BIRKBECK

(University of London)

MSc EXAMINATION FOR INTERNAL STUDENTS

MSc Computer Science

MSc Data Science

Department of Computer Science and Information Systems

Principles of Programming II

BUCI064H7

DATE OF EXAMINATION: Wednesday, 2nd May 2018

DURATION OF PAPER: One hour

Mock practical paper

WITH OUTLINE SOLUTIONS

RUBRIC:

1. Candidates should attempt ALL 4 questions on this paper.
2. You are advised to look through the entire examination paper before getting started, in order to plan your strategy.
3. Simplicity and clarity of expression in your answers is important.
4. All programming questions should be answered using the Java programming language.
5. Electronic calculators are **NOT** allowed.
6. Start each question on a new page.

Question:	1	2	3	4	Total
Marks:	10	14	13	13	50

Question 1 ..... Total: 10 marks

Write a recursive method `isReverse` that accepts two strings as a parameter and returns `true` if the two strings contain the same sequence of characters as each other but in the opposite order (ignoring capitalisation), and `false` otherwise.

For example, the string `"hello"` backwards is `"olleh"`, so a call of

```
isReverse("hello", "olleh")
```

would return `true`. Since the method is case-insensitive, you would also get a `true` result from a call of

```
isReverse("Hello", "oLLEh")
```

The empty string, as well as any one-letter string, is considered to be its own reverse. The string could contain characters other than letters, such as numbers, spaces, or other punctuation; you should treat these like any other character.

The key aspect is that the first string has the same sequence of characters as the second string, but in the opposite order, ignoring case.

You may assume that the strings passed are not `null`. You are not allowed to construct any structured objects other than `String`s (no array, `List`, `Scanner`, etc.) and you may not use any loops to solve this problem; you must use recursion. If you like, you may declare other methods to help you solve this problem, subject to the previous rules.

Solution: Four possible solutions are shown:

```
package worksheet;

public class IsReverse {
  public static boolean isReverse(String s1, String s2) {
    if (s1.length() == 0 && s2.length() == 0) {
      return true;
    } else if (s1.length() == 0 || s2.length() == 0) {
      return false;  // not same length
    } else {
      String s1first = s1.substring(0, 1);
      String s2last = s2.substring(s2.length() - 1);
      return s1first.equalsIgnoreCase(s2last) &&
               isReverse(s1.substring(1),
                 s2.substring(0, s2.length() - 1));
    }
  }

  public static boolean isReverseTwo(String s1, String s2) {
    if (s1.length() != s2.length()) {
      return false;  // not same length
    } else if (s1.length() == 0 && s2.length() == 0) {
      return true;
    } else {
      s1 = s1.toLowerCase();
      s2 = s2.toLowerCase();
```

```
      return s1.charAt(0) == s2.charAt(s2.length() - 1) &&
             isReverseTwo(s1.substring(1, s1.length()),
               s2.substring(0, s2.length() - 1));
    }
  }

  public static boolean isReverseThree(String s1, String s2) {
    if (s1.length() == s2.length()) {
      return isReverse(s1.toLowerCase(), 0,
        s2.toLowerCase(), s2.length() - 1);
    } else {
      return false;   // not same length
    }
  }

  private static boolean isReverse(String s1, int i1,
                                   String s2, int i2) {
    if (i1 >= s1.length() && i2 < 0) {
      return true;
    } else {
      return s1.charAt(i1) == s2.charAt(i2) &&
             isReverse(s1, i1 + 1, s2, i2 - 1);
    }
  }


  public static boolean isReverseFour(String s1, String s2) {
    return reverse(s1.toLowerCase()).equals(s2.toLowerCase());
  }

  private static String reverse(String s) {
    if (s.length() == 0) {
      return s;
    } else {
      return reverse(s.substring(1)) + s.charAt(0);
    }
  }
}
```

Question 2 ..... Total: 14 marks

Consider the following class definition:

```
public abstract class Date {
    private int day;          // from 1 to 31
    private int month;        // from 1 to 12
    private int year;         // from 2000 upwards

    public void advance() {
    }  // "advance" to next day
}
```

(a) Implement a constructor that initialises new objects of the `Date` class to be set to the 1st of January 2000. [3 marks]

(b) Implement getters and setters for each of the private fields. [2 marks]

(c) Provide appropriate `hashCode`, `toString`, and `equals` methods for the class. (No credit for auto-generated versions.) [4 marks]

(d) Provide an implementation for the `advance` method which advances the date to the next day. You should ensure that all the private fields are updated appropriately using their respective getters and setters. (You may assume that all months have the same number of days.) [5 marks]

Solution:

```
package datesol;

public abstract class Date {
    private int day;          // from 1 to 31
    private int month;        // from 1 to 12
    private int year;         // from 2000 upwards

    public Date() {
        this(1, 1, 2000);
    }

    public Date(int day, int month, int year) {
        setDay(day);
        setMonth(month);
        setYear(year);
    }

    // "advance" to next day
    public void advance() {
        if (day < 31) {
            setDay(day + 1);
        } else {
            if (month < 12) {
                setDay(1);
                setMonth(month + 1);
            } else {
                setYear(year + 1);
                setMonth(1);
                setDay(1);
```

```
            }
        }
    }

    public int getDay() {
        return day;
    }

    public void setDay(int day) {
        this.day = day;
    }

    public int getMonth() {
        return month;
    }

    public void setMonth(int month) {
        this.month = month;
    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }

    @Override
    public String toString() {
        return "Day " + day + " Month " + month + " Year " + year;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) return false;
        if (this == obj) return true;
        if (!(obj instanceof Date)) return false;
        Date d = (Date) obj;
        return this.getDay() == d.getDay()
                && this.getMonth() == d.getMonth()
                && this.getYear() == d.getYear();
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + getDay();
        result = prime * result + getMonth();
```

```
        result = prime * result + getYear();
        return result;
    }
}
```

Question 3 ..... Total: 13 marks

Write a method `rarestAge` that accepts as a parameter a map from students' names (strings) to their ages (integers), and returns the *least* frequently occurring age. Consider a map variable `m` containing the following key/value pairs:

```
{Alyssa=22, Char=25, Dan=25, Jeff=20, Casey=20, Kim=20,
 Morgan=25, Ryan=25, Stef=22}
```

Three people are age 20 (Jeff, Casey, and Kim), two people are age 22 (Alyssa and Stef), and four people are age 25 (Char, Dan, Morgan, and Ryan). So a call of `rarestAge(m)` returns `22` because only two people are that age.

If there is a tie (two or more rarest ages that occur the same number of times), return the youngest age among them. For example, if we added another pair of `Kelly=22` to the map above, there would now be a tie of three people of age 20 (Jeff, Casey, Kim) and three people of age 22 (Alyssa, Kelly, Stef). So a call of `rarestAge(m)` would now return `20` because 20 is the smaller of the rarest values.

If the map passed to your method is `null` or empty, your method should throw an `IllegalArgumentException`. You may assume that no key or value stored in the map is `null`. Otherwise you should not make any assumptions about the number of key/value pairs in the map or the range of possible ages that could be in the map.

You may create one new set or map as auxiliary storage to solve this problem. You can have as many simple variables as you like. You should not modify the contents of the map passed to your method.

Solution: Four possible solutions are shown:

```
package worksheet;

import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

public class RarestAge {
  public static int rarestAge(Map<String, Integer> m) {
    if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
    }

    Map<Integer, Integer> counts = new TreeMap<>();
    for (String name : m.keySet()) {
      int age = m.get(name);
      if (counts.containsKey(age)) {
        counts.put(age, counts.get(age) + 1);
      } else {
        counts.put(age, 1);
      }
    }

    int minCount = m.size() + 1;
    int rareAge = -1;
```

```
    for (int age : counts.keySet()) {
      int count = counts.get(age);
      if (count < minCount) {
        minCount = count;
        rareAge = age;
      }
    }
    return rareAge;
  }

  public static int rarestAgeTwo(Map<String, Integer> m) {
    if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
    }

    Map<Integer, Integer> counts = new TreeMap<>();
    for (int age : m.values()) {
      if (!counts.containsKey(age)) {
        counts.put(age, 0);
      }
      counts.put(age, counts.get(age) + 1);
    }

    int rareAge = -1;
    for (int age : counts.keySet()) {
      if (rareAge < 0 || counts.get(age) < counts.get(rareAge)) {
        rareAge = age;
      }
    }
    return rareAge;
  }

  public static int rarestAgeThree(Map<String, Integer> m) {
    if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
    }

    Map<Integer, Integer> counts = new TreeMap<>();
    for (String name : m.keySet()) {
      if (counts.containsKey(m.get(name))) {
        counts.put(m.get(name), counts.get(m.get(name)) + 1);
      } else {
        counts.put(m.get(name), 1);
      }
    }

    int minCount = Integer.MAX_VALUE;
    // really big number to be overwritten
```

```
    for (int age : counts.keySet()) {
      minCount = Math.min(minCount, counts.get(age));
    }
    for (int age : counts.keySet()) {
      if (counts.get(age) == minCount) {
        return age;
      }
    }
    return -1;   // won't reach here
  }

  public static int rarestAgeFour(Map<String, Integer> m) {
    if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
    }

    Map<Integer, Integer> counts = new HashMap<>();
    for (String name : m.keySet()) {
      if (!counts.containsKey(m.get(name))) {
        counts.put(m.get(name), 0);
      }
      counts.put(m.get(name), counts.get(m.get(name)) + 1);
    }

    int minCount = Integer.MAX_VALUE;
    // really big number to be overwritten

    for (int age : counts.keySet()) {
      minCount = Math.min(minCount, counts.get(age));
    }
    int rareAge = -1;
    for (int age : counts.keySet()) {
      if (counts.get(age) == minCount &&
          (rareAge < 0 || age < rareAge)) {
        rareAge = age;
      }
    }
    return rareAge;
  }
}
```

Question 4 ..... Total: 13 marks

Consider the following Java code for circular linked lists.

```
public class CList extends Thread {
  private int item;
  private Clist next;
  private int updateCount=0;

  public synchronized void replace(int old, int neww, Clist startedAt) {
      if(item==old) {item=neww; updateCount++; }
      if(next!=startedAt) next.replace(old, neww, startedAt);
  }

  public void run() {
      for(int i=0; i<100; i++)
          replace(i, i-1, this);
  }

  public Clist(int item, Clist next)  { this.item=item;  this.next=next; }

  public static void main(String args[]) {
      Clist clist3 = new Clist(60, null);
      Clist clist2 = new Clist(40, clist3);
      Clist clist1 = new Clist(20, clist2);
      clist3.next = clist1;
      clist1.start();
      clist2.start();
      clist3.start();
  }
}
```

(a) Explain why it is possible for this code to deadlock. Include a description of a situation where a deadlock occurs. 5 marks

Solution: This code may deadlock because clist1, clist2 and clist3 together form a cycle — repeatedly following the "next" instance variable leads back to the original CList object. The three threads each run replace, which will lock the object, then the next object, and then the third object. So, we can have a deadlock if two threads commence a call to replace at about the same time, since each will lock its own CList object, then then try to obtain locks on the other two CList objects. E.g., we can have the following sequence.

```
thread 1: calls replace, obtains the lock on CList1
thread 2: calls replace, obtains the lock on CList2
thread 1: calls next.replace, waits for the lock on CList2
thread 2: calls next.replace, obtains the lock on CList3
thread 2: calls next.replace, waits for lock on CList1
DEADLOCKED
```

(b) Show two different ways that this code may be fixed in order to avoid deadlocks. For each, write the code for the methods that need to be changed and also, in your comments, explain why deadlocks are no longer possible. 8 marks

Solution:

(a) The deadlock may be resolved by using the class object lock, as follows.

```
public void replace(int old, int neww, Clist startedAt) {
        synchronized(CList.class) {
                if(item==old) {item=neww; updateCount++; }
                if(next!=startedAt) next.replace(old, neww, startedAt);
    }
}
```

This code can no longer deadlock because any thread that holds the class object lock will be able to complete the original call to replace without waiting for any more locks, since it never waits for any other lock. So, it will eventually release the lock, and since this is the only lock other threads could be waiting for, they will eventually get to continue.

(b) Another way to fix this code is to reduce the duration that locks are held, as follows:

```
public void replace(int old, int neww, Clist startedAt) {
          synchronized(this) {
              if(item==old) {item=neww; updateCount++; }
          }
          if(next!=startedAt) next.replace(old, neww, startedAt);
      }
```

This code cannot deadlock because threads only hold one lock at a time. So, if one thread is waiting for a lock held by a second thread, we know that the second thread only hold the lock for a short time (while executing the first `if` above), and in particular it will release the lock before it waits on another lock. Hence, the first thread will be able to obtain the lock and will not wait forever.