Family Name: . . . . . **Other Names:** . . . . . .

Student ID: . . . . . **Signature** . . . . . .

COMP 103 : Test 3

2022, Jan 31

Instructions

- Time allowed: **50 minutes**
- Attempt **all** questions. There are 30 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 is provided with the test
- You may use dictionaries.
- You may write notes and working on this paper, but make sure your answers are clear.

Questions	**Marks**	
1. Maps	[10]	
2. Big-O	[10]	
3. Recursion	[10]	
	TOTAL:	

Question 1. Maps **[10 marks]**

You are writing a program to store data from a number of water quality sensors located in rivers across New Zealand. The Sensor class includes the following fields and method:

```
class Sensor {
        private int latitude ;
        private int longitude ;
        private int uniqueID;
        private String riverName;

        public String getRiverName() { return riverName; }
        ...
}
```

(a) **[8 marks]** Write a method called sensorCoverage that is given a list of river names, and a list of sensors, and returns a Map mapping each river name to a list of sensors installed *in that river*.
Note: if the location of a sensor is not in the given list of river names, it should be ignored.

```
public Map<String, List<Sensor>> sensorCoverage(List<String> riverNames,
                                                List<Sensor> sensors) {




}
```

(Question 1 continued on next page)

(Question 1 continued)

(b) **[2 marks]** If there are n rivers and m sensors, then what is the Big-O cost of the method you wrote?

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 2. Big-O **[10 marks]**

(a) **[6 marks]** What is the Big-O cost of the following method (in the worst case)?
- write the Big-O cost of performing each line with a comment
- write the number of times each of the lines will be performed (at most),
- write the total cost of the algorithm (Big-O) at the bottom of the box

```
public void removeBadWords(List<String> words){

    for( int i=0; i<words.size(); i++) {
        if ( words.get(i).equals("keyword") ) {  // cost= O(      )   times=
            words.remove(word));                 // cost= O(      )   times=
            words.add(0,""));                    // cost= O(      )   times=
        }
    }
}                                                Total Cost = O(          )
```

(b) **[4 marks]** What is the Big-O cost of the following method, which uses a HashSet?

```
public Set<String> removeBadWords(List<String> words){
    Set<String> cleaned = new HashSet<String>();
    for( String word : words) {
        cleaned.add(word);                       // cost= O(      )   times=
    }
    for( String word : words) {
        if ( word.equals("keyword") ) {          // cost= O(      )   times=
            cleaned.remove(word);                // cost= O(      )   times=
            cleaned.add("");                     // cost= O(      )   times=
        }
    }
    return cleaned;
}                                                Total Cost = O(          )
```

Question 3. Recursion **[10 marks]**

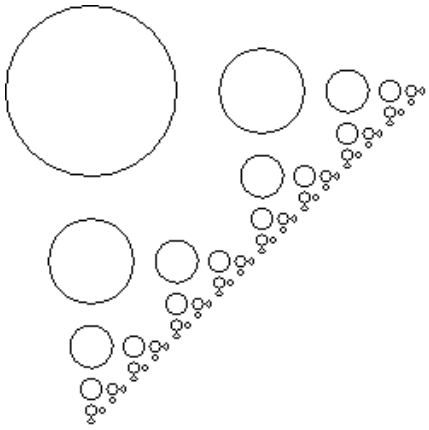
(a) **[5 marks]** Complete the following recursive drawPattern(x, y, dist) method to draw a triangular pattern made of circles of decreasing size.

The pattern consists of

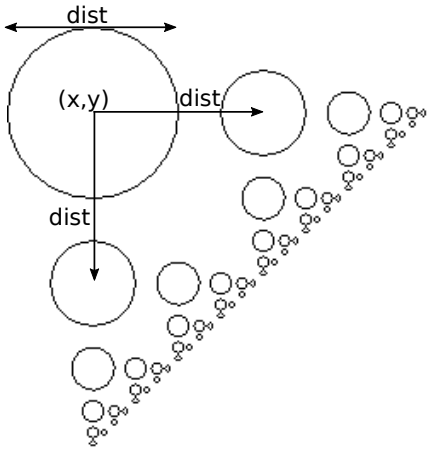
- a circle of diameter dist at the top left, centered at (x, y),
- a half-size pattern, dist units to the right, and
- a half-size pattern, dist units below

If dist is less than 4, the Pattern has only the one circle, with no half-size patterns.

For example, drawPattern(50, 50, 100) should output the following:



The pattern



The pattern with dimensions

```
public void drawPattern(double x, double y, double dist){




}
```

Note: use UI.drawOval(...) to draw a circle.

(Question 3 continued on next page)

(Question 3 continued)

(b) **[5 marks]** Consider the following *increment* method that is given a list of numbers and a value, and returns a new list that is a modified version of the given list. The method also prints out the numbers in the list as it processes them.

```
public List<Double> increment(List<Double> numbers, double value){
    if (source.isEmpty()){
        return new ArrayList<Double>();
    }
    else {
        double num = source.remove(0);
        List<Double> answer = increment(source, value);
        UI.println(num);
        answer.add(num+value);
        return answer;
    }
}
```

Write a non-recursive version of the increment method which uses a for loop instead of recursion, but returns the same list and prints the numbers in the same order as the recursive version.

```
public List<Double> increment(List<Double> source, double value){









}
```

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

Documentation

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

Note: E stands for the type of the 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)
   public void sort((E e1, E e2)->{...});            // cost: O(n log(n)), but O(n) if almost sorted

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