* find @ O notation

$f(n) = 5n^2+2n+1$

we know that $c_1 g(n) \le f(n) \le c_2 g(n)$

$5n^2+2n+1$

$5n^2$ | $5n^2+2n+1$ | $8n^2$

$5n^2 = c_1 g(n)$

$8n^2 = c_2 g(n)$

(i) Big-oh (O):

$f(n) \le c_2 \cdot g(n)$

$5n^2+2n+1 \le 8 \cdot n^2$

for $n=0$ $1 \le 0$ false

for $n=1$ $8 \le 8$ true

$f(n) = c_2 \cdot g(n)$ $\forall$ $n \ge 1$

(ii) Omega (Ω)

$f(n) \ge c_1 \cdot g(n)$

$5n^2+2n+1 \ge 5n^2$ true

for $n=0$ $1 \ge 0$ true

n	$5n^2$	$5n^2+2n+1$	$8n^2$	
0	0	1	0	true
1	5	8	8	true
2	20	25	32	true
3	45	52	72	true

$f(n) \ge c_1 \cdot g(n)$ $\forall$ $n \ge 0$

(iii) Theta O:

$c_1 g(n) \le f(n) \le c_2 g(n)$

$5n^2 \le 5n^2+2n+1 \le 8n^2$

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

$5n^2+2n+1 = O(n^2)$ $\forall$ $n \ge 0$

(1) Big oh (O)

if $f(n) \le c_2 g_n$

$f(n) = c_2 g_n$

(2) omega (Ω)

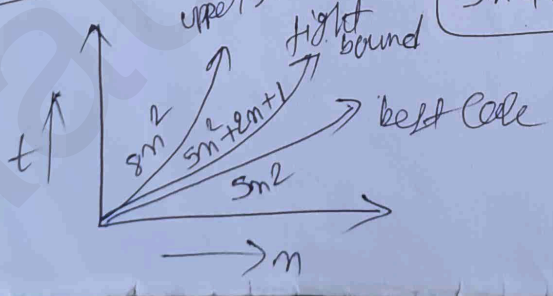
if $f(n) \ge c_1 \cdot g_n$

$f(n) = c_1 \cdot g_n$

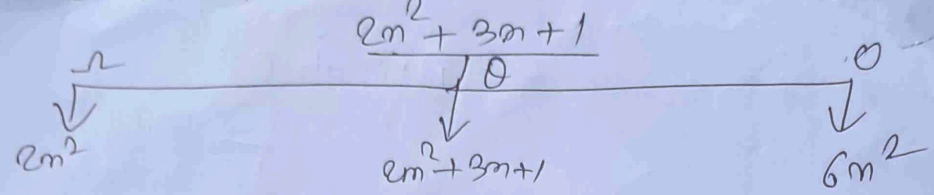
(3) Theta (Θ)

$c_1 g(n) \le f_n \le c_2 g(n)$

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



(a) $2n^2 + 3n + 1$



We know that $\Theta \Rightarrow$ ~~$c_1 \cdot g(n) \le f(n) \le c_1 \cdot g(n)$~~

$c_1 \cdot g(n) \le f(n) \le c_1 \cdot g(n)$

$\boxed{f(n) = \Theta(g(n))}$

$2n^2 \le 2n^2 + 3n + 1 \le 6n^2$

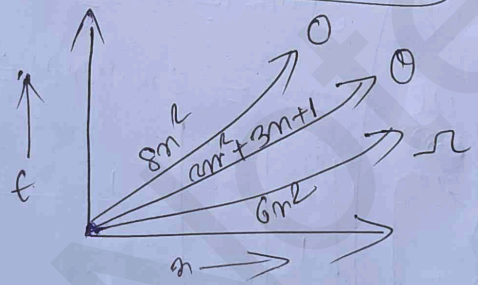
n	$2n^2$	$2n^2+3n+1$	$6n^2$	
0	0	1	0	true
1	2	6	6	true
2	8	15	24	true
3				

hence

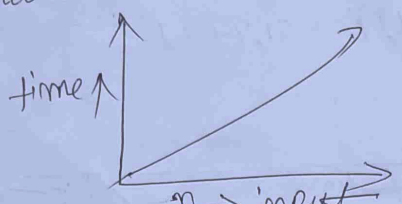
$\therefore c_1 \cdot g(n) \le f(n) \le c_1 g(n)$

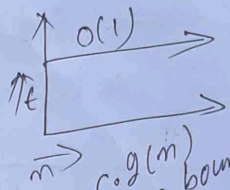
$\therefore \boxed{f(n) = \Theta(gn)}$

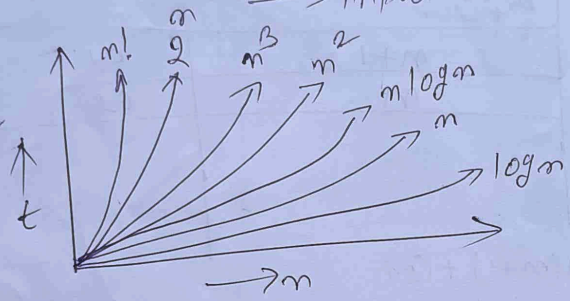
$\boxed{2n^2 + 3n + 1 = \Theta(n^2)}$ ~~$\forall$~~ $n \ge 0$

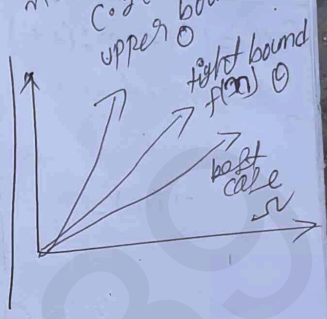


Time Complexity: Rate of growth of time taken by an algorithm with respect to input size is called Time Complexity.





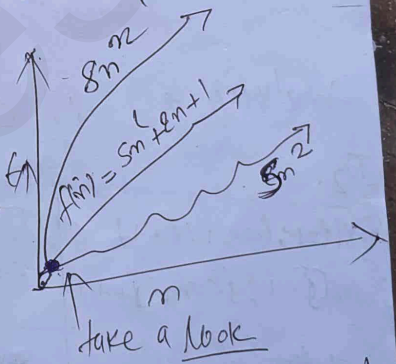




(*) $f(n) = 5n^2 + 2n + 1 \Rightarrow f(n) \leq 8n^2$ {multiplying n to $((2n+1) \times n)$ $5n^2 + 2n + n = 8n^2$}

$\Rightarrow 5n^2 \leq f(n) \leq 8n^2$ $\Rightarrow$ for $n \geq 0$

f(n)	$8n^2$	$5n^2$	n
1	0	0	0
8	8	5	1
20+4+1=25	32	20	2
45+6+1=52	72	45	3



Q Compute the tight bound for (i) $\frac{3n^2 + n}{2n^2 + 3n + 1}$ (2M)

perform Time complexity @ Analysis of following code snippet (CEM) MST

Q1. ~~final~~ for(i=1; i<=n; i++)
{ statement
}

for(i=1; i<=n; i++)	No of time execute	cost / exe
{ statement	n+1	c1
}	n	c2
T(n) =		

$T(n) = c_1(n+1) + c_2 n$

$T(n) = (c_1 + c_2)n + c_1$

. it is of the form: $T(n) = an + b$

where $a = c_1 + c_2$

$b = c_1$

Analysis → ∴ O(n)

Q2.

for(i=1; i<=n; i++)	No of time execute	cost / exe	
{for(j=1; j<=n; j++)	n+1	c1	$\propto n$
{ statement	n(n+1)	c2	$\propto n^2$
}	n^2	c3	$\propto n^2$

$T(n) = c_1(n+1) + c_2(n(n+1)) + c_3(n * n)$

$T(n) = c_1(n+1) + c_2(n^2 + n) + c_3(n^2)$

$T(n) = c1n + c1 + c2n^2 + c2n + c3n^2$

$T(n) = n(c1+c2) + n^2(c2+c3) + c1$

$\boxed{T(n) = an^2 + bn + c}$

Analysis :: $O(n^2)$

Q3.

```
for(i=1; i<=n; i++)
{ for(j=1; j<=n; j++)
  for(k=1; k<=n; k++)
  { statement; }
}
```

No of time exe	cost/exe
$n+1$	$c1$
$n(n+1)$	$c2$
$n^2(n+1)$	$c3$
n^3	$c4$

$T(n) = c1(n+1) + c2[n(n+1)] + c3[n^2(n+1)] + c4n^3$

$T(n) = c1n + c1 + c2(n^2+n) + c3(n^3+n^2) + c4n^3$

$T(n) = c1n + c1 + c2n^2 + c2n + c3n^3 + c3n^2 + c4n^3$

$T(n) = c1n + c2n + c2n^2 + c3n^2 + c3n^3 + c4n^3 + c1$

$T(n) = n(c1+c2) + n^2(c2+c3) + n^3(c3+c4) + c1$

$\boxed{T(n) = an^3 + bn^2 + cn + d}$

Analysis $O(n^3)$

Q4. P

```
for(i=1; i<=n; i++)
{
  for(j=1; j<=i; j++)
  {
    statement;
  }
}
```

i	j	Statement
n=1	j=1	1
n=2	j=1,2	1,2
n=3	j=1,2,3	1,2,3
n=4	j=1,2,3,4	1,2,3,4
n=5	1,2,3,4,5	
n+1	1+2+3+4+5+----	

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

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

$\Rightarrow O(n^2) = TC$

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

$1+2+3+4+5+6$

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

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

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

```
for(i=1; i<=10; i++)
{
  for(j=1; j<=i; j++)
  {
    code
  }
}
```

No of time ele	cost
n+1	c1
$\frac{(n+1)(n+2)}{2} - 1$	c2
$\frac{n(n+1)}{2}$	c3

$T(n) = an^2 + bn + c$

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

Statements

Perform the complexity analysis of the following

for(i=1; i<=n; i=i*2)
{
statement
}

i	i*2
2^0	2^1
2^1	$2 \times 2 = 2^2$
2^2	$2^2 \times 2 = 2^3$
2^3	$2^3 \times 2 = 2^4$

$2^k = n$

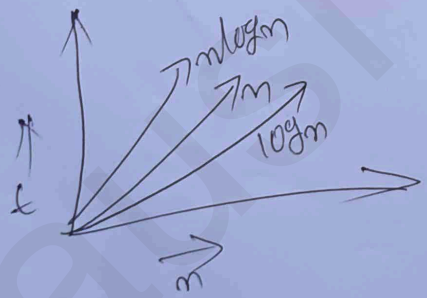
taking log both side

$\log_2 2^k = \log_2 n$

$k \cdot \log_2 2 = \log_2 n$

$k = \log_2 n$

$O(\log_2 n)$



Insertion Sort

8, 19, 2, 13, 50, 1 $\quad n = 6$

i=1 i=2 i=3

for(i=2 to n)

8 19 2 | 13 50 1

8 2 19 | 13 50 1

for j = 2 No of comparision

j=3	1+1=2
j=4	1+1+1=3
j=5	4
j=6	5

$1+2+3+4+5+\cdots+(n-1)$

No of Comparisons

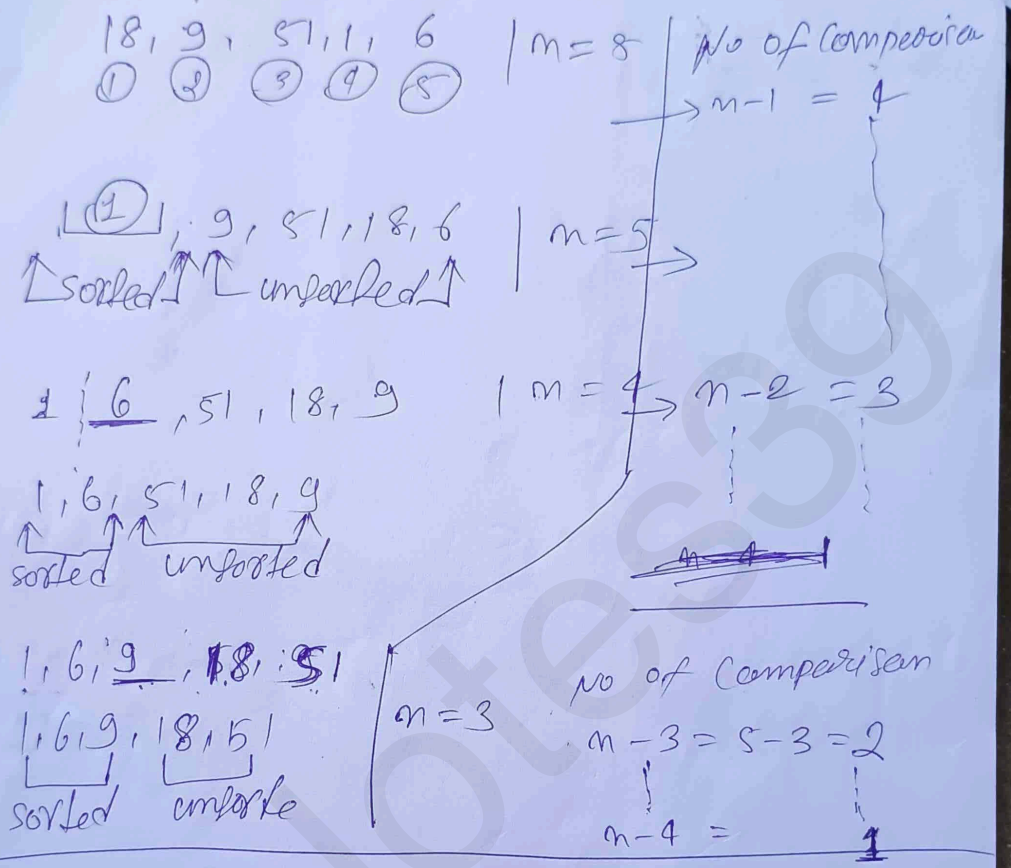
$1+2+3+4+\cdots+n$

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

$\frac{(n-1)[n-1+1]}{2}$

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

*perform the complexity analysis of
⑥ selection sort:



Total no of com = (n-1) + (n-2) + (n-3) + ---

= 1 + 2 + 3 + 4 + 5 + 6 + ---- + (n-1)

∵ → $\frac{n(n+1)}{2}$ ⇒ ∴ $\frac{(n-1)((n-1)+1)}{2} = \frac{n.(n-1)}{2}$

≈ $O(n^2)$

practice:

(c) perform complexity analysis of bubble sort.

```
for(i=1 to n-1)
    for(j=1 to n-i)
    {
        statement {A[j], A[j+1] if required}
    }
```

index $i=1$

$j=1 \Rightarrow (n-i) = (5-1) = 4$

9, 18, 5, 1, 6

$j=2$ A[j], A[j+1]

9, 18, 5, 1, 6

$j=3$ 9, 18, 1, 5, 1, 6

$j=4$ 9, 18, 1, 6, 5 | sorted

No of comparison

$i=5 \Rightarrow 4 = (n-1)$

$i=4 \Rightarrow 3 = (n-2)$

$i=3 \Rightarrow 2 = (n-3)$

$i=2 \Rightarrow 1 = (n-4)$

$i=1 \Rightarrow$ X

Total no of Comp. $(n-1)+(n-2)+(n-3)+ \dots 1$

$= 1+2+3+4+5+ \dots (n-1) \leq \frac{n(n+1)}{2}$

$\Rightarrow \frac{(n-1)(n-1+1)}{2} \Rightarrow \frac{n(n-1)}{2} \simeq O(n^2)$ TC

Bubble Sort: It states that by running n-1 iteration we will compare adjacent element and we will try to push larger element in end side by doing swapping and

1st Iteration
[4,1,5,2,3]
[1,4,5,2,3]
[1,4,5,2,3]
[1,4,2,5,3]
[1,4,2,3,5] 5 sorted

2nd
[1,4,2,3,5]
[1,4,2,3,5]
[1,2,4,3,5]
[1,2,3,4,5] sorted

3rd
[1,2,3,4,5]
[1,2,3,4,5]
[1,2,3,4,5] sorted

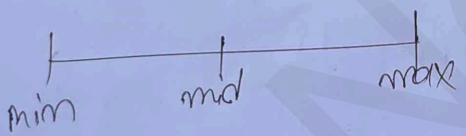
4th
[1,2,3,4,5]
[1,2,3,4,5]
sorted

```
for(i=0; i<n-1; i++)
{ for(j=0; j<n-i-1; j++)
  { if(arr[j]>arr[j+1])
    { to swap(arr[j], arr[j+1]);
}}}
```

3) max - Mix ele
Divide & method:
Arr = [8, 19, 2
max = 19
min =
Total

* Divide & Conquer Algorithm Design Technique

Binary Search Algo: sorted



$n \to \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \dots 1$

$\frac{n}{2^k} = 1 \quad n = 2^k$ taking $\log_2$ both side

$\log_2 n = \log_2 2^k \quad \because \log_2 2 = 1$

$\log_2 n = k \cdot \log_2 2$

$k = O(\log_2 n)$ time complexity

n-1 iterations and we in endside by

(8) Max_Mix ele in an array:

(1) Direct Method:

Arr = [8, 19, 2, 1, 7] (1c, 2c, 3c, 4c)

Max = 8 19 no of Comparison = 4 = 5-1] (n-1)

Min = 8 2 1 no of comparison = 4 = 5-1] (n-1)

Total no of comparison required = (n-1)+(n-1) = 2(n-1)

(2) Divide and Conquer Method

Case 1: Arr = 70 (i, j)

```
if(i==j)
{ //only one ele inside an array
  max=70; //no of comparison =0
  min=70;
}
```

Case 2: Arr = 70, 2 (i, j)

```
if(i==(j-1))
{ //only two ele in an array
  if(arr[i]>arr[j])
  { max=arr[i];
    min=arr[j]; //No of Comparison = 1
  }
  else{ max=arr[j];
        min=arr[i];
  }
```

Case 3: arr = [70, 2, 80] (i, mid, j)

when n>2

else mid = $\frac{i+j}{2}$

max_min(i, mid);

max_min(mid+1, j);

max_min(i, mid) | max_min(mid+1, j)

70 | 2 | 80 → 80

70 | 2

max1 = 70

min1 = 2

max2 = 80

min2 = 80

compare both

max_ele = max(max1, max2); // 80

min_ele = min(min1, min2); // 2

No of comparisons = 2

$$T(n) = \begin{cases} n = 1, & \text{no of comparisons} = 0 \\ n = 2, & 1 \\ n > 2, & T(n/2) + T(n/2) + 2 \end{cases}$$

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

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

$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + 2\right] + 2$ put value of $T(n)$ in eqn (1)

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

$T(n) = 4T\left(\frac{n}{4}\right) + 6$

$T(n) = 4T\left(\frac{n}{2^2}\right) + 8 - 2$ here $T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) \cdots T(2) = 1$

$T(n) = 4T(2) + 8 - 2$

$T(n) = 2^{3-1} \cdot 1 + 2^3 - 2$

$T(n) = 2^{k-1} \cdot + 2^k - 2$

$\boxed{T(n) = \frac{2^k}{2} + 2^k - 2}$

$8 = 2^3 = 2^k$

$\because T(2) = 1$

$\because 2^3 = 2^k$

Put $2^k = n$

$T(n) = \frac{n}{2} + n - 2 \Rightarrow T(n) = \frac{n+2n}{2} - 2$

$\boxed{T(n) = \frac{3n}{2} - 2}$ D&C

$\boxed{T(n) = 2(n-1)}$ Direct Method

Compare D&C with direct Method

$\boxed{T(n) = \frac{3n}{2} - 2 < 2(n-1)}$

D & C Method

no of Comparison $= \frac{3n}{2} - 2$

taking $n = 100$

$\frac{3 \times 100}{2} - 2 = \frac{300}{2} - 2$

$= 150 - 2$

$= \underline{148}$

Direct Method

No of Comparison $= 2(n-1)$

taking $n = 100$

$= 2(100-1)$

$= 2(99)$

$= 198$

$\Rightarrow \frac{148}{198} \times 100 = 75\%$ hence D&C saves 25% of comparisons

```
Void maxMin(int i, int j, Type &max, Type &min)
{
  if(i==j) max=min=a[j];
  else if(i==j-1)
  {
    if(a[i]<a[j])
    {max=a[j];
     min=a[i];
    }
    else{
        max=a[i];
        min=a[j];
    }
  }
  else{
    int mid=(i+j)/2; Type max1, min1;
    maxMin(i, mid, max, min);
    maxMin(mid+1, j, max1, min1);
     if(max<max1) max=max1;
     if(min>min1). min=min1;
}}
```

*Quick Sort
Step 1: Choose
Step 2: To

*Quick sort Algorithm: (pg8)

Step1: Choose an element as pivot.

Step2: Take two pointer at left & right of the array excluding pivot, left at starting ele of array and right at the end of the array.

Step3: if left is less than or equal to pivot element then increase left idx and same if right element is greater than the pivot ele then decrease right idx.

Step 4: If left pointer is greater than pivot element & right pointer is less than pivot element then swap both left to right, right to left by considering that left do not cross each other

Step5: The moment left and right both crossing each other break the loop hence array is sorted.

Ex ⇒ 7 1 8 2 (5) 6 3 4
↑ left, ↑ right, pivot

(pg9) Illustrate the performance of Quicksort algorithm for its average and worst cases.

Average case	Worst case
pivot divides the array into approximately two half.	when the pivot is either the first or last element in an array.
$T(n) = T(n/2) + T(n/2) + cn$ —(1)	$T(n) = T(1) + T(n-1) + cn$ —(1)

Average case

$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + cn$ — (i)

$T(n) = 2T(\frac{n}{2}) + cn$ — (ii)

Find $T(\frac{n}{2})$ by putting $n = \frac{n}{2}$ in eq^n (i)

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

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

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + \frac{cn}{2}$ — (3)

put (3) in eq^n (ii)

$T(n) = 2[2T(\frac{n}{4}) + \frac{cn}{2}] + cn$

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

we will divide until a single element will left

$T(n) = 4T(\frac{n}{2^2}) + 2cn$

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

$T(n) = 2^k T(\frac{n}{2^k}) + kcn$

$T(n) = 4T(1) + 2cn$

$T(n) = 4T(1) + 2cn$

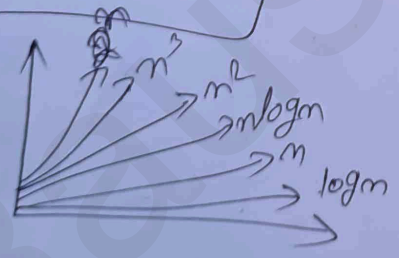
$T(n) = 2^2 T(1) + 2cn$

$T(n) = 2^k a + kcn$

$T(n) = 2^k a + kcn$ $\quad \{2^k = n$

$T(n) = na + cn\log n \quad \{k = \log n$

$T(n) = n\log n$



Worst case

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

$T(n) = T(n-1) + cn$ — (1)

find $T(n-1)$ by putting $n = n-1$ in eq^n (1)

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

$T(n-1) = T(n-2) + c(n-1)$ — (3)

put eq (3) in (1)

$T(n) = T(n-2) + c(n-1) + cn$ — (5)

Again find $T(n-2)$ by putting $n = n-2$ in eq (1)

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

$T(n-2) = T(n-3) + c(n-2)$ — (4)

put eq (4) in (5)

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

Do the same process till $T(1)$

$T(n) = T(1) + c(n-2) + c(n-1) + cn$

$T(n) = c(n-2) + c(n-1) + cn$

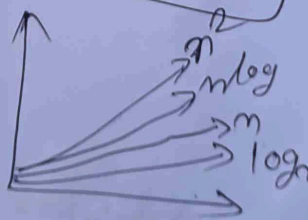
$T(n) = c[n-2 + n-1 + n]$

$T(n) = c[n + n-1 + (n-2) + \dots 1]$

$T(n) = [c(n + n-1 + (n-2) + \dots 1) - 1]$

$T(n) = c[\frac{n(n+1)}{2} - 1]$

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



* Greedy Algo

Worst case

$T(n) = T(n-1) + T(1) + cn$ — (1)

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

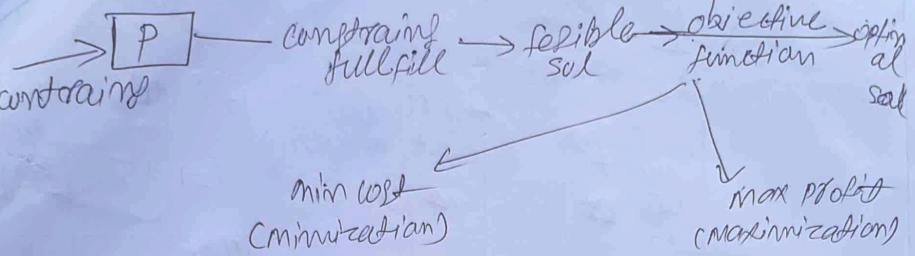
Find $T(n-1)$ by putting $n = n-1$

eq (1) $(n-1-1) + c(n-1)$ — (3)

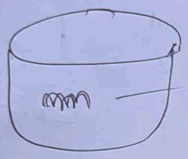
$+ c(n-1)$ — (5)

$n-2$

(c) * Greedy Algorithm:



* Knapsack problem by Greedy Method:



$i = 1, 2 \ldots n$ — weight (w_i)

x_i = function profit (p_i)

$0 \le x_i \le 1 \rightarrow \{0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4} \ldots \frac{1}{c}\}$

$$\sum_{1 \le i \le n} w_i x_i \le m$$

optimal sol

maximum $\sum p_i x_i$

P	W	profit/weight
100	50	2
90	25	3.5

$m = 25$

$\frac{50}{2} = 25$

$w_1 x_1 = 50 \times \frac{1}{2} = 25$

$P_1 x_1 = 100 \times \frac{1}{2} = 50$

Q Capacity = 20

no of ele = 3

profit = (25, 24, 15)

weight = (18, 15, 10)

round(5) = 4

Compute profit/weight Step 1:

$PW = (1.38, 1.6, 1.5)$ (E1, E2, E3)

Step 2: make a table

Element	profit per weight	weight	pi profit	our capacity (U)	xi (part of the element)	wixi	profit earn
E2	1.6	15	24	20	xi = 1	15	24 × 1 = 24
E3	1.5	10	15	5	10 / 10 = 1/2	10 × 1/2 = 5	15 × 1/2 = 7.5
E1	1.38	18	25	U = 5 − 5 = 0	xi = 0	0	0
							$\sum p_i x_i = 31.5$

Step 3: $\sum_{1 \le i \le n} w_i x_i \le m$

$w_i x_i = 15 + 5 = 20$, $m = 20$

$w_i x_i \le m$

any ⇒

$\langle E1, E2, E3 \rangle$

$\langle 0, 1, 1/2 \rangle$

$\langle x3, x1, x2 \rangle$

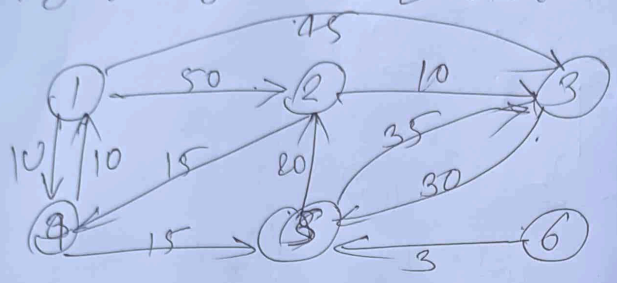
Maximum profit = 31.5

Maximize $\sum_{1 \le i \le n} p_i x_i$ $\quad 0 \le x_i \le 1$ (Greedy para) fractional knapsack

$x_i = 0$ or 1 zero one knapsack 0/1 → (Greedy failed)

* Minimum greedy
A tree

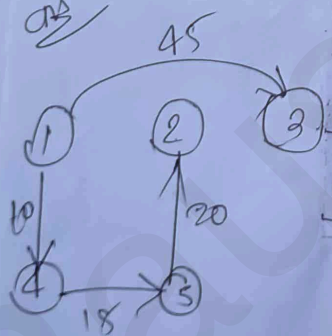
Q) Single source shortest paths problem (using greedy method [Dijkstra's Algo])

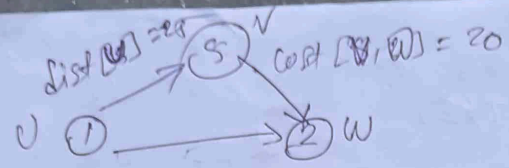


Source (1) shortest paths from source to every other node

Vertices / 1 - cost		
Vertices	cost	Selected vertex
(1), (4)	10	(4)
(1), (5)	unreachable (∞)	X
(1), (4), (5)	10+15=25	(5)
(1), (2)	50	X
(1), (4), (5), (2)	10+15+20=45	(2)
(1), (2), (3)	50+10=60	X
(1), (4), (5), (2), (3)	10+15+20+10=55	X
(1), (3)	45	(3)
(1), (6)	unreachable (∞)	

Ans





dist[w] = 50

dist[w] > dist[v] + cost[v, w]

dist[w] = dist[v] + cost[v, w]

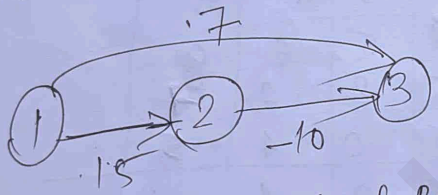
dist[1] = 0
dist[2] = 50
dist[3] = 45
dist[4] = 10
dist[5] = ∞
dist[6] = ∞

Time = O(V+E) log V

if you use Red Black Tree : O((V+E) log V)

Fail on this graph ⇒ Dijkstra's failed on negative edge

greedy ~~algorithm~~

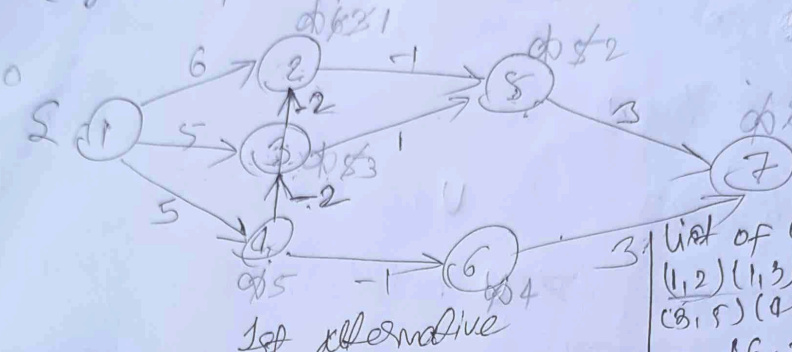


(Pyq) Compare the efficiency of Bellman and ford algorithm with Dijkstra's algorithm to solve single source path problem in a graph.

OR

Defend the statement that Bellman and ford algorithm is superior than Dijkastra's Algorithm

* Single source shortest paths problems:
(Dynamic pro) [Bellman & ford]:



1st alternative

List of edge :
(1,2) (1,3) (1,4) (2,5) (3,2)
(3,5) (4,3) (4,6) (5,7)
(6,7)

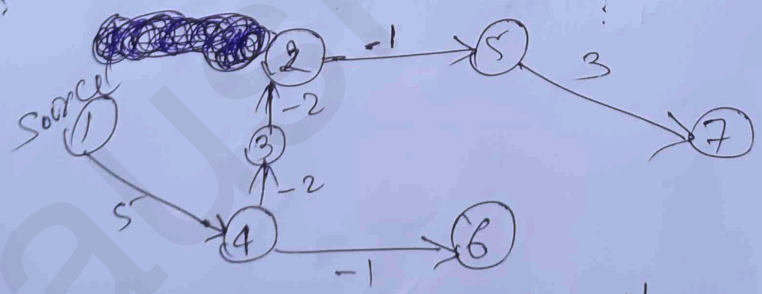
dist [2] = directed edge = 6

dist'[2] = 6

principle of optimality	1	2	3	4	5	6	7
1st Altern	0	6	5	5	∞	∞	∞
2nd Alt	0	3	3	5	5	4	∞
3rd Alt	0	1	3	5	2	4	7
4th alt	0	1	3	5	0	4	5
5th alt	0	1	3	5	0	4	3
6th alt	0	1	3	5	0	4	3

dist[2] = min (dist'[2], via node 3 by iff first alternative)

dist^3[2] = min(dist^2[2], via node 4 by iff second alternative)



Sorted path

Dijkstra's algorithm
Dijkstra's algorithm
path from a
vertices
. It
small

Dijkstra's Algorithm (Greedy Approach):

Dijkstra's algorithm used to find the shortest path from a single source to all other vertices in a graph with non-negative edge weights. It uses a priority queue (min heap) to choose the smallest known distance. It is a greedy technique but it fails on negative edge weight.

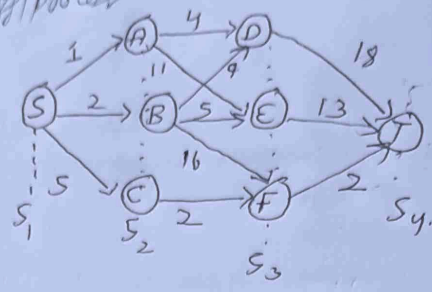
Bellman-ford Algorithm: Bellman-ford algorithm is used to find the shortest path from a single source to all vertices, it also work on negative edge weight, it is a dynamic technique which have more than one decision sequence.

Comparing the efficiency of Bellman and ford algorithm.

Aspect	Dijkstra's Algo	Bellman-ford Algo
worst-Case Time Com	$O(V+E)\log V \approx O(E \log V)$	$O(VE)$
Best Case Time com	$O(V \log V)$ {if $E = V$}	$O(VE)$
Space Complexity	$O(V+E)$ (Adjacency List) + (priority queue)	$O(V)$ {Distance array}
Edge-weight	work only with non-negative edge weights	works with negative weights too.
Decision sequence	only one decision sequence since it uses greedy technique.	More than one decision sequence, since it uses dynamic prog approach.

Hence, based on the comparision Bellman ford algorithm is superior than Dijkstra's algorithm.

(Q5) Dynamic programming for solving multistage graph using forward & backward approach.



Stage 4 :- T

Stage 3 :- $d(D,T) = 18$, $d(E,T) = 13$, $d(F,T) = 2$

Stage 2 :- $d(A,T) = \min\{4 + d(D,T), 11 + d(E,T)\}$.

$\min\{4+18, 11+13\} = 22$.

$A \rightarrow D \rightarrow T$.

$d(B,T) = \min\{5 + d(E,T), 16 + d(F,T), 9 + d(D,T)\}$.

$\min\{5+13, 16+2, 9+18\}$.

18, $B \rightarrow E \rightarrow T$.

$B \rightarrow F \rightarrow T$.

$d(C,T) = 2 + d(F,T) = 4$.

$C \rightarrow F \rightarrow T$.

Stage 1 :- $d(S,T) = \min\{1 + d(A,T), 2 + d(B,T), 5 + d(C,T)\}$

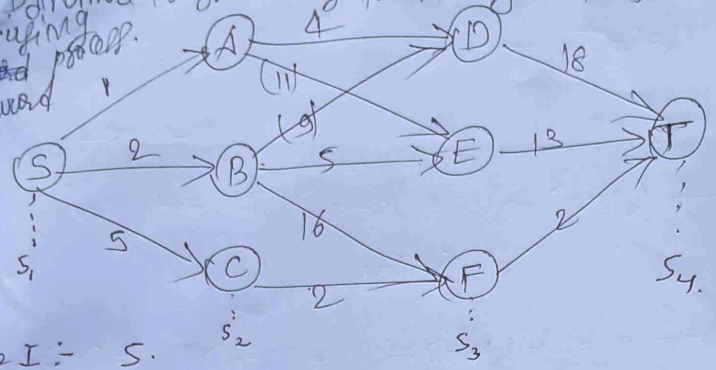
$\min\{1+22, 2+18, 5+4\}$

(9).

$S \rightarrow C \rightarrow F \rightarrow T$.

Dynamic programming for Multi-stage graph

pgm using backward process.



backward process

Stage I :- 5.

Stage II :- $d(S,A) = 1$, $d(S,D) = 2$, $d(S,C) = 5$.

Stage III :- $d(S,D) = \min\{1 + d(A,D),\ 2 + d(B,D)\}$

$\{1+4,\ 2+7\} = 5$

$S \to A \to D$

$d(S,E) = \min\{1 + d(A,E),\ 2 + d(B,E)\}$

$\min\{1+11,\ 2+5\} = 7.$

$S \to B - E.$

$d(S,F) = \min\{2 + d(B,F),\ 5 + d(C,F)\}$

$\{2+16,\ 5+2\} = 7.$

$S \to C \to F.$

Stage IV

$d(S,T) = \min\{d(S,D) + d(D,T),\ d(S,E) + d(E,T),$

$d(S,F) + d(F,T)\}.$

$\min\{5+18,\ 7+13,\ 7+2\}$

$= 9$ $S \to C \to F \to T$

* Knapsack problem using dynamic programming:

0/1

$n=3$ $m=6$

{0/1 → Dynamic
Fractio → Greedy

(P_1, W_1)	(P_2, W_2)	(P_3, W_3)
(1,2)	(2,3)	(5,9)

S^9 ← not including ele 9 in knapsack

S^9_1 ← including ele 9 in knapsack

$S^1 = \{(0,0)\}$
$S^1_1 = \{(1,2)\}$ } merge & apply purge(discard)

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

case I:

case II: (P_k, W_k) $W_k > m$

$S^2_1 = \{(2,3)(3,5)\}$ { 1 2 3 / 3 5 }

$S^3 = \{(0,0)(1,2)(2,3)(3,5)\}$ { profit should inc & weight should inc

$S^3_1 = \{(5,9);(6,6)(7,7),(8,9)\}$

$S_4 = \{(0,0)(1,2)(2,3)(3,5)(5,9)(6,6)(7,7)(8,9)\}$

decreasing

S^4_k = Maximum profit = 6,6 {optimal soln}

E1 E2 E3

<1, 0, 1> → answer

6,6
5 9
1 2
1 2
0 0

Dynamic
Greedy

Slove 0/1 knap sack problem using greedy method and proove how greedy failed on 0/1 knap sack and propose a optimal solution to slove that problem (Dynamic programming).

Soln:

P1,w1	P25,18	
P2,w2	24,15	
P3,w3	15,10	

Capacity = 20

Slowing 0/1 knap sack problem using greed.

ele	Prof, wt	Profit/wt	cap size	xi	Profit
P1,w1	25,18	1.38	5	0	0
P2,w2	24,15	1.6	20	1	24
P3,w3	15,10	1.5	5	0	0

profit = 24

Greedy will take the element with high profit/weight i.e (24,15) is added but here still capacity = 5 remains since 0/1 knap will not allow fraction of any ele so only one ele is added to knap.

But as we can clearly seee that there is an ele which (25,18). has high profit (25) and its weight is also less than [m = 20]. Therefore Greedy fails in this case.

Now, proposing a optimal solution to slove 0/1 knap using dynamic prog.

Dynamic ⇒ $S_0^1 = \{(0,0)\}$

$S_1^1 = \{(25,18)\}$ } merge & purge

$S^2 = \{(0,0)(25,18)\}$

$S_1^2 = \{(24,15)(49,33)\}$] merge & purge

→ weight is > m discard / purge

$S^3 = \{(0,0)(25,18)(24,15)\}$

$S_1^3 = \{(15,10)(40,28)(39,25)\}$ } → merge & purge

→ wei > m

$S_4^4 = \{(0,0)(25,18)(24,15)(15,10)\}$

S_i^4 This will not execute as there is not 4th ele.

Now check the maximum profit & backtrack the element which is responsible for maximum profit

Maximum profit = 25, 18

<E1, E2, E3>

<1, 0, 0>

Compare and Knap cost

① Compare the working and performance of prim's and kruskal's algorithms to compute minimum cost spanning tree.

Minimum spanning Tree: A minimum spanning tree (MST) is a subgraph of a connected, weighted and undirected graph that (CPYES)

(i) connects all vertices
(ii) Has no cycle
(iii) Has the minimum possible total edge weight among all spanning trees.
(iv) In MST V vertices and |V-1| edges.

prim's Algorithm: prim's Algorithm is a greedy approach. It is used to find minimum spanning tree. It is suitable for dense graph.

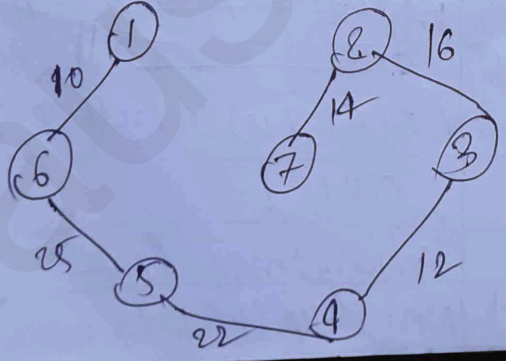
Steps of Algorithm

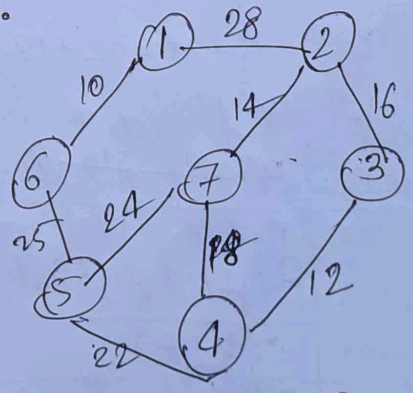
Step1: From your graph first of all you select a minimum cost edge.

Step2: Make sure that always select a minimum cost edge but should connected already selected vertices.

Step3: while selecting edge if any cycle forms discard it to add in MST.

Example: $|E| = V-1$





Cost = 10+25+22+12+16+14 = 99

* Kruskal's algorithm: Kruskal's algorithm is a greedy approach, it is used to used to find minimum spanning tree. It is suitable for sparse graph.

Steps of Algorithm

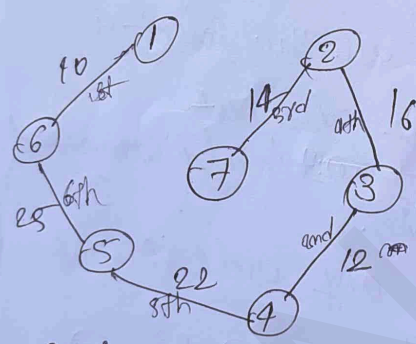
Step 1: From your graph first of all you select a minimum cost ~~tree~~ edge.

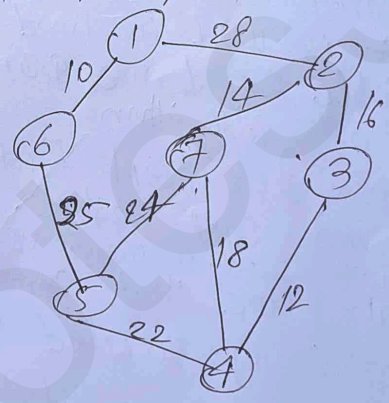
Step 2: Always select a minimum cost edge but it is not required that edge should connected to already selected edge/nodes.

Step 3: while selecting edge if any cycle forms discard it to add in MST.

Example:

Soln:





Cost = 10+25+22+12+16+14 = 99

|E| = V-1

Comparing performance of both algos.

Features	~~Prim's Algo~~ Kruskal's Algo	~~Kruskal's Algo~~ prim's algo
Best for	Dense graph sparse graph	~~sparse graph~~ Dense graph
Time com	O(E log E) (Sorting + union-find)	O(E log V) (min Heap)
Space com	O(V+E)	O(V+E)
graph representation	Edge list	Adjacency list
DS used	union-Find (Disjoint set)	Min Heap (Priority queue)

* All pair shortest

(b) (PYQ) 4M

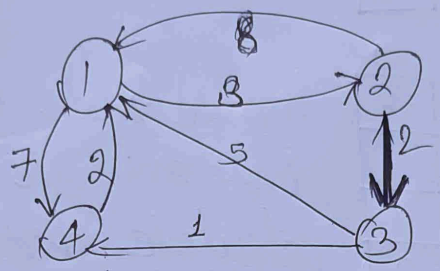
Apply dynamic programming to slove All pair shortest path problem.(Floyed warshall Algorithm).

Soln: The all-pairs shortest path problem is a fundamental problem in graph theory. It involves finding the shortest path between every pair of vertices in a weighted graph.

Floyed warshall Algorithm: Floyed warshall algorithm is a fundamental algorithm in computer science and graph theory. It is used to find the shortest

④ between all pairs of vertices in a weighted graph. This algorithm is highly efficient and can handle graphs with both negative and positive edge weights, making it a versatile tool for solving a wide range of network and connectivity problems.

problem:



A_0 =

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

~~Create vertex~~ initially the distance matrix using input graph such that distance = ∞ if there is no edge from i to j.

A_1 =

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	8	∞	0

Create vertex 1 as an intermediate node and calculate the distance

$A_0[2,3]$ $A_0[2,1] + A_0[1,3]$

2 < 8+∞

$A_0[2,4]$ $A_0[2,1] + A_0[1,4]$

∞ > 8+7

$A_0[3,2]$ $A_0[3,1] + A_0[1,2]$

∞ 5+8

and so on.

$A_2 =$

	1	2	3	4
1	0	3	5	17
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

Treate Node 1 & 2 as an intermediate node and calculate distance

$A_3 =$

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

Treate Node 1, 2 & 3 as an intermediate node and calculate distance.

$A_4 =$

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

Treate Node 1, 2, 3 & 4 as an intermediate node and calculate distance

Formula is:

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

Time complexity $:= O(V^3)$ where V is no of Vertices

Algo:

```
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])
      }
   }
}
```

Assignment-1

Name: RAUSHAN KUMAR
CRN : 2221139
URN : 2203751
Sec : IT(B2)

Q1 @ Differentiate b/w greedy algorithm and dynamic programming (2 Marks)

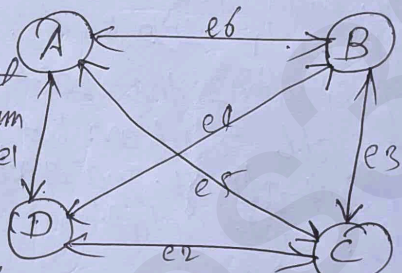
Sol^n:

Features	Greedy Algorithm	Dynamic programming
Definition	A greedy algorithm makes the best possible choice at each step in the hope of finding the best overall solution but it doesn't always guarantee the best solution.	Dynamic programming solves complex problems by breaking them into smaller overlapping subproblems, solving each subproblem once and storing the results to avoid redundant calculations. It ensures an optimal solution
Optimal solution	A greedy algo does not always guarantee an optimal solution.	It always guarantees an optimal solution.
Decision sequence	only one decision sequence.	many decision sequence
Intermediate result storage	It does not store intermediate results.	It stores results of subproblems to avoid redundancy
Time complexity	Generally faster	It takes more time
Approach type	~~bottom-up~~ ~~top-down~~ approach	~~Top-down~~ ~~Bottom-up~~ approach
Efficiency	It is more efficient than dynamic programming	It is less efficient than greedy algorithm
Example	~~0/1~~ Fractional knapsack, Minimum cost Spanning Tree.	0/1 knapsack, multi-stage problem

BABC

(b) Compare the performance of greedy method & dynamic programming for solving Travelling sales person problem (3 marks).

ans ⇒ Travelling sales man problem: In this problem a sales man visits all the cities and after sell they have to come it's starting home city. here we are considering four cities A, B, C, D where city A is the home city, and A 2-D matrix is given that consists of all the cost of all possible edges b/w cities.



Greedy: Greedy tries to find out the path/route with the minimum cost such that visiting all cities once and return back to the source city is achieved. The path through which we can achieve that, can be represented as A→B→C→D→A; Here we started from city A and ended on the same visiting all other cities once. The cost of our path/route is calculated as follows.

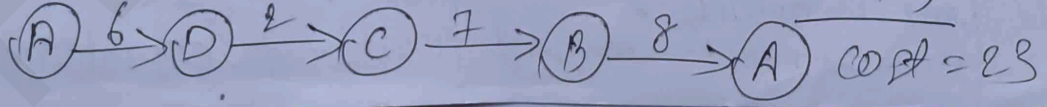
	A	B	C	D
A	∞	16	11	6
B	8	∞	13	16
C	4	7	∞	9
D	5	12	2	∞

A → D || 6
D → C || 2
C → B || 7
B → A || 8

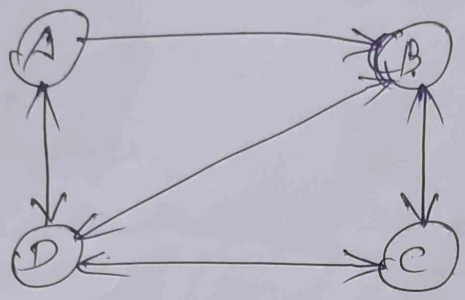
Hence total cost = 6+2+7+8 = 23

Time complexity: $O(N^2)$

Space complexity: $O(N)$



If this edge ~~from~~ e6 from B to A will be not in the graph then greedy also gets stuck hence greedy is failed, four generation failed because it has only one decision sequence hence greedy failed. (1.1)

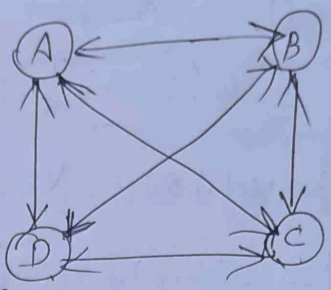


No edge from B to A node

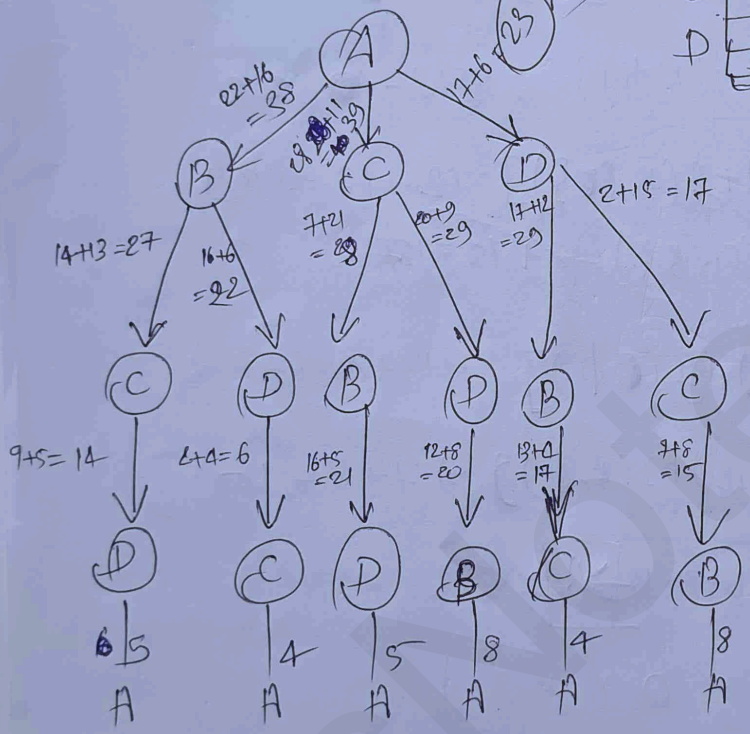
from B to A will be not in (1)

Dynamic programming :

Dynamic programming takes more than one Decision sequence to find out the optimal solution it is slow but it guarantee that it will give you one of the optimal solution. It looks for all possible solution that takes minimum cost to reach one city to another city.



	A	B	C	D
A	0	16	11	6
B	8	0	13	16
C	4	7	0	9
D	5	12	2	0



It have more than one decision sequence.

$Dp \Rightarrow g(i,S) = \min(w(i,j) + g(j,\{S-j\}))\ j \in S$

i = starting vertex
S = set of vertices
w = weight
j = another vertex

$i = A$

$g(A,\{B,C,D\}) = \min[w(A,B) + g(B,\{C,D\})] = 16+22 = 38$

$w(A,C) + g(C,\{B,D\}) = 11+28 = 39$

$w(A,D) + g(D,\{B,C\}) = 6+17 = 23$ Minimum

$g(B,\{C,D\}) = w(B,C) + g(C,\{D\}) = 13+14 = 27$

$w(B,C) + g(D,\{C\}) = 16+6 = 22$

$g(C,\{D\}) = w(C,D) + g(D,\phi) \Rightarrow 9+5 = 14$

$g(D,\{C\}) = w(D,C) + g(C,\phi) \Rightarrow 2+4 = 6$

$g(C,\{B,D\}) = \min\{w(C,B) + g(B,\{D\}) \Rightarrow 7+21 = 28$

$w(C,D) + g(D,\{B\}) \Rightarrow 9+20 = 29$

$g(B,\{D\}) = \min[w(B,D) + g(D,\phi)] \Rightarrow 16+5 = 21$

$g(D,\{B\}) = \min[w(D,B) + g(B,\phi)] \Rightarrow 12+8 = 20$

$g(D,\{B,C\}) = \min[w(D,B) + g(B,\{C\})] = 12+17 = 29$

$w(D,C) + g(C,\{B\}) = 2+15 = 17$

$g(B,\{C\}) = \min[w(B,C) + g(C,\phi)] = 13+4 = 17$

$g(C,\{B\}) = \min[w(C,B) + g(B,\phi)] \Rightarrow 7+8 = 15$

$A \rightarrow D \rightarrow C \rightarrow B \rightarrow A = 23$

Time complexity $= O(n^2 * 2^n)$

Space complexity $= O(n * 2^n)$

Now, Comparing the performance of greedy method and dynamic programming for solving Travelling salesperson problem

	TC	SC
Greedy	$O(N^2)$	$O(N)$
Dynamic Pro	$O(n^2 * 2^n)$	$O(n * 2^n)$

$T(n) = T(n-1) + \log n$ ... (2)

Back Substit.

(i) — $T(n) = T(n-1) + \log n$.

$n \to n-1$

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

Put eq. ii in i

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

Again, Put $n \to n-2$. in eq i always.

$T(n-2) = T(n-3) + \log(n-2)$ — (iv)

Put iv in (iii)

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

$T(n) = T(1) + \log 1 + \log 2 + \dots \log n$.

$= T(1) + \log[1 \times 2 \times 3 \times 4 \dots n]$ {We can write like this}

$= 1 + \log(1 \cdot 2 \dots n)$

$= 1 + \log(n!)$.

$= 1 + n\log n - n$

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

{$\log(n!) = n\log n - n$}

{$\log(1 \times 2 \times 3 \dots n) = \log(n!)$}

(a) $a_n = a_{n-1} + n$ with $a_0 = 4$

when $n=1$

$$a_1 = a_0 + 1 \quad \text{---} (1)$$

now when $n=2$, $a_2 = a_1 + 2$ — (2)

put eq (1) in (2)

$a_2 = a_0 + 1 + 2$

$a_2 = a_0 + 3$ — (3)

when $n=3$, $a_3 = a_2 + 3$ — (4)

put (3) in (4)

$a_3 = a_0 + 3 + 3$

$a_3 = a_0 + 6$

put eq (1) in (2)

$a_2 = (a_0 + 1) + 2$ — (3)

when $a_n = 3$, $a_3 = a_2 + 3$ — (4)

put eq (3) in (4)

$a_3 = \big((a_0 + 1) + 2\big) + 3$

(b) we notice a pattern. Each time we take the previous term and add the corresponding index so

$a_n = ((((a_0 + 1) + 2) + 3) + \cdots + n-1) + n$ // put $a_0 = 4$

Regrouping terms.

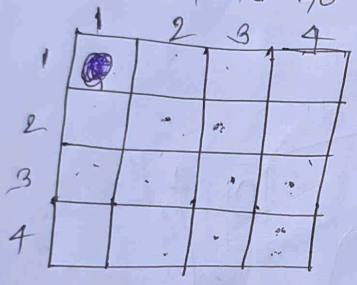
$$a_n = 4 + \frac{n(n+1)}{2}$$

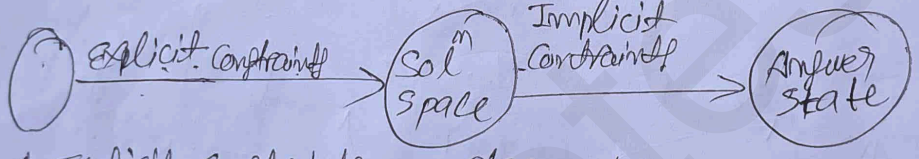
$T(n) = O(n^2)$

4 Queens problem using Backtracking:

Soln: The 4 queens problem is a smaller version of the classic N-queens problem, where the goal is to place 4 queens on a 4×4 chessboard such that no two queens threaten each other. That means no two queens should be in the same row, column or diagonal.

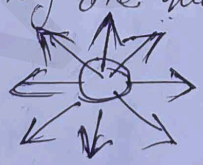
Backtracking approach: Backtracking is a recursive algorithm that tries to build a solution step by step and removes (backtracks) a step when it determines that the current path cannot lead to a valid solution.





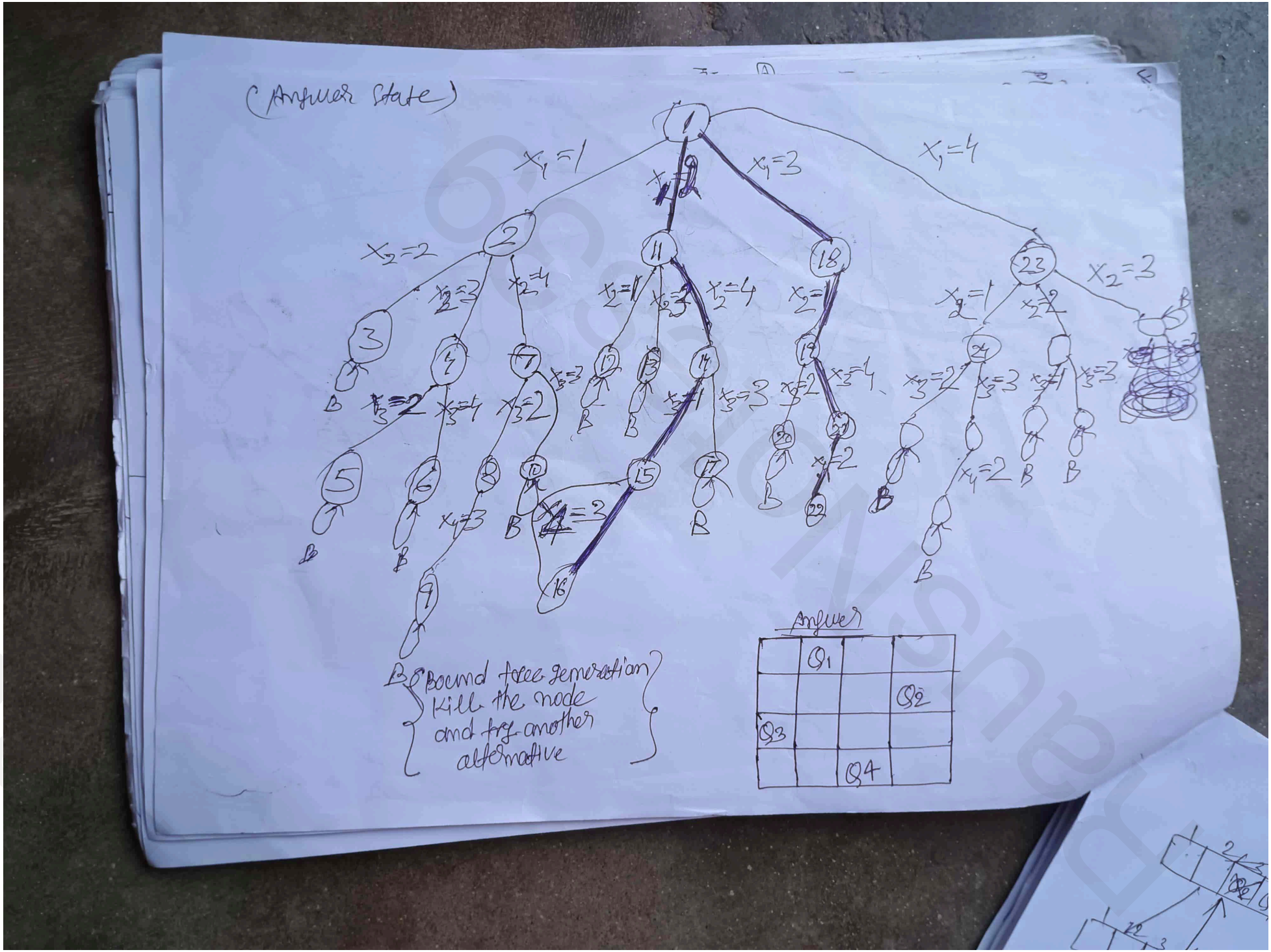
* Explicit Constraints = Place 4 queens
only one queen can be placed per row.
only one queen can be placed per col.

* Implicit Constraints =



Tree generation for solution space


depth first Search
(solution space)


(Answer State)
Bound tree generation
Kill the node
and try another
alternative
Answer
Q1
Q2
Q3
Q4

Sum of Subsets problem using Backtracking

Soln: Given a set of positive integers and a value m determine all subsets of the set whose sum is exactly m.

Constraints

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

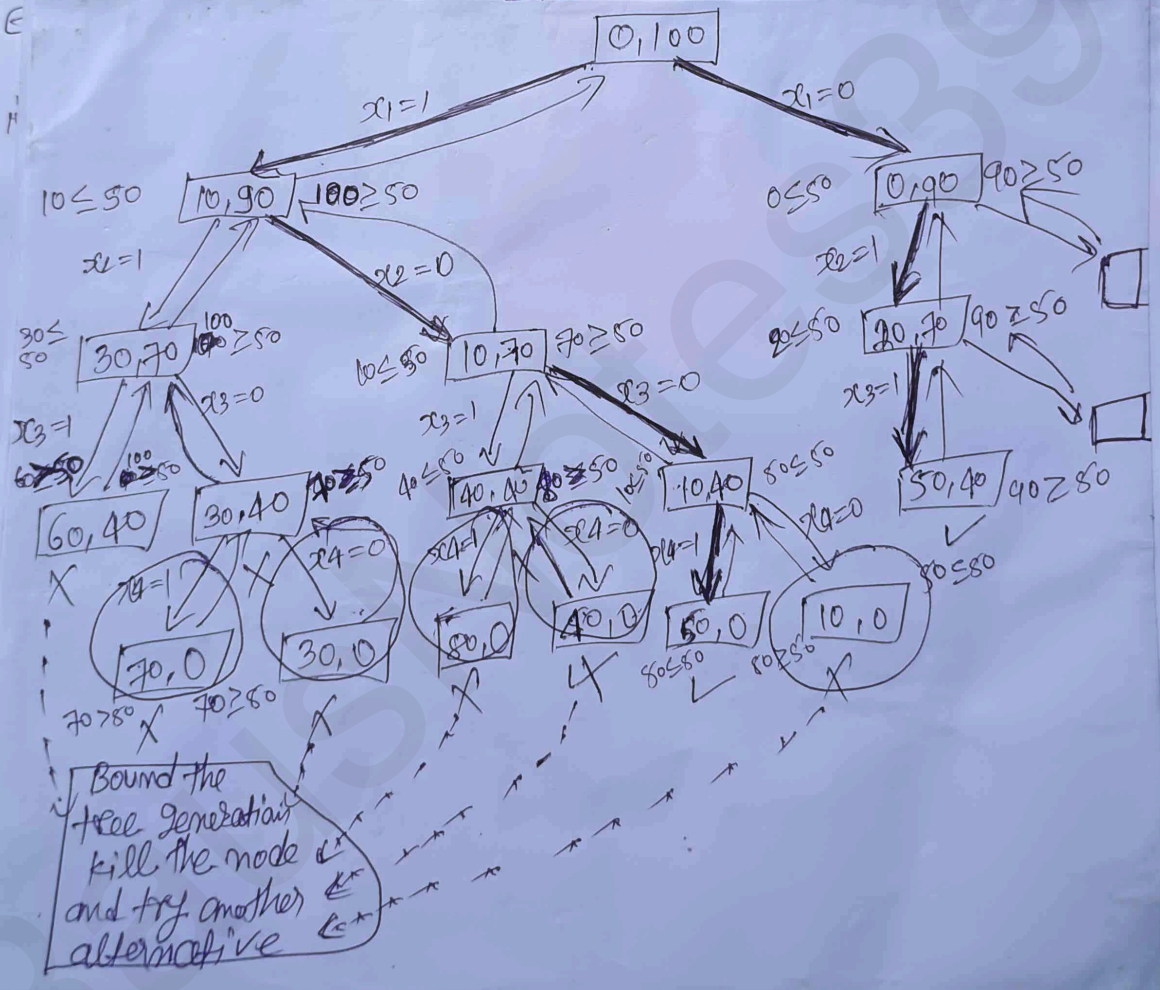
$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

$x_i = 0$ or 1
↳ no ele is selected
↳ ele is selected

problem = $\{10, 20, 30, 40\}$ $x_1\ x_2\ x_3\ x_4$ Sum = 80, m = 50

x_1	x_2	x_3	x_4	
1	0	0	1	(10+40)=50(m)
0	1	1	0	(20+30)=50(m)

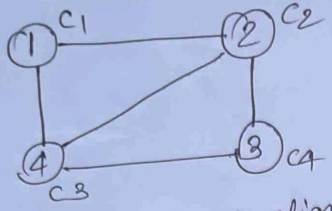
any = $\{10, 40\}$, $\{20, 30\}$



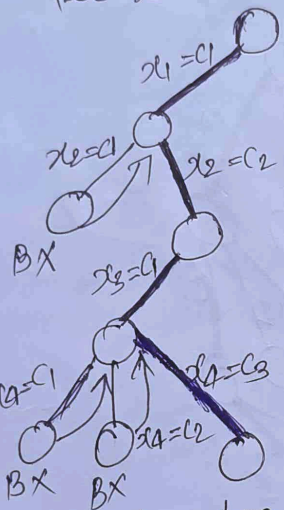
Q. Compute the chromatic number for the given graph G1 by using backtracking algorithmic design technique and demonstrate the tree-generation.

[Adjacent Nodes cannot have the same color] chromatic No = 3

(1)



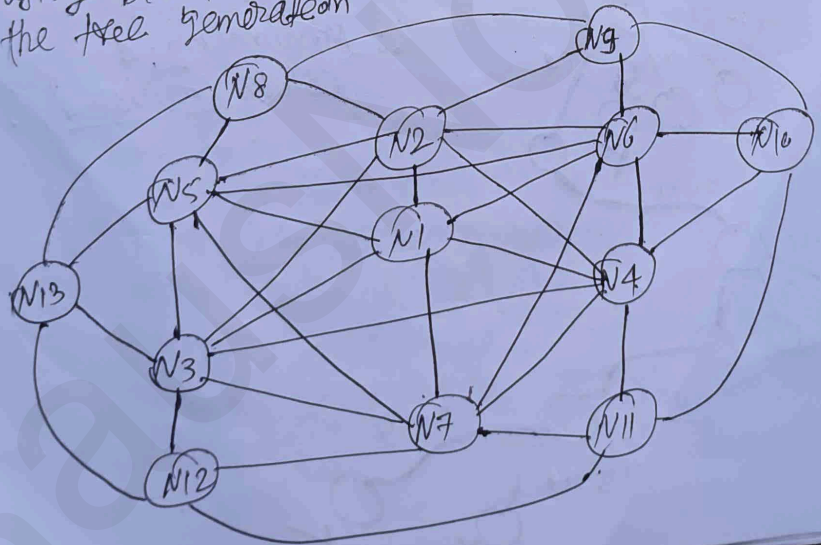
Tree generation

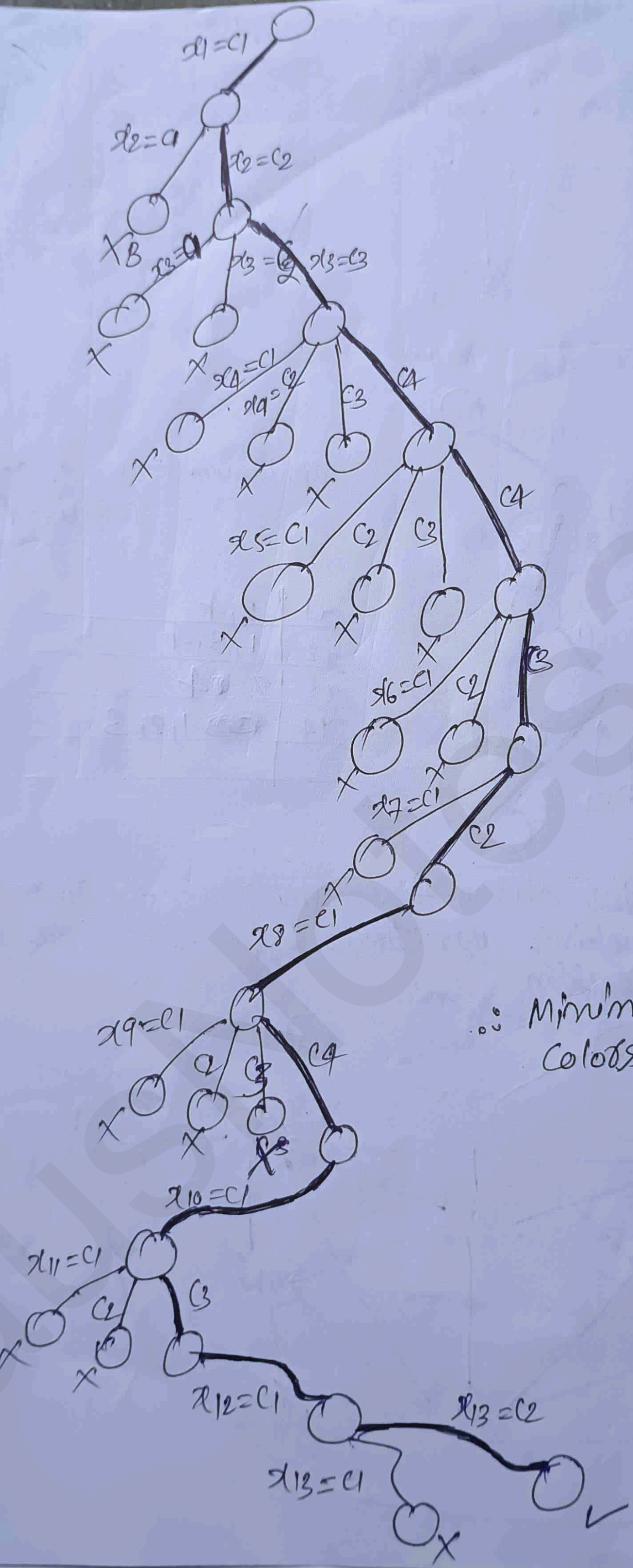


Minimum :- There colors are required to clore the graph.

x1	2,4
x2	1,3,4
x3	2,4
x4	1,2,3

(PYQ) Compute the chromatic number for the given Graph G1 by using backtracking algorithmic design technique and demonstrate the tree generation





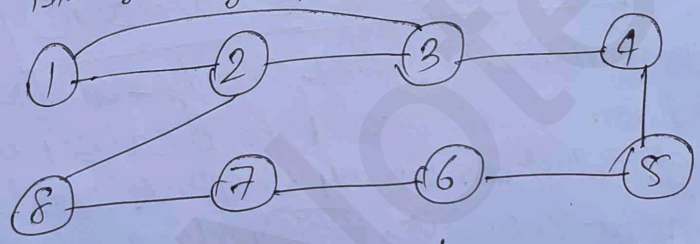
∴ Minimum 4 colors required

x1	x2, x3, x4, x5, x6, x7
x2	x1, x3, x5, x6, x8, x9, x4
x3	x1, x2, x5, x7, x12, x13, x4
x4	x1, x2, x3, x6, x7, x8, x13
x5	x1, x2, x3, x6, x7, x8, x13
x6	x1, x2, x9, x5, x7, x9, x10
x7	x1, x3, x4, x5, x6, x11, x12
x8	x2, x5, x9, x13
x9	x2, x6, x8, x10
x10	x4, x6, x9, x11
x11	x4, x7, x11, x13
x12	x3, x7, x11, x13
x13	x3, x5, x8, x12

(Pg0) Design a backtracking algorithm to find all the Hamiltonian cycles in a graph.

Hamiltonian cycle: It is a round-trip along an edges of the graph that visits every vertex once and returns to its starting position.

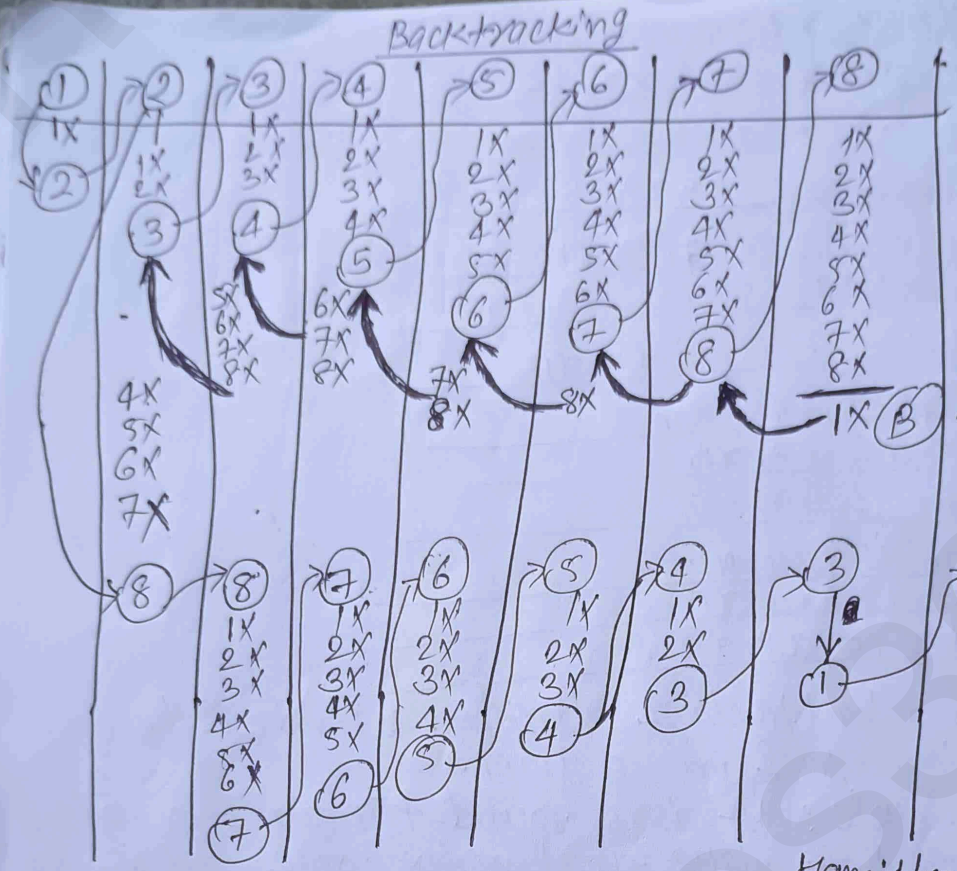
Ex ⇒



1, 2, 8, 7, 6, 5, 4, 3, 1
is the Hamiltonian cycle of the given graph

choice of Node

- Node : should not be visited earlier
- Node : should have an edge i.e must be connected.



⇒ Solution :- Hamiltonian cycle

(1), (2), (8), (7), (6), (5), (4), (3), (1)

(pyq) Access the importance of using greedy method and relaxing the condition of $x_i = 0$ or 1 to $0 \le x_i \le 1$ while computing optimal solution for 0/1 knapsack problem using a ~~recursion ba~~ recursive backtracking algorithm.

or

Elaborate how the backtracking technique is applied to solve 0/1 knapsack problem.

To access the importance of using the greedy method and relaxing the condition of $x_i = 0$ or 1 to $0 \le x_i \le 1$ in the context of computing an optimal solution to the 0/1 knapsack problem using recursive backtracking.

1. Relaxation of Constraint: $x_i = 0$ or $1 \Rightarrow 0 \leq x_i \leq 1$
This relaxation converts the 0/1 knapsack problem into the fractional knapsack problem, which is
- Easier to solve and
- solvable optimally using a greedy method

2. Use of Greedy Method in Recursive Backtracking

In the 0/1 knapsack problem, greedy alone doesn't always give the optimal solution, but greedy (with relaxation) help in bounding and pruning the recursive search space:

Importance in Recursive Backtracking

- The greedy solution to the fractional knapsack (relaxed problem) provides an upper bound on the maximum value that can be achieved from the current state.
- During recursive backtracking, if the upper bound is less than or equal to the best solution already found, that path can be pruned.

Elaborating how the backtracking technique is applied to solve 0/1 knapsack.

Backtracking is a problem-solving technique that explores all possible solutions to a problem by building a solution incrementally and abandoning a path as soon as it determines that the path cannot lead to a valid or optimal solution. In the 0/1 knap-sack problem, backtracking is used to explore all subsets of items to find the maximum value that can be carried in the knapsack without exceeding its capacity.

0/1 knapsack problem (using Backtracking)

To solve 0/1 knapsack problem using backtracking we need upper bound value i.e expected profit. UB (upper bound) is computed using greedy method (& allowing fractional values)

Eg:=>

P	45	30	45	10
W	3	5	9	5
P/W	15	6	5	2

m = 16
UB = 115

P: 45 + 30 + (45 × 8/9) = 115
W: (3) (5)

greedy Method

Tree generation for backtracking

Note: Breadth first Search

1
Cp = 0
CW = 0, UB = 115

Cp = current profit
CW = current weight
UB = upper Bound

$x_1 = 1$ $x_1 = 0$

2
Cp = 45, CW = 3
UB = 115

3
Cp = 0, CW = 0
UB = 79

recompute UB by not including x_1

115 > 79

proceed where max profit is expected

$x_2 = 1$ $x_2 = 0$

4
Cp = 75, CW = 8
UB = 115

5
Cp = 45, CW = 3
UB = 98

98 > 85 > 79 (proceed where max profit is expected)

115 > 98 > 79

$x_3 = 1$ $x_3 = 0$ $x_3 = 1$ $x_3 = 0$

6
Cp = 120, CW = 17
UB = 115

(B) Bound ✗ exceeds m = 16 the Capacity

7
Cp = 75, CW = 8
UB = 85

Cp = 90, CW = 12
UB = 98

Cp = 45, CW = 3
UB = 55

98 > 85 > 79 > 55

$x_4 = 1$ $x_4 = 0$

Cp = 100
CW = 17
UB = 98

(B) Bound ✗ exceeds m = 16

Cp = 90, CW = 12
UB = 90

Answer ✓

Answer :-
<x1, x2, x3, x4>
<1, 0, 1, 0>
final profit = 90
final weight = 12

String Matching Algorithms.

① Brute force:

ⓘ Text : A B A B A B C C A (represented by [n] = 9)
pattern : A B A B C → Not match (represented by [m] = 5)
Compare 1 2 3 4 5X

Once mismatch occure it will move one step ahead.

T: A B A B A B C C A
— A B A B C
6X again mismatch
Again move one step ahead

T: A B A B A B C C A
— — A B A B C → final Answer
No of Comparison 7 8 9 10 11

T:	A	B	A	B	A	B	C	C	A	
P:	A	B	A	B	C					mismatch
P:	—	A	B	A	B	C				again mismatch
P:	—	—	A	B	A	B	C			matched found

Total No of comparison = 11
Complexity = o(n * m) o(nm)
if $m = \frac{n}{2}$ then time complexity = $o(n * \frac{n}{2})$ $O(n^2)$

It is inefficient method (multiply)
those who are efficient (Addition)

* Rabin-karp Algorithm : O(n+m) time complexity

T: 2 3 5 9 0 2 3 1 4 1 5 --- n
P: 3 1 4 1 5 m = 5 window size

Step 1: It takes any prime no $\Rightarrow q = 13$ it compile $r = p \bmod q$
31415 mod 13
r = 7 remainder

Step ②: Take text of same window size
23590 mod 13 = 8 8 ≠ 7 Not proceed

Step 3: shift the window by 1
2 [3 5 9 0 2] 3 1 4 1 5
shifting of window : ~~Horners rule~~ Horner's rule

old No: (23590 − 10,000 × 2)

⇒ 3590 × 10 + 2 ← Add element No that you want

⇒ Temt window size = 35902

Step4: 35902 mod 13 = 9

7 ≠ 9 temt window ≠ pattern window

shift ~~for~~ the window by 1 - Apply ~~Hoor~~

(35902 − 10,000 × 3)

5902 × 10 + 3 =

~~59023~~ 59023

new temt window = 59023

Step5: Again 59023 mod 13 = .

... will continue till 31415

Step6: Compare and Matches found

Rabin Karp Algorithm:

T: B A C B A A
p: B A A

Step1: Take any prime no q and compute $\gamma = p \bmod q$ where p is pattern [q = 5381]

B A A.

B(2) - 2
A(1) - 1
A(0) - 1
] 26

$((2\times26^2) \bmod 5381) + ((1\times26^1) \bmod 5381) + ((1\times26^0) \bmod 5381))$

$\Rightarrow 1352 + 26 + 1 = 1379$ $\quad \gamma = 1379$

Step2: Take the substring from text with the same window size. and compute $\gamma = str \bmod q$

str = B A C

$\gamma = (((2\times26^2) \bmod 5381) + ((1\times26^1 \bmod 5381)) + ((3\times26^0) \bmod 5381))$

$\gamma = 1352 + 26 + 3 = 1381$

Step3: Compare value of γ of text and pattern. If not matches then Apply Horner's rule and shift character by 1 in Text. and compare γ.

$\gamma(P) = 1379$
$\gamma(str) = 1381$

$1379 \neq 1381$

∴ Horner's rule $= \left(1381 - (2\times26^2)\right)\times26 + (2\times26^0)$

AC = BAC - B

$\Rightarrow 756$

Step 4: Compare $756 \neq 1379$

Again Apply Horner's rule

$\left((756 - (1\times26^2))\times26 + (1\times26^0)\right) = 2081 \neq 1379$

CB + A

$$\underbrace{\left(\left(2081-(3\times 26^{2})\right)\times 26+1\times 26^{0}\right)}_{BA} + \underbrace{\quad}_{A}$$

$\Rightarrow 1379 = 1379$ Match found

Q. T: 23590231415267399 21 and

I P: 31415

Ans $\Rightarrow$ $Q = 13$ $r = p \bmod Q = 31415 \bmod 13$

$r = 7$

$r = Str \bmod Q \Rightarrow r = 23590 \bmod 13$

$r = 8$

$r(T) \neq r(Str) \Rightarrow 7 \neq 8$

apply Horner's rule $\Rightarrow$

$(23590 - 2\times 10000)\times 10 + 2 = 35902$

$r = 35902 \bmod 13 \Rightarrow 9 \neq 7$ not match

again $\Rightarrow$ $(35902 - 3\times 10000)\times 10 + 3 = 59023$

$r = 59023 \bmod 13 \Rightarrow 3 \neq 7$ No Match

again $\Rightarrow$ $(59023 - 10000\times 5)\times 10 + 1 = 90231 \bmod 13 \Rightarrow 1 \neq 7$

again $\Rightarrow$ ~~59023~~ $(90231 - 9\times 10000)\times 10 + 4) = 02314$

$r = 02314 \bmod 13 = 0 \neq 7$ No match

$\Rightarrow (02314 - 0\times 10000)\times 10 + 1) = 23141 \bmod 13 = 1 \neq 7$

$\Rightarrow (23141 - 2\times 10000)\times 10 + 5 \Rightarrow 31415$

$r = 31415 \bmod 13 = 7 = 7$ Match found

Q2 (PYQ) Compare the Boyer Moore and Boyer Moore Hospool Algorithm.

Ans → Boyer Moore Algorithm: The Boyer-Moore algorithm is a powerful and efficient string matching/ Searching algorithm used to find occurrence of a pattern within a text.

Boyer moore uses 2 techniques
(i) Bad character Heuristic
(ii) Good Suffix Heuristic

Steps of Boyer moore algorithm

T: c a t n p c a n d y

P: c a n d y

T be the string of length n

P be the string of length m

1. Start matching from right: Compare the pattern p to a window of T but start comparing them from the end of pattern.

T: c a t n p c a n d y

P: c a n d y

(2) Mismatch use Heuristic to shift

Bad character rule: Move the pattern so that the mismatched character in T aligns with the last occurrence of that character in P or move part if it doesn't exists in P.

After first shift.

1. T[5] = c, P[0+1] = a → mismatch

T: c a t n p c a n d y
p: _ _ _ _ _ _ c a n d y

Here we get a mateh, if mateh does not occurs.

3. Repeat the process untill the pattern is found.

* Good suffix Rule: If a suffix of p matched before a mismatch, shift p so that this suffix matches another occurrance of the same suffix or the prefix of p.

T: c a t c a p c a n d y
p: c a n d y

After first shift

T: c a t c a p c a n d y
p: _ _ _ c a n d y

Again Apply Good suffix rule

T: c a t c a p c a n d y
p: c _ _ _ _ _ c a n d y

After third shift we get the matched pattern.

Time Complexity
Best/Avg case: O(n/m)
worst case: O(n+m)

Booyer - Moore Horspoal Algorithm

It is a simplified and faster variant of the Booyer moore algorithm that only uses the Bad character Heuristic but in a much more optimized way.

Algorithm steps

(i) primarily make Bad match Table for every character in the pattern except the last one.

(ii) Compare pattern to text, starting from rightmost character in the pattern.

(iii) If mismatch, move pattern forward corresponding to value in the table.

Text : t r u s t h a r d t o o t h b r u s h e s

pattern: t o o t h length = 5

Construct Bad match Table

Value = length - index - 1

t = 5 - 0 - 1 = 4
o = 5 - 1 - 1 = 3
t = 5 - 2 - 1 = 2
o = 5 - 3 - 1 = 1
h = 5 - 4

t o o t h
0 1 2 3 4

Letter	t	o		*
value	2	1		5 ← always length value

T : T R U S T H A R D T O O T H B R U S H E S

P : T O O T H → mismatch see T value from table and shift p value times.

T : T R U S T H A R D T O O T H B R U S H E S

P : _ T O O T H

mismatch S = Not Present in the table then use value of * = 5 shift by 5.

T : T R U S T H A R D T O O T H B R U S H E S

P : T O O T H mismatch value of O = 2 shift by 2

T: TRUSTHARDTOOTHBRUSHES
P: ----------TOOTH

pattern matched at idx = 9

Time Complexity
worst case = O(n * m)

(PYQ) Compare the performance of Boyer-Moore algorithm and Boyer-Moore Horspool algorithm.

Feature	Boyer Moore Algo	Boyer Moore Horspool Algo
Heuristics used	Bad character + Good suffix Heuristics	Only Bad character Heuristics
Implementation complexity	More complex due to multiple heuristics	simple and easier to implement due to only one.
Worst case performance	O(n+m)	O(n*m)
prefix rule	Yes	No
Use in case suitability	ideal for large texts and patterns with few repeats	Best for long patterns and large alphabet
Practical performance	Very fast but harder to Code.	slightly slower than BM but easy to code.

(PYQ) Examine how Boyer-Moore algorithm is working efficiently than Boyer moore Horspool algorithm for the worst case time complexity.

The Boyer-Moore and Boyer-Moore Horspool algorithm are efficient string matching algorithms, but Boyer-Moore is generally more efficient than Boyer-Moore-Horspool in the worst case time complexity. due to its use of two heuristics rather than only one.

Differences in Efficiency

Algorithm	Heuristics used	Worst case Time complexity
Boyer-Moore	Bad character + Good suffix	O(n+m)
Boyer Moore Horspool	Bad character	O(n*m)

Now here I am going to show that how Boyer-Moore algorithm is working efficiently than Boyer Moore Horspool using example → using Horspool algorithm

Example : Text : a a a a a a a a a a a a a a a b

pattern : a a a a b n = 5

Construct Bad match table

Value = length - index - 1

a : 5 - 0 - 1 = 4
a : 5 - 1 - 1 = 3
a : 5 - 2 - 1 = 2
a : 5 - 3 - 1 = 1

Letter	a	*
Value	4 3 2 1	5

T : a a a a a a a a a a a a a a a b (18)

P : a a a a b mismatch look a in the table and shift p value times

T : a a a a a a a a a a a a a a a b

a a a a b → mismatch shift by 2

T : a a a a a a a a a a a a a a a b

a a a a b mismatch shift by 2

T : a a a a a a a a a a a a a a a b

a a a a b

T : a a a a a a a a a a a a a a a b

a a a a b

T : a a a a a a a a a a a a a a a b

a a a a b

Text: aaaaaaaaaab
pattern: aab

① Construct bad match table
value = length - index - 1

Letter	a	b
value	1	3

len = 3
a: 3 - 0 - 1 = 2
a: 3 - 1 - 1 = 1

T: aaaaaaaaaab
P: aab mismatched look a in the table and shift 'P' value times. shift by 1.

pattern

T: aaaaaaaaaab
P: aab shift by 1

T: aaaaaaaaaab
P: --aab shift by 1

T: aaaaaaaaaab
P: ---aab shift by 1

T: aaaaaaaaaab
P: ----aab shift by 1

T: aaaaaaaaaab
P: -----aab shift by 1

T: aaaaaaaaaab
P: ------aab shift by 1

T: aaaaaaaaaab
P: -------aab shift by 1

T: aaaaaaaaaab
P: --------aab match found

Some example using Boyer moore algorithm
// using Bad character rule

T: a a a a a a a a a b
P: a a b mismatched

T: a a a a a a a a a b
P: — — a a b mismatched

T: a a a a a a a a a b
P: — — — — a a b mismatched

T: a a a a a a a a a b
P: — — — — — a a b mismatched

T: a a a a a a a a a | a a b
P: — — — — — — — — — | a a b Match found

Horspool

Since, Boyer Moore have taken 9 steps and 11 comparisons whereas Boyer Moore (using Bad character) have taken 5 steps and 7 comparisons. Therefore we can say that Boyer Moore is more efficient than Boyer Moore horspool algorithm in worst case time complexity.

Q1. Demonstrate the working of KMP algorithm with the help of an example.

ans→ The KMP (Knuth-Morris-Pratt) algorithm is an efficient pattern searching algorithm that avoids unnecessary comparisons in a string search by pre-processing the pattern using a Longest prefix suffix array (LPS).

LPS: For the pattern, it stores the length of the longest proper prefix which is also a suffix.

Example: String : a b a b c a b c a b a b d

pattern : a b a b d

Step 1: Compute the LPS (Longest prefix suffix) Array.

pattern:

	1	2	3	4	5
	a	b	a	b	d
LPS = 0	0	0	1	2	0

Step 2: pattern matching

Use two pointers: i = 1, j = 0

Match step by step:

Text = a b a b c a b c a b a b d

i = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

	j=0	1	2	3	4	5
pattern =		a	b	a	b	d
LPS =		0	0	1	2	0
T[i] =		P[j+1]				

1. T[1] = a, P[0+1] = a → match → move i = 2, j = 1
2. T[2] = b, P[1+1] = b → match → move i = 3, j = 2
3. T[3] = a, P[2+1] = a → match → move i = 4, j = 3
4. T[4] = b, P[3+1] = b → match → move i = 5, j = 4
5. T[5] = c, P[4+1] = d → mismatch → move j = LPS[j] = LPS[4] = 2
6. T[5] = c, P[2+1] = a → mismatch → move j = LPS[j] = LPS[2] = 0
7. T[5] = c, P[0+1] = a → mismatch → So at j = 0 again mismatch

We cannot move j back further so move i = 6

T[6] = a, P[0+1] = a → match → move i = 7, j = 1

T[7] = b, P[1+1] = b → match → move i = 8, j = 2

T[8] = c, P[2+1] = a → mismatch → move j = TSP[j] = TSP[2] = 0

T[8] = c, P[0+1] = a → mismatch → ~~move~~ since j cannot move back further hence move i = 9

T[9] = a, P[0+1] = a → match → move → i = 10, j = 1

T[10] = b, P[1+1] = b → match → move → i = 11, j = 2

T[11] = a, P[2+1] = a → match → move → i = 12, j = 3

T[12] = b, P[3+1] = b → match → move → i = 13, j = 4

T[13] = a, P[4+1] = d → mismatch → move j = TSP[4] = 2

T[13] = a, P[2+1] = a → match → move i = 14, j = 3

T[14] = b, P[3+1] = b → match → move i = 15, j = 4

T[15] = d, P[4+1] = d → match → move i = 16, j = 5

So here end of string and end of pattern and we get our pattern in string

pattern = ababd

match found at ⇒ i - j = 16 - 5 = 9

NP Completness and Approximation Algo

Term & terminology

(1) Polynomial Time problem(P): A problem is called a polynomial time problem if it can be solved by an algorithm whose running time is a polynomial (n, n^2, n^3 --)

Ex⇒ Selection sort, Binary Search

(2) Non-polynomial Time problem: A non-polynomial time problem is one ~~where the~~ who takes exponential time complexity $2^n, n!$ to solve a problem.

example ⇒ Travelling salesman problem.

(3) Deterministic Algorithm: A deterministic algorithm is one that, for a given input always produces the same output and follow the same sequence of steps.

Example: Binary search.

(4) Non Deterministic Algorithm: A non-Deterministic algorithm can make guesses and explore multiple computation paths simultaneously, it's theoretical and mainly used in the context of NP problems.

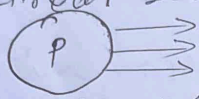
(5) Decision problem: A ~~per~~ decision problem is one where the answer is simply Yes or No.

(6) Optimization problem: An optimization problem involves finding the best solution among many possible options.

* P problem: A problem is said to be polynomial problem (P) if it can be solved in polynomial time using deterministic algorithm like ($O(n^k)$) where k is constant. polynomial problems can be solve and verify in polynomial time.

Example: Linear search, binary search, merge sort.

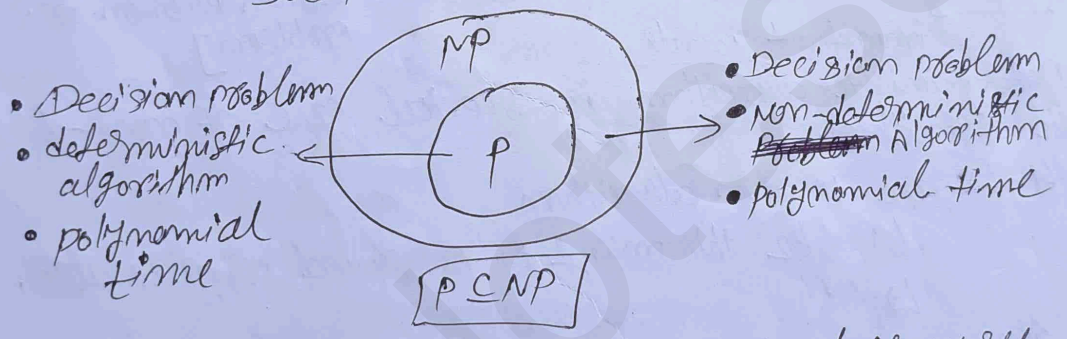
$P \subseteq NP$



not

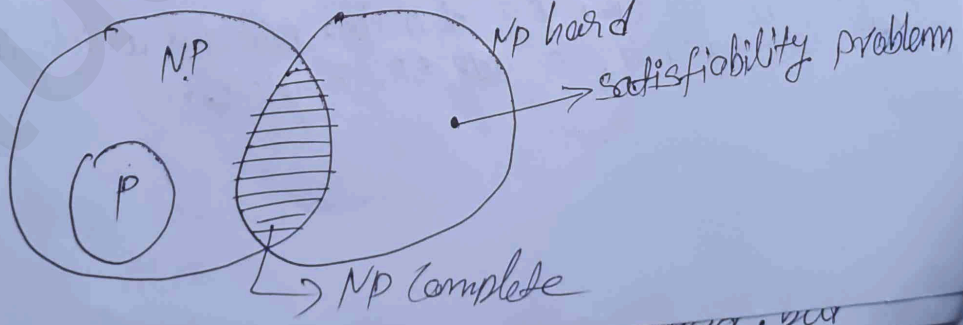
* NP problem: A problem that can be solved in polynomial time but it can be verifiable in polynomial time using non-deterministic algorithm is known as NP (Non Deterministic polynomial) problem.

Example: TSP (Travelling salesman problem)
Sudoku



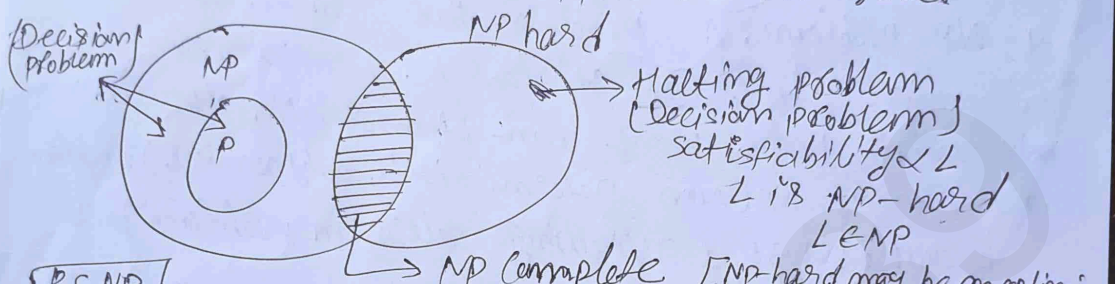
* NP-hard: A problem is NP-hard if every problem in NP can be reduced to it in polynomial time. NP-hard problems may or may not be in NP.

Ex ⇒ Halting problem, Travelling salesman problem



* NP-Complete : A problem is NP-Complete if
1. It is in NP.
2. It is NP-hard.

These are the hardest problems in NP, and if we can solve any NP-Complete problem in polynomial time, we can solve all NP problems in polynomial time. Ex→ subset sum, Hamiltonian cycle.



important points

- Deterministic algorithm is special case of non-deterministic algorithm.
- L1, L2 are known algorithm
  L1 ∝ L2 this means L1 is solved using L2 algo.
  L2 ∝ L3
  then L1 ∝ L3 (Transitive relation)
- A Decision problem may belongs to NP-Complete class.
- An optimization problem can never belongs to NP-Complete class. ~~L1 ∝ L2~~
- Eg: Halting problem is a decision problem which is NP-hard but can never be a NP-Complete..

(Q9) Judge the correctness of the statement that "Np hard decision problem is always NP-complete" If you judge it as correct or incorrect, give a detailed evidence to support your answer.

Statement: "Np-hard decision problem is always NP-Complete"

Judgement: The statement is incorrect.

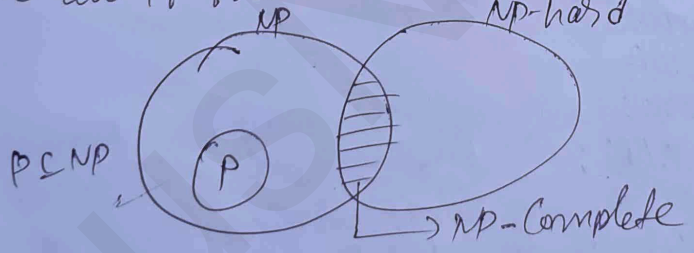
Detailed explanation: To understand why the statement is incorrect, here are the definitions of both.

~~(1) NP (Nondeterministic Polynomial)~~

(i) Np-hard: A problem is NP-hard if every problem in NP can be reduced to it in polynomial time. Np hard problem may or may not be in NP.

(ii) NP-complete: A problem is NP-complete if
- (i) it is in NP.
- (ii) it is NP-hard

These are the hardest problems in NP, and if we can solve any NP complete in polynomial time, we can solve all NP problems in polynomial time.



Key point: NP-Complete problems is a subset of NP-hard problem, that means

- All the NP-complete problems are NP-hard, but

• Not all NP-hard problems are NP-Complete.

Evidence of @ incorrectness.

Halting problem.

Halting problem is a decision problem which is NP-hard but can never become a NP-Complete.

Halting problem: Halting problem is a NP-hard problem. It is a decision problem which answers whether a machine will halt or not. i.e. the machine will halt or enter into a infinite loop. The halting problem for an arbitary deterministic algorithm and an input I is undecidable [satisfiability ∝ L, L is NP-hard ∴ satisfiability ∝ Halting problem ∴ Halting pro is a NP-hard problem]

There exists no algorithm to solve it. It cannot be solved in polynomial time.

for problem to be NP-complete. It should be a NP-hard and NP.

Algo → For input I, find the required answer if found halt otherwise enter into infinite loop.

```
Algo(I)
{
    do {
        compute
        request
        break;
    } while(1);
}
```

∴ Halting problem is NP-hard But It is unsolvable in polynomical time.

It cannot be NP

Therefore It can never be NP-complete.

Conclusion: The statement is incorrect because an NP-hard decision problem is not necessarily NP complete. It must also be in NP to be NP-complete.

* Approximation Algorithm: An approximation Algorithm is a way of dealing with NP-completeness for an optimization problem. This technique does not guarantee the best solution. The goal of the approximation algorithm is to come as close as possible to the optimal solution in polynomial time. Such algorithm are called approximation algorithms or heuristic algorithm.

Features of Approximation Algorithm

(i) An approximation algorithm guarantees to run in polynomial time though it does not guarantee the most effective solution.

(ii) An approximation algorithm guarantee to seek out high accuracy and top quality solution (within 1% of optimum)

(iii) Approximation algorithm are used to get an answer near the (optimal) solution of an optimization problem in polynomial time.

Ex→ Travelling salesman problem
Subset Sum problem.

1. Absolute Approximation Algorithm: An algorithm is an absolute approximation algorithm if difference between the optimal solution value and the approximate solution value is always less than or equal to a constant K, for all problem instances.

Mathematically.

For all instance I of a problem P:

$$|f^*(I) - \hat{f}(I)| \leq k$$

where $f^*(I)$ = optimal solution value
$\hat{f}(I)$ = solution value by the algorithm
k = some fixed constant

Example: Let's say we have 3 items in a knapsack problem:

profits: {20, 10, 19}
weights: {65, 20, 35}
Max weight(capacity) = 100

A feasible solution is:
$(x1, x2, x3) = (1, 1, 0) \rightarrow$ Total profit = 30

But the optimal solution is:
$(x1, x2, x3) = (1, 0, 1)$ - Total profit = 39

So, the difference = 39 − 30 = 9

f(n)-Approximate Algorithm (Illustrate the concept of f(n) approximation algorithm).

f(n)-Approximation Algorithm: An algorithm is an f(n)-approximate algorithm if the relative error between the approximate and optimal solution is bounded by a function f(n) (dependent on input size n).

Mathematically:

$$\frac{|F^*(I) - \hat{f}(I)|}{F^*(I)} \leq f(n) \quad \text{for } F^*(I) > 0$$

Example: optimal value $F^*(I) = 100$ input size $n = 10$
Approximate value $\hat{f}(I) = 85$

Then, $\frac{|100 - 85|}{100} = 0.15$ which is $\leq f(10)$

if $f(10) = 0.2$, the algorithm is f(n)-approximate.

ε-Approximation algorithm
i.e. Approx

ε-Approximate Algorithm (what are ε-approximate algorithms).

ε-Approximate Algorithm: An algorithm is an ε-approximate algorithm if

$$\frac{|F^*(I) - \hat{F}(I)|}{F^*(I)} \leq \varepsilon$$

where ε is a constant such that $0 < \varepsilon < 1$

$f(n) \leq \varepsilon$

Example: If $\varepsilon = 0.3$ and optimal value is 50, then approximate value must be at least 35 for maximization:

$$\left|\frac{50 - \hat{F}(I)}{50}\right| \leq 0.3 \Rightarrow F(I) \geq 35$$

$\left[\frac{35}{50} \times 100 = 70\right]$ { This ensures the approximate solution is within 70% of the optimal }

(a) prove the equation (1): for an instance I of the scheduling problem involving:

number of processors $m = 3$

number of tasks $n = 7$

task times $(t1, t2, t3, t4, t5, t6, t7) = (5, 5, 4, 4, 3, 3, 3)$

Consider $F^*(I)$ be the finish time of an optimal schedule and $F^\wedge(I)$ be the finish time of an Longest processing Time (LPT) schedule.

$$\frac{|F^*(I) - F^\wedge(I)|}{F^*(I)} \leq \frac{1}{3} - \frac{1}{3m} \quad \longrightarrow (1)$$

Sol: For an instance I of the scheduling problem involving

- number of processors $m = 3$
- number of tasks $n = 7$
- task times $(t1, t2, t3, t4, t5, t6, t7) = (5, 5, 4, 4, 3, 3, 3)$

Considering $F^*(I)$ be the finish time of an optimal schedule and $F^\wedge(I)$ be the finish time of a longest processing Time (LPT) schedule.

Steps to slove:

(1) Calculate $F^*(I)$ ~~optimal~~ : we can found it by considering lower bounds.

- Lower bound 1: sum of all tasks ~~times~~ divided by the number of processors.
- Lower bound 2: The maximum task time
- For optimal scheduling, we aim to balance the load among processors as much as possible.

(2) calculate $F^\wedge(I)$: Sort the tasks in ~~decreas~~ descending order of their processing times.

- Assign tasks to the processor that becomes free earliest

(3) Substitute the values into the inequality and verify.

Step 1: calculate $F^*(I)$ (optimal makespan)

Total processing time $\sum t_i = 5+5+4+4+3+3+3 = 27$

• lower bound 1 : $\frac{\sum t_i}{m} = \frac{27}{3} = 9$

• lower bound 2 (maximum task time) $\max(t_i) = 5$

So $F^*(I) \geq 9$ Let's ~~try~~ to construct a schedule that achieves a makespan of 9.

We have 3 processors (P1, P2, P3)

Tasks : 5, 5, 4, 4, 3, 3, 3

Let's try to distribute tasks to achieve 9 on each processor

P1 : 5 + 4 = 9

P2 : 5 + 4 = 9

P3 : 3 + 3 + 3 = 9 This is a valid schedule with a makespan of 9 Therefore $\boxed{F^*(I) = 9}$

Step 2 : Calculate $F(I)$ (LPT Makespan)

Sort tasks in descending order : (5, 5, 4, 4, 3, 3, 3)

initialize processor loads : P1 = 0, P2 = 0, P3 = 0

1. Assign task 5:
   - P1 = 5
   - P2 = 0
   - P3 = 0

2. Assign task 5:
   - P1 : 5
   - P2 : 5
   - P3 : 0

3. Assign task : 4
   - P1 : 5
   - P2 : 5
   - P3 : 4

④ Assign task : 4 (P3 is free first)
   - P1 : 5
   - P2 : 5
   - P3 : 4 + 4 = 8

5. Assign task = 3 P1 is free first after 5 cmut
   - P1 : 5 + 3 = 8
   - P2 = 5
   - P3 = 8

6. Assign task : 3 P2 is free first after 5 cmut
   - P1 : 8
   - P2 : 5 + 3 = 8
   - P3 : 8

7. Assign task : 3 P1, P2, P3 are all at 8, so any can take it.
   - P1 : 8
   - P2 : 8
   - P3 : 8 + 3 = 11

So make span for LPT schedule is $F(I) = 11$

Step 3 : subsitute and verify the inequality :

The inequality is : $\frac{|F^*(I) - F(I)|}{F^*(I)} \leq \frac{1}{3} - \frac{1}{3m}$

LHS : $\frac{|9 - 11|}{9} = \left|\frac{-2}{9}\right| = \frac{2}{9}$, R.H.S : $\frac{1}{3} - \frac{1}{3 \times 3} = \frac{1}{3} - \frac{1}{9} = \frac{2}{9}$

$\boxed{\frac{2}{9} \leq \frac{2}{9}}$ This is true Therefor, the equation (1) is proven for this specific instance

(Q5) State and Explain Cook's theorem.

Statement of Cook's theorem: "The satisfiability problem (SAT) is NP Complete".

Means, every problem in NP class can be reduced in polynomial time to the Boolean satisfiability problem.

$$\boxed{P = NP}$$

Explanation:

what is SAT: SAT means can we give true/false values to variables in a Boolean expression so that whole expression becomes true?

Ex ⇒ (X OR NOT Y) AND (NOT X OR Y)

if x = true, y = true the expression become true so this expression is satisfiable.

Cook's theorem says that: The SAT problem is the first NP-complete problem. This means if you can solve SAT in polynomial time, you can solve all NP problems in polynomial time".