CSC 360 Programming Assignment #4: Recursion
Due date: ~~Wednesday, September 24~~ Monday, September 29

We talked about the Missionaries and Cannibals problem in class. You will implement a solution to this problem in this assignment. In this problem, there are 3 Missionaries (M) and 3 Cannibals (C) on the left shore of a river and they want to cross the river to the right shore. There is a boat capable of taking 2 people at a time across the river. What order will the M and C get into and out of the boat to cross the river such that at no time will the Cs outnumber the Ms on either shore (lest the Cs eat the Ms on that shore)? One solution looks like this:

Left shore			Right shore	
MMMCCC				
MMCC	MC			
MMCC		MC		
MMCC		M	C	
MMCC	M		C	
MMC	MC		C	
MMC		MC	C	
MMC		M	CC	
MMC	M		CC	
MM	MC		CC	
MM		MC	CC	
MM		M	CCC	
MM	M		CCC	
M	MM		CCC	
M		MM	CCC	(both Ms get out of the boat while 2 Cs get in the boat)
M		CC	MMC	
M	CC		MMC	
C	MC		MMC	(1 C gets out of the boat, 1 M gets in)
C		MC	MMC	
C		C	MMMC	
C	C		MMMC	
	CC		MMMC	
		CC	MMMC	
			MMMCCC	

At any turn, we see that up to 2 of either M or C can get into or out of the boat in some combination. We see in the above for instance that 1 C gets out of the boat or 1 C gets out and 1 M gets in, we also see at one point that 2Ms get out of the boat while 2Cs get into the boat. The typical solution to this problem uses a 5-tuple to represent the state of the problem. We will instead use a 7-tuple as follows: a, b, c, d, e, f, g where a=Ms on left shore, b=Cs on left shore, c=Ms in the boat, d=Cs in the boat, e=Ms on right shore, f=Cs on right shore, g=location of boat. These are all int values where g=0 if the boat is on the left shore and 1 if it is on the right shore. To solve this problem, we use recursion. The basic recursive algorithm is as follows being called with parameters (3,3,0,0,0,0,0).

```
MandC(int a, int b, int c, int d, int e, int f, int g) {
      boolean temp;
      if(test for solution) return true;
      else if(test for repeated state) return false;
      else if(test for illegal state) return false;
      else { // recursive portion
```

```
                // add this state to the tried states list
                temp=MandC(//try next possible move);
                if(temp) // add the move you just tried in our
                           // solution (an ArrayList)
                else {
                     temp=MandC(//try next possible move);
                     if(temp) // add the move to solution
                     else {
                          …     // continue trying more moves
                     }          // using a nested if-else structure
                }
           }
           return temp;     // return result – did this attempt find
      }                     // a solution or reach a dead end?
```

Illegal states are those in which we have a negative number of people on the left shore, right shore or in the boat (any of a, b, c, d, e, f < 0) or a number more than 3 (for a, b, e, f) or a number more than 2 (c, d), or c+d > 2, or Cs outnumbering Ms (a < b and a > 0 or e < f and e > 0).

We will maintain a list of *tried* states using an ArrayList<String>. This ArrayList will store each 6-tuple we have visited. To create this String, concatenate the 6 variables of the Ms and Cs (a, b, c, d, e, f). For instance, "330000" denotes the starting state. Each time we reach a new state, we need to see if we have been at this state before. If so, we don't want to continue with this recursive call, so we immediately return false (indicating a dead end). Once we establish that this state has not been visited, is legal, and is not the solution, we add this state to the *tried* list.

The solution is determined as (0, 0, 0, 0, 3, 3, 1). Legal moves are any that have 0, 1 or 2 Ms getting into or out of the boat and 0, 1 or 2 Cs get into or out of the boat as long as the result is not illegal or tried before, as described above.

As an example of a recursive call to MandC, you might use notation like this:

```
      if(g==0) temp=MandC(a-2,b,c+2,d,e,f,1);
```

Indicating that with the boat on the left shore, 2 Ms leave the left shore and get into the boat (a-2, c+2). Note that if someone is already in the boat, then c + d > 2, and thus the move is illegal. If there is only 1 M on the left shore, then a will become -1 which is also illegal. A more complicated situation is when one or more get out of the boat while others get in. Here we see 2 Cs get out of the boat while 2 Ms get into the boat while the boat is on the right shore (g=1).

```
      if(g==1) temp=MandC(a,b,c+2,d-2,e-2,f+2,0);
```

Notice how we toggle g. If the boat is currently on the left shore, the recursive call will have g=1, and if the boat is currently on the right shore, the recursive call will have g=0.

In order to report the solution, we will use a second ArrayList<String>. We will add the successful move only if the result from MandC is true. If this second ArrayList is called *list*, we can add to it the step that was successful. For instance, if our call to MandC is `if(g==1) temp=MandC(a,b,c+2,d-2,e-2,f+2,0);` then if temp is true, we would use `list.add("Move 2 Cs from boat to R and 2 Ms from R to boat");`

Both ArrayList variables should be declared as instance data for the class, not local variables to the recursive method. Solve the problem as stated above. If you are struggling to figure out how to implement the solution, you can search the Internet for solutions to this problem. Please try to figure it out on your own though before seeking any help online. Note: find a recursive solution but not a dynamic programming solution. The common solution uses a 5-tuple instead of a 7-tuple and the solutions that I found only return true or false, they do not record the actual moves to make to solve the problem. You will have to figure out how to add the ArrayList to your solution.

There are many different solutions to the M and C problem. The one on the previous page is only one possible solution. Another merely alternates M and C getting into and out of the boat, for instance: (3,3,0,0,0,0) → (2,2,1,1,0,0) → (2,2,1,0,0,1) → (2,1,1,1,0,1) → (2,1,0,1,1,1) → (1,1,1,1,1,1) → (1,1,0,1,2,1) → (0,1,1,1,2,1) → (0,1,1,0,2,2) → (0,0,1,1,2,2) → (0,0,0,0,3,3). The actual solution your code will find is based on the order of your attempts in the recursive portion of your code.

Extra Credit: If you are able to solve the problem with 3 Ms and 3 Cs, a variant uses 7 Ms and 7 Cs where the boat can hold up to 3 at a time. The same restrictions apply but we add the restriction that at no time can there be more Cs than Ms in the boat or else those Cs eat the one M. This version of the program will be very similar to the 3/3 version except that you will have a lot more possible combinations of moves in each turn, so your nested if-else code will lengthen.

Hand in your commented source code to the 3/3 problem (it should be a single Java file) and the solution that your program found. If you successfully complete the 7/7 problem, hand it in as well as the solution it finds.