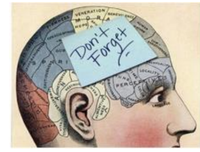
Memoization Problems



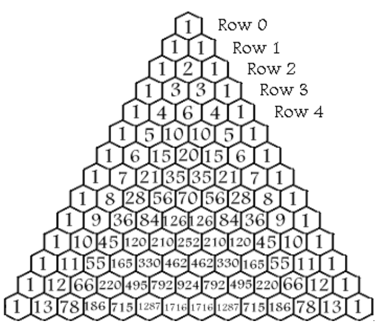
If you have competed in a programming contest, at some point you may have received a "time limit exceeded" message when submitting a solution to a problem that works with the sample (small) data sets. Usually, these problems use recursive backtracking in some form to test all possible combinations of something. With the judges' much larger data sets, a seemingly working solution takes too long and is rejected!

The reason for this is that sub-problems are constantly being re-computed. ***Memoization*** is a process where solutions to sub-problems are cached, rather than re-calculated. This technique turns many problems with exponential time complexity into linear time complexity, without having to relearn the algorithms as you would to use a technique called ***dynamic programming*** (which we'll do next). Create a separate class for each method outlined below and any needed helper methods. Create a client class with a main method that will have test cases for all your methods.

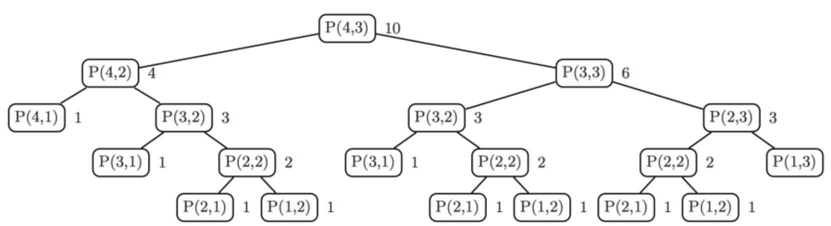
1. Write a method `pascal(int row, int col)` that returns the value of [Pascal's triangle] at `row, col` given the following recursive definition: Every number in Pascal's triangle is defined as the sum of the item above it and the item to the left of it. If there is a position that doesn't have an entry, we treat it as if we had a 0 there (a base case?). Write another method to (attempt to) print the 40th row of the triangle.

```
printPascalRow(9) >>> 1 9 36 84 126 126 84 36 9 1

printPascalRow(40) >>> ??????
```



Below is an invocation tree for `Pascal(4, 3)` in which each call `Pascal(r,c)` has been abbreviated `P(r,c)`:



Once you have the algorithm down, memoize the results to avoid recalculation. Make a wrapper class (I called mine Pair) that wraps a `row, col` pair into a single object, such that you can map an `r, c` location to its value. Make sure you override the `equals` AND `hashCode` methods, so that the HashMap class knows how to find (hash) Pair objects *and* test for equivalency. Re-print the first 40 rows and observe the run time difference.

```
/* Of course the previous problem could be solved with simple variables / arrays, but memoizing their recursive solutions is good practice */
```

2. Write a method `int numPaths(int[][] grid)` that, for the supplied `m` by `n` matrix, returns the number of unique paths from top-left to bottom-right, given you can only go either down or right. Use an `Integer[][]` to cache intermediate results (being able to check for `null` is helpful).

```
numPaths(new int[2][4]) >>> 4

numPaths(new int[3][4]) >>> 10

numPaths(new int[20][12]) >>> 54627300
```

(Advanced) Create a copy of your method that will return the *optimal* path sum (the path with the largest sum) from top-left to bottom-right.

3. Write a method `int minCoins(int total, int[] coins)` that uses recursive backtracking to return the minimum number of coins required to make change for `total`, given the denominations in `coins`. Use a `Map<Integer, Integer>` to store sub-solutions (i.e. min. coins for a specific value). You must use recursive backtracking and memoization to receive credit. Write two versions of this method using a Map and an array to memoize results. (e.g. minCoinsArray, minCoinsMap)

```
minCoins(11, new int[] {9, 6, 5, 1}) >>> 2

minCoins(1000, new int[] {12, 8, 5, 2, 1}) >>> 84
```