# Dijkstra, BFS, Heap

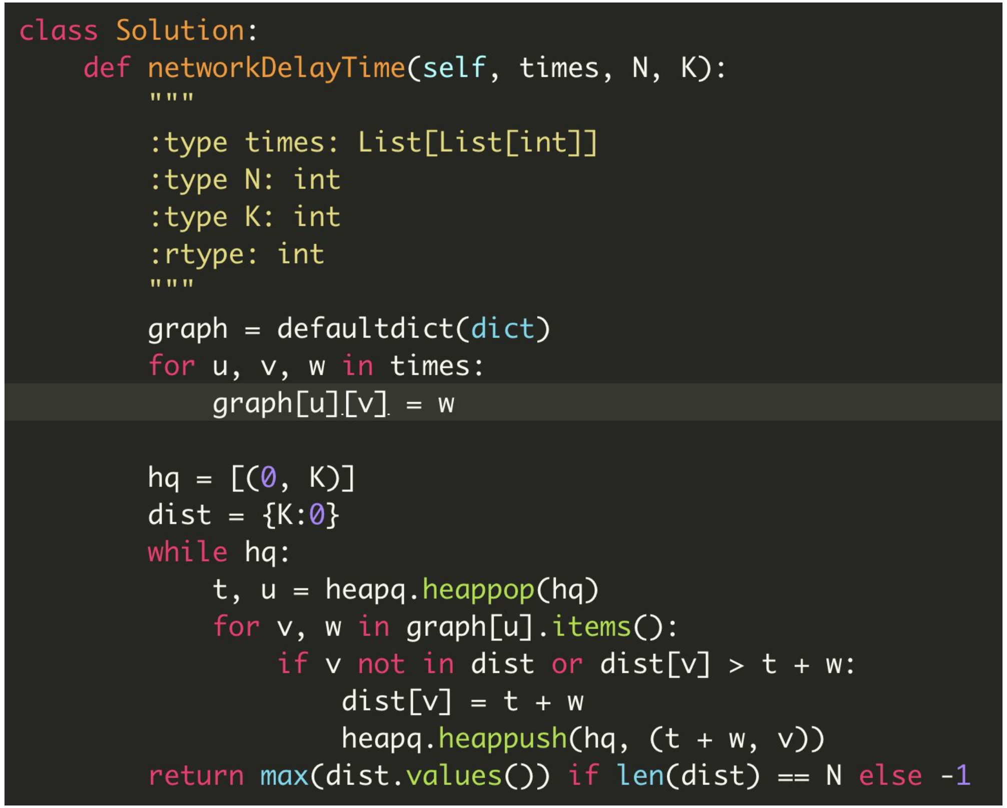
Mar 2020

## 743. Network Delay Time

We will send a signal from a given node k. Return the time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

```
Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
```

```
Output: 2
```

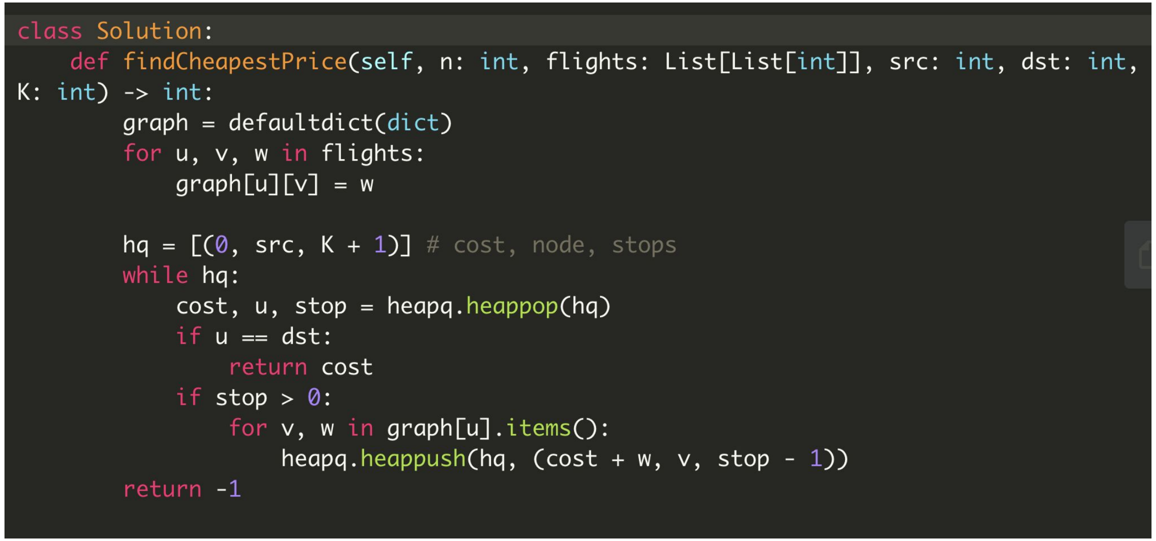


```python
class Solution:
    def networkDelayTime(self, times, N, K):
        """
        :type times: List[List[int]]
        :type N: int
        :type K: int
        :rtype: int
        """
        graph = defaultdict(dict)
        for u, v, w in times:
            graph[u][v] = w

        hq = [(0, K)]
        dist = {K:0}
        while hq:
            t, u = heapq.heappop(hq)
            for v, w in graph[u].items():
                if v not in dist or dist[v] > t + w:
                    dist[v] = t + w
                    heapq.heappush(hq, (t + w, v))
        return max(dist.values()) if len(dist) == N else -1
```

# 787. Cheapest Flights Within K Stops

There are `n` cities connected by `m` flights. Each flight starts from city `u` and arrives at `v` with a price `w`.

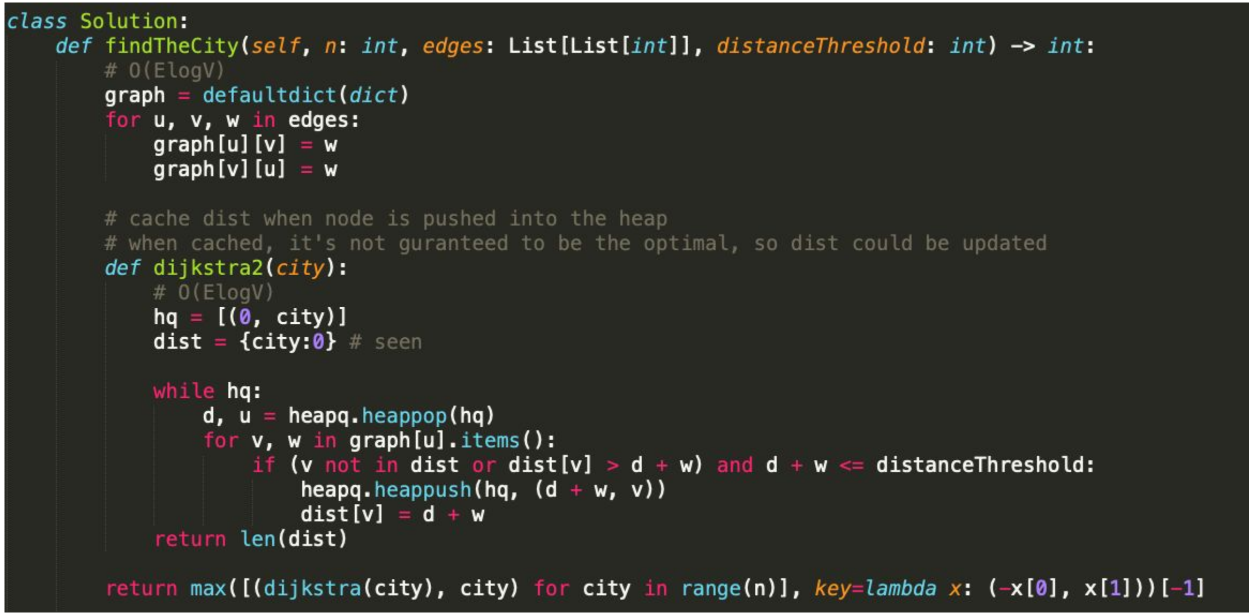
Now given all the cities and flights, together with starting city `src` and the destination `dst`, your task is to find the cheapest price from `src` to `dst` with up to `k` stops. If there is no such route, output `-1`.



```python
class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: int,
K: int) -> int:
        graph = defaultdict(dict)
        for u, v, w in flights:
            graph[u][v] = w

        hq = [(0, src, K + 1)] # cost, node, stops
        while hq:
            cost, u, stop = heapq.heappop(hq)
            if u == dst:
                return cost
            if stop > 0:
                for v, w in graph[u].items():
                    heapq.heappush(hq, (cost + w, v, stop - 1))
        return -1
```

# 1334. Find the City With the Smallest Number of Neighbors at a Threshold Distance

Return the city with the smallest number of cities that are reachable through some path and whose distance is at most distanceThreshold, If there are multiple such cities, return the city with the greatest number.



```python
class Solution:
    def findTheCity(self, n: int, edges: List[List[int]], distanceThreshold: int) -> int:
        # O(ElogV)
        graph = defaultdict(dict)
        for u, v, w in edges:
            graph[u][v] = w
            graph[v][u] = w

        # cache dist when node is pushed into the heap
        # when cached, it's not guranteed to be the optimal, so dist could be updated
        def dijkstra2(city):
            # O(ElogV)
            hq = [(0, city)]
            dist = {city:0} # seen

            while hq:
                d, u = heapq.heappop(hq)
                for v, w in graph[u].items():
                    if (v not in dist or dist[v] > d + w) and d + w <= distanceThreshold:
                        heapq.heappush(hq, (d + w, v))
                        dist[v] = d + w
            return len(dist)

        return max([(dijkstra(city), city) for city in range(n)], key=lambda x: (-x[0], x[1]))[-1]
```

# 1514. Path with Maximum Probability

Given two nodes `start` and `end`, find the path with the maximum probability of success to go from `start` to `end` and return its success probability.

If there is no path from `start` to `end`, return 0. Your answer will be accepted if it differs from the correct answer by at most 1e-5.

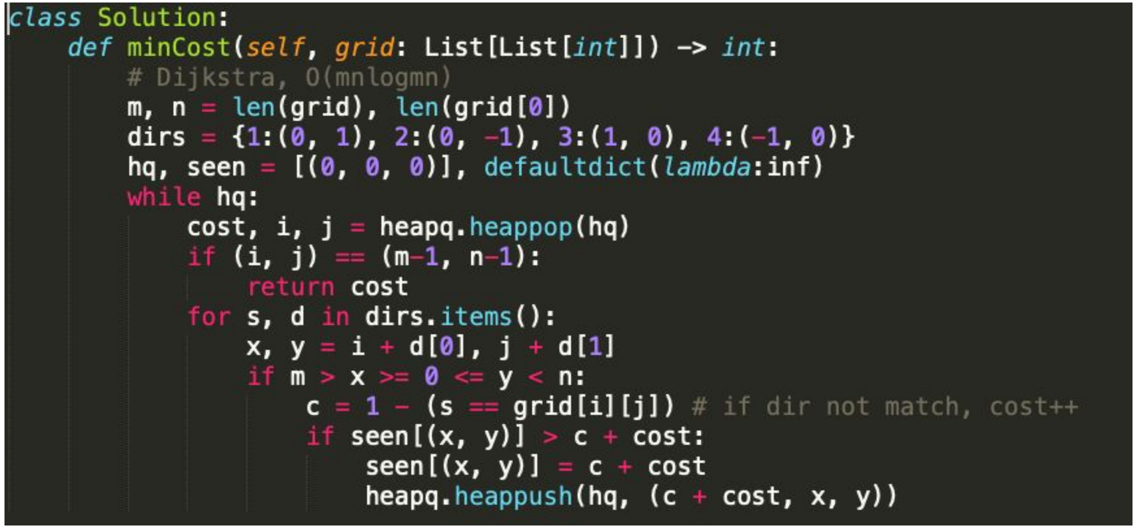
```python
class Solution:
    def maxProbability(self, n: int, edges: List[List[int]], succProb: List[float],
start: int, end: int) -> float:
        graph = defaultdict(dict)
        for i, (u, v) in enumerate(edges):
            graph[u][v] = succProb[i]
            graph[v][u] = succProb[i]

        seen = {start: -1}
        bfs = [(-1, start)]
        while bfs:
            p, node = heapq.heappop(bfs)
            if node == end:
                return -p

            for nxt in graph[node]:
                if nxt not in seen or -seen[nxt] < - p * graph[node][nxt]:
                    heapq.heappush(bfs, (p * graph[node][nxt], nxt))
                    seen[nxt] = p * graph[node][nxt]
        return 0.0
```

# 1368. Minimum Cost to Make at Least One Valid Path in a Grid

You will initially start at the upper left cell `(0,0)`. A valid path in the grid is a path which starts from the upper left cell `(0,0)` and ends at the bottom-right cell `(m - 1, n - 1)` following the signs on the grid. The valid path doesn't have to be the shortest.

You can modify the sign on a cell with `cost = 1`. You can modify the sign on a cell one time only.

Return *the minimum cost* to make the grid have at least one valid path.



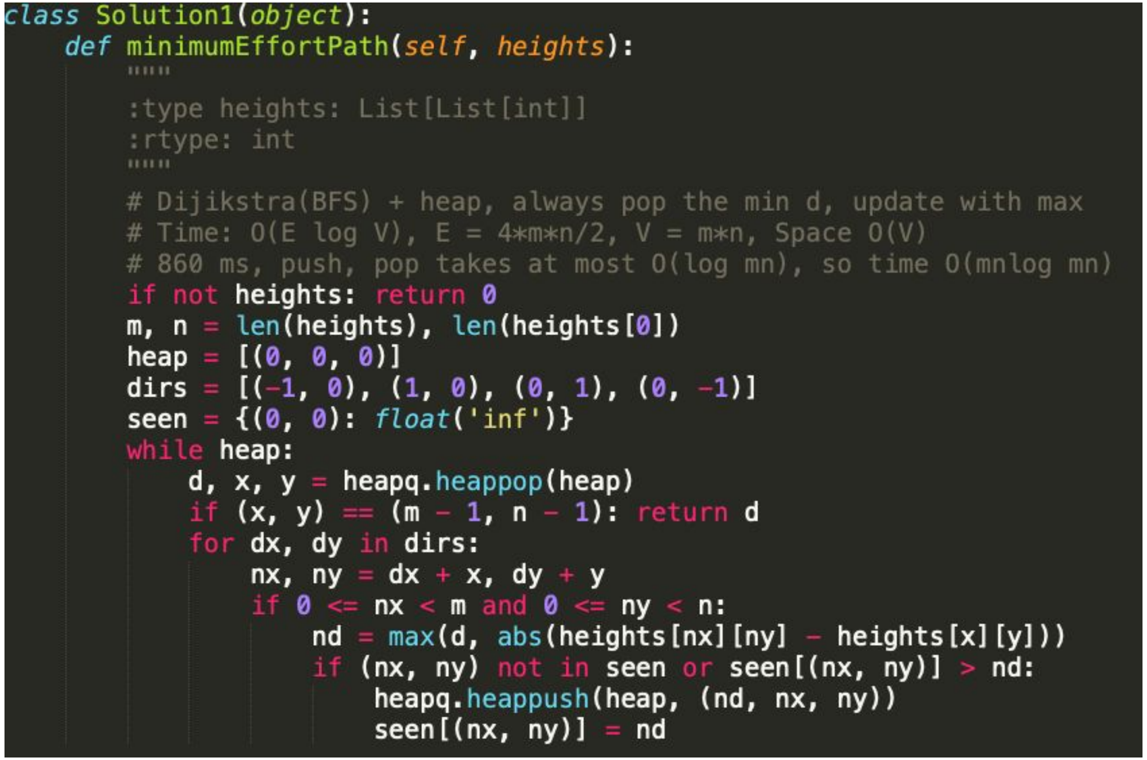
```
class Solution:
    def minCost(self, grid: List[List[int]]) -> int:
        # Dijkstra, O(mnlogmn)
        m, n = len(grid), len(grid[0])
        dirs = {1:(0, 1), 2:(0, -1), 3:(1, 0), 4:(-1, 0)}
        hq, seen = [(0, 0, 0)], defaultdict(lambda:inf)
        while hq:
            cost, i, j = heapq.heappop(hq)
            if (i, j) == (m-1, n-1):
                return cost
            for s, d in dirs.items():
                x, y = i + d[0], j + d[1]
                if m > x >= 0 <= y < n:
                    c = 1 - (s == grid[i][j]) # if dir not match, cost++
                    if seen[(x, y)] > c + cost:
                        seen[(x, y)] = c + cost
                        heapq.heappush(hq, (c + cost, x, y))
```

# 1631. Path With Minimum Effort

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.*



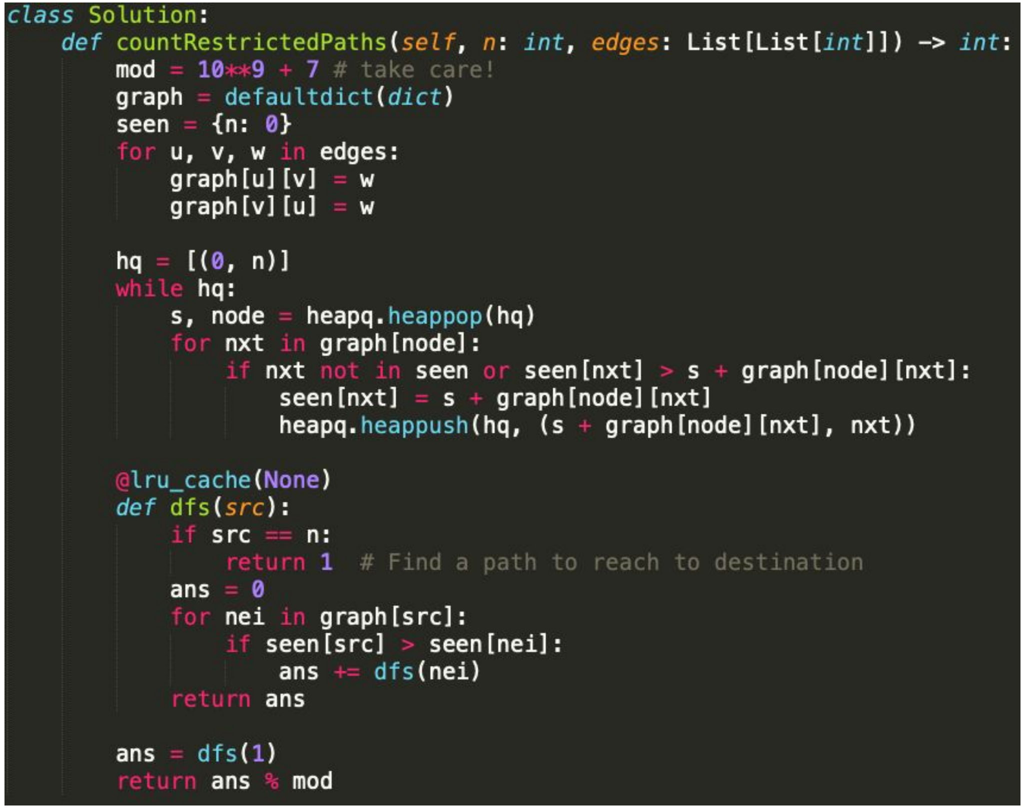
```python
class Solution1(object):
    def minimumEffortPath(self, heights):
        """
        :type heights: List[List[int]]
        :rtype: int
        """
        # Dijikstra(BFS) + heap, always pop the min d, update with max
        # Time: O(E log V), E = 4*m*n/2, V = m*n, Space O(V)
        # 860 ms, push, pop takes at most O(log mn), so time O(mnlog mn)
        if not heights: return 0
        m, n = len(heights), len(heights[0])
        heap = [(0, 0, 0)]
        dirs = [(-1, 0), (1, 0), (0, 1), (0, -1)]
        seen = {(0, 0): float('inf')}
        while heap:
            d, x, y = heapq.heappop(heap)
            if (x, y) == (m - 1, n - 1): return d
            for dx, dy in dirs:
                nx, ny = dx + x, dy + y
                if 0 <= nx < m and 0 <= ny < n:
                    nd = max(d, abs(heights[nx][ny] - heights[x][y]))
                    if (nx, ny) not in seen or seen[(nx, ny)] > nd:
                        heapq.heappush(heap, (nd, nx, ny))
                        seen[(nx, ny)] = nd
```

# 1786. Number of Restricted Paths From First to Last Node

A path from node `start` to node `end` is a sequence of nodes `[z0, z1, z2, ..., zk]` such that `z0 = start` and `zk = end` and there is an edge between `zi` and `zi+1` where `0 <= i <= k-1`.

The distance of a path is the sum of the weights on the edges of the path. Let `distanceToLastNode(x)` denote the shortest distance of a path between node `n` and node `x`. A restricted path is a path that also satisfies that `distanceToLastNode(zi) > distanceToLastNode(zi+1)` where `0 <= i <= k-1`.

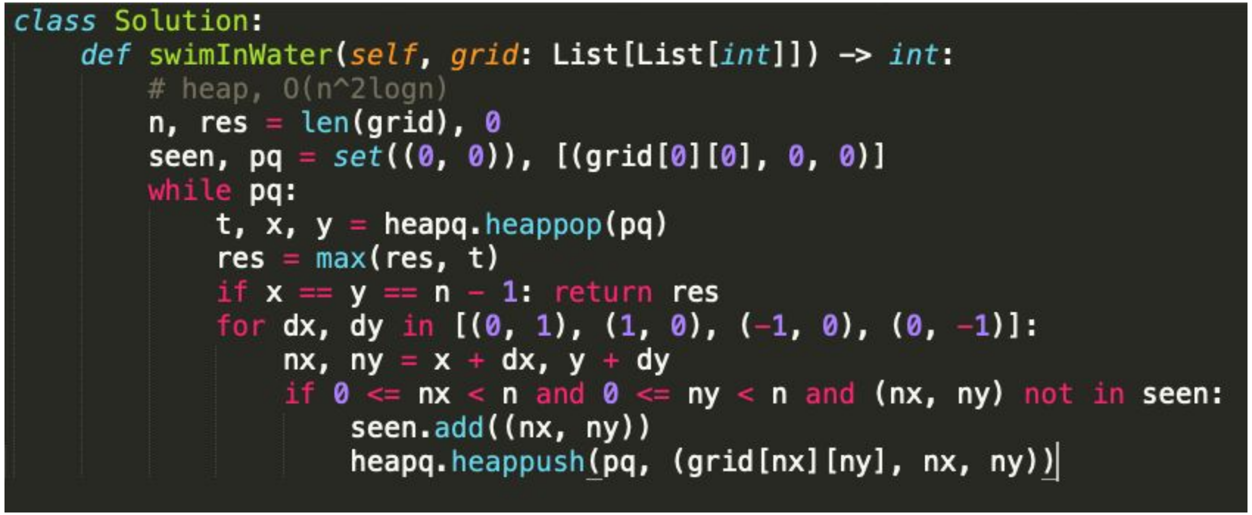
Return *the number of restricted paths from node* `1` *to node* `n`. Since that number may be too large, return it modulo `109 + 7`.



```python
class Solution:
    def countRestrictedPaths(self, n: int, edges: List[List[int]]) -> int:
        mod = 10**9 + 7 # take care!
        graph = defaultdict(dict)
        seen = {n: 0}
        for u, v, w in edges:
            graph[u][v] = w
            graph[v][u] = w

        hq = [(0, n)]
        while hq:
            s, node = heapq.heappop(hq)
            for nxt in graph[node]:
                if nxt not in seen or seen[nxt] > s + graph[node][nxt]:
                    seen[nxt] = s + graph[node][nxt]
                    heapq.heappush(hq, (s + graph[node][nxt], nxt))

        @lru_cache(None)
        def dfs(src):
            if src == n:
                return 1  # Find a path to reach to destination
            ans = 0
            for nei in graph[src]:
                if seen[src] > seen[nei]:
                    ans += dfs(nei)
            return ans

        ans = dfs(1)
        return ans % mod
```

# 778. Swim in Rising Water

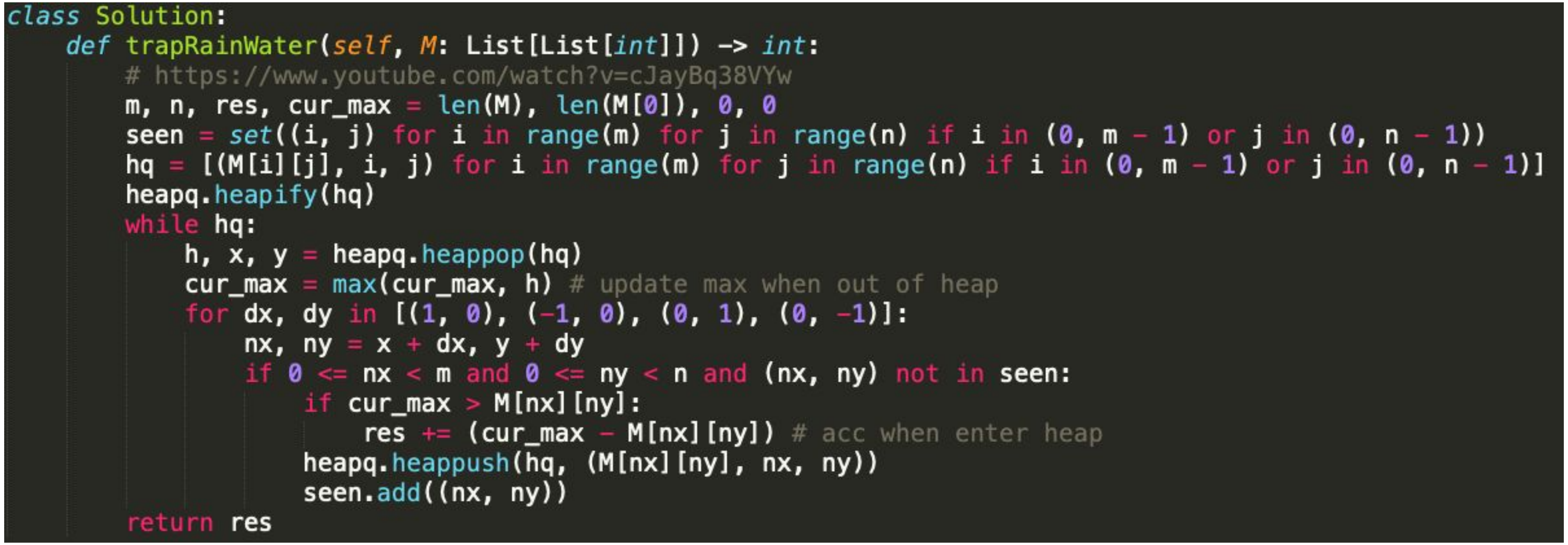
Now rain starts to fall. At time t, the depth of the water everywhere is t. You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most t. You can swim infinite distance in zero time. Of course, you must stay within the boundaries of the grid during your swim.



```python
class Solution:
    def swimInWater(self, grid: List[List[int]]) -> int:
        # heap, O(n^2logn)
        n, res = len(grid), 0
        seen, pq = set((0, 0)), [(grid[0][0], 0, 0)]
        while pq:
            t, x, y = heapq.heappop(pq)
            res = max(res, t)
            if x == y == n - 1: return res
            for dx, dy in [(0, 1), (1, 0), (-1, 0), (0, -1)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < n and 0 <= ny < n and (nx, ny) not in seen:
                    seen.add((nx, ny))
                    heapq.heappush(pq, (grid[nx][ny], nx, ny))
```

# 407. Trapping Rain Water II

https://www.youtube.com/watch?v=cJayBq38VYw



```python
class Solution:
    def trapRainWater(self, M: List[List[int]]) -> int:
        # https://www.youtube.com/watch?v=cJayBq38VYw
        m, n, res, cur_max = len(M), len(M[0]), 0, 0
        seen = set((i, j) for i in range(m) for j in range(n) if i in (0, m - 1) or j in (0, n - 1))
        hq = [(M[i][j], i, j) for i in range(m) for j in range(n) if i in (0, m - 1) or j in (0, n - 1)]
        heapq.heapify(hq)
        while hq:
            h, x, y = heapq.heappop(hq)
            cur_max = max(cur_max, h) # update max when out of heap
            for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < m and 0 <= ny < n and (nx, ny) not in seen:
                    if cur_max > M[nx][ny]:
                        res += (cur_max - M[nx][ny]) # acc when enter heap
                    heapq.heappush(hq, (M[nx][ny], nx, ny))
                    seen.add((nx, ny))
        return res
```