1. Code Comprehension

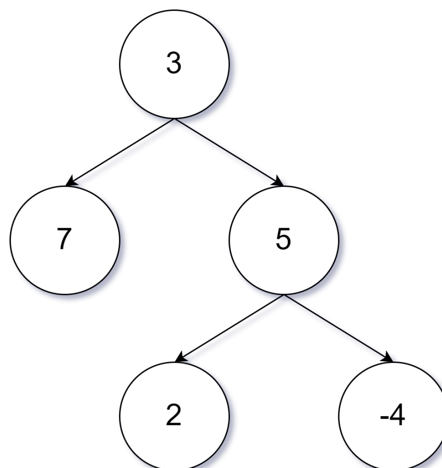
Part A: (Select all that apply) Which of these statements are true about inheritance?

- ☐ Inheritance allows a subclass to access all private properties and methods of its parent class.
- ☑ Calling a method using `super` will go to the parent class one level above the current subclass.
- ☑ To indicate that class `OrangeJuice` is a subclass of `Juice`, we write the following:
  `public class OrangeJuice extends Juice`
- ☐ Parent classes can call the methods of their subclasses.
- ☐ If the class `Mammal` is the parent class of `Dog`, then the following statement is legal:
  `Dog a = new Mammal();`

Part B: Consider the following method in the **`IntTree`** class:

```
1     public List<String> method() {
2          List<String> result = new ArrayList<>();
3          methodHelper(overallRoot, result, "");
4          return result;
5      }
6
7      private void methodHelper(TreeNode root, List<String> result, String s) {
8          if (root != null) {
9              s += root.data;
10             if (root.left == null && root.right == null) {
11                 result.add(s);
12             } else {
13                 s += ", ";
14                 methodHelper(root.left, result, s);
15                 methodHelper(root.right, result, s);
16             }
17         }
18     }
```

Provide a tree that, if called by the above method, would result in a list of size 3.



Part C: Consider the following scenario:

In your free time during Thanksgiving break, you and a friend decided to start working on an online platform in which users can display their academic and career accomplishments. After a quick discussion of how a user should be represented, you settled on the following fields:

```
1  private String name;
2  private List<User> connections;
3  private Set<String> skills;
```

On your new platform, you want to allow users to apply to open positions at companies (think engineer, artist, plumber, etc.). You also want to provide companies an ordered collection of applicants that are best suited for the role. Being a near 123 graduate, you recognized the potential of using the **`Comparable`** interface to define an ordering between applicants (where the first element after sorting would be considered the "best" applicant for a role). You tell this to your collaborator who suggests the following implementation for the compareTo method:

```
1  public int compareTo(User other) {
2      if (this.connections.size() == other.connections.size()) {
3          if (this.skills.size() == other.skills.size()) {
4              return this.name.compareTo(other.name);
5          }
6          return other.skills.size() - this.skills.size();
7      }
8      return other.connections.size() - this.connections.size();
9  }
```

Write an appropriate method comment for the above `compareTo` implementation.

This method compares two **`User`** objects, returning a positive number if 'this' comes after 'other', a negative number if 'this' comes before 'other', or 0 if there's a tie. Initially, it compares users by their total number of connections, placing users with more before those with fewer. If the number of connections is equal, the method then sorts by the number of skills, this time prioritizing users with more skills. Lastly, if both the number of connections and skills are identical for two users, the method sorts them lexicographically/alphabetically by their names.

Describe a potential concern a client of this platform might have with its design, and explain a high-level change you could make to address that concern. Your critique should be from the perspective of a _client_, not the implementer.

An answer with reasonable justification would receive full credit. Below are two sample answers:

1. A client might be concerned that applicants are being primarily prioritized by the number of connections they already have. Doing so might reduce the diversity of an applicant pool as those with fewer industry ins are pushed towards the end of the recommendation list. To fix this, we could remove the connection comparison altogether and order on other factors.
2. A client might be concerned that applicants are being prioritized by factors entirely external to the job they're applying for (someone with many skills related to computer science might not be a good fit for work as a plumber). To fix this, after sorting we could do another pass shifting applicants unsuited for a role towards the end.

2. Code Tracing

Part A: For each of the following, **`list`** represents the starting node. Write the code necessary to convert the following sequences of **`ListNode`** objects in the **Before** into the sequence of **`ListNode`** objects in the **After**:

Before:	**After:**
`list -> [5] -> [4] -> [3] /`	`list -> [4] -> [5] -> [3] /`

```
ListNode temp = list.next;
list.next = list.next.next;
temp.next = list;
list = temp;
```

Before:	**After:**
`list  -> [1] -> [2] /` `list2 -> [3] -> [4] /`	`list  -> [4] -> [1] /` `list2 -> [2] -> [3] /`

```
list.next.next = list2;
list2.next.next = list;
list = list2.next;
list2 = list.next.next;
list.next.next = null;
list2.next.next = null;
```

Part B: Consider the following method:

```
1  public int mystery(int n, int d) {
2        if (d < 0 || d > 9) {
3              throw new IllegalArgumentException();
4        }
5
6        if (n < 0) {
7              return -mystery(-n, d);
8        } else if (n == 0) {
9              return 0;
10       } else if (n % 10 == d) {
11             return mystery(n / 10, d) * 100 + d * 11;
12       } else {
13             return mystery(n / 10, d) * 10 + n % 10;
14       }
15 }
```

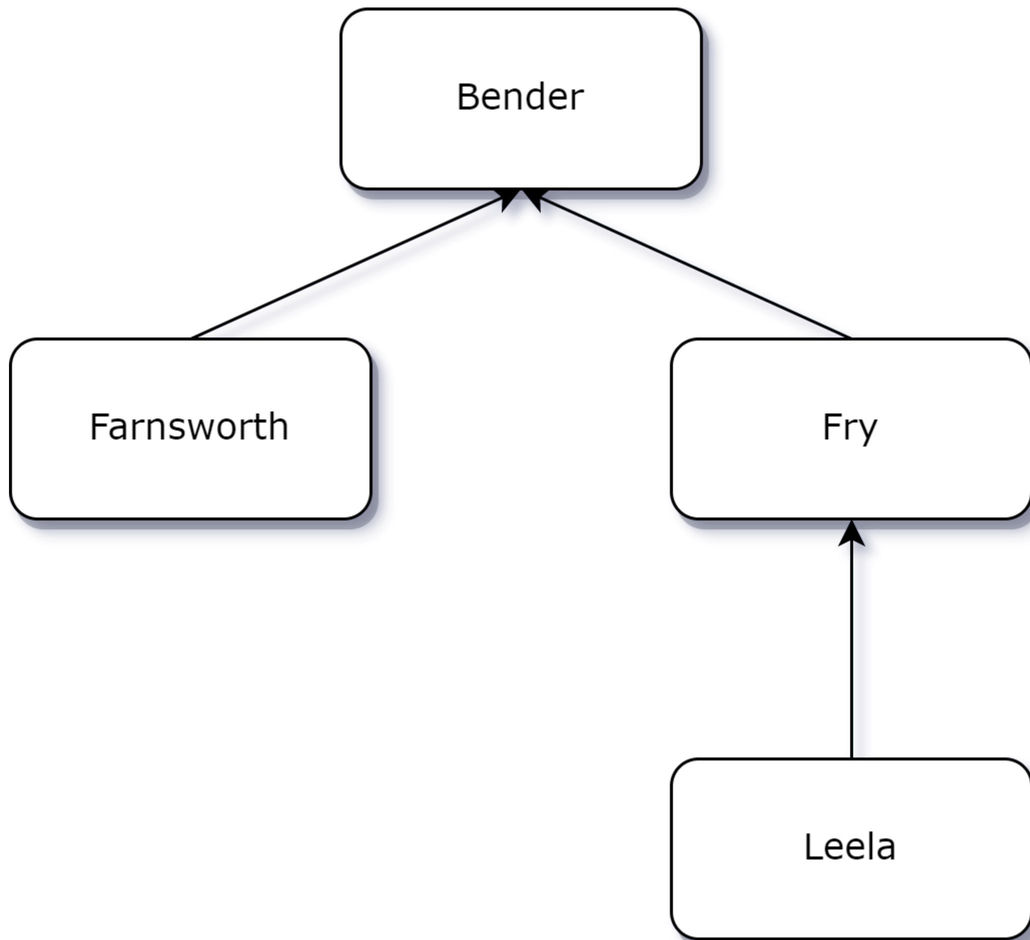
For each call below, indicate what is returned:

Call	Return
mystery(3445, 5)	**34455**
mystery(0, 0)	**0**
mystery(-101, 1)	**-11011**
mystery(323, 3)	**33233**
mystery(12345, 6)	**12345**

Part C: Consider the following classes:

```
public class Leela extends Fry {
    public void method1() {
        System.out.print("Leela1 ");
    }

    public void method2() {
        System.out.print("Leela2 ");
        super.method2();
    }
}

public class Farnsworth extends Bender {
    public void method1() {
        System.out.print("Farnsworth1 ");
    }

    public String toString() {
        return "Good news everyone!";
    }
}
```

```
public class Fry extends Bender {
    public void method2() {
        System.out.print("Fry2 ");
        super.method2();
    }
}

public class Bender {
    public void method1() {
        System.out.print("Bender1 ");
    }

    public void method2() {
        System.out.print("Bender2 ");
        method1();
    }

    public String toString() {
        return "We're doomed!";
    }
}
```



Given the classes above, what output is produced by the following code?

```
Bender[] rodriguez = {new Leela(), new Bender(), new Farnsworth(), new Fry()};
for (int i = 0; i < rodriguez.length; i++) {
    rodriguez[i].method2();
    System.out.println();
    System.out.println(rodriguez[i]);
    rodriguez[i].method1();
    System.out.println();

}
```

Indicate what the output of each of the following statements would be.

elements[0]: Leela	Leela2 Fry2 Bender2 Leela1 We're doomed! Leela1
elements[1]: Bender	Bender2 Bender1 We're doomed! Bender1
elements[2]: Farnsworth	Bender2 Farnsworth1 Good news everyone! Farnsworth1
elements[3]: Fry	Fry2 Bender2 Bender1 We're doomed! Bender1

3. Linked List Debugging

Consider a method in the **LinkedIntList** class called **removeAll** that removes all occurrences of a particular value. For example, suppose the variable **list** contains the following list:

list = [3, 9, 4, 2, 3, 8, 17, 4, 3, 18]

Then, after a call to **list.removeAll(3)** is made, the list would then store the elements:

list = [9, 4, 2, 8, 17, 4, 18]

If the list is empty or the value doesn't appear in the list at all, then the list should not be changed by your method. You must preserve the original order of the elements of the list.

Consider the following implementation of **removeAll**:

```
1  public void removeAll(int n) {
2      if (front != null) {
3          while (front.data == n) {
4              front = front.next;
5          }
6          ListNode current = front;
7          while (current != null && current.next != null) {
8              if (current.next.data == n) {
9                  current.next = current.next.next;
10             } else {
11                 current = current.next;
12             }
13         }
14     }
15 }
```

Part A: When reviewing this implementation, you discover that the code contains a bug that is causing it to not work as intended. In order to communicate what the bug is, you decide to write a test case. First, you consider what elements **list** needs to have to expose the bug. Then, you consider what input **x** should be, such that calling **list.removeAll(x)** triggers the bug. Finally, you consider what the expected **list** should look like, assuming it was implemented correctly.

What should **list** be to expose the bug? (Provide your answer in the form of a list "[1, 2, 3]")

[3, 3, 3]

What does **x** need to be such that **list.removeAll(x)** would trigger the bug?

list.removeAll(3)

Assuming the method was implemented correctly, what should the expected output of **list** be after the method call from above?

[]

Part B: Annotate (write on) the code below to indicate how you would fix the bug. You may add (using arrows to indicate where to insert), remove (by crossing out), or modify (with a combination) any code you choose. However, the fix should not require a lot of work.

```
1  public void removeAll(int n) {
2      if (front != null) {
3          while (front != null && front.data == n) {
4              front = front.next;
5          }
6          ListNode current = front;
7          while (current != null && current.next != null) {
8              if (current.next.data == n) {
9                  current.next = current.next.next;
10             } else {
11                 current = current.next;
12             }
13         }
14     }
15 }
```

4. Inheritance Programming

You have been asked to extend a pre-existing class `Student` that represents a college student. A student has a name, a year (such as 1 for freshman and 4 for senior), and a set of courses that he/she is taking. The `Student` class is as follows:

```
public class Student {
	private String name;
	private int year;
	private Set<String> courses;

	public Student(String name, int year) {
		this.name = name;
		this.year = year;
		this.courses = new HashSet<>();
	}

	public void addCourse(String name) {
		courses.add(name);
	}

	public void dropAll() {
		courses.clear();
	}

	public int getCourseCount() {
		return courses.size();
	}

	public String getName() {
		return name;
	}

	public String getTuition() {
		return 1234.50 * courses.size();
	}

	public int getYear() {
		return year;
	}
}
```

You are to define a new class called **`GradStudent`** that extends **`Student`** through inheritance. A **`GradStudent`** should behave like a **`Student`** except for the following differences:

- A grad student keeps track of a research advisor, which is a professor working with the student.
- Grad students are considered to be 4 years further ahead than typical students. So, for example, a grad student in year 1 of grad school is really in year 5 of school overall.
- Grad students can enroll in a maximum of 3 courses at a time. If a grad student tries to add additional courses beyond 3, the course is not added to the student's set of courses.
- If grad students work too much, they become "burnt out." A burnt-out student is one who is in his/her 5th or higher year of grad school (9th or higher year of school overall) or one who is taking 3 courses.

You should provide the same methods as the superclass, as well as the following new behavior.

Constructor/Method	**Description**
`public GradStudent(String name, int year, String advisor)`	Constructs a graduate student with the given name, given year (where 1 is the first year of grad school, 5th year of school overall; etc.), and research advisor
`public String getAdvisor()`	Returns this grad student's research advisor
`public boolean isBurntOut()`	Returns true if grad student is "burnt out" (in at least the 5th year of grad school, and/or taking 3 courses)

You must also make **`GradStudent`** objects comparable to each other using the **`Comparable`** interface. **`GradStudent`**s are compared by year in ascending order, breaking ties by count of courses in ascending order, and further breaking ties by advisor in ascending alphabetical order. In other words, a **`GradStudent`** object with a lower year is considered to be "less than" one with a higher year. If two students have the same year, the one with a lower count of courses is considered to be "less than" one with a higher course count. If the two students have the same year and the same number of courses, the one whose advisor's name comes first in alphabetical order is considered "less." (You should compare the strings as-is and not alter their casing, spacing, etc.). If the two objects are the same by all three of these criteria, they are considered to be "equal."

Write your solution to problem #4 here:

One possible solution

```
public class GradStudent extends Student implements Comparable<GradStudent> {
    private String advisor;

      public GradStudent(String name, int year, String advisor) {
        super(name, year + 4);
        this.advisor = advisor;
    }

    public String getAdvisor() {
        return this.advisor;
    }

    public boolean isBurntOut() {
        return super.getCourseCount() >= 3 || super.getYear() >= 9;
    }

    @Override
    public void addCourse(String name) {
        if (super.getCourseCount() < 3) {
            super.addCourse(name);
        }
    }

    @Override
    public int compareTo(GradStudent other) {
        if (this.getYear() != other.getYear()) {
            return this.getYear() - other.getYear();
        } else if (this.getCourseCount() != other.getCourseCount()) {
            return this.getCourseCount() - other.getCourseCount();
        } else {
            return this.advisor.compareTo(other.advisor);
        }
    }
}
```

5. Recursive Programming

You are standing at the base of a staircase and are heading to the top. A small stride will move up one stair, and a large stride will advance two. You want to count the number of ways to climb the entire staircase based on different combinations of large and small strides. For example, a staircase of three steps can be climbed in three different ways: three small strides, one small stride followed by one large stride, or one large followed by one small.

Write a recursive method **`waysToClimb`** that takes a non-negative integer value representing the number of stairs and prints each unique way to climb a staircase of that height, taking strides of one or two stairs at a time. Your method should output each way to climb the stairs on its own line, using a 1 to indicate a small stride of 1 stair, and a 2 to indicate a large stride of 2 stairs. For example, the call of **`waysToClimb(3)`** should produce the following output:

```
[1, 1, 1]
[1, 2]
[2, 1]
```

The call of **`waysToClimb(4)`** would produce the following output:

```
[1, 1, 1, 1]
[1, 1, 2]
[1, 2, 1]
[2, 1, 1]
[2, 2]
```

You may print out the possible ways to climb the stairs in any order, so long as you list the right overall set of ways. There are no ways to climb zero stairs, so your method should produce no output if 0 is passed.

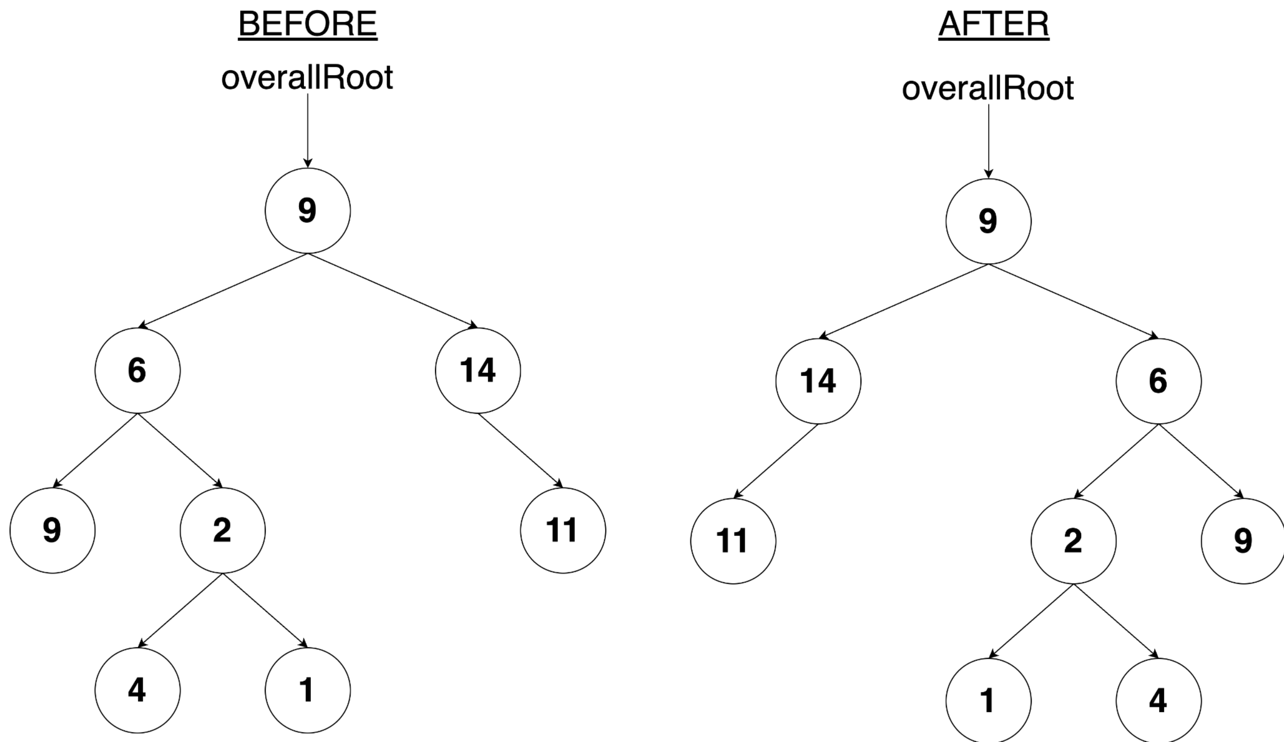
Write your solution to problem #5 here:

One possible solution

```
public void waysToClimb(int n) {
    if (n > 0) {
        int[] steps = {1, 2};
        List<Integer> result = new ArrayList<>();
        waysToClimb(n, steps, result);
    }
}

private void waysToClimb(int n, int[] steps, List<Integer> result) {
    if (n == 0) {
        System.out.println(result);
    } else if (n > 0) {
        for (int step : steps) {
            result.add(step);
            waysToClimb(n - step, steps, result);
            result.remove(result.size() - 1);
        }
    }
}
```

6. Binary Tree Programming

Write a method **`mirror`** that converts a binary tree of integers to its mirror image. For example, if a variable called **`t`** stores a reference to a binary tree then **`t.mirror()`** then the links of the tree should be rearranged so that after the call, **`t`** stores the mirror image of what it stored before the call. Below is a specific example of how a tree would change:



After the call is made, the tree stores the structure that you would see if you were to hold the original tree's diagram up to a mirror. More precisely, the overall root (if it exists) remains in the same place in the mirror image and every other node is moved so that it's new path from the overall root is composed of the old path from the overall root with left and right links exchanged. For example, the node that stores 4 in the example above has the path (left, right, left) in the original tree. In the mirror image, it has path (right, left, right). The node that stores 1 has the path (left, right, right) in the original tree. In the mirror image it has path (right, left, left).

You are writing a method that will become part of the **`IntTree`** class. You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class. You may not construct any new nodes, and you may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc). You also may not change any data fields of the nodes. **<u>You MUST solve this problem by rearranging the links of the tree.</u>**

Write your solution to problem #6 here:

One possible solution

```
public void mirror() {
    overallRoot = mirror(overallRoot);
}

public IntTreeNode mirror(IntTreeNode root) {
    if (root == null) {
        return null;
    } else {
        IntTreeNode mirrorLeft = mirror(root.left);
        IntTreeNode mirrorRight = mirror(root.right);
        root.left = mirrorRight;
        root.right = mirrorLeft;
        return root;
    }
}
```

This page intentionally left blank for scratch work