Coding Quiz

- Do not open the quiz until you are directed to do so.
- Read **all** the instructions first.
- The quiz contains 7 pages, including this one.
- The quiz contains two problems. You have 120 minutes to earn 100 points.
- Read the questions very carefully. Part of the challenge here is ensuring that your properly understand the specification being implemented.
- Please submit **only** the .java files specified in the problem. Please name your files as specified. Please compress your submission into a single .zip file **containing your name**.
- The .zip file should be submitted on IVLE before the end of the session.
- Document your code well. For each algorithm, explain your solution in comments. For each function/method, explain what it is supposed to do and how it does it. At least 10% of the points will be related to your documentation.
- Write your code clearly using good Java style. Points will be deducted for badly written code.
- Be sure to test your code for correctness. A correct, working solution will always get significant credit.
- The quiz is closed book, except for one sheet of paper.
- You may also use any documentation available on the Oracle Java website, i.e., `http://download.oracle.com/javase/1.4.2/docs`. You may *not* use any other sites on the internet, your mobile phone, or any other electronic device. We may be monitoring the outgoing network connections to detect cheating.
- Read through all the problems before starting. Read the problems carefully.
- Do not spend too much time on any one problem.
- Good luck!

Problem #	**Name**	**Possible Points**	**Achieved Points**
1	Finding the Top	50	
2	Cellular Automaton	50	
Total:		100	

A Note on Cheating

- Students will be taking the coding quiz on March 10, March 11, and March 13. The Coding Quiz officially ends when announced in class on Friday March 14.
- **Any discussion of the quiz with other students prior to the announcement in class on March 14 will be considered cheating.** For example, it is considered cheating if a friend asks, "How hard was the quiz?" and you respond, "Not so bad!"
- You may not discuss any aspect of the quiz, whether it is easy or hard, whether you did well or badly, etc. And you certainly may not discuss the problems on the quiz. Even if your friend has already completed the quiz, you may not discuss the quiz with them. There is to be no discussion of the quiz until the quiz is declared over on March 14 inc lass. The rules are clearly defined so that there will be no question as to what is and is not allowed.
- It is easy to cheat and think you have committed only a minor offense. Don't. There are more important things in life than getting a slightly higher or slightly lower score on a quiz, and being an honest and upright individual is one of those. Don't be a cheater.
- There may be small differences between different versions of the quiz, and if you simply copy answers from a different version, your cheating will be clear.
- You may not leave the room with a copy of the quiz, and you may not e-mail a copy of the quiz to anyone. After the quiz ends on Thursday, you will be provided with a copy for future reference.
- Any cheating that is detected or reported will be treated harshly to the most severe extent allowed by NUS, including a zero for the entire module (not just the quiz), and referral for potential further disciplinary action (which can lead to expulsion from NUS).

Problem 1. Finding the Top

Your goal in this problem is to find the maximum item in a sorted array. Unfortunately, you do not know where the sorted sequence starts. For example, consider the following sorted array:

$$57 \;\; 63 \;\; 84 \;\; 10 \;\; 18 \;\; 23 \;\; 32 \;\; 42$$

Notice that this is, in fact, a sorted sequence starting with the fourth element, 10, and wrapping around to the beginning: $10, 18, 23, 32, 42, 57, 63, 84$. You might think of the sorted sequence as rotated in the array. For this problem, you will write a method `searchMax` that returns the largest element in such an array.

To be precise, the input to your method is an array A of n distinct items, where for some (unknown) non-negative integer value r, the sequence:

$$A[0 + r \mod n],\; A[1 + r \mod n],\; A[2 + r \mod n],\; \ldots,\; A[n - 1 + r \mod n]$$

is sorted from smallest to largest. Assume that every element in the array is distinct, i.e., there are no duplicate values.

There are two requirements for your solution to get full credit. (Partial credit will be given for solutions that do not satisfy these requirements.)

- Your method should be generic, i.e., it should find the largest element in any sorted array of items, as long as the items are `Comparable`.
- Your method should run in $O(\log n)$ time.

In addition, your method should be static. See the file `RotatedSearch.java`, which contains the following three examples of how your method should work:

```
Integer[] integers = {39, 47, 53, 3, 13, 14, 16, 18, 25, 31};
System.out.println(searchMax(integers));
// Prints: 53

Double[] doubles = {16.69, 23.89, 27.61, 33.05, 34.48, 36.63, 46.62, 5.96, 8.3, 11.44};
System.out.println(searchMax(doubles));
// Prints: 46.62

String[] names = {"Franny", "Glen", "Harry", "Isabelle", "Julia", "Alice",
                  "Bob", "Collin", "David", "Elissa"};
System.out.println(searchMax(names));
// Prints: Julia
```

Submission details. We have provided you with a file `RotatedSearch.java` that contains some test cases that may help you. Submit the file `RotatedSearch.java`, zipped (along with the solution to Problem 2) in a file containing your name.

Problem 2. Cellular Automaton

Overview. In this problem, you will implement a one-dimensional cellular automaton. (Perhaps you are already familiar with Conway's Game of Life, a two-dimensional cellular automaton.) A cellular automaton (for this problem) consists of an array of '0's and '1's. At every time step, the cells in the array are modified according to a specific rule. For example, a cellular automaton of size 11 may be represented as follows:

0 0 0 0 0 1 0 0 0 0 0

This is the initial state of the cellular automaton, all '0's, with a single '1' in the middle position. (See later for more details on initialization.)

The update rule for cell i in the array is very simple: it depends only on the value of cell $i-1$, cell i, and cell $i+1$, i.e., it depends on the value of the cell and its two neighbors. Since each of these three cells can be either '0' or '1', there are $2^3 = 8$ possible updates. Thus, a rule can be represented as an 8-bit string, with one bit specifying the outcome for each of the 8 possibilities. For example, consider the rule "**0 1 0 0 1 0 0 0**". This rule can be interpreted as follows:

rule index	0	1	2	3	4	5	6	7
left middle right	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
new middle value	0	1	0	0	1	0	0	0

If, say, cell $i-1$ of the cellular automaton is '1', cell i of the cellular automaton is '0', and cell $i+1$ of the cellular automaton is '0', then the update rule specifies that cell i should be changed to a '1'. Two important details as to how rules are applied:

- If the cellular automaton is of size n, consisting of cells $C[0], C[1], \ldots, C[n-1]$, we will assume that there are two special cells $C[-1] = 0$ and $C[n] = 0$ that are permanently fixed to 0 and that never change. (This ensures that the rule can be applied consistently to cell 0 and to cell $n-1$.)
- When the cellular automaton is updated, all the new values are calculated based on the old values, i.e., even if you have already calculate a new value for cell $i-1$, when you calculate the new value for cell i, you must use the old value. Be careful not to overwrite (and lose) the old values as you calculate the new values.

As an example, consider the following cellular automaton of size 11 (with the dummy entries for -1 and n draw in gray), executing the rule "**0 1 0 0 1 0 0 0**".

0 0 1 1 1 0 1 0 0 1 0 0 0

The result is:

0 1 0 0 0 0 0 1 1 0 1 0 0

Details. Your solution should implement the `ICA` interface provided with this problem. There are two methods in the interface:

- `initialize(int[] rule)`: This method takes an array that consists of 8 digits, each of which is either '0' or '1'. (If there is any other value in the rule, or if it is not 8 digits long, then it is an error.) This defines the rule for the cellular automaton. E.g., `rule[5]` specifies the new value for cell i if cell $i-1$ equals '1', cell i equals '0', and cell $i+1$ equals '1'.

  Whenever the cellular automaton is initialized, it is reset to its initial state. In its initial state, all the cells are '0', except for the middle cell which is '1'. For example, in a cellular automaton of size 11, the initial state would look like:

  0 0 0 0 0 0 1 0 0 0 0 0 0

  If the size is even, then initially set both the "middle" cells to '1', i.e.,:

  0 0 0 0 0 1 1 0 0 0 0 0

  If the cellular automaton has not been initialized, then any attempt to execute it is an error.

- `step()`: This method executes the rule that was provided during initialization. (If the cellular automaton has not been initialized, then this is an error.) An implementation of this method might create a temporary array, set the value of each entry in the temporary array (based on the existing state of the automaton and the rule), and update the state of the automaton.

Your class should also have a constructor that takes one parameter: the size of the cellular automaton. For example, if you class is called `CA`, then it should have a constructor `CA(int size)`.

Finally, the `ICA` interfaces extends `Iterable<String>` and hence your class must also implement the `Iterable<String>` interface. Your class should return an iterator that can be used to iterate through the different steps of the cellular automaton. The first time the iterator is executed, it should return the initial state of the cellular automaton, represented as a string. Each time after that you call `next()` on the iterator, it should return a string representing the next state of the cellular automaton (i.e., after the rule is executed).

Example. Consider the following example:

```
// Create a new cellular automaton of size 31
CA example = new CA(31);

// Create a new rule
int[] rule = {0,1,0,0,1,0,0,0};

// Initialize the CA with the rule
example.initialize(rule);

// Get an iterator
Iterator<String> iterator = example.iterator();
```

```
    // Run the cellular automaton for 17 steps
    for (int i=0; i<17; i++){
      if (iterator.hasNext()){
        String s = iterator.next();
        System.out.println(s);
      }
    }
```

This will create the following output:

```
0000000000000001000000000000000
0000000000000010100000000000000
0000000000000100010000000000000
0000000000001010101000000000000
0000000000010000000100000000000
0000000000101000001010000000000
0000000001000100010001000000000
0000000010101010101010100000000
0000000100000000000000010000000
0000001010000000000000101000000
0000010001000000000001000100000
0000101010100000000010101010000
0001000000010000000100000001000
0010100000101000001010000010100
0100010001000100010001000100010
1010101010101010101010101010101
0000000000000000000000000000000
```

Notice that this is easier to see if the ‘0’s are replaced with spaces, as follows:

```
               1
              1 1
             1   1
            1 1 1 1
           1       1
          1 1     1 1
         1   1   1   1
        1 1 1 1 1 1 1 1
       1               1
      1 1             1 1
     1   1           1   1
    1 1 1 1         1 1 1 1
   1       1       1       1
  1 1     1 1     1 1     1 1
 1   1   1   1   1   1   1   1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Submission details. We have provided you with a file `CA.java` that contains some template code which may help you. Be aware the template makes some specific design decisions (e.g., storing the state in an array of integers, and using the first and last entry of the array to represent the permanently-zero boundaries of the automaton). (You are not required to follow this template, but it is intended to make things easier for you.) We have also provided the interface file `ICA.java` which contains the interface. Submit the file `CA.java`, zipped (along with the solution to Problem 2) in a file containing your name.

Just for fun, another interesting pattern is generated by the rule: "**0 1 1 1 1 0 0 0**".