Unit - 4

Dynamic Programming

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

Let's understand this approach through an example.

Consider an example of the Fibonacci series. The following series is the Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,…

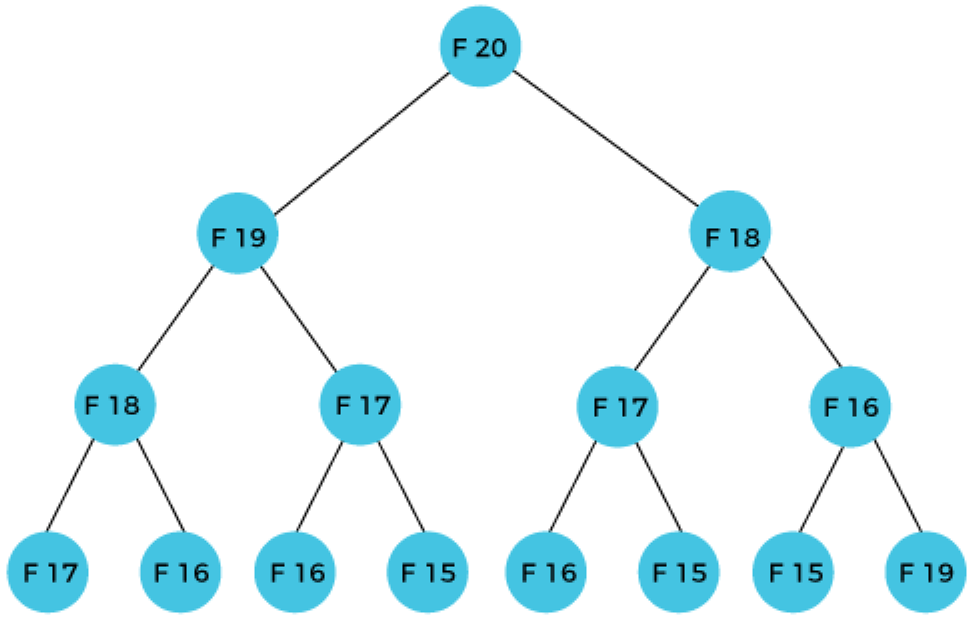
The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

F(n) = F(n-1) + F(n-2),

With the base values F(0) = 0, and F(1) = 1. To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum **f(0)** and **f(1),** which is equal to 1.

How can we calculate F(20)?

The F(20) term will be calculated using the nth formula of the Fibonacci series. The below figure shows that how F(20) is calculated.



As we can observe in the above figure that F(20) is calculated as the sum of F(19) and F(18). In the dynamic programming approach, we try to divide the problem into the similar subproblems. We are following this approach in the above case where F(20) into the similar subproblems, i.e., F(19) and F(18). If we recap the definition of dynamic programming that it says the similar subproblem should not be computed more than once. Still, in the above case, the subproblem is calculated twice. In the above example, F(18) is calculated two times; similarly, F(17) is also calculated twice. However, this technique is quite useful as it solves the similar subproblems, but we need to be cautious while storing the results because we are not particular about storing the result that we have computed once, then it can lead to a wastage of resources.

In the above example, if we calculate the F(18) in the right subtree, then it leads to the tremendous usage of resources and decreases the overall performance.

The solution to the above problem is to save the computed results in an array. First, we calculate F(16) and F(17) and save their values in an array. The F(18) is calculated by summing the values of F(17) and F(16), which are already saved in an array. The computed value of F(18) is saved in an array. The value of F(19) is calculated using the sum of F(18), and F(17), and their values are already saved in an array. The computed value of F(19) is stored in an array. The value of F(20) can be calculated by adding the values of F(19) and F(18), and the values of both F(19) and F(18) are stored in an array. The final computed value of F(20) is stored in an array.

How does the dynamic programming approach work?

The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (memoization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.

The above five steps are the basic steps for dynamic programming. The dynamic programming is applicable that are having properties such as:

Those problems that are having overlapping subproblems and optimal substructures. Here, optimal substructure means that the solution of optimization problems can be obtained by simply combining the optimal solution of all the subproblems.

In the case of dynamic programming, the space complexity would be increased as we are storing the intermediate results, but the time complexity would be decreased.

Approaches of dynamic programming

There are two approaches to dynamic programming:

- Top-down approach
- Bottom-up approach

Top-down approach

The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

Advantages

- It is very easy to understand and implement.
- It solves the subproblems only when it is required.

- It is easy to debug.

Disadvantages

It uses the recursion technique that occupies more memory in the call stack. Sometimes when the recursion is too deep, the stack overflow condition will occur.

It occupies more memory that degrades the overall performance.

Let's understand dynamic programming through an example.

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

In the above code, we have used the recursive approach to find out the Fibonacci series. When the value of 'n' increases, the function calls will also increase, and computations will also increase. In this case, the time complexity increases exponentially, and it becomes 2^n.

One solution to this problem is to use the dynamic programming approach. Rather than generating the recursive tree again and again, we can reuse the previously calculated value. If we use the dynamic programming approach, then the time complexity would be O(n).

When we apply the dynamic programming approach in the implementation of the Fibonacci series, then the code would look like:

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

```
  error;
 if(n==0)
 return 0;
 if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
memo[n] = sum;
}
```

In the above code, we have used the memorization technique in which we store the results in an array to reuse the values. This is also known as a top-down approach in which we move from the top and break the problem into sub-problems.

Bottom-Up approach

The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

There are two ways of applying dynamic programming:

- **Top-Down**
- **Bottom-Up**

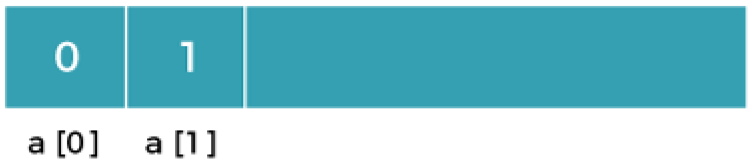
The bottom-up is the approach used to avoid the recursion, thus saving the memory space. The bottom-up is an algorithm that starts from the beginning, whereas the recursive algorithm starts from the end and works backward. In the bottom-up approach, we start from the base case to find the answer for the end. As we know, the base cases in the Fibonacci series are 0 and 1. Since the bottom approach starts from the base cases, so we will start from 0 and 1.

Key points

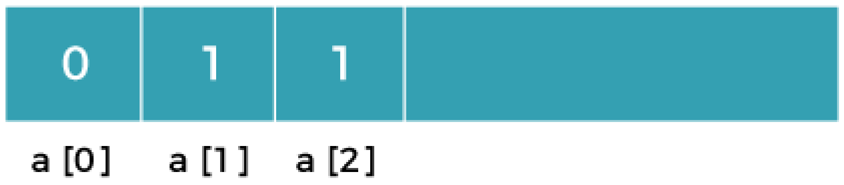
- We solve all the smaller sub-problems that will be needed to solve the larger sub-problems then move to the larger problems using smaller sub-problems.
- We use for loop to iterate over the sub-problems.
- The bottom-up approach is also known as the tabulation or table filling method.

Let's understand through an example.

Suppose we have an array that has 0 and 1 values at a[0] and a[1] positions, respectively shown as below:

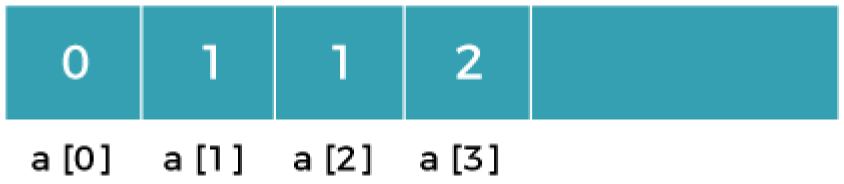


Since the bottom-up approach starts from the lower values, so the values at a[0] and a[1] are added to find the value of a[2] shown as below:

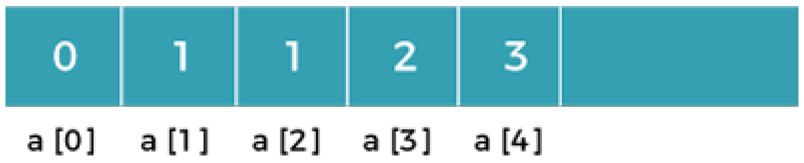


The value of a[3] will be calculated by adding a[1] and a[2], and it becomes 2 shown as below:

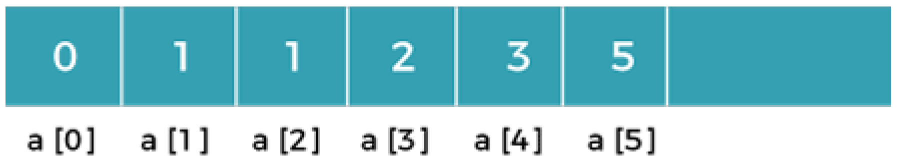
Advertisement



The value of a[4] will be calculated by adding a[2] and a[3], and it becomes 3 shown as below:



The value of a[5] will be calculated by adding the values of a[4] and a[3], and it becomes 5 shown as below:



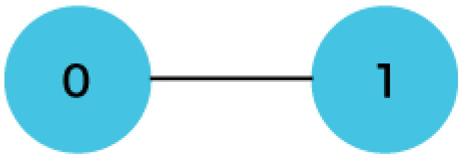
The code for implementing the Fibonacci series using the bottom-up approach is given below:

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

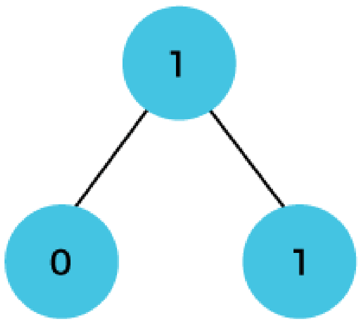
In the above code, base cases are 0 and 1 and then we have used for loop to find other values of Fibonacci series.

Let's understand through the diagrammatic representation.

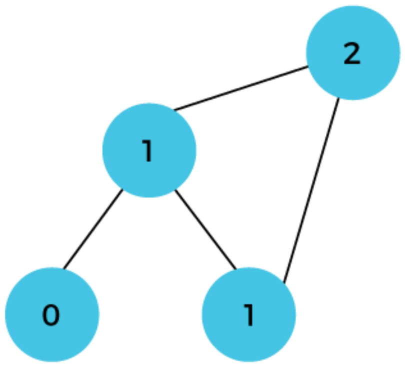
Initially, the first two values, i.e., 0 and 1 can be represented as:



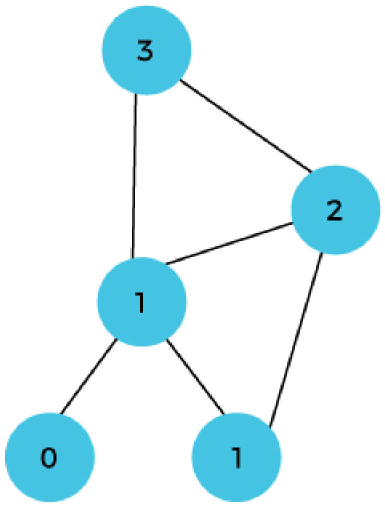
When i=2 then the values 0 and 1 are added shown as below:



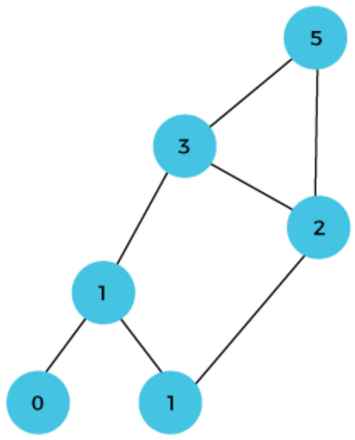
When i=3 then the values 1and 1 are added shown as below:



When i=4 then the values 2 and 1 are added shown as below:



When i=5, then the values 3 and 2 are added shown as below:



In the above case, we are starting from the bottom and reaching to the top.

All-Pairs Shortest Path (APSP) problem (Floyd-Warshall)

The **All-Pairs Shortest Path (APSP)** problem in **Design and Analysis of Algorithms (DAA)** is about finding the shortest paths between every pair of vertices in a weighted graph.

One of the most common algorithms used to solve the APSP problem is **Floyd-Warshall**.

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is an efficient method for APSP, especially when the graph has both positive and negative edge weights but no negative cycles.

The algorithm has a time complexity of O(V3), where V is the number of vertices.

Algorithm Steps:

1. Initialize a distance matrix dist[][] where dist[i][j] is set to the edge weight between nodes i and j. If there is no edge, set it to infinity (∞), except for dist[i][i] which should be 0.
2. Use each vertex as an intermediate point and update the matrix if a shorter path is found via that intermediate vertex.
3. Repeat for all vertices, and the final dist[][] will contain the shortest distances between every pair of vertices.

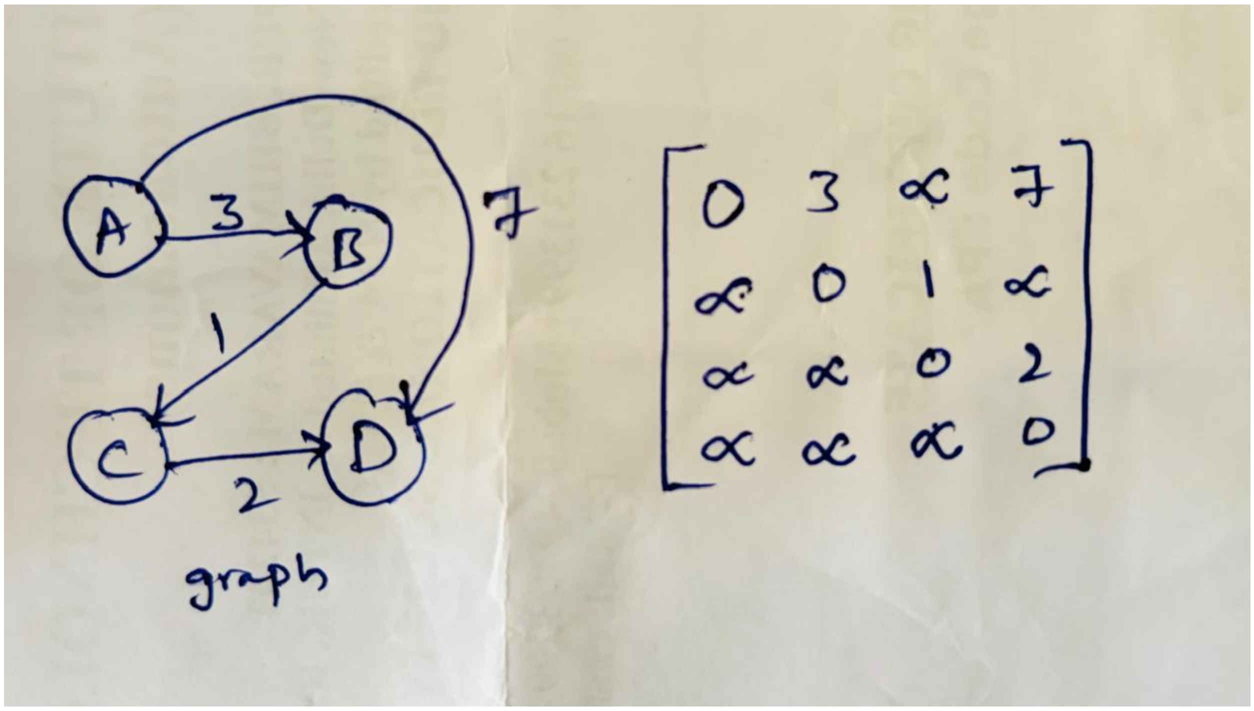
Example

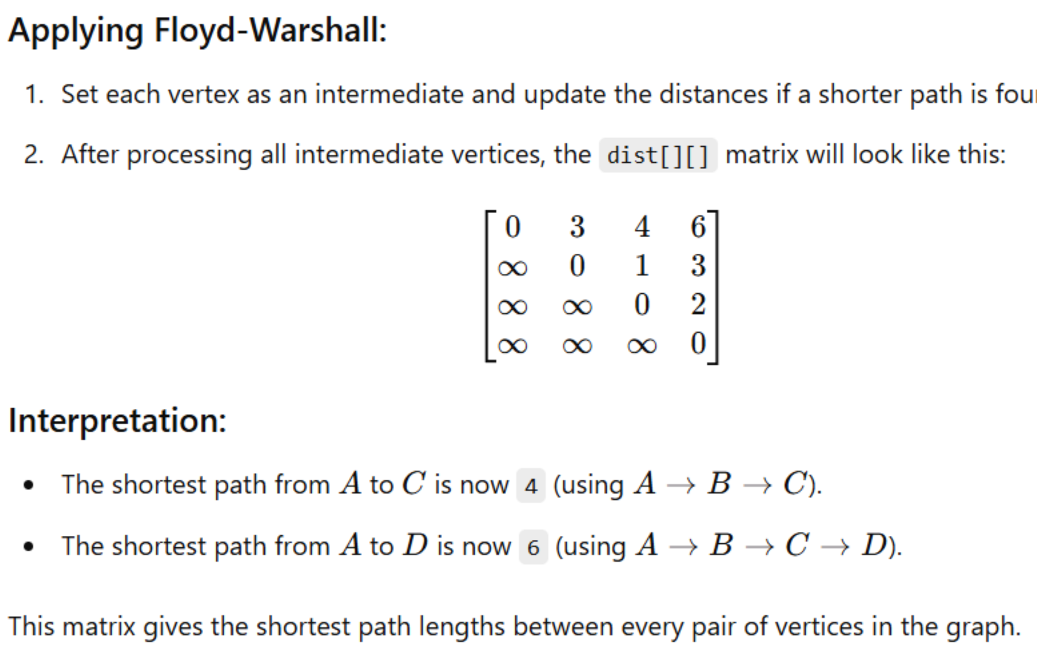
Consider a graph with 4 vertices and the following weighted edges:

$$\begin{array}{ll} A \to B & \text{weight} = 3 \\ A \to D & \text{weight} = 7 \\ B \to C & \text{weight} = 1 \\ C \to D & \text{weight} = 2 \end{array}$$

Representing this graph as a matrix `dist[][]` with `∞` indicating no direct edge:

$$\begin{bmatrix} 0 & 3 & \infty & 7 \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & 0 \end{bmatrix}$$





Applying Floyd-Warshall:

1. Set each vertex as an intermediate and update the distances if a shorter path is found.
2. After processing all intermediate vertices, the `dist[][]` matrix will look like this:

$$\begin{bmatrix} 0 & 3 & 4 & 6 \\ \infty & 0 & 1 & 3 \\ \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

Interpretation:

- The shortest path from A to C is now `4` (using $A \to B \to C$).
- The shortest path from A to D is now `6` (using $A \to B \to C \to D$).

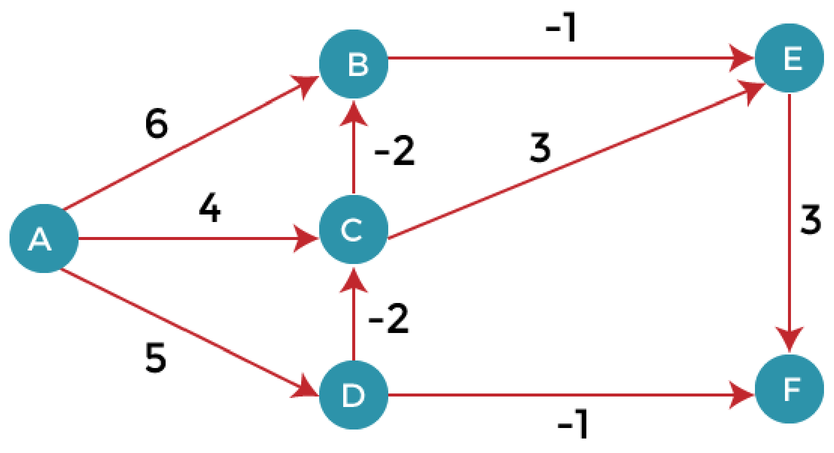
This matrix gives the shortest path lengths between every pair of vertices in the graph.

Single source shortest paths (Bellman Ford Algorithm)

Bellman ford algorithm is a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph. There are various other algorithms used to find the shortest path like Dijkstra algorithm, etc. If the weighted graph contains the negative weight values, then the Dijkstra algorithm does not confirm whether it produces the correct answer or not. In contrast to Dijkstra algorithm, bellman ford algorithm guarantees the correct answer even if the weighted graph contains the negative weight values.

Rule of this algorithm

1. We will go on relaxing all the edges (n - 1) times where,
2. n = number of vertices
   Consider the below graph:



As we can observe in the above graph that some of the weights are negative. The above graph contains 6 vertices so we will go on relaxing till the 5 vertices. Here, we will relax all the edges 5 times. The loop will iterate 5 times to get the correct answer. If the loop is iterated more than 5 times then also the answer will be the same, i.e., there would be no change in the distance between the vertices.

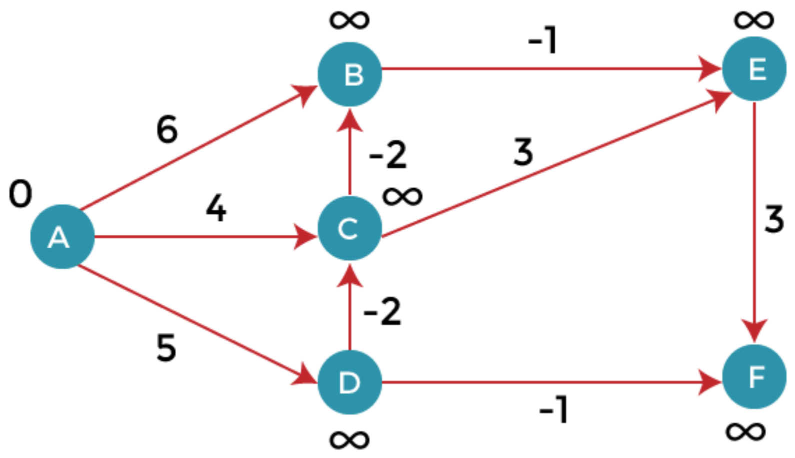
Relaxing means:

1. If (d(u) + c(u , v) < d(v))
2. d(v) = d(u) + c(u , v)

To find the shortest path of the above graph, the first step is note down all the edges which are given below:

(A, B), (A, C), (A, D), (B, E), (C, E), (D, C), (D, F), (E, F), (C, B)

Let's consider the source vertex as 'A'; therefore, the distance value at vertex A is 0 and the distance value at all the other vertices as infinity shown as below:



Since the graph has six vertices so it will have five iterations.

First iteration

Consider the edge (A, B). Denote vertex 'A' as 'u' and vertex 'B' as 'v'. Now use the relaxing formula:

$d(u) = 0$

$d(v) = \infty$

$c(u , v) = 6$

Since $(0 + 6)$ is less than ∞, so update

$d(v) = d(u) + c(u , v)$
$d(v) = 0 + 6 = 6$

Therefore, the distance of vertex B is 6.

Consider the edge (A, C). Denote vertex 'A' as 'u' and vertex 'C' as 'v'. Now use the relaxing formula:

$d(u) = 0$

$d(v) = \infty$

$c(u , v) = 4$

Since $(0 + 4)$ is less than ∞, so update

$d(v) = d(u) + c(u , v)$
$d(v) = 0 + 4 = 4$

Therefore, the distance of vertex C is 4.

Consider the edge (A, D). Denote vertex 'A' as 'u' and vertex 'D' as 'v'. Now use the relaxing formula:

$d(u) = 0$

$d(v) = \infty$

$c(u , v) = 5$

Since $(0 + 5)$ is less than ∞, so update

d(v) = d(u) + c(u , v)
d(v) = 0 + 5 = 5

Therefore, the distance of vertex D is 5.

Consider the edge (B, E). Denote vertex 'B' as 'u' and vertex 'E' as 'v'. Now use the relaxing formula:

d(u) = 6

d(v) = ∞

c(u , v) = -1

Since (6 - 1) is less than ∞, so update

d(v) = d(u) + c(u , v)
d(v) = 6 - 1= 5

Therefore, the distance of vertex E is 5.

Consider the edge (C, E). Denote vertex 'C' as 'u' and vertex 'E' as 'v'. Now use the relaxing formula:

d(u) = 4

d(v) = 5

c(u , v) = 3

Since (4 + 3) is greater than 5, so there will be no updation. The value at vertex E is 5.

Consider the edge (D, C). Denote vertex 'D' as 'u' and vertex 'C' as 'v'. Now use the relaxing formula:

d(u) = 5

d(v) = 4

c(u , v) = -2

Since (5 -2) is less than 4, so update

d(v) = d(u) + c(u , v)
d(v) = 5 - 2 = 3

Therefore, the distance of vertex C is 3.

Consider the edge (D, F). Denote vertex 'D' as 'u' and vertex 'F' as 'v'. Now use the relaxing formula:

d(u) = 5

d(v) = ∞

c(u , v) = -1

Since (5 -1) is less than ∞, so update

d(v) = d(u) + c(u , v)
d(v) = 5 - 1 = 4

Therefore, the distance of vertex F is 4.

Consider the edge (E, F). Denote vertex 'E' as 'u' and vertex 'F' as 'v'. Now use the relaxing formula:

d(u) = 5

d(v) = ∞

c(u , v) = 3

Since (5 + 3) is greater than 4, so there would be no updation on the distance value of vertex F.

Consider the edge (C, B). Denote vertex 'C' as 'u' and vertex 'B' as 'v'. Now use the relaxing formula:

d(u) = 3

d(v) = 6

c(u , v) = -2

Since (3 - 2) is less than 6, so update

1. d(v) = d(u) + c(u , v)
   d(v) = 3 - 2 = 1

Therefore, the distance of vertex B is 1.

Now the first iteration is completed. We move to the second iteration.

Second iteration:

In the second iteration, we again check all the edges. The first edge is (A, B). Since (0 + 6) is greater than 1 so there would be no updation in the vertex B.

The next edge is (A, C). Since (0 + 4) is greater than 3 so there would be no updation in the vertex C.

The next edge is (A, D). Since (0 + 5) equals to 5 so there would be no updation in the vertex D.

The next edge is (B, E). Since (1 - 1) equals to 0 which is less than 5 so update:

d(v) = d(u) + c(u, v)

d(E) = d(B) +c(B , E)

= 1 - 1 = 0

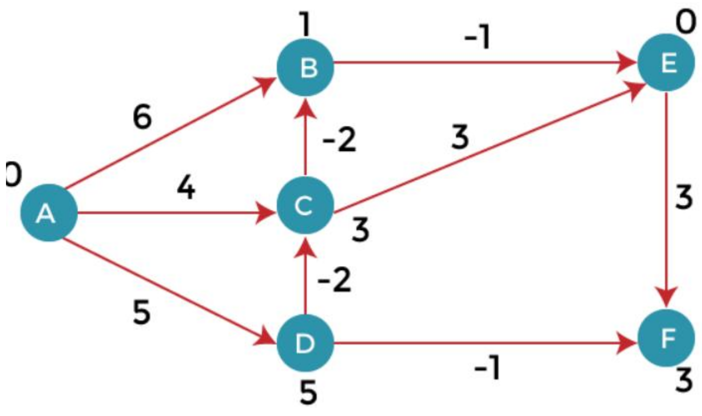
The next edge is (C, E). Since (3 + 3) equals to 6 which is greater than 5 so there would be no updation in the vertex E.

The next edge is (D, C). Since (5 - 2) equals to 3 so there would be no updation in the vertex C.

The next edge is (D, F). Since (5 - 1) equals to 4 so there would be no updation in the vertex F.

The next edge is (E, F). Since (5 + 3) equals to 8 which is greater than 4 so there would be no updation in the vertex F.

The next edge is (C, B). Since (3 - 2) equals to 1` so there would be no updation in the vertex B.



Third iteration

We will perform the same steps as we did in the previous iterations. We will observe that there will be no updation in the distance of vertices.

```
The following are the distances of vertices:
A: 0
B: 1
C: 3
D: 5
E: 0
F: 3
```

Time Complexity

The time complexity of Bellman ford algorithm would be O(E|V| - 1).

```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
      previous[V] <- NULL
  distance[S] <- 0

  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U

  for each edge (U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V}
      Error: Negative Cycle Exists

  return distance[], previous[]
```

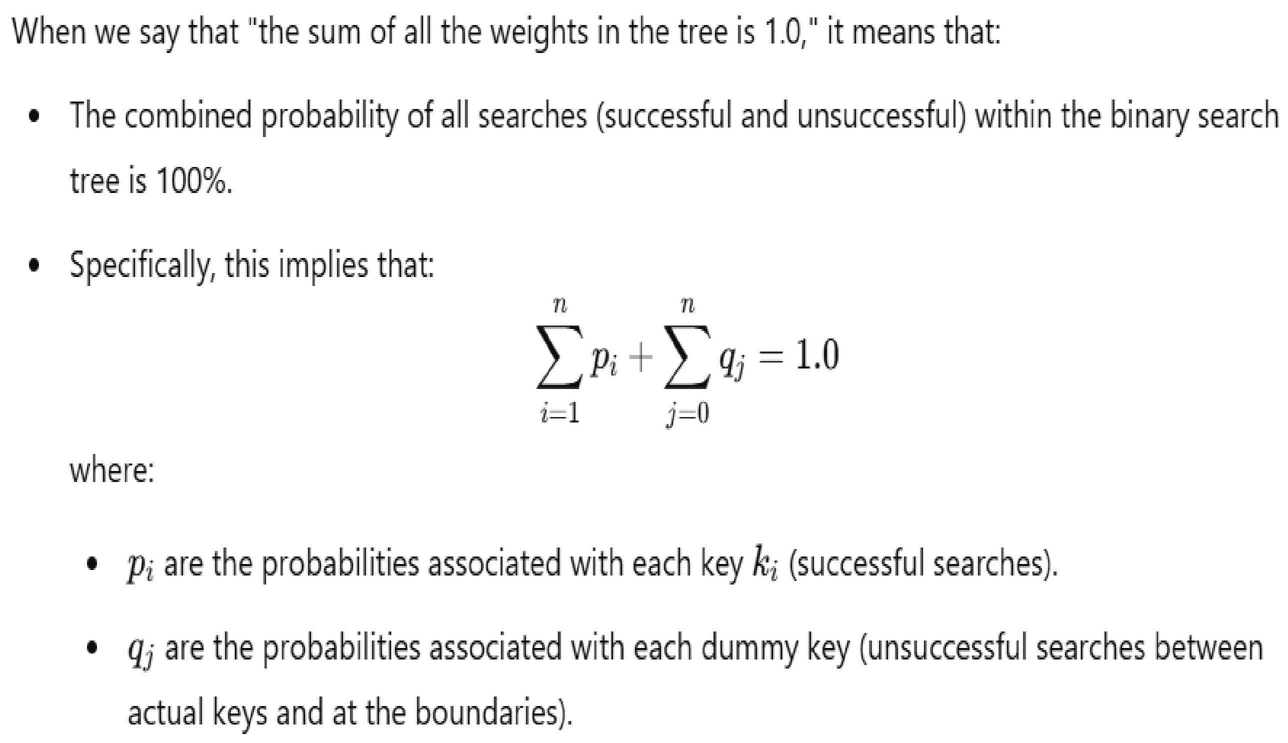
Optimal Binary Search Tree

An Optimal Binary Search Tree (OBST), also known as a Weighted Binary Search Tree, is a binary search tree that minimizes the expected search cost.

In a binary search tree, the search cost is the number of comparisons required to search for a given key.

In an OBST, each node is assigned a weight that represents the probability of the key being searched for.

The sum of all the weights in the tree is 1.0.



When we say that "the sum of all the weights in the tree is 1.0," it means that:

- The combined probability of all searches (successful and unsuccessful) within the binary search tree is 100%.
- Specifically, this implies that:

$$\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1.0$$

  where:

  - p_i are the probabilities associated with each key k_i (successful searches).
  - q_j are the probabilities associated with each dummy key (unsuccessful searches between actual keys and at the boundaries).

The expected search cost of a node is the sum of the product of its depth and weight, and the expected search cost of its children.

To construct an OBST, we start with a sorted list of keys and their probabilities.

We then build a table that contains the expected search cost for all possible sub-trees of the original list.

We can use dynamic programming to fill in this table efficiently. Finally, we use this table to construct the OBST.

The time complexity of constructing an OBST is O(n^3), where n is the number of keys. However, with some optimizations, we can reduce the time complexity to O(n^2).

Once the OBST is constructed, the time complexity of searching for a key is O(log n), the same as for a regular binary search tree.

Given a sorted array key *[0.. n-1]* of search keys and an array *freq[0.. n-1]* of frequency counts, where *freq[i]* is the number of searches for *keys[i]*.

Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

Let us first define the cost of a BST. The cost of a BST node is the level of that node multiplied by its frequency. The level of the root is 1.

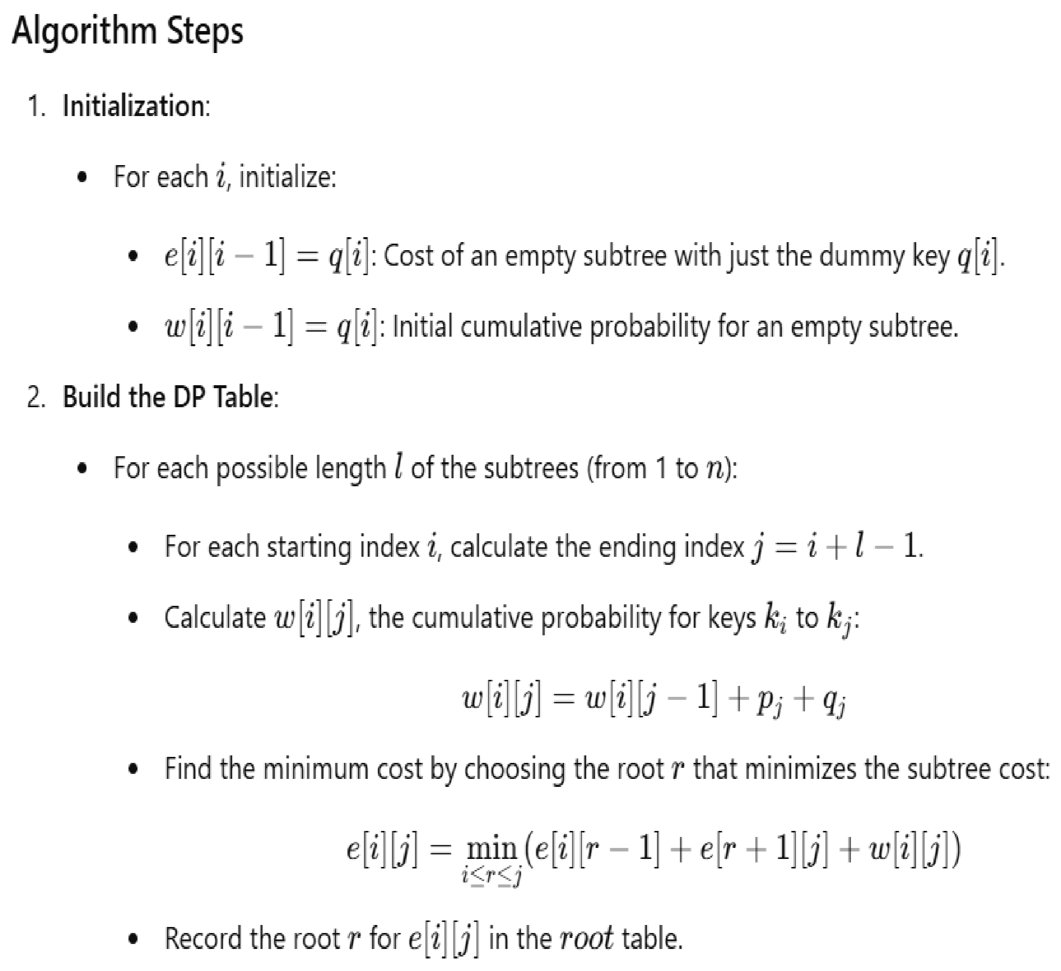
Examples:

```
Input:  keys[] = {10, 12}, freq[] = {34, 50}
There can be following two possible BSTs
        10                       12
          \                     /
           12                 10
          I                     II
Frequency of searches of 10 and 12 are 34 and 50 respectively.
The cost of tree I is 34*1 + 50*2 = 134
The cost of tree II is 50*1 + 34*2 = 118


Input:  keys[] = {10, 12, 20}, freq[] = {34, 8, 50}
There can be following possible BSTs
    10                12                  20         10           20
      \             /    \              /              \         /
      12          10     20           12               20      10
        \                            /                 /         \
         20                        10                12           12
     I               II             III             IV             V

Among all possible BSTs, cost of the fifth BST is minimum.
Cost of the fifth BST is 1*50 + 2*34 + 3*8 = 142
```

Dynamic Programming Approach

We'll use dynamic programming to solve this problem efficiently. Let's define:

- e[i][j]: Minimum search cost for subtree containing keys from ki to kj.
- w[i][j]: Sum of probabilities from ki to kj (includes all successful and unsuccessful searches within that range).
- root[i][j]: Root of the optimal subtree for keys ki to kj.



Algorithm Steps

1. **Initialization**:
   - For each i, initialize:
     - $e[i][i-1] = q[i]$: Cost of an empty subtree with just the dummy key $q[i]$.
     - $w[i][i-1] = q[i]$: Initial cumulative probability for an empty subtree.
2. **Build the DP Table**:
   - For each possible length l of the subtrees (from 1 to n):
     - For each starting index i, calculate the ending index $j = i + l - 1$.
     - Calculate $w[i][j]$, the cumulative probability for keys k_i to k_j:

$$w[i][j] = w[i][j-1] + p_j + q_j$$

     - Find the minimum cost by choosing the root r that minimizes the subtree cost:

$$e[i][j] = \min_{i \leq r \leq j} (e[i][r-1] + e[r+1][j] + w[i][j])$$

     - Record the root r for $e[i][j]$ in the $root$ table.

Reconstruction of the Tree:

    - The root table stores the root for each subtree, which can be used to construct the optimal tree.

0/1 Knapsack Problem (Dynamic Programming Approach)

Knapsack Problem-

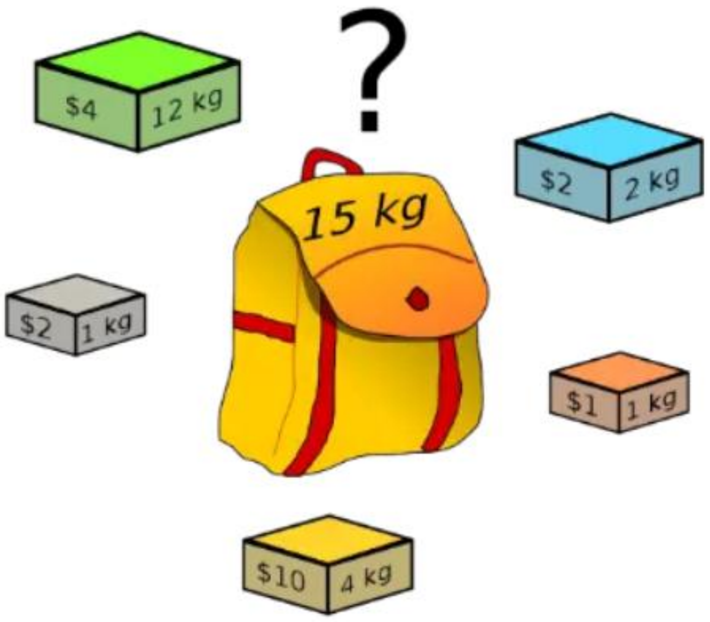
You are given the following-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

The problem states-

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.



Knapsack Problem

Knapsack Problem Variants-

Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

0/1 Knapsack Problem-

In 0/1 Knapsack Problem,

- As the name suggests, items are indivisible here.
- We can not take the fraction of any item.
- We have to either take an item completely or leave it completely.

- It is solved using dynamic programming approach.

The 0/1 Knapsack Problem can be efficiently solved using a **Dynamic Programming (DP)** approach. Here's how it can be implemented step-by-step.

Dynamic Programming Approach

1. **Problem Restatement:** Given n*n* items, each with a weight wi and value *vi*, and a knapsack with a capacity *W*, we want to maximize the total value without exceeding the knapsack's weight limit.
2. **DP Table Definition:** Let dp[i][j] represent the maximum value achievable using the first i*i* items with a knapsack capacity of j.
3. **DP Table Initialization:**
    - dp[0][j] = 0 for all j (if there are no items, the maximum achievable value is 0).
    - dp[i][0] = 0 for all i*i* (if the knapsack capacity is 0, no items can be included).
4. **Transition Formula:** For each item i and each capacity j:
    - If the weight wi of the item is greater than j, we can't include it:

      $$dp[i][j]=dp[i-1][j]$$

    - Otherwise, we have two options: either exclude the item or include it. We take the maximum value between these two choices:

      $$dp[i][j]=\max(dp[i-1][j], vi+dp[i-1][j-wi])$$

5. **Final Solution:** The maximum achievable with n*n* items and knapsack capacity W is found at dp[n][w].

Algorithm:

The **0/1 Knapsack Problem** algorithm using dynamic programming involves building a solution iteratively by breaking down the problem into smaller subproblems. Below is a step-by-step guide and pseudocode for the algorithm.

Steps of the 0/1 Knapsack Algorithm

Given:

- nn: number of items
- WW: capacity of the knapsack
- v[i]$v[i]$: value of the ii-th item
- w[i]$w[i]$: weight of the ii-th item

Objective:

- Maximize the total value of items included in the knapsack without exceeding capacity WW.

Step 1: Define the DP Table

- Let dp[i][j] represent the maximum value achievable using the first ii items with a knapsack capacity of jj.

Step 2: Initialize the DP Table

- If there are no items or the capacity is 0, the maximum value is 0:
    - Set dp[0][j] = 0 for all jj.
    - Set dp[i][0] = 0 for all ii.

Step 3: Populate the DP Table

- For each item ii (from 1 to nn):
    - For each weight jj (from 1 to WW):
        - If the weight of the current item w[i−1]$w[i-1]$ is greater than jj, skip it:dp[i][j]=dp[i−1][j]$dp[i][j]=dp[i-1][j]$

- Otherwise, decide whether to include the item or not by choosing the maximum of these two options:
    - Exclude the item: dp[i-1][j]
    - Include the item: v[i-1] + dp[i-1][j - w[i-1]]

dp[i][j]=max⁡(dp[i−1][j],v[i−1]+dp[i−1][j−w[i−1]])*dp*[*i*][*j*]=max(*dp*[*i*−1][*j*],*v*[*i*−1]+*dp*[*i*−1][*j*−*w*[*i*−1]])

Step 4: Get the Result

- The maximum value achievable is stored in dp[n][W], where n*n* is the number of items and W*W* is the knapsack capacity.

Travelling Salesperson problem

In the following tutorial, we will discuss the Travelling Salesman Problem with its solution and implementation in different programming languages using different approaches.

So, let's get started.

Understanding the Travelling Salesman Problem

The Travelling Salesman Problem (also known as the Travelling Salesperson Problem or TSP) is an NP-hard graph computational problem where the salesman must visit all cities (denoted using vertices in a graph) given in a set just once. The distances (denoted using edges in the graph) between all these cities are known. We are requested to find the shortest possible route in which the salesman visits all the cities and returns to the origin city.

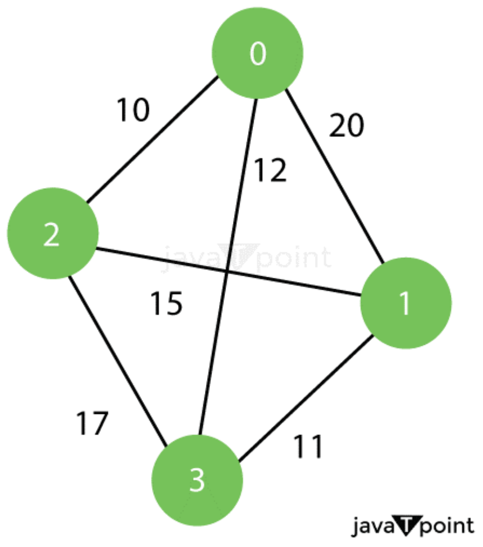
Note:

There is a difference between Hamiltonian Cycle and TSP. The Hamiltonian Cycle problem is the problem where we have to find if there exists a tour that visits every city accurately once. However, in this problem, we already know that Hamiltonian Tour exists as the graph is complete, and in fact, there exist many such tours. Therefore, the problem is to find a minimum weight Hamiltonian Cycle.

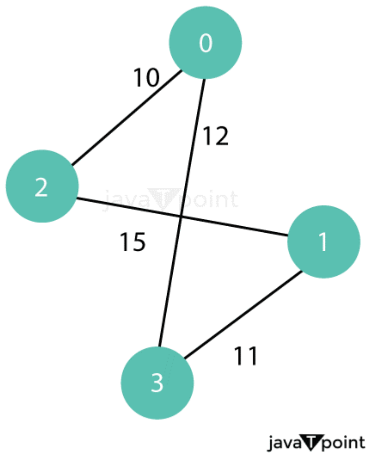
Let us consider the following examples demonstrating the problem:

Example 1 of Travelling Salesman Problem

Input:

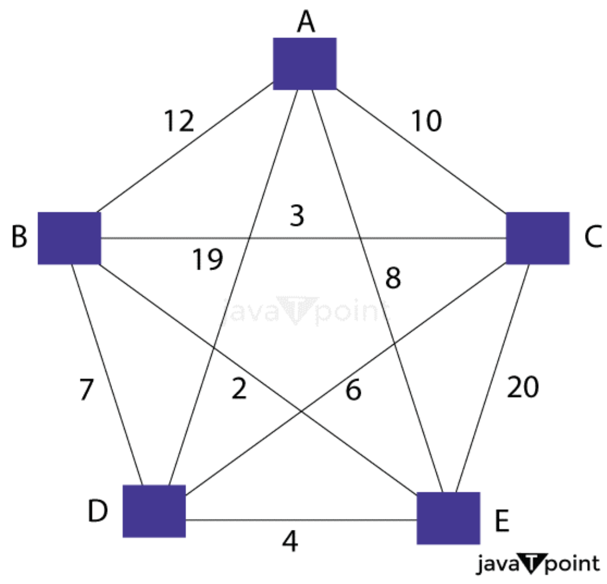


Output:



Example 2 of Travelling Salesman Problem

Input:



Output:

```
Minimum Weight Hamiltonian Cycle: EACBDE = 32
```

Solution of the Travelling Salesman Problem

In the following section, we will discuss different approaches that we will use to solve the problem and their implementations in different programming languages like C, C++, Java, and Python.

The following are some approaches that we are going to use:

1. Simple or Naive Approach
2. Dynamic Programming Approach
3. Greedy Approach

Solving Travelling Salesman Problem Using Dynamic Programming Approach

In the following approach, we will solve the problem using the steps mentioned below:

Step 1: In travelling salesman problem algorithm, we will accept a subset N of the cities that require to be visited, the distance among the cities, and starting city S as inputs. Each city is represented by a unique city id (let's say 1, 2, 3, 4, 5, ..., n).

Step 2: Let us consider 1 as the starting and ending point of the problem. For every other node V (except 1), we will calculate the minimum cost path having 1 as the starting point, V as the ending point, and all nodes appearing exactly once.

Step 3: Let the cost of this path cost (i), and the cost of the corresponding cycle would be cost(i) + distance(i, 1) where the distance(i, 1) from V to 1. Finally, we will return the minimum of all [cost(i) + distance(i, 1)] values.

The above procedure looks quite easy. However, the main question is how to calculate the cost(i)? In order to calculate the cost(i) with the help of Dynamic Programming, we are required to have some recursive relations in terms of sub-problems.

Let us define a term C(S, i) be the cost of the minimum cost path visiting each node in set S exactly once, beginning at 1 and ending at i. We will begin with all subsets of size 2 and calculate C(S, i) for all subsets S of size 3 and so on.

Note:

Node 1 must be present in every subset.

Implementation of Dynamic Programming Approach to Solve Travelling Salesman Problem in Different Programming Languages

We will now see the implementation of the Dynamic Programming solution using the Top-Down Recursive + Memorized approach in different programming languages like C++, Java, and Python.

To maintain the subsets, we can utilize the bitmasks in order to represent the remaining nodes in the subset. Since the operations are carried out faster using bits, and there are only a few nodes in the graph, bitmasks are better for usage.

For instance:

10100 signifies that nodes 2 and 4 are left in set for processing.

010010 signifies that nodes 1 and 4 are left in the subset.

BACKTRACKING ALGORITHM: GENERAL METHOD

The **backtracking algorithm** is a problem-solving approach that tries out all the possible solutions and chooses the best or desired ones. Generally, it is used to solve problems that have multiple solutions. The term backtracking suggests that for a given problem, if the current solution is not suitable, eliminate it and then backtrack to try other solutions.

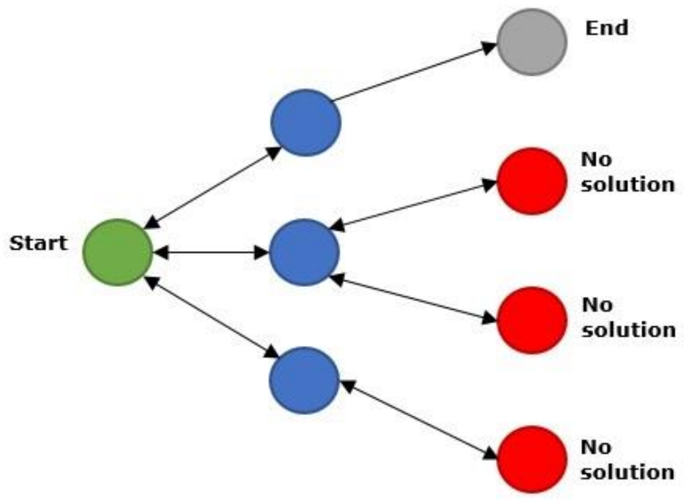
When do we use backtracking algorithm?

Backtracking algorithm can be used for the following problems −

- The problem has multiple solutions or requires finding all possible solutions.
- When the given problem can be broken down into smaller subproblems that are similar to the original problem.
- If the problem has some constraints or rules that must be satisfied by the solution

How does backtracking algorithm work?

The backtracking algorithm explores various paths to find a sequence path that takes us to the solution. Along these paths, it establishes some small checkpoints from where the problem can backtrack if no feasible solution is found. This process continues until the best solution is found.



In the above figure, **green** is the start point, **blue** is the intermediate point, **red** are points with no feasible solution, **grey** is the end solution.

When backtracking algorithm reaches the end of the solution, it checks whether this path is a solution or not. If it is the solution path, it returns that otherwise, it backtracks to one step behind in order to find a solution.

Algorithm

Following is the algorithm for backtracking −

```
1. Start
2. if current_position is goal, return success.
3. else
4. if current_position is an end point, return failed.
5. else-if current_position is not end point, explore and repeat above steps.
6. Stop
```

Complexity of Backtracking

Generally, the time complexity of backtracking algorithm is exponential ($O(k^n)$). In some cases, it is observed that its time complexity is factorial ($O(N!)$).

Types of Backtracking Problem

The backtracking algorithm is applied to some specific types of problems. They are as follows −

- **Decision problem** − It is used to find a feasible solution of the problem.
- **Optimization problem** − It is used to find the best solution that can be applied.
- **Enumeration problem**− It is used to find the set of all feasible solutions of the problem.

Examples of Backtracking Algorithm

The following list shows examples of Backtracking Algorithm −

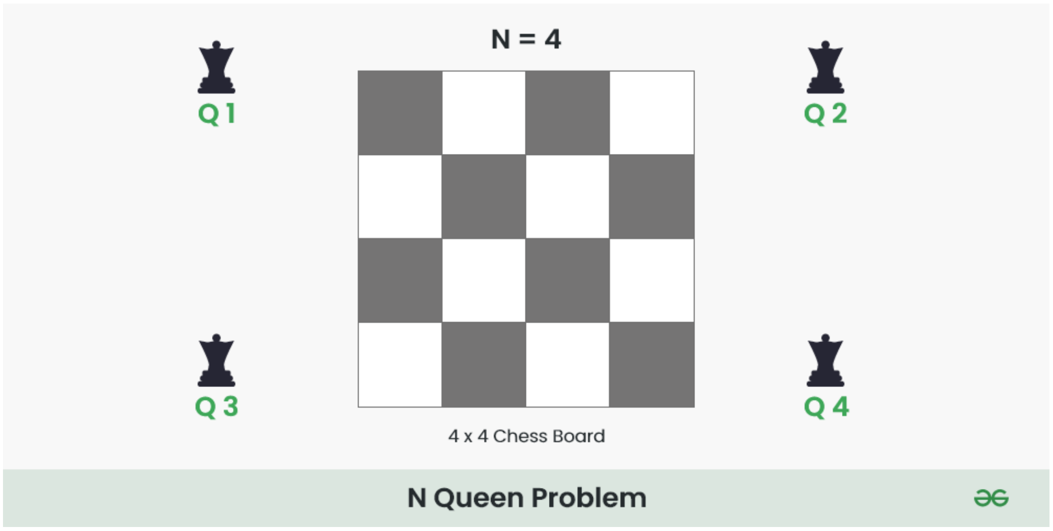
- Hamiltonian Cycle
- M-Coloring Problem
- N Queen Problem
- Rat in Maze Problem
- Cryptarithmetic Puzzle
- Subset Sum Problem
- Sudoku Solving Algorithm
- Knight-Tour Problem
- Tug-Of-War Problem
- Word Break Problem
- Maximum number by swapping problem

8 Queens Problem using Backtracking

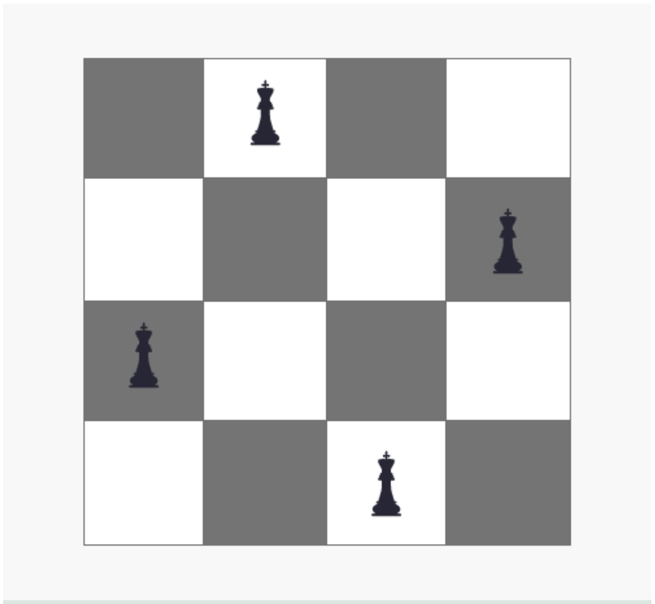
N Queen Problem

What is N-Queen problem?

The **N** Queen is the problem of placing **N** chess queens on an **N×N** chessboard so that no two queens attack each other.



For example, the following is a solution for the 4 Queen problem.


Solution Of 4 Queen Problem

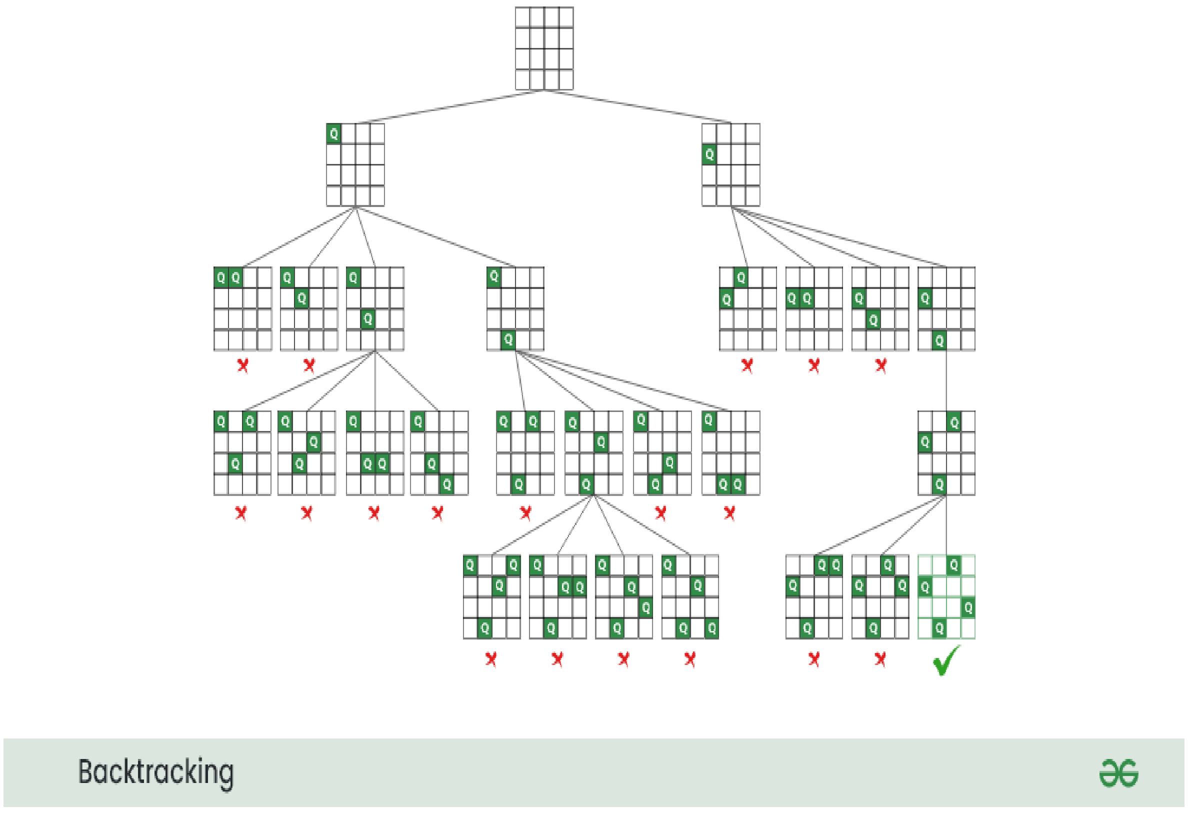
The expected output is in the form of a matrix that has ‘**Q**‘s for the blocks where queens are placed and the empty spaces are represented by **‘.’** . For example, the following is the output matrix for the above 4-Queen solution.

```
. Q . .
. . . Q
Q . . .
. . Q .
```

N Queen Problem using **Backtracking**:

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return ***false****.*

Below is the recursive tree of the above approach:



Recursive tree for N Queen problem

Algorithm:

Follow the steps mentioned below to implement the idea:

- Start in the leftmost column
- If all queens are placed return true
- Try all rows in the current column. Do the following for every row.
    - If the queen can be placed safely in this row
        - Then mark this **[row, column]** as part of the solution and recursively check if placing queen here leads to a solution.
        - If placing the queen in **[row, column]** leads to a solution then return **true**.
        - If placing queen doesn't lead to a solution then unmark this **[row, column]** then backtrack and try other rows.
    - If all rows have been tried and valid solution is not found return **false** to trigger backtracking.

// C program to solve N Queen Problem using backtracking

```
#define N 4
#include <stdbool.h>
#include <stdio.h>

// A utility function to print solution
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if(board[i][j])
                printf("Q ");
            else
                printf(". ");
        }
        printf("\n");
    }
}

// A utility function to check if a queen can
// be placed on board[row][col]. Note that this
// function is called when "col" queens are
// already placed in columns from 0 to col -1.
// So we need to check only left side for
// attacking queens
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    // Check this row on left side
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
```

```
        if (board[i][j])
            return false;

    return true;
}

// A recursive utility function to solve N
// Queen problem
bool solveNQUtil(int board[N][N], int col)
{
    // Base case: If all queens are placed
    // then return true
    if (col >= N)
        return true;

    // Consider this column and try placing
    // this queen in all rows one by one
    for (int i = 0; i < N; i++) {

        // Check if the queen can be placed on
        // board[i][col]
        if (isSafe(board, i, col)) {

            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col]
            // doesn't lead to a solution, then
            // remove queen from board[i][col]
            board[i][col] = 0; // BACKTRACK
        }
    }

    // If the queen cannot be placed in any row in
    // this column col  then return false
    return false;
}
```

```
// This function solves the N Queen problem using
// Backtracking. It mainly uses solveNQUtil() to
// solve the problem. It returns false if queens
// cannot be placed, otherwise, return true and
// prints placement of queens in the form of 1s.
// Please note that there may be more than one
// solutions, this function prints one  of the
// feasible solutions.
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

// Driver program to test above function
int main()
{
    solveNQ();
    return 0;
}

// This code is contributed by Aditya Kumar (adityakumar129)
```

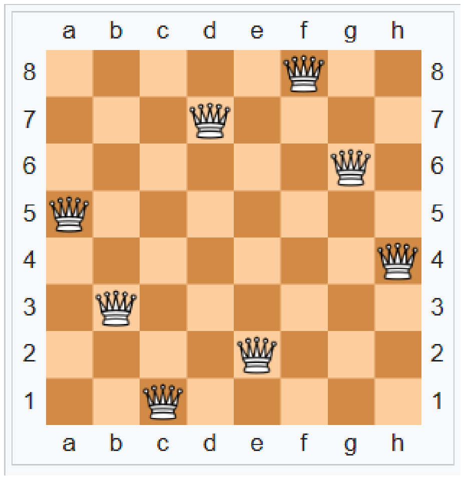
Output

```
. . Q .

Q . . .

. . . Q

. Q . .
```

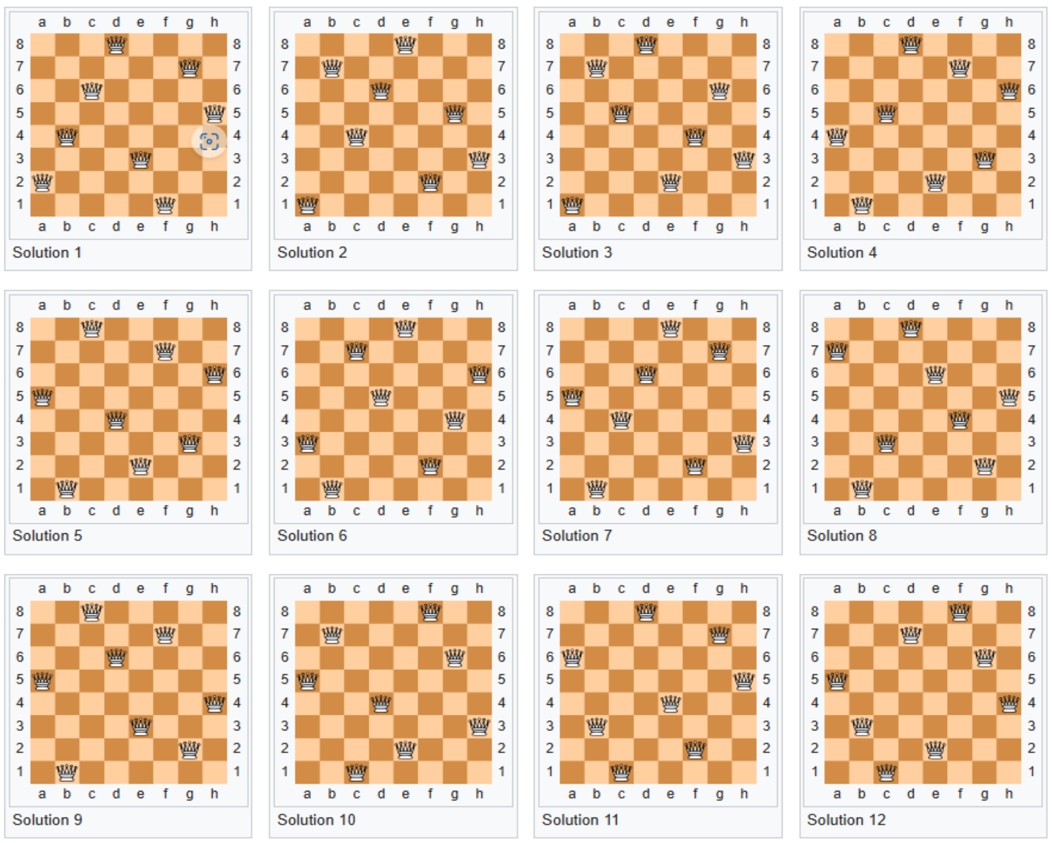
Time Complexity: O(N!)
Auxiliary Space: O(N^2)

8 Queens Problem:

The 8 Queens problem is a classic puzzle in which the goal is to place 8 queens on a standard 8x8 chessboard such that no two queens threaten each other. This means no two queens can share the same row, column, or diagonal. Here's a step-by-step guide to solving this problem using backtracking:



All fundamental solutions are presented below:



Steps for Solving the 8 Queens Problem

1. **Initialize the Board:** Start with an empty 8x8 chessboard.
2. **Place Queens:** Begin placing queens one by one in different rows.
3. **Check Safety:** Before placing a queen, check if the position is safe from attack by any previously placed queens.
4. **Backtracking:** If placing the queen in a current position leads to a dead-end, backtrack by removing the queen and trying the next position.

Code Example

Here's a simple Python implementation using backtracking:

```
N = 8 # (size of the chessboard)
def solveNQueens(board, col):
    if col == N:
        print(board)
        return True
    for i in range(N):
        if isSafe(board, i, col):
                board[i][col] = 1
                if solveNQueens(board, col + 1):
                    return True
                board[i][col] = 0
    return False

def isSafe(board, row, col):
    for x in range(col):
        if board[row][x] == 1:
            return False
    for x, y in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False
    for x, y in zip(range(row, N, 1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False
    return True

board = [[0 for x in range(N)] for y in range(N)]
if not solveNQueens(board, 0):
    print("No solution found")
```

Output

```
[[1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 1,
0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1], [0, 1, 0, 0, 0, 0, 0, 0], [0, 0,
0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 0], [0, 0, 1, 0, 0, 0, 0,
0]]
```

Time Complexity : O((m + q) log^2 n)
Space Complexity : O((m + q) log n)

Subset Sum Problem using Backtracking

Given a **set[]** of non-negative integers and a value **sum**, the task is to print the subset of the given set whose sum is equal to the given **sum**.

Examples:
Input: *set[] = {1,2,1}, sum = 3*
Output: *[1,2],[2,1]*
Explanation: *There are subsets [1,2],[2,1] with sum 3.*

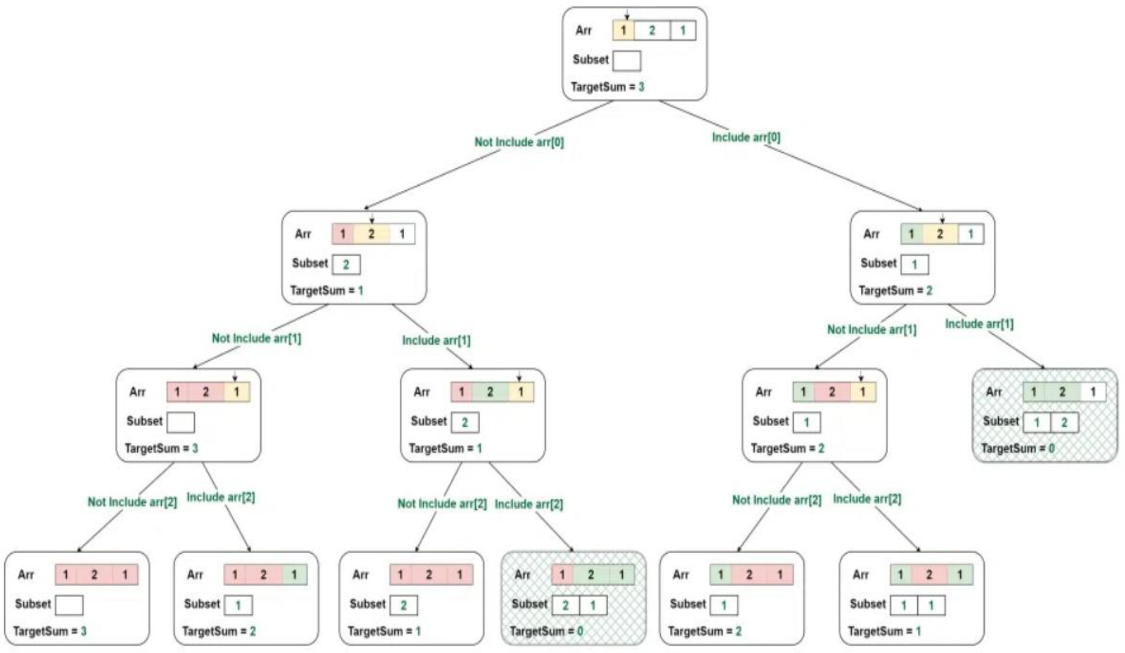
Input: *set[] = {3, 34, 4, 12, 5, 2}, sum = 30*
Output: *[]*

Explanation: *There is no subset that add up to 30.*
Subset Sum Problem using Backtracking

Subset sum can also be thought of as a special case of the 0–1 Knapsack problem. For each item, there are two possibilities:

- ***Include** the current element in the subset and recur for the remaining elements with the remaining **Sum**.*
- ***Exclude** the current element from the subset and recur for the remaining elements.*

*Finally, if **Sum** becomes **0** then print the elements of current subset. The recursion's **base case** would be when **no items are left**, or the **sum** becomes **negative**, then simply return.*



Subset Sum Problem

```
# Print all subsets if there is at least one subset of set[]
# with a sum equal to the given sum
flag = False

def print_subset_sum(i, n, _set, target_sum, subset):
    global flag
    # If targetSum is zero, then there exists a subset
    if target_sum == 0:
        # Prints valid subset
        flag = True
        print("[", end=" ")
        for element in subset:
            print(element, end=" ")
        print("]", end=" ")
        return

    if i == n:
        # Return if we have reached the end of the array
        return

    # Not considering the current element
    print_subset_sum(i + 1, n, _set, target_sum, subset)
```

```
        # Consider the current element if it is less than or equal to targetSum
        if _set[i] <= target_sum:
                # Push the current element into the subset
                subset.append(_set[i])

                # Recursive call for considering the current element
                print_subset_sum(i + 1, n, _set, target_sum - _set[i], subset)

                # Remove the last element after recursive call to restore subset's
original configuration
                subset.pop()

# Driver code
if __name__ == "__main__":
        # Test case 1
        set_1 = [1, 2, 1]
        sum_1 = 3
        n_1 = len(set_1)
        subset_1 = []
        print("Output 1:")
        print_subset_sum(0, n_1, set_1, sum_1, subset_1)
        print()
        flag = False
```

```
# Test case 2

set_2 = [3, 34, 4, 12, 5, 2]

sum_2 = 30

n_2 = len(set_2)

subset_2 = []

print("Output 2:")

print_subset_sum(0, n_2, set_2, sum_2, subset_2)

if not flag:

    print("There is no such subset")
```

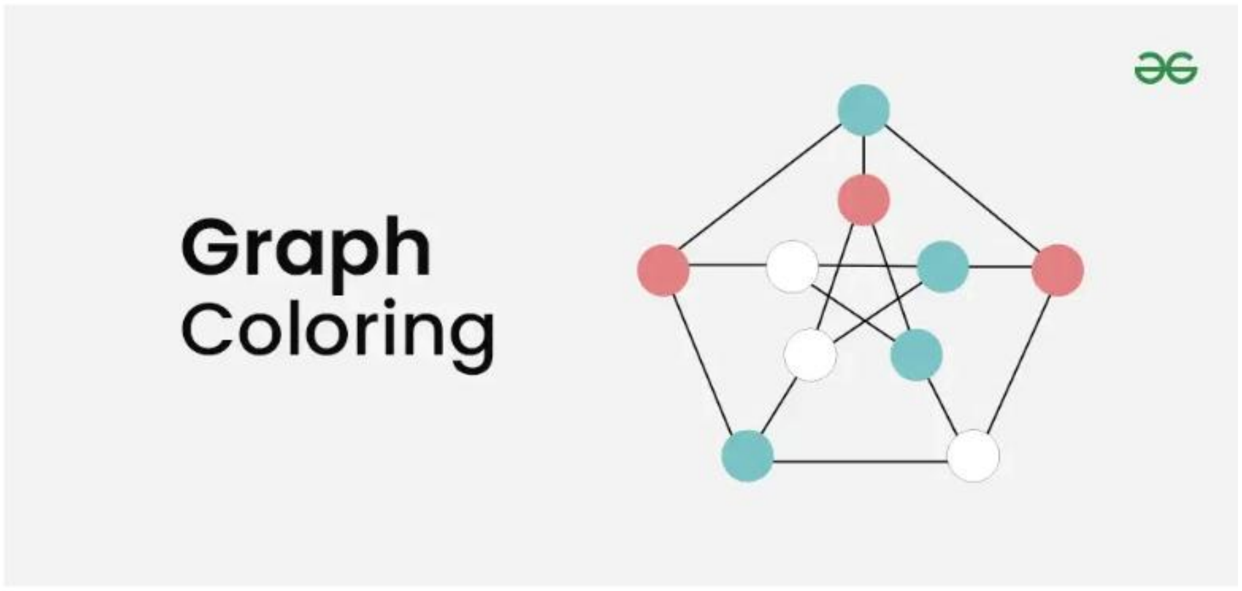
```
Output 1:
[ 2 1 ][ 1 2 ]
Output 2:
There is no such subset
```

Complexity analysis:

- **Time Complexity:** O(2^n) The above solution may try all subsets of the given set in the worst case. Therefore time complexity of the above solution is **exponential**.
- **Auxiliary Space:** O(n) where n is recursion stack space.

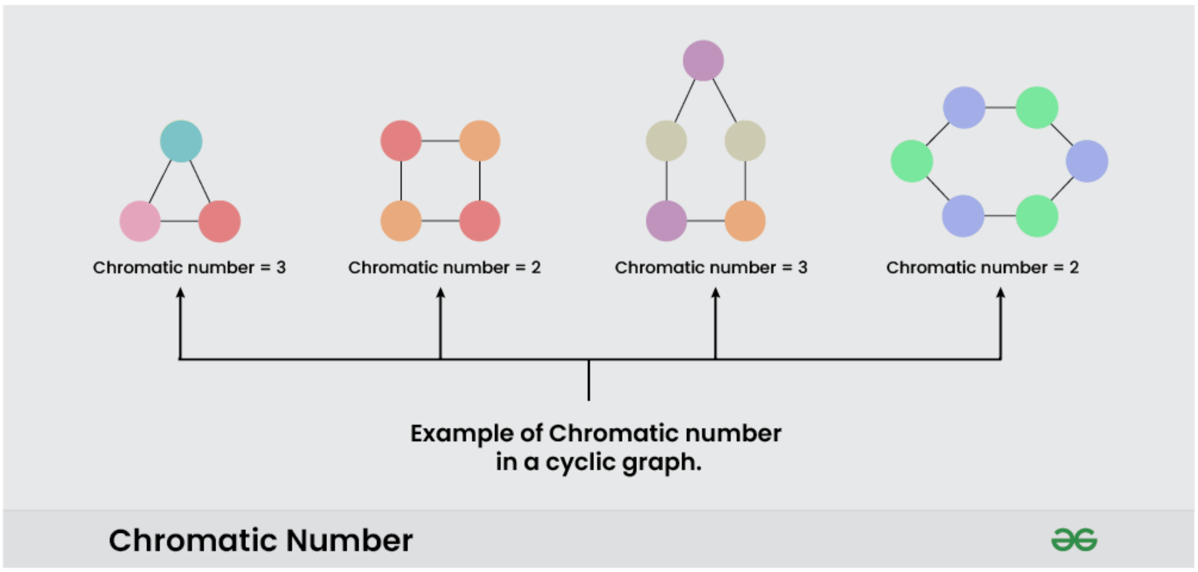
Graph coloring:

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 m colors, it is called m-coloring.*



Chromatic Number:

The ***minimum*** *number of* ***colors*** *needed to* ***color*** *a graph is called its* ***chromatic*** *number. For example, the following can be colored a minimum of 2 colors.*



Example of Chromatic Number

The problem of finding a chromatic number of a given graph is NP-complete. Graph coloring problem is both, a **decision problem** as well as an **optimization problem**.

- A decision problem is stated as, "With given M colors and graph G, whether a such color scheme is possible or not?".
- The optimization problem is stated as, "Given M colors and graph G, find the minimum number of colors required for graph coloring."

Algorithm of Graph Coloring using **Backtracking**:

*Assign colors one by one to different vertices, starting from vertex **0**. Before assigning a color, check if the adjacent vertices have the same color or not. If there is any color assignment that does not violate the conditions, mark the color assignment as part of the solution. If no assignment of color is possible then backtrack and return false.*

Follow the given steps to solve the problem:

- Create a recursive function that takes the graph, current index, number of vertices, and color array.
- If the current index is equal to the number of vertices. Print the color configuration in the color array.
- Assign a color to a vertex from the range (1 to m).

- For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) and recursively call the function with the next index and number of vertices else return **false**
- If any recursive function returns **true** then break the loop and return true
- If no recursive function returns **true** then return **false**

```
V = 4
def print_solution(color):
    print("Solution Exists: Following are the assigned colors")
    print(" ".join(map(str, color)))

def is_safe(v, graph, color, c):
    # Check if the color 'c' is safe for the vertex 'v'
    for i in range(V):
        if graph[v][i] and c == color[i]:
            return False
    return True

def graph_coloring_util(graph, m, color, v):
    # Base case: If all vertices are assigned a color, return true
    if v == V:
        return True

    # Try different colors for the current vertex 'v'
    for c in range(1, m + 1):
        # Check if assignment of color 'c' to 'v' is fine
        if is_safe(v, graph, color, c):
            color[v] = c

            # Recur to assign colors to the rest of the vertices
            if graph_coloring_util(graph, m, color, v + 1):
                return True

            # If assigning color 'c' doesn't lead to a solution, remove it
            color[v] = 0

    # If no color can be assigned to this vertex, return false
    return False

def graph_coloring(graph, m):
    color = [0] * V

    # Call graph_coloring_util() for vertex 0
    if not graph_coloring_util(graph, m, color, 0):
        print("Solution does not exist")
        return False

    # Print the solution
    print_solution(color)
    return True

# Driver code
if __name__ == "__main__":
    graph = [
        [0, 1, 1, 1],
        [1, 0, 1, 0],
        [1, 1, 0, 1],
        [1, 0, 1, 0],
    ]

    m = 3

    # Function call
    graph_coloring(graph, m)
```

Output

```
Solution Exists: Following are the assigned colors
 1  2  3  2
```

Applications of Graph Coloring:

- Design a timetable.
- Sudoku.
- Register allocation in the compiler.
- Map coloring.
- Mobile radio frequency assignment.

0/1 knapsack problem using backtracking

0/1 knapsack problem using backtracking example and algorithm

Sure! Let's dive into the 0/1 knapsack problem using backtracking with a detailed algorithm and an example to illustrate how it works.

Algorithm

Here's a step-by-step algorithm for solving the 0/1 knapsack problem using backtracking:

1. **Define the Function**: Create a function that takes the list of item values, item weights, the remaining capacity of the knapsack, and the number of items left to consider.
2. **Base Case**: If there are no items left or the remaining capacity is zero, return a value of 0.
3. **Recursive Case**:
    - If the weight of the current item exceeds the remaining capacity, skip this item and move to the next.
    - Otherwise, compute two scenarios:
        1. Including the current item (add its value to the result and reduce the capacity by its weight).
        2. Excluding the current item (just move to the next item).
    - Return the maximum value obtained from the two scenarios.

Example

Let's consider an example with the following items:

- **Values**: [60, 100, 120]

- **Weights**: [10, 20, 30]
- **Knapsack Capacity**: 50

Python Code

Here's the Python code that implements this algorithm:

```
def knapsack_backtracking(values, weights, W, n):
    # Base case: no items left or no remaining capacity
    if n == 0 or W == 0:
        return 0

    # If weight of the nth item is more than knapsack capacity W, then this item
cannot be included
    if weights[n-1] > W:
        return knapsack_backtracking(values, weights, W, n-1)

    # Return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    else:
        include_item = values[n-1] + knapsack_backtracking(values, weights, W -
weights[n-1], n-1)
        exclude_item = knapsack_backtracking(values, weights, W, n-1)
        return max(include_item, exclude_item)

# Example usage
values = [60, 100, 120]
weights = [10, 20, 30]
W = 50
n = len(values)

max_value = knapsack_backtracking(values, weights, W, n)
print(f"The maximum value that can be obtained is {max_value}")
```

Explanation

1. **Base Case**: If there are no items left (n == 0) or the remaining capacity is zero (W == 0), the function returns 0 because no more items can be added to the knapsack.

2. **Weight Check**: If the weight of the current item is greater than the remaining capacity, the item cannot be included. The function calls itself recursively, skipping this item.
3. **Include/Exclude Scenarios**: If the item can be included, the function calculates the value of including the item (adding its value and reducing the capacity) and the value of excluding the item. The maximum of these two values is returned.

In the example provided, the maximum value that can be obtained with a knapsack capacity of 50 is 220.