26/02/2024

GRAPHS CLASS - 5

1. Course Schedule (Leetcode-207)

Problem Statement:

There are a total of **numCourses** courses you have to take, labeled from **0** to **numCourses - 1**. You are given an array **prerequisites** where **prerequisites[i] = [ai, bi]** indicates that you must take course **bi** first if you want to take course **ai**.

For example, the **pair [0, 1]**, indicates that to take **course 0** you have to first take **course 1**.

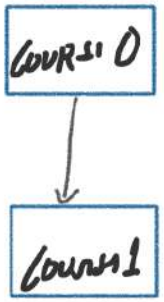
Return **true** if you can finish all courses. Otherwise, return **false**.

Example 1:

Input: numCourses = 2, prerequisites = [[1,0]]

Output: true

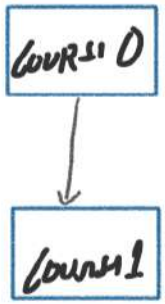
GRAPH



Course 1 lene ke liye pahle course 0 lena ho.
it means course 1 depends on course 0 so we can apply topological sorting to store the dependency order of course.

```
Example 1:
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
```

GRAPH



STEP1

topoOrder

0	1
0	1

STEP2

```
if ( topoOrder.size() == numCourses){
        // All courses are finished
        ↳ return true
}
ELSE
        // All courses are not finished
}
```

STEP1 Create topoOrder using topological sort

STEP2 Check All courses are finished or not

Example 2:
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false

0 1

GRAPH

If you want to take course 1. you have to finish course 0 first.

If you want to take course 0. you have to finish course 1 first.

STEP1

Course 0's Indegree = 1
Course 1's Indegree = 1

STEP2 TopoOrder.size != numCourses
↳ return false

TopoOrder is Always Empty because Topological sorting Algo. push a node jisKi indegree 0 Ho. It means wo node Independent Hai. But yaaha par DONO COURSE 0 and course 1 EK DUSRE PAR HE DEPEND KAR RAHE HAI. jiska matlab courses nahi Finished.

```cpp
// 1. Course Schedule (Leetcode-207)

class Solution {
public:
    void topoSortUsingBFS(int n, vector<int> &topoOrder, unordered_map<int, list<int>> &adjList){...}

    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        // Create adjList first
        unordered_map<int, list<int>> adjList;
        for(auto i: prerequisites){
            int u = i[0];
            int v = i[1];
            adjList[v].push_back(u);
        }

        // Step 1: Create the topo order for checking all courses are finished or not
        vector<int> topoOrder;
        topoSortUsingBFS(numCourses, topoOrder, adjList);

        // Step 2: Check all courses are finished or not
        if(topoOrder.size() == numCourses){
            // Cycle does not present: means all course are finished
            return true;
        }
        else{
            // Cycle does present: means all course are never finished
            return false;
        }
    }
};
```

STEP2

```cpp
void topoSortUsingBFS(int n, vector<int> &topoOrder, unordered_map<int, list<int>> &adjList){
    queue<int> q;
    unordered_map<int,int> indegree;

    // Step 1: initialize the indegree
    for(auto i: adjList){
        for(auto neighbour: i.second){
            indegree[neighbour]++;
        }
    }

    // Step 2: push all nodes jinki indegree zero hai
    for(int node = 0; node < n; node++){
        if(indegree[node] == 0){
            q.push(node);
        }
    }

    // Step 3: BFS on queue to print the order dependency wise
    while(!q.empty()){
        auto frontNode = q.front();
        q.pop();
        topoOrder.push_back(frontNode);

        for(auto neighbour: adjList[frontNode]){
            indegree[neighbour]--;

            // check neighbour node indegree is zero or not
            if(indegree[neighbour] == 0){
                q.push(neighbour);
            }
        }
    }
```

STEP1

2. Course Schedule II (Leetcode-210)

Problem Statement:

There are a total of **numCourses** courses you have to take, labeled from **0** to **numCourses - 1**. You are given an array **prerequisites** where **prerequisites[i] = [ai, bi]** indicates that you must take course **bi** first if you want to take course **ai**.

For example, the **pair [0, 1],** indicates that to take **course 0** you have to first take **course 1**.

Return the **ordering of courses** you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an **empty array**.

Example 1:
Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]

STEP 2 Create Topo Order

0	1
0	1

STEP 2 Return topoOrder when all courses are finished other wise return Empty Array

↳ return

0	1
0	1

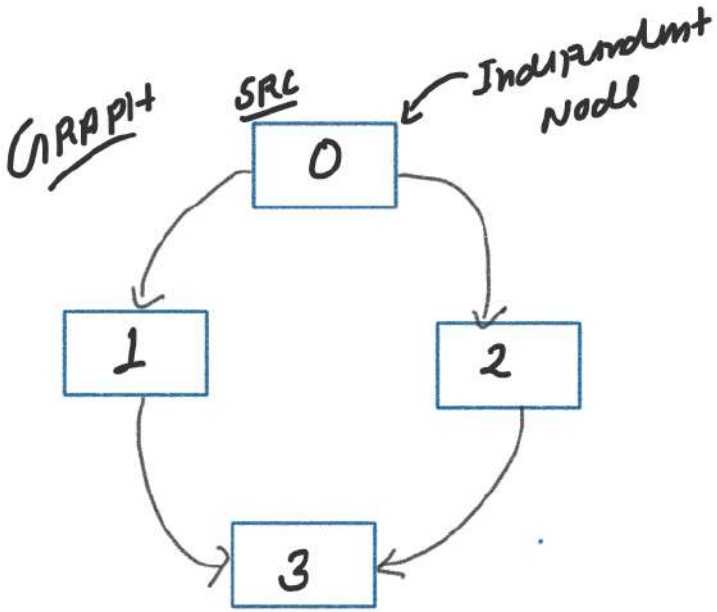
GRAPH

0's Indegree = 0

1's Indegree = 1 → It means 1 depends on 0.

Example 2:

```
Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,2,1,3]
```



Indegree

0's Indegree = 0

1's Indegree = 1

2's Indegree = 1

3's Indegree = 2

STEP 1

TopoOrder

0	1	2	3
0	1	2	3

This is a dependency order

STEP 2

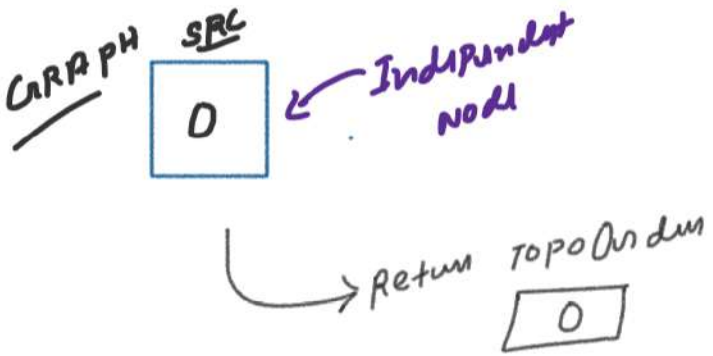
if(TopoOrder.size == Num(v))

↳ return TopoOrder;

Else

↳ return {};

Example 3:
Input: numCourses = 1, prerequisites = []
Output: [0]



STEP 2

if(TopoOrdem.size == NUM(num))
↳ return TopoOrdem;

Else
↳ return {};

```cpp
// 2. Course Schedule II (Leetcode-210)

class Solution {
public:
    void topoSortUsingBFS(int n, vector<int> &topoOrder, unordered_map<int, list<int>> &adjList){...}
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        // Create adjList first
        unordered_map<int, list<int>> adjList;
        for(auto i: prerequisites){
            int u = i[0];
            int v = i[1];
            adjList[v].push_back(u);
        }

        // Create the topo order for checking all courses are finished or not
        vector<int> topoOrder;
        topoSortUsingBFS(numCourses, topoOrder, adjList);

        if(topoOrder.size() == numCourses){
            // Cycle does not present: means all course are finished
            return topoOrder;
        }
        else{
            // Cycle does present: means all course are never finished
            return {};
        }
    }
};
```

STEP2

```cpp
void topoSortUsingBFS(int n, vector<int> &topoOrder, unordered_map<int, list<int>> &adjList){
    queue<int> q;
    unordered_map<int,int> indegree;

    // Step 1: initialize the indegree
    for(auto i: adjList){
        for(auto neighbour: i.second){
            indegree[neighbour]++;
        }
    }

    // Step 2: push all nodes jinki indegree zero hai
    for(int node = 0; node < n; node++){
        if(indegree[node] == 0){
            q.push(node);
        }
    }

    // Step 3: BFS on queue to print the order dependency wise
    while(!q.empty()){
        auto frontNode = q.front();
        q.pop();
        topoOrder.push_back(frontNode);

        for(auto neighbour: adjList[frontNode]){
            indegree[neighbour]--;

            // check neighbour node indegree is zero or not
            if(indegree[neighbour] == 0){
                q.push(neighbour);
            }
        }
    }
```

STEP1

3. Path with Minimum Effort (Leetcode-1631)

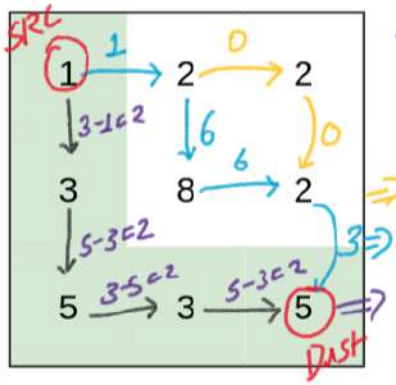
Problem Statement:
*You are a hiker preparing for an upcoming hike. You are given **heights**, a 2D array of size **rows x columns**, where **heights[row][col]** represents the height of cell **(row, col)**. You are situated in the top-left cell, **(0, 0)**, and you hope to travel to the bottom-right cell, **(rows-1, columns-1)** (i.e., **0-indexed**). You can **move up, down, left,** or **right,** and you wish to find a route that requires the minimum **effort**.*

*A route's **effort** is the **maximum absolute difference** in heights between two consecutive **cells** of the route.*

*Return the **minimum effort** required to travel from the top-left cell to the bottom-right cell.*

SRC Dest

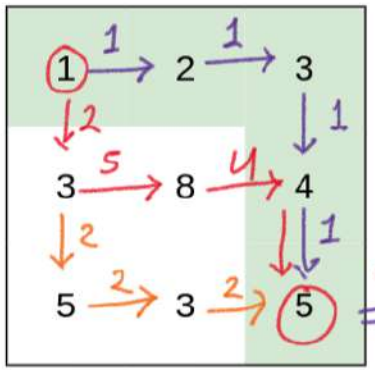
Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2



$\Rightarrow \max(1,0,0,3) = 3$
$\Rightarrow \max(1,6,6,3) = 6$
$\Rightarrow \max(2,2,2,2)$
$\Rightarrow 2$

$\min(3,6,2) = 2$

Example 2:
Input: heights = [[1,2,3],[3,8,4],[5,3,5]]
Output: 1



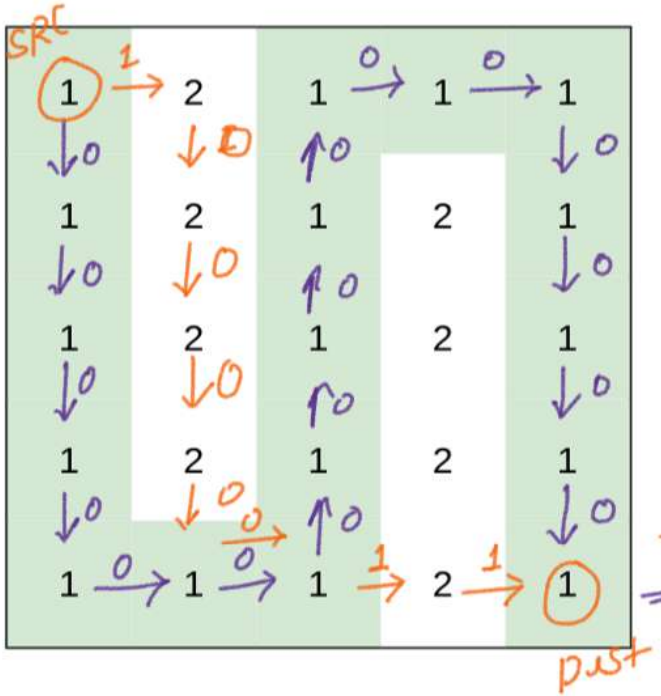
$\Rightarrow \max(2,2,2,2) = 2$
$\Rightarrow \max(2,5,4,1) = 5$
$\Rightarrow \max(1,1,1,1) = 1$

Maximum Effort

Minimum Effort

$\min(2,5,1) = 1$

Example 3:
Input: heights =
[[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,2,1,2,1],[1,1,1,2,1]]
Output: 0



maximum Efforts

$\Rightarrow$ maxEffort = 1

$\Rightarrow$ maxEffort = 0

minimum Efforts

$\hookrightarrow$ min(1,0)

= 0 output

Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2

H

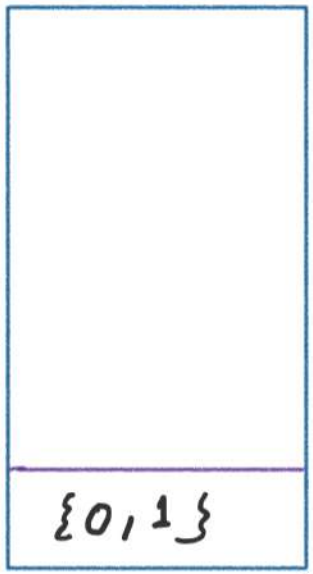
SRC 1	2	2
3	8	2
5	3	5 Dest

Heights

D

0	∞	∞
∞	∞	∞
∞	∞	∞

Difference



MinHeap (mini)

Source = H[x][y]
Dest = H[row-1][col-1]

Initial state

⇒ D[0][0] = 0

⇒ mini.push{0, {0, 0}}

curDiff curX curY

H[curX][curY] = CELL

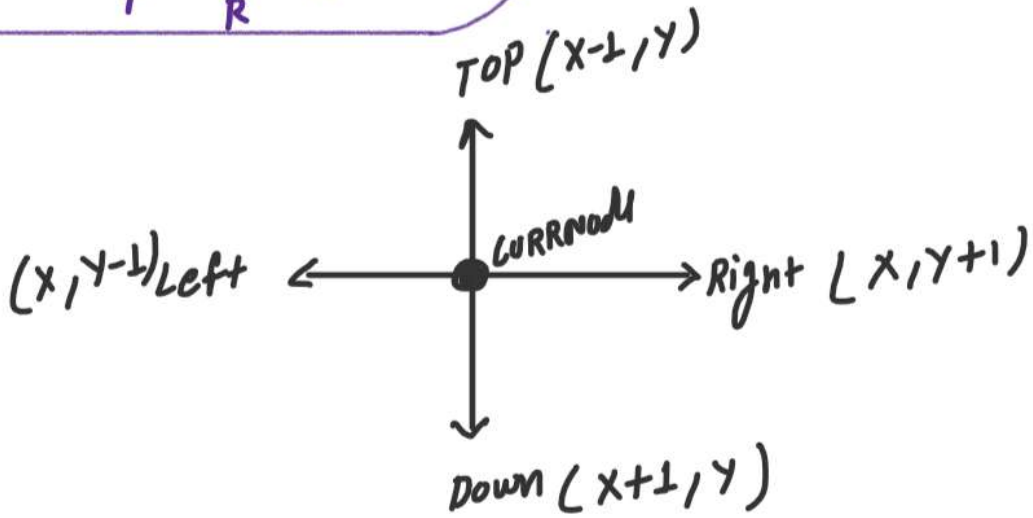
curNode

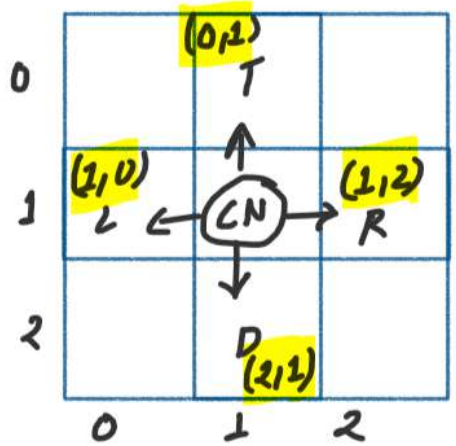
Now we can travel to all NBRNode
i.e. **Top, Right, Down, Left**

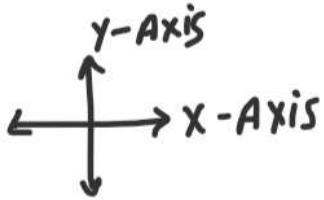
```
int dx[] = {-1,0,1,0};
int dy[] = {0,1,0,-1};
```

T R D L

-1 ⇒ X and Y decrement by 1
0 ⇒ X and Y remains same
1 ⇒ X and Y Increment by 1







Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2

Item 1

H	0	1	2
0	1	2	2
1	3	8	2
2	5	3	5

Heights

D			
	0	∞	∞
	∞	∞	∞
	∞	∞	∞

Differences

STEP3

D[0][0] < D[0][1]
0 < ∞ ⇒ isSafe = True

D[0][0] < D[1][0]
0 < ∞ ⇒ True

STEP1 Get TopPair and pop it ⇒ currD = 0
⇒ currN = 1

STEP2 We can Travel to All Nbr

START

END L ← T → R, D

STEP3 Check the isSafe to update the diff when isSafe is True

• D[0][1] = min(maxD, ∞)
= 1

maxD = (0,1)
= 1

R

0 — 1 → 2

1

CURRN

2 → 3

maxD = (0,2)
= 2

D

• D[1][0] = min(maxD, ∞)
= 2

{2, 3}
{1, 2}
~~{0, 1}~~

MinHeap (mini)

```
Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2
```

Item 2

H	0	1	2
0	1	2	2
1	3	8	2
2	5	3	5

Heights

D			
	0	1	∞
	2	∞	∞
	∞	∞	∞

Differences

STEP3

$D[0][1] < D[0][2]$

$1 < \infty \Rightarrow$ isSafe = True

$D[0][1] < D[1][1]$

$1 < \infty \Rightarrow$ True

{6, 8}
{1, 2}
{2, 3}
~~{1, 2}~~

MinHeap (mini)

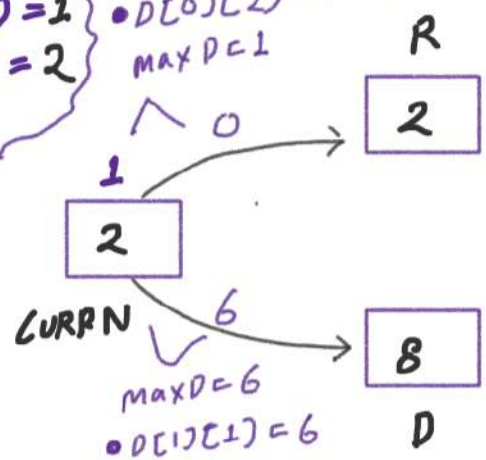
STEP1 Get TopPair and pop it $\Rightarrow$ currD = 1, $\Rightarrow$ currN = 2

• D[0][2] = min(1, ∞) = 1

maxD = 1

STEP2 We can Travel to All Nbr

STEP3 Check the isSafe to update the diff when isSafe is True



maxD = 6

• D[1][1] = 6

Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2

Iteration 3

STEP3 Down ⇒ (1 < ∞) ✓

H	0	1	2
0	1	2	2
1	3	8	2
2	5	3	5

Heights

D			
	0	1	1
	2	6	∞
	∞	∞	∞

Differences

{1,2}
{6,8}
~~{1,2}~~
{2,3}

MinHeap (mini)

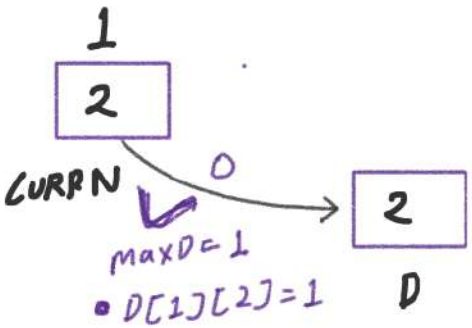
STEP1 Get TopPair and pop it ⇒ currD = 1
⇒ currN = 2

STEP2 We can Travel to All Nbr

START

END

STEP3 Check the isSafe to update the diff when isSafe is True



Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2

Item 4

H	0	1	2
0	1	2	2
1	3	8 ← 6	2 ↓ 3
2	5	3	5

Heights

D			
	0	1	1
	2	6	1
	∞	∞	∞

Differences

STEP3

Down ⇒ 1 < ∞ ✓
Left ⇒ 1 < 6 ✓

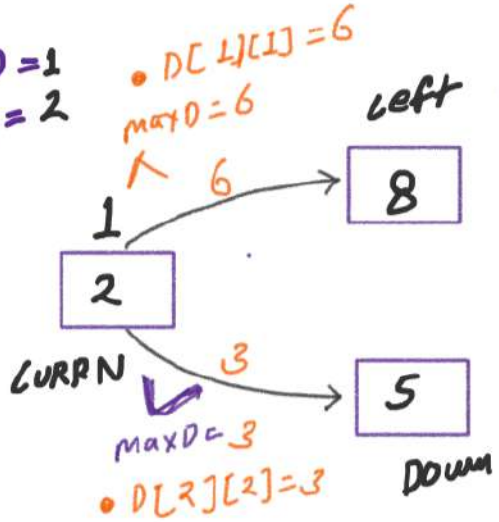
~~{1,2}~~
{6, 8}
{3, 5}
{2, 3}

MinHeap (mini)

STEP1 Get TopPair and pop it ⇒ currD = 1
⇒ currN = 2

STEP2 We can Travel to All Nbr

STEP3 Check the isSafe to update the diff when isSafe is True



Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2

Iteration 5

H	0	1	2
0	1	2	2
1	3	8	2
2	5	3	5

Heights

D	0	1	1
	2	6	1
	∞	∞	3

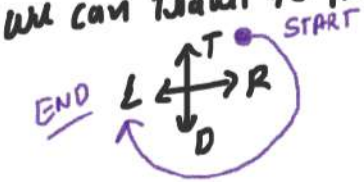
Differences

STEP3

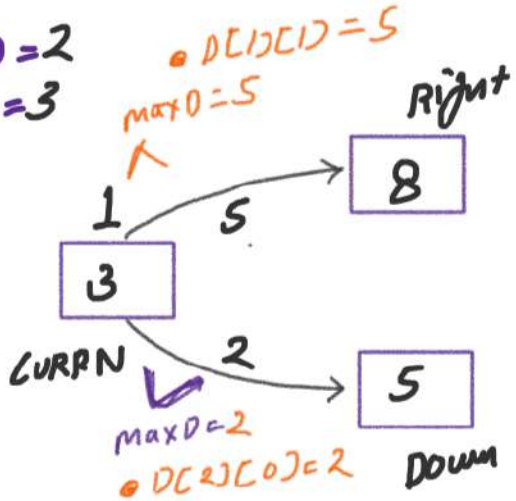
Right ⇒ 2 < 6 ✓
Down ⇒ 2 < ∞ ✓

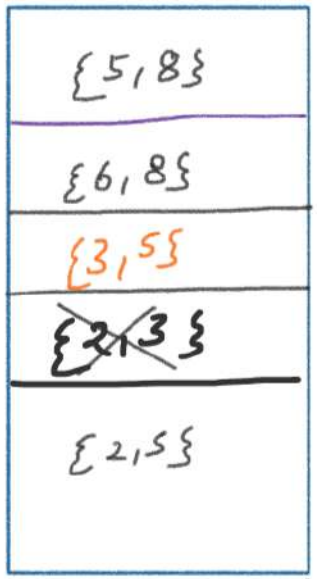
STEP1 Get TOP pair and pop it ⇒ currD = 2
⇒ currN = 3

STEP2 We can Travel to All Nbr



STEP3 Check the isSafe to update the diff when isSafe is True





MinHeap (mini)

Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2

Item6

H	0	1	2
0	1	2	2
1	3	8	2
2	5 → 2	3	5

Heights

STEP3

Right ⇒ 2 < ∞ ✓

D			
	0	1	1
	2	5	1
	2	∞	3

Differences

{5, 8}
{6, 8}
{3, 5}
{2, 3}
~~{2, 5}~~

MinHeap (mini)

STEP1 Get TopPair and pop it ⇒ currD = 2
⇒ currN = 5

STEP2 We can Travel to All Nbr

START

END

T, L, R, D

STEP3 Check the isSafe to update the diff when isSafe is True

D[2][1] = 2

max = 2

Right: 3

2

5

CURRN

Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2

Item 7

H	0	1	2
0	1	2	2
1	3	8	2
2	5	3	5

Heights

D			
	0	1	1
	2	5	1
	2	2	3

Differences

STEP3

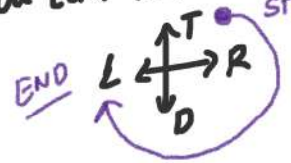
TOP ⇒ (2 < 5) ✓
Right ⇒ (2 < 3) ✓

STEP1 Get TOP pair and pop it ⇒ currD = 2
⇒ currN = 3

STEP2 We can travel to all nbr

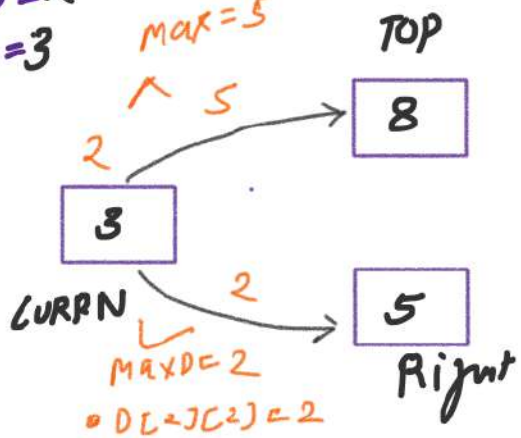
START

END



STEP3 Check the isSafe to update the diff when isSafe is True

• D[1][1] = 5
max = 5



MAXD = 2
• D[2][2] = 2

{5, 8}
{6, 8}
{3, 5}
~~{2, 3}~~
{2, 5}

MinHeap (mini)

Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2

Item 8

H	0	1	2
0	1	2	2
1	3	8	2
2	5	3	5

Heights

STEP3

D		
0	1	1
2	5	1
2	2	2

Differences

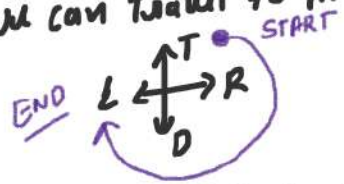
TOP ⇒ (2 < 1) X
Right ⇒ Array Index X
Down ⇒ Array Index X
Left ⇒ (2 < 2) X

{5, 8}
{6, 8}
{3, 5}
~~{2, 5}~~

MinHeap (mini)

STEP1 Get TopPair and pop it ⇒ currD = 2
⇒ currN = 5

STEP2 We can Travel to All Nbrs



STEP3 Check the isSafe to update the diff when isSafe is True

2
5
CURRN

Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2

Iteration 9

STEP3

H	0	1	2
0	1	2	2
1	3	8	2
2	5	3	5

↑5

Heights

D			
	0	1	1
	2	5	1
	2	2	2

Differences

TOP ⇒ (2 < 5) ✓

{5, 8}
{6, 8}
~~{3, 5}~~

MinHeap (mini)

STEP1 Get TOP pair and pop it ⇒ currD = 5
⇒ currN = 3

STEP2 We can travel to all nbr

START

END

T, L, R, D

STEP3 Check the isSafe to update the diff when isSafe is True

TOP

8

5

3

CURRN

5

maxD = 5

• DC[U][J] = 5

Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2

Iteration 10

STEP3

H	0	1	2
0	1	2	2
1	3	8	2
2	5	3	5

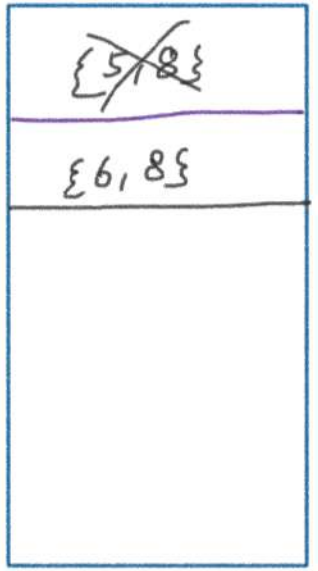
Heights

D

0	1	1
2	5	1
2	2	2

Differences

TOP ⇒ 5 < 1 ✗
Right ⇒ 5 < 1 ✗
Down ⇒ 5 < 2 ✗
Left ⇒ 5 < 2 ✗



MinHeap (mini)

STEP1 Get TopPair and pop it ⇒ currD = 5
⇒ currN = 8

STEP2 We can Travel to All Nbr

STEP3 Check the isSafe to update the diff when isSafe is True

5

8

CURRN

```
Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2
```

Iteration 11

STEP3

H	0	1	2
0	1	2	2
1	3	8	2
2	5	3	5

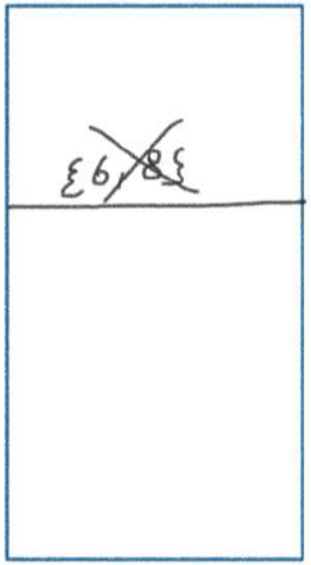
Heights

D

0	1	1
2	5	1
2	2	2

Differences

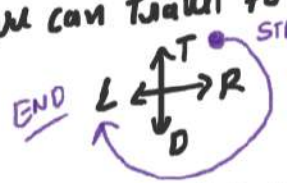
TOP ⇒ 5 < 1 ✗
Right ⇒ 5 < 1 ✗
Down ⇒ 5 < 2 ✗
Left ⇒ 5 < 2 ✗



MinHeap (mini)

STEP1 Get TopPair and pop it ⇒ currD = 6
⇒ currN = 8

STEP2 We can Travel to All Nbr



STEP3 Check the isSafe to update the diff when isSafe is True

5

8

CURRN

Example 1:
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2

H	0	1	2
0	1	2	2
1	3	8	2
2	5	3	5

Heights

D			
	0	1	1
	2	5	1
	2	2	2

Differences

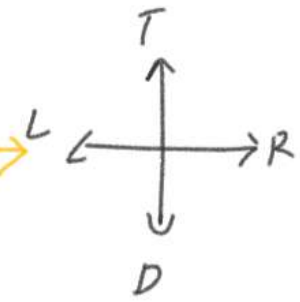
Destination

return [2][2]
⇒ 2
Output

Empty
Now

MinHeap (mini)

```cpp
// 3. Path With Minimum Effort (Leetcode-1631)

class Solution {
public:

    bool isSafe(int newX, int newY, int row, int col,vector<vector<int> >& diff, int currX, int currY ) {
        if(newX >=0 && newY >=0 && newX < row && newY < col && diff[currX][currY] < diff[newX][newY]) {
            return true;
        }
        else {
            return false;
        }
    }

    int minimumEffortPath(vector<vector<int>>& heights) {

        priority_queue< pair<int, pair<int,int> > , vector<pair<int,pair<int,int> > >,
        greater<pair<int,pair<int,int> > > > mini;

        int row = heights.size();
        int col = heights[0].size();
        vector<vector<int> > diff(row, vector<int>(col, INT_MAX));
        int destX = row-1;
        int destY = col-1;

        // Initial state:
        // Set the src difference ==> 0
        diff[0][0] = 0;
        // Insert MinHeap Entry for src ==> {diff of src,{position cell of src}}
        mini.push({0,{0,0}});

        while(!mini.empty()) {
            // Step 1: get the topPair and pop it
            pair<int, pair<int,int>> topPair = mini.top();
            mini.pop();
            int currDiff = topPair.first;
            pair<int,int>  currNodeIndexPair = topPair.second;
            int currX = currNodeIndexPair.first;
            int currY = currNodeIndexPair.second;

            // Step 2: Now we can travel to all nbr i.e. top, right, down, left
            int dx[] = {-1,0,1,0};
            int dy[] = {0,1,0,-1};
            for(int i=0; i<4; i++) {
                int newX = currX + dx[i];
                int newY = currY + dy[i];

                // Step 3: Check the isSafe to update the diff
                if(isSafe(newX, newY,row,col,diff,currX, currY)) {
                    int maxDiff = max(currDiff, abs(heights[currX][currY]-heights[newX][newY]));
                    // Update the minDiff in the diff array
                    diff[newX][newY] = min(diff[newX][newY], maxDiff);
                    // Create new entry for minHeap
                    if(newX != destX || newY != destY) {
                        mini.push({diff[newX][newY], {newX, newY}});
                    }
                }
            }
        }
        // Return destination minimum difference effort
        return diff[destX][destY];
    }
};
```



Github par jakar code check karle
Agar visible nahi ho rahe hai to
phash.