Name: ______________________________________

USC loginid (e.g., ttrojan):____________________________

CS 455 Final Exam
Fall 2011 [Bono]

December 12, 2011

There are 3 problems on the exam, with 68 points total available. There are 7 pages to the exam, including this one; make sure you have all of them. There is also a separate double-sided one-page code handout. If you need additional space to write any answers, you may use the backs of exam pages (just direct us to look there).

Note: if you give multiple answers for a problem, we will only grade the first one. Avoid this issue by labeling and **circling** your final answers and crossing out any other answers you changed your mind about (though it's fine if you show your work).

Put your name and USC loginid at the top of the exam. Please read over the whole test before beginning. Good luck!

Remote DEN students only: Do not write on the backs of pages. If additional space is needed, ask proctor for additional blank page(s), put your name on them, and attach them to the exam.

	value	**score**
Problem 1	17 pts.	
Problem 2	15 pts.	
Problem 3A	9 pts.	
Problem 3B	20 pts.	
Problem 3CD	7 pts.	
TOTAL	68 pts.	

Problem 1 [17 pts. total]

Part A (15 pts.) Implement the **Java** method `fillArray`, such that a call to `fillArray(n)` returns an array filled with the following sequence (complete array contents shown on multiple lines to emphasize structure of contents):

```
[1,
 1, 2,
 1, 2, 3,
 .
 .
 .
 1, 2, . . . , (n-1), n]
```

Some additional examples:

n	return value of `fillArray(n)`
0	*[]*
1	*[1]*
2	*[1, 1, 2]*
4	*[1, 1, 2, 1, 2, 3, 1, 2, 3, 4]*

Hint: the sum of the numbers from `1` to `n` is exactly `n*(n+1)/2`.

```
// PRE: n > 0
public int[]  fillArray(int n) {
```

Part B (2 pts). Give the worst-case big-O time for your method:

Problem 2 [15 pts. total]

In lecture and a lab we discussed Java code to create a concordance, that is, the number of occurrences of each distinct word in a file. Here we're going to make a version of our Concord class that creates a concordance that computes the *locations* of each distinct word in a file. We will use the line number as a word's location.

Here is an example of some input, and the corresponding output when we build a concordance for it and print it (just using toString-type format):

```
the big dog went to the big dog
and the other big dog went to the
the big elephant

{and=[2], big=[1, 2, 3], dog=[1, 2], elephant=[3], other=[2], the=[1, 2, 3],
to=[1, 2], went=[1, 2]}
```

Note that even if a word appears more than once on a line, we only list that line number once. Some more details of the problem:

- You don't have to worry about putting words into some canonical form for the purposes of this problem (e.g., you may assume there is no punctuation or capital letters in the input).
- For a file with *n* words your code must build the concordance in at worst *nlogn* time, and be able to print it out in alphabetical order in linear time (you are not required to write the toString code).
- It is expected that you will use the Java library to make your code short and fast. See the code handout for a reminder of some classes and methods.
- We have written the code to read the file for you below. You need to fill in the private data, complete the constructor, and complete `build` by implementing the private method `processLine.`

Write your answer in the spaces provided on the next page.

Problem 2 (cont.)

```
public class Concord {
   // put private data here:





   // creates an empty concordance
   public Concord() {




   }

   // returns string version of concordance in alphabetical order by word.
   // for each word, it prints all the line numbers it occurs on.
   public String toString() {
       // you don't have to write this (code not shown)
   }

   // builds the concordance from data read from "in"
   public void build(Scanner in) {

     int currLineNum =  1;

     while (in.hasNext()) {
        String currLine = in.nextLine();
        processLine(currLine, currLineNum);  // call helper function below
        currLineNum++;
     }
   }

   // (helper func) adds all the words in this line to the concordance
   // as occurring on lineNum
   private void processLine(String line, int lineNum) {















   }
}
```

Problem 3 [36 pts. total]

Note: there are four parts to this problem over this and the following two pages. Part B is to implement the function described; but there are two shorter parts that come after that; you can do them in pretty much any order.

Consider the **C++** function `removeKAtLoc` ("remove k at loc") which removes the `k` elements starting at location `loc` in the given linked list. You may assume the location numbers start at 0, so for a list with 5 elements, `removeKAtLoc(list, 2, 2)`, would remove the third and fourth elements in the list (i.e., the ones numbered 2 and 3); that is the first example shown below. *Note: to make this problem simpler, you may assume that the function never removes the first element in the list – this is reflected in the precondition shown on the next page.* It's legal to pass in a `loc` or `k` that would go off the end of the list: see examples below for behavior:

`list` before call	`loc`	`k`	`list` after call to `removeKAtLoc(list, loc, k)`
(0 1 2 3 4)	*2*	*2*	*(0 1 4)*
(2 6 7 5)	*1*	*2*	*(2 5)*
(2 6 7 5)	*1*	*3*	*(2)*
(2 6 7 5)	*1*	*43*	*(2)*
(2 6 7 5)	*4*	*2*	*(2 6 7 5)*
(2)	*1*	*2*	*(2)*
(2 6)	*1*	*0*	*(2 6)*

Part A (9 pts). Come up with a thorough set of test cases for this function, as you would use for unit testing. We don't want to see the exact values for the test cases. *What is required is to describe the category of each test you would need to do.* We'll give one such case here as an example (len refers to the length of the list):

-- *list with len > 1, loc > 1 and < (len-1), k > 1 and < (len-loc)*

Problem 3 (cont.)

Part B (20 pts). Implement the function. (Reminder: this is a C++ function.) For full credit, reclaim the memory used by the removed nodes. See code handout for `Node` type.

```
// Remove the k elements starting at location loc from the list.
// Location numbers start from 0 (like array indices in C++ and Java)
// PRE: list is a well-formed linked list with at least one element
//      loc > 0
//      k >= 0
void removeKAtLoc(Node * & list, int loc, int k) {
```

(problem continued on next page)

Problem 3 (cont.)

Part C (2 pts). Give the worst case big-O time for the implementation you wrote for part B (you can get credit for this part even if you didn't complete part B):

Part D (5 pts). For this function and the way it's specified we don't actually have to define it to pass the `Node*` by reference, but we did so in part to signal to the caller that it modifies the list. Below is an example of an alternate header where we don't pass it by reference.

```
void removeKAtLocV2(Node * list, int loc, int k)
```

i. Why can we do it either way (i.e., passing the `Node*` by value or by reference)?

ii. To illustrate the situation when we *don't* pass the first parameter by reference, draw a box-and-pointer diagram for an example call to `removeKAtLocV2` and its results.