CS 61BL Summer 2025

Midterm Friday, July 18th, 2025

Solutions last updated: 7/21/25

PRINT Your Name: _____

PRINT Your Student ID: _____

PRINT Student name to your left: _____

PRINT Student name to your right: _____

You have 110 minutes. There are 8 questions of varying credit. (100 points total)

Question:	1	2	3	4	5	6	7	8	Total
Points:	12	8	9	19	18	14	20	0	100

For questions with **circular bubbles**, select only one choice (there is only one correct answer).

- ○ Unselected option (Completely unfilled)
- ◌ Don't do this (it will be graded as incorrect)
- ● Only one selected option (completely filled)

For questions with **square boxes**, you may select one or more choices (select all that apply).

- ■ You can select
- ■ multiple squares
- ☑ Don't do this (it will be graded as incorrect)

- Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers or your answer is ambiguous, we will grade the **worst** interpretation.
- Unless otherwise specified, all data structures and algorithms behave according to their implementation in lecture or lab, with no additional optimizations, and all relevant libraries have been imported.
- If an implementation detail (e.g. tiebreaking scheme, linked list topology) is relevant, it will be explicitly noted in the question.
- You may write at most one statement per blank and you may not use more blanks than provided.
- Your answer will be reformatted according to the 61B/L style guidelines. For example, any method, constructor, or if-statement requires at least three lines for the purposes of determining line count.
- You may not use ternary operators, lambdas, streams, or multiple assignment.

Read the honor code below and sign your name.

> By signing below, I affirm that all work on this exam is my own work. I have not referenced any disallowed materials, nor collaborated with anyone else on this exam. I understand that if I cheat on the exam, I may face the penalty of an "F" grade and a referral to the Center for Student Conduct.

SIGN your name: _____

Q1 *Amy's Ponies* **(12 points)**

Consider the code below.

```
public interface Magical {
    default void useMagic() {
        System.out.println("ZAP!");
    }
}

public class Horse {
    public static String species = "equus";
    private int weight;

    public Horse() {
        this.weight = 900;
        System.out.println("Horse");
    }

    public Horse(int weight) {
        this.weight = weight;
        System.out.println("Weight: " + weight);
    }

    public void neigh() {
        System.out.println("Neigh!");
    }
}

public class Unicorn extends Horse implements Magical {
    private int magicLevel;

    public Unicorn(int magicLevel, int weight) {
        super(weight);
        this.magicLevel = magicLevel;
    }

    @Override
    public void neigh() {
        System.out.println("Friendship!");
    }

    @Override
    public void useMagic() {
        System.out.println("MAGIC!");
    }
}
```

Q1.1 (12 points)

Write what each line would output in the box to the right of that line.

If the line would result in a compiler error, leave the box blank and bubble "CE". If the line would result in a runtime error, leave the box blank and bubble "RE". In either error case, continue as if that line was never run.

If no output is printed, and the line would not error, write the word "nothing" (without the quotes).

The first few lines have been given for you.

`System.out.println("Welcome!");`	○ CE ○ RE	Welcome!
`String to = "the";`	○ CE ○ RE	nothing
`int exam = "!";`	● CE ○ RE	
`Horse appleJack = new Horse();`	○ CE ○ RE	
`Magical pinkiePie = new Horse();`	○ CE ○ RE	
`Unicorn twilight = new Unicorn(10, 50);`	○ CE ○ RE	
`Magical sparkle = twilight;`	○ CE ○ RE	
`Unicorn princessCelestia = new Horse();`	○ CE ○ RE	
`Unicorn rarity = (Unicorn) appleJack;`	○ CE ○ RE	
`twilight.neigh();`	○ CE ○ RE	
`twilight.useMagic();`	○ CE ○ RE	

Solution:

`System.out.println("Welcome!");`	○ CE ○ RE	Welcome!
`String to = "the";`	○ CE ○ RE	nothing
`int exam = "!";`	● CE ○ RE	
`Horse appleJack = new Horse();`	○ CE ○ RE	Horse
`Magical pinkiePie = new Horse();`	● CE ○ RE	
`Unicorn twilight = new Unicorn(10, 50);`	○ CE ○ RE	Weight: 50
`Magical sparkle = twilight;`	○ CE ○ RE	nothing
`Unicorn princessCelestia = new Horse();`	● CE ○ RE	
`Unicorn rarity = (Unicorn) appleJack;`	○ CE ● RE	
`twilight.neigh();`	○ CE ○ RE	Friendship!
`twilight.useMagic();`	○ CE ○ RE	MAGIC!

Q2 *Addicted Drinkers of Tea (ADTs)* **(8 points)**

Noah, Wilson, and Karen all want to buy boba for every `Student` in the world. For each of the below subquestions, choose the *best* abstract data type (ADT) to use. Each subpart is independent from the other subparts.

Your options for ADTs are `List`, `Set`, `Map`, `Queue`, or `Stack`.

Q2.1 (2 points) Noah would like to know how far different buildings (represented by the `Building` class) are from CS61BobaShop. For example, Evans Hall is 61.8 kilometers from CS61BobaShop.

Write your chosen ADT, including the generic type(s):

Example answers: `List<Building>` or `Map<Integer, String>`

`Map<Building, Double> or Map<Building, Float> or Map<Building, String>`

Q2.2 (2 points) Students (represented by the `Student` class) who order last receive their boba first (slightly unfair, yes). Wilson wants to keep track of the next student to receive their boba. Students are tracked immediately after ordering.

Write your chosen ADT, including the generic type(s):

`Stack<Student>`

Q2.3 (2 points) Karen asks students (represented by the `Student` class) to go into a tree formation, where every student represents a node. When distributing boba, Karen gives them to students in increasing order of distance from the root node (breaking ties arbitrarily). As she distributes boba, Karen wants to know which student is the next to receive boba.

Write your chosen ADT, including the generic type(s):

`Queue<Student>`

Q2.4 (2 points) Every time a student (represented by the `Student` class) mentions a boba type (represented by the `Boba` class), Michelle wants to know whether or not that boba type was mentioned before by *any* student.

Write your chosen ADT, including the generic type(s):

`Set<Boba>`

Solution:

2.1 Since we want to be able to represent decimals, we use a map from `Building`s to `Double`s or `Float`s. Note that it was not specified whether something like `"61.8 kilometers"` is valid, so we also accept `Map<Building, String>` as a valid answer.

2.2 A `Stack` follows LIFO (last-in-first-out) order, which is exactly what we want here.

2.3 A `Queue` follows FIFO (first-in-first-out) order, which is required for a level order traversal. The students who arrive earlier will be closer to the root of the tree, and will get their boba first.

2.4 `Set`s hold unique values with no copies and are perfect to use as a check off for previous mentions. Note that we don't actually care about which `Student` mentioned the particular `Boba`, only whether the `Boba` was mentioned at all.

Q3 *Would You Like to Get Foo'd* **(9 points)**

Q3.1 (9 points) Consider the following code:

```
class Foo {
    private int x;
    public Foo(int x) {
        this.x = x;
    }
}
```

Codey is considering implementations for `boolean equals(Object obj) { ... }` in `Foo`.

We instantiate three `Foo` objects `foo1`, `foo2`, `foo3`, each with an arbitrary (not necessarily unique) value of `x`.

Then, we create a new `Set` according to some `Set` implementation, and insert `foo1`, `foo2`, and `foo3`, in that order, into this newly created `Set`.

Select whether each `equals` implementation always, sometimes, or never results in the respective final resulting `Set` after inserting all of `foo1`, `foo2`, `foo3`, in that order.

Note: Assume nonempty `Set`s call `equals` when checking for duplicates.

	Final result after inserting all of `foo1`, `foo2`, `foo3` into an empty `Set`		
`equals` implementation	`{foo1}`	`{foo1, foo2}`	`{foo1, foo2, foo3}`
`return this == obj;`	○ Always ○ Sometimes ● Never	○ Always ○ Sometimes ● Never	● Always ○ Sometimes ○ Never

`Foo f = (Foo) obj;`
`return ((this.x % 2)`
`== (f.x % 2));`
	○ Always
	● Sometimes
	○ Never
		○ Always
		● Sometimes
		○ Never
			○ Always
			● Sometimes
			● Never

`Foo f = (Foo) obj;`
`this.x = f.x;`
`return this.x == f.x;`
	● Always
	○ Sometimes
	○ Never
		○ Always
		○ Sometimes
		● Never
			○ Always
			○ Sometimes
			● Never

Solution:

- For the first `equals` implementation, we are directly checking for matching memory addresses. Since `foo1`, `foo2`, and `foo3` are all separate objects (each made with a separate `new` keyword), none of their memory addresses will be the same, and thus they will all be added to the `Set`.
- For the second `equals` implementation, the original intention was to check whether or not the `x` instance variable is divisible by 2 (even versus odd). If `foo1` and `foo2` are both even, for example, only `foo1` would get added to the `Set`. If `foo1` is even and `foo2` is odd (or the other way around), they would both get added to the `Set`. However, `foo3` can never be added to the set if both `foo1` and `foo2` are already added.

  However, Java's `%` operator returns a negative value for negative numbers, so there are three choices for `x` in `foo1`, `foo2`, `foo3`: even, odd, and negative odd. Since this is a bit of an obscure Java quirk (staff forgot about this!), both Never and Sometimes are accepted for the third box of this `equals` implementation.
- For the third `equals` implementation, we set the value of `foo1.x` to be whatever the value of `foo1.x` was, and similarly for `foo3`. So the `Set` will only contain `foo1`, as both `foo2` and `foo3` will have the same value of `x` as the `foo1` in the `Set`.

This page intentionally left (mostly) blank

The exam continues on the next page.

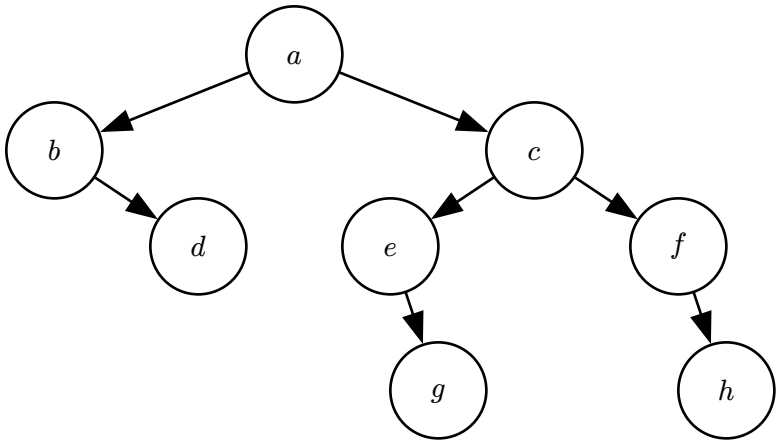
Q4 *Field of Hopes and Dreams* **(19 points)**

Lancer has built a devious forest maze in the shape of a Binary Search Tree, with an `int` label at each node. As with standard BSTs, there are no duplicate labels.

Q4.1 (6 points)

Below is one of the BST mazes that Lancer could have built, with labels hidden by unknown variables a through h (not in alphabetical order).

For the following logical comparisons, identify whether they are always, sometimes, or never `true` depending on the values of the variables a through h.



$a > c$

○ Always ○ Sometimes ● Never

$b == d$

○ Always ○ Sometimes ● Never

$d < e$

● Always ○ Sometimes ○ Never

$(b + d) > a$

○ Always ● Sometimes ○ Never

$(c - h) < 0$

● Always ○ Sometimes ○ Never

$(b + d) == 0$

○ Always ● Sometimes ○ Never

Solution: $a > c$: Following the BST property, all values to the right of a must be greater than a. Thus, c will never be less than a.

$b == d$: BSTs do not have duplicates.

$d < e$: The BST property forces all values left of a to be less than a and all values right of a to be greater than a. Since $d < a$ and $e > a$, then transitively, d must always be less than e.

$(b + d) > a$: If b was 2 and d was 3 and a was 4, the BST property would hold and $(2 + 3) > 4$.

$(c - h) < 0$: The BST property forces h to always be greater than c. Thus, $(c - h)$ will always be negative.

$(b + d == 0)$: If b was −3 and d was 3, the BST property would hold and $(b + d == 0)$.

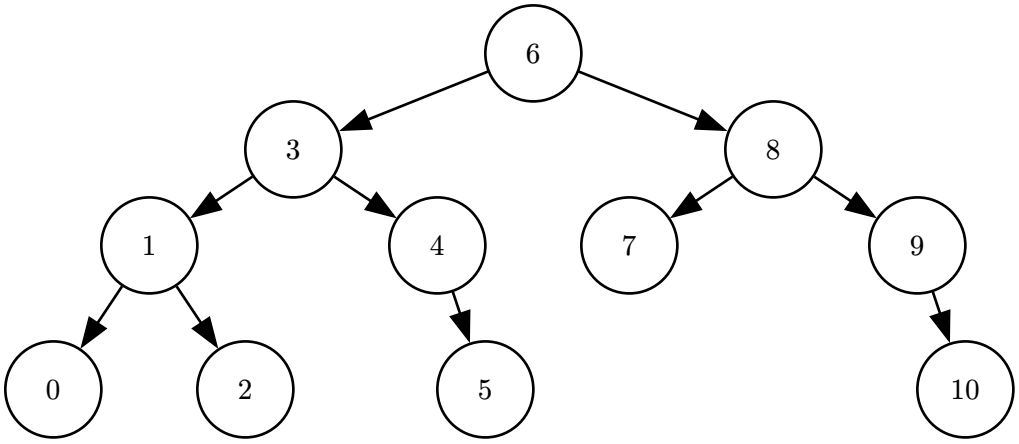
Q4.2 (13 points) Lancer builds a `ForestPathNode` class to represent his BST maze.

```
public class ForestPathNode {
    public int label;
    public ForestPathNode left;
    public ForestPathNode right;

    // ... methods not shown ...
}
```

Suppose that the heroes know the correct `label` of their desired destination `dest`. Build a `ForestPathIterator` that returns the correct path from the input node to the node with the given label, using the directions `"left"` and `"right"`.

Below is an example BST forest maze:



- A `ForestPathIterator` that takes in the root node of the above forest maze and a `dest` of `5` should exhaust its elements after `next()` returns `"left"`, then `"right"`, then `"right"`.
- A `ForestPathIterator` that takes in the root node and a `dest` of `8` should exhaust its elements after `next()` returns `"right"`.

Assume that the given `dest` is in the maze.

Your solution should construct in $\Theta(1)$ time.

The skeleton code begins on the next page.

(Question 4 continued...)

```
public class ForestPathIterator implements Iterator<String> {

    private ForestPathNode curr;

    private  int  dest;

    public ForestPathIterator(ForestPathNode root, int dest) {

        this.curr = root;

        this.dest = dest;
    }


    @Override

    public boolean hasNext() {

        return this.curr.label != dest;
    }



    @Override

    public String next() {

        if (this.dest > curr.label) {

          curr = curr.right;

          return "right";
        } else {

            curr = curr.left;

            return "left";
        }
    }
}
```

Solution: We know that our `Iterator` spits out `String`s; we also know that our pointer points to `Node`s and our given `dest` is an integer. We set up these instance variables in our constructor. `hasNext()` should return `true` until we have arrived at our `dest`, which is guaranteed since it is assumed `dest` is in the tree. Finally, `next()` is in charge of figuring out whether we go left or right by making a comparison to the current node and returns the direction `"left"` or `"right"` accordingly.

Q5 *Recycling* **(18 points)**

Consider the following raw, singly-linked `SLNode` class:

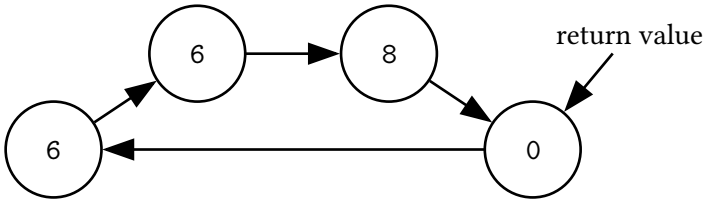
```
public class SLNode<T> {
    public T item;
    public SLNode<T> next;
    public SLNode(T item) {
        this.item = item;
        this.next = null;
    }
}
```

Q5.1 (8 points)

Complete `cyclify`, which takes in a `int[] arr` and returns a new, *circular*, `SLNode<Integer>` where each node corresponds to an element in the `int[]`. **Return the last node** in the resulting circular linked list.

Assume that `arr` is nonempty.

For example, `cyclify(new int[]{6, 6, 8, 0})` should build the circular linked list below, and return the `SLNode` corresponding to `0`.




```
public SLNode<Integer> cyclify(int[] arr) {

    SLNode<Integer> curr = new SLNode<>(arr[0]);

    SLNode<Integer> head = curr;

    for (int i =  1  ; i < arr.length; i += 1) {

        curr.next = new SLNode<>(arr[i]);

        curr = curr.next;
    }

    curr.next = head;

    return curr;
}
```

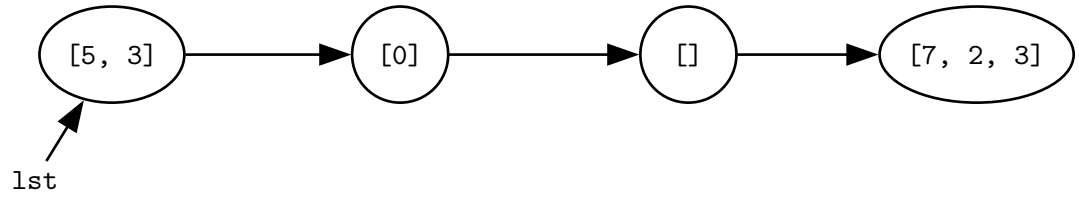
Solution: Since the array is assumed to be nonempty, we can create a node with the value at `arr[0]`, which will be the starting point of our returned list. Then, we iterate through the array, appending a node for each item in the list, until we get to the end of the array. We set the `next` pointer of the last node to the head of our list, and return the last node.

Q5.2 (10 points)

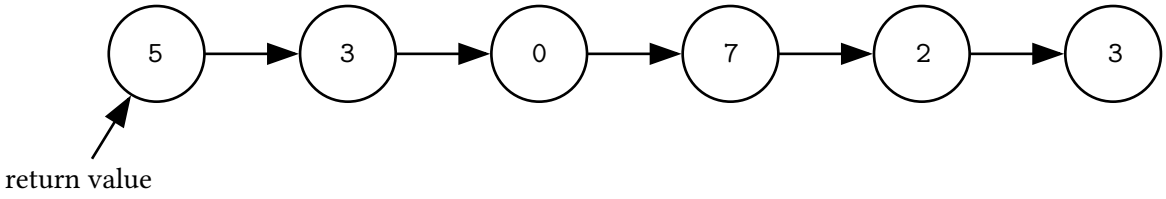
Complete **`flatten`**, which takes in a (non-circular) **`SLNode<int[]> lst`** and flattens it into a (non-circular) **`SLNode<Integer>`** as shown in the following diagram. You may use **`cyclify`** as defined in part (a).

An example is given in the following diagram:

`lst` begins at the **`SLNode`** containing **`[5, 3]`**.



after **`flatten`**ing:




```
public SLNode<Integer> flatten(SLNode<int[]> lst) {
    if (lst == null) {
        return null;
    }

    SLNode<Integer> rest = flatten(lst.next);

    if (lst.item.length == 0) {
        return rest;
    }

    SLNode<Integer> lastOfCurr = cyclify(lst.item);

    SLNode<Integer> result = lastOfCurr.next;

    lastOfCurr.next = rest;
    return result;
}
```

Solution: We make heavy use of the recursive leap of faith here.

The base case is when you have an empty `lst` (i.e., `lst == null`). In that case, the list is already flattened, so we just return null.

Otherwise, we flatten the rest of the list via recursion.

For the current node, we cyclify it to get something that is almost flat, and get the head of the list by using the circular structure. Then we can reassign the next pointer of the last node in the cyclified node to point to the rest of the list, and finally return the head of the list (which we saved in a temp variable).

Q6 *Critical Thinking* **(14 points)**

Noah is brainstorming `Idea`s for the midterm, and he wants to know which are bad `Idea`s.

Fortunately, Noah knows there's always a `Critic` who can `review` any `Idea`:

```
public interface Idea {
    // methods not shown
}
```

```
public interface Critic {
    /* Returns how good a given Idea is. */
    int review(Idea idea);
}
```

Q6.1 (4 points) Complete `CriticComparator`, which takes in some `Idea`. It compares two `Critic`s according to their `review` values of that `Idea` (in natural order).

```
public class CriticComparator implements Comparator<Critic> {
    // Add instance variables (if any needed) here

    private Idea idea;

    public CriticComparator(Idea idea) {

        this.idea = idea;

    }

    @Override
    public int compare(Critic c1, Critic c2) {

        return c1.review(idea) - c2.review(idea);

    }
}
```

Solution: We need to keep track of the `Idea` passed into the constructor of `CriticComparator` (note that `public Idea idea` or `Idea idea` also works). Then, when `compare` is called, we `review` this `Idea` according to each `Critic` and return a positive, negative or `0` integer accordingly.

Q6.2 (10 points)

Complete **IdeaComparator**, which considers an array of **Critic**s called **consideredCritics**.

It compares two **Idea**s (in natural order) according to their highest respective **Critic** review across all **Critic**s being considered.

You may assume there is at least one **Critic** being considered when **compare** is called.

You may use **CriticComparator** as defined in the previous subpart in addition to any classes or methods on the reference sheet.

```
public class IdeaComparator implements Comparator<Idea> {
    private Critic[] consideredCritics;
    public IdeaComparator() { ... }

    @Override
    public int compare(Idea idea1, Idea idea2) {

        CriticComparator c1 = new CriticComparator(idea1);

        CriticComparator c2 = new CriticComparator(idea2);

        Critic bestCritic1 = Collections.max(consideredCritics, c1);

        Critic bestCritic2 = Collections.max(consideredCritics, c2);

        return bestCritic1.review(idea1) - bestCritic2.review(idea2);
    }
}
```

Solution: First, we need to find the `Critic` in `critics` that gives the highest review for each `Idea` passed in to `compare`. To do this, we create two new `CriticComparator`s and use `Collections.max` (example on the reference sheet) to find the highest reviewing critic of each idea. Finally, we return which `Idea`s best `Critic` review was higher.

Alternate solution with sort:

Note that `sort` is destructive, so creating a new `Critic[]` is invalid, and `Collections.sort` does not work on arrays. Arrays also do not have `sort` as an instance method.

```
public class IdeaComparator implements Comparator<Idea> {
    private Critic[] consideredCritics;
    public IdeaComparator() { ... }

    @Override
    public int compare(Idea idea1, Idea idea2) {

        Arrays.sort(consideredCritics, new CriticComparator(idea1));

        Critic c1 = consideredCritics[consideredCritics.length-1];

        Arrays.sort(consideredCritics, new CriticComparator(idea2));

        Critic c2 = consideredCritics[consideredCritics.length-1];

        return c1.review(idea1) - c2.review(idea2);

    }
}
```

Q7 *Un-Noah-ble* **(20 points)**

Q7.1 (4 points) What is the runtime of `naa(N)` in terms of N?

```
public void naa(int N) {
    int b = 61;
    if (N <= 1) {
        return;
    }
    if (Math.random() <= 0.0001) {
        b = 8;
        naa(N - 1);
    }
    for (int i = 1; i < b; i += 7) {
        System.out.println("yum");
    }
}
```

Best case:

◯ $\Theta(1)$	◯ $\Theta(\log(\log N))$	◯ $\Theta(\log N)$	◯ $\Theta((\log N)^2)$	◯ $\Theta\left(\sqrt{N}\right)$
◯ $\Theta(N)$	◯ $\Theta(N \log N)$	◯ $\Theta(N^2)$	◯ $\Theta(N^2 \log N)$	◯ $\Theta(N^3)$
◯ $\Theta(N^3 \log N)$	◯ $\Theta(2^N)$	◯ $\Theta(N!)$	◯ $\Theta(N^N)$	◯ Infinite loop

Solution: $\Theta(1)$. Note that the `for` loop runs in constant time.

In the best case, the `Math.random` if-case is never entered, so the runtime is $\Theta(1)$, as there is no recursion.

Worst case:

◯ $\Theta(1)$	◯ $\Theta(\log(\log N))$	◯ $\Theta(\log N)$	◯ $\Theta((\log N)^2)$	◯ $\Theta\left(\sqrt{N}\right)$
◯ $\Theta(N)$	◯ $\Theta(N \log N)$	◯ $\Theta(N^2)$	◯ $\Theta(N^2 \log N)$	◯ $\Theta(N^3)$
◯ $\Theta(N^3 \log N)$	◯ $\Theta(2^N)$	◯ $\Theta(N!)$	◯ $\Theta(N^N)$	◯ Infinite loop

Solution: $\Theta(N)$. In the worst case, we enter the `Math.random` case at every recursive call. We have constant work per node and the `N-1` recursive calls will lead to a linked list recursive tree with height N. This is $1 + 1 + ... + 1$ runtime with N terms, which is $\Theta(N)$.

Q7.2 (4 points) What is the runtime of `melo(N)` in terms of N?

```
public void melo(int N) {
    if (N <= 1) {
        return;
    }
    melo(N / 2);
    if (N % 2 == 0) {
        melo(N / 2);
    }
}
```

Best case:

◯ $\Theta(1)$	◯ $\Theta(\log(\log N))$	◯ $\Theta(\log N)$	◯ $\Theta((\log N)^2)$	◯ $\Theta\left(\sqrt{N}\right)$
◯ $\Theta(N)$	◯ $\Theta(N \log N)$	◯ $\Theta(N^2)$	◯ $\Theta(N^2 \log N)$	◯ $\Theta(N^3)$
◯ $\Theta(N^3 \log N)$	◯ $\Theta(2^N)$	◯ $\Theta(N!)$	◯ $\Theta(N^N)$	◯ Infinite loop

Solution: $\Theta(\log N)$. In the best case, the if case is never entered. This would result in a linked list recursive tree with height $\log N$, with constant work per node, for a total runtime of $\Theta(\log N)$.

Note that a value of N always exists for which the if case is never entered, namely values of $N = 2^k - 1$ for some k (one less than a power of two).

Worst case:

◯ $\Theta(1)$	◯ $\Theta(\log(\log N))$	◯ $\Theta(\log N)$	◯ $\Theta((\log N)^2)$	◯ $\Theta\left(\sqrt{N}\right)$
◯ $\Theta(N)$	◯ $\Theta(N \log N)$	◯ $\Theta(N^2)$	◯ $\Theta(N^2 \log N)$	◯ $\Theta(N^3)$
◯ $\Theta(N^3 \log N)$	◯ $\Theta(2^N)$	◯ $\Theta(N!)$	◯ $\Theta(N^N)$	◯ Infinite loop

Solution: $\Theta(N)$. In the worst case, the if case is always entered. This results in a recursive tree with height $\log N$ that splits in half at each level, with each node doing constant work. Summing each level gives $1 + 2 + 4 + 8 + ... + \frac{N}{2}$ which is $\Theta(N)$.

Note that a value of N always exists for which the if case is always entered, namely values of $N = 2^k$ for some k (powers of two).

Q7.3 (6 points) What is the runtime of `baco(N)` in terms of N?

```
public void baco(int N) {
    int[] spaghett = new int[N];
    spaghett[0] = Math.round(Math.random());

    for (int i = 0; i < N - 1; i += 1) {
        spaghett[i + 1] = 0;
        for (int x = 0; x < spaghett[i]; x += 1) {
            spaghett[i + 1] += 2;
        }
    }
}
```

Best case:

◯ $\Theta(1)$	◯ $\Theta(\log(\log N))$	◯ $\Theta(\log N)$	◯ $\Theta((\log N)^2)$	◯ $\Theta\left(\sqrt{N}\right)$
◯ $\Theta(N)$	◯ $\Theta(N \log N)$	◯ $\Theta(N^2)$	◯ $\Theta(N^2 \log N)$	◯ $\Theta(N^3)$
◯ $\Theta(N^3 \log N)$	◯ $\Theta(2^N)$	◯ $\Theta(N!)$	◯ $\Theta(N^N)$	◯ Infinite loop

Solution: $\Theta(N)$. The inner loop with x never runs if `spaghett[0]` is 0, so in the best case, only the i outer loop runs N times with constant work per iteration, for a runtime of $\Theta(N)$.

Worst case:

◯ $\Theta(1)$	◯ $\Theta(\log(\log N))$	◯ $\Theta(\log N)$	◯ $\Theta((\log N)^2)$	◯ $\Theta\left(\sqrt{N}\right)$
◯ $\Theta(N)$	◯ $\Theta(N \log N)$	◯ $\Theta(N^2)$	◯ $\Theta(N^2 \log N)$	◯ $\Theta(N^3)$
◯ $\Theta(N^3 \log N)$	◯ $\Theta(2^N)$	◯ $\Theta(N!)$	◯ $\Theta(N^N)$	◯ Infinite loop

Solution: $\Theta(2^N)$. In the worst case, the inner loop will be entered at each iteration. Then the loop corresponding to `spaghett[0]` runs once, which will update `spaghett[1]` to be `2`. The loop corresponding to `spaghett[1]` will run N times, causing `spaghett[2]` to be $2 + 2 = 4$. The loop corresponding to `spaghett[2]` runs 4 times, then the one corresponding to `spaghett[3]` will run 8 times, and so on, until we reach `spaghett[N - 2]`, which will run 2^{N-2} times. The runtime is then $1 + 2 + 4 + 8 + ... + 2^{N-2} \in \Theta(2^N)$.

Q7.4 (6 points) What is the runtime of `rame(N)` in terms of N?

```
public void rame(int N) {
    for (int i = 1; i < N; i *= 2) {
        sush(i);
    }
}

private void sush(int i) {
    if (i >= 1) {
        sush(i / 2);
    }
}
```

Best case:

◯ $\Theta(1)$	◯ $\Theta(\log(\log N))$	◯ $\Theta(\log N)$	◯ $\Theta((\log N)^2)$	◯ $\Theta\left(\sqrt{N}\right)$
◯ $\Theta(N)$	◯ $\Theta(N \log N)$	◯ $\Theta(N^2)$	◯ $\Theta(N^2 \log N)$	◯ $\Theta(N^3)$
◯ $\Theta(N^3 \log N)$	◯ $\Theta(2^N)$	◯ $\Theta(N!)$	◯ $\Theta(N^N)$	◯ Infinite loop

Worst case:

◯ $\Theta(1)$	◯ $\Theta(\log(\log N))$	◯ $\Theta(\log N)$	◯ $\Theta((\log N)^2)$	◯ $\Theta\left(\sqrt{N}\right)$
◯ $\Theta(N)$	◯ $\Theta(N \log N)$	◯ $\Theta(N^2)$	◯ $\Theta(N^2 \log N)$	◯ $\Theta(N^3)$
◯ $\Theta(N^3 \log N)$	◯ $\Theta(2^N)$	◯ $\Theta(N!)$	◯ $\Theta(N^N)$	◯ Infinite loop

Solution: Best and worst case is $\Theta((\log N)^2)$. `rame` will create a bunch of nodes in the recursive tree with values $i = 1, 2, 4, 8, ..., N$. Then, `sush(i)` will result in several branches according to each of those nodes. The first will have height 1, the second has height 2, the third has height 3, then 4, and so on, until the last branch has height $\log N$. So in total the runtime is $1 + 2 + 3 + 4 + ... + \log N \in \Theta((\log N)^2)$, and the best and worst cases are identical.

Q8 *Am I Worth Nothing to You?* **(0 points)**

Congrats on finishing the exam! The below questions are just for fun; **nothing on this page is worth any points.**

Q8.1 (0 points) Which people, if any, are telling the truth?

Karen: "If Noah is telling the truth, then I am lying."

Noah: "Karen and Wilson are both liars!"

Wilson: "If Noah is lying, then all of us are lying."

- ■ Karen
- ☐ Noah
- ☐ Wilson
- ○ None of the above

Solution: Suppose Noah is truthful. Then Karen and Wilson are both lying. Since Noah is truthful, Karen's statement reduces to "I am lying", a true statement! This contradicts Noah's testimony (that Karen is lying), so Noah must be lying.

This means either Karen or Wilson (or both) must be truthful. This also reduces Wilson's statement to "all of us are lying".

Suppose Wilson is truthful. Then everyone is lying, which means Wilson must be lying, contradicting the assumption that Wilson is truthful. So Wilson must also be lying.

If Karen was lying, then Wilson's statement (everyone is lying) would be true, but Wilson is a liar, so Wilson's statement can't be true!

So Karen must be telling the truth.

Q8.2 (0 points; one cookie from Dawn) Who is NOT on CS61BL staff this semester?

☐ Samuel	■ Eric	■ Erik	■ Kanav	■ Kevin
■ Stacey	☐ Miller	■ Alonzo	☐ Lawrence	☐ Dennis
☐ Andrew	☐ Benjamin	■ Gabe	■ Noelle	■ Anniyat
■ David	■ LeBron	■ Susie	☐ Stanley	☐ Amanda
■ Teresa	■ Curtis	☐ Yinqi	☐ Yashna	■ Apollo
■ Rico	☐ Julian	■ Jonah	■ Sophia	■ Circle

Q8.3 (0 points) Leave any feedback, comments, and/or drawings in the box below!