CS 367: Data Structures **Summer 2011**
Exam 1: Lists, Stacks, Queues, Recursion, Searching

You have 75 minutes to complete this exam.

You are allowed nothing except a writing utensil and a 3" x 5" index card, which MUST BE TURNED IN WITH YOUR EXAM.

Be sure to show your work and explain all your answers. Partial credit may be awarded for incorrect answers with thorough explanations, and a correct answer with no work or explanation may be awarded little or no credit.

If you are asked to write code but do not know the exact Java syntax required, approximate as best you can. If you feel your code is not enough to demonstrate your understanding of the problem, describe your algorithm in English as well. Don't make me guess at what you're thinking.

BE SURE TO WRITE YOUR NAME ON THIS PAGE.

1. Perform a radix sort on the following list of numbers. Be sure to show the bins (queues) at each step of the algorithm: `[312, 8, 21, 16, 201, 1007, 9, 2851, 1]`

ANSWER:

One's digit:
0:
1: 21, 201, 2851, 1
2: 312
3:
4:
5:
6: 16
7:
8: 8
9: 9

[21, 201, 2851, 1, 312, 16, 1007, 8, 9]

Ten's digit:
0: 201, 1, 1007, 8, 9
1: 312, 16
2: 21
3:
4:
5: 2851
6:
7:
8:
9:

[201, 1, 1007, 8, 9, 312, 16, 21, 2851]

Hundred's digit:
0: 1, 1007, 8, 9, 16, 21
1:
2: 201
3: 312
4:
5:
6:
7:
8: 2851
9:

[1, 1007, 8, 9, 16, 21, 201, 312, 2851]

Continued on next page...

Thousand's digit:

0: 1, 8, 9, 16, 21, 201, 312
1: 1007
2: 2851
3:
4:
5:
6:
7:
8:
9:

[1, 8, 9, 16, 21, 201, 312, 1007, 2851]

After this, if we kept track of the largest integer in our array, we know we can stop. Otherwise, if we perform one more round, we will see that all of the numbers go into the 0th bin.

2. Consider the following implementation of a list. My class contains three fields: a `head` variable, which anchors the linked list; a `count` variable which keeps track of the length n of the list; and a `shortcuts` variable which anchors a second linked list of length $\sqrt{n}$. (For simplicity, you may assume that n is a perfect square, so $\sqrt{n}$ is an integer.)
Each node in this second linked list contains as its data a pointer to a node in the linked list anchored at `head`. These shortcut pointers are evenly spaced throughout the big linked list. For example, if $n = 100$, then the `shortcuts` list will have 10 nodes, pointing to nodes 0, 10, 20, ..., 80, 90 of the big linked list. Using whatever combination of words, pictures, and code that you need to get your point accross, describe how you would implement the `get()` method of this new data structure. Also explain what this method's runtime would be.

ANSWER:

Here is a rough implementation of the `get()` method. You could also have shown this with pictures, pseudocode, etc.

```
 public E get(int pos) {
   Node current = shortcuts;
   int count = 0;
   while (count + Math.sqrt(n) < pos) {
      current = current.getNext();
      count += Math.sqrt(n);
   }
   current = current.getData();
   while (count < pos) {
      current = current.getNext();
      count++;
   }
   return current.getData();
}
```

This will have a runtime of $O(\sqrt{n})$, where n is the length of the bigger linked list. This is because each loop can run a maximum of $\sqrt{n}$ times. The first loop can at most traverse through each of its $\sqrt{n}$ elements. The second loop will only ever have a subsection of the big list of length $\sqrt{n}$ to traverse through as well—we would never need to traverse any further, because otherwise, we'd have skipped one more ahead in the `shortcuts` list. All of the other operations are constant time, so we have $O(\sqrt{n} + \sqrt{n}) = O(\sqrt{n})$.

3. In our implementation of the GCD function, we used the modular operator % which computes the remainder when dividing two integers. For this problem, you will write this function yourself. Your method will take two integers, `a` and `b`, and return their remainder. One way to do this would be to repeatedly subtract `b` until you get to the remainder—though of course, it is important to know when to stop! The method signature looks like this:

```
int remainder(int a, int b) {...}
```

Write a recursive version of this method:

ANSWER:

```
 int remainder(int a, int b) {
   if (a < b) return a;
   return remainder(a-b, b);
}
```

Write an iterative version of this method:

ANSWER:

```
 int remainder(int a, int b) {
   while (a >= b) {
      a -= b;
   }
   return a;
}
```

Describe the runtime of either of your methods. (*Hint: it will depend on a and b.*)

ANSWER:

The recursive method will have the same number of recursive calls as the iterative method has iterations of the while-loop: namely, a/b. There are nothing but constant-time operations within each recursive call or loop iteration, so this makes our run-time $O(a/b)$.

4. In this problem, you will implement a method that takes two pieces of input: a *sorted* array of integers arr and an integer v. It returns a boolean value—true if v is contained in arr and false otherwise. Your implementation should have a runtime of $O(\log n)$, where n is the length of arr. Explain why your method has this runtime. You may assume that arr and v are not null, that arr contains no null values, and that arr is sorted.

ANSWER:

```
 public static boolean arrayContains(int[] arr, int v) {
   int min = 0;
   int max = arr.length - 1;
   int mid = max + min / 2;
   while (max - min > 1) {
      if (arr[mid] == v) {
         return true;
      } else if (arr[mid] < v) {
         min = mid;
         mid = (min + max) / 2;
      } else {
         max = mid;
         mid = (min + max) / 2;
      }
   }
   return false;
}
```

Runtime justification: This method just implements binary search. It has a runtime of $O(\log n)$ because we are halving the size of our search space with each iteration of the loop. We can only halve the space $\log_2 n$ times. The rest of the operations are all $O(1)$, so we are looking at $O(\log_2 n) = O(\log n)$.