Project Euler: Problem 15

Contents

1 Recursive Solution

By inspection, the number of routes from $(0, 0)$ to (m, n) is equal to the number of routes from $(0, 0)$ to $(m - 1, n)$ plus the number of routes from $(0, 0)$ to $(m, n - 1)$. This implies a very simple recursive algorithm:

Algorithm 1 Recursive route-counting function

```
function countRoutes(m,n)
    if n = 0 OR m = 0 then
        return 1
    end if
    return countRoutes(m, n − 1) + countRoutes(m − 1, n)
end function
```

However, it is too slow to give the answer in reasonable time: consider that countRoutes$(20, 20)$ calls countRoutes$(20, 19)$ and countRoutes$(19, 20)$. The former calls countRoutes$(20, 18)$ and countRoutes$(19, 19)$, while the latter calls countRoutes$(19, 19)$ and countRoutes$(18, 20)$.

Notice how countRoutes$(19, 19)$ is present in both of those branches two levels down from the top. Our algorithm is recomputing the same results multiple times. This problem only gets worse as we delve deeper into the recursion. This can be avoided by using *memoization*, a top-down programming technique of caching results for later use. This way each result is only calculated once.

Algorithm 2 Recursive route-counting function, $O(mn)$ time

```
cache ← dict()                                   ▷ Dictionary object

function countRoutes(m,n)
    if n = 0 OR m = 0 then
        return 1
    end if

    if cache[(m, n)] is defined then
        return cache[(m, n)]
    end if

    cache[(m, n)] ← countRoutes(m, n − 1) + countRoutes(m − 1, n)
    return cache[(m, n)]
end function
```

Notice that the cache-lookup code comes after the 0-check for m and n. These two pieces could technically go in either order and work just fine for our purposes, but there's really no need to spend time/resources performing a dictionary lookup if we can "quit" early.

2 Iterative Solution

While the memoized recursion may be suitable for the purposes of the problem, memoized recursion often requires more memory, and some languages have trouble with deeply-nested recursive calls.

We can translate our recursion into an *iterative* solution using a bottom-up programming technique called *dynamic programming*. Instead of starting from $n = 20$ and $m = 20$ and working our way down, we can start from the bottom "base cases" and work our way up, saving the results in a two-dimensional array (which we'll denote via "*grid*") where $grid[i][j]$ corresponds to the result $\text{COUNTROUTES}(i, j)$.

A "base case" is typically a simple case where the recursive algorithm ends. For us, that's any time m or n equals 0, for which the result is always 1. So we initialize all $grid[i][0]$ and $grid[0][j]$ to 1 and then iterate across all $i \leq m$ and $j \leq n$ to calculate each result, storing the results in the array as we go. At the very end, $grid[m][n]$ will have the desired answer.

Algorithm 3 Iterative route-counting function, $O(mn)$ time

```
function COUNTROUTES(m,n)
    grid ← array[m + 1][n + 1]      ▷ A two-dimensional 0-indexed array of size m + 1 by n + 1

    for i = 0 to m do
        grid[i][0] ← 1
    end for

    for j = 0 to n do
        grid[0][j] ← 1
    end for

    for i = 1 to m do
        for j = 1 to n do
            grid[i][j] ← grid[i − 1][j] + grid[i][j − 1]
        end for
    end for

    return grid[m][n]
end function
```


Author: Pim Spelier

3 Combinatorial Solution

However, these solutions are both $O(mn)$, and we can do better using combinatorics. In a grid of size m by n, we know that no matter what path we take, there will be exactly m movements to the right (R) and n movements down (D). This means the pathway can be represented as a string of R's and D's of length $m + n$.

How many ways can we allocate m R's and n D's in such a string? Note that for a single configuration, once we place the the R's we immediately know where the D's must go (as they cannot go anywhere else). All we really have to know is how many ways we can place the R's. The number of ways we can do this, mathematically, is denoted with the binomial coefficient $\binom{m+n}{m}$.

This is called a *combination*, the number of ways of choosing k items from a group of n items where the order of the k items does not matter[1]:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \times (n-1) \times ... \times (n-k+1)}{k!} \tag{3.0.1}$$

Using this formula, we can calculate $\binom{m+n}{m}$ for general n and m, but seeing as our grid is a perfect square ($m = n$):

$$\binom{2n}{n} = \frac{2n \times (2n-1) \times ... \times (n+2) \times (n+1)}{n \times (n-1) \times ... \times 2 \times 1} = \prod_{i=1}^{n} \frac{n+i}{i} \tag{3.0.2}$$

This gives us a much faster $O(n)$ solution:

Algorithm 4 Combination-based route-counting function, $O(n)$ time and $O(1)$ memory

```
function COUNTROUTES(n)                          ▷ n by n grid
    result = 1

    for i = 1 to n do
        result ← result × (n + i)/i
    end for

    return result
end function
```

[1] In equation 3.0.1, let $x!$ be the *factorial function*, $x! = x \times (x-1) \times (x-2) \times ... \times 2 \times 1$

Author: Pim Spelier