VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Analysis and Design of Algorithms

Submitted by

NEHAL A K (1BM2CS176)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
April-2024 to August-2024

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated to Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Analysis and Design of Algorithms" carried out by NEHAL A K (1BM22CS176), who is bonafide student of B.M.S. College of Engineering. It is in partial fulfillment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the academic semester April-2024 to August-2024. The Lab report has been approved as it satisfies the academic requirements in respect of an Analysis and Design of Algorithms (23CS4PCADA) work prescribed for the said degree.

Vikranth BM	Dr. Jyothi S Nayak
Associate Professor	Professor and Head
Department of CSE BMSCE, Bengaluru	Department of CSE , BMSCE, Bengaluru

Index Sheet

9	➢ Implement 0/1 Knapsack problem using dynamic programming. ➢ Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm	35-41
10	➢ From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm. ➢ Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm. ➢ Implement Fractional Knapsack using Greedy technique. ➢ Implement "N-Queens Problem" using Backtracking.	42-53

Course Outcome

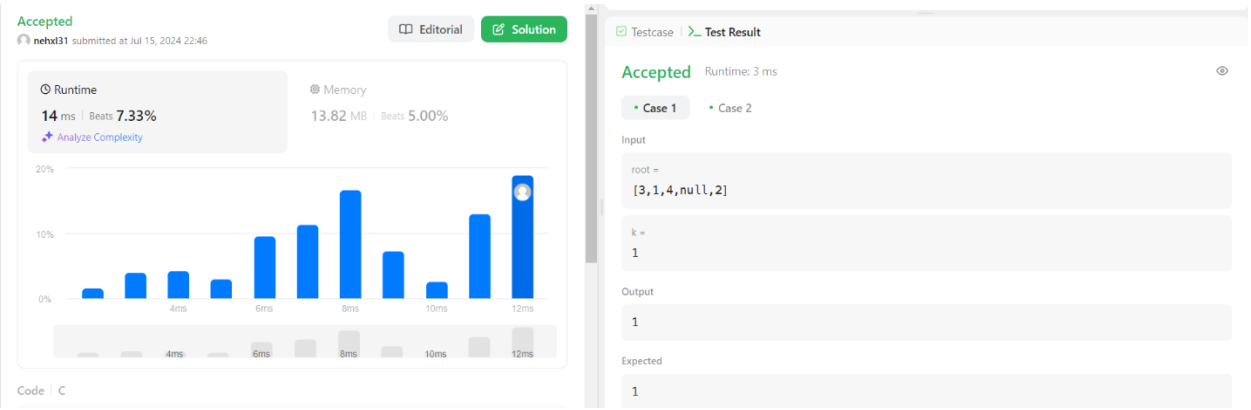
CO1	Analyze time complexity of Recursive and Non-recursive algorithms using asymptotic notations.
CO2	Apply various design techniques for the given problem.
CO3	Apply the knowledge of complexity classes P, NP, and NP-Complete and prove certain problems are NP-Complete
CO4	Design efficient algorithms and conduct practical experiments to solve problems.

LAB-1:

Leetcode-1:Kth smallest element in a BST

```c
int kthSmallest(struct TreeNode* root, int k) {
    struct TreeNode* stack[100];
    int top = -1;

    struct TreeNode* curr = root;

    int count = 0;


    while (curr != NULL || top != -1) {
        while (curr != NULL) {
            stack[++top] = curr;
            curr = curr->left;
        }

        curr = stack[top--];
        count++;
        if (count == k) {
            return curr->val;
        }
        curr = curr->right;
    }
    return -1;
}
```

Output:

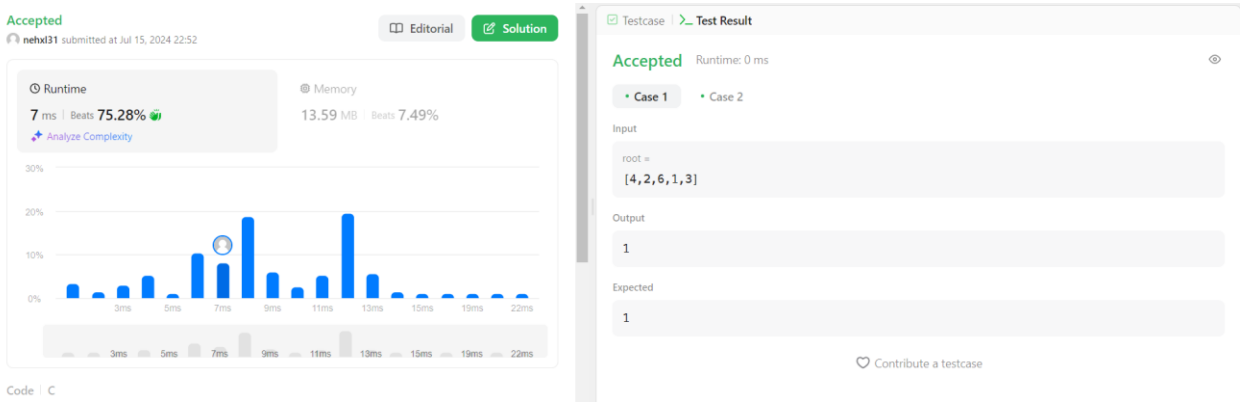


LAB-2:

Leetcode-2: Minimum Absolute difference in BST

```
void inorderTraversal(struct TreeNode* root, struct TreeNode** prevNode, int*
minDiff) {
    if (root == NULL) return;
    inorderTraversal(root->left, prevNode, minDiff);
    if (*prevNode != NULL) {
        *minDiff = (*minDiff < abs(root->val - (*prevNode)->val)) ? *minDiff :
abs(root->val - (*prevNode)->val);
    }
    *prevNode = root;

    inorderTraversal(root->right, prevNode, minDiff);
}
int getMinimumDifference(struct TreeNode* root) {
    int minDiff = INT_MAX;
    struct TreeNode* prevNode = NULL;
    inorderTraversal(root, &prevNode, &minDiff);
    return minDiff;
}
```

Output:

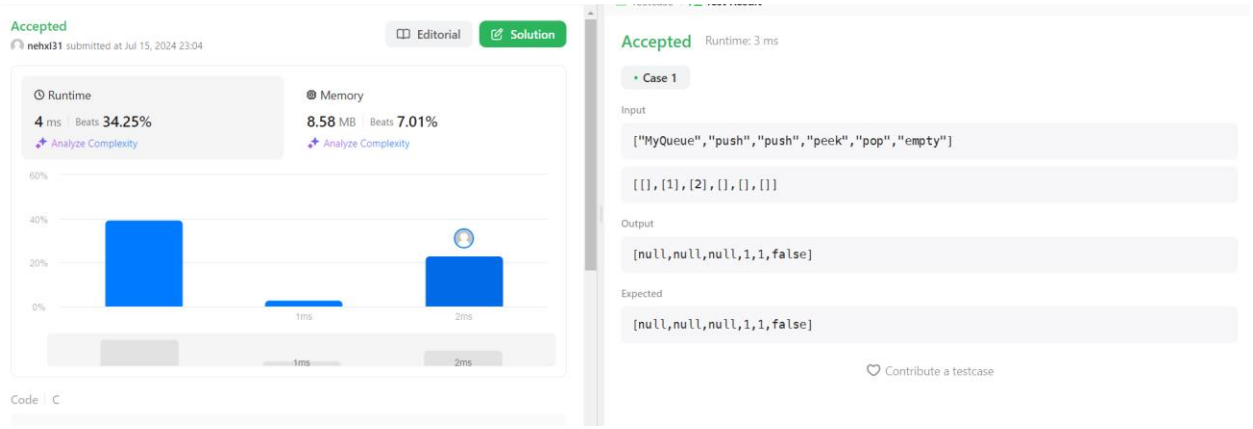


LAB-3:

Leetcode-3:Implement Queue using Stacks

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

```c
#define SIZE 1000
typedef struct {
    int stack1[SIZE];
    int top1;
    int stack2[SIZE];
    int top2;
} MyQueue;
MyQueue* myQueueCreate() {
    MyQueue* obj = (MyQueue*)malloc(sizeof(MyQueue));
    obj->top1 = -1;
    obj->top2 = -1;
    return obj;
}
void myQueuePush(MyQueue* obj, int x) {
    if (obj->top1 == SIZE - 1) {
        return;
    }
    obj->stack1[++obj->top1] = x;
}
int myQueuePop(MyQueue* obj) {
    if (obj->top2 == -1) {
        while (obj->top1 != -1) {
            obj->stack2[++obj->top2] = obj->stack1[obj->top1--];
        }
    }
    if (obj->top2 == -1) {
       return 0;
    }
    return obj->stack2[obj->top2--];
}
int myQueuePeek(MyQueue* obj) {
    if (obj->top2 == -1) {
        while (obj->top1 != -1) {
            obj->stack2[++obj->top2] = obj->stack1[obj->top1--];
        }
    }
    if (obj->top2 == -1) {
        return 0;
    }
    return obj->stack2[obj->top2];
}
bool myQueueEmpty(MyQueue* obj) {
    return obj->top1 == -1 && obj->top2 == -1;
}
void myQueueFree(MyQueue* obj) {
    free(obj);
}
```



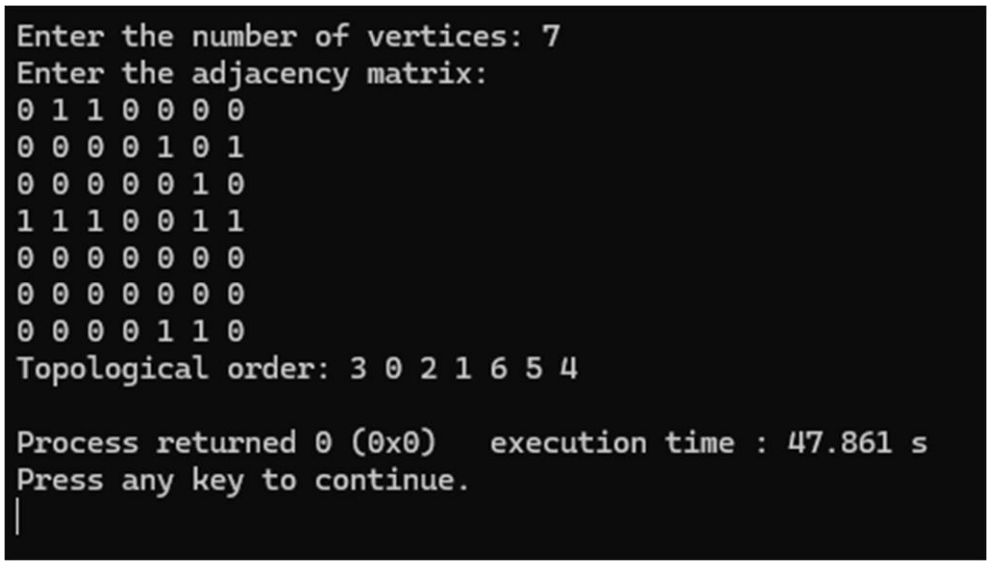
LAB-4:

Topological ordering using DFS

```
#include <stdio.h>

#include <stdlib.h>

#define MAX_VERTICES 100

int s[MAX_VERTICES] = {0};

int res[MAX_VERTICES];

int j = 0;

void DFS(int u, int n, int a[MAX_VERTICES][MAX_VERTICES]) {

   s[u] = 1;     for (int v = 0; v <

n; v++) {        if (a[u][v] == 1 &&

s[v] == 0) {

          DFS(v, n, a);

 }}

   res[j++] = u;

} int

main(

) {

   int n;
```

```
   printf("Enter the number of vertices: ");
scanf("%d", &n);
   int a[MAX_VERTICES][MAX_VERTICES];
   printf("Enter the adjacency
matrix:\n");     for (int i = 0; i < n;
i++) {        for (int j = 0; j < n; j++) {
scanf("%d", &a[i][j]);
       }
   }
   for (int u = 0; u < n; u++)
{       if (s[u] == 0) {
          DFS(u, n, a);
       }
   }
   printf("Topological order: ");
   for (int i = j - 1; i >= 0; i--) {
printf("%d ", res[i]);
   }
printf("\n;
   return 0;
}
```

Output:



Topological sorting using Source Removal Method

```
#include <stdio.h>

#include <stdlib.h>


int

st[100];

int top =

-1;


void degree(int adj[][20],

int n) {    int indegree[20];

int sum = 0;    for (int j = 0;

j < n; j++) {       sum = 0;

for (int i = 0; i < n; i++) {

sum = sum + adj[i][j];

      }

      indegree[j] = sum;

   }

   for (int i = 0; i < n;

i++) {       if
```

```
(indegree[i] == 0) {
top++;          st[top] =
i;
      }
   }

   while (top != -1) {
      int u = st[top];
top--;
      printf("%d ", u);
for (int v = 0; v < n; v++) {
if (adj[u][v] == 1) {
indegree[v]--;              if
(indegree[v] == 0) {
                top++;
st[top] = v;
             }
          }
      }
   }
} int
main(
) {
   int n;
   printf("Enter the number of nodes: ");
scanf("%d", &n);

   int adj[20][20];
```

```
    printf("Enter the adjacency
matrix:\n");     for (int i = 0; i < n;
i++) {        for (int j = 0; j < n; j++) {
scanf("%d", &adj[i][j]);
       }
    }
    printf("Topological order of nodes: ");
degree(adj, n);

    return 0;
}
```

Output:

```
Enter the number of vertices: 7
Enter the adjacency matrix:
0 1 1 0 0 0 0
0 0 0 0 1 0 1
0 0 0 0 0 1 0
1 1 1 0 0 1 1
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 1 1 0
Topological order: 3 0 2 1 6 5 4

Process returned 0 (0x0)   execution time : 47.861 s
Press any key to continue.
```

LAB-5 :

Merge Sort

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
void split(int[],int,int);
void
combine(int[],int,int,int);
void main()
{   int a[15000],n, i,j,ch,
temp;   clock_t
start,end;   while(1)
  {
 printf("\n1:For manual entry of N value and array elements");  printf("\n2:To display time
taken for sorting number of elements N in the range 500 to 14500");
 printf("\n3:To exit");
printf("\nEnter your
choice:");     scanf("%d",
&ch);     switch(ch)
   {
     case 1:  printf("\nEnter the number of elements: ");
                    scanf("%d",&n);
                   printf("\nEnter array elements: ");
                    for(i=0;i<n;i++)
                    {
                      scanf("%d",&a[i]);
                    }
                    start=clock();
                    split(a,0,n-1);
```

```
                    end=clock();
                    printf("\nSorted array is: ");
                     for(i=0;i<n;i++)
                    printf("%d\t",a[i]);
          printf("\n Time taken to sort %d numbers is %f
Secs",n,  (((double)(endstart))/CLOCKS_PER_SEC));
                   break;
    case 2:
               n=500;
while(n<=14500) {
for(i=0;i<n;i++)
                     {
                       //a[i]=random(1000);
              a[i]=n-i;
                     }
                start=clock();
                split(a,0,n-1);
         //Dummy loop to create delay
              for(j=0;j<90000000;j++){ temp=38/600;}
end=clock();
printf("\n Time taken to sort %d numbers is %f
Secs",n,  (((double)(endstart))/CLOCKS_PER_SEC));  n=n+1000;
                } break;
  case 3: exit(0);
  }
  getchar();
   }
```

```
} void split(int a[],int
low,int high)
{  int
mid;
if(low<
high)
 {
  mid=(low+high)/2;
  split(a,low,mid);
split(a,mid+1,high);
combine(a,low,mid,high);
 }
}

void combine(int a[],int low,int mid,int high)
{  int
c[15000],i,j,
k;
 i=k=low;
 j=mid+1;
while(i<=mid&&j<=high)
 {
if(a[i]<
a[j])   {
c[k]=a[
i];
++k;
   ++i;
}
```

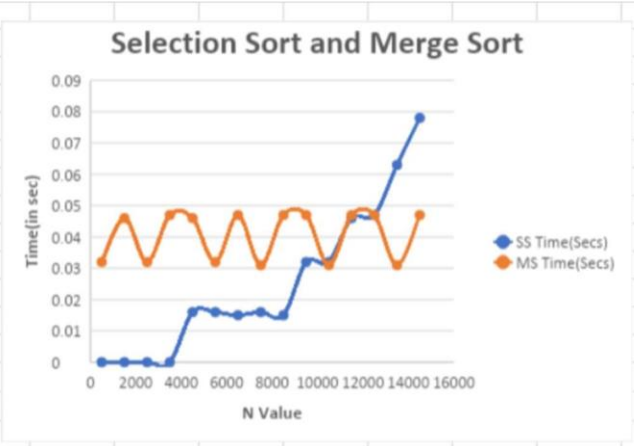
```
else
{
c[k]=
a[j];
++k;
  ++j; } }
 if(i>mid)
 {
  while(j<=high)
  {
c[k]=a[
j];
++k;
  ++j;
  } }
if(j>
high
) {
  while(i<=mid)
  {
c[k]=a[
i];
++k;
++i;  }
 } for(i=low;i<=high;i++)
 {
a[i]=c
[i];
 } }
```

Output:

```
Enter the number of elements: 4

Enter array elements: 44 33 22 11

Sorted array is: 11      22      33      44
 Time taken to sort 4 numbers is 0.000000 Secs
1:For manual entry of N value and array elements
2:To display time taken for sorting number of elements N in the range 500 to 14500
3:To exit
Enter your choice:2

 Time taken to sort 500 numbers is 0.032000 Secs
 Time taken to sort 1500 numbers is 0.046000 Secs
 Time taken to sort 2500 numbers is 0.032000 Secs
 Time taken to sort 3500 numbers is 0.047000 Secs
 Time taken to sort 4500 numbers is 0.046000 Secs
 Time taken to sort 5500 numbers is 0.032000 Secs
 Time taken to sort 6500 numbers is 0.047000 Secs
 Time taken to sort 7500 numbers is 0.031000 Secs
 Time taken to sort 8500 numbers is 0.047000 Secs
 Time taken to sort 9500 numbers is 0.047000 Secs
 Time taken to sort 10500 numbers is 0.031000 Secs
 Time taken to sort 11500 numbers is 0.047000 Secs
 Time taken to sort 12500 numbers is 0.047000 Secs
 Time taken to sort 13500 numbers is 0.031000 Secs
 Time taken to sort 14500 numbers is 0.047000 Secs
1:For manual entry of N value and array elements
2:To display time taken for sorting number of elements N in the range 500 to 14500
3:To exit
Enter your choice:3

Process returned 0 (0x0)   execution time : 95.566 s
Press any key to continue.
```

Graph:

N value	SS Time(Secs)	MS Time(Secs)
500	0	0.032
1500	0	0.046
2500	0	0.032
3500	0	0.047
4500	0.016	0.046
5500	0.016	0.032
6500	0.015	0.047
7500	0.016	0.031
8500	0.015	0.047
9500	0.032	0.047
10500	0.032	0.031
11500	0.046	0.047
12500	0.047	0.047
13500	0.063	0.031
14500	0.078	0.047



LAB 6:

Quick Sort:

```
#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

#include<time.h>
```

```
void swap(int *a, int
*b){    int temp = *a;
*a = *b;
   *b = temp;
}

int partition(int a[], int low, int
high){    int pivot = a[low];    int i
= low + 1;    int j = high;

   while (i <= j) {        while (i <=
j && a[i] <= pivot)
          i++;
       while (i <= j && a[j] > pivot)

j--;
if (i < j)
          swap(&a[i], &a[j]);
   }

   swap(&a[low], &a[j]);
   return j;
}

void quicksort(int a[],int low,int high){
   int point;    if(low<high){
point=partition(a,low,high);
quicksort(a,low,point-1);
quicksort(a,point+1,high);
```

```
   }

}
void main(){
   int a[15000], n, i, j, ch, temp;
clock_t start, end;

   while (1) {        printf("\n 1:For manual entry of N value and
array elements");
      printf("\n 2:To display time taken for sorting number of elements N in the range 500 to
14500");
      printf("\n 3:To exit");
      printf("\nEnter your choice:");
scanf("%d", &ch);

      switch (ch) {
         case 1:
            printf("\nEnter the number of
elements: ");              scanf("%d", &n);
printf("Enter array elements: ");              for (i
= 0; i < n; i++) {                 scanf("%d", &a[i]);
            }
            start = clock();
quicksort(a,0,n-1);              end
= clock();
printf("\nSorted array is: ");
for (i = 0; i < n; i++) {
printf("%d\t", a[i]);
```

```
            }
            printf("\nTime taken to sort %d numbers is %f Secs", n, (((double)(end - start)) /
CLOCKS_PER_SEC));
            break;

         case 2:
            n = 500;
while (n <= 14500) {
for (i = 0; i < n; i++) {
a[i] = n - i;   }
               start = clock();
quicksort(a, 0, n - 1);
for (j = 0; j < 500000; j++) {
temp = 38 / 600;
               }
               end = clock();
               printf("\nTime taken to sort %d numbers is %f Secs", n, (((double)(end - start)) /
CLOCKS_PER_SEC));
               n = n + 1000;
            }
break;
         case 3:
            exit(0);
      }
      getchar();
   }
}
```

Output:

```
 1:For manual entry of N value and array elements
 2:To display time taken for sorting number of elements N in the range 500 to 14500
 3:To exit
Enter your choice:1

Enter the number of elements: 8
Enter array elements: 5 3 1 9 8 2 4 7

Sorted array is: 1       2       3       4       5       7       8       9
Time taken to sort 8 numbers is 0.000000 Secs
 1:For manual entry of N value and array elements
 2:To display time taken for sorting number of elements N in the range 500 to 14500
 3:To exit
Enter your choice:2

Time taken to sort 500 numbers is 0.000000 Secs
Time taken to sort 1500 numbers is 0.016000 Secs
Time taken to sort 2500 numbers is 0.000000 Secs
Time taken to sort 3500 numbers is 0.000000 Secs
Time taken to sort 4500 numbers is 0.000000 Secs
Time taken to sort 5500 numbers is 0.015000 Secs
Time taken to sort 6500 numbers is 0.000000 Secs
Time taken to sort 7500 numbers is 0.000000 Secs
Time taken to sort 8500 numbers is 0.031000 Secs
Time taken to sort 9500 numbers is 0.016000 Secs
Time taken to sort 10500 numbers is 0.031000 Secs
Time taken to sort 11500 numbers is 0.031000 Secs
Time taken to sort 12500 numbers is 0.047000 Secs
Time taken to sort 13500 numbers is 0.032000 Secs
Time taken to sort 14500 numbers is 0.062000 Secs
 1:For manual entry of N value and array elements
 2:To display time taken for sorting number of elements N in the range 500 to 14500
 3:To exit
Enter your choice:3

Process returned 0 (0x0)   execution time : 33.657 s
Press any key to continue.
```

Graph:

N-Value	Time Taken(in sec)
500	0
1500	0.016
2500	0
3500	0
4500	0
5500	0.015
6500	0
7500	0
8500	0.031
9500	0.016
10500	0.031
11500	0.031
12500	0.047
13500	0.032
14500	0.062

LAB 7:

Johnson Trotter Algorithm:

```
#include <stdio.h>

#include <stdlib.h>

int flag = 0; int

swap(int *a,int

*b) {    int t

=*a;    *a = *b;
```

```
   *b = t; } int search(int arr[],int
num,int mobile)
{
in
t
g;
   for(g=0; g<num; g++) {
      if(arr[g] == mobile)
return g+1;
      else          {
flag++;

}
}
   return  -1;  }  int  find_Moblie(int
arr[],int d[],int num)
{    int
mobile = 0;
int mobile_p
=0;
   int i;
   for(i=0; i<num; i++) {
      if((d[arr[i]-1] == 0) && i != 0) {
if(arr[i]>arr[i-1] && arr[i]>mobile_p) {
            mobile = arr[i];
mobile_p = mobile;
         }          else          {
flag++;
         }
```

```
      } else if((d[arr[i]-1] == 1) & i != num-1) {
if(arr[i]>arr[i+1] && arr[i]>mobile_p) {
            mobile = arr[i];
mobile_p = mobile;
         }        else        {
flag++;              }         }
else {          flag++;
      }
   }
   if((mobile_p == 0) && (mobile == 0))       return 0;
   else       return mobile;
}

void permutations(int arr[],int d[],int num)
{
   int i;
   int mobile = find_Moblie(arr,d,num);    int pos =
search(arr,num,mobile);    if(d[arr[pos-1]-1]==0)
swap(&arr[pos-1],&arr[pos-2]);
   else
      swap(&arr[pos-1],&arr[pos]);
   for(int i=0; i<num; i++) {       if(arr[i] >
mobile) {          if(d[arr[i]-1]==0)
d[arr[i]-1] = 1;          else              d[arr[i]-
1] = 0;
      }
   }
   for(i=0; i<num; i++) {
printf(" %d ",arr[i]);
```

```
   }
}

int factorial(int k)
{
int f
= 1;
int i
= 0;
   for(i=1; i<k+1; i++) {
      f =
f*i;    }
return f;
} int
main() {
int num
=0;

int i;
int j;
   int z =0;    printf("Johnson trotter algorithm to find all permutations of
given numbers \n");    printf("Enter the number\n");    scanf("%d",&num);
int arr[num],d[num];

   z = factorial(num);    printf("total
permutations = %d",z);    printf("\nAll
possible permutations are: \n");    for(i=0;
i<num; i++) {
```

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

Output:

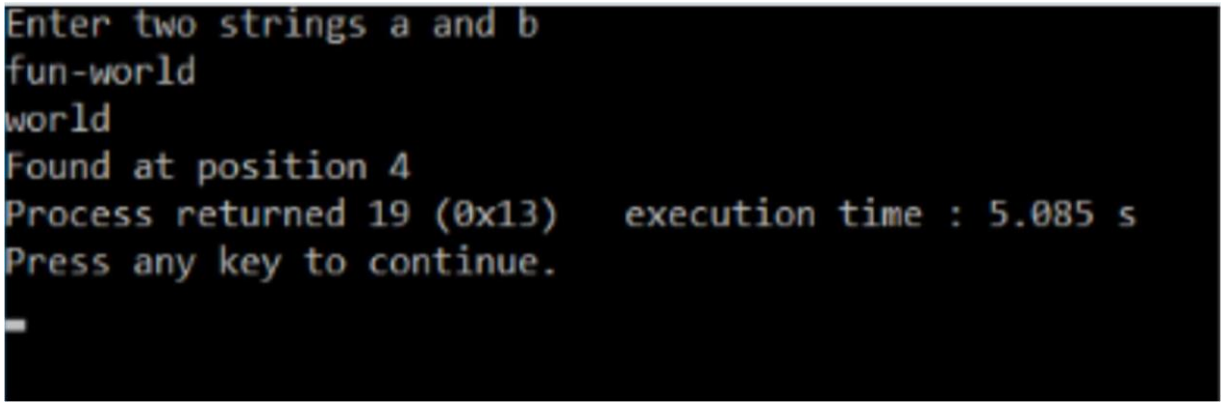
```
Johnson trotter algorithm to find all permutations of given numbers
Enter the number
3
total permutations = 6
All possible permutations are:
 1  2  3
 1  3  2
 3  1  2
 3  2  1
 2  3  1
 2  1  3
```

Pattern Matching Program:

```
#include<stdio.h>
#include<string.h> void main() {
char a[50], b[50];    printf("Enter
two strings a and b\n");
```

```
scanf("%s",a);    scanf("%s",b);
int n = strlen(a);    int m =
strlen(b);    int j, flag = 0;

   for(int i = 0; i <= n-m; i++)
   {
j =
0;
      while(j < m && a[i+j] == b[j])
         j = j + 1;

         if(j ==
m)          {
            printf("%d\n",j);
flag = 1;
printf("%d",i);
break;

}   }
   if(flag == 0)
printf("String not
matched");

}
```

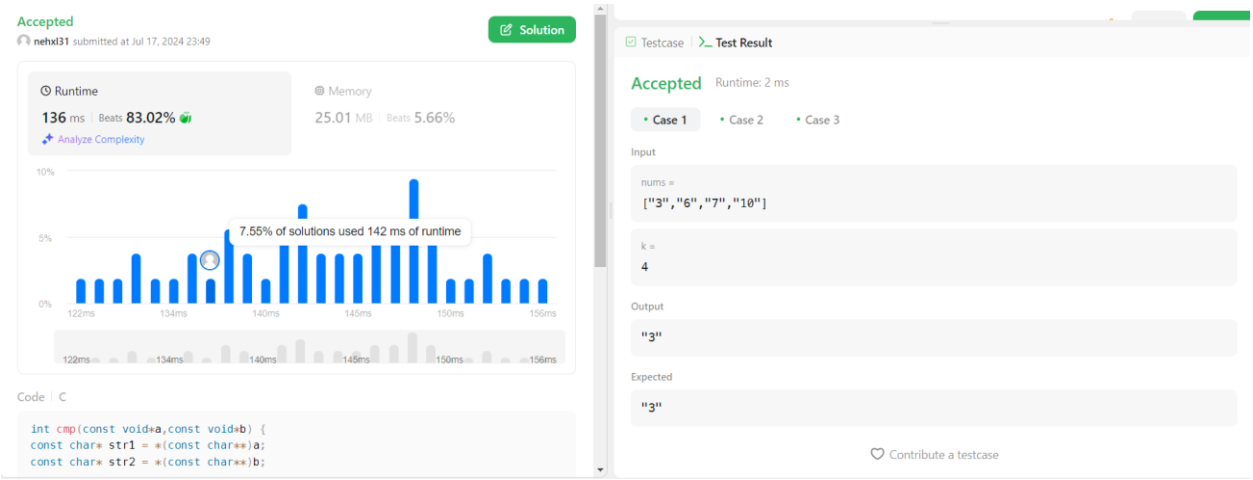
Output:



3.Leetcode:find the Kth largest Integer in the array

Output:

```
int cmp(const void*a,const void*b) {
const char* str1 = *(const char**)a;
const char* str2 = *(const char**)b;

if (strlen(str1) == strlen(str2)) {
return strcmp(str1, str2);
}
return strlen(str1) - strlen(str2);
}

char* kthLargestNumber(char** nums, int numsSize, int k) {
qsort(nums, numsSize, sizeof(char*), cmp);
return nums[numsSize - k];
}
```



LAB 8:

Heap Sort:

```
#include<stdio.h>

#include<stdlib.h>

#include<time.h>


void bottom_up_heapify(int n, int

a[]); void swap(int *a, int *b); void

heap_sort(int n, int a[]);


void bottom_up_heapify(int n, int a[]) {

   int p, item, c;     for (p =

(n-1)/2; p >= 0; p--) {

item = a[p];        c = 2 * p +

1;


       while (c <= n - 1) {           if (c + 1 <= n

- 1 && a[c] < a[c + 1]) {

            c++;

         }
```

```
        if (item >=
a[c]) {
break;          }
        a[p] =
a[c];           p =
c;          c = 2
* p + 1;       }
a[p] = item;
   }
}

void swap(int *a, int
*b) {    int temp =
*a;    *a = *b;
   *b = temp;
}

void heap_sort(int n, int a[]) {
   int i;
   bottom_up_heapify(n, a);

   for (i = n - 1; i >= 0; i--) {
swap(&a[0], &a[i]);
bottom_up_heapify(i, a);
   }
} int main() {    int
a[15000], n, i, j, ch, temp;
clock_t start, end;
while (1) {
```

```
        printf("\n1: For manual entry of N value and array elements");
        printf("\n2: To display time taken for sorting number of elements N in the range 500 to
14500");
        printf("\n3: To exit");
printf("\nEnter your choice:
");        scanf("%d", &ch);
switch (ch) {
            case 1:
                printf("\nEnter the number of
elements: ");                scanf("%d", &n);
printf("\nEnter array elements: ");                for
(i = 0; i < n; i++) {                    scanf("%d", &a[i]);
                }
                start = clock();
heap_sort(n, a);                end =
clock();
printf("\nSorted array is: ");
for (i = 0; i < n; i++)
printf("%d\t", a[i]);
                printf("\nTime taken to sort %d numbers is %f Secs", n, (((double)(end - start)) /
CLOCKS_PER_SEC));
                break;

            case 2:
                n = 500;                while
(n <= 14500) {                    for (i
= 0; i < n; i++) {
//a[i] = rand() % 1000;
a[i] = n - i;
```

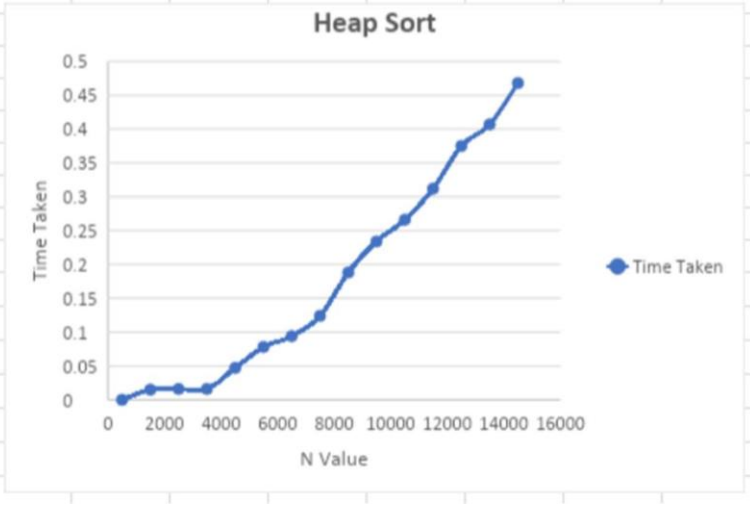
```
            }
            start = clock();
heap_sort(n, a);
            //Dummy loop to create
delay               for (j = 0; j < 500000;
j++) {                 temp = 38 / 600;
            }
end = clock();
            printf("\nTime taken to sort %d numbers is %f Secs", n, (((double)(end - start)) /
CLOCKS_PER_SEC));
            n = n + 1000;
          }
break;
case 3:
          exit(0);
     }
     getchar();
  }
  return 0;
}
```

Output:

```
1: For manual entry of N value and array elements
2: To display time taken for sorting number of elements N in the range 500 to 14500
3: To exit
Enter your choice: 1

Enter the number of elements: 11

Enter array elements: 5 35 25 45 30 55 25 45 50 10 30

Sorted array is: 5      10      25      25      30      30      35      45      45      50      55
Time taken to sort 11 numbers is 0.000000 Secs
1: For manual entry of N value and array elements
2: To display time taken for sorting number of elements N in the range 500 to 14500
3: To exit
Enter your choice: 2

Time taken to sort 500 numbers is 0.000000 Secs
Time taken to sort 1500 numbers is 0.015000 Secs
Time taken to sort 2500 numbers is 0.016000 Secs
Time taken to sort 3500 numbers is 0.016000 Secs
Time taken to sort 4500 numbers is 0.047000 Secs
Time taken to sort 5500 numbers is 0.078000 Secs
Time taken to sort 6500 numbers is 0.094000 Secs
Time taken to sort 7500 numbers is 0.124000 Secs
Time taken to sort 8500 numbers is 0.188000 Secs
Time taken to sort 9500 numbers is 0.234000 Secs
Time taken to sort 10500 numbers is 0.266000 Secs
Time taken to sort 11500 numbers is 0.312000 Secs
Time taken to sort 12500 numbers is 0.375000 Secs
Time taken to sort 13500 numbers is 0.407000 Secs
Time taken to sort 14500 numbers is 0.468000 Secs
1: For manual entry of N value and array elements
2: To display time taken for sorting number of elements N in the range 500 to 14500
3: To exit
Enter your choice: 3

Process returned 0 (0x0)   execution time : 29.531 s
Press any key to continue.
```

Graph:

N Value	Time Taken
500	0
1500	0.015
2500	0.016
3500	0.016
4500	0.047
5500	0.078
6500	0.094
7500	0.124
8500	0.188
9500	0.234
10500	0.266
11500	0.312
12500	0.375
13500	0.407
14500	0.468



Floyd's Algorithm:

```
#include <stdio.h>

#define V 5

#define INF 99999

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

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

   /* Add all vertices one by one to
the set of intermediate vertices.
---> Before start of an iteration, we
have shortest distances between
all       pairs of vertices such that
the shortest       distances consider
only the       vertices in set {0, 1, 2,
.. k-1} as       intermediate vertices.
----> After the end of an iteration,
vertex no. k is added to the set of
intermediate vertices and the set
becomes {0, 1, 2, .. k} */     for (k =
0; k < V; k++) {
      // Pick all vertices as source one by
one        for (i = 0; i < V; i++) {
          // Pick all vertices as destination for the
          // above picked
source           for (j = 0; j <
V; j++) {
             // If vertex k is on the shortest path from              // i to j, then update the value of
             // dist[i][j]              if
(dist[i][k] + dist[k][j] < dist[i][j])
dist[i][j] = dist[i][k] + dist[k][j];
          }
```

```
        }
    }

    printSolution(dist);
} void printSolution(int
dist[][V])
{
    printf(
        "The following matrix shows the shortest distances"
        " between every pair of
vertices \n");     for (int i = 0; i < V;
i++) {         for (int j = 0; j < V; j++) {
if (dist[i][j] == INF)
printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
} int main() {     int graph[V][V] = { { 0,
4, INF, 5,INF },
                    { INF, 0, 1, INF,6 },
                    { 2,INF, 0, 3,INF },
                   { INF, INF, 1, 0,2 } ,
{1,INF,INF,4,0}};
    floydWarshall(graph);
    return 0;
}
```

Output:

```
The following matrix shows the shortest distances between every pair of vertices
      0      4      5      5      7
      3      0      1      4      6
      2      6      0      3      5
      3      7      1      0      2
      1      5      5      4      0

Process returned 0 (0x0)   execution time : 0.153 s
Press any key to continue.
```

LAB 9:

Knapsack using Dynamic Programming:

```
#include <stdio.h>


#define N 4

#define CAPACITY 7


int max(int a,

int b) {

if(a>b){

return a;

   }

   return b;


}


void knapsack(int weights[], int profits[]) {

   int i, w;

   int dp[N + 1][CAPACITY + 1];


   for (i = 0; i <= N; i++) {

      for (w = 0; w <= CAPACITY; w++) {
```

```
        if (i == 0 || w == 0)               dp[i][w] = 0;           else if (weights[i - 1]
<= w)               dp[i][w] = max(profits[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i -
1][w]);
          else
             dp[i][w] = dp[i - 1][w];
       }
   }

   int maxProfit = dp[N][CAPACITY];
printf("Maximum profit: %d\n", maxProfit);

   int
selectedObjects[N];
int k = N, c = CAPACITY;
while (k > 0 && c > 0) {
       if (dp[k][c] != dp[k - 1][c])
{           selectedObjects[k - 1]
= 1;           c =c- weights[k -
1];
       } else {
selectedObjects[k - 1] = 0;

}
k--;
   }

   printf("\nTable Values (DP Table):\n");
   for (i = 0; i <= N; i++) {
      for (w = 0; w <= CAPACITY; w++) {
```

```
            printf("%d\t", dp[i][w]);
        }
        printf("\n");
    }

    printf("\nObjects selected in the knapsack:\n");     for (i = 0; i < N; i++) {
if (selectedObjects[i] == 1)            printf("Object %d (Weight: %d, Profit:
%d)\n", i + 1, weights[i], profits[i]);
    }
}

int main() {
int
weights[N];
int
profits[N];

    printf("Enter              the
weights:\n");           for(int
i=0;i<N;i++){
scanf("%d",&weights[i]);
    }

    printf("Enter           the
profits:\n");           for(int
i=0;i<N;i++){
scanf("%d",&profits[i]);
    }
```

```
    printf("Knapsack Capacity: %d\n", CAPACITY);
    printf("Objects:\n");
    for (int i = 0; i < N; i++) {        printf("Object %d - Weight: %d, Profit: %d\n", i
+ 1, weights[i], profits[i]);
    }
    knapsack(weights, profits);


    return 0;
}
```

Output:

```
Enter the weights:
1 3 4 5
Enter the profits:
1 4 5 7
Knapsack Capacity: 7
Objects:
Object 1 - Weight: 1, Profit: 1
Object 2 - Weight: 3, Profit: 4
Object 3 - Weight: 4, Profit: 5
Object 4 - Weight: 5, Profit: 7
Maximum profit: 9

Table Values (DP Table):
0       0       0       0       0       0       0       0
0       1       1       1       1       1       1       1
0       1       1       4       5       5       5       5
0       1       1       4       5       6       6       9
0       1       1       4       5       7       8       9

Objects selected in the knapsack:
Object 2 (Weight: 3, Profit: 4)
Object 3 (Weight: 4, Profit: 5)

Process returned 0 (0x0)   execution time : 14.113 s
Press any key to continue.
```

Prims Algorithm:

```
#include <stdio.h>
#include <limits.h>
#define MAX 100
void prims(int n, int cost[MAX][MAX], int INF) {
int S[MAX], d[MAX], p[MAX], T[MAX][2];
```

```
   int i, j, min, source, sum = 0, k = 0, u;
min = INF;     source = 0;     for (i = 0; i < n;
i++) {        for (j = 0; j < n; j++) {            if
(cost[i][j] != 0 && cost[i][j] < min) {
min = cost[i][j];                source = i;
          }


}
}
   for (i = 0; i < n;
i++) {        S[i] = 0;
d[i] =
cost[source][i];
p[i] = source;     }
   S[source] = 1;
for (i = 1; i < n; i++)
{        min = INF;
       u = -1;        for (j = 0; j < n;
j++) {           if (S[j] == 0 && d[j]
<= min) {
              min = d[j];
              u = j;
          }
       }
       T[k][0] = u;
T[k][1] = p[u];
k++;
```

```
        sum += cost[u][p[u]];        S[u] =
1;        for (j = 0; j < n; j++) {            if
(S[j] == 0 && cost[u][j] < d[j]) {
                d[j] = cost[u][j];
p[j] = u;
            }
        }
    }

    if (sum >= INF) {        printf("Spanning
tree does not exist.\n");
    } else {
        printf("Spanning tree exists and MST is:\n");
        for (i = 0; i < n - 1; i++) {
printf("%d -> %d\n", T[i][1], T[i][0]);
        }
        printf("The cost of Spanning tree is MST is %d\n", sum);
    }
}

int main() {     int n,
cost[MAX][MAX], i, j;
int INF = INT_MAX;

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

    printf("Enter the cost adjacency
matrix:\n");     for (i = 0; i < n; i++) {        for
(j = 0; j < n; j++) {            scanf("%d",
```

```
&cost[i][j]);           if (cost[i][j] == 9999 ) {
cost[i][j] = INF;
        }
      }
   }
   prims(n, cost, INF);
return 0;
}
```

Output:

```
Enter the number of vertices: 5
Enter the cost adjacency matrix:
0 5 15 20 9999
5 0 25 9999 9999
15 25 0 30 37
20 9999 30 0 35
9999 9999 37 35 0
Spanning tree exists and MST is:
0 -> 1
0 -> 2
0 -> 3
3 -> 4
The cost of Spanning tree is MST is 75

Process returned 0 (0x0)   execution time : 48.002 s
Press any key to continue.
```

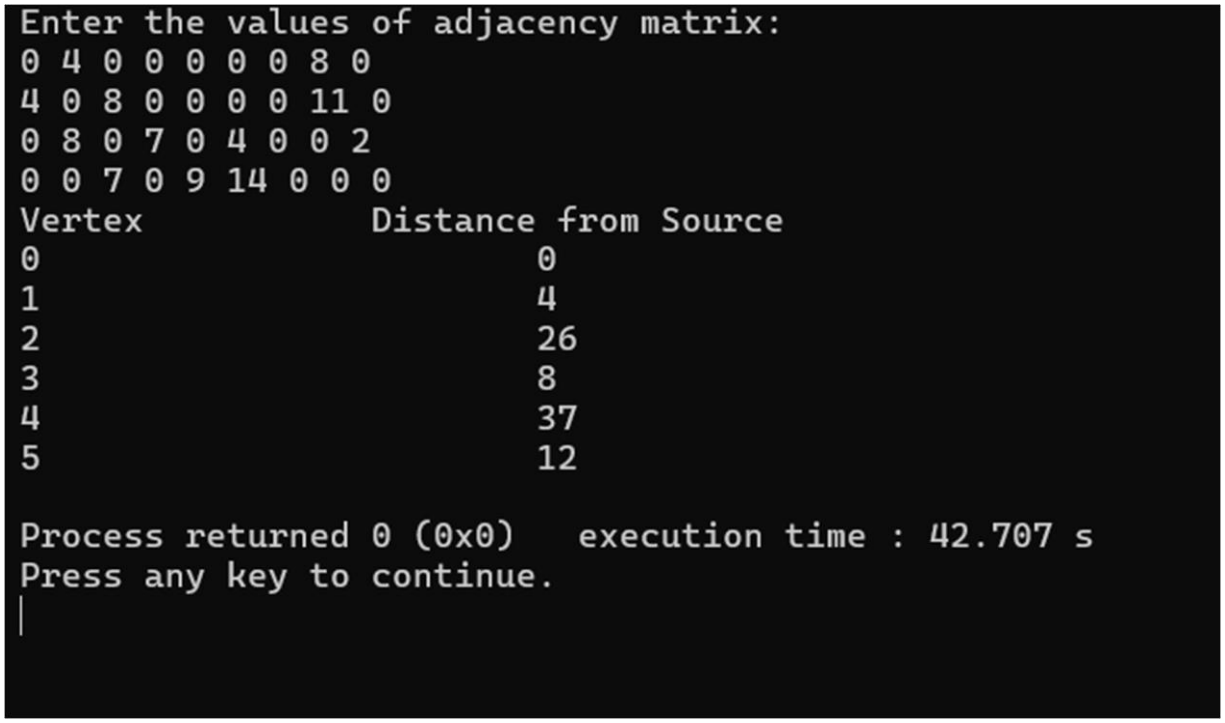
LAB 10:

Dijkstra's Algorithm:

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h> #define V 6 int
minDistance(int dist[], bool
sptSet[])
{
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)        if
(!sptSet[v] && dist[v] <= min) {
min = dist[v];            min_index =
v;
       }

   return min_index;

void printSolution(int dist[])
{         printf("Vertex  \t\t  Distance  from
Source\n");       for  (int  i  =  0;  i  <  V;  i++)
printf("%d \t\t\t %d\n", i, dist[i]);
} void dijkstra(int graph[V][V],
int src)
{    int
dist[V];
bool
sptSet[V];
for (int i =
```

```
0; i < V; i++)
dist[i] =
INT_MAX,
sptSet[i] =
false;
   dist[src] = 0;     for (int count = 0;
count < V - 1; count++) {        int u =
minDistance(dist, sptSet);        sptSet[u]
= true;

       for (int v = 0; v < V; v++)           if
(!sptSet[v] && graph[u][v]               &&
dist[u] != INT_MAX               && dist[u]
+ graph[u][v] < dist[v])               dist[v]
= dist[u] + graph[u][v];
   }

   printSolution(dist);
} int main() {     int graph[V][V];     printf("Enter
the values of adjacency matrix:\n");     for (int
i = 0; i < V; i++) {         for (int j = 0; j < V; j++) {
scanf("%d", &graph[i][j]);
       }
   }

   dijkstra(graph, 0);

   return 0;
}
```

Output:



Kruskal's Algorithm:

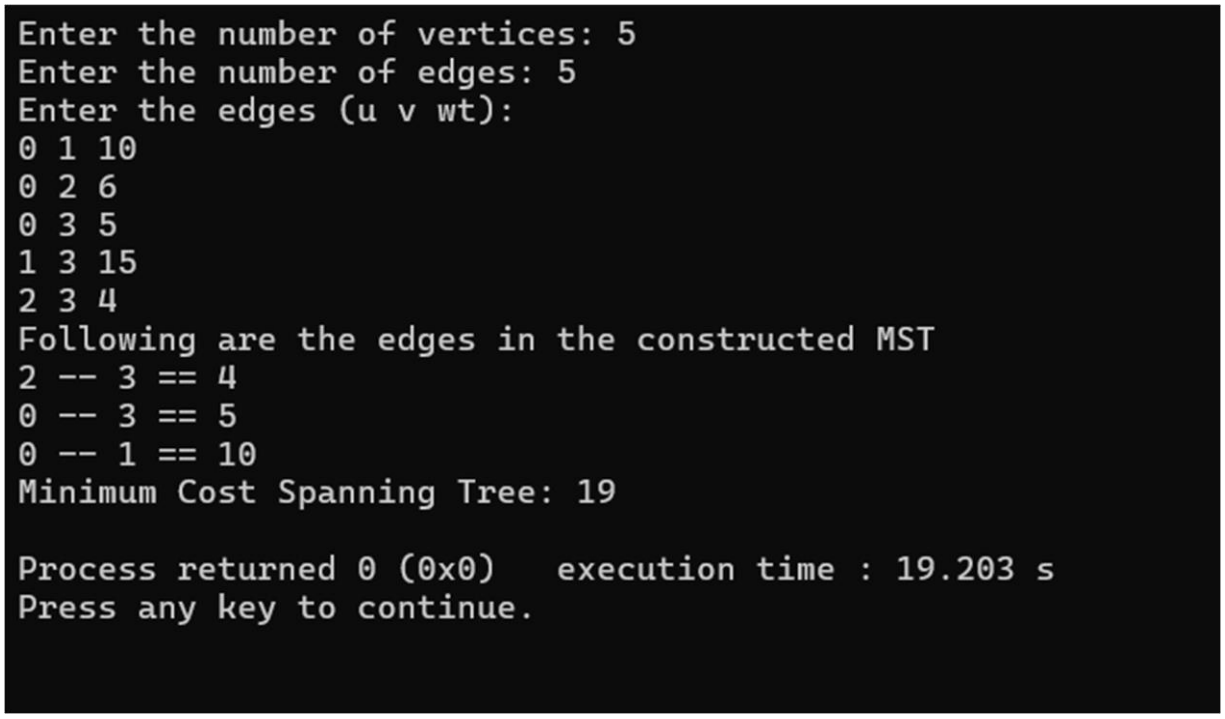
```
#include <stdio.h> #include <stdlib.h> int
comparator(const void* p1, const void*
p2)
{    const int(*x)[3]
= p1;    const
int(*y)[3] = p2;

   return (*x)[2] - (*y)[2];
} void makeSet(int parent[], int rank[],
int n)
{
   for (int i = 0; i < n;
i++) {       parent[i] = i;
rank[i] = 0;
```

```
   }
} int findParent(int parent[], int
component)
{    if (parent[component] != component) {
parent[component] = findParent(parent,
parent[component]);
   }
   return parent[component];
} void unionSet(int u, int v, int parent[], int
rank[])
{    u =
findParent(parent, u);
v = findParent(parent,
v);    if (rank[u] <
rank[v]) {
parent[u] = v;
   } else if (rank[u] >
rank[v]) {        parent[v] =
u;    } else {        parent[v]
= u;        rank[u]++;
   }
}

void kruskalAlgo(int n, int edges[][3], int e)
{    qsort(edges, e, sizeof(edges[0]),
comparator);    int parent[n];

   int rank[n];    makeSet(parent, rank, n);    int minCost
= 0;    printf("Following are the edges in the constructed
MST\n");
```

```
   for (int i = 0; i < e; i++) {        int u =
edges[i][0];        int v = edges[i][1];        int
wt = edges[i][2];        int parent_u =
findParent(parent, u);        int parent_v =
findParent(parent, v);        if (parent_u !=
parent_v) {            unionSet(parent_u,
parent_v, parent, rank);            minCost +=
wt;            printf("%d -- %d == %d\n", u, v,
wt);
       }
   }
   printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

int
main(
) {
   int n, e;
   printf("Enter the number of
vertices: ");    scanf("%d", &n);
printf("Enter the number of edges:
");    scanf("%d", &e);    int
edges[e][3];    printf("Enter the
edges (u v wt):\n");    for (int i = 0; i
< e; i++) {
       scanf("%d %d %d", &edges[i][0], &edges[i][1], &edges[i][2]);
   }
   kruskalAlgo(n, edges, e);
   return 0;
```

}

Output:



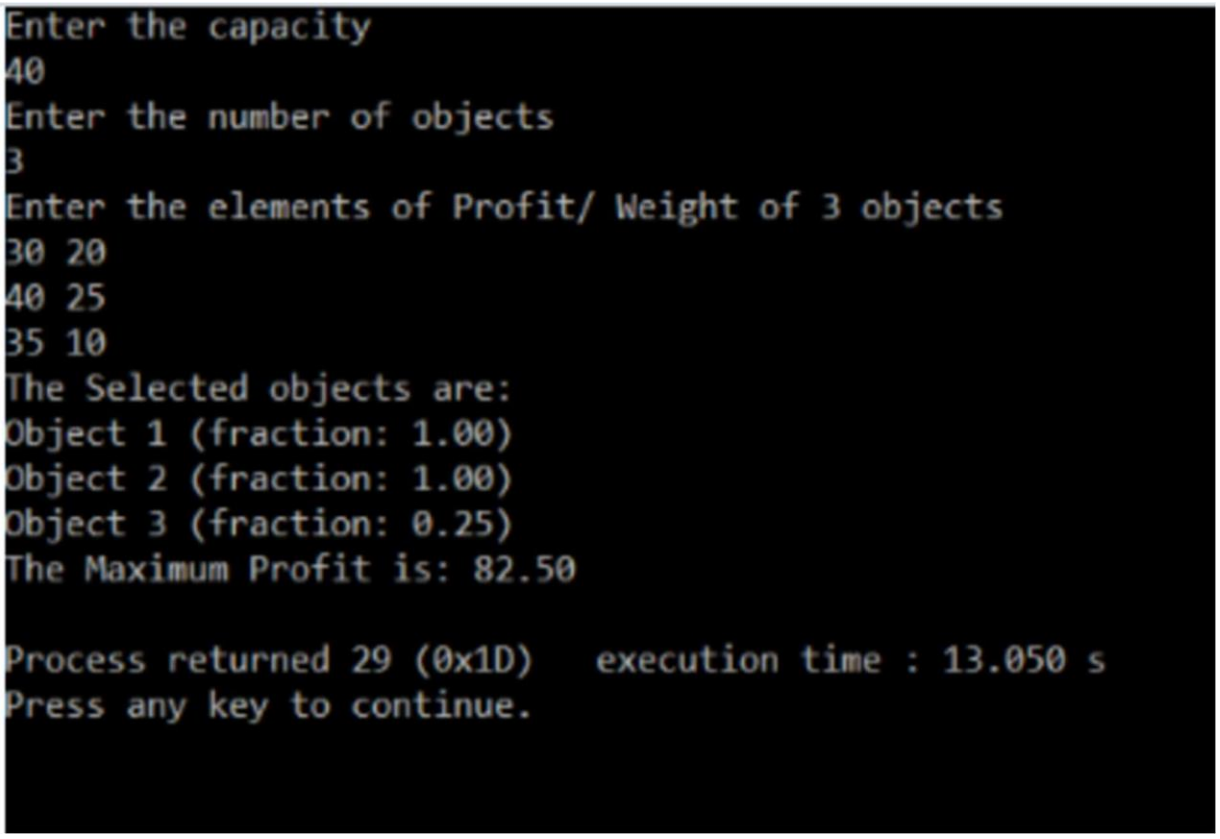
Greedy knapsack Problem:

```
#include <stdio.h>
void main() {
   int n;    float m;
printf("Enter the
capacity\n");    scanf("%f",
&m);

   printf("Enter the number of objects\n");    scanf("%d", &n);
printf("Enter the elements of Profit/ Weight of %d objects\n",
n);
   float w[n], p[n], x[n];
float ratio[n];    for (int i = 0;
i < n; i++) {       scanf("%f
%f", &p[i], &w[i]);
```

```
      x[i] = 0;
ratio[i] = p[i] / w[i];
   }

   for (int i = 0; i < n - 1; i++) {
for (int j = 0; j < n - i - 1; j++) {
if (ratio[j] < ratio[j + 1]) {
// Swap profits              float
tp = p[j + 1];              p[j + 1] =
p[j];              p[j] = tp;
// Swap weights              float
tw = w[j + 1];              w[j + 1]
= w[j];              w[j] = tw;
// Swap ratios              float tr
= ratio[j + 1];              ratio[j +
1] = ratio[j];              ratio[j] =
tr;
         }
      }    }    float rc =
m;    float mp = 0;
for (int i = 0; i < n; i++)
{
      // If weight is less than remaining capacity        if (w[i] <= rc) {          // make it visited          x[i]
= 1;
         // Subtract weight from remaining capacity
         rc -= w[i];
// Add to total Profit
mp += p[i];
      }
```

```
        // If weight is greater than capacity
        else {
            // Take portion of remaining capacity
            x[i] = rc / w[i];           //
add to profit           mp += x[i] *
p[i];           break; // No more
capacity left
        }
    }

    printf("The Selected objects
are:\n");     for (int i = 0; i < n; i++) {
        if (x[i]) {
            printf("Object %d (fraction: %.2f)\n", i + 1, x[i]);
        }
    }
    printf("The Maximum Profit is: %.2f\n", mp);
}
```

Output:



N Queens Problem:

```
#include <stdio.h>
#include <stdbool.h>
#define N 8  // You can change N to any number to solve for different board size
void printSolution(int board[N][N]) {
   for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++) {
printf("%2d ", board[i][j]);
      }
      printf("\n");
   }
}

bool isSafe(int board[N][N], int row, int col) {
   int i, j;
```

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

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

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

   return true;
}

bool solveNQUtil(int board[N][N], int col) {
   // If all queens are
placed     if (col >= N)
return true;

   // Consider this column and try placing this queen in all rows one by one
   for (int i = 0; i < N; i++) {
      // Check if the queen can be placed on
board[i][col]        if (isSafe(board, i, col)) {
```

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

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

        // If placing queen in board[i][col] doesn't lead to a solution
        // then backtrack
        board[i][col] = 0; // Remove queen from board[i][col]
      }
   }
   // If the queen cannot be placed in any row in this column, return false
return false;

bool solveNQ() {
int board[N][N] =
{0};

   if (!solveNQUtil(board, 0)) {
printf("Solution does not exist");
return false;
   }
   printSolution(board);
return true;
} int
main() {
```

```
solveN
Q();
   return 0;
}
```

Output: