Name: ______________________________________

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

CS 455 Final Exam
Fall 2012 [Bono]

Dec. 17, 2012

There are 6 problems on the exam, with 70 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 two-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!

	value	**score**
Problem 1	6 pts.	
Problem 2	10 pts.	
Problem 3	15 pts.	
Problem 4	6 pts.	
Problem 5	13 pts.	
Problem 6	20 pts.	
TOTAL	70 pts.	

Problem 1 [6 pts. total]

Consider the following **Java** method computes the sum of the elements in a `LinkedList` object:

```
public static int sumList(LinkedList<Integer> list) {
   int sum = 0;
   for (int i = 0; i < list.size(); i++) {
      sum += list.get(i);
   }
   return sum;
}
```

This problem concerns the *implementation* of `sumList` – assume we cannot change the interface to the function.

Part A (4). The code works, but is written in a bad way. What's the problem with the code? (Be specific.)

Part B (2). Describe how we can improve the code, and how your improvement helps. (You should not rewrite the function; just write one sentence or so.)

Problem 2 [10 pts]

Part A (8). Implement the boolean **Java** method `sortByScore`, which sorts an `ArrayList` of students so they are in decreasing order by score. *You must use the Java sort utility* (more info about that on the code handout), and include any additional code necessary to make it work (outside of the `sortByScore` method).

Here is the `Student` class it uses (only shows interface, since that's all we are using here). This is an already completed class; *you may not modify* `Student` *for this problem.*

```
public class Student implements Comparable<Student> {

   public Student(String name, int score) { . . . }

   public String getName() { . . . }

   public int getScore() { . . . }

   public void changeScore(int newScore) { . . . }

   // compares this student with b by name: if this student's name comes before
   // b's name returns a negative value, positive if it comes after, and 0 if
   // they are equal
   public int compareTo(Student b) { . . . }

   . . .  // private stuff not shown

}


public static void sortByScore(ArrayList<Student> students) {
```

Part B (2). Using big-O notation give the worst-case time for your method as a function of, *n*, the size of the array list.

Problem 3 [15 pts]

We can think of a Map as describing a function that maps from keys to values. Write the **Java** boolean function `isOneToOne` which tells if its `Map` argument is a one-to-one function, that is, whether no two *(key, value)* entries have the same *value* part. Here are some examples (maps shown as they would be by a call to `toString`):

map	**isOneToOne(map)**	
{joe=3, sally=2, sam=3}	*false*	*(both* `joe` *and* `sam` *have the value* 3*)*
{joe=3, sally=2, sam=7}	*true*	*(no two keys have the same value)*
{}	*true*	*(you may assume an empty map is one-to-one)*

To receive full credit your function must run in at most O(*n*) time, where *n* is the number of entries in the map (Hint: take advantage of the Java library).

```
public static boolean isOneToOne(Map<String, Integer> map) {
```

Problem 4 [6 pts. total]

Assume we are trying to compile the **C++** `grades` program we did for assignment 5. Here's a reminder of the file organization:

`Table.h`	Header file for the `Table` class (contains `Table` class definition)
`Table.cpp`	Implementation file for the `Table` class (contains `Table` method definitions)
`grades.cpp`	A program that uses the `Table` class (contains `main`)

Consider the following non-ideal `Makefile` for the program (compile commands are numbered at the right for your convenience):

```
grades: grades.o Table.o

      g++ -ggdb -Wall grades.o Table.o -o grades                  (1)

Table.o: Table.cpp Table.h

      g++ -ggdb -Wall -c Table.cpp                                (2)

grades.o: grades.cpp Table.cpp Table.h

      g++ -ggdb -Wall -c grades.cpp                               (3)
```

Suppose we have already compiled our `grades` program successfully once using this `Makefile`. Give a scenario such that the `Makefile` given will do more work than necessary to create the updated executable for our program the next time we call gmake.

a. File(s) modified in your scenario:

b. Step(s) done by the `Makefile` above on the next call to the

```
gmake grades
```

command: (Note: you may use the numbers shown above; numbers must be given in a valid order)

c. Minimum step(s) that *would have been* necessary to create the updated executable for our program under this scenario: (Note: you may use the numbers shown above; numbers must be given in a valid order):

Problem 5 [13 pts.]

Consider the following **C++** program:
(Note: the `Node` class is on code handout.)

```
void swap(Node * a, Node * b) {
   Node * tmp = a;
   a = b;
   b = tmp;
   cout << a->data << " " << b->data << endl;
}

int main() {
   Node * x = new Node(3);
   Node * y = new Node(8);
   cout << x->data << " " << y->data << endl;
   swap(x, y);
   cout << x->data << " " << y->data << endl;
   return 0;
}
```

Part A [7]. In the space below, draw a box-and-pointer diagram (a.k.a., memory diagram) showing all variables, objects, and their state as they have changed during the code sequence. This includes showing all of `swap`'s variables and parameters.

Part B [6]. In the space below show the output of the program:

Problem 6 [20 pts]

Implement the **C++** function `removeEveryOther` which removes every other element from a linked list of `int`s starting with the first element. *For full credit you need to reclaim memory no longer used.* The `Node` and `ListType` definitions are on the code handout.

`list` before call	**`list`** after call to **`removeEveryOther(list)`**
(1 2 3)	*(2)*
(7 6 7 5 7 8)	*(6 5 8)*
(1 2 3 4)	*(2 4)*
(1 2 3 4 5)	*(2 4)*
()	*()*
(3)	*()*

```
// PRE: list is a well-formed linked list
void removeEveryOther(ListType & list) {
```