DRONACHARYA
Group of Institutions

B-27, Knowledge Park – III, Greater Noida Uttar Pradesh - 201308
Approved by: All India Council for Technical Education (AICTE), New Delhi
Affiliated to: Dr. A. P. J. Abdul Kalam Technical University (AKTU), Lucknow

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

# Design and Analysis of Algorithm Lab

### SUBJECT CODE: BCS-553

### B.Tech., Semester -V

### Session: 2024-25, ODD Semester

# Table of Contents

# DRONACHARYA GROUP OF INSTITUTIONS GREATER NOIDA

## VISION

- Instilling core human values and facilitating competence to address global challenges by providing Quality Technical Education.

## MISSION

- M1 - Enhancing technical expertise through innovative research and education, fostering creativity and excellence in problem-solving.
- M2 - Cultivating a culture of ethical innovation and user-focused design, ensuring technological progress enhances the well-being of society.
- M3 - Equipping individuals with the technical skills and ethical values to lead and innovate responsibly in an ever-evolving digital land

# DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

## VISION

Promoting technologists by imparting profound knowledge in information technology, all while instilling ethics through specialized technical education.

## MISSION

- Delivering comprehensive knowledge in information technology, preparing technologists to excel in a rapidly evolving digital landscape.
- Building a culture of honesty and responsibility in tech, promoting smart and ethical leadership.
- Empowering individuals with specialized technical skills and ethical values to drive positive change and innovation in the tech industry.

## Program Outcomes (POs)

**PO1: Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

**PO2: Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

**PO3: Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

**PO4: Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

**PO5: Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

**PO6: The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

**PO7: Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

**PO8: Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of engineering practice.

**PO 9: Individual and teamwork:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

**PO10: Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

**PO11: Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply theseto one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

**PO12: Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

## Programme Educational Objectives (PEOs)

**PEO1:** To enable graduates to pursue higher education and research, or have a successful career in industries associated with Computer Science and Engineering, or as entrepreneurs.

**PEO2:** To ensure that graduates will have the ability and attitude to adapt to emerging technological changes.

**PEO3:** To prepare students to analyze existing literature in an area of specialization and ethically develop innovative methodologies to solve the problems identified.

## Program Specific Outcomes (PSOs)

**PSO1:** To analyze, design and develop computing solutions by applying foundational concepts of Computer Science and Engineering.

**PSO2:** To apply software engineering principles and practices for developing quality software for scientific and business applications.

**PSO3:** To adapt to emerging Information and Communication Technologies (ICT) to innovate ideas and solutions to existing/novel problems.

| Cos | COURSE OUTCOMES |
|---|---|
| **BCS-553.1** | Understand and implement algorithm to solve problems by iterative approach. |
| **BCS-553.2** | Understand and implement algorithm to solve problems by divide and conquer approach. |
| **BCS-553.3** | Understand and implement algorithm to solve problems by Greedy algorithm approach |
| **BCS-553.4** | Understand and analyze algorithm to solve problems by Dynamic programming, backtracking.. |
| **BCS-553.5** | Understand and analyze the algorithm to solve problems by branch and bound approach. |

## Mapping of Program Outcomes with Course Outcomes (COs)

| CO-PO Matrix | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Course Outcomes | **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO10** | **PO11** | **PO12** |
| **BCS-553.1** | 2 | 2 | 3 | 3 | 3 | - | - | - | - | - | - | 2 |
| **BCS-553.2** | 3 | 3 | 3 | 2 | 2 | - | - | - | - | - | - | 3 |
| **BCS-553.3** | 2 | 3 | 3 | 3 | 3 | - | - | - | - | - | - | 2 |
| **BCS-553.4** | 2 | 3 | 2 | 2 | 2 | - | - | - | - | - | - | 2 |
| **BCS-553.5** | 2 | 3 | 2 | 2 | 2 | - | - | - | - | - | - | 3 |

**CO-PSO Mapping:**

| PSO1 | PSO2 | PSO3 |
|---|---|---|
| 1 | 2 | 1 |
| 1 | 3 | 1 |
| 1 | 3 | 1 |
| 2 | 2 | 1 |
| 2 | 2 | 1 |

# Course Overview

This course equips students with the skills to innovate and optimize computational solutions across diverse domains.

- Design and implement algorithms for various computational problems.
- Analyze algorithms to determine their efficiency in terms of time and space.
- Apply appropriate algorithmic paradigms for real-world challenges.
- Understand the limitations of algorithms and explore alternative solutions

# Course Objectives

- **Practical Implementation**: Enable students to implement algorithms using programming languages, reinforcing theoretical concepts learned in lectures.

- **Algorithm Analysis**: Develop skills to analyze the efficiency and performance of algorithms through empirical testing and comparison.

- **Problem-Solving Skills**: Enhance the ability to apply appropriate algorithms to solve complex computational problems effectively.

| Design and Analysis of Algorithm Lab (BCS553) | | |
|---|---|---|
| **Course Outcome ( CO)** | | **Bloom's Knowledge Level (KL)** |
| **At the end of course , the student will be able to:** | | |
| CO 1 | Understand and implement algorithm to solve problems by iterative approach. | $K_2$, $K_4$ |
| CO 2 | Understand and implement algorithm to solve problems by divide and conquer approach. | $K_3$, $K_5$ |
| CO 3 | Understand and implement algorithm to solve problems by Greedy algorithm approach. | $K_4$, $K_5$ |
| CO 4 | Understand and analyze algorithm to solve problems by Dynamic programming, backtracking. | $K_4$, $K_5$ |
| CO 5 | Understand and analyze the algorithm to solve problems by branch and bound approach. | $K_3$, $K_4$ |

**DETAILED SYLLABUS**

1. Program for Recursive Binary & Linear Search.
2. Program for Heap Sort.
3. Program for Merge Sort.
4. Program for Selection Sort.
5. Program for Insertion Sort.
6. Program for Quick Sort.
7. Knapsack Problem using Greedy Solution
8. Perform Travelling Salesman Problem
9. Find Minimum Spanning Tree using Kruskal's Algorithm
10. Implement N Queen Problem using Backtracking
11. 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 non graph sheet. The elements can be read from a file or can be generated using the random number generator. Demonstrate using Java how the divide and- conquer method works along with its time complexity analysis: worst case, average case and best case.
12. 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 non graph sheet. The elements can be read from a file or can be generated using the random number generator. Demonstrate how the divide and-conquer method works along with its time complexity analysis: worst case, average case and best case.
13.6. Implement , the 0/1 Knapsack problem using
   (a) Dynamic Programming method
   (b) Greedy method.
14. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.
15.Find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm. Use Union-Find algorithms in your program.
16. Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.
17. Write programs to (a) Implement All-Pairs Shortest Paths problem using Floyd's algorithm.
   (b) Implement Travelling Sales Person problem using Dynamic programming.
18. Design and implement 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. For example, if S ={1, 2, 5, 6, 8} and d= 9, there are two solutions {1,2,6}and {1,8}. Display a suitable message, if the given problem instance doesn't have a solution.
19. Design and implement to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle.

**Note: The Instructor may add/delete/modify/tune experiments, wherever he/she feels in a justified manner**
**It is also suggested that open source tools should be preferred to conduct the lab ( C, C++ etc)**

**List Of Programs:**

| | |
|---|---|
| Program1 | Write a program for iterartive and recursive Binary Search |
| Program 2 | Write a program for Quick sort. |
| Program 3 | Write a program for Merge Sort |
| Program 4 | Write a program for Heap Sort |
| Program 5 | Write a program for insertion Sort |
| Program 6 | Write a program for for Selection Sort |
| Program 7 | a. To implement knapsack problem using Greedy Technique<br>b. To implement 0/1 Knapsack problem using Dynamic Programming |
| Program 8 | Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm |
| Program 9 | To Implement Floyd's warshall algorithm. |
| Program 10 | To Implement N Queen Problem using Backtracking. |

## 1. Program to implement Recursive and iterative Linear and Binary Search

**Linear Search**:

- Iterative: Loops through the array to find the element.
- Recursive: Calls itself with the next index until the element is found or the end is reached.

**Binary Search** (for sorted arrays):

- Iterative: Uses a while loop to narrow down the search range.
- Recursive: Divides the range into two halves, searching recursively in the appropriate half.

```
#include <stdio.h>


// Function prototypes

int iterative_linear_search(int arr[], int n, int key);

int recursive_linear_search(int arr[], int n, int key, int index);

int iterative_binary_search(int arr[], int n, int key);

int recursive_binary_search(int arr[], int left, int right, int key);


// Main function

int main() {

    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15};

    int n = sizeof(arr) / sizeof(arr[0]);

    int key;


    printf("Enter the number to search: ");
```

```
    scanf("%d", &key);

    // Iterative Linear Search
    int linear_iter_result = iterative_linear_search(arr, n, key);
    printf("Iterative Linear Search: Element %s found\n",
           (linear_iter_result == -1) ? "not" : "is");

    // Recursive Linear Search
    int linear_recur_result = recursive_linear_search(arr, n, key, 0);
    printf("Recursive Linear Search: Element %s found\n",
           (linear_recur_result == -1) ? "not" : "is");

    // Iterative Binary Search
    int binary_iter_result = iterative_binary_search(arr, n, key);
    printf("Iterative Binary Search: Element %s found\n",
           (binary_iter_result == -1) ? "not" : "is");

    // Recursive Binary Search
    int binary_recur_result = recursive_binary_search(arr, 0, n - 1, key);
    printf("Recursive Binary Search: Element %s found\n",
           (binary_recur_result == -1) ? "not" : "is");

    return 0;
}
```

```
// Iterative Linear Search
int iterative_linear_search(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}

// Recursive Linear Search
int recursive_linear_search(int arr[], int n, int key, int index) {
    if (index >= n)
        return -1;
    if (arr[index] == key)
        return index;
    return recursive_linear_search(arr, n, key, index + 1);
}

// Iterative Binary Search
int iterative_binary_search(int arr[], int n, int key) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
```

```
        if (arr[mid] == key)
            return mid;
        if (arr[mid] < key)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}

// Recursive Binary Search
int recursive_binary_search(int arr[], int left, int right, int key) {
    if (left > right)
        return -1;
    int mid = left + (right - left) / 2;
    if (arr[mid] == key)
        return mid;
    if (arr[mid] < key)
        return recursive_binary_search(arr, mid + 1, right, key);
    return recursive_binary_search(arr, left, mid - 1, key);
}
```

**Input:**

```css
Enter the number to search: 7
```

**Output:**

```sql
Iterative Linear Search: Element is found
Recursive Linear Search: Element is found
Iterative Binary Search: Element is found
Recursive Binary Search: Element is found
```

## 2. Write a Program to implement Quick Sort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

There are many different versions of QuickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in QuickSort is partition. Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

```
include <stdio.h>
include <time.h>
voidExch(int *p, int *q){
    int temp = *p;
    *p = *q;
    *q = temp;
}
voidQuickSort(int a[], int low, int
    high){ int i, j, key, k;
    if(low>=high)
        return;
    key=low;
```

```
i=low+1;
j=high;
while(i<=j){
        while ( a[i] <= a[key] )
                i=i+1;
        while ( a[j] > a[key] )
                j=j -1;
         if(i<j)
                Exch(&a[i], &a[j]);
}
```

```
                Exch(&a[j],
                &a[key]);
                QuickSort(a,
                low,        j-1);
                QuickSort(a,
                j+1, high);
}
void main(){
        int n, a[1000],k;
        clock_tst,et; double ts; clrscr();
        printf("\n Enter How many
        Numbers: "); scanf("%d", &n);
        printf("\nThe Random
        Numbers are:\n"); for(k=1;
        k<=n; k++){
            a[k]=rand();
            printf("%d\t",a[k])
            ;
        }
        st=clock();
        QuickSor
        t(a, 1, n);
        et=clock();
        ts=(double)(et-st)/CLOCKS
        _PER_SEC; printf("\nSorted
        Numbers are: \n "); for(k=1;
        k<=n; k++)
       printf("%d\t", a[k]); printf("\nThe time taken is %e",ts);
```

```
Unsorted array: 10 7 8 9 1 5
Sorted array: 1 5 7 8 9 10
```

**3. WRITE A PROGRAM TO IMPLEMENT MERGE SORT**

```
#include <stdio.h>

// Function prototypes
void mergeSort(int arr[], int left, int right);
void merge(int arr[], int left, int mid, int right);
void printArray(int arr[], int size);

// Main function
int main() {
   int arr[] = {12, 11, 13, 5, 6, 7};
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("Unsorted array: ");
   printArray(arr, n);

   mergeSort(arr, 0, n - 1);

   printf("Sorted array: ");
   printArray(arr, n);

   return 0;
}

// Merge Sort function
void mergeSort(int arr[], int left, int right) {
   if (left < right) {
      // Calculate the mid-point
      int mid = left + (right - left) / 2;

      // Recursively sort the left and right halves
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);

      // Merge the sorted halves
      merge(arr, left, mid, right);
   }
}

// Merge function
void merge(int arr[], int left, int mid, int right) {
   int n1 = mid - left + 1; // Size of left subarray
   int n2 = right - mid;    // Size of right subarray

   // Temporary arrays
   int L[n1], R[n2];

   // Copy data to temporary arrays L[] and R[]
   for (int i = 0; i < n1; i++)
      L[i] = arr[left + i];
   for (int j = 0; j < n2; j++)
      R[j] = arr[mid + 1 + j];

// Merge the temporary arrays back into arr[left..right]
```

```
    int i = 0; // Initial index of left subarray
    int j = 0; // Initial index of right subarray
    int k = left; // Initial index of merged subarray

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

**4. Write a program to implement heap sort**

```
#include <stdio.h>

// Function prototypes
void heapify(int arr[], int n, int i);
void heapSort(int arr[], int n);
void printArray(int arr[], int n);

// Main function
int main() {
   int arr[] = {12, 11, 13, 5, 6, 7};
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("Unsorted array: ");
   printArray(arr, n);

   heapSort(arr, n);

   printf("Sorted array: ");
   printArray(arr, n);

   return 0;
}

// Function to heapify a subtree rooted at index i
void heapify(int arr[], int n, int i) {
   int largest = i; // Initialize largest as root
   int left = 2 * i + 1; // Left child index
   int right = 2 * i + 2; // Right child index

   // If left child is larger than root
   if (left < n && arr[left] > arr[largest])
      largest = left;

   // If right child is larger than the largest so far
   if (right < n && arr[right] > arr[largest])
      largest = right;

   // If the largest is not the root
   if (largest != i) {
      // Swap arr[i] with arr[largest]
      int temp = arr[i];
      arr[i] = arr[largest];
      arr[largest] = temp;

      // Recursively heapify the affected subtree
      heapify(arr, n, largest);
   }
}




// Function to perform heap sort
void heapSort(int arr[], int n) {
```

```
    // Build a max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from heap one by one
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to the end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

**5. PROGRAM TO IMPLEMENT INSERTION SORT.**

```
#include<stdio.h>
#include<conio.h>
void main() {
int a[50],i,j,key,n;
clrscr();
printf("\n Enter how many
no:"); scanf("%d",&n);
printf("\n Enter the array
elements:"); for(i=0;i<n ;i++)
scanf("%d",&a[i]);
for(j=1;j<n;j++) {
key=a[j];
i=j-1;
while(i>=0 && a[i]>key)
  {
a[i+1]=a[i];
i=i-1;
  }
a[i+1]=key;
  }
printf("\n Sorted array
is:\n"); for(i=0;i<n;i++)
printf("%d\n",a[i]);
getch();
}
```

**6. WRITE A PROGRAM FOR FOR SELECTION SORT**

```
#include <stdio.h>

// Function prototypes
void selectionSort(int arr[], int n);
void printArray(int arr[], int n);

// Main function
int main() {
   int arr[] = {64, 25, 12, 22, 11};
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("Unsorted array: ");
   printArray(arr, n);

   selectionSort(arr, n);

   printf("Sorted array: ");
   printArray(arr, n);

   return 0;
}

// Function to perform selection sort
void selectionSort(int arr[], int n) {
   for (int i = 0; i < n - 1; i++) {
      // Find the index of the minimum element
      int minIndex = i;
      for (int j = i + 1; j < n; j++) {
         if (arr[j] < arr[minIndex]) {
            minIndex = j;
         }
      }

      // Swap the found minimum element with the first element of the unsorted part
      int temp = arr[minIndex];
      arr[minIndex] = arr[i];
      arr[i] = temp;
   }
}

// Function to print the array
void printArray(int arr[], int n) {
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}
```

## 7. A. TO IMPLEMENT KNAPSACK PROBLEM USING GREEDY TECHNIQUE

```
#include <stdio.h>

// Structure to store items
typedef struct {
   int weight;
   int value;
   double ratio;
} Item;

// Function to compare items based on value/weight ratio
int compare(const void *a, const void *b) {
   Item *item1 = (Item *)a;
   Item *item2 = (Item *)b;
   if (item1->ratio < item2->ratio) return 1;
   else if (item1->ratio > item2->ratio) return -1;
   else return 0;
}

// Function to solve the fractional knapsack problem using the greedy technique
double knapsack(Item items[], int n, int capacity) {
   qsort(items, n, sizeof(Item), compare);  // Sort items by value/weight ratio

   double totalValue = 0.0;
   for (int i = 0; i < n; i++) {
      if (capacity >= items[i].weight) {
         // Take the full item
         totalValue += items[i].value;
         capacity -= items[i].weight;
      } else {
         // Take a fraction of the item
         totalValue += items[i].value * ((double)capacity / items[i].weight);
         break;  // Knapsack is full
      }
   }
   return totalValue;
}

// Main function
int main() {
   int n, capacity;

   printf("Enter the number of items: ");
   scanf("%d", &n);

   Item items[n];
   printf("Enter the weight and value of each item:\n");
   for (int i = 0; i < n; i++) {
      printf("Item %d - Weight: ", i + 1);
      scanf("%d", &items[i].weight);
      printf("Item %d - Value: ", i + 1);
      scanf("%d", &items[i].value);
      items[i].ratio = (double)items[i].value / items[i].weight;  // Calculate value/weight ratio
   }
```

```
printf("Enter the capacity of the knapsack: ");
   scanf("%d", &capacity);


   double maxValue = knapsack(items, n, capacity);
   printf("Maximum value in Knapsack = %.2f\n", maxValue);

   return 0;
}
```

### b. TO IMPLEMENT 0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

```
/* 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; }

// Returns 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));
}

int main()
{
```

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

OUTPUT: 220

**8. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm**

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

#define MAX 100

// Structure to represent an edge
typedef struct {
   int src, dest, weight;
} Edge;

// Structure to represent a graph
typedef struct {
   int vertices, edges;
   Edge edge[MAX];
} Graph;

// Structure to represent a subset for Union-Find
typedef struct {
   int parent;
   int rank;
} Subset;

// Function to find set of an element using path compression
int find(Subset subsets[], int i) {
   if (subsets[i].parent != i)
      subsets[i].parent = find(subsets, subsets[i].parent);
   return subsets[i].parent;
}

// Function to perform union of two sets
void Union(Subset subsets[], int x, int y) {
   int xroot = find(subsets, x);
   int yroot = find(subsets, y);

   if (subsets[xroot].rank < subsets[yroot].rank)
      subsets[xroot].parent = yroot;
   else if (subsets[xroot].rank > subsets[yroot].rank)
      subsets[yroot].parent = xroot;
   else {
      subsets[yroot].parent = xroot;
      subsets[xroot].rank++;
   }
}

// Compare function for qsort to sort edges by weight
int compare(const void *a, const void *b) {
```

```
    Edge *e1 = (Edge *)a;
    Edge *e2 = (Edge *)b;
    return e1->weight - e2->weight;
}

// Kruskal's Algorithm to find Minimum Cost Spanning Tree
void KruskalMST(Graph *graph) {
    int vertices = graph->vertices;
    Edge result[MAX];  // To store the resulting MST
    int e = 0;         // Number of edges in MST
    int i = 0;         // Index variable for sorted edges

    // Step 1: Sort all the edges in non-decreasing order of weight
    qsort(graph->edge, graph->edges, sizeof(graph->edge[0]), compare);

    // Allocate memory for subsets
    Subset *subsets = (Subset *)malloc(vertices * sizeof(Subset));

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

    // Step 2: Pick the smallest edge and check if it forms a cycle
    while (e < vertices - 1 && i < graph->edges) {
        Edge next_edge = graph->edge[i++];

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

        // If including this edge does not form a cycle, include it in result
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }


 // Print the result
    printf("Edges in the Minimum Cost Spanning Tree:\n");
    int minimumCost = 0;
    for (i = 0; i < e; ++i) {
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
        minimumCost += result[i].weight;
    }
    printf("Minimum Cost = %d\n", minimumCost);

    free(subsets);


}

int main() {
```

```
    int vertices, edges;
    printf("Enter the number of vertices and edges: ");
    scanf("%d %d", &vertices, &edges);

    Graph *graph = (Graph *)malloc(sizeof(Graph));
    graph->vertices = vertices;
    graph->edges = edges;

    printf("Enter the edges (source destination weight):\n");
    for (int i = 0; i < edges; i++) {
        scanf("%d %d %d", &graph->edge[i].src, &graph->edge[i].dest, &graph->edge[i].weight);
    }

    KruskalMST(graph);
    free(graph);
    return 0;
}
```

- **Kruskal's Algorithm**:
  - Sort edges by weight.
  - Use a Union-Find data structure to detect cycles.
  - Add the smallest edge that does not form a cycle to the MST.
- **Steps**:
  - Sort edges in non-decreasing order.
  - Use the Union-Find technique to add edges one by one while ensuring no cycles are formed.
  - Stop when (V−1)(V - 1)(V−1) edges are added to the MST.
- **Key Components**:
  - **Edge Structure**: Stores the source, destination, and weight of edges.
  - **Union-Find**: Helps to find cycles and connect components efficiently.
  - **Sorting**: Edges are sorted by weight using `qsort`.

## Input/Output Example:

**Input:**

```mathematica
Enter the number of vertices and edges: 4 5
Enter the edges (source destination weight):
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4
```

**Output:**

```lua
Edges in the Minimum Cost Spanning Tree:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost = 19
```

9. To Implement Floyds Warshall Algorihm.

```
#include <stdio.h>

#define INF 99999  // Representation of infinity
#define V 4        // Number of vertices in the graph

// Function to print the solution matrix
void printSolution(int dist[V][V]) {
   printf("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");
   }
}

// Floyd-Warshall Algorithm
void floydWarshall(int graph[V][V]) {
   int dist[V][V], i, j, k;

   // Initial
```

```
Shortest distances between every pair of vertices:
      0      3      7      5
      2      0      6      4
    INF      1      0      7
    INF    INF      2      0
```

10. Implement N Queen's problem using Back Tracking.

```
#include <stdio.h>
#define MAX 20 // Maximum size of the chessboard
int board[MAX][MAX]; // Chessboard to represent queens' positions
// Function to print the solution
void printSolution(int n) {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         if (board[i][j])
            printf("Q ");
         else
            printf(". ");
      }
      printf("\n");
   }
   printf("\n");
}
// Function to check if a queen can be placed on board[row][col]
bool isSafe(int row, int col, int n) {
   // Check this column on the upper side
   for (int i = 0; i < row; i++) {
      if (board[i][col])
         return false;
   }
```

```
    // Check the upper-left diagonal
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j])
            return false;
    }
    // Check the upper-right diagonal
    for (int i = row, j = col; i >= 0 && j < n; i--, j++) {
        if (board[i][j])
            return false;
    }

    return true;
}
// Function to solve the N-Queens problem using backtracking
bool solveNQueens(int row, int n) {
    // Base case: If all queens are placed, return true
    if (row >= n) {
        printSolution(n);
        return true;
    }
    bool hasSolution = false;

    // Try placing a queen in all columns of the current row
    for (int col = 0; col < n; col++) {
        if (isSafe(row, col, n)) {
            // Place the queen
            board[row][col] = 1;
```

```
            // Recur to place the rest of the queens
            hasSolution = solveNQueens(row + 1, n) || hasSolution;

            // Backtrack: Remove the queen
            board[row][col] = 0;
        }
    }

    return hasSolution;
}

int main() {
    int n;

    // Input the size of the chessboard
    printf("Enter the number of queens (N): ");
    scanf("%d", &n);

    // Initialize the chessboard
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            board[i][j] = 0;
        }
    }
```

```
    // Solve the N-Queens problem
    if (!solveNQueens(0, n)) {
        printf("No solution exists for %d queens.\n", n);
    } else {
        printf("Solutions displayed above.\n");
    }

    return 0;
}
```

- The program uses **backtracking** to explore all possible configurations of queens.
- It ensures no two queens attack each other by validating column and diagonal constraints.
- The time complexity is approximately O(N!)O(N!)O(N!) for NNN-Queens.

This program works for any N≥1N \geq 1N≥1 and prints all possible solutions for the N-Queens problem

**Input**

mathematica Copy code

```
Enter the number of queens (N): 4
```

**Output**

css Copy code

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

Q . . .
. . . Q
. Q . .
. . Q .

Solutions displayed above.
```