CSE 123 Summer 2024 Final Exam

Name of Student: ____

Section (e.g., AA):____________________ Student Number (7 digits):____________________

Do not turn the page until you are instructed to do so.

Rules/Guidelines:

- You must not begin working before time begins, and you must stop working **promptly** when time is called. Any modifications to your exam (writing *or* erasing) before time begins or after time is called will result in a penalty.
- You are allowed one page of notes, no larger than 8.5 x 11 inches. You may not access any other resources or use any electronic devices (including calculators, phones, or smart watches, among others) during the exam. Using unauthorized resources or devices will result in a penalty.
- In general, you are limited to Java concepts or syntax covered in class. You may not use `break`, `continue`, a `return` from a `void` method, `try/catch`, or Java 8 stream/functional features.
- You are limited to the standard Java classes and methods listed on the provided reference sheet. You do not need to write import statements.
- If you abandon one answer and write another, ***clearly cross out*** the answer(s) you do not want graded and ***draw a circle or box*** around the answer you do want graded. When in doubt, we will grade the answer that appears in the space indicated, and the first such answer if there is more than one.
- If you require scratch paper, raise your hand and we will bring some to you.
- If you write an answer on scratch paper, please ***write your name and clearly label*** which question you are answering on the scratch paper, and ***clearly indicate*** on the question page that your answer is on scratch paper. Staple all scratch paper you want graded to the ***end*** of the exam before turning in.
- Answers must be written as proper Java code. Pseudocode or comments will not be graded.
- The exam is not graded on code quality. You are not required to include comments.
- You are also allowed to abbreviate `System.out.print` and `System.out.println` as `S.o.p` and `S.o.pln` respectively. You may **NOT** use any other abbreviations.

ABIDE BY ANY AND ALL SENTENCE LIMITS PER PROBLEM. NOT DOING SO WILL RESULT IN DEDUCTIONS

Grading:

- Each problem will receive a single E/S/N grade.
- Minor syntax errors will be ignored as long as it is unambiguous what was intended (e.g. forgetting a semicolon, misspelling a variable name where there is only one close option). Major syntax errors, or errors where it is unclear what was intended, may have an impact on your grade.

Advice:

- Read all questions carefully. Be sure you understand the question *before* you begin your answer.
- The questions are not necessarily in order of difficulty. Be sure you at least attempt every question.
- Write clearly and legibly. We cannot award credit for answers we cannot read.
- If you have questions, raise your hand to ask. The worst that can happen is we will say "I can't answer that."
- Ask questions as soon as you have them. Do not wait until you have several questions.

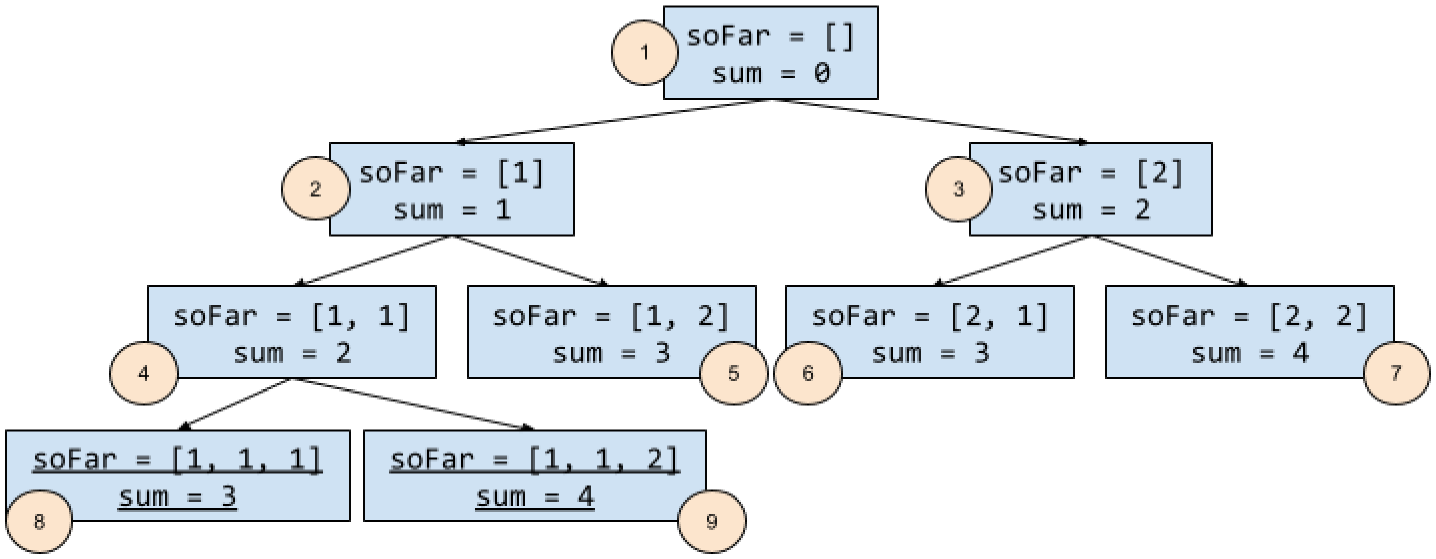
Initial here to indicate you have read and agreed to these rules:

1. Recursive Backtracking

Part A: Decision Trees

Consider the following decision tree for a recursive backtracking problem using a `Stack<Integer>` as an accumulator. The desired results that are printed to the console are underlined.

Note that each of the squares within the diagram is attached to a circle containing a unique identifier which will be used to answer the following question.



Based on the diagram, correctly locate nodes corresponding to each component of the general recursive backtracking pattern (*base case, dead-end case, recursive case*) and copy their identifiers into the appropriate boxes below. While there may be multiple correct identifiers, you only need to provide **one** for each box.

1. Base Case: **8, 9**

2. Dead-end Case: **5, 6, 7**

3. Recursive Case: **1, 2, 3, 4**

Part B: Boolean Expressions

Given your answers above, synthesize Java boolean expressions for both the base and dead-end cases of the backtracking algorithm pictured above. Importantly, these expressions should evaluate to **true** if we're currently in a base / dead-end case respectively.

1. Base Case: **soFar.size() == 3**
***other expressions that apply only to 8, 9 accepted**

2. Dead-end Case: **sum > 2 / sum == 3 || sum == 4**
***other expressions that apply only to 5, 6, 7 accepted**

Part C: Exhaustive Search vs. Recursive Backtracking

In 1-3 sentences, explain why in recursive backtracking we need to explicitly unchoose the last chosen option (choose, explore, unchoose) in comparison to exhaustive search.

Hint: Remember that recursive backtracking uses data structures in some capacity (e.g. List<String>, Stack<String>, Queue<String>, etc.) whereas exhaustive search does not.

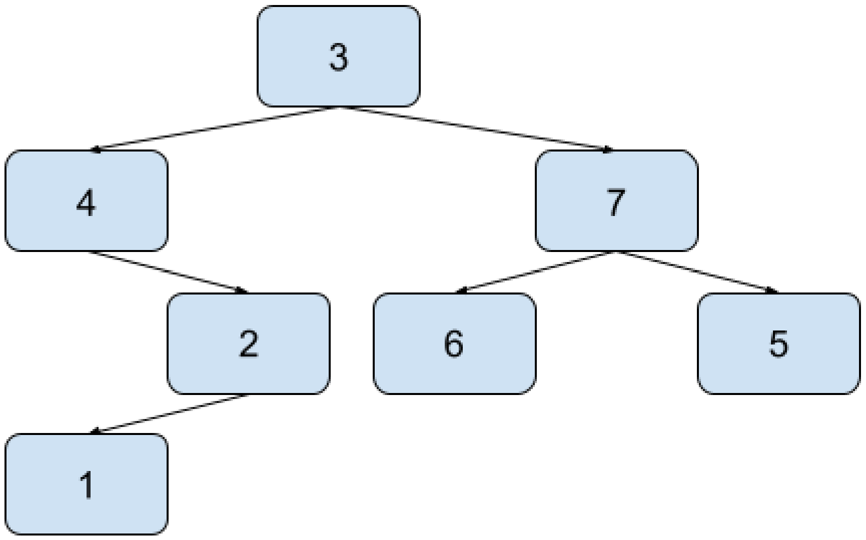
Reference semantics - all recursive calls share a reference to a single data structure, meaning changes are global. Unchoosing is necessary to return to the original state.

Exhaustive search uses non-objects, meaning every recursive call has its own copy of values and changes are local not global.

2. Binary (Search) Trees

Part A: Binary Tree Traversals

Consider the following binary tree:



Provide the values within the tree in the order called for each of the possible traversal patterns:

Pre-order:	[3, 4, 2, 1, 7, 6, 5]
In-order:	[4, 1, 2, 3, 6, 7, 5]
Post-order:	[1, 2, 4, 6, 5, 7, 3]

Part B: Binary Tree Comprehension

Consider the following code snippet implemented within the `IntTree` class:

```
public boolean mystery() {
    return mystery(this.overallRoot);
}

public boolean mystery(IntTreeNode root) {
    if (root == null || root.left == null && root.right == null) {
        return true;
    } else if (root.left == null || root.right == null) {
        return false;
    } else {
        return mystery(root.left) && mystery(root.right);
    }
}
```

Continued on next page -->

In 1-2 sentences, provide a plain english explanation as to what the `mystery` method accomplishes. Remember, that your explanation should be at a high-level, avoiding implementation details.

Determines whether or not each node in a tree has either two children or no children (no single branches).

Accepted terms such as: full, complete, balanced, etc.

Part C: Binary Search Trees

In 1 sentence, provide the tree traversal which when used printing out a `BinarySearchTree` (BST) will show all values in their natural (sorted) order.

- ☐ Pre-order Traversal
- ☑ **In-order Traversal**
- ☐ Post-order Traversal

Draw a picture below of the binary search tree that would result from inserting the following words into an empty binary search tree in the following order: **`Fettuccine, Ravioli, Spaghetti, Conchiglie, Macaroni, Farfalle`**

Assume the search tree uses alphabetical ordering to compare words.

```
              Fettuccine
             /          \
     Conchiglie          Ravioli
            \           /      \
          Farfalle  Macaroni   Spaghetti
```

3. Redemption Programming

Part A: Comparable

Implement the `compareTo` method for the following `Celebrity` class as described in the method comment below.

```
public class Celebrity implements Comparable<Celebrity> {
    private String name;
    private double netWorth;
    // Constructor omitted for brevity

    // Compares this celebrity to the provided other, ordering by netWorth (largest to
    // smallest) then name (alphabetical).
    public int compareTo(Celebrity other) {
        if (this.netWorth > other.netWorth) {
            return -1;
        } else if (this.netWorth < other.netWorth) {
            return 1;
        } else {
            return this.name.compareTo(other.name);
        }
    }
}
```

Now consider the `Youtuber` class, which is a subclass of `Celebrity`. fill in the class declaration blank appropriately and implement its `compareTo` method as described in the method comment below.

```
public class Youtuber extends Celebrity, implements Comparable<Youtuber> {
    private boolean canceled;
    // Constructor omitted for brevity

    // Compares this celebrity to the provided other, ordering by canceled status (not
    // canceled to canceled) netWorth (largest to smallest) then name (alphabetical).
    public int compareTo(Youtuber other) {
        if (!canceled && other.canceled) {
            return -1;
        } else if (canceled && !other.canceled) {
            return 1;
        } else {
            return super.compareTo(other);
        }
    }
}
```

Part B: ArrayIntList -> LinkedIntList

Implement the following constructor within the `LinkedIntList` class abiding by the provided comment. You may ***not*** assume that any other instance methods implemented in section/lecture are implemented here, and your solution should ***not*** include a size field / back pointer. The resulting implementation should run in at worst `O(n^2)` complexity.

Your solution can be either iterative or recursive - whichever is most helpful in solving the problem.

```
// Reverse iterative O(N)
public LinkedIntList(int[] elementData, int size) {
    for (int i = size - 1; i >= 0; i-) {
        front = new ListNode(elementData[i], front);
    }
}

// Normal iterative O(N)
public LinkedIntList(int[] elementData, int size) {
    if (size != 0) {
        front = new ListNode(elementData[0]);
        ListNode curr = front;
        for (int i = 1; i < size; i++) {
            curr.next = new ListNode(elementData[i]);
            curr = curr.next;
        }
    }
}

// Slow iterative O(N^2)
public LinkedIntList(int[] elementData, int size) {
    for (int i = 0; i < size; i++) {
        if (front == null) {
            front = new ListNode(elementData[i]);
        } else {
            ListNode curr = front;
            while (curr.next != null) {
                curr = curr.next;
            }
            curr.next = new ListNode(elementData[i]);
        }
    }
}
```

```
// Recursive O(N)
public LinkedIntList(int[] elementData, int size) {
    front = build(elementData, size, 0);
}

public ListNode build(int[] elementData, int size, int i) {
    if (x >= size) {
        return null;
    }
    return new ListNode(elementData[i],
                        build(elementData, size, i+1));
}

// Slow recursive O(N^2)
public LinkedIntList(int[] elementData, int size) {
    for (int i = 0; i < size; i++) {
        build(front, elementData[i]);
    }
}

public ListNode build(ListNode front, int val) {
    if (front == null) {
        return new ListNode(val);
    }
    front.next = build(front.next, val);
    return front;
}
```