Introduction to Analysis and Design of Algorithms Laboratory

The **Analysis and Design of Algorithms Laboratory** is a practical course designed to complement theoretical studies in the field of algorithms. This lab focuses on the implementation, testing, and evaluation of various algorithms to gain hands-on experience with algorithm design principles and the intricacies of algorithm analysis. The laboratory setting provides students with an opportunity to experiment with real-world problems, analyze algorithmic solutions, and understand the underlying principles of efficiency in computation.

Objective of the Laboratory

The primary objective of the Analysis and Design of Algorithms Laboratory is to enable students to:

1. **Understand the theory** behind algorithmic design techniques such as Divide and Conquer, Greedy algorithms, Dynamic Programming, and Backtracking.
2. **Implement algorithms** in a programming language (such as C++, Java, or Python) and explore their correctness, time complexity, and space complexity.
3. **Analyze and evaluate the performance** of various algorithms by running them on different test cases and understanding their computational efficiency.
4. **Compare the performance** of different algorithms in terms of time complexity, space complexity, and real-world applicability.

Key Concepts Covered in the Laboratory

The laboratory allows students to apply theoretical concepts to real coding problems, focusing on the following areas:

1. **Algorithm Design Strategies**

- **Divide and Conquer**: This strategy involves breaking a problem into smaller subproblems, solving them independently, and then combining their results. The laboratory exercises typically involve the implementation of **Merge Sort**, **Quick Sort**, and **Binary Search**. Students will analyze how this approach improves the efficiency of sorting and searching operations.
- **Greedy Algorithms**: These algorithms make a sequence of choices that are locally optimal at each step. In the laboratory, students will work on problems such as **Activity**

Selection, **Fractional Knapsack**, and **Huffman Coding**. The focus is on the trade-offs between simplicity and optimality.

- **Dynamic Programming (DP)**: DP is used for problems with overlapping subproblems, where previously computed results are stored to avoid redundant work. Students will implement algorithms for problems like **Fibonacci Sequence**, **Knapsack Problem**, and **Longest Common Subsequence**. They will learn how to optimize solutions for problems that would otherwise be inefficient.
- **Backtracking**: This method systematically searches for a solution by trying potential candidates and abandoning them if they don't meet the problem's constraints. In the lab, students will solve problems like the **N-Queens Problem** and **Sudoku Solver**. The emphasis is on exploring all possible solutions and pruning invalid ones early.

2. **Algorithm Analysis**

- **Time Complexity**: Students will learn how to measure and analyze the time taken by different algorithms to execute. They will compare different sorting algorithms like **Bubble Sort** (O(n^2)) and **Quick Sort** (O(n log n)) to understand how the choice of algorithm impacts performance with respect to input size.
- **Space Complexity**: Along with time complexity, space complexity is analyzed to determine how much memory is required by an algorithm. For example, recursive algorithms often use more memory due to function call stacks, and the lab will allow students to compare iterative vs. recursive approaches for various problems.
- **Big O Notation**: The laboratory emphasizes the importance of **Big O** notation in classifying algorithms based on their worst-case or average-case performance. By analyzing different algorithms using Big O, students will gain insights into the scalability of algorithms and their feasibility in large-scale applications.

3. **Practical Implementation and Testing**

- **Problem Solving**: Students will be presented with different algorithmic problems to solve, ranging from sorting and searching to optimization and pathfinding problems. The lab provides a hands-on experience of designing and coding solutions in a programming language of choice.
- **Test Cases and Edge Cases**: A significant part of the laboratory work involves testing the implemented algorithms with various datasets, including worst-case and edge cases.

For example, testing sorting algorithms with nearly sorted data, reverse-sorted data, or random data helps students understand how different inputs affect algorithm performance.

- **Efficiency Comparisons**: The lab exercises often involve running multiple algorithms on the same dataset and comparing their performance. Students will analyze how their algorithms scale with increasing input sizes and how different data structures (like arrays, linked lists, trees, and hash tables) affect performance.

Tools and Techniques Used in the Laboratory

1. **Programming Languages**: The lab typically uses popular programming languages like C, **C++**, **Java**,. These languages are chosen because of their flexibility, ease of use, and availability of libraries to facilitate algorithm implementation.
2. **Complexity Analysis Tools**: Tools such as **Big-O calculators** or **profilers** may be used to measure the runtime and memory usage of algorithms, helping students quantify performance.
3. **Online Coding Platforms**: In some cases, students might use online platforms like **LeetCode**, **HackerRank**, or **Codeforces** to practice algorithmic problems and evaluate their solutions in real-time.

Evaluation and Outcomes

The outcome of the **Analysis and Design of Algorithms Laboratory** is not just the completion of algorithmic problems but the development of skills that can be used to analyze and optimize real-world systems. Key takeaways include:

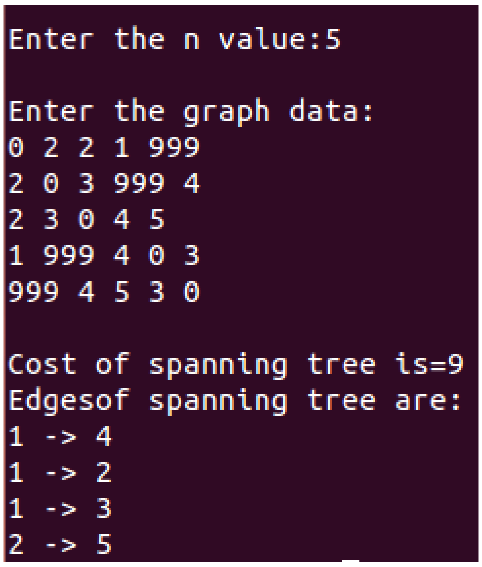
- **Problem-Solving Skills**: Students will become proficient in designing and implementing algorithms that solve complex problems efficiently.
- **Analytical Skills**: Students will develop the ability to assess an algorithm's performance and determine the most suitable approach for different problems based on time and space considerations.
- **Critical Thinking**: The laboratory provides an opportunity to think critically about problem decomposition and the trade-offs between different algorithms and design strategies.
- **Implementation Skills**: By working through multiple algorithms and data structures, students will improve their programming skills and learn how to write clean, efficient, and maintainable code.

1. **Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.**

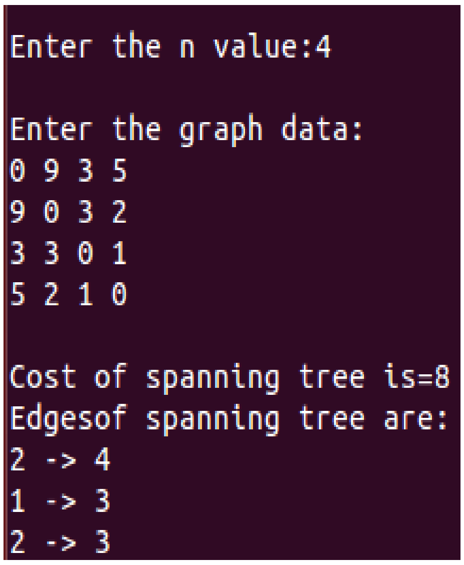
Program:

```
#include<stdio.h>
#define INF 999
#define MAX 100
int p[MAX], c[MAX][MAX], t[MAX][2];
int find(int v)
{
        while (p[v])
        v = p[v];
        return v;
}
void union1(int i, int j)
{
        p[j] = i;
}
void kruskal(int n)
{
        int i, j, k, u, v, min, res1, res2, sum = 0;
        for (k = 1; k < n; k++)
        {
                min = INF;
                for (i = 1; i < n - 1; i++)
                {
                        for (j = 1; j <= n; j++)
                        {
                                if (i == j) continue;
                                if (c[i][j] < min)
                                {
                                        u = find(i);
                                        v = find(j);
                                        if (u != v)
                                        {
                                                res1 = i;
                                                res2 = j;
                                                min = c[i][j];
                                        }
                                }
                        }
                }
        union1(res1, find(res2));
        t[k][1] = res1;
        t[k][2] = res2;
        sum = sum + min;
        }
        printf("\nCost of spanning tree is=%d", sum);
        printf("\nEdgesof spanning tree are:\n");
        for (i = 1; i < n; i++)
        printf("%d -> %d\n", t[i][1], t[i][2]);
}
int main()
{
        int i, j, n;
        printf("\nEnter the n value:");
        scanf("%d", & n);
        for (i = 1; i <= n; i++)
        p[i] = 0;
        printf("\nEnter the graph data:\n");
        for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
        scanf("%d", & c[i][j]);
        kruskal(n);
        return 0;
}
```

Output:



```
Enter the n value:5

Enter the graph data:
0 2 2 1 999
2 0 3 999 4
2 3 0 4 5
1 999 4 0 3
999 4 5 3 0

Cost of spanning tree is=9
Edgesof spanning tree are:
1 -> 4
1 -> 2
1 -> 3
2 -> 5
```



```
Enter the n value:4

Enter the graph data:
0 9 3 5
9 0 3 2
3 3 0 1
5 2 1 0

Cost of spanning tree is=8
Edgesof spanning tree are:
2 -> 4
1 -> 3
2 -> 3
```

Explanation:

This program implements **Kruskal's Algorithm** to find the **Minimum Spanning Tree (MST)** of a connected, undirected graph. The graph is represented as an adjacency matrix, where each element indicates the weight (cost) of the edge between two vertices. Below is a detailed explanation of the program:

❖ **Kruskal's Algorithm**: This is a greedy algorithm used for finding the Minimum Spanning Tree (MST) of a graph. The algorithm follows these steps:
  - Sort all the edges in the graph based on their weights.
  - Add edges to the MST in increasing order of weight, ensuring that no cycles are formed (using a union-find data structure to check for cycles).

❖ **#include<stdio.h>**: Includes the standard input/output library for using functions like scanf() and printf().

❖ **#define INF 999**: Defines a constant INF to represent an infinite or large weight. In the context of graph representation, INF is used to indicate the absence of an edge between nodes.

❖ **#define MAX 100**: Defines a constant MAX representing the maximum number of nodes in the graph (100 nodes in this case).

- **int p[MAX]**: This is an array that will hold the parent of each vertex during the union-find process. Initially, each vertex is its own parent.
- **int c[MAX][MAX]**: This 2D array represents the adjacency matrix of the graph, where c[i][j] holds the weight of the edge between vertex i and vertex j. If there's no edge, it might hold INF or some large value.
- **int t[MAX][2]**: This 2D array will store the edges of the minimum spanning tree (MST). Each row in t will contain two integers: the two vertices that form an edge in the MST.
- **int find(int v)**: This function finds the root of the set that contains vertex v (with path compression). It returns the representative (root) of the set.
- **if (p[v] == 0)**: This checks if vertex v is its own parent, meaning it's the root of its set.
- **p[v] = find(p[v]);**: This performs path compression. Instead of returning the parent directly, it recursively finds the parent and sets the parent of each vertex along the path directly to the root, making future find() calls faster.
- **return p[v];**: This returns the root of the set that vertex v belongs to
- **void union1(int i, int j)**: This function performs a union operation, merging the sets containing vertices i and j. After this operation, vertex j will be part of vertex i's set.
- **p[j] = i;**: This sets the parent of vertex j to be vertex i, effectively joining the two sets
- **void kruskal(int n)**: This function implements Kruskal's algorithm. It takes n, the number of vertices, as an argument.
- **int i, j, k, u, v, min, res1, res2, sum = 0;**: Declares various variables:
- i, j, k are loop indices.
- u, v store the representative (root) of the two vertices being checked.
- min stores the minimum edge weight found in each iteration.
- res1, res2 store the vertices of the minimum weight edge that will be added to the MST.
- sum is the total weight of the MST.
- **for (k = 1; k < n; k++)**: This loop iterates over each edge that will be added to the MST. The MST will have n - 1 edges.
- **min = INF;**: Initializes min to INF at the start of each iteration.
- **for (i = 1; i <= n; i++) and for (j = 1; j <= n; j++)**: These nested loops iterate over all pairs of vertices in the graph to find the minimum edge.
- **if (i == j) continue;**: Skips checking the edge from a vertex to itself.
- **if (c[i][j] < min)**: If the weight of the edge between vertices i and j is smaller than the current min, it updates the minimum edge.
- **u = find(i); v = find(j);**: Finds the root (representative) of the sets containing vertices i and j using

the find() function.

- **if (u != v)**: If the vertices i and j are not in the same set (i.e., they are not yet connected in the MST), the edge is a valid candidate for the MST.
- **res1 = i; res2 = j; min = c[i][j];**: Stores the vertices i and j as the minimum edge, and the edge weight min.
- **union1(res1, find(res2));**: Joins the sets of res1 and res2 by setting the parent of res2 to res1, indicating that the edge between res1 and res2 is part of the MST.
- **t[k][0] = res1; t[k][1] = res2;**: Stores the edge in the t array, indicating that the edge from res1 to res2 is part of the MST.
- **sum = sum + min;**: Adds the weight of the selected edge to the total weight sum.
- **printf("\nCost of spanning tree is=%d", sum);**: Prints the total weight of the MST.
- **printf("\nEdges of spanning tree are:\n");**: Prints a header for the edges.
- **for (i = 1; i < n; i++)**: Loops through the t array to print each edge in the MST.
- **printf("%d -> %d\n", t[i][0], t[i][1]);**: Prints each edge in the MST.
- **int main()**: The main function where the program starts execution.
- **int i, j, n;**: Declares the loop indices and the number of vertices n.
- **printf("\nEnter the number of vertices in the graph: ");**: Prompts the user to input the number of vertices in the graph.
- **scanf("%d", &n);**: Reads the number of vertices from the user.
- **for (i = 1; i <= n; i++) { p[i] = 0; }**: Initializes the parent array p to 0 for each vertex, indicating that initially, each vertex is its own parent.
- **printf("\nEnter the graph adjacency matrix:\n");**: Prompts the user to input the adjacency matrix of the graph.
- **for (i = 1; i <= n; i++) and for (j = 1; j <= n; j++)**: Loops to read the adjacency matrix values.
- **scanf("%d", &c[i][j]);**: Reads the weight of the edge between vertices i and j into the adjacency matrix c.
- **kruskal(n);**: Calls the kruskal function to find the MST of the graph.
- **return 0;**: Exits the program with a return code of 0, indicating successful execution.

2. Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.

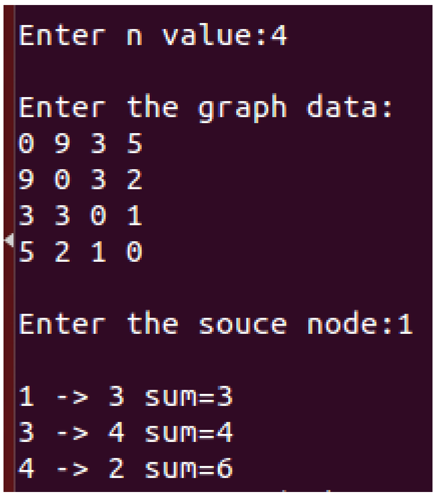
Program:

```
#include<stdio.h>

#define INF 999

int prim(int c[10][10],int n,int s)

{

   int v[10],i,j,sum=0,ver[10],d[10],min,u;

   for(i=1; i<=n; i++)

   {

      ver[i]=s;

      d[i]=c[s][i];

      v[i]=0;

   }

   v[s]=1;

   for(i=1; i<=n-1; i++)

   {

      min=INF;

      for(j=1; j<=n; j++)

      if(v[j]==0 && d[j]<min)

         {

            min=d[j];
```

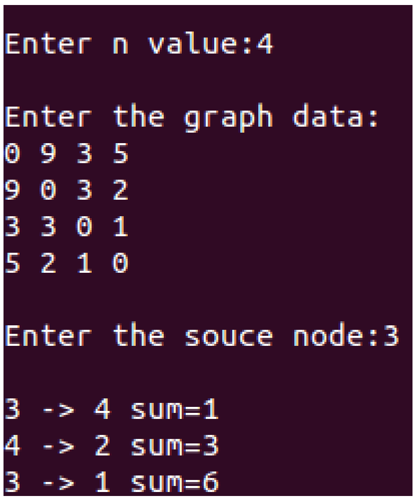
```
                u=j;

            }

        v[u]=1;

        sum=sum+d[u];

        printf("\n%d -> %d sum=%d",ver[u],u,sum);

        for(j=1; j<=n; j++)

            if(v[j]==0 && c[u][j]<d[j])

            {

                d[j]=c[u][j];

                ver[j]=u;

            }

    }

    return sum;

}

int main()

{

    int c[10][10],i,j,res,s,n;

    printf("\nEnter n value:");

    scanf("%d",&n);

    printf("\nEnter the graph data:\n");

    for(i=1; i<=n; i++)

    for(j=1; j<=n; j++)
```

```
        scanf("%d",&c[i][j]);

        printf("\nEnter the souce node:");

        scanf("%d",&s);

        res=prim(c,n,s);

        printf("\nCost=%d",res);

        return 0;

    }
```

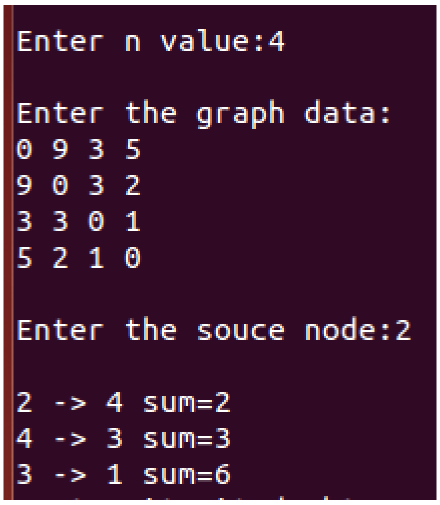
Output:



```
Enter n value:4

Enter the graph data:
0 9 3 5
9 0 3 2
3 3 0 1
5 2 1 0

Enter the souce node:1

1 -> 3 sum=3
3 -> 4 sum=4
4 -> 2 sum=6
```



```
Enter n value:4

Enter the graph data:
0 9 3 5
9 0 3 2
3 3 0 1
5 2 1 0

Enter the souce node:3

3 -> 4 sum=1
4 -> 2 sum=3
3 -> 1 sum=6
```



```
Enter n value:4

Enter the graph data:
0 9 3 5
9 0 3 2
3 3 0 1
5 2 1 0

Enter the souce node:2

2 -> 4 sum=2
4 -> 3 sum=3
3 -> 1 sum=6
```

Explanation:

This program implements **Prim's Algorithm** for finding the **Minimum Spanning Tree (MST)** of a connected, undirected graph. Prim's algorithm is a greedy algorithm that builds the MST by growing it one edge at a time, starting from an arbitrary node and always adding the smallest edge that connects a vertex in the MST to a vertex outside it.

❖ **Header and Definitions**

- #include<stdio.h>: Includes the standard input-output library to enable usage of printf() and scanf().
- #define INF 999: Defines a constant INF to represent a large value that acts as an "infinity" when comparing edge weights.

❖ Prim's Algorithm Function

- **int prim(int c[10][10], int n, int s)**: This is the implementation of Prim's algorithm, where:
  - c[10][10]: Adjacency matrix representing the graph's edge weights.
  - n: Number of vertices in the graph.
  - s: The source vertex where the algorithm starts.

- **int v[10], i, j, sum = 0, ver[10], d[10], min, u;**:
  - v[10]: An array that marks whether a vertex is included in the MST (1 if included, 0 if not).
  - ver[10]: Stores the vertex that was added before the current vertex in the MST (i.e., parent vertex).
  - d[10]: Stores the minimum distance to the MST for each vertex. This value is updated as the algorithm progresses.
  - min: Variable to store the minimum weight edge found in the loop.
  - u: Stores the vertex with the smallest edge weight to add to the MST.

❖ Initialization of Arrays

- Loop over all vertices:
  - ver[i] = s;: Initializes the ver array to store the source vertex for all other vertices.
  - d[i] = c[s][i];: Sets the minimum distance d[i] for each vertex as the weight of the edge between the source vertex s and the other vertex i. If no edge exists, c[s][i] will be a large value.
  - v[i] = 0;: Marks all vertices as not yet included in the MST (set v[i] to 0).
- v[s] = 1;: Marks the source vertex s as part of the MST.

❖ Main Loop: Selecting the Minimum Edge

- for (i = 1; i <= n - 1; i++): Loop runs for n-1 iterations because a tree with n vertices has n-1 edges.
- min = INF;: Initializes min to a large value (INF) for each iteration.
- Inner loop for (j = 1; j <= n; j++): Iterates through each vertex j.
  - if (v[j] == 0 && d[j] < min): Checks if vertex j is not yet in the MST (v[j] == 0) and if the current minimum distance d[j] is smaller than the current min.
  - min = d[j]; u = j;: Updates the min value with d[j] and stores vertex j as u, the vertex with the smallest edge weight.

❖ **Add Vertex to MST**

- v[u] = 1;: Adds vertex u to the MST by marking it as visited.
- sum = sum + d[u];: Adds the weight of the selected edge (d[u]) to the total sum.
- printf("\n%d -> %d sum=%d", ver[u], u, sum);: Prints the edge added to the MST (ver[u] -> u) and the running sum of the MST's weight.

❖ **Update Distances**

- **Loop over all vertices j**: Checks each vertex j that is not yet in the MST (v[j] == 0).
- **if (v[j] == 0 && c[u][j] < d[j])**: If vertex j is not in the MST and the weight of the edge from u to j (c[u][j]) is smaller than the current minimum distance (d[j]), it updates the distance.
- **d[j] = c[u][j];**: Updates the minimum distance for vertex j.
- **ver[j] = u;**: Updates the parent of vertex j to u, meaning vertex u is the vertex that connects j to the MST.

❖ **Return the Total Cost**

- return sum;: After the loop finishes, it returns the total weight (sum) of the Minimum Spanning Tree (MST).

❖ **Main Function**

- int c[10][10], i, j, res, s, n;: Declares necessary variables. c[10][10] is the adjacency matrix representing the graph.
- printf("\nEnter n value:");: Prompts the user to enter the number of vertices (n).
- scanf("%d", &n);: Reads the number of vertices from the user.
- **printf("\nEnter the graph data:\n");**: Asks the user to input the graph's adjacency matrix.
- **for (i = 1; i <= n; i++) and for (j = 1; j <= n; j++)**: Loops to read the adjacency matrix.
- **scanf("%d", &c[i][j]);**: Reads the weight of the edge between vertices i and j.
- **printf("\nEnter the source node:");**: Prompts the user to enter the source vertex (s) from where the MST will be generated.
- **scanf("%d", &s);**: Reads the source vertex from the user.
- **res = prim(c, n, s);**: Calls the prim() function with the adjacency matrix c, the number of vertices n, and the source vertex s, and stores the result (total cost) in res.
- **printf("\nCost=%d", res);**: Prints the total cost of the MST.
- **return 0;**: Exits the program with a success code (0).

3. **a. Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm.**

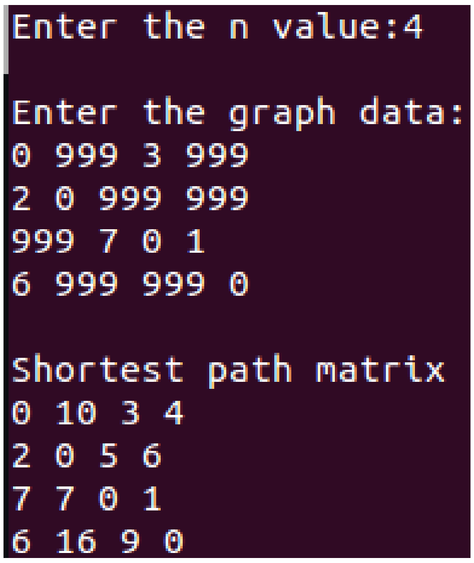
   b. Design and implement C/C++ Program to find the transitive closure using Warshal's algorithm.

Program - a:

```
#include<stdio.h>
#define INF 999
int min(int a,int b)
{
   return(a<b)?a:b;
}
void floyd(int p[][10],int n)
{
   int i,j,k;
   for(k=1; k<=n; k++)
   for(i=1; i<=n; i++)
   for(j=1; j<=n; j++)
   p[i][j]=min(p[i][j],p[i][k]+p[k][j]);
}
int main()
{
   int a[10][10],n,i,j;
   printf("\nEnter the n value:");
   scanf("%d",&n);
   printf("\nEnter the graph data:\n");
   for(i=1; i<=n; i++)
   for(j=1; j<=n; j++)
   scanf("%d",&a[i][j]);
   floyd(a,n);
   printf("\nShortest path matrix\n");
```

```
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n; j++)
        printf("%d ",a[i][j]);
        printf("\n");
    }
    return 0;
}
```

Output:



```
Enter the n value:4

Enter the graph data:
0 999 3 999
2 0 999 999
999 7 0 1
6 999 999 0

Shortest path matrix
0 10 3 4
2 0 5 6
7 7 0 1
6 16 9 0
```

Explanation:

This C program implements the Floyd-Warshall algorithm for finding the shortest paths between all pairs of nodes in a weighted graph.

❖ **Includes and Definitions**

- **#include<stdio.h>**: Includes the standard input/output library for using functions like scanf() and printf().
- **#define INF 999**: Defines a constant INF to represent an infinitely large distance. This is used to indicate the absence of a direct edge between two nodes in the graph.
- **int min(int a, int b)**: A helper function that returns the minimum of two integers a and b. This is used to update the shortest distance during the Floyd-Warshall algorithm.
- **return (a < b) ? a : b;**: The ternary operator is used to return the smaller value between a and

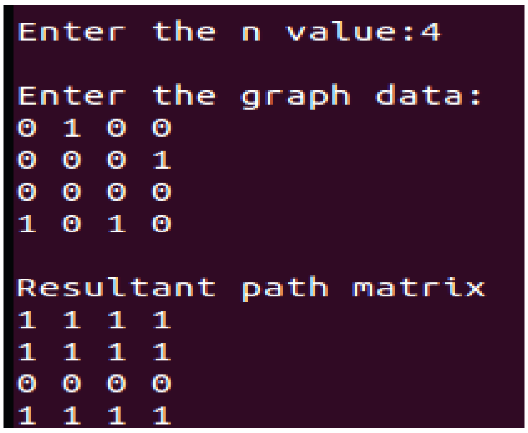
b.

- **void floyd(int p[][10], int n)**: This is the function implementing the Floyd-Warshall algorithm. It takes two arguments:
  - p[][10]: A 2D array representing the graph's distance matrix (adjacency matrix). The matrix p[i][j] stores the shortest distance from node i to node j.
  - n: The number of vertices in the graph.
- **Triple nested loops**:
  - The outer loop (for (k = 1; k <= n; k++)) selects each intermediate node k in the graph.
  - The two inner loops (for (i = 1; i <= n; i++) and for (j = 1; j <= n; j++)) are used to iterate through all pairs of nodes (i, j).
- **p[i][j] = min(p[i][j], p[i][k] + p[k][j]);**:
  - This updates the shortest path from i to j by checking if a path through k is shorter than the current known shortest path. If p[i][k] + p[k][j] is shorter, the value of p[i][j] is updated with this new shorter distance.
  - **int a[10][10], n, i, j;**: Declares a 2D array a[10][10] to store the adjacency matrix (graph data) and other variables n, i, and j.
  - **printf("\nEnter the n value:");**: Asks the user to input the number of vertices (n) in the graph.
  - **scanf("%d", &n);**: Reads the value of n from the user.
  - **printf("\nEnter the graph data:\n");**: Prompts the user to input the graph's adjacency matrix.
  - **for (i = 1; i <= n; i++)** and **for (j = 1; j <= n; j++)**: These loops iterate through all elements of the a array (the adjacency matrix) and ask the user to input the weights for each edge.
  - a[i][j] stores the distance (or weight) of the edge between node i and node j.
  - If there's no edge between two nodes, the weight should be set to INF (or a very large value).
  - **`floyd(a, n);`**: Calls the `floyd()` function to calculate the shortest paths for all pairs of nodes in the graph. The adjacency matrix `a` is updated to contain the shortest path distances after this call.
  - **return 0;**: Returns 0 to indicate that the program executed successfully.

Program-b:

```
#include<stdio.h>
void warsh(int p[][10],int n)
{
   int i,j,k;
   for(k=1; k<=n; k++)
   for(i=1; i<=n; i++)
   for(j=1; j<=n; j++)
   p[i][j]=p[i][j] || p[i][k] && p[k][j];
}
int main()
{
   int a[10][10],n,i,j;
   printf("\nEnter the n value:");
   scanf("%d",&n);
   printf("\nEnter the graph data:\n");
   for(i=1; i<=n; i++)
   for(j=1; j<=n; j++)
   scanf("%d",&a[i][j]);
   warsh(a,n);
   printf("\nResultant path matrix\n");
   for(i=1; i<=n; i++)
   {
      for(j=1; j<=n; j++)
      printf("%d ",a[i][j]);
      printf("\n");
   }
   return 0;
}
```

Output:



```
Enter the n value:4

Enter the graph data:
0 1 0 0
0 0 0 1
0 0 0 0
1 0 1 0

Resultant path matrix
1 1 1 1
1 1 1 1
0 0 0 0
1 1 1 1
```

Explanation:

❖ Warshall’s Algorithm Function: warsh

Warshall's Algorithm finds the transitive closure of a graph. The idea behind it is that for each pair of nodes (i, j), we check if there is a path from i to j through any intermediate node k.

❖ Main Function: main

Input Section:

- **#include<stdio.h>**: Includes the standard input/output library to use printf() and scanf() for taking inputs and displaying outputs.

- **void warsh(int p[][10], int n)**: This function implements Warshall's algorithm to compute the transitive closure of the graph.
  - **Parameters**:
    - p[][10]: A 2D matrix representing the adjacency matrix of the graph (where p[i][j] is 1 if there is a direct path from vertex i to vertex j, and 0 otherwise).
    - n: The number of vertices in the graph.
- <u>**Triple nested loops**</u>:
  - The outer loop (for(k = 1; k <= n; k++)) iterates over all possible intermediate vertices k.
  - The middle loop (for(i = 1; i <= n; i++)) iterates over the starting vertices i.
  - The inner loop (for(j = 1; j <= n; j++)) iterates over the ending vertices j.
- **p[i][j] = p[i][j] || (p[i][k] && p[k][j]);**: This is the key line of Warshall's algorithm. It updates the matrix p[i][j] to indicate whether there is a path from vertex i to vertex j through some intermediate vertex k. It works as follows:

- **p[i][j]**: This is the current value (whether there's a direct path from i to j).
- **p[i][k]**: This checks if there's a path from i to k.
- **p[k][j]**: This checks if there's a path from k to j.
- The logical OR (||) means that if there is a path either directly from i to j or via k, p[i][j] is updated to 1.

Main Function:

- **int a[10][10], n, i, j;**: Declares a 2D array a[10][10] to represent the graph's adjacency matrix, as well as variables n, i, and j.
- **printf("\nEnter the n value:");**: Prompts the user to input the number of vertices n in the graph.
- **scanf("%d", &n);**: Reads the number of vertices n from the user.
- **printf("\nEnter the graph data:\n");**: Prompts the user to input the adjacency matrix for the graph.
- **for(i = 1; i <= n; i++)** and **for(j = 1; j <= n; j++)**: These loops iterate through the adjacency matrix a and allow the user to input values. The values should be either 0 (no path) or 1 (direct path between vertices i and j).

warsh(a, n);: Calls the warsh() function to calculate the transitive closure of the graph. After this call, the matrix a is updated to reflect whether there is a path between any two vertices.

- P**rintf("\nResultant path matrix\n");**: Prints a message to indicate the result.
- **for(i = 1; i <= n; i++)**: Loops through each row of the adjacency matrix.
- **for(j = 1; j <= n; j++)**: Loops through each column of the adjacency matrix.
- **printf("%d ", a[i][j]);**: Prints the value at a[i][j], which now represents whether there is a path between node i and node j after applying Warshall's algorithm.
- **printf("\n");**: Prints a newline after each row to format the output.
- **return 0;**: Returns 0, indicating that the program executed successfully.

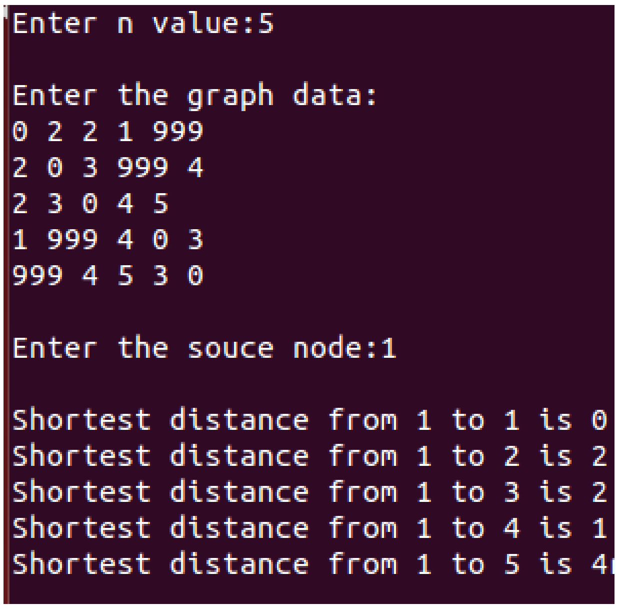
4. Design and implement C/C++ Program to find shortest paths from a given vertex in a weighted connected graph to other vertices using Dijkstra's algorithm.

Program:

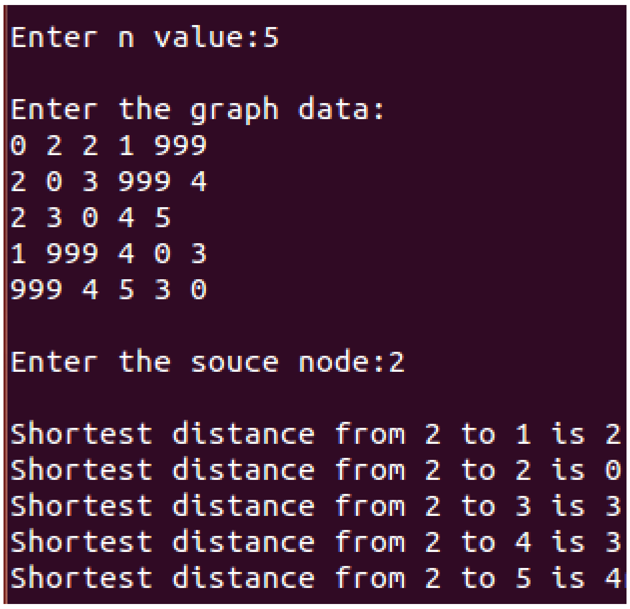
```
#include<stdio.h>
#define INF 999
void dijkstra(int c[10][10],int n,int s,int d[10])
{
   int v[10],min,u,i,j;
   for(i=1; i<=n; i++)
   {
      d[i]=c[s][i];
      v[i]=0;
   }
   v[s]=1;
   for(i=1; i<=n; i++)
   {
      min=INF;
      for(j=1; j<=n; j++)
         if(v[j]==0 && d[j]<min)
         {
            min=d[j];
            u=j;
         }
      v[u]=1;
      for(j=1; j<=n; j++)
      if(v[j]==0 && (d[u]+c[u][j])<d[j])
      d[j]=d[u]+c[u][j];
   }
}
```

```
int main()
{
    int c[10][10],d[10],i,j,s,sum,n;
    printf("\nEnter n value:");
    scanf("%d",&n);
    printf("\nEnter the graph data:\n");
    for(i=1; i<=n; i++)
    for(j=1; j<=n; j++)
    scanf("%d",&c[i][j]);
    printf("\nEnter the souce node:");
    scanf("%d",&s);
    dijkstra(c,n,s,d);
    for(i=1; i<=n; i++)
    printf("\nShortest distance from %d to %d is %d",s,i,d[i]);
    return 0;
}
```

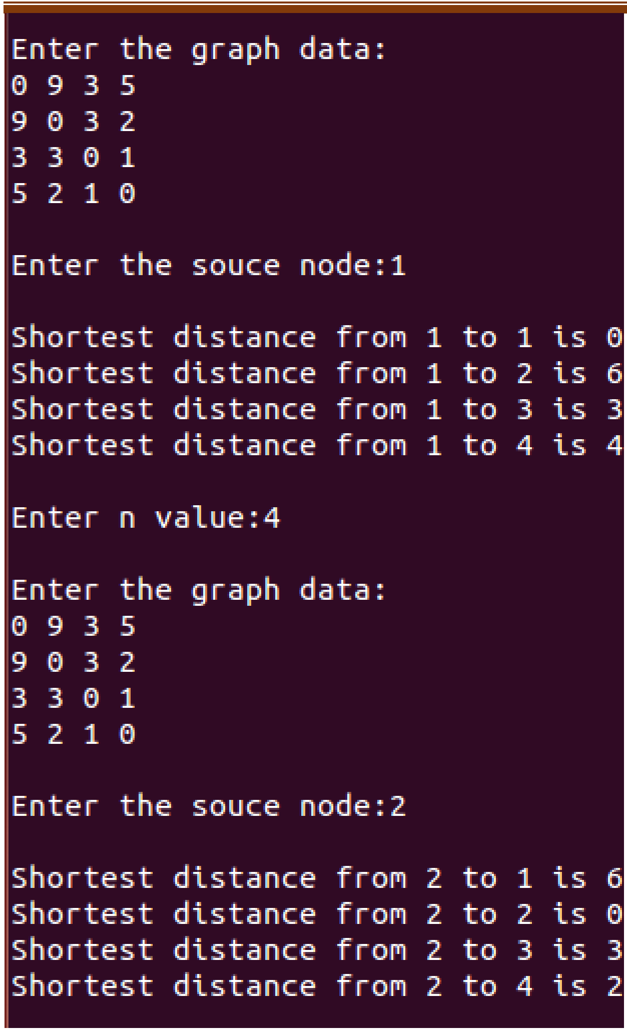
Output:



```
Enter n value:5

Enter the graph data:
0 2 2 1 999
2 0 3 999 4
2 3 0 4 5
1 999 4 0 3
999 4 5 3 0

Enter the souce node:1

Shortest distance from 1 to 1 is 0
Shortest distance from 1 to 2 is 2
Shortest distance from 1 to 3 is 2
Shortest distance from 1 to 4 is 1
Shortest distance from 1 to 5 is 4
```



```
Enter n value:5

Enter the graph data:
0 2 2 1 999
2 0 3 999 4
2 3 0 4 5
1 999 4 0 3
999 4 5 3 0

Enter the souce node:2

Shortest distance from 2 to 1 is 2
Shortest distance from 2 to 2 is 0
Shortest distance from 2 to 3 is 3
Shortest distance from 2 to 4 is 3
Shortest distance from 2 to 5 is 4
```



```
Enter the graph data:
0 9 3 5
9 0 3 2
3 3 0 1
5 2 1 0

Enter the souce node:1

Shortest distance from 1 to 1 is 0
Shortest distance from 1 to 2 is 6
Shortest distance from 1 to 3 is 3
Shortest distance from 1 to 4 is 4

Enter n value:4

Enter the graph data:
0 9 3 5
9 0 3 2
3 3 0 1
5 2 1 0

Enter the souce node:2

Shortest distance from 2 to 1 is 6
Shortest distance from 2 to 2 is 0
Shortest distance from 2 to 3 is 3
Shortest distance from 2 to 4 is 2
```

Explanation:

This C program implements **Dijkstra's Algorithm**, which is used to find the **shortest path** from a **single source node** to all other nodes in a **weighted graph**. Dijkstra's algorithm is commonly used in routing and navigation systems.

❖ Includes and Definitions

- **`#include<stdio.h>`**: This is the standard input/output library, which is required for functions like `printf()` and `scanf()`.
- **`#define INF 999`**: This defines a constant `INF` (infinity) as `999`. This value is used to represent unreachable nodes, i.e., when no path exists between two nodes.

❖ **Dijkstra Function: `Dijkstra`**

Input Parameters:

- **#include<stdio.h>**: Includes the standard input/output library for input and output operations.
- **#define INF 999**: Defines a constant INF (infinity) to represent unreachable nodes or large values for initial comparison. It is set to 999 here, but could be a larger number in practice for bigger graphs.

Dijkstra Function:

- **void dijkstra(int c[10][10], int n, int s, int d[10])**: This is the function that implements Dijkstra's algorithm.

- **Parameters**:
    - c[10][10]: The adjacency matrix representing the graph where c[i][j] holds the weight of the edge between nodes i and j.
    - n: The number of nodes in the graph.
    - s: The source node from which shortest paths are calculated.
    - d[10]: The array where the shortest distances from the source node s to all other nodes will be stored.

- **Initialization**:

- This loop initializes the d[] array with the direct distances from the source node s to all other nodes, and the v[] array to mark all nodes as unvisited (except for the source node).

- **Main Loop**:
    - **First inner loop (for(j=1; j<=n; j++))**:
        - Finds the unvisited node u with the smallest distance (min).
        - d[j] stores the current shortest distance from the source to node j. We find the smallest of these distances for an unvisited node and set it as the next node to process.
    - **Mark the node u as visited (v[u] = 1)**:
        - This means we've finished processing the node u.
    - **Second inner loop (for(j=1; j<=n; j++))**:
        - For all unvisited nodes, we check if going through node u offers a shorter path. If so, we update the distance to that node.

Main Function:

int c[10][10], d[10], i, j, s, sum, n;: Declares variables and arrays:

- c[10][10]: The adjacency matrix of the graph.
- d[10]: The array for storing shortest distances from the source node.
- i, j: Loop variables.
- s: The source node from which shortest paths are computed.
- sum: Not used in this code but might be intended for calculating total distance or sum of shortest paths.
- n: Number of nodes in the graph.

- **printf("\nEnter the graph data:\n");**: Prompts the user to enter the graph data.

- **for(i=1; i<=n; i++) and for(j=1; j<=n; j++)**: These nested loops take input from the user to fill the adjacency matrix c[i][j].
   - **printf("\nEnter the source node:");**: Prompts the user to input the source node.
   - **scanf("%d", &s);**: Takes the source node as input.
- **dijkstra(c, n, s, d);**: Calls the dijkstra function to calculate the shortest distances from the source node s to all other nodes.
   - **for(i=1; i<=n; i++)**: Loops through all the nodes and prints the shortest distance from the source node s to node i.
   - **return 0;**: Ends the program.

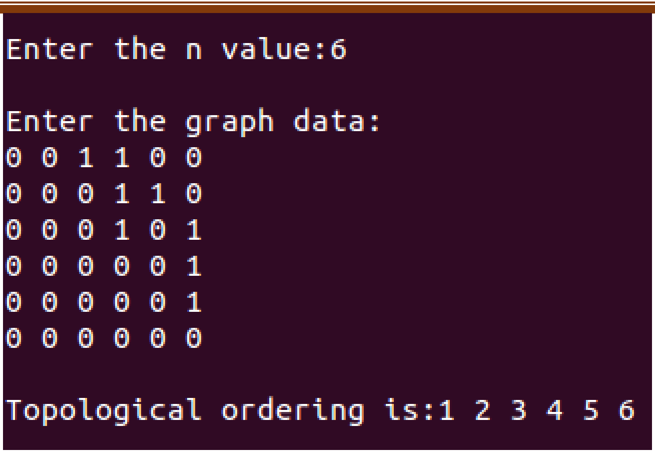
5. Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.

Program:

```
#include<stdio.h>
int temp[10],k=0;
void sort(int a[][10],int id[],int n)
{
   int i,j;
   for(i=1; i<=n; i++)
   {
      if(id[i]==0)
      {
         id[i]=-1;
         temp[++k]=i;
         for(j=1; j<=n; j++)
         {
            if(a[i][j]==1 && id[j]!=-1)
               id[j]--;
         }
         i=0;
      }
```

```
    }
}
int main()
{
    int a[10][10],id[10],n,i,j;
    printf("\nEnter the n value:");
    scanf("%d",&n);
    for(i=1; i<=n; i++)
    id[i]=0;
    printf("\nEnter the graph data:\n");
    for(i=1; i<=n; i++)
        for(j=1; j<=n; j++)
        {
            scanf("%d",&a[i][j]);
            if(a[i][j]==1)
            id[j]++;
        }
    sort(a,id,n);
    if(k!=n)
    printf("\nTopological ordering not possible");
    else
    {
        printf("\nTopological ordering is:");
        for(i=1; i<=k; i++)
        printf("%d ",temp[i]);
    }
 return 0;
}
```

Output:



```
Enter the n value:6

Enter the graph data:
0 0 1 1 0 0
0 0 0 1 1 0
0 0 0 1 0 1
0 0 0 0 0 1
0 0 0 0 0 1
0 0 0 0 0 0

Topological ordering is:1 2 3 4 5 6
```

Explanation:

- This C program implements a **Topological Sort** algorithm for a **Directed Acyclic Graph (DAG)**. Topological sorting is used to order the nodes (or vertices) of a directed graph in a linear sequence such that for every directed edge u → v, node u comes before node v in the ordering. This algorithm is especially useful in scheduling problems or tasks that have dependencies.

- Includes and Global Variables
- This includes the standard input-output library, which is needed for functions like printf() and scanf().
- Declares an array temp to store the topological order of nodes. It's sized 10, which assumes the graph will not have more than 10 nodes.
- k is an integer initialized to 0 and will be used to track how many nodes have been added to the topological order (temp array).
- Defines a function sort() that will perform the topological sorting. The function takes:
- a[][10]: A 2D adjacency matrix representing the graph.
- id[]: An array that tracks the incoming degree (number of edges pointing to) each node.
- n: The number of nodes in the graph.
- Declares two integer variables i and j for use in the loops inside the sort() function.
- Starts a loop that will iterate over all the nodes (from 0 to n-1), which represent the nodes in the graph.
- Inside the loop, checks if the current node i has no incoming edges. This is determined by id[i] == 0, meaning the node can be processed (no dependencies).
- Marks the node i as processed by setting id[i] to -1.
- Adds the node i to the temp array (which stores the topological order).

- Since the problem expects the node numbering to start from 1, we add 1 to i before storing it in temp.
- Increments k after the assignment, indicating that one more node has been added to the order.
- Starts an inner loop that goes through all the nodes j to check if node i has directed edges to any other nodes.
- Checks if there is an edge from node i to node j (i.e., a[i][j] == 1).
- Also checks if id[j] is not -1, meaning node j has not been processed yet.
- If there is an edge from i to j, it decreases the incoming degree (id[j]) of node j by 1.
- This is because we've "processed" node i, which reduces the number of dependencies (incoming edges) for node j.
- Resets i to -1, forcing the outer loop to restart from the beginning. This is to check if any new nodes have become available for processing (i.e., if their incoming degree has become 0).
- Ends the sort() function.
-

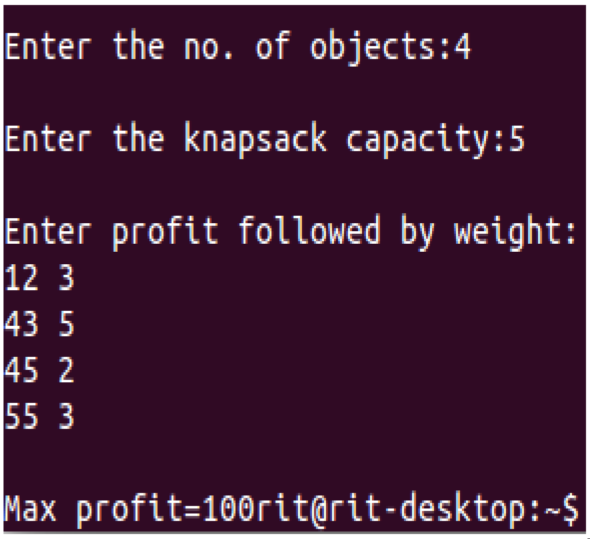
- **main function explanation**
- Declares variables:
- a[10][10]: A 2D array (adjacency matrix) to represent the graph.
- id[10]: An array to track the number of incoming edges (degree) for each node.
- n: An integer to store the number of nodes in the graph.
- i, j: Loop variables.
- Prompts the user to enter the number of nodes in the graph and stores the value in n.
- Initializes the id array. All nodes are initially assumed to have zero incoming edges.
- Prompts the user to enter the adjacency matrix (a[][]) that defines the directed edges between nodes.
- If there is an edge from node i to node j (i.e., a[i][j] == 1), the incoming degree of node j (id[j]) is incremented.
- Calls the sort() function to perform the topological sort on the graph. This function will populate the temp array with the topologically sorted nodes.
- Checks if k (the number of nodes processed) is equal to n (the total number of nodes).
- If k is not equal to n, it means that not all nodes could be processed, which implies that the graph has a cycle and topological sorting is not possible.
- If all nodes were successfully processed (i.e., there was no cycle), it prints the topological order stored in the temp array.
- Returns 0 from main() to indicate the program executed successfully.

6. Design and implement C/C++ Program to solve 0/1 Knapsack problem using Dynamic Programming method.

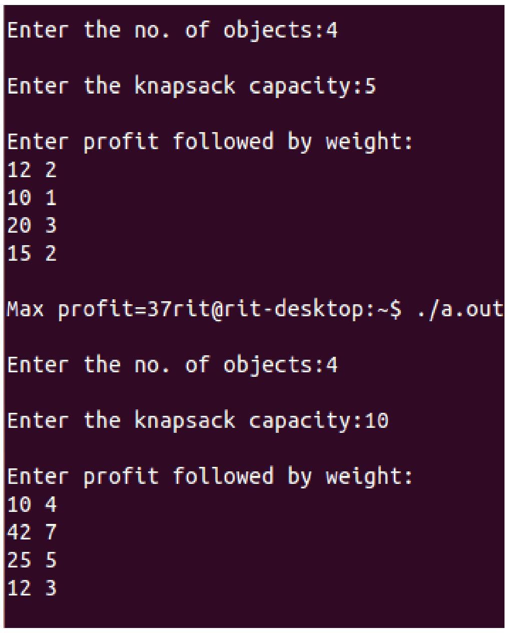
Program:

```
#include<stdio.h>
int w[10],p[10],n;
int max(int a,int b)
{
   return a>b?a:b;
}
int knap(int i,int m)
{
   if(i==n) return w[i]>m?0:p[i];
   if(w[i]>m) return knap(i+1,m);
   return max(knap(i+1,m),knap(i+1,m-w[i])+p[i]);
}
int main()
{
   int m,i,max_profit;
   printf("\nEnter the no. of objects:");
   scanf("%d",&n);
   printf("\nEnter the knapsack capacity:");
   scanf("%d",&m);
   printf("\nEnter profit followed by weight:\n");
   for(i=1; i<=n; i++)
   scanf("%d %d",&p[i],&w[i]);
   max_profit=knap(1,m);
   printf("\nMax profit=%d",max_profit);
   return 0;
}
```

Output:



```
Enter the no. of objects:4

Enter the knapsack capacity:5

Enter profit followed by weight:
12 3
43 5
45 2
55 3

Max profit=100rit@rit-desktop:~$
```



```
Enter the no. of objects:4

Enter the knapsack capacity:5

Enter profit followed by weight:
12 2
10 1
20 3
15 2

Max profit=37rit@rit-desktop:~$ ./a.out

Enter the no. of objects:4

Enter the knapsack capacity:10

Enter profit followed by weight:
10 4
42 7
25 5
12 3
```

Explanation:

❖ **Global Variables and Definitions**

- This includes the standard input-output library, which is necessary for using functions like printf() and scanf() to handle input/output operations.
- `w[10]`: An array to store the weights of up to 10 items (this is a fixed size of 10, but you could make it dynamic in a more general case).
- `p[10]`: An array to store the profits associated with up to 10 items.
- `n`: An integer variable to store the number of items
- Defines a helper function max() that returns the larger of the two integers a and b. This is used later to decide whether to include an item in the knapsack or not based on profit maximization.
- **`knap(i, m)`** is the recursive function that calculates the maximum profit. It considers the first `i` items and a knapsack with capacity `m`.
- **Base Case**: `if(i == n)`: If we've considered all items (i.e., `i == n`), it returns 0 if the weight of the current item exceeds the remaining capacity (`w[i] > m`), otherwise, it returns the profit of the item `p[i]`.
- **If the weight of the current item exceeds the remaining capacity**: `if(w[i] > m)`, then skip this item and move to the next item by calling `knap(i + 1, m)`.
- **Recursive case**: `return max(knap(i + 1, m), knap(i + 1, m - w[i]) + p[i]);`

- This checks two possibilities:
    - The maximum profit without including the current item (`knap(i + 1, m)`).
    - The maximum profit including the current item. If included, the remaining capacity is reduced by `w[i]` and the profit is increased by `p[i]` (`knap(i + 1, m - w[i]) + p[i]`).
- It returns the larger of these two values.

main Function Explanation:

- Declares variables m to store the knapsack's capacity, i for looping, and max_profit to store the final result.
- Asks the user to input the number of items (n) and reads the value.
- Asks the user for the knapsack's capacity (m) and reads the value.
- Prompts the user to enter the profit and weight for each item. It reads the profit (p[i]) and weight (w[i]) of each item and stores them in the respective arrays.
- Calls the knap() function starting from the first item (i = 1) and with the full capacity (m), storing the result (maximum profit) in max_profit.
- Finally, prints the maximum profit that can be obtained by selecting items to fit in the knapsack.
- The main() function returns 0, indicating successful execution of the program.

 Output**:**

- The program prints the **maximum profit** that can be obtained by selecting a subset of items whose total weight does not exceed the knapsack’s capacity.

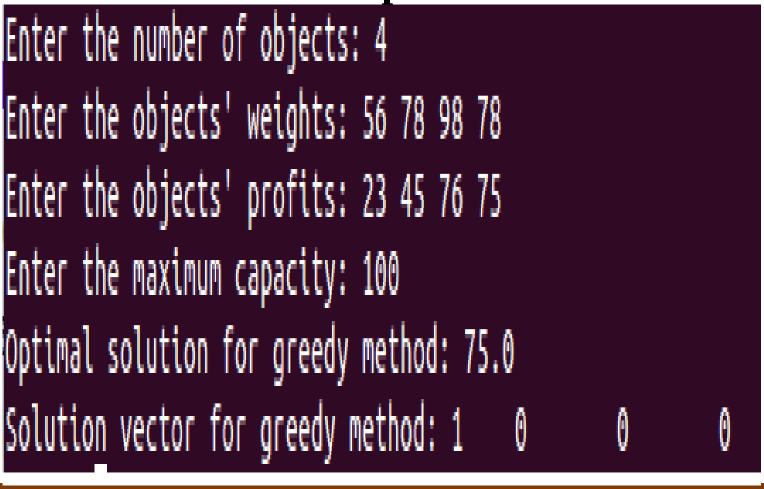
7. Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.

Program:

```
#include <stdio.h>
#define MAX 50
int p[MAX], w[MAX], x[MAX];
double maxprofit;
int n, m, i, j;
void greedyKnapsack(int n, int w[], int p[], int m)
{
   double ratio[MAX];
   // Calculate the ratio of profit to weight for each item
   for (i = 0; i < n; i++)
   {
      ratio[i] = (double)p[i] / w[i];
   }
   // Sort items based on the ratio in non-increasing order
   for (i = 0; i < n - 1; i++)
   {
      for (j = i + 1; j < n; j++)
      {
         if (ratio[i] < ratio[j])
         {
            double temp = ratio[i];
            ratio[i] = ratio[j];
            ratio[j] = temp;

            int temp2 = w[i];
            w[i] = w[j];
            w[j] = temp2;

            temp2 = p[i];
            p[i] = p[j];
            p[j] = temp2;
         }
      }
   }
   int currentWeight = 0;
   maxprofit = 0.0;
   // Fill the knapsack with items
   for (i = 0; i < n; i++)
   {
      if (currentWeight + w[i] <= m)
```

```
        {
            x[i] = 1; // Item i is selected
            currentWeight += w[i];
            maxprofit += p[i];
        }
        else
        {
            // Fractional part of item i is selected
            x[i] = (m - currentWeight) / (double)w[i];
            maxprofit += x[i] * p[i];
            break;
        }
    }
    printf("Optimal solution for greedy method: %.1f\n", maxprofit);
    printf("Solution vector for greedy method: ");
    for (i = 0; i < n; i++)
    printf("%d\t", x[i]);
}
int main()
{
    printf("Enter the number of objects: ");
    scanf("%d", &n);
    printf("Enter the objects' weights: ");
    for (i = 0; i < n; i++)
    scanf("%d", &w[i]);
    printf("Enter the objects' profits: ");
    for (i = 0; i < n; i++)
    scanf("%d", &p[i]);
    printf("Enter the maximum capacity: ");
    scanf("%d", &m);
    greedyKnapsack(n, w, p, m);
    return 0;
}
```

Output:



```
Enter the number of objects: 4
Enter the objects' weights: 56 78 98 78
Enter the objects' profits: 23 45 76 75
Enter the maximum capacity: 100
Optimal solution for greedy method: 75.0
Solution vector for greedy method: 1    0       0       0
```

Explanation:

This program implements a greedy algorithm for the fractional knapsack problem. The goal of this problem is to maximize the total profit by selecting items with given weights and profits, subject to the constraint that the total weight cannot exceed a specified maximum capacity of the knapsack**.**

- **#include <stdio.h>**: This includes the standard input-output library that provides functions for input (scanf) and output (printf).
- **#define MAX 50**: This defines a constant MAX with the value 50, which will be used to limit the size of arrays. In this case, it limits the maximum number of items the knapsack can handle to 50.

- **p[MAX]**: Array to store the profits of each item.
- **w[MAX]**: Array to store the weights of each item.
- **x[MAX]**: Array to store the solution vector, where each element represents the fraction of the corresponding item included in the knapsack. x[i] = 1 means the full item is selected, and x[i] < 1 means a fraction of the item is selected.
- **maxprofit**: Variable to store the maximum profit calculated using the greedy algorithm.
- **n**: The number of items.
- **m**: The maximum capacity of the knapsack.
- **i, j**: Loop variables for iterating over the items.

Greedy Knapsack Function

- The function greedyKnapsack is defined to solve the fractional knapsack problem using a greedy approach.
  - **n**: The number of items.
  - **w[]**: Array of item weights.
  - **p[]**: Array of item profits.
  - **m**: Maximum capacity of the knapsack.
  - **ratio[MAX]**: An array to store the profit-to-weight ratio for each item.
  - This loop calculates the **profit-to-weight ratio** for each item. The ratio is calculated as:
- **ratio[i] = p[i] / w[i]** for each item i, where p[i] is the profit and w[i] is the weight.
- The ratio represents how much profit we gain per unit of weight for each item.
- This **sorting block** uses a **bubble sort algorithm** to sort the items in **non-increasing order** of their profit-to-weight ratios.

- For each pair of items i and j, if the ratio of item i is less than the ratio of item j, it swaps their corresponding ratios, weights (w[i], w[j]), and profits (p[i], p[j]).
- After this sorting step, the items are arranged in order where the item with the highest profit-to-weight ratio comes first. This is important because the greedy algorithm prioritizes these items.
   - **currentWeight = 0**: A variable to keep track of the total weight of the items selected so far.
   - **maxprofit = 0.0**: A variable to keep track of the total profit obtained by adding items (or fractions of items) to the knapsack.
- This loop goes through each item i in the sorted list to decide whether to include it fully or fractionally in the knapsack.
- **if (currentWeight + w[i] <= m)**: If the weight of the current item w[i] can fit into the remaining capacity (m - currentWeight), the item is **fully selected**. We update:
   - x[i] = 1: Item i is fully included.
   - **currentWeight += w[i]**: The weight of item i is added to the knapsack's current weight.
   - **maxprofit += p[i]**: The profit of item i is added to the total profit.
- If the current item cannot be fully included (i.e., the knapsack will overflow), we calculate the **fractional part** of the item that can be included:
   - **x[i] = (m - currentWeight) / (double)w[i]**: The fraction of the item that can fit is determined by the remaining capacity divided by the item's weight.
   - **maxprofit += x[i] * p[i]**: The fractional profit is added to the total profit.
   - The loop **breaks** since no more items can be added (after adding a fractional item, the knapsack is full).
- After completing the greedy algorithm, the function prints the **maximum profit** obtained using the greedy method (maxprofit).
- The solution vector x[] is also printed, showing the fraction of each item selected. If x[i] = 1, the full item is selected; if x[i] < 1, only a fraction of the item is selected.

<u>Main Function</u>

- The program starts by asking the user to input the number of items (n) and reads the value.
- The program asks the user to input the weights of the n items and stores them in the w[] array.
- The program asks the user to input the profits of the n items and stores them in the p[] array.
- The program asks for the knapsack's maximum capacity m and reads the value.
- The function greedyKnapsack() is called with the number of items n, the arrays of weights w[] and profits p[], and the knapsack's capacity m. It calculates and prints the solution.
- The main() function ends and returns 0, indicating successful execution.

- **Input**: The program takes the number of items, their weights, profits, and the maximum capacity of the knapsack as input.
- **Greedy Approach**: The items are sorted based on their profit-to-weight ratio in descending order. The program then iterates over the sorted items, adding them (or fractions of them) to the knapsack until the capacity is full.
- **Output**: The program prints the maximum profit and the solution vector, which indicates which items are selected and in what fraction.

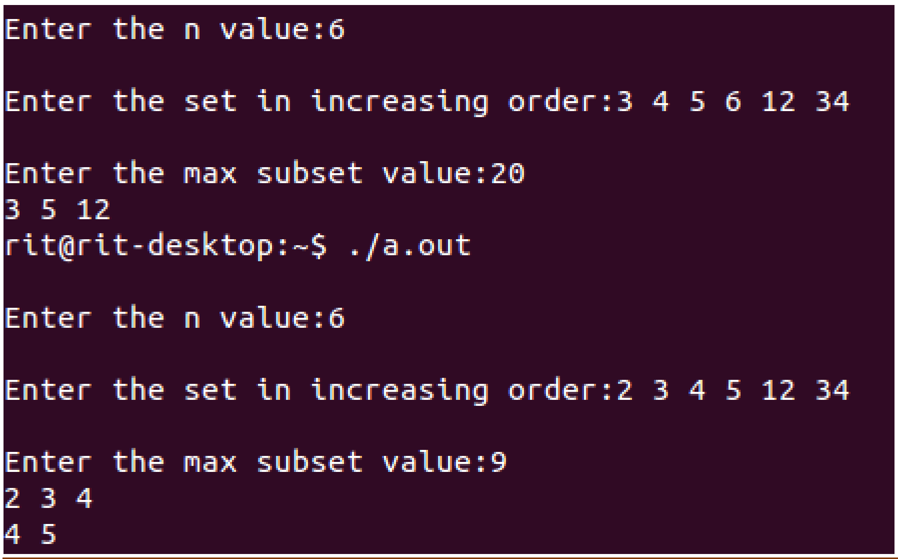
8. Design and implement C/C++ Program to find a subset of a given set S = {sl , s2,.....,sn} of n positive integers whose sum is equal to a given positive integer d.

Program:

```
#include<stdio.h>
#define MAX 10
int s[MAX],x[MAX],d;
void sumofsub(int p,int k,int r)
{
   int i;
   x[k]=1;
   if((p+s[k])==d)
   {
      for(i=1; i<=k; i++)
         if(x[i]==1)
         printf("%d ",s[i]);
      printf("\n");
   }
      else if(p+s[k]+s[k+1]<=d)
      sumofsub(p+s[k],k+1,r-s[k]);
      if((p+r-s[k]>=d) && (p+s[k+1]<=d))
   {
      x[k]=0;
      sumofsub(p,k+1,r-s[k]);
```

```
    }
}
int main()
{
    int i,n,sum=0;
    printf("\nEnter the n value:");
    scanf("%d",&n);
    printf("\nEnter the set in increasing order:");
    for(i=1; i<=n; i++)
    scanf("%d",&s[i]);
    printf("\nEnter the max subset value:");
    scanf("%d",&d);
    for(i=1; i<=n; i++)
     sum=sum+s[i];
    if(sum<d || s[1]>d)
    printf("\nNo subset possible");
    else
    sumofsub(0,1,sum);
    return 0;
}
```

Output:



```
Enter the n value:6

Enter the set in increasing order:3 4 5 6 12 34

Enter the max subset value:20
3 5 12
rit@rit-desktop:~$ ./a.out

Enter the n value:6

Enter the set in increasing order:2 3 4 5 12 34

Enter the max subset value:9
2 3 4
4 5
```

Explanation:

This program is designed to solve the Subset Sum Problem using backtracking. The problem asks to find all subsets of a given set of integers whose sum is equal to a specified target value (d).

Variables and Constants:

- s[MAX]: An array to store the input set of integers. This is where the set is stored in increasing order.
- x[MAX]: An array used for backtracking to track whether an element is included in the current subset (x[i] = 1 means item i is included in the subset, and x[i] = 0 means it is excluded).
-

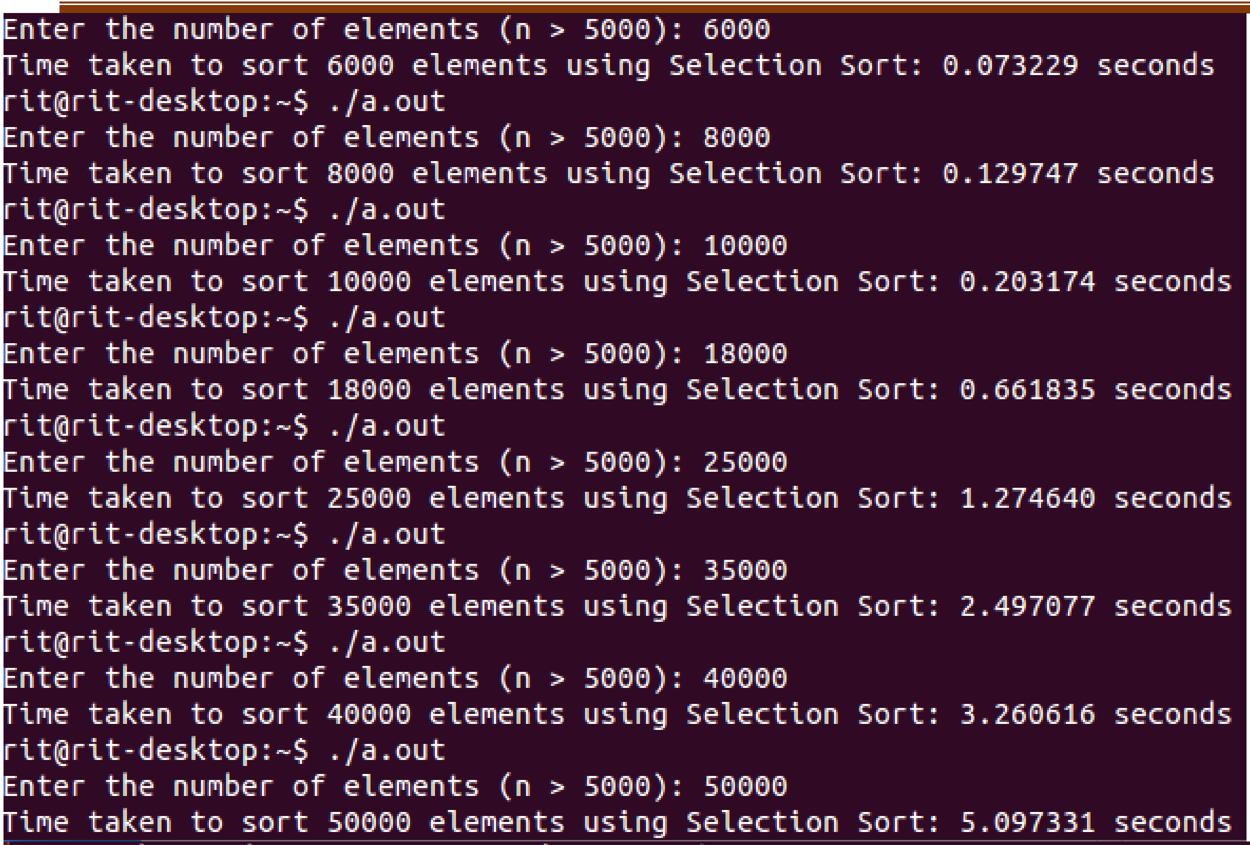
9. Design and implement C/C++ Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Program:

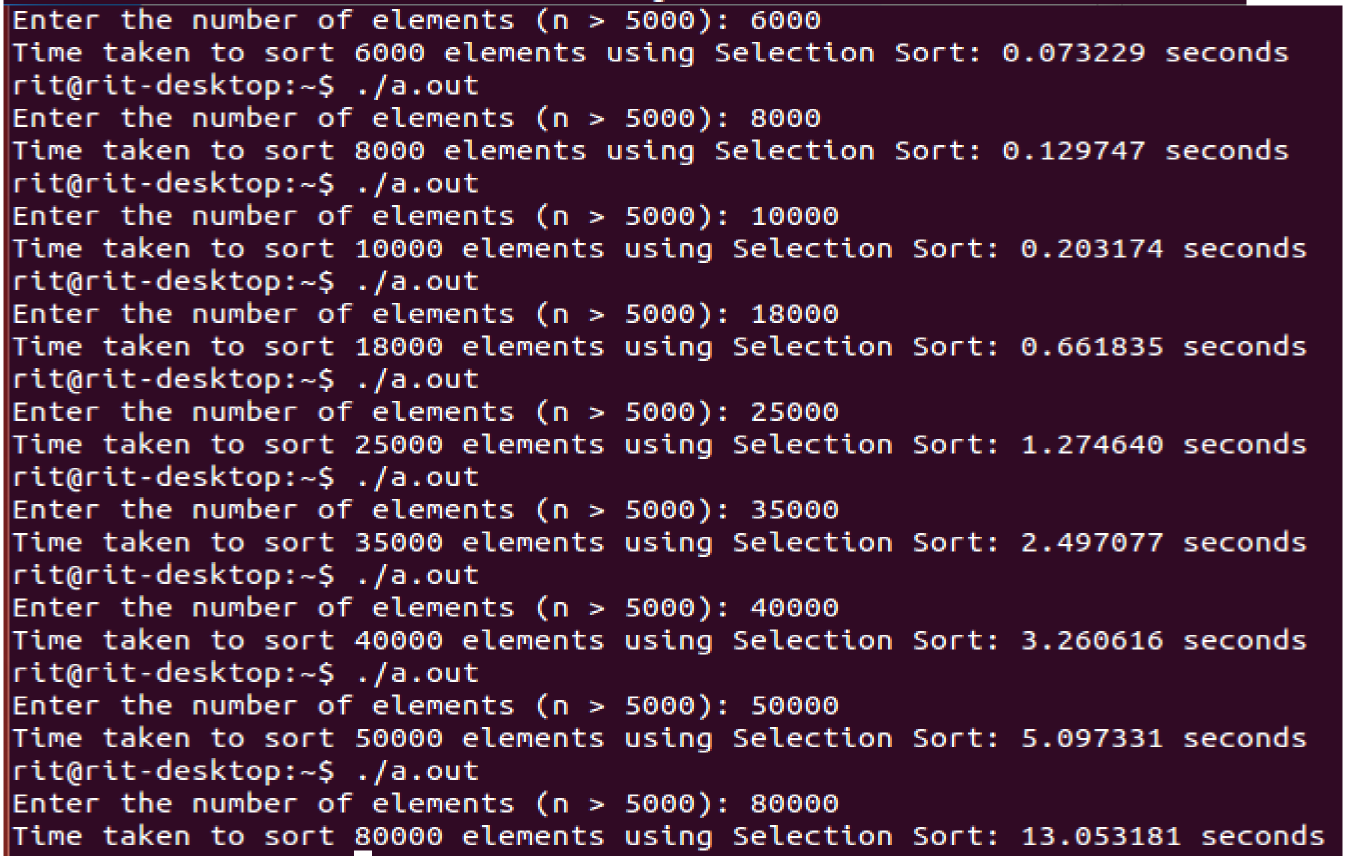
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
// Function to perform Selection Sort
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;
    for (i = 0; i < n - 1; i++)
   {
        min_idx = i;
        for (j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[min_idx])
            {
                min_idx = j;
            }
        }
        swap(&arr[min_idx], &arr[i]);
    }
}

// Function to generate random numbers and store in array
void generateRandomNumbers(int arr[], int n)
{
     int i;
     for ( i = 0; i < n; i++)
```

```
        {
        arr[i] = rand() % 100000; // Generate numbers in range 0-99999
        }
}
int main()
{
    int n;
    clock_t start, end;
    double time_taken;
    printf("Enter the number of elements (n > 5000): ");
    scanf("%d", &n);

    if (n <= 5000)
    {
        printf("Please enter a value greater than 5000.\n");
        return 1;
    }

    int *arr = (int *)malloc(n * sizeof(int));

    // Generate random numbers
    generateRandomNumbers(arr, n);

    // Measure sorting time
    start = clock();
    selectionSort(arr, n);
    end = clock();

    // Calculate time taken
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken to sort %d elements using Selection Sort: %lf seconds\n", n,
time_taken);

    // Free allocated memory
    free(arr);
    return 0;
}
```

Output:



```
Enter the number of elements (n > 5000): 6000
Time taken to sort 6000 elements using Selection Sort: 0.073229 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 8000
Time taken to sort 8000 elements using Selection Sort: 0.129747 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 10000
Time taken to sort 10000 elements using Selection Sort: 0.203174 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 18000
Time taken to sort 18000 elements using Selection Sort: 0.661835 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 25000
Time taken to sort 25000 elements using Selection Sort: 1.274640 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 35000
Time taken to sort 35000 elements using Selection Sort: 2.497077 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 40000
Time taken to sort 40000 elements using Selection Sort: 3.260616 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 50000
Time taken to sort 50000 elements using Selection Sort: 5.097331 seconds
```



```
Enter the number of elements (n > 5000): 6000
Time taken to sort 6000 elements using Selection Sort: 0.073229 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 8000
Time taken to sort 8000 elements using Selection Sort: 0.129747 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 10000
Time taken to sort 10000 elements using Selection Sort: 0.203174 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 18000
Time taken to sort 18000 elements using Selection Sort: 0.661835 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 25000
Time taken to sort 25000 elements using Selection Sort: 1.274640 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 35000
Time taken to sort 35000 elements using Selection Sort: 2.497077 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 40000
Time taken to sort 40000 elements using Selection Sort: 3.260616 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 50000
Time taken to sort 50000 elements using Selection Sort: 5.097331 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 80000
Time taken to sort 80000 elements using Selection Sort: 13.053181 seconds
```

Explanation:

This C program implements the Selection Sort algorithm and measures the time it takes to sort an array of randomly generated integers.

1. Headers and Function Prototypes:

- **#include <stdio.h>**: This includes the standard input-output library for functions like printf and scanf.
- **#include <stdlib.h>**: This includes the standard library for dynamic memory allocation (malloc) and other utility functions.
- **#include <time.h>**: This includes the time library, which provides functions for time measurement (clock(), CLOCKS_PER_SEC).

- This function **swaps two integers** by using a temporary variable temp.
  - *x and *y represent the values at the memory addresses x and y.
  - The values of *x and *y are exchanged using the temporary variable temp.

Selection Sort Function

- This function implements **Selection Sort**.
  - **Outer loop** (for (i = 0; i < n - 1; i++)): This loop iterates over each element, treating it as the "current" element to be sorted.
  - **Inner loop** (for (j = i + 1; j < n; j++)): For each "current" element arr[i], the inner loop checks all the remaining unsorted elements to find the minimum value.
  - **Finding the minimum**: If an element arr[j] is smaller than arr[min_idx], it updates min_idx to the index j.
  - **Swapping**: After finding the minimum element, the function swaps the element at index i with the one at index min_idx. This places the smallest unsorted element at the correct position.
  - The process continues until the array is fully sorted.

Random Number Generator Function

- **int n;**: Variable to store the number of elements the user wants to sort.
- **clock_t start, end;**: Variables to store the starting and ending clock ticks for measuring execution time.
- **double time_taken;**: Variable to store the total time taken for sorting in seconds
- Prompts the user to enter the number of elements (n) they want to sort.
- **scanf("%d", &n);**: Reads the input number and stores it in the variable n.

Input validation: The program checks if the value of n is greater than 5000. If it's not, the program prints an error message and exits with return 1;.

Dynamic Memory Allocation: Allocates memory dynamically for the array arr[] to store n integers.

- malloc(n * sizeof(int)) allocates memory for n integers. The cast (int *) ensures the pointer is of the correct type.

- Calls the generateRandomNumbers() function to populate the array arr[] with random numbers.

Measure Time: The program uses the clock() function to measure the time before and after the sorting process.

- **start = clock();**: Records the current clock time before sorting starts.
- **selectionSort(arr, n);**: Calls the selectionSort() function to sort the array arr[] with n elements.
- **end = clock();**: Records the current clock time after the sorting ends.
- **Calculate Execution Time**: The total time taken to sort the array is calculated by subtracting start from end to get the number of clock ticks spent on sorting.
    - **CLOCKS_PER_SEC**: This constant defines how many clock ticks correspond to one second. Dividing by CLOCKS_PER_SEC converts the time to seconds.
- **Print the Time Taken**: Displays the time it took to sort n elements using Selection Sort.
- **Free Allocated Memory**: After sorting, the program releases the dynamically allocated memory for the array arr[] using the free() function.
- **return 0;**: Returns 0 from the main() function, indicating successful execution.

10. Design and implement C/C++ Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

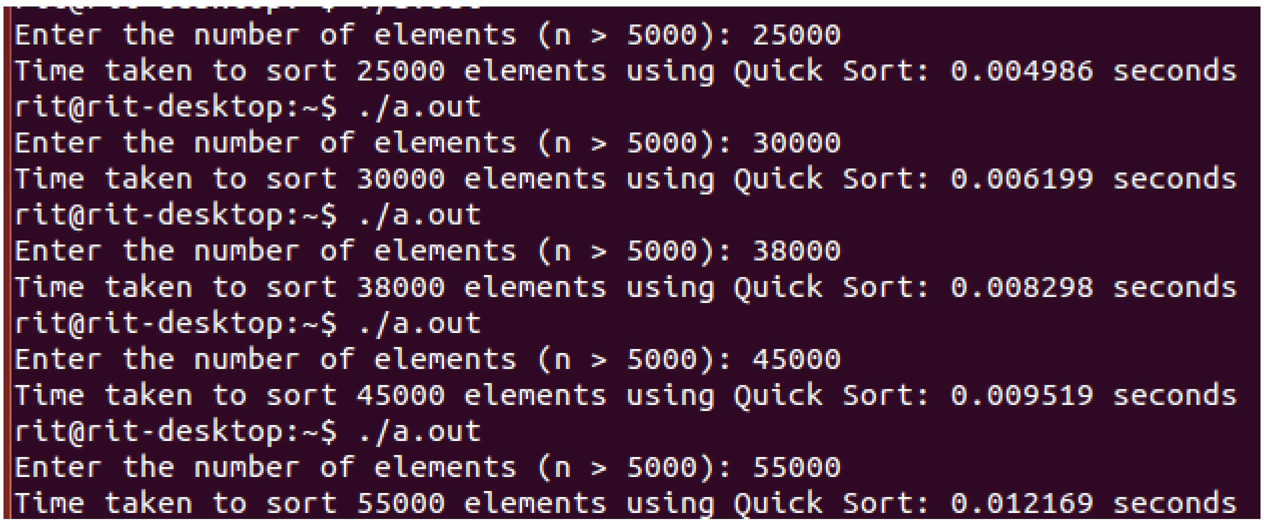
Program:

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to swap two elements

void swap(int* a, int* b)

 {

   int t = *a;

   *a = *b;
```

```
    *b = t;

}



// Partition function for Quick Sort

int partition(int arr[], int low, int high)

{

    int j;

    int pivot = arr[high]; // Choosing the last element as the pivot

    int i = (low - 1); // Index of smaller element



    for (j = low; j < high; j++)

          {

            if (arr[j] < pivot)

                    { // If element is smaller than pivot

                    i++;

                     swap(&arr[i], &arr[j]);

          }

          }

          swap(&arr[i + 1], &arr[high]);

          return (i + 1);

}
```

```
// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to generate random numbers
void generateRandomNumbers(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
          {
        arr[i] = rand() % 100000; // Generate numbers in range 0-99999
           }
}

int main() {
    int n;
    clock_t start, end;
    double time_taken;


    printf("Enter the number of elements (n > 5000): ");

    scanf("%d", &n);

    if (n <= 5000)

  {

        printf("Please enter a value greater than 5000.\n");

        return 1;

    }


    int *arr = (int *)malloc(n * sizeof(int));


    // Generate random numbers

    generateRandomNumbers(arr, n);


    // Measure sorting time

    start = clock();

    quickSort(arr, 0, n - 1);

    end = clock();


    // Calculate time taken

    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken to sort %d elements using Quick Sort: %lf seconds\n", n,
time_taken);

    // Free allocated memory

    free(arr);

    return 0;

}
```

Output:



```
Enter the number of elements (n > 5000): 25000
Time taken to sort 25000 elements using Quick Sort: 0.004986 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 30000
Time taken to sort 30000 elements using Quick Sort: 0.006199 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 38000
Time taken to sort 38000 elements using Quick Sort: 0.008298 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 45000
Time taken to sort 45000 elements using Quick Sort: 0.009519 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 55000
Time taken to sort 55000 elements using Quick Sort: 0.012169 seconds
```

Explanation:

This C program implements the Quick Sort algorithm and measures the time it takes to sort an array of randomly generated integers.

1. Headers and Libraries:

- #include <stdlib.h>: Includes functions like malloc() for dynamic memory allocation, rand() for generating random numbers, and free() for deallocating memory.
- #include <time.h>: Includes functions for measuring time, specifically clock() to capture the time taken for sorting.

2. swap Function:

This function swaps two elements in the array.

- It uses a temporary variable t to swap the values of a and b.
- This is a helper function used in the partitioning step of Quick Sort.

3. partition Function:

This function is used to partition the array into two halves based on a pivot element. Elements smaller than the pivot go to the left, and elements larger than the pivot go to the right.

- Pivot Selection: The last element (arr[high]) is selected as the pivot.
- Partitioning Process:
    - The loop checks each element from low to high - 1.
    - If an element is smaller than the pivot, it gets swapped with the element at index i + 1.
    - After the loop, the pivot is swapped with the element at i + 1 to ensure it is in its correct sorted position.
- The function returns the index where the pivot is located after partitioning.

4. quickSort Function:

This is the main function that implements the Quick Sort algorithm. Quick Sort is a divide-and-conquer algorithm where the array is recursively divided into smaller sub-arrays, which are then sorted.

- The function first checks if the low index is less than the high index. If not, it exits, as the array is already sorted or has one element.
- The partition function is called to place the pivot element in its correct position. Then, the array is divided into two sub-arrays: one from low to pi - 1 and the other from pi + 1 to high.
- The quickSort function is recursively called on these sub-arrays until the entire array is sorted.

5. generateRandomNumbers Function:

This function generates random integers for the array.

- It uses rand() to generate random integers, and % 100000 ensures the numbers are between 0 and 99999.

6. main Function:

The main function is responsible for user interaction, memory allocation, invoking the sorting algorithm, and printing the results.

- User Input: The program first asks the user for the number of elements (n) and checks if it's greater than 5000. If n <= 5000, the program asks the user to enter a value greater than 5000 and terminates.
- Dynamic Memory Allocation: The program dynamically allocates memory for the array of size n. If memory allocation fails, it prints an error message and exits.
- Generating Random Numbers: It then calls generateRandomNumbers to populate the array with random integers.
- Sorting: The program uses clock() to measure the time taken to sort the array using quickSort.
- Time Calculation: The time is computed by subtracting start from end and dividing by CLOCKS_PER_SEC to convert the result to seconds.
- Memory Deallocation: Finally, the program frees the dynamically allocated memory before exiting.

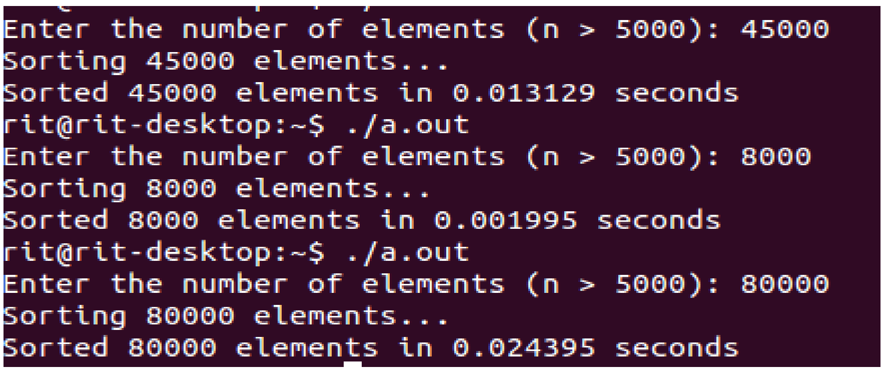
11. Design and implement C/C++ Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator

Program:

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Merge function to combine two sorted subarrays

void merge(int arr[], int left, int mid, int right)

{

   int i, j, k;

   int n1 = mid - left + 1;

   int n2 = right - mid;


// Temporary arrays

   int L[n1], R[n2];

   for (i = 0; i < n1; i++)

      L[i] = arr[left + i];

   for (j = 0; j < n2; j++)

      R[j] = arr[mid + 1 + j];

   i = 0; j = 0; k = left;

   while (i < n1 && j < n2)

    {

           if (L[i] <= R[j])

          {

             arr[k] = L[i];
```

```
                i++;

            }

             else

            {

                 arr[k] = R[j];

                  j++;

           }

        k++;

    }


    while (i < n1)

   {

        arr[k] = L[i];

        i++;

        k++;

    }

    while (j < n2)

    {

        arr[k] = R[j];

        j++;

        k++;

    }

}

// Merge Sort function

void mergeSort(int arr[], int left, int right)

 {

    if (left < right)
    {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
// Function to generate a random array
void generateRandomArray(int arr[], int n)
{
            int i;
          for (i = 0; i < n; i++)
        arr[i] = rand() % 100000;  // Random numbers between 0 and 99999
}
// Function to measure execution time
void measureTime(int n)
 {
       int *arr = (int *)malloc(n * sizeof(int));
        if (arr == NULL)
      {
        printf("Memory allocation failed\n");
        return;
      }
    generateRandomArray(arr, n);
    printf("Sorting %d elements...\n", n);
    clock_t start = clock();
    mergeSort(arr, 0, n - 1);
    clock_t end = clock();

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("Sorted %d elements in %f seconds\n", n, time_taken);

    free(arr);

  }

  // Main function

  int main()

   {

    int n;

    printf("Enter the number of elements (n > 5000): ");

    scanf("%d", &n);

    if (n <= 5000)

    {

      printf("Please enter a value greater than 5000.\n");

      return 1;

    }

    measureTime(n);

    return 0;

  }
```

Output:



```
Enter the number of elements (n > 5000): 45000
Sorting 45000 elements...
Sorted 45000 elements in 0.013129 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 8000
Sorting 8000 elements...
Sorted 8000 elements in 0.001995 seconds
rit@rit-desktop:~$ ./a.out
Enter the number of elements (n > 5000): 80000
Sorting 80000 elements...
Sorted 80000 elements in 0.024395 seconds
```

Explanation:

This C program implements the Merge Sort algorithm and measures the time it takes to sort an array of randomly generated integers. The sorting process is repeated multiple times to help generate a measurable amount of time for performance evaluation.

Headers and Libraries:

- #include <stdio.h>: Standard I/O library to handle user input and output.
- #include <stdlib.h>: Provides functions for memory allocation (malloc), random number generation (rand), and memory deallocation (free).
- #include <time.h>: Allows us to measure the time taken for sorting using the clock() function.

2. **Merge Function:**

This function merges two sorted sub-arrays into a single sorted array.

Algorithm:

- Input: Two sorted sub-arrays: one from left to mid and the other from mid + 1 to right.
- Output: The merged and sorted array stored in arr[left] to arr[right].
- Temporary Arrays: L[] and R[] are used to hold the left and right halves of the array during merging.
- Merging Logic: The elements from L[] and R[] are compared, and the smaller element is placed in the correct position in the original array arr[].
- After the merging process, any leftover elements from either L[] or R[] are copied back into arr[].

3. **MergeSort Function:**

The main function implementing the Merge Sort algorithm, which divides the array into two halves and recursively sorts them using merge().

Algorithm:

- **Base Case:** If left is not less than right, return because the array has one element (already sorted).
- **Recursive Case:** The array is divided into two halves, each of which is sorted recursively using mergeSort. After sorting, the two halves are merged using the merge

function.

- **Dividing the Array:** The middle index is calculated as mid = left + (right - left) / 2. The array is recursively divided into two sub-arrays.
- **Merge:** After the two sub-arrays are sorted, the merge() function is called to combine them back together in sorted order.

4. generateRandomArray Function:

This function generates random integers and stores them in the array arr[]. The random numbers are between 0 and 99999.

5. main Function:

The main function serves as the entry point for the program. It handles user input, memory allocation, sorting, and timing.

- User Input: The program first asks for the number of elements (n). If n is less than or equal to 5000, it prompts the user to enter a larger value and exits.
- Memory Allocation: Dynamically allocates memory for the array of integers. If memory allocation fails, the program prints an error and exits.
- Random Array Generation: The program populates the array with random integers using generateRandomArray().
- Sorting and Timing: It measures the time taken to sort the array 1000 times by calling mergeSort() in a loop. The clock() function is used to capture the start and end time, and the time per sorting iteration is calculated.
- Memory Deallocation: After the sorting process is complete, the program frees the allocated memory.

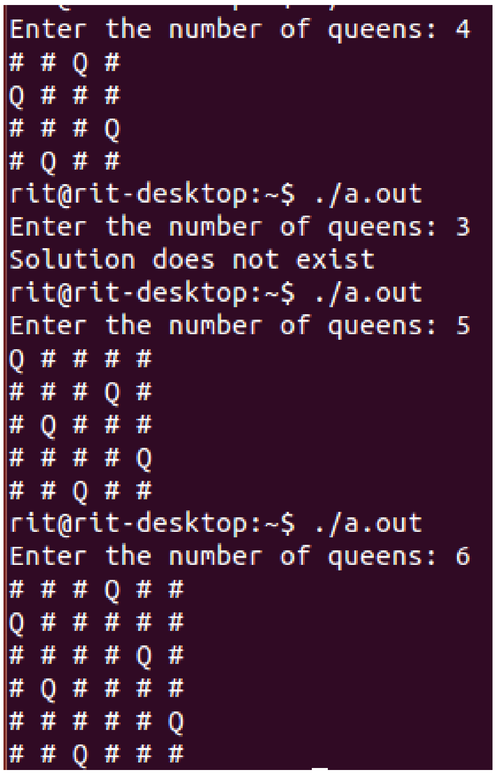
12. Design and implement C/C++ Program for N Queen's problem using Backtracking.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
// Function to print the solution
void printSolution(int **board, int N)
{
  int i,j;
   for ( i = 0; i < N; i++)
   {
      for ( j = 0; j < N; j++)
      {
         printf("%s ", board[i][j] ? "Q" : "#");
      }
      printf("\n");
   }
}
// Function to check if a queen can be placed on board[row][col]
bool isSafe(int **board, int 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, int N, int col)
{
     int i;
    // 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
```

Output:



```
Enter the number of queens: 4
# # Q #
Q # # #
# # # Q
# Q # #
rit@rit-desktop:~$ ./a.out
Enter the number of queens: 3
Solution does not exist
rit@rit-desktop:~$ ./a.out
Enter the number of queens: 5
Q # # # #
# # # Q #
# Q # # #
# # # # Q
# # Q # #
rit@rit-desktop:~$ ./a.out
Enter the number of queens: 6
# # # Q # #
Q # # # # #
# # # # Q #
# Q # # # #
# # # # # Q
# # Q # # #
```

Explanation:

This program solves the N-Queens problem using backtracking. The problem involves placing N queens on an N x N chessboard such that no two queens threaten each other. A queen can attack another queen if they share the same row, column, or diagonal.

1. **Headers and Libraries:**

- #include <stdio.h>: Provides standard I/O functions such as printf and scanf.
- #include <stdlib.h>: Used for memory allocation and deallocation (specifically malloc and free).
- #include <stdbool.h>: Enables the use of bool data type for logical values (true and false).

2. **printSolution Function:**

- Purpose: To print the chessboard after finding a valid arrangement of queens.
- Input: A 2D array board of size N x N and the size N of the board.
- Logic:
    - Iterates through the board and prints "Q" for a queen (board[i][j] == 1) and "#" for an empty space (board[i][j] == 0).
    - The board is printed row by row.

3. **isSafe Function:**

- Purpose: To check whether placing a queen at board[row][col] is safe.
- Input: board (the chessboard), N (size of the board), row (row index), col (column index).
- Logic:
    - Row Check: It checks all columns to the left of the current column to see if there's a queen placed on the same row (board[row][i]).
    - Upper Diagonal Check: Checks the upper-left diagonal for any queens. The diagonal is checked by decrementing both i (row) and j (column).
    - Lower Diagonal Check: Checks the lower-left diagonal by incrementing i (row) and j (column).
- Return: Returns true if the queen can be safely placed, otherwise returns false.

4. solveNQUtil Function:

- Purpose: The core recursive backtracking function that attempts to place queens on the board.
- Input: board (the chessboard), N (size of the board), col (current column to place the queen).
- Logic:
    - If col >= N, it means all queens are placed successfully, and the function returns true.
    - For each row in the current column (col), it checks if placing a queen is safe using isSafe(). If it is, the queen is placed at board[i][col].
    - It then recursively tries to place queens in the next columns.
    - If placing the queen in the current position does not lead to a solution, it backtracks by removing the queen (board[i][col] = 0), which is done using the backtracking concept (undoing a decision that led to failure).
- Return: Returns true if a solution is found, otherwise false.

5. solveNQ Function:

- Purpose: Initializes the chessboard and calls the recursive function solveNQUtil to solve the N-Queens problem.
- Input: N (the size of the board, i.e., the number of queens).
- Logic:
    - Allocates memory for the board as a 2D array of size N x N (dynamically allocated).
    - Initializes the board with all entries set to 0 (indicating empty squares).
    - Calls solveNQUtil to attempt to solve the problem starting from column 0.
    - If a solution is found, it prints the solution using printSolution(). If no solution is found, it prints "Solution does not exist".
    - Finally, the allocated memory for the board is freed.

Example Output:

For an input of N = 4, the program will output a possible solution like this:

```
Enter the number of queens: 4

# Q # #

# # Q #

Q # # #

# # # Q
```

This shows one valid configuration of placing 4 queens on a 4x4 chessboard.

Key Concepts:

1. Backtracking: The core of the algorithm is backtracking, where the program tries placing a queen in each row of the current column and recursively moves to the next column. If placing the queen leads to a solution, it moves forward; otherwise, it backtracks and tries another position.
2. Safety Check: The function isSafe() ensures that no two queens threaten each other by checking the current row, column, and diagonals for any existing queens.
3. Dynamic Memory Allocation: The board is dynamically allocated using malloc, and memory is freed after the solution is printed or if no solution is found.

Viva Questions

1. What is an algorithm?

Answer: An algorithm is a well-defined, step-by-step procedure or set of rules to solve a specific problem or perform a computation. It takes an input, processes it, and produces an output. The algorithm must be finite, unambiguous, and must terminate after a certain number of steps.

2. What are the different types of algorithms?

Answer: Algorithms can be classified into several types based on their design approach:

- **Divide and Conquer**: Divides the problem into smaller subproblems, solves them, and combines the results. Example: Merge Sort, Quick Sort.
- **Greedy Algorithm**: Makes the locally optimal choice at each stage with the hope of finding a global optimum. Example: Dijkstra's algorithm.
- **Dynamic Programming**: Solves problems by breaking them down into overlapping subproblems and storing the results to avoid redundant computations. Example: Fibonacci sequence, Knapsack problem.
- **Backtracking**: Tries to build a solution incrementally and abandons the solution if it leads to an invalid state. Example: N-Queens problem, Sudoku solver.
- **Brute Force**: Tries all possible solutions until the correct one is found. Example: Linear search, Bubble sort.

3. What is the difference between time complexity and space complexity?

Answer:

- **Time Complexity**: Refers to the amount of time an algorithm takes to complete as a function of the size of the input. It is often expressed using Big O notation (e.g., O(n), O(n^2)).
- **Space Complexity**: Refers to the amount of memory an algorithm requires to complete as a function of the size of the input. It also is often expressed using Big O notation (e.g., O(1), O(n)).

4. What is the significance of Big O notation?

Answer: Big O notation is used to describe the upper bound of the time complexity or space complexity of an algorithm. It helps in analyzing the worst-case scenario and allows comparison between different algorithms to choose the most efficient one for large inputs. For example, O(n) implies linear time complexity, while O(n^2) implies quadratic time complexity.

5. What is the difference between a recursive and an iterative algorithm?

Answer:

- **Recursive Algorithm**: A recursive algorithm solves a problem by calling itself with smaller inputs until it reaches a base case. Example: Factorial calculation.
- **Iterative Algorithm**: An iterative algorithm solves a problem by repeating a set of instructions until a certain condition is met, usually with loops like for or while. Example: Linear search.

6. What is Divide and Conquer? Explain with an example.

Answer: Divide and Conquer is a strategy where a problem is divided into smaller subproblems, solved independently, and then combined to form a solution. The subproblems are usually easier to solve than the original problem.

Example: **Merge Sort**:

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the sorted halves to get the final sorted array.

7. What is Dynamic Programming?

Answer: Dynamic Programming (DP) is a method used for solving problems by breaking them down into simpler subproblems. It stores the results of these subproblems to avoid redundant computations, which significantly improves efficiency. It is useful for optimization problems.

Example: **Fibonacci Sequence**:

- Recursive Fibonacci has overlapping subproblems, which DP can optimize by storing the results in a table to avoid recalculating them.

8. Explain the concept of Greedy Algorithms with an example.

Answer: Greedy Algorithms make the best choice at each step with the hope that these choices will lead to an optimal solution. They do not guarantee an optimal solution for all problems, but they work well for certain problems.

Example: **Activity Selection Problem**:

- Given a set of activities with start and finish times, the greedy approach selects the activity with the earliest finish time that is compatible with the already selected activities.

9. What are the advantages and disadvantages of using Recursion?

Answer:

- **Advantages**:
    - Simplifies code and reduces the complexity of solving problems.
    - Especially useful for problems that have a recursive structure, like tree traversals or backtracking problems.
- **Disadvantages**:
    - May lead to stack overflow if recursion depth is too deep.
    - Inefficient in terms of memory usage, as each recursive call requires additional memory for function calls.

10. Explain the concept of Backtracking with an example.

Answer: Backtracking is a technique used to find solutions to problems by trying possible candidates and abandoning them if they fail to satisfy the constraints.

Example: **N-Queens Problem**:

- The problem is to place N queens on an NxN chessboard such that no two queens attack each other. The algorithm places a queen in a position, then recursively tries to place the next queen. If placing a queen leads to an invalid configuration, it backtracks by removing the queen and trying a different position.

11. What is the difference between Depth-First Search (DFS) and Breadth-First Search (BFS)?

Answer:

- **Depth-First Search (DFS)**: Explores as far as possible along a branch before backtracking. It uses a stack data structure (either implicitly through recursion or explicitly using a stack).
- **Breadth-First Search (BFS)**: Explores all neighbors at the present depth before moving on to nodes at the next depth level. It uses a queue data structure.

12. What is a greedy algorithm, and how does it differ from dynamic programming?

Answer: A **greedy algorithm** makes a series of choices that appear to be the best at the moment, hoping that these local optimal choices will lead to a global optimum. **Dynamic programming**, on the other hand, solves a problem by breaking it down into overlapping subproblems and stores the solutions to these subproblems, avoiding redundant calculations.

- **Greedy Algorithms**: Make the best choice at the moment.
- **Dynamic Programming**: Solves problems by solving subproblems and using the results to solve the overall problem.

13. What are the types of searching algorithms, and how do they differ?

Answer:

- **Linear Search**: Sequentially checks each element until the desired element is found or the end is reached. Time complexity: O(n).
- **Binary Search**: Searches for an element by repeatedly dividing the sorted array in half. Time complexity: O(log n).
- **Hashing**: Uses a hash function to map keys to positions in an array, allowing for faster lookups. Average time complexity: O(1).

14. What is the difference between a graph's adjacency matrix and adjacency list representation?

Answer:

- **Adjacency Matrix**: A 2D array where each cell (i, j) contains a value (1 or 0) indicating whether there is an edge between vertex i and vertex j.
- **Adjacency List**: An array of lists. The index represents the vertex, and each list contains the vertices connected to it.
- **Adjacency Matrix**: Takes O(V^2) space, even if the graph is sparse.
- **Adjacency List**: Takes O(V + E) space, making it more efficient for sparse graphs