Family Name:..... **First Name:**.....

Student ID:.....

COMP 103 : Test ** WITH SOLUTIONS **

September 8, 2022

Instructions

- Time allowed: **50 minutes**
- Write your Family Name and Student ID on top of the first page and your Student ID on top of the rest of the pages.
- Attempt **all** questions. There are 50 marks in total.
- Write your answers in this test paper and hand in all sheets.
- If you think some question is unclear, ask for clarification.
- Brief Java documentation for Collections is provided with the test
- This test contributes 15% of your final grade
- You may use dictionaries.
- You may write notes and working on this paper, but make sure your answers are clear.

Questions	Marks	
1. Properties of Collections	[9]	
2. Using Collections	[8]	
3. More on using Collections	[18]	
4. Cost of Algorithms	[7]	
5. Recursion	[8]	
	TOTAL:	

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 1. Properties of Collections **[9 marks]**

(Terminology: A collection is an object that groups multiple elements into a single unit.)

For these questions, circle "true" or "false" for each property.

(a) **[3 marks]** For a collection which is an implementation of the Set interface, state whether each property is true or false.

true/false : The last element added to the collection is the first element to be removed

true/false : Items must have a natural ordering to be stored in the collection

true/false : The collection cannot contain duplicate elements

(b) **[3 marks]** For a collection which is an implementation of the List interface, state whether each property is true or false.

true/false : The collection records the number of copies of each element

true/false : An element can be removed from the collection at any time, regardless of when the element was added to the collection.

true/false : An element can only be removed by specifying its index

(c) **[3 marks]** For a collection which is an implementation of the Map interface, state whether each property is true or false.

true/false : Values must be added with a key

true/false : The collection records the number of copies of each element

true/false : The order of elements of the collection can be reversed

Marking:
3 marks for getting all 3 properties right;
1.5 marks for getting 2 properties right;
0 marks for less than 2 right.

Question 2. Using Collections **[8 marks]**

(a) **[6 marks]** Complete the following tidyList method which is given a list of words (Strings), and returns a Stack. The method should go through the list of words, adding those that have not been added previously to a Stack. You should use a Set to keep track of the words that have already occurred.

```
public Stack<String> tidyList( List<String> L){
    // put your code here



/**
 * Go through the list of words provided.
 * If the word has not occured before, add it to a Stack.
 * At the end of the list, return that stack.
 */
    Set<String> used = new HashSet<String>();
    Stack<String> stk = new Stack<String>();
    for (String s : L) {
        if (!used.contains(s)) {
            stk.push(s); // add might work too?
            used.add(s);
        }
    }
    return stk;
}



}
```

(b) **[2 marks]** The following code uses the tidyList method. What should it produce, assuming the method is correct?

```
String s = "s e s o g e n o i m g s a s o n l f o e";
// populating myList with all the letters of the String s.
List<String> myList = Arrays.asList(s. split (" "));

Stack<String> myStack = tidyList(myList);
while (!myStack.isEmpty())
    UI. print (mytack.pop());
```

Answer:

flamingoes (no spaces between letters)

Question 3. More on using Collections **[18 marks]**

This question concerns a group of players of some game. Each player has a name (String) and a skill level (integer).

(a) **[5 marks]** The Player class is given below. By adding or altering code, make the Player class *Comparable*, so that players possess an ordering based on their skill level such that the player with the highest skill comes first.

```
public class Player implements Comparable<Player> {

    // Fields
    private String name;
    private int skill ;

    // Constructor
    public Player (Scanner data) {
        name = data.next();
        skill = data.nextInt();
    }

    // Methods

    public int getSkill () {return skill ;}

    public String toString() {
        return(String.format("%s (skill %d)", name, skill));
    }

    /**
     * Compare this Player with another Player.
     * Natural ordering is by skill: the most skilled come first.
     */
    public int compareTo(Player other){
        if (this.skill > other.skill )      {return -1;}
        else if (this.skill < other.skill ){return 1;}
        else                                {return 0;}
    }



}
```

(Question 3 continued on next page)

(Question 3 continued)

A school needs an application to organise children (players) into two teams plus an ordered list of the remaining children (substitutes) who were not paired.

You must modify the class PairsOrganiser on the facing page to achieve this.

You can assume the method loadPlayersFromFile() exists and works as intended.

You need to implement the method organise() to achieve the goal using the algorithm below.

The queues (queueA and queueB) contain two lists of players. They do not contain any null entries, but they can be empty. All players have unique names.

Use queueA and queueB to build a Map of players.

Each entry in the Map must contain a key (the next available player from queueA) and a value (the next available player from queueB.)

Continue filling the Map while there are players available in both queues.

When one (or both) of the queues is empty, print the pairs of matched players. For an entry where "Bob" is the key, and "Elsie" is the value, the algorithm must print "Bob paired with Elsie" on a single new line.

Finally, if one of the queues has any remaining players, it must be converted into a List called subs. The subs List must be sorted according to the players' skill level, where high-level players appear before low-level players in the List. The List must be printed out at the end.

Completing all of the above functionalities is worth 13 marks.

Partial marks can be awarded as follows:

(b) **[6 marks]** Correctly build a Map of pairs from the two queues.

(c) **[3 marks]** Print out all the pairs in the Map by looping through the Map.

(d) **[4 marks]** Create and print the sorted List of the remaining (unpaired) players.

Note you will be awarded partial marks for printing out the entries in the Map, even if you did not build the Map correctly.

(Question 3 continued on next page)

(Question 3 continued)

```
public class PairsOrganiser {

    private Queue<Player> queueA, queueB;

    // Constructor
    public PairsOrganiser() {
        queueA = loadPlayersFromFile("poolA.txt");
        queueB = loadPlayersFromFile("poolB.txt");
        organise();
    }

    public void organise() {

        // Build the Map
        Map<Player, Player> pairs = new HashMap<Player, Player>();
        while (!queueA.isEmpty() && !queueB.isEmpty()) {
            pairs.put(queueA.poll(), queueB.poll());
        }



        // Print out the pairs, by looping over the Map
        for (Player f1 : pairs.keySet())  // (Note: could also loop over Map Entries)
            UI.println(f1 + " paired with " + pairs.get(f1));



        // Print out the unmatched players, using their natural ordering :
        // For this, you can assume that Players are Comparable.
        List<Player> subs = new ArrayList<Player>();
        subs.addAll(queueA);
        subs.addAll(queueB);
        Collections.sort(subs);
        for (Player p : subs) UI.println("Unmatched: " + p);



    }

    public Queue<Player> loadPlayersFromFile(String filename) { ... etc ... }
}
```

Question 4. Cost of Algorithms **[7 marks]**

What is the big-O cost of the following method finding all the numbers that are repeated at least 3 times in a list of numbers? Suppose the list of numbers, called values, contains n numbers.

- write the big-O cost of performing each line with a comment
- write the number of times each of the lines will be performed
- write the total cost of the algorithm (big-O) at the bottom of the box

```
public void findTripleDuplicate (List<Integer> values){
    Set<Integer> result = new TreeSet<Integer>();      // cost = O( 1 )    times= 1
    for(int value: values){
        int count = 0;                                 // cost = O( 1 )    times= n
        for(int other: values){
            if(value == other)                         // cost = O( 1 )    times= n^2
                count++;                               // cost = O( 1 )    times= n^2
        }
        if(count >= 3)                                 // cost = O( 1 )    times= n
            result.add(value);                         // cost = O( log(n) ) times= n
    }
}                                                      Total Cost = O( n^2 )
```

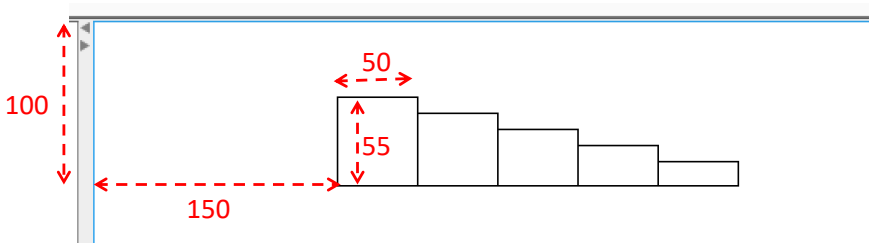
Marking:
Each correct line is 1 mark
Cost and Times worth 0.5 mark for each line

Question 5. Recursion **[8 marks]**

(a) **[4 marks]** Complete the following recursive drawStair(x, y, width, height) method to draw a stair which is a list of rectangles. Each rectangle is a stair step.

- You must use a "first-and-rest" recursion: drawStair should draw the first rectangle and then call itself recursively to draw the rest of the rectangles.
- (x, y) is the **bottom left** of the first rectangle.
- width and height are the width and height of the first rectangle.
- The next rectangle should be **adjacent** to the first rectangle, and its height should be **10 pixels** smaller than the height of the first rectangle.
- It should continue to draw rectangles as long as the height is larger than 10.

For example, drawStair(150, 100, 50, 55) should output the following:



The dashed double-arrow lines and the numbers are for indication, you do not need to draw them.

```
public void drawStair(double x, double y, double width, double height){
    if (height > 10){
        UI.drawRect(x, y-height, width, height);
        drawStair(x+width, y, width, height-10);
    }

}
```

Hint: you can use the method UI.drawRect(left, top, width, height) to draw a rectangle.

(Question 5 continued on next page)

(Question 5 continued)

(b) **[4 marks]** Given the following fun(x) method, what will be the outputs of fun(1), fun(2), and fun(4)?

```
public void fun( int x){
    if (x>0){
        fun(x−1);
        UI. print (x + "  ");
        fun(x−2);
    }
}
```

Hint: fun(1) and fun(2) can be used to find the output of fun(4)

Answer:

fun(1): 1

fun(2): 1 2

fun(4): 1 2 3 1 4 1 2

* * * * * * * * * * * * * * * *

Documentation

Brief, simplified specifications of some relevant Java collection types and classes.

Note: E stands for the type of the element (item) in the collection.

```
interface Collection<E>
   public boolean isEmpty()                    // cost: O(1) for all standard collection classes
   public int size()                           // cost: O(1) for all standard collection classes
   public void clear()
   public boolean add(E item)
   public boolean contains(Object item)
   public boolean remove(Object element)

interface List<E> extends Collection<E>
   // Implementations: ArrayList
   public boolean isEmpty()
   public int size()
   public void clear()
   public E get(int index)                     // cost: O(1)
   public E set(int index, E element)          // cost: O(1)
   public boolean contains(Object item)        // cost: O(n)
   public void add(int index, E element)       // cost: O(n) (unless index is close to the end.)
   public E remove(int index)                  // cost: O(n) (unless index is close to the end.)
   public boolean remove(Object element)       // cost: O(n)

interface Set extends Collection<E>
   // Implementations: HashSet, TreeSet
   public boolean isEmpty()
   public int size()
   public void clear()
   public boolean add(E item)                  // cost: O(1) for HashSet, O(log(n)) for TreeSet
   public boolean contains(Object item)        // cost: O(1) for HashSet, O(log(n)) for TreeSet
   public boolean remove(Object element)       // cost: O(1) for HashSet, O(log(n)) for TreeSet

interface Map<K, V>
   // Implementations: HashMap, TreeMap
   public V get(K key)                 // cost: O(1) for HashMap, O(log(n)) for TreeMap
   public V put(K key, V value)        // cost: O(1) for HashMap, O(log(n)) for TreeMap
   public V remove(K key)              // cost: O(1) for HashMap, O(log(n)) for TreeMap
   public boolean containsKey(K key)   // cost: O(1) for HashMap, O(log(n)) for TreeMap
   public Set<K> keySet()              // cost: O(1)
   public Collection<V> values()       // cost: O(1)
   // (get returns null if the key is not present)
   // (get put and remove return the old value, if any)
```

```
interface Queue<E> extends Collection<E>
   // Implementations: ArrayDeque, LinkedList, PriorityQueue
   public boolean isEmpty()
   public int size()
   public void clear()
   public E peek()                  // cost: O(1) for ArrayDeque, LinkedList, O(1) for PriorityQueue
   public E poll()                  // cost: O(1) for ArrayDeque, LinkedList, O(log(n)) for PriorityQueue
   public boolean offer(E element) // cost: O(1) for ArrayDeque, LinkedList, O(log(n)) for PriorityQueue
   // (peek and poll return null if the queue is empty)

class Stack<E> implements Collection<E>
   public boolean isEmpty()
   public int size()
   pubic void clear()
   public E peek()                  // cost: O(1)
   public E pop()                   // cost: O(1)
   public E push(E element)         // cost: O(1)
   // (peek and pop return null if the queue is empty)


class Collections
   public void sort(List<E> list);                          // cost = O(n log(n)), but O(n) if almost sorted
   public void sort(List<E> list, (E e1, E e2)->{...});     // cost = O(n log(n)), but O(n) if almost sorted
   public void swap(List<E> list, int i, int j);            // cost = O(1)
   public void reverse(List<E> list);                       // cost = O(n)
   public void shuffle(List<E> list);                       // cost = O(n)


interface Comparable<E> // All Comparable objects have a compareTo method:
   public int compareTo(E other);
          // returns
          //     -ve if this comes before other;
          //     +ve if this comes after other,
          //       0 if this and other are the same
```