

CSE 143 Sample Midterm Exam #2

1. **ArrayList Mystery.** Consider the following method:

```
public static void mystery2(ArrayList<Integer> list) {
    for (int i = list.size() - 1; i > 0; i--) {
        if (list.get(i) < list.get(i - 1)) {
            int element = list.get(i);
            list.remove(i);
            list.add(0, element);
        }
    }
    System.out.println(list);
}
```

Write the output produced by the method when passed each of the following ArrayLists:

List

Output

(a)

[2, 6, 1, 8]

(b)

[30, 20, 10, 60, 50, 40]

(c)

[-4, 16, 9, 1, 64, 25, 36, 4, 49]

- ArrayList Programming.** Write a method `isConsecutive` that accepts an `ArrayList` of integers as a parameter and returns `true` if the list contains a sequence of consecutive integers and `false` otherwise. Consecutive integers are integers that come one after the other in ascending order, as in 5, 6, 7, 8, 9, etc. For example, if a variable called `list` stores the values `[3, 4, 5, 6, 7, 8, 9]`, then the call of `list.isConsecutive()` should return `true`. If the list instead stored `[3, 4, 5, 6, 7, 12, 13]` then the call should return `false` because the numbers 7 and 12 are not consecutive. The list `[3, 2, 1]` might seem to be consecutive, but the elements appear in reverse order, so the method would return `false` in that case. Any list with fewer than two values should be considered to be consecutive. You may assume that the list passed is not `null`.

4. **Collections Programming.** Write a method `union` that accepts two maps (whose keys and values are both integers) as parameters, and returns a new map that represents a merged union of the two original maps. For example, if two maps `m1` and `m2` contain these pairs:

```
{7=1, 18=5, 42=3, 76=10, 98=2, 234=50}    m1
{7=2, 11=9, 42=-12, 98=4, 234=0, 9999=3}   m2
```

The call of `union(m1, m2)` should return a map that contains the following pairs:

```
{7=3, 11=9, 18=5, 42=-9, 76=10, 98=6, 234=50, 9999=3}
```

The "union" of two maps *m1* and *m2* is a new map that contains every key from *m1* and every key from *m2*. Each value stored in your "union" map should be the sum of the corresponding value(s) for that key in *m1* and *m2*, or if the key exists in only one of the two maps, that map's corresponding value should be used. For example, in the maps above, the key 98 exists in both maps, so the result contains the sum of its values from the two maps, $2 + 4 = 6$. The key 9999 exists in only one of the two maps, so its sole value of 3 is stored as its value in the result map.

You may assume that the maps passed are not `null`, though either map (or both) could be empty. Though the pairs are shown in sorted order by key above, you should not assume that the maps passed to you store their keys in sorted order, and the map you return does not need to store its keys in any particular order.

You may create one collection of your choice 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 maps passed to your method. For full credit your code must run in less than $O(n^2)$ time where n is the combined number of pairs in the two maps.

6. **Linked List Programming.** Write a method `removeLast` that could be added to the `LinkedList` class that removes the last occurrence (if any) of a given integer from the list of integers. For example, suppose that a variable named `list` stores this sequence of values:

```
[3, 2, 3, 3, 19, 8, 3, 43, 64, 1, 0, 3]
```

If we repeatedly make the call of `list.removeLast(3)`;, then the list will take on the following sequence of values after each call:

```
after first call: [3, 2, 3, 3, 19, 8, 3, 43, 64, 1, 0]
after second call: [3, 2, 3, 3, 19, 8, 43, 64, 1, 0]
after third call: [3, 2, 3, 19, 8, 43, 64, 1, 0]
after fourth call: [3, 2, 19, 8, 43, 64, 1, 0]
after fifth call: [2, 19, 8, 43, 64, 1, 0]
after sixth call: [2, 19, 8, 43, 64, 1, 0]
```

Notice that once we reach a point where no more 3's occur in the list, calling the method has no effect.

Assume that we are adding this method to the `LinkedList` class as seen in lecture and as shown below. You may not call any other methods of the class to solve this problem.

```
public class LinkedList {
    private ListNode front;

    methods
}
```

7. **Recursive Tracing.** For each call to the following method, indicate what output is produced:

```
public static void mystery(int x, int y) {  
    if (x > y) {  
        System.out.print("*");  
    } else if (x == y) {  
        System.out.print("=" + y + "=");  
    } else {  
        System.out.print(y + " ");  
        mystery(x + 1, y - 1);  
        System.out.print(" " + x);  
    }  
}
```

Call	Output
mystery(3, 3);	
mystery(5, 1);	
mystery(1, 5);	
mystery(2, 7);	
mystery(1, 8);	

8. **Recursive Programming.** Write a recursive method `repeat` that accepts a string `s` and an integer `n` as parameters and that returns a string consisting of `n` copies of `s`. For example:

Call	Value Returned
<code>repeat("hello", 3)</code>	<code>"hellohellohello"</code>
<code>repeat("this is fun", 1)</code>	<code>"this is fun"</code>
<code>repeat("wow", 0)</code>	<code>""</code>
<code>repeat("hi ho! ", 5)</code>	<code>"hi ho! hi ho! hi ho! hi ho! hi ho! "</code>

You should solve this problem by concatenating strings using the `+` operator. String concatenation is an expensive operation, so it is best to minimize the number of concatenation operations you perform. For example, for the call `repeat("foo", 500)`, it would be inefficient to perform 500 different concatenation operations to obtain the result. Most of the credit will be awarded on the correctness of your solution independent of efficiency. The remaining credit will be awarded based on your ability to minimize the number of concatenation operations performed.

Your method should throw an `IllegalArgumentException` if passed a negative value for `n`. You are not allowed to construct any structured objects other than `Strings` (no `array`, `List`, `Scanner`, etc.) and you may not use any loops to solve this problem; you must use recursion.

Solution Key

1.

List

(a) [2, 6, 1, 8]

(b) [30, 20, 10, 60, 50, 40]

(c) [-4, 16, 9, 1, 64, 25, 36, 4, 49]

Output

[1, 2, 6, 8]

[10, 30, 40, 20, 60, 50]

[-4, 1, 25, 4, 16, 9, 64, 36, 49]

2.

```
public static boolean isConsecutive(ArrayList<Integer> list) {  
    for (int i = 0; i < list.size() - 1; i++) {  
        if (list.get(i) + 1 != list.get(i + 1)) {  
            return false;  
        }  
    }  
    return true;  
}
```

3.

```
public static void interleave(Queue<Integer> q) {
    if (q.size() % 2 != 0) {
        throw new IllegalArgumentException();
    }
    Stack<Integer> s = new Stack<Integer>();
    int halfSize = q.size() / 2;
    for (int i = 0; i < halfSize; i++) {
        s.push(q.remove());
    }
    while (!s.isEmpty()) {    // s2q(s, q);
        q.add(s.pop());
    }
    for (int i = 0; i < halfSize; i++) {
        q.add(q.remove());
    }
    for (int i = 0; i < halfSize; i++) {
        s.push(q.remove());
    }
    while (!s.isEmpty()) {
        q.add(s.pop());
        q.add(q.remove());
    }
}
```

4. Two solutions are shown.

```
public static Map<Integer, Integer> union(Map<Integer, Integer> m1,
                                         Map<Integer, Integer> m2) {
    Map<Integer, Integer> result = new TreeMap<Integer, Integer>();
    for (int key : m1.keySet()) {
        result.put(key, m1.get(key));
    }
    for (int key : m2.keySet()) {
        if (result.containsKey(key)) {
            result.put(key, result.get(key) + m2.get(key));
        } else {
            result.put(key, m2.get(key));
        }
    }
    return result;
}
```

```
public static Map<Integer, Integer> union(Map<Integer, Integer> m1,
                                         Map<Integer, Integer> m2) {
    Map<Integer, Integer> result = new TreeMap<Integer, Integer>();
    for (int key : m1.keySet()) {
        int value = m1.get(key);
        if (m2.containsKey(key)) {
            int value2 = m2.get(key);
            value += value2;
        }
        result.put(key, value);
    }
    for (int key : m2.keySet()) {
        int value = m2.get(key);
        if (!result.containsKey(key)) {
            result.put(key, value);
        }
    }
    return result;
}
```

```

5. ListNode list2 = list.next.next;           // list2 -> 3
   list.next.next.next.next = list.next;     // 4 -> 2
   ListNode temp = list;                     // temp -> 1
   list = list.next.next.next;               // list -> 4
   list2.next = temp;                         // 3 -> 1
   list.next.next = null;                    // 2 /
   list2.next.next = null;                   // 1 /

```

```

6. public void removeLast(int n) {
    if (front != null) {
        ListNode current = front;
        ListNode spot = null;
        while (current.next != null) {
            if (current.next.data == n) {
                spot = current;
            }
            current = current.next;
        }
        if (spot != null) {
            spot.next = spot.next.next;
        } else if (front.data == n) {
            front = front.next;
        }
    }
}

```

7.

Call	Output
mystery(3, 3);	=3=
mystery(5, 1);	*
mystery(1, 5);	5 4 =3= 2 1
mystery(2, 7);	7 6 5 * 4 3 2
mystery(1, 8);	8 7 6 5 * 4 3 2 1

```

8. public static String repeat(String s, int n) {
    if (n < 0) {
        throw new IllegalArgumentException();
    } else if (n == 0) {
        return "";
    } else if (n == 1) {
        return s;
    } else if (n % 2 == 0) {
        String temp = repeat(s, n / 2);
        return temp + temp;
        // alternative without temp: return repeat(s + s, n / 2);
    } else {
        return s + repeat(s, n - 1);
    }
}

```