Lab 1

Implementation of Bubble , selection and Insertions Sort and Number of step required.

Bubble sort

Theory:

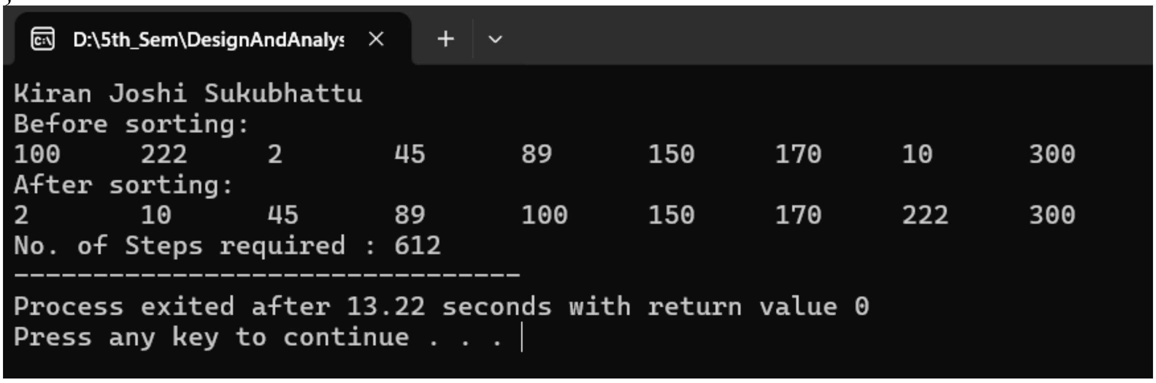
Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

Algorithm:

1. We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after a second pass, the second largest element goes to second last position and so on.
2. In every pass, we process only those elements that have already not moved to the correct position. After k passes, the largest k elements must have been moved to the last k positions.
3. In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.

```
#include<iostream>
using namespace std;
int count = 0;
void BubbleSort(int A[],int n)
{
   for(int i=0;i<n;i++)
   {
      for(int j=0;j<n-1;j++)
      {
         if(A[j]>A[j+1])
         {
            int t= A[j];
            A[j]=A[j+1];
            A[j+1]=t;

         }

         count =count+8;

      }
      count = count+4;
   }
}
int main()
{
cout<<"Kiran Joshi Sukubhattu\n";
int a[9] = {100,222,2,45,89,150,170,10,300};

cout<<"Before sorting: \n";
  for(int i=0 ;i<9 ;i++)
      {
   cout<<a[i]<<"\t" ;


      }

   cout<<endl;

BubbleSort(a,9);
```

```
cout<<"After sorting: \n";
 for(int i=0 ;i<9 ;i++)
        {
    cout<<a[i]<<"\t" ;
        }
cout<<endl;
cout<<"No. of Steps required : "<<count;

return 0;
}
```

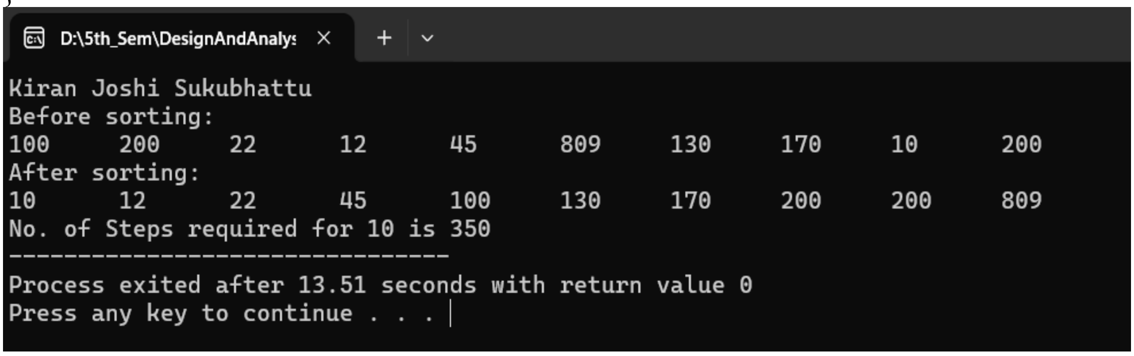


Selection Sort
Theory:
Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element.

Algorithm:
1. First, we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among the remaining elements (or second smallest) and swap it with the second element.
3. We keep doing this until we get all elements moved to the correct position.

```
 #include<iostream>
using namespace std;
int count = 0;
void SelectionSort(int A[],int n)
{
    for(int i=0;i<n;i++)
    {
        int least =A[i];
        int loc =i;

        for(int j=i+1;j<n;j++)
        {
            if(A[j]<least)
            {
                least = A[j];
                loc =j;
            }

            count =count+6;

        }
        A[loc] = A[i];
        A[i] =least;
        count = count+8;
```

```
    }
}
int main()
{
cout<<"Kiran Joshi Sukubhattu\n";
int a[] = {100,200,22,12,45,809,130,170,10,200};

int n= sizeof(a)/sizeof(int);
cout<<"Before sorting: \n";
   for(int i=0 ;i<n ;i++)
        {
    cout<<a[i]<<"\t" ;
        }

    cout<<endl;

SelectionSort(a,n);
cout<<"After sorting: \n";
 for(int i=0 ;i<n ;i++)
        {
    cout<<a[i]<<"\t" ;
        }
cout<<endl;
cout<<"No. of Steps required for "<<n<<" is "<<count;

return 0;
}
```



Insertion Sort

Theory:

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands.

Algorithm:

1. We start with the second element of the array as the first element is assumed to be sorted.
2. Compare the second element with the first element if the second element is smaller than swap them.
3. Move to the third element, compare it with the first two elements, and put it in its correct position
4. Repeat until the entire array is sorted.

```
#include<iostream>
using namespace std;
int count =0;
void InsertionSort(int A[],int n)
{
    int key;
    int j;
    for(int i=1;i<=n-1;i++)
    {
```

```
        key =A[i];

        for(j=i-1;A[j]>key && j>=0;j--)
        {
           A[j+1] =A[j];
           count += 7;
        }
        A[j+1] = key;
           count += 7;
    }
}

int main()
{
    cout<<"Kiran Joshi Sukubhattu\n";
    int a[] = {12,13,25,10,5,29,30,100,8,2};
    int n = sizeof(a)/sizeof(int);

    cout<<"Before sorting :\n";
    for(int i =0 ;i<n;i++)
    {
        cout<<a[i]<< "\t";

    }
     InsertionSort(a,n);

    cout<<"\nAfter sorting :\n";
    for(int i =0 ;i<n;i++)
    {
        cout<<a[i]<< "\t";

    }

    cout<<"\nRequired Number of steps for "<<n << "sized Data: "<<count;
    return 0;
}
```

```
D:\5th_Sem\DesignAndAnaly:
Kiran Joshi Sukubhattu
Before sorting :
12      13      25      10      5       29      30      100     8       2
After sorting :
2       5       8       10      12      13      25      29      30      100
Required Number of steps for 10sized Data: 224
--------------------------------
Process exited after 13.21 seconds with return value 0
Press any key to continue . . .
```

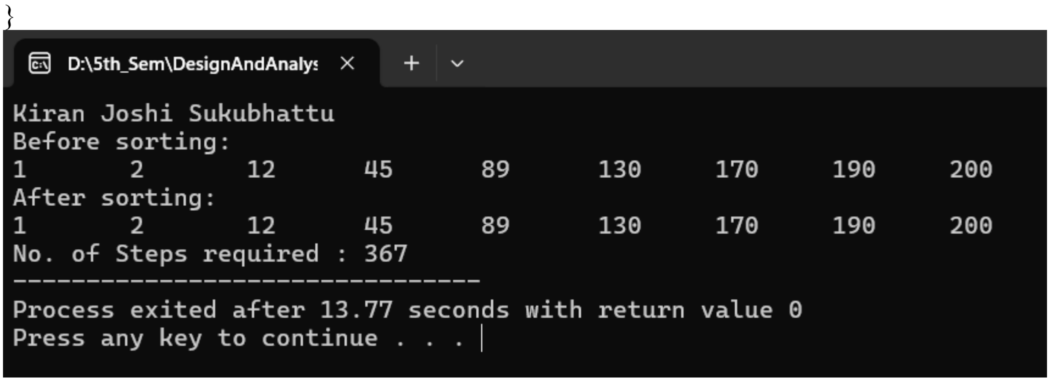
Lab 2
Implementation of Merge sort.

Theory:
Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

Algorithm:
1. Divide the array into two halves.
2. Sort each half
3. Merge the sorted halves back together.
4. Repeat this process until the entire array is sorted.

```
#include<iostream>
using namespace std;
int B[9];
int count = 0;
void Merge(int A[],int l, int m ,int r)
{
   int x= l;
   int y= m;
   int k= l;
   count = count + 3;
   while(x<m && y<=r)
   {
      if(A[x]<A[y])
      {
      B[k]=A[x];
      k++;
      x++;
      count = count +    6;
      }
      else
      {
      B[k]=A[y];
      k++;
      y++;
      count = count +    6;
      }
   }
      while(x<m)
      {
      B[k]=A[x];
      k++;
      x++;
      count =count+6;
      }

      while(y<=r)
      {
      B[k]=A[y];
      k++;
      y++;
      count =count+6;
      }

      for(int i=l ;i<=r;i++)
```

```
        {
        A[i] = B[i];
        count = count+    5;
        }
}

void MergeSort(int A[9],int l ,int r)
{
    if(l<r)
    {
        count = count+3;
        int m = (l+r)/2;
        MergeSort(A,l,m);
        MergeSort(A,m+1,r);
        Merge(A,l,m+1,r);
    }
}

int main()
{
cout<<"Kiran Joshi Sukubhattu\n";
int a[9] = {1,2,12,45,89,130,170,190,200};

cout<<"Before sorting: \n";
   for(int i=0 ;i<9 ;i++)
        {
    cout<<a[i]<<"\t" ;
        }

    cout<<endl;

MergeSort(a,0,8);
cout<<"After sorting: \n";
 for(int i=0 ;i<9 ;i++)
        {
    cout<<a[i]<<"\t" ;
        }
cout<<endl;
cout<<"No. of Steps required : "<<count;

return 0;
}
```



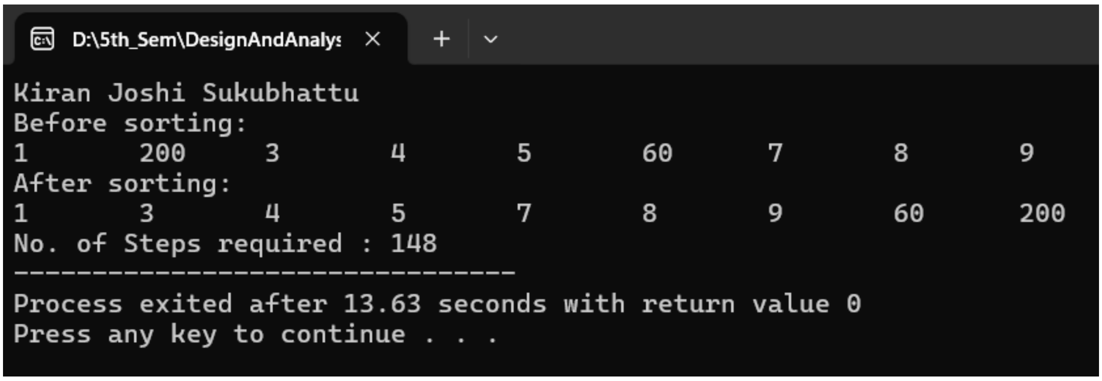
Lab 3
Implementation of Quick Sort

Theory:
Quicksort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Algorithm:
1. Choose a Pivot – Select a pivot element from the array (e.g., first, last, middle, or random).
2. Partition the Array – Rearrange elements so that:
   - Elements smaller than the pivot are on the left.
   - Elements greater than the pivot are on the right.
3. Recursively Apply Quick Sort – Perform the same process on the left and right subarrays.
4. Base Case – If the subarray has one or zero elements, return (already sorted).
5. Combine the Sorted Parts – The final sorted array is obtained when all partitions are sorted.

```
#include<iostream>
using namespace std;
int count=0;
int partition(int A[],int l, int r)
{
   int x=l;
   int y=r;
   int pivot = A[l];

   count = count+3;

   while(x<y)
   {
      while(A[x]<=pivot)
      {
         x++;
         count =count+3;
      }

      while(A[y]>pivot)
      {
          y--;
         count =count+3;
      }

      if(x<y)
      {
         int t = A[x];
         A[x]=A[y];
         A[y] = t;
         count =count+4;

      }

   }
    A[l] =A[y];
   A[y]= pivot;
   count = count+3;
   return y;
```

```
}
void QuickSort(int A[],int l ,int r)
{
    if(l<r)
    {
        count++;
    int p = partition(A,l,r);
    QuickSort(A,l,p-1);
    QuickSort(A,p+1,r);

    }
}

int main()
{
    cout<<"Kiran Joshi Sukubhattu\n";
//int a[9] = {100,22,12,45,809,130,170,10,200};
int a[9] = {1,200,3,4,5,60,7,8,9};

cout<<"Before sorting: \n";
   for(int i=0 ;i<9 ;i++)
        {
    cout<<a[i]<<"\t" ;
        }

    cout<<endl;

QuickSort(a,0,8);
cout<<"After sorting: \n";
 for(int i=0 ;i<9 ;i++)
        {
    cout<<a[i]<<"\t" ;
        }
cout<<endl;
cout<<"No. of Steps required : "<<count;

return 0;
}
```



Lab 4
Implementation of Randomized Quick Sort

Theory:

Randomized quick sort is designed to decrease the chances of the algorithm being executed in the worst-case time complexity of $O(n^2)$. The worst case time complexity of quick sort arises when the input given is an already sorted list, leading to n(n 1) comparisons.

Algorithm:

1. Choose a random pivot from the array.
2. Swap the pivot with the last element.
3. Partition the array so that elements smaller than the pivot are on the left and greater ones on the right.
4. Recursively apply Quick Sort on the left and right subarrays.
5. Stop when subarrays have one or zero elements.

```
#include<iostream>
#include<cstdlib>
using namespace std;
int count=0;

int partition(int A[],int l, int r)
{
    int x=l;
    int y=r;
    int pivot = A[l];

    count = count+3;

    while(x<y)
    {
        while(A[x]<=pivot)
        {
            x++;
            count =count+3;
        }
        while(A[y]>pivot)
        {
             y--;
            count =count+3;
        }
        if(x<y)
        {
            int t = A[x];
            A[x]=A[y];
            A[y] = t;
            count =count+4;
        }
    }
     A[l] =A[y];
    A[y]= pivot;
    count = count+3;
    return y;
}

int randpartition(int A[],int l, int r)
{

int k = l+rand()%(r-l);
```

```
int t = A[l];
A[l]=A[k];
A[k]=t;
count =count +4;
return partition(A,l,r);
}
void RandQuickSort(int A[],int l ,int r)
{
   if(l<r)
   {
      count++;
   int p = randpartition(A,l,r);
   RandQuickSort(A,l,p-1);
   RandQuickSort(A,p+1,r);

   }
}

int main()
{
   cout<<"Kiran Joshi Sukubhattu\n";
//int a[9] = {100,22,12,45,809,130,170,10,200};
int a[9] = {1,2,3,4,5,6,7,8,9};

cout<<"Before sorting: \n";
  for(int i=0 ;i<9 ;i++)
      {
   cout<<a[i]<<"\t" ;
      }

   cout<<endl;

RandQuickSort(a,0,8);
cout<<"After sorting: \n";
 for(int i=0 ;i<9 ;i++)
      {
   cout<<a[i]<<"\t" ;
      }
cout<<endl;
cout<<"No. of Steps required : "<<count;

return 0;
}
```

```
D:\5th_Sem\DesignAndAnaly

Kiran Joshi Sukubhattu
Before sorting:
1       2       3       4       5       6       7       8       9
After sorting:
1       2       3       4       5       6       7       8       9
No. of Steps required : 130
--------------------------------
Process exited after 13.29 seconds with return value 0
Press any key to continue . . .
```

Lab 5

Implementation of 0//1 Knapsack problem using Dynamic approach

Theory:

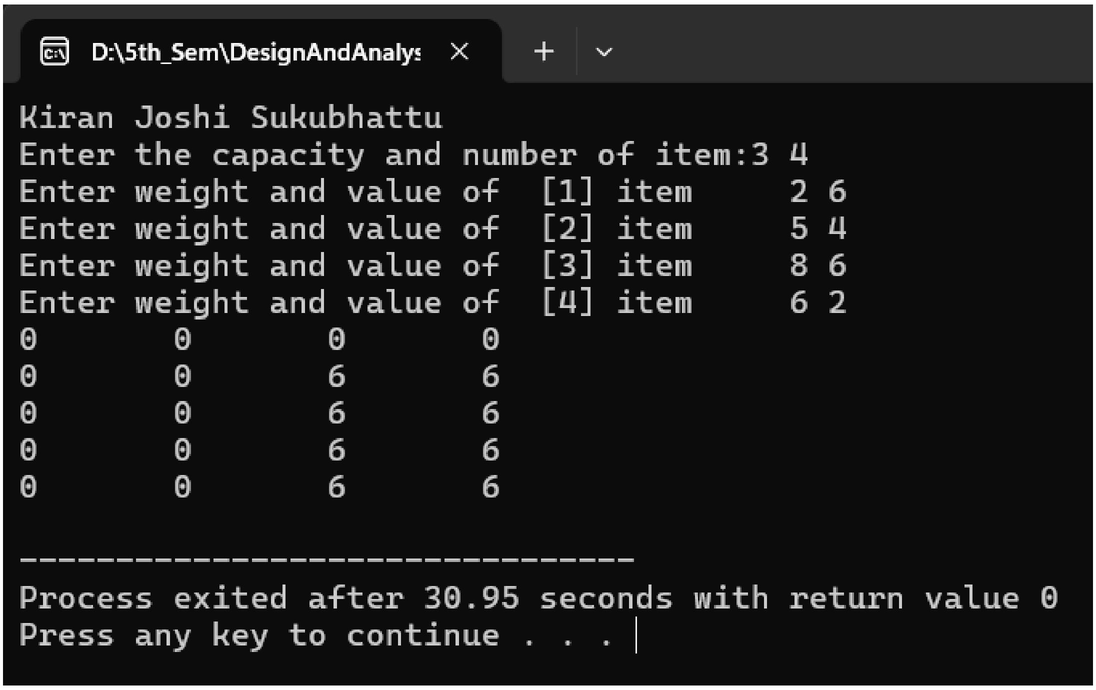
The Knapsack Problem using Dynamic Programming is a method to maximize the total value of items that can be placed in a knapsack of limited capacity. It uses a table to store optimal subproblem solutions, avoiding redundant calculations.

Algorithm:

1. Create a table dp[i][w], where i is the item index and w is the weight capacity.
2. Initialize the first row and column with zeros.
3. For each item, check if its weight is less than or equal to w:
   - If yes, update dp[i][w] with the maximum of including or excluding the item.
   - If no, inherit the previous value.
4. The final value at dp[n][W] gives the maximum profit.

```
#include<stdio.h>
#include<conio.h>

int c[100][100];

void Knapsack(int W , int n , int wt[] ,int v[]){
int i,w;
for(i=0;i<=n;i++)
c[i][0]=0;
for(w=0;w<=W;w++)
c[0][w]=0;

for(i=1;i<=n;i++){
for(w=1;w<=W;w++){
if(wt[i-1]>w)
c[i][w]=c[i-1][w];
else{
if((v[i-1]+c[i-1][w-wt[i-1]])>c[i-1][w]){
c[i][w]=v[i-1]+c[i-1][w-wt[i-1]];
}else{
c[i][w]=c[i-1][w];
}
}
}
}
}

int main(){

cout<<"Kiran Joshi Sukubhattu\n";
int w[100];
int v[100];
int W,n,i,wt;

printf("Enter the capacity and number of item:");
scanf("%d%d",&W,&n);

for(i=0;i<n;i++){
printf("Enter weight and value of  [%d] item\t",(i+1));
scanf("%d%d",&w[i],&v[i]);
}
Knapsack(W,n,w,v);
```

```
for(i=0;i<=n;i++){
for(wt=0;wt<=W;wt++){
printf("%d\t",c[i][wt]);
}

printf("\n");
}
return 0;
}
```



Lab 6

Implementation Of Matrix chain Multiplication Problem

Theory:

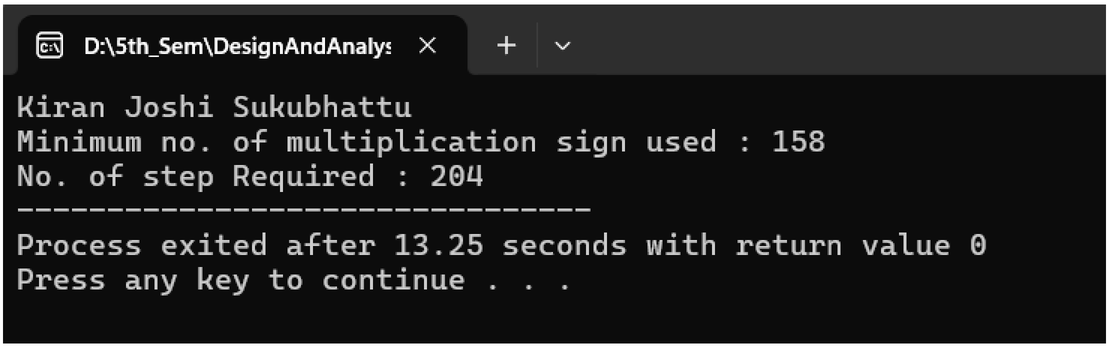
Matrix Chain Multiplication using Dynamic Programming finds the most efficient way to multiply a sequence of matrices by minimizing the total number of scalar multiplications. It avoids redundant computations using a table to store optimal subproblem results.

Algorithm:

1. Create a table dp[i][j] to store the minimum cost of multiplying matrices from index i to j.
2. For each possible chain length, compute dp[i][j] by selecting the best split point k, updating the cost as dp[i][j] = min(dp[i][k] + dp[k+1][j] + cost of multiplication).
3. The final result dp[1][n] gives the minimum multiplication cost.

```
#include<bits/stdc++.h>
using namespace std;
int c=0;
int MatrixChainOrder(int p[], int n)
{
  int m[n][n];
  int i, j, k, L, q;

  for (i=1; i<n; i++)
  {
  m[i][i] = 0;
  c++;
  }

  for (L=2; L<n; L++)
  {
    for (i=1; i<n-L+1; i++)
    {
     j = i+L-1;
     m[i][j] = INT_MAX;
     c =c+4;
     for (k=i; k<=j-1; k++)
     {
       q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
       c =c+7;
       if (q < m[i][j])
       {
        m[i][j] = q;
        c++;
        }
      c =c+5;
     }
     c=c+6;
    }
    c=c+4;
  }
  return m[1][n-1];
}
int main()
{
cout<<"Kiran Joshi Sukubhattu\n";
  int arr[] = {5,4,6,2,7};
  int size = 5;
cout<<"Minimum no. of multiplication sign used : "<<MatrixChainOrder(arr,size)<<endl;
```

```
cout<<"No. of step Required : "<<c;
}
```



Lab 7
Implementation of String Editing

Theory:
String Editing (Edit Distance) using Dynamic Programming calculates the minimum number of operations (insertion, deletion, substitution) required to convert one string into another. It uses a table to store solutions to subproblems, ensuring efficiency.

Algorithm:
1. Create a table dp[i][j], where i represents characters of the first string and j represents the second.
2. Initialize the first row and column to represent transformations from an empty string.
3. For each character pair, update dp[i][j] using the minimum of insertion, deletion, or substitution costs.
4. The final value at dp[m][n] gives the minimum edit distance.

```
#include <iostream>
using namespace std;
int c =0;
// Utility function to find the minimum of three numbers
int min(int x, int y, int z) { return min(min(x, y), z); }
int editDistDP(string str1, string str2, int m, int n)
{
  // Create a table to store results of subproblems
  int dp[m + 1][n + 1];
  // Fill d[][] in bottom up manner
  for (int i = 0; i <= m; i++) {
    for (int j = 0; j <= n; j++) {

      if (i == 0)
      {
      dp[i][j] = j; // Min. operations = j
      c++;
      }
      else if (j == 0)
      {
      dp[i][j] = i; // Min. operations = i
      c++;
      }
      else if (str1[i - 1] == str2[j - 1])
      {
        dp[i][j] = dp[i - 1][j - 1];
        c =c+6;
      }
      else
      {
      dp[i][j]
          = 1    + min(dp[i][j - 1], // Insert
                  dp[i - 1][j], // Remove
                  dp[i - 1][j - 1]); // Replace
                  c =c+10;
      }
        c =c+4;
    }
    c=c+4;
  }
  return dp[m][n];
}
```

```
int main()
{
    cout<<"Kiran Joshi Sukubhattu\n";
    string str1 = "sunday";
    string str2 = "saturday";
    cout << "Maximum no. of edit required : "<<editDistDP(str1, str2, str1.length(),
                str2.length())<<endl;
                cout<<"No. of steps Required : "<<c;
    return 0;
}
```

```
D:\5th_Sem\DesignAndAnaly:
Kiran Joshi Sukubhattu
Before sorting :
12      13      25      10      5       29      30      100     8       2
After sorting :
2       5       8       10      12      13      25      29      30      100
Required Number of steps for 10sized Data: 224
--------------------------------
Process exited after 13.7 seconds with return value 0
Press any key to continue . . .
```

Lab 8

Program for Floyd Warshall Algorithm

Theory:

The Floyd-Warshall Algorithm is a dynamic programming approach used to find the shortest paths between all pairs of vertices in a weighted graph. It iteratively updates distances by considering each vertex as an intermediate point.

Algorithm:

1. Create a distance matrix dist[][], initializing direct edge weights and setting infinity for unreachable pairs.

2. For each vertex k, update dist[i][j] as dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]).

3. Repeat for all vertices to compute the shortest paths between all pairs.

```
// C++ Program for Floyd Warshall Algorithm
using namespace std;
#include<iostream>
#define V 4
#define INF 99999

void printSolution(int dist[][V]);
int count = 0;
void floydWarshall(int dist[][V])
{
    int i, j, k;

    for (k = 0; k < V; k++) {

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

            for (j = 0; j < V; j++) {

                if (dist[i][j] > (dist[i][k] + dist[k][j])
                    && (dist[k][j] != INF
                        && dist[i][k] != INF))
                        {
                            dist[i][j] = dist[i][k] + dist[k][j];
                            count = count +    6;
                        }
                count = count +    4;
            }
            count = count +    4;
        }
        count = count +    4;
    }

    printSolution(dist);
}

void printSolution(int dist[][V])
{
    cout << "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)
                cout << "INF"
                    << " ";
            else
```

```
            cout << dist[i][j] << " ";
        }
        cout << endl;
    }
}

int main()
{
    cout<<"Kiran Joshi Sukubhattu\n";
    int graph[V][V] = { { 0, 5, INF, 10 },
                        { INF, 0, 3, INF },
                        { 5, INF, 0, 1 },
                        { INF, INF, 3, 0 } };

    floydWarshall(graph);
    cout<<"Required No of steps : " <<count;
    return 0;
}
```

```
D:\5th_Sem\DesignAndAnalys
Kiran Joshi Sukubhattu
The following matrix shows the shortest distances between every pair of vertices
0 5 8 9
8 0 3 4
5 10 0 1
8 13 3 0
Required No of steps : 378
--------------------------------
Process exited after 14.1 seconds with return value 0
Press any key to continue . . .
```

Lab 9
Program for Dijkstra's single source shortest path

Theory:
Dijkstra's Algorithm finds the shortest path from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It uses a priority queue to always expand the nearest unvisited vertex first.

Algorithm:
1. Initialize distances from the source to all vertices as infinity, except the source (0).
2. Use a priority queue (or min-heap) to extract the vertex with the smallest distance.
3. Update the distances of its adjacent vertices if a shorter path is found.
4. Repeat until all vertices are processed.

```
//program for Dijkstra's single source shortest path
#include <iostream>
using namespace std;
#include <limits.h>
#define V 9
int count=0;
int minDistance(int dist[], bool sptSet[])
{

   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      if (sptSet[v] == false && dist[v] <= min)
      {
      min = dist[v], min_index = v;
      count = count+4;
      }
   count = count+4;
   return min_index;
}

void printSolution(int dist[])
{
   cout << "Vertex \t Distance from Source" << endl;
   for (int i = 0; i < V; i++)
      cout << i << " \t\t\t\t" << dist[i] << endl;
}

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;
      count = count +3;

   for (int i = 0; i < V - 1; i++) {

      int u = minDistance(dist, sptSet);
      sptSet[u] = true;
      count++;

      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];
                count = count+6;
                }
        count = count +5;
    }

    printSolution(dist);
}

int main()
{
cout<<"Kiran Joshi Sukubhattu\n";
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);
cout<<"No. of steps required :"<<count;
    return 0;
}
```

```
D:\5th_Sem\DesignAndAnalys

Kiran Joshi Sukubhattu
Vertex   Distance from Source
0                               0
1                               4
2                               12
3                               19
4                               21
5                               11
6                               9
7                               8
8                               14
No. of steps required :191
--------------------------------
Process exited after 13.43 seconds with return value 0
Press any key to continue . . .
```

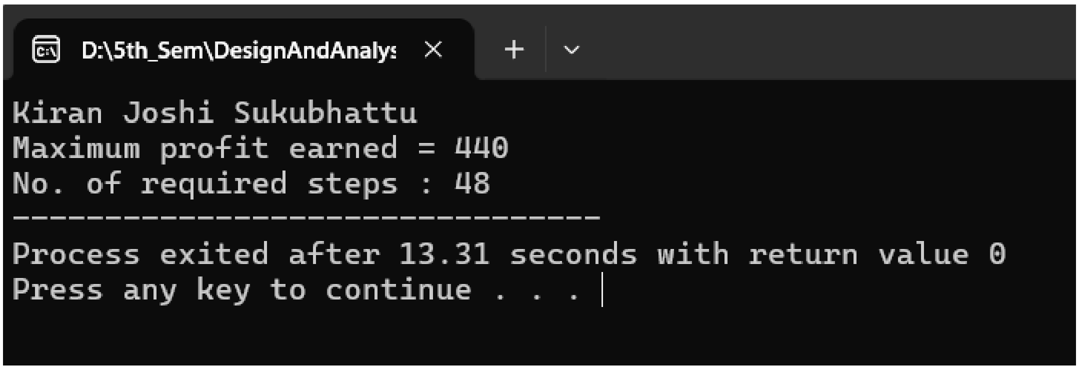
Lab 10
Program to solve fractional Knapsack Problem

Theory:
The Fractional Knapsack Problem is solved using the Greedy approach, where items are selected based on the highest value-to-weight ratio. Unlike the 0/1 Knapsack, fractions of items can be taken to maximize total value.

Algorithm:
1. Sort items by their value-to-weight ratio in descending order.
2. Pick items fully until the knapsack is full; if an item exceeds capacity, take a fraction of it.
3. Stop when the knapsack reaches its maximum capacity.

```
// C++ program to solve fractional Knapsack Problem
#include <bits/stdc++.h>
using namespace std;

struct Item {
    int value, weight;
    // Constructor
    Item(int value, int weight)
        : value(value), weight(weight)
    {
    }
};
int c =0;
bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
     c  = c + 4;
    return r1 > r2;
}

double fractionalKnapsack(struct Item arr[],
                        int N, int size)
{
    // Sort Item on basis of ratio
    sort(arr, arr + size, cmp);
    // Current weight in knapsack
    int curWeight = 0;
    // Result (value in Knapsack)
    double finalvalue = 0.0;
    c += 2;
    for (int i = 0; i < size; i++) {

        if (curWeight + arr[i].weight <= N)
        {
            curWeight += arr[i].weight;
            finalvalue += arr[i].value;
            c  += 4;
        }

        else {
            int remain = N - curWeight;
            finalvalue += arr[i].value
                        * ((double)remain
                           / arr[i].weight);
```

```
            c += 6;
            break;
        }
        c = c+4;
    }
    // Returning final value
    return finalvalue;
}
// Driver Code
int main()
{   cout<<"Kiran Joshi Sukubhattu\n";
    // Weight of knapsack
    int N = 60;
    // Given weights and values as a pairs
    Item arr[] = { { 100, 10 },
                { 280, 40 },
                { 120, 20 },
                { 120, 24 } };
    int size = sizeof(arr) / sizeof(arr[0]);
    // Function Call
    cout << "Maximum profit earned = "<< fractionalKnapsack(arr, N, size)<<endl;
    cout<<"No. of required steps : "<<c;
    return 0;
}
```



Lab 11

Program to solve N Queen Problem using backtracking

Theory:

The N-Queens problem involves placing N queens on an N×N chessboard such that no two queens threaten each other. The solution is found using backtracking by placing queens one by one and ensuring no conflicts arise.

Algorithm:

1. Place a queen in the first available column of the current row.
2. Check for conflicts with queens already placed in previous rows.
3. If no conflicts, move to the next row and repeat. If a conflict occurs, backtrack by removing the queen and trying the next column.
4. Continue until all N queens are placed or no solution exists.

```
//program to solve N Queen Problem using backtracking
#include<iostream>
#define N 4
using namespace std;
int count=0;
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
        if(board[i][j])
            cout << "Q ";
        else cout<<". ";
        printf("\n");
    }
}

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

    for (i = 0; i < col; i++)
    {
    if (board[row][i])
    {
        count++;
        return false;
    }
    count = count+4;
    }


    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
    {
        if (board[i][j])
        {
            count = count +1;
        return false;
        }

        count = count +6;

    }
```

```
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
    {
    if (board[i][j])
    {
        count = count +1;
        return false;
    }
    count = count +6;
    }

    return true;
}

bool solveNQUtil(int board[N][N], int col)
{
    if (col >= N)
    {
        count++;
        return true;
    }


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

        if (isSafe(board, i, col)) {

            board[i][col] = 1;
             count++;

            if (solveNQUtil(board, col + 1))
            {
            count++;
            return true;
            }


            board[i][col] = 0;
        }
    }

    return false;
}

bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };
    if (solveNQUtil(board, 0) == false) {
        cout << "Solution does not exist";
        return false;
    }
    printSolution(board);
    return true;
}
```

```
int main()
{
cout<<"Kiran Joshi Sukubhattu\n";
   solveNQ();
    cout<<"No. of required steps :"<<count;
    return 0;
}
```

```
D:\5th_Sem\DesignAndAnalys

Kiran Joshi Sukubhattu
. . Q .
Q . . .
. . . Q
. Q . .
No. of required steps :381
--------------------------------
Process exited after 13.43 seconds with return value 0
Press any key to continue . . .
```

Lab 12

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

Theory:

Kruskal's Algorithm finds the Minimum Spanning Tree (MST) of a connected, undirected graph by selecting edges in increasing order of weight while avoiding cycles. It uses a disjoint-set (union-find) data structure to efficiently manage connected components.

Algorithm:

1. Sort all edges in the graph by their weight in ascending order.
2. Initialize a disjoint-set to track connected components.
3. For each edge, check if it forms a cycle by checking the sets of the two vertices. If no cycle, include the edge in the MST and unite the sets.
4. Repeat until there are (V-1) edges in the MST, where V is the number of vertices.

```
// program for Kruskal's algorithm to find Minimum Spanning Tree of a given connected, undirected and weighted graph
#include<bits/stdc++.h>
using namespace std;

typedef pair<int, int> iPair;

struct Graph
{
   int V, E;
   vector< pair<int, iPair> > edges;
   // Constructor
   Graph(int V, int E)
   {
      this->V = V;
      this->E = E;
   }

   void addEdge(int u, int v, int w)
   {
      edges.push_back({w, {u, v}});
   }
   int kruskalMST();
};

struct DisjointSets
{
   int *parent, *rnk;
   int n;

   DisjointSets(int n)
   {

      this->n = n;
      parent = new int[n+1];
      rnk = new int[n+1];

      for (int i = 0; i <= n; i++)
      {
         rnk[i] = 0;
         parent[i] = i;
      }
   }
```

```
    int find(int u)
    {

        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }

    void merge(int x, int y)
    {
        x = find(x), y = find(y);

        if (rnk[x] > rnk[y])
            parent[y] = x;
        else
            parent[x] = y;
        if (rnk[x] == rnk[y])
            rnk[y]++;
    }
};

int Graph::kruskalMST()
{
    int mst_wt = 0;
    sort(edges.begin(), edges.end());
    DisjointSets ds(V);

    vector< pair<int, iPair> >::iterator it;
    for (it=edges.begin(); it!=edges.end(); it++)
    {
        int u = it->second.first;
        int v = it->second.second;
        int set_u = ds.find(u);
        int set_v = ds.find(v);

        if (set_u != set_v)
        {
            cout << u << " - " << v << endl;
            mst_wt += it->first;
            ds.merge(set_u, set_v);
        }
    }
    return mst_wt;
}

int main()
{
    cout<<"Kiran Joshi Sukubhattu\n";
    int V = 9, E = 14;
    Graph g(V, E);
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
```

```
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);
    cout << "Edges of MST are \n";
    int mst_wt = g.kruskalMST();
    cout << "\nWeight of MST is " << mst_wt;
    return 0;
}
```

```
D:\5th_Sem\DesignAndAnaly:
Kiran Joshi Sukubhattu
Edges of MST are
6 - 7
2 - 8
5 - 6
0 - 1
2 - 5
2 - 3
0 - 7
3 - 4

Weight of MST is 37
--------------------------------
Process exited after 13.23 seconds with return value 0
Press any key to continue . . .
```

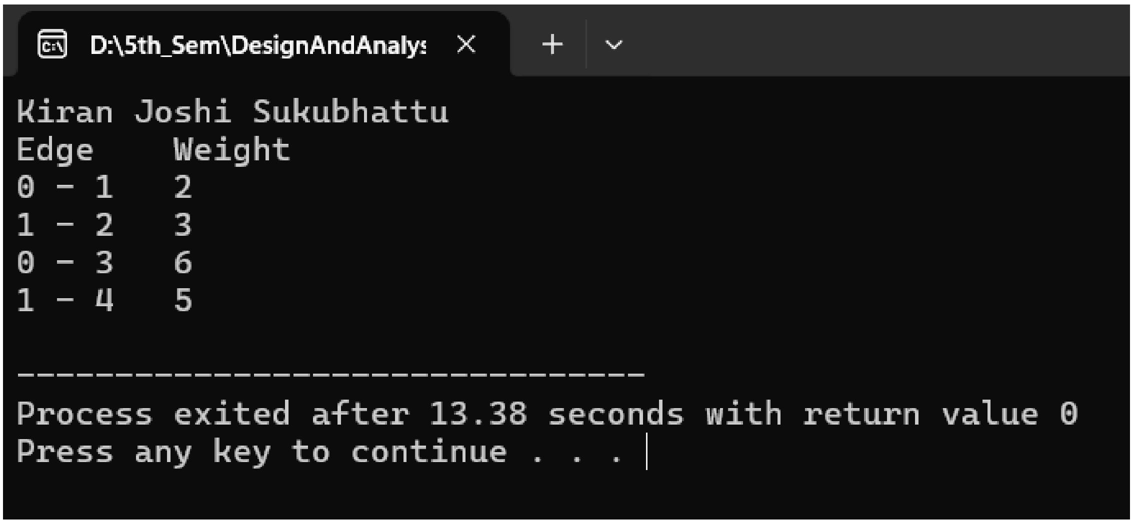
Lab 13
Program for Prim's Minimum

Theory:
Prim's Algorithm is a greedy approach to find the Minimum Spanning Tree (MST) of a connected, undirected graph. It starts with an arbitrary vertex and grows the MST by repeatedly adding the smallest edge that connects a vertex inside the MST to one outside.

Algorithm:
1. Initialize a set of vertices for the MST, starting with an arbitrary vertex.
2. Create an array to store the minimum edge weight for each vertex, and set the starting vertex's weight to 0.
3. Repeat until all vertices are in the MST:
   - Choose the vertex with the smallest weight not yet in the MST.
   - Update the weights of its adjacent vertices if a smaller edge weight is found.
4. The edges chosen form the Minimum Spanning Tree.

```
// A C++ program for Prim's Minimum

#include <bits/stdc++.h>
using namespace std;
// Number of vertices in the graph
#define V 5

int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;
    return min_index;
}

void printMST(int parent[], int graph[V][V])
{
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " \t"
            << graph[i][parent[i]] << " \n";
}

void primMST(int graph[V][V])
{

    int parent[V];
    int key[V];
    bool mstSet[V];
    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;
    key[0] = 0;
    parent[0] = -1;
    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {


        int u = minKey(key, mstSet);
```

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

        for (int v = 0; v < V; v++)

            if (graph[u][v] && mstSet[v] == false
                && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }
    // Print the constructed MST
    printMST(parent, graph);
}
// Driver's code
int main()
{
cout<<"Kiran Joshi Sukubhattu\n";
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };
    // Print the solution
    primMST(graph);
    return 0;
}
```



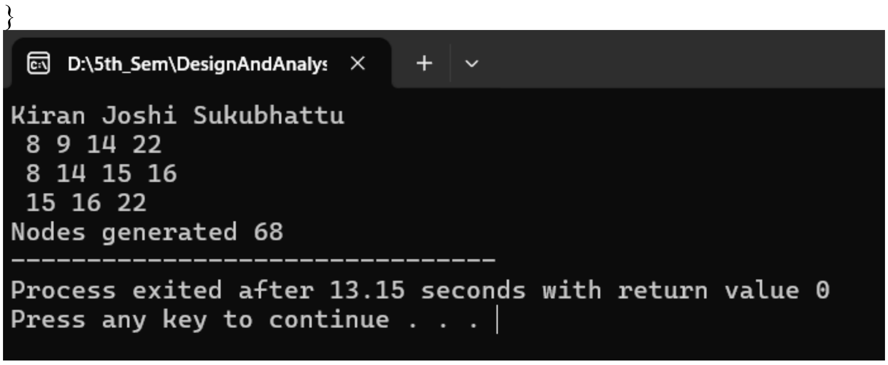
Lab 14
Implementation of Subset sum problem

Theory:
The Subset Sum Problem asks whether there is a subset of a given set of integers that sums up to a specific target value. It can be solved using dynamic programming by storing intermediate results of subproblems to avoid redundant calculations.

Algorithm:
1. Create a 2D table dp[i][j], where i represents the first i elements, and j represents the sum.
2. Initialize the first column of the table (dp[i][0] = true) because the sum 0 is always achievable with an empty subset.
3. For each element and sum, update dp[i][j] as dp[i-1][j] (exclude current element) or dp[i-1][j - arr[i-1]] (include current element).
4. The value at dp[n][target] gives whether the target sum is achievable.

```
#include <bits/stdc++.h>
using namespace std;
#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))
static int total_nodes;

void printSubset(int A[], int size)
{
   for(int i = 0; i < size; i++)
   {
      cout<<" "<< A[i];
   }
   cout<<"\n";
}
// qsort compare function
int comparator(const void *pLhs, const void *pRhs)
{
   int *lhs = (int *)pLhs;
   int *rhs = (int *)pRhs;
   return *lhs > *rhs;
}

void subset_sum(int s[], int t[],
            int s_size, int t_size,
            int sum, int ite,
            int const target_sum)
{
   total_nodes++;
   if( target_sum == sum )
   {
      // We found sum
      printSubset(t, t_size);
      // constraint check
      if( ite + 1 < s_size && sum - s[ite] + s[ite + 1] <= target_sum )
      {

         // Exclude previous added item and consider next candidate
         subset_sum(s, t, s_size, t_size - 1, sum - s[ite], ite + 1, target_sum);
      }
      return;
   }
   else
   {
```

```
        // constraint check
        if( ite < s_size && sum + s[ite] <= target_sum )
        {

            // generate nodes along the breadth
            for( int i = ite; i < s_size; i++ )
            {
                t[t_size] = s[i];
                if( sum + s[i] <= target_sum )
                {

                    // consider next level node (along depth)
                    subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
                }
            }
        }
    }
}
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuplet_vector = (int *)malloc(size * sizeof(int));
    int total = 0;
    // sort the set
    qsort(s, size, sizeof(int), &comparator);
    for( int i = 0; i < size; i++ )
    {
        total += s[i];
    }
    if( s[0] <= target_sum && total >= target_sum )
    {
        subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);
    }
    free(tuplet_vector);
}

int main()
{
cout<<"Kiran Joshi Sukubhattu\n";
    int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
    int target = 53;
    int size = ARRAYSIZE(weights);
    generateSubsets(weights, size, target);
    cout << "Nodes generated " << total_nodes;
    return 0;
}
```



```
D:\5th_Sem\DesignAndAnalys
Kiran Joshi Sukubhattu
 8 9 14 22
 8 14 15 16
 15 16 22
Nodes generated 68
--------------------------------
Process exited after 13.15 seconds with return value 0
Press any key to continue . . .
```

Lab 15
Implementation of job sequence in deadlines

Theory:
The Job Sequencing Problem with Deadlines is a scheduling problem where the goal is to maximize profit by completing jobs within their respective deadlines. Each job has a deadline and a profit, and only one job can be scheduled at a time.

Algorithm:
1. Sort the jobs in descending order of their profit.
2. Initialize a time slot array to track available slots for job scheduling.
3. For each job, find the latest available slot before its deadline and assign the job to that slot if it's available.
4. Repeat until all jobs are scheduled or no more slots are available. The total profit is the sum of the profits of the scheduled jobs.

```
// C++ code for the above approach
#include <algorithm>
#include <iostream>
using namespace std;

struct Job {
   char id;
   int dead;
   int profit;

};

bool comparison(Job a, Job b)
{
   return (a.profit > b.profit);
}

void printJobScheduling(Job arr[], int n)
{

   sort(arr, arr + n, comparison);
   int result[n];
   bool slot[n];

   for (int i = 0; i < n; i++)
      slot[i] = false;

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

      for (int j = min(n, arr[i].dead) - 1; j >= 0; j--) {

         if (slot[j] == false) {
            result[j] = i;
            slot[j] = true;
            break;
         }
      }
   }

   for (int i = 0; i < n; i++)
      if (slot[i])
```

```
        cout << arr[result[i]].id << " ";
}

int main()
{
cout<<"Kiran Joshi Sukubhattu\n";
    Job arr[] = { { 'a', 2, 100 },
                  { 'b', 1, 19 },
                  { 'c', 2, 27 },
                  { 'd', 1, 25 },
                  { 'e', 3, 15 } };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Following is maximum profit sequence of jobs "
        "\n";

    printJobScheduling(arr, n);
    return 0;
}
```