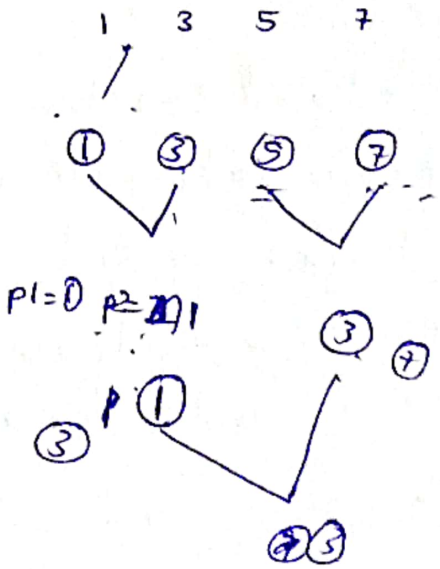
Algorithm Maxpos(A[l.....r])
//Description : Algorithm to find maximum element position using DAC

//Input : Array of element
//output: Index of output element.

```
if( l==r  return l;
else
    p1 ← Maxpos(A[l....(l+r)/2])
    p2 ← Maxpos(A[(l+r)/2 + 1 ..... r])

    if A[p1] > A[p2]
        return p1
    else
        return p2
```

Time complexity :-

$$T(n) = 2T\left(\frac{n}{2}\right) + C$$



To count the number of leaf nodes in a BT using DAC

Algorithm Countleaf(root)

//Description : Returns number of leafnodes

// Input : Address of root node

// Output : Number of leafnodes

```
if root = ϕ return 0
if root_left = ϕ and root_right = ϕ return 1
else
    return Countleaf(root_left) + Countleaf(root_right)
```

Efficiency :

QUICK SORT

It is based on the concept of partition, in partition, we take either first element or last element as pivot and the end of the partition the pivot element will be placed such that all element all element to be left of pivot is less than or equal and all elements towards right of pivot are greather number.

```
pivot ≥ A[i]    i++
pivot < A[j]    j++
if (i<j)
    Swap a(i) and a(j)
else
    swap (a[i]) and pivot return j
```

[10] 12 7 2 24 15 8

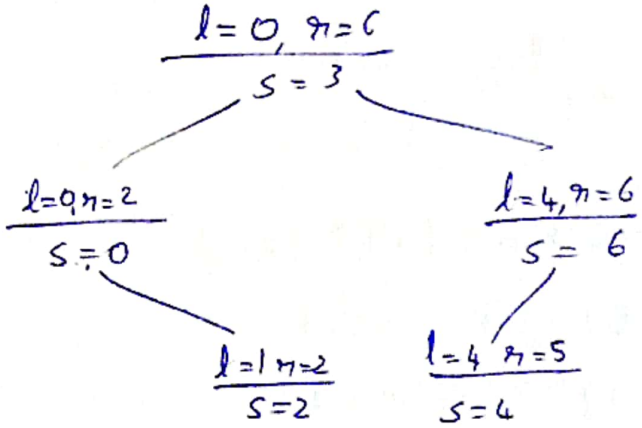
pivot

[10] 8 7 2 24 15 12

(2 8 7) [10] (24 15 12)

[2] (8 7) [10] (12 15) [24]

[2] → [7] → [8] → [10] → [12] → [15] → [24]



Trace quicksort for the input 5, 3, 1, 9, 8, 2, 4, 7 & draw recursive tree

[5] 3 1 9 8 2 4 7

[5] 3 1 4 8 2 9 7

[5] 3 1 4 2 8 9 7

2 3 1 4 [5] [8] 9 7

p p

2 1 3 4

i j i++

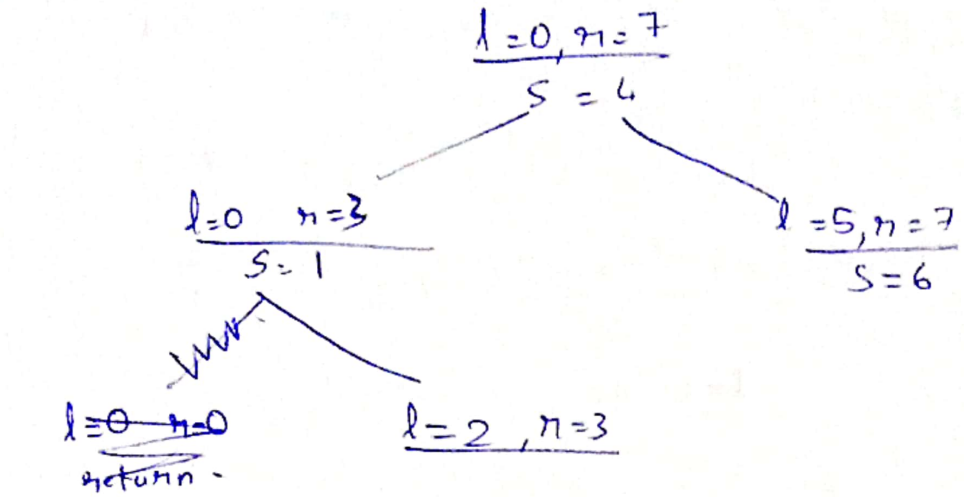
[1] [2] [3] 4

r

[1] [2] [3] [4]

9 [8] 9

[7] [8] [9]



Algorithm Quicksort (A[l.... r])
// Description: Returns sorted array
// Input: Array of elements
// Output = sorted array

if $l < r$
    $s \leftarrow$ partition (A, l, r)
    Quicksort (A[l....s-1])
    Quicksort (A[s+1.....r])

Algorithm partition
// Description: Partition a subarray from l to r
// Input. Subarray A
// Output: Places the pivot element into actual position in sorted array

```
pivot ← A[l]
i ← l+1
j ← r
while{TRUE do
    ~~pivot ≥ A~~ while pivot ≥ A[i] && i <= r
                i++
          while pivot ≤ A[j] && j != l
                decrement j
          if i<j
              swap A[i] and A[j]
          else
              swap A[j] and A[l]
          return j
```

Efficiency

Basic Operation is comparision

Best case: On each call of partition, if problem is divided into 2 equal parts then it is best case.

$$C_{best}(n) = 2C\left(\frac{n}{2}\right) + n+1$$

Applying master theorem

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

Worst case: If input elements are in ascending or descending then quick sort exhibits worst case

$$C_{worst}(n) = C(n-1) + n+1$$

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

$$= \frac{n^2+3n-4}{6} \Rightarrow \Theta = n^2$$

Average case: In average case position of partition is between l and r

$$C_{avg}(n) = 1.38 n \log n$$

Strassen's matrix multiplication.

To multiply 2 matrix of order n: The brute force method has a time complexity of n^3. Using divide and conquer we reduce complexity to $n^{2.807}$.

Let A and B be two matrix denoted by

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

The resultant matrix

$$C = A \times B = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$$ where ~~C_1, C_2, C_3, C_4~~

$C_1 = E + I + J - G_1$

$C_2 = D + G_1$

$C_3 = E + F$

$C_4 = D + H + J - F$

$D = A_1 (B_2 - B_4)$

$E = A_4 (B_3 - B_1)$

$F = (A_3 + A_4) B_1$

$G_1 = (A_1 + A_2) B_4$

$H = (A_2 - A_1)(B_1 + B_2)$

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

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

$T(n) + 7T\left(\frac{n}{2}\right) + n$ → Apply Master theorem

~~$D = A_1 (B_2 - B_4)$~~

$a = 7$

$b = 2$

$n^{\log_2 7} = n^{2.807}$

Apply Strassen's matrix multiplication for matrix

$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ $B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$

$A_1 = 1$ $A_2 = 2$ $A_3 = 3$ $A_4 = 4$

$B_1 = 5$ $B_2 = 6$ $B_3 = 7$ $B_4 = 8$

$\left(\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right)$

$D = 1\,(6-8) = -2$

$E = 4\,(7-5) = 8$

$F = (3+4)\,5 = 35$

$G = (1+2)\,8 = 24$

$H = (3-1)\,(5+6) = 22$

$I = (2-4)\,(7+8) = -30$

$J = (1+4)\,(5+8) = 65$

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

$C_2 = -2 + 24 = 22$

$C_3 = 8 + 35 = 43$

$C_4 = -2 + 22 + 65 - 35 = 50$

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

Efficiency

The following recurrence can be used to denote total number of multiplication done

$T(n) = 7T(\frac{n}{2}) + n$ on applying M.T

$T(n) = O(n^{2.8073})$

Multiplication of two large integers:

Using Brute force method to multiply two numbers of size 'n' time taken is n^2. Using divide and conquer the time complexity is reduced to $n^{1.5849}$. Let A & B be two n-number whose product is $C = a \times b$

$$a = a_1 a_0 \qquad b = b_1 b_0$$

Product c is calculated as.

1. $C_0 = a_0 \times b_0$
2. $C_2 = a_1 \times b_1$
3. $C_1 = (a_0 + a_1)(b_0 + b_1) - (C_2 + C_0)$
4. $C = C_2 10^n + C_1 10^{n/2} + C_0$

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

$$n^{\log_2 3}$$

$$n^{1.5849}$$

Let A be 72 and B = 43

$a_1 = 7 \quad a_0 = 2 \quad b_1 = 4 \quad b_0 = 3$

$C_0 = 2 \times 3 = 6$

$C_2 = 7 \times 4 = 28$

$C_1 = (7+2)(4+3) - (28+6)$

$= 9(7) - 34$

$= 63 - 34$

$= 29$

$C = 28(10)^2 + 29(10)^{4/2} + 6$

$= 2800 + 290 + 6$

$= 3096$

Time complexity :

The following recurrence can be used to denote total number of multiplication done

$T(n) = 3T\left(\frac{n}{2}\right) + n$. Applying master theorem

$n^{\log_2 3}$

$n^{1.5849}$

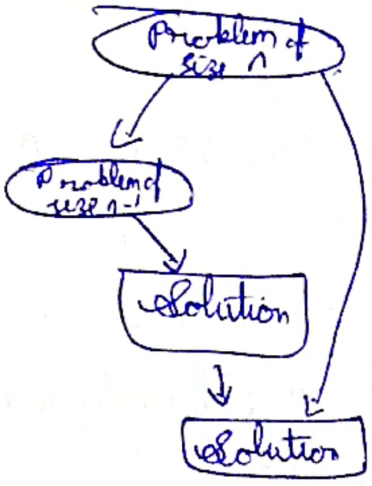
n

Decrease and conquer

The general approach for decrease and conque is

* Reduce the ~~smaller~~ problem instance into smaller instance of the same problem and extend solution of smaller instance to obtained the solution for the original problem.

* Reduce to small inst
* Solve the smaller inst
* Extend.



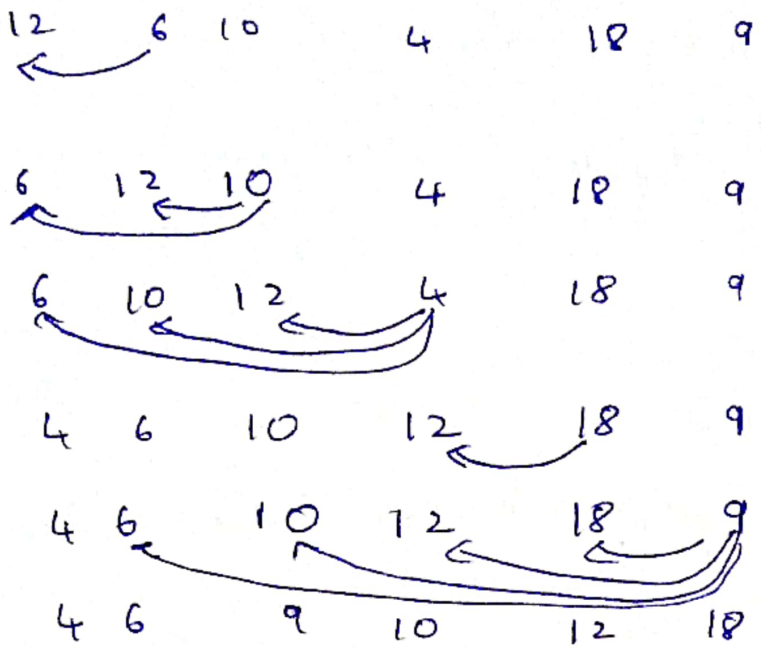
Types.

i) Decrease by one
ii) Decrease by constant factor
iii) Variable size decrease.

Decrease by one.

I.

Insertion sort



Best case :- If elements are already sorted we just do 1 comparision in each iteration

Worst case : If elements are there in reverse order it exhibits worst case

Algorithm Insertion (A[0....n-1])
// Input : Array
// Output : Sorted array
// Description : function returns 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
```

Efficiency: Basic operation is comparision.

Best case = $\sum_{i=1}^{n-1} 1$

$= n-1-1+1$

$= (n-1)$

Best case $= O(n)$

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

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

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

$= 1 + 2 + \ldots\ldots n-1$

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

$= \theta(n^2)$

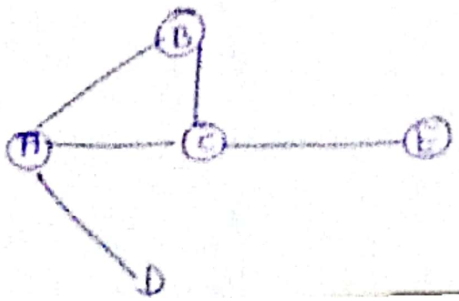
Average case : $\Theta(n^2)$

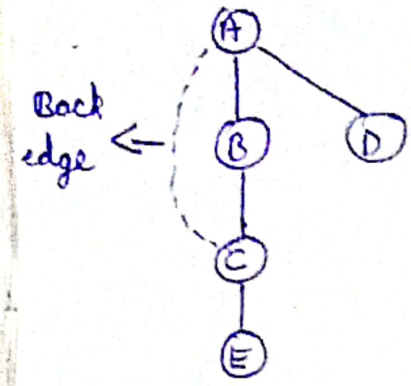
Graph traversal:

i) DFS → Depth first search.

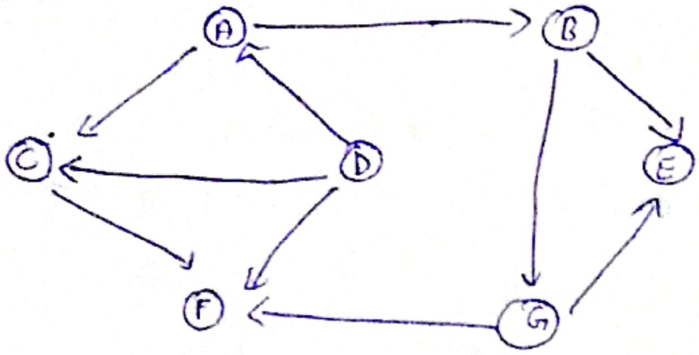
ii) BFS → Breath first search.

DFS: Perform DFS traversal on following graph

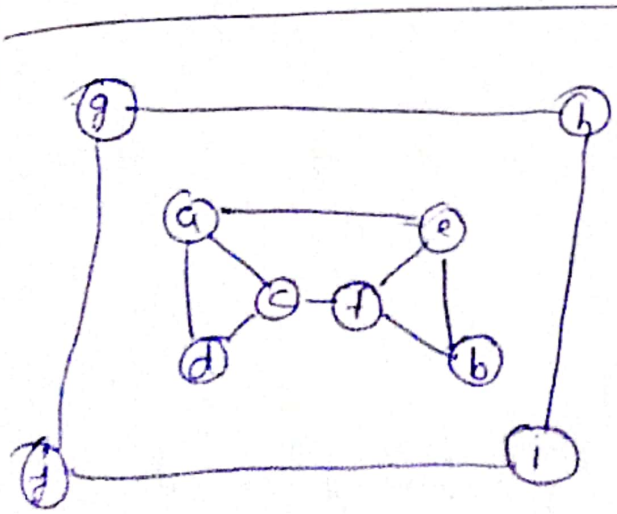




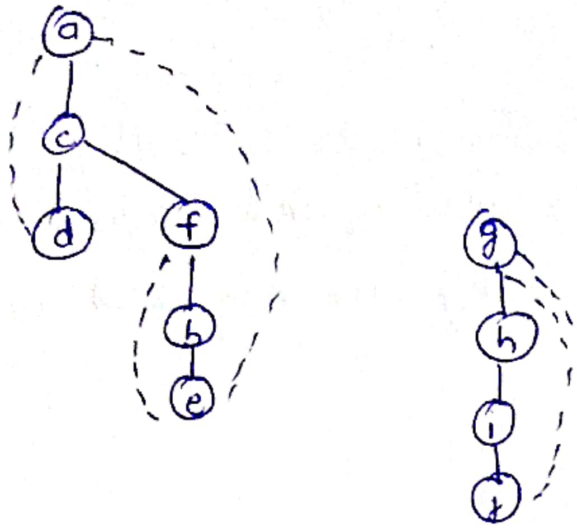
$A_{1,1}$
$B_{2,3}$
$C_{3,2}$
$E_{4,1}$ ← Dead end
$D_{5,4}$



$A_{1,6}$
$B_{2,4}$
$E_{3,1}$
$G_{4,3}$
$F_{5,2}$
$C_{6,5}$
$D_{7,7}$



$a_{1,6}$
$c_{2,5}$
$d_{3,1}$
$f_{4,4}$
$b_{5,3}$
$e_{6,2}$
$g_{7,10}$
$h_{8,9}$



$i_{a,8}$

$d_{10,7}$

Algorithm DFS(G)

// Input: Graph G.

// Output: Graph G with its vertices marked with consecutive integers in the order they were visited.

// Description :- Implements depth first search on a graph G $=\{V, E\}$

Mark each vertex v in V with 0 as a mark of being unvisited.

count ← 0

for each vertex v in V

if v is marked with zero

dfs(v)

Algorithm dfs(v)

// Description : visits recursively all the unvisited vertices connected to small v and assigns them an integer number in the order they are encountered via variable count

// Input: Vertex

// Output:

count ← count+1

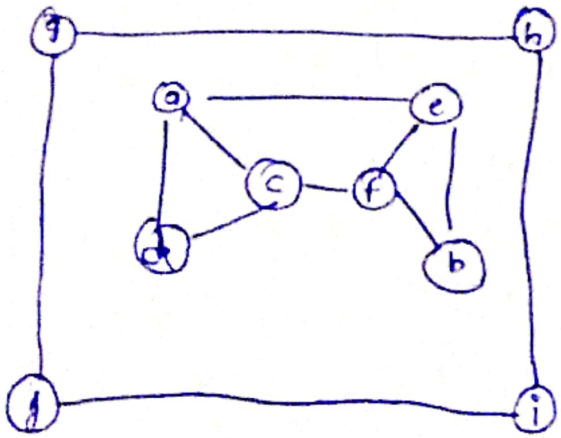
mark v with count

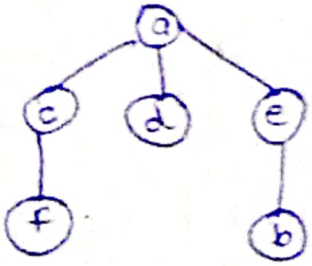
for each vertex w adjacent to ~~small~~ v

if w is marked with zero

dfs(w)

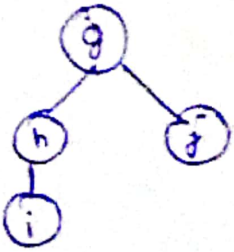
~~Algorithm bfs(v)~~ BFS (Breath first search)

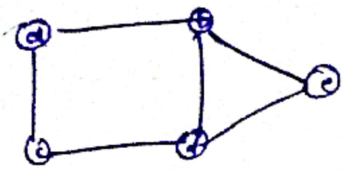


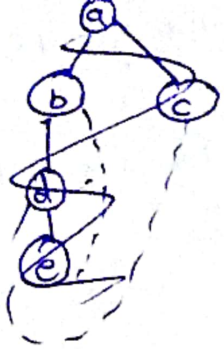


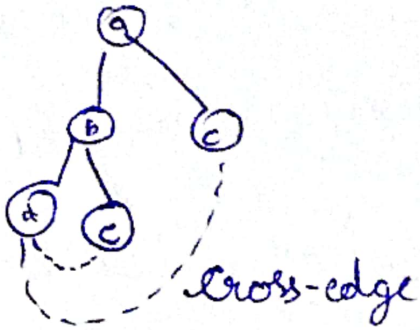
a c d e f b

g h j i









Algorithm BFS (G)
// Input : Graph G
// Output: Output in which graph is visited.

Mark each vertex v in V with zero as mark of being unvisited

```
count ← 0
for each vertex v in V
        if v is marked with zero
            bfs(v)
```

Algorithm bfs(v)
// Description : Visits all the unvisited vertices connected to v and assigns them an integer number in order they were visited by global variable count
// Input : Vertex
// Output:

```
count ← count+1
mark v with count and initialize a queue with v
while ~~q is not~~ queue is not empty do.
        for each vertex w adjacent to v
            if w is marked with 0
                count ← count+1
```

add w to the queue

remove a vertex from the queue.

Difference between

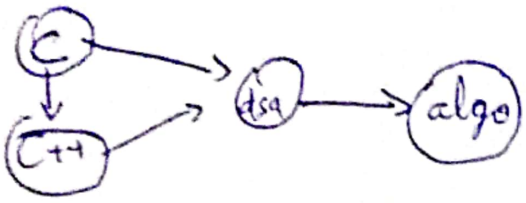
	DFS	BFS
D.S	Stack	Queue.
Ordering	Two	one
Edges Types	Tree edge & back edge	Tree edge & Cross edge.
Application	Graph connectivity Articulation point	Graph connectivity minimum edge path.
Time complexity	$\|V\|^2$	$\|V\|^2$
	$\|V\| + \|E\|$ for adjacency.	

Articulation point → A vertex of a connected graph is said to be it's articulation point if its removal ~~all edges incident to it~~ breaks ~~all edges~~ graph into disjoint subgraphs.

Topological ordering

Topological order of a directed acyclic graph (DAG) is the the linear ordering of vertices if there is an edge from $U \rightarrow V$, U comes before v in the order.

Ex:

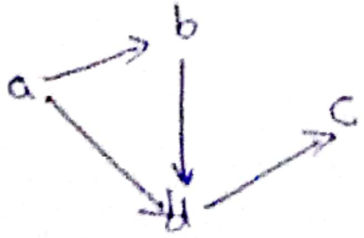


C, C++, dsa, algo

Topological ordering can be achieved in two different ways, using DFS or, Vertex deletion method (source removal method).

DFS method

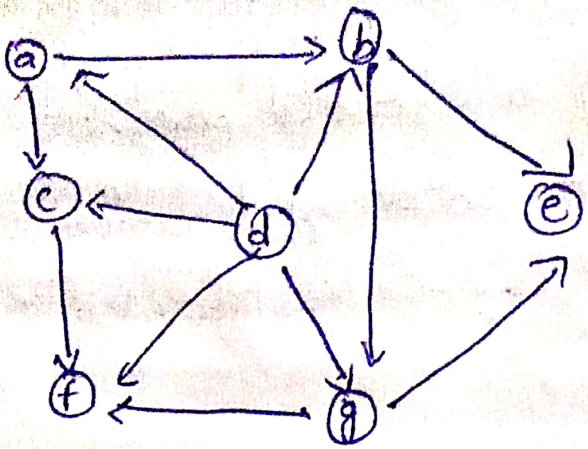
* Traverse the given graph using DFS
* Note down the order in which nodes became deadend
* Reverse the above order.
* We will get Topological order

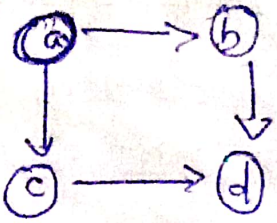


a, 4
b, 2, 3
d 3, 2
c 4, 1

c, d, b, a

a, b, d, c





	a	b	c	d
a	0	1	1	0

$a_{1,6}$

$b_{2,4}$

$e_{3,1}$

$g_{4,3}$

$f_{5,2}$

$c_{6,5}$

~~$f_{7,8}$~~

$d_{7,7}$

e, f, g, b, c, a, d

b

c

d

d, a, c, b, g, f, e

Exp2: Topo

```
int topo[10], k
int main() {

int n, a[10][10], i, v[10], j
for(int i=0 printf("Enter value of n\n");
scanf("%d", &n);
printf("Enter adjacency matrix\n");
for (i=1 ; i<=n; i++)
        for (j =1 ; j<=n; j++)
            scanf("%d", &a[i][j]);
```

```
for(i=1; i<=n; i++)
    v[i] = 0;

for (i=0; i<=n ; i++) {
        for(j=1; j<=n; j++)
    if(v[i] ==0)
        dfs =(a,n, v,i);
    }
printf("\n The topological ordering is \n");
for(i = k; i >=1; i--)
        printf("%d\t", topo[i]);

return 0;
}

void dfs (int a[10][10], int n, int v[10], int source){
  int i;
  v[source] = 1;
  for (i=1; i<=n; i++)
     {  if(v[i]==0 && a[source][i] ==1)
              dfs (a, n, v, i);
     }
  top[++k] = source;
```

Part-B (To check graph connectivity using BFS)

```
int main() {
    int n, a[10][10], i, j, v[10]; count=0;
    printf("\nEnter the no of nodes\n");
    scanf("%d", &n);
    printf("\n Enter the elements\n");
    for (i=1; i<=n; i++)
        v[i] =0;
    for (i=1; i<=n; i++) {
        if (v[i]==0) {
            bfs(a,n,v,i);
            count++; }
    }
    if (count==1)
        printf("\n Graph is connected");
    else
        printf("\n Graph is not connected and has %d
    components," count);
    return 0;
}
void bfs(int a[10][10], int n, int v[10], int source) {
    int j; q[10], f=0, r=-1, node
    v[source]=1;
    for(i=1; i<=n; i++)
    {
        if(v[i]==0 && a[source][i]==1)
            bfs(a,n,v,i);
    }
```

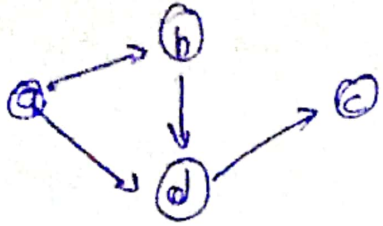
~~topo[++k] = source;~~
~~}~~

```
q[++r] = source;
while (f <= r) {
    node = q[f++]; printf("%d\n", node);
    for(i=1; i<=n; i++){
        if (v[i]==0 && a[node][i]==1) {
            q[++r] = i;
            v[i] = 1;
        }
    }
}
```

Topological ordering using source removal method;

Procedure to be used is

* Note down indegree of all the nodes of the graph
* Place all those nodes which have an indegree of 0
* Decrement the indegree of those nodes where there is an incoming edge from the nodes which are placed in step 2
* Repeat step 2 & 3 until all nodes have been placed.



Generating permutation :-

Johnsben Trottes algorithm :

According to this algorithm it is possible to get the same order of permutation for n elements with explicity generating permutation for the smaller values of N.

The idea here is to associate a direction with each of the element as

$\overrightarrow{1}$ $\overrightarrow{2}$ $\overleftarrow{3}$

3 is termed as mobile digit or integer

Algorithm Johnson Trottes (n)

Input :// value of n

Output :// Permutation list of the set (1.....n)

Description :// Implements Johnson Trottes for generating permutation.

Initialize the first permutation as $\overleftarrow{1}, \overleftarrow{2}, \overleftarrow{3} \ldots \overleftarrow{n}$

while there exists a mobile integer do

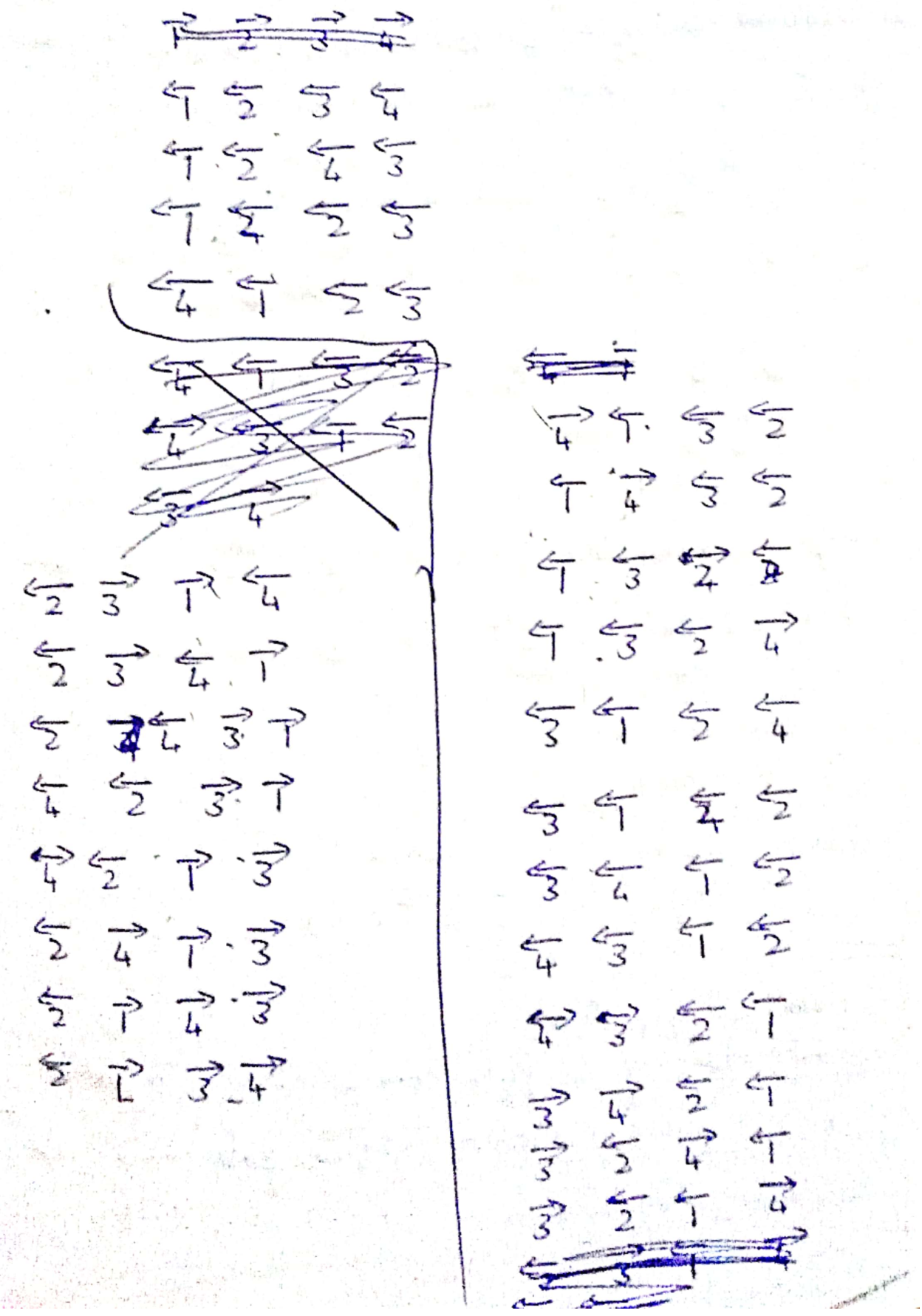
for the largest mobile integer k

Swap k and adj integer its arrow points to

Reverse the direction of all integer that are larger than k

add the new permutation to the least.

n=4

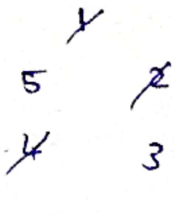


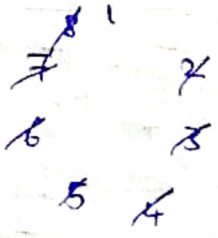
Decrease by constant factor method

1) Josephus's problem

Let n be the number of people numbered from 1 to n standing in a circle starting the count with person 1 we eliminate every 2nd person until 1 person is left.

n = 5





5 3

= 3

= 1.

i) If n is even then

$J(n) = J(2k)$ $\quad J(1) = 1$

$= 2[J(k) - 1$

ii) If n is odd

$J(n) = J(2k+1)$

$= 2J(k) + 1$

---

2) Fake coin problem:

The objective of fake coin problem is to find a fake coin among n coins. Assume fake coin weigh less than original coins

* Divide the given coins into 2 groups and weigh them
* Further divide the set which weighs less and continue

Efficiency:

Basic operation : Weighing coins

$W(n) \Rightarrow$ Total number of weighing operation done

$$W(n) = W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1$$

$$n^{\log_b a} = n^0 = 1$$

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

Variable size decrease. (4 marks)

1) Selection problem

We find the kth smallest element among n elements, if k value is $n/2$ then kth element is called median.

To find median, we will use concept of partition as used in quick sort. Let s denotes the position of partition.

if k is $\lceil n/2 \rceil$, then k^{th} element is called median.

i) if $s = k$, s^{th} element is median.

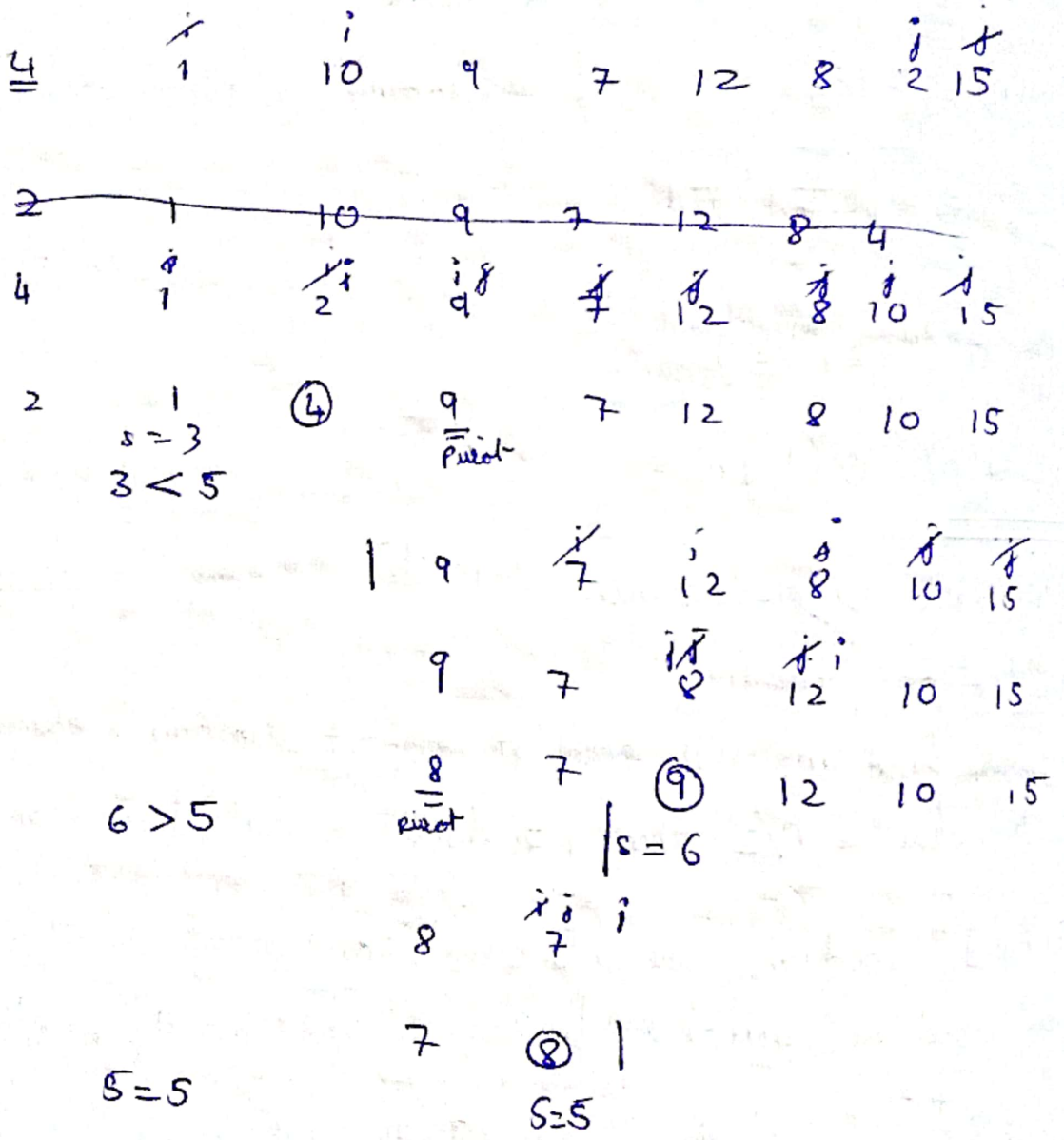
ii) if $s < k$, we need to do partition on right side

iii) if $s > k$, we need to do partition on left side

Compute median for the elements.

4, 1, 10, 9, 7, 12, 8, 2, 15

$n = 9 \quad \left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{9}{2} \right\rceil = 5$

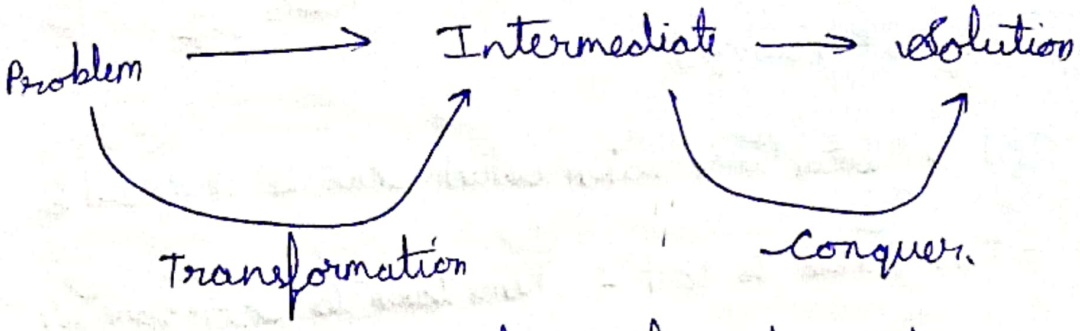


Median = 5

Unit -3

Transform and Conquer

In transform and conquer a given problem instance is transformed into an intermediate instance and then each intermediate instance we find solution



There are 3 variants of tranformation stage

* Instance simplification (Presorting, AVL tree)
* Representation change (Heap sort, 2-3 tree
* Problem reduction: (finding LCM using GCD)

Instance simplification :

Pre-sorting is not a direct problem solving technique, it is just an intermediate state for finding the solution to a given problem

1) Element uniqueness

Algorithm Ele_Uniq (A[0.....n-1])
// Input : Array of n elements
// Output : Returns where array unique or not
// Description : Checks Uniqueness

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

Complexity [efficiency]:-

Let T(n) be total time taken which can be expressed as

T(n) = Time taken to sort + Time taken to compare adj. elements.

= nlogn + n

T(n) = O(nlogn)

*****

To find mode using Pre-sorting.

mode:- most frequency element among n elements.

```
Algorithm Presort_mode (A[0 .... n-1])
// Description :- to find mode
// Input       :- Array of element
// Output      :- mode

sort A
i ← 0
modefrequency ← 0
while i <= n-1 do
    runlength ← 1
    runvalue ← A[i]
```

while i+runlength <= n-1 and A[i+runlength] = runvalue do

runlength ← runlength+1

if runlength > modefrequency

modefrequency ← runlength

modevalue ← runvalue

i = i + runlength $O(n^2)$

return modeValue

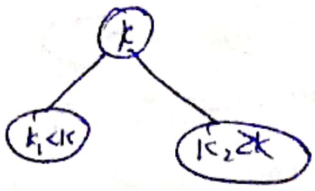
2 5 10 10 10 10 10 15 15 20 20 20

Representation change

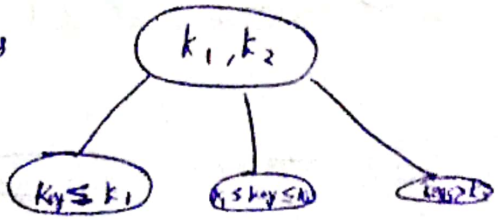
1) 2-3 tree

2-3 tree is a balanced search tree with following property.

* A node can have 1 key like



* A node can have 2 keys as



* All leaf nodes should be at same level.

ht 2-3 '0' = 2

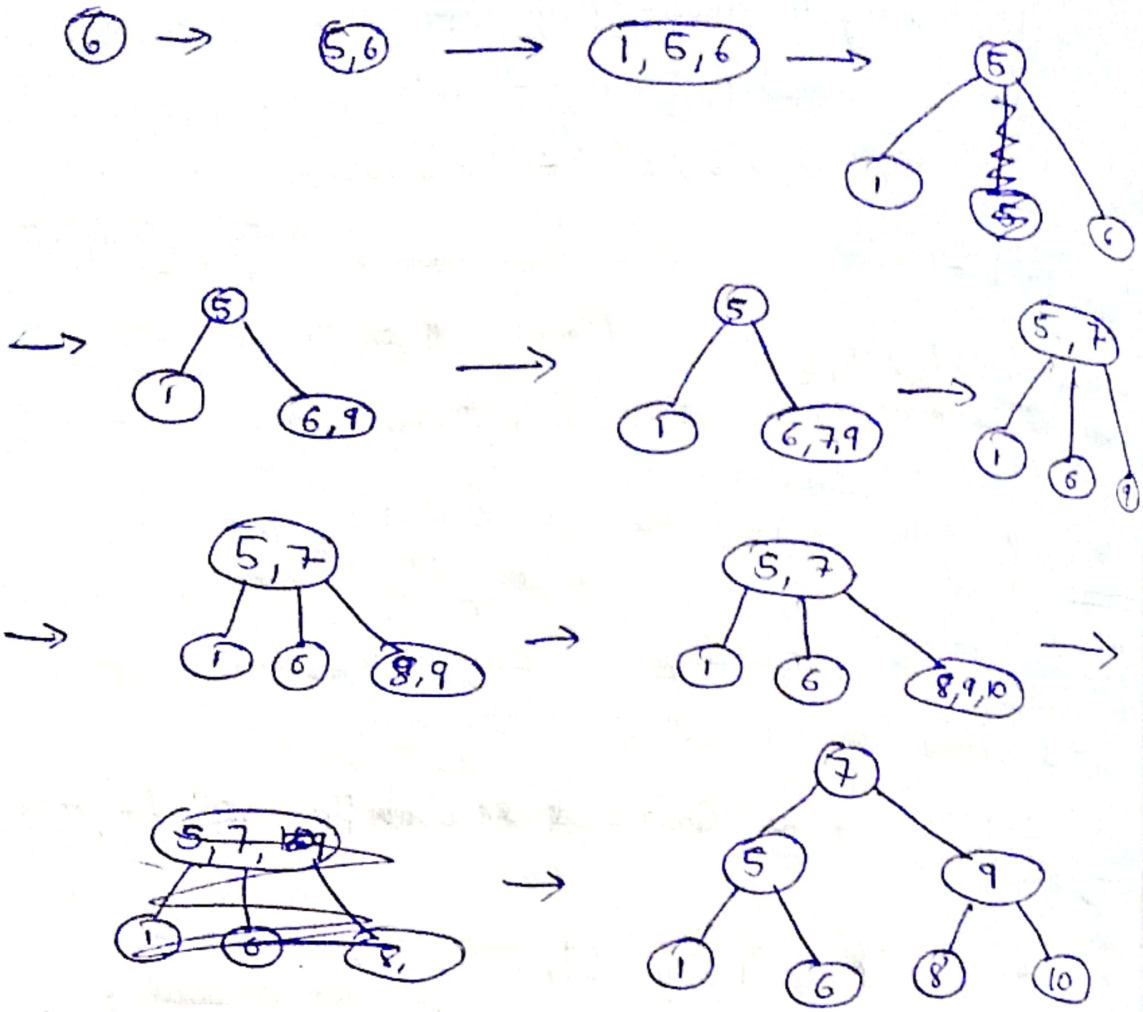
ht 2-3 '1' = 8

ht 2-3 '2' = 26

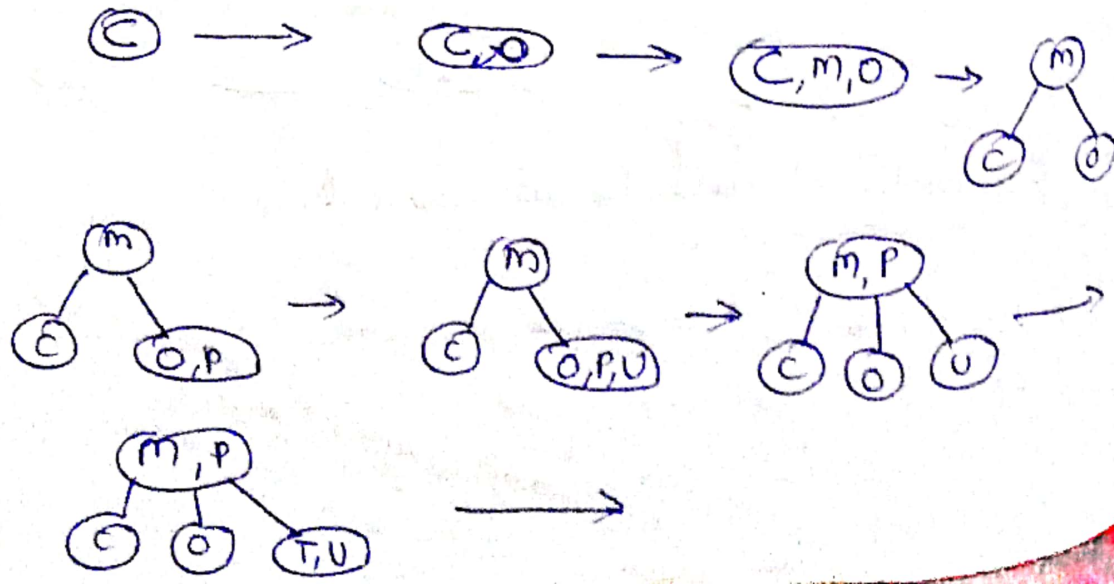
ht 2-3 '3' = 80

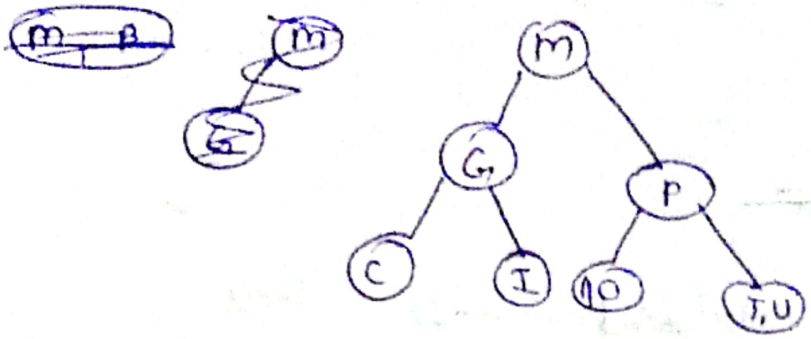
Create a 2-3 tree for value.

6 5 1 9 7 8 10



* COMPUTERING





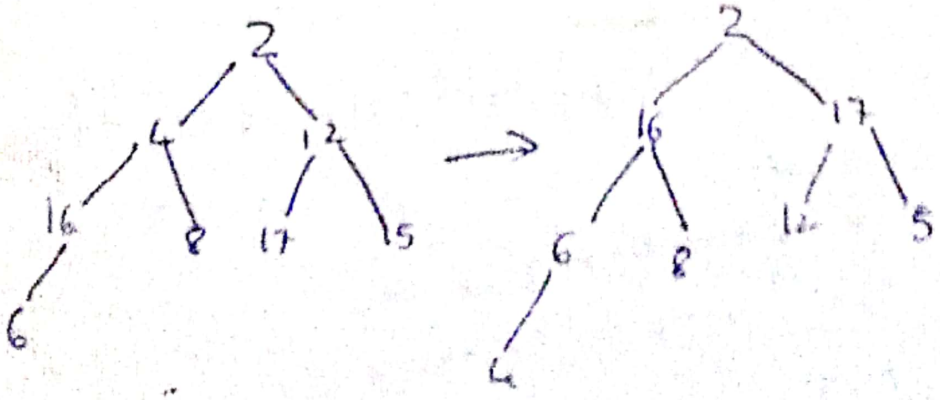
HeapSort:

Procedure to perform heap sort:

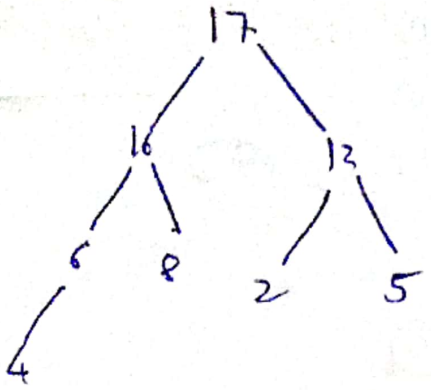
1) Construct a heap for given set of n-elements (max heap)
2) Perform <u>delete root operation</u> n-1 times

Construct max heap

2, 4, 12, 16, 8, 17, 15, 6



→



Algorithm heapify(H[0.....n-1]) $\Theta(n)$

//Description : Construct max heap using bottom up approach

An array H of elements

//Output : max heap

Heapify = $\Theta(n)$

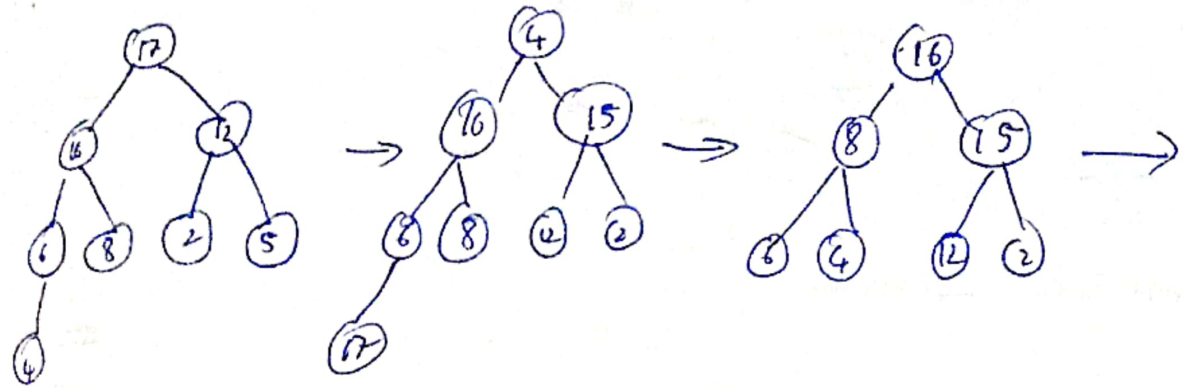
Delete $= O((n-1)\Theta(\log n))$

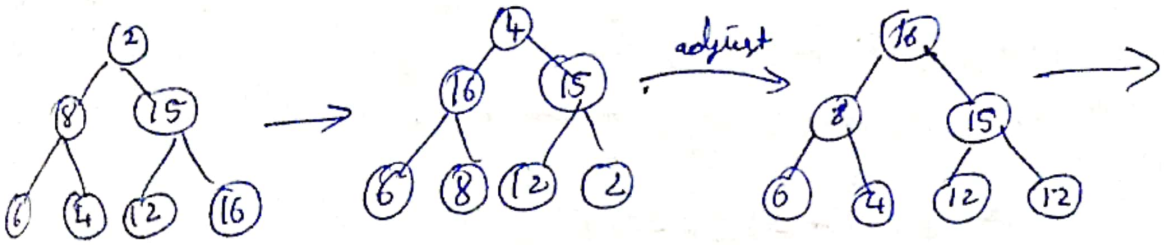
```
for i ← ⌊n/2⌋ down to 1
    k ← i
    v ← H[k]
    Heap ← False
    while not heap and 2*k <= n do
        j = 2k
        if j < n
            if H[j] < H[j+1]  j ← j+1
        if v >= H[j]
            Heap ← True
        else
            H[k] = H[j]
            k ← j
```

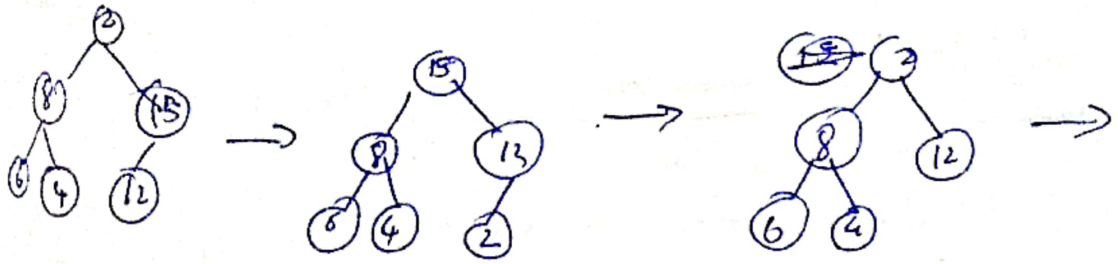
H[k] = V

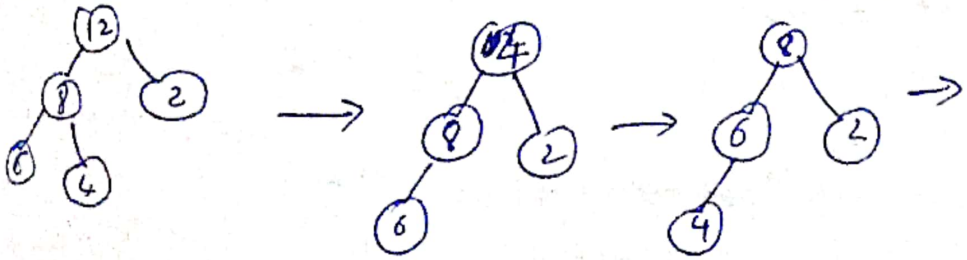
Delete root operation [~~O(n) O(log n)~~]

*Exchange the root's key with the last key [k] of the heap
*Decrease the size of the heap by 1.
*Heapify the smaller tree by shifting k down the tree









17 16 15 6 8 12 2 4

4 8 4 2

16 8 15 6 4 12 2 4

15 8 6 4 12 2

12 8 2 6 4

8 6 2 4

6

Efficiency of heap sort depends on time taken to construct heap ($O(n)$) and time taken to perform max operation $n-1$ times is $(n-1)\log n$.

$$\text{Total time} = O(n) + O(n\log n)$$
$$= O(n\log n)$$

---

p-3

```
#include <stdio.h>

void heapify (int a[], int n) {
  int , i, k, j, v, heap ;
  for (i = n/2 ; i >= 1 ; i--) {
      k = i ;
      v = a[k] ;
    heap = 0 ;
     while ( !heap && 2*k <= n) {
         j = 2*k ;
        if ( j < n)
           if (a[j] < a[j+1])
                 j = j+1 ;
        if (v >= a[j]) {
            heap = 1 ;
        }
      else {
          a[k] = a[j] ;
           k = j ; }
```

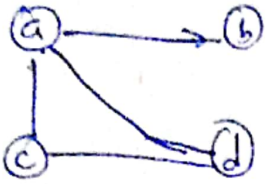
```
        a[k] = v;
    }
}

void heapsort(int a[], n)
{
    int i, temp;
    for (i=n; i>=1; i--)
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        heapify (a, i-1)
    }
}

int main() {
    int a[100], n;
    // Read n
    // Read a[100]
    heapify (a, n);
    heapify (a, n);
    heapsort (a, n)
    // print a[100];
    return 0;
}
```

Problem Reduction :

Problem reduction is a strategy that involves a transformation of problem A to another type of problem B. It is assumed that solution of problem B already exists.

S → B

Eg : $lcm(a,b) = \frac{a \times b}{gcd(a,b)}$



$$A' = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

A' indicates number of paths of length 1 between between the corresponding vertices of graph.

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

Matrix A^2 indicates number of paths of length two between corresponding vertices of graph.

Time and Space Tradeoff :-

1) Input enhancement :

String matching :

Objective of string matching is to find substring in main string.

substring is termed as pattern denoted with 'P', we assume length of P is n characters

main string is called as text denoted with 'T', we assume length of text is n characters

Brute force String Matching :

Best case = Ω(m)
Worst case = (n-m+1) x (m)

Algorithm BruteForce_StringMatch (T[0....n-1], P[0.....m-1])
//Description : Implements naive string matching algorithm
//Input : Text of n chars and pattern of m chars
//Output : value 1 if P is found in T, else value of -1

```
for i ← 0 to n-m do
    j ← 0
    while j < m && P[j] == T[i+j] do
        j = j+1
    if j == m return i
return -1
```

Basic operation : Comparision

Let C(n) denotes total number of comparisions which is denoted as

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

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

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

$$= nm - m^2 + m$$

$$C(n) = O(nm)$$

Find the no of comparisions done in finding the pattern 0001 in a text of 10000 0's 10,000 1's

$(n-m+1)\,m$

$(10,000-4+1)\,4$

$(9,997)\,4$

$= 39988$

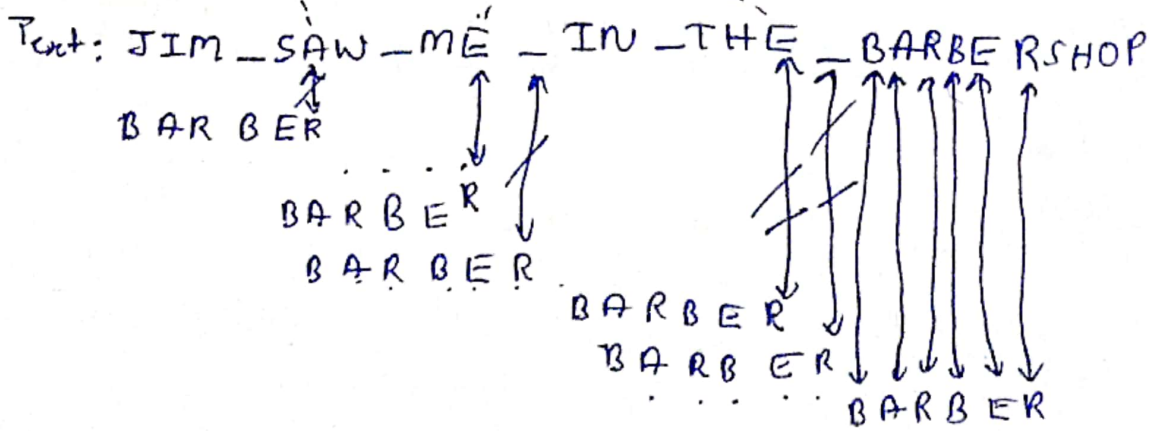
$= 10,000-4+1$

$= 9,9997$ comparisions

Horspool string matching algorithm.

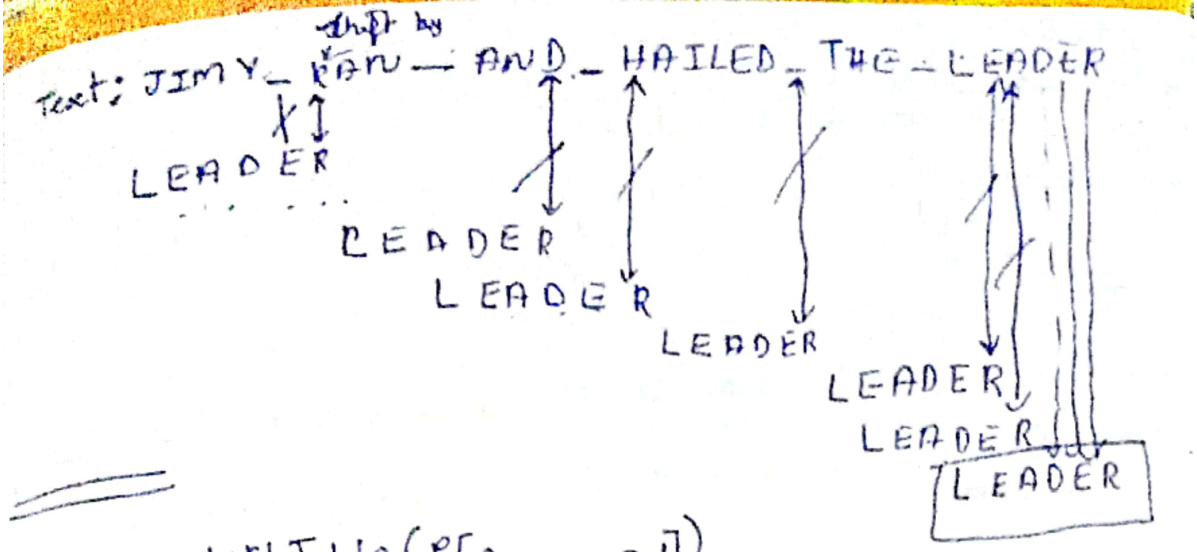
Let pattern be BARBER, the shift table for harbor is

B	A	R	B	E	R
2	4	3	2	1	3



Pattern as LEADER

L	E	A	D	E	R
5	1	3	2	1	6



Algorithm shiftTable(P[0.....n-1])

//Description :- Fills the shift table used in horspool/ Boyer moore string matching

//Input : Pattern P of m characters

//Output: shift table [0.....256-1]

Initialize all elements of table SIZE with m.

for i ← 0 to m-2 do

Table[P[i]] ← m-1-i

Algorithm horspool(T[0.....n-1], P[0....m-1])

//Description : Implements horspool string matching algorithm

//Input : A text of n characters and pattern of m-characters

//Output : Returns position of pattern in text (1st occurance) else returns -1

~~shiftTable~~

shiftTable(P)

while i <= n-1 do

k ← 0 // number of matched characters

while k < m and p[m-1-k] = p[i-k] do

k = k+1

if k = m

return i-m+1

else

$i \leftarrow i + Table(P[i])$

return -1

Find number of comparision in a text of 100 zeros while searching a pattern 10000 using horspool.

$(n-m+1)\,m$

$= (100-4+1)\,4$

$= 97 \times 4$

$= 388$

$(n-m+1)\,m$

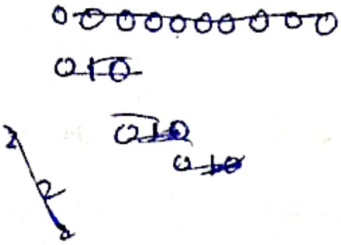
$(100-5+1)\,5$

96×5

480

Number of shift = 96

Pattern 010



Time complexity of horspool string matching also $O(mn)$

For random input = $O(n)$

Exp -4

```
#include <stdio.h>
#include <string.h>
#define SIZE 256

int Table[SIZE]

void shiftTable(char P[]) {
    int i,m;
    m= strlen(P)

    for(i<-0 to; i< SIZE; i++)
        Table[i] = m

    for(i=0 ; i<=m-2 ; i++)
        Table[P[i]] = m-1-i ;
}

int horspool(char T[], char P[]) {
    int i,j,k,m,n;
    n=strlen(T);
    m=strlen(P);
    shiftTable(P)
    i=m-1
    while (i<=n-1) {
        k=0
        while k<m && P[m-1-k] = T[i-k] do
            k<-k+1
        if k=m
            return i-m+1
        else
    }       i<- i + Table[T[i]]
```

```
    return -1;
}

int main() {
    char T[100], P[20];
    //printf("Enter the text\n");
    scanf(" //Read T
    printf("Enter the pattern\n");
    //Read P;
    printf(" Pattern is present at %d", horspool(T,P));
}
```

Boyer Moore's string matching algorithm

In boyer moore's string matching algorithm the number of matched character is also taken into consideration while shifting pattern in any iteration

In this algorithm we calculate two tables namely bad shift table (d_1) and good suffix table (d_2)

Bad shift table (d_1)

case i)

Text:

```
    X E R
B A R B E R
```

$d_1 = st(a) - k$

$= 6 - 2$

$= 4$

Prefix ← same
shift = len

Case ii)

Text: X

BARBER

$d_1 = 6 - 0 = 6$

Note:- If $st(c) - k$ is negative, shift the pattern by one position to its right.

$d_1 = \max\{st(c) - k, 1\}$

Good suffix table (d_2)

1) ABCBAB

k	Pattern	d_2
1	ABCBAB	2
2	ABCBAB	4
3	ABCBAB	4

3) AT_THAT

k	Pattern	d_2
1	AT_THAT	3
2	AT_THAT	5
3	AT_THAT	5

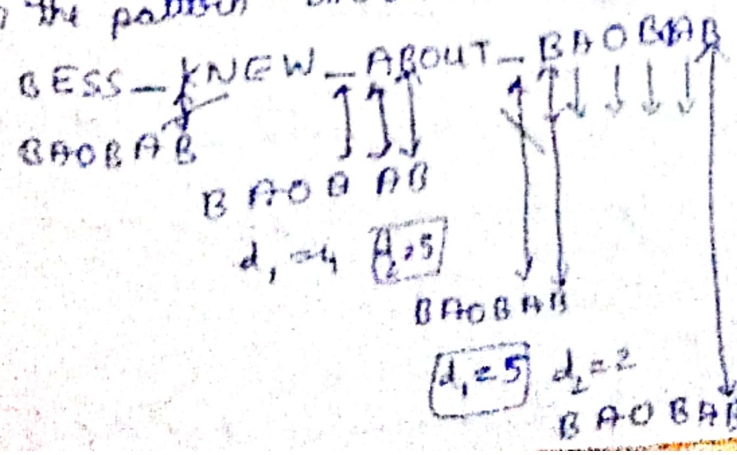
2) BAOBAB

k	Pattern	d_2
1	BAOBAB	2
2	BAOBAB	5
3	BAOBAB	5

4) 01001

k	Pattern	d_2
1	01001	5
2	01001	3
3	01001	3

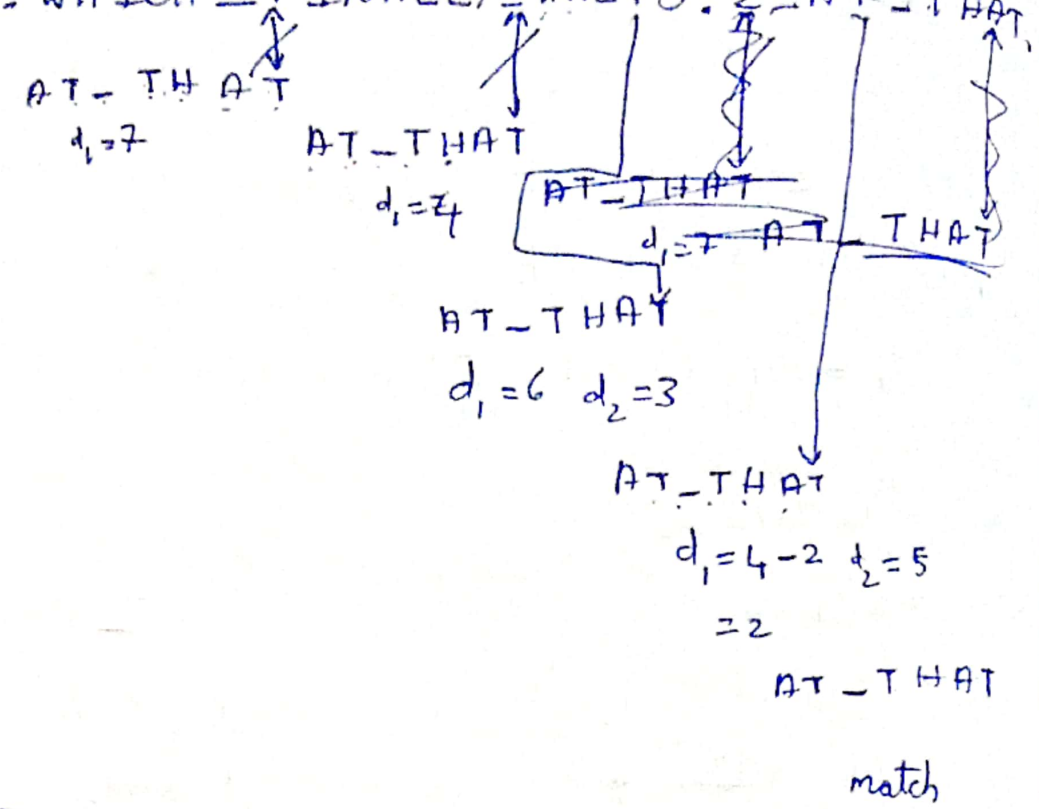
Search the pattern BAOBAB in the text



12 comparisons

Pattern : AT_THAT

Text : WHICH_FINALLY_HALTS.2_AT_THAT.



Procedure for boyer moore algorithm.

Step 1 : For given pattern and alphabet used both in pattern and text construct bad shift table (d_1)

$$d_1 = \max[s(t) - k, 1]$$

Step 2 : Using the pattern construct good suffix table.

Step 3 : Align the pattern against beginning of the text

Step 4 : Repeat the following step until a matching substring is found or the pattern reaches beyond the last character of text.

Starting with last character of pattern compare.

corresponding characters in pattern & text (k is no of matching characters do not match after some comparision.

shift is done based on following.

$$d = \begin{cases} d_1 \text{ if } k = 0 \\ \max[d, d_2] \text{ if } k > 0 \end{cases}$$

---

comparision, count sort.:

	62	31	84	96	19	47
	0	0	0	0	0	0
	3	0	1	1	0	0
		1	2	2	0	1
			4	3	0	1
				5	0	1
					0	2
	3	1	4	5	0	2

final array

0	1	2	3	4	5
19	31	47	62	84	96

Algorithm Comparision CountSort (A[0....n-1])

//Description : Sorts an array using comparision sort

//Input : Array

//Output: ~~Sorted array~~ Array S of n elements of A's elements in sorted order.

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

Time complexity :

B.O → Comparision.

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

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

$$= (n-1) + (n-2) + (n-3) + \dots 1$$

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

Distribution count start:

Elements →	13	11	12	11	12	12	13
Elements	11	12	13				
frequency	2	3	2				
Distribution value	2	5	7				

→ O(n)

	1	2	3	4	5	6	7
Output	11	11	12	12	12	13	13

Algorithm

Unit - 4

Dynamic Programming:

It is similar to divide and conquer, where given problem instance is divided into smaller problem instance.

The overlapping subproblem is solved only once in dynamic programming whereas in divide and conquer overlapping subproblem is solved multiple points.

The solution of subproblem is stored or recorded in table which is used further.

Binomial Co-efficient.

* Objective is to find number of combination of k elements in given set of elements.

$^{n}C_{k}$ (or) $C(n,k)$

The recursive definition to find $C(n,k)$ is

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

	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 binomial coefficient (n,k)

// Description: computes $C(n,k)$ using D.P

// Input: Two non-negative integers n, k

// Output: value of $C(n,k)$

```
for i ← 0 to n do
    for j ← 0 to min(i,k) do
        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)
```

Basic operation is addition :

Total number of addition can be denoted as $C(n) = \sum_{i=0}^{n} \sum_{j=0}^{\min(i,k)} 1$

$$= \sum_{i=0}^{n} k$$

$$C(n) = nk$$

The recurrence relation which gives total number of additions done is

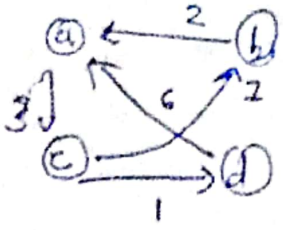
$$A(n,k) = 0, \text{ for } k=0, k=n$$

$$A(n,k) = A(n-1,k) + A(n-i, k-1) + 1$$

Floyd's algorithm [Floyds - Warshall algorithm]

Objective is to find all pair shortest path ~~alg~~

Tracing of floyd's algorithm:



	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	~~∞~~ 0	1
d	6	∞	∞	0

①

	0	∞	3	∞
0	0	∞	3	∞
2	2	~~∞~~ 0	5	∞
∞	∞	(∞) 7	0	1
6	6	∞	9	0

②

	2	0	5	∞
∞	0	∞	3	∞
0 (∞)	2	0	5	∞
7	9	7	0	1
∞	6	∞	9	0

③

	9	7	0	1
3	0	10	3	4
5	2	0	5	6
0	9	7	0	1
9	(6)	16	9	0

④

	6	16	9	0
4	0	10	3	4
6	2	0	5	6
1	~~3~~ 7	7	0	1
0	(6)	16	9	0

↗ Shortest path

$\Theta(n^3)$

Algorithm floyds ($W[1....n][1....n]$)

// Description :- Implements floyds algorithm using DP

// Input : ~~weighted~~ weighted graph represented through matrix W

// Output : matrix which gives all pair shortest path

```
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
```

Time complexity is $O(n^3)$

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

①

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

②

	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

③

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

④

	∞	∞	2	0	3
4'	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

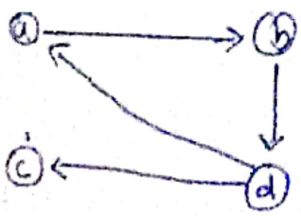
⑤

	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	9	0	3
0	3	5	6	4	0

Warshall's algorithm:

It is used to find transitive closureness.

Transitive closureness of directed graph for n nodes can be represented as an n×n boolean matrix $T = \{t_{ij}\}$ in which element in the i^{th} row and j^{th} column is 1 if there exist a non-trivial path from i to j; otherwise it will be 0.



	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

①

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

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

②

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

③

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

④

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

Algorithm Warshall's (A[1.....n, 1.....n])

//Description :- Finds transitive closureness

//Directed graph represented through adjacent matrix

//Output :- Transitive closureness based matrix

$R^0 \leftarrow 0$

for $k \leftarrow 1$ to n do

for i ← 1 to n do

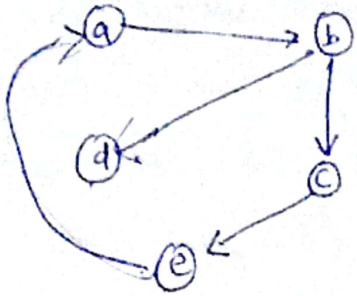
for j ← 1 to n do

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

return $R^n(i,j)$

Note :- Time complexity of Warshall's algorithm is order of $n^3 = O(n^2)$

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



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

→

②

	0	0	1	1	0
0	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

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

→

	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	1	1	1	1	1

④

	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

→

knapsack problem:

In knapsack problem there will be 'n' objects of known weight denoted with $w_1, w_2, w_3, \ldots, w_n$ and profits denoted with $v_1, v_2, v_3, \ldots, v_n$ and knapsack capacity denoted with W. The objective is to add more profitable objects into the knapsack with constraint that sum of weight of added objects should not exceed knapsack capacity.

Let x_i denotes whether the object is included or not

if $x_i = \begin{cases} 1, & \text{it's included} \\ 0, & \text{not included.} \end{cases}$

The knapsack problem can be denoted as

$$\text{maximize} = \sum_{i=1}^{n} x_i v_i$$

subjected to $\sum_{i=1}^{n} x_i w_i \leq W$

Initialization:

n \ knapsack capacity (W)	0	1	2	3	4	.....	W
0	0	0	0	0	0	.....	0
1	0						
2	0						
3	0						
⋮							
n	0						

(objects)

$$V[i,j] = \begin{cases} V[i-1, j] & ; j - w_i < 0 \\ \max\{[V[i-1,j], V[i-1, j-w_i]] + v_i\} & \text{otherwise} \end{cases}$$

Solve given instance of knapsack problem using D.P.

$n = 4$

$W = 5$

$w = \{2, 3, 1, 2\}$

$v = \{8, 8, 16, 16\}$

		0	1	2	3	4	5
	0	0	0	0	0	0	0
②	1	0	0	8	8	8	8
③	2	0	0	8	8	8	14
①	3	0	16	16	24	24	24
②	4	0	16	16	27	27	35

$V(4,5) \neq V(3,5)$ (4th is included)

$V(3,3) \neq V(2,3)$ (3rd obj is included)

$V(2,2) = V(1,2)$ (2nd obj is excluded)

$V(1,2) \neq V(0,2)$ (1st obj is included)

	0	0	0	0	0	0
2	0	0	12	12	12	12
3	0	10	12	22	22	22
3	0	10	12	22	30	32
2	0	10	15	25		37

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

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

$V(2,3) \neq V(1,3)$

$V(1,2)$

Inc

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

excluded

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

$V(2,1) =$

Exp - 4

```
#include <stdio.h>

int max(int a, int b) {
    return a > b ? a : b;
}

int n, W, w[10], v[10], V[10][10], X[10] = {0};

int main() {
    int i;
    printf("Read number of objects\n");
    scanf("%d", &n);
    printf("Read weights of objects\n");
    for(i=0; i<=n; i++)
        scanf("%d", &w[i]);
```

Read the profits.

```
    for(i=0; i<=n; i++)
        scanf("%d", &v[i]);
```

Read knapsack capacity

```
    scanf("%d", &W);

    for(i=0; i<=n; i++) {
        for(int j=0; j<=W; j++){
            if(i==0 || j==0)
                V[i][j] = 0;
            else if (j-w[i]) < 0)
                V[i][j] = V[i-1][j];
```

```
        else
            V[i][j] = max(V[i-1][j], V[i-1][j-w[i]] + V[i]);
        }
    }
print();
printf("Max profit = %d", V[n][j]);
printf("\n ------ ");
for(i=1; i<=n; i++)
    { if (x[i]==1)
        printf("\n%d obj is included", i);
    }
return 0;
}

void print() {
    int i,j;
    i=n; j=W;
    while(i!=0 || j!=0)
    { if(V[i][j] != V[i-1][j]) {
        x[i] = 1
        j = j - w[i];
      }
      i=i-1;
    }
}
```

$\Rightarrow O(nW)$

Memory function:

It is recursive in nature, it uses concept of both top-down and bottom up approach, initially all the entries of table apart from 0th row and 0th column are -1

Algorithm ~~memory function~~ MFK (i, j)

//Description: Implements memory function based method to solve napsack

//Input: Two non-negative integers i & j (j: Knapsack capacity; i: Number of first object being considered)

//Output: Values of optimal feasible ~~subset~~ set of first n objects.

Note: -1st row $\leftarrow 0$, 1st col $\leftarrow 0$ & rem values are -1.

```
if V[i,j] < 0
    if j < weight[i]
        value ← MFK(i-1, j)
    else
        value ← max(~~MFknapsack~~ MFK(i-1, j), MFK[i-1
    V[i, j] ← value.
return V[i][j]
```

$mF(4,5) = \max(mF(3,5), MF(3,3)+11)$

$mF(3,5) = \max(mF(2,5), mF(2,4)+16)$

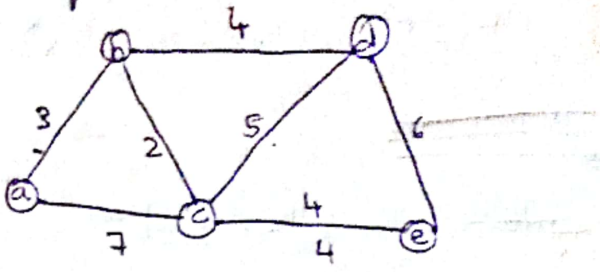
$mF(3,3) = \max(mF(2,3), mF(2,0)+6)$

$mF(2,5) = \max(mF(1,5), mF(2,$

Dijkstra's Algo

→ find shortest path from source vertex

→ if there is -ve edge, it fails



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

Soln ← ϕ

source vertex = {a}

$W = V$

vertex	d	p
a	~~0~~	Nil
b	∞	Nil
c	∞	Nil
d	∞	Nil
e	∞	Nil
f	∞	Nil

① $W = \{a, b, c, d, e\}$

↳ extractmin

Soln = {a}

Vertex	d	p
a	0	a
b	3	a
c	7	a
d	∞	Nil
e	∞	Nil

② $W = \{b, c, d, e\}$

↳ extractmin

Soln = {a, b}

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

Vertex	d	p
a	0	a
b	3	a
c	5	b
d	7	b
e	∞	Nil

③ $W = \{c, d, e\}$
↳ extract min

$Soln = \{a, b, c\}$
$W = \{d, e\}$

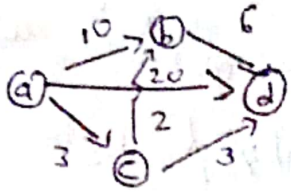
Vertex	d	p
a	0	a
b	8	a
c	5	b
d	7	b
e	9	c

④ $W = \{d, e\}$
$Soln = \{a, b, c, d\}$
$W = \{e\}$

Vertex	d	p
a	0	a
b	3	a
c	5	b
d	7	b
e	9	c

$W = \{a, b, c, d\}$ → Extract min
$Soln = \{a\}$

Vertex	d	p
a	0	a
b	~~10~~ 10	a
c	3	a
d	20	a



② $W = \{b, c, d\}$ → extract min
$Soln = \{a, c\}$

Vertex	d	p
a	0	a
b	~~10~~ 5	~~b~~ c
c	3	a
d	~~7~~ 6	~~b~~ c

$W = \{c, d\}$ → extract min
$Soln = \{a, c, b\}$

Vertex	d	p
a	0	a
b	5	c
c	3	a
d	6	c

Algorithm Dijkstra's (G, s)

// Description : Implement single source, shortest path

// Input : Connected weighted graph G, source s.

// Output : Shortest path from source vertex to all vertex.

```
for each v in V do
    t[v] ← ∞
    p[v] ← Nil
d[s] ← 0
W ← V
Soln ← ϕ
while W ≠ ϕ do
    u ← extract_min (W)
  Solution ← Solution Union {u}
  W ← W - {u}
  for each vertex v in V adjacent to u
      if d[v] > d[u] + cost(u,v)
          d[v] ← d[u] + c(u,v)
          p[v] ← u
return d
```

Note :-

Time complexity is $O(V^2)$ if input is adjacency matrix

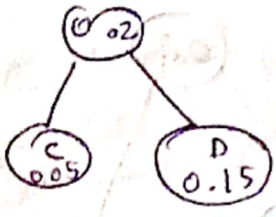
It is $O(E \log N)$ if input is adjacency list and binary heap is constructed for edges.

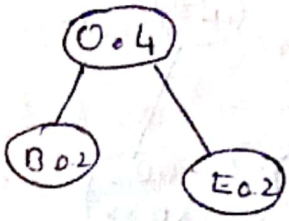
Huffman Tree:

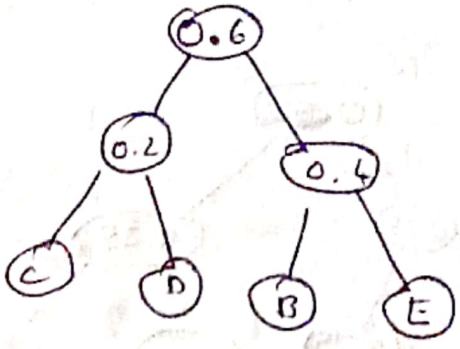
↳ Data encoding (Data compression)
  ↳ Fixed
  ↳ Variable length

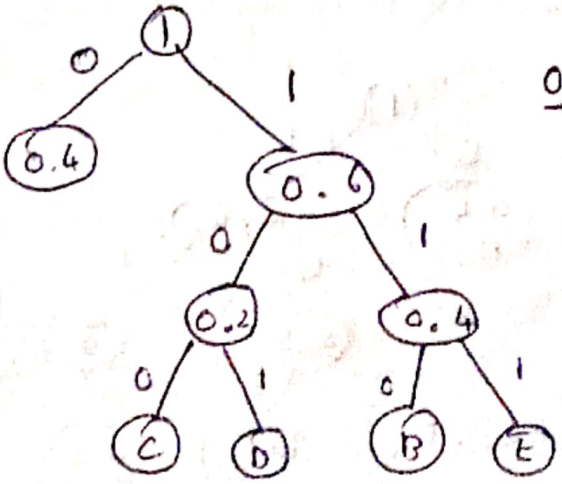
Construct a huffman Tree for following input.

	A	B	C	D	E
f	0.4	0.2	0.05	0.15	0.20









$0.4 \times 1 + 0.6 \times 3$

$= 2.2$

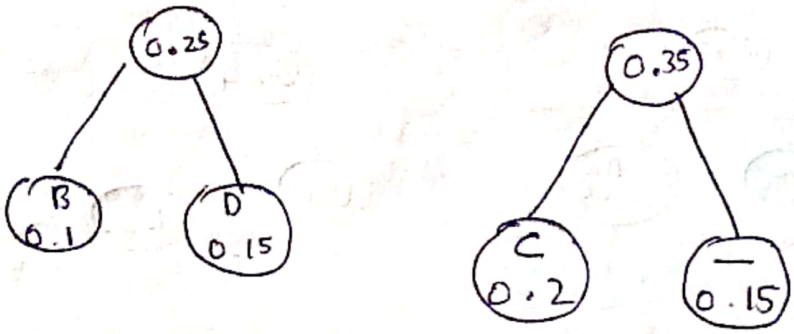
A - 00
B - 110
C - 100
D - 101
E = 111

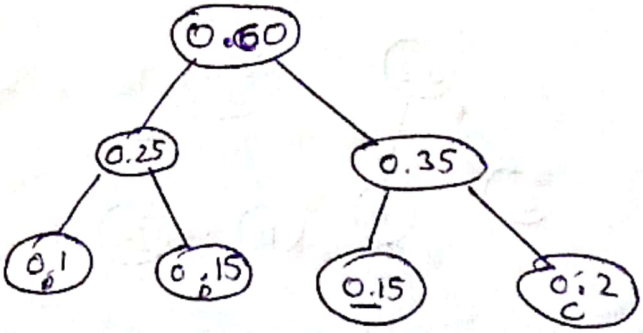
Avg length = 2.2

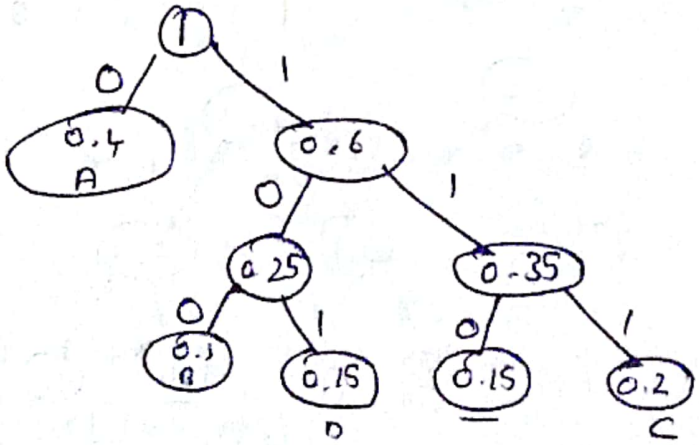
Compression ratio $= \frac{3 - 2.12}{3} \times 100$

$= 26.66$

A	B	C	D	—
0.4	0.1	0.2	0.15	0.15







A = 0

B = 100

C = 111

D = 101

— = 110

$1 \times 0.4 + 0.6 \times 3$

$= 2.2$

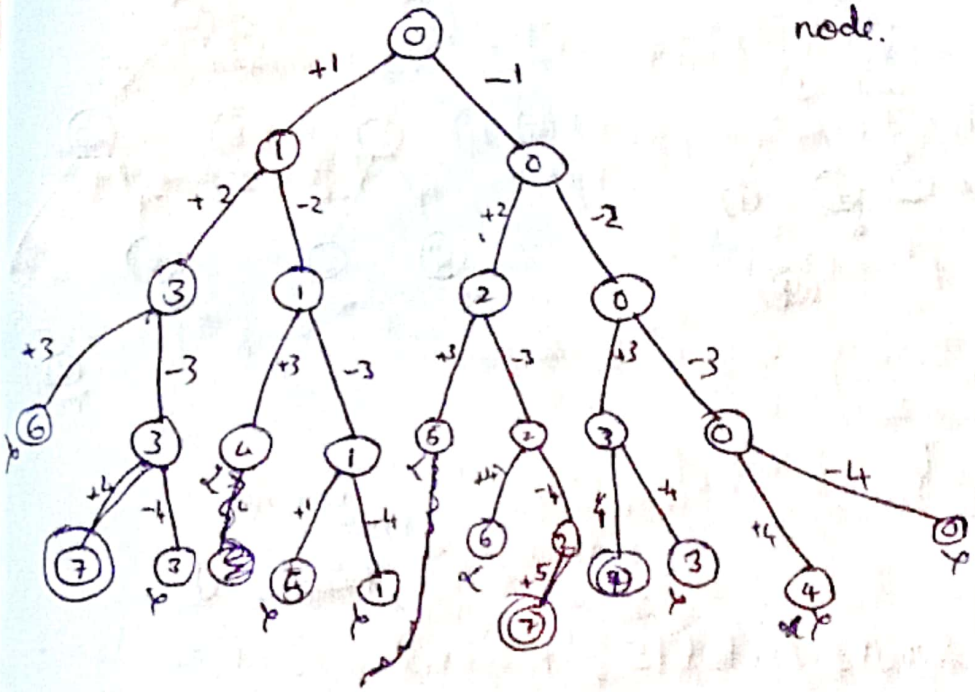
Back-tracking :

Sum of subset problem :

Objective is to find the subsets from the given set S such that the summation of elements of ~~subset~~ subset is equal to a given value 'd'.

Let $S = \{1, 2, 3, 4, 5\}$ $d = 7$

χ = non-promising dead node.



Promising nodes → leads to final ~~nodes~~ solution.

Non-Promising nodes → doesn't leads to final solution

$S_1 = \{1, 2, 4\}$

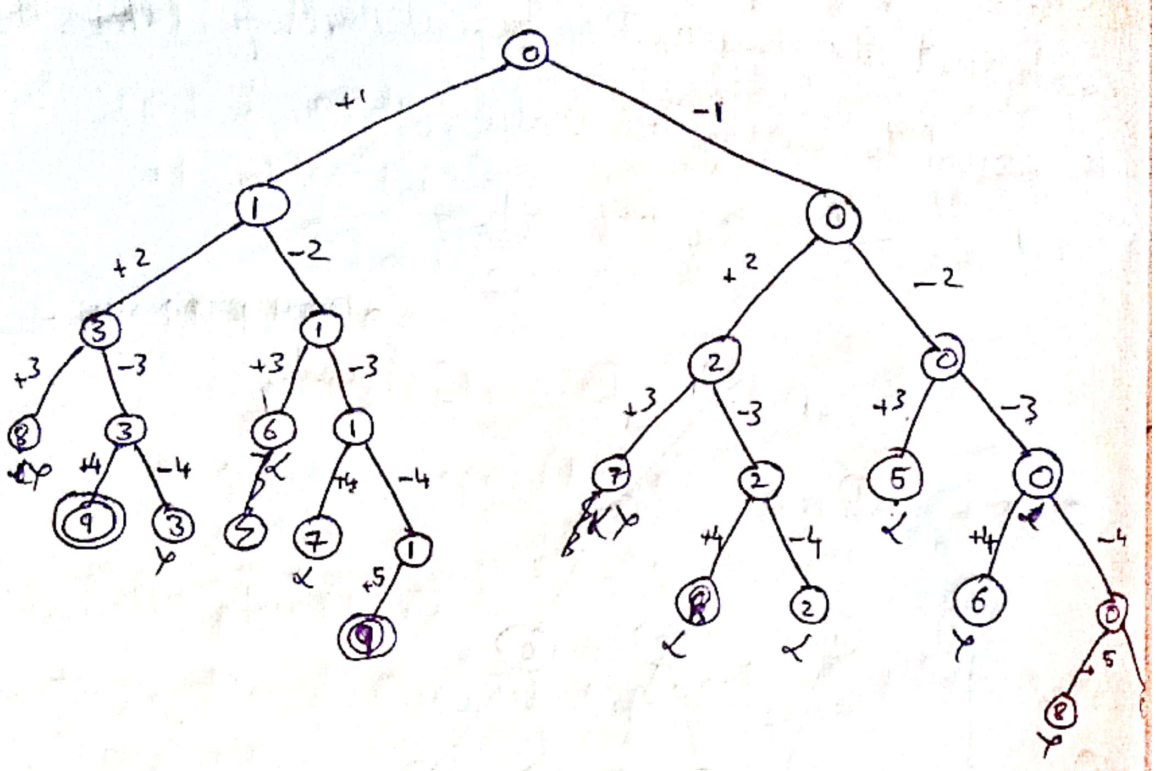
$S_2 = \{2, 5\}$

$S_3 = \{3, 4\}$

Note :- Elements in the given set S should always be in ascending order, if they are not in ~~ascending~~ ascending order, ~~not~~ sort them out.

Find the solution to sum of subsubject of given problem

$S = \{1, 2, 5, 6, 8\}$ $d = 9$



$S_1 = \{1, 2, 6\}$

$S_2 = \{1, 8\}$

Algorithm Sum_of_subset (S , k , r) (initialsum, element index no., ~~final~~ : d.)

//Description : solve sum of subset using backtracking

//Input : W[1 ..... n] which holds elements in increasing order & a integer d

//Output : subsets whose summation is d

```
X[k] ← 1
if( W[k] + S == d)
    write( X[1 .... n])
else if( S + W[k] + W[k+1] <= d)
        Sum_of_subset. (S + W[k], k+1, r - W[k])
if ( S + r - W[k] >= d and S + W[k+1] < d)
```

x[k] ← 0

Sum-of_subset(s, k+1, r - w[k])

N-Queen's problem:

Objectives of N-queen's problems place n queen's on a chess board of nxn such that no 2 queens attack each other.

Note: Queens attack each other if they are on same colums, same rows and same diagonal

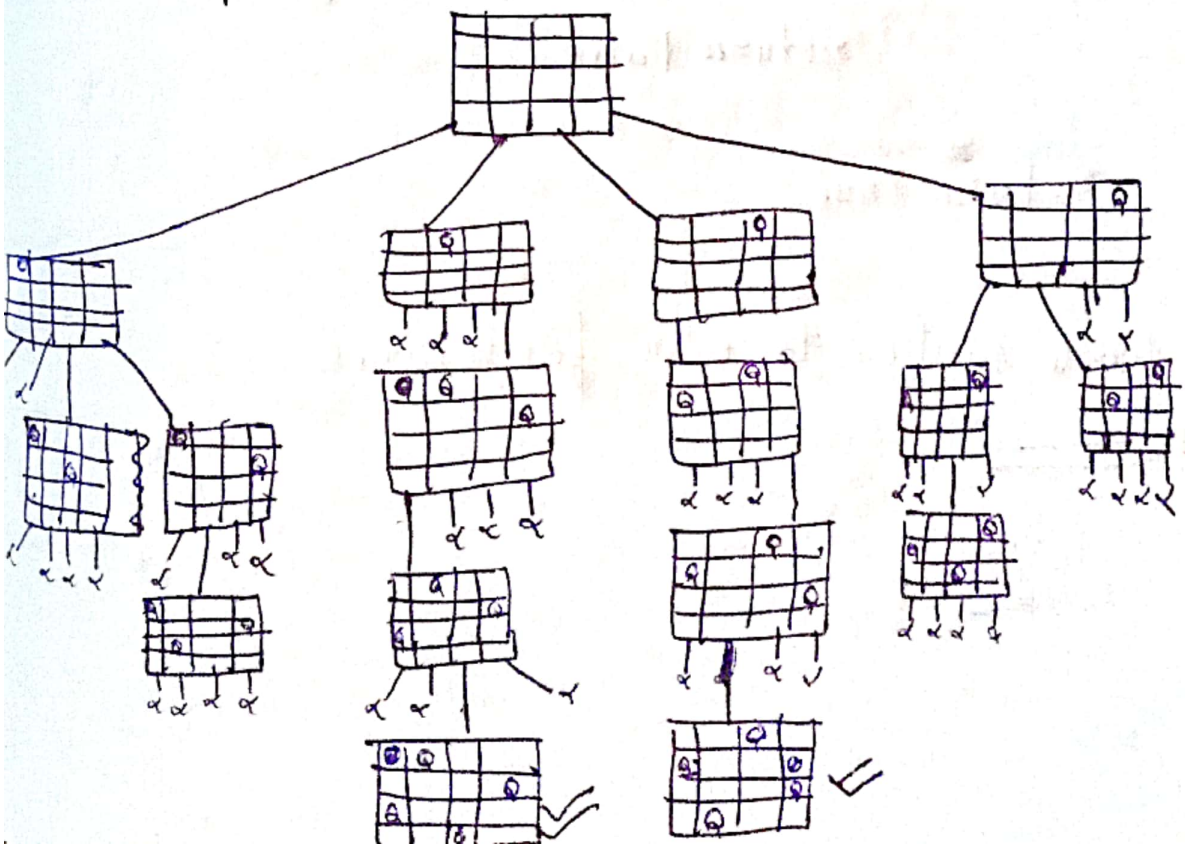
For n=2, n=3 no solution

for n=4 possible solutions are.

*	Q	*	*
*	*	*	Q
Q	*	*	*
*	*	Q	*

*	*	Q	*
Q	*	*	*
*	*	*	Q
*	Q	*	*

state spaced tree for spacing 4 queens.



Algorithm N-Queens (k, n). ← number of queens
↳ Queens number (initial value 1)

// Prints all possible combinations of feasible solution of backtracking

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

Algorithm place (k, i)

// returns true if Queen k is placed at k^{th} row and i^{th} column.

// Let x be an global array.

```
for j ← 1 to k-1 do
    if (x[j] == i or abs(x[j] - i) == abs(j - k))
        return False
return True.
```

Write solution to solve for 5 Queens.

Branch and bound:

* Branch and bound gives the optimal solution
* Branch and bound uses both dfs as well as bfs
* Branch and bound uses bounding function.

First problem

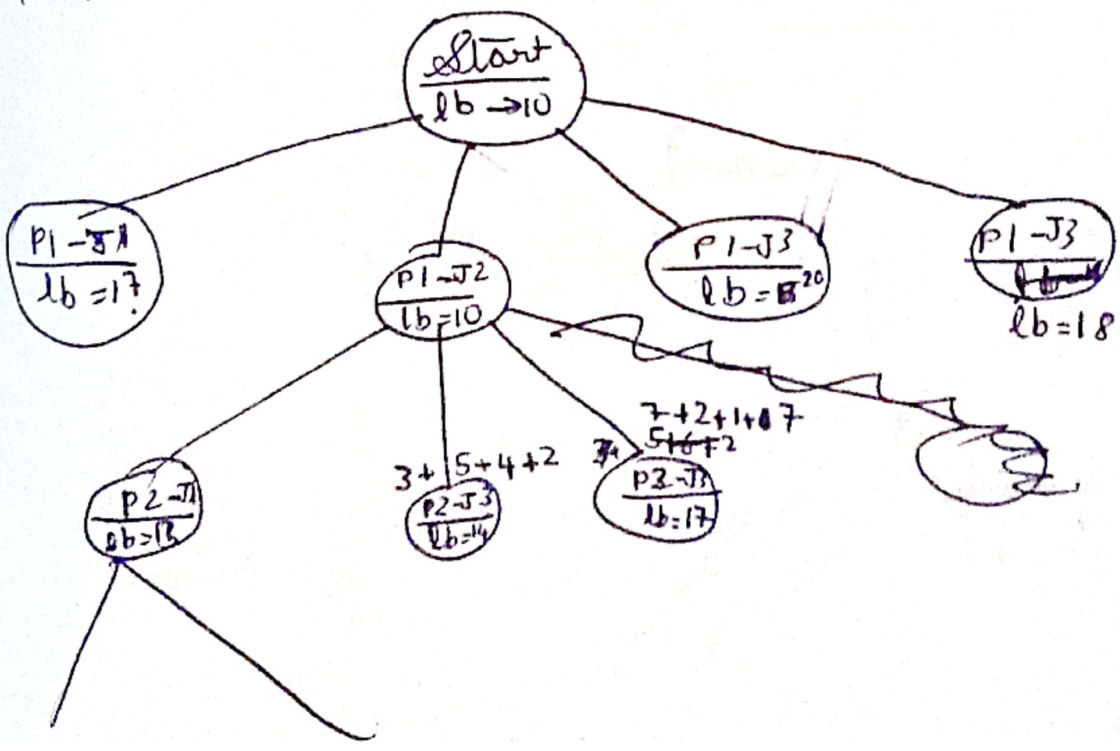
Assignment problem:

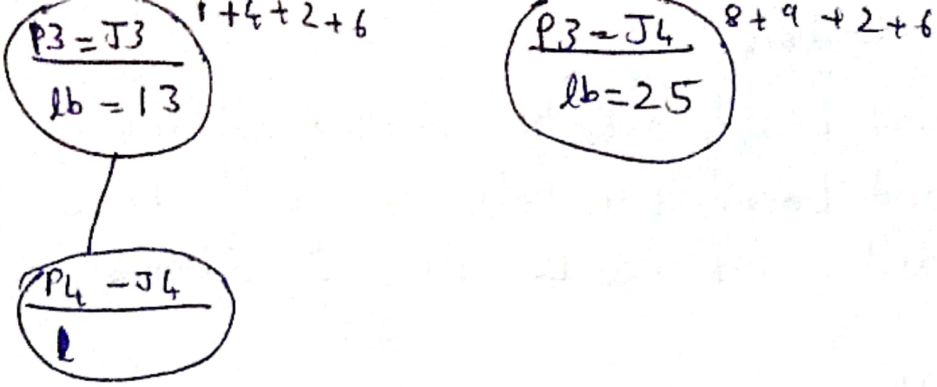
- Objective of assignment problem is to assign 'n' job to n processes such that total cost of assignment is as small as possible

This is a minimization problem.

The lower bound is calculated as summation of the least value in each row.

	J_1	J_2	J_3	J_4
P_1	9	(2)	7	8
P_2	(6)	4	3	7
P_3	5	8	(1)	8
P_4	7	6	9	(4)





Solve

9	11	14	11	7
6	15	13	13	10
12	13	6	8	8
11	9	10	12	9
7	12	14	10	14

7+6+6+9+10

Start lb

lb →

knapsack problem :

$n = 4$

$w = \{4, 7, 5, 3\}$

$v = \{40, 42, 25, 12\}$

$m = 10$

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \frac{v_3}{w_3} \geq \frac{v_4}{w_4}$$

$\frac{v_1}{w_1} = \frac{40}{4} = 10$

$\frac{v_2}{w_2} = \frac{42}{7} = 6$

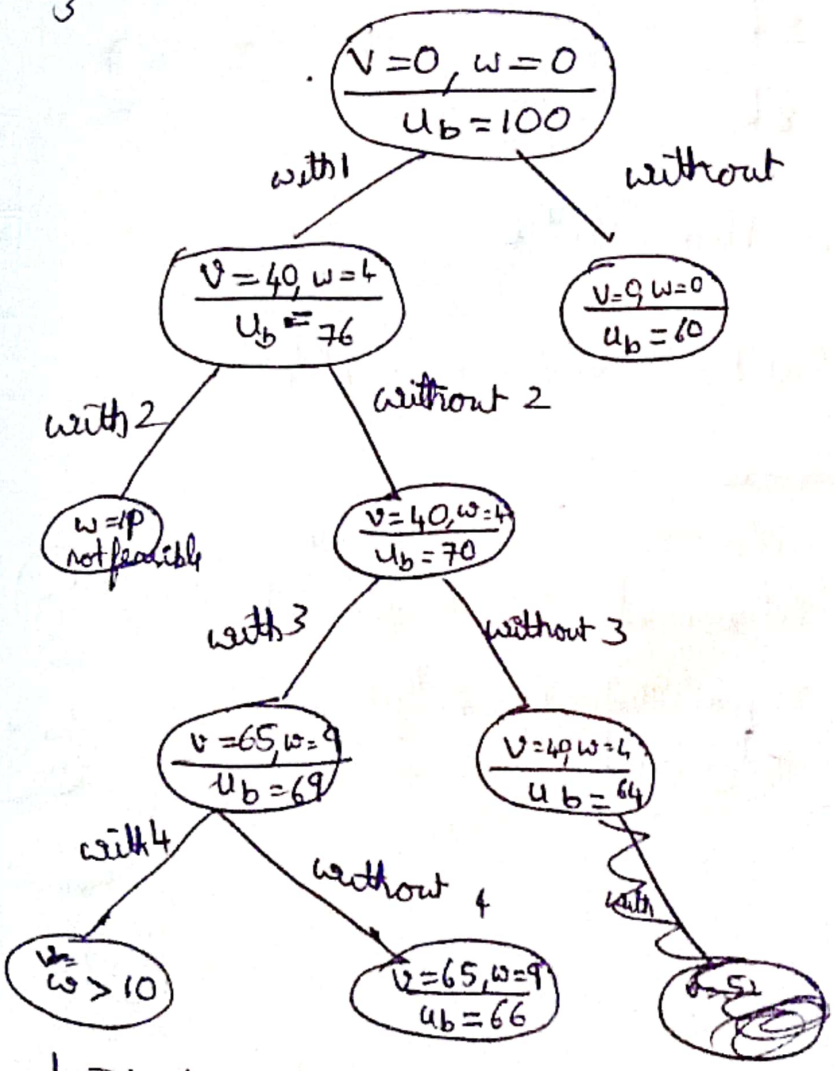
$\frac{v_3}{w_3} = \frac{25}{5} = 5$

$\frac{v_4}{v_{41}} = \frac{12}{3} = 4$

$$U_b = v + (m - w) \cdot \frac{v_{i+1}}{w_{i+1}}$$

If v_i doesn't exist

$\frac{v_i}{w_i} = 1$



object to be included = {1, 3}

max profit = 65

$N = 4$

$m = 5$

$w = \{2, 1, 3, 2\}$

$p = \{8, 6, 15, 10\}$

$\frac{v_i}{w_i} =$

$\frac{v_1}{w_1} = \frac{8}{2} = 4$

$\frac{v_2}{w_2} = \frac{6}{1} = 6$

$\frac{v_3}{w_3} = \frac{15}{3} = 5$

$\frac{v_4}{w_4} = \frac{10}{2} = 5$

$w = \{1, 3, 2, 2\}$

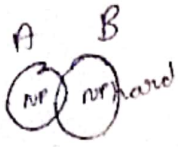
$v_i = \{6, 15, 10, 8\}$

Post-man Problem - Video

Define P, NP, NP hard and NP complete.

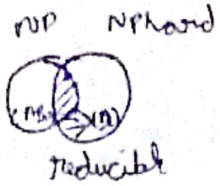
P class ~~are those~~

	Solved	Verify
P	Polynomial	Polynomial
NP	Exponential (or) Polynomial	Polynomial



NP Hard

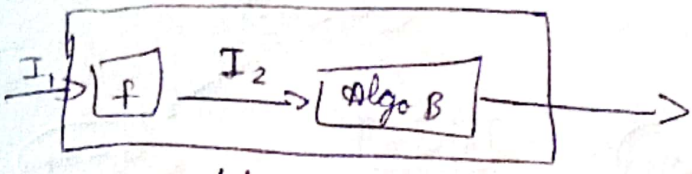
$(A \propto B)$



NP hard $A \propto B$

A → Input I_1 → Algo A

B → Inp I_2 → Algo B



Algo A

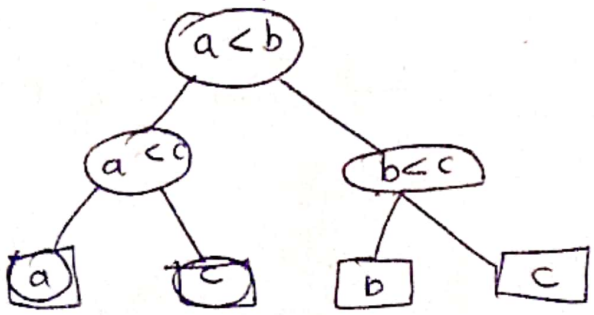
NP hard is that problem through which we can reduce the problem of NP

Decision tree

A decision tree is a full binary tree that shows the comparision between elements that are executed by an appropriate algorithm (basic ~~algo~~ comparision) of operation on a input of given size

* If input size is n, we will have max of n! leaves.

Decision tree to find minimum of 3 elements.



Decision tree to sort 3 elements. ****