UNIT - V	**Dynamic Programming & Backtracking**	9 Hrs
Dynamic Programming: General method, applications- 0/1 knapsack problem, All pairs shortest path problem, Travelling salesperson problem, Reliability design. **Backtracking**: General method, applications-n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles. **Introduction to NP-Hard and NP-Complete problems**: Basic Concepts.		

Dynamic Programming

Greedy Algorithms vs Dynamic Programming(2 Marks)

- Greedy Algorithms are similar to dynamic programming in the sense that they are both tools for optimization.
- However, **greedy algorithms** look for locally optimum solutions or in other words, a greedy choice, in the hopes of finding a global optimum. Hence greedy algorithms can make a guess that looks optimum at the time but becomes costly down the line and do not guarantee a globally optimum.
- **Dynamic programming**, on the other hand, finds the optimal solution to subproblems and then makes an informed choice to combine the results of those subproblems to find the most optimum solution.
- **Dynamic programming can be implemented in two ways –**
    - Memoization**(2 Marks)**
    - Tabulation**(2 Marks)**
- **Memoization –** Memoization uses the top-down technique to solve the problem i.e. it begin with original problem then breaks it into sub-problems and solve these sub-problems in the same way.
- **Tabulation –** Tabulation is the typical Dynamic Programming approach. Tabulation uses the bottom-up approach to solve the problem, i.e., by solving all related sub-problems first, typically by storing the results in an array. Based on the results stored in the array, the solution to the "top" / original problem is then computed.

PROPERTIES OF DYNAMIC PROGRAMMING(2 Marks)

The problems that can be solved by using Dynamic Programming has the following two main properties-

1. Overlapping sub-problems
2. Optimal Substructure

1) Overlapping Subproblems:

- Overlapping subproblems is a property in which a problem can be broken down into subproblems which are used multiple times.
- Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a array so that these don't have to recomputed. So Dynamic Programming is not useful when there are no overlapping subproblems because there is no point storing the solutions if they are not needed again.

2) Optimal substructure

Optimal substructure is a property in which an optimal solution of the original problem can be constructed efficiently from the optimal solutions of its sub-problems.

Example :

Consider a program to generate Nth fibonacci number

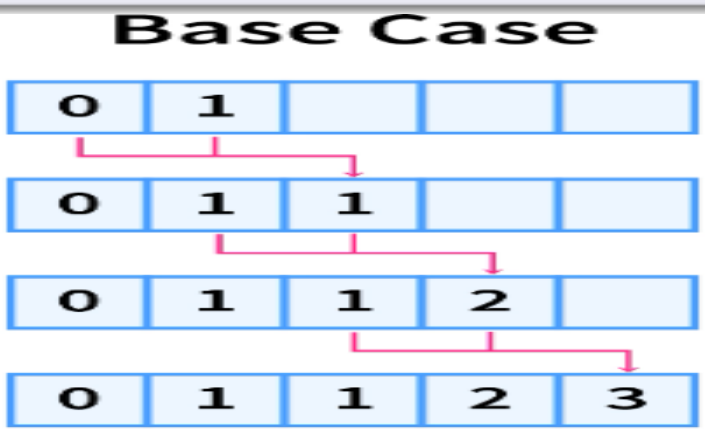
Let's find the fibonacci sequence upto 5th term. A fibonacci series is the sequence of numbers in which each number is the sum of the two preceding ones. For example, 0, 1, 1, 2, 3. Here, each number is the sum of the two preceding numbers.

Naive Recursive Function

```
int Fib(int n){
  if(n==1 || n==2)
    return 1;
  return Fib(n-1)+Fib(n-2)
}
```

DP Solution O(n)

```
Fib[1] = Fib[2] = 1;
for(i=3;i<N;i++)
  Fib[i] = Fib[i-1]+Fib[i-2]
```



Applications of dynamic programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. All pair Shortest path problem
4. Travelling salesperson problem
5. Longest common subsequence (LCS)
6. Flight control and robotics control
7. Time sharing: It schedules the job to maximize CPU usage
8. Reliability design.

0/1 KNAPSACK PROBLEM

- The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. It uses dynamic programming
- Dynamic programming divides the problem into small sub-problems. Let V is an array of the solution of sub-problems. V[i, j] represents the solution for problem size j with first i items. The mathematical notion of the knapsack problem is given as :

$$V[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1, j] & \text{if } j < wi \\ \max\{V[i-1, j], vi + V[i-1, j-wi]\} & \text{if } j \geq w \end{cases}$$

Algorithm

Algorithm for binary knapsack using dynamic programming is described below

```
Algorithm DP_BINARY_KNAPSACK (V, W, M)
// Description: Solve binary knapsack problem using dynamic programming
    for i ← 1 to n do
       V[i, 0] ← 0
    end

    for i ← 1 to M do
       V[0, i] ← 0
    end

    for V[0, i] ← 0 do
       for j ← 0 to M do
          if w[i] ≤ j then
             V[i, j] ← max{V[i-1, j],  v[i]  + V[i – 1, j –  w[i]]}
          else
             V[i, j] ← V[i – 1, j]    // w[i]>j
          end
       end
    end
```

Example

Example: Find an optimal solution for following 0/1 Knapsack problem using dynamic programming: Number of objects n = 4, Knapsack Capacity M = 5, Weights (W_1, W_2, W_3, W_4) = (2, 3, 4, 5) and profits (P_1, P_2, P_3, P_4) = (3, 4, 5, 6).

Solution-

Given-

- Knapsack capacity (w) = 5 kg
- Number of items (n) = 4

Step-01:

- Draw a table say 'T' with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with 0.

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

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$

Finding T(1,1)-

We have,

- i = 1
- j = 1
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,1) = \max \{ T(1-1, 1), 3 + T(1-1, 1-2) \}$

$T(1,1) = \max \{ T(0,1), 3 + T(0,-1) \}$

$T(1,1) = T(0,1)$ { Ignore $T(0,-1)$ }

$T(1,1) = 0$

Finding T(1,2)-

We have,

- i = 1
- j = 2
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,2) = \max \{ T(1-1, 2), 3 + T(1-1, 2-2) \}$

$T(1,2) = \max \{ T(0,2), 3 + T(0,0) \}$

$T(1,2) = \max \{0, 3+0\}$

$T(1,2) = 3$

Finding T(1,3)-

We have,

- i = 1
- j = 3
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,3) = \max \{ T(1-1, 3), 3 + T(1-1, 3-2) \}$

$T(1,3) = \max \{ T(0,3), 3 + T(0,1) \}$

$T(1,3) = \max \{0, 3+0\}$

$T(1,3) = 3$

Finding T(1,4)-

We have,

- i = 1
- j = 4
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,4) = \max \{ T(1-1, 4), 3 + T(1-1, 4-2) \}$

$T(1,4) = \max \{ T(0,4), 3 + T(0,2) \}$

$T(1,4) = \max \{0, 3+0\}$

$T(1,4) = 3$

Finding T(1,5)-

We have,

- i = 1
- j = 5
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,5) = max { T(1-1 , 5) , 3 + T(1-1 , 5-2) }
T(1,5) = max { T(0,5) , 3 + T(0,3) }
T(1,5) = max {0 , 3+0}
T(1,5) = 3

Finding T(2,1)-

We have,

- i = 2
- j = 1
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,1) = max { T(2-1 , 1) , 4 + T(2-1 , 1-3) }
T(2,1) = max { T(1,1) , 4 + T(1,-2) }
T(2,1) = T(1,1) { Ignore T(1,-2) }
T(2,1) = 0

Finding T(2,2)-

We have,

- i = 2
- j = 2
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,2) = max { T(2-1 , 2) , 4 + T(2-1 , 2-3) }
T(2,2) = max { T(1,2) , 4 + T(1,-1) }
T(2,2) = T(1,2) { Ignore T(1,-1) }
T(2,2) = 3

Finding T(2,3)-

We have,

- i = 2
- j = 3
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,3) = max { T(2-1 , 3) , 4 + T(2-1 , 3-3) }
T(2,3) = max { T(1,3) , 4 + T(1,0) }
T(2,3) = max { 3 , 4+0 }
T(2,3) = 4

Finding T(2,4)-

We have,

- i = 2
- j = 4
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

$T(2,4) = max \{ T(2-1 , 4) , 4 + T(2-1 , 4-3) \}$
$T(2,4) = max \{ T(1,4) , 4 + T(1,1) \}$
$T(2,4) = max \{ 3 , 4+0 \}$
$T(2,4) = 4$

Finding T(2,5)-

We have,

- i = 2
- j = 5
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

$T(2,5) = max \{ T(2-1 , 5) , 4 + T(2-1 , 5-3) \}$
$T(2,5) = max \{ T(1,5) , 4 + T(1,2) \}$
$T(2,5) = max \{ 3 , 4+3 \}$
$T(2,5) = 7$

Similarly, compute all the entries.
After all the entries are computed and filled in the table, we get the following table-

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

T-Table

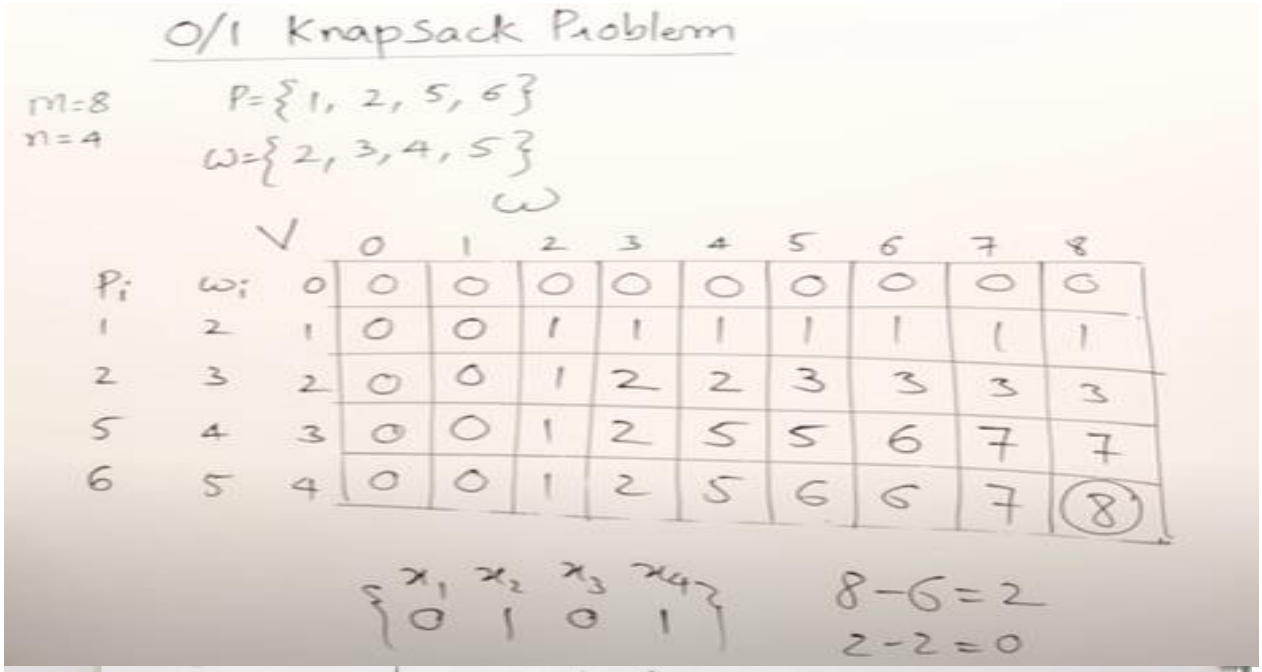
- The last entry represents the maximum possible value that can be put into the knapsack.
- So, maximum possible value that can be put into the knapsack = 7.

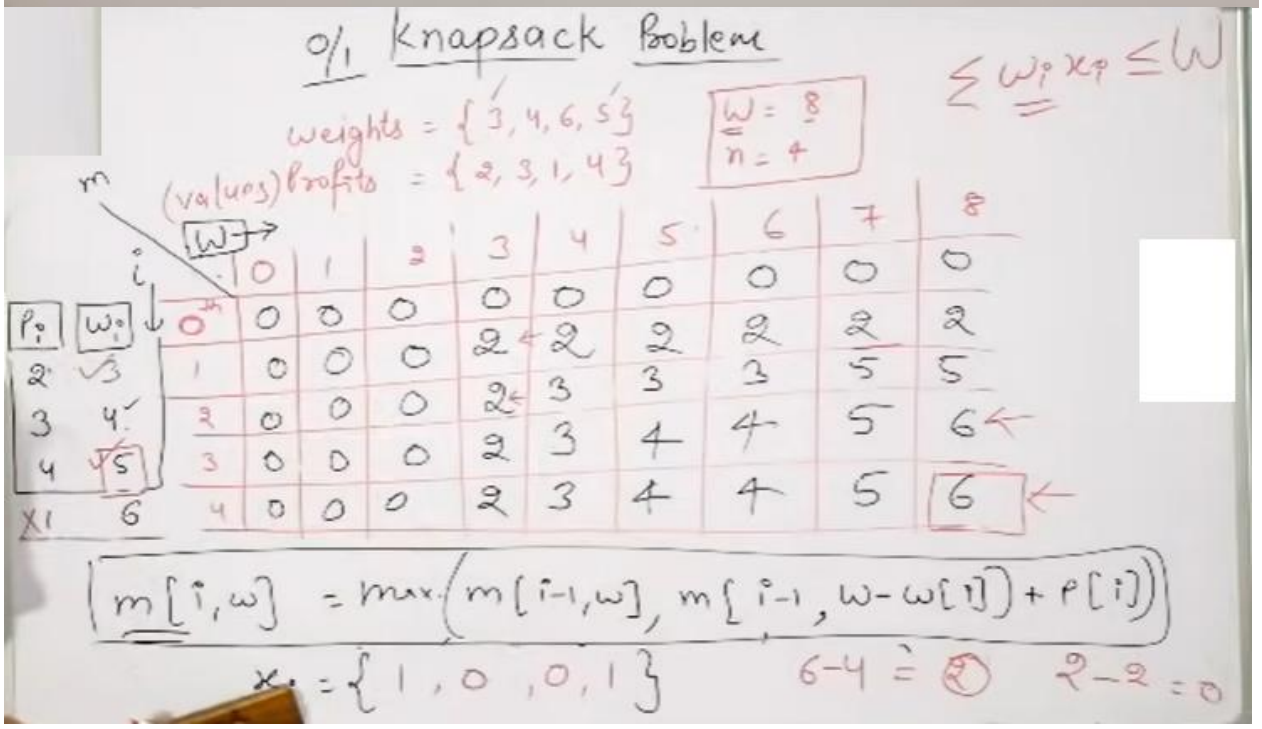
Identifying Items To Be Put Into Knapsack-

Following Step-04,

- We mark the rows labelled “1” and “2”.
- Thus, items that must be put into the knapsack to obtain the maximum value 7

Example 2 & 3:





Complexity analysis

- With n items, there exist 2^n subsets, the brute force approach examines all subsets to find the optimal solution. Hence, the running time of the brute force approach is $O(2^n)$. This is unacceptable for large n.
- Dynamic programming finds an optimal solution by constructing a table of size n ´ M, where n is a number of items and M is the capacity of the knapsack. This table can be filled up in O(nM) time, same is the space complexity.
    - Running time of Brute force approach is $O(2^n)$.
    - Running time using dynamic programming with memorization is O(n * M).

****END****

All pairs shortest path problem

- All Pairs Shortest Path Algorithm is also known as the Floyd-Warshall algorithm. And this is an optimization problem that can be solved using dynamic programming.
- Let G = <V, E> be a directed graph, where V is a set of vertices and E is a set of edges with nonnegative length. Find the shortest path between each pair of nodes.
  - L = Matrix, which gives the length of each edge
  - L[i, j] = 0, if i == j // Distance of node from itself is zero
  - L[i, j] = ∞ , if $i \neq j$ and $(i, j) \notin E$
  - L[i, j] = w (i, j), if $i \neq j$ and $(i, j) \in E$ // w(i, j) is the weight of the edge (i, j)

Algorithm:

```
procedure ALL_PATHS(COST, A, n)
    integer i, j, k, n; real COST(n, n), A(n, n)
    for i ← 1 to n do
      for j ← 1 to n do
        A(i, j) ← COST(i, j)   //copy COST into A//
      repeat
    repeat
    for k ← 1 to n do   //for a path with highest vertex index k//
      for i ← 1 to n do   //for all possible pairs of vertices//
        for j ← 1 to n do
          A(i, j) ← min{A(i, j), A(i, k) + A(k, j)}
        repeat
      repeat
    repeat
end ALL_PATHS
```

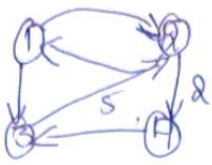
Algorithm 5.3 Procedure to compute lengths of shortest paths

Time Complexity:

- Floyd Warshall Algorithm consists of three loops over all the nodes.
- The inner most loop consists of only constant complexity operations.
- Hence, the asymptotic complexity of Floyd Warshall algorithm is $O(n^3)$.
- Here, n is the number of nodes in the given graph.

All pairs shortest path Example:- (Floyd warshal algorithm)

* Find the All pairs shortest path for given graph



Initial distance matrix for the given matrix is

$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & \infty & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

* optimal substructure formula for Floyd's algorithm is

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

Iteration 1:- $K=1$

$$D^1[i,j] = \min\{D^0[i,j], D^0[i,k] + D^0[k,j]\}$$

* For D^1 first row and first column is same as D^0 and also diagonal

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array}$$

* find 2,3 (i j) by using formula

$$D^1[2,3] = \min\{D^0[2,3], D^0[2,1] + D^0[1,3]\}$$

(from D^0 matrix)

$$= \min\{\infty, 6+(-4)\}$$
$$= \min\{\infty, 2\}$$
$$= 2$$

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & 2 & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array}$$

* Next 2,4

$$D^1[2,4] = \min\{D^0[2,4], D^0[2,1] + D^0[1,4]\}$$
$$= \min\{2, 6+\infty\}$$
$$= 2$$

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & 2 & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

Find 3,2

$D^1[3,2] = \min\{5, \infty+9\} = 5$

$D^1[3,4] = \min\{\infty, \infty+2\} = \infty$

$D^1[4,2] = \min\{D^0[4,2], D^0[4,3]+D^0[3,2]\}$

$= \min\{\infty, \infty+9\} = \infty$

$D^1[4,3] = \min\{D^0[4,3], D^0[4,3]+D^0[3,3]\} = \{1, \infty+(-4)\} = 1$

$$\therefore D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & 2 & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array} \quad //$$

Iteration 2 $k=2$

$D^2[i,j] = \min\{D^1[i,j], D^1[i,k]+D^1[k,j]\}$

* for D^2 second row, second coloumn and diagonal same as D^1

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & - & - \\ 6 & 0 & 2 & 2 \\ - & 5 & 0 & - \\ - & \infty & - & 0 \end{bmatrix} \end{array}$$

$D^2[1,3] = \min\{D^1[1,3], D^1[1,2]+D^1[2,3]\} = \min\{-4, 9+2\} = -4$

$D^2[1,4] = \min\{\infty, 9+2\} = 11$

$D^2[3,1] = \min\{\infty, 5+6\} = 11$

$D^2[3,4] = \min\{\infty, 5+2\} = 7$

$D^2[4,1] = \min\{\infty, \infty+6\} = \infty$

$D^2[4,3] = \min\{1, \infty+2\} = 1$

Therefore

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & 11 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

$\min\{D^2[i,j], D^2[i,3]+D^2[3,j]\}$

Iteration 3 $k=3$:

$D^3[i,j] = \min\{D^2[i,j], D^2[1,3]+D^2[3,j]\}$

* For D^3 Third row, Third column, and diagonal save as D^2

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & - & -4 & - \\ - & 0 & 2 & - \\ 11 & 5 & 0 & 7 \\ - & - & 1 & 0 \end{bmatrix} \end{array}$$

$D^3[1,2] = \min\{9, -4+5\} = 1$
$D^3[1,4] = \min\{11, -4+7\} = 3$
$D^3[2,1] = \min\{6, 0+6\} = 6$
$D^3[2,4] = \min\{2, 0+2\} = 2$
$D^3[4,1] = \min\{\infty, 1+11\} = 12$
$D^3[4,2] = \min\{\infty, 1+5\} = 6$

$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{bmatrix} \end{array}$$

Iteration 4 K=4:-

$$D^4[i,j] = \min\{D^3[i,j], D^3[i,4] + D^3[4,j]\}$$

* For D^4 Fourth row, column and diagonal are same

$$D^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & - & - & 3 \\ - & 0 & - & 2 \\ - & - & 0 & 7 \\ 12 & 6 & 1 & 0 \end{bmatrix} \end{array}$$

$D^4[1,2] = \min\{D^3[1,2], D^3[1,4] + D^3[4,2]\}$
$= \min\{1, 3+6\} = 1$

$D^4[1,3] = -4$
$D^4[2,1] = 6$
$D^4[2,3] = 2$
$D^4[3,1] = 11$
$D^4[3,2] = 5$

$$D^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{bmatrix} \end{array}$$

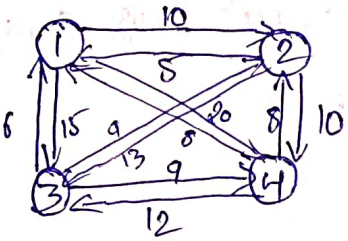
//.

Travelling salesman problem using dynamic programming

* Travelling salesman problem is to find the shortest path in a graph with the condition of visiting all the nodes only one time and returning to the origin node or city.
* For example, Given a set of n cities and distance b/w each pair of cities, find the minimum length path such that it covers each city exactly once and terminates the tour at starting city.
* This problem can be solved by finding Hamiltonian cycle of the graph
* Hamiltonian path is a path that visits each vertex exactly once and Hamiltonian cycle is Hamiltonian path such that there is edge from last vertex to first vertex.
* The distance b/w cities is best described by the weighted graph, where edge(u,v) indicates the path from city u to v and w(u,v) represents distance b/w u & v
* The optimal path can be find by using

$$g(i,S) = \min_{j \in S} \{ C_{ij} + g(j, S-\{j\}) \} \quad \text{where } i \notin S$$

Example:-



(c) Distance matrix from graph

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Solution:-

* Initially, we will find the distance b/w city 1 to {2,3,4} without any intermediate city.

$$g(2,\phi) = C_{21} = 5$$
$$g(3,\phi) = C_{31} = 6$$
$$g(4,\phi) = C_{41} = 8$$

* Next step to find the minimum distance by visiting one city. By using above formula we obtain

$$g(2,\{3\}) = C_{23} + g(3,\phi) \Rightarrow 9 + C_{31} \Rightarrow 9+6 \Rightarrow 15$$
$$g(2,\{4\}) = C_{24} + g(4,\phi) \Rightarrow C_{24} + C_{41} \Rightarrow 10+8 = 18$$
$$g(3,\{2\}) = C_{32} + g(2,\phi) \Rightarrow C_{32} + C_{21} \Rightarrow 13+5 = 18$$

$g(3,\{4\}) = c_{34} + g(4,\phi) \Rightarrow c_{34} + c_{41} = 12+8 = 20$

$g(4,\{2\}) = c_{42} + g(2,\phi) \Rightarrow c_{42} + c_{21} \Rightarrow 8+5 \Rightarrow 13$

$g(4,\{3\}) = c_{43} + g(3,\phi) \Rightarrow c_{43} + c_{31} \Rightarrow 9+6 = 15$

* next, we compute $g(i,S)$ where $i \neq 1$

$g(2,\{3,4\}) = \min\{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\}$

$= \min\{9+20, 10+15\}$

$= \min\{29, 25\} = \underline{25}$

$g(3,\{2,4\}) = \min\{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\}$

$= \min\{13+18, 12+13\}$

$= \min\{31, 25\} = \underline{25}$

$g(4,\{2,3\}) = \min\{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\}$

$= \min\{8+15, 9+18\}$

$= \min\{23, 27\} = \underline{23}$

* Finally from ① to remaining nodes

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

$= \min\{25+10, 25+15, 23+20\}$

$= \min\{35, 40, 43\}$

$= \boxed{35}$

- The minimum cost path is $\boxed{35}$

The optimal tour path is $1 \to 2 \to 4 \to 3 \to 1$ //

Time complexity:-

* In the dynamic programming for TSP, the no. of possible subsets can be at most $N \times 2^N$. Each subset can be solved in $O(N)$ time.
* Therefore, the time complexity of this algorithm is $O(N^2 \times 2^N)$

Space complexity:-

The space complexity is $O(2^N)$

Applications:-

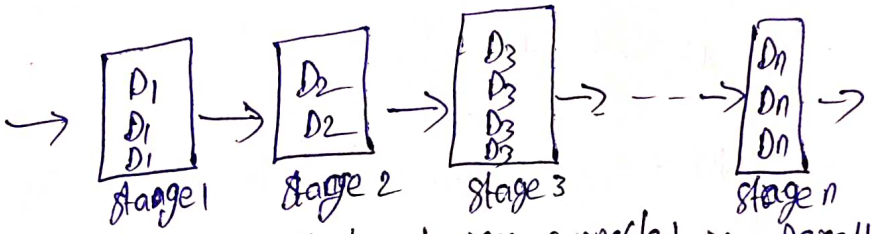
* DNA sequencing * Astronomy * optimal control problem
* manufacturing microchips.

Reliability Design

* Reliability means the ability of an system to consistently perform its intended function without degradation
* Reliability Design using dynamic programming is used to solve a problem with a multiplicative optimization function.
* The problem is to design a system which is composed of several devices connected in series.

devices connected in series

* Let γ be a reliability of device D then reliability of entire system is $\prod \gamma_i$
* To increase reliability of entire system we can use more than one device



multiple devices connected in parallel

* Let m_i be the no of devices of D_i are connected in parallel with reliability.
* $(1-\gamma_i)$ is the probability that one copy of the device will malfunction
* $(1-\gamma_i)^{m_i}$ is the probability of all devices malfunction same time.
* Hence the reliability of device (D_i) i.e probability of working good is

$$\phi_i(m_i) = 1-(1-\gamma_i)^{m_i}$$

* Our problem is to use device duplication to maximize reliability. This maximization is to be carried out under a cost constraint.
* A dynamic programming solution may be obtained for upper bound as

$$u_i = \left\lfloor \frac{c + c_i - \sum_{j=1}^{n} c_j}{c_i} \right\rfloor$$

where c_i is the cost of ith device, c is cost of entire system

* Finaly optimal solution $f_i(x)$ is

$$f_i(x) = \max_{1 \le m_i \le u_i} \{ \phi_i(m_i) f_{i-1}(c - c_i m_i) \}$$

Dominance Rule:-

* (f_1, x_1) dominates (f_2, x_2) if $f_1 > f_2$ and $x_1 < x_2$. Then dominated pair is removed from s^i. ~~i.e pair with less~~ which spend more cost for less reliability

Example:-

To design a three stage system with device type D_1, D_2 & D_3. The costs are 30, 15 & 20 respectively. The cost of the system is to be more than 105. The reliability of each device to be 0.9, 0.8 & 0.5 respectively.

Sol:- Given data.

$c_1 = 30$, $c_2 = 15$, $c_3 = 20$

$c = 105$

$r_1 = 0.9$, $r_2 = 0.8$, $r_3 = 0.5$

D_i	c_i	r_i	u_i
D_1	30	0.9	2
D_2	15	0.8	3
D_3	20	0.5	3

Step1:- compute upper bound i.e no.of copies of device using

$$u_i = \left\lfloor \frac{c + c_i - \sum_{j=1}^{n} c_j}{c_i} \right\rfloor$$

$$u_1 = \left\lfloor \frac{c + c_1 - (c_1+c_2+c_3)}{c_1} \right\rfloor = \left\lfloor \frac{105 + 30 - (30+15+20)}{30} \right\rfloor = 2$$

$$u_2 = \left\lfloor \frac{c + c_2 - (c_1+c_2+c_3)}{c_2} \right\rfloor = \left\lfloor \frac{105 + 15 - (30+15+20)}{15} \right\rfloor = 3$$

$$u_3 = \left\lfloor \frac{c + c_3 - (c_1+c_2+c_3)}{c_3} \right\rfloor = \left\lfloor \frac{105 + 20 - (30+15+20)}{20} \right\rfloor = 3$$

* Finaly $\Rightarrow$ 2 copies of D_1
  $\Rightarrow$ 3 copies of D_2
  $\Rightarrow$ 3 copies of D_3

Step2:- compute S^0 & S^1 $\quad S^0 = \{(r, x)\}$

* Initialy we will take $S^0 = \{1, 0\}$ ($r \rightarrow$ reliability, $x \rightarrow$ cost; reliability, cost)

* now compute S^1 where u_1 is 2 so it requries two copies S^1_1 & S^1_2

* S^1_1 calculated as $i=1$, $j=1$, $m_1 = 1$ $\quad r_1 = 0.9$

reliability of device $\phi(m_1) = 1 - (1-r_i)^{m_i} = 1 - (1-0.9)^1$

$= 1 - 0.1$

$= 0.9$

remove (0.95, 90) because of more cost always in increasing order

$S^2 = \{(0.72, 45), (0.864, 60), (0.892, 75), (0.98, 105)\}$

Step 4: S^3 is calculated from S_1^3, S_2^3 & S_3^3

* S_1^3 is calculated as $i=3$, $j=1$, $m_3=1$ $\gamma_3 = 0.5$, $c_3 = 20$

$$\phi(m_3) = 1-(1-\gamma_3)^{m_3} = (1-(1-0.5)^1 = 0.5$$

$S_1^3 = \{(0.72*0.5+45+20), (0.864*0.5, 60+20), (0.892*0.5, 75+20), (0.98*0.5, 105+20$

$= \{(0.36, 65), (0.432, 80), (0.446, 95), (0.49, 125)\}$

↓ remove the pair which exceeds 105

$= \{(0.36, 65), (0.432, 80), (0.446, 95)\}$

* S_2^3 is calculated as $i=3$, $j=2$, $m_3=2$

$$\phi(m_3) = (1-(1-\gamma_3))^{m_3} = (1-(1-0.5))^2 = 1-(0.5)^2 = 0.75$$

$S_2^3 = \{(0.72*0.75+45+2*20), (0.864*0.75, 60+2*20), (0.892*0.75, 75+2*20),$
$(0.98*0.75, 105+2*20)\}$ remove which exceeds cost 105

$= \{(0.54, 85), (0.648, 100)\}$

* S_3^3 is calculated as $i=3$, $j=3$, $m_3=3$

$$\phi(m_3) = (1-(1-\gamma_3)^{m_3} = 1-(1-0.5)^3 = 0.875$$

$S_3^3 = \{(0.72*0.875, 45+3*20), (0.864*0.875, 60+3*20), (0.892*0.875, 75+3*20),$
$(0.98*0.875, 105+3*20)\}$ remove which exceeds 105

$= \{(0.63, 105)\}$

* S^3 computed by merging S_1^3, S_2^3 & S_3^3

$S^3 = \{(0.36, 65), (0.432, 80), (0.446, 95), (0.54, 85), (0.648, 100), (0.63, 105$

$0.446 < 0.54$ && $95 > 85$ remove $(0.446, 95)$ pair

$0.648 - 100$
$0.63 - 105$ so remove $(0.63, 105)$

$= \{(0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100)\}$

* The best 0.648 reliability with cost of 100 which is obtained from S_2^3.
* $(0.648, 100)$ is again obtained from $(0.864, 60)$ which is present in S_2^2
* $(0.864, 60)$ is again obtained from $(0.9, 30)$ which is present in S^1

$S_1' = \{0.9*1, 0+30\}$ Here we multiply reliability and we add cost with S^0

$= \{0.9, 30\}$

* Now S_2' calculated as $i=1, J=2, m_2=2$

$\phi(m_2) = 1-(1-\gamma_1)^{m_2} = 1-(1-0.9)^2$

$= 1-(0.1)^2$

$= 0.99$

$S_2' = \{0.99, 2*30\} = \{0.99, 60\}$

S' can be obtained by merging S_1' and S_2'

$S' = \{(0.9, 30), (0.99, 60)\}$

Step 3:- compute s^2 it has three copies S_1^2, S_2^2, S_3^2

* S_1^2 calculated as $i=2, J=1, m_2=1$ $c_2=15, \gamma_2=0.8$

$\phi(m_2) = 1-(1-\gamma_2)^{m_2} = 1-(1-0.8)^1 = 0.8$

$S_1^2 = \{(0.9*0.8, 30+15), (0.99*0.8, 60+15)\}$ multiply with S' reliab and add cost

$= \{(0.72, 45), (0.792, 75)\}$

* S_2^2 calculated as $i=2, J=2, m_2=2$ $c_2=15, \gamma_2=0.8$

$\phi(m_2) = 1-(1-\gamma_2)^{m_2} = 1-(1-0.8)^2 = 0.96$

$S_2^2 = \{(0.9*0.96, 30+2*15), (0.99*0.96, 60+2*15)\}$

$= \{(0.864, 60), (0.95, 90)\}$

* S_3^2 calculated as $i=2, j=3, m_2=3$

$\phi(m_2) = 1-(1-\gamma_2)^{m_2} = 1-(1-0.8)^3 = 0.992$

$S_3^2 = \{(0.9*0.992, 30+3*15), (0.99*0.992, 60+3*15)\}$

$= \{(0.892, 75), (0.982, 105)\}$

Now S^2 can be obtained by merging S_1^2, S_2^2 & S_3^2

$S^2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.95, 90), (0.892, 75), (0.982, 105)\}$

By applying dominance rule blw $(0.792, 75)$ & $(0.864, 60)$

$0.792, 75$ using more cost so remove this pair

$S^2 = \{(0.72, 45), (0.864, 60), (0.95, 90), (0.892, 75), (0.982, 105)\}$

* Finally we require

D_1 - 1 copy i.e $S^1_{(1)}$

D_2 - 2 copies i.e $S^2_{(2)}$

D_3 - 2 copies i.e $S^3_{(2)}$

Backtracking

- Backtracking is an improvement of the bruteforce approach. It tries to search for a solution to a problem among all the available options
- Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again.

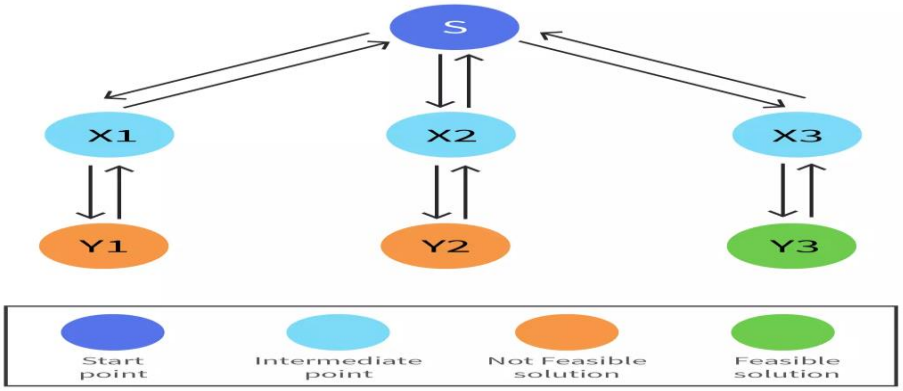
When to use a Backtracking algorithm?

- When we have multiple choices, then we make the decisions from the available choices. In the following cases, we need to use the backtracking algorithm:
  - A piece of sufficient information is not available to make the best choice, so we use the backtracking strategy to try out all the possible solutions.
  - Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In this case, we need to use the backtracking strategy.

How Backtracking Algorithms Works?

- Backtracking is the technique to solve programs recursively. In backtracking problems, you will have a number of options and you must choose one of these options.
- After you make your choice you will explore a new set of options, if you reach the feasible solution through those choices you will print the solution otherwise you will backtrack and return to explore the other paths and choices that can possibly lead to the solution.

For Instance: Consider the following space state tree. With the help of the below space state tree let's understand how the backtracking algorithm actually works.



- We start from S which is the starting point of our problem. We go to find a solution to Y1 from intermediate point X1 and then we find that Y1 is not a feasible solution to our problem therefore we backtrack and go back to X1 via Y1 then go back to S and then again try to find out the feasible solution by following another path S->X2->Y2 and again we will find that Y2 is not a feasible solution so we will again return to S by following path Y2->X2->S and then we will ultimately reach out to feasible solution Y3 by following the path S->X3->Y3. So we can summarise the backtracking algorithm as follows:
  1. We select one possible path and try to move forward towards finding a feasible solution.
  2. If we will reach a point from where we can't move towards the solution then we will backtrack.
  3. Again we try to find out the feasible solution by following other possible paths.

Types of Backtracking Problems

Before start solving the problem we must be able to recognize if it can be solved using a backtracking algorithm. There are the following types of problems that can be solved using backtracking:

1. **Decision Problem:** In this type of problem we always search for a feasible solution.
2. **Optimization Problem:** In this type of problem we always search for the best possible solution.
3. **Enumeration Problem:** In this type of problem we try to find all feasible solutions.

The terms related to the backtracking are:

- **Live node:** The nodes that can be further generated are known as live nodes.
- **E node:** The nodes whose children are being generated and become a success node.
- **Success node:** The node is said to be a success node if it provides a feasible solution.
- **Dead node:** The node which cannot be further generated and also does not provide a feasible solution is known as a dead node.

Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints, and these constraints are of two types:

- **Implicit constraint:** It is a rule in which how each element in a tuple is related.

Common examples of explicit constraints are

$x_i \geq 0$ or $S_i = \{\text{all nonnegative real numbers}\}$

$x_i = 0 \text{ or } 1$ or $S_i = \{0, 1\}$

$l_i \leq x_i \leq u_i$ or $S_i = \{a: l_i \leq a \leq u_i\}$

- **Explicit constraint:** The rules that restrict each element to be chosen from the given set.

General Method Algorithm:

- To apply backtracking method, the desired solution must be expressible as an n-tuple ($x_1 \ldots x_n$) where xi is chosen from some finite set Si.
- The problem is to find a vector, which maximizes or minimizes a criterion function $P(x_1 \ldots . x_n)$.
- The major advantage of this method is, once we know that a partial vector ($x_1, \ldots x_i$) will not lead to an optimal solution that ($m_i+1 \ldots \ldots \ldots .m_n$) possible test vectors may be ignored entirely.

```
procedure BACKTRACK(n)
    integer k, n; local X(1:n)
    k ← 1
    while k > 0 do
      if there remains an untried X(k) such that
        X(k) ∈ T(X(1), ..., X(k − 1)) and B_k(X(1), ..., X(k)) = true
      then if (X(1), ..., X(k)) is a path to an answer node
           then print (X(1), ..., X(k))  endif
           k ← k + 1   //consider the next set//
      else k ← k − 1   //backtrack to previous set//
      endif
    repeat
end BACKTRACK
```

Algorithm 7.1 General backtracking method

Applications of Backtracking

- N-queen problem
- Sum of subset problem
- Graph coloring
- Hamiliton cycle

Difference between the Backtracking and Recursion

- Recursion is a technique that calls the same function again and again until you reach the base case.
- Backtracking is an algorithm that finds all the possible solutions and selects the desired solution from the given set of solutions.

N-Queen Problem

What is N Queen Problem?

- N Queen problem is the classical Example of backtracking. N-Queen problem is defined as, “given N x N chess board, arrange N queens in such a way that no two queens attack each other by being in same row, column or diagonal”.
  - For N = 1, this is trivial case. For N = 2 and N = 3, solution is not possible. So we start with N = 4 and we will generalize it for N queens.

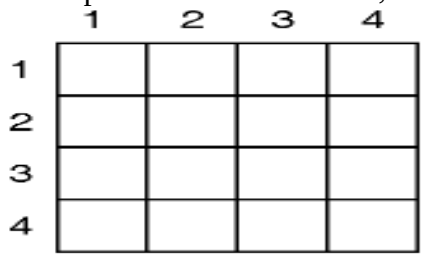
Algorithm:

START
1. begin from the leftmost column
2. if all the queens are placed,
   return true
3. check for all rows in the current column
   a) if queen placed safely, mark row and column; and recursively check if we approach in the current configuration, do we obtain a solution or not
   b) if placing yields a solution, return true
   c) if placing does not yield a solution, unmark and try other rows
4. if all rows tried and solution not obtained, return false and backtrack

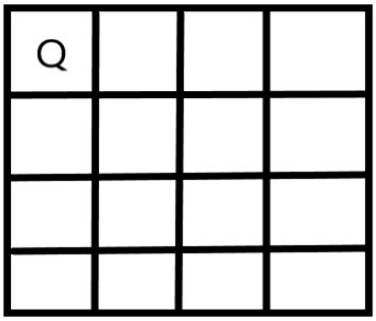
END

4-Queen Problem

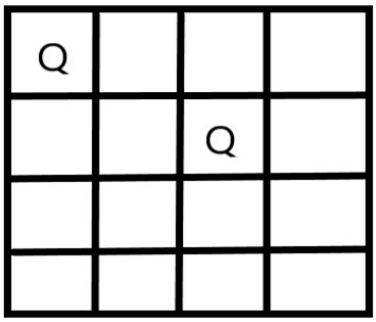
Problem : Given 4 x 4 chessboard, arrange four queens in a way, such that no two queens attack each other. That is, no two queens are placed in the same row, column, or diagonal.



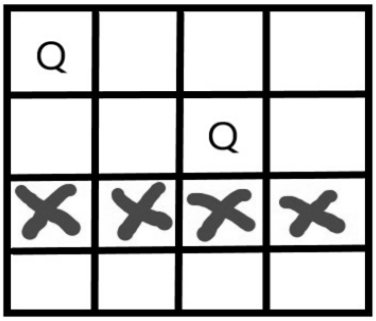
4 x 4 Chessboard



let us first consider an empty chessboard and start by placing the first queen on cell chessboard[0][0]

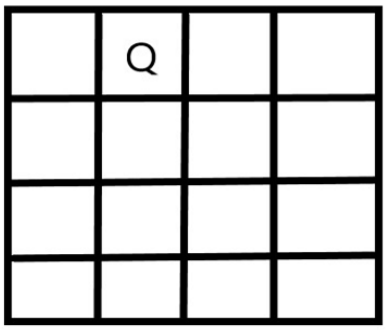


chessboard[1][2] is the only possible position where the second queen can be placed

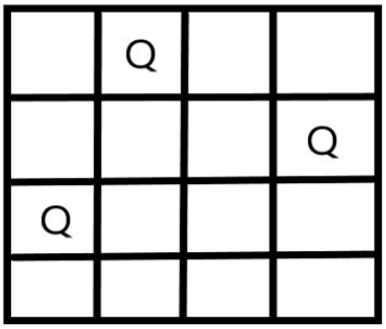


no queen can be placed further as queen 1 is in column 1, queen 2 is diagonally opposite to columns 2 and 3; and column 3 has queen 2 in it. Since number of queens is not 4, this is an infeasible solution and will not be printed

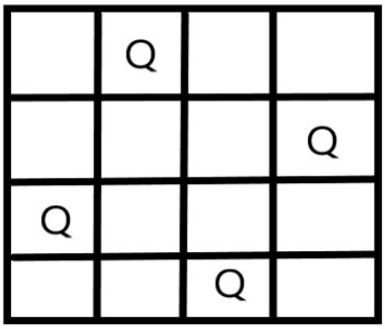
- As the above combination was not possible, we will go back and go for the next iteration. This means we will change the position of the second queen.



the next iteration of our algorithm will begin with the second column and start placing the queens again



queens 2 and 3 are easily placed onto the chessboard without creating the possibility of attacking one another.



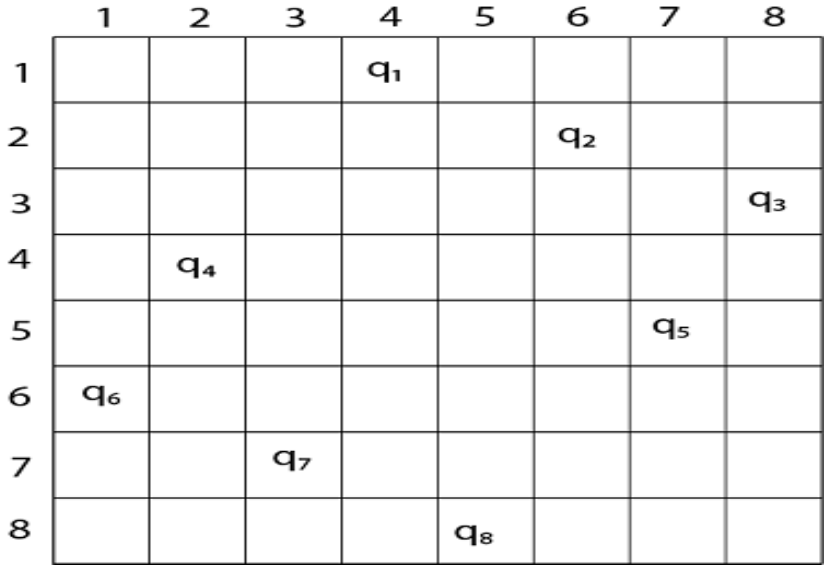
the 4th queen has also been placed accordingly. Since the number of queens = 4, this solution will be printed.

In this, we found a solution.
Now let's take a look at the backtracking algorithm and see how it works:

- The idea is to place the queens one after the other in columns, and check if previously placed queens cannot attack the current queen we're about to place.
- If we find such a row, we return true and put the row and column as part of the solution matrix. If such a column does not exist, we return false and backtrack*

8-Queen Problem

Problem : Given an 8 x 8 chessboard, arrange 8 queens in a way such that no two queens attack each other.



Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).

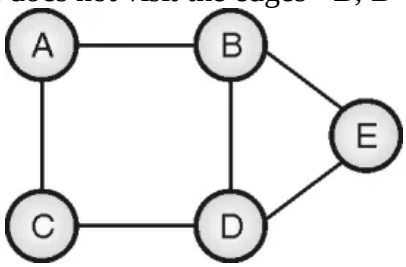
Time Complexity Analysis

1. the isPossible method takes O(n) time
2. for each invocation of loop in nQueenHelper, it runs for O(n) time
3. the isPossible condition is present in the loop and also calls nQueenHelper which is recursive

- Adding this up, the recurrence relation is: T(n) = O(n^2) + n * T(n-1)
- Solving the above recurrence by iteration or recursion tree,
    - The time complexity of the NQueen problem is = O(N!)

Hamiltonian cycle using the backtracking approach

- The Hamiltonian cycle is the cycle in the graph which visits all the vertices in graph exactly once and terminates at the starting node. It may not include all the edges
- The Hamiltonian cycle problem is the problem of finding a Hamiltonian cycle in a graph if there exists any such cycle.
- The input to the problem is an undirected, connected graph. For the graph shown in Figure (a), a path A – B – E – D – C – A forms a Hamiltonian cycle. It visits all the vertices exactly once, but does not visit the edges <B, D>.

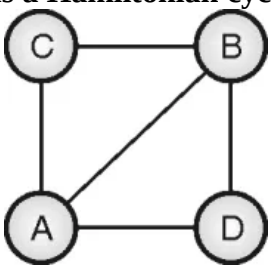


- The problem can be solved using a backtracking approach. Follow the below steps to solve the problem.

Algorithm:

- Create an empty path array.
- Add the vertex 0 to the array.
- Start adding vertex 1 and other connected nodes and check if the current vertex can be included in the array or not.
- This can be done by using a visiting array and check if the vertex has already been visited or is adjacent to previously added vertex.
- If any such vertex is found, add it to the array and backtrack from that node.
- Try every possible combinations and if a path returns false, ignore the vertex and start iterating from the next vertex till all the nodes has been visited.

Example: Show that given graph has a Hamiltonian cycle



Solution:

- We can start with any random vertex. Let us start with vertex A. Neighbors of A are {B, C, D}. The inclusion of B does not lead to a complete solution. So explore it as shown in Figure (c).
- Adjacent vertices to B are {C, D}. The inclusion of C does not lead to a complete solution. All adjacent vertices of C are already members of the path formed so far. So it leads to a dead end.
- Backtrack and go to B and explore its next neighbor i.e. D.
- The inclusion of D does not lead to a complete solution, and all adjacent vertices of D are already a member of the path formed so far. So it leads to a dead end.

Figur
e: (b)
initial
tree

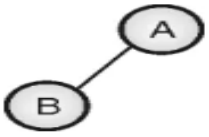


Figure (c):
Added node
B

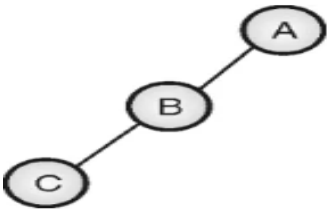


Figure (d): Node C
added, dead-end

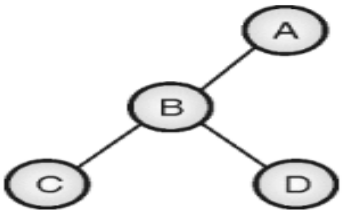
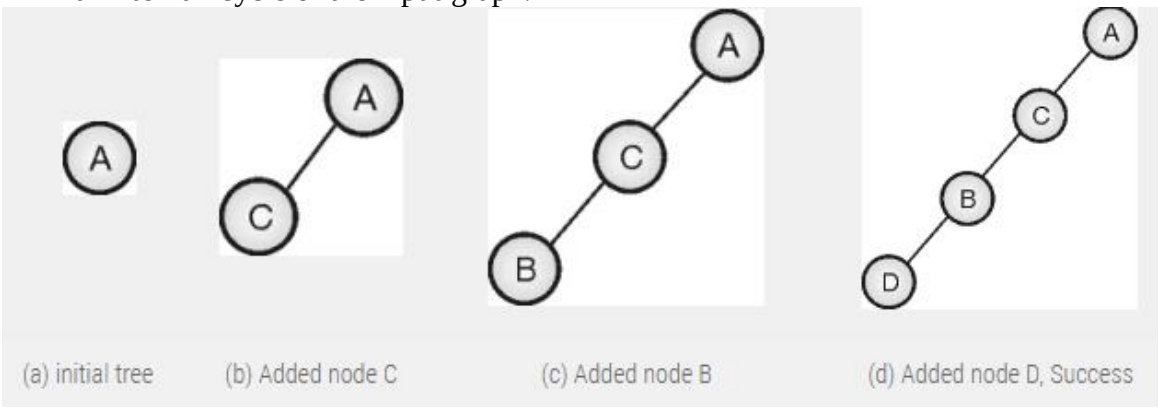


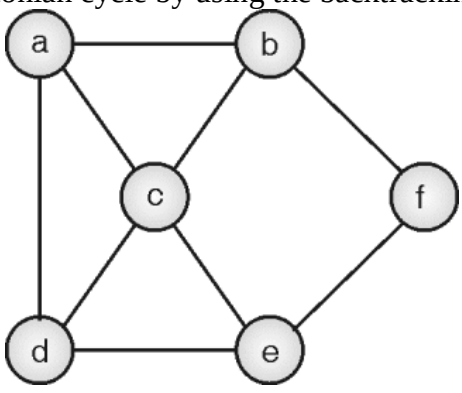
Figure (e): Node D
added, dead-end

- Backtrack and go to B. Now B does not have any more neighbors, so backtrack and go to A. Explore the next neighbor of A i.e. C. By repeating the same procedure, we get the state space trees as shown below. And path A – C – B – D – A is detected as the Hamiltonian cycle of the input graph.

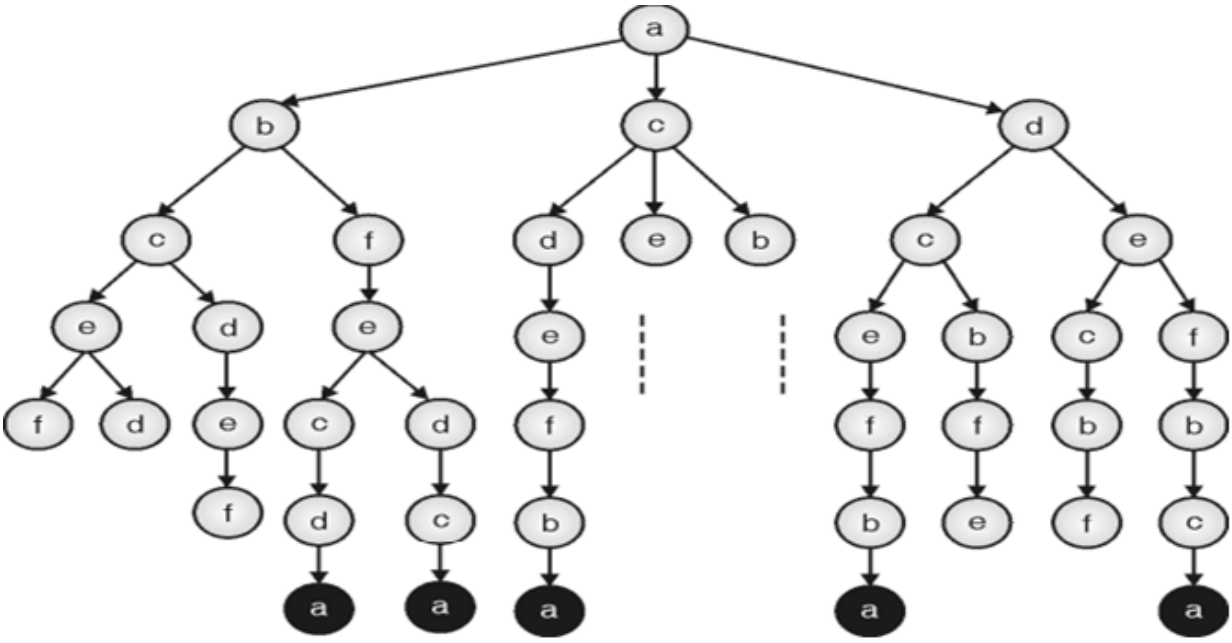


Another Hamiltonian cycle with A as a start vertex is A – D – B – C – A.

Example 2: Find the Hamiltonian cycle by using the backtracking approach for a given graph.



Solution:



Time Complexity :

- **O(N!),** where N is number of vertices.

Auxiliary Space :

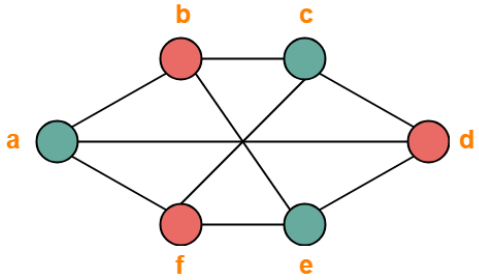
- **O(1),** since no extra space used.

*****END******

Graph coloring using Backtracking

Graph coloring refers to the problem of coloring vertices of a graph in such a way that no two adjacent vertices have the same color. This is also called the vertex coloring problem.

- If coloring is done using at most k colors, it is called k-coloring.
- The smallest number of colors required for coloring graph is called its chromatic number.
- The chromatic number is denoted by X(G).



- Minimum number of colors used to color the given graph are 2.
- Therefore, Chromatic Number of the given graph = 2.

Applications of Graph Coloring Problem

- Design a timetable.
- Sudoku
- Register allocation in the compiler
- Map coloring

- Mobile radio frequency assignment:

Approach: Backtracking

The backtracking approach to solving the graph coloring problem can be to assign the colors one after the other to the various vertices. The coloring will start with the first index only but before assigning any color, we would first check if it satisfies the constraint or not (i.e. no two adjacent vertices have the same color). If the current color assignment does not violate the condition then add it into the solution else, backtrack by returning false.

Pseudo code can be:

1. Define a recursive function that takes the current index (i), several vertices, and the color array for output.
2. Check if the current color is safe i.e. no two adjacent vertices have the same color. Then print the current color configuration and break the loop.
3. Assign a color to the vertex. The range of assigned colors is from 1 to m.
4. Now, for every color assigned, call the recursive function.
5. If the recursive call returns true then the coloring is possible. So, break the loop and return true. Else, backtrack.

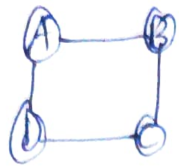
Time Complexity:

- O(mV). Since backtracking is also a kind of brute force approach, there would be total O(mV) possible color combinations. It is to be noted that the upperbound time complexity remains the same but the average time taken will be less due to the refined approach.

Space Complexity:

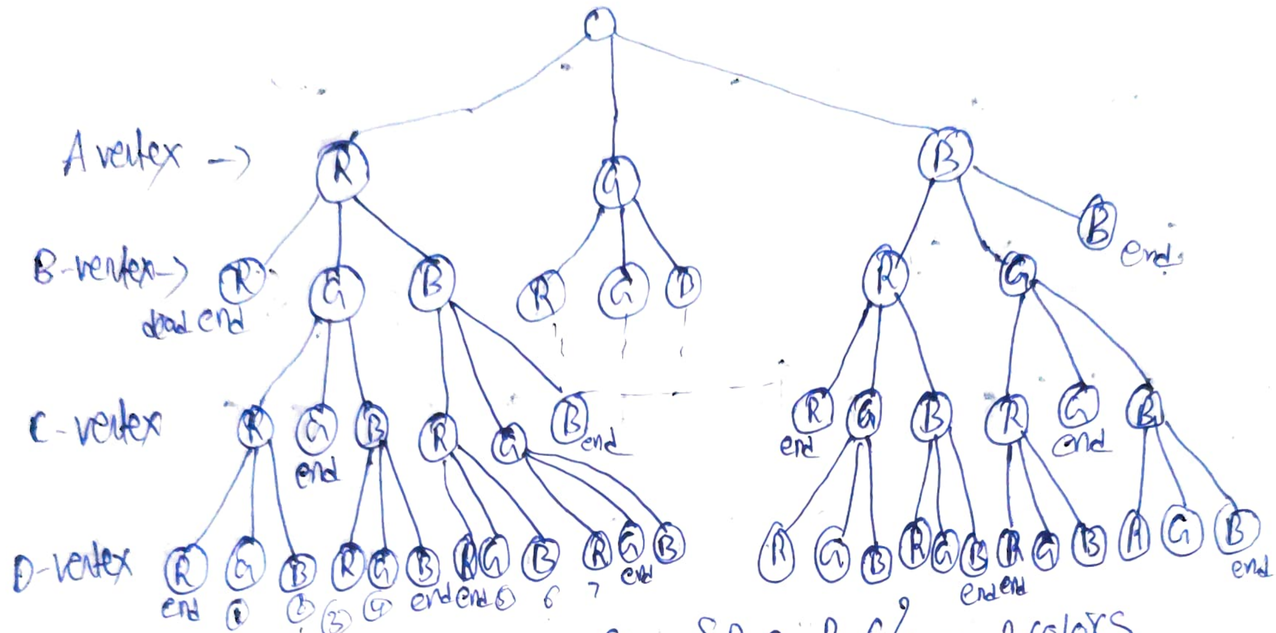
- O(V) for storing the output array in O(V) space

Example:- Graph colouring using Backtracking



no. of vertices = 4 = {A, B, C, D}

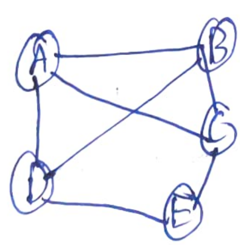
no of colors = 3 = {R, G, B}



① - {R, G, R, G} — 2 colors

② - {R, G, R, B} — 3 colors

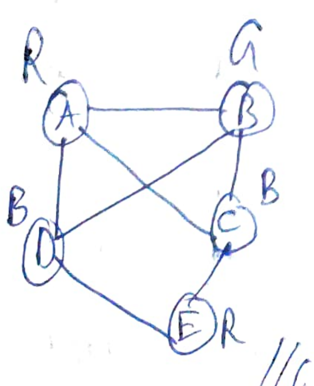
Example 2:-

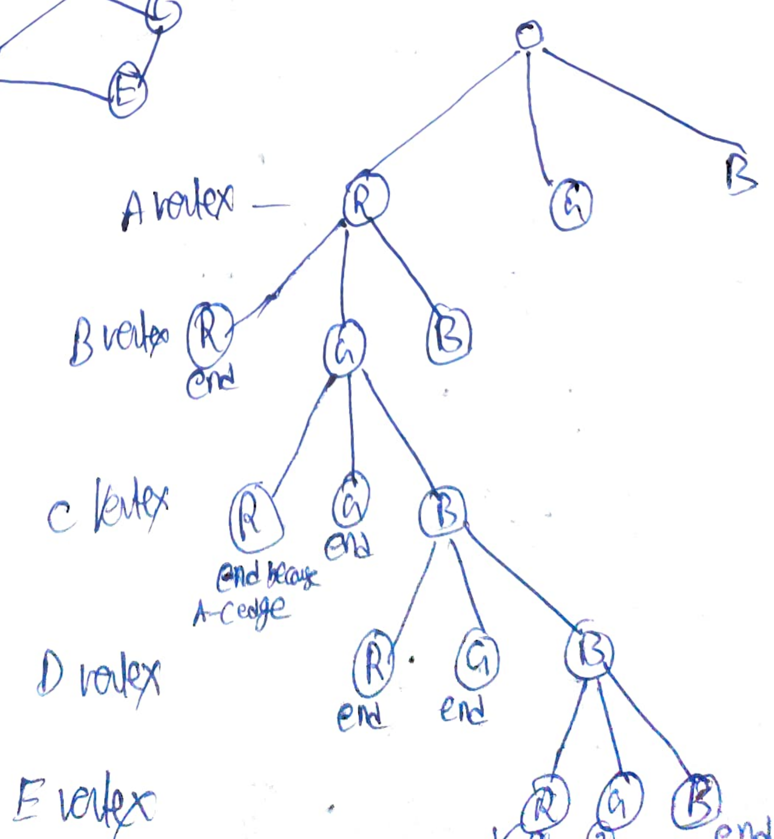


A B C D E

① = {R, G, B, B, R}

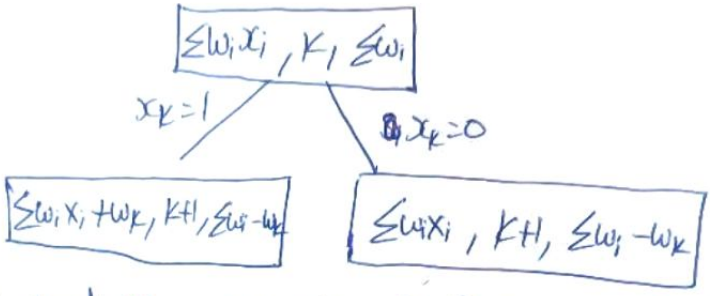
② = {R, G, B, B, G}





Sum of subsets problem using Backtracking

* Suppose we are given 'n' distinct positive numbers and we desire to find all combinations of these numbers whose sum is 'M'. This is called as the "sum of subsets problem".
* Sum of subset Problem problem using back tracking approach will take $O(2^n)$ time complexity which is faster than the recursive approach
* The state space tree can be drawn as follows



* The Bounding functions we use $B_k(x_1 x_2 \cdots x_k)$ = true if and only if

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq M \quad \text{and} \quad \sum_{i=1}^{k} w_i x_i + w_{(k+1)} \leq M$$

Algorithm :-

```
Algorithm sumofsubsets(S, k, r)
// S = Σ_{i=1}^{k-1} w[i]*x[i]  and  r = Σ_{i=k}^{n} w_i
    x[k] := 1
    if (S+w[k] == M) then write x[1:k]
    else if S+w[k] ≤ M then
        sumofsubsets(S+w[k], k+1, r-w[k])
    endif
// Generate right child
    if (S+r-w[k] ≥ M) and (S+w[k+1] ≤ M) then
        x[k] := 0;
        sumofsubsets(S, k+1, r-w[k])
    endif
end sumofsubsets
```

steps of algorithm:-

1. start with an empty set
2. Include the next element from list to set
3. If the numbers in the set sum up to given target sum, it is a solution set
4. If the set doesnot sum upto target sum (or) reached the end then backtrack the set until we find a solution set
5. if we get a feasible solution goto step 2
6. if we have traversed all the elements and if no backtracking is possible then stop

Example: let $n=4$, $M=31$ and $w[1:4] = \{7, 11, 13, 24\}$ Draw state space tree

$\sum_{i=1}^{n} w_i = 7+11+13+24 = 55$

Example 2:

Set = {10, 7, 5, 18, 12, 20, 15}

Sum = 30

Output = {{10, 5, 15}, {10, 20}, {7, 5, 18}, {18, 12}}

Complexity Analysis of sum of subsets

Time complexity:-

In the state space tree at level i, the tree has 2^i nodes.

So, given n items, the total no of nodes the tree would be

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

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

Space complexity:-

Space complexity $= O(1)$

NP-Hard and NP-Complete Problem

Deterministic and non-deterministic algorithms

- **Deterministic:** The algorithm in which every operation is uniquely defined is called deterministic algorithm.
- **Non-Deterministic:** The algorithm in which the operations are not uniquely defined but are limited to specific set of possibilities for every operation, such an algorithm is called non-deterministic algorithm

The Class P

- P: the class of problems that have polynomial-time deterministic algorithms.
  - That is, they are solvable in O(p(n)), where p(n) is a polynomial on n
  - A deterministic algorithm is (essentially) one that always computes the correct answer
- Sample Problems in P
  - Fractional Knapsack
  - MST
  - Sorting

The class NP

- NP: the class of decision problems that are solvable in polynomial time on a nondeterministic machine (or with a nondeterministic algorithm)
  - (A determinstic computer is what we know)
  - A nondeterministic computer is one that can "guess" the right answer or solution
- Think of a nondeterministic computer as a parallel machine that can freely spawn an infinite number of processes
- Thus NP can also be thought of as the class of problems "whose solutions can be verified in polynomial time"
- Note that NP stands for "Nondeterministic Polynomial-time"
- Sample Problems in NP
  - Fractional Knapsack
  - MST
  - Traveling Salesman
  - Graph Coloring

NP-hard

- A problem is NP-hard if all problems in NP are polynomial time reducible to it
  Ex:- Hamiltonian Cycle
- Every problem in NP is reducible to Hamiltonian Cycle in polynomial time.
  Ex:- Travelling salesman problem is reducible to Hamiltonian Cycle.

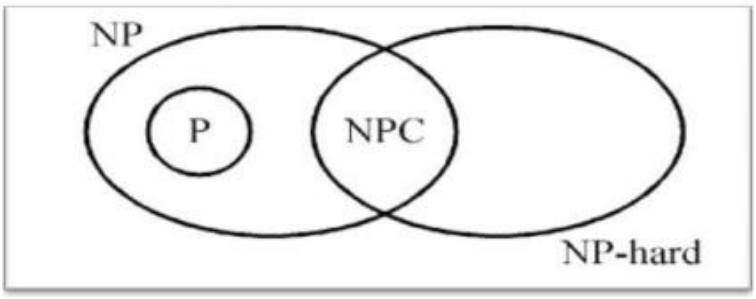
- In symbols,

$$P_i \text{ is NP-hard if, for every } P_j \in \mathbf{NP}, P_j \xrightarrow{\text{poly}} P_i.$$

- Note that this doesn't require P_i to be in NP.

NP-complete

- A problem is NP-complete if the problem is both NP-hard and NP.
- If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time.
- All NP-complete problems are NP-hard, but all NP- hard problems are not NP-complete.
- The class of NP-hard problems is very rich in the sense that it contains many problems from a wide variety of disciplines



The Satisfiability Problem (SAT)

- An expression *E* is *satisfiable* if there exists a truth assignment to the variables in *E* that makes *E* true.
- The *satisfiability problem* (SAT) is to determine whether a given boolean expression is satisfiable.
- We can view SAT as the language { *E* | *E* is the encoding of a satisfiable boolean expression }.
- In 1971 using a slightly different definition of NP-completeness, Steven Cook showed that SAT is NP-complete. At roughly the same time, Leonid Levin independently showed that SAT was NP-complete.
- SAT can be used to prove that other problems are NP complete by showing that the other problem is in NP and that SAT can be reduced to the other problem in polynomial time.

Difference Between NP-Hard and NP-Complete Problem

Parameters	**NP-Hard Problem**	**NP-Complete Problem**
Meaning and Definition	One can only solve an NP-Hard Problem X only if an NP-Complete Problem Y exists. It then becomes reducible to problem X in a polynomial time.	Any given problem X acts as NP-Complete when there exists an NP problem Y- so that the problem Y gets reducible to the problem X in a polynomial line.
Presence in NP	The NP-Hard Problem does not have to exist in the NP for anyone to solve it.	For solving an NP-Complete Problem, the given problem must exist in both NP-Hard and NP Problems.
Decision Problem	This type of problem need not be a Decision problem.	This type of problem is always a Decision problem (exclusively).
Example	Circuit-satisfactory, Vertex cover, Halting problems, etc., are a few examples of NP-Hard Problems.	A few examples of NP-Complete Problems are the determination of the Hamiltonian cycle in a graph, the determination of the satisfaction level of a Boolean formula, etc.