**Recursive Solution:**

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. This approach takes a long time though and would take forever for a bigger grid size.

**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()

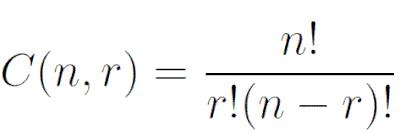
result = recPath(gridSize)

elapsed = time.time() - start

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

**The math approach (kinda lame tbh):**

We know that any path will have 40 moves (20 right + 20 down), so for C(n,r) – n will equal 40 and r (the number of right moves) will equal 20 – C(40,20). All you have to do is follow the formula:



import math

n = 20

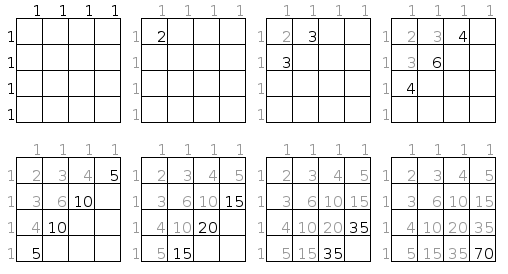
result = math.factorial(2\*n)//(math.factorial(n)\*\*2)

print(result)

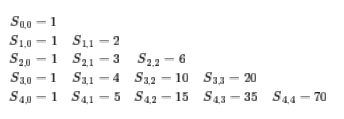
**Efficient approach (many other approaches too):**

The recursive approach above does a lot of similar operations over and over again. Can we learn anything from the smaller cases and build up from there?

Consider a grid of rows **m** and columns **n**. Counting from the upper-left and starting at zero, denote the intersection/node in the **i-th** row and **j-th** column by **.** Thus, the upper-left node is ., the bottom-left is and the bottom-right is **.** Clearly, the number of paths from to any node along the far left or far top of the grid is only 1 (since we may only proceed down or left). In order to determine the total number of paths to any node we only need to sum together the total number of paths to and This is understood graphically in the following diagram, where each new integer represents the number of paths to a node.



Thus, in a 4x4 grid (example above), there are 70 non-backtracking paths. So suppose is the number of paths to node . If we write out the sequence for we obtain the lower diagonal sequence embedded in the diagram above.



Implementation bellow:

**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)