Lab Document

Nusrat Farzana Choudhury
Student
ID: 221-35-990
Section: B
Department of SWE
Daffodil International University

Table of contents:

Topic: Searching

i) Linear Search

```
#include<stdio.h>
int main()
{
	int search, i,n;
	printf("Enter the number of elements in array: ");
	scanf("%d",&n);
	int a[n]; //array initialization

	printf("\nEnter %d integer(s):\n",n);

	for(i=0; i<n; i++)
	{
		printf("Index-%d: ",i);
		scanf("%d",&a[i]);
	}
	printf("\nEnter the element to be searched for: ");
	scanf("%d",&search); //taking user input for searching
	printf("\n----Search result----\n");
	for(i=0; i<n; i++)
	{
		if (search==a[i]) //If required element is found
		{
			printf("The element '%d' is found at index %d.\n",search,i);
			break;
		}
	}

	if(i==n)
	{
		printf("\nThe element '%d' is not found.\n",search);
	}
	return 0;
}
```

ii) Binary Search

```
#include<stdio.h>
int main()
{
	int c, min, max, mid, n, search, d, swap;

	printf("Enter the number of elements in array: ");
	scanf("%d",&n);
	int a[n]; //array initialization

	printf("\nEnter %d integer(s):\n",n);
	for(c=0; c<n; c++)
	{
		printf("Index-%d: ",c);
		scanf("%d",&a[c]);
	}

	/*Bubble sort code*/
	for(c=0;c<n-1; c++){
		for(d=0; d<n-c-1;d++){
		  if (a[d]>a[d+1]){
			swap=a[d];
			a[d]=a[d+1];
			a[d+1]=swap;
		  }
		}
	}

	printf("\nSorted list in ascending order:\n");
	for(c=0;c<n;c++){
		printf("Index-%d: %d\n",c,a[c]);
	}
	printf("\nEnter a value to find: ");
	scanf("%d",&search); //taking user input for searching

	min=0;
	max=n-1;
	mid=(min+max)/2;

	while(min<=max){
		if(a[mid]<search){
			min=mid+1;
		}
		else if (a[mid]==search){
			printf("\n%d is found at index %d.\n",search,mid);
			break;
		}
		else{
			max=mid-1;
		}

		mid=(min+max)/2;
	}
	if(min>max){
		printf("\nNot found! %d isn't present in the list.\n",search);
	}
	return 0;
}
```

Topic: Sorting

i) Insertion sort

```
#include<stdio.h>
int main(){
	int i,j,n,temp;
	printf("Enter the size of the array: ");
	scanf("%d",&n);
	int a[n];
	printf("Enter the elements of the array:\n");
	for(i=0;i<n;i++){
		printf("Index-%d: ",i);
		scanf("%d",&a[i]);
	}
	for(i=1;i<n;i++){
		temp=a[i];
		j=i-1;
		while(j>=0 && a[j]>temp){
			a[j+1]=a[j];
			j--;
		}
		a[j+1]=temp;
	}
	 printf("\nSorted array in ascending order is:\n");
	 for(i=0;i<n;i++){
		printf("Index-%d: %d\n",i,a[i]);
	 }

	return 0;
}
```

ii) Selection sort

```
#include<stdio.h>
int main()
{
	int i,j,n,temp,position;

	printf("Enter the size of the array: ");
	scanf("%d",&n);
	int a[n]; //array initialization

	printf("\nEnter the elements of the array:\n",n);
	for(i=0; i<n; i++)
	{
		printf("Index-%d: ",i);
		scanf("%d",&a[i]);
	}

	for(i=0;i<n-1;i++){
		position=i;
		for(j=i+1;j<n;j++){
			if(a[j]>a[position]){ //used '>' for descending order
				position=j;
			}
		}

		if(position!=i){
			temp=a[i];
			a[i]=a[position];
			a[position]=temp;
		}
	}

	printf("\nSorted array in ascending order:\n");
	for(i=0;i<n;i++){
		printf("Index-%d: %d\n",i,a[i]);
	}

	return 0;
}
```

iii) Merge sort

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

void merge(int arr[], int left[], int left_size, int right[],int right_size) {
   int i=0, j=0, k=0;

   while (i < left_size && j < right_size) {
      if (left[i] <= right[j]) {
         arr[k++] = left[i++];
      }
                                 else {
         arr[k++] = right[j++];
      }
   }

   while (i < left_size) {
      arr[k++] = left[i++];
   }

   while (j < right_size) {
      arr[k++] = right[j++];
   }
}

void merge_sort(int arr[], int size) {
   if (size < 2) {
      return;
   }
   int mid=size/2;
   int left[mid], right[size-mid];
   for(int i=0;i<mid;i++){
                left[i]=arr[i];
                }
                for(int i=mid;i<size;i++){
                                right[i-mid]=arr[i];
                }
                merge_sort(left,mid);
                merge_sort(right,size-mid);
                merge(arr,left,mid,right,size-mid);

}

int main() {
   int n,i;
   printf("Enter the size of the array: ");
   scanf("%d", &n);
   int a[n];
   printf("\nEnter the elements of the array:\n");
   for (int i = 0; i < n; i++) {
                printf("Index-%d: ",i);
      scanf("%d", &a[i]);
   }

   // Call merge sort
   merge_sort(a,n);

   printf("\nSorted array in ascending:\n");
   for (int i = 0; i < n; i++) {
      printf("Index-%d: %d\n",i, a[i]);
   }
   printf("\n");

   return 0;
}
```

iv) Bubble sort:

```
#include <stdio.h>
int main()
{
	int n, i, j, tmp;

	printf("Enter number of elements: ");
	scanf("%d", &n);
	char array[n];

	printf("Enter %d letters:\n", n);
	for (i = 0; i < n; i++){
		printf("Index-%d: ",i);
		scanf("%s", &array[i]);
	}
	for (i = 0 ; i < n - 1; i++)
	{
		for (j = 0 ; j < n - i - 1; j++)
		{
			if (array[j] < array[j+1]) //used '<' for ascending order
			{
				tmp	= array[j];
				array[j]   = array[j+1];
				array[j+1] = tmp; //swap
			}
		}
	}

	printf("\nSorted array in descending order is:\n");
	for (i = 0; i < n; i++){
		printf("\t  %c\t", array[i]);
	}
	printf("\n");
	for (i = 0; i < n; i++){
		printf("\tIndex-%d\t",i);
	}
	return 0;
}
```

v) Quick sort:

```
#include<stdio.h>

void quicksort(int number[],int first,int last)
{
	int i, j, pivot, temp;
	if(first<last)
	{
		pivot=first;
		i=first;
		j=last;
		while(i<j)
		{
			while(number[i]<=number[pivot]&&i<last)
				i++;
			while(number[j]>number[pivot])
				j--;
			if(i<j)
			{
				temp=number[i];
				number[i]=number[j];

				number[j]=temp;
			}
		}
		temp=number[pivot];
		number[pivot]=number[j];
		number[j]=temp;
		quicksort(number,first,j-1);
		quicksort(number,j+1,last);
	}
}
int main()
{
	int i, n;
	printf("Enter the number of elements: ");
	scanf("%d",&n);
	int arr[n];
	printf("Enter %d integers:\n",n);
	for(i=0; i<n; i++){
		printf("Index-%d: ",i);
		scanf("%d",&arr[i]);
	}
	quicksort(arr,0,n-1);
	printf("\nOrder of Sorted elements in ascending:\n");
	for(i=0; i<n; i++){
		printf("Index-%d: %d\n",i,arr[i]);
	}

	return 0;
}
```

Coin change:

```
#include <stdio.h>

int minCoins(int coins[], int numCoins, int target) {
    int dp[target + 1];
    dp[0] = 0;

    for (int i = 1; i <= target; i++) {
        dp[i] = target + 1;
    }

    for (int i = 1; i <= target; i++) {
        for (int j = 0; j < numCoins; j++) {
            if (coins[j] <= i) {
                int subResult = dp[i - coins[j]];
                if (subResult != target + 1 && subResult + 1 < dp[i]) {
                    dp[i] = subResult + 1;
                }
            }
        }
    }

    return dp[target];
}

int maxCoins(int coins[], int numCoins, int target) {
    int dp[target + 1];
    dp[0] = 0;

    for (int i = 1; i <= target; i++) {
        dp[i] = -1;
    }

    for (int i = 1; i <= target; i++) {
        for (int j = 0; j < numCoins; j++) {
            if (coins[j] <= i) {
                int subResult = dp[i - coins[j]];
                if (subResult != -1 && subResult + 1 > dp[i]) {
                    dp[i] = subResult + 1;
                }
            }
        }
    }

    return dp[target];
}

int main() {
    int numCoins;
    printf("Enter the number of coins: ");
    scanf("%d", &numCoins);

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

    int target;
    printf("Enter the target value: ");
    scanf("%d", &target);

    int minCount = minCoins(coins, numCoins, target);
    int maxCount = maxCoins(coins, numCoins, target);

    printf("Minimum number of coins required: %d\n", minCount);
    printf("Maximum number of coins required: %d\n", maxCount);

    return 0;
}
```

BFS:

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

#define MAX_SIZE 100

// Queue implementation
typedef struct {
    int items[MAX_SIZE];
    int front;
    int rear;
} Queue;

void enqueue(Queue* queue, int item) {
    if (queue->rear == MAX_SIZE - 1) {
        printf("Queue is full\n");
    } else {
        if (queue->front == -1) {
            queue->front = 0;
        }
        queue->rear++;
        queue->items[queue->rear] = item;
    }
}

int dequeue(Queue* queue) {
    int item;
    if (queue->front == -1 || queue->front > queue->rear) {
        printf("Queue is empty\n");
        return -1;
    } else {
        item = queue->items[queue->front];
        queue->front++;
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
        return item;
    }
}

bool isEmpty(Queue* queue) {
    return queue->front == -1;
}

// BFS traversal
void BFS(int adjacencyMatrix[][MAX_SIZE], int vertices, int startVertex) {
    bool visited[MAX_SIZE] = { false };

    Queue queue;
    queue.front = -1;
    queue.rear = -1;

    visited[startVertex] = true;
    enqueue(&queue, startVertex);

    printf("BFS traversal: ");

    while (!isEmpty(&queue)) {
        int currentVertex = dequeue(&queue);
        printf("%d ", currentVertex);

        for (int i = 0; i < vertices; ++i) {
            if (adjacencyMatrix[currentVertex][i] == 1 && !visited[i]) {
                visited[i] = true;
                enqueue(&queue, i);
            }
        }
    }

    printf("\n");
}

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

    int adjacencyMatrix[MAX_SIZE][MAX_SIZE];

    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < vertices; ++i) {
        for (int j = 0; j < vertices; ++j) {
            scanf("%d", &adjacencyMatrix[i][j]);
        }
    }

    int startVertex;
    printf("Enter the starting vertex: ");
    scanf("%d", &startVertex);

    BFS(adjacencyMatrix, vertices, startVertex);

    return 0;
}
```

DFS:

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

#define MAX_SIZE 100

void DFS(int adjacencyMatrix[][MAX_SIZE], int vertices, int startVertex, bool visited[]) {
    visited[startVertex] = true;

    for (int i = 0; i < vertices; i++) {
        if (adjacencyMatrix[startVertex][i] == 1 && !visited[i]) {
            DFS(adjacencyMatrix, vertices, i, visited);
        }
    }

    printf("%d ", startVertex); // Print the vertex after traversing its left and right subtrees
}

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

    int adjacencyMatrix[MAX_SIZE][MAX_SIZE];

    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &adjacencyMatrix[i][j]);
        }
    }

    int startVertex;
    printf("Enter the starting vertex: ");
    scanf("%d", &startVertex);

    bool visited[MAX_SIZE] = { false };

    printf("Inorder traversal: ");
    DFS(adjacencyMatrix, vertices, startVertex, visited);
    printf("\n");

    return 0;
}
```

Adjacency input:

```
0 1 1 0 0 0
1 0 0 1 1 0
1 0 0 0 0 0
0 1 0 0 0 1
0 1 0 0 0 0
0 0 0 1 0 0
```

LCS:

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

#define MAX_LENGTH 100

int max(int a, int b) {
    return (a > b) ? a : b;
}

void printLCS(char lcs[MAX_LENGTH][MAX_LENGTH], char* X, int m, int n) {
    if (m == 0 || n == 0) {
        return;
    }

    if (lcs[m][n] == 'd') {
        printLCS(lcs, X, m - 1, n - 1);
        printf("%c", X[m - 1]);
    } else if (lcs[m][n] == 'u') {
        printLCS(lcs, X, m - 1, n);
    } else {
        printLCS(lcs, X, m, n - 1);
    }
}

void findLCS(char* X, char* Y) {
    int m = strlen(X);
    int n = strlen(Y);

    int L[m + 1][n + 1];
    char lcs[MAX_LENGTH][MAX_LENGTH];

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                L[i][j] = 0;
            } else if (X[i - 1] == Y[j - 1]) {
                L[i][j] = L[i - 1][j - 1] + 1;
                lcs[i][j] = 'd'; // diagonal arrow
            } else {
                if (L[i - 1][j] >= L[i][j - 1]) {
                    L[i][j] = L[i - 1][j];
                    lcs[i][j] = 'u'; // upward arrow
                } else {
                    L[i][j] = L[i][j - 1];
                    lcs[i][j] = 'l'; // leftward arrow
                }
            }
        }
    }

    printf("\nLongest Common Subsequence (LCS): ");
    printLCS(lcs, X, m, n);

    printf("\nLength of LCS: %d\n", L[m][n]);
}

int main() {
    char X[MAX_LENGTH];
    char Y[MAX_LENGTH];

    printf("Enter the first sequence: ");
    scanf("%s", X);

    printf("Enter the second sequence: ");
    scanf("%s", Y);

    findLCS(X, Y);

    return 0;
}
```

LIS:

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

int max(int a, int b) {
    return (a > b) ? a : b;
}

void printLIS(int* arr, int* lis, int n) {
    int maxLength = lis[0];
    int maxIndex = 0;
    for (int i = 1; i < n; i++) {
        if (lis[i] > maxLength) {
            maxLength = lis[i];
            maxIndex = i;
        }
    }

    int* lisSeq = (int*)malloc(sizeof(int) * maxLength);
    lisSeq[maxLength - 1] = arr[maxIndex];
    int j = maxLength - 1;

    for (int i = maxIndex - 1; i >= 0; i--) {
        if (arr[i] < arr[maxIndex] && lis[i] == lis[maxIndex] - 1) {
            lisSeq[--j] = arr[i];
            maxIndex = i;
        }
    }

    printf("\nLongest Increasing Subsequence (LIS): ");
    for (int i = 0; i < maxLength; i++) {
        printf("%d ", lisSeq[i]);
    }
    printf("\n");

    printf("Length of LIS: %d\n", maxLength);

    free(lisSeq);
}

void findLIS(int* arr, int n) {
    int* lis = (int*)malloc(sizeof(int) * n);

    for (int i = 0; i < n; i++) {
        lis[i] = 1;
    }

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (arr[i] > arr[j]) {
                lis[i] = max(lis[i], lis[j] + 1);
            }
        }
    }

    printLIS(arr, lis, n);

    free(lis);
}

int main() {
    int n;
    printf("Enter the number of elements in the sequence: ");
    scanf("%d", &n);

    int* arr = (int*)malloc(sizeof(int) * n);

    printf("Enter the elements of the sequence:\n");
    for (int i = 0; i < n; i++) {
                    printf("Element-%d: ",i+1);
        scanf("%d", &arr[i]);
    }

    findLIS(arr, n);

    free(arr);

    return 0;
}
```

Knapsack:

```
# include<stdio.h>

void knapsack(int n, float weight[], float profit[], float capacity)
{
	float x[n], tp = 0;
	int i, j, u;
	u = capacity;

	for (i = 0; i < n; i++)
		x[i] = 0.0;

	for (i = 0; i < n; i++)
	{
		if (weight[i] > u)
			break;
		else
		{
			x[i] = 1.0;
			tp = tp + profit[i];
			u = u - weight[i];
		}
	}

	if (i < n)
	{
		x[i] = u / weight[i];
	}
	tp = tp + (x[i] * profit[i]);

	printf("\nMaximum profit is:- %f", tp);

}

int main()
{
	int num, i, j;
	printf("Enter the no. of objects: ");
	scanf("%d", &num);

	float weight[num], profit[num], capacity;
	float ratio[num], temp;
```

```
printf("Enter the weights and profits of each object:-\n");
printf("\n\tObject no.\t\tWeight\t\tProfit\n");
for (i = 0; i < num; i++)
{
        printf("\tObject-%d\t\t",i+1);
        scanf("%f", &weight[i]);
        printf("\t\t");
        scanf("%f",&profit[i]);
        printf("\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b");
}

printf("\nEnter the capacity of knapsack: ");
scanf("%f", &capacity);

for (i = 0; i < num; i++)
{
        ratio[i] = profit[i] / weight[i];
}

for (i = 0; i < num; i++)
{
        for (j = i + 1; j < num; j++)
        {
                if (ratio[i] < ratio[j])
                {
                        temp = ratio[j];
                        ratio[j] = ratio[i];
                        ratio[i] = temp;

                        temp = weight[j];

                        weight[j] = weight[i];
                        weight[i] = temp;

                        temp = profit[j];
                        profit[j] = profit[i];
                        profit[i] = temp;
                }
        }
}

knapsack(num, weight, profit, capacity);
return(0);
```

}
Prim’s Algorithm:

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

#define MAX_SIZE 100

int findMinKey(int key[], bool mstSet[], int vertices) {
    int min = INT_MAX, minIndex;

    for (int v = 0; v < vertices; v++) {
        if (mstSet[v] == false && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }

    return minIndex;
}

void printMST(int parent[], int graph[][MAX_SIZE], int vertices) {
    printf("Minimum Spanning Tree:\n");
    for (int i = 1; i < vertices; i++) {
        printf("%c - %c\tWeight: %d\n", parent[i]+'A', i+'A', graph[i][parent[i]]);
    }
}

void primMST(int graph[][MAX_SIZE], int vertices, char vertexNames[]) {
    int parent[MAX_SIZE];
    int key[MAX_SIZE];
    bool mstSet[MAX_SIZE];

    for (int i = 0; i < vertices; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < vertices - 1; count++) {
        int u = findMinKey(key, mstSet, vertices);
        mstSet[u] = true;
```

```
        for (int v = 0; v < vertices; v++) {
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    printMST(parent, graph, vertices);

    int minWeight = 0;
    for (int i = 1; i < vertices; i++) {
        minWeight += graph[i][parent[i]];
    }

    printf("Minimum Weight of MST: %d\n", minWeight);
}

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

    int graph[MAX_SIZE][MAX_SIZE];

    char vertexNames[MAX_SIZE];
    printf("Enter the names of vertices (in uppercase letters):\n");
    for (int i = 0; i < vertices; i++) {
        scanf(" %c", &vertexNames[i]);
    }

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

    primMST(graph, vertices, vertexNames);

    return 0;
}
```

Adjacency input:

0 6 8 4 7 0
5 0 0 0 0 3
8 0 0 2 3 0
4 0 2 1 0 3
7 0 3 0 0 0
0 5 0 3 0 0

Kruskal's Algorithm:

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

#define MAX_SIZE 100

typedef struct {
    int src;
    int dest;
    int weight;
} Edge;

typedef struct {
    Edge* edges[MAX_SIZE];
    int numEdges;
} Graph;

Edge* createEdge(int src, int dest, int weight) {
    Edge* edge = (Edge*)malloc(sizeof(Edge));
    edge->src = src;
    edge->dest = dest;
    edge->weight = weight;
    return edge;
}

Graph* createGraph() {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numEdges = 0;
    return graph;
}

void addEdge(Graph* graph, int src, int dest, int weight) {
    Edge* edge = createEdge(src, dest, weight);
```

```
    graph->edges[graph->numEdges++] = edge;
}

int compareEdges(const void* a, const void* b) {
    Edge* edge1 = *(Edge**)a;
    Edge* edge2 = *(Edge**)b;
    return edge1->weight - edge2->weight;
}

int findParent(int parent[], int vertex) {
    if (parent[vertex] == vertex)
        return vertex;
    return findParent(parent, parent[vertex]);
}

void unionSets(int parent[], int rank[], int vertex1, int vertex2) {
    int root1 = findParent(parent, vertex1);
    int root2 = findParent(parent, vertex2);

    if (rank[root1] < rank[root2])
        parent[root1] = root2;
    else if (rank[root1] > rank[root2])
        parent[root2] = root1;
    else {
        parent[root2] = root1;
        rank[root1]++;
    }
}

void calculateMST(Graph* graph, int vertices, char vertexNames[]) {
    Edge* result[MAX_SIZE];
    int parent[MAX_SIZE];
    int rank[MAX_SIZE];

    for (int v = 0; v < vertices; v++) {
        parent[v] = v;
        rank[v] = 0;
    }

    qsort(graph->edges, graph->numEdges, sizeof(Edge*), compareEdges);

    int numEdges = 0;
    int i = 0;
```

```
    while (numEdges < vertices - 1 && i < graph->numEdges) {
        Edge* nextEdge = graph->edges[i++];

        int root1 = findParent(parent, nextEdge->src);
        int root2 = findParent(parent, nextEdge->dest);

        if (root1 != root2) {
            result[numEdges++] = nextEdge;
            unionSets(parent, rank, root1, root2);
        }
    }

    int minimumWeight = 0;

    printf("\nMinimum Spanning Tree (MST):\n");
    for (int e = 0; e < numEdges; e++) {
        Edge* edge = result[e];
        printf("%c - %c\tWeight: %d\n", vertexNames[edge->src], vertexNames[edge->dest],
edge->weight);
        minimumWeight += edge->weight;
    }

    printf("\nMinimum Weight of MST: %d\n", minimumWeight);
}

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

    char vertexNames[MAX_SIZE];
    int weights[MAX_SIZE];
    int adjacencyMatrix[MAX_SIZE][MAX_SIZE];

    printf("Enter the names of the vertices:\n");
    for (int i = 0; i < vertices; i++) {
        scanf(" %c", &vertexNames[i]);
    }



    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
```

```
            scanf("%d", &adjacencyMatrix[i][j]);
        }
    }

    Graph* graph = createGraph();

    for (int i = 0; i < vertices; i++) {
        for (int j = i + 1; j < vertices; j++) {
            if (adjacencyMatrix[i][j] != 0) {
                addEdge(graph, i, j, adjacencyMatrix[i][j]);
            }
        }
    }

    calculateMST(graph, vertices, vertexNames);

    return 0;
}
```

Adjacency input:

```
0 5 8 4 7 0
5 0 0 4 0 2
8 0 0 2 5 0
4 4 0 0 0 3
7 0 5 0 0 0
0 2 0 3 0 0
```