**QUESTION:**

Write a C program to find a pattern in a given text using Knuth-Morris-Pratt algorithm.

**PSEUDOCODE:**

function computeLPSArray(char\* pattern, int patternLength, int\* lPSArray):

    length, lPSArray[0], index = 0, 0, 1

    while index < patternLengthgth:

        if pattern[index] = pattern[length]:

            length++

            lPSArray[index] = length

            index++

        else:

            if length not 0 then length = lPSArray[length - 1]

            else:

                lPSArray[index] = 0

                index++

function knuthMorrisPratt(char\* text, char\* pattern):

    textLength, patternLength = len(text), len(pattern)

    computeLPSArray(pattern, patternLength, lPSArray)

    index, secondaryIndex = 0

    while index < textLength:

        if pattern[secondaryIndex] = text[index]:

            secondaryIndex++

            index++

        if secondaryIndex = patternLength:

            printf("Pattern found at index %d\n", index - secondaryIndex)

            secondaryIndex = lPSArray[secondaryIndex - 1]

        else if index < textLength and pattern[secondaryIndex] not text[index]:

            if secondaryIndex not 0 then secondaryIndex = lPSArray[secondaryIndex - 1]

            else index = index + 1

function main():

    char text[1000], pattern[1000] = input()

    knuthMorrisPratt(text, pattern)

**CODE:**

#include <stdio.h>

#include <string.h>

void computeLPSArray(char\* pattern, int patternLength, int\* lPSArray)

{

    int length = 0

    lPSArray[0] = 0;

    int index = 1;

    while (index < patternLengthgth)

    {

        if (pattern[index] == pattern[length])

        {

            length++;

            lPSArray[index] = length;

            index++;

        }

        else

        {

            if (length != 0) length = lPSArray[length - 1];

            else

            {

                lPSArray[index] = 0;

                index++;

            }

        }

    }

}

void knuthMorrisPratt(char\* text, char\* pattern)

{

    int textLength = strlen(text);

    int patternLength = strlen(pattern);

    int lPSArray[patternLength];

    computeLPSArray(pattern, patternLength, lPSArray);

    int index = 0;

    int secondaryIndex = 0;

    while (index < textLength)

    {

        if (pattern[secondaryIndex] == text[index])

        {

            secondaryIndex++;

            index++;

        }

        if (secondaryIndex == patternLength)

        {

            printf("Pattern found at index %d\n", index - secondaryIndex);

            secondaryIndex = lPSArray[secondaryIndex - 1];

        }

        else if (index < textLength && pattern[secondaryIndex] != text[index])

        {

            if (secondaryIndex != 0) secondaryIndex = lPSArray[secondaryIndex - 1];

            else index = index + 1;

        }

    }

}

int main()

{

    printf("Name: Afraaz Hussain\nAdmission number: 20BDS0374\n\n\n");

    char text[1000], pattern[1000];

    printf("Enter the text: ");

    scanf("%s", text);

    printf("Enter the pattern: ");

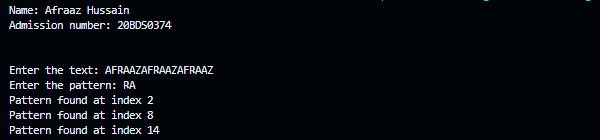
    scanf("%s", pattern);

    knuthMorrisPratt(text, pattern);

    return 0;

}

**OUTPUT:**



**QUESTION:**

Consider set of points Q as input and find the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior using Graham’s Scan algorithm.

**PSEUDOCODE:**

structure:

    int: xCoordinate, yCoordinate

as Point

function orientation(Point p, Point q, Point r):

    value = (q.yCoordinate - p.yCoordinate) \* (r.xCoordinate - q.xCoordinate) - (q.xCoordinate - p.xCoordinate) \* (r.yCoordinate - q.yCoordinate)

    // For co-linear points

    if value = 0 return 0

    // For clockwise pairs

    else if value > 0 return 1

    // For counter-clockwise pairs

    else return 2

function comparePoints(const void\* vp1, const void\* vp2):

    Point\* pointOne = (Point\*)vp1

    Point\* pointTwo = (Point\*)vp2

    int currentOrientation = orientation((Point){0, 0}, *\*pointOne, \**pointTwo)

    if currentOrientation = 0:

        int distanceOne = pointOne -> xCoordinate \* pointOne -> xCoordinate + pointOne -> yCoordinate \* pointOne -> yCoordinate

        int distanceTwo = pointTwo -> xCoordinate \* pointTwo -> xCoordinate + pointTwo -> yCoordinate \* pointTwo -> yCoordinate

        return distanceOne - distanceTwo

    else return -1 if currentOrientation = 2 else 1

function printConvexHull(Point\* points, int numberOfPoints):

    // When there aren't enough ponts for a convex hull

    if numberOfPoints < 3 then return

    qsort(points, numberOfPoints, sizeof(Point), comparePoints)

    int: hull[numberOfPoints]

    top, hull[0], hull[1] = 2, 0, 1

    for index from 2 to numberOfPoints:

        while (top > 0 and orientation(points[hull[top - 1]], points[hull[top]], points[index]) not 2) do top--

        hull[++top] = index

    print("Smallest convex polygon: ")

    for index from 0 to top:

        print("(%d, %d) ", points[hull[index]].xCoordinate, points[hull[index]].yCoordinate)

    print("\n")

function main():

    int: numberOfPoints, index

    numberOfPoints = input()

    Point\* points = malloc(numberOfPoints \* sizeof(Point))

    for index from 0 to numberOfPoints:

        &points[index].xCoordinate, &points[index].yCoordinate = input()

    printConvexHull(points, numberOfPoints)

    free(points)

    return

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

typedef struct

{

    int xCoordinate, yCoordinate;

} Point;

int orientation(Point p, Point q, Point r)

{

    int value = (q.yCoordinate - p.yCoordinate) \* (r.xCoordinate - q.xCoordinate) - (q.xCoordinate - p.xCoordinate) \* (r.yCoordinate - q.yCoordinate);

    // For co-linear points

    if (value == 0) return 0;

    // For clockwise pairs

    else if (value > 0) return 1;

    // For counter-clockwise pairs

    else return 2;

}

int comparePoints(const void\* vp1, const void\* vp2)

{

    Point\* pointOne = (Point\*)vp1;

    Point\* pointTwo = (Point\*)vp2;

    int currentOrientation = orientation((Point){0, 0}, \*pointOne, \*pointTwo);

    if (currentOrientation == 0)

    {

        int distanceOne = pointOne -> xCoordinate \* pointOne -> xCoordinate + pointOne -> yCoordinate \* pointOne -> yCoordinate;

        int distanceTwo = pointTwo -> xCoordinate \* pointTwo -> xCoordinate + pointTwo -> yCoordinate \* pointTwo -> yCoordinate;

        return distanceOne - distanceTwo;

    }

    else return (currentOrientation == 2) ? -1 : 1;

}

void printConvexHull(Point\* points, int numberOfPoints)

{

    // When there aren't enough ponts for a convex hull

    if (numberOfPoints < 3) return;

    qsort(points, numberOfPoints, sizeof(Point), comparePoints);

    int hull[numberOfPoints];

    int top = 2;

    hull[0] = 0;

    hull[1] = 1;

    for (int index = 2; index < numberOfPoints; index++)

    {

        while (top > 0 && orientation(points[hull[top - 1]], points[hull[top]], points[index]) != 2) top--;

        hull[++top] = index;

    }

    printf("Smallest convex polygon: ");

    for (int index = 0; index <= top; index++) printf("(%d, %d) ", points[hull[index]].xCoordinate, points[hull[index]].yCoordinate);

    printf("\n");

}

int main()

{

    printf("Name: Afraaz Hussain\nAdmission number: 20BDS0374\n\n\n");

    int numberOfPoints, index;

    printf("Enter the number of points: ");

    scanf("%d", &numberOfPoints);

    Point\* points = malloc(numberOfPoints \* sizeof(Point));

    for (index = 0; index < numberOfPoints; index++)

    {

        printf("Enter the coordinates of point %d: ", index + 1);

        scanf("%d %d", &points[index].xCoordinate, &points[index].yCoordinate);

    }

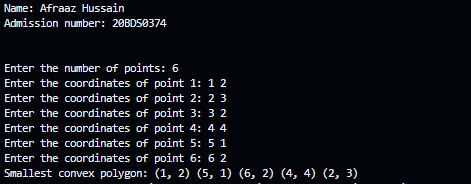
    printConvexHull(points, numberOfPoints);

    free(points);

    return 0;

}

**OUTPUT:**



**QUESTION:**

Consider set of points Q as input and find the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior using Jarvis March algorithm.

**PSEUDOCODE:**

structure:

    int: xCoordinate, yCoordinate

as Point

function orientation(Point p, Point q, Point r):

    value = (q.yCoordinate - p.yCoordinate) \* (r.xCoordinate - q.xCoordinate) - (q.xCoordinate - p.xCoordinate) \* (r.yCoordinate - q.yCoordinate)

    // For co-linear points

    if value = 0 then return 0

    // For clockwise pairs

    else if value > 0 then return 1

    // For counter-clockwise pairs

    else return 2

function printConvexHull(Point\* points, int numberOfPoints):

    // When there aren't enough ponts for a convex hull

    if numberOfPoints < 3 then return

    int\*: hull = malloc(numberOfPoints \* sizeof(int))

    leftmost = 0

    for index from 1 to numberOfPoints do if (points[index].xCoordinate < points[leftmost].xCoordinate) then leftmost = index

    int p = leftmost, q

    int index = 0

    do:

        hull[index++] = p

        q = (p + 1) % numberOfPoints

        for secondaryIndex from 0 to numberOfPoints do if (orientation(points[p], points[secondaryIndex], points[q]) = 2) then q = secondaryIndex

        p = q

    while p not leftmost

    print("\nSmallest convex polygon: ")

    for index from 0 to numberOfPoints:

        secondaryIndex = hull[index]

        print("(%d, %d) ", points[secondaryIndex].xCoordinate, points[secondaryIndex].yCoordinate)

    print("\n")

    free(hull)

function main():

    int: numberOfPoints, index

    numberOfPoints = input()

    Point\* points = malloc(numberOfPoints \* sizeof(Point))

    for index from 0 to numberOfPoints:

        &points[index].xCoordinate, &points[index].yCoordinate = input()

    printConvexHull(points, numberOfPoints)

    free(points)

    return

**CODE:**

#include <stdio.h>

#include <stdlib.h>

typedef struct

{

    int xCoordinate, yCoordinate;

} Point;

int orientation(Point p, Point q, Point r)

{

    int value = (q.yCoordinate - p.yCoordinate) \* (r.xCoordinate - q.xCoordinate) - (q.xCoordinate - p.xCoordinate) \* (r.yCoordinate - q.yCoordinate);

    // For co-linear points

    if (value == 0) return 0;

    // For clockwise pairs

    else if (value > 0) return 1;

    // For counter-clockwise pairs

    else return 2;

}

void printConvexHull(Point\* points, int numberOfPoints)

{

    // When there aren't enough ponts for a convex hull

    if (numberOfPoints < 3) return;

    int\* hull = malloc(numberOfPoints \* sizeof(int));

    int leftmost = 0;

    for (int index = 1; index < numberOfPoints; index++)  if (points[index].xCoordinate < points[leftmost].xCoordinate) leftmost = index;

    int p = leftmost, q;

    int index = 0;

    do

    {

        hull[index++] = p;

        q = (p + 1) % numberOfPoints;

        for (int secondaryIndex = 0; secondaryIndex < numberOfPoints; secondaryIndex++) if (orientation(points[p], points[secondaryIndex], points[q]) == 2) q = secondaryIndex;

        p = q;

    }

    while (p != leftmost);

    printf("\nSmallest convex polygon: ");

    for (int index = 0; index < numberOfPoints; index++)

    {

        int secondaryIndex = hull[index];

        printf("(%d, %d) ", points[secondaryIndex].xCoordinate, points[secondaryIndex].yCoordinate);

    }

    printf("\n");

    free(hull);

}

int main()

{

    printf("Name: Afraaz Hussain\nAdmission number: 20BDS0374\n\n\n");

    int numberOfPoints;

    printf("Enter the number of points: ");

    scanf("%d", &numberOfPoints);

    Point\* points = malloc(numberOfPoints \* sizeof(Point));

    for (int index = 0; index < numberOfPoints; index++)

    {

        printf("Enter the coordinates of point %d: ", index + 1);

        scanf("%d %d", &points[index].xCoordinate, &points[index].yCoordinate);

    }

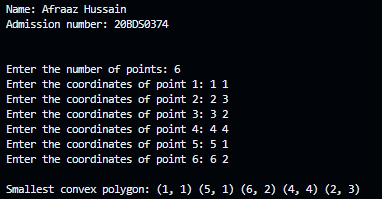
    printConvexHull(points, numberOfPoints);

    free(points);

    return 0;

}

**OUTPUT:**



**QUESTION:**

Implement Floyd-Warshall algorithm and find the shortest path between each pair of vertices in the given graph G.

**PSEUDOCODE:**

define maximumNumberOfVertices 100 // maximum number of vertices in the graph

int: graph[maximumNumberOfVertices][maximumNumberOfVertices] // adjacency matrix of the graph

int: distance[maximumNumberOfVertices][maximumNumberOfVertices] // matrix to store the shortest distanceances between pairs of vertices

function floydWarshall(int numberOfVertices):

    int: index, secondaryIndex, tertiaryIndex

    for index from 0 to numberOfVertices do for secondaryIndex from 0 to numberOfVertices do distance[index][secondaryIndex] = graph[index][secondaryIndex]

    for tertiaryIndex from 0 to numberOfVertices:

        for (index from 0 to numberOfVertices:

            for secondaryIndex from 0 to numberOfVertices:

                if (distance[index][tertiaryIndex] not INT\_MAX and distance[tertiaryIndex][secondaryIndex] not INT\_MAX and distance[index][tertiaryIndex] + distance[tertiaryIndex][secondaryIndex] < distance[index][secondaryIndex]):

                    distance[index][secondaryIndex] = distance[index][tertiaryIndex] + distance[tertiaryIndex][secondaryIndex]

function main():

    int: index, secondaryIndex, u, v, w

    numberOfVertices, numberOfEdges = input()

    // initialize the adjacency matrix with all entries set to infinity

    for (index = 0 index < numberOfVertices index++) for (secondaryIndex = 0 secondaryIndex < numberOfVertices secondaryIndex++) graph[index][secondaryIndex] = INT\_MAX

    // read the edges and their weights

    print("Enter the edges and their weights (u v w):\n")

    for index from 0 to numberOfEdges:

        u, v, w = input()

        graph[u][v] = w

    // run the Floyd-Warshall algorithm

    floydWarshall(numberOfVertices)

    // print the shortest distanceances between all pairs of vertices

    print("Shortest distanceances between all pairs of vertices:\n\n")

    for index from 0 to numberOfVertices:

        for (secondaryIndex from 0 to numberOfVertices:

            if distance[index][secondaryIndex] = INT\_MAX then print("INF\t")

            else print("%d\t", distance[index][secondaryIndex])

        print("\n")

    return

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define maximumNumberOfVertices 100 // maximum number of vertices in the graph

int graph[maximumNumberOfVertices][maximumNumberOfVertices]; // adjacency matrix of the graph

int distance[maximumNumberOfVertices][maximumNumberOfVertices]; // matrix to store the shortest distanceances between pairs of vertices

void floydWarshall(int numberOfVertices)

{

    int index, secondaryIndex, tertiaryIndex;

    for (index = 0; index < numberOfVertices; index++) for (secondaryIndex = 0; secondaryIndex < numberOfVertices; secondaryIndex++) distance[index][secondaryIndex] = graph[index][secondaryIndex];

    for (tertiaryIndex = 0; tertiaryIndex < numberOfVertices; tertiaryIndex++)

    {

        for (index = 0; index < numberOfVertices; index++)

        {

            for (secondaryIndex = 0; secondaryIndex < numberOfVertices; secondaryIndex++)

            {

                if (distance[index][tertiaryIndex] != INT\_MAX && distance[tertiaryIndex][secondaryIndex] != INT\_MAX && distance[index][tertiaryIndex] + distance[tertiaryIndex][secondaryIndex] < distance[index][secondaryIndex])

                {

                    distance[index][secondaryIndex] = distance[index][tertiaryIndex] + distance[tertiaryIndex][secondaryIndex];

                }

            }

        }

    }

}

int main()

{

    printf("Name: Afraaz Hussain\nAdmission number: 20BDS0374\n\n\n");

    int numberOfVertices, numberOfEdges, index, secondaryIndex, u, v, w;

    printf("Enter the number of vertices in the graph: ");

    scanf("%d", &numberOfVertices);

    printf("Enter the number of edges in the graph: ");

    scanf("%d", &numberOfEdges);

    // initialize the adjacency matrix with all entries set to infinity

    for (index = 0; index < numberOfVertices; index++) for (secondaryIndex = 0; secondaryIndex < numberOfVertices; secondaryIndex++) graph[index][secondaryIndex] = INT\_MAX;

    // read the edges and their weights

    printf("Enter the edges and their weights (u v w):\n");

    for (index = 0; index < numberOfEdges; index++)

    {

        scanf("%d %d %d", &u, &v, &w);

        graph[u][v] = w;

    }

    // run the Floyd-Warshall algorithm

    floydWarshall(numberOfVertices);

    // print the shortest distanceances between all pairs of vertices

    printf("Shortest distanceances between all pairs of vertices:\n\n");

    for (index = 0; index < numberOfVertices; index++)

    {

        for (secondaryIndex = 0; secondaryIndex < numberOfVertices; secondaryIndex++)

        {

            if (distance[index][secondaryIndex] == INT\_MAX) printf("INF\t");

            else printf("%d\t", distance[index][secondaryIndex]);

        }

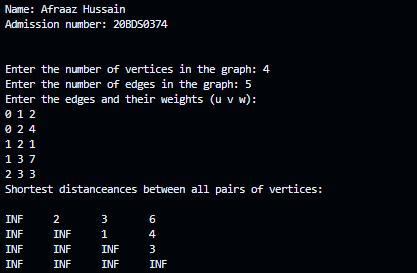
        printf("\n");

    }

    return 0;

}

**OUTPUT:**



**QUESTION:**

Implement Ford Fulkerson algorithm to compute the max-flow of a given graph.

**PSEUDOCODE:**

define maximumNumberOfVertices 100 // maximum number of vertices in the graph

int: graph[maximumNumberOfVertices][maximumNumberOfVertices] // adjacency matrix of the graph

int: parent[maximumNumberOfVertices] // array to store the parent of each vertex in the augmenting path

int: visited[maximumNumberOfVertices] // array to keep track of visited vertices during DFS

int: numberOfVertices // number of vertices in the graph

function min(int firstNumber, int secondNumber) return (firstNumber < secondNumber) ? firstNumber : secondNumber

// find an augmenting path from the source to the sink in the residual graph

function depthFirstSearch(int source, int sink, int minimumFlow):

    int index

    visited[source] = 1

    if source = sink then return minimumFlow

    for index from 0 to numberOfVertices:

        if not visited[index] and graph[source][index] > 0:

            parent[index] = source

            flow = depthFirstSearch(index, sink, min(minimumFlow, graph[source][index]))

            if flow > 0:

                graph[source][index] -= flow

                graph[index][source] += flow

                return flow

    return

// compute the maximum flow from the source to the sink in the given graph

function maximumFlow(int source, int sink):

    maximumFlow = 0

    while True:

        memset(visited, 0, sizeof(visited))

        int flow = depthFirstSearch(source, sink, INT\_MAX)

        if flow = 0 then break

        maximumFlow += flow

    return maximumFlow

function main():

    numberOfVertices = input()

    numberOfEdges = input()

    // initialize the adjacency matrix with all entries set to 0

    memset(graph, 0, sizeof(graph))

    // read the edges and their weights

    print("Enter the edges and their weights (u v w):\n")

    for index from 0 to numberOfEdges:

        scanf("%d %d %d", &u, &v, &w)

        graph[u][v] += w

    source, sink = input()

    // compute the maximum flow from the source to the sink

    maximumFlowValue = maximumFlow(source, sink)

    // print the maximum flow

    print("The maximum flow from vertex %d to vertex %d is: %d\n", source, sink, maximumFlowValue)

    return

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#define maximumNumberOfVertices 100 // maximum number of vertices in the graph

int graph[maximumNumberOfVertices][maximumNumberOfVertices]; // adjacency matrix of the graph

int parent[maximumNumberOfVertices]; // array to store the parent of each vertex in the augmenting path

int visited[maximumNumberOfVertices]; // array to keep track of visited vertices during DFS

int numberOfVertices; // number of vertices in the graph

int min(int firstNumber, int secondNumber) return (firstNumber < secondNumber) ? firstNumber : secondNumber;

// find an augmenting path from the source to the sink in the residual graph

int depthFirstSearch(int source, int sink, int minimumFlow)

{

    int index;

    visited[source] = 1;

    if (source == sink) return minimumFlow;

    for (index = 0; index < numberOfVertices; index++)

    {

        if (!visited[index] && graph[source][index] > 0)

        {

            parent[index] = source;

            int flow = depthFirstSearch(index, sink, min(minimumFlow, graph[source][index]));

            if (flow > 0)

            {

                graph[source][index] -= flow;

                graph[index][source] += flow;

                return flow;

            }

        }

    }

    return 0;

}

// compute the maximum flow from the source to the sink in the given graph

int maximumFlow(int source, int sink)

{

    int index, secondaryIndex, maximumFlow = 0;

    while (1)

    {

        memset(visited, 0, sizeof(visited));

        int flow = depthFirstSearch(source, sink, INT\_MAX);

        if (flow == 0) break;

        maximumFlow += flow;

    }

    return maximumFlow;

}

int main()

{

    printf("Name: Afraaz Hussain\nAdmission number: 20BDS0374\n\n\n");

    int numberOfEdges, index, secondaryIndex, u, v, w, source, sink;

    printf("Enter the number of vertices in the graph: ");

    scanf("%d", &numberOfVertices);

    printf("Enter the number of edges in the graph: ");

    scanf("%d", &numberOfEdges);

    // initialize the adjacency matrix with all entries set to 0

    memset(graph, 0, sizeof(graph));

    // read the edges and their weights

    printf("Enter the edges and their weights (u v w):\n");

    for (index = 0; index < numberOfEdges; index++) {

        scanf("%d %d %d", &u, &v, &w);

        graph[u][v] += w;

    }

    printf("Enter the source and sink vertices: ");

    scanf("%d %d", &source, &sink);

    // compute the maximum flow from the source to the sink

    int maximumFlowValue = maximumFlow(source, sink);

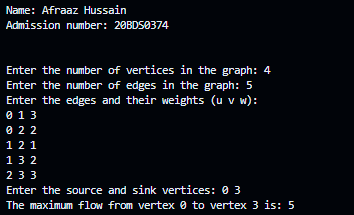
    // print the maximum flow

    printf("The maximum flow from vertex %d to vertex %d is: %d\n", source, sink, maximumFlowValue);

    return 0;

}

**OUTPUT:**

****