United International University

Final Report

<u>Submitted By</u>

Milat Hossain

ID:011191121

Course title: Data Structure and Algorithms II Laboratory

Course code: CSI 228

Section: C

<u>Submitted To</u>

Mohammad Imam Hossain

Department of Computer Science and Engineering,

United International University

Graph

1. Adjacency list representation for weighted undirected graph. Explain its insertion, update and deletion run time complexities.

```
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

int main()
{


    vector<int> adj_list[100];

    ifstream file_obj;
    file_obj.open("graph.txt", ios::in);

    if(file_obj.is_open()){
        int n, e;
        file_obj>>n>>e;

        for(int ind=0;ind<e;ind++){
            int node1, node2;

            file_obj>>node1>>node2;

            adj_list[node1].push_back(node2);
            adj_list[node2].push_back(node1);
        }


        for(int ind=1;ind<=n;ind++){
            cout<<ind<<" : ";

            vector<int> neighbors=adj_list[ind];

            for(int i=0;i<neighbors.size();i++){
                cout<<neighbors[i]<<" ";
            }
            cout<<endl;
        }

    }
    else{
        cout<<"Couldn't open the file"<<endl;
    }

    return 0;
}
```

Explanation And Time Complexity
Insertion-O(1) Update-O(n) Delete-O(n)

2.Implement the Minimum Spanning Tree (Kruskal's) algorithm with the help of Disjoint set. Also explain its time complexity.

```
#include <iostream>
#include <vector>
#include <fstream>
#include <algorithm>

using namespace std;

int parent[100];
int rnk[100];
vector<pair<int, pair<int, int>>> all_edge;

void make_set(int node_number)
{
    parent[node_number] = node_number;
    rnk[node_number] = 0;
}

int find_set(int node_number)
{
    if (node_number == parent[node_number])
    {
        return node_number;
    }
    else
    {

        int myparent = parent[node_number];
        int neta = find_set(myparent);
        parent[myparent] = neta;

        return neta;
    }
}
```

```
void union_set(int node1, int node2)
{
    int neta1 = find_set(node1);
    int neta2 = find_set(node2);

    if (rnk[neta1] > rnk[neta2])
    {

        parent[neta2] = neta1;
    }
    else if (rnk[neta1] < rnk[neta2])
    {

        parent[neta1] = neta2;
    }
    else
    {

        parent[neta2] = neta1;
        rnk[neta1] = rnk[neta1] + 1;
    }
}
bool myfun(pair<int, pair<int, int>> a,
pair<int, pair<int, int>> b)
{
    if (a.first < b.first)
        return true;
    return false;
}
```

```
void kruskals(int leftnode, int rightnode)
{

    vector<pair<int, int>> A;

    if (find_set(leftnode) != find_set(rightnode))
    {
        union_set(leftnode, rightnode);
        A.push_back({leftnode, rightnode});
    }

    for (int i = 0; i < A.size(); i++)
    {
        cout << A[i].first << " " << A[i].second << endl;
    }
}


int main()
{
    ifstream file_obj;
    file_obj.open("a.txt", ios::in);

    if (file_obj.is_open())
    {
        int n, e, result = 0;
```

```
        file_obj >> n >> e;

        for (int node = 1; node <= n; node++)
        {
            make_set(node);
        }

        for (int edge = 0; edge < e; edge++)
        {
            int leftnode, rightnode, weight;

            file_obj >> leftnode >> rightnode >> weight;

            pair<int, int> innerpair(leftnode, rightnode);
            pair<int, pair<int, int>> finalpair(weight, innerpair);
        }

        sort(all_edge.begin(), all_edge.end());

        for (int edge = 0; edge < e; edge++)
        {

            pair<int, pair<int, int>> curedge_weight = all_edge[edge];
            pair<int, int> curedge = curedge_weight.second;

            int leftnode = curedge.first;
            int rightnode = curedge.second;

            kruskals(leftnode, rightnode);

            if (find_set(leftnode) != find_set(rightnode))
            {
                result += curedge_weight.first;
            }
        }

        cout << "Minimum weight of MST" << endl;
        cout << result << endl;
    }
    else
    {
        cout << "Couldn't open the file" << endl;
    }

    return 0;
}
```

Explanation And Time Complexity

The temporal complexity of Kruskal's Algorithm is O(E log V), where V is the number of vertices.

3.Implement the Dijkstra's algorithm for shortest path search problem. Also explain its time complexity.

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

#define MX 105
#define INF 1000000000
#define NIL -9999

struct node
{
    int val;
    int cost;
};

vector<node> G[MX];
bool vis[MX];
int dist[MX];

class cmp
{
public:
    bool operator()(node &A, node &B)
    {
        if (A.cost > B.cost)
            return true;
        return false;
    }
};

void dijkstra(int source, int n, int s)
{
    priority_queue<node, vector<node>, cmp> PQ;

    PQ.push({source, 0});

    int parent[n];

    while (!PQ.empty())
    {
        node current = PQ.top();
        PQ.pop();

        int val = current.val;
        int cost = current.cost;

        if (vis[val] == 1)
            continue;

        dist[val] = cost;
        vis[val] = 1;

        for (int i = 0; i < G[val].size(); i++)
        {
            int nxt = G[val][i].val;
            int nxtCost = G[val][i].cost;

            if (vis[nxt] == 0 && dist[nxt] > dist[val] + cost)
            {
                PQ.push({nxt, cost + nxtCost});
                parent[nxt] = val;
            }
        }
```

```
        }

        for (int i = 2; i < n; i++)
        {
            int x = i;
            cout<<"Shortest path from 1 to ";
            while (x != -1)
            {

                if (parent[x] == NIL)
                    break;
                cout<<"<--"<<x;
                x = parent[x];
            }

            cout << endl;
        }
    }

    int main()
    {

        int nodes, edges,source;
        cin >> nodes >> edges;

        for (int i = 1; i <= edges; i++)
        {

            int n, e, w;
            cin >> n >> e >> w;
            G[n].push_back({e, w});
        }

        cout << "enter source: ";
        cin >> source;

        dijkstra(source, edges, nodes);

        return 0;
    }
```

Explanation And Time Complexity
Dijkstra's Algorithm has a time complexity of O (V 2), but with a min-priority queue, it lowers to O (V + E l o g V).

4. Implement the Bellman Ford for shorted path search problem. Also explain its time complexity.

```
#include <iostream>
#include <fstream>
#include <vector>

#define INF 999999
#define NIL -1

using namespace std;

int dist[100];
int parent[100];

void print_shortest_path(int src, int dest){
    if(src==dest){
        cout<<"----->"<<src;
    }
    else{
        print_shortest_path(src, parent[dest]);
        cout<<"----->"<<dest;
    }
}

bool Bellman_ford(vector< pair<int,int> > all_edge, vector<double> weight, int nodes, int
startnode){
    ///initialize the graph
    for(int node=1;node<=nodes;node++){
        dist[node]=INF;
        parent[node]=NIL;
    }
    dist[startnode]=0;

    ///performing bellman ford iterations
    for(int loop=1;loop<=nodes-1;loop++){

        ///accessing all edge
        for(int ind=0;ind<all_edge.size();ind++){
            pair<int,int> current_edge=all_edge[ind];
            double edge_weight = weight[ind];


            int srcnode=current_edge.first;
            int destnode=current_edge.second;

            if( dist[destnode] > dist[srcnode]+edge_weight ){
                dist[destnode]=dist[srcnode]+edge_weight;
                parent[destnode]=srcnode;
            }
        }
    }

    ///negative cycle check
    ///accessing all edge
    for(int ind=0;ind<all_edge.size();ind++){
        pair<int,int> current_edge=all_edge[ind];
        double edge_weight = weight[ind];


        int srcnode=current_edge.first;
        int destnode=current_edge.second;
```

```
        if( dist[destnode] > dist[srcnode]+edge_weight ){
            ///cycle exists
            return false;
        }
    }

    return true;
}


int main()
{
    vector< pair<int,int> > all_edge; ///vector<>( (1,3), (2,5), (4,7)  )
    vector<double> weight;            ///vector<>(   5   ,   7            )

    ifstream file_obj;
    file_obj.open("graph.txt");

    if(file_obj.is_open()){
        int nodes, edges;

        file_obj>>nodes>>edges;

        for(int i=0;i<edges;i++){
            int s,d;
            double wt;

            file_obj>>s>>d>>wt;

            pair<int,int> p(s,d);

            all_edge.push_back(p);
            weight.push_back(wt);
        }


        ///all_edge vector contains all edges
        ///weight   vector contains all corresponding weights
        Bellman_ford(all_edge, weight, nodes, 1);

        print_shortest_path(1, 5);

    }
    else{
        cout<<"Couldn't open the file"<<endl;
    }

    return 0;
}
```

Explanation And Time Complexity
Bellman Ford is used to verify if there is a negative cycle in a graph. The time complexity of the Bellman Ford technique is rather large, O (V E) in case E = V 2 and O (V 3) in case E = V 3.

Divide and Conquer

1. Implement the Binary Search algorithm and explain its time complexity.

```
#include <iostream>

using namespace std;

bool myself(int arr[], int start, int stop, int searchval){
    if(start>stop){  ///for zero length array
        return false;
    }
    else{
        int midind = (start+stop)/2;   ///or, start+(stop-start)/2

        if(arr[midind]==searchval){
            return true;
        }
        else if(searchval<arr[midind]){
            bool friendresult=myself(arr,start,midind-1,searchval);

            return friendresult;
        }
        else{
            bool friend1result=myself(arr, midind+1, stop, searchval);

            return friend1result;
        }
    }
}


int main()   ///CEO
{
    int sorted_arr[]={1,3,6,10,13,17,21,22,23};
    int sz=sizeof(sorted_arr)/sizeof(int);
    int searchval=25;

    cout<< myself(sorted_arr, 0, sz-1, searchval) <<endl;

    return 0;
}
```

Explanation And Time Complexity
The binary search algorithm has a time complexity O(log n). When the central index directly matches the required value, the best-case time complexity is O(1).

2. Implement the Merge Sort algorithm and explain its time complexity

```
#include <iostream>
using namespace std;


void merge_sorted_halfs(int arr[], int startind, int midind, int endind)
{
    ///copying the left half to leftarr
    int leftarrsz = (midind - startind + 1);
    int leftarr[100];
    for (int ind = 0; ind < leftarrsz; ind++)
    {
        leftarr[ind] = arr[startind + ind];
    }
    ///copying the right half to rightarr
    int rightarrsz = (endind - (midind + 1) + 1);
    int rightarr[100];
    for (int ind = 0; ind < rightarrsz; ind++)
    {
        rightarr[ind] = arr[midind + 1 + ind];
    }
    ///merging the left and right halves and placing into the main array, arr
    int leftind = 0;
    int rightind = 0;
    for (int ind = startind; ind <= endind; ind++)
    {
        if (leftind == leftarrsz)
        {
            ///when all the left array elements are already copied
            ///we only need to copy the right array elements
            arr[ind] = rightarr[rightind];
            rightind++;
        }
        else if (rightind == rightarrsz)
        {
            ///when all the right array elements are already copied
            ///we only need to copy the left array elements
            arr[ind] = leftarr[leftind];
            leftind++;
        }
        else if (leftarr[leftind] <= rightarr[rightind])
        {
            arr[ind] = leftarr[leftind];
            leftind++;
        }
        else
        {
            arr[ind] = rightarr[rightind];
            rightind++;
        }
    }
}
```

```cpp
void merge_sort(int arr[], int startind, int endind)
{
    ///startind > endind: array is empty. This case won't happen
    if (startind == endind)
    {
        ///only 1 elements in the array
        return;
    }
    else if (startind < endind)
    {
        ///array contains more than 1 elements
        int midind = (startind + endind) / 2;
        merge_sort(arr, startind, midind);
        merge_sort(arr, midind + 1, endind);
        merge_sorted_halfs(arr, startind, midind, endind);
    }
}

int main()
{
    int arr[] = {14, 7, 3, 12, 9, 11, 6, 2};
    int sz = sizeof(arr) / sizeof(int);

    merge_sort(arr, 0, sz - 1);

    for (int ind = 0; ind < sz; ind++)
        cout << arr[ind] << " ";
    cout << endl;

    return 0;
}
```

Explanation And Time Complexity
The time complexity of Merge Sort is **O(n*Log n)**

3.Implement the Quick Sort algorithm and explain its time complexity.

```
#include <iostream>
using namespace std;

int partition_arr(int arr[], int startind, int endind)
{

    int pivot = arr[endind];
    int k = startind - 1;
    int i = startind;

    while (i < endind)
    {

        if (arr[i] < pivot)
        {
            k++;
            int temp = arr[i];
            arr[i] = arr[k];
            arr[k] = temp;
        }
        i++;
    }

    int temp = arr[endind];
    arr[endind] = arr[k + 1];
    arr[k + 1] = temp;
    return k + 1;
}

void QuickSort(int arr[], int startind, int endind)
{
    if (startind > endind)
    {
        return;
    }
    else if (startind == endind)
    {
        return;
    }
    else
    {
        int pivotpos = partition_arr(arr, startind, endind);
        QuickSort(arr, startind, pivotpos - 1);
        QuickSort(arr, pivotpos + 1, endind);
    }
}

int main()
{
    int arr[] = {2, 8, 7, 1, 3, 5, 6, 4};
    int sz = sizeof(arr) / sizeof(int);
    QuickSort(arr, 0, sz - 1);

    for (int ind = 0; ind < sz; ind++)
    {
        cout << arr[ind] << " ";
    }
    cout << endl;
    return 0;
}
```

Explanation And Time Complexity
The time complexity of Quick Sort is **O(nLog n)**

4. Describe the difference between Merge and Quick sort.

Merge Sort	Quick sort
➢ Merge sort divides the arrays into two subarrays again and again until one element is left.	➢ Quick sort works as sorting the elements by comparing each element with the pivot.
➢ It operates fine on any type of array	➢ 2.It works well on smaller array
➢ Merge sort consists consistent speed in all type of data sets.	➢ Quick sort is faster than other sorting algorithms for small data set.
➢ Merge sort's stability status is "Stable"	➢ Quick sort's stability status is "Not Stable"
➢ Merge sort is preferred for Linked Lists	➢ Quick sort is preferred for arrays
➢ -Worst case time complexity - O(n log n)	➢ Worst case time complexity - $O(n^2)$

5.Implement the Maximum subarray sum problem.

```
#include <iostream>
using namespace std;
#define INT_MIN 10000

int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

int max(int a, int b, int c)
{
    if (a >= b && a >= c)
        return a;

    else if (b >= a && b >= c)
        return b;

    else
        return c;
}
```

```cpp
int maxCrossingSum(int arr[], int low, int mid, int high)
{

    int sum = 0;
    int leftsubsum = INT_MIN;
    for (int i = mid; i >= low; i--)
    {
        sum = sum + arr[i];
        if (sum > leftsubsum)
            leftsubsum = sum;
    }

    sum = 0;
    int rightsubsum = INT_MIN;
    for (int i = mid + 1; i <= high; i++)
    {
        sum = sum + arr[i];
        if (sum > rightsubsum)
            rightsubsum = sum;
    }

    int sumOfLeftright = leftsubsum + rightsubsum;

    return max(sumOfLeftright, leftsubsum, rightsubsum);
}

int maximumSubSum(int arr[], int low, int high)
{

    if (low == high)
    {
        return arr[low];
    }

    int mid = (low + high) / 2;


    return max(maximumSubSum(arr, low, mid),
           maximumSubSum(arr, mid + 1, high),
           maxCrossingSum(arr, low, mid, high));
}

int main()
{
    int n, arr[n];
    cin >> n;

    for (int i = 0; i < n; i++){
        cin >> arr[i];
    }

    int sum = maximumSubSum(arr, 0, n - 1);

    cout << sum << endl;

    return 0;
}
```

Recursion and Dynamic Programming

1. Find out the leave nodes of a binary tree (represented as an array) using recursion.

➢ **Using Recursion**

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

using namespace std;

struct node {
  int data;
  struct node *left;
  struct node *right;
};

struct node *newNode(int data) {
  struct node *node = (struct node *)malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return (node);
}

bool isLeaf(struct node *node){
    if(node->left == NULL && node->right == NULL){
            return true;
    }else{
        return false;
    }
}

void printLeaves(struct node *node) {
    // base case
    if (node == NULL){
        return;

    } if (isLeaf(node)) {
        cout << node->data;

    }
    printLeaves(node->left);
    printLeaves(node->right);

}

int main() {

  struct node *root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);

  printLeaves(root);

  return 0;
}
```

2. Implement the Fibonacci Number problem both using recursion and using Dynamic programming.

➢ Using Recursion

```cpp
#include <iostream>

using namespace std;

int fib(int term){
    if(term==0){
        return 0;
    }
    else if(term==1){
        return 1;
    }
    else{
        ///recursive call
        int friend1=fib(term-1);
        int friend2=fib(term-2);

        int myresult=friend1+friend2;
        return myresult;
    }
}
```

```cpp
int main()
{
    cout<< fib(6) <<endl;

    return 0;
}
```

➢ Using Dynamic programming

```cpp
#include <iostream>
using namespace std;

int dptable[100];

int dp_fib(int term){
    dptable[0]=0;
    dptable[1]=1;

    for(int next=2;next<=term;next++){
        dptable[next]=dptable[next-1]+dptable[next-2];
    }

    return dptable[term];
}
```

```cpp
int main()
{
    cout<< dp_fib(6) <<endl;

    return 0;
}
```

3.Implement the Binomial Coefficient problem (nCr) both using recursion and using Dynamic programming.

➢ Using Recursion

```
#include<iostream>
using namespace std;

int a,b;
int ncr(int c,int r){

    if(r==0 )return 1;
    if(r>=c)return 1;

     a=ncr(c-1,r);
     b=ncr(c-1,r-1);
    int ans=a+b;
    return ans;
}
```

```
int main(){

    cout<<ncr(5,2) <<endl;

}
```

➢ Using Dynamic programming

```
#include <iostream>
using namespace std;

int dptable[100][100];

int BC(int n_limit, int r_limit){
    ///BC(n,r)=1 when r=0
    for(int n=0;n<=n_limit;n++){
        dptable[n][0]=1;
    }

    ///BC(n,r)=1 when r=n
    for(int n=0;n<=n_limit;n++){
        dptable[n][n]=1;
    }

    ///gradually filling up the table
    for(int n=2;n<=n_limit;n++){
        for(int r=1;r<=n-1;r++){
            dptable[n][r]= dptable[n-1][r]
                          +dptable[n-1][r-1];
        }
    }

    return dptable[n_limit][r_limit];
}
```

```
int main()
{
    cout<< BC(10,0) <<endl;
    return 0;
}
```

3. Implement the Coin Change problem both using recursion and using Dynamic programming.

➢ Using Recursion

```
#include <iostream>

using namespace std;

int myself(int coins[], int sz, int amount){
    if(amount==0){
        return 1;
    }
    else if(amount<0){
        return 0;
    }
    else if(sz==0 && amount>0){
        return 0;
    }
    else{
        ///way 1: considering
        int rem_amount=amount-coins[sz-1];
        int friend1result = myself(coins,sz,rem_amount);


        ///way 2: not considering
        int friend2result = myself(coins, sz-1, amount);

        int myresult = friend1result+friend2result;

        return myresult;
    }
}


int main()    ///CEO
{
    int coins[]={2,3,5};
    int sz=3;
    int amount=10;

    cout<< myself(coins, sz, amount) <<endl;

    return 0;
}
```

➢ Using Dynamic programming

```
#include <iostream>
using namespace std;

#define infinty 9999999

int DpChange(int M,int c[],int d){
    int bestNumCoins[M+1],sol[M+1];


    bestNumCoins[0]=0;

    for(int m=1;m<=M;m++){
        bestNumCoins[m]= infinty;

        for(int i=0;i<d;i++){
            if(m>=c[i]){
                if(bestNumCoins[m-c[i]]+1 < bestNumCoins[m]){
                    bestNumCoins[m]= bestNumCoins[m-c[i]]+1;

                    sol[m]= c[i];
                }
            }
        }
    }

    int m=M;

    while(m>0){
        // printf("%d ",sol[m]);
        m=m-sol[m];
    }

    return bestNumCoins[M];
}

int main(){
    int arr[]={1,3,5};

    int count = DpChange(7,arr,3);

    printf("%d",count);

    return 0;
}
```

5. Implement the Subset sum problem both using recursion and using Dynamic programming.

➢ Using Dynamic programming

```
#include <iostream>
using namespace std;

bool dptable[100][100]={0};

bool SS(int st[], int st_sz, int st_sum){
    ///SS(sz, sum=0) = True
    for(int sz=0;sz<=st_sz;sz++){
        dptable[sz][0]=true;
    }

    ///SS(sz=0, sum>0) = False
    for(int sum=1;sum<=st_sum;sum++){
        dptable[0][sum]=false;
    }

    ///filling the table
    for(int sz=1;sz<=st_sz;sz++){
        for(int sum=1;sum<=st_sum;sum++){
            bool friend1=dptable[sz-1][sum];

            int friend2_sumval=sum-st[sz-1];
            if(friend2_sumval>=0){
                ///valid column
                bool friend2=dptable[sz-1][friend2_sumval];
                dptable[sz][sum]= friend1 || friend2;
            }
            else{
                dptable[sz][sum]=friend1;
            }
        }
    }

    return dptable[st_sz][st_sum];
}

int main()
{
    int st[]={1,3,5,2};
    int sz=sizeof(st)/sizeof(int);
    int sum=26;

    bool result=SS(st,sz,sum);

    if(result){
        cout<<"Possible"<<endl;
    }
    else{
        cout<<"Not Possible"<<endl;
    }

    return 0;
}
```

6. Implement the 0/1 knapsack problem both using recursion and using Dynamic programming.

➢ Using Recursion

```
#include<iostream>
using namespace std;

int k(int W, int wt[], int val[], int n)
{


    if (n == 0 || W == 0)return 0;


    if (wt[n - 1] > W) return k(W, wt, val, n - 1);

    else
      int one=val[n - 1] + k(W - wt[n - 1], wt, val, n - 1);
      int two=kk(W, wt, val, n - 1);

     return one+two;
}

int main()
{
    int st[]={3, 34, 4, 12, 5, 2};
    int sz=sizeof(st)/sizeof(int);
    int sum=9;

    bool result=k(st,6,sum);

    if(result){
        cout<<"Possible"<<endl;
    }
    else{
        cout<<"Not Possible"<<endl;
    }

    return 0;
}
```

➢ Using Dynamic programming

```
#include<iostream>
using namespace std;


bool dptable[100][100]={0};

bool SS(int st[], int st_sz, int st_sum){
    ///SS(sz, sum=0) = True
    for(int sz=0;sz<=st_sz;sz++){
        dptable[sz][0]=true;
    }

    ///SS(sz=0, sum>0) = False
    for(int sum=1;sum<=st_sum;sum++){
        dptable[0][sum]=false;
    }

    ///filling the table
    for(int sz=1;sz<=st_sz;sz++){
        for(int sum=1;sum<=st_sum;sum++){
            bool friend1=dptable[sz-1][sum];
```

```cpp
            int friend2_sumval=sum-st[sz-1];
            if(friend2_sumval>=0){
                ///valid column
                bool friend2=dptable[sz-1][friend2_sumval];
                dptable[sz][sum]= friend1 || friend2;
            }
            else{
                dptable[sz][sum]=friend1;
            }
        }
    }

    return dptable[st_sz][st_sum];
}

int main()
{
    int st[]={3, 34, 4, 12, 5, 2};
    int sz=sizeof(st)/sizeof(int);
    int sum=9;

    bool result=SS(st,sz,sum);

    if(result){
        cout<<"Possible"<<endl;
    }
    else{
        cout<<"Not Possible"<<endl;
    }

    return 0;
}
```

7. Implement the Longest common subsequence problem using Dynamic programming.

➢ Using Dynamic programming

```
#include <iostream>
using namespace std;


int dptable[100][100];

int LCS(string str1, int st1_sz1, string str2, int st2_sz2){
    ///LCS(sz1, sz2=0)=0
    for(int sz1=0;sz1<=st1_sz1;sz1++){
        dptable[sz1][0]=0;
    }

    ///LCS(sz1=0, sz2)=0
    for(int sz2=0;sz2<=st2_sz2;sz2++){
        dptable[0][sz2]=0;
    }

    ///rest of the table
    for(int sz1=1;sz1<=st1_sz1;sz1++){
        for(int sz2=1;sz2<=st2_sz2;sz2++){

            if(str1[sz1-1]==str2[sz2-1]){
                dptable[sz1][sz2]=1+dptable[sz1-1][sz2-1];
            }
            else{
                int friend1=dptable[sz1][sz2-1];
                int friend2=dptable[sz1-1][sz2];

                dptable[sz1][sz2]=max(friend1, friend2);
            }

        }
    }

    return dptable[st1_sz1][st2_sz2];
}

int main()
{
    string str1="GXTXAYB";
    int sz1=str1.size();

    string str2="AGGTAB";
    int sz2=str2.size();

    cout<< LCS(str1, sz1, str2, sz2) <<endl;

    return 0;
}
```

Greedy Approach

1.Implement the fractional knapsack problem

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool sortfn(pair<int,int> item1, pair<int,int> item2){
    double item1perUnitValue=item1.first/item1.second;
    double item2perUnitValue=item2.first/item2.second;

    return item1perUnitValue>item2perUnitValue;

}

double FracKS(int v[], int w[], int sz, int c){
    vector<pair<int,int> > myitems;

    for(int ind=0;ind<sz;ind++){
        pair<int,int> p(v[ind], w[ind]);
        myitems.push_back(p);
    }

sort(myitems.begin(), myitems.end(), sortfn);


    double total_gain=0;
    for(int ind=0;ind<sz;ind++){
        pair<int,int> p = myitems[ind];
        double valueperunit=p.first/p.second;
        int weight=p.second;

        if(weight<=c){
            total_gain+=(weight*valueperunit);
            c=c-weight;
        }
        else{
            ///weight>c
            total_gain+=(c*valueperunit);
            c=0;
            break;
        }
    }

    return total_gain;

}

int main()
{
    int v[]={100,200,300,400,500};
    int w[]={5  ,20,  60, 80, 25};
    int sz=sizeof(v)/sizeof(int);
    int c = 95;

    cout<< FracKS(v,w,sz,c) <<endl;

    return 0;
}
```

2. Implement the Activity selection problem.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool sortfn(pair<int,int> item1, pair<int,int> item2){
    return item1.second < item2.second;
}

void Act_Selection(int S[],int F[],int sz){
    vector<pair<int,int>> tasks;

    for(int ind=0;ind<sz;ind++){
        pair<int,int> p(S[ind], F[ind]);
        tasks.push_back(p);
    }

    sort(tasks.begin(), tasks.end(), sortfn);

    vector<pair<int,int>> selected_tasks;
    pair<int,int> firsttask=tasks[0];
    selected_tasks.push_back(firsttask);
    int last_finish_time=firsttask.second;

    for(int ind=1;ind<sz;ind++){
        pair<int,int> curtask=tasks[ind];
        int curtask_start=curtask.first;

        if(last_finish_time<curtask_start){
            selected_tasks.push_back(curtask);
            last_finish_time=curtask.second;
        }
    }

    ///print selected tasks
    for(int ind=0;ind<selected_tasks.size();ind++){
        pair<int,int> curtask=selected_tasks[ind];

        cout<< curtask.first <<" "<<curtask.second<<endl;
    }
}

int main()
{

    int S[]={12,1,3, 8,0,5,3,5,6,  8, 2};
    int F[]={14,4,5,11,6,7,8,9,10,12,13};
    int sz=sizeof(S)/sizeof(int);

    Act_Selection(S,F,sz);

    return 0;
}
```

Thank You