Dynamic Programming (Binomial Coefficient)

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

2) A binomial coefficient \( C(n, k) \) also gives the number of ways, disregarding order, that \( k \) objects can be chosen from among \( n \) objects; more formally, the number of \( k \)-element subsets (or \( k \)-combinations) of an \( n \)-element set.

The Problem
Write a function that takes two parameters \( n \) and \( k \) and returns the value of Binomial Coefficient \( C(n, k) \). For example, your function should return 6 for \( n = 4 \) and \( k = 2 \), and it should return 10 for \( n = 5 \) and \( k = 2 \).

1) Optimal Substructure
The value of \( C(n, k) \) can recursively calculated using following standard formula for Binomial Coefficients.

\[
C(n, k) = C(n-1, k-1) + C(n-1, k)
\]
\[
C(n, 0) = C(n, n) = 1
\]

2) Overlapping Subproblems
Following is simple recursive implementation that simply follows the recursive structure mentioned above.

```c
// A Naive Recursive Implementation
#include<stdio.h>

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    // Base Cases
    if (k==0 || k==n)
        return 1;

    // Recur
    return binomialCoeff(n-1, k-1) + binomialCoeff(n-1, k); 
}

/* Driver program to test above function*/
int main()
{ 
    int n = 5, k = 2;
    printf("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k));
    return 0;
}
```

www.studentsfocus.com
It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for \( n = 5 \) an \( k = 2 \). The function \( C(3, 1) \) is called two times. For large values of \( n \), there will be many common subproblems.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Binomial Coefficient problem has both properties of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array \( C[][] \) in bottom up manner. Following is Dynamic Programming based implementation.

```c
// A Dynamic Programming based solution that uses table C[][] to calculate the Binomial Coefficient
#include<stdio.h>

// Prototype of a utility function that returns minimum of two integers
int min(int a, int b);

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
  int C[n+1][k+1];
  int i, j;

  // Calculate value of Binomial Coefficient in bottom up manner
  for (i = 0; i <= n; i++)
  {
    for (j = 0; j <= min(i, k); j++)
    {
      // Base Cases
      if (j == 0 || j == i)
        C[i][j] = 1;

      // Calculate value using previously stored values
      else
        C[i][j] = C[i-1][j-1] + C[i-1][j];
    }
  }
  return C[n][k];
}

// A utility function to return minimum of two integers
```
```c
int min(int a, int b)
{
    return (a<b)? a: b;
}

/* Drier program to test above function*/
int main()
{
    int n = 5, k = 2;
    printf("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k) );
    return 0;
}

Time Complexity: O(n*k)
Auxiliary Space: O(n*k)

Following is a space optimized version of the above code. The following code only uses O(k).
Thanks to AK for suggesting this method.

// A space optimized Dynamic Programming Solution
int binomialCoeff(int n, int k)
{
    int* C = (int*)calloc(k+1, sizeof(int));
    int i, j, res;
    C[0] = 1;
    for(i = 1; i <= n; i++)
    {
        for(j = min(i, k); j > 0; j--)
            C[j] = C[j] + C[j-1];
    }
    res = C[k]; // Store the result before freeing memory
    free(C); // free dynamically allocated memory to avoid memory leak
    return res;
}

Time Complexity: O(n*k)
Auxiliary Space: O(k)
```

www.studentsfocus.com
Dynamic Programming (Floyd Warshall Algorithm)

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

**Input:**

```
graph[][] = { {0,   5,  INF, 10},
             {INF,  0,   3,  INF},
             {INF, INF, 0,   1},
             {INF, INF, INF, 0} }
```

which represents the following graph

```
10
(0)------->(3)
|         / 目
5 |          | 1
|          | /
\ /        | 3
(1)------->(2)
```

Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

**Output:**

Shortest distance matrix

```
0      5      8      9
INF    0      3      4
INF    INF    0      1
INF    INF    INF    0
```

**Floyd Warshall Algorithm**

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices \{0, 1, 2, .. k-1\} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.
Following is C implementation of the Floyd Warshall algorithm.

```c
#include<stdio.h>

#define V 4

#define INF 99999

void printSolution(int dist[][V]);

void floydWarshall (int graph[][V])
{
    int dist[V][V], i, j, k;

    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
    }

    printSolution(dist);
}
```

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
void floydWarshall (int graph[][V])
{
    int dist[V][V], i, j, k;

    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
    }

    printSolution(dist);
```
// i to j, then update the value of dist[i][j]
if (dist[i][k] + dist[k][j] < dist[i][j])
    dist[i][j] = dist[i][k] + dist[k][j];
}
}

// Print the shortest distance matrix
printSolution(dist);

/* A utility function to print solution */
void printSolution(int dist[][V])
{
    printf("Following matrix shows the shortest distances" 
    "between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }

    // driver program to test above function
    int main()
    {
        /* Let us create the following weighted graph
           10
           (0)-------(3)
           |      /|
           5 |     1
           |      |
           (1)-------(2)
           3 */
        int graph[V][V] = {
            {0, 5, INF, 10},
            {INF, 0, 3, INF},
            {INF, INF, 0, 1},
            {INF, INF, INF, 0}
        };

        // Print the solution
        floydWarshell(graph);
        return 0;
    }

Output:

Following matrix shows the shortest distances between every pair of vertices
The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.

Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

#include<limits.h>
#define INF INT_MAX

..........................
if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] <
    dist[i][j])
    dist[i][j] = dist[i][k] + dist[k][j];
...........................
Dynamic Programming (Optimal Binary Search Tree)

Given a sorted array $keys[0..n-1]$ of search keys and an array $freq[0..n-1]$ of frequency counts, where $freq[i]$ is the number of searches to $keys[i]$. 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 level of that node multiplied by its frequency. Level of root is 1.

**Example 1**
Input: $keys[] = {10, 12}$, $freq[] = {34, 50}$
There can be following two possible BSTs

```
10                       12
\                       /  \
12                    10
```

$Freq$ 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 $50*1 + 34*2 = 118$

**Example 2**
Input: $keys[] = {10, 12, 20}$, $freq[] = {34, 8, 50}$
There can be following possible BSTs

```
10                12                 20
\                       /  \
12          10   20
```

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

1) **Optimal Substructure:**
The optimal cost for $freq[i..j]$ can be recursively calculated using following formula.

We need to calculate $optCost(0, n-1)$ to find the result.

The idea of above formula is simple, we one by one try all nodes as root ($r$ varies from $i$ to $j$ in second term). When we make $r$th node as root, we recursively calculate optimal cost from $i$ to $r-1$ and $r+1$ to $j$.

We add sum of frequencies from $i$ to $j$ (see first term in the above formula), this is added because every search will go through root and one comparison will be done for every search.

2) **Overlapping Subproblems**
Following is recursive implementation that simply follows the recursive structure mentioned above.

```cpp
// A naive recursive implementation of optimal binary search tree problem
```
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

// A recursive function to calculate cost of optimal binary search tree
int optCost(int freq[], int i, int j)
{
    // Base cases
    if (j < i)      // If there are no elements in this subarray
        return 0;
    if (j == i)     // If there is one element in this subarray
        return freq[i];

    // Get sum of freq[i], freq[i+1], ... freq[j]
    int fsu = sum(freq, i, j);

    // Initialize minimum value
    int min = INT_MAX;

    // One by one consider all elements as root and recursively find cost
    // of the BST, compare the cost with min and update min if needed
    for (int r = i; r <= j; ++r)
    {
        int cost = optCost(freq, i, r-1) + optCost(freq, r+1, j);
        if (cost < min)
            min = cost;
    }

    // Return minimum value
    return min + fsu;
}

// The main function that calculates minimum cost of a Binary Search Tree.
// It mainly uses optCost() to find the optimal cost.
int optimalSearchTree(int keys[], int freq[], int n)
{
    // Here array keys[] is assumed to be sorted in increasing order.
    // If keys[] is not sorted, then add code to sort keys, and rearrange
    // freq[] accordingly.
    return optCost(freq, 0, n-1);
}

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
        s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
int freq[] = {34, 8, 50};
int n = sizeof(keys)/sizeof(keys[0]);
printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));
return 0;
}

Output:

Cost of Optimal BST is 142

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. We can see many subproblems being repeated in the following recursion tree for freq[1..4].

Dynamic Programming Solution
Following is C/C++ implementation for optimal BST problem using Dynamic Programming. We use an auxiliary array cost[n][n] to store the solutions of subproblems. cost[0][n-1] will hold the final result. The challenge in implementation is, all diagonal values must be filled first, then the values which lie on the line just above the diagonal. In other words, we must first fill all cost[i][i] values, then all cost[i][i+1] values, then all cost[i][i+2] values. So how to fill the 2D array in such manner? The idea used in the implementation is same as Matrix Chain Multiplication problem, we use a variable ‘L’ for chain length and increment ‘L’, one by one. We calculate column number ‘j’ using the values of ‘i’ and ‘L’.

// Dynamic Programming code for Optimal Binary Search Tree Problem
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

/* A Dynamic Programming based function that calculates minimum cost of a Binary Search Tree. */
int optimalSearchTree(int keys[], int freq[], int n)
{
    /* Create an auxiliary 2D matrix to store results of subproblems */
    int cost[n][n];

    /* cost[i][j] = Optimal cost of binary search tree that can be formed from keys[i] to keys[j].
       cost[0][n-1] will store the resultant cost */

    // For a single key, cost is equal to frequency of the key
    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];

    // Now we need to consider chains of length 2, 3, ..., L.
    // L is chain length.
    for (int L=2; L<=n; L++)
    {
        // i is row number in cost[]
        for (int i=0; i<=n-L+1; i++)
        {

www.studentsfocus.com
// Get column number j from row number i and chain length L
int j = i+L-1;
cost[i][j] = INT_MAX;

// Try making all keys in interval keys[i..j] as root
for (int r=i; r<=j; r++)
{
    // c = cost when keys[r] becomes root of this subtree
    int c = ((r > i)? cost[i][r-1]:0) +
            ((r < j)? cost[r+1][j]:0) +
            sum(freq, i, j);
    if (c < cost[i][j])
        cost[i][j] = c;
}
return cost[0][n-1];

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
        s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is \d ", optimalSearchTree(keys, freq, n));
    return 0;
}

Output:

Cost of Optimal BST is 142

Notes
1) The time complexity of the above solution is O(n^4). The time complexity can be easily reduced to O(n^3) by pre-calculating sum of frequencies instead of calling sum() again and again.

2) In the above solutions, we have computed optimal cost only. The solutions can be easily modified to store the structure of BSTs also. We can create another auxiliary array of size n to store the structure of tree. All we need to do is, store the chosen ‘r’ in the innermost loop.
Dynamic Programming (0-1 Knapsack Problem)

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

1) Optimal Substructure:
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.
Therefore, the maximum value that can be obtained from n items is max of following two values.
1) Maximum value obtained by n-1 items and W weight (excluding nth item).
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

2) Overlapping Subproblems
Following is recursive implementation that simply follows the recursive structure mentioned above.

```c
/* A Naive recursive implementation of 0-1 Knapsack problem */
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// A recursive function to return the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    // Base Case
    if (n == 0 || W == 0)
        return 0;

    // If weight of the nth item is more than Knapsack capacity W, then
    // this item cannot be included in the optimal solution
    if (wt[n-1] > W)
        return knapSack(W, wt, val, n-1);

    // Return the maximum of two cases: (1) nth item included (2) not included
    else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                     knapSack(W, wt, val, n-1) );
}
```

www.studentsfocus.com
It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).

In the following recursion tree, K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W. The recursion tree is for following sample inputs.

wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

Since subproblems are evaluated again, this problem has Overlapping Subprolems property. So the 0-1 Knapsack problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

// A Dynamic Programming based solution for 0-1 Knapsack problem
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n+1][W+1];
// Build table K[][] in bottom up manner
for (i = 0; i <= n; i++)
{
    for (w = 0; w <= W; w++)
    {
        if (i==0 || w==0)
            K[i][w] = 0;
        else if (wt[i-1] <= w)
            K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
        else
            K[i][w] = K[i-1][w];
    }
}

return K[n][W];

int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}

Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.

Greedy Algorithms (Prim’s Minimum Spanning Tree MST))

It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.
A group of edges that connects two set of vertices in a graph is called cut in graph theory. So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

How does Prim’s Algorithm Work? The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Algorithm
1) Create a set mstSet that keeps track of vertices already included in MST.
2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

3) While mstSet doesn’t include all vertices
   … a) Pick a vertex u which is not there in mstSet and has minimum key value.
   … b) Include u to mstSet.
   … c) Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

Let us understand with the following example:

The set mstSet is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in mstSet. So mstSet becomes {0}. After including to mstSet, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.

Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.
Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes \{0, 1, 7\}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).

Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes \{0, 1, 7, 6\}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.

We repeat the above steps until mstSet doesn’t include all vertices of given graph. Finally, we get the following graph.
**How to implement the above algorithm?**

We use a boolean array `mstSet[]` to represent the set of vertices included in MST. If a value `mstSet[v]` is true, then vertex `v` is included in MST, otherwise not. Array `key[]` is used to store key values of all vertices. Another array `parent[]` to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

// A C / C++ program for Prim's Minimum Spanning Tree (MST) algorithm. // The program is for adjacency matrix representation of the graph

```c
#include <stdio.h>
#include <limits.h>

#define V 5

// A utility function to find the vertex with minimum key value, from // the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed MST stored in parent[]
int printMST(int parent[], int n, int graph[V][V])
{
    printf("Edge   Weight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d    %d    \
", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented using adjacency // matrix representation
void primMST(int graph[V][V])
{
    int parent[V]; // Array to store constructed MST
    int key[V];   // Key values used to pick minimum weight edge in cut
    bool mstSet[V]; // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    key[0] = 0;   // Make key 0 so that this vertex is picked as first vertex
    parent[0] = -1; // First node is always root of MST
```
// The MST will have V vertices
for (int count = 0; count < V - 1; count++)
{
    // Pick the minimum key vertex from the set of vertices
    // not yet included in MST
    int u = minKey(key, mstSet);

    // Add the picked vertex to the MST Set
    mstSet[u] = true;

    // Update key value and parent index of the adjacent vertices of
    // the picked vertex. Consider only those vertices which are not yet
    // included in MST
    for (int v = 0; v < V; v++)
    {
        // graph[u][v] is non zero only for adjacent vertices of m
        // mstSet[v] is false for vertices not yet included in MST
        // Update the key only if graph[u][v] is smaller than key[v]
        if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
    } 

    // print the constructed MST
    printMST(parent, V, graph);
}

// driver program to test above function
int main()
{
    /* Let us create the following graph

    2    3
    (0) --(1) --(2)
      |    / \
     6    8   5    7
      |    /   \
    (3)-------(4)

    */
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0}};

    // Print the solution
    primMST(graph);

    return 0;
}

Output:

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 1</td>
<td>2</td>
</tr>
<tr>
<td>1 - 2</td>
<td>3</td>
</tr>
</tbody>
</table>
Time Complexity of the above program is $O(V^2)$. If the input graph is represented using adjacency list, then the time complexity of Prim’s algorithm can be reduced to $O(E \log V)$ with the help of binary heap. We will soon be discussing $O(E \log V)$ algorithm as a separate post.
Greedy Algorithms (Kruskal’s Minimum Spanning Tree Algorithm)

What is Minimum Spanning Tree?
Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?
A minimum spanning tree has \((V - 1)\) edges where \(V\) is the number of vertices in the given graph.

What are the applications of Minimum Spanning Tree?

Below are the steps for finding MST using Kruskal’s algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are \((V-1)\) edges in the spanning tree.

The step#2 uses Union-Find algorithm to detect cycle. So we recommend to read following post as a prerequisite.
Union-Find Algorithm | Set 1 (Detect Cycle in a Graph)
Union-Find Algorithm | Set 2 (Union By Rank and Path Compression)

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example:

Consider the below input graph.

![Graph Image]

The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having \((9 - 1) = 8\) edges.
After sorting:

<table>
<thead>
<tr>
<th>Weight</th>
<th>Src</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Now pick all edges one by one from sorted list of edges

1. Pick edge 7-6: No cycle is formed, include it.

2. Pick edge 8-2: No cycle is formed, include it.

3. Pick edge 6-5: No cycle is formed, include it.

4. Pick edge 0-1: No cycle is formed, include it.
5. Pick edge 2-5: No cycle is formed, include it.

6. Pick edge 8-6: Since including this edge results in cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.

8. Pick edge 7-8: Since including this edge results in cycle, discard it.

9. Pick edge 0-7: No cycle is formed, include it.

10. Pick edge 1-2: Since including this edge results in cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.

Since the number of edges included equals \( V - 1 \), the algorithm stops here.

// Kruskal's algorithm to find Minimum Spanning Tree of a given connected, // undirected and weighted graph

www.studentsfocus.com
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected and weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges. Since the graph is
    // undirected, the edge from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
```c
int xroot = find(subsets, x);
int yroot = find(subsets, y);

// Attach smaller rank tree under root of high rank tree
// (Union by Rank)
if (subsets[xroot].rank < subsets[yroot].rank)
    subsets[xroot].parent = yroot;
else if (subsets[xroot].rank > subsets[yroot].rank)
    subsets[yroot].parent = xroot;

// If ranks are same, then make one as root and increment
// its rank by one
else
{
    subsets[yroot].parent = xroot;
    subsets[xroot].rank++;
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing order of their weight
    // If we are not allowed to change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V subsets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
```
while (e < V - 1)
{
    // Step 2: Pick the smallest edge. And increment the index
    // for next iteration
    struct Edge next_edge = graph->edge[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    // If including this edge doesn't cause cycle, include it
    // in result and increment the index of result for next edge
    if (x != y)
    {
        result[e++] = next_edge;
        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display the built MST
printf("Following are the edges in the constructed MST\n");
for (i = 0; i < e; ++i)
    printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
return;
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
       
       10
       0------1
       |       |
       6|      5|15
       |       |
       2------3
       4

       */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;

    return 0;
}
graph->edge[2].dest = 3;
graph->edge[2].weight = 5;

// add edge 1-3
graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 15;

// add edge 2-3
graph->edge[4].src = 2;
graph->edge[4].dest = 3;
graph->edge[4].weight = 4;

KruskalMST(graph);

    return 0;
}

Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10

**Time Complexity:** O(ElogE) or O(ElogV). Sorting of edges takes O(ElogE) time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost O(LogV) time. So overall complexity is O(ElogE + ElogV) time. The value of E can be atmost V^2, so O(LogV) are O(LogE) same. Therefore, overall time complexity is O(ElogE) or O(ElogV)