Backtracking

* It is a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

* Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fails to satisfy the constraints of the problem at any point of time

* Backtracking uses Brute-force approach.

* Backtracking is not for optimization problem

* Backtracking is used when, you have multiple solutions and we need all those solutions

* State space tree

* In Backtracking algorithm or technique can be solved can be represented by a tree: the solution tree is also called as State space tree the problem and solution

* Its root represents an initial state before the search

The problems that we can solve by Backtracking have the constraints.

The Backtracking try to check those constraints and get the solution which satisfy the constraints.

bounding function

* If no bounding function is applied untill it reaches the last level, then we get solutions in Backtracking.

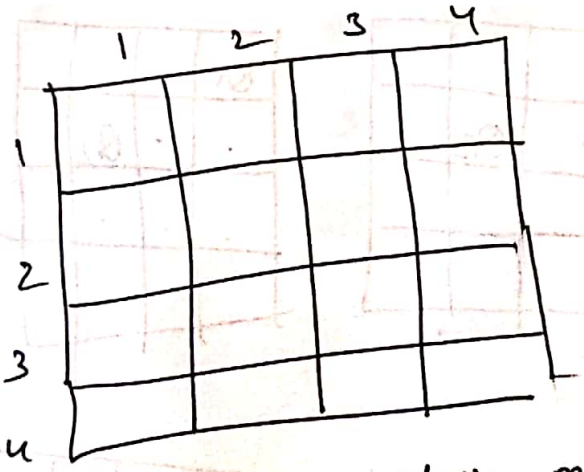
* Backtracking uses DFS approaches.

N Queen Problem

* The N Queen is the problem of placing N chess queen on n x n Chessboard so that no two queens atta each other

* In chess, a queen can attack each other when they are placed at
  - Horizontally
  - vertically
  - Diagonally.

* ∴ no two queens are on the same row, colou and diagonal



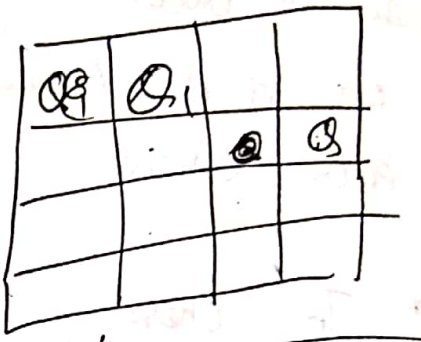
4x4 matrix

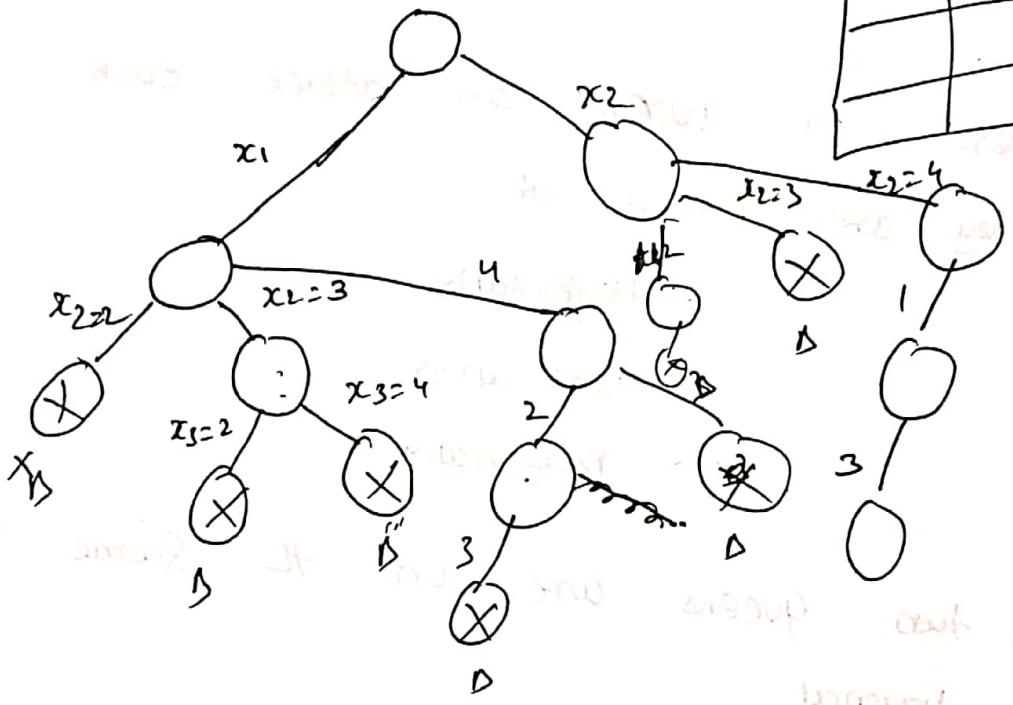
* We can place queens that are not under attac

* There are more than one solution, & we nee all those solution.

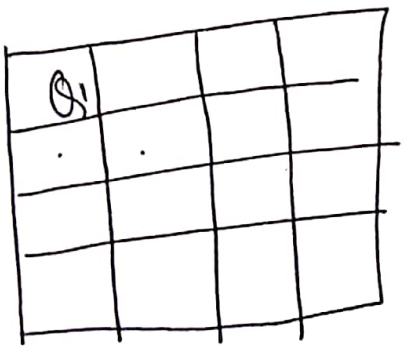
* Backtracking technique can be applied to solve the

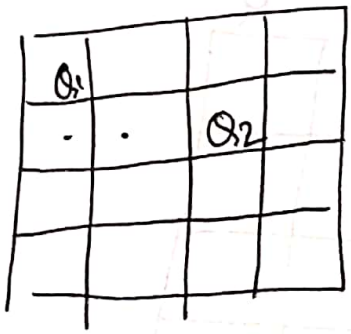
Problem

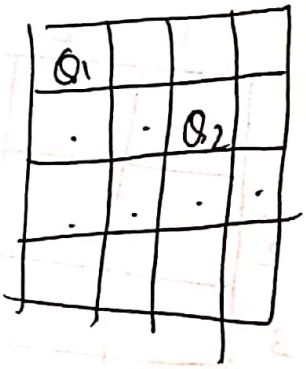


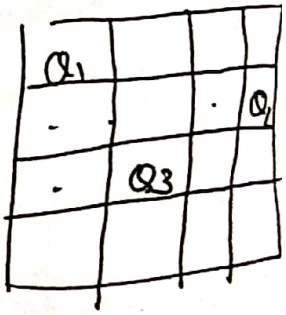


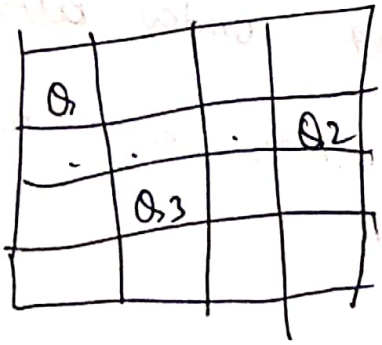
2	4	1	3

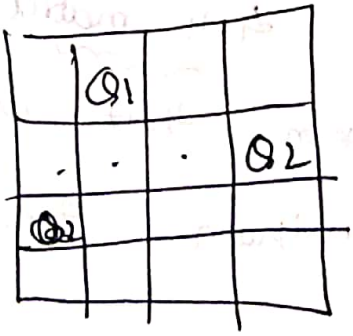


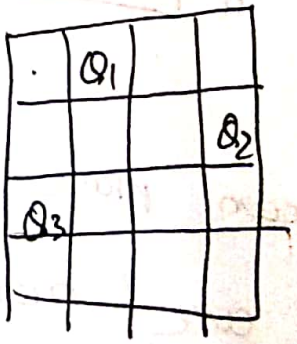


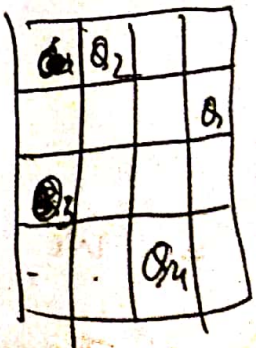












∴ Solution = 2, 4, 1, 3

Algorithm

```
Queen(n)
{
    k=1;        // select the first Queen
    a[k]=0;     // but dont place on the board
    while(k!=0) // check a queen exist ?
    {
        a[k]=a[k]+1;   // place the kth Queen on the next col
        while(a[k]<=n && !Place(k)) // satisfy condition
        {
            a[k]++;    // place Queen in next column
        }
        if(a[k]<=n)    // if queen placed successfully on the board ?
        {
            if(k==n)   // if all queens are placed
            {
                print(x1,x2...xn) Print Solution
            }
            else
            {
                k++;   // select next queen
                a[k]=0; // but dont place
            }
        }
        else
        {
            k--;       // Backtrack and select previous Queen.
        }
    }
}
```

Algorithm for place

```
int place (int pos)
{
    int i;
    for(i=1; i< pos; i++)
    {
        if (( a[i] == a[pos]) ||((abs (a[i] - a[pos]) ==
                                  abs (i - pos))
        {
            return 0;
        }
    return 1;
}

void Print san(int n)
{
    int i, j;
    count ++;
    printf(" \n Solution = %d \n", count);
    for(i=1; i<=n; i++)
    { for(j=1; j<=n; j++)
        { if (a[i] == j)
            { printf ("& I \t");
            }
        else
        { printf (" 0 \t");
        }
```

Time complexity

	B.T	A.C	W.C
n-Queen	O(n!)	O(n!)	O(n!)

Sum of Subset Problem

* It is defined as, "Given 'n' positive numbers $n_1, n_2, n_3 \dots n_n$ and m and it is required to find all subsets of n_i where $1 \le i \le n$ whose sum is m.

Example :-

$w[1:5] = \{1, 2, 5, 6, 8\}$ and

$n = 5$ → total objects

$m = 9$ → sum of subset

* Two solution can be found

$\{1, 2, 6\} \Rightarrow 9$

$\{1, 8\} \Rightarrow 9$

// construct the state-space tree for sum of subset problem given the following data $w = \{5, 10, 12, 13, 15, 18\}$ and $n = 6$ and $m = 30$

Bounding function :-

x_1	x_2	x_3	x_4	x_5	x_6
1	1	0	0	1	0

$= 30$

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

2) $\sum_{i=k+1}^{n} w_i x_i + \sum_{i=k+1}^{n} w_i > m$

Example

Bounding Function for Program

```
if(k<=n && x[k]==1)
{
    if(s+w[k]==d)          // Bounding function
    {
        printf("Print the solution");
        for(i=1; i<=n; i++)
        {
            if(a[i]==1)
            {
                printf("%d it", w[i]);
            }
        }
        println("in");
        x[k]=0;
    }
    else if(s+w[k]<d)      // Bounding function
    {
        s+=w[k];
    }
    else
    {
        x[k]=0;
    }
}
```

else // Backtrack

{

```
k--;
while(k>0 && x[k]==0)
{
    k--;        // once again backtrack
}
if(k==0)
    Break;
x[k]=0;         // dont consider that object
s=s-w[k];       // subtract it not include object weig
```

}

```
k=k+1           // select next object
x[k]=1;         // include next object
```

problem :-

w = {3, 5, 6, 7}

n = 4

m = 15

Bounding function

1) s + w[k] < d

sum of the object + weight of nex
object must be not be greater
the total value (d)
* If it is greater kill that no

2) $s + \sum_{j=i+1}^{n} a_j > d$

sum of the object + remaining weight
the object should be not be lesser
than total value
* If its lesser, kill that node

Algorithm for subset sum problem

Step1: Initialization of Object's weight

$x[k] = 1$

Step2: Bounding Function:

* If we have the value of a node/object is equal to d, we have the solution to the problem.

```
If (weightsoFar + w[k] == d)
        // Print the Solution
        for i ← 1 to n do
            If a[i] == 1
                printf("%d ", w[i]);
            end if
        end for
end if
```

Step3: If above condition is not satisfy then check the condition

```
If (weightsoFar + w[k] < d)
        $weightsoFar = weightsoFar + w[k];
end if
else
        x[k] ← 0;   // dont consider this object.
```

Step 4 :- Backtrack : If the value of a node/object is not equal to d, continue by backtracking to the node's parent to get the remaining solution if any

```
k--;            Backtrack

while (k>0 && x[k]==0)
        k--;        backtrack

x[k]=0;      // dont consider that object
weightsofar = weightsofar - w[k]   // subtract the
                                      not included object weight
```

and Repeat step 1.

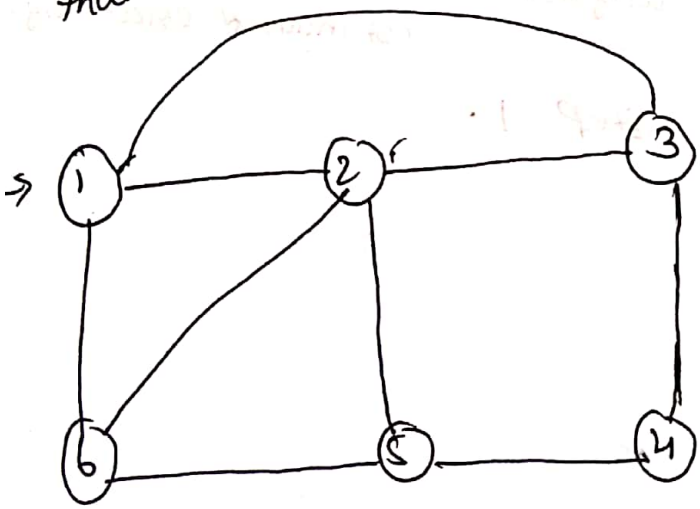
Hamiltonian Circuit Problem

Hamiltonian Circuit or Cycle

* Hamiltonian cycle is an Hamiltonian path such that there is an edge from the last vertex to the first vertex of the Hamiltonian path

Hamiltonian path

* Hamiltonian path in an undirected graph is the path that visits each vertex exactly once.

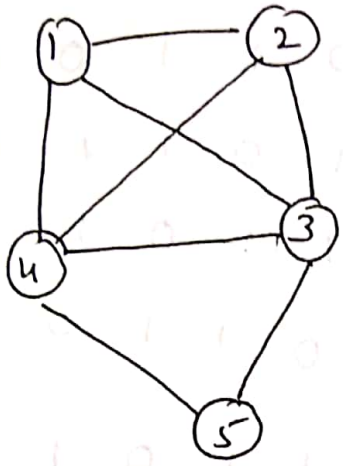


1, 2, 3, 4, 5, 6, 1 → Hamiltonian cycle

1, 2, 6, 5, 4, 3, 1 → Hamiltonian cycle

1, 6, 2, 5, 4, 3, 1 → Hamiltonian cycle

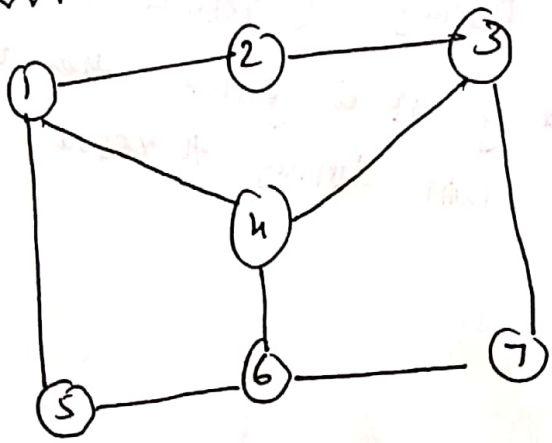
Example 2



1, 2, 3, 5, 4, 1 → 1 @ cycle

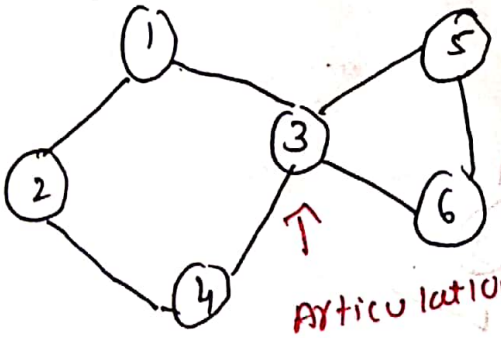
1, 2, 4, 5, 3, 1 → 2^{nd} cycle

Example 3



Hamiltonian path is not Exist in this grap

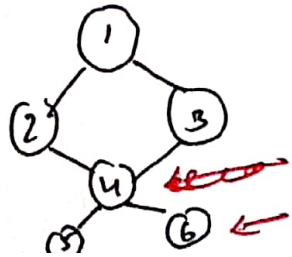
Example 4



Articulation point

* If any graph that contain articulation poin then Hamiltonian Cycle is n possible

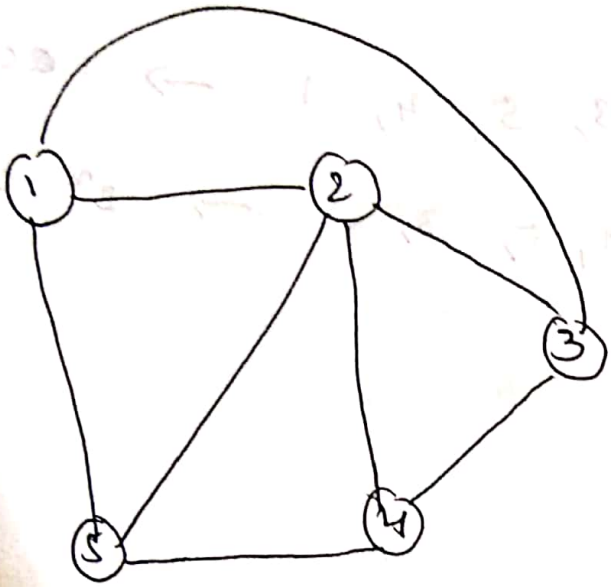
Example 5



Pendant vertex

* If any graph that contain pendant vertices or vertex, then Hamiltonian cycle is not possib

Hamiltonian cycle



$G =$

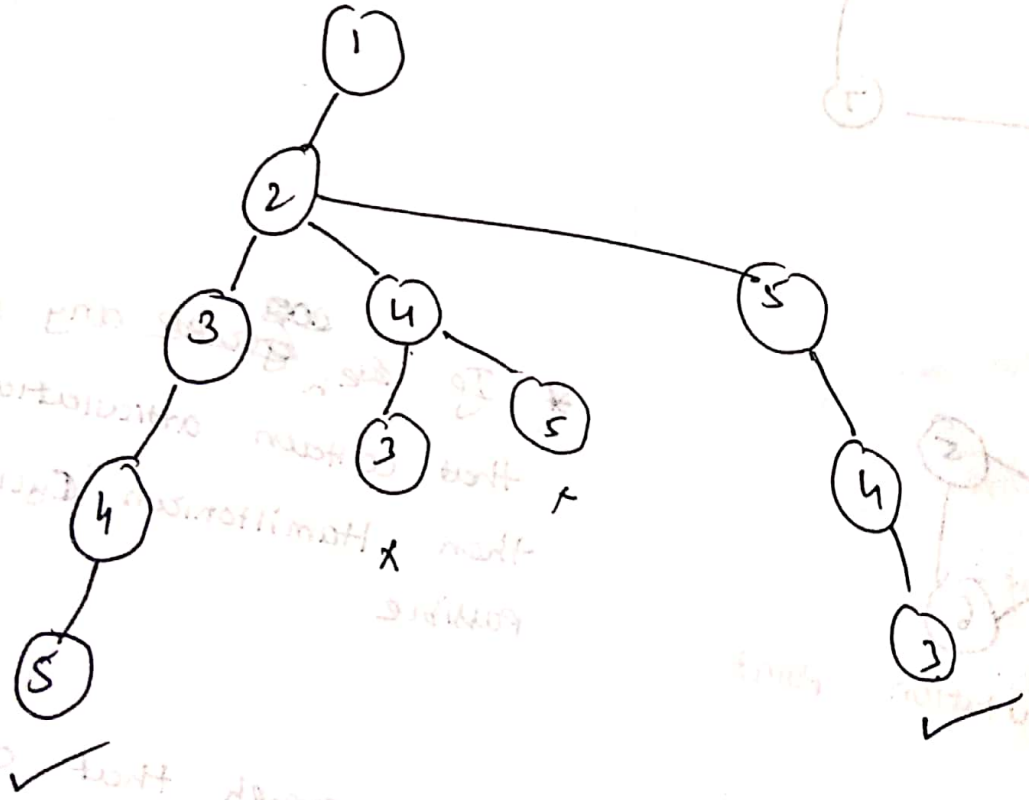
	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	1	1
3	1	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

x

0	0	0	0	0
1	2	3	4	5

← * This array is used to Display the results

* If it is filled, then we will display the result



Algorithm

```
Hamiltonian (k)
{
    do
    {
        nextvertex(k)
        if (x[k]==0)
            return;
        if (k==n)
            print (solution x[1:n])
        else
            Hamiltonian (k+1);
    } while (true);
}
```

Algorithm nextvertex (k)

```
{
    do
    {
        x[k] = (x[k]+1) mod (n+1);   // for generating no of v
        if (x[k] ==0) return;
        if ( G[x[k-1], x[k]≠0)   // there should be edge to previous vertex
            for j=1 to k-1 do   ← for checking duplicates
                if (x[j] == x[k]
                    break;
        if (j==k)
            if (k<n or (k==n) && G[x[n], x[1]] ≠ 0
                return;
    } while (true);
}
```

Algorithm	Time complexity
1) fact(n)	$O(n)$
2)* fib(n)	$O(2^n)$
3) Sequential search	Best case → $O(1)$ Average case → $O(n)$ Worst case → $O(n)$
4) Binary search	→ B.C → $O(1)$ A.C → $O(\log_2 n)$ W.C → $O(\log n)$
5) Insertion, Deletion into BST	→ $O(\log_2 n)$
6) Mergesort	→ $O(n \log n)$
7) Quick sort	→ Best.C → $O(n \log n)$ W.C → $O(n^2)$
8) Bubble sort	→ $O(n^2)$
9) Selection sort	→ $O(n^2)$

10) Insertion sort

B.C → O(n)

W.C → $O(n^2)$

A.C → $O(n^2)$

11) Searching, insertion or deleting key

- BST → O(log n) → O(n)
- AVL → O(log n)
- BT → O(log n) → O(n)
- B^+ → $O(\log_m n)$

12) Knapsack (greedy method) → O(n log n)

13) Optimal merge patter → O(n log n)

14) Prim's → O(V+E) log V) [∵ Priority Que]

$O(n^2)$ [∵ Adjacency matrix]

15) Krushkal's algorithm → $O(n^2)$ → [∵ Adjacency matrix]

O(n log n) → [∵ Priority Queue]

16) Dijkstra's algorithm → $O(n^2)$

17) DFS → O(V+E)

18) BFS → O(V+E)

19) Topological Sorting → O(V+E)

20) TSP → $O(n^2 2^n)$

21) 0/1 Knapsack → $O(2^n)$

22) All pair Shortest path → $O(n^3)$

23) Finding connected component → $O(V+E)$

24) Finding Bi-connected component → $O(V+E)$

25) Hamiltonian path → $O(2^n)$

26) N Queen → $O(n!)$

27) Sum of subset → $O(2^n)$

28) multistage graph → $O(V+E)$

29) Construction of

BT → $O(n^2)$

BST → $O(n \log n)$ — $O(n^2)$

AVL → $O(n \log n)$

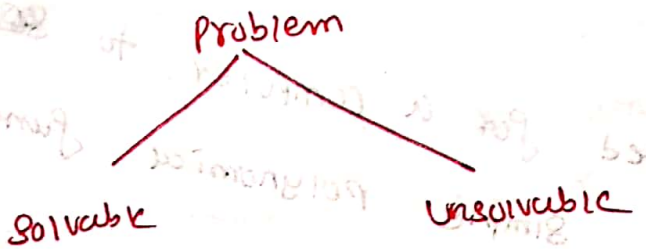
B tree → $O(n \log_2 n)$

B+ tree → $O(n \log_m n)$

P, NP and NP-complete Problem

What is Problem

* A problem is anything you're trying to solve computationally.
* TL problems typically specify an input and a desire output
* Problem in computer science can be catergorie it two parts
  - Solvable
  - Unsolvable



Solvable Problem :-

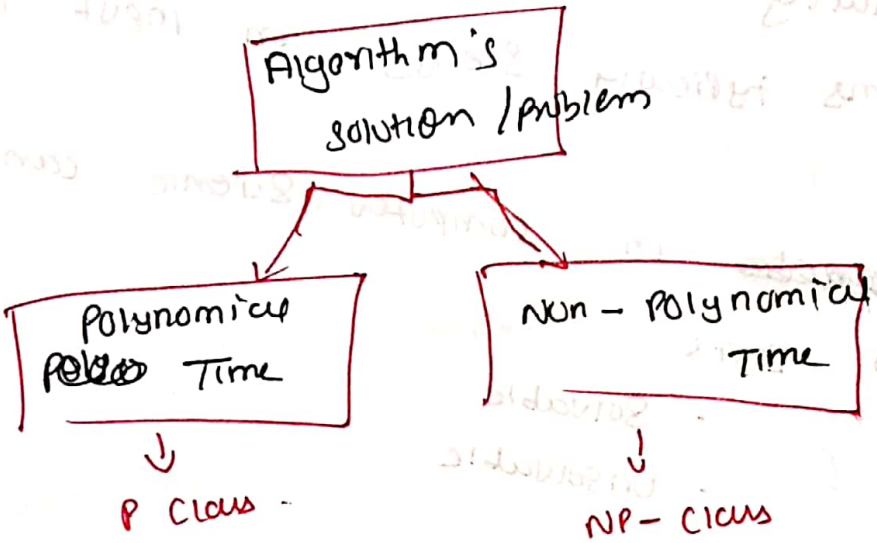
A problem is solvable if you can solve it
Example: Adding two number

Unsolvable problem :-

A problem is unsolvable if you cannot solve it right now but in future you may solve it.
Example: - video call in 16th century was unsolvable
- but right now its solvable.

* If a problem is given, we can write an algorithm for that problem.

* So, Solving of algorithm can be of two types



Polynomial Time

* The time required for a computer to solve a problem where time is simple polynomical function of the size of the input

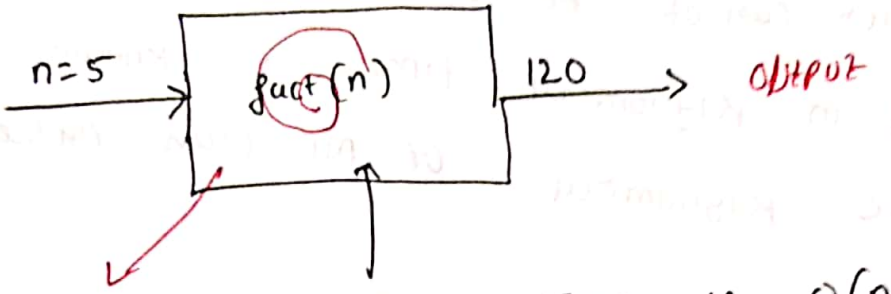
P-class

* A problem which can be solve in polynomial time are called P - class problem

* All the algorithm listed in previous page except comes under P- class problem [except 2, 20, 21, 25]

Example: . Searching algorithm
. Sorting algorithm etc

Example



Deterministic algorithm

* In computer science, a deterministic algorithm is an algorithm which
  - given a particular input, will always produce the same output.

* An algorithm whose behaviour can be completely predicted from the input.

Ex: 2+3 = 5, all sorting algorithm & searching algorithm

Nondeterministic algorithm

* In computer science, a nondeterministic algorithm is an algorithm which may produce different results for same input

* An algorithm whose behaviour can not be completely predicted from the input

Ex: choice [1, 2, 3, 4, 5]

NP - Class Problem

* A problem which cannot be solved on polynomial time but is verified in polynomial time is known as non-deterministic polynomial or NP-class problem

* A problem which can be verified in polynomial time is called NP-class problem

Example:-

- SU-DO-KU
- Travelling Salesman
- [illegible] problem
- prime factor
- scheduling etc

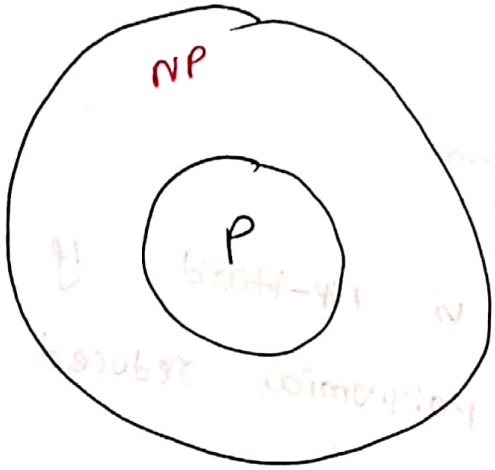
Note:

* All the problems which can be solved in polynomial time can also verified in polynomial time so P is subset of NP i.e $P \subseteq NP$

Prove $P \subseteq NP$

NP class

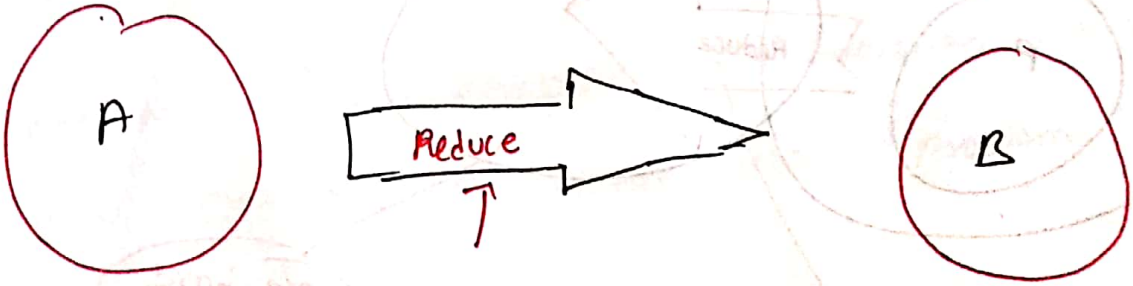
- Hard to solve and
- Easy to verify in Polynomial time
- takes Exponential time



P class

- Easy to solve
- Easy to verify
- Takes polynomial Time

Reduction



Polynomial Time

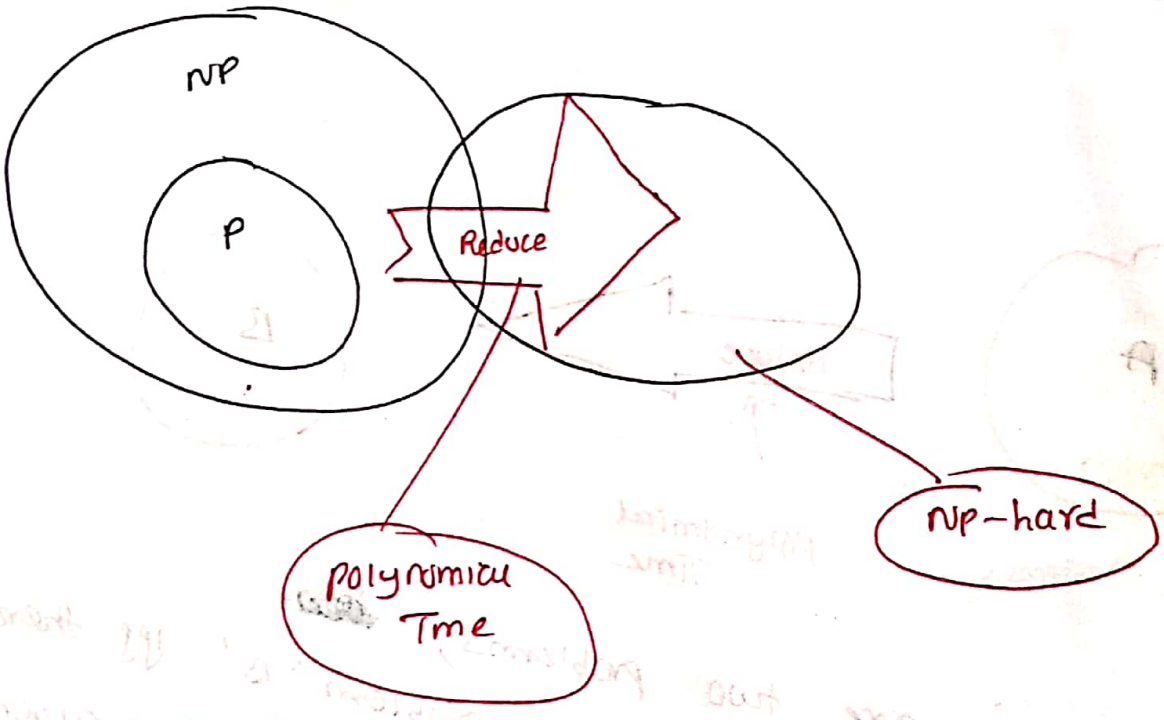
* Let 'A' & 'B' are two problems,
* Then problem 'A' reduces to problem 'B' iff there is a way to solve 'A' by deterministic algorithm that solve 'B' in polynomial time
* If 'A' is reducible to 'B' we denoted by $A \propto B$

properties of Reduction

1. If A is reducible to B and B in P then A in P
2. A is not in P implies B is not in P

NP- Hard Problem

* A problem is NP-Hard if every problem in NP can be polynomial reduced to it.

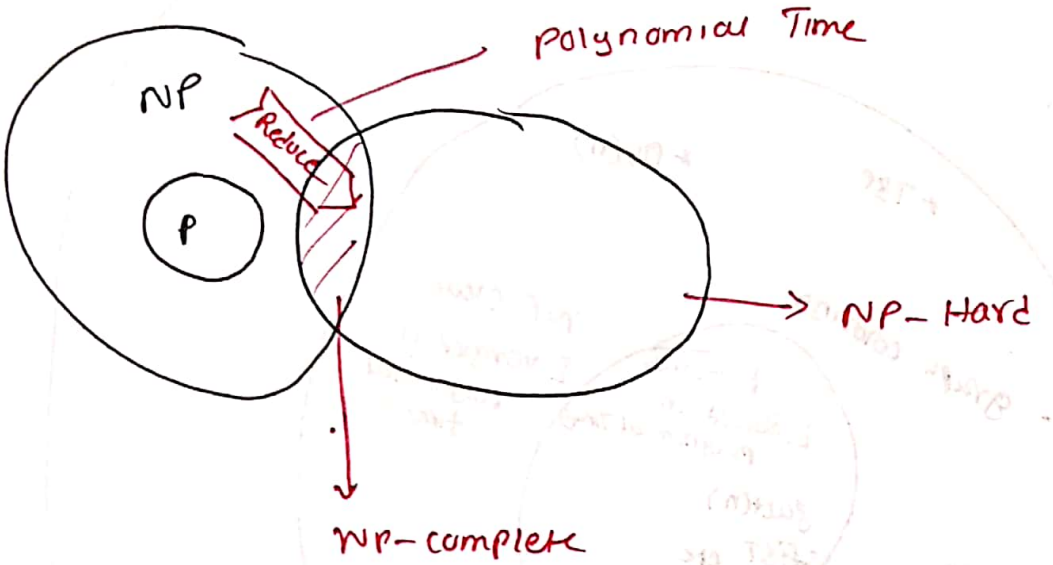


Simple definication

* A problem which cannot be solved in polynomical Time is called NP-hard

NP- Complete

* A problem is NP- complete if it is NP and it is NP hard



* NP- complete problem is the intersection of the NP & NP-hard class

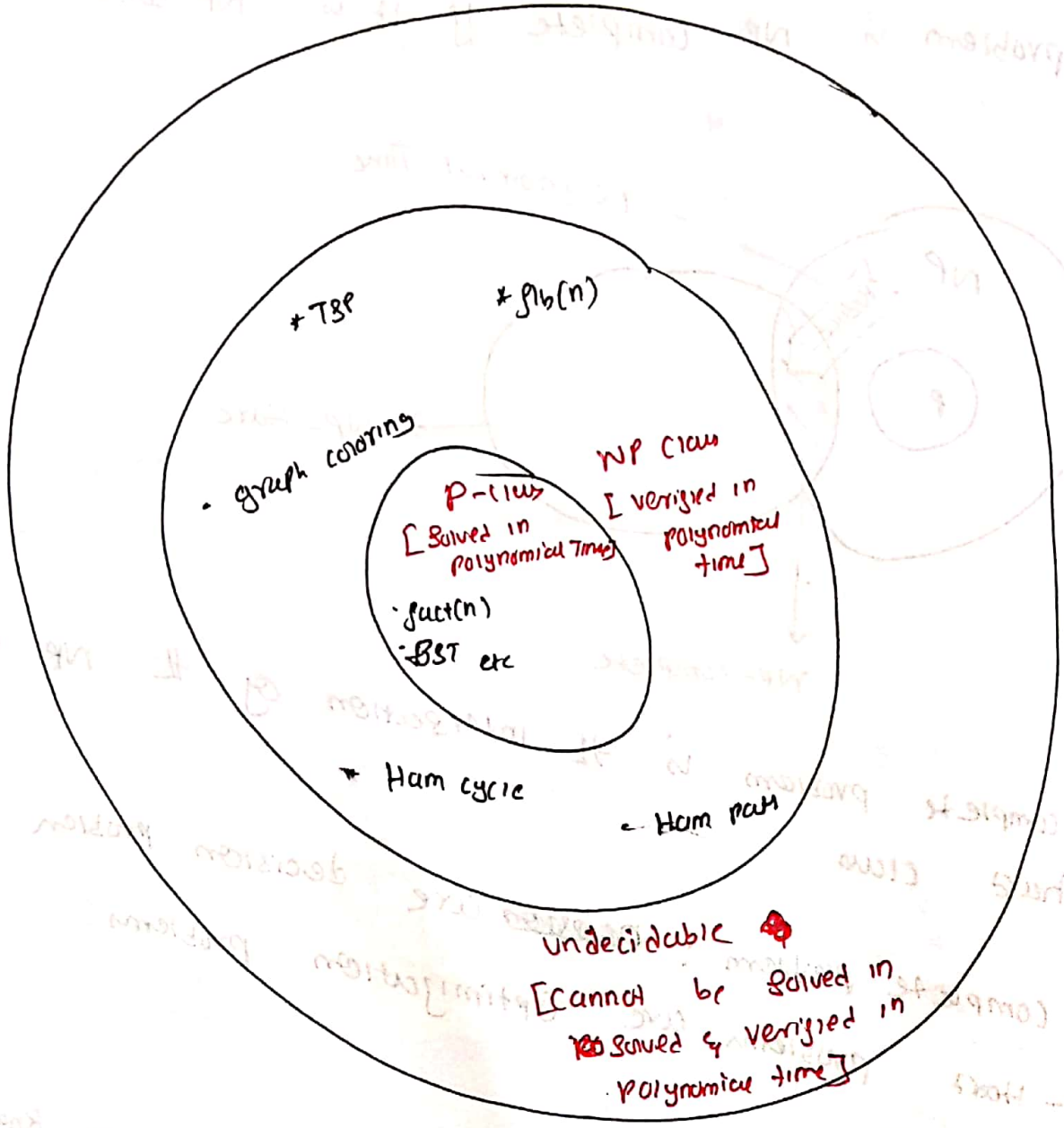
* NP complete problem : are decision problem

* NP- Hard problems are Optimization problems.

Example :

* Knapsack decision problem can be converted to Knapsack optimization problem

* All NP- Complete problems are NP-Hard but All NP- Hard problems are not NP- Complete



ven diagram