

```

    }
}

public void swap(HeapNode[] heap, int index1, int index2) {
    HeapNode tmp = heap[index1];
    heap[index1] = heap[index2];
    heap[index2] = tmp;
}

```

边界都是 1 的最大正方形大小

【题目】

给定一个 $N \times N$ 的矩阵 `matrix`，在这个矩阵中，只有 0 和 1 两种值，返回边框全是 1 的最大正方形的边长长度。

例如：

```

0  1  1  1  1
0  1  0  0  1
0  1  0  0  1
0  1  1  1  1
0  1  0  1  1

```

其中，边框全是 1 的最大正方形的大小为 4×4 ，所以返回 4。

【难度】

尉 ★★☆☆

【解答】

先介绍一个比较容易理解的解法：

1. 矩阵中一共有 $N \times N$ 个位置。 $O(N^2)$
2. 对每一个位置都看是否可以成为边长为 $N-1$ 的正方形左上角。比如，对于 $(0,0)$ 位置，依次检查是否是边长为 5 的正方形左上角，然后检查边长为 4、3 等。 $O(N)$
3. 如何检查一个位置是否可以成为边长为 N 的正方形的左上角呢？遍历这个边长为 N 的正方形边界看是否只由 1 构成，也就是走过 4 个边的长度 $(4N)$ 。 $O(N)$

所以普通方法总的时间复杂度为 $O(N^2) \times O(N) \times O(N) = O(N^4)$ 。

本书提供的方法的时间复杂度为 $O(N^3)$ ，基本过程也是如上三个步骤。但是对于步骤 3，

可以把时间复杂度由 $O(N)$ 降为 $O(1)$ 。具体地说，就是能够在 $O(1)$ 的时间内检查一个位置假设为 (i, j) ，是否可以作为边长为 $a(1 \leq a \leq N)$ 的边界全是 1 的正方形左上角。关键是使用预处理技巧，这也是面试经常使用的技巧之一，下面介绍得到预处理矩阵的过程。

1. 预处理过程是根据矩阵 `matrix` 得到两个矩阵 `right` 和 `down`。`right[i][j]` 的值表示从位置 (i, j) 出发向右，有多少个连续的 1。`down[i][j]` 的值表示从位置 (i, j) 出发向下有多少个连续的 1。

2. `right` 和 `down` 矩阵如何计算？

1) 从矩阵的右下角 $(n-1, n-1)$ 位置开始计算，如果 `matrix[n-1][n-1]==1`，那么，`right[n-1][n-1]=1` 且 `down[n-1][n-1]=1`，否则都等于 0。

2) 从右下角开始往上计算，即在 `matrix` 最后一列上计算，位置就表示为 $(i, n-1)$ 。对 `right` 矩阵来说，最后一列的右边没有内容，所以，如果 `matrix[i][n-1]==1`，则令 `right[i][n-1]=1`，否则为 0。对 `down` 矩阵来说，如果 `matrix[i][n-1]==1`，因为 `down[i+1][n-1]` 表示包括位置 $(i+1, n-1)$ 在内并往下有多少个连续的 1，所以，如果位置 $(i, n-1)$ 是 1，那么，令 `down[i][n-1]=down[i+1][n-1]+1`；如果 `matrix[i][n-1]==0`，则令 `down[i][n-1]=0`。

3) 从右下角开始往左计算，即在 `matrix` 最后一行上计算，位置可以表示为 $(n-1, j)$ 。对 `right` 矩阵来说，如果 `matrix[n-1][j]==1`，因为 `right[n-1][j+1]` 表示包括位置 $(n-1, j+1)$ 在内右边有多少个连续的 1。所以，如果位置 $(n-1, j)$ 是 1，则令 `right[n-1][j]=right[n-1][j+1]+1`；如果 `matrix[n-1][j]==0`，则令 `right[n-1][j]=0`。对 `down` 矩阵来说，最后一列的下边没有内容，所以，如果 `matrix[n-1][j]==1`，令 `down[n-1][j]=1`，否则为 0。

4) 计算完步骤 1) ~ 步骤 3) 之后，剩下的位置都是既有右，也有下，假设位置表示为 (i, j) ：

如果 `matrix[i][j]==1`，则令 `right[i][j]=right[i][j+1]+1`，`down[i][j]=down[i+1][j]+1`。

如果 `matrix[i][j]==0`，则令 `right[i][j]=0`，`down[i][j]=0`。

预处理的具体过程请参看如下代码中的 `setBorderMap` 方法。

得到 `right` 和 `down` 矩阵后，如何加速检查过程呢？比如现在想检查一个位置，假设为 (i, j) 。是否可以作为边长为 $a(1 \leq a \leq N)$ 的边界全为 1 的正方形左上角。

1) 位置 (i, j) 的右边和下边连续为 1 的数量必须都大于或等于 $a(\text{right}[i][j] \geq a \ \&\& \ \text{down}[i][j] \geq a)$ ，否则说明上边界和左边界的 1 不够。

2) 位置 (i, j) 向右跳到位置 $(i, j+a-1)$ ，这个位置是正方形的右上角，那么这个位置的下边连续为 1 的数量也必须大于或等于 $a(\text{down}[i][j+a-1] \geq a)$ ，否则说明右边界的 1 不够。

3) 位置 (i, j) 向下跳到位置 $(i+a-1, j)$ ，这个位置是正方形的左下角，那么这个位置的右边

连续为 1 的数量也必须大于或等于 $a(\text{right}[i+a-1][j] \geq a)$, 否则说明下边界的 1 不够。

以上三个条件都满足时, 就说明位置 (i, j) 符合要求, 利用 `right` 和 `down` 矩阵之后, 加速的过程很明显, 不需要遍历边长上的所有值了, 只看 4 个点即可。

全部过程请参看如下代码中的 `getMaxSize` 方法。

```
public void setBorderMap(int[][] m, int[][] right, int[][] down) {
    int r = m.length;
    int c = m[0].length;
    if (m[r - 1][c - 1] == 1) {
        right[r - 1][c - 1] = 1;
        down[r - 1][c - 1] = 1;
    }
    for (int i = r - 2; i != -1; i--) {
        if (m[i][c - 1] == 1) {
            right[i][c - 1] = 1;
            down[i][c - 1] = down[i + 1][c - 1] + 1;
        }
    }
    for (int i = c - 2; i != -1; i--) {
        if (m[r - 1][i] == 1) {
            right[r - 1][i] = right[r - 1][i + 1] + 1;
            down[r - 1][i] = 1;
        }
    }
    for (int i = r - 2; i != -1; i--) {
        for (int j = c - 2; j != -1; j--) {
            if (m[i][j] == 1) {
                right[i][j] = right[i][j + 1] + 1;
                down[i][j] = down[i + 1][j] + 1;
            }
        }
    }
}

public int getMaxSize(int[][] m) {
    int[][] right = new int[m.length][m[0].length];
    int[][] down = new int[m.length][m[0].length];
    setBorderMap(m, right, down);
    for (int size = Math.min(m.length, m[0].length); size != 0; size--) {
        if (hasSizeOfBorder(size, right, down)) {
            return size;
        }
    }
    return 0;
}

public boolean hasSizeOfBorder(int size, int[][] right, int[][] down) {
    for (int i = 0; i != right.length - size + 1; i++) {
        for (int j = 0; j != right[0].length - size + 1; j++) {
            if (right[i][j] >= size && down[i][j] >= size
```

```

        && right[i + size - 1][j] >= size
        && down[i][j + size - 1] >= size) {
            return true;
        }
    }
    return false;
}

```

不包含本位置值的累乘数组

【题目】

给定一个整型数组 `arr`，返回不包含本位置值的累乘数组。

例如，`arr=[2,3,1,4]`，返回`[12,8,24,6]`，即除自己外，其他位置上的累乘。

【要求】

1. 时间复杂度为 $O(N)$ 。
2. 除需要返回的结果数组外，额外空间复杂度为 $O(1)$ 。

【进阶题目】

对时间和空间复杂度的要求不变，而且不可以使用除法。

【难度】

士 ★☆☆☆

【解答】

先介绍可以使用除法的实现，结果数组记为 `res`，所有数的乘积记为 `all`。如果数组中不含 0，则设置 `res[i]=all/arr[i](0<=i<n)` 即可。如果数组中有 1 个 0，对唯一的 `arr[i]==0` 的位置令 `res[i]=all`，其他位置上的值都是 0 即可。如果数组中 0 的数量大于 1，那么 `res` 所有位置上的值都是 0。具体过程请参看如下代码中的 `product1` 方法。

```

public int[] product1(int[] arr) {
    if (arr == null || arr.length < 2) {
        return null;
    }
    int count = 0;

```