CSC 148H1S, Winter 2006

Section L5101

Midterm test

No aids allowed.

Time allowed: 50 minutes.

ANSWERS

1	
2	
3	
4	
5	
Total	

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

1. [5 marks] *This is the only question worth 5 marks; the rest are each worth 10.*

Here are the declaration and initialization of a two-dimensional array `a`:

```
int[][] a;
... // a receives values here
```

The array `a` is not necessarily rectangular.

In the space below, write a few lines of code (not a complete method) to print the number of elements in `a`. Print to the standard output with `System.out`.

```
int count = 0;
for (int i = 0; i < a.length; i++)
  count += a[i].length;
System.out.println(count);
```

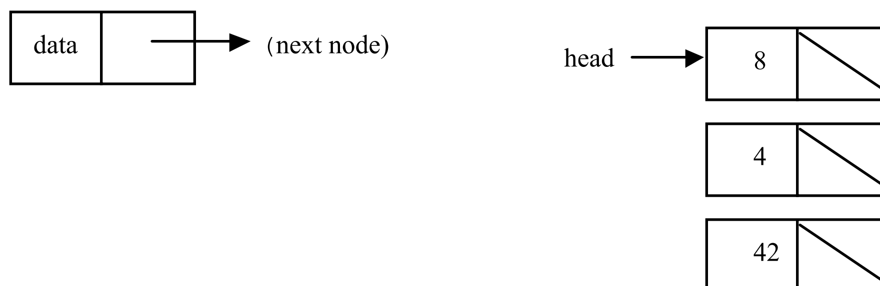
2. [10 marks = 7 + 3]

Consider this class:

```
public class Links {
  private static class Node {
    private int data;
    private Node next;
    private Node(int data) { this.data = data; }
  }
  public static void main(String[] args) {
    Node x, y, z;
    x = new Node(4);
    y = new Node(8);
    z = new Node(42);
    Node head = y;    // line A  <-------------------
    y.next = x;
    x.next = z;
    print(head);
  }
  private static void print(Node head) {
    if (head == null) {
      return;
    }
    else {
      System.out.println(head.data);
      print(head.next);
    }
  }
}
```

The code above compiles and runs without errors.

(a) Draw a picture of `head` and the objects created in `main()` at the time when the line marked "`line A`" has finished executing. Draw `Node`s in the standard simplified style, as shown in the sketch below. (You may use the full memory model if you prefer, and you may use the blank page opposite if you need more space.)



(b) What is the output from the program above?

```
8
4
42
```

3. [10 marks = 5 + 5]

Ackermann's function is useful in big-O analysis of operations on, for example, data structures representing sets. This function grows astonishingly fast — much faster than the exponential function. Here is the definition of Ackermann's function in terms of its two parameters, i and j:

$$A(1, j) = 2^j \text{ for } j \geq 1$$

$$A(i,1) = A(i-1,2) \text{ for } i \geq 2$$

$$A(i, j) = A(i-1, A(i, j-1)) \text{ for } i, j \geq 2$$

(a) Complete the Java method `twoto(n)` to return 2^n. You may not use any classes from the Java API. You may use recursion if you like.

```
/** Return 2 raised to the power n.
 * Precondition: n >= 0 */
public static int twoto(int n) {
  int pow = 1;
  for (int i = 0; i < n; i++) {
    pow *= 2;
  }
  return pow;
}
```

OR

```
  if (n == 0)
    return 1;
  else
    return 2 * twoto(n – 1);
}
```

(b) Complete the Java method `ack(i,j)` to return Ackermann's function. You may use `twoto()` from part (a).

```
/** Return Ackermann's function A(i,j).
 * Precondition: i >= 1, j >= 1 */
public static int ack(int i, int j) {

  if (i == 1) {
    return twoto(j);
  }
  else if (j == 1) {
    return ack(i - 1, 2);
  }
  else {
    return ack(i - 1, ack(i, j - 1));
  }
}
```

4. [10 marks = 8 + 2]

Here is a segment of code:

```
// Read n from input.
int n = Integer.parseInt(br.readLine());

for (int i = n*n; i > 1; i -= i/2) {
  if (i % 2 == 0) {
    System.out.println(i + " is even");
  }
}
```

(a) What is the complexity (the "big-O time of execution") of this code, in terms of `n`?

Show your work, and get a result in simplest form.

Cost of loop body (the "if" statement): O(1), because there is no repetition (iteration or recursion)

Number of loop iterations = number of times n^2 can be divided by 2 before getting to 1

= $\log n^2$

= $2 \log n$

So cost of loop = $2 \log n \times O(1) = O(\log n)$

Cost of "int n = …": O(1), because there is no repetition

So total cost = $O(\log n) + O(1) = O(\log n)$

(b) Why is the "keep going" condition of the `for` statement `i>1` rather than `i>0`, as you might expect?

Because if i = 1, the next value of i is 1 – 1/2 = 1 – 0 = 1, so the loop is infinite.

5. [10 marks]

Here is an abstract class representing a kind of list:

```
import java.util.Iterator;

public abstract class MyList<E> implements Iterable {
  public abstract void add(E item);

  /** Return an Iterator of the usual type, with the usual methods:
   *  hasNext() and next().
   */
  public abstract Iterator<E> iterator();

  public boolean contains(E item) {...}
}
```

There are only three methods, and two of them are abstract. Your job is to write `contains()`, which is not abstract. Its purpose is to return `true` or `false` depending on whether the parameter `item` is found in the list. The standard `equals()` method can be used to check whether `item` matches an object in the list. You may assume that no entry in the list is `null`.

You may not change any part of the class `MyList` except the body of the `contains()` method. Note that this means you that you cannot add new methods, class or instance variables. In addition, you may not use any class from the standard API except `Iterable` and `Iterator`, which are already imported.

Here is the header again to get you started:

```
public boolean contains(E item) {

    for (Iterator it = iterator(); it.hasNext(); ) {
      if (it.next().equals(item)) {
        return true;
      }
    }
    return false;
  }

OR (with a small change in the class header):

    for (E el : this) {
      if (el.equals(item)) {
        return true;
      }
    }
    return false;
  }
```