<https://leetcode.com/problems/dungeon-game/>

The demons had captured the princess and imprisoned her in **the bottom-right corner** of a dungeon. The dungeon consists of m x n rooms laid out in a 2D grid. Our valiant knight was initially positioned in **the top-left room** and must fight his way through dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

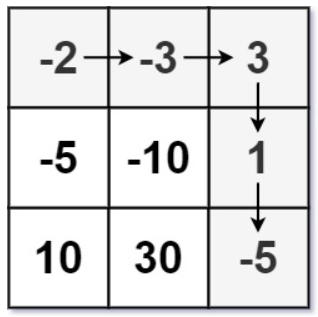
Some of the rooms are guarded by demons (represented by negative integers), so the knight loses health upon entering these rooms; other rooms are either empty (represented as 0) or contain magic orbs that increase the knight's health (represented by positive integers).

To reach the princess as quickly as possible, the knight decides to move only **rightward** or **downward** in each step.

Return *the knight's minimum initial health so that he can rescue the princess*.

**Note** that any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

**Example 1:**



Input: dungeon = [[-2,-3,3],[-5,-10,1],[10,30,-5]]

Output: 7

Explanation: The initial health of the knight must be at least 7 if he follows the optimal path: RIGHT-> RIGHT -> DOWN -> DOWN.

**Example 2:**

Input: dungeon = [[0]]

Output: 1

**Constraints:**

* m == dungeon.length
* n == dungeon[i].length
* 1 <= m, n <= 200
* -1000 <= dungeon[i][j] <= 1000

**Attempt 1: 2023-08-23**

**Solution 1: Native DFS (10 min, TLE 41/45)**

class Solution {

public int calculateMinimumHP(int[][] dungeon) {

return helper(dungeon, 0, 0);

}

private int helper(int[][] dungeon, int i, int j) {

if(i >= dungeon.length || j >= dungeon[0].length) {

return Integer.MAX\_VALUE;

}

// e.g

// If reach the bottom right cell (where princess stay),

// (1) If the value is -5, how many health we need to

// keep knight alive ? We need minimum as 1 - (-5) = 6

// (2) If the value is 5, how many health we need to

// keep knight alive ? We only need 1 not 5 to make sure

// the knight able to reach this cell, because the condition

// on description is "If at any point his health point drops

// to 0 or below, he dies immediately." -> which means to reach

// princess cell the knight has to maintain at least 1 health

if(i == dungeon.length - 1 && j == dungeon[0].length - 1) {

return dungeon[i][j] <= 0 ? 1 - dungeon[i][j] : 1;

}

int go\_down = helper(dungeon, i + 1, j);

int go\_right = helper(dungeon, i, j + 1);

int min\_require = Math.min(go\_down, go\_right) - dungeon[i][j];

// min\_require <= 0 means for visiting dungeon[i][j] we don't

// need more health, but for reaching the cell which adjacent

// to dungeon[i][j] still need maintain health as 1, for

// min\_require > 0 means we actually need this number of health

return min\_require <= 0 ? 1 : min\_require;

}

}

**Solution 2: DFS + Memoization (10 min)**

class Solution {

public int calculateMinimumHP(int[][] dungeon) {

Integer[][] memo = new Integer[dungeon.length][dungeon[0].length];

return helper(dungeon, 0, 0, memo);

}

private int helper(int[][] dungeon, int i, int j, Integer[][] memo) {

if(i >= dungeon.length || j >= dungeon[0].length) {

return Integer.MAX\_VALUE;

}

// e.g

// If reach the bottom right cell (where princess stay),

// (1) If the value is -5, how many health we need to

// keep knight alive ? We need minimum as 1 - (-5) = 6

// (2) If the value is 5, how many health we need to

// keep knight alive ? We only need 1 not 5 to make sure

// the knight able to reach this cell, because the condition

// on description is "If at any point his health point drops

// to 0 or below, he dies immediately." -> which means to reach

// princess cell the knight has to maintain at least 1 health

if(i == dungeon.length - 1 && j == dungeon[0].length - 1) {

return dungeon[i][j] <= 0 ? 1 - dungeon[i][j] : 1;

}

if(memo[i][j] != null) {

return memo[i][j];

}

int go\_down = helper(dungeon, i + 1, j, memo);

int go\_right = helper(dungeon, i, j + 1, memo);

int min\_require = Math.min(go\_down, go\_right) - dungeon[i][j];

// min\_require <= 0 means for visiting dungeon[i][j] we don't

// need more health, but for reaching the cell which adjacent

// to dungeon[i][j] still need maintain health as 1, for

// min\_require > 0 means we actually need this number of health

return memo[i][j] = min\_require <= 0 ? 1 : min\_require;

}

}

**Solution 3: 2D DP (10 min)**

**Style 1: Initial 2D dp array as same size of 2D dungeon array**

**Note: Fully based on recursion "top" as (0, 0) and recursion "bottom" as (dungeon.length - 1, dungeon[0].length - 1) standard which exactly reflect the relation from Solution 1 Native DFS, in dp traversal, since compare to DFS solution, the dp solution will save much more time as no recursion stack push from "top" to "bottom" procedure like DFS, instead it directly process from "bottom" to "top", we should start with "bottom" and trace back to "top", the final solution will come out from dp[0][0]**

class Solution {

public int calculateMinimumHP(int[][] dungeon) {

int m = dungeon.length;

int n = dungeon[0].length;

int[][] dp = new int[m][n];

dp[m - 1][n - 1] = (dungeon[m - 1][n - 1] <= 0 ? 1 - dungeon[m - 1][n - 1] : 1);

// Initialize last column separately since no more column

for(int i = m - 2; i >= 0; i--) {

int min\_require = dp[i + 1][n - 1] - dungeon[i][n - 1];

dp[i][n - 1] = (min\_require <= 0 ? 1 : min\_require);

}

// Initialize last row separately since no more row

for(int j = n - 2; j >= 0; j--) {

int min\_require = dp[m - 1][j + 1] - dungeon[m - 1][j];

dp[m - 1][j] = (min\_require <= 0 ? 1 : min\_require);

}

for(int i = m - 2; i >= 0; i--) {

for(int j = n - 2; j >= 0; j--) {

int min\_require = Math.min(dp[i + 1][j], dp[i][j + 1]) - dungeon[i][j];

dp[i][j] = (min\_require <= 0 ? 1 : min\_require);

}

}

return dp[0][0];

}

}

**Style 2: Initial 2D dp array as one more column and row than original 2D dungeon array**

**Note: Create one more row and one more column which helps uniform the formula make it even able to apply to last column and last row, even it strictly follow the conditions in Native DFS, still need to handle original last column and row specially, the difference between int[][] dp = new int[m][n] style is here the additional last column and row provide a way to do Math.min() as a uniform style as it always have a rightwards, downwards one to compare, which also mapping to base condition 1 in Native DFS**

class Solution {

public int calculateMinimumHP(int[][] dungeon) {

int m = dungeon.length;

int n = dungeon[0].length;

int[][] dp = new int[m + 1][n + 1];

// Setup the cornerstone dp[m - 1][n - 1]

dp[m - 1][n - 1] = (dungeon[m - 1][n - 1] <= 0 ? 1 - dungeon[m - 1][n - 1] : 1);

// Initialize last column

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

dp[i][n] = Integer.MAX\_VALUE;

}

// Initialize last row

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

dp[m][j] = Integer.MAX\_VALUE;

}

// e.g

// Till now the 2D DP array looks like below:

// \_\_\_, \_\_\_, \_\_\_, max

// \_\_\_, \_\_\_, \_\_\_, max

// \_\_\_, \_\_\_, \_6\_, max

// max, max, max, max

for(int i = m - 1; i >= 0; i--) {

for(int j = n - 1; j >= 0; j--) {

if(i == m - 1 && j == n - 1) {

// dp[m - 1][n - 1] is the cornerstone already setup earlier,

// don't overwrite

continue;

}

int min\_require = Math.min(dp[i + 1][j], dp[i][j + 1]) - dungeon[i][j];

dp[i][j] = (min\_require <= 0 ? 1 : min\_require);

}

}

return dp[0][0];

}

}

**Solution 4: 2 rows DP (10 min)**

**Style 1: Initial dpPrev, dp array as same size of 2D dungeon array first dimension size**

class Solution {

// Why we have to initialize last row and penultimate row last column element

// in 2 rows DP solution ?

// Because in 2D DP solution the derive formula to calculate dp[i][j] depends

// on its next row and same column element as dp[i + 1][j] and its next column

// and same row element as dp[i][j + 1]:

// int min\_require = Math.min(dp[i + 1][j], dp[i][j + 1]) - dungeon[i][j]

// To use 2 rows DP, to rollout full current row in each iteration, we have

// to prepare next row and current row last column first, so next row define

// in 2 rows DP as 'dpPrev', and current row define as 'dp'

public int calculateMinimumHP(int[][] dungeon) {

int m = dungeon.length;

int n = dungeon[0].length;

// dpPrev -> dp[i + 1] row, equal to 2D dp[i + 1][...]

// dp -> dp[i] row, equal to 2D dp[i][...]

int[] dpPrev = new int[n];

int[] dp = new int[n];

// Initialize bottom right corner cell where princess stay as cornerstone

// to initialize last row and last column elements

dpPrev[n - 1] = (dungeon[m - 1][n - 1] <= 0 ? 1 - dungeon[m - 1][n - 1] : 1);

// Initialize last row and assign to 'dpPrev'

// e.g for input: dungeon

// {{-2,-3,3},

// {-5,-10,1},

// {10,30,-5}}

// we create last row for dpPrev = {1,1,6}

for(int j = n - 2; j >= 0; j--) {

int min\_require = dpPrev[j + 1] - dungeon[m - 1][j];

dpPrev[j] = (min\_require <= 0 ? 1 : min\_require);

}

// Since last row already initialized as 'dpPrev'

// now we only need to iterate from penultimate row

for(int i = m - 2; i >= 0; i--) {

// Initialize last column element for 'dp' in each round

int min\_require = dpPrev[n - 1] - dungeon[i][n - 1];

dp[n - 1] = (min\_require <= 0 ? 1 : min\_require);

// Now we have next full row initialized as 'dpPrev' and last element

// on current row initialized in 'dp', we can build up full current row

// as 'dp' based on same formula as 2D DP solution, just replace

// dp[i + 1][j] to dpPrev[j] and dp[i][j + 1] to dp[j + 1]

// int min\_require = Math.min(dp[i + 1][j], dp[i][j + 1]) - dungeon[i][j]

// --> int min\_require\_1 = Math.min(dpPrev[j], dp[j + 1]) - dungeon[i][j];

for(int j = n - 2; j >= 0; j--) {

int min\_require\_1 = Math.min(dpPrev[j], dp[j + 1]) - dungeon[i][j];

dp[j] = (min\_require\_1 <= 0 ? 1 : min\_require\_1);

}

// Update 'dpPrev' to 'dp' prepare for next iteration

dpPrev = dp.clone();

}

return dpPrev[0];

}

}

**Style 2: Initial dpPrev, dp array as one more column of 2D dungeon array first dimension size**

class Solution {

public int calculateMinimumHP(int[][] dungeon) {

int m = dungeon.length;

int n = dungeon[0].length;

// dpPrev -> dp[i + 1] row, equal to 2D dp[i + 1][...]

// dp -> dp[i] row, equal to 2D dp[i][...]

int[] dpPrev = new int[n + 1];

int[] dp = new int[n + 1];

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

dpPrev[i] = Integer.MAX\_VALUE;

}

// Initialize bottom right corner cell where princess stay as cornerstone

// to initialize last row and last column elements, also need to initialize

// last column of dp

// e.g for input: dungeon

// {{-2,-3,3},

// {-5,-10,1},

// {10,30,-5}}

// we create penultimate row for dp = {\_\_\_,\_\_\_,6,max}

dp[n] = Integer.MAX\_VALUE;

dp[n - 1] = (dungeon[m - 1][n - 1] <= 0 ? 1 - dungeon[m - 1][n - 1] : 1);

// Now we have dp and dpPrev as below and want to derive full dp (but don't overwrite 6)

// dp = {\_\_\_,\_\_\_,\_6\_,max} -> penultimate row

// dpPrev = {max,max,max,max} -> last row

for(int i = m - 1; i >= 0; i--) {

for(int j = n - 1; j >= 0; j--) {

// dp[n - 1] is the cornerstone already setup earlier, don't overwrite

if(i == m - 1 && j == n - 1) {

continue;

}

int min\_require = Math.min(dpPrev[j], dp[j + 1]) - dungeon[i][j];

dp[j] = (min\_require <= 0 ? 1 : min\_require);

}

dpPrev = dp.clone();

}

return dpPrev[0];

}

}

**Solution 5: 1 row DP (360 min)**

**Style 1: Initial dpPrev array as same size of 2D dungeon array first dimension size**

class Solution {

public int calculateMinimumHP(int[][] dungeon) {

int m = dungeon.length;

int n = dungeon[0].length;

// Only have one array 'dpPrev' now

int[] dpPrev = new int[n];

// Initialize bottom right corner cell where princess stay as cornerstone

// to initialize last row and last column elements

dpPrev[n - 1] = (dungeon[m - 1][n - 1] <= 0 ? 1 - dungeon[m - 1][n - 1] : 1);

// Initialize last row and assign to 'dpPrev'

// e.g for input: dungeon

// {{-2,-3,3},

// {-5,-10,1},

// {10,30,-5}}

// we create last row for dpPrev = {1,1,6}

for(int j = n - 2; j >= 0; j--) {

int min\_require = dpPrev[j + 1] - dungeon[m - 1][j];

dpPrev[j] = (min\_require <= 0 ? 1 : min\_require);

}

// Since last row already initialized as 'dpPrev'

// now we only need to iterate from penultimate row

for(int i = m - 2; i >= 0; i--) {

// Update last column element for 'dpPrev' in each round

// e.g for input: dungeon

// {{-2,-3,3}, {{7,5,2},

// {-5,-10,1}, => dpPrev final result = {6,11,5},

// {10,30,-5}} {1,1,6}}

// Previously we initialize dpPrev = {1,1,6}

// now in for loop 1st iteration we update last column

// element from 6 to 5 -> dpPrev = {1,1,5}

// the last element as 5 now which successfully update

// from last row's value to penultimate row's value

int min\_require = dpPrev[n - 1] - dungeon[i][n - 1];

dpPrev[n - 1] = (min\_require <= 0 ? 1 : min\_require);

for(int j = n - 2; j >= 0; j--) {

// Now based on dpPrev = {1,1,5} we can continue update the whole 'dpPrev'

// array value from last row's value to penultimate row's value

// {1,1[a],5[b]}, here '1' at position [a] represents the last row's value,

// in another word its the old value in 'dpPrev' pending on update, which

// represents by 'dpPrev[j]' in below formula, '5' at position [b]

// represents the penultimate row's value, in another word its already

// the new value in 'dpPrev' updated earlier(6 -> 5), which represents by

// 'dpPrev[j + 1]', graphically it can be represented as

// 1 1->? -- 5[b] -> penultimate row as updated earlier on last column

// |

// 1 1[a] 6 -> last row as initial dpPrev

// The formula below is used to calculate the question mark in (1->?)

// and its downwards old value 'dpPrev[j] = 1' at 'a' and rightwards new

// value 'dpPrev[j + 1] = 5' at 'b' contribute on it

// Finally result '?' is 11 as below:

// 1 1-> Math.min(1,5)-(-10)=11 -- 5[b]

// 1 1[a] 6 -> last row as initial dpPrev

// And if we rewind final target dpPrev, we have finish all (\*) elements

// calculation till now with only 1 row dpPrev array

// {{7,5,2},

// dpPrev final result = {6,11\*,5\*},

// {1\*,1\*,6\*}}

int min\_require\_1 = Math.min(dpPrev[j], dpPrev[j + 1]) - dungeon[i][j];

dpPrev[j] = (min\_require\_1 <= 0 ? 1 : min\_require\_1);

}

}

return dpPrev[0];

}

}

**Style 2: Initial dpPrev array as one more column of 2D dungeon array first dimension size**

class Solution {

public int calculateMinimumHP(int[][] dungeon) {

int m = dungeon.length;

int n = dungeon[0].length;

// dpPrev -> dp[i + 1] row, equal to 2D dp[i + 1][...]

int[] dpPrev = new int[n + 1];

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

dpPrev[i] = Integer.MAX\_VALUE;

}

// Initialize bottom right corner cell where princess stay as cornerstone

// to initialize last row and last column elements

// e.g for input: dungeon

// {{-2,-3,3},

// {-5,-10,1},

// {10,30,-5}}

// we create penultimate row for dpPrev = {max,max,6,max}

dpPrev[n - 1] = (dungeon[m - 1][n - 1] <= 0 ? 1 - dungeon[m - 1][n - 1] : 1);

// Now we have dpPrev as below (don't overwrite 6)

// dpPrev = {max,max,\_6\_,max} -> last row

for(int i = m - 1; i >= 0; i--) {

for(int j = n - 1; j >= 0; j--) {

// dpPrev[n - 1] is the cornerstone already setup earlier, don't overwrite

if(i == m - 1 && j == n - 1) {

continue;

}

int min\_require = Math.min(dpPrev[j], dpPrev[j + 1]) - dungeon[i][j];

dpPrev[j] = (min\_require <= 0 ? 1 : min\_require);

}

}

return dpPrev[0];

}

}

**Refer to**

<https://leetcode.com/problems/dungeon-game/solutions/745340/post-dedicated-to-beginners-of-dp-or-have-no-clue-how-to-start/>

**before starting with DP one must master the art of recursion** most of the post shows you the optimal solution, briefly explains how it works, but doesn't tell you how to arrive at that solution. it goes as "give man a fish, he can eat it whole day, tell then how to catch one, he can have it whole life". here is how you should approach these recursive / DP problems. hopefully you will learn something.

**step1 : understand the problem** here we have been given a matrix we need to start from top and find a way to get to bottom right, we need the min cost that is required to do this

**step 2: logic building and deriving recurrence relation**

**1st sub step : asking questions**

1. At any point if our health gets zero of below we dies, anthem so : we need 1 + (-mat[i][j]) for our health to be one.
2. What if we get some health if we arrive at some cell ? my guess is we still need 1 health in first case to arrive at that cell - cases like these need to be figure out by yourself.
3. at any cell what health do we need ? - since we can go only down and right therefore min health required will be minimum health required if we go right or down, ( further explained in arriving at recurrence relation heading )

for brief answers/explanation for above point 1 and 2 , assume a 1D matrix this is what is meant by 1st and 2nd point.

[[-10]] : ans = 1 + (-(-10)) = 11 (explanation to first point mentioned)

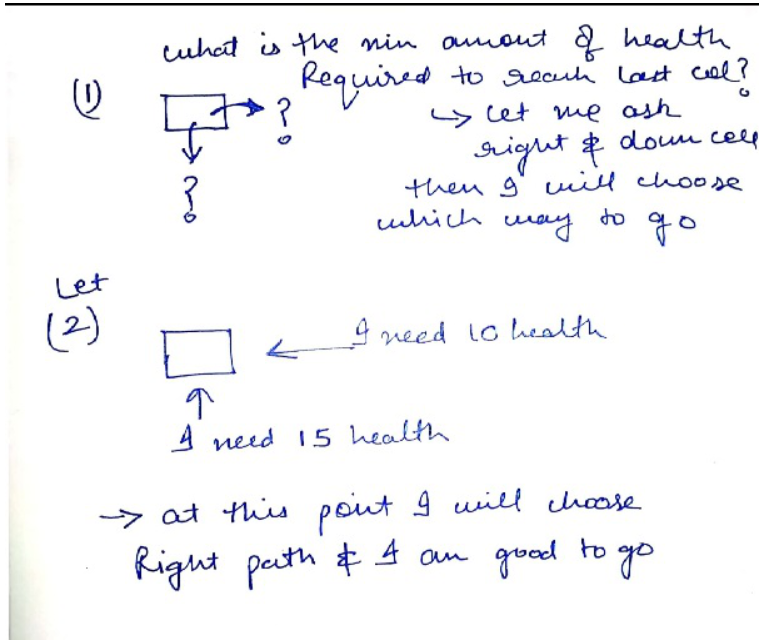
[[10]] : ans = 1 as we still need 1 health at first place to get there (explanation to second point mentioned)

[[-2,-3,3,-5,-10]] : ans = 1 + (-(-17)) = 18 same as 1st case

[[2,3,3,5,10]] : ans = 1 same as 2nd test case, explanation to second point mentioned to asking question

**2nd sub step : Arriving at recurrence relation**

recurrence relation is pretty straight forward at any cell ,if we are at any particular cell we must ask should we go right or down ? if we know the answer for min health req if we go right vs we go down, then we can easily choose



**step 3: code recursive solution - (TLE)**

int getVal(vector<vector<int>> &mat, int i=0, int j=0)

{

int n = mat.size();

int m = mat[0].size();

// Base case : we have crossed the matrix, ie. out of bound

/// if current row crosses then my row is below the princess or

/// if current column crosses then my column is ahead the column of princess

/// and beacause we can go only down and right so we wont be able reach princess

if(i == n || j == m) return 1e9;

// Base Case : we have reached our destination ie. last cell

/// we reached princess , cheers return this cost;

if(i == n-1 and j == m-1)

return (mat[i][j] <= 0) ? -mat[i][j] + 1 : 1;

/// now we must try all possible paths , we ask our right and and down cell

int IfWeGoRight = getVal(mat , i , j+1);

int IfWeGoDown = getVal(mat , i+1 , j);

/// min of either values and then cost of this cell

int minHealthRequired = min(IfWeGoRight , IfWeGoDown) -mat[i][j];

/// point 2 as explained

return ( minHealthRequired <= 0 ) ? 1 : minHealthRequired;

}

int calculateMinimumHP(vector<vector<int>>& dungeon) {

return getVal(dungeon);

}

Now you know how to solve this recursively, lets now observe time complexity ... ... yep its exponential ! . but luckily we are doing same task over and over again. ( for example we are asking 6 times the last cell its cost in 3X3 matrix), we can overcome this task by storing the values of cost at each cell ( aka memoization ).

**step4 : memoization-top-down** lets maintain a DP matrix which will store the calculated values for its cell. next time if we arrive at this cell we will return this calculated value to save recursive calls.

int getVal(vector<vector<int>> &mat, vector<vector<int>> &dp , int i=0, int j=0)

{

int n = mat.size();

int m = mat[0].size();

if(i == n || j == m) return 1e9;

if(i == n-1 and j == m-1)

return (mat[i][j] <= 0) ? -mat[i][j] + 1 : 1;

/// if we know the answer for this cell then no need to recalculate those, simply return those values

if( dp[i][j] != 1e9)

return dp[i][j];

int IfWeGoRight = getVal(mat , dp , i , j+1);

int IfWeGoDown = getVal(mat , dp , i+1 , j);

int minHealthRequired = min(IfWeGoRight , IfWeGoDown) -mat[i][j];

/// before returning the values, we must store the answers for this cell which we hacve calculated

/// in next recurssive call this value will be used to save some computation, aka repetative work which we are doing.

dp[i][j] = ( minHealthRequired <= 0 ) ? 1 : minHealthRequired;

return dp[i][j];

}

int calculateMinimumHP(vector<vector<int>>& dungeon) {

int n = dungeon.size();

int m = dungeon[0].size();

vector<vector<int>> dp(n , vector<int>(m , 1e9));

return getVal(dungeon, dp);

}

so now you know the recursive solution , you have also tried memoization, so try coming up with bottomUp solution yourself, HINT : as you might have observed final destination is the last cell, so why dont we start with the bottom cell itself. then work all the way up to first cell.

**step 5 : You know the base cases , you know the sub problems so try coming up with bottom up solution yourself ( hint in above paragraph )**

int calculateMinimumHP(vector<vector<int> > &dungeon) {

int n = dungeon.size();

int m = dungeon[0].size();

vector<vector<int> > dp(n + 1, vector<int>(m + 1, 1e9));

dp[n][m - 1] = 1;

dp[n - 1][m] = 1;

for (int i = n - 1; i >= 0; i--)

{

for (int j = m - 1; j >= 0; j--)

{

int need = min(dp[i + 1][j], dp[i][j + 1]) - dungeon[i][j];

// store this value

dp[i][j] = need <= 0 ? 1 : need;

}

}

return dp[0][0];

}

**Refer to**

<https://leetcode.wang/leetcode-174-Dungeon-Game.html>

题目描述，任务是从左上角（K）走到右下角（P），初始的时候有一个生命值 HP。只能向右和向下走，格子上边的数值代表增加 HP 和减少 HP，一旦变为 0，就立刻结束，问初始的 HP 最小可以取多少，才能从 K 走到 P。注意如果 P 点是负值，也要保证到达 P 点后将 P 点的值减去后， HP 的值依旧大于 0。

# **解法一 回溯法**

最直接暴力的方法就是做搜索了，在每个位置无非就是向右向下两种可能，然后去尝试所有的解，然后找到最小的即可，也就是做一个 DFS 或者说是回溯法。

//全局变量去保存最小值

int minHealth = Integer.MAX\_VALUE;

public int calculateMinimumHP(int[][] dungeon) {

//calculateMinimumHPHelper 四个参数

//int x, int y, int health, int addHealth, int[][] dungeon

//x, y 代表要准备到的位置，x 代表是哪一列，y 代表是哪一行

//health 代表当前的生命值

//addHealth 代表当前已经增加的生命值

//初始的时候给加 1 点血，addHealth 和 health 都是 1

calculateMinimumHPHelper(0, 0, 1, 1, dungeon);

return minHealth;

}

private void calculateMinimumHPHelper(int x, int y, int health, int addHealth, int[][] dungeon) {

//加上当前位置的奖励或惩罚

health = health + dungeon[y][x];

//此时是否需要加血，加血的话就将 health 加到 1

if (health <= 0) {

addHealth = addHealth + Math.abs(health) + 1;

}

//是否到了终点

if (x == dungeon[0].length - 1 && y == dungeon.length - 1) {

minHealth = Math.min(addHealth, minHealth);

return;

}

//是否加过血

if (health <= 0) {

//加过血的话，health 就变为 1

if (x < dungeon[0].length - 1) {

calculateMinimumHPHelper(x + 1, y, 1, addHealth, dungeon);

}

if (y < dungeon.length - 1) {

calculateMinimumHPHelper(x, y + 1, 1, addHealth, dungeon);

}

} else {

//没加过血的话，health 就是当前的 health

if (x < dungeon[0].length - 1) {

calculateMinimumHPHelper(x + 1, y, health, addHealth, dungeon);

}

if (y < dungeon.length - 1) {

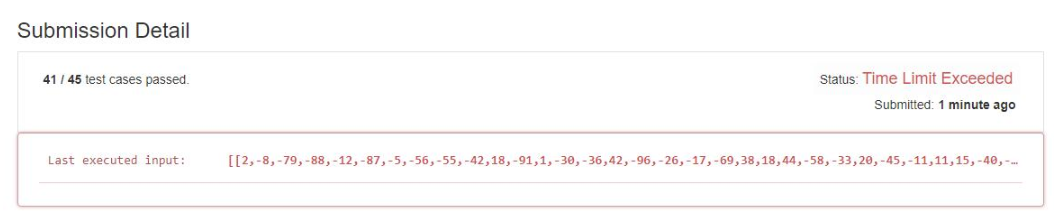
calculateMinimumHPHelper(x, y + 1, health, addHealth, dungeon);

}

}

}

然后结果是意料之中的，会超时。



然后我们就需要剪枝，将一些情况提前结束掉，最容易想到的就是，如果当前加的血已经超过了全局最小值，那就可以直接结束，不用进后边的递归。

if (addHealth > minHealth) {

return;

}

然后发现对于给定的 test case 并没有什么影响。

之所以超时，就是因为我们会经过很多重复的位置，比如

0 1 2

3 4 5

6 7 8

如果按 DFS，第一条路径就是 0 -> 1 -> 2 -> 5 -> 8

然后通过回溯，第二次判断的路径就会是 0 -> 1 -> 4 -> 5 -> ...

我们会发现它又会来到 5 这个位置

其他的也类似，如果表格很大的话，不停的回溯，一些位置会经过很多次

接下来，就会想到用 map 去缓冲我们过程中求出的解，key 话当然是 x 和 y 了，value 呢？存当前的 health 和 addhealth？那第二次来到这个位置的时候，我们并不能做什么，比如举个例子。

第一次来到 (3,5) 的时候，health 是 5，addhealth 是 6。

第二次来到 (3,5) 的时候，health 是 4，addhealth 是 7，我们什么也做不了，我们并不知道未来它会走什么路。

因为走的路是由 health 和 addhealth 共同决定的，此时来到相同的位置，由于 health 和 addhealth 都不一样，所以未来的路也很有可能变化，所以我们并不能通过缓冲结果来剪枝。

我们最多能判断当 x、y、health 和 addhealth 全部相同的时候提前结束，但这种情况也很少，所以并不能有效的加快搜索速度。

这条路看起来到死路了，我们换个思路，去用动态规划。

动态规划的关键就是去定义我们的状态了，这里直接将要解决的问题定义为我们的状态。

用 dp[i][j] 存储从起点 (0, 0) 到达 (i, j) 时候所需要的最小初始生命值。

到达 (i,j) 有两个点，(i-1, j) 和 (i, j-1)。

接下来就需要去推导状态转移方程了。

\* \* 8 \*

\* 7 ! ?

? ? ? ?

假如我们要求上图中 ! 位置的 dp，假设之前的 dp 已经都求出来了。

那么 dp 是等于感叹号上边的 dp 还是等于它左边的 dp 呢？选较小的吗？

但如果 8 对应的当时的 health 是 100，而 7 对应的是 5，此时更好的选择应该是 8。

那就选 health 大的呗，那 dp 不管了吗？极端的例子，假如此时的位置已经是终点了，很明显我们应该选择从左边过来，也就是 7 的位置过来，之前的 health 并不重要了。

所以推到这里会发现，因为我们有两个不确定的变量，一个是 dp ，也就是从起点 (0, 0) 到达 (i, j) 时候所需要的最小初始生命值，还有一个就是当时剩下的生命值。

当更新 dp 的时候我们并不知道它应该是从上边下来，还是从左边过来有利于到达终点的时候所需的初始生命值最小。

换句话讲，依赖过去的状态，并不能指导我们当前的选择，因为还需要未来的信息。

所以到这里，我再次走到了死胡同，就去看 Discuss 了，这里分享下别人的做法。

# **解法二 递归**

看到 [这里](https://leetcode.com/problems/dungeon-game/discuss/52790/My-AC-Java-Version-Suggestions-are-welcome) 评论区的一个解法。

所需要做的就是将上边动态规划的思路逆转一下。

↓

→ \*

之前我们考虑的是当前这个位置，它应该是从上边下来还是左边过来会更好些，然后发现并不能确定。

现在的话，看下边的图。

\* → x

↓

y

我们现在考虑从当前位置，应该是向右走还是向下走，这样我们是可以确定的。

如果我们知道右边的位置到终点的需要的最小生命值是 x，下边位置到终点需要的最小生命值是 y。

很明显我们应该选择所需生命值较小的方向。

如果 x < y，我们就向右走。

如果 x > y，我们就向下走。

知道方向以后，当前位置到终点的最小生命值 need 就等于 x 和 y 中较小的值减去当前位置上边的值。

如果算出来 need 大于 0，那就说明我们需要 need 的生命值到达终点。

如果算出来 need 小于等于 0，那就说明当前位置增加的生命值很大，所以当前位置我们只需要给一个最小值 1，就足以走到终点。

举个具体的例子就明白了。

如果右边的位置到终点的需要的最小生命值是 5，下边位置到终点需要的最小生命值是 8。

所以我们选择向右走。

如果当前位置的值是 2，然后 need = 5 - 2 = 3，所以当前位置的初始值应该是 3。

如果当前位置的值是 -3，然后 need = 5 - (-3) = 8，所以当前位置的初始值应该是 8。

如果当前位置的值是 10，说明增加的生命值很多，need = 5 - 10 = -5，此时我们只需要将当前位置的生命值初始为 1 即可。

然后每个位置都这样考虑，递归也就出来了。

递归出口也很好考虑， 那就是最后求终点到终点需要的最小生命值。

如果终点位置的值是正的，那么所需要的最小生命值就是 1。

如果终点位置的值是负的，那么所需要的最小生命值就是负值的绝对值加 1。

public int calculateMinimumHP(int[][] dungeon) {

return calculateMinimumHPHelper(0, 0, dungeon);

}

private int calculateMinimumHPHelper(int i, int j, int[][] dungeon) {

//是否到达终点

if (i == dungeon.length - 1 && j == dungeon[0].length - 1) {

if (dungeon[i][j] > 0) {

return 1;

} else {

return -dungeon[i][j] + 1;

}

}

//右边位置到达终点所需要的最小值，如果已经在右边界，不能往右走了，赋值为最大值

int right = j < dungeon[0].length - 1 ? calculateMinimumHPHelper(i, j + 1, dungeon) : Integer.MAX\_VALUE;

//下边位置到达终点需要的最小值，如果已经在下边界，不能往下走了，赋值为最大值

int down = i < dungeon.length - 1 ? calculateMinimumHPHelper(i + 1, j, dungeon) : Integer.MAX\_VALUE;

//当前位置到终点还需要的生命值

int need = right < down ? right - dungeon[i][j] : down - dungeon[i][j];

if (need <= 0) {

return 1;

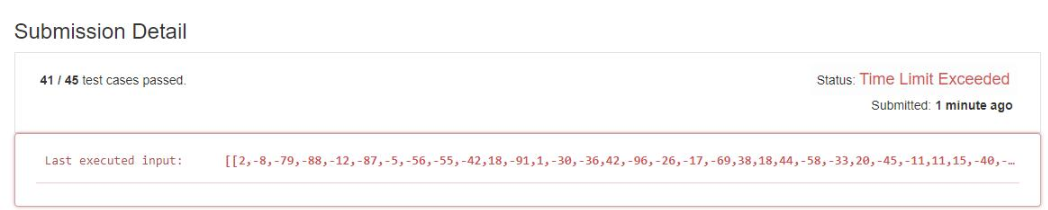
} else {

return need;

}

}

当然还是意料之中的超时了。



不过不要慌，还是之前的思想，我们利用 map 去缓冲中间过程的值，也就是 memoization 技术。

这个 map 的 key 和 value 就显而易见了，key 是坐标 i,j，value 的话就存当最后求出来的当前位置到终点所需的最小生命值，也就是 return 前同时存进 map 中。

public int calculateMinimumHP(int[][] dungeon) {

return calculateMinimumHPHelper(0, 0, dungeon, new HashMap<String, Integer>());

}

private int calculateMinimumHPHelper(int i, int j, int[][] dungeon, HashMap<String, Integer> map) {

if (i == dungeon.length - 1 && j == dungeon[0].length - 1) {

if (dungeon[i][j] > 0) {

return 1;

} else {

return -dungeon[i][j] + 1;

}

}

String key = i + "@" + j;

if (map.containsKey(key)) {

return map.get(key);

}

int right = j < dungeon[0].length - 1 ? calculateMinimumHPHelper(i, j + 1, dungeon, map) : Integer.MAX\_VALUE;

int down = i < dungeon.length - 1 ? calculateMinimumHPHelper(i + 1, j, dungeon, map) : Integer.MAX\_VALUE;

int need = right < down ? right - dungeon[i][j] : down - dungeon[i][j];

if (need <= 0) {

map.put(key, 1);

return 1;

} else {

map.put(key, need);

return need;

}

}

# **解法三 动态规划**

其实解法二递归写完以后，很快就能想到动态规划怎么去解了。虽然它俩本质是一样的，但用动态规划可以节省递归压栈的时间，直接从底部往上走。

我们的状态就定义成解法二递归中返回的值，用 dp[i][j] 表示从 (i, j) 到达终点所需要的最小生命值。

状态转移方程的话和递归也一模一样，只需要把函数调用改成取直接取数组的值。

因为对于边界的情况，我们需要赋值为最大值，所以数组的话我们也扩充一行一列将其初始化为最大值，比如

奖惩数组

1 -3 3

0 -2 0

-3 -3 -3

dp 数组

终点位置就是递归出口时候返回的值，边界扩展一下

用 M 表示 Integer.MAXVALUE

0 0 0 M

0 0 0 M

0 0 4 M

M M M M

然后就可以一行一行或者一列一列的去更新 dp 数组，当然要倒着更新

因为更新 dp[i][j] 的时候我们需要 dp[i+1][j] 和 dp[i][j+1] 的值

然后代码就出来了，可以和递归代码做个对比。

public int calculateMinimumHP(int[][] dungeon) {

int row = dungeon.length;

int col = dungeon[0].length;

int[][] dp = new int[row + 1][col + 1];

//终点所需要的值

dp[row - 1][col - 1] = dungeon[row - 1][col - 1] > 0 ? 1 : -dungeon[row - 1][col - 1] + 1;

//扩充的边界更新为最大值

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

dp[row][i] = Integer.MAX\_VALUE;

}

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

dp[i][col] = Integer.MAX\_VALUE;

}

//逆过来更新

for (int i = row - 1; i >= 0; i--) {

for (int j = col - 1; j >= 0; j--) {

if (i == row - 1 && j == col - 1) {

continue;

}

//选择向右走还是向下走

dp[i][j] = Math.min(dp[i + 1][j], dp[i][j + 1]) - dungeon[i][j];

if (dp[i][j] <= 0) {

dp[i][j] = 1;

}

}

}

return dp[0][0];

}

如果动态规划做的多的话，必不可少的一步就是空间复杂度可以进行优化，比如 [5题](https://leetcode.windliang.cc/leetCode-5-Longest-Palindromic-Substring.html)，[10题](https://leetcode.windliang.cc/leetCode-10-Regular-Expression-Matching.html)，[53题](https://leetcode.windliang.cc/leetCode-53-Maximum-Subarray.html?h=%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92)，[72题](https://leetcode.wang/leetCode-72-Edit-Distance.html) ，[115 题](https://leetcode.wang/leetcode-115-Distinct-Subsequences.html) 等等都已经用过了。

因为我们的 dp 数组在更新第 i 行的时候，我们只需要第 i+1 行的信息，而 i+2，i+3 行的信息我们就不再需要了，我们我们其实不需要二维数组，只需要一个一维数组就足够了。

public int calculateMinimumHP(int[][] dungeon) {

int row = dungeon.length;

int col = dungeon[0].length;

int[] dp = new int[col + 1];

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

dp[i] = Integer.MAX\_VALUE;

}

dp[col - 1] = dungeon[row - 1][col - 1] > 0 ? 1 : -dungeon[row - 1][col - 1] + 1;

for (int i = row - 1; i >= 0; i--) {

for (int j = col - 1; j >= 0; j--) {

if (i == row - 1 && j == col - 1) {

continue;

}

dp[j] = Math.min(dp[j], dp[j + 1]) - dungeon[i][j];

if (dp[j] <= 0) {

dp[j] = 1;

}

}

}

return dp[0];

}

# **总**

回过来看这道题，其实有时候只是一个思维的逆转，就可以把问题解决了。

开始的时候，想求出从起点出发到任点的所需的最小生命值，然后发现走到了死胡同，因为根据当前的信息无法指导未来的方向。而思维逆转过来，从未来往回走，去求出任一点到终点所需要的最小生命值，问题瞬间得到了解决。

第一次遇到这样的动态规划题目，之前的动态规划无论从左上角到右下角，还是从右下角到左上角都是可以做的。而这个题由于有两个变量，所以只允许一个方向才能解题，很有意思。所以，最根本的原因就是终点到起点和起点到终点所需要的最小生命值并不一定是相同的。

遇到问题到了死胡同，不如逆过来去解决问题，太妙了！

**Refer to**

<https://segmentfault.com/a/1190000003884349>

### 动态规划

#### 复杂度

时间 O(N) 空间 O(N) 递归栈

#### 思路

骑士向右或者向下走，如果血量小于0就死掉了，这会使得计算变得很复杂。如果我们从后往前看，从最后一个格子逆推回去，就会简单很多。每个格子可以是它下方或者右方的格子逆推回来，那么要让其实的血量最少，我们则要保证逆推的每一步都处于活着的状态，且选择活着的状态中，血量较小的那一种。假设health[i][j]表示点i和j的血量，dungeon[i][j]表示走到i和j要扣除的血量。如果从下方逆推回上面，则血量为health[i][j] = health[i + 1][j] - dungeon[i][j]，但要考虑，如果该格子如果扣血扣太多的，则这样相减血量会成为负数，说明骑士就已经死了，这样的话我们要保证扣完血后骑士还活着，则该点的血量就应该为1。所以其实是health[i][j] = Math.max(health[i + 1][j] - dungeon[i][j], 1)。同理，如果从右边逆推回来，则health[i][j] = Math.max(health[i][j] - dungeon[i][j + 1], 1)。最后，我们在这两个逆推的值中，取较小的那个就行了。

#### 注意

* 由于最下面一行和最右面一列比较特殊，只有一种逆推方法，所以我们要先单独处理一下。
* 最右下角那个节点没有待逆推的节点，所以我们假设其逆推节点的血量为1。

#### 代码

public class Solution {

public int calculateMinimumHP(int[][] dungeon) {

if(dungeon == null || dungeon.length == 0) return 1;

int m = dungeon.length;

int n = dungeon[0].length;

int[][] health = new int[m][n];

health[m - 1][n - 1] = Math.max(1 - dungeon[m - 1][n - 1], 1);

// 逆推最后一列的血量

for(int i = m - 2; i >= 0; i--){

health[i][n - 1] = Math.max(health[i + 1][n - 1] - dungeon[i][n - 1], 1);

}

// 逆推最后一行的血量

for(int j = n - 2; j >= 0; j--){

health[m - 1][j] = Math.max(health[m - 1][j + 1] - dungeon[m - 1][j], 1);

}

// 对于每个节点，从其下方和右方逆推回来

for(int i = m - 2; i >= 0; i--){

for(int j = n - 2; j >= 0; j--){

int down = Math.max(health[i + 1][j] - dungeon[i][j], 1);

int right = Math.max(health[i][j + 1] - dungeon[i][j], 1);

health[i][j] = Math.min(down, right);

}

}

return health[0][0];

}

}