Backtracking Problems

Recursive backtracking is a problem-solving technique that utilizes recursive method calls to find solutions to a problem by exploring all possible combinations (subsets) of that problem.

When a backtracking algorithm reaches the "bounds" of a problem, it has two options: return the solution, or realize that the current attempt is not a valid solution and "backtrack" to a previous attempt (location).

Review the background info in Canvas. Understanding in a general way how backtracking algorithms work will greatly increase your chances of success in writing the following methods.

1. Complete the method `void climbStairs(int steps)` that prints the all the different ways you can climb a staircase with `steps` number of steps. Each step, you can take either a large stride or a small stride. A large stride will move up two stairs, a small stride will move up one stair.

```
climbStairs(4):     1, 1, 1, 1
                    1, 1, 2
                    1, 2, 1
                    2, 1, 1
                    2, 2
```

2. You're in the middle of the woods, and you know that your campsite is in the north-east corner of the woods. Complete the method `void campsite(int x, int y)` that prints the different routes you can take to get to your campsite at coordinates `<x, y>`. Your starting position will be `0, 0` and you can move (one mile at a time) east (E), north (N), or north-east (NE).

```
campsite(2, 1):     E E N
                    E N E
                    E NE
                    N E E
                    NE E
```

3. Complete the method `int getMax(List<Integer> nums, int limit)` that returns the maximum sum that can be generated by adding elements of `nums` once without going over `limit`.

```
getMax(Arrays.asList(30, 2, 8, 22, 6, 4, 20), 19) >>> 18
```

   Hint: If you're done traversing the list, return the potential max sum or, if the current sum is over the `limit`, a very small number. If neither of the previous cases is true, return the larger of both possible courses of action (adding the current element and skipping the current element).

4. Complete the method `int makeChange(int amount)` that returns the number of different ways you can make change (given standard US coins of 1, 5, 10, 25) for `amount`. What is the time complexity (the Big-O) of this recursive algorithm?

```
makeChange(25)  >>> 13
makeChange(100) >>> 242
```

5. Complete the method `void makeChangeCoinCount(int amount)`. It is similar to `makeChange` but it now prints out all the different ways change can be made, in this format:

```
 P  N  D  Q
------------
[1, 0, 1, 0]
[1, 2, 0, 0]
[6, 1, 0, 0]
[11, 0, 0, 0]
```

6. Complete the method `String longestCommonSub(String a, String b)` that returns the "longest common subsequence" that appears in both Strings. A common subsequence is a sequence of characters that appear in both Strings in the same relative order. This problem is somewhat less intuitive that the previous problems; try it on your own first, but if you get stuck, algorithm help can be found in the lab folder.

```
longestCommonSub("ABCDEFG", "BGCEHAF") >>> "BCEF"
longestCommonSub("12345", "54321 21 54321") >>> "123"
```