求最大子矩阵的大小

【题目】

给定一个整型矩阵 map，其中的值只有 0 和 1 两种，求其中全是 1 的所有矩形区域中，最大的矩形区域为 1 的数量。

例如：

```
1  1  1  0
```

其中，最大的矩形区域有 3 个 1，所以返回 3。

再如：

```
1  0  1  1
1  1  1  1
1  1  1  0
```

其中，最大的矩形区域有 6 个 1，所以返回 6。

【难度】

校 ★★★☆

【解答】

如果矩阵的大小为 $O(N \times M)$，本题可以做到时间复杂度为 $O(N \times M)$。解法的具体过程为：

1．矩阵的行数为 N，以每一行做切割，统计以当前行作为底的情况下，每个位置往上的 1 的数量。使用高度数组 height 来表示。

例如：

```
map  =  1  0  1  1
        1  1  1  1
        1  1  1  0
```

以第 1 行做切割后，height={1,0,1,1}，height[j]表示目前的底上（第 1 行），j 位置往上（包括 j 位置）有多少连续的 1。

以第 2 行做切割后，height={2,1,2,2}，注意到从第一行到第二行，height 数组的更新是十分方便的，即 height[j] = map[i][j]==0 ? 0 : height[j]+1。

以第 3 行做切割后，height={3,2,3,0}。

2．对于每一次切割，都利用更新后的 height 数组来求出以每一行为底的情况下，最大

的矩形是什么。那么这么多次切割中，最大的那个矩形就是我们要的。

整个过程就是如下代码中的 maxRecSize 方法。步骤 2 的实现是如下代码中的 maxRecFromBottom 方法。

下面重点介绍一下步骤 2 如何快速地实现，这也是这道题最重要的部分，如果 height 数组的长度为 M，那么求解步骤 2 的过程可以做到时间复杂度为 $O(M)$。

对于 height 数组，读者可以理解为一个直方图，比如{3,2,3,0}，其实就是如图 1-6 所示的直方图。

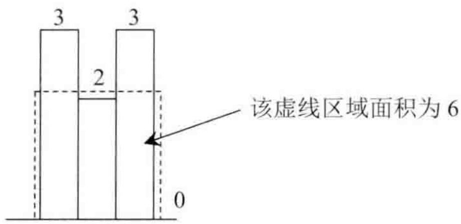

图 1-6

也就是说，步骤 2 的实质是在一个大的直方图中求最大矩形的面积。如果我们能够求出以每一根柱子扩展出去的最大矩形，那么其中最大的矩形就是我们想找的。比如：

- 第 1 根高度为 3 的柱子向左无法扩展，它的右边是 2，比 3 小，所以向右也无法扩展，则以第 1 根柱子为高度的矩形面积就是 3*1==3；
- 第 2 根高度为 2 的柱子向左可以扩 1 个距离，因为它的左边是 3，比 2 大；右边的柱子也是 3，所以向右也可以扩 1 个距离，则以第 2 根柱子为高度的矩形面积就是 2*3==6；
- 第 3 根高度为 3 的柱子向左没法扩展，向右也没法扩展，则以第 3 根柱子为高度的矩形面积就是 3*1==3；
- 第 4 根高度为 0 的柱子向左没法扩展，向右也没法扩展，则以第 4 根柱子为高度的矩形面积就是 0*1==0；

所以，当前直方图中最大的矩形面积就是 6，也就是图 1-6 中虚线框住的部分。

考查每一根柱子最大能扩多大，这个行为的实质就是找到柱子左边刚比它小的柱子位置在哪里，以及右边刚比它小的柱子位置在哪里。这个过程怎么计算最快呢？用栈。

为了方便表述，我们以 height={3,4,5,4,3,6}为例说明如何根据 height 数组求其中的最大矩形。具体过程如下：

1．生成一个栈，记为 stack，从左到右遍历 height 数组，每遍历一个位置，都会把位

置压进 stack 中。

2．遍历到 height 的 0 位置，height[0]=3，此时 stack 为空，直接将位置 0 压入栈中，此时 stack 从栈顶到栈底为{0}。

3．遍历到 height 的 1 位置，height[1]=4，此时 stack 的栈顶为位置 0，值为 height[0]=3，又有 height[1]>height[0]，那么将位置 1 直接压入 stack。这一步体现了遍历过程中的一个关键逻辑：只有当前 i 位置的值 height[i]大于当前栈顶位置所代表的值(height[stack.peek()])，则 i 位置才可以压入 stack。

所以可以知道，stack 中从栈顶到栈底的位置所代表的值是依次递减，并且无重复值，此时 stack 从栈顶到栈底为{1,0}。

4．遍历到 height 的 2 位置，height[2]=5，与步骤 3 的情况完全一样，所以直接将位置 2 压入 stack，此时 stack 从栈顶到栈底为{2,1,0}。

5．遍历到 height 的 3 位置，height[3]=4，此时 stack 的栈顶为位置 2，值为 height[2]=5，又有 height[3]<height[2]。此时又出现了一个遍历过程中的关键逻辑，即如果当前 *i* 位置的值 height[i]小于或等于当前栈顶位置所代表的值(height[stack.peek()])，则把栈中存的位置不断弹出，直到某一个栈顶所代表的值小于 height[i]，再把位置 *i* 压入，并在这期间做如下处理：

1）假设当前弹出的栈顶位置记为位置 *j*，弹出栈顶之后，新的栈顶记为 *k*。然后我们开始考虑位置 *j* 的柱子向右和向左最远能扩到哪里。

2）对位置 *j* 的柱子来说，向右最远能扩到哪里呢？

如果 height[j]>height[i]，那么 *i*-1 位置就是向右能扩到的最远位置。因为 *j* 之所以被弹出，就是因为遇到了第一个比位置 *j* 值小的位置。

如果 height[j]==height[i]，那么 *i*-1 位置不一定是向右能扩到的最远位置，只是起码能扩到的位置。那怎么办呢？

可以肯定的是，在这种情况下，*i* 位置的柱子向左必然也可以扩到 *j* 位置。也就是说，*j* 位置的柱子扩出来的最大矩形和 *i* 位置的柱子扩出来的最大矩形是同一个。

所以，此时可以不再计算 *j* 位置的柱子能扩出来的最大矩形，因为位置 *i* 肯定要压入到栈中，那就等位置 *i* 弹出的时候再说。

3）对位置 *j* 的柱子来说，向左最远能扩到哪里呢？

肯定是 *k*+1 位置。首先，height[k+1..j-1]之间不可能有小于或等于 height[k]的值，否则 *k* 位置早从栈里弹出了。

然后因为在栈里 *k* 位置和 *j* 位置原本是相邻的，并且从栈顶到栈底的位置所代表的值是依次递减并且无重复值，所以在 height[k+1..j-1]之间不可能有大于或等于 height[k]，同时

又小于或等于 height[j]的，因为如果有这样的值，k 和 j 在栈中就不可能相邻。

所以，height[k+1..j-1]之间的值必然是既大于 height[k]，又大于 height[j]的，所以 j 位置的柱子向左最远可以扩到 k+1 位置。

4）综上所述，j 位置的柱子能扩出来的最大矩形为(i-k-1)*height[j]。

以例子来说明：

① i==3，height[3]=4，此时 stack 的栈顶为位置 2，值为 height[2]=5，故 height[3]<=height[2]，所以位置 2 被弹出（j==2），当前栈顶变为 1（k==1）。位置 2 的柱子扩出来的最大矩形面积为(3-1-1)*5==5。

② i==3，height[3]=4，此时 stack 的栈顶为位置 1，值为 height[1]=4，故 height[3]<=height[1]，所以位置 1 被弹出（j==1），当前栈顶变为 1（k==0）。位置 1 的柱子扩出来的最大矩形面积为(3-0-1)*4==8，这个值实际上是不对的（偏小），但在位置 3 被弹出的时候是能够重新正确计算得到的。

③ i==3，height[3]=4，此时 stack 的栈顶为位置 0，值为 height[0]=3，这时 height[3]<=height[2]，所以位置 0 不弹出。

④ 将位置 3 压入 stack，stack 从栈顶到栈底为{3,0}。

6．遍历到 height 的 4 位置，height[4]=3。与步骤 5 的情况类似，以下是弹出过程：

1）i==4，height[4]=3，此时 stack 的栈顶为位置 3，值为 height[3]=4，故 height[4]<=height[3]，所以位置 3 被弹出（j==3），当前栈顶变为 0（k==0）。位置 3 的柱子扩出来的最大矩形面积为(4-0-1)*4==12。这个最大面积也是位置 1 的柱子扩出来的最大矩形面积，在位置 1 被弹出时，这个矩形其实没有找到，但在位置 3 这里找到了。

2）i==4，height[4]=3，此时 stack 的栈顶为位置 0，值为 height[0]=3，故 height[4]<=height[0]，所以位置 0 被弹出（j==0），当前没有了栈顶元素，此时可以认为 k==-1。位置 0 的柱子扩出来的最大矩形面积为(4-(-1)-1)*3==12，这个值实际上是不对的（偏小），但在位置 4 被弹出时是能够重新正确计算得到的。

3）栈已经为空，所以将位置 4 压入 stack，此时从栈顶到栈底为{4}。

7．遍历到 height 的 5 位置，height[5]=6，情况和步骤 3 类似，直接压入位置 5，此时从栈顶到栈底为{5,4}。

8．遍历结束后，stack 中仍有位置没有经历扩的过程，从栈顶到栈底为{5,4}。此时因为 height 数组再往右不能扩出去，所以认为 i==height.length==6 且越界之后的值极小，然后开始弹出留在栈中的位置：

1）i==6，height[6]极小，此时 stack 的栈顶为位置 5，值为 height[5]=6，故

height[6]<=height[5]，所以位置 6 被弹出（j==6），当前栈顶变为 4（k==4）。位置 5 的柱子扩出来的最大矩形面积为(6-4-1)*6==6。

2）i==6，height[6]极小，此时 stack 的栈顶为位置 4，值为 height[4]=3，故 height[6]<=height[4]，所以位置 4 被弹出（j==4），栈空了，此时可以认为 k==-1。位置 4 的柱子扩出来的最大矩形面积为(6-(-1)-1)*3==18。这个最大面积也是位置 0 的柱子扩出来的最大矩形面积，在位置 0 被弹出的时候，这个矩形其实没有找到，但在位置 4 这里找到了。

3）栈已经空了，过程结束。

9．整个过程结束，所有找到的最大矩形面积中 18 是最大的，所以返回 18。

研究以上 9 个步骤时我们发现，任何一个位置都仅仅进出栈 1 次，所以时间复杂度为 $O(M)$。既然每做一次切割处理的时间复杂度为 $O(M)$，一共做 N 次，则总的时间复杂度为 $O(N\times M)$。

全部过程参看如下代码中的 maxRecSize 方法。9 个步骤的详细过程参看代码中的 maxRecFromBottom 方法。

```java
public int maxRecSize(int[][] map) {
	if (map == null || map.length == 0 || map[0].length == 0) {
		return 0;
	}
	int maxArea = 0;
	int[] height = new int[map[0].length];
	for (int i = 0; i < map.length; i++) {
		for (int j = 0; j < map[0].length; j++) {
			height[j] = map[i][j] == 0 ? 0 : height[j] + 1;
		}
		maxArea = Math.max(maxRecFromBottom(height), maxArea);
	}
	return maxArea;
}

public int maxRecFromBottom(int[] height) {
	if (height == null || height.length == 0) {
		return 0;
	}
	int maxArea = 0;
	Stack<Integer> stack = new Stack<Integer>();
	for (int i = 0; i < height.length; i++) {
		while (!stack.isEmpty() && height[i] <= height[stack.peek()]) {
			int j = stack.pop();
			int k = stack.isEmpty() ? -1 : stack.peek();
			int curArea = (i - k - 1) * height[j];
			maxArea = Math.max(maxArea, curArea);
		}
```

```
            stack.push(i);
        }
        while (!stack.isEmpty()) {
            int j = stack.pop();
            int k = stack.isEmpty() ? -1 : stack.peek();
            int curArea = (height.length - k - 1) * height[j];
            maxArea = Math.max(maxArea, curArea);
        }
        return maxArea;
    }
```

最大值减去最小值小于或等于 num 的子数组数量

【题目】

给定数组 arr 和整数 num，共返回有多少个子数组满足如下情况：

max(arr[i..j]) - min(arr[i..j]) <= num

max(arr[i..j])表示子数组 arr[i..j]中的最大值，min(arr[i..j])表示子数组 arr[i..j]中的最小值。

【要求】

如果数组长度为 N，请实现时间复杂度为 $O(N)$的解法。

【难度】

校 ★★★☆

【解答】

首先介绍普通的解法，找到 arr 的所有子数组，一共有 $O(N^2)$个，然后对每一个子数组做遍历找到其中的最小值和最大值，这个过程时间复杂度为 $O(N)$，然后看看这个子数组是否满足条件。统计所有满足的子数组数量即可。普通解法容易实现，但是时间复杂度为 $O(N^3)$，本书不再详述。最优解可以做到时间复杂度 $O(N)$，额外空间复杂度 $O(N)$，在阅读下面的分析过程之前，请读者先阅读本章“生成窗口最大值数组”问题，本题所使用到的双端队列结构与解决“生成窗口最大值数组”问题中的双端队列结构含义基本一致。

生成两个双端队列 qmax 和 qmin。当子数组为 arr[i..j]时，qmax 维护了窗口子数组 arr[i..j]的最大值更新的结构，qmin 维护了窗口子数组 arr[i..j]的最小值更新的结构。当子数组 arr[i..j]向右扩一个位置变成 arr[i..j+1]时，qmax 和 qmin 结构可以在 $O(1)$的时间内更新，并且可以