Computer Science 201
Fall 2012
Midterm #1

Your Name: _____

Your NetID: _____

Sign on the line below to confirm that you have completed this test in accordance with the Duke Community Standard.

This is a 75-minute, 75-point test: each point should take about one minute. Perfect Java syntax is not required: get your point across without worrying about perfect semicolon, curly-brace, and indentation hygiene. It should still look like Java: pseudocode and English aren't good enough. *Write carefully: if we can't read it, we won't grade it.*

Question	Possible	Score
0	16	
1	6	
2	6	
3	14	
4	6	
5	7	
6	10	
7	10	
Total	75	

0 .compareTo, .equals, .hashCode (16 points total)

You are given a `String[]` of names and a corresponding `int[]` of ages. (That is, `names[i]` is `ages[i]` years old.) You would like to sort these people such that long names are before short names, breaking ties such that younger people are before older people. You will do this by implementing the `LongNameFirst` class, which is used by `buildAndSort`, below, to sort these arrays. *Note that you do not modify `buildAndSort`.* For example, if we ran this code

```
String[] names = {"Mark", "Sam", "Erin", "Haley", "Connor"};
int[] ages = {17, 12, 10, 8, 2};
ArrayList<LongNameFirst> sorted = buildAndSort(names, ages);
```

and then printed out the names in the variable `sorted`, we would get

```
"Connor" "Haley" "Erin" "Mark" "Sam"
```

The `buildAndSort` method looks like this:

```
public ArrayList<LongNameFirst> buildAndSort(String[] names, int[] ages)
{
  ArrayList<LongNameFirst> nameList = new ArrayList<LongNameFirst>();
  for(int i = 0; i < names.length; i++) {
    nameList.add(new LongNameFirst(names[i], ages[i]));
  }
  Collections.sort(nameList);
  return nameList;
}
```

0.1 Instance variables (3 points)

Declare the instance variables for `LongNameFirst`.

```
class LongNameFirst implements Comparable<LongNameFirst>{
  // TODO: Declare instance variables
```

0.2 Constructor (3 points)

Write the constructor for `LongNameFirst`.

```
public LongNameFirst(                                    ) {




}
```

Question continues on next page.

0.3 compareTo (5 points)

Write the `compareTo` method for `LongNameFirst`.

```
public int compareTo(LongNameFirst other) {









}
```

0.4 hashCode (3 points)

Write a non-trivial `hashCode` method (and any helper methods) for `LongNameFirst`.

```
public int hashCode() {




}
```

0.5 equals (2 points)

Write the `equals` method for `LongNameFirst`.

```
public boolean equals(Object other) {
  if (other == null) {
    return false;
  }
  LongNameFirst lnf = (LongNameFirst)other;
  // TODO: Your code starts here.




}
```

1 Syntax (6 points total)

For each of the following, circle the choice that compiles.

1.1 Types (2 points)

`int x = "Hello";`	`int x = 17.4;`
`char x = "Robots!";`	`int x = 42;`

1.2 Return values (2 points)

`void doSomething() {`
`  System.out.println("Go!");`
`}`
	`void doSomething() {`
	`  return 17;`
	`}`

`int doSomething(double d) {`
`  return d * 2;`
`}`
	`String doSomething() {`
	`  System.out.println("Stop!");`
	`}`

1.3 Loops (2 points)

`String[] v = {"Hamlet",`
`  "MacBeth",`
`  "Othello"};`
`for (int i = 0, i < v.length, ++i) {`
`  System.out.println(v[i]);`
`}`
	`String[] v = {"Hamlet",`
	`  "MacBeth",`
	`  "Othello"};`
	`for (String i : v) {`
	`  System.out.println(i);`
	`}`

`String[] v = {"Hamlet",`
`  "MacBeth",`
`  "Othello"};`
`for (String i : v) {`
`  System.out.println(v[i]);`
`}`
	`String[] v = {"Hamlet",`
	`  "MacBeth",`
	`  "Othello"};`
	`for (String i = 0; i < v.length; ++i) {`
	`  System.out.println(v[i]);`
	`}`

2 Code output (6 points total)

What is the output of the following code?

2.1 (3 points)

```
public static void main(String[] args){
    int[] aArray = {1, 2, 3, 4, 5};
    int[] bArray = {6, 7, 8, 9, 10};
    int[] cArray = aArray;

    aArray[3] = aArray[3] + 1;
    bArray[3] = bArray[3] - 1;
    cArray[3] = cArray[3] + 1;

    System.out.println(aArray[3]);
    System.out.println(bArray[3]);
    System.out.println(cArray[3]);
}
```

2.2 (3 points)

```
public static void doSomething(int[] someInts)
{
  for(int i = 0; i < someInts.length; i++) {
    someInts[i] = someInts[i] + i * 2;
  }
}


public static void main(String[] args)
{
  int[] dArray = {10, 8, 6};
  doSomething(dArray);

  for(int i = 0; i < dArray.length; ++i)
  {
    System.out.print(dArray[i] + " ");
  }
}
```

3 Big-O (14 points total)

3.1 Running Times (12 points; 2 each)

Provide the Big-O running time of each of the following methods, and *briefly* justify your answer. Every answer is one of $O(\log n)$, $O(n)$, $O(1)$, $O(n^2)$, of $O(\sqrt{n})$.

A (2 points)

```
public String numberZero(int n) {
  return "Alpha";
}
```

B (2 points)

```
public String[] numberOne(int n) {
  String[] result = new String[n];
  for (int i = 0 ; i < n ; i += 2) {
    result[i] = "Bravo";
  }
  return result;
}
```

C (2 points)

```
public int numberTwo(int n) {
  int k = 0;
  for (int i = 0 ; i < n ; ++i) {
    for (int j = 0 ; j < n ; j += 2) {
      k += 2;
    }
  }
  return k;
}
```

D (2 points)

```
public int numberThree(int n) {
  int total = 0;
  for (int i = 0 ; i < n ; ++i) {
    for (int j = 0 ; j < n ; j += 2) {
      total += 1;
    }
  }
  for (int i = 0 ; i < 10 * n ; ++i) {
    total += 1;
  }
  return total;
}
```

Question continues on the next page.

E (2 points)

```
public void numberFour(int n) {
  int now = 1;
  while (now * now < n) {
    now++;
  }
}
```

F (2 points)

```
public int numberFive(int n) {
  int foo = 17;
  for (int i = 1 ; i < n ; i *= 2) {
    foo *= 3;
  }
  return foo;
}
```

3.2 Ordering (2 points)

Order methods A–F from fastest-running to slowest-running (that is, best to worst). There may be ties.

4 Tradeoffs (6 points)

In statistics, the *mode* of a list is the element that occurs most often in that list. Your boss has asked for an implementation of mode to be used on very large inputs (hundreds of thousands of elements). Your coworker has produced `modeOne`; you produced `modeTwo`. Both correctly compute the mode, but you are confident that your boss should prefer your implementation. Explain why.

A

```
int modeOne(int[] values) {
  int bestCount = 0;
  int bestIdx = 0;
  for (int i = 0 ; i < values.length ; ++i) {
    int count = 0;
    for (int j = 0 ; j < values.length ; ++j) {
      if (values[j] == values[i]) {
        count++;
      }
    }
    if (count > bestCount) {
      bestCount = count;
      bestIdx = i;
    }
  }
  return values[bestIdx];
}
```

B

```
int modeTwo(int[] values) {
  HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
  for (int i = 0 ; i < values.length ; ++i) {
    if (!map.containsKey(values[i])) {
      map.put(values[i], 0);
    }
    int old = map.get(values[i]);
    map.put(values[i], old + 1);
  }
  int best = 0;
  int bestVal = 0;
  for (int i : map.keySet()) {
    if (map.get(i) > best) {
      best = map.get(i);
      bestVal = i;
    }
  }
  return bestVal;
}
```

---

5 Number of Wins (7 points)

Complete the method `findChampion` below. This method takes in a `String[]` where each `String` represents the score of a single game, and returns a single `String` containing the team that won the most games (you may assume that the champion is unique). The `Strings` are in this format:

```
"DUKE UNC 85 84"
```

which means that Duke scored 85 points and UNC scored 84. The provided `findWinner` method, below, computes the winner of a single game.

```
// Provided method. See findChampion, below.
public String findWinner(String game) {
  String[] tokens = game.split(" ");  // Divide up the String
  String winner = tokens[0];
  // See if the second team had a higher score than the first
  if (Integer.parseInt(tokens[3]) > Integer.parseInt(tokens[2])) {
    winner = tokens[1];
  }
  return winner;
}

// TODO: Complete this method!
public String findChampion(String[] games) {
```

6 Duplicates (10 points total)

You've just been hired by Very Small Cell Phones Incorporated, and given your first programming task: writing a method to find the unique values in a list of `Strings` in $O(N)$ time. However, because it's for a phone, the version of Java you'll be using is incomplete: it has `ArrayList`, but no `Map` or `Set` classes. However, you have one advantage: you've been told that your input will be in sorted order! This means that duplicate elements will be adjacent in the input.

Complete the method `computeUniqueWords`, below. You should not remove any of the provided code; only add to it. The argument `words` will have at least one element, and will be in sorted order. The returned `ArrayList<String>` should also be in sorted order, and should contain the `Strings` from `words`, but without duplicates. The method must run in $O(N)$ time, where N is the number of `Strings` in `words`. You may not use any `Set` or `Map` classes.

```
public ArrayList<String> computeUniqueWords(ArrayList<String> words) {
  ArrayList<String> result = new ArrayList<String>();
  result.add(words.get(0));
  for (int i = 1 ; i < words.size() ; ++i) {









  return result;
}
```

7 Recursion (10 points)

Keeping track of parentheses and braces can be a pain in the neck, but when writing a compiler, it's an absolute necessity: they must match! Below, you will complete the recursive method `areMatched`, which checks to see if a `String` contains *only* nested, matched pairs of parentheses and braces ('{' and '}'). Here are some example input-output pairs:

```
areMatched("")          →   true
areMatched("(())")      →   true
areMatched("{()}")      →   true
areMatched("{({z})}")   →   false   Not just paretheses and braces.
areMatched("((((()))")  →   false   Not matched.
```

```
boolean areMatched(String s) {

  // Base case #1
  if (s.length() == 0) {
    // TODO: Return the correct value to complete this base case.



  }

  // Base case #2
  if (s.length() == 1) {
    // TODO: Return the correct value to complete this base case.




  }

  // TODO: check to see if the first and last characters of s match.

















  // TODO: Make the recursive call and complete the method.







}
```