1. Ans in c
2. #include<stdio.h>
3. #include<stdlib.h>
4. #include<stdbool.h>
5. #define MAX 50
6. //structure to rep. ecah point in the grid
7. typedef struct{
8. int row, col, dist;
9. } Point;
10. //dire:fwd,r,l,bkwd
11. int dir[4][2];
12. //set the move rules based on the i/p moves
13. void setMoveRule(int x, int y){
14. //fwd
15. dir[0][0]=x;
16. dir[0][1]=y;
17. //r(90)
18. dir[1][0]=y;
19. dir[1][1]=-x;
20. //l(-90)
21. dir[2][0]=-y;
22. dir[2][1]=x;
23. //bkwd(180)
24. dir[3][0]=-x;
25. dir[3][1]=-y;
26. }
27. //check if the move is valid(within bounds & cell=0)
28. bool isValidMove(int grid[MAX][MAX], int row, int col, int m, int n){
29. return (row>=0 && row<m && col>=0 && col<n && grid[row][col]==0);
30. }
31. //BFS to find the shortest path
32. int bfs(int grid[MAX][MAX], int m, int n, int srcX, int srcY, int destX, int destY, int x, int y){
33. //set the move rule
34. setMoveRule(x, y);
35. //queue for bfs
36. Point queue[MAX\* MAX];
37. int front=0, rear=0;
38. //visited array
39. bool visited[MAX][MAX]={false};
40. //start from the src point
41. queue[rear++]=(Point){srcX, srcY, 0};
42. // queue[rear].row=srcX;
43. // queue[rear].col=srcY;
44. // queue[rear].dist=0;
45. // rear++;
46. visited[srcX][srcY]=true;
48. //BFS Loop
49. while(front<rear){
50. //pop the front element
51. Point curr=queue[front]; // 2 lines in 1 line=Point curr=queue[front++];
52. front++;
53. //if destination is reached,return the distance
54. if(curr.row==destX && curr.col==destY){
55. return curr.dist;
56. }
57. //check for all 4 possible moves
58. for(int i=0; i<4; i++){
59. int newRow=curr.row+dir[i][0];
60. int newCol=curr.col+dir[i][1];
61. //if the move is valid & not visited yet
62. if(isValidMove(grid, newRow, newCol, m, n) && !visited[newRow][newCol]){
63. //mark the cell as visited
64. visited[newRow][newCol]=true;
65. //push the cell into the queue
66. queue[rear].row=newRow;
67. queue[rear].col=newCol;
68. queue[rear].dist=curr.dist+1;
69. rear++;
70. //queue[rear++]=(Point){newRow, newCol, curr.dist+1};
71. }
72. }
73. }
74. //if destination is not reachable
75. return -1;
76. }
77. int main(){
78. int m, n;
79. int grid[MAX][MAX];
80. printf("Enter the number of rows and columns: ");
81. scanf("%d %d", &m, &n);
82. printf("\nEnter the grid: \n");
83. for(int i=0; i<m; i++){
84. for(int j=0; j<n; j++){
85. scanf("%d", &grid[i][j]);
86. }
87. }
88. //input source & destination coordinates
89. int srcX, srcY, destX, destY;
90. printf("\nEnter the source coordinates: ");
91. scanf("%d %d", &srcX, &srcY);
92. printf("\nEnter the destination coordinates: ");
93. scanf("%d %d", &destX, &destY);
94. //input the move rule
95. int x, y;
96. printf("\nEnter the move rule coordinates: ");
97. scanf("%d %d", &x, &y);
98. //call bfs function
99. int res=bfs(grid, m, n, srcX, srcY, destX, destY, x, y);
100. printf("\nMinimum number of moves required: %d\n", res);
101. return 0;
102. }

Or

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 50

// Structure to represent each point in the grid

typedef struct {

int row, col, dist;

} Point;

// Directions: forward, right, left, and backward

int directions[4][2];

// Set the movement rules based on the input move (x, y)

void setMoveRules(int x, int y) {

directions[0][0] = x; directions[0][1] = y; // Forward

directions[1][0] = y; directions[1][1] = -x; // Right (90 degrees)

directions[2][0] = -y; directions[2][1] = x; // Left (-90 degrees)

directions[3][0] = -x; directions[3][1] = -y; // Backward (180 degrees)

}

// Check if the move is valid (within bounds and the cell is 0)

bool isValidMove(int grid[MAX][MAX], int row, int col, int M, int N) {

return (row >= 0 && row < M && col >= 0 && col < N && grid[row][col] == 0);

}

// BFS function to find the shortest path

int bfs(int grid[MAX][MAX], int M, int N, int srcRow, int srcCol, int destRow, int destCol, int x, int y) {

setMoveRules(x, y); // Set up movement rules

// Queue for BFS

Point queue[MAX \* MAX];

int front = 0, rear = 0;

// Visited array

bool visited[MAX][MAX] = { false };

// Start from the source point

queue[rear++] = (Point){srcRow, srcCol, 0};

visited[srcRow][srcCol] = true;

// BFS Loop

while (front < rear) {

Point current = queue[front++];

// If destination is reached, return the distance (number of moves)

if (current.row == destRow && current.col == destCol) {

return current.dist;

}

// Try all 4 possible moves

for (int i = 0; i < 4; i++) {

int newRow = current.row + directions[i][0];

int newCol = current.col + directions[i][1];

// If the move is valid and not visited yet

if (isValidMove(grid, newRow, newCol, M, N) && !visited[newRow][newCol]) {

visited[newRow][newCol] = true;

queue[rear++] = (Point){newRow, newCol, current.dist + 1}; // Move and increment distance

}

}

}

// Return -1 if destination can't be reached

return -1;

}

int main() {

int M, N;

int grid[MAX][MAX];

// Input grid dimensions M and N

scanf("%d %d", &M, &N);

// Input the grid

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

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

scanf("%d", &grid[i][j]);

}

}

// Input source and destination coordinates

int srcRow, srcCol, destRow, destCol;

scanf("%d %d", &srcRow, &srcCol);

scanf("%d %d", &destRow, &destCol);

// Input the move rule

int x, y;

scanf("%d %d", &x, &y);

// Call BFS to find the minimum moves and print the result

int result = bfs(grid, M, N, srcRow, srcCol, destRow, destCol, x, y);

printf("%d\n", result);

return 0;

}

Ans in java

import java.util.LinkedList;

import java.util.Queue;

import java.util.Scanner;

class SimpleGame {

// Class to represent each position in the grid

static class Point {

int row, col, dist;

Point(int r, int c, int d) {

row = r;

col = c;

dist = d;

}

}

// Directions: forward, right, left, and backward

static int[][] directions = new int[4][2];

// Set the movement rules based on the input move (x, y)

static void setMoveRules(int x, int y) {

directions[0] = new int[]{x, y}; // Forward

directions[1] = new int[]{y, -x}; // Right (90 degrees)

directions[2] = new int[]{-y, x}; // Left (-90 degrees)

directions[3] = new int[]{-x, -y}; // Backward (180 degrees)

}

// Check if the move is valid (within bounds and the cell is 0)

static boolean isValidMove(int[][] grid, int row, int col) {

return (row >= 0 && row < grid.length && col >= 0 && col < grid[0].length && grid[row][col] == 0);

}

// BFS function to find the shortest path

static int bfs(int[][] grid, int[] src, int[] dest, int x, int y) {

setMoveRules(x, y); // Set up movement rules

Queue<Point> queue = new LinkedList<>();

boolean[][] visited = new boolean[grid.length][grid[0].length];

queue.add(new Point(src[0], src[1], 0)); // Start from source

visited[src[0]][src[1]] = true; // Mark source as visited

while (!queue.isEmpty()) {

Point current = queue.poll();

// If destination is reached, return the distance (number of moves)

if (current.row == dest[0] && current.col == dest[1]) {

return current.dist;

}

// Try all 4 possible moves

for (int[] move : directions) {

int newRow = current.row + move[0];

int newCol = current.col + move[1];

// If the move is valid and not visited yet

if (isValidMove(grid, newRow, newCol) && !visited[newRow][newCol]) {

visited[newRow][newCol] = true;

queue.add(new Point(newRow, newCol, current.dist + 1)); // Move and increment distance

}

}

}

// Return -1 if destination can't be reached

return -1;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// Input grid dimensions M and N

int M = sc.nextInt();

int N = sc.nextInt();

// Input the grid

int[][] grid = new int[M][N];

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

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

grid[i][j] = sc.nextInt();

}

}

// Input source and destination coordinates

int[] src = new int[]{sc.nextInt(), sc.nextInt()};

int[] dest = new int[]{sc.nextInt(), sc.nextInt()};

// Input the move rule

int x = sc.nextInt();

int y = sc.nextInt();

// Call BFS to find the minimum moves and print the result

int result = bfs(grid, src, dest, x, y);

System.out.println(result);

}

}

2. ans in c

#include <stdio.h>

#include <string.h>

#include <corecrt\_malloc.h>

// Function to find if x can be formed from y

int can\_form(char\* x, char\* y, int lenX, int lenY, int S, int R, int\* res) {

int\* dp = (int\*)malloc((lenX + 1) \* sizeof(int)); // Dynamic programming array to store the minimum number of substrings needed

int\* path = (int\*)malloc((lenX + 1) \* sizeof(int)); // To track from where the substring is taken (0 = from Y, 1 = from reversed Y)

dp[0] = 0; // No substrings needed to form an empty string

// Iterate over the length of X

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

dp[i] = 1000000000; // Set to a large value initially

// Try matching a substring from Y

for (int j = 1; j <= lenY && j <= i; j++) {

if (strncmp(x+ i - j, y + lenY - j, j) == 0) {

if (dp[i - j] + 1 < dp[i]) {

dp[i] = dp[i - j] + 1;

path[i] = 0; // From Y

}

}

// Try matching a substring from reversed Y

if (strncmp(x + i - j, y, j) == 0) {

if (dp[i - j] + 1 < dp[i]) {

dp[i] = dp[i - j] + 1;

path[i] = 1; // From reversed Y

}

}

}

}

if (dp[lenX] == 1000000000) return 0; // If dp[lenX] is unchanged, X cannot be formed from Y

int subY = 0, subRevY = 0;

for (int i = lenX; i > 0;) {

if (path[i] == 0) subY++;

else subRevY++;

i -= (subY + subRevY); // Move back by the number of substrings

}

\*res = subY \* S + subRevY \* R; // Calculate the String Factor

free(dp);

free(path);

return 1;

}

int main() {

char x[10001], y[10001], revY[10001];

int S, R;

int lenX=0,lenY=0;

// Input strings X and Y

scanf("%s", x);

scanf("%s", y);

// Input S and R

scanf("%d %d", &S, &R);

int lenX = strlen(x);

int lenY = strlen(y);

int res=0;

if(can\_form(x, y, lenX, lenY, S, R,&res)) {

printf("%d\n", res);

} else{

printf("-1\n");

}

return 0;

}

3.ans in c

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

// Function to check if a word is a valid number

int is\_number(char \*word) {

int i;

for (i = 0; word[i] != '\0'; i++) {

if (word[i] < '0' || word[i] > '9') {

return 0;

}

}

return 1;

}

// Function to convert a word to an integer

int word\_to\_int(char \*word) {

int num = 0;

int i;

for (i = 0; word[i] != '\0'; i++) {

num = num \* 10 + (word[i] - '0');

}

return num;

}

// Function to evaluate the expression

int evaluate\_expression(char \*\*words, int num\_words) {

int stack[100];

int top = -1;

int i;

for (i = 0; i < num\_words; i++) {

if (strcmp(words[i], "add") == 0) {

stack[top - 1] = stack[top - 1] + stack[top];

top--;

} else if (strcmp(words[i], "sub") == 0) {

stack[top - 1] = stack[top - 1] - stack[top];

top--;

} else if (strcmp(words[i], "mul") == 0) {

stack[top - 1] = stack[top - 1] \* stack[top];

top--;

} else if (strcmp(words[i], "rem") == 0) {

stack[top - 1] = stack[top - 1] % stack[top];

top--;

} else if (strcmp(words[i], "pow") == 0) {

stack[top - 1] = pow(stack[top - 1], stack[top]);

top--;

} else if (is\_number(words[i])) {

stack[++top] = word\_to\_int(words[i]);

} else {

printf("expression evaluation stopped invalid words present\n");

return -1;

}

}

if (top != 0) {

printf("expression is not complete or invalid\n");

return -1;

}

return stack[0];

}

int main() {

char expression[100];

char \*words[20];

int num\_words = 0;

int result;

fgets(expression, 100, stdin);

// Tokenize the expression into words

words[num\_words] = strtok(expression, " ");

while (words[num\_words] != NULL) {

num\_words++;

words[num\_words] = strtok(NULL, " ");

}

// Evaluate the expression

result = evaluate\_expression(words, num\_words);

if (result != -1) {

printf("%d\n", result);

}

return 0;

}