Unique Paths

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Above is a 3 x 7 grid. How many possible unique paths are there?

Note: *m* and *n* will be at most 100.

Solution 1

Binomial coefficient:

```
class Solution {
    public:
        int uniquePaths(int m, int n) {
            int N = n + m - 2;// how much steps we need to do
            int k = m - 1; // number of steps that need to go down
            double res = 1;
            // here we calculate the total possible path number
            // Combination(N, k) = n! / (k!(n - k)!)
            // reduce the numerator and denominator and get
            // C = ( (n - k + 1) * (n - k + 2) * ... * n ) / k!
            for (int i = 1; i <= k; i++)
                res = res * (N - k + i) / i;
            return (int)res;
        }
    };
```

First of all you should understand that we need to do n + m - 2 movements : m - 1 down, n - 1 right, because we start from cell (1, 1).

Secondly, the path it is the sequence of movements(go down / go right), therefore we can say that two paths are different when there is i-th (1 .. m + n - 2) movement in path1 differ i-th movement in path2.

So, how we can build paths. Let's choose (n - 1) movements(number of steps to the right) from (m + n - 2), and rest (m - 1) is (number of steps down).

I think now it is obvious that count of different paths are all combinations (n - 1) movements from (m + n-2).

written by d40a original link here

Solution 2

This is a fundamental DP problem. First of all, let's make some observations.

Since the robot can only move right and down, when it arrives at a point, there are only two possibilities:

1. It arrives at that point from above (moving down to that point);
2. It arrives at that point from left (moving right to that point).

Thus, we have the following state equations: suppose the number of paths to arrive at a point `(i, j)` is denoted as `P[i][j]`, it is easily concluded that `P[i][j] = P[i - 1][j] + P[i][j - 1]`.

The boundary conditions of the above equation occur at the leftmost column (`P[i][j - 1]` does not exist) and the uppermost row (`P[i - 1][j]` does not exist). These conditions can be handled by initialization (pre-processing) --- initialize `P[0][j] = 1, P[i][0] = 1` for all valid `i, j`. Note the initial value is `1` instead of `0`!

Now we can write down the following (unoptimized) code.

```
class Solution {
    int uniquePaths(int m, int n) {
        vector<vector<int> > path(m, vector<int> (n, 1));
        for (int i = 1; i < m; i++)
            for (int j = 1; j < n; j++)
                path[i][j] = path[i - 1][j] + path[i][j - 1];
        return path[m - 1][n - 1];
    }
};
```

As can be seen, the above solution runs in `O(n^2)` time and costs `O(m*n)` space. However, you may have observed that each time when we update `path[i][j]`, we only need `path[i - 1][j]` (at the same column) and `path[i][j - 1]` (at the left column). So it is enough to maintain two columns (the current column and the left column) instead of maintaining the full `m*n` matrix. Now the code can be optimized to have `O(min(m, n))` space complexity.

```
class Solution {
    int uniquePaths(int m, int n) {
        if (m > n) return uniquePaths(n, m);
        vector<int> pre(m, 1);
        vector<int> cur(m, 1);
        for (int j = 1; j < n; j++) {
            for (int i = 1; i < m; i++)
                cur[i] = cur[i - 1] + pre[i];
            swap(pre, cur);
        }
        return pre[m - 1];
    }
};
```

Further inspecting the above code, we find that keeping two columns is used to recover `pre[i]`, which is just `cur[i]` before its update. So there is even no need to use two vectors and one is just enough. Now the space is further saved and the code also gets much shorter.

```
class Solution {
    int uniquePaths(int m, int n) {
        if (m > n) return uniquePaths(n, m);
        vector<int> cur(m, 1);
        for (int j = 1; j < n; j++)
            for (int i = 1; i < m; i++)
                cur[i] += cur[i - 1];
        return cur[m - 1];
    }
};
```

Well, till now, I guess you may even want to optimize it to `O(1)` space complexity since the above code seems to rely on only `cur[i]` and `cur[i - 1]`. You may think that 2 variables is enough? Well, it is not. Since the whole `cur` needs to be updated for `n - 1` times, it means that all of its values need to be saved for next update and so two variables is not enough.

written by jianchao.li.fighter original link here

Solution 3

```
public class Solution {
    public int uniquePaths(int m, int n) {
        Integer[][] map = new Integer[m][n];
        for(int i = 0; i<m;i++){
            map[i][0] = 1;
        }
        for(int j= 0;j<n;j++){
            map[0][j]=1;
        }
        for(int i = 1;i<m;i++){
            for(int j = 1;j<n;j++){
                map[i][j] = map[i-1][j]+map[i][j-1];
            }
        }
        return map[m-1][n-1];
    }
}
```

The assumptions are

1. When (n==0||m==0) the function always returns 1 since the robot can't go left or up.
2. For all other cells. The result = uniquePaths(m-1,n)+uniquePaths(m,n-1)

Therefore I populated the edges with 1 first and use DP to get the full 2-D array.

Please give any suggestions on improving the code.

written by yx3110 original link here