# Analysis and Design of Algorithms Lab - BCSL404

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

Program Code

```
#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("\nEdges of 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:
1 3 4 6 2
1 7 6 9 3
5 2 8 99 45
1 44 66 33 6
12 4 3 2 0

Cost of spanning tree is=11

Edges of spanning tree are:
2 -> 1
1 -> 5
3 -> 2
1 -> 4
```

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

Program Code

```
#include<stdio.h>
#include<conio.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;
}
void 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);
    getch();
}
```

## Output

```
Enter n value:4

Enter the graph data:
4 5 2 1
7 5 9 2
1 7 6 9
0 2 8 5

Enter the souce node:4

4 -> 1 sum=0
4 -> 2 sum=2
1 -> 3 sum=4
Cost=4
```

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

## Program Code

```
#include<stdio.h>
#include<conio.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]);
}
void 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");
    }
      getch();
    }
```

## 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
```

## 3b. Design and implement C/C++ Program to find the transitive closure using Warshal’s algorithm.

### Program Code

```
#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

## 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.

<u>Program Code</u>

```
#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:4

Enter the graph data:
444 767 987 12
999 87 56 45
1 0 999 678
444 678 235 0

Enter the souce node:1

Shortest distance from 1 to 1 is 444
Shortest distance from 1 to 2 is 247
Shortest distance from 1 to 3 is 247
Shortest distance from 1 to 4 is 12
```

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

## Program Code

```
#include<stdio.h>
#include<conio.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;
        }
    }
}
void 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]);
    }
    getch();
}
```

## Output 1

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

## Output 2

Enter the n value:4

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

Topological ordering not possible

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

### Program Code

```
#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=100

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

Program Code

```
#include <stdio.h>
#define MAX 50
int p[MAX], w[MAX], x[MAX];
double maxprofit;
int n, m, i;
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 (int 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 78
Enter the maximum capacity: 100
Optimal solution for greedy method: 78.0
Solution vector for greedy method: 1  0   0   0
```

## 8. Design and implement C/C++ Program to find a subset of a given set S = {s1 , s2,…..,sn} of n positive integers whose sum is equal to a given positive integer d.

Program Code

```
#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:9

Enter the set in increasing order:1 2 3 4 5 6 7 8 9

Enter the max subset value:9

1 2 6
1 3 5
1 8
2 3 4
2 7
3 6
4 5
9

# 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.

### Step 1: Implement the Selection Sort Algorithm

The Selection Sort algorithm works by repeatedly finding the minimum element from the unsorted part and putting it at the beginning.

## Program Code

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

// Function to perform selection sort on an array
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;
    for (i = 0; i < n-1; i++)
    {
        min_idx = i;  // Assume the current element is the minimum
```

```
        for (j = i+1; j < n; j++)
        {
            if (arr[j] < arr[min_idx])
            {
                min_idx = j;  // Update min_idx if a smaller element is found
            }
        }
        // Swap the found minimum element with the current element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

// Function to generate an array of random numbers
void generateRandomNumbers(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 10000;  // Generate random numbers between 0 and 9999
    }
}

int main()
{
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);  // Read the number of elements from the user

    if (n <= 5000)
    {
        printf("Please enter a value greater than 5000\n");
        return 1;  // Exit if the number of elements is not greater than 5000
    }

    // Allocate memory for the array
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL)
    {
        printf("Memory allocation failed\n");
        return 1;  // Exit if memory allocation fails
    }

    // Generate random numbers and store them in the array
    generateRandomNumbers(arr, n);

    // Measure the time taken to sort the array
    clock_t start = clock();
    selectionSort(arr, n);
    clock_t end = clock();
```

```
    // Calculate and print the time taken to sort the array
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);

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

**Step 2: Measure Time Taken**

The above program generates n random numbers, sorts them using the Selection Sort algorithm, and measures the time taken for the sorting process.

**Step 3: Run the Program for Various Values of n**

To collect data, run the program with different values of n greater than 5000, such as 6000, 7000, 8000, etc., and record the time taken for each.

## Output

```
Enter number of elements: 6000
Time taken to sort 6000 elements: 0.031000 seconds

Enter number of elements: 7000
Time taken to sort 7000 elements: 0.034000 seconds

Enter number of elements: 8000
Time taken to sort 8000 elements: 0.047000 seconds

Enter number of elements: 9000
Time taken to sort 9000 elements: 0.052000 seconds

Enter number of elements: 10000
Time taken to sort 10000 elements: 0.077000 seconds
```

**Step 4: Plot the Results**

You can use a graphing tool like Python with matplotlib to plot the results.

```
import matplotlib.pyplot as plt
# data collected
n_values = [6000, 7000, 8000, 9000, 10000]
time_taken = [0.031000, 0.034000, 0.047000, 0.052000, 0.077000]  # replace with actual times recorded

plt.plot(n_values, time_taken, marker='o')
```

```
plt.title('Selection Sort Time Complexity')
plt.xlabel('Number of Elements (n)')
plt.ylabel('Time taken (seconds)')
plt.grid(True)
plt.show()
```


## 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.

### Step 1: Implement the Quick Sort Algorithm

Quick Sort is a divide-and-conquer algorithm that works by selecting a ‘pivot’ element and partitioning the array into elements less than and greater than the pivot.

Program Code

```
#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 pivot = arr[high]; // Pivot element
   int i = (low - 1); // Index of smaller element

   for (int j = low; j <= high - 1; j++)
   {
      if (arr[j] < pivot)
      {
         i++; // Increment index of smaller element
         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);

      // Recursively sort elements before and after partition
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}

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

int main()
{
   int n;
```

```
    printf("Enter number of elements: ");
    scanf("%d", &n); // Read the number of elements from the user

    if (n <= 5000)
    {
        printf("Please enter a value greater than 5000\n");
        return 1; // Exit if the number of elements is not greater than 5000
    }

    // Allocate memory for the array
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL)
    {
        printf("Memory allocation failed\n");
        return 1; // Exit if memory allocation fails
    }

    // Generate random numbers and store them in the array
    generateRandomNumbers(arr, n);

    // Measure the time taken to sort the array
    clock_t start = clock();
    quickSort(arr, 0, n - 1);
    clock_t end = clock();

    // Calculate and print the time taken to sort the array
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);

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

### Step 2: Measure Time Taken

This program generates n random numbers, sorts them using the Quick Sort algorithm, and measures the time taken for the sorting process.

### Step 3: Run the Program for Various Values of n

To collect data, run the program with different values of n greater than 5000, such as 6000, 7000, 8000, etc., and record the time taken for each if you didn’t get time then increase the value of n for example 20000, 40000, 60000 etc….

## 0utput

Enter number of elements: 10000
Time taken to sort 10000 elements: 0.0000 seconds

Enter number of elements: 20000
Time taken to sort 20000 elements: 0.015000 seconds

Enter number of elements: 30000
Time taken to sort 30000 elements: 0.011000 seconds

Enter number of elements: 35000
Time taken to sort 35000 elements: 0.003000 seconds

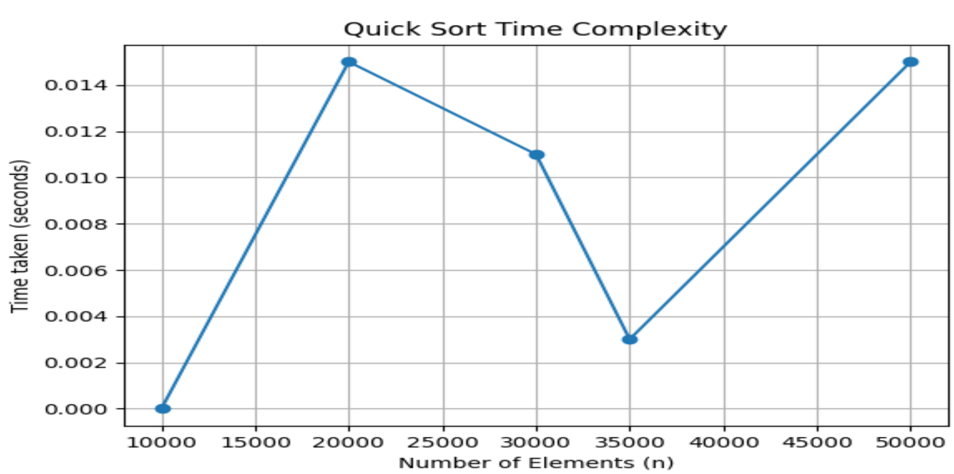
Enter number of elements: 50000
Time taken to sort 50000 elements: 0.015000 seconds

### Step 4: Plot the Results

You can use a graphing tool like Python with matplotlib to plot the results.

```
import matplotlib.pyplot as plt
#data collected
n_values = [10000, 20000, 30000, 35000, 50000]
time_taken = [0.0000, 0.015000, 0.011000, 0.003000, 0.015000]

plt.plot(n_values, time_taken, marker='o')
plt.title('Quick Sort Time Complexity')
plt.xlabel('Number of Elements (n)')
plt.ylabel('Time taken (seconds)')
plt.grid(True)
plt.show()
```



**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.**

**Step 1: Implement the Merge Sort Algorithm**

Merge Sort is a divide-and-conquer algorithm that splits the array into values, sorts each half, and then merges the sorted values.

<u>Program Code</u>

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

// Function to merge two sorted arrays
void merge(int arr[], int left, int mid, int right)
{
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    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++;
    }

    free(L);
    free(R);
}

// Function to implement Merge Sort
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 random integers
void generateRandomArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        arr[i] = rand() % 100000; // Generate random integers between 0 and 99999
}

int main()
{
    int n;
    printf("Enter the number of elements: ");
```

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

    if (n <= 5000)
    {
        printf("Please enter a value greater than 5000\n");
        return 1; // Exit if the number of elements is not greater than 5000
    }

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL)
    {
        printf("Memory allocation failed\n");
        return 1; // Exit if memory allocation fails
    }

    generateRandomArray(arr, n);

    // Repeat the sorting process multiple times to increase duration for timing
    clock_t start = clock();
    for (int i = 0; i < 1000; i++)
    {
        mergeSort(arr, 0, n - 1);
    }
    clock_t end = clock();

    // Calculate the time taken for one iteration
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC / 1000.0;

    printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);

    free(arr);
    return 0;
}
```

**Step 2: Measure Time Taken**

This program generates `n` random numbers, sorts them using the Merge Sort algorithm, and measures the time taken for the sorting process.

**Step 3: Run the Program for Various Values of `n`**

To collect data, run the program with different values of `n` greater than 5000, such as 6000, 7000, 8000, etc., and record the time taken for each.

## <u>Output</u>

Enter number of elements: 6000
Time taken to sort 6000 elements: 0.000709 seconds

Enter number of elements: 7000
Time taken to sort 7000 elements: 0.000752 seconds

Enter number of elements: 8000
Time taken to sort 8000 elements: 0.000916 seconds

Enter number of elements: 9000
Time taken to sort 9000 elements: 0.001493 seconds

Enter number of elements: 10000
Time taken to sort 10000 elements: 0.001589 seconds

Enter number of elements: 11000
Time taken to sort 11000 elements:  0.002562 seconds

Enter number of elements: 12000
Time taken to sort 12000 elements: 0.001944 seconds

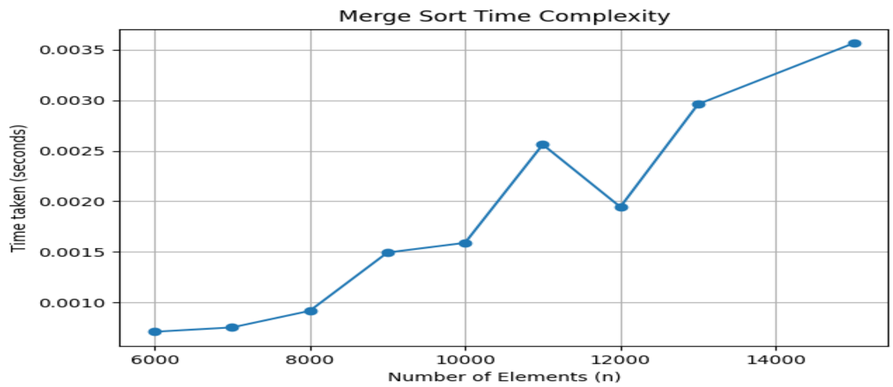
Enter number of elements: 13000
Time taken to sort 13000 elements: 0.002961 seconds

Enter number of elements: 15000
Time taken to sort 15000 elements: 0.003563 seconds

### Step 4: Plot the Results

You can use a graphing tool like Python with matplotlib to plot the results.

```
import matplotlib.pyplot as plt
# data collected n_values = [6000, 7000, 8000, 9000, 10000, 11000, 12000, 13000, 15000]
time_taken = [0.000709, 0.000752, 0.000916, 0.001493, 0.001589, 0.002562, 0.001944, 0.002961,
0.003563]
plt.plot(n_values, time_taken, marker='o')
plt.title('Merge Sort Time Complexity')
plt.xlabel('Number of Elements (n)')
plt.ylabel('Time taken (seconds)')
plt.grid(True)
plt.show()
```




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

Program Code

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

// Function to print the solution
void printSolution(int **board, int N)
{
    for (int i = 0; i < N; i++)
    {
        for (int 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)
{
    // If all queens are placed, then return true
    if (col >= N)
    {
        return true;
    }

    // Consider this column and try placing this queen in all rows one by one
    for (int i = 0; i < N; i++)
    {
        if (isSafe(board, N, i, col))
        {
            // Place this queen in board[i][col]
            board[i][col] = 1;

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

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

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

```
    return false;
}
```

**// This function solves the N Queen problem using Backtracking**
**// It mainly uses solveNQUtil() to solve the problem**
**// It returns false if queens cannot be placed, otherwise, return true and prints the placement of queens**

```
bool solveNQ(int N)
{
    int **board = (int **)malloc(N * sizeof(int *));
    for (int i = 0; i < N; i++)
    {
        board[i] = (int *)malloc(N * sizeof(int));
        for (int j = 0; j < N; j++)
        {
            board[i][j] = 0;
        }
    }

    if (!solveNQUtil(board, N, 0))
    {
        printf("Solution does not exist\n");
        for (int i = 0; i < N; i++)
        {
            free(board[i]);
        }
        free(board);
        return false;
    }

    printSolution(board, N);

    for (int i = 0; i < N; i++)
    {
        free(board[i]);
    }
    free(board);
    return true;
}

int main()
{
    int N;
    printf("Enter the number of queens: ");
    scanf("%d", &N);
    solveNQ(N);
    return 0;
}
```

Output 1:
Enter the number of queens: 4
# # Q #
Q # # #
# # # Q
# Q # #

Output 2:
Enter the number of queens: 3
Solution does not exist

********************************************************************************************