Prelim 1 Solutions

CS2110, October 2, 2014, 7:30PM

	1	2	3	4	5	Extra	Total
Question	TrueFalse	Multiple	Object Oriented	Recursion	Lists	Extra Credit	
Max	20	20	30	15	15	5	100
Score							
Grader							

The exam is closed book and closed notes. Do not begin until instructed.

You have **90 minutes**. Good luck!

Write your name and Cornell *netid* at the top of EACH page! There are 5 questions on 13 numbered pages, front and back. Check that you have all the pages. When you hand in your exam, make sure your booklet is still stapled together. If not, please use our stapler to reattach all your pages!

We have scrap paper available. If you do a lot of crossing out and rewriting, you might want to write code on scrap paper first and then copy it to the exam, so that we can make sense of what you handed in.

Write your answers in the space provided. Ambiguous answers will be considered incorrect. You should be able to fit your answers easily into the space provided.

In some places, we have abbreviated or condensed code to reduce the number of pages that must be printed for the exam. In others, code has been obfuscated to make the problem more difficult. This does not mean that its good style.

1. True / False (20 points)

a)	T	F	A method in an abstract class must have no body.
b)	T	F	If variable a is declared to be of type A and class A does not define function `toString()`, `return "" + a;` is illegal.
c)	T	F	`double[][] aa = new double[2][];` is valid syntax for constructing a ragged array.
d)	T	F	You cannot append elements to an existing array.
e)	T	F	In a doubly-linked list, a node can be removed in constant time if you have a pointer to the node.
f)	T	F	Executing `e= null;` causes the object pointed by `e` to be deleted from every place it is referenced.
g)	T	F	If a method contains a loop that looks like this: `for (int i=0; i < n; i= i+1)` {...}, then space is allocated for `i` whenever execution of the loop starts and deallocated when execution of the loop ends.
h)	T	F	If A is an abstract class, a declaration `A v;` is legal.
i)	T	F	The `protected` methods of a superclass can be called from a subclass.
j)	T	F	A field of type `Boolean` will have a value of `null` if the constructor does not assign it a value.
k)	T	F	Suppose abstract class A declares abstract method `m()` and non-abstract class B extends A. Then if B does not declare `m()`, the java compiler will insert the declaration `m(){}` in B.
l)	T	F	The largest value in an unsorted array with n values can always be found in $O(\log n)$ steps.
m)	T	F	In the average case, Quicksort will sort an array of size n in time proportional to $O(n \log n)$.
n)	T	F	A Java class can extend many different classes, but only if all but one of the classes is abstract.
o)	T	F	Even though `Integer` is a subtype of `Object`, `int` is not a subtype of `Object`.
p)	T	F	Elements are removed from a queue in last-in-first-out order.
q)	T	F	`int[]` is a primitive type.
r)	T	F	Evaluation of `new ArrayList<Integer>()` will call the constructor in the Object partition of the object being created.
s)	T	F	Suppose class `Dog` contains static method `m()` and non-static field `id`. Then, assigning a value to `id` will change the value of `id` in every instance of `Dog`.
t)	T	F	If A extends B and C extends B, then `B b= new A(); A a= b;` is illegal and results in a compile-time error.

2. Multiple Choice (20 points)

In each part, circle **all** the correct answers.

(a) (3 points) The *travelling salesman problem* (TSP) asks this question: Given a list of cities and the distances between each pair of cities, what is the shortest route that visits each city exactly once and returns to the original city? The brute-force method (i.e. trying every possible circuit) is $O(n!)$ —for n cities, it takes time proportional to n factorial. On our laptop, which executes 4 billion instructions per second, it took 1 year to find the solution for a 19 city problem! If we add 1 more city, giving 20 cities, and run the program again on our laptop, how long can we expect to wait for it to finish?

A. 2 years B. 20 years C. 40 years D. $2 * 20!$ years

(b) (3 points) Which of the following statements about Collections classes is false?

A. An `ArrayList` has $O(1)$ insertion and removal time except when the backing array is being increased or decreased in size.

B. The elements of a `HashSet` are indexed from 0 to `size() - 1`, and element i can be accessed via method `get`.

C. A `LinkedList` has $O(n)$ removal time for an arbitrary value if you only have a reference to the first and last node.

Questions (c) and (d) refer to the following class:

```
public class A {
    public static double m(int x) {
        int y = x;
        try {
            System.out.println("one");
            y= 5/x;
            System.out.println("two");
            return 5/Math.max(0, x);
        } catch (RuntimeException e) {
            System.out.println("three");
            y= 5/(x+1);
            System.out.println("four");
        }
        System.out.println("five");
        y= 4/(x + 2)
        System.out.println("six");
        return 1/(x + 1);
    }
}
```

(c) (4 points) If `m(x)` is called for some `x` < 0, which of the following will definitely be printed?
A. "three" B. "four" C. "six" D. None of These

(d) (4 points) Which value of `x` will cause an exception to be thrown out of `m`.
A. -2 B. 0 C. 1 D. None of These

(e) (3 points) Which of the following operations take constant time?

A. Look at the first ten items in a list of n items

B. Access element number i of a linked list of length n

C. Access element number i of an array of length n

(f) (3 points) Which of the following uses of collection classes is correctly justified?

A. A `LinkedList` should be used to implement a stack because adding and removing an element at the beginning of the list takes constant time

B. An `ArrayList` should be used to implement a queue because we can access any element in constant time.

C. An `ArrayList` should be used to implement a stack because adding and removing an element at the beginning can be done in constant time.

D. A `HashSet` should be used for implementing a stack that disallows the same item from being on the queue twice because it doesn't allow duplicate elements.

3. Object Oriented Design (30 points)

You and your friend are writing a game in Java with two types of characters: Snakes and Apes. Your friend gives you the following classes, which you are now tasked with improving.

```
public class Game {
   private List<Snake> snakes;
    private List<Ape> apes;

    public void updateAll() {
        for (Snake s : snakes) { update(s); }
        for (Ape a : apes) { update(a); }
    }

    public void update(Snake s) {
        s.x += s.xSpeed;
        s.y += s.ySpeed;
        s.doStuff();
    }

    public void update(Ape a) {
        a.x += a.xSpeed;
        a.y += a.ySpeed;
        a.doStuff();
    }
}

public class Ape {
    public int x;
    public int y;
    public int xSpeed;
    public int ySpeed;

    public void doStuff() {
        System.out.println("doing ape stuff");
    }
}

public class Snake {
    public int x;
    public int y;
    public int xSpeed;
    public int ySpeed;

    public void doStuff() {
        System.out.println("doing snake stuff");
    }
}
```

(a) 5 points Identify *two* design problems with these three classes from an object-oriented perspective. Explain why these two are problems in at most 2 sentences per problem. Hint: There are at least four or five problems of various natures, some dealing with syntax and access modifiers and others dealing with what we discussed in the recitation on abstract classes.

Solution There are four possible answers, of which they must get at least 2:

- Ape and Snake duplicate code; since the behavior is the same, the logic should be shared in a superclass.
- Ape and Snake have public fields which should be encapsulated.
- In Game, the same logic is duplicated for Apes and Snakes in both update methods and the updateAll method.
- The update methods are logically related to the Ape and Snake behavior, so they belong in those classes, not in the Game class.

(b) 15 points Rewrite the above program by improving the object-oriented design elements. You should:

- Create a new abstract class `GameCharacter`.
- Rewrite classes `Ape` and `Snake` to extend class `GameCharacter`.
- Rewrite class `Game` to take advantage of class `GameCharacter`.

Solution

```
public class Game {
    private List<GameCharacter> gameCharacters;

    public void updateAll() {
        for (GameCharacter gc : gameCharacters) {
            gc.move();
            gc.doStuff();
        }
    }
}

public abstract class GameCharacter {
    private int x;
    private int y;
    private int xSpeed;
    private int ySpeed;

    public abstract void doStuff();
    public void move() {
        x += xSpeed;
        y += ySpeed;
    }
}

public class Ape extends GameCharacter {
    public void doStuff() {
        System.out.println("doing ape stuff");
    }
}

public class Snake extends GameCharacter {
    public void doStuff() {
        System.out.println("doing snake stuff");
    }
}
```

(c) 10 points Suppose class `Game` has a list of `GameCharacter`s. You want to sort this list so that all Snakes come before all Apes by calling `Collections.sort(listOfGameCharacters)`, which sorts the list in ascending order. In order to do this, GameCharacter must implement `Comparable<GameCharacter>`.
Implement function `compareTo` below according to its specification. Assume that `Ape` and `Snake` are the only subclasses of `GameCharacter`.

Solution

```
public abstract class GameCharacter implements Comparable<GameCharacter> {
    /**
    * Compare this object with obj. Return a negative integer, zero,
    * or a positive integer depending on whether this object is less
    * than, equal to, or greater than obj.
    *
    * Note: the comparison need ensure only that Snake objects come
    * before Ape objects.
    */
    public int compareTo(GameCharacter obj) {
        if (this instanceof Ape && obj instanceof Snake)
            return 1;
        } else if (this instanceof Snake && obj instanceof Ape) {
            return -1;
        } else {
            return 0;
        }
    }
}
```

4. Recursion (15 Points)

(a) 5 points Recall the fibonacci numbers: fibonacci number a_n is defined as the sum of the 2 previous fibonacci numbers: $a_n = a_{n-1} + a_{n-2}$, with $a_0 = a_1 = 1$. Based on fibonacci numbers we define the jamesinacci numbers as the list of numbers where jamesinacci number j_n is defined as the sum of the previous 3 jamesinacci numbers if n is even and the sum of the previous 2 jamesinacci numbers if n is odd. The first three jamesinacci numbers j_0, j_1, and j_2 are 1.

Implement recursive function `jamesinacci` below according to its specification.

Solution

```
/** Return jamesinacci number jn.
 *  Precondition n >= 0. */
public int jamesinacci(int n) {
    if(n < 3) {
        return 1;
    }
    if(n % 2 == 1) {
        return jamesinacci(n - 1) + jamesinacci(n - 2);
    }
    return jamesinacci(n - 1) + jamesinacci(n - 2) + jamesinacci(n - 3);
}
```

(b) 10 points Consider class `ListNode` below. `ListNode` is a Node of a singly-linked list. The relevant fields are shown below:

```
/** A Node of a Singly-Linked List */
public class ListNode<E> {
    E value; // The value stored in this ListNode
    ListNode<E> next; // Next ListNode, null if this is the last node

}
```

Complete function `reverse` below according to its specification. Do not use an array or any other data structure. Do not create new ListNodes. Do not use a loop. Use only recursion.

Solution

```
/** Return the list, starting at node n, in reverse order.
 * Each value should be separated by a space.
 * The list is NOT modified */
public static String reverse(ListNode<E> n) {
    if(n == null) {
        return ""
    }
    return reverse(n.next) + " "  + n.value;
}
```

5. Loop Invariants (15 Points)

(a) 10 points Implement function `findPartition` below. Your algorithm must use a loop that uses the invariant drawn below. For full credit, it must also run in $O(\log b.length)$ time.

Solution

```
/** Return the value k such that b[0..k-1] < 0 < b[k..]
 *  Precondition: b contains no zeroes and all the
 *  negative values come before all the positive ones
 */
public static int findPartition(int[] b) {
    int h = -1;
    int k = b.length;
    while(k != h + 1) {
        int j = (h + k) / 2;
        if(b[j] > 0) h = j;
        else  k = j;
    }
    return k;
}
```

(b) 5 points You and your friend are tasked with writing a loop to sum the values of array segment $b[0..h]$. Your loop (with initialization) must use the following invariant:

Your friend writes the following loop:

```
t= h;
while(t >= 0) {
    s= s + b[t];
    t= t - 1;
}
```

Write down the four loopy questions and, for each, state whether it is correctly satisfied by your friend's loop.

Solution

- **How does the initialization of loop variables make the invariant true?** This question is not correctly satisfied. The initialization creates a state that violates the invariant.
- **How does the loop body preserve the invariant after each iteration?** This question is not correctly satisified. The loop body actually preserves a different invariant than the one stated and is not correct.
- **How does the loop body make progress towards termination each iteration?** This question is correctly satisfied as the loop reduces the number of elements that remain to be added to s.
- **How do we know that when the loop condidition is false, the post condition is true** This question is not correctly satisified. When the loop is finished s does hold the sum of all elements in b, but t is 0 when it should be h

6. Extra Credit (5 Points)

Write the body of the function `findCommonList`, whose specification and header appear below. Like the recursion problem, `ListNode<E>` has two fields, `E value` and `ListNode<E> next`. Feel free to add subfunctions.

Your solution must run in $O(n)$, be completely functionally correct to receive credit, and not use any additional data structures. You cannot use more than constant ($O(1)$) space.

Solution

```
/** The singly linked list in the problem has two heads, n1 and n2, that merge
 *  at a common node. Return the first common node that is accessible from both
 *  n1 and n2. This must run in O(n) time.
 */
public static<E> ListNode<E> findCommonList(ListNode<E> n1, ListNode<E> n2) {
    int length1 = getLength(n1);
    int length2 = getLength(n2);

    if (length1 > length2)
        n1 = advance(n1, length1 - length2);
    else
        n2 = advance(n2, length2 - length1);

    while (n1 != n2) {
        n1 = n1.next;
        n2 = n2.next;
    }
    return n1;
}


private static<E> ListNode<E> advance(ListNode<E> n, int k) {
    while (k > 0) {
        n = n.next;
        k--;
    }
    return n;
}


private static<E> int getLength(ListNode<E> n) {
    int total = 0;
    while (n != null) {
        total++;
        n = n.next;
    }
    return total;
}
```