the sol to the toggle problem

#include<iostream>

using namespace std;

int N,M,K;

void populateHash(int input[][20], int row, int hashmap[][21], int N, int M, int &candidateRowCount)

{

    bool found = false;

    for (int i=0; i<candidateRowCount; i++)

    {

        for (int j=0; j<M; j++)

        {

            if(input[row][j] !=hashmap[i][j])

                break;

            else

            {

                if (j== M-1)

                {

                    found = true;

                    hashmap[i][M] +=1;

                    break;

                }

            }

        }

    }

    if (!found)

    {

        for (int i=0; i<M; i++)

        {

            hashmap[candidateRowCount][i] = input[row][i];

        }

        hashmap[candidateRowCount][M] +=1;

        candidateRowCount++;

    }

}

int getMaxCount(int input[][20], int N , int M, int K)

{

    int candidateRowCount=0;

    int hashmap[N][21];

    for(int i=0;i<N;i++)

    for(int j=0;j<M+1;j++)

    hashmap[i][j]=0;

    for(int i=0; i<N; i++)

    {

        int count = 0;

        for (int j=0; j<M; j++)

        {

            if(input[i][j]==0)

                count++;

        }

            int diff= K-count;

            if ((diff >= 0) && ((diff %2)== 0))

                populateHash(input, i, hashmap, N, M, candidateRowCount);

    }

    int maxVal = 0;

    for (int i=0; i<candidateRowCount; i++)

    {

        if(hashmap[i][M]>maxVal)

            maxVal = hashmap[i][M];

    }

    return maxVal;

}

int main()

{

        cin>>N>>M>>K;

        int input[N][20];

        for(int i=0;i<N;i++)

        {

            for(int j=0;j<M;j++)

            cin>>input[i][j];

        }

        int val= getMaxCount(input, N, M, K);

        cout<<val<<"\n";

    return 0;

}

the sol to the rare element problem

#include<iostream>

using namespace std ;

#define qCAP 500

bool ar[21][21];

bool visited[21][21];

int ans[21][21];

int xa[]={0,-1,0,1};

int yb[]={1,0,-1,0};

int n;

typedef struct Point{ int x,y;} P;

typedef struct C{ int x,y; int dis;} C;

bool issafe(int x,int y)

{

    return (x>=0 && x<n && y>=0 && y<n && ar[x][y] && !visited[x][y]);

}

C q[qCAP];

int qsize  = 0;

int qrear  = qCAP-1;

int qfront = 0;

bool isEmpty()

{ return (qsize == 0); }

bool isFull()

{ return (qsize == qCAP); }

C peek()

{

    if(!isEmpty())

    {

        C temp;

        temp.x = q[qfront].x;

        temp.y = q[qfront].y;

        temp.dis = q[qfront].dis;

        return temp;

    }

}

bool push(C in)

{

    if (isFull())

        return false;

    qrear = (qrear + 1) % qCAP;

    qsize++;

    q[qrear].x = in.x;

    q[qrear].y = in.y;

    q[qrear].dis = in.dis;

    return true;

}

void pop()

{

    if (isEmpty())

        return;

    qfront = (qfront + 1) % qCAP;

    qsize--;

}

void bfs(int x,int y)

{

    C in;

    in.x=x;

    in.y=y;

    in.dis=0;

    push(in);

    visited[x][y]=1;

    while(qsize>0)

    {

        C c = peek();

        pop();

        int a=c.x;

        int b=c.y;

        int d=c.dis;

        ans[a][b]=d;

        for(int i=0;i<4;i++)

        {

            int nx=a+xa[i];

            int ny=b+yb[i];

            if(issafe(nx,ny))

            {

                visited[nx][ny]=1;

                in.x=nx;

                in.y=ny;

                in.dis=d+1;

                push(in);

            }

        }

    }

}

int main()

{

    cin>>n;

    for(int i=0;i<n;i++)

        for(int j=0;j<n;j++)

            cin>>ar[i][j];

    int q;

    cin>>q;

    P rare[q];

    int fans=10000;

    int mx=-1;

    for(int i=0;i<q;i++)

    {

        int a,b;

        cin>>a>>b;

        rare[i].x=a;

        rare[i].y=b;

    }

    for(int i=0;i<n;i++)

    {

        for(int j=0;j<n;j++)

        {

            for(int i=0;i<21;i++)

            for(int j=0;j<21;j++)

            ans[i][j] = 9999999;

            int flag=0;

            for(int i=0;i<21;i++)

            for(int j=0;j<21;j++)

            visited[i][j] = 0;

            mx=-1;

            if(ar[i][j])

            {

                bfs(i,j);

                    for(int k=0;k<q;k++)

                        mx=max(mx,ans[rare[k].x][rare[k].y]);

                fans=min(fans,mx);

            }

        }

    }

    cout<<fans<<endl;

}

Given a 2 D matrix where 1 represent the places where the frog can jump and 0 represent the empty spaces, the frog can move freely in horizontal direction (on 1’s only) without incurring any cost (jump). A vertical jump from a given point of the matrix to other point on the matrix can be taken (on 1’s only) with cost as the number of jumps taken.

Given a source and destination, the frog has to reach the destination minimizing the cost (jump).

0-1 BFS

1 0 0 0

1 1 0 1

1 0 0 1

1 1 1 1

source - (2, 0)

dest - (1, 3)

o/p - 3

#include<bits/stdc++.h>

#define in\_range(x, y, r, c) (x >= 0 && x < r && y >= 0 && y < c)

using namespace std;

int mat[4][4];

int dis[4][4];

bool vis[4][4];

void init(){

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

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

dis[i][j] = INT\_MAX;

vis[i][j] = false;}}

}

void bfs(int startX, int startY, int r, int c)

{

dis[startX][startY] = 0;

vis[startX][startY] = true;

deque< pair<int, int> > q;

pair<int, int> p;

q.push\_front(make\_pair(startX, startY));

while(!q.empty())

{

p = q.front();

q.pop\_front();

int x = p.first;

int y = p.second;

vis[x][y] = true;

int ha[] = {0, 0}, hb[] = {-1, 1};

for(int i=0;i<2;i++)

{

int tmpX = x + ha[i];

int tmpY = y + hb[i];

if(in\_range(tmpX, tmpY, r, c))

{

if(!vis[tmpX][tmpY]&&mat[tmpX][tmpY]==1&&dis[tmpX][tmpY]>dis[x][y])

{

                    q.push\_front({tmpX, tmpY});

dis[tmpX][tmpY] = dis[x][y];

}

}

}

int va[] = {-1, 1}, vb[] = {0, 0};

for(int i=0;i<2;i++)

{

int tmpX = x + va[i];

int tmpY = y + vb[i];

if(in\_range(tmpX, tmpY, r, c))

{

if(!vis[tmpX][tmpY]&&mat[tmpX][tmpY]==1&&dis[tmpX][tmpY]>dis[x][y]+1)

{

                    q.push\_front({tmpX, tmpY});

dis[tmpX][tmpY] = dis[x][y]+1;

}

}

}

}

}

int main(){

    ios\_base::sync\_with\_stdio(false);

    cin.tie(NULL);

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

for(int j=0;j<4;j++)

cin>>mat[i][j];

}

init();

bfs(2, 0, 4, 4);

cout << dis[1][3] << endl;

    return 0;

}

There is an island surrounded by oil mines. You will be given n companies and m oil mines having values. You have to distribute the mines to "n" companies in fair manner. Remember the companies can have oil mines adjacent to each other and not in between of each others.After distributing them compute the difference of oil mines from the company getting highest and company getting lowest. This number should be minimum.(then only the distribution can be termed as fair).

Example

Input

2

2 4

6 13 10 2

2 4

6 10 13 2

output

5

1

#include<iostream>

using namespace std;

void calculateDiff(int comp[],int n, int &an)

{

int maxa=0,mina=1000;

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

{

maxa = max(maxa,comp[i]);

mina = min(mina,comp[i]);

}

an = min(an,maxa-mina);

}

void calculateTotal(int end,int curr,int oil[],int comp[],int n,int m,int compNo, int &an)

{

if((curr+1)%m==end)

{

for(int j = compNo;j<n;j++)

{

comp[j]+=oil[curr];

calculateDiff(comp,n,an);

comp[j]-=oil[curr];

}

return;

}

if(compNo==n)

return;

comp[compNo]+=oil[curr];

calculateTotal(end,(curr+1)%m,oil,comp,n,m,compNo,an);

calculateTotal(end,(curr+1)%m,oil,comp,n,m,compNo+1,an);

comp[compNo]-=oil[curr];

calculateTotal(end,curr,oil,comp,n,m,compNo+1,an);

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

int t;

cin>>t;

while(t--)

{

    int n,m;

    int an = 10000;

    cin>>n>>m;

    int oil[m];

    for(int i=0;i<m;i++)

cin>>oil[i];

    int comp[n];

    for(int i=0;i<n;i++)

comp[i]=0;

    for(int i=0;i<m;i++)

    calculateTotal(i,i,oil,comp,n,m,0,an);

    printf("%d\n",an);

}

return 0;

}

Given a graph print either of the set of the vertices that are colored with the same

color. And if the graph is not bipartite print “-1”. Test cases also included the cases

when a graph is not connected.

#include <bits/stdc++.h>

using namespace std;

#define MAX 20

int g[MAX][MAX];

int color[MAX];

bool graphcolor(int g[][MAX], int color[], int s, int N)

{

    if(g[s][s]==1) return false;

    color[s] = 1;

    queue<int> q;

    q.push(s);

    while(!q.empty())

    {

        int u = q.front();

        q.pop();

        if(g[u][u]==1) return false;

        for(int i=0;i<N;i++)

        {

            if(g[u][i]==1 && color[i]==-1)

            {

                color[i] = 1-color[u];

                q.push(i);

            }

            else if(g[u][i]==1 && color[i]==color[u])

            return false;

        }

    }

    return true;

}

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

int N;

cin>>N;

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

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

cin>>g[i][j];

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

color[i] = -1;

bool ifpossible = true;

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

{

    if(color[i]==-1)

    {

        ifpossible = (ifpossible && graphcolor(g,color,i,N));

        if(!ifpossible)

        break;

    }

}

if(!ifpossible)

cout<<"-1\n";

else

{

    for(int i=0;i<N;i++)

    {

        if(color[i]==1)

        cout<<i<<" ";

    }

}

return 0;

}

Picking up Jewels

There is a maze that has one entrance and one exit.

Jewels are placed in passages of the maze.

You want to pick up the jewels after getting into the maze through the entrance and before getting out of it through the exit.

You want to get as many jewels as possible, but you don’t want to take the same passage you used once.

When locations of a maze and jewels are given,

find out the greatest number of jewels you can get without taking the same passage twice, and the path taken in this case.

Time limit : 1 sec (Java : 2 sec)

[Input]

There can be more than one test case in the input file. The first line has T, the number of test cases.

Then the totally T test cases are provided in the following lines (T ≤ 10 )

In each test case,

In the first line, the size of the maze N (1 ≤ N ≤ 10) is given. The maze is N×N square-shaped.

From the second line through N lines, information of the maze is given.

“0” means a passage, “1” means a wall, and “2” means a location of a jewel.

The entrance is located on the upper-most left passage and the exit is located on the lower-most right passage.

There is no case where the path from the entrance to the exit doesn’t exist.

[Output]

From the first line through N lines, mark the path with 3 and output it.

In N+1 line, output the greatest number of jewels that can be picked up.

Each test case must be output separately as a empty

[Input Example]

2

5

0 0 0 2 0

2 1 0 1 2

0 0 2 2 0

0 1 0 1 2

2 0 0 0 0

6

0 1 2 1 0 0

0 1 0 0 0 1

0 1 2 1 2 1

0 2 0 1 0 2

0 1 0 1 0 1

2 0 2 1 0 0

[Output Example]

3 0 3 3 3

3 1 3 1 3

3 0 3 2 3

3 1 3 1 3

3 3 3 0 3

6

3 1 2 1 0 0

3 1 3 3 3 1

3 1 3 1 3 1

3 2 3 1 3 2

3 1 3 1 3 1

3 3 3 1 3 3

4

#include<iostream>

#define f(a,b,c) for(int a=b;a<c;a++)

using namespace std;

int N, max\_jewel;

bool visited[11][11];

int T,t, arr[11][11],i,j;

int path[11][11];

int out[11][11];

int xrow[] = {0,0,1,-1};

int ycol[] = {1,-1,0,0};

void save\_path(){

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

    for(int j=0;j<N;j++)

        out[i][j]=path[i][j];

}

bool isSafe(int arr[][11], bool visited[][11], int i, int j)

{

    return (i>=0 && i<N && j>=0 && j<N && arr[i][j]!=1 && !visited[i][j]);

}

void search(int row, int col, int total)

{

    visited[row][col] = true;

    path[row][col] = 3;

    if(row==0 && col==0 && arr[row][col]==2)

    total++;

    if(row==N-1 && col==N-1)

    {

     if(total>max\_jewel)

     {

        max\_jewel = total;

        save\_path();

     }

     visited[row][col]=false;

     return;

    }

    f(i,0,4)

    {

        int nextx = row + xrow[i];

        int nexty = col + ycol[i];

        if(isSafe(arr,visited,nextx,nexty))

        {

           if(arr[nextx][nexty]==2)

           search(nextx,nexty,total+1);

           else

           search(nextx,nexty,total);

           path[nextx][nexty] = arr[nextx][nexty];

        }

    }

    visited[row][col] = false;

    return;

}

int main(){

    ios\_base::sync\_with\_stdio(false);

    cin.tie(NULL);

    int total\_zero;

    cin>>T;

    f(t,0,T)

    {

        cin>>N;

        f(i,0,11)

        f(j,0,11)

        visited[i][j]=false;

        total\_zero=0;

        max\_jewel = -1;

        f(i,0,N)

        f(j,0,N)

        {

            cin>>arr[i][j];

            if(arr[i][j]==0)

            total\_zero++;

        }

        if(total\_zero<N\*N)

        {

            search(0,0,0);

            f(i,0,N)

            f(j,0,N)

            if(out[i][j]!=3)

            out[i][j]=arr[i][j];

            f(i,0,N)

            {

                f(j,0,N)

                printf("%d ",out[i][j]);

                printf("\n");

            }

            printf("%d\n\n",max\_jewel);

        }

        //print all 0's if each cell is empty

        else

        {

            f(i,0,N)

            {

                f(j,0,N)

                printf("0 ");

                printf("\n");

            }

            printf("0\n\n");

        }

    }

return 0;

}

space-ship wormhole problem

There is a source (S) and destination (D) and a spacecraft has to go from S to D. There are N number of wormholes in between

which has following properties:

 Each wormhole has an entry and an exit.

 Each wormhole is bi-directional i.e. one can enter and exit from any of the ends.

The time to cross the wormhole is given and the space craft may or may not use the wormhole

to reach D.

The time taken to travel outside wormhole between two points (x1, y1) and (x2, y2) is given by a formula

|x1 - x2| + |y1 - y2|

where, (x1, y1) and (x2, y2) are the co-ordinates of two points.

The co-ordinates of S and D are given and we have to find the minimum time to reach D from S.

Note: It’s not mandatory to consider all the wormholes

#include<iostream>

#define INF 2147483647

using namespace std;

int ans,mask[100],w[100][5],n;

// returns dis = |sx-dx| + |sy-dy|

int distance(int sx,int sy,int dx,int dy) {

int xd=(sx>dx)?(sx-dx):(dx-sx);

int yd=(sy>dy)?(sy-dy):(dy-sy);

return (xd+yd);

}

void cal(int sx,int sy,int dx,int dy,int dis)

{

    //distance to reach dest without using wormhole

ans=min(ans,distance(sx,sy,dx,dy)+dis);

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

{

if(mask[i]==0)

{

mask[i]=1;

//source to start of a wormhole then to it's end

int temp=distance(sx,sy,w[i][0],w[i][1])+dis+w[i][4];

cal(w[i][2],w[i][3],dx,dy,temp);

//source to end of a wormhole then to it's start

temp=distance(sx,sy,w[i][2],w[i][3])+dis+w[i][4];

cal(w[i][0],w[i][1],dx,dy,temp);

mask[i]=0;

}

}

}

int main()

{

        ios\_base::sync\_with\_stdio(false);

        cin.tie(NULL);

        cin>>n;

        int sx,sy,dx,dy;

cin>>sx>>sy>>dx>>dy;

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

{

mask[i]=0;

for(int j=0;j<5;j++)

cin>>w[i][j];

}

ans=INF;

cal(sx,sy,dx,dy,0);

cout<<ans;

return 0;

}

Mr. Kim has to deliver refrigerators to N customers. From the office, he is going to

visit all the customers and then return to his home. Each location of the office, his

home, and the customers is given in the form of integer coordinates (x,y) (0≤x≤100,

0≤y≤100) . The distance between two arbitrary locations (x1, y1) and (x2, y2) is

computed by |x1-x2| + |y1-y2|, where |x| denotes the absolute value of x; for instance,

|3|=|-3|=3. The locations of the office, his home, and the customers are all distinct.

You should plan an optimal way to visit all the N customers and return to his among

all the possibilities.

You are given the locations of the office, Mr. Kim’s home, and the customers; the

number of the customers is in the range of 5 to 10. Write a program that, starting at

the office, finds a (the) shortest path visiting all the customers and returning to his

home. Your program only have to report the distance of a (the) shortest path.

Constraints

5≤N≤10. Each location (x,y) is in a bounded grid, 0≤x≤100, 0≤y≤100, and x, y are

integers.

Input

You are given 10 test cases. Each test case consists of two lines; the first line has N,

the number of the customers, and the following line enumerates the locations of the

office, Mr. Kim’s home, and the customers in sequence. Each location consists of the

coordinates (x,y), which is reprensented by ‘x y’.

Output

Output the 10 answers in 10 lines. Each line outputs the distance of a (the) shortest

path. Each line looks like ‘#x answer’ where x is the index of a test case. ‘#x’ and

‘answer’ are separated by a space.

I/O Example

Input (20 lines in total. In the first test case, the locations of the office and the home

are (0, 0) and (100, 100) respectively, and the locations of the customers are (70, 40),

(30, 10), (10, 5), (90, 70), (50, 20).)

5 ← Starting test case #1

0 0 100 100 70 40 30 10 10 5 90 70 50 20

6 ← Starting test case #2

88 81 85 80 19 22 31 15 27 29 30 10 20 26 5 14

10 ← Starting test case #3

5/14

39 9 97 61 35 93 62 64 96 39 36 36 9 59 59 96 61 7 64 43 43 58 1 36

Output (10 lines in total)

#1 200

#2 304

#3 366

#include <iostream>

#define INF 2147483647

using namespace std;

int x2,y2;

int x[12];

int y[12];

void solve(int a[],int c,int n,int p, int vertex\_included, int &ans)

{

    if(vertex\_included==n)

{

int s2=c+abs(x[p]-x2)+abs(y[p]-y2);

ans = min(ans,s2);

return;

}

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

{

if(a[i]==0)

{

    a[i]=1;

    solve(a,(c+abs(x[p]-x[i])+abs(y[p]-y[i])),n,i,vertex\_included+1,ans);

    a[i]=0;

    }

}

}

int main()

{

    ios\_base::sync\_with\_stdio(false);

    cin.tie(NULL);

int t;

cin>>t;

for(int j=1;j<=t;j++)

{

    int n,i,c,x1,y1,ans = INF;

    cin>>n>>x1>>y1>>x2>>y2;

    for(i=0;i<n;i++)

    cin>>x[i]>>y[i];

    int a[n] = {0};

    for(i=0;i<n;i++)

    {

     a[i]=1;

     c=abs(x1-x[i])+abs(y1-y[i]);

     solve(a,c,n,i,1,ans);

     a[i]=0;

    }

    printf("#%d %d\n",j,ans);

}

return 0;

}

There is a mobile piece and a stationary piece on the N×M chessboard. The available

moves of the mobile piece are the same as set out in the image below. You need to

capture the stationary piece by moving the mobile piece with the minimum amount of

moves.

Write a program to find out the minimum number moves to catch a piece.

Time limit:1 second (java: 2 seconds)

Input

Several test cases can be included in the inputs. T, the number of cases is given in

the first row of the inputs. After that, the test cases as many as T (T ≤ 20) are given in

a row. N, the numbers of the rows and M, the number of columns of the chessboard

are given in the first row of each test case. R & C is the location information of the

attacking piece and S & K is the location of the defending pieces and are given in the

row at the second line. However, the location of the uppermost end of the left end is

(1, 1)

Output

For each test case, you should print "Case #T" in the first line where T means the

case number. For each test case, you should output the minimum number of

movements to catch a defending piece at the first line of each test case. If not

moveable, output equals ‘-1’.

2

9 9

3 5 2 8

20 20

2 3 7 9

#include <bits/stdc++.h>

using namespace std;

typedef struct

{

    int x; int y; int level;

}data;

int mv[8][2] = {{-2,1},{-1,2},{1,2},{2,1},{2,-1},{1,-2},{-1,-2},{-2,-1}};

int main()

{

    int T;

    cin>>T;

    for(int t=1; t<=T; t++)

    {

        int n,m,r1,c1,r2,c2;

        cin>>n>>m;

        int a[n+1][m+1] = {{0}};

        cin>>r1>>c1>>r2>>c2;

        data d,d1;

        queue<data> q;

        d.x = r1; d.y = c1; d.level = 0;

        q.push(d);

        a[d.x][d.y] = 1; //mark visited

        int tmx,tmy,tml;

        int steps = 0;

        bool f = false;

        while(!q.empty())

        {

            if(f)

                break;

            d = q.front();

            q.pop();

            for(int k=0; k<8; k++)

            {

                tmx = d.x + mv[k][0];

                tmy = d.y + mv[k][1];

                tml = d.level + 1;

                if(tmx && tmx<=n && tmy && tmy<=m && a[tmx][tmy]==0)

                {

                    if(tmx == r2 && tmy == c2)

                    {

                        steps = tml; f = true; break;

                    }

                    d1.x = tmx; d1.y = tmy; d1.level = tml;

                    q.push(d1);

                    a[d1.x][d1.y] = 1; //mark visited

                }

            }

        }

        if(!f)

            steps = -1;

        cout<<"Case #"<<t<<"\n";

        cout<<steps<<"\n";

        cout<<steps<<"\n";

    }

    return 0;

}

You are busy to promote a newly released film in a movie theatre . the title is

'Biochemical Laughing Bomb' which is about terror. Guerillas drop a biochemical

laughing bomb in the middle of a city. once exposed, you have to laugh all your life.

The bomb will contaminate four people around it during t second, and another four

around each of them during another one second. However, you won't be

contaminated if you are not in the adjacent four directions. as the below shows the

9/14

location of the bomb and affected people , and shows contamination process in

seconds and you can figure out that the whole city is contaminated in 8 seconds. In

order to protect the city from the epidemic, create a program that figures out when

the city will be contaminated by the bomb for the last.

Input

Several test cases can be included in the inputs. T, the number of cases is given in

the first row of the inputs. After that, the test cases as many as T (T ≤ 30) are given in

a row. The row and column of the city, N and M are given by being separated with a

blank on the first row of each test case. (1 ≤ N, M ≤ 100) The status within city is

given by being separated with a blank from the second row to N number rows. 1

means people exist and 0 means people do not exist. The coordinate of the row and

column on which the bomb fall is given by being separated with a blank on the last

row.

Output

For each test case, you should print "Case #T" in the first line where T means the

case number. For each test case, you should output how long does it take to

contaminate al people on the first row of each test case.

2

8 7

0 0 1 1 0 0 0

1 1 1 1 0 1 0

0 0 1 1 1 1 1

0 1 1 1 1 1 1

0 1 0 0 1 1 0

0 1 1 1 1 0 0

0 0 1 0 1 1 1

0 0 0 0 1 0 0

2 5

10 10

1 1 1 1 0 1 1 0 0 0

0 1 1 1 1 1 0 1 1 0

0 0 1 1 0 1 0 1 1 1

0 1 1 1 1 1 0 0 1 1

0 1 0 1 0 1 1 1 1 0

0 0 0 0 0 1 1 0 0 0

1 0 1 0 1 0 1 1 0 0

0 0 1 1 1 1 1 1 1 1

1 0 1 0 0 1 0 1 1 0

1 1 1 0 0 0 0 1 1 1

2 2

#include <bits/stdc++.h>

using namespace std;

typedef struct

{

    int x;

    int y;

    int level;

}data;

int main()

{

    ios\_base::sync\_with\_stdio(false);

    cin.tie(NULL);

    int mv[4][2] = {{1,0},{0,1},{-1,0},{0,-1}};

    int T;

    cin>>T;

    for(int t=1; t<=T; t++)

    {

        int n,m,r,c;

        cin>>n>>m;

        int a[n+1][m+1] = {{0}};

        for(int i=1; i<=n; i++)

        {

            for(int j=1; j<=m; j++)

                cin>>a[i][j];

        }

        cin>>r>>c;

        data d,d1;

        queue<data> q;

        int tmx,tmy,tml;

        d.x = r; d.y = c; d.level = 0;

        q.push(d);

        a[d.x][d.y] = 2; //mark visited

        while(!q.empty())

        {

            d = q.front();

            q.pop();

            for(int k=0; k<4; k++)

            {

                tmx = d.x + mv[k][0];

                tmy = d.y + mv[k][1];

                tml = d.level + 1;

                if(a[tmx][tmy] == 1)

                {

                    d1.x = tmx; d1.y = tmy; d1.level = tml;

                    a[d1.x][d1.y] = 2;  //mark visited

                    q.push(d1);

                }

            }

        }

        cout<<"Case #"<<t<<"\n";

        cout<<tml<<"\n";

    }

    return 0;

}

There are N pots. Every pots have some water in it. They may be partially filled. So there is a Overflow Number 0 associated with every pot which tell how many minimum stone pieces are require for that pot to overflow. So if for a pot 0-value is 5 it means minimum 5 stone pieces should be put in that pot to make it overflow. Initially a crow watched those pots and by seeing the water level he anticipated 0-value correctly for every pot ( that is he knew 01 to On). But when he came back in evening he found that every pot is painted from outside and he is not able to know which pot has what 0-value. Crow wants some K pots to overflow so that he can serve his child appropriately. For overflow of pots he need to search for stone in forest( assume that every stone has same size). He wants to use minimum number of stones required to overflow K pots. But only he know the 0-value of pots he doesn't know now which pot has what 0-value. So the task is that in what minimum number of stones he can make K pots overflow in worst case.

Input/Output Specifications Input Specification: 1) A array 0 corresponding to 0-value of N pots {01, 02, On} 2) Number of pots 3) K -value ( number of pots which the crow wants to overflow}

Output Specification: Minimum number of stones required to make K pots overflow in worst case. Or -1 if input is invalid

Example: Let say there are two pots pot 1 has 0 value of 5 , 01= 5 pot 2 has 0 value of 58, 02= 58 Let say crow wants to make one of the pot to overflow. If he know which pot has what 0-value he would simple search for 5 stones and put then in pot 1 to make it overflow. But in real case he doesn't know which pot has what 0-value so just 5 stones may not always work. However he does know that one pot has 0-value S and other has 58. So even in worst case he can make one of the pot overflow just by using 10 stones. He would put 5 stones in one pot if it doesn't overflow he would try the remaining 5 in the other pot which would definitely overflow because one of the pot has 0-value of 5. So the answer for above question is minimum 10 stones even in worst case. Input : Input 1= {5,58} Input 2= 2 Input 3= 1 Output : 10

#include <iostream>

using namespace std;

void merge(int arr[], int l, int mid, int r)

{

    int i,j,k,n1,n2;

    n1 = mid-l+1;

    n2 = r-mid;

    int L[n1];

    int R[n2];

    for(int i=0;i<n1;i++)

    L[i] = arr[l+i];

    for(int j=0;j<n2;j++)

    R[j] = arr[mid+j+1];

    i=0;j=0;k=l;

    while(i<n1 && j<n2)

    {

        if(L[i]<=R[j])

            arr[k++] = L[i++];

        else

            arr[k++] = R[j++];

    }

    while(i<n1)

    {

        arr[k++] = L[i++];

    }

    while(j<n2)

    {

        arr[k++] = R[j++];

    }

}

void mergeSort(int arr[], int l, int r)

{

    if(l<r)

    {

        int mid = l+(r-l)/2;

        mergeSort(arr,l,mid);

        mergeSort(arr,mid+1,r);

        merge(arr,l,mid,r);

    }

}

int Minstones(int arr[],int k,int n)

{

    if(n<k)

    return -1;

    int s1=0,s2=0,i,count;

    mergeSort(arr,0,n-1);

    int maxlimit = arr[k-1];

    for(i=0;i<k;i++)

    s1 += arr[n-1-i];

    for(i=0,count=0;count<k;i++)

    {

        if(arr[n-1-i]>maxlimit)

        s2 += maxlimit;

        else

        {

            s2 += arr[n-1-i];

            count++;

        }

    }

    //for cases like 1 3 3 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 with k=3

    //s1 is less, for cases like 1 3 3 9 9 9 9 with k=3 s2 is less

    //so a last comparison is a must

    //also for cases like 65 69 with k=1, 69 is the answer which is stored in s1

    return (s1>s2)?s2:s1;

}

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

int t;

cin>>t;

while(t--)

{

    int n;

    cin>>n;

    int arr[n];

    for(int i=0;i<n;i++)

    cin>>arr[i];

    int k;

    cin>>k;

    cout<<Minstones(arr,k,n)<<"\n";

}

return 0;

}

5

10

10 6 8 7 4 3 2 1 5 9

6

14

1 3 3 9 9 9 9 9 9 9 9 9 9 9

3

2

58 5

1

18

1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 8 11

2

2

65 69

1

45

27

10

8

69

There are n balloons and n bullets and each balloon is assigned with a particular

number (point). Whenever a particular balloon is shot the no of points increases by

1.the multiplication of point assigned to balloon on left and that of right side.

2.point assigned to left if no right exists

3.point assigned to right if no left exists.

4.the point assigned to itself if no other balloon exists.

You have to output the maximum no of points possible.1 2 3 4

40

1 2 3 4 5

110

3 1 5 8

167

#include <iostream>

using namespace std;

int burstballoon(int nums[], int n)

{

    int M[n+2][n+2] = {{0}};

    for(int l=1;l<=n;++l)

            for(int i=1;i<=n-l+1;++i) //to calculate through all subsets

            {

                int j=i+l-1;          //last index of a subset

                for(int k=i;k<=j;++k)

                    M[i][j] = max(M[i][j],nums[i-1]\*nums[k]\*nums[j+1]+M[i][k-1]+M[k+1][j]);

            }

        return M[1][n];

}

int main() {

//for balloons 3 1 5 8, we'll pass 1 3 1 5 8 1 in function;

int a[] = {1,3,1,5,8,1};

cout<<burstballoon(a,4);

return 0;

}

You have to place an electronic banner of a company as high as it can be, so that whole the city can view the banner standing on top of TWO PILLERS.

The height of two pillers are to be chosen from given array.. say [1, 2, 3, 4, 6]. We have to maximise the height of the two pillars standing side by side, so that the pillars are of EQUAL HEIGHT and banner can be placed on top of it.

In the above array, (1, 2, 3, 4, 6) we can choose pillars like this, say two pillars as p1 and p2..

Then pillars can be,

p1 = 3 unit… Choosing element (3) from array,

Similarly p2 = 3 choosing (2 + 1) from array.

Since, two pillars are equal, we can put board on it…

But we have two maximise the height of the pillars,

And if we check for other heights, we can see p1 = 6 p2 = 4 + 2 which is greater than 3 ( the previous height)..

We have to see if we can further maximize the height… Yes it can be 8.

I.e. p1 = 6 + 2 = 8. p2 = 4 + 3 + 1 = 8.

Both pillars are equal and banner can be placed… And since this is the maximum height attainable for two pillars, we print the answer as 8. In case, there is no combination possible, print 0 (zero).

INPUT :

1

5

1 2 3 4 6

First line is  T  number of test cases to be followed.

Second line of input is number of different pillars.

Third line of input is  different available heights of pillars.

Note : heights of given pillars can be same .. I.e. array can have same elements repeated.

Output.

8

#include<iostream>

using namespace std;

int dp[50][50];

bool exist(int dp[][50],bool vis[],int arr[],int n,int sum)

{

    if(n<0||(sum<0)||(n==0 && sum!=0))

    return 0;

    if(n>=0 && sum==0)

    return 1;

    if(dp[n][sum]==0)

    return 0;

    if(vis[n-1]==1)

    return exist(dp,vis,arr,n-1,sum);

    else

    return exist(dp,vis,arr,n-1,sum)||exist(dp,vis,arr,n-1,sum-arr[n-1]);

}

int height(int arr[],int n,int sum)

{

    for(int i=1;i<=sum;i++)

    dp[0][i]=0;

    for(int i=1;i<=n;i++)

    dp[i][0]=1;

    dp[0][0]=1;

    for(int i=1;i<=n;i++)

    {

        for(int j=1;j<=sum;j++)

        {

            if(j>=arr[i-1])

            dp[i][j]=dp[i-1][j]||dp[i-1][j-arr[i-1]];

            else

            dp[i][j]=dp[i-1][j]; //upar se copy

        }

    }

    for(int k=sum;k>=0;k--)

    {

        bool vec[n] = {0};

        if(dp[n][k]==1)

        {

            int i=n;

            int j=k;

            while( j!=0 && i>=0 )

            {

                if(dp[i-1][j]==1)

                 i--;

                else

                {

                    j=j-arr[i-1];

                    vec[i-1]=1;

                    i--;

                }

            }

            bool tot = exist(dp,vec,arr,n,k);

            if(tot)

            return k;

        }

    }

    return 0;

}

int main()

{

    ios\_base::sync\_with\_stdio(false);

    cin.tie(NULL);

    int t;

    cin>>t;

    while(t--)

    {

        int n;

        cin>>n;

        int arr[n];

        int sum=0;

        for(int i=0;i<n;i++)

        {

            int temp;

            cin>>temp;

            arr[i]=temp;

        }

        for(int i=0;i<n;i++)

        sum+=arr[i];

        sum=sum/2;

        cout<<height(arr,n,sum)<<"\n";

    }

    return 0;

}

[4:02 AM, 11/20/2018] भास्कर शरण: Company S has developed an industrial endoscope available to explore inner part of

the decrepit water pipes. It is possible to explore the inner part of the pipes putting

the endoscope in a certain part of the pipe. The endoscope can be moved in the pipe

only. Meanwhile, when the pipes are connected to each other, if the length of the

endoscope is long enough to explore ,then it is able to inspect the connected pipes.

However, we cannot observe every pipe because the length of endoscope is limited.

When the map of the ground water pipe, the location where the endoscope to out in,

and the length of the endoscope is given, calculate the number of pipe which are

available to explore. Length of endoscope means the range upto which endoscope

can explore. There are seven kind of pipes, and description for each pipe are shown

below:

[4:02 AM, 11/20/2018] भास्कर शरण: When the map of the ground water pipe, the location where the endoscope to out in,

and the length of the endoscope is given, calculate the number of pipe which are

available to explore. Length of endoscope means the range upto which endoscope

can explore.

Input

In the first line, T, the number of total test cases is given. From the second line, T test

cases are given. In the first line of each test case, N, the height of the map of the

ground water pipes, M, the width, R, the vertical location of the water pipe where to

put in the endoscope, C, the horizontal location of it, and the length of the endoscope

L are given. In the following N lines information of the map of ground water pipe is

given. Each line has M numbers. Each number (from 1 to 7) means the type of water

pipe for that point. 0 means there is no water pipe buried in that place.

Output

Print the respective answer for T test cases in total for T lines. The answer is the

number of water pipes which is available to observe using the endoscope.

Constraints

1≤ T ≤100

1≤ N, M ≤50

0≤ X < N

0≤ Y < M

1≤ L ≤ 20

[4:11 AM, 11/20/2018] भास्कर शरण: #include <iostream>

using namespace std;

#define Max 1000

#define row\_size 3

#define col\_size 4

int x = 1, y = 0; // starting index(x, y),

int l = 4; // length of probe tool

// input matrix containing the pipes

int mt[row\_size][col\_size] = { { 0, 0, 4, 0 },

{ 1, 3, 6, 0 },

{ 5, 0, 0, 0 } };

// visited matrix checks for cells already visited

int vi[row\_size][col\_size];

// calculates the depth of connection for each cell

int depth[row\_size][col\_size];

int f = 0;

int r = 0;

// queue for BFS

struct node {

int x;

int y;

int d;

};

node q[Max];

void push(int a, int b, int d) // push function

{

node temp;

temp.x = a;

temp.y = b;

temp.d = d;

q[r++] = temp;

vi[a][b] = 1;

}

node pop() // pop function

{

node temp;

temp.x = q[f].x;

temp.y = q[f].y;

temp.d = q[f].d;

f++;

return temp;

}

bool s1(int i, int j) //if this is open from left side

{

if (i >= 0 && i < row\_size && j >= 0 &&

j < col\_size && vi[i][j] == 0 && (mt[i][j] == 1 ||

mt[i][j] == 3 || mt[i][j] == 6 || mt[i][j] == 7))

return true;

else

return false;

}

bool s2(int i, int j) //if this is open from up

{

if (i >= 0 && i < row\_size && j >= 0 && j < col\_size &&

vi[i][j] == 0 && (mt[i][j] == 1 || mt[i][j] == 2 ||

mt[i][j] == 4 || mt[i][j] == 7))

return true;

else

return false;

}

bool s3(int i, int j) //if this is open from right side

{

if (i >= 0 && i < row\_size && j >= 0 && j < col\_size &&

vi[i][j] == 0 && (mt[i][j] == 1 || mt[i][j] == 3 ||

mt[i][j] == 4 || mt[i][j] == 5))

return true;

else

return false;

}

bool s4(int i, int j) //if this is open from down

{

if (i >= 0 && i < row\_size && j >= 0 && j < col\_size &&

vi[i][j] == 0 && (mt[i][j] == 1 || mt[i][j] == 2 ||

mt[i][j] == 6 || mt[i][j] == 5))

return true;

else

return false;

}

void bfs(int x, int y, int d)

{

push(x, y, d);

while (r > f) //queue not empty

{

node temp = pop();

int i = temp.x;

int j = temp.y;

int c = temp.d;

depth[i][j] = c;

        //if current pipe is open from right side,

        //check if next adjacent is open from left side

if (mt[i][j] == 1 || mt[i][j] == 3 ||

mt[i][j] == 4 || mt[i][j] == 5) {

if (s1(i, j + 1))

push(i, j + 1, c + 1);

}

//if current pipe is open from bottom side,

        //check if next adjacent is open from up side

if (mt[i][j] == 1 || mt[i][j] == 2 ||

mt[i][j] == 6 || mt[i][j] == 5) {

if (s2(i + 1, j))

push(i + 1, j, c + 1);

}

//if current pipe is open from left side,

        //check if next adjacent is open from right side

if (mt[i][j] == 1 || mt[i][j] == 3 ||

mt[i][j] == 7 || mt[i][j] == 6) {

if (s3(i, j - 1))

push(i, j - 1, c + 1);

}

//if current pipe is open from up side,

        //check if next adjacent is open from bottom side

if (mt[i][j] == 1 || mt[i][j] == 2 ||

mt[i][j] == 4 || mt[i][j] == 7) {

if (s4(i - 1, j))

push(i - 1, j, c + 1);

}

}

}

int main()

{

f = 0;

r = 0;

// matrix

for (int i = 0; i < row\_size; i++) {

for (int j = 0; j < col\_size; j++) {

vi[i][j] = 0;

depth[i][j] = Max;

}

}

if (mt[x][y] != 0) //false condition for BFS to start at source

bfs(x, y, 1);

int nc = 0;

for (int i = 0; i < row\_size; i++)

{

for (int j = 0; j < col\_size; j++)

{

if (depth[i][j] <= l)

nc++;

}

}

cout<<nc<< "\n";

}