CSC 148H1S, Winter 2005

Section L5101

Midterm test

No aids allowed.

Time allowed: 50 minutes.

ANSWERS

1	
2	
3	
Total	

Answer all questions on this test paper. Do not separate the pages of the test.

1. [10 marks]

A doubly-linked list is being used to store Objects. If all the Objects are Strings, the list might look like this:

Notice that there is a "head" variable, but there is no "tail". The last node's "next" pointer and the first node's "prev" pointer are both null. Here's what the class looks like:

```
public class DoubleList {
  private class Node {
    private Object data; // visible inside DoubleList in spite of "private"
    private Node next, prev;
  }
  private Node head;
  // Omitted: constructor, methods such as insert(), etc.

  // printForward() prints all the data in the list from the head to the
  // tail. It is included to remind you of the list-printing pattern.
  // But be careful when using it as a model!
  public void printForward() {
    Node p;
    for (p = head; p != null; p = p.next) {
      System.out.println(p.data);
    }
  }
  ... Your method goes here.
}
```

Write a non-recursive method `printAlternatelyBackward()` that prints *every other* item in the list, *ending with the first item*. For example, with the list sketched above, the output would be:

```
coat
hat
my
```

If there were only four items in the list, for example "my", "red", hat" and "and", then the output would be:

```
hat
my
```

There is space on the next page to write your answer.

1. (continued)

Here is the method header to get you started:

```
public void printAlternatelyBackward () {

  if (head == null) {
    return;
  }
  Node p;
  int count = 0;
  for (p = head; p.next != null; p = p.next) {
    count += 1;
  }
  // Now count is 1 less than the number of items in the list,
  // because we didn't increment it at the last list item.
  // If count+1 is odd, we should print the last item, otherwise
  // not.
  if (count % 2 == 1) { // Don't print the last item.
    p = p.prev;
  }
  while (p != null) {
    System.out.println(p.data);
    // Skip back two places to get the next thing to print.
    p = p.prev;
    if (p != null) {
      p = p.prev;
    }
  }
}
```

2. [10 marks]

Here is a method that finds the median of an unsorted array of integers. (We define the median as the element bigger than half the other elements. That definition is not always reliable, but we assume that it works here.)

```
public static int median (int[] array) {

  // Any element could be the median. We check them all.
  for (int candidate = 0; candidate < array.length; candidate++) {
    // Count the smaller elements of array.
    int count = 0;
    for (int other = 0; other < array.length; other++) {
      if (array[other] < array[candidate]) {
        count++;
      }
    }
    if (count == array.length/2) {
      return array[candidate];
    }
  }
  return 0; // just to keep the compiler happy
}
```

If the number of elements in the array is *N*, what is the complexity of `median()`? (In other words, what big-O function of *N* describes how long `median()` takes to execute?)

Hint: You can ignore the effect of the "return" statement inside the loop on the number of loop iterations.

The actual answer is worth 1 mark. To get the rest of the marks for this question, you must show how you derived the answer.

Note that array.length = N.

The inner for loop:
  Each execution of the loop body takes O(1) time.
  There are N iterations of this inner loop (for each execution of the outer loop body).
  The cost of the inner loop is therefore N*O(1) = O(N).

The outer for loop:
  Each execution of the loop body takes O(1) + O(N) + O(1) time, which is O(N) provided N isn't 0. (You don't have to say that "provided" part.)
  There may be up to N iterations of the outer loop.
  (There may be fewer, because of the "return array[candidate];" statement, but we're told not to worry about that.)
  Therefore the cost of the outer loop is N*O(N), or O(N^2).

The final return statement would cost O(1) if it were ever executed, which it isn't. So the total cost of the method is O(N^2) + O(1), which is O(N^2).

3. [10 marks = 5 + 5]

(a) The method `countEs()`, begun below, is intended to return the number of upper-case Es in a string.

Complete the method, ***using recursion***.

```
public static int countEs (String s) {

  if (s.length() == 0) {
    return 0;
  }
  else {
    int restEs = countEs(s.substring(1));
    if (s.charAt(0) == 'E') {
      return restEs + 1;
    }
    else {
      return restEs;
    }
  }
}
```

3. (continued)

(b) The method `secondLargest()`, begun below, returns the second-largest element in an array of integers.

(The "second-largest" is the element that is bigger than all the other elements except one; equivalently, it is the element that is smaller than exactly one other element. To ensure that this definition works, you may assume that there are at least two elements in the array, and that all the array elements are different. For example, the array might have elements 4, 5, 2, 9, and –13; then the second-largest element is 5, which is smaller than 9 but larger than all the other elements.)

Complete `secondLargest`, ***using recursion***. Hint: you need at least one helper method.

```
public static int secondLargest (int[] array) {

  return findFromI(array, 0);
}

private static int findFromI(int[] array, int start) {
  if (countBigger(array, start, 0) == 1) {
    return array[start];
  }
  else {
    return findFromI(array, start + 1);
  }
}

private static int countBigger(int[] array, int start, int compareAt) {
  if (compareAt == array.length) {
    return 0;
  }
  else if (array[compareAt] > array[start]) {
    return 1 + countBigger(array, start, compareAt + 1);
  }
  else {
    return countBigger(array, start, compareAt + 1);
  }
}
```