JASON'S CODE BLOG

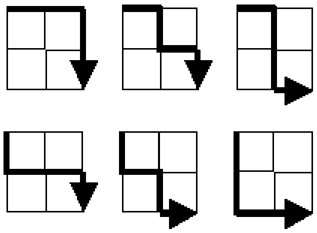
SOME STUFF I FIND USEFUL

Jun 17 (http://code.jasonbhill.com/python/project-euler-problem-15/)

PROJECT EULER PROBLEM 15

BY JASON NO COMMENTS YET

Problem: Starting in the top left corner of a 2×2 grid and moving only down and right, there are 6 routes to the bottom right corner.



How many routes are there through a 20×20 grid?

A GREAT INTERVIEW PROBLEM

This is precisely the sort of problem I expect to see in a technical coding interview, and knowing the various ways of solving a problem such as this will help you get far in those situations. But, even if you’re an experienced coder, the solutions may not be obvious at first. I want to walk through a few potential solutions and see how they perform.

RECURSIVE PYTHON SOLUTION

I think the easiest solution, and one you should know (but possible don’t) is the recursive approach. The main idea here is to develop a function that will call itself, inching along to the right and down in all possible combinations, returning a value of 1 whenever it reaches the bottom-right and summing all of those 1s along the way. You should convince yourself that the procedure actually terminates (at what we call “the base case”) and returns the correct solution. Try doing it on paper on the 2×2 grid, or a 3×3 grid, and see what happens. Here’s what the Python code looks like.

```
#!/usr/bin/python

import time


gridSize = [20,20]

def recPath(gridSize):
    """
    Recursive solution to grid problem. Input is a list of x,y moves remaining.
    """
    # base case, no moves left
    if gridSize == [0,0]: return 1
    # recursive calls
    paths = 0
    # move left when possible
    if gridSize[0] > 0:
        paths += recPath([gridSize[0]-1,gridSize[1]])
    # move down when possible
    if gridSize[1] > 0:
        paths += recPath([gridSize[0],gridSize[1]-1])

    return paths

start = time.time()
result = recPath(gridSize)
elapsed = time.time() - start

print "result %s found in %s seconds" % (result, elapsed)
```

That’s great, and it will actually work, but it may take some time. Actually it takes a lot of time. By that, I mean it really, really takes a lot of time. When we run it on the 2×2 output, we get the following.

```
result 6 found in 9.05990600586e-06 seconds
```

When we run it on the 20×20 input, as the problem requires, it runs for about 4 hours before I kill it. Python is OK at recursive function calls, and it can handle/collapse the memory required moderately well, but what we’re doing here is manually constructing ALL possible paths to a solution, which isn’t incredibly efficient. Still, during a technical interview, this is definitely the first idea for a solution that should pop into your head. It’s quick and easy to write, and many problems can be solved like this.

DYNAMIC PYTHON SOLUTION

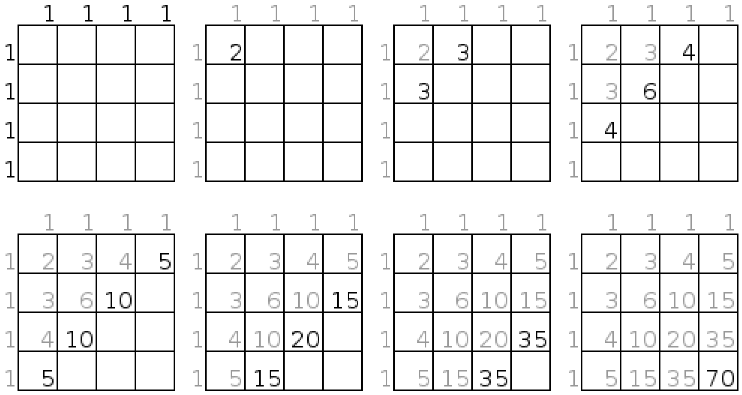
Our recursive approach suffers from the problem that we’re doing a lot of similar operations over and over. Can we learn anything from the smaller cases and build up from there? In this case, the answer is yes, but it requires us to build up some mathematics a bit more. This is actually quite easy if approached correctly. The idea is to construct the solution recursively, but differently this time.

Claim: Let n be any natural number and consider the 2-dimensional sequence $S_{i,j}$ defined by

$$S_{i,j} = \begin{cases} 1 & \text{if } j = 0 \\ S_{i,j-1} + S_{i-1,j} & \text{if } 0 < j < i \\ 2S_{i,j-1} & \text{if } i = j \end{cases}$$

where $0 \leq i \leq n$ and $0 \leq j \leq i$. Then the number of non-backtracking paths from top-left to bottom-right through an $n \times n$ grid is $S_{n,n}$.

Proof: Consider a grid of m rows and n columns (we do not need to assume that the grid is square). Counting from the upper-left and starting at zero, denote the intersection/node in the i-th row and j-th column by $N_{i,j}$. Thus, the upper-left node is $N_{0,0}$, the bottom-left is $N_{m,0}$ and the bottom-right is $N_{m,n}$. Clearly, the number of paths from $N_{0,0}$ to any node along the far left or far top of the grid is only 1 (since we may only proceed down or left). Now, consider how many paths there are to $N_{1,1}$. We must first travel through $N_{0,1}$ or $N_{1,0}$. This yields only two paths to $N_{1,1}$. We can continue this process. In order to determine the total number of paths to any node $N_{i,j}$, we only need to sum together the total number of paths to $N_{i,j-1}$ and $N_{i-1,j}$. The process is understood graphically in the following diagram, where each new integer represents the number of paths to a node.



Thus, in a 4×4 grid, there are 70 non-backtracking paths. How does this relate to the sequence $S_{i,j}$? Simply put, $S_{i,j}$ is the number of paths to node $N_{i,j}$. If we write out the sequence $S_{i,j}$ for $0 \leq i \leq 4$ and $0 \leq j \leq i$, we simply obtain the lower diagonal sequence embedded in the diagram above.

$$\begin{array}{lllll}
S_{0,0} = 1 & & & & \\
S_{1,0} = 1 & S_{1,1} = 2 & & & \\
S_{2,0} = 1 & S_{2,1} = 3 & S_{2,2} = 6 & & \\
S_{3,0} = 1 & S_{3,1} = 4 & S_{3,2} = 10 & S_{3,3} = 20 & \\
S_{4,0} = 1 & S_{4,1} = 5 & S_{4,2} = 15 & S_{4,3} = 35 & S_{4,4} = 70
\end{array}$$

That completes the proof.

Let's code that into a Python solution and see how fast it runs. My bet is that this will be considerably faster than our initial recursive solution. We can record the sequence $S_{i,j}$ as a single dimensional list that is simply rewritten at each iteration.

```
#!/usr/bin/python

import time

def route_num(cube_size):
    L = [1] * cube_size
    for i in range(cube_size):
        for j in range(i):
            L[j] = L[j]+L[j-1]
        L[i] = 2 * L[i - 1]
    return L[cube_size - 1]

start = time.time()
n = route_num(20)
elapsed = (time.time() - start)
print "%s found in %s seconds" % (n,elapsed)
```

When executed, we get the following.

```
137846528820 found in 0.000205039978027 seconds
```

CYTHON SOLUTION

We'll recode things in Cython and see how much faster we can get the result returned.

```
%cython

import time
from libc.stdlib cimport malloc, free

cdef route_num(short cube_size):
    cdef unsigned long *L = <unsigned long *>malloc((cube_size + 1) * sizeof(unsigned long))
    cdef short j,i = 0
    while i <= cube_size:
        L[i] = 1
        i += 1
    i = 1
    while i <= cube_size:
        j = 1
        while j < i:
            L[j] = L[j]+L[j-1]
            j += 1
        L[i] = 2 * L[i - 1]
        i += 1
    cdef unsigned long c = L[cube_size]
    free(L)
    return c

start = time.time()
cdef unsigned long n = route_num(20)
elapsed = (time.time() - start)
print "%s found in %s seconds" % (n,elapsed)
```

We now get the result a bit more quickly.

```
137846528820 found in 2.21729278564e-05 seconds
```

The Cython code executes roughly 9 times as fast as the Python.

POSTED IN CYTHON, PROJECT EULER, PYTHON

← Prev post

Next post →

Comments are closed