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

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

COMP 103 : Test 6

2021, Feb 12

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
- This test contributes 10% 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. Collections	[10]	
2. Complexity	[10]	
3. Using Stacks	[10]	
	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. Collections **[10 marks]**

For these questions, circle "true" or "false" for each property. (Note some properties may be true for more than one type.)

(a) **[2 marks]** For a Priority Queue, state whether each property is true or false.

true/false	:	Items must be added with a key
true/false	:	Items must be comparable or a comparator must be provided
true/false	:	The first item added is the first item to be removed
true/false	:	The "best" item added is the first item to be removed

(b) **[2 marks]** For a List, state whether each property is true or false.

true/false	:	The collection is not ordered
true/false	:	Items can be added at a specified location
true/false	:	The collection cannot contain any duplicate values
true/false	:	Items can only be removed from one end

(c) **[2 marks]** For a Map, state whether each property is true or false.

true/false	:	A value can be only be removed by specifying its index
true/false	:	The order of values in the collection is unimportant
true/false	:	To remove a value, you must specify its key
true/false	:	The collection cannot contain any duplicate values

(Question 1 continued on next page)

(Question 1 continued)

(d) **[4 marks]** Complete the comparator below, which should sort Card objects by **suit** (largest-to-smallest) and then by **rank** (smallest-to-largest).

Card objects have the following methods:

```
class Card {
    public int getSuit(); // returns the suit (hearts = 0, diamonds = 1, clubs = 2, spades = 3)
    public int getRank()  // returns the rank (A = 1, J = 11, Q = 12, K = 13)
}
```

For example, if you had the four cards:

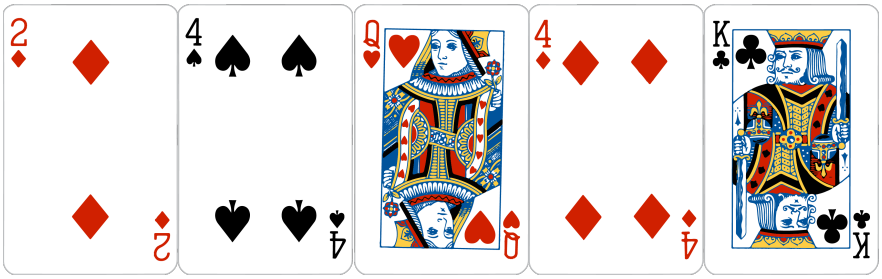

Figure 1: Unsorted Cards

They would be sorted first by suit (into the order Spades, Clubs, Diamonds, Hearts) then by rank (from smallest to largest).

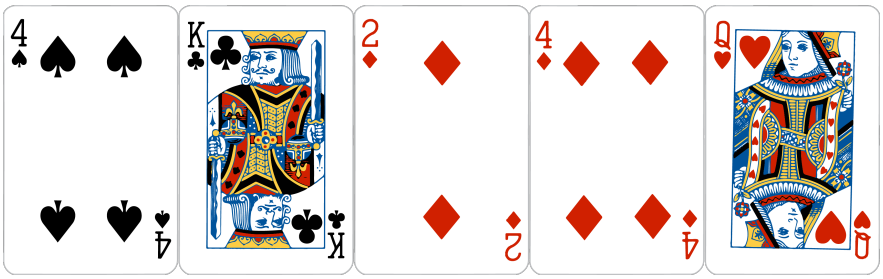

Figure 2: Sorted Cards

```
List<Card> cards = ... // Assume a list of Card objects

Collections.sort(cards, (Card a, Card b) -> {




});
```

Question 2. Complexity **[10 marks]**

(a) **[5 marks]** Work out the cost (in Big-O notation) of the code snippet below by:

- working out the cost of performing each line once.
- working out the number of times each line will be performed.
- computing the total cost.

What is the Big-O cost of the fragment of code below? Assume the size of the list is n.

```
ArrayList<Integer> list = ... // assume a list of n integers

int n = list.size();                                  // cost= O(      ) times=
for (int i = 0; i < n-1 ; i++) {
    for (int j = 0; j < n-i-1; j++) {
        if ( list.get(j) > list.get(j+1)) {           // cost= O(      ) times=
            // swap values
            int temp = list.remove(j);                // cost= O(      ) times=
            list.add(j+1, temp);                      // cost= O(      ) times=
        }
    }
}
 // Total Cost = O(          )
```

(b) **[5 marks]** A program that handles online purchases has a HashMap that contains the mappings between order ids and the order objects.

When the HashMap has 1,000 orders, the program takes about 12 milliseconds to look up a specific order.

If the HashSet had 10,000 orders, how long would you expect the program to take to look up a specific order? Why?

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 3. Using Stacks **[10 marks]**

Suppose you are writing a program for a simple solitaire card game.

The player has a hand of cards which initially contains 5 cards. The rest of the deck is placed as a stack of cards on the table, with only the top card visible.

The program has two fields with the following collections for the hand and the stack of cards:

```
private List<Card> hand;
private Stack<Card> stack;
```

Note: The Card class has two methods that return the suit and the rank of a card.

```
public int getSuit(); // returns the suit (hearts = 0, diamonds = 1, clubs = 2, spades = 3)
public int getRank()  // returns the rank (A = 1, J = 11, Q = 12, K = 13)
```

At each turn, a player has several choices, including:

- Add the top card from the stack to their hand (as long as the stack isn't empty).
- Discard three cards which must have the same suit.
  - These cards are removed from the hand.
  - If one of the three cards has the same number as the card on the top of the stack, then that card is added to the top of the stack, rather than discarded.
  - Discarded cards are removed from the game (ie: they are no longer in the hand or the stack).

Complete the following methods for these two choices. (The methods do not need to redisplay the hand or stack.)

(a) **[4 marks]** Add the top card from the stack to the players hand (if the stack is not empty).

```
/** Take top card on stack (if any) and add to hand */
public void takeFromStack(){




}
```

(Question 3 continued on next page)

(Question 3 continued)

(b) **[6 marks]** Discard (remove) three cards from the hand. The parameter is an array of Card objects that are the cards to be removed.

If any of the removed cards have the same *rank* as the current card on top of the stack, add the removed card to the stack.

```
/** Discard three cards (of same suit) from hand.
    Add card to stack if it has the same number as the top of the stack */
public void discard(Card[] toDiscard){









}
```

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

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
```