CS125 : Introduction to Computer Science

Lecture Notes #24
Recursive Counting

Lecture 24 : Recursive Counting

Let's consider another problem: grid traversal. Say you have a 2-D array grid

```
 (0, 0)                    (0, 3)

    -------------------------
   |      |        |        |
   |      |        |        |
   |______|________|________|
   |      |        |        |
   |      |        |        |
   |______|________|________|
 (2, 0)                   (2, 3)
```

Travelling only right or down (i.e. increasing numbers only), how many ways are there to get to (2, 3)?

More generally...

```
   (curR, curC)

      -------
     |
     |   ...


                      |
                   ___|
                        (maxR, maxC)
```

How many ways are there to get from (curR, curC) to (maxR, maxC) travelling only right or downward?

From every point, you have two choices.

```
   (curR, curC)
     ----------->    (curR, curC + 1)
     |
     |
     |
    \|/

   (curR + 1, curC)
```

Once you take one of those two choices, then you have a number of options from that point. For example, if you move right, then there are a number of paths to take from (curR, curC + 1). Likewise, if you move down, there are a number of paths to take from (curR + 1, curC). The important point, however, is that those are the only two moves you can make. You either move right and then explore from (curR, curC + 1), or move down and then explore from (curR + 1, curC).

So, the total number of paths from (curR, curC) should be equal to the number of paths that begin by travelling to the right, plus the number of paths that begin by travelling down. There are no other paths to count because those two possible first moves cover everything you can do as your first move, and thus cover all possible options.

That gives us the following so far – it's still incomplete, but it's a beginning:

```
public static int CountPaths(int curR, int curC, int maxR, int maxC)
{
   return CountPaths(curR + 1, curC, maxR, maxC) +
                              CountPaths(curR, curC + 1, maxR, maxC);
}
```

Now, if we've passed up the maximum row, or maximum column, we can't go back upward or to the left to reach it again. So any point where `curR > maxR` or `curC > maxC` automatically has zero paths to the solution. We've gone too far down, or too far to the right, at that point and so the destination is now unreachable. That gives us the following modification to our code so far:

```
public static int CountPaths(int curR, int curC, int maxR, int maxC)
{
   if ((curR > maxR) || (curC > maxC))
      return 0;
   else
      return CountPaths(curR + 1, curC, maxR, maxC) +
                             CountPaths(curR, curC + 1, maxR, maxC);
}
```

So now we have a base case. The problem is, that since the base case always returns zero, the recursive calls will produce, at best, zero, and so the function will always return 0. When do we actually have a correct path that we can count?

And the answer there is, when we've reached the destination, we will count that as a correct path. That is, if we always have to move to the right or downward, we can never stop, so we'll allow ourselves a third operation as well – if we are at the destination, we can stop and count that as a path. That gives us the correct code:

```
public static int CountPaths(int curR, int curC, int maxR, int maxC)
{
   if ((curR > maxR) || (curC > maxC))
      return 0;
   else if ((curR == maxR) && (curC == maxC))
      return 1;
   else
      return CountPaths(curR + 1, curC, maxR, maxC) +
                              CountPaths(curR, curC + 1, maxR, maxC);
}
```

Another example is making change. You have various coin amounts. What is the maximum numbers of ways you can have a given quantity of money?

Consider: Say you had 62 dollars, and could use at most, 10 dollar bills, and as small as pennies. You have the following possibilities for money:

```
10 dollar bills
 5 dollar bills
 2 dollar bills   // yes, such things exist!
 1 dollar bills
 50 cent pieces
 dimes
 nickels
 pennies
```

Now, if we want to know how to make 62 dollars out of the above money, we could say, well, first, let's ask how many 10 dollar bills we have? It's possible we have none. It's possible we have 1. It's also possible we have 2, or 3, or 4, or 5, or 6. But those are the only possibilities. We can't have fewer than 0, and if we have more than 6, we already have 70 dollars and that is too much.

So, if we add together all combinations that total 62 dollars, where we have no 10 dollar bills, plus all combinations that total 62 dollars, where we have exactly 1 ten dollar bill, plus all combinations that total 62 dollars where we have exactly 2 ten dollar bills, and so on, up to all combinations of 62 dollars where we have exactly 6 ten dollar bills, then that total will be the total number of possible ways to have 62 dollars using the above money types – every combination has been represented exactly once in our total since every combination has either exactly 0, or exactly 1, or exactly 2, or exactly 3, or exactly 4, or exactly 5, or exactly 6 ten dollar bills.

The only problem is, okay, let's imagine we want to count all combinations that have exactly 1 ten dollar bill. How do we do that? Well, if we have one ten dollar bill, that means the other 52 dollars needs to come from money that is NOT ten dollar bills! Otherwise, we'd secretly be adding additional ten dollar bills! That is, the rest of what's left over needs to come from the rest of the list of money:

```
 5 dollar bills
 2 dollar bills   // yes, such things exist!
 1 dollar bills
 50 cent pieces
 dimes
 nickels
 pennies
```

That gives us the following:

```
0 tens, plus 62 dollars without using 10 dollar bills
1 ten, plus 52 dollars without using 10 dollar bills
2 tens, plus 42 dollars without using 10 dollar bills
3 tens, plus 32 dollars without using 10 dollar bills
4 tens, plus 22 dollars without using 10 dollar bills
5 tens, plus 12 dollars without using 10 dollar bills
6 tens, plus 2 dollars without using 10 dollar bills
```

That should cover every possibility. We decide how many 10 dollar bills we want, and then force ourselves to make the rest of the change from the lower dollar amounts, to guarantee we don't accidentally add any additional 10 dollar bills. Add up the total size of each category above and you have your answer.

In other words, we can say that the original problem was to "make 62 dollars using *at most* bills worth 10 dollars". In that case, the subproblems whose solutions we add together for our answer, are:

```
0 tens, plus 62 dollars using at most bills worth 5 dollars
1 ten, plus 52 dollars using at most bills worth 5 dollars
2 tens, plus 42 dollars using at most bills worth 5 dollars
3 tens, plus 32 dollars using at most bills worth 5 dollars
4 tens, plus 22 using at most bills worth 5 dollars
5 tens, plus 12 dollars using at most bills worth 5 dollars
6 tens, plus 2 dollars using at most bills worth 5 dollars
```

Once we are down to our lowest money type – pennies in this case – we just return 1, since if I say "make X dollars using only pennies", then there is exactly one way to do that – a pile of pennies large enough to total X dollars.

In both this problem and the paths problem, the key is to divide your collection of items you are counting, into piles, such that:

- every item is in at least one pile
- no item is in more than one pile

That is to say, divide your collection into piles such that every item in the collection is in *exactly one* of the new piles. Then, you recursively count those piles, and add those counting results together to get your total answer.