

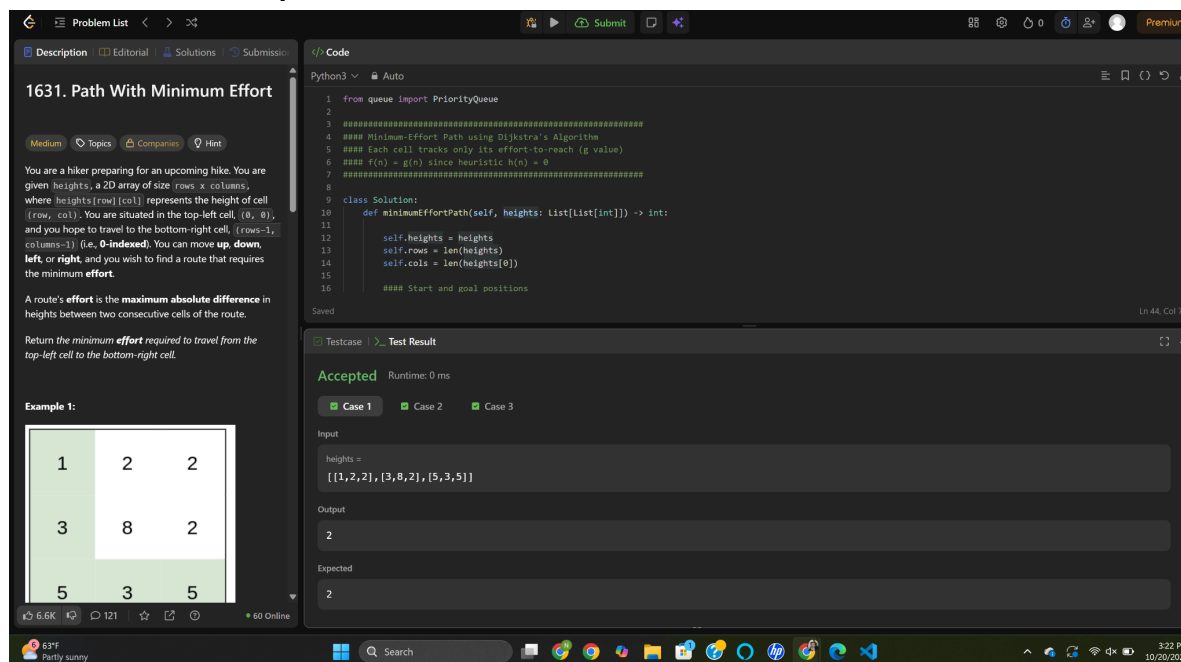
Leetcode Problem:

<https://leetcode.com/problems/path-with-minimum-effort/>

Explanation of Approach:

The goal of this problem was to find a path from the top-left to the bottom-right of a grid where the cost of each path is equal to the absolute difference in height between adjacent cells. The goal was to minimize the maximum effort, meaning that the optimal path would be the one that has the lowest maximum height difference. I decided to use Dijkstra's algorithm to solve this problem because of the fact that each edge cost was different, and the A* algorithm would rely on having a heuristic. Due to the fact that an A* algorithm would rely on a heuristic to guide the search and that there does not seem to be a meaningful heuristic to use, I opted for Dijkstra's algorithm instead. By using Dijkstra's algorithm, correctness and optimality are guaranteed, even though there is no heuristic that narrows down the search space. Since the grid/search space is relatively small, the absence of a heuristic does not significantly impact the performance of the algorithm.

Screenshot of Accepted Solution:



The screenshot shows the LeetCode interface for the problem "1631. Path With Minimum Effort". The problem description is on the left, and the code editor on the right shows a Python solution using Dijkstra's algorithm. The test results show "Accepted" with a runtime of 0 ms.

Problem Description:

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.

Example 1:

1	2	2
3	8	2
5	3	5

Code:

```
1 from queue import PriorityQueue
2
3 #####
4 ##### Minimum-Effort Path using Dijkstra's Algorithm
5 ##### Each cell tracks only its effort-to-reach (g value)
6 ##### f(n) = g(n) since heuristic h(n) = 0
7 #####
8
9 class Solution:
10     def minimumEffortPath(self, heights: List[List[int]]) -> int:
11
12         self.heights = heights
13         self.rows = len(heights)
14         self.cols = len(heights[0])
15
16         ##### Start and goal positions
```

Testcase: Test Result

Accepted Runtime: 0 ms

Case 1 **Case 2** **Case 3**

Input:

```
heights =
[[1,2,2], [3,8,2], [5,3,5]]
```

Output:

```
2
```

Expected:

```
2
```

Code:

```
from queue import PriorityQueue
```

```
#####
#### Minimum-Effort Path using Dijkstra's Algorithm
#### Each cell tracks only its effort-to-reach (g value)
#### f(n) = g(n) since heuristic h(n) = 0
#####
```

class Solution:

def minimumEffortPath(self, heights: List[List[int]]) -> int:

self.heights = heights
self.rows = len(heights)
self.cols = len(heights[0])

Start and goal positions

self.start = (0, 0)
self.goal = (self.rows - 1, self.cols - 1)

Initialize effort grid (g values)

g = [[float('inf')] * self.cols for _ in range(self.rows)]
g[self.start[0]][self.start[1]] = 0

Priority queue for open set

open_set = PriorityQueue()
open_set.put((0, self.start)) # (effort, position)

Movement directions: down, up, right, left

directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

while not open_set.empty():

current_f, (x, y) = open_set.get()

Stop if goal is reached

if (x, y) == self.goal:
return g[x][y]

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < self.rows and 0 <= ny < self.cols:

Step cost = height difference

step_cost = abs(self.heights[nx][ny] - self.heights[x][y])

Path effort = max step seen so far

new_g = max(g[x][y], step_cost)

Update if better path found

if new_g < g[nx][ny]:
g[nx][ny] = new_g
open_set.put((new_g, (nx, ny)))