

CSE 143 SAMPLE MIDTERM SOLUTION

1. (5 points) In some methods, you wrote code to check if a certain precondition was held. If the precondition did not hold, then you threw an exception. This leads to robust code by catching client code misusing your methods. This seems like a great idea, so when would you **NOT** want to check for the precondition in a method? Why wouldn't you? (You may use no more than 30 words.)

Answers (or close derivatives thereof) we accepted:

- It would be too inefficient to do so.
- The precondition is ensured by the other methods in the class.

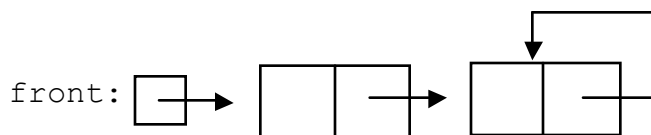
2. (5 points) Recall the definition of the `ListNode` class:

```
public class ListNode {
    int data;
    ListNode next;
}
```

We can print a linked list using the following method:

```
public static void print(ListNode front) {
    ListNode current = front;
    while (current != null) {
        System.out.println(current.data + " ");
        current = current.next;
        if (current == front) {
            break;
        }
    }
}
```

Starting from the node pointed to by `front`, draw one or more nodes and link them together in such a way that the call `print(front)` will **not** “behave” properly.



3. (8 points) Consider the following method:

```
public void mystery(int n) {
    if (n >= 7) {
        System.out.println(n);
    } else {
        System.out.print(n);
        System.out.print(n);
        mystery(n + 7);
    }
}
```

For each call below, indicate what output is produced by the method. If the call results in infinite recursion, write out the first 5 characters followed by "...".

Method Call	Output Produced
mystery(-1)	<u>-1-16613</u>
mystery(711)	<u>711</u>
mystery(0)	<u>007</u>
mystery(-5)	<u>-5-5229</u>

4. (8 points) Consider the following method:

```
public void mystery(int n) {
    if (n == 0) {
        System.out.print(n);
    } else {
        System.out.print("(");
        mystery(n - 1);
        System.out.print(n);
        mystery(n - 1);
        System.out.print(")");
    }
}
```

For each call below, indicate what output is produced by the method. If the call results in infinite recursion, write out the first 5 characters followed by "...".

Method Call	Output Produced
mystery(2)	<u>((010)2(010))</u>
mystery(0)	<u>0</u>
mystery(-1)	<u>(((..._</u>
mystery(1)	<u>(010)</u>

5. (20 points) A palindrome reads the same backward or forward. The number 12321 is a palindrome, because if you start from the left side or the right side, it reads exactly the same way. Write a method `isPalindrome` that takes an integer array `nums` and returns `true` or `false` if the array of numbers constitute a palindrome in the sense that the first element matches the last element; the second element matches the second-to-last element, and so on. Do **NOT** use recursion.

Examples where `isPalindrome` would return `true`:

123	2	24	2	123
-----	---	----	---	-----

123	2	214	99	99	214	2	123
-----	---	-----	----	----	-----	---	-----

Examples where `isPalindrome` would return `false`:

123	2	24	1	123
-----	---	----	---	-----

123	2	214	99	98	214	2	125
-----	---	-----	----	----	-----	---	-----

```
public boolean isPalindrome(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] != nums[nums.length - 1 - i]) {
            return false;
        }
    }
    return true;
}
```

6. (10 points) What is the running time of this method? Circle one: **O(1)** O(n)

```
public int add100(int[] array) {
    if (array.length < 100) {
        return 0;
    }

    int sum = 0;
    for (int i = 0; i < 100; i++) {
        sum += array[i];
    }
    return sum;
}
```

Explain your answer in 30 words or less.

We are interested in the running time as the array gets larger and larger. After the array exceeds a size of 100, the size of the array is irrelevant, and a constant number of steps is needed to solve the problem.

7. (20 points) Recall the definition of the `ListNode`:

```
public class ListNode {
    int data;
    ListNode next;
}
```

Assume that we define a class `SortedLinkedList` that is similar to `LinkedList`, except that the data stored in the nodes are in sorted order:

```
public class SortedLinkedList {
    private ListNode front;

    <methods>
}
```

Write a method `mode(void)` for the `SortedLinkedList` class that returns the mode of the numbers in the list. The mode of a set of numbers is the number that appears the most frequently. If there is a tie, return any of the numbers that share the highest frequency. If the list is empty, throw an `IllegalStateException`.

For example, if the list contained `[1,1,1,4,4,4,4,4,5,6,6,70,99,99]`, then a call to `mode()` would return 4, since it appears the most frequently (six times). If the list contained `[1,4,4,7,7,11,25,25,99]`, then your method can return either 4, 7, or 25, because they all share the highest frequency (two times).

```
public int mode(void) {
    if (front == null) {
        throw new IllegalStateException();
    }

    int value = front.data;
    int maxRun = 1;
    int currentRun = 1;

    ListNode current = front.next;
    ListNode last = front;
    while (current != null) {
        if (current.data == last.data) {
            currentRun++;
            if (currentRun > maxRun) {
                maxRun = currentRun;
                value = current.data;
            }
        } else {
            currentRun = 1;
        }
        last = current;
        current = current.next;
    }
    return value;
}
```

Stack Interface

```
// Interface Stack defines a set of operations for manipulating a
// LIFO (Last In First Out) structure that can be used to store
// objects.
```

```
public interface Stack<E> {
    // post: given value is pushed onto the top of the stack
    public void push(E value);

    // pre : !isEmpty()
    // post: removes and returns the value at the top of the stack
    public E pop();

    // post: returns true if the stack is empty, false otherwise
    public boolean isEmpty();

    // post: returns the current number of element in the stack
    public int size();
}
```

Queue Interface

```
// Interface Queue defines a set of operations for manipulating a
// FIFO (First In First Out) structure that can be used to store
// objects.
```

```
public interface Queue<E> {
    // post: given value inserted at the end of the queue
    public void enqueue(E value);

    // pre : !isEmpty()
    // post: removes and returns the value at the front of the queue
    public E dequeue();

    // post: returns true if the queue is empty, false otherwise
    public boolean isEmpty();

    // post: returns the current number of element in the queue
    public int size();
}
```

8. (30 points) Write a method `pushNumTimes` that takes a `Queue q` as an argument and returns a `Stack`. Assume that `q` is storing `Integer` objects. Each integer value `i` at position `n` (where the object at the front of the queue has position 1) in the queue will be replaced by an `Integer` object with an integer value of $(i * n)$ on the stack. The contents of the queue do **not** have to be preserved.

For illustration purposes, let a queue's contents be represented as a list of numbers, where the leftmost number represents the front of the queue; let a stack's contents be represented as a list of numbers where the leftmost number represents the top of the stack. Suppose `q` initially contained `[6, 4, 5, 3]`, then the stack returned from a call to `pushNumTimes(q)` will have `[6, 8, 15, 12]`.

For your convenience, the `Queue` and `Stack` interfaces are on the previous page. The names of the classes that implement those interfaces are `LinkedList` and `ArrayStack`, respectively. Both classes have constructors that take no arguments.

If `q` initially contained:

6	4	5	3
---	---	---	---

position: 1 2 3 4

Then the returned stack would contain:

6
8
15
12

```
public Stack<Integer> pushNumTimes(Queue<Integer> q) {
    Stack<Integer> s = new ArrayStack<Integer>();
    int counter = 1;
    while (!queue.isEmpty()) {
        int n = queue.dequeue();
        s.push(n * counter);
        counter++;
    }
    while (!s.isEmpty()) {
        q.enqueue(s.pop());
    }
    while (!q.isEmpty()) {
        s.push(q.dequeue());
    }
    return s;
}
```

9. (20 points) Consider the following definitions:

```
public class Apple extends Date {
    public void method2() {
        System.out.println("Apple2");
    }
}

public class Banana {
    public void method2() {
        System.out.println("Banana2");
    }
}

public class Cherry extends Date {
    public void method2() {
        System.out.println("Cherry2");
    }

    public void method1() {
        super.method1();
        System.out.println("Cherry1");
    }
}

public class Date extends Banana {
    public void method1() {
        System.out.println("Date1");
    }

    public void method2() {
        System.out.println("Date2");
        method1();
    }
}
```

And assuming the following variables have been defined:

```
Object var1 = new Cherry();
Banana var2 = new Date();
Banana var3 = new Cherry();
Apple var4 = new Apple();
```


In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the line breaks with slashes as in "a/b/c" to indicate three lines of output with "a" followed by "b" followed by "c". If the statement causes an error, fill in the right-hand column with either the phrase "compiler error" or "runtime error" to indicate when the error would be detected.

Statement	Output
<code>var1.method1 ();</code>	<u>compiler error</u>
<code>var2.method1 ();</code>	<u>compiler error</u>
<code>var3.method1 ();</code>	<u>compiler error</u>
<code>var4.method1 ();</code>	<u>Date1</u>
<code>((Cherry) var2).method1 ();</code>	<u>runtime error</u>
<code>((Date) var2).method2 ();</code>	<u>Date2 / Date1</u>
<code>((Date) var1).method2 ();</code>	<u>Cherry2</u>
<code>((Cherry) var1).method1 ();</code>	<u>Date1 / Cherry1</u>
<code>((Cherry) var4).method1 ();</code>	<u>compiler error</u>
<code>((Date) var3).method2 ();</code>	<u>Cherry2</u>

10. (20 points) **Using recursion**, write a method `times` that takes two integers `a` and `b` as parameters and returns their product, *i.e.*, $a * b$. You are to multiply them by using a series of additions. You may use `+`, `-`, comparison operators (`>`, `>=`, `<`, `<=`, `==`, `!=`), and Boolean operators (`&&`, `||`). You may NOT use `*`, `/`, any loops (*i.e.*, `for` or `while`) or any method from the Java library.

You may find the following equality useful: $a * b = b + (a - 1) * b$

For at most $\frac{3}{4}$ credit (*i.e.*, 15 points), you may assume that both arguments are non-negative. If you would like that option, sign here: _____

```
public int times(int a, int b) {
    if (a < 0) {
        return -times(-a, b);
    }
    if (a == 0) {
        return 0;
    }
    return b + times(a-1, b);
}
```