机器人位于一个*m* x *n*网格的左上角（在下图中标记为“开始”）。

机器人只能随时向下或向右移动。机器人正在尝试到达网格的右下角（在下图中标记为“完成”）。有多少可能的独特路径？



题目：路径的遍历，典型dp例题。

首先，让我们来看看。

由于机器人只能向右和向下移动，当它到达某一点时，只有两种可能性：

1. 它从上面到达那个点（移到那个点）;
2. 它从左边到达那个点（向右移动到那一点）。

因此，我们有下面的状态方程：假设到达某一点的路径数(i, j)表示为P[i][j]，很容易得出结论P[i][j] = P[i - 1][j] + P[i][j - 1]。

上述等式的边界条件出现在最左列（P[i][j - 1]不存在）和最上列（P[i - 1][j]不存在）。这些条件可以通过初始化（预处理）来处理 - P[0][j] = 1, P[i][0] = 1对所有有效的初始化i, j。请注意，初始值1不是0！

现在我们可以写下下面的（未优化的）代码。

public static int uniquePaths(int m,int n){

int[][] dp = new int[m][n];//用于记录当前路径的m\*n规格的dp数组

//初始化dp数组，第0行，第0列都为1

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

dp[i][0]=1;

}

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

dp[0][i]=1;

}

//开始计算非0行非0列的路径，即dp[i][j]的值

//dp[i][j] = dp[i-1][j] + dp[i][j-1]

for(int i = 1;i<m;i++){

for(int j=1;j<n;j++){

dp[i][j]=dp[i-1][j]+dp[i][j-1];

}

}

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

}

时间复杂度O(m\*n),空间复杂度O(m\*n)

可以看出，上述解决方案O(n^2)及时运行，成本O(m\*n)空间较大。但是，您可能已经观察到，每当我们更新时path[i][j]，我们只需要path[i - 1][j]（在同一列）和path[i][j - 1]（在左侧列）。所以维护两列（当前列和左列）就足够了，而不是维护整个m\*n矩阵。现在代码可以被优化以具有O(min(m, n))空间复杂性。

下面是上面最基本的dp数组的平面图示例

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 3 | 6 | 10 | 15 | 21 |

我们看到边界都是1，我们如何压缩dp数组呢？

首先path[i][j]=path[i-1][j] + path[i][j-1] ，

我们考虑path[1][1] = path[0][1]+path[1][0]

Path[1][2]=path[0][2]+path[1][1]

我们会发现，当前位置的路径数等于之前一列的路径数+当前列的上一个路径数。

我们试图把数组压缩，脑袋里想我们给数组压憋，合并成1\*3可以吗，我们从左到右逐步合并。当我们计算完第二列的路径，第一列数据已经没有用了，我们给覆盖掉，同理第三列覆盖掉第二列，…，这样我们就从左到右维护一个1\*3的数列就可以了。

|  |
| --- |
| 1 |
| 1 |
| 1 |

我们从左到右逐步维护。上面为第一列，因为处于边界，所有数组初始化都是1.

Java中初始化填充数组的方法是Arrays.fill(dp,1);

我们开始计算第二列

第二列第一个数字为边界值不用管始终为1，也就是path[0]=1.

第二列的第二个数字，根据公式，第二列的path[1]=第一列的path[1] + 第二列的path[0]。

然后我们用计算出来的第二列的path[1]替换掉第一列的path[1]。

同理第二列的第三个数字。

依次类推，直到最后一列。

最后一列的最后一个数为终点，所以path[2]就是最大路径数。

压缩后的空间复杂度为O(min(m,n))，时间复杂度O(m\*n)没变化

代码

public static int uniquePaths(int m,int n){

//这里默认m最小

//如果m>n进行处理,m和n互换

if(m>n)

return uniquePaths(n,m);

int[] dp = new int[m];//用于记录路径数的dp数组

//dp数组的初始化，全部初值为1

Arrays.fill(dp,1);

//计算非边界的路径数

//因为非边界值是非0行或者非0列，所以我们都从1开始

for(int i = 1;i<n;i++){

for(int j=1;j<m;j++){

//本行的dp[j]=上一行的dp[j]+本行的dp[j-1]

dp[j]=dp[j]+dp[j-1];

}

}

return dp[m-1];

}