CIS 303 Assignment Chapter 4 Due date: Tuesday, February 28, 2017

1. The *Josephus* problem is as follows: *N* people, numbered 1 to *N*, are sitting in a circle. Starting with person 1, a stone is passed. After *M* passes, the person holding the stone is eliminated; the circle closes ranks, and the game continues with the person that was sitting after the eliminated person picking up the stone. The last remaining person wins. For example, if *M* = 0 and *N* = 5, players are eliminated in order, and player 5 wins. If *M*=1 and *N*=5, the order of elimination is 2, 4, 1, 5, and player 3 wins.

   a) (12 pts) Open `Four1.java`, and write a method to solve the Josephus problem. Your method will have the heading: `public static void josephus(int [] people, int M)` , where people is an array of *N* integers 1 to *N* representing the original group of *N* people. *M* is, of course, the number of passes to use. Implement the most efficient solution you can devise. You are allowed to write helper methods and say, reassign the people to a linked list or some other structure that you choose to use. However, if you use a linked list or another structure, **you must write it yourself** (that is, **no using Java's Collection Framework**). Maintain the top level call as given in the `main()` method. Report the order of elimination and who wins. Here is some example input/output for *N* = 6, and *M* = 25:

```
grabowlm$ java Four1 6 25
removing person 2
removing person 3
removing person 5
removing person 1
removing person 6
Person remaining: 4
```

   Submit your work: `submit 303 Four1.java`

   b) (12 pts) Analyze the runtime of your program in terms of big-Oh (upper bound). Give a careful analysis. You may want to insert counts into your method to verify your analysis. If you do, provide that evidence as part of your analysis. If you use a different data structure, for example, build a circular linked list of *N* people, do NOT count construction time in your counts, just count the work that must be done within your algorithm with the data structure in place to eliminate everyone except that last person. Is your implementation dependent on *M*? *N*? Both?

2. For this problem, you will implement an alternate strategy for deleting from a list and compare it to an existing deletion method. Open `ArrayList.java` and `Four2.java`. Study the implementation of deletion in the `ArrayList` class. `Four2.java` constructs an `ArrayList` of the size you specify, fills it, and then removes everything from it at position 0 so that elements must repeatedly shift down. Use `Four2.java` to time `ArrayLists` of size 1,000, 10,000, 100,000, 1,000,000. Keep this data, since you'll need it in your analysis in Part b, following.

   a) (12 pts) An alternative deletion strategy is called *lazy deletion*. To delete an element, we simply mark it as deleted (using an extra field associated with each data element). Implement lazy deletion by extending the `ArrayList` class with the `LazyArrayList` class. In order to compare the performance of `LazyArrayList` with the original `ArrayList`, you will need to keep track of the number of deleted elements as a field of the `LazyArrayList`. When there are as many deleted elements as non-deleted elements in the `ArrayList`, traverse the list performing the standard deletion algorithm on all nodes marked as deleted. In other words, when number of deleted elements equals the number of elements remaining, actually remove the deleted elements from the list. Submit your modified class electronically: `submit 303 LazyArrayList.java.`

b) (12 pts) Time your performance on the `ArrayList` that uses lazy deletion using the code in `Four2.java`. Report your results (for both `ArrayList` and `LazyArrayList` performance times) and explain your results. Of course, feel free to devise your own tests.

3. [This question is based on 4.17 in your text.] Open `Four3.java`.

a) (6 pts) Implement a recursive algorithm to compute the value of the recurrence relation:

$$\mathrm{T}(n) = \mathrm{T}(\lceil n/2 \rceil) + \mathrm{T}(\lceil n/2 \rceil) + n; \mathrm{T}(1) = 1$$

Call it `Tr(long n)` and place it in `Four3.java` where there is a stub for this method.

b) (9 pts) Now, reimplement your algorithm for part a) to use a stack to simulate the recursive calls. Call it `Ts(long n)` and place it in `Four3.java` where there is a stub for this method. You must *truly simulate* what you do in part a). Therefore, all information, including intermediate results, must be stored on the stack. If you need a local variable, use the input parameter, `n.`

Make both your implementations as efficient as possible.

Extra Credit: (18 pts) Implement $\mathrm{T}(n) = \mathrm{T}(\lceil n/2 \rceil) + \mathrm{T}(\lfloor n/2 \rfloor) + n;\ \mathrm{T}(1) = 1$ with a Stack. If you choose to do this, copy `Four3.java` to `Four3Bonus.java`, and put your implementation there as `Ts(long n)`.