Unit - 1
Problem Solving and Basics of Algorithm

Page No.
Date

- Notion of an Algorithm - finite set of instr^n which performed particular task.

problem
↓
Algorithm
↓
Input → [Computer] → output

- properties of Algorithm
1) Input - zero or more quantities are externally supplied
2) output - At least one quantity is produced.
3) Definiteness - each inst^n is clear & unambiguous
4) Effectiveness - Every instr^n must be very basic.
5) Finiteness - In all cases, the algo must terminates after a finite no. of steps.

* Q Define tractable problem and Intractable problem?

classification of problems
a) P class (polynomial time)
b) NP class (Non deterministic)
c) NP-C class (NP-complete)

1) Tractable - Solved (can be) by a polynomial-time algo.
~~polynomial fn~~ Tractable problems can run in a reasonable amount of time, even for very large amounts of input data.

2) Intractable - Cannot be solved by polynomial-time algo, and have lower bound that is exponential.
- Intractable problems require huge amt of time, even for modest input sizes.

Problem solving principles -

a. Identify the problem
b. Understand the problem
c. Identify alternative way to solve the problem
d. select the best way to solve the problem from the list of alternative solutions.
e. evaluate the solution.

Classification of Time complexity.

Time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input.

Time complexity depends on execution time, where execution time is length of the input

length of the input indicates the number of operⁿ.
- Time complexity depends on length of the input. it gives info about variation (increase or decrease) in execution in time

Types of time complexities

1) Constant time - $O(1)$
2) Linear time - $O(n)$
3) Logarithmic time - $O(\log n)$
4) Quadratic time - $O(n^2)$ — No of for loops. 2 nested for loops
5) Cubic time - $O(n^3)$
6) Exponential time - $O(2^n)$
7) Factorial time - $O(n!)$

Big O Notation

- It gives relation between the input data size (n) and the No of operations (N) with respect to time.
- Relation is denoted as Order of Growth & Time complexity is denoted O[n]

1) Constant time - O(1)
- When not dependent on the input size 'n'., the runtime will always be same.

e.g- a) A[1000] contains 1000 elements, it takes same time to access 10th element & 999th element
b) I++, this statement takes constant time
c) C= a+b
d) push/pop operⁿ in stack.

2) Linear time - O(n)
When the running time increase linearly with length of the input.

e.g- linear search
```
for (i=0; i<n; i++)
{ statement }
```

3) Logarithmic time - O(log n)
When it reduces the size of the input data in each step.
- No of operⁿ reduced & input size increase.
e.g - Binary tree, Binary search funⁿ

4) Quadratic Time - $O(n^2)$
- Has growth rate of n^2.
If input size is 2, it will perform 4 oper^n
—11— 8, —11— 64 oper^n

e.g - Bubble sort - $O(n^2)$

5) Cubic time - $O(n^3)$
processing time of an algo is proportional to the cube of the input element.

e.g - 3 Nested for loops.

```
for( ){
 for( ){
  for( )
  { statement }
}}
```

6) Exponential time - $O(2^n)$
Exponential (base 2) running time means that the calculations performed by an algo doubles every time as the input grows.

e.g -
a) Power set - finding all subsets on a set
Powerset ('ab') : ['', 'a', 'b', 'ab']
$f(n) = 4$
$2^2 = 4$

b) Travelling salesman problem using dynamic programming.

6) Polynomial time - $O(n^k)$
Number of steps require to solve an algo with input size 'n' is $O(n^k)$.
- No of oper^n ∝ power 'k' of the input size
- $O(n^2)$ & $O(n^3)$ are special type of polynomial time

7) Factorial time - $O(n!)$
Multiplication of all positive integers numbers less than itself.
e.g - $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
a) permutation of string
b) Solving travelling salesman problem using Brute Force Method.

$O(n)$, $O(c^n)$, $O(n^c)$ - Worst
$O(n \log n)$ - bad
$O(n)$ - Fair
$O(\log n)$ - Good
$O(1)$ - Best

Performance Analysis -
Efficiency of an algo is calculated on 2 parameter
1) Space Complexity
2) Time Complexity.

1) Space Complexity -
Amount of memory it needs to run to complete space needed by any algo
$S(P) = C$ (fixed part) + S_p (variable part)

1) Fixed part -
Independent of instance characteristics.
e.g - int, float

2) Variable part -
dependent on particular problem Instance
e.g - Array, class, structure

Time Complexity.

$$T(P) = \text{Compile time} + \text{execution time}$$
$$\boxed{T(P) = tp\,(\text{execution time})}$$

step count -
1) for algo heading = 0 time
2) for braces = 0 time
3) for expression = 1 unit time
4) for any looping statement = No of times the loop is repeating

e.g -
for i=1 to n → (n+1) true, for false.

Asymptotic Analysis -
The study of change in performance of an algo with the change in the order of input is defined as Asymptotic Analysis.

Best case - when the input size is very less
Average Case - —"— medium
Worst case - —"— big.

- The fun $f(n)$ is said to be in $O(g(n))$, iff $f(n)$ is bounded above by some const. multiple c of $g(n)$ for all large n.

Page No.
Date

- Represents the upper bound of running time of an algo.

① Big O ⇒ Worst Case. Complexity. Time

$$\boxed{f(n) \leq c \cdot g(n)}$$

$f(n) = 2n+2$; $g(n) = n^2$

$n=1$

$f(n) = 2+2 = 4$; $g(n) = 1^2 = 1$

$\therefore f(1) > g(1)$

$n=2$

$f(n) = 4+2 = 6$; $g(n) = 2^2 = 4$

$\therefore f(2) > g(2)$

$n=3$

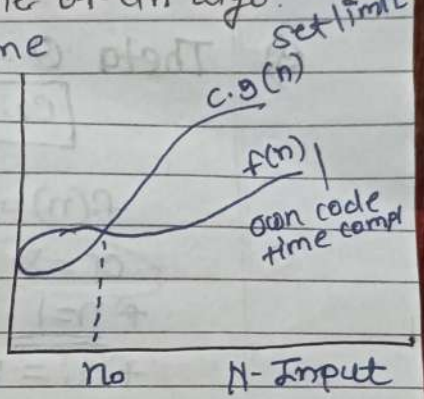
$f(3) = 6+2 = 8$; $g(3) = 3^2 = 9$

$\therefore f(3) < g(3)$

$n=4$

$f(4) = 8+2 = 10$; $g(4) = 4^2 = 16$

$\therefore f(4) < g(4)$



$\boxed{n_0 = 3}$ $f(n) \leq c \cdot g(n)$ for every $n \geq n_0$.

~~Represents lower bound~~

② Omega Ω ⇒ Best case.

A fun $f(n)$ is said to be in $\Omega(g(n))$, if $f(n)$ is bounded below by some const multiple of c of $g(n)$ for all large n

$$\boxed{f(n) \geq c \cdot g(n)} \text{ for all } n \geq n_0.$$

$f(n) = n^2$; $g(n) = 2n+2$

$n=1$

$f(1) = 1$; $g(1) = 2+2 = 4$

$f(1) < g(1)$

$n=2$

$f(2) = 4$; $g(2) = 4+2 = 6$.

$n=3$

$f(3) = 3^2 = 9$; $g(3) = 6+2 = 8$.

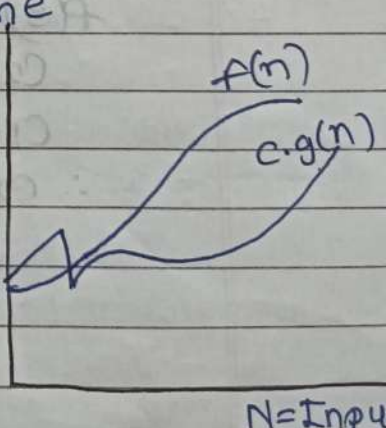
$f(3) > g(1)$

$\boxed{n_0 = 3}$

$n=4$

$f(4) = 4^2 = 16$; $g(4) = 8+2 = 10$.

$f(4) > g(4)$



Page No.
Date

3) Theta $\Theta \Rightarrow$ Average Case

$$\boxed{c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)}$$

$f(n) = 2n+8$; $g(n) = n$

$c_1 = 8$; $c_2 = 2$

$n=1$

$f(1) = 2+8 = 10$; $g(1) = 1$ $\quad \boxed{n_0 = 2}$

$c_2 \cdot g(n) = 2 \times 1 = 2$

$c_1 \cdot g(n) = 8 \times 1 = 8$

$2 \leq 10 \leq 8$ // Not satisfied

$\therefore c_2 \cdot g(1) \leq f(n) \leq c_1 \cdot g(n)$ ×

$n=2$

$f(2) = 4+8 = 12$; $g(n) = 2$

$c_2 \cdot g(n) = 2 \times 2 = 4$

$c_1 \cdot g(n) = 8 \times 2 = 16$

$\therefore c_2 \cdot g(2) \leq f(n) \leq c_1 \cdot g(2)$

$4 \leq 12 \leq 16$ // satisfied

$n=3$

$f(3) = 6+8 = 14$; $g(n) = 3$

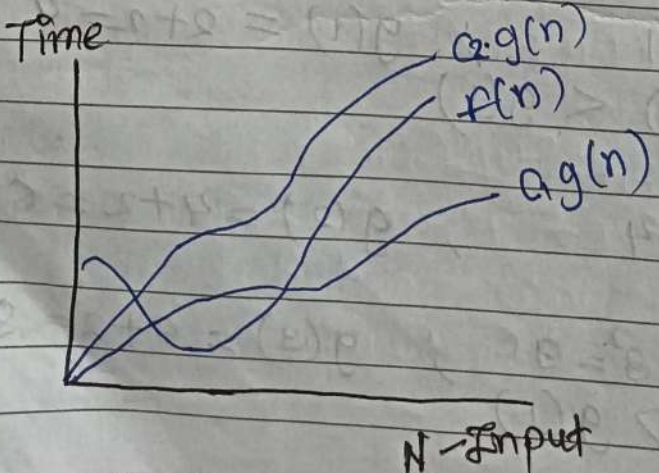
$c_2 \cdot g(n) = 2 \times 3 = 6$

$c_1 \cdot g(n) = 8 \times 3 = 24$

$\therefore c_2 \cdot g(3) \leq f(3) \leq c_1 \cdot g(3)$

$6 \leq 14 \leq 24$ // satisfied

A fun^n $f(n)$ is said to be $\Theta(g(n))$ if $f(n) = \Theta(g(n))$, if $f(n)$ is bounded both above & below by some constant multiple of $g(n)$ for all large n.



4) little oh o
5) little omega ω

4) Little oh o ⇒

$$\boxed{f(n) < c \cdot g(n)}$$

$f(n) = 2n+2$; $g(n) = n^2$

$n=1$

$f(1) = 2+2 = 4$; $g(1) = 1$

$n=2$

$f(2) = 4+2 = 6$; $g(2) = 4$

$n=3$

$f(3) = 6+2 = 8$; $g(3) = 9$

$f(3) < c \cdot g(3)$ // satisfied.

$n=4$

$f(4) = 8+2 = 10$; $g(4) = 16$

$f(4) < c \cdot g(4)$ // satisfied

5) Little omega ω

$$\boxed{f(n) > c \cdot g(n)}$$

$f(n) = n^2$; $g(n) = 2n+2$.

$n=1$

$f(1) = 1$; $g(1) = 2+2 = 4$

$n=2$

$f(2) = 2^2 = 4$; $g(2) = 4+2 = 6$

$n=3$

$f(3) = 3^2 = 9$; $g(3) = 6+2 = 8$

$f(3) > c \cdot g(n)$

Types of Algorithm and Applications -

1) Randomized Algorithm
2) Approximate Algorithm
3) Exact Algorithm

1) Randomized Algorithms
- An algo that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.
- Used to reduce time complexity or space complexity in algo & most times helpful to avoid infinite looping.

- types-
1) Las Vegas Algorithms
2) monte - Carlo Algorithms

1) Las vegas Algorithms
- Use the randomness to speed up the computation
- but algo must always return the correct answer to the input.
- let input be random
- The restriction that the expected runtime be finite.
- Arise frequently in search problems.
- Time & Space complexity must be less then we can use this algo.

2) Monte-Carlo Algorithm
- Do not have the former restrictions.
- they are allowed to give wrong return values.
- Answer may be incorrect with a certain (typically small) probability.
e.g- Karger-stein algo & minimum feedback arc set

2) Exact Algorithms-
- finds the solution to the problem asked.
- solve an optimization problem to optimality

3) Approximate Algorithm -
- one must content oneself with approx
- when finding optimal solution is intractable, but can also be used in situation where near-optimal sol^n can be found quickly & exact solution is not needed.
- returned sol^n to the optimal one.
- come as close as possible to optimal sol^n in polynomial time (P)

function - 0
declare variable - 0
value assign variable - 1
for 1 to n → n+1
calculate → till n → n
return → 1
for algo heading → 0
braces → 0
expres^n → 1
loopin → no of times loop is repeating.

Unit 2 - Divide & Conquer and Greedy strategy.

Page No.
Date

Divide & Conquer

- divide that problem into subproblems
- solve this subproblems
- Combine the solution of all the subproblems together to get the solution of the original problem.

Greedy -

It has 3 parts

1) Divide - Divide the problem into no. of sub problem that are similar instances of the same problem.

2) Conquer - Conquer the sub-problem by solving them recursively, if they are small enough, solve the sub-problem as base cases.

3) Combine - Combine the solution to the sub-problem into the sol^n for the original problem.

Pros & Cons.

1) sub-problem can be solved parallely.
2) No dependency of an sub-problem.

Cons

In this approach, most of the algo are design using recursion, hence
Memory management is very high.
For this recursive fun^n, stack is used, where the recursive fun^n stay needs to be stored.

Technical, given a fun^n to compute on n inputs, divide & conquer strategy suggests splitting the inputs into k distinct subset

$1 < K < n$

giving sub-problem

Application

1. Binary search
2. Merge sort
3. Strassens Matrix Multiplication
4. Finding min & max of a sequence of numbers.

Control Abstraction of D & C -

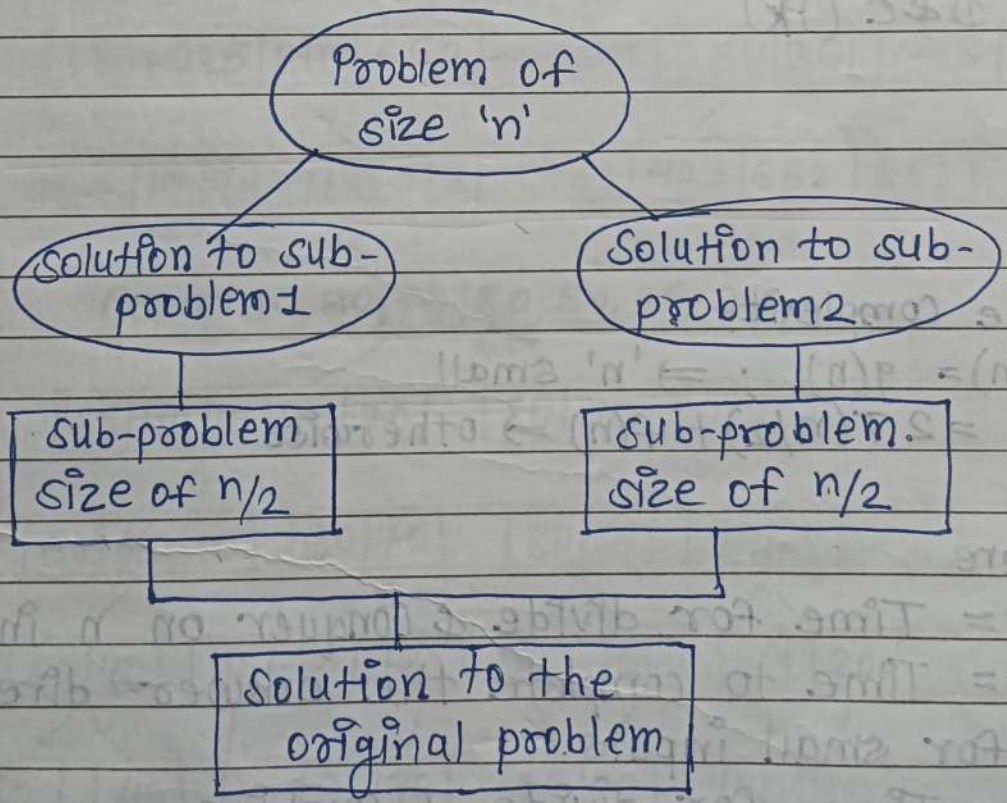


fig. Control Abstraction of D & C technique

Control Abstraction of D & C

```
Algorithm D&C (P) // Problem to be solved
{
 if small (P) then
   return S(P)
 else
 {
 Divide P into smaller instances
  P1, P2, P3, ---, Pk;     K >= 1

//Apply D&C to all these sub-problems

  return combine (D&C (P1), D&C (P2), D&C (P3), ---
    D&C (Pk).
 }
}
```

- Time complexity,

$T(n) = g(n)$; $\Rightarrow$ 'n' small

$= 2T(n/2) + f(n) \Rightarrow$ otherwise

where

$T(n)$ = Time for divide & conquer on n inputs

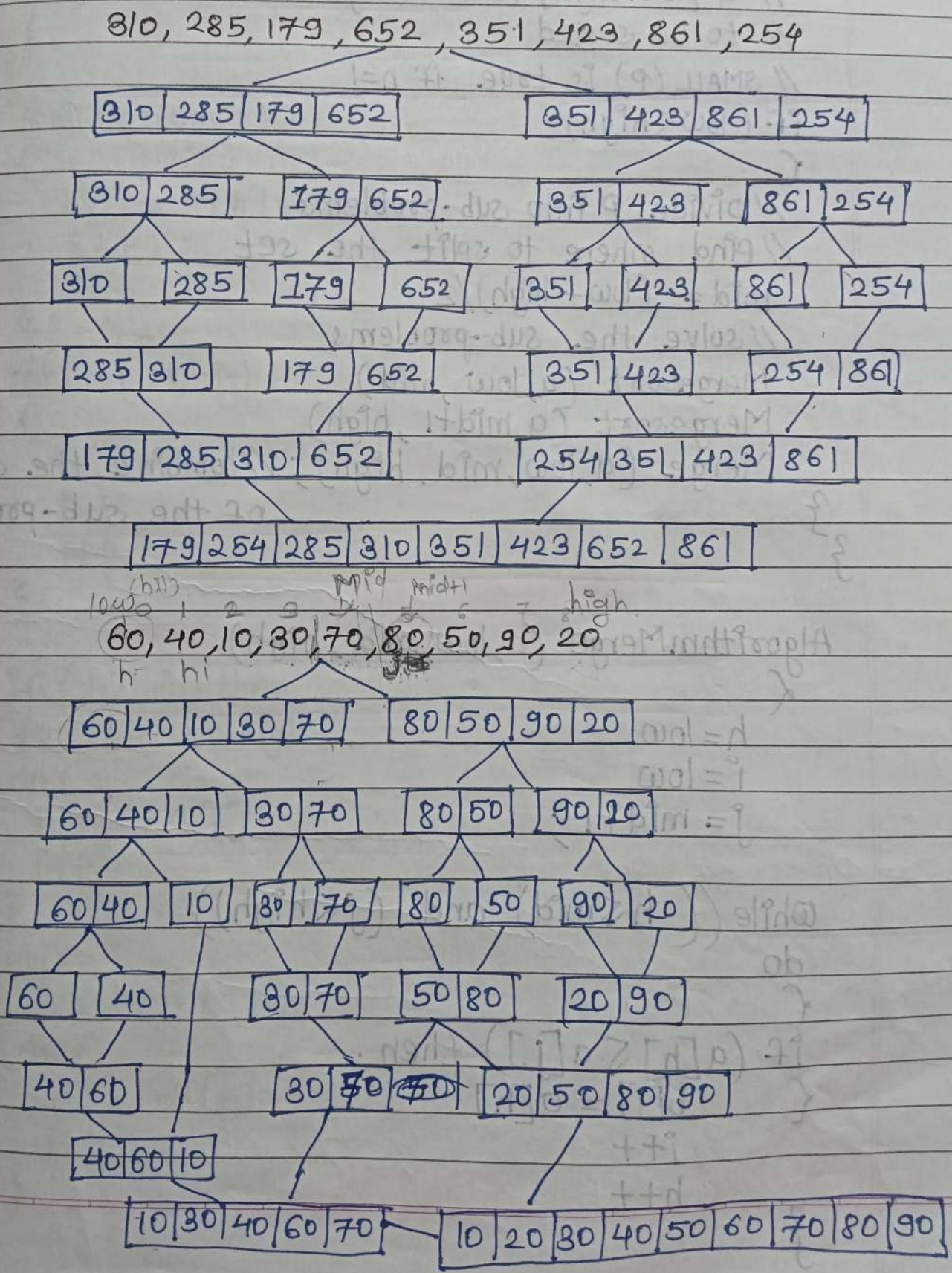
$g(n)$ = Time to complete the answer directly for small inputs.

$f(n)$ = Time for divide & combine

Merge sort -

① Divide

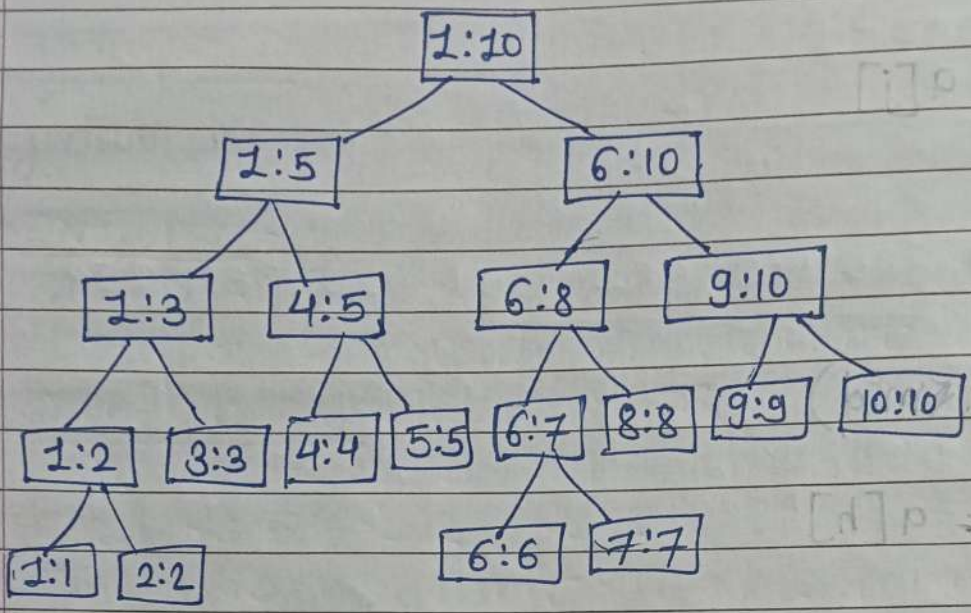
② Merge



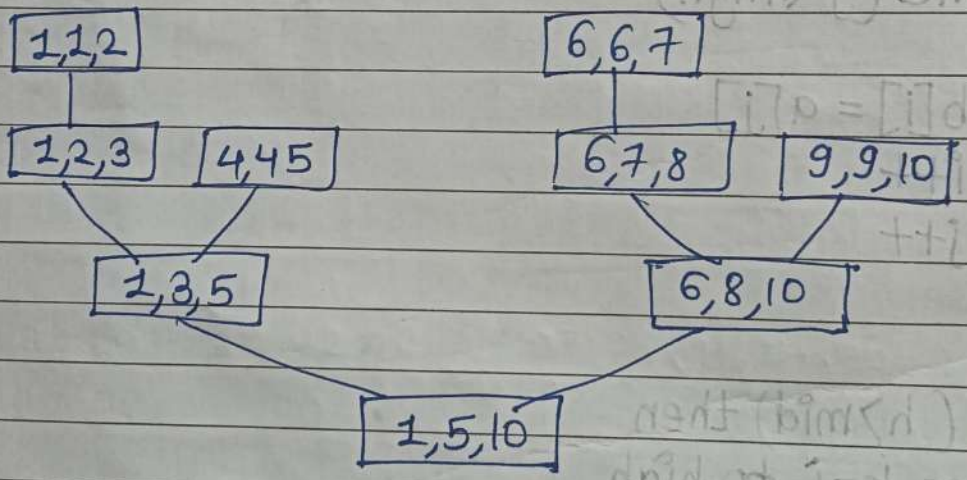
Merge sort

```
Algorithm Mergesort (a, low, high)
{
   // a [low: high] is an array
   // to be sorted
   // SMALL (P) is true if n=1
   if (low < high)
   {
   // Divide P into sub-problems
   // Find where to split the set
    mid = (low + high)/2
    // solve the sub-problems
    Mergesort (a, low, mid)
    Mergesort (a, mid+1, high)
    Merge (a, low, mid, high); // combine the sol^n
                                  of the sub-problem
   }
}

Algorithm Merge (a, low, mid, high)
  {
     h = low
     i = low
     j = mid+1;

  while ((h ≤ mid) and (j ≤ high))
  do
  {
  if (a[h] ≤ a[j]) then
  {     b[i] = a[h]
        i++
        h++
  }
```

```
else
{ b[i] = a[j]
   i++
   j++
}
}
while (h<=mid)
{
   b[i] = a[h]
   h++
   i++
}
while (j<=high)
{
   b[i] = a[j]
   i++
   j++
}

if (h>mid) then
for k=j to high
do{
 b[i] = a[k]
 i++
}
else
 for k=h to mid
 do{
  b[i] = a[k]
  i++
}
```



Fig① Tree of Calls of Merge Sort (1:10)



Fig② Tree of Call of Merge

Recurrence Relⁿ

$T(n) = a \quad ; \quad n=1$

$= 2 \cdot T(n/2) + cn \quad ; \quad n>1$

c is constant

$T(n) = 2 \cdot T(n/2) + cn$ —①

But what is $T(n/2)$?

Page No.
Date

Substitute $(n/2)$ at 'n' in (1)

$$T(n/2) = 2.T(n/2/2) + cn/2$$
$$= 2.T(n/4) + c.n/2$$

$$T(n/2) = 2.T(n/2^2) + c.n/2 \quad -(2)$$

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

$$= 4.T\frac{n}{2^2} + \cancel{2}c \times \frac{n}{\cancel{2}} + cn$$

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

But $T(n/4)$

Substitute $(n/4)$ at n in eqn (1)

$$T(n/4) = 2.T(n/4/2) + c.n/4$$
$$T(n/4) = 2.T(n/8) + c.n/4 \quad -(4)$$

$$= 2^2.[2.T(n/8) + c.n/4] + 2cn$$

$$= 2^3.T(n/8) + 2^2.cn/4 + 2cn$$

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

After k iterations

$$2^k.T(n/2^k) + kcn$$

But $n = 2^k \Rightarrow k = \log_2 n$

$= 2^{\log_2 n} \cdot T(n/2 \log_2 n) + \log_2 n \cdot cn$

$= n \cdot T \cdot (n/n) + c \cdot n \log_2 n$

$= n \cdot T(1) + c \cdot n \log_2 n$

$= n \cdot a + c \cdot n \log_2 n$

$T(n) = O(n \cdot \log_2 n)$

Greedy Strategy

1. Greedy
2. Dynamic Programming
3. Branch & bound.

- Feasible solution -
  Solutions which satisfy the given conditions.
- Optimal solution -
  solution which achieves the objectives of a given problem
- Optimization problem -
  A problem that requires a minimum or maximum value or result is called as optimization problem.

Technically

At current, whatever we feel is best soln we select it, without worrying for the final o/p.

Page No.
Date

A greedy strategy works if a problem has foll^{n} 2 properties.

1) Greedy choice property
2) Optimal substructure

1) Greedy choice property -
If an optimal solution to a problem can be found by choosing the best choice at each step without reconsidering the previous step once chosen, the problem can be solved using a greedy approach.
This property is called greedy choice property.

2) Optimal substructure -
Optimal solutions will always contains optimal sub solutions, then the problem can be solved using greedy approach
This property is called optimal substructure.

- Algorithm

① Initially, the solution set (set containing result) is empty.
{ } = 0
② At each step, we select an item & add it to the solution set.
③ If the solution set is feasible, the current selected item is kept in the sol^{n} set.

④ Else, it is removed from the solution set and never considered again.

- **example -**

problem -

Make a change of an amount using the smallest possible number of coins

Amount = Rs 18, and the available coins are Rs 5, Rs 2, Rs 1.

There is No limit to the number of each coin you can use.

⇒ **Solution -**

1) Soln set : { }
2) Select Rs 5
   {5} ; 5<18
3) Select Rs 5
   {5, 5} ; 10<18
4) Select Rs 5
   {5, 5, 5} ; 15<18
5) Select Rs 5
   {5, 5, 5, 5} ; 20>18 ×
   select Rs 2
   {5, 5, 5, 2} ; 17<18
6) Select Rs 1
   {5, 5, 5, 2, 1} ; 18=18

① Iteration 1:
solution set = {5} and
sum = 5

② Iteration 2:
solution set = {5,5},
sum = 10

③ Iteration 3:
solution set = {5,5,5},
sum = 15

④ Iteration 4:
solution set = {5,5,5,2}
sum = 17

⑤ Iteration 5:
solution set = {5,5,5,2,1}
sum = 18

Note:
Always select the coin with largest value until the sum is greater than 18, sum > 18

Control Abstraction of Greedy strategy.

```
Algorithm Greedy (a,n)
{
 // a[1:n] contains 'n' i/ps (subsolutions)
 solution = 0; // initialize the solution
 for i to n do
 {
   x = select(a);
   if (feasible(soln, x)) then
   solution = union(soln, x);
 }
 return soln;
}
```

Advantages

3) easy to implement 4) efficient for certain problems.
1) Simple to describe
2) It performs better than others (but not in all cases.

Disadvantages

2) Can be slow 3) No backtracking
1) It does not guarantee for optimal solution always. 4) Dependency on problem structure

Knapsack Problem / Fractional Knapsack problem

1. We are given a bag (knapsack) with capacity M.
   ex- M=15, 18, 20
2. We are given 'n' no of objects.
   ex - n=3, 7, 10....
3. An object 'i' is associated with some weight value 'wi' and profit value 'pi'
4. If a fraction 'xei', $0 \leq xi \leq 1$ of object 'i' is placed into knapsack then a profit of 'pixei' is earned
5. The objective is to obtained a filling of the knapsack, that maximizes the total profit earned.
6. Since the knapsack capacity is 'M' we require the total weight of all choosen object to be at most capacity of knapsack i.e M.

⑦ Formally the problem can be stated as-

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

subject to $\sum_{i=1}^{n} w_i x_i \leq M$ ——— ② and

$0 \leq x_i \leq 1, \quad 1 \leq i \leq n$ ——— ③

⑧ The profits & weights are positive numbers

⑨ A feasible sol^n is any set $(x_1, x_2, \ldots, x_n)$ satisfying eq^n ② & ③

⑩ An optimal sol^n is a feasible sol^n for which eq^n ① is maximized.

ex-

① Let $n=3$, $M=20$, $(P_1, P_2, P_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$ find feasible solutions and optimal solution using knapsack Problem.

⇒ Given-

$n=3$, $M=20$

object	1	2	3
Profit	25	24	15
Weight	18	15	10

Max $\sum_{i=1}^{n} p_i x_i$, $0 \leq x_i \leq 1$

$\sum_{i=1}^{n} w_i x_i \leq M$

Sol^n - Select 1st object completely

$W_1 = 18 \times 1 = 18$

$P_1 = 25 \times 1 = 25$

Remaining knapsack capacity.

$= 20 - 18 = \boxed{2}$

Select 2nd object ~~completely~~ 2/15

$W_2 = 15 \times 2/15 = 2$

$P_2 = 24 \times 2/15 = 3.2$

Remaining knapsack capacity

$20 - 18 - 2 = 0$

Add weights together & profit togethers

$\sum w_i x_i = 18 + 2 = 20$

$\sum p_i x_i = 25 + 3.2 = \underline{28.2}$

Sol^n 2 - select 1st object 1/2

$W_1 = 18 \times 1/2 = 9$

$P_1 = 25 \times 1/2 = 12.5$

Remaining knapsack capacity

$= 20 - 9 = \boxed{11}$

select 2nd object 10/15

$W_2 = 15 \times 10/15 = 10$

$P_2 = 24 \times 10/15 = 16$

Remaining knapsack capacity.

$= 20 - 9 - 10 = \boxed{1}$

select 3rd object = 1/10

$W_3 = 10 \times 1/10 = 1$

$P_3 = 15 \times 1/10 = 1.5$

Remaining knapsack capacity

$20 - 9 - 10 - 1 = 0$

Add weights together & profit togethers

$\Sigma w_i x_i = 9 + 10 + 1 = 20$

$\Sigma p_i x_i = 12.5 + 16 + 1.5$

$= \underline{\underline{30}}$

solution 3

select 2nd object 10/15

$W_2 = 15 \times 10/15 = 10$

$P_2 = 24 \times 10/15 = 16$

Remaining knapsack capacity

$= 20 - 10 = 10$

select 3rd object completely

$W_3 = 10 \times 1 = 10$

$P_3 = 15 \times 1 = 15$

Remaining knapsack capacity

$20 - 10 - 10 = 0$

Add weights together & profit togethers

$\Sigma w_i x_i = 10 + 10 = 20$

$\Sigma p_i x_i = 16 + 15 = \underline{31}$

Page No.
Date

solution 4

Select 3rd object completely

$W_3 = 10 \times 1 = 10$

$P_3 = 15 \times 1 = 15$

Remaining knapsack capacity

$20 - 10 = 10$

Select 1st object 10/18

$W_1 = 18 \times 10/18 = 10$

$P_1 = 25 \times 10/18 = 13.8$

Remaining knapsack capacity

$20 - 10 - 10 = 0$

Add weights together & profit togethers

$\Sigma w_i x_i = 10 + 10 = 20$

$\Sigma p_i x_i = 15 + 13.8 = \underline{\underline{28.8}}$

object	1	2	3
profit	25	24	15
weight	18	15	10
P/W	1.4	1.6	1.5

Soln	x_1	x_2	x_3	$\Sigma w_i x_i$	$\Sigma p_i x_i$
1	1	2/15	0	20	28.2
2	1/2	10/15	1/10	20	30
3	0	1	1	20	31
4	10/18	0	1	20	28.8

- To find optimal solution -
  ① Take (P/W) Ratio
  ② choose the objects with the decreasing value of (P/W) Ratio.

select object 2 completely.
$W_2 = 15 \times 1 = 15$
$P_2 = 24 \times 1 = 24$
Rem ~~w~~ knapsack capacity =
$20 - 15 = \underline{5}$

Select object 3 $5/10$
$W_3 = 10 \times 5/10 = 5$
$P_3 = 15 \times 5/10 = 7.5$
Remaining knapsack capacity =
$20 - 15 - 5 = \underline{0}$

$\sum w_i x_i = w_2 + w_3$
$= 15 + 5 = 20$
$\sum p_i x_i = P_2 + P_3$
$= 24 + 7.5 = \underline{31.5}$

Page No.
Date

without assumption

Greedy knapsack()
{for i=1 to n
compute (Pi/wi)
sort objects on decreasing value
of (Pi/wi)
for (i=1 to n (from sorted list)
if (M>0 && wi≤M)
M = M - wi
P = P + Pi
else
break;

If (M>0)
P = P + Pi (M/wi)

Time Complexity -

O(n.logn)

Applications
Job scheduling
Activity selection

Assumption :-

- P[1:n] & w[1:n] contain the p & w respectively of n objects ordered such that

$$\frac{P[i]}{w[i]} \geq \frac{P[i+1]}{w[i+1]}$$

M is the knapsack capacity
x[1:n] is the solution vector

```
Algo Gknapsack (M,n)
{ for i=1 to n do
    x[i]= 0.0
    u = M
  for (i=1 to n) do
  {
   if (w[i]>u) then
     break;
   x[i]=1.0;
    u=u-w[i];
  }
if (i≤n) then
   x[i]= u/w[i];
```

3. Dynamic Programming.

Page No.
Date

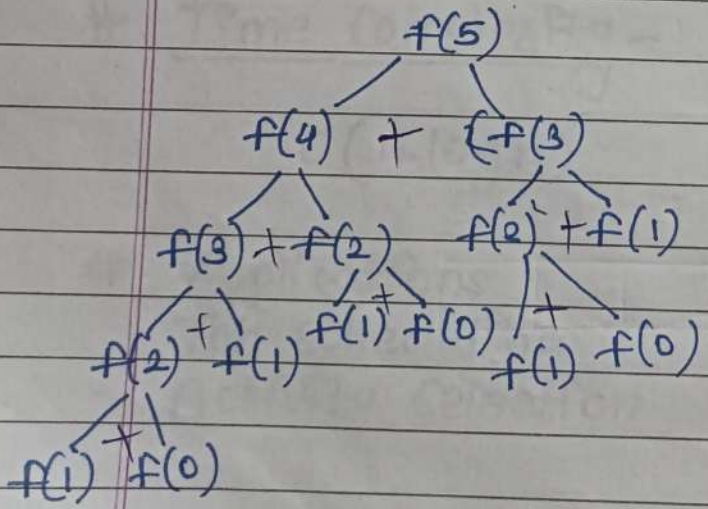
Working Principle -

1) Divide the complex problems into number of sub-problems
2) Solve these sub-problems by finding optimal solution.
3) Save those optimal sol^n of the sub-problems (memoization)
4) Use these stored result if same sub-problem is encountered again, instead of re-computing.
5) Combine the sol^n of these sub-problems together to get final result.

Properties of (DP)

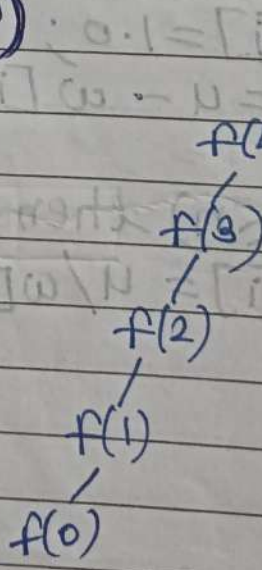
1) Optimal Substructure
2) Overlapping subproblems.

ex- n=5



$O(2^n)$

Memoization
Dynamic programming



O(n)

Page No.
Date

Abonacci series using Recursion

```
static int cnt=0;
int f(int n)
{
 cnt++;
  if(n<0)
  return error;
  if (n==0)
   return 0;
  if (n==1)
   return 1;
sum = f(n-1) + f(n-2)
    return sum;
 }

int main() {
 int n;
 cout<<"Enter number";
 cin>>n;
 cout<<f(int n);
```

n=0,1 → cnt=1
n=2 → cnt=3
n=3 → cnt=5
n=4 → cnt=9
n=5 → cnt=15
n=6 → cnt=25
n=10 → cnt=177
n=20 → cnt=21891 ≈ 22,000

exponential time ⇒ $O(2^n)$

Memorization feature of DP ⇓ $O(n)$

- **DP**

```
# static int cnt=0
int fib(int n)
{ if (memo[n] != NULL)
   return memo[n]
 cnt++;
 if (n<0)
   error
if (n==0)
   return 0
```

```
if (n==1)
  return 1
sum= fib(n-1) + fib(n-2);
memo[n] = sum;
return sum;
```

n=5
memo[5]

0	1	2	3	4	5
-1	-1	-1	-1	-1	-1

NULL=(-1)

f(0)=0 } base conditions
f(1)=1 } already given in the program

0	1	-1	-1	-1	-1
n= 0	1	2	3	4	5

fib(2) = fib(1) + fib(0)
= 1 + 0
fib(2) = 1

0	1	1	-1	-1	-1
n= 0	1	2	3	4	5

fib(3) = fib(2) + fib(1)
= 1 + 1
= 2

0	1	1	2	-1	-1
n= 0	1	2	3	4	5

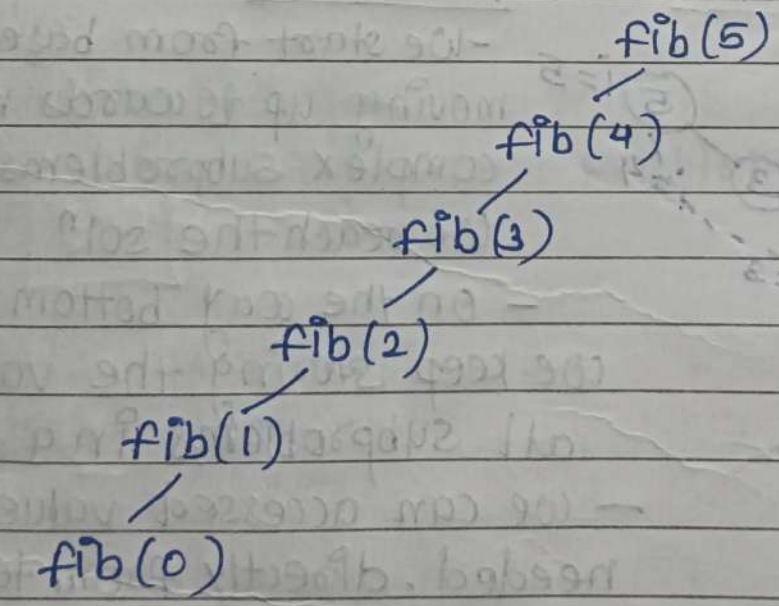
Date

fib(4) = fib(3) + fib(2)
= 2 + 1
= 3

0	1	1	2	3	-1
n = 0	1	2	3	4	5

fib(5) = fib(4) + fib(3)
= 3 + 2
= 5

0	1	1	2	3	5
n = 0	1	2	3	4	5



To reduce the exponential time complexity to lineae time complexity.

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

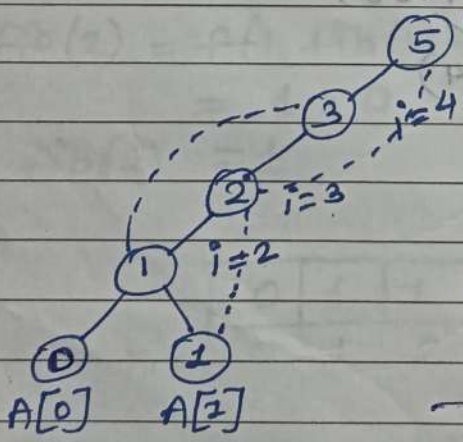
2 Methods.

2 Approaches of DP

① Top-to-down approach → Memoization
② Bottom-to-up approach → tabulation

```
int a[n];
int f(int n);
a[0]=0;
a[1]=1;
for(i=2; i<=n; i++){
  a[i]=a[i-1]+a[i-2]
}
return a[n];
}
```

Memoization - to find the soln to prob we start from top, moving down to smaller sub-problems.
- from top to down, we keep the storing the values of all the subproblems which are computed so they can be reused, This is memoization.
e.g - Fibonacci series



Tabulation-
- We start from base cases moving up towards more complex subproblems until we reach the soln.
- on the way bottom to top, we keep storing the values of all subproblems in a table.
- we can accessed values if needed. directly from table.

Principle of optimality

A problem is set to satisfy the principle of optimality if the sub-solutions of an optimal solution of the problem are themselves optimal solutions for their sub-problems.

problem can be solved by taking sequence of decisions.

Page No.
Date

- In an optimal sequence of decisions or choices each subsequence must be optimal

Control Abstraction of DP-

```
Algo DP(P)
if solved(P) then
  return (lookup(P))
else
  Ans:= SOLVE(P)
   store (P, Ans);
end;

algo SOLVE (P)
if SMALL(P) then
   solution (P)
else
Divide the problem P into
no. of smaller sub-problems
P1, P2, P3, ---, Pn

Ans1 : = DP(P1)
Ans2: = DP (P2)
Ans3: = DP(P3)
   :
Ans n: = DP(Pn)
return (Combine (Ans1, Ans2, Ans 3, ... Ansn));
end;
```

Time complexity - O(n)

- Dynamic programming problem can be solved using a top-down approach or bottom-up approach.

- both give the same solutions.
- In very large problems, bottom-up approach is beneficial as it does not lead to stack overflow.
- e.g - If the input is 10,000, top-down approach will give maximum calls to the stack which will exceed the size, but bottom up approach will give immediate solution

- We cannot eliminate recursive function completely from the program, we always have to define a recursive relation irrespective of the approach we used.

Binomial Coefficients -

Binomial coefficient of n and k is written as $c(n,k)$ or nc_k or $\binom{n}{k}$.

$\boxed{nc_k}$

Mathematically, we find binomial coefficient as

$$c(n,k) = \frac{n!}{k!(n-k)!} \quad \text{---} \quad (1)$$

$\frac{n!}{k!(n-k)!}$

Also binomial formula can be given as

$$(a+b)^n = nc_0 a^n b^0 + nc_1 a^{n-1} b^1 + nc_2 a^{n-2} b^2 + nc_3 a^{n-3} b^3 + \text{----} + nc_n a^0 b^n$$

where, $nc_0, nc_1, nc_2, \text{---} nc_n$ are called binomial coefficient.

• **Definition** -

A binomial coefficient $C(n,k)$ can be defined as the coefficient of x^k in the expansion of $(1+x)^n$

ex-

Expanding $(1+x)^2 \rightarrow 1x^0 + 2x^1 + 1x^2$

$k = 0, 1, 2$

$(1+x)^2 = 1 + 2x + x^2$

$\downarrow x^0 \quad \downarrow x^1 \quad \downarrow x^2$

Here x^0

① $C(2,0) = 1$

② $C(2,1) = 2$

③ $C(2,2) = 1$

coefficients of $x^0 = 1$

$x^1 = 2$

$x^2 = 1$

Another definition -

A binomial coefficient $C(n,k)$ also gives the number of ways in which k objects can be choosen from 'n' objects.

⇒ ~~the num~~

Using the mathematical formula, lets compute

$C(2,0)$

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

$$C(2,1) = \frac{2!}{1!(2-1)!} = 2$$

$$C(2,0) = \frac{2!}{0!(2-0)!}$$

$$C(2,2) = \frac{2!}{1!(2-2)!} = 1$$

$C(2,0) = 1$

- **Problem statement -**

Write a function that takes two parameters n & k and returns the value of binomial coefficient C(n,k)

→ we can compute C(n,k) for any n, n & k using the foll^{n} recurrence relation as

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

→ base condition

~~ex~~ - C(n,k)

C(n,k) = 1

k=0, n=k

C(n,0) = 1 } base cond^{n}

n & k value same C(n,n) = 1 }

for other values

$\boxed{C(n-1,k-1) + C(n-1,k)}$ → formula

example - C(4,2)

C(4,2)

n=4, k=2

C(4,2) = C(3,1) + C(3,2)

C(3,1) = C(2,0) + C(2,1)

= 1 + C(2,1)

C(2,1) = C(1,0) + C(1,1)

= 1 + 1 = 2

C(3,1) = 1 + 2 = 3

$c(3,2) = c(2,1) + c(2,2)$

$c(2,1) = c(1,0) + c(1,1)$

$= 1 + 1 = 2.$

$c(3,2) = c(2,1) + c(2,2)$

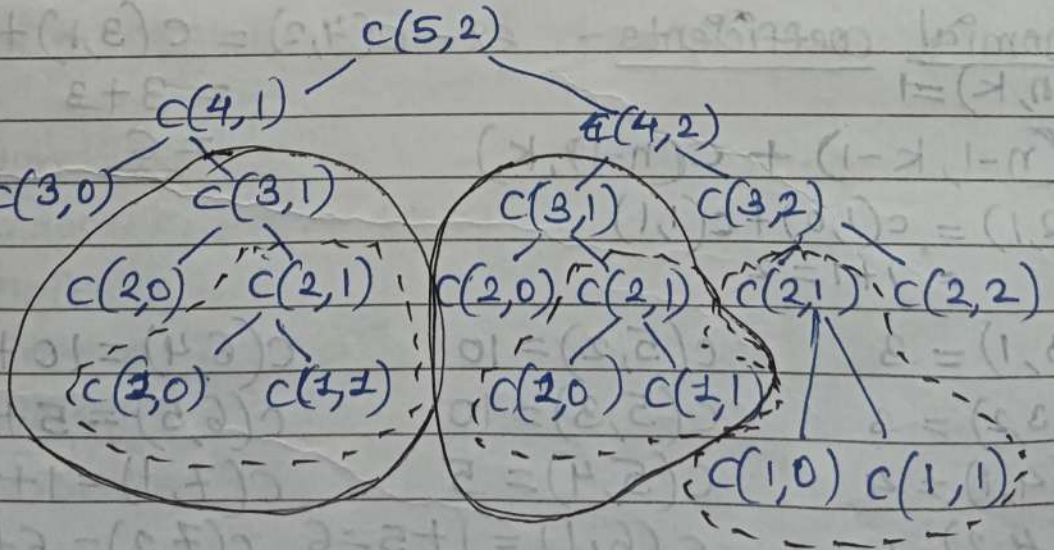
$= 2 + 1 = \underline{3}$

$c(4,2) = 3 + 3 = 6$

Recursive function to calculate binomial coefficient

```
int binomialCoefficient (int n, int k)
{
 if(k==0 || k==n) // Base case
  return 1;
 else
 return binomialCoefficient (n-1, k-1) +
   binomialCoefficient (n-1, k);
}
```

Recursive tree for binomial coefficient



- And following shows how this is done

```
int binomialcoefficient( int n, int k)
{
  int c[n+1][k+1];
  int i,j;
  for(i=0, i<=n ; i++)
  {
   for(j=0; j<=k ;j++)
   {
    if (j==0 || j==i) // base condition
      c[i][j]=1;
    else
    c[i][j]= c[i-1][j-1] + c[i-1] [j];
    }
  }
   return c[n][k];
}
```

- Binomial coefficients - ex- $C(4,2) = C(3,1) + C(3,2)$

$C(n,k) = 1$

$= 3+3$

$= 6$

$C(n-1, k-1) + C(n-1, k)$

$C(2,1) = c(1,0) + c(1,1)$

$= 1+1 = 2$

$C(3,1) = 3$ $C(5,2) = 10$ $C(6,4) = 10+5 = 15$

$C(3,2) = 3$ $C(5,3) = 10$ $C(6,5) = 5+1 = 6$

$C(4,1) = 4$ $C(5,4) = 5$ $C(7,1) = 1+6 = 7$

$C(4,2) = 6$ $C(6,1) = 1+5 = 6$ $C(7,2) = 6+15 = 21$

$C(4,3) = 4$ $C(6,2) = 5+10 = 15$ $C(7,3) = 15+20 = 35$

$C(5,1) = 5$ $C(6,3) = 10+10 = 20$ $C(7,4) = 20+15$

$= 35$

$C(7,5) = 15+6 = 21$ $C(7,6) = 6+1$

$= 7$

Page No.
Date

cond^{n} - $n>k>0$ $0<k<n$

n/K	0	1	2	3	4	5	6	7
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

$K=0 \rightarrow 1$
$n=k \rightarrow 1$

n+1 } Grid
K+1 }

Not compulsory to take 8 by 8.

Time complexity = O(n*k)
Space complexity = O(n*k)

0/1 Knapsack Problem -

0 = Object is not selected.
1 = Object selected completely.

Given :-
Knapsack/bag = M → capacity
Objects = n
Each objects = i
Weights = wi
Profit = Pi

Consider the problem having weights & profits as $W = \{3, 4, 6, 5\}$, $P = \{2, 3, 1, 4\}$

$M = 8$, $n = 4$

$\Rightarrow$

$xi = \{1,0,0,0\}$, $= \{0,1,0,0\}$, $\{0,0,1,1\}$, $\{0,1,0,$
$= \{1,1,0,0\}$, $= \{0,1,1,0\}$, $\{0,0,0,1\}$
$= \{1,1,1,0\}$, $= \{0,1,1,1\}$, $\{1,1,0,1\}$
$= \{1,1,1,1\}$, $= \{0,0,1,0\}$ $\{1,0,0,1\}$

As the number of objects are getting increasi it is not possible to check each and every possibility / condition so as to avoid that we use dynamic programming.

Pi	Wi	i ↓ W →	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0	0
2	3	1	0	0	0	2	2	2	2	2	2
3	4	2	0	0	0	2	3	3	3	5	5
4	5	3	0	0	0	2	3	4	5	5	6
1	6	4	0	0	0	2	3	4	4	5	(6)

as $M = 8 \Rightarrow$ Take columns 0 to 8
$n = 4 \Rightarrow$ Take rows 0 to 4
consider objects in their ascending order of their weight values

Here we have not taken the weight 8 directly, but instead it is divided into sub-problems 0, 1, 2, ---8 : the solutions of the subproblems will be same in the cells & the optimal solution will be stored in the last cell.

step 1 -

$i=0, w=0$

Initially no item is selected, so store '0' in 0^{th} column & 0^{th} row

M[4][8] = 6
M[3][8] = 6 } → selected.
M[2][8] = 5

xei = {1, 0, 0, 0}
xi xe2 xe3 xe4

6-4 = 2
M[2][3] = 2 → ~~selected~~
M[1][3] = 2 → selected

M = 3+5 = 8
P = 2+4 = 6

- **Formula -**

$M[i,w] = \max(m[i-1, w], m[i-1, w-w[i]] + P[i]$

If no of objects are greater then we cannot do it manually then this formula is used.

i = 2, w = 4

we have 2 objects ⇒ 4 and 3

$M[2,4] = \max(m[2-1, 4], m[2-1, 4-w[2]] + P[2]$

$= \max(m[1,4], m[1, 4-4] + 3)$
$= \max(2, m[1, 0] + 3)$
$= \max(2, 0+3)$
$= \max(2, 3)$

$M[2,5] = \max(m[2-1, 5], m[2-1, 5-w[2]] + P[$
$= \max(m[1,5], m[1, 5-4] + 3)$

$= \max(2, m[1,1] + 3)$

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

$= \max(2, 3)$ ← selected.

If we get -cell then we have to keep it zero.

$M[3,1] = \max(m[2,1], m[2, 1-4] + 3)$

$= \max(0, [2,-1] + 3)$ ← due to this (-ve) sign

$= 0$

2) $M = 4$

$N = 3$

$W = \{4, 5, 1\} = \{1, 4, 5\}$

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

$x_1\ x_2\ x_3$

⇒

Pi	Wi	i ↓ w →	0	1	2	3	4
	0	0	0	0	0	0	0
3	1	1	0	0			
1	4	2	0				
2	5	3	0				

$M[1,1] = \max(m[0,1], m[0, 1-1] + 3)$

$= \max(0, m[0,0] + 3)$

$= \max(0, 0+3)$ → (-ve)

$= \max(0, 3)$

$= 0$

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

Algorithm - 0/1 Knapsack

```
main()
{
 int p[5]={0,1,2,5,6};
 int wt[5]={0,2,3,4,5};
 int n=4, m=8;
int k[5][9];

 for (i=0; i<=n; i++)
 {
  for (w=0; w<=m; w++)
  {
   if (i==0 || w==0)
    k[i][w]=0;
  elseif (wt[i]<=w)
k[i][w] = max (p[i]+k[i-1][w-wt[i]], k[i-1][w]);
  else
   k[i][w] = k[i-1][w];
  }
}
 cout<<k[n][w];
  i=n, j=m;
 while (i>0 && j>0)
 {
 if (k[i][j]== k[i-1][j])
 {
  cout<<i<<"=0";
  i--;
 }
```

```
else {
cout << i << "=1";
j--;
j = j - wt[j];
}}}
```

Time complexity -

4. Backtracking and Branch-n-Bound

Page No.
Date

1) ~~Brack~~ Backtracking -

1) Backtracking represents one of the most general technique.
2) Many problems which deal with searching for a set of ~~p~~ solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation
3) In any many appl^n backtracking, the desired solution is expressed as an n-tuple $(x_1, x_2, \ldots, x_n)$, where x_i are choosen from some finite set S_i.
4) often the problem to be solved calls for finding one vector that maximizes or minimizes a criteria function $P(x_1, x_n)$

for example -

let the problem be of sorting the array of integers in a[1:n].

Its solution is express by an n-tuple where x_i is the index of i^{th} smallest element in a
5) The criteria function ~~f~~ be $a[x_i] < a[x_{i+1}]$ $1 \le i \le n$.
6) The set S_i is finite and includes the integers 1 to n.
7) Backtracking uses the concepts of brute-force technique to find out the solution.

~~Brute~~ force approach finds all the possible solutions where ~~the~~ backtracking finds the solutions which satisfies the given constraints. -
8) The basic idea of backtracking is to build up the solution vector one component at a time & to use modified criteria fun^n $P_i(x_1, \ldots x_i)$ sometime

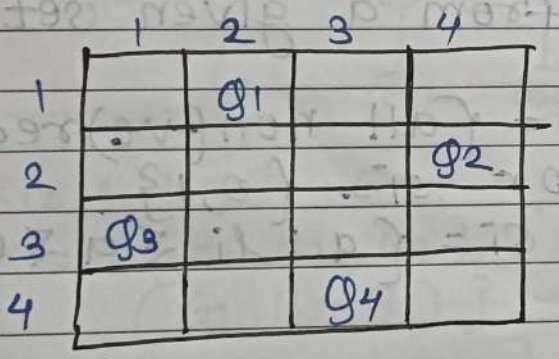
called bounding function)

9) To ~~test~~ whether the vector P from has any chance of ~~chan~~ success.

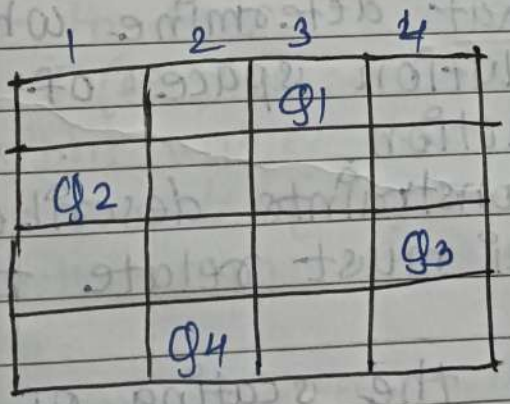
~~10) The measure~~

10) The measure advantage of backtracking:
If it is realize that partial vector $x_1, x_2, \ldots x_i$ can in no way lead to an optimal solution then ~~mi+o~~ m_{i+1} to m_n possible vectors can be ignored entirely.

4 queen problem (q_1, q_2, q_3, q_4)



$sol^n = \{2, 4, 1, 3\}$



$sol^n = \{3, 1, 4, 2\}$

Page No.
Date

Backtracking -

- finite set Si
- Constraints P
- solution vector $x = \{x_1, x_2, \ldots x_n\}$
- solution, If $x = \{x_1, x_2, \ldots, x_i\}$
- Not going for optimal soln we ignore next mi+1 to mn sub-soln sub ∉ backtrack.

$SP = \{2, 5, 1, 3, 4\}$
$x = \{6, 7, 8, 9\}$

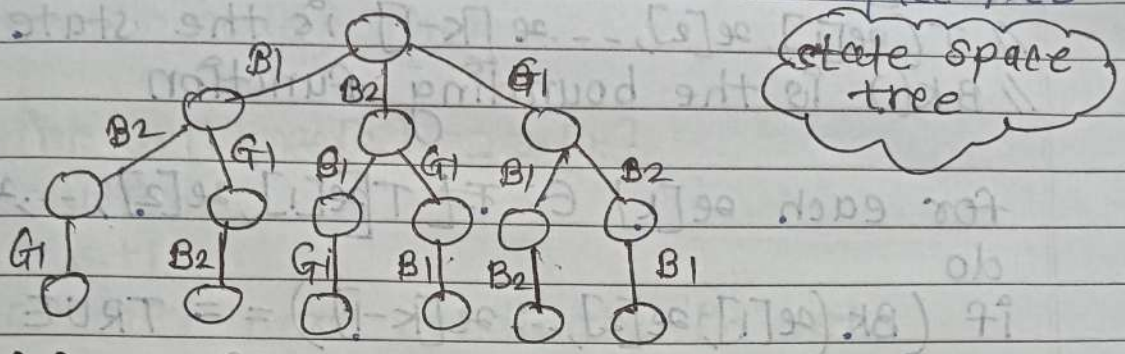
1) Explicit constraints -
Are rules that restrict each x_i to take an values only from a given set.
ex -
$x_i \geq 0$ or $S_i = \{$all non-ve) real numbers$\}$
$x_i = 0$ or 1 or $S_i = \{0, 1\}$
$l_i \leq x_i \leq u_i$ or $S_i = \{a : l_i \leq a \leq u_i\}$

2) Implicit constraints -
Are the rules that determine which of the tuples in the solution space of I satisfy the criteria function.
Thus, implicit constraints describe the way in which the x_i must relate to each other

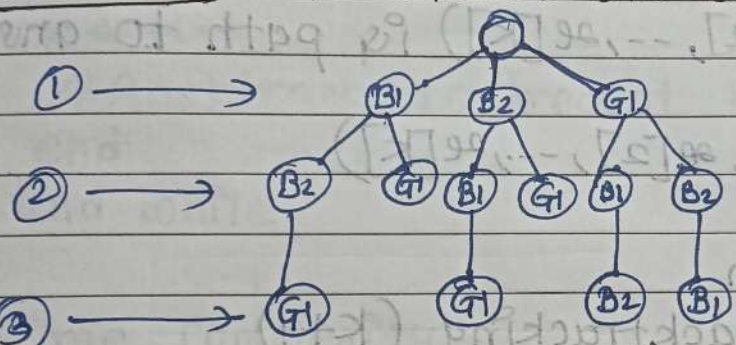
Problem - Arranging the seating of 3 students 2 boys - B1 and B2 and one girl - G1

⇒ we will find all possible soln using Brute force technique

All the solu^{n} can be represented in the form of a tree which is termed as "state space tree".



condition - G1 cannot sit bet^{n} B1 & B2



$S_i = \{B_1, B_2, G_1\}$

$X = \{(B_1, B_2, G_1), (B_2, B_1, G_1), (G_1, B_1, B_2), (G_1, B_2, B_1)\}$

- let $(x_1, x_2, \ldots x_k)$ be the path from the root to the i^{th} node
- let $T(x_1, x_2, \ldots x_k)$ be the set of all possible values for next node x_{k+1} such that $(x_1, x_2, \ldots, x_{k+1})$ is also a path from the root to a problem state.

General Algorithm discussed above finds all answer nodes not just one.

Page No.
Date

Algorithm Recursive Backtracking (k)
// x[1..k-1] is the solution vector
// T(x[1], x[2], ... x[k-1] is the state space tree
// BK() is the bounding function

for each x[k] ∈ T[T[x[1], x[2], ..., x[k-1]
do
if (BK(x[1], x[2], ... x[k-1]) == TRUE then
// check for feasible solution
if (x[1], x[2], ..., x[k]) is path to answer node then
print (x[1], x[2], ..., x[k]).
end
if k<n then
Recursive - backtracking (k+1)
end
end
end do

Iterative Backtracking

Algorithm Iterative - Backtrack (n)

// x[1...k-1] is the solution vector
// T(x[1], x[2], ..., x[k-1]) is the state space tree
// Bk() is the bounding function.

k ← 1
while k ≠ 0
do

if (untried $x[k] \in T[x[1], x[2], \ldots, x[k-1]]$)
AND ($B_k(x[1], x[2], \ldots, x[k])$ is path to answer node) then

print $x[1], x[2], \ldots x[k]$
//solution
$k \leftarrow k+1$
// consider next candidate in set
else
$k \leftarrow k-1$
// Backtrack to recent node
end
end while

- Time complexity -
The performance of backtracking depends on 4 factors.
1) Time to compute the tuple $x[k]$
2) No. of $x[k]$ which satisfy the explicit constraint.
3) Time taken by bounding function B_k to generate a feasible solution.
4) A number of $x[k]$ which satisfy the bounding function B_k for all k.

$O(p(n) \cdot n!)$

$p(n)$: Time to execute first 3 steps
$n!$: time to execute 4th step.

Unit-5

Complexity theory

Page No.
Date

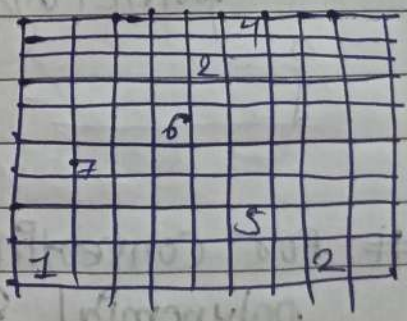
Algorithms

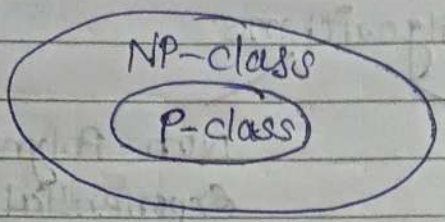
Polynomial Time	Non-Polynomial Time / Exponential time algo.
ex- Linear search - $O(n)$	ex- 0/1 knapsack - 2^n
Binary search - $O(\log n)$	sudoku - 2^n
Insertion sort - $O(n^2)$	TSP - 2^n
Merge sort - $O(n \cdot \log n)$	Graph coloring - 2^n
Matrix Multiplication - $O(n^3)$	Job scheduling - 2^n

Hard

1) **P-class** -
A problem which can be solved in polynomial time is called as P-class problem
e.g - Lineae search, Binary search

2) **NP class** - (Non-deterministic polynomial time.)
A problem that cannot be solved in polynomial time, but whose solution can be verified in polynomial time is called as np-class problem.
e.g - sudoku
for solving sudoku problem it takes hardly 20 min but to verify its solution it takes polynomial time.





$P \subseteq NP$

Deterministic Algorithm -
The problems for which we can write finite and accurate algorithmic steps, knowing working of each & every step are deterministic Algorithm.

Non-deterministic Algorithm -
The problems for which we cannot write finite & accurate algorithmic steps, we don't know, how the task will be implemented are non-deterministic Algorithm.

ex -

```
algo eleseaech (A, n, key)
{
  i = choice ();
  if (A[i] == key)
  print (i); // success
  print (0) // failure.
}
```

For converting exponential time Algo to into polynomial time, we need to focus on only 2 points

① Writing non-deterministic problem:
② And try to showing Relationship between the exponential time algorithms:

- Consideration of base problem.

Base Condition ⇒

Satisfiability problem → definition

Boolean formula

0/1 ↳ CNF - satisfiability formula

↓

Conjunctive Normal Form

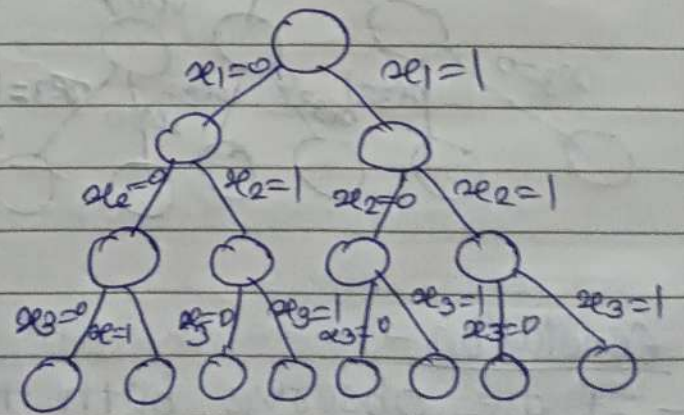
⇒ Variables : $X_i = \{x_1, x_2, x_3, \dots\}$

ex- $x_i = \{\overline{x_1} \vee x_2 \vee \overline{x_3}\} \wedge \{x_1 \vee \overline{x_2} \vee x_3\}$

clause C_1 clause C_2

① Insert the values of x_i and check for which values of x : this CNF formula is satisfied.

x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1



0/1 Knapsack Problem

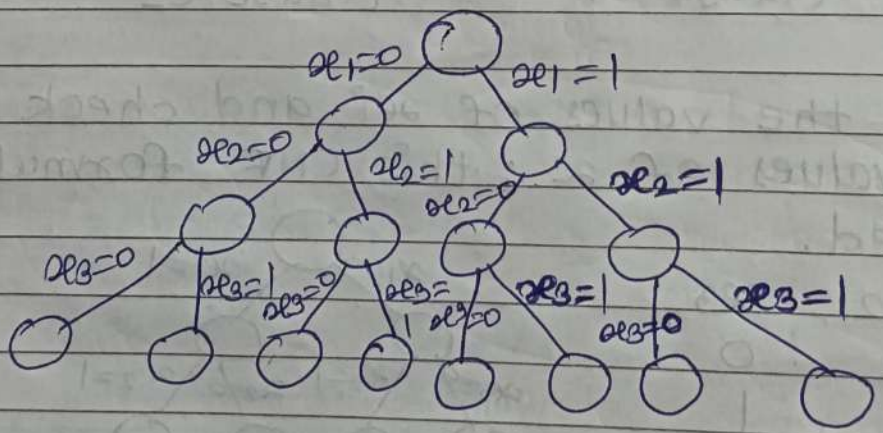
P	10	15	20
W	4	3	5

1 → selected
0 → Rejected.

$\Rightarrow Sol^n = \{obj1\ 0/1, obj2\ 0/1, obj3\ 0/1\}$

obj 1	obj 2	obj 3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Trying all possible values it will take,
$2^3 = 8$ units of time
$n \Rightarrow (2^n)$
time



Conclusion -

- If we observe both the state space tree we can say that it is same for CNF satisfiability problem which takes 2^n as it the exponential time and 0/1 knapsack problem also takes 2^n time complexity.

- Hence, above statespacetree can be solved in polynomial time then CNF satisfiability algo and 0/1 knapsack algo can also be solved in polynomial time.
- That's How we could show the relation using CNF satisfiability algo as the base problem.

Reduction -

- Satisfiability → NP Hard.
- 0/1 knapsack
- sudoku
- Scheduling
- TSP

} Hard

!

Reduction -

Satisfiability (CNF) I_1 $\propto$ 0/1 knapsack I_2

reduces to

Polynomial time

If $I_1 \propto I_2$ and
If I_2 can be solved in polynomial time using some algo, then I_1 can also be solved in polynomial time using the same algo.

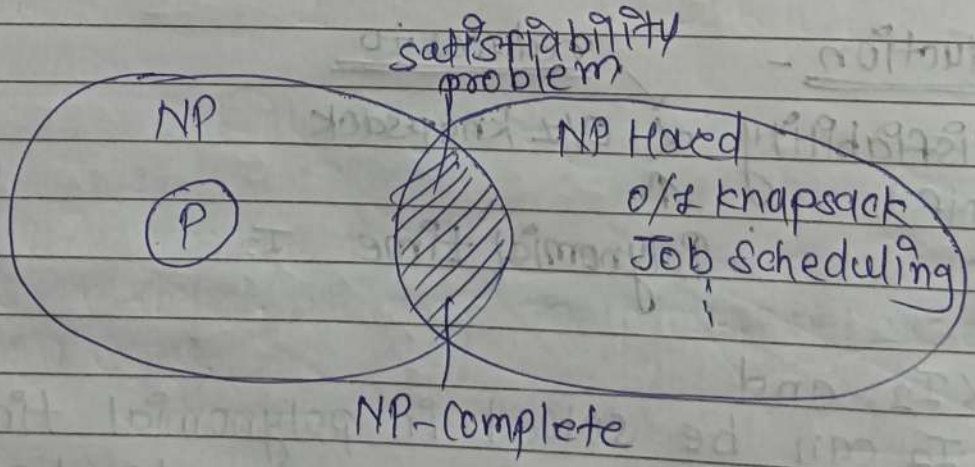
If this reduction ~~in~~ to is possible in polynomial time then we say that satisfiability problem and 0/1 knapsack problem are related to each other.

Now, we say that, satisfiability is problem is NP-hard (already proved).
- So if satisfiability problem can be reduced to 0/1 knapsack problem, then due to the relation, 0/1 knapsack problem also becomes NP-hard.
- Hence Relationship is proved between them.

NP-Hard and ~~NP~~ NP-Complete Problem

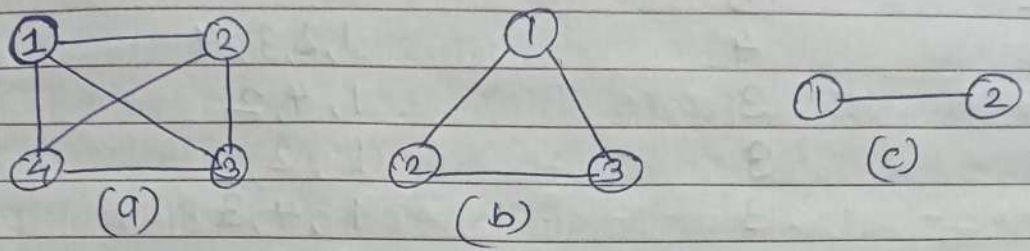
Satisfiability problem → NP-Hard
↑
Non-deterministic.
↑
already proved

0/1 knapsack
Job Scheduling
Graph coloring
} Hard.

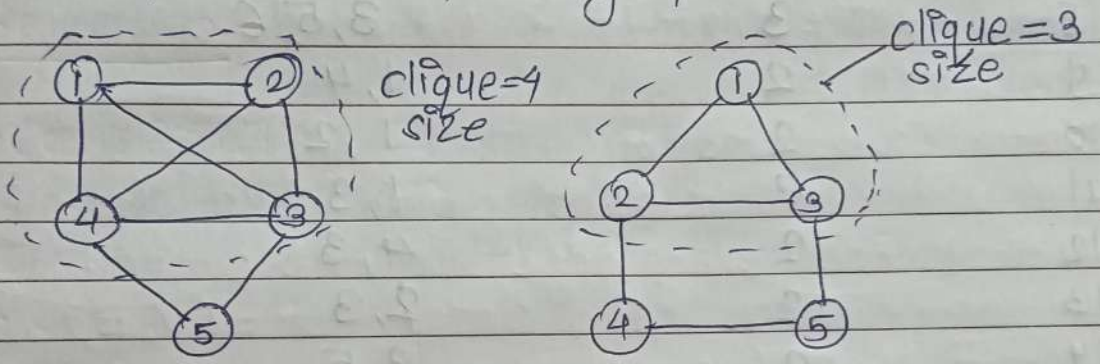


- We have non-deterministic algo for satisfiability problem (already proved).
- So if we can write non-deterministic algo for exponential time NP-hard problems, then this problems are called as NP-complete problems.

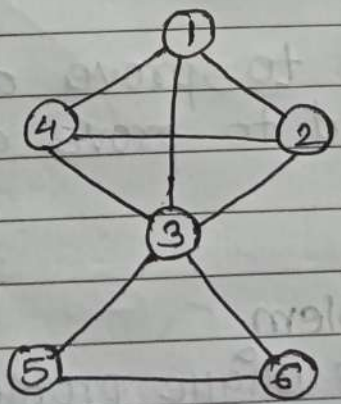
NP-Hard Problem :- Clique Decision Problem (CDP) problem - Prove that CDP is NP-Hard.



examples of complete graph.



This subgraph are complete graph, but graph is not complete graph, so that subgraph is called as clique



Sr.No.	Clique Size	Subgraph vertices:
1	6	—
2	5	—
3	4	1,2,3,4
4	3	1,4,2
5	3	1,2,3
6	3	1,4,3
7	3	2,4,3
8	3	3,5,6
9	2	1,4
10	2	1,2
11	2	1,3
12	2	4,3
13	2	2,3
14	2	3,5
15	2	3,6
16	2	5,6

Max-clique size = (4)

Whenever we need to prove a given problem is NP-hard, we need to prove a problem in two steps.

Step 1 - Decision problem

Try to convert this clique problem into decision problem, where by asking some kind of question we can take some sort of decisions.

Step 2 - NP-Hard

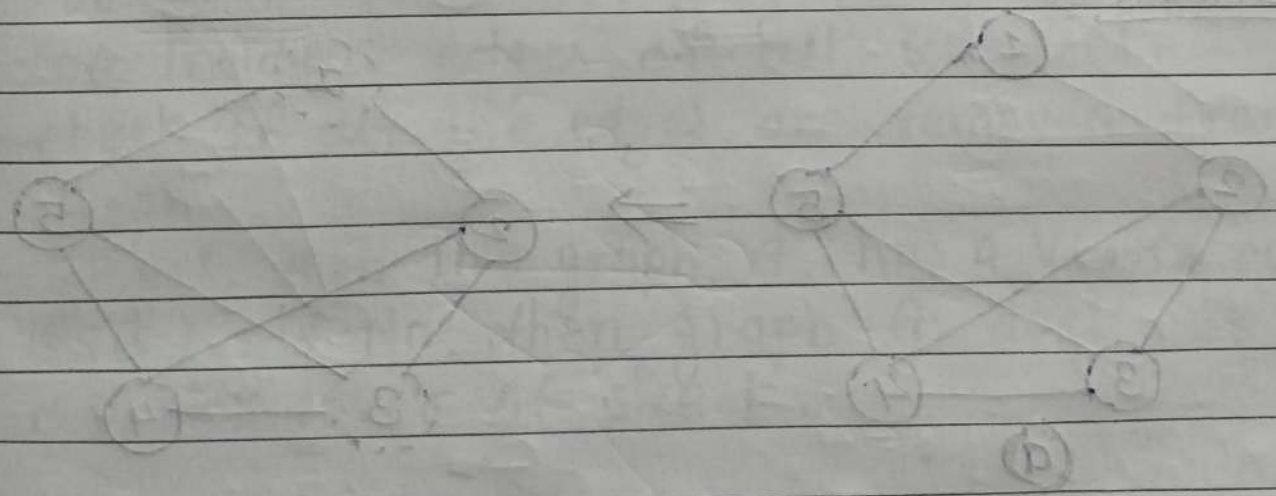
Prove that clique is NP-Hard.

Questions - Decision problem

1) Is there any ~~clik~~ clique of size 4?
2) Is there any clique of size 8?
3) What is the max-clique size in the graph.

In step (1) we could convert the clique problem into decision problem. i.e we could ask decision based questions about clique.

Hence we proved step (2)



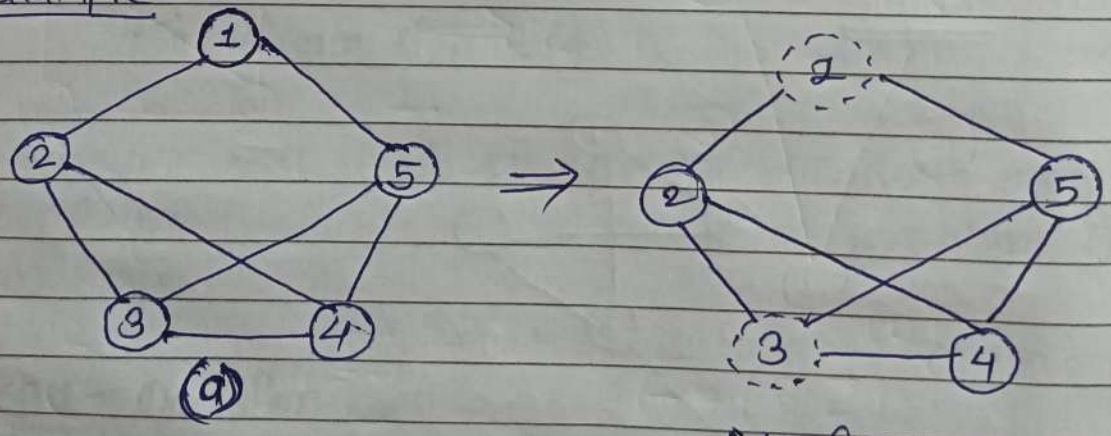
NP-Complete Problem - Vertex cover problem
prove that vertex cover problem is NPC.

solution -
3 steps to be proved

step 1 - Prove that it is Decision problem
step 2 - Prove that it is NP problem } NP-complete.
step 3 - prove that it is NP-hard }

Vertex Cover problem
- A vertex cover of a graph is a set of vertices that touches every edge in the graph.
- A vertex cover of a graph $G = (V, E)$ is a subset $V_c \subseteq V$ such that
if $(a,b) \in E$ then either $a \in V_c$ or $b \in V_c$ or both $\in V_c$

Example -



$V_c = (2,4,5)$
$|k| = 3$

In vertex cover problem, it is set of vertices that cover all the edges by removing unwanted vertex.

Step 1 - Prove that it is decision problem

I/P = Graph G & K

Statement - Vertex cover graph takes input as graph G and integer K.
- Now ask, is there any vertex cover of size K ? the answer can be either yes or No.
- In our case, check is there any vertex cover of size 3 ?
  Ans ⇒ yes.
- Hence proved that vertex cover is a decision problem.

Step 2 -

Statement - Given V_c, Vertex cover of $G = (V, E)$ and $|V_c| = K$ (size of V_c).
- For each vertex belonging to V_c, remove all the incident edges of all vertices.
- check if all the edges are removed from the graph
- If yes, then this graph G, has a vertex cover of size K, if No, then graph G does not have a vertex cover of size K.

Now, This can be easily be done in polynomial time. the non-deterministic algo would take all vertices at a time and remove all incident edges parallely.
This problem can be solved in polynomial time but by non-deterministic algorithm.
Hence proved it is NP problem.

Step 3 - To prove that it is NP-Hard

Reduce clique decision problem to vertex cover prob

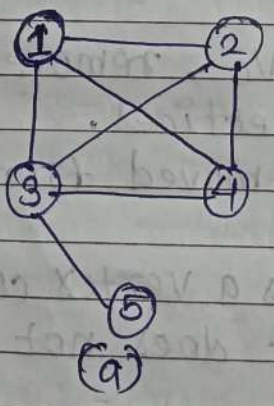
Clique decision problem $\leq_p$ Vertex Cover problem

reduces to $\leq_p$ in polynomial time

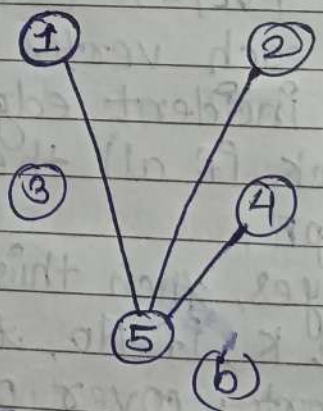
Given for CDP

$G=(V,E)$, $|V|=n$, clique $=k$ reduces to $\leq_p$ in polynomial time $\bar{G}=(V,\bar{E})$ if vertex cover of size $|V|-k$

Example -



(a)



(b)

$G=(V,E)$, clique $=4$

$\bar{G}=(V,\bar{E})$.

Vertex cover

$k=1$

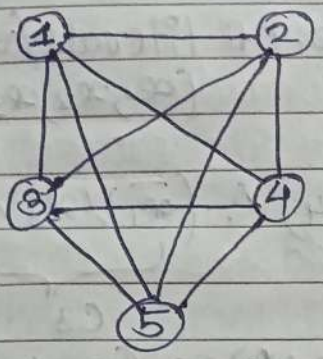
$V_c=(5)$

$|V|-k=5-4=1$

↓

clique value size

By combining both 2 graphs.



let input for clique problem be graph $G=(V,E)$ and an integer K, such that Graph G has a max clique of K if & only if $\overline{G}=(V,E)$ has a vertex cover of size $|V|-K$. → (A) K is clique value:- size K.

- Graph (a) ⇒ clique size = 4
- Graph (b) ⇒ vertex cover K=1
- statement (A) has been proved.

Hence Vertex-Cover is NP Hard
Vertex cover is NP as well as NP Hard, hence it is proved that it is NP-complete.

NP-Complete Problem :- 3-Sat problem
To prove that 3-sat problem is NP-Complete

<u>3 steps proof</u> -

1) step 1 - Prove that it is Decision Problem.
2) step 2 - Prove that it is NP-class problem. } NP-
3) step 3 - Prove that it is NP-hard problem. } output

3-sat problem ?
sat = satisfiability problem
↓
CNF formula → Boolean formula

Page No.
Date

↓
Conjunctive Normal form formula
↓
No. of clauses with same no. of literals/variables ($x_1, x_2, x_3 \ldots$)

3 set ?

ex- $(x_1 \vee \bar{x}_1 \vee x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee x_3 \vee x_4)$

C_1 C_2 C_3

$x_1 = 1$ $(1 \vee 0 \vee 0) \wedge (1 \vee 0 \vee \phi) \wedge (0 \vee 0 \vee \phi)$
$x_2 = 1$ $= 1 \wedge 1 \wedge 1$
$x_3 = 0$ $= 1$
$x_4 = 1$. $=$ True

Step 1 - Prove that it is a decision problem.

→ After substituting the value as 1/0 in the given 3-CNF formula, are we getting the result in terms of 0/1 ?

→ Yes ⇒ for the example that we considered, we got the ans, as 1.
So, we proved that 3-sat problem is a decision problem.

Step 2 - Prove that ~~it is~~ 3-sat is NP-class problem.

for this, we can consider a non-deterministic algo that takes a CNF-boolean formula with 3 variable per clause as the input, Compute the values for the clauses & finally check whether the formula evaluates to the values = 1 or True,

then we say that 3-sat problem is a NP-class problem.

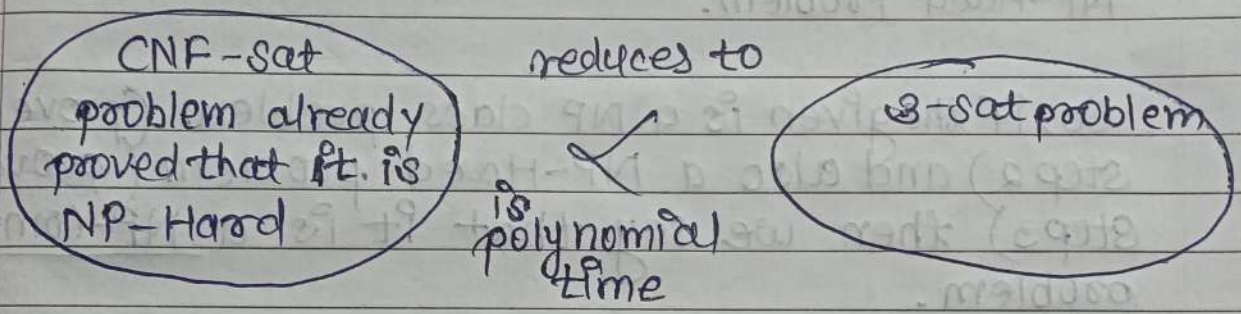
ex-

$(m+n+z)\cdot(m+n+\bar{z})=1$

Hence it is proved that 3-sat problem is a NP-class problem.

Step3 - Prove that 3-sat problem is NP-Hard.

→ Reduction



ex-

Consider a 2-variable CNF formula

$C_i = (m+n) \Rightarrow (1,2,3)$ 3-sat

$C_i = (m+n).1$

$C_i = (m+n)\cdot(z+\bar{z})$ // $z=1$, $\bar{z}=0$.

$\Downarrow$

$(m+n+z)\cdot(m+n+\bar{z})$

ex- CNF formula with 1-variable

$C_i = m$

$C_i = m\cdot 1$

$C_i = m.(n+\bar{n})$

$C_i = (m+n)\cdot(m+\bar{n})$

$C_i = (m+n)\cdot(m+\bar{n})\cdot 1$

$C_i = (m+n)\cdot(m+\bar{n})\cdot(z+\bar{z})$

$C_i = (m+n+z) \cdot (m+n+\bar{z}) \cdot (m+\bar{n}+z) \cdot (m+\bar{n}+\bar{z})$

We could convert 1-variable and 2-variable CNF boolean formula into 3-CNF formula
But, CNF sat formula is already proved as a NP-Hard problem.
Due to the reduction of 1-Variable & 2-Variable CNF into 3-CNF formula
we proved that 3-sat problem is also a NP-Hard Problem.

So, if the given is a NP class problem (proved in step 2) and also a NP-Hard problem (proved in step 3) then we say that it is a NP-Complete problem.
Hence, 3-sat problem is a NP-Complete problem.

Unit 6 :

Parallel Computing ....

* **Parallelism** – It is a process of executing several set of instructions simultaneously. Parallelism can be performed using parallel computers.

* **Parallel Computers** – Computers having more than one processors.

* **Algorithm** – It is a set of instructions followed to solve a problem.

Algorithm

Sequential algorithm	Parallel algorithm
– single processor	– multiple processor
– executes instructions in sequence, one after another.	– tasks divided into processors, & all work in parallel & later on the results are combined.
– low performance	– High performance

(a) Problem → Processor

fig. Serial Computing

(b)

Problem → Task 1 → Processor 1
Task 2 → Processor 2
Task 3 → Processor 3

fig- Parallel Computing

IMP

* **Principles of Parallel Algorithm –**

Design –

① Identify the portions of problem / work that can be executed concurrently.

② Map concurrent portions of work onto multiple processes running in parallel.

③ Distribute programs input, output & intermediate data.

④ Manage access to stored data & avoid conflicts.

⑤ Synchronize the processes at stages of the parallel program execution.

Nodes = represents tasks
Directed Edge = represents control dependencies.

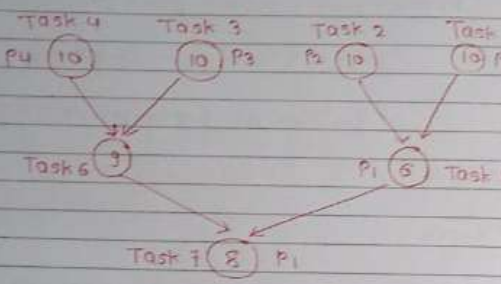


fig- Parallel Computing

IMP
* Parallel Algorithm Approaches -

① Existing sequential algorithms are molded to form parallel algorithms.

② Developing a new parallel algorithm.

③ Picking of the shelf parallel algorithm according to our requirements.
(i.e. selecting one from already available parallel algorithms)

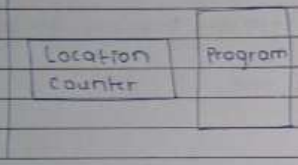
IMP
* Diff between Sequential Processing s/m & Parallel processing s/m -

Sequential processing	Parallel Processing
① Single processor system.	Multiple processor system.
② Processor performs the tasks in	

IMP
* RAM & PRAM Models -

① RAM - (Random Access Machine Model)
- Algorithms can be measured in a machine-independent way using RAM model.
- This model assumes a single processor.

Read only input tape | x_1 | x_2 | ..... | x_n |



* Optimal Parallel Algorithms -

Parallel algorithms are beneficial as compared to sequential algos.

↑

Less time for its execution

Speedup = Ratio of parallel time to Sequential time.

Ratio of computational time from the sequential algorithm to the time for the parallel algorithm is speedup.

Exact optimization method guarantees finding ~~att~~ an optimal soln whereas

Heuristic optimization method does not guarantee for optimal soln.

* Multithreading -