Sorting Algorithms:-

Insertion sort

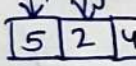
Input:- A sequence of numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output:- A permutation (reordering) $\langle a_1', a_2', \ldots a_n' \rangle$ of the input sequence such that $a_1' \le a_2' \le \ldots \le a_n'$

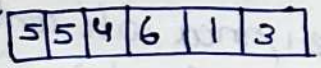
Algorithm:- Insertion sort(A)

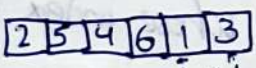
	Cost	Times
1) for j=2 to A.length	c_1	n
2) key = A[j]	c_2	$n-1$
3) //Insert A[j] in the sorted sequence A[1...j-1]	c_3	0
4) i = j-1	c_4	$n-1$
5) while i>0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
6) A[i+1] = A[i]	c_6	$\sum_{j=2}^{n} (t_j-1)$
7) i = i-1	c_7	$\sum_{j=2}^{n} (t_j-1)$
8) A[i+1] = key	c_8	$n-1$

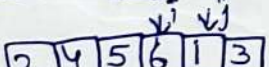
Example:- A = <5,2,4,6,1,3>

a) 

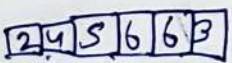
key = 2 A[i] = 5, 5 > 2



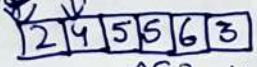


d) 

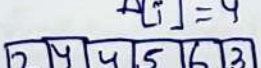
A[i] = 6 key = 1



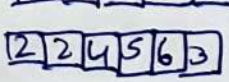
A[i] = 5



A[i] = 4

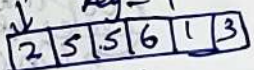


A[i] = 2



b) 

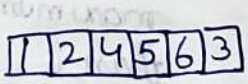
key = 4



A[i+1] = 5 changes both

c) 

key = 6 A[i] = 5

e) 

f) 

Time complexity:-

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Best case:- Array is already sorted

$t_j = 1$

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} 1 + c_8(n-1)$

$= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$

$= an + b$

$T(n)$ is a linear function of n

worst case:- Array is sorted in reverse order

$t_j = j$

$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ $\quad \sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (t_j - 1)$

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right)$

$= an^2 + bn + c$ where a, b, c are constants

$T(n)$ is a quadratic function of n $\boxed{T(n) = O(n^2)}$ $\boxed{T(n) = \theta(n^2)}$

Proof of correctness:-

```
Algorithm max(A)
{ //A is an array
1) max = A[0]
2) for i=1 to A.length-1 step 2o
3)   if A[i] > max then
4)       max = A[i];
5) return max;
}
```

A → 1

Loop invariant:-
↳ which must hold true during entire execution of the loop.

For any given value i, max will contain maximum of elements whose index is smaller than i.

Merge sort

Algorithm:- Merge sort(A, P, r) ⟶ $T(n)$

1) if $p < r$
2) then $q \leftarrow \lfloor (p+r)/2 \rfloor$

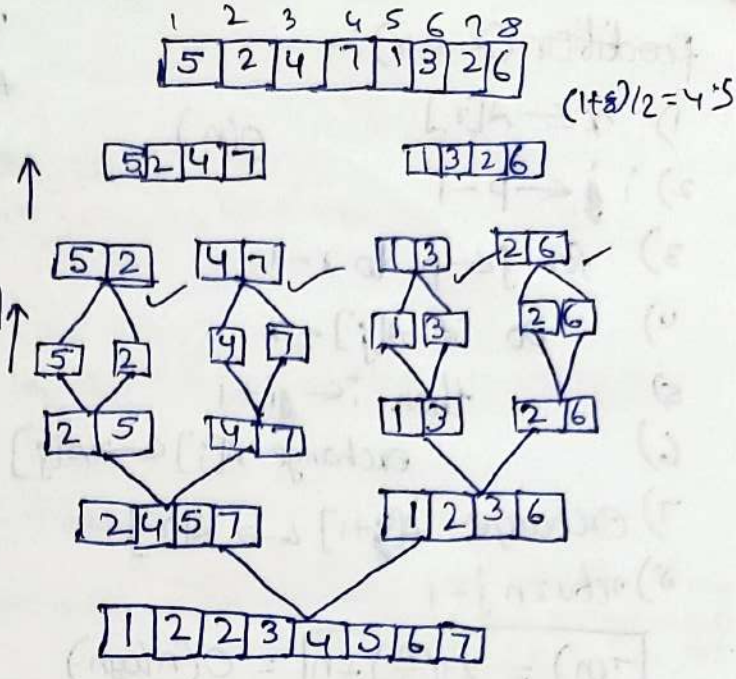
MERG SORT (A, P, q) ⟶ $T(\frac{n}{2})$

MERG SORT $(A, q+1, r)$ ⟶ $T(\frac{n}{2})$

MERGE (A, P, q, r) ⟶ $O(n)$

Merge algorithm:-

```
MERG (A,P,q,r)
1) n1 ← q-p+1
2) n2 ← r-q
3) Create array L[1....n1+1] and R[1--n2+1]
4) For i ← 1 ton1
5)    L[i] ← A[p+i-1]
6) for j ← 1 to n2
7)    R[j] ← A[q+j]
8) L[n1+1] ← ∞
9) R[n2+1] ← ∞
10) i ← 1
11) j ← 1
12) For k ← p tor
13) if L[i] ≤ R[j]
    than A[k] ← L[i]
         j ← k+1
    else A[k] ← R[j]
         j ← j+1
```



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

by master's theorem, $\Theta(n \log n)$

Best, worst, average case all are same

Quick sort

→ It is based on divide and conquer paradigm

divide:- $A[p \ldots r]$ into $A[p..q-1]$ and $A[q+1, r]$ such that

$A[p \ldots q-1] \le A[q] < A[q+1 \ldots r]$

conquer:- sort sub array recursively

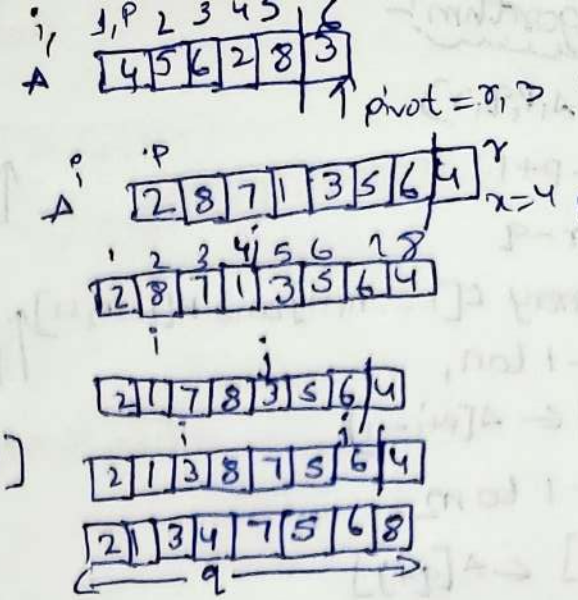
combine:- Entire array $A[p \ldots r]$ is now sorted.

Quick sort:- Quicksort(A,P,r)

```
1) if p<r
   than q ← partition(A,P,r) → O(n)
2)
3) Quicksort(A,P,q-1)  → T(n/2)
4) Quicksort(A,q+1,r)  → T(n/2)
```

partition(A, p, r):-

1) $x \leftarrow A[r]$ $O(n)$
2) $i \leftarrow p-1$
3) for $j \leftarrow p$ to $r-1$
4) do if $A[j] \le x$
5) then $i \leftarrow i+1$
6) exchange $A[i] \leftrightarrow A[j]$
7) exchange $A[i+1] \leftrightarrow A[r]$
8) return $j+1$



$\boxed{T(n) = 2T(\frac{n}{2}) + n} = O(n \log n)$

worst case $= T(n) = O(n-1) + O(0) + n = O(n-1) + 1 + n$
$= O(n-1) + n$
$= n^2$ $O(n^2)$

Largest sum contiguous subarray:-

Initialize:- max_so_far = 0
max_ending_here = 0

Loop for each element of the array
(a) max_ending_here = max_ending_here + a[i]
(b) if (max_ending_here < 0)
max_ending_here = 0
(c) if (max_so_far < max_ending_here)
max_so_far = max_ending_here
return max so far

both greedy & dp

Simple idea of Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this) And keep track of maximum sum contiguous segment among all positive segments (max_so_far is used for this)
Each time we get a positive sum compare it with max_so_far and update max_so_far if it is greater than max_so_far.

Example:- $\{-2, -3, 4, -1, -2, 1, 5, -3\}$

max_so_far = max_ending_here = 0

for i = 0, a[0] = -2

max_ending_here = max_ending_here + (-2)

set max_ending_here = 0 because max_ending_here < 0

for i = 1, a[1] = -3

max_ending_here = max_ending_here + (-3)

set max_ending_here = 0 because max_ending_here < 0

for i = 2, a[2] = 4

max_ending_here = max_ending_here + (4)

max_ending_here = 4

max_so_far is updated to 4 because max_ending_here greater than max_so_far which was 0

for i = 3, a[3] = -1

max_ending_here = max_ending_here + (-1)

max_ending_here = 3

for i = 4, a[4] = -2

max_ending_here = max_ending_here + (-2)

max_ending_here = 1

for i = 5, a[5] = 1

max_ending_here = max_ending_here + (1)

max_ending_here = 2

for i = 6, a[6] = 5

max_ending_here = max_ending_here + (5)

max_ending_here = 7

max_so_far is updated to 7 bcoz max_ending_here is greater than max_so_far

for i = 7, a[7] = -3

max_ending_here = max_ending_here + (-3)

max_ending_here = 4

→ Algorithm doesn't work for all negitive nos, it simply returns 0

Time complexity :- O(n)

Algorithmic paradigm :- Dynamic programming

Algorithm :-

```
int maxsubarray(int a[], int size)
{
  int max_so_far = 0;
  int max_end_here = 0;
  for (int i = 0; i < size; i++)
  {
    max_end_here = max_end_here + a[i];
    if (max_end_here < 0)
      max_end_here = 0;
    if (max_so_far < max_end_here)
      max_so_far = max_end_here;
  }
  return max_so_far;
}
```

use else if there for optimization

Problem statement :-

* Given a one dimensional array that may contain both positive and negative numbers (integers), find the sum of contiguous subarray of numbers which has the largest sum and return its sum.

Example :- Input :- [-2, -5, 6, 2, -3, 1, 5, -6]

Output :- 7

Explanation :- [6, -2, -3, 1, 5] has the largest sum which is 7

Algorithm formulation/Idea :-

a) Divide the given array in two halves.

b) Return the maximum of following three

1) Recursively calculate the maximum subarray in left half
2) Recursively calculate the maximum subarray in right half
3) Recursively calculate the maximum subarray sum such that the subarray crosses the midpoint.

(i) Find the maximum sum starting from midpoint and ending at some point on left of mid.

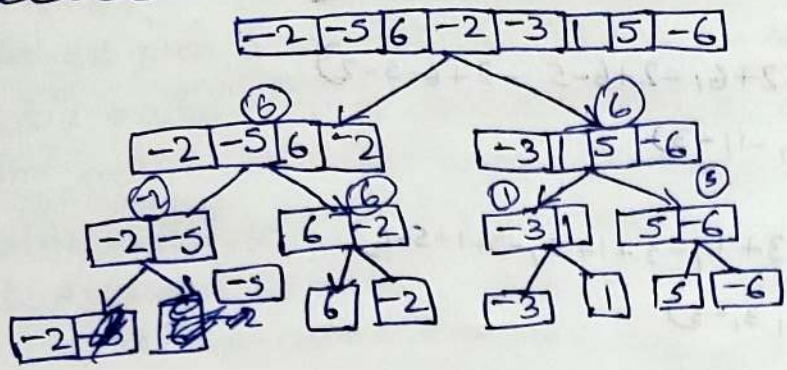
(ii) Find the maximum sum starting from mid+1 and ending with some point on right of mid+1

(iii) Finally combine the two and return.

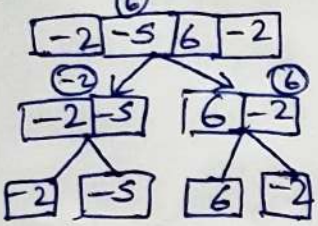
Pseudocode :-

```
Max-Subarray(A, left, Right)
    if (right == left)
        return(left, right, A[left])
    else mid = [(left + right)/2]
    l1 = Find-max_subarray(A, left, mid)
    r1 = Find-max_subarray(A, mid+1, right)
    m1 = Find-max-crossing-subarray(A, left, mid, right)
    if sum(l1) > sum(r1) and sum(l1) > sum(m1)
    return l1
    else if sum(r1) > sum(l1) and sum(r1) > sum(m1)
    return r1
    Else return m1
```

simulation / illustration:-



Left part:-

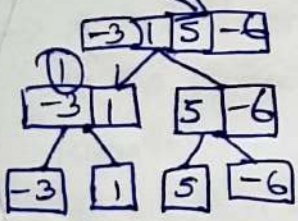


Calculations:-
pivot element = -2
left sum = -2
right sum = -5
Cross sum calculations:-
left sub sum = -2
right sub sum = -5
Cross sum = -2-5 = -7
max sum = max(left sum, right sum, cross sum) = -2

Calculations
pivot element = -5
left sum = -2
right sum = 6
Cross sum calculations:-
Left sub sum = max(-5, -5-2) = -5
right sub sum = max(6, 6-2) = 6
cross sum = -5+6 = 1
max sum = max(left sum, right sum, cross sum) = 6

→ total array part

Right part:-



Calculations:-
pivot element = -3
left sum = -3
right sum = 1
Cross sum calculations:-
left sub sum = -3
right sub sum = 1
Cross sum = -3+1 = -2
max sum = max(left sum, right sum, cross sum) = 1

Calculations
pivot element = 5
left sum = 5
right sum = -6
cross sum calculation:-
left sub sum = 5
right sub sum = -6
cross sum = 5-6 = -1
max sum = max(left sum, right sum, cross sum) = 5

Left part
Calculations:- pivot element = 6
left sum = 6
right sum = 6-2
cross calculation:-
left sub sum = 6
right sub sum = -2
Cross sum = 6-2 = 4
max sum = max(left sum, right sum, cross sum) = 6

Right part:-
Calculations:- pivot = 1
left sum = 1
right sum = 5
Cross sum calculation:-
left sub sum = max(1, 1-3) = 1
right sub sum = max(5, 5-6) = 5
cross sum = 1+5 = 6
max sum = max(left sum, right sum, cross sum) = 6

Cross sum calculation:-

-2	-5	6	-2	-3	1	5	-6

pivot element = -2

Left sub sum = max(-2, -2+6, -2+6-5, -2+6-5-2)

= max(-2, 4, -1, -3)

= 4

right sub sum = max(-3, -3+1, -3+1+5, -3+1+5-6)

= max(-3, -2, 3, -3)

= 3

cross sum = 4+3-7

max sum = max(left sum, right sum, cross sum)

= 7

So, the maximum subarray sum is 7.

Time complexity analysis:-

Find max-cross subarray takes : O(n) time

two recursive calls on input size n/2 take : 2T(n/2) time

Hence, T(n) = 2T(n/2) + O(n)

Here,

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

$= 2[2T(n/2^2) + n/2]$

$T(n) = 2^2 T(n/2^2) + n + n$

$= 2^2 [2T(n/2^3) + n/2^2] + 2n$

$= 2^3 T(n/2^3) + 3n$

$T(n) = 2^k T(n/2^k) + kn$

Use case / variations:-

→ Basis of efficient algorithms such as quick sort, merge sort

→ Multiplying large number (e.g. the karatsuba algorithm)

→ Finding the closest pair of points

→ syntactic analysis (e.g. top down parsers)

→ Computing the discrete fourier transform (FFT)

Alternative solutions of solving:-

* Brute force solution * Greedy algorithm * Kadane's algorithm (or)

Matrix chain multiplication :-

→ We are given a sequence (chain) $\{A_1, A_2, \ldots A_n\}$ of n matrices to be multiplied

→ Matrix multiplication is associative, and so all parenthesizations yield the same product.

$$A_{2\times3}\begin{vmatrix} a & b & c \\ d & e & f \end{vmatrix} \qquad B_{3\times2}\begin{vmatrix} g & h \\ i & j \\ k & l \end{vmatrix}$$

```
Matrix multiply (A,B)
if A.columns ≠ B.rows
    error "incompatible dimensions".
else let C be a new A.rows x B.columns matrix
    for i=1 to A.rows
        for j=1 to B.columns
            Cij = 0
            for k=1 to A.columns
                cij = cij + aik . bkj
return C
```

$$C_{2\times2}\begin{vmatrix} a\times g+b\times i+c\times k & a\times h+b\times j+c\times l \\ d\times g+e\times i+f\times k & d\times h+e\times j+f\times l \end{vmatrix}$$

→ We can multiply two matrices A and B only if they are compatible. The number of columns of A must equal the number of rows of B.

→ If $A = p\times q$ matrix and $B = q\times r$ matrix, the resulting matrix $C = p\times r$ matrix

→ A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parantheses.

→ If n = Number of matrices − 1

→ Number of distinct parenthesizations possible = $\frac{2nC_n}{n+1}$ [Catalan number]

→ For ex, $\{A_1, A_2, A_3, A_4\}$, then 5 distinct ways i.e $n = 4-1 = 3$

→ $(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, $((A_1(A_2A_3))A_4)$, $(((A_1A_2)A_3)A_4)$.

$$\frac{2\times3C_3}{3+1} = \frac{6C_3}{4} = \frac{6!}{3\times3!\times4!} = 5$$

→ Ex: consider the problem of a chain $\{A_1, A_2, A_3\}$ of three matrices. the dimensions of the matrices are 10×100, 100×5, and 5×50 respectively.

→ $(A_1A_2)A_3)$: $10*100*5 = 5000$ scalar multiplications to compute 10×5 matrix product A_1A_2 plus another $10*5*50 = 2500$ scalar multiplications to multiply this matrix by A_3, total of 7500.

→ we shall implement the tabular, bottom up method in the procedure MATRIXCHAIN-ORDER

→ This procedure assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$. Its input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$ where $p.length = n+1$

→ This procedure uses an auxiliary table $m[1\ldots n, 1\ldots n]$ for storing the $m[i,j]$ costs and another auxiliary table $s[1\ldots n-1, 2\ldots n]$ that records which index of k achieved the optimal cost in computing $m[i,j]$. We shall use table s to construct an optimal solution.

→ The m table uses only the upper triangle.

→ $$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Pseudocode:- $i \le k < j$

```
Matrix-chain-order(p)
    n = p.length - 1
    let m[1...n, 1...n] and s[1...n-1, 2...n] be new tables
    for i = 1 to n
        m[i,i] = 0
    for l = 2 to n        // l is the chain length
        for i = 1 to n-l+1
            j = i+l-1, m[i,j] = ∞
            for k = i to j-1
                q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
                if q < m[i,j]
                    m[i,j] = q, s[i,j] = k
    return m and s
```

→ The nested loop structure of matrix-chain-order yields a running time of $O(n^3)$ for the algorithm.

→ The algorithm requires $\Theta(n^2)$ space to store the m and s tables.

Ex:- matrix $A_1 \quad A_2 \quad A_3 \quad A_4$

dimension $3\times2 \quad 2\times4 \quad 4\times2 \quad 2\times5$

$p_0 \quad p_1 \quad p_2 \quad p_3 \quad p_4$

	1	2	3	4
1	0	24	28	58
2		0	16	36
3			0	40
4				0

(m)

	2	3	4
1	1	1	3
2		2	3
3			3

(s)

for which k value we got $m[1,2]$

$m[1,2] = i=1, j=2$ possible $k = 1 \le k < 2$ $\{m[1,1] + m[2,2] + P_0 \times P_1 \times P_2\} = 3+2+4 = 24$

$m[2,3] = 2+4+2 = 16$

$m[3,4] = 4+2+5 = 40$

$m[1,3] = \min \{k=1 : m[1,1] + m[2,3] + P_0 P_1 P_3\} = 0+16+3\times2\times2 = 28$ }28
$k=2 : m[1,2] + m[3,3] + P_0 P_2 P_3\} = 24+0+3*4*2 = 48$ }

$m[2,4] = \min \{k=2 : m[2,2] + m[3,4] + P_1 P_2 P_4 = 0+40+2+4+5 = 86$ }
$k=3 : m[2,3] + m[4,4] + P_1 P_3 P_4 = 16+0+2\times2+5 = 36$ } = 36

$m[1,4] = \min \{k=1 : m[1,1] + m[2,4] + P_0 P_1 P_4 = 0+36+3+2+5 = 66$ }
$k=2 : m[1,2] + m[3,4] + P_0 P_2 P_4 = 24+40+3\times4\times5 = 124$ } 58
$k=3 : m[1,3] + m[4,4] + P_0 P_3 P_4 = 28+0+3\times2\times5 = 58$ }

	1	2 (j)	3	4	m
1	0	24	28	58	1
2		0	16	36	2
3			0	40	3
4				0	4

	2	3 (j)	4	s
1	1	1	3	1
2		2	3	2 (i)
3			3	3

$((A_1 (A_2 A_3))A_4)$

→ bag of maximum weight.

0/1 Knapsack problem :-

→ We are given n no. of objects with weights w_i and profits P_i where i varies from 1 to n and also a knapsack with capacity M.

→ The problem is, we have to fill the bag with the help of n objects and the resulting profit to be maximum.

→ This problem is similar to ordinary knapsack problem but we may not take a fraction of a object

maximize $\sum_1^n P_i x_i$

subjected to $\sum_1^n w_i x_i \le M$

where $1 \le i \le n$ and $x_i = 0$ or 1.

Step 1 :- * Let s^i be a pair of (P,w) where P is profit and w is weight of an object

* initially $s^0 = \{(0,0)\}$

→ Compute $s^{i+1} = \{$merge s^i and $s^i\}$

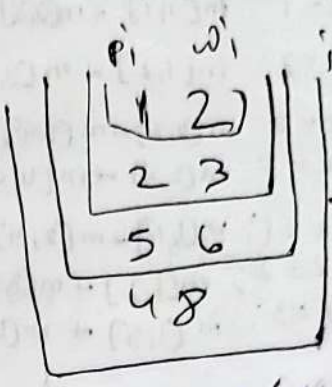
~~Longest common subsequence~~:-

~~→ A substring is a contiguous sequence of characters within a string~~

Step 2:- Generate a sequence of decisions using the following formula,

$(P_1, P_2, P_3) = (1,4,5)$

$(w_1, w_2, w_3) = (2,3,4)$ $m=6, n=3$



if w_i is under bag capacity from 2nd step consider the sum of w_i at 1st & w_i at next i.e $w_i = 2+3$ in 2nd step

$S^0 = (P_1, w_1) = (1,2)$

$S_1^0 = (0+1, 0+2) = (1,2)$

$S^1 = \{(0,0)\ (1,2)\}$

Tabular method:- $P = \{1,4,5,6\}$ $w = \{2,3,4,8\}$ Capacity $m=8$, $n=4$

weight of the bag check it value can fill if can't copy above cell value

no of objects

w → weight

Pi	wi	v / w	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5											
6	4	3	0	0	1	2	5	5	6	7	7
	5	4	0	0	1	2	5	6	6	7	8

here we added $w_i + w_{i-1}$ i.e $2+3=5$ so from the previous row will be $1+3=3$.

in this row as w_1 is 2 it can't placed in w 1 so from above ring fill the profit value.

→ for empty boxes check upper and left values and fill

w[4,8]

$= \max\{V(3,8), V[3,8-5]+4\}$

$V(0,(1,2))$

$V[i,w] = \max(V[i-1,w], V[i-1, w-w[i]] + P[i])$

$V[4,5] = \max(V[3,5], V[3,5-5]+6)$

$= \max(5, 0+6)$

x_1	x_2	x_3	x_4
0	1	0	1

we get max capacity 8 at $v=4$, $\therefore x_4 = 1$

and perform m − Pi value of $v=4$, $8-6=2$

from where 2 is starting in table take that value i.e

$x_2 = 1$

Pseudocode:-

```
int knapsack(int m, int wt[], int val[], int n)
{
 int i, w;
 int K[n+1][m+1];
 for(i=0; i<=n; i++)
 {
```

```
for (w=0; w<=m; w++)
{
  if (i==0 || w==0)
      k[i][w] = 0;
  else if (wt[i] <= w)
      k[i][w] = max(val[i] + k[i-1][w-wt[i]], k[i-1][w]);
  else
      k[i][w] = k[i-1][w];
}
return k[n][m]
```

set method:- example :- m=6

$(w_1, w_2, w_3) = (2,3,4)$

$(P_1, P_2, P_3) = (1,2,5)$

$S^0 = \{(0,0)\}$ initially no weight and profit

$S_1^0 = \{(1,2)\}$; S^1 = merge of S^0 and S_1^0

$= \{(0,0)\ (1,2)\}$

$S_1^1 = \{(2,3)\ (3,5)\}$

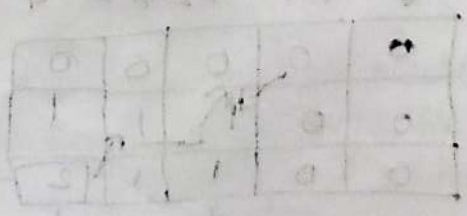
$S^2 = \{(0,0)\ (1,2)\ (2,3)\ (3,5)\}$

$S_1^2 = \{(5,4)\ (6,6)\ (7,7)\ (8,9)\}$ → discard

↳ discard more than max cap

$S^3 = \{(0,0), (1,2)\ (2,3)\ (5,4)\ (6,6)\}$

Dynamic programming follows principle of optimality.



Longest common subsequence (LCS):-

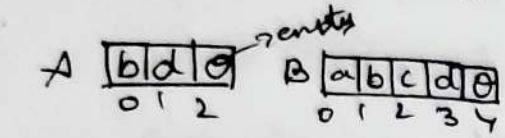
string 1: a b c d e f g h i j
string 2: c d g i

cdgi is longest common subsequence

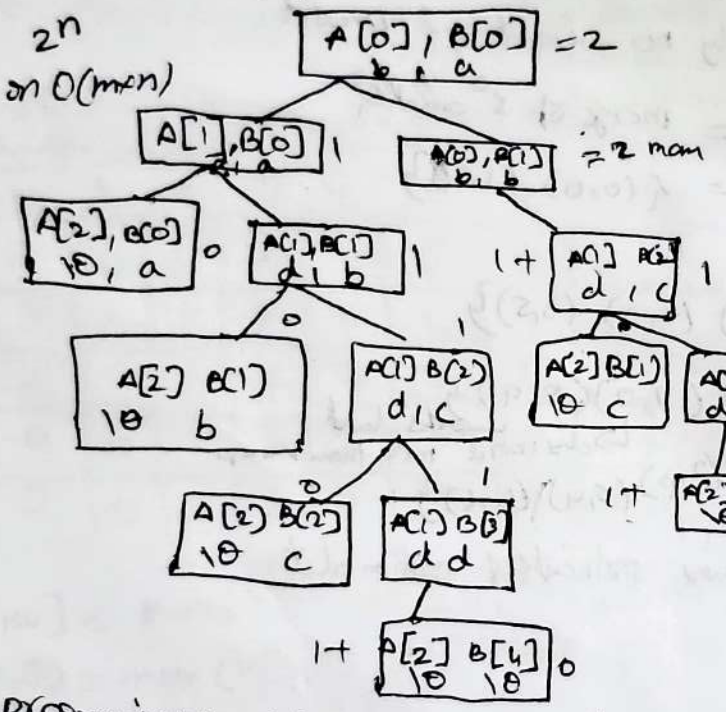
string 1: a b d a c e b a c e
string 2: b d a b c e a b c e

string 1: a b c d e f g h i j
string 2: e c d g i

'c' and 'd' we shouldn't take bcoz they are intersecting
if we take from 'e' we get "egi"
if we start from 'c' we get "cdgi" (LCS)



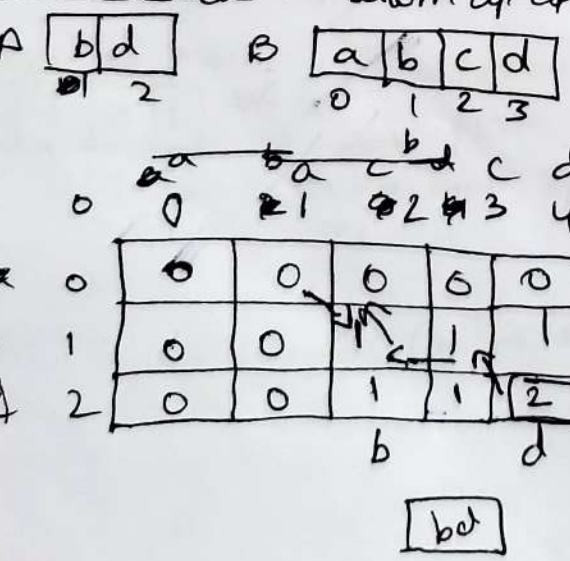
recursive 2^n
memorization $O(mn)$



Pseudocode:-

```
int LCS(i,j)
{ if (A[i]=='\0' || B[j]=='\0')
    return 0;
  else if (A[i]==B[j])
    return (1+LCS(i+1, j+1));
  else return max(LCS(i+1,j),
                  LCS(i,j+1));
```

Dynamic programming:- bottom up approach



Pseudo code

```
if(A[i]=B[j])
  LCS[i,j] = 1+LCS[i-1,j-1]
else
  LCS[i,j] = max(LCS[i-1,j],
                 LCS[i,j-1])
```

1 -> if match
i.e 1+ diagonal ele

-> size of longest subsequence

$\Theta(m \times n)$

bd

ex:- str1 :- stone
str2 :- longest

T.C : O(m×n)
m is rows n is colums

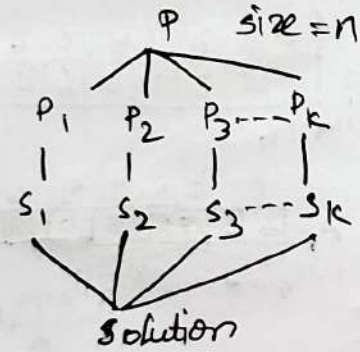
		L	O	N	G	E	S	T
	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
S 1	0	0	0	0	0	0	1	1
t 2	0	0	0	0	0	0	1	2
O 3	0	0	1	1	1	1	1	2
N 4	0	0	1	2	2	2	2	2
E 5	0	0	1	2	2	3	3	3

size of LCS

O N E

∴ The longest common subsequence is (one)$_3$

Divide and conquer :-



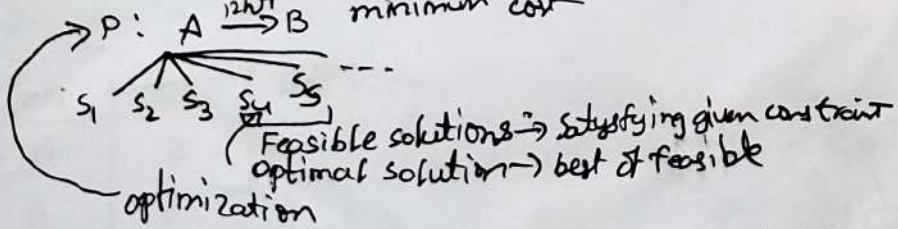
recursive in nature

pseudo code :-

```
DAC (P)
{
 if (small(P))
 {
   S(P);   -----> solve it directly
 }
 else
 {
   divide P into P1, P2, P3 -- Pk
   Apply DAC(P1), DAC(P2), ---
   Combine (DAC(P1), DAC(P2), ---)
 }
}
```

applications :-
1) Binary search
2) Finding max and min
3) merge sort
4) quick sort
5) strassen's matrix multiplication

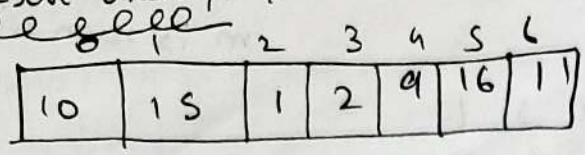
Greedy method :-



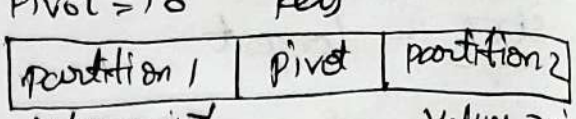
Greedy, Dynamic programming, Branch & bound
optimization problems

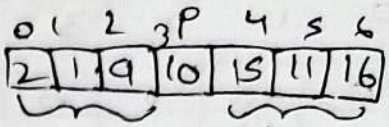
```
Algorithm Greedy (a, n)
{
  for i = 1 to n do
  {
    x = select (a);
    if Feasible (x) then
        solution = solution + x;
  }
}
```

Quicksort example:-

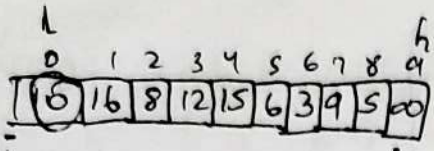


Pivot = 10 key



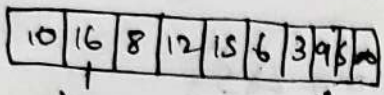


0-2 & 4-6 are 2 arrays

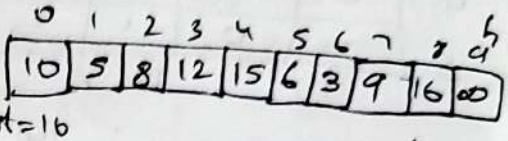


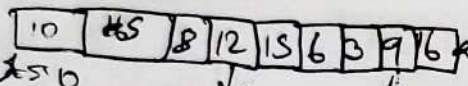
i searches for element greater than pivot

j searches for element greater than pivot and decrement

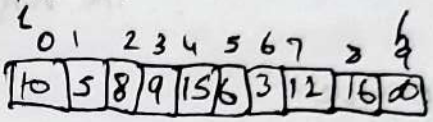


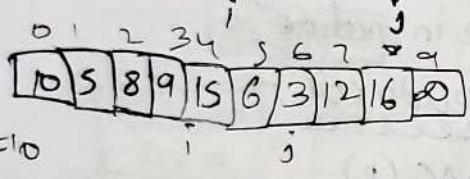
swap(A[i],A[j]) A



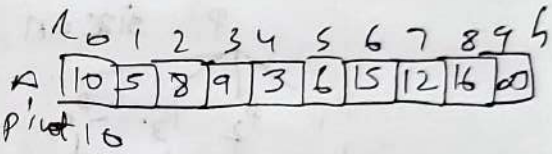


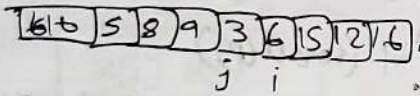
swap A



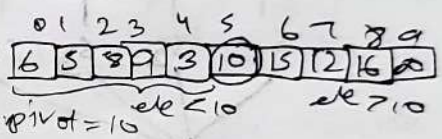


swap





do swap



when j<i swap j with pivot

Continue same for these sub arrays

```
Partition(l,h)
{
  pivot = A[l];
  i = l; j = h;
do
{ i++;
 } while (A[i] ≤ pivot);
do
{
  j--;
} while (A[j] > pivot)
  if (i<j)
      swap(A[i], A[j]);
}
  swap(A[l], A[j]);
  return j;
```

Huffman coding

message - BCCABBDDAECCBBAEDDCC

length - 20

ASCII - 8 bit

A - 65 - 01000001 } Binary

B - 66 - 01000010

C - 67

D - 68

E - 69

size of the message = 8*20 = 160 bits

character	count/frequency		code
A	3	3/20	000
B	5	5/20	001
C	6	6/20	010
D	4	4/20	011
E	2	2/20	100
	20		

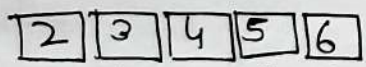
5*8 bit (↑ characters) — 5*3 (↑ codes)

40 + 15 = 55

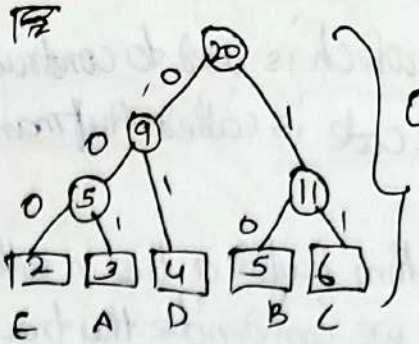
message - 60 bits

table - 55 bits

115 bits } size reduced



char	count	code	
A	3	001	by optimal merge tree
B	5	10	
C	6	11	
D	4	01	
E	2	000	
	20		



on the left hand side edges write 0s

on the right hand side edges write 1s

→ for each alphabet follow from root onwards.

10 10 01

BCCABBDDAECCBBAEDDCC

11 11 001 10 01001

char	count	code	
A	3	001	3+3=9
B	5	10	5+2=10
C	6	11	6+2=12
D	4	01	4+2=8
E	2	000	2+3=6
5*8 bit	20	12 bits	45 bits

40 bits msg 45 bits table - 52 bits

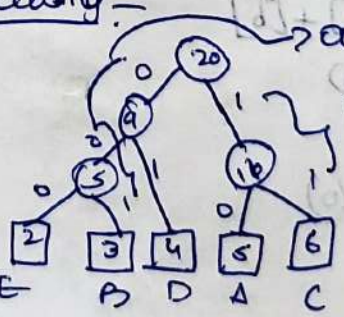
total - 97

$\sum a_i * f_i = 3*2 + 3*3 + 2*4 + 2*5 + 2*6$

= 45 bit

shortcut to find size directly from tree

Decoding :-



001 - B

11 - C

So from binary we can get the message

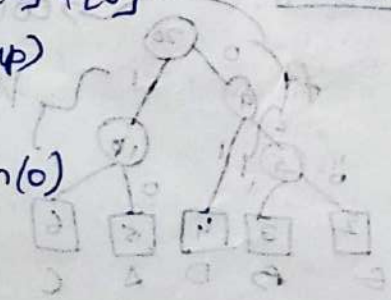
Algorithm:-

The method which is used to construct optimal prefix code is called Huffman Coding

→ This algorithm builds a tree in bottom up manner. We can denote this tree by T. Let |c| be number of leaves.

|c| - 1 are number of operations required to merge the nodes. Q be the priority queue which can be used while constructing binary heap.

```
Algorithm Huffman(c)
{
  n = |c|
  Q = c
  for i ← 1 to n-1
  do
  {
   temp ← get node()
  left[temp] Get_min(2) right [temp] Get min(a)
  a = left[temp] b = right[temp]
  F[temp] ← f[a] + [b]
  insert (Q, temp)
  }
return Get_min(0)
}
```



simple words:-

→ Create a priority queue Q consisting of each unique characters

→ Sort them in ascending order of their frequencies

for all the unique characters:

create a newNode

→ extract minimum value from Q and assign it to left child of newNode

→ extract minimum value from Q and assign it to right child of newNode

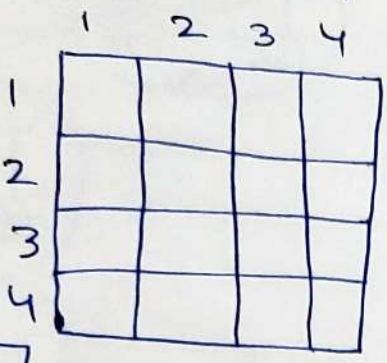
→ Calculate the sum of these two minimum values and assign it to the value of newNode insert this newNode into the three

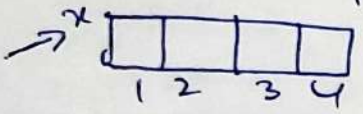
return rootNode.

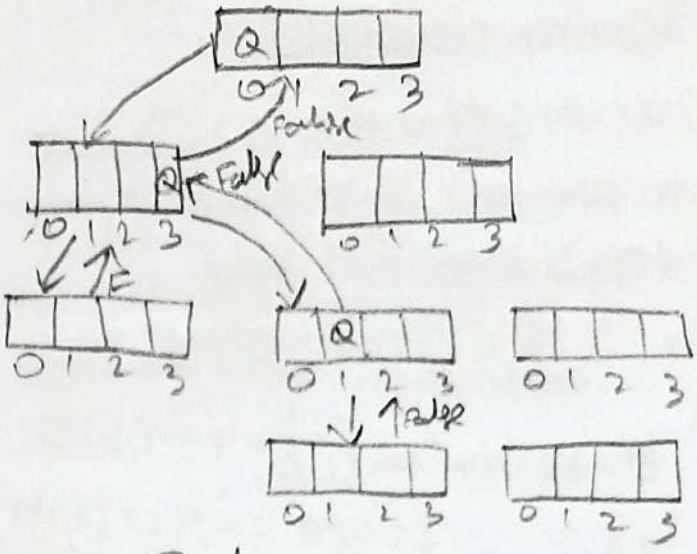
N-Queens

Q_1, Q_2, Q_3, Q_4

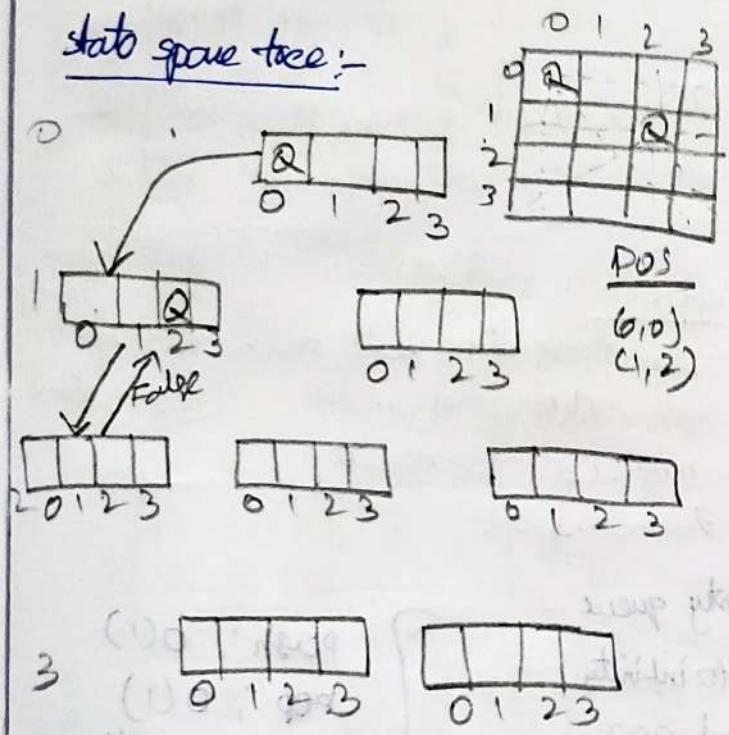
Same
- → row
- → column
- → diagonal



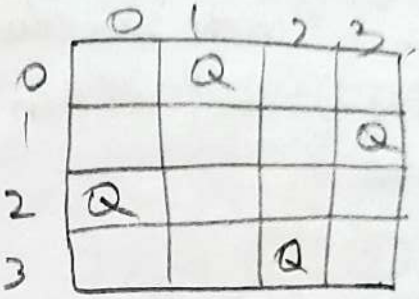


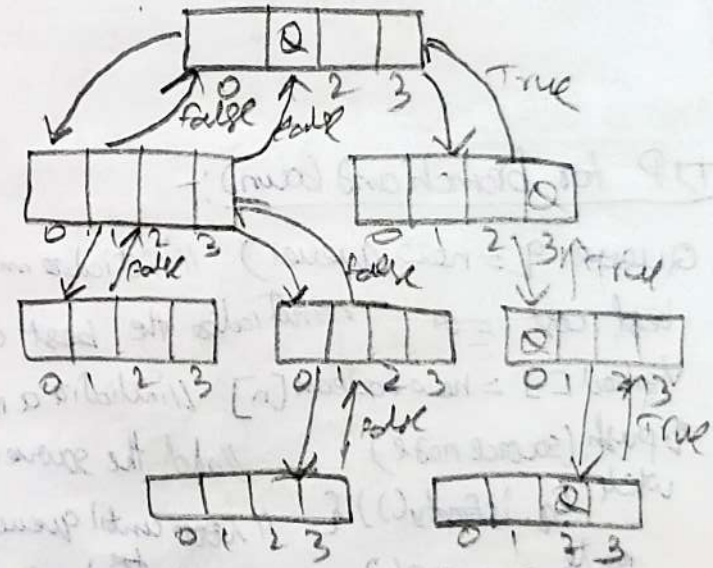


state space tree :-

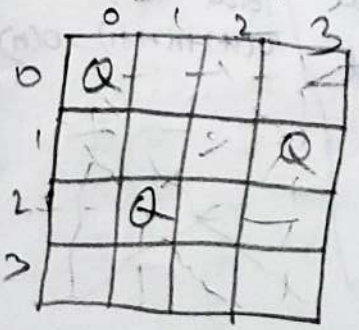


pos
(0,0)
(1,2)





→ False represents no position for a queen can be find so return to previous state and change position of previous queen.



pos
(0,0)
(1,3)
(2,1)

Algorithm :- Nqueens(k, n)

```
{
for i = 1 to n do
{
if place(k, i) then
{
x[k] = i
if (k = n) then write (x[1 : n]);
else Nqueens(k+1, n);
}
}
}
```

Algorithm place(k,i)
// returns true if 8 is placed at
kth row and ith column
// otherwise returns false
// x[] is a global array

```
{ for j:=1 to (k-1) do
  { if((x[j]=i) // in the same column
    or(Abs(x[j]-i)=Abs(j-k)) then
    return false //
  }
  return true
}
```

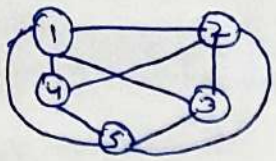
Simple words :-

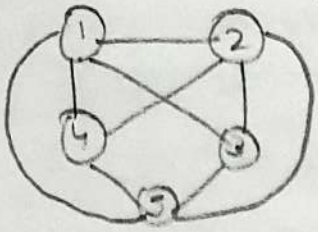
1) P for branch and bound :-

```
Queue q=new queue() //initialize an empty queue
best cost =∞  //initialize the best cost to infinity
visited[] = new boolean[n] //initialize a visited array
q.push(source node)  //add the source node to queue
while(!q.isEmpty()) { // loop until queue is empty
  node = q.pop() //get the next node from the queue
  visited[node] =true // mark node as visited
  Cost = calculate cost(node) //calculate the cost of visiting the node
if (cost < best cost) { //check if the cost is better than best cost
    best cost =cost //update the best cost
for(i=0;i<n;i++){ //check if the node has not been visited
if(!visited[i]) { //add the node to queue.
  q.push(i)
}}}
return best cost  //return the best cost
```

push : O(1)
pop : O(1)
Iterate through the nodes : O(n)
Calculate cost: O(1)
check if node has been visited : O(1)
Total tc:
O(1+1+n+1+1) = O(n)

Travelling Salesperson branch-n-bound:-





	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞

reducing the matrix, take minimum value in each row and subtract remaining values by that value

	1	2	3	4	5	
1	∞	10	20	0	1	10
2	13	∞	14	2	0	2
3	1	3	∞	0	2	2
4	16	3	15	∞	0	3
5	12	0	3	12	∞	4

minimum value of each row before

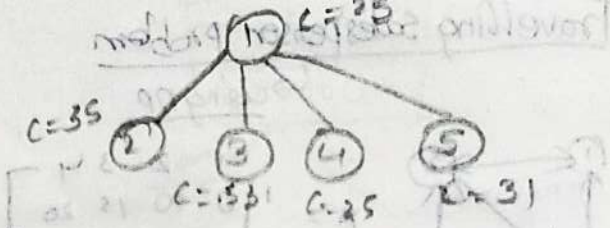
21

reduce the matrix, using minimum value in column now

	1	2	3	4	5	
1	∞	10	17	0	1	10
2	12	∞	11	2	0	2
3	0	3	∞	0	2	2
4	15	3	12	∞	0	3
5	11	0	0	12	∞	4
	0	3	0	0		

minimal of column

21+4=25



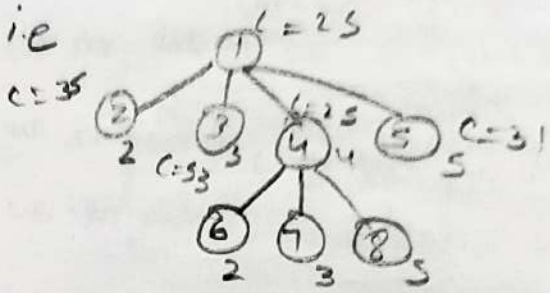
for finding (1,2) cost make 1st row and 2nd column elements as ∞. if all are zeroes it is reduced.

$c(1,2) + r + \hat{r} = 25 + 10 + 0 = 35$

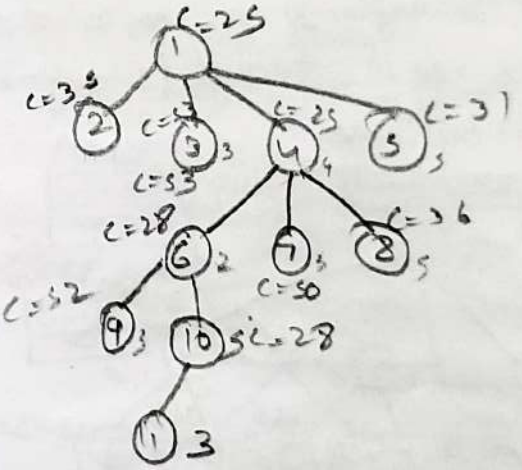
$c(1,3) + r + \hat{r} = 17 + 25 + 11 = 53$

here cost at node 4 is minimum.

i.e



$c(4,2) = c(1) + \hat{r} = 3 + 25 + 0 = 28$



1 - 4 - 2 - 5 - 3 - 1

minimum cost tour.

for (1,2), make 1st row and 2nd column as infinite and position (2,1) as infinite

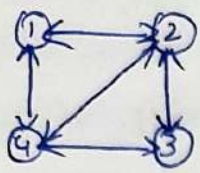
∞	∞	∞	∞	∞
∞	∞	11	2	0
0	∞	∞	0	2
15	∞	12	∞	0
11	∞	0	12	∞

is reduced, ∴ r = 0 → from reduced

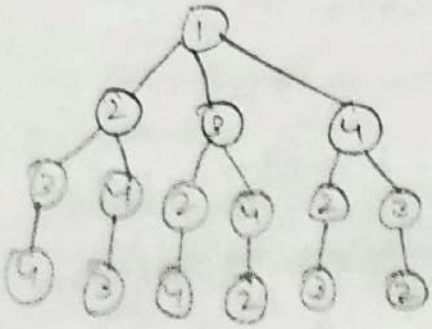
$c(1,2) = c + r + \hat{r} = 25 + 10 + 0 = 35$

Travelling salesperson problem

↳ using DP



$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix} \end{array}$$



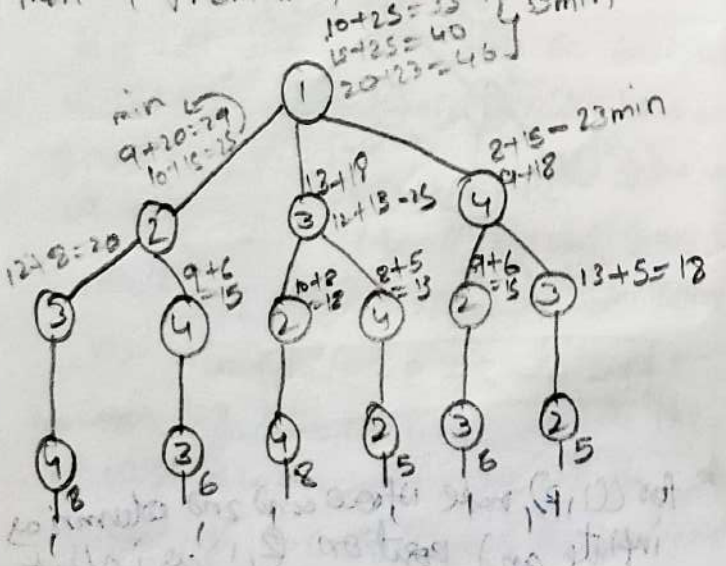
by using bottom up approach

4 to 1 — cost 8

3 to 1 — cost 6

2 to 1 — cost 5

if he is in vertex 3 and has to return to 1, he has do travel to 4 and then 1 from there



$$g(1,\{2,3,4\}) = \min\{c_{1k} + g(k,\{2,3,4\} - \{k\})\}$$

$$g(i,S) = \min\{c_{ik} + g(k, S - \{k\})\}$$

using formula,

$g(2,\phi) = 5$; $g(3,\phi) = 6$

$g(4,\phi) = 8$

$g(2,\{3,4\}) = 15$ $g(3,\{4\}) = 8$

$g(3,\{2\}) = 5$ $g(3,\{4\}) = 8$

$g(4,\{2,3\}) = 5$, $g(4,\{3\}) = 6$

$g(2,\{3,4\}) = 25$ $g(3,\{2,4\}) = 25$

$g(4,\{2,3\}) = 23$

$$g(1,\{2,3,4\}) = \min\{c_{12} + g(2,\{3,4\}),\ c_{13} + g(3,\{2,4\}),\ c_{14} + g(4,\{2,3\})\}$$

$$= 35$$

Algorithm:-

```
Data: s: starting point; N: a subset of input cities; dist(): distance among the cities
Result: Cost: TSP result
visited[N] = 0;
cost = 0;
procedure TSP(N, S)
    visited[s] = 1;
if |N| = 2 and K ≠ s then
    cost(N,K) = dist(s,K);
    Return cost;
else
  for j ∈ N do
    for i ∈ N and visited[i] = 0 do
      if j ≠ i and j ≠ s then
        cost(N,j) = min(TSP(N - {i}, i) + dist(j,i))
        visited[j] = 1;
      end
    end
  end
end
return cost;
end
```

TC:- In DP for TSP the no. of possible subsets can be at most $N \times 2^N$. Each subset can be solved in $O(N)$ times. Therefore, the TC is $O(N^2 \times 2^N)$

0/1 Knapsack using branch and bound :-

We will use LC-BB

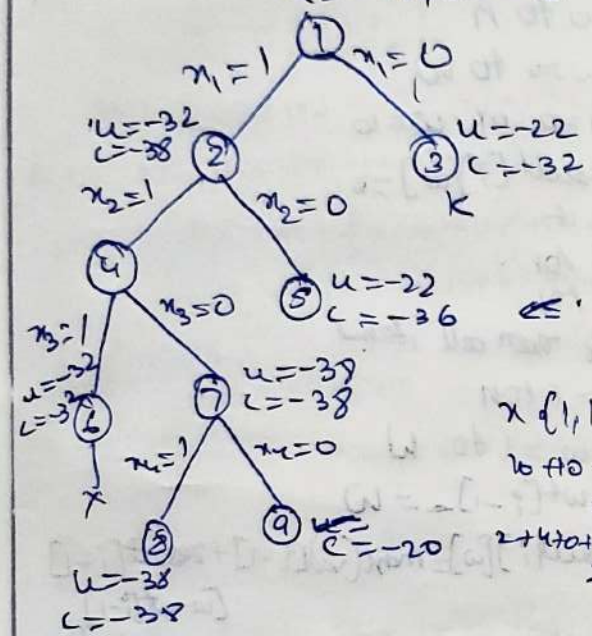
profit	10	10	12	18
weight	2	4	6	9

$m = 15, n = 4,$

$u = \sum_{i=1}^{n} p_i x_i$

$c = \sum_{i=1}^{n} p_i x_i$ (with fraction)

$S = \{x_1, x_3\}$, $S = \{1, 0, 1\}$



$x\{1, 1, 0, 1\}$

10 + 10 + 18 = 38 max profit

2 + 4 + 9 = 15

for node ① finding cost

$c = 10 + 10 + 12 + \frac{18}{9} \times 3$

2 4 6

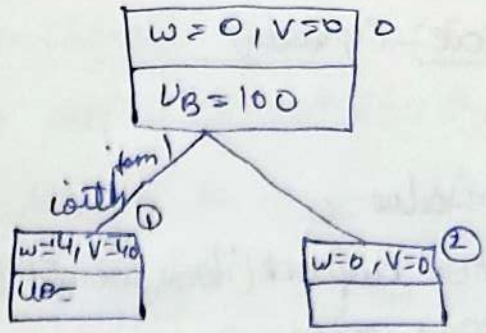
12 but $m = 15$ so 3 remaining multiply

Illustration-2

Item	weight	value	v_i/w_i
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

$u_B = v + (w - w)(v_{i+1}/w_{i+1})$ bag capacity

$u_B = 0 + (10 - 0) \times 10 = 10 \times 10 = 100$



UB for node ①, $UB = 40 + (10 - 4) \times 6$

$= 40 + 6 \times 6 = 76$

UB for node ②, $UB = 0 + (10 - 0)\ 6$

$= 10 \times 6 = 60$

UB for node ③,

$UB = 40 + (10 - 4) \times 5$

$= 40 + 6 \times 5 = 40 + 30 = 70$

UB for node ⑤, $UB = 65 + (10 - 9) \times 4$

$= 65 + 1 \times 4 = 69$

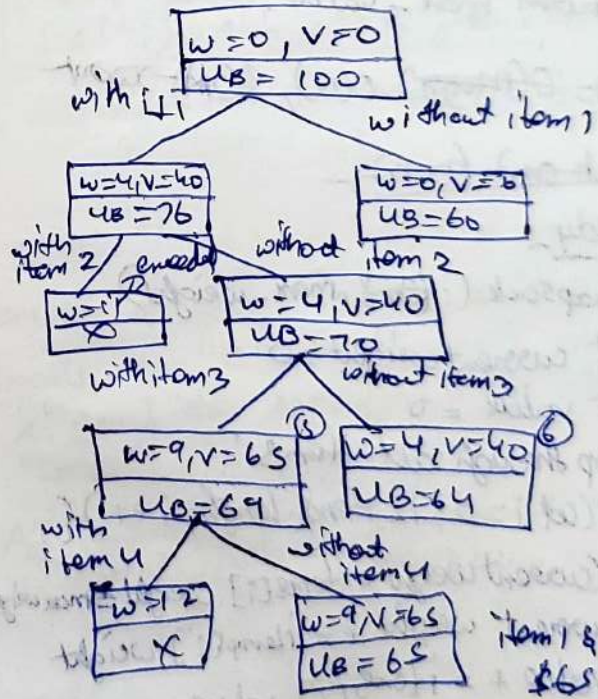
UB for node ⑥, $UB = 40 + (10 - 4) \times 4$

$= 40 + 6 \times 4 = 64$

UB for node ⑦, $UB = 65 + (10 - 9) \times 0$

$= 65 + 0 = 65$

item 1 & item 3 $65



Pseudocode:- (greedy)

Begin

→ max_value = 0

function knapsack (items, weight_limit)

current_weight = 0

current_value = 0

for items in items

if (current_weight + item_weight ≤ weight_limit)

current_weight += item.weight

current_value += item.value

if (current_value > max value)

max_value = current_value)

if (items is not empty): //recursively call knapsack

knapsack (items without the first item, weight_limit)

//END

//return the max_value

return max_value.

~~T(1) = O(n log n)~~ O(N), O(2^N) ~~worst~~

~~branch and bound~~:-

greedy:-

knapsack (items, max_weight)

let current_weight = 0

let value = 0

//loop through all items

for (let i = 0; i < items.length; i++){

if (current weight + items[i].weight ≤ max weight){

current weight += items[i].weight

value += items[i].value

}

}

return value O(n)

b & b :-

knapsack (wt, val, n)

W = knapsack capacity

wt = array of item weights

val = array of item values

n = no. of items

//Create an array to store results of subproblems result[0-n, 0-w]

//Initialize a 2D array result[][]

for i = 0 to n

for w = 0 to W

if i = 0 or w = 0

result[i][w] = 0

End if

End for

End for

//Iterate over all items

for i = 1 to n

for w = 1 to W

if wt[i-1] <= W

result[i][w] = max(val[i-1] + result[i-1][w-wt[i-1]], result[i-1][w])

else

result[i][w] = result[i-1][w]

end if

end for

end for

//return the maximum value in the knapsack.

return result[n][w]

END Function

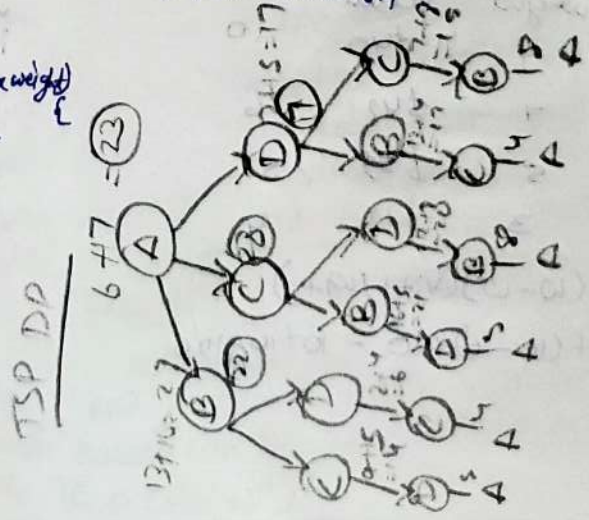


Illustration :-

n=7 objects 0 1 2 3 4 5 6 7

m=15

profits P	10	5	15	7	6	18	3
weights w	2	3	5	7	1	4	

This is optimization, maximization problem

profit / weight 5, 1.3, 3, 1, 6, 4.5, 3

consider highest profit/weight ratio value

15kg	15-1=14	5, 1.3, 3, 1, 6, 4.5, 3 selected
	14-2=12	1 1 1 1 1 1
	12-4=8	1 2/3 1 0 1 1
	8-5=3	
	3-1=2	
	2-2=0 ✓	

$1\times2+\frac{2}{3}\times3+1\times5+0+7+1\times1+1\times4+1\times1$.

so, $x = 2+2+5+0+1+4+1 = 15$

for profits, $7+0+\frac{2}{3}\times5+1+15+1\times(7)\times18+1\times3$

$= 10+2+1.3+15+6+18+3 = 54.6$

constraint :- $\sum x_i w_i \le m$

objective :- max $\sum x_i P_i$

coin change :- (POC)

The proof of correctness for the above is

Let A(i,j) be the no. of ways to make change for j amount of money using coins 0 to coins [i], we can prove that A(i,j) is correct by induction on i.

Base case :- when i=0, A(0,j) is the no. of ways to make change for j amount of money using only coins [0]. we know that if coin[0] > j, then A(0,j) = 0. If coin[0] == j then A(0,j) = 1, since the only way to make change for j is to use the coin coins[0] j times.

Inductive step :- Assume A(i-1, j) is correct we need to prove that A(i,j) is also correct A(i,j) is the no of ways to make change for j amount of money using coin[0] to coin[i]. if coin[i] > j, then only way to make change is to use coin(0) to coins[i-1], which is A(i-1,j)

∴ A(i,j) = A(i-1, j) + A(i, j - coin[i]).

Proof of correctness :-

Initialization :- For i=0, the invariant is respected; the sub-array A[1...0] is sorted, trivially (it contains no element).

Maintenance :- Given the subarray A[1...n-1] sorted. Iteration n inserts at position n the smallest of the remaining unsorted elements of A[n... A.size], as computed by the ; loop. A[1... n-1] contains only elements smaller than A[n... size], and A[n] is smaller than any element in A[n+1, ... A. size], then A[1...n] is sorted and the invariant is preserved.

Termination :- At the last iteration, A[1, ..., A. size, -1] is sorted, and all elements in A[A.size-1, ... A.size] are larger than elements in A[1... A.size-1]. Hence A[1...A.size] is sorted.

String matching Illustration:-

1) Naive Algorithm:-

Algorithm:-

→ It is the simplest method which uses the brute force approach.

→ It is a straight forward approach of solving the problem.

→ it compares first character of pattern with searchable text. If match is found, pointers in both strings are advanced. If match not found, pointer of text is incremented & pointer of pattern is reset. This process is repeated until end of the text.

→ It doesn't require any pre-processing. It directly starts comparing both strings character by character.

→ $TC = O(m * (n-m))$

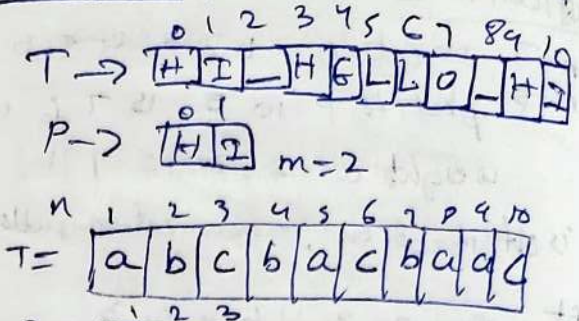
```
Algorithm Naive(T,P)
for i ← 0 to n-m do
if P[1...m] = = T[i+1...i+m] then
print "match found".
end
end.
```

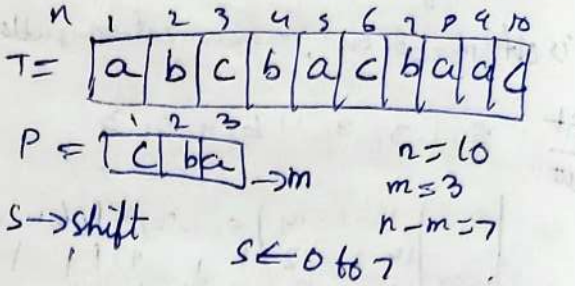
→ The time complexity of the algorithm depends on the length of the text string n and the length of the pattern string m.

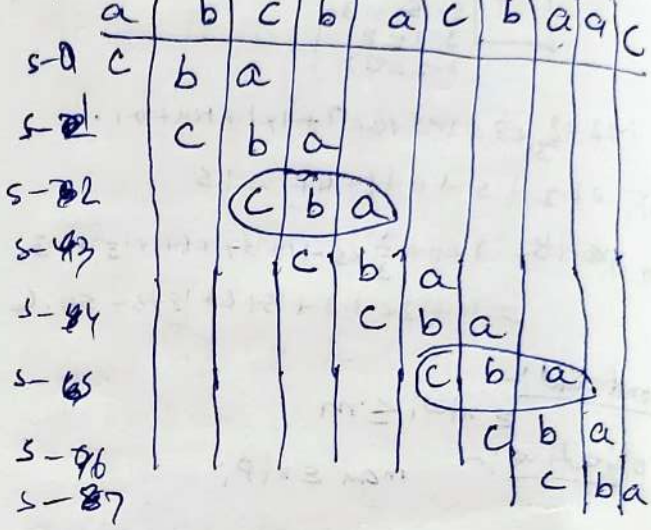
→ In the worst case, the algorithm needs to compare all m characters of the pattern with all n characters of the text, for every possible starting position in the text.

∴ The worst case complexity is $O(nm)$

However, if the pattern is much shorter than the text (i.e $m < n$) the time comp can be closer to $O(n)$, as the no. of iterations in the loop is proportional to $n-m+1$.







s-0
a = c [false]

s-1
b = c [false]

s-2
c = c [True]
b = b [True]
a = a [True]

s-3
b = c [false]

s-4 a = c [False]

s-5
c = c [True]
b = b [True]
a = a [True]

s-6
b = c [False]

s-7 :- a = c [false]

2) Rabin Karp :-

It is based on hashing technique.

→ Rabin karp algorithm is a string matching algorithm, that uses a "block of characters" for comparing a pattern P with text T.

The key idea of rabin karp algorithm is to proun the given text "T" to a sequence of integer. similarly pattern P is also converted to an integer.

① Successful Hit :- One can find the location of an exact pattern "P" in the text "T".

② Unsuccessful hit :- Here search fails completely

③ Spurious hit :- Here, a match between the hash code and the presence of a pattern is reported. However the report is generated is incorrect since the hash fn value is same for multiple string and the characters of the pattern do not match with the text.

Pseudocode :-

```
n = t.length
m = p.length
h = d^(m-1) mod q
p = 0
t0 = 0
for i = 1 to m
    p = (d*p + P[i]) mod q
    t0 = (d*t0 + t[i]) mod q
for s = 0 to n-m
    if p = ts
        if P[1..m] = t[s+1..s+m]
            print "pattern found at position" s.
    if s < n-m
        ts+1 = (d(ts - t[s+1]h) + t[s+m+1]) mod q
```

Illustration :-

Text : c c a c c a a e d b a
(2→3 as 6 to 8 a to 10) match

a - 1
b - 2
c - 3
d - 4
e - 5

pattern : d b a Hash value = 4+2+1 = 7
m = 3

if in case of spurious use hash function

$4*10^2 + 2*10^1 + 1*10^0 = 421$

$P[1]*10^{m-1} + P[2]*10^{m-2} + P[3]*10^{m-3}$

for c c a → $3*10^2 + 3*10^1 + 1*1^0 = 331$ ✗

second, c a c → $3*10^2 + 1*10^1 * 3*10^0$

$300 + 10 + 3 = 313$

Ex :- T = a a a a a b

P = a a b → Hash fn = 4

① a a a → 1+1+1 = 3 (3 ≠ 4)

② a a a a b → 1+1+3 (3 ≠ 4)

a a a b → 1+1+1 (3 ≠ 4)

a a b → (1+1+2 = 4) ✓

If In order to reduce the complexity and under the spurious hit → to use complex hash function,

as a result it complexity is O(n-m+1)

Time complexity :-

The rabin-karp algorithm is a string matching algorithm that uses hashing to compare a pattern with multiple substrings in a text. It has an average case time complexity of O(n+m), where n is the length of the text and m is the length of the pattern.

The hash function used in the rabin-karp algorithm should have following properties :

→ It should be computable in O(1) time.

→ It should produce the same hash value for two strings if and only if they are the same.

→ It should produce different hash values for most strings.

If a good hash function is used, the expected running time of the rabin-karp algorithm is $O(n+m)$. However in the worst case, when the hash function produces many collisions, the running time can be $O(nm)$.

Overall, the rabin-karp algorithm is a good choice for string matching when the pattern length is small relative to the text length, and a good hash function.

knuth morris pratt

KMP is an algorithm, which checks the characters from left to right. When a pattern has a sub-pattern appears more than one in a sub-pattern, it uses that property to improve the time complexity, also for in the worst case

Pseudocode:-

~~Compute-prefix-function(P)~~

~~1. m ← length[P]~~

~~2.~~ prefix(pattern, m, pref array)

```
Begin
  length := 0
  pref array[0] := 0
  for all characters index i of pattern, do
  if pattern[i] = pattern[length], then
     increase length by 1
     prefArray[i] := length
  else
     if length ≠ 0 then
        length := prefArray[length-1]
        decrease i by 1
     else
        prefArray[i] := 0
  done
end
```

```
kmp(text, pattern)
  n ← length[text]
  m ← length[pattern]
  Call findprefix(pattern, m, prefArray)
  while i<n, do
     if text[i] = pattern[j], then
        increase i & j by 1
     if j = m, then
        print(pos(i-j))
        j ← prefArray[j-1]
     else if i<n and pattern[j] ≠ text[i] then
        if j ≠ 0 then
           i ← prefArray[j-1]
        else
           increase i by 1
  done
end.
```

Π-table / Longest prefix-suffix (LPS):-

string: a b a b c a b a b a b d

	1	2	3	4	5
pattern	a	b	a	b	d
	0	0	1	2	0

pattern start at 0

compare e string[i] & pattern[j+1]

~~if true i++, j++~~

if no match move i only not j

if false match come to pattern index of variable shown in table.

P[i] = a b a b a a c a

Π[i] = 0 0 1 2 3 0 0

0 1 2 3 4 5 6 7

for same value = add 1 to previous value

The knuth morris pratt algorithm is an efficient string matching algorithm that searches for a pattern in a text string in linear time complexity

Time complexity analysis.

1. Building the prefix table: O(m)
   m is length of pattern
2. searching the pattern in the text O(n), where n is the length of the pattern

Therefore, the overall time complexity of the KMP algorithm is O(m+n).

The prefix table is built by comparing the pattern with itself, and therefore its time complexity is linear in the length of the pattern. The table is used to skip over previously matched characters in the text string without starting over from the beginning of the pattern. This optimization reduces the no. of comparisons required to find the pattern in the text leading to linear time complexity of the algorithm.

→efficient for large strings & patterns.

0	1	2	3	4	5	6
b	c	a	b	c	x	x
0	0	0	1	2	0	0

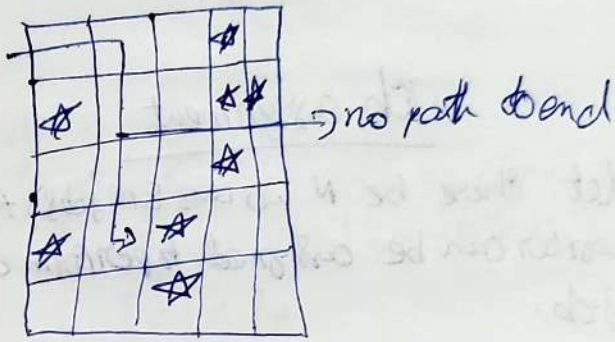
↙ b index of i ↓ index i+1=2

Check for possible path in 2D matrix

Given, a 2D array (m×n). The task is to check if there is any path from top left to bottom right in the matrix, -1 is considered as blockage, and 0 considered as path cell.

Note: Top left is always 0

Ex:- {{0,0,0,-1,0}, {-1,0,0,-1,-1}, {0,0,0,-1,0}, {-1,0,-1,0,0}, {0,0,-1,0,0}}



Logic:- This algorithm will perform BFS but the need for extra space will be eliminated by marking the array. So first run a loop and check which element of the 1st column and 1st row is accessible from (0,0) by using only 1st row & 1st column. mark them as 1, Now traverse the matrix from start to end row-wise in increasing index of rows & columns. If the cell is not blocked then check that any of its adjacent cells is marked 1 or not, if marked 1 then mark the cell 1

pseudocode:-

```
Path (int arr[row][col])
{ arr[0][0]=1;
  for(i=1; i<row; i++)
    if (arr[i][0] != -1)
      arr[i][0] = arr[i-1][0];
```

```
for(j=1;j<col;j++)
  if(arr[0][j]!=-1)
    arr[0][j] = arr[0][j-1];
for(i=1;i<row;i++)
  for(j=1;j<col;j++)
    if(arr[i][j]!=-1)
      arr[i][j] = max(arr[i][j-1],
                      arr[i-1][j]);
return(arr[row-1][col-1]==1);
```

TC:-

Every element of the matrix is traversed so tc is O(R*C).

Job assignment

Let there be N workers & N jobs; Any worker can be assigned to perform any job.

Branch & bound:-

2 approaches:-

1) For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row)

2) For each job, we choose a worker with lowest cost for that job from list of unassigned workers. (minimum entry from each column)

	J-1	J-2	J-3	J-4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Job 2 assigned to worker A

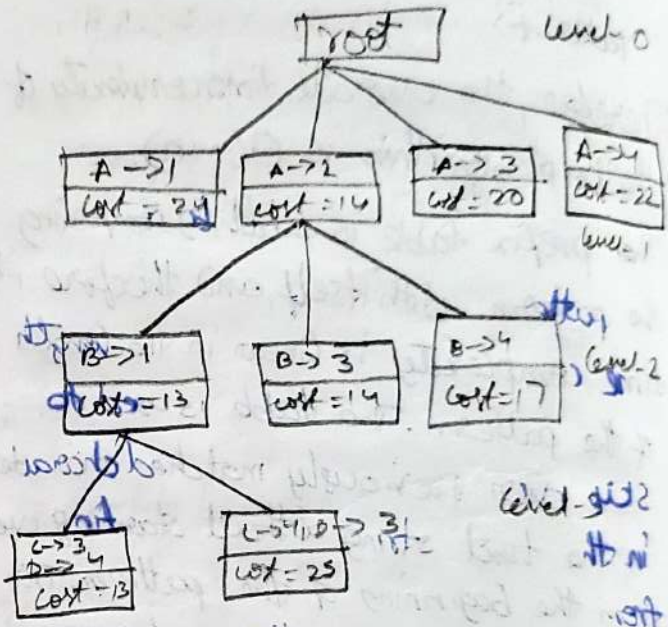
now cost = 2, now job 2 & worker 2 becomes unavailable

now we assign job 3 to worker B as it has min cost. now,

cost = 2+3 = 5 J-3, W-B also unavailable

finally job 1 gets assigned to worker C as it has minimum cost

Cost = 2+3+5+4 = 14



Algorithm:-

```
node {
    int job_no;
    int worker_no;
    node parent;
    int cost;
}
findmincost( cost matrix mat[][])
{
  //initialize list of live nodes
  //with root of search tree
  while(true)
{
//Find a live node with least estimated cost
E = least();
 if (E is a leaf node)
{
 printSolution();
```

```
return;
}
for each child x of E
{
add(x);
x->parent = E;
}
}
```

TC :-

The TC of this algo is O(M*N)
This is because the algorithm uses a double for loop to iterate through the MxN matrix.

karatsuba algorithm

problem of multiplying large integers :-

→ If n is the number of digits, then addition and subtraction algorithms run in O(n) time.

→ but the standard algorithm for multiplication runs in $O(n^2)$ time, which is quite costly.

n digit n=8

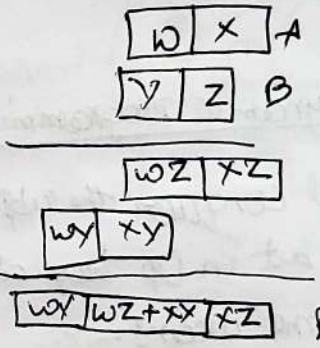
$n/2$ $n/2$

A[n-1 . . . . 0]

let $m = \frac{n}{2}$

let $w = A[n-1, \ldots m]$. $x = A[m-1 \ldots 0]$

$y = B[n-1 \ldots m]$, $z = B[m-1 \ldots 0]$



If w, x, y and z are $\frac{n}{2}$ digit number,

then $A = w \cdot 10^m + x$

$B = y \cdot 10^m + z$

and their product is :

$A \cdot B = (w \cdot y) 10^{2m} + (w \cdot z + x \cdot y) 10^m + (x \cdot z)$

ex :- multiply 1234 and 1234

A = 1234 B = 1234

$m = \frac{n}{2} = \frac{4}{2} = 2$

w = 12 x = 34

y = 12 z = 34

$A \cdot B = (w \cdot y) 10^{2m} + (w \cdot z + x \cdot y) 10^m + (x \cdot z)$

$= (12 \times 12) 10^4 + (12 \times 34 + 34 \times 12) 10^2 + (34 \times 34)$

A*B = 1522756

Pseudocode:-

```
procedure karatsuba (num1, num2)
  if (num1 < 10) or (num2 < 10)
    return num1 * num2
  /* Calculate the size of the numbers */
  m = max(size_base10(num1), size_base10(num2))
  m2 = m/2
  /* Split the digit sequence about the middle */
  high1, low1 = split_at(num1, m2)
  high2, low2 = split_at(num2, m2)
  /* 3 calls made to numbers approximately half the size */
  z0 = karatsuba(low1, low2)
  z1 = karatsuba((low1 + high1), (low2 + high2))
  z2 = karatsuba(high1, high2)
  return (z2 * 10^(2*m2)) + ((z1 - z2 - z0) * 10^(m2)) + (z0)
```

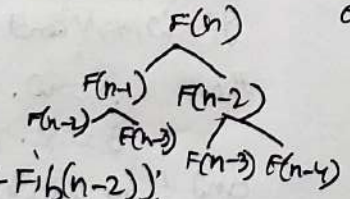
Dynamic programming

→ In divide and conquer the subproblems are independent but in DP they are connected by some means.

eg:-

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

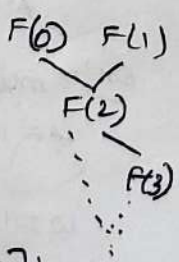
TC: $O(2^n)$



DP

```
fib_DP(n)
  f[0] = 0;
  f[1] = 1;
  for i = 2 to n
    f[i] = f[i-1] + f[i-2];
  return f[n];
```

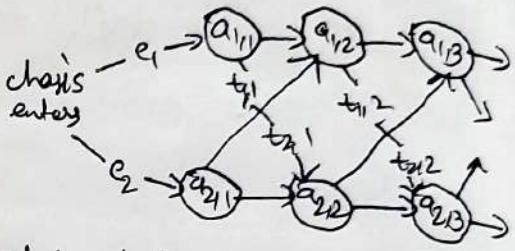
TC: $O(n)$



Assembly line scheduling problem

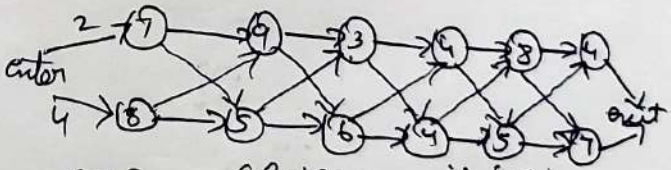
→ Auto mobile factory has 2 assembly lines.

* Each line has n stations: $S_{1,1}, \ldots, S_{1,n}$ & $S_{2,1}, \ldots, S_{2,n}$.
* Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$
* Entry times are e_1 and e_2 and exit times are x_1 & x_2



Qn) What stations should be chosen from line 1 and line 2 in order to minimize the total time through the factory for one car?

example:-



$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

	1	2	3	4	5	6
$l_1[j]$	1	1	2	1	1	2
$l_2[j]$	2	1	2	1	2	2

Pseudocode:-

```
Fastest_way(a, t, e, x, n)
  f1[1] ← e1 + a1,1
  f2[1] ← e2 + a2,1
  for i ← 2 to n
  do if f1[j-1] + a1,i ≤ f2[i-1] + t2,j-1 + a1,j
     then f1[j] ← f1[j-1] + a1,j
          l1[j] ← 1
     else f1[j] ← f2[j-1] + t2,j-1 + a1,j
          l1[j] ← 2
  if f2[j-1] + a2,j ≤ f1[j-1] + t1,j-1 + a2,j
  then f2[j] ← f2[j-1] + a2,j
       l2[j] ← 2
  else f2[j] ← f1[j-1] + t1,j-1 + a2,1
       l2[j] ← 1
if f1[n] + x1 ≤ f2[n] + x2
then f* = f1[n] + x1
     l* = 1
else f* = f2[n] + x2
     l* = 2
```

compute initial values of f_1 & f_2

compute values of $f_1[j]$ & $l_1[j]$

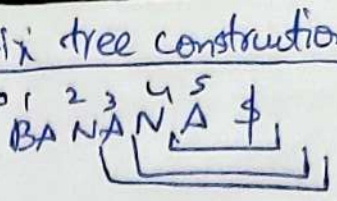
compute $f_2[j]$ & $l_2[j]$

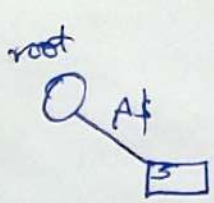
compute the values of fastest time through the entire factory.

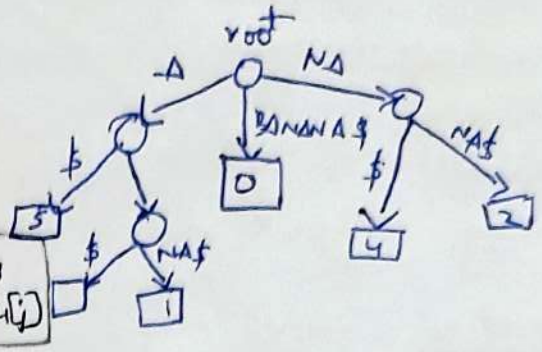
```
print stations(l, n)
  i ← l*
  print "line" i "," station n
  for j ← n down to 2
  do i ← li[j]
     print "line" i "," station j-1
```

minimize
0 1 2 3 4 5 6 7

Suffix tree construction



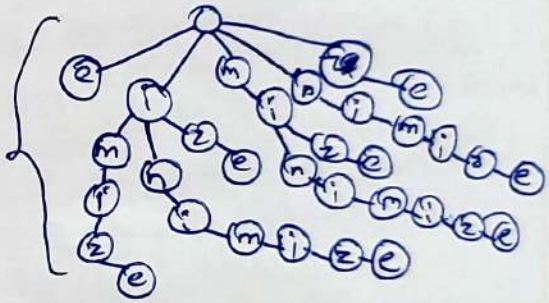




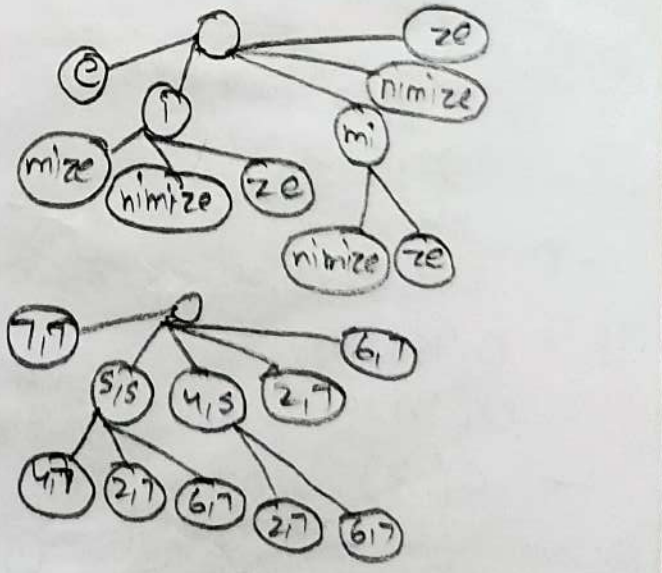
an:- minimize suffixes:-

e
ze
ize
mize
imize
nimize
inimize
minimize

S



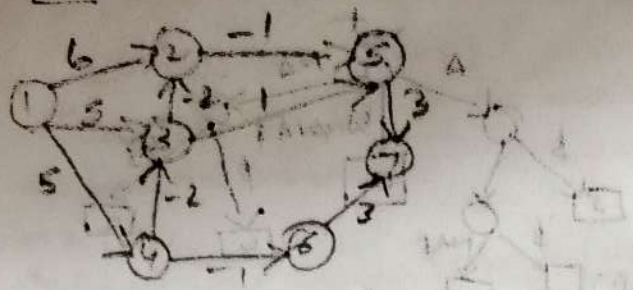
standard trie



Single source shortest path

Drawback :- when there is a negative weight cycle the bellman ford fails

Bellman ford :- uses DP



relaxation :- (u,v)

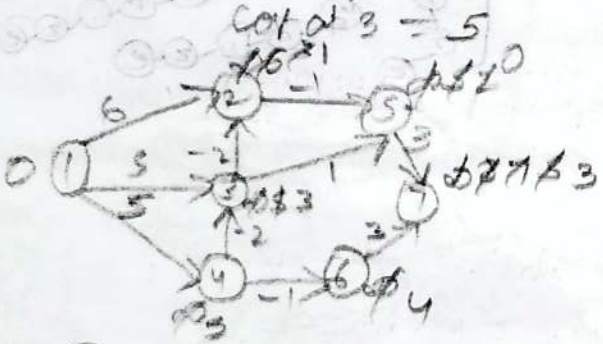
if (d[u] + c(u,v) < d[v])
d[v] = d[u] + c(u,v)

edge list -> (1,2)(1,3)(1,4)(2,5)(2,2)(3,5)
(4,3)(4,6)(5,7)(6,7)

(1,2) initially mark ① as 0 and remaining ∞

(1,2) -> 0 + 6 < ∞

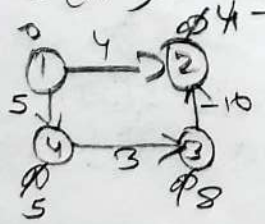
(1,5) -> 0 + 5 < ∞



1 - 0, 2 - 1, 3 - 3, 4 - 5, 5 - 0, 6 - 4, 7 - 3

TC :- O(|E| |V| - 1)
O(|V| |E|)
O(n²) if graph is complete, worst can O(n³)

ex :-



edges :-
(3,2)(4,3)(1,4)(1,2)
n = 4 - 1 = 3
2 relaxations 3 times

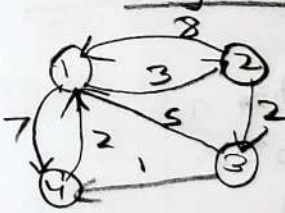
1 - 0
2 - (-2)
3 - 8
4 - 5

pseudocode :-

```
bellmanford (G, s)        -> graph, -> source vertex
for each vertex v in G
dist[v] ← ∞
prev[v] ← 0
  dist [s] ← 0
  for each vertex v in G
  for each edge (u,v) in G
  tempdist ← dist [u] + edgeweight(u,v)
    if tempdist < dist[v]
        dist[v] ← tempdist
        prev[v] ← u
  for each edge (u,v) in G
  if dist[u] + edgeweight(u,v) < dist[v]
    error : negative cycle exists
    return dist [], previ[].
```

TC :- BC :- O(V³)
AC :- O(EV)
BC :- O(E)

Floyd - warshall :-



$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 12 \\ 2 & 8 & \infty & 0 \end{bmatrix}$$

$A^0[2,3]$ $A^0[2,1] + A^0[1,3]$
$2 < 8 + \infty$

$A^0[2,4]$ $A^0[2,1] + A^0[1,4]$
$\infty > 8 + 7$

$A^0[3,2]$: $A^0[3,1]$

∞ > 5+3

$$A^2 = \begin{bmatrix} 0 & 3 & & \\ 8 & 0 & 2 & 15 \\ & 8 & 0 & \\ 8 & & & 0 \end{bmatrix}$$

$A^1[1,3]$. $A^1[1,2] + A^1[2,3]$

∞ > 3+2

$A^1[1,4]$ $A^1[1,2] + A^1[2,4]$

7 < 3+15

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

similarly,

$$A^3 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \quad A^4 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

$A^k[i,j] = \min\{A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j]\}$

Logic :-

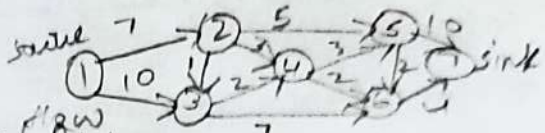
```
for(k=1; k<=n; k++)
  {
   for(i=1; i<=n; i++)
    {
     for(j=1; j<=n; j++)
      {
       A[i,j] = min(A[i,j], A[i,k] + A[k,j])
      }
    }
  }
```

pseudocode :-

```
for each d ∈ V do
    distance[d][d] ← 0;
end
for each edge (s,p) ∈ E do
    distance[s][p] ← weight(s,p);
end
n = cardinality(V)
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do
      if (distance[i][j] > distance[i][k] + distance[k][j])
        dist[i][j] ← dist[i][k] + dist[k][j];
      end
    end
  end
end
```

TC :- $O(n^3)$

Ford fulkerson algo

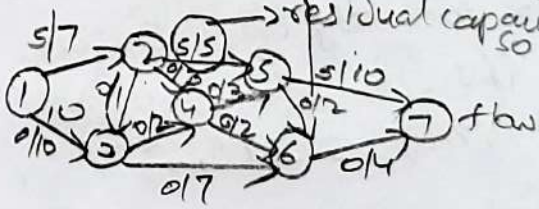


so flow 4/5 ← capacity

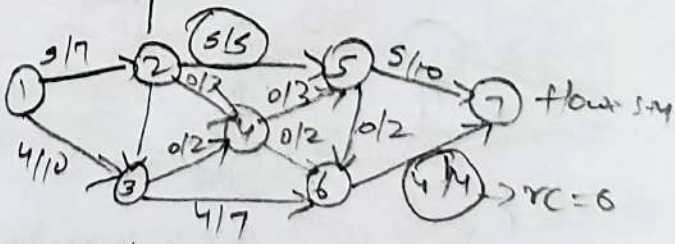
bottleneck capacity :- minimum capacity of any edge on the path

residual capacity :- every edge has a residual capacity which is equal to original capacity minus current

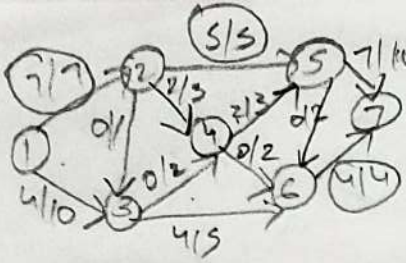
augmented path (source to sink path)	bottleneck capacity (min capacity)
1-2-5-7	5



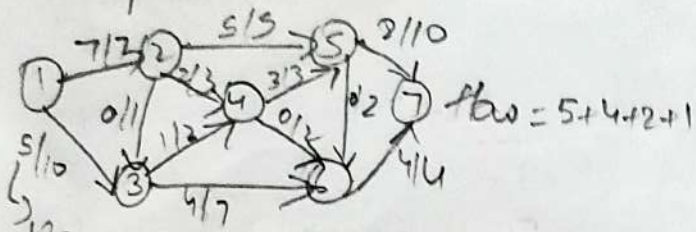
ap	bottleneck
1-3-6-7	4



ap	bn
1-2-4-6-7	2



ap	bn
1-3-4-5-7	1



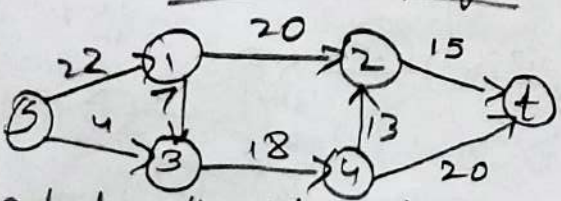
we are adding bottleneck capacities of created taken path.

∴ max flow = 5+4+2+1

Pseudocode:- TC:- $O(|E|*f)$ |E| no of edges, f is max flow

```
FordFulkerson(B,S,t)
    for each edge (u,v) ∈ E(G) do
        f(u,v) ← 0
        f(v,u) ← 0
    end
    while ∃a augumented path p ∈ Gf do
        Cf(p) ← min{Cf(u,v) : (u,v) ∈ p};
        for each edge (u,v) ∈ p do
            f(u,v) ← f(u,v) + Cf(p)
            f(v,u) ← -f(u,v)
    end
end
```

Edmonds-Karp algo



Select path with minimum edges

Paths:-
S → 1
S → 3

children:
- S: 1, 3

Paths

S → 1 → 2
S → 1 → 3
S → 3 → 4

S → 1 → 2 → t
shortest path

children

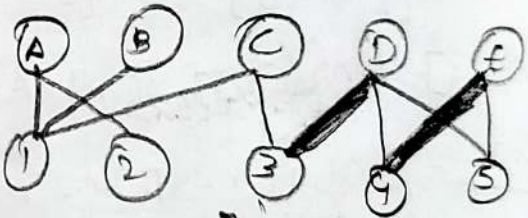
1: 2, 3
3: 4
4: 2, t
2: t
3: 4

Step 2:- select path with minimum edges

→ Calculate max flow for the path.
→ update residual graph
- reduce flows
- add return edges

Maximum matching in a bipartite graph

Let M be the matching in bipartite graph G.



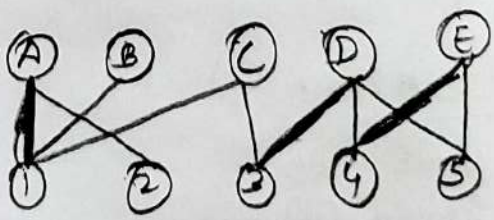
Sol which are not in

Find free vertex matching

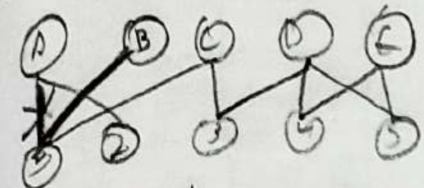
X = {A, B, C, D, E}

Y = {1, 2, 3, 4, 5}

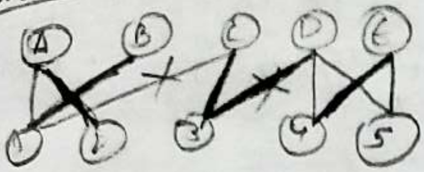
Lets consider vertex A :- M = {(A,1), (D,3), (E,4)}



consider B :- $M_c = \{(A,2), (B,1), (D,4), (E,4)\}$

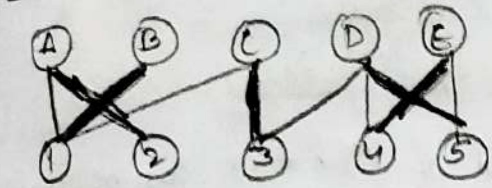


consider vertex C :-



$M_a = \{(A,2) (B,1) (C,3) (E,4)\}$

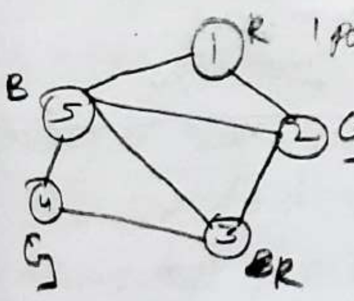
consider vertex D & E :-



Thus the maximum match for the bipartite graph is

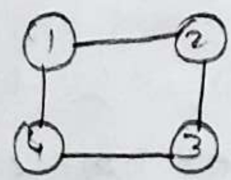
$M = \{(A,2), (B,1), (C,3), (D,5) (E,4)\}$

Graph coloring



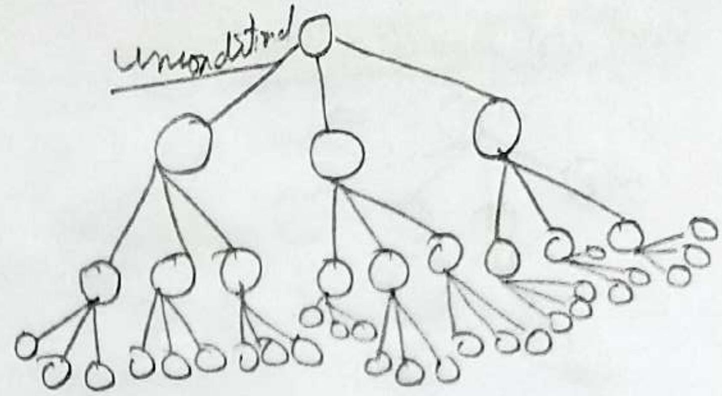
1 possible combo

eg :-

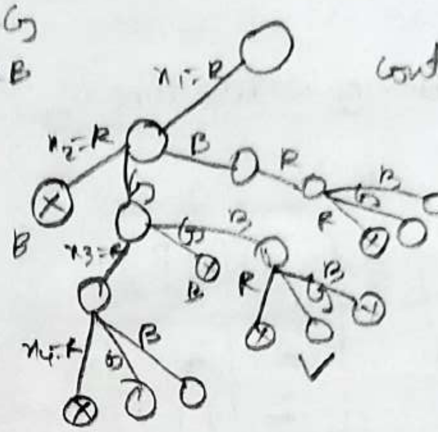


$m = 3$
$\{R, G, B\}$

unconstrained



1) RGRG
2) RGRB



continue like this

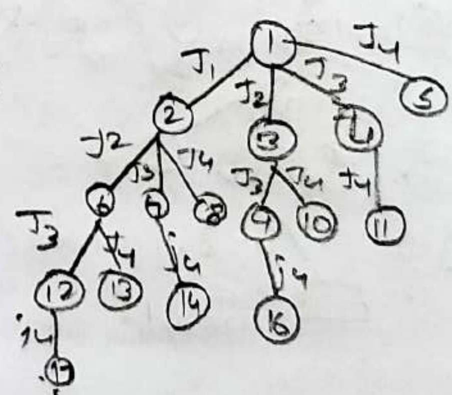
Branch and Bound :-

FIFO :- Jobs = $\{J_1, J_2, J_3, J_4\}$

$P = \{10, 5, 8, 3\}$

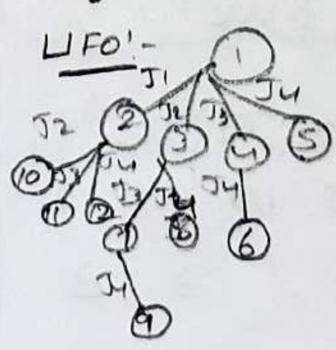
$d = \{1, 2, 1, 2\}$

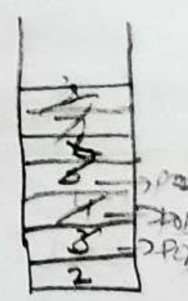
$S_1 = \{J_1, J_4\}$ $S_2 = \{1, 0, 0, 1\}$



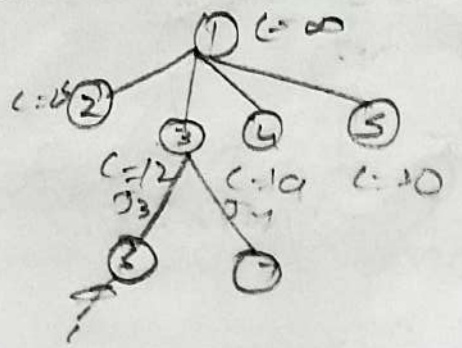
queue is used

LIFO :-





Least cost branch & bound :-

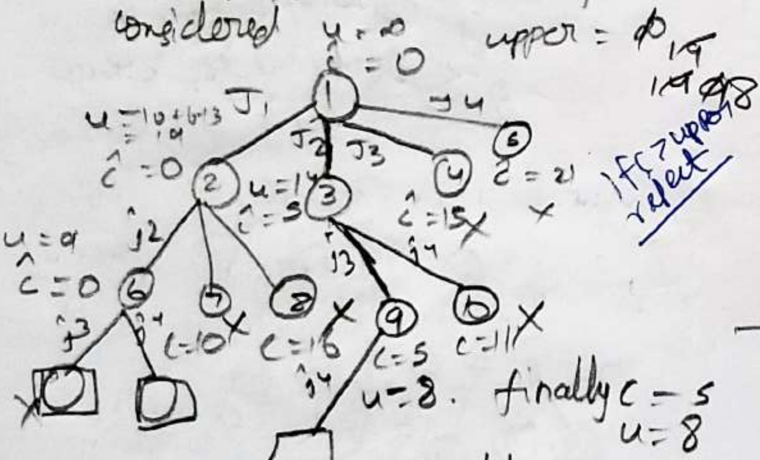


Job sequencing with deadline :-

Jobs	1	2	3	4
penality	5	10	6	3
deadline	1	3	2	1
time	1	2	1	1

u = sum of all penalities except that included in soln

$\hat{c}$ = sum of penalities till last job considered



finally c = 5
u = 8

penality = 8

Job sequencing with deadline (greedy) :-

n = 5

Jobs	J_1	J_2	J_3	J_4	J_5
profits	20	15	10	5	1
deadline	2	2	1	3	3

Slots OV| = 0–1 | 1–2 | 2–3 as max deadline is 3

Job consider	Slot assign	Solution	profit
—	—	ϕ	0
J_1	[1,2]	J_1	20
J_2	[0,1] [1,2]	J_1, J_2	20+15
J_3 ×	[0,1] [1,2]	J_1, J_2	20+15
J_4	[0,1], [1,2], [2,3]	J_1, J_2, J_3	20+15 +5
J_5	"	"	40

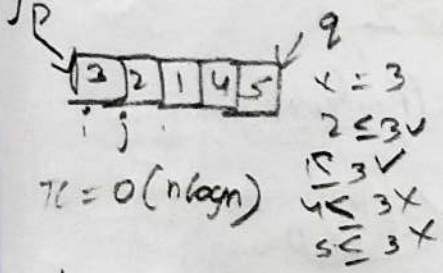
Greedy pseudocode :-

```
Algorithm greedyjob(d,J,n)
//J is set of jobs that can be completed
  by their deadlines
{
  J = {1};
  for i = 2 to n do
  {
    if (all jobs in J U {i} can be
        completed by their deadlines)
    then  J = J U {i};
  }
}
```

Randomized quicksort

Pseudocode:-

```
RQS(A,P,q)
{
if (P<q)
{
r = RGEC(P,q)   -> random generating function.
swap(A[P], A[r])
m = partition(A,P,q)
RQS(A, P, m-1)
RQS(A, m+1, q)
}
}
```



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

when an sorted array is input quicksort complexity is $O(n^2)$
to reduce we use random quicksort

The hiring problem:-

→ Suppose you need to hire a new 'office assistant'

→ Assume that your previous attempts at hiring have been 'unsuccessful' & you decide to use an "Employment agency"

→ The 'employment agency' sends you one candidate each day.

→ Now your role is to interview that person & then decide either to hire or not

→ Therefore, after interviewing each applicant if that applicant is better qualified than the current office assistant then you will fire the current office assistant and hire new

Algorithm:-

```
Hire-Assistant(n)
{
best ← 0;
for i = 1 to n do
{
interview candidate i;
if (candidate i better than candidate best) then
best = i
hire candidate i;
}
}
```

→ Interviewing a candidate has a low cost, denoted by 'c_i' where i is hiring cost.

→ Let 'm' be the no. of people hired then total cost associated with this algorithm is $O(c_i + c_n)$

→ In worst case, we actually hire every candidate that we interview. This occurs if candidates comes in increasing order of quality. ∴ total cost of hiring is $O(nc_n)$.

global minimum cut :-

A cut of connected graph is obtained by dividing vertex set V of graph G into 2 sets V_1 & V_2

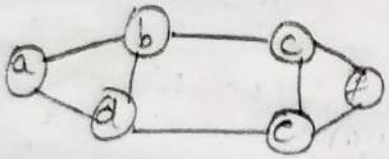
(i) There are no common vertices in V_1 & V_2 that is 2 sets are disjoint

(ii) $V_1 \cup V_2 = V$

random_mincut(G)

1) repeat step 2 to 4 until only two vertices are left
2) pick an edge e(u,v) at random
3) merge u & v
4) remove all self loops from E
5) return |E|

ex)



Sol) The graph is G(V,E) where

V = {a, b, c, d, e, f}

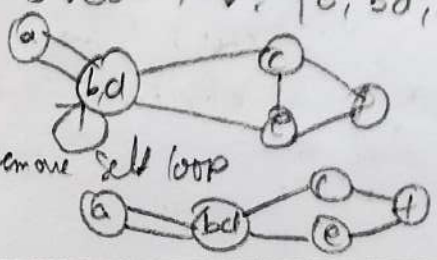
E = {(a,b), (a,d), (b,c), (b,d), (d,e), (c,e), (c,f)}

st①: check |V| = 6 > 2, yes

st②:- pick an edge at random say (b,d)

st③:- merge b & d now we have a new vertex (b,d)

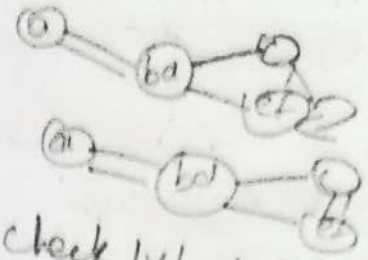
all edg incident on b are incident on this vertex, V: {a, bd, c, e, f}



check |V| = 5 > 2 yes

Pick an edge at random (e,f)

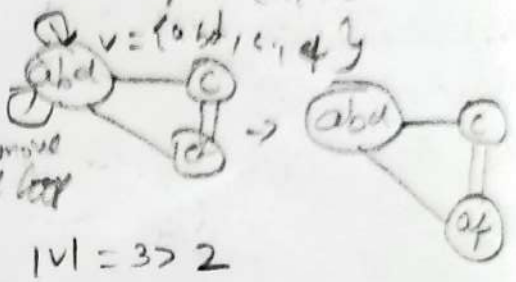
V = {a, bd, cef}



check |V| = 3 > 2 yes

pick an edge (a, bd)

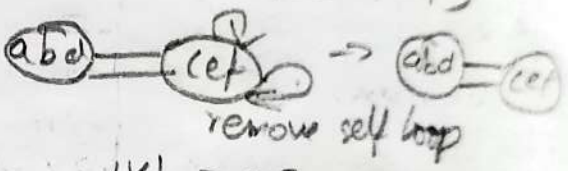
V = {abd, c, ef}



|V| = 3 > 2

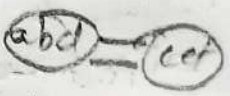
pick an edge (c, ef)

V = {abd, cef}



|V| = 2 > 2, no

return |E|



Pseudocode :-

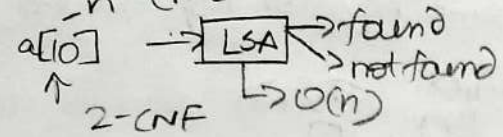
```
while |V| > 2 do
    select e ∈ E uniformly at random
    G = G/e :
od;
return |E|.
```

Input :- connected loop free multigraph with atleast 2 vertices

output :- an upper bound on the minimum cut of G.

P and NP class

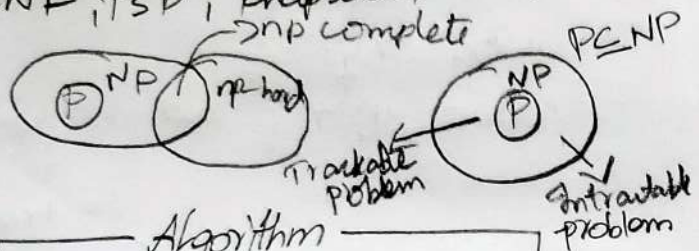
Class P :- problem solvable in polynomial time → $O(n^k)$. k - constant



Class NP :- It consists of those problem that are "verifiable" in polynomial time

→ Decision problems solvable in non-deterministic polynomial time.

3CNF, TSP, knapsack, 3 SAT,



Algorithm

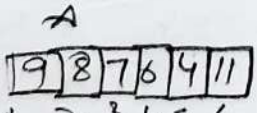
- Polynomial time
  - → Linear search (O(n))
  - → Binary search (O(log n))
  - → Insertion sort ($O(n^2)$)
  - → Merge sort (O(n log n))
- non polynomial time exponential time
  - → 0/1 knapsack - 2^n
  - → TSP - 2^n
  - → graph colouring - 2^n
  - → Sudoku - 2^n
  - → scheduling - 2^n

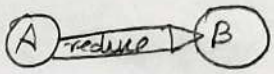
Deterministic Algo :-

The algo we use generally linear, binary

non Deterministic Algo :-

```
kundra search(A,n, key)
{
  j = choice();
  if (j == key)
  {
    print (j)
  //success
    print (0)
  //failure
}
```





Let A and B are two problem A reduces to problem B, if there is a way to solve A by deterministic algo that solve B in polynomial time

If A is reducible to B we denote $A \propto B$

satisfiability problem I_1 $\propto$ 0/1 knapsack I_2

→ If L_1 is reducible by satisfiable problem S_1 then L_1 is NP hard

satisfiability - 2^n

↑ np hard ↑ np complete

SAT problem :-

→ SAT is the problem of determining if a boolean formula is satisfiable or unsatisfiable.

satisfiable :- If the boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the formula is satisfiable

unsatisfiable :-

→ If it is not possible to assign such values, then we say that the formula is unsatisfiable.

$F = A \wedge \neg B$ is satisfiable, A = TRUE, B = False, F = TRUE

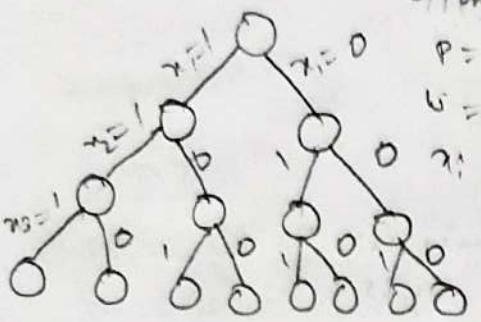
2-SAT is a special case of boolean satisfiability problem and can be solved in polynomial time.

2-SAT → Only 2 terms in CNF

→ It is proved that SAT is NP-complete

→ A boolean formula is in 3-conjugative normal form of 3-CNF if each clause has exactly 3 distinct literals

$(x_1 \wedge \neg x_1 \wedge \neg x_2) \vee (x_3 \wedge x_2 \wedge x_4) \vee (\neg x_1 \wedge \neg x_3 \wedge \neg x_4)$



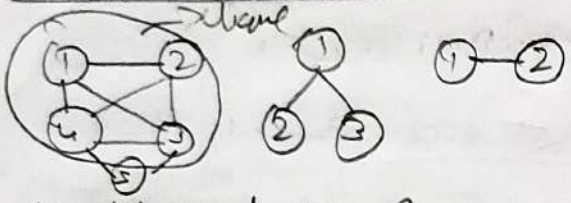
0/1 knapsack

$P = \{10, 8, 12\}$

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

$x_i = \{0/1, 0/1, 0/1\}$

Clique decision problem :-

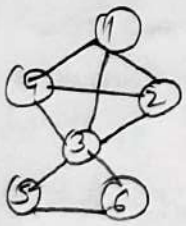


complete graph $|V| = n$

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

Clique is a subgraph of a graph which is complete.

or)



$k = 4$

$k = 3$

$k = 2$

Decision problem:- find if a graph is having a clique of k.

Optimization problem:- Find max clique in a graph.

Sat ∝ CDD

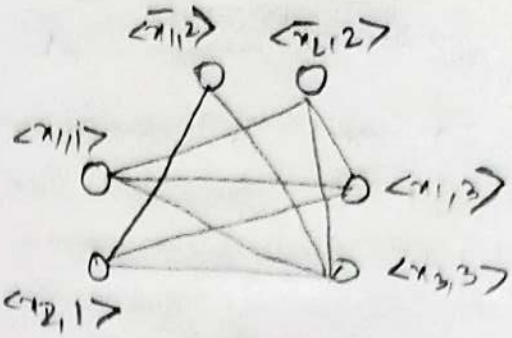
$F = \bigwedge_{i}^{k} C_i \quad F = (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (x_1 \vee x_3)$

$C_1 \quad C_2 \quad C_3$

rules for graph :-

1) same clause connection not allowed,

2) to same negation not allowed.



$k = 3$ here we have a clique of 3

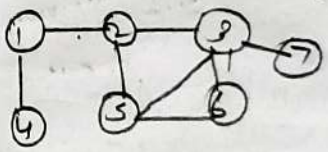
x_1	x_2	x_3	put in f we get
0	1	1	1

so true

Vertex cover problem

→ To find minimum subset of vertices that covers all the edges.

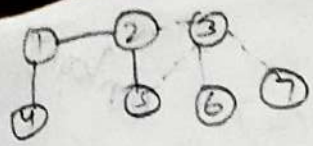
Ex :- given an undirected graph



Degree :- no. of edges that are connected to a vertex

vertex	Degree
1	→ 2
2	→ 3
3	→ 4
4	→ 1
5	→ 3
6	→ 3
7	→ 1

→ remove max degree vertex in the above & remove all associated edges



→ up 2 to degrees, ① → 2, ② → 2, ④ → 1, ⑤ → 2

→ select ⑤ & remove associated edges

→ remove vertex 1

subset of vertices $i \in \{3, 5, 1\}$

Set Cover problem

→ mathematical problem

→ Computational complexity theory

U – n elements

collection of subsets

$S = \{S_1, S_2, \ldots S_n\}$

minimum cost sub collection of S

Ex:- $U = \{1, 2, 3, 4, 5\}$

$S = \{S_1, S_2, S_3\}$

$S_1 = \{4, 1, 3\}$ cost $(S_1) = 5$

$S_2 = \{2, 5\}$ cost $(S_2) = 10$

$S_3 = \{1, 3, 2\}$ cost $(S_3) = 6$

$\{S_1, S_2\}$

$\{4, 1, 3, 2, 5\}$ - cost = 15

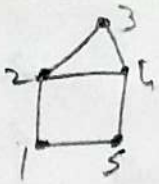
$\{S_2, S_3\} = \{1, 2, 3, 4, 5\} = 13$

Application:- 1) Is NP complete

TSP

1) Consider the graph of $G = (V, E)$ be an instance of Hamiltoncycle, we need to construct an instance of TCP.

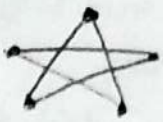
$G(V, E) =$



2) To form a TSP

2(a) construct a complete graph

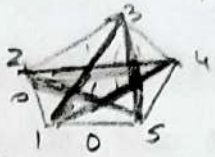
E' = complement graph



$G(V, E) \rightarrow$

2)b define the cost fn

$$c(i, j) = \begin{cases} 0 & \text{if } i, j \in E \\ 1 & \text{if } i, j \notin E \end{cases}$$

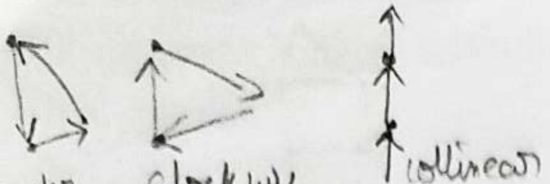


Conclusion:-

We now show that graph G has a hamiltonian edge if and only if graph G' has a cost of cost almost 0.

Geometric Algorithms

```
Segment_intersect(P1, P2, P3, P4)
d1 = DIR(P3, P4, P1); d2 = DIR(P3, P4, P2);
d3 = DIR(P1, P2, P3); d4 = DIR(P1, P2, P4);
if ((d1 > 0 && d2 < 0) || (d1 < 0 && d2 > 0)) &&
   ((d3 > 0 && d4 < 0) || (d3 < 0 && d4 > 0))
return TRUE
else if (d1 == 0) AND ON_SEG(P3, P1, P4)
return TRUE;
else if (d2 == 0) AND ON_SEG(P3, P1, P2)
return TRUE;
else if (d3 == 0) AND ON_SEG(P1, P2, P3)
return TRUE
else if (d4 == 0) AND ON_SEG(P1, P2, P4)
return TRUE
else return False.
```

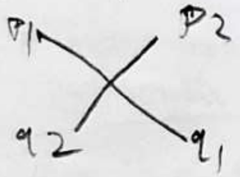


1) general case :-

- (p_1, q_1, p_2) & (p_1, q_1, q_2) have different orientations
- (p_2, q_2, p_1) and (p_2, q_2, q_1) have different orientations

2) Special case :-

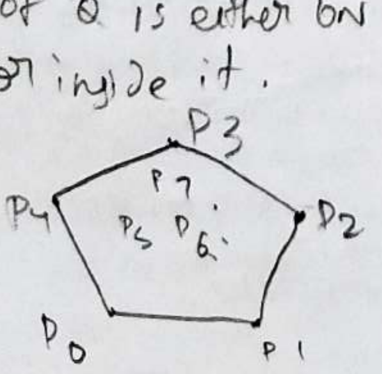
- (p_1, q_1, p_2), (p_1, q_1, q_2), (p_2, q_2, p_1) & (p_2, q_2, q_1) are all collinear &
- the x-projections of (p_1, q_1) & (p_2, q_2) intersect
- the y projections of (p_1, q_1) and (p_2, q_2) intersect.



(p_1, q_1, p_2) & (p_1, q_1, q_2) have different orientations
(p_2, q_2, p_1) and $(p_2, q_2,$

Convex Hull

-> set Q of points is the smallest convex polygon P for which each point of Q is either on boundary of P or inside it.



Pseudocode :-

GRAHAM SCAN (Q) $O(n \log n)$

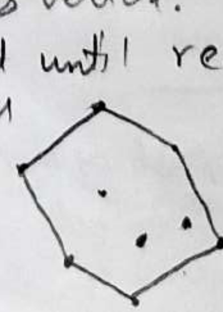
1) Let P_0 be the point in Q with the minimum y-coordinate $O(n)$
2) Let $\langle P_1, P_2, \ldots P_m \rangle$ be remaining points in Q, sorted by polar angle in counterclockwise. $O(n \log n)$
3) top[S] ← 0 $O(1)$
4) push(P_0, S) $O(1)$
5) push(P_1, S)
6) push(P_2, S)

for i ← 3 to m
  if (P_i left of top(S) by next to top(S)
  push(P_i, S)) makes a non left turn
  else pop(S) n+n+n = 3n
  push(P_i, S)
return S

1st sort points by their polar angle

Jarvis march

-> select the point with the lowest Y-coordinate - the 1st vertex
-> Select the point with smallest counterclockwise in reference to previous vertex.
-> repeat until reaching the start vertex



O(nh) where h is no of vertices.

Pseudocode:-

```
hull = [ ]
x0 = the leftmost point,
hull.push(x0)
loop hull.last() != hull.first()
  candidate = S.first()
for each P in S do
  if P! = hull.last() and P is on the
  right of the segment "hull.last()
                        candidate"
    candidate = P
if candidate != hull.first then hull.push
                                (candidate)
else
  break
return hull
```

$O(nh)$

(or)

```
Jarvis-march(P)
  P = {P0, P1 ... , Pn-1}
    P = P0, the lowest point
while p not highest point
  find point Pm with minimum polar
    angle from P → add Pm to hull
while P! = P0
  find point Pm with minimum polar
  angle from ← P add Pm to convex
                                hull
  P = Pm
```