# CSE143 Midterm
Autumn 2019

Name of Student: ______________________________________________________

Section (e.g., AA):_____________________ Student Number: ________________

The exam is divided into six questions with the following points:

| # | Problem Area | Points | Score |
|---|---|---|---|
| 1 | Recursive Tracing | 15 | _____ |
| 2 | Recursive Programming | 15 | _____ |
| 3 | ListNodes | 15 | _____ |
| 4 | ArrayIntList Programming | 10 | _____ |
| 5 | Collections | 20 | _____ |
| 6 | Stacks/Queues | 25 | _____ |
| | Total | 100 | _____ |

Do not begin work on this exam until instructed to do so. Any student who starts early or who continues to work after time is called will receive a 10 point penalty.

This is a closed-book/closed-note exam. Space is provided for your answers. There is a "cheat sheet" at the end that you can use as scratch paper. You are not allowed to access any of your own papers during the exam. You are NOT to use any electronic devices while taking the test, including calculators. Anyone caught using an electronic device will receive a 10 point penalty.

The exam is not, in general, graded on style and you do not need to include comments. For the stack/queue and collections questions, however, you are expected to use generics properly and to declare variables using interfaces when possible. You may only use the methods on the cheat sheet for the data structures listed. You are not allowed to use programming constructs like break, continue, or returns from a void method on this exam.

Do not abbreviate code, such as "ditto" marks or dot-dot-dot ... marks.

You are allowed to ask for scratch paper to use as additional space when writing answers, but you must indicate on the original page for the problem that part of the solution is on scratch paper. Failure to do so may result in your work on scratch paper not being graded.

If you finish the exam early, please hand your exam to the instructor and exit quietly through the front door.

Initial here to indicate you have read and agreed to these rules: ____________________

1. **Recursive Tracing, 15 points:** Consider the following method:

```
public void mystery(int n) {
    System.out.print("+");
    if (n >= 10) {
        mystery(n / 10);
    }

    if (n % 2 == 0) {
        System.out.print("-");
    } else {
        System.out.print("*");
    }
}
```

For each call below, indicate what output is produced:

| **Method Call** | **Output Produced** |
|---|---|
| mystery(5) | ________________________________________ |
| mystery(15) | ________________________________________ |
| mystery(304) | ________________________________________ |
| mystery(9247) | ________________________________________ |
| mystery(43269) | ________________________________________ |

2. **Recursive Programming, 15 points:** Write a recursive method called **doubleDigit** that takes an integer n and a digit d as parameters and that returns the integer obtained by replacing all occurrences of d in n with two of that digit. For example, doubleDigit(3797, 7) would return 377977 because the two occurrences of the digit 7 are replaced with two of that digit. The table below includes more examples.

| Method Call | Value Returned | Method Call | Value Returned |
|---|---|---|---|
| doubleDigit(2, 2) | 22 | doubleDigit(2001, 0) | 200001 |
| doubleDigit(0, 0) | 0 | doubleDigit(12345, 6) | 12345 |
| doubleDigit(8, 6) | 8 | doubleDigit(72773, 7) | 77277773 |
| doubleDigit(55, 2) | 55 | doubleDigit(3445, 5) | 34455 |
| doubleDigit(33, 3) | 3333 | doubleDigit(54224, 4) | 5442244 |
| doubleDigit(-101, 1) | -11011 | doubleDigit(-624243, 4) | -62442443 |
| doubleDigit(323, 3) | 33233 | doubleDigit(5340909, 0) | 534009009 |

Notice that the number can be negative. Your method should throw an IllegalArgumentException if the value of d is not a 1-digit number (i.e., not between 0 and 9 inclusive). You are not allowed to construct any structured objects to solve this problem (no string, array, ArrayList, StringBuilder, Scanner, etc) and you may not use a while loop, for loop or do/while loop to solve this problem; you must use recursion.

3. **Linked Lists, 15 points**: Fill in the "code" column in the following table providing a solution that will turn the "before" picture into the "after" picture by modifying links between the nodes shown. You are not allowed to change any existing node's data field value and you are not allowed to construct any new nodes, but you are allowed to declare and use variables of type ListNode (often called "temp" variables). You are limited to at most two variables of type ListNode for each of the four subproblems below.

You are writing code for the ListNode class discussed in lecture:

```
    public class ListNode {
        public int data;       // data stored in this node
        public ListNode next;  // link to next node in the list

        <constructors>
    }
```

As in the lecture examples, all lists are terminated by null and the variables p and q have the value null when they do not point to anything.

| before | after | code |
|---|---|---|
| p->[1]<br><br>q->[2]->[3] | p<br><br>q->[2]->[3]->[1] | |
| p->[1]->[2]->[3]<br><br>q | p->[2]->[3]<br><br>q->[1] | |
| p->[1]->[2]<br><br>q->[3]->[4] | p->[2]<br><br>q->[4]->[3]->[1] | |
| p->[1]->[2]<br><br>q->[3]->[4]->[5] | p->[4]->[2]->[3]<br><br>q->[5]->[1] | |

4. **ArrayIntList Programming, 10 points:** Write a method called **removeRange** that takes as parameters a "from" index (inclusive) and "to" index (exclusive) and that removes the values in the given range from a list of integers. For example, if a variable called list stores the following values:

```
    [8, 13, 17, 4, 9, 12, 98, 41, 7, 23, 0, 92]
```

If you were to call list.removeRange(3, 8), then the values starting at index 3 (value 4) and up to but not including index 8 (value 7) are removed:

```
    [8, 13, 17, 4, 9, 12, 98, 41, 7, 23, 0, 92]
               |  to remove   |
               +--------------+
```

thus, the list should store the following values after the call:

```
    [8, 13, 17, 7, 23, 0, 92]
```

Your method should throw an IllegalArgumentException if either index is not legal. Start has to be less than or equal to stop and both have to be between 0 and the size of the list inclusive. If start equals stop, the method should not change the list because the specified sublist is empty.

You are writing a method for the ArrayIntList class discussed in lecture:

```
    public class ArrayIntList {
        private int[] elementData; // list of integers
        private int size;          // current # of elements in the list
        <methods>
    }
```

You may not call any other methods of the ArrayIntList class to solve this problem, you are not allowed to define any auxiliary data structures (no array, String, ArrayList, etc), and your solution must run in O(n) time.

5. **Collections Programming, 20 points:** Write a method called **whatToCook** that takes a map of recipes and a set of ingredients in your kitchen as parameters. The recipes map uses recipe names as keys (strings) and has sets of ingredients as values (each ingredient is a string). Your method should return a set of all the recipe names in the recipes map that can be made with the ingredients in your kitchen (case insensitive).

For example, a variable called recipes might contain the following map:

```
{"cacio e pepe"=["spaghetti", "cheese", "pepper"], "sandwich"=["bread", "Cheese"],
 "basic pasta"=["sPaGHetTi"], "just some air"=[],
 "ratatouille"=["eggplant", "zucchini", "onion", "tomato"]}
```

and a variable called pantry might contain the following ingredients

```
["cheese", "pepper", "spaghetti"]
```

In this example, the recipes map indicates that to make a "sandwich" you need "bread" and "cheese" while to make "basic pasta" you only need "spaghetti". Also, in this example, the pantry set indicates you only have "cheese", "pepper", and "spaghetti" in your pantry. Suppose that the following call is made:

```
whatToCook(recipes, pantry);
```

This call would return the set

```
["basic pasta", "just some air", "cacio e pepe"]
```

since these are the only recipes that can be made with the ingredients in your kitchen. For example, you can make "basic pasta" because you have "spaghetti", but you can't make a "sandwich" because you don't have "bread". The recipe "just some air" is a valid recipe in the result in this example since there is no ingredient in that recipe that you don't have in the pantry.

The returned set should favor lookup speed over presenting in sorted order. It's possible for the returned set to be empty. If the map of recipes is empty, your method should throw an IllegalArgumentException.

You may assume that the given maps are not null and you may assume none of the contents of the maps contain null values. The method should not modify the provided map or any of the structures it references.

Use the space on the next page to write your answer. Do not write your solution on this page.

6. **Stacks/Queues, 25 points:** Write a method named **expand** that accepts two stacks of integers of the same length. For each pair of numbers between the stacks, we will make copies of the values in the first stack, using the values in the second stack to determine the number of copies to make. This is clearer with an example.

Assume we had the following stacks, s1 and s2. In the rest of this description, bottom is denoted with a B while the top is denoted with a T.

```
   s1:     B [4, 5, 6] T                          s2:     B [1, 2, 3] T
```

After the call expand(s1, s2), s1 and s2 should store the following values.

```
   s1:     B [4, 5, 5, 6, 6, 6] T                 s2:     B [1, 2, 3] T
```

The pairs are determined by being at the same position in each stack. In this example, the pairs are (6, 3), (5, 2), (4, 1) since they are in the same position in each stack. For each pair, we use the value in the second stack to determine how many values of the first stack to make; for the top pair, this means there should be one 4 in the result. This is why the result above shows the method modified s1 to store one 4, two 5s, and three 6s. The method should only modify the first stack and the second one is unchanged.

Alternatively, assume we had the original s1 and s2 but made the call expand(s2, s1). In this case the pairs will be (3, 6), (2, 5), (1, 4). This means the method should modify s2 to have four 1s, five 2s, and six 3s. The resulting state of both stacks is shown below (s1 is unchanged in this example since it is the second parameter).

```
   s1:   B [4, 5, 6] T            s2:  B [1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3] T
```

The values should appear in the stack in the same relative order they appeared originally. The second stack parameter should not be modified after the call finishes. You may assume that the stacks passed to your method are not null and they do not contain any null values. You may assume all values in the second stack will be at least 1.

Your method should throw an IllegalArgumentException if the stacks are not the same length.

For full credit, obey the following restrictions in your solution. A solution that does not follow these restrictions can get some partial credit.

* You may use one queue as auxiliary storage. You may not use other structures (arrays, lists, etc.), but you can have as many simple variables as you like.
* Use the Queue interface and Stack/LinkedList classes discussed in class.
* Use stacks/queues in stack/queue-like ways only. Do not use index-based methods such as get, search, or set, or for-each loops or iterators. You may call add, remove, push, pop, peek, isEmpty, and size.
* You may not solve the problem recursively.

You have access to the following two methods and may call them as needed to help you solve the problem:

```
    public void s2q(Stack<Integer> s, Queue<Integer> q) {
        while (!s.isEmpty()) {
            q.add(s.pop());
        }
    }

    public void q2s(Queue<Integer> q, Stack<Integer> s) {
        while (!q.isEmpty()) {
            s.push(q.remove());
        }
    }
```

# ^_^ CSE 143 MIDTERM EXAM CHEAT SHEET ^_^

*(DO NOT WRITE ANY WORK YOU WANT GRADED ON THIS SHEET. IT WILL NOT BE GRADED)*

## Examples of Constructing Various Collections

```
List<Integer> list = new ArrayList<Integer>();
Queue<Double> queue = new LinkedList<Double>();
Stack<String> stack = new Stack<String>();
Set<String> words = new HashSet<String>();
Map<String, Integer> counts = new TreeMap<String, Integer>();
```

## Methods Found in ALL collections (Lists, Stacks, Queues, Sets, Maps)

| | |
|---|---|
| `equals(`**collection**`)` | returns `true` if the given other collection contains the same elements |
| `isEmpty()` | returns `true` if the collection has no elements |
| `size()` | returns the number of elements in the collection |
| `toString()` | returns a string representation such as `"[10, -2, 43]"` |

## Methods Found in both Lists and Sets (ArrayList, LinkedList, HashSet, TreeSet)

| | |
|---|---|
| `add(`**value**`)` | adds value to collection (appends at end of list) |
| `addAll(`**collection**`)` | adds all the values in the given collection to this one |
| `contains(`**value**`)` | returns `true` if the given value is found somewhere in this collection |
| `iterator()` | returns an Iterator object to traverse the collection's elements |
| `clear()` | removes all elements of the collection |
| `remove(`**value**`)` | finds and removes the given value from this collection |
| `removeAll(`**collection**`)` | removes any elements found in the given collection from this one |
| `retainAll(`**collection**`)` | removes any elements *not* found in the given collection from this one |

## `List<E>` Methods

| | |
|---|---|
| `add(`**index**, **value**`)` | inserts given value at given index, shifting subsequent values right |
| `indexOf(`**value**`)` | returns first index where given value is found in list (-1 if not found) |
| `get(`**index**`)` | returns the value at given index |
| `lastIndexOf(`**value**`)` | returns last index where given value is found in list (-1 if not found) |
| `remove(`**index**`)` | removes/returns value at given index, shifting subsequent values left |
| `set(`**index**, **value**`)` | replaces value at given index with given value |

## `Stack<E>` Methods

| | |
|---|---|
| `pop()` | removes the top value from the stack and returns it; `pop` throw an `EmptyStackException` if the stack is empty |
| `push(`**value**`)` | places the given value on top of the stack |
| `peek()` | returns the top from the stack without removing it; throws a `EmptyStackException` if the stack is empty |

## `Queue<E>` Methods

| | |
|---|---|
| `add(`**value**`)` | places the given value at the back of the queue |
| `remove()` | removes the value from the front of the queue and returns it; throws a `NoSuchElementException` if the queue is empty |
| `peek()` | returns the value at the front of the queue without removing it; throws a `NoSuchElementException` if the queue is empty |

# ^_^ CSE 143 MIDTERM EXAM CHEAT SHEET ^_^

*(DO NOT WRITE ANY WORK YOU WANT GRADED ON THIS SHEET. IT WILL NOT BE GRADED)*

## `Map<K, V>` Methods

| | |
|---|---|
| `containsKey(`**key**`)` | `true` if the map contains a mapping for the given key |
| `get(`**key**`)` | the value mapped to the given key (`null` if none) |
| `keySet()` | returns a `Set` of all keys in the map |
| `put(`**key**`, `**value**`)` | adds a mapping from the given key to the given value |
| `putAll(`**map**`)` | adds all key/value pairs from the given map to this map |
| `remove(`**key**`)` | removes any existing mapping for the given key |
| `toString()` | returns a string such as `"{a=90, d=60, c=70}"` |
| `values()` | returns a `Collection` of all values in the map |

## `Iterator<E>` Methods

| | |
|---|---|
| `hasNext()` | returns `true` if there is another element in the iterator |
| `next()` | returns the next value in the iterator and progresses the iterator forward one element |
| `remove()` | removes the previous value returned by the next call. Can only be called once after each call to `next()` |

## `String` Methods

| | |
|---|---|
| `charAt(`**i**`)` | the character in this String at a given index |
| `contains(`**str**`)` | `true` if this String contains the other's characters inside it |
| `endsWith(`**str**`)` | `true` if this String ends with the other's characters |
| `equals(`**str**`)` | `true` if this String is the same as *str* |
| `equalsIgnoreCase(`**str**`)` | `true` if this String is the same as *str*, ignoring capitalization |
| `indexOf(`**str**`)` | first index in this String where given String begins (-1 if not found) |
| `lastIndexOf(`**str**`)` | last index in this String where given String begins (-1 if not found) |
| `length()` | number of characters in this String |
| `isEmpty()` | `true` if this String is the empty string |
| `startsWith(`**str**`)` | `true` if this String begins with the other's characters |
| `substring(`**i**`, `**j**`)` | characters in this String from index *i* (inclusive) to *j* (exclusive) |
| `Substring(`**i**`)` | characters in this String from index *i* (inclusive) to the end |
| `toLowerCase(), toUpperCase()` | a new String with all lowercase or uppercase letters |

## `Math` Methods

| | |
|---|---|
| `abs(`**x**`)` | returns the absolute value of `x` |
| `max(`**x**`, `**y**`)` | returns the larger of `x` and `y` |
| `min(`**x**`, `**y**`)` | returns the smaller of `x` and `y` |
| `pow(`**x**`, `**y**`)` | returns the value of `x` to the `y` power |
| `random()` | returns a random number between `0.0` and `1.0` |
| `round(`**x**`)` | returns `x` rounded to the nearest integer |