1 Asymptotics

1.1 Give a tight asymptotic bound for `mystery`.

```
void mystery(int N) {
    for (int i = 1; i <= N * N; i *= 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("moo");
        }
    }
}
```

1.2 Give a tight asymptotic runtime bound for `mysterySearch` as a function of N, the size of the array, in the *best case, worst case, and overall.* Assume the array is sorted.

```
public static boolean mysterySearch(int[] a, int value) {
    if (Math.random() < 0.5) {
        return linearSearch(a, value, 0);
    } else {
        return binarySearch(a, value, 0, a.length);
    }
}
```

2 T,F,G,V,E

2.1 State if the following statements are True or False, and justify. For all graphs, assume that edge weights are positive and distinct, unless otherwise stated.

(a) If a graph G has a unique MST, it must have unique edge weights.

(b) Adding some positive constant k to every edge weight does not change the minimum spanning tree.

(c) Doubling every edge weight does not change the minimum spanning tree.

(d) Let $(S, V - S)$ be a specific cut of the graph. If an edge e is not the lightest edge across this cut, it cannot be a part of any MST.

3 Redundant Connections

3.1 For any connected graph with N vertices and N edges, there is at least 1 edge whose removal will keep the graph connected (graph would become a tree).

Given a graph represented with an **edge set**, design an algorithm that would find the K redundant edges of minimum weight. If there are not K redundant edges, return all redundant edges.

Hint: Consider how Kruskal's Algorithm checks for a cycle.

This problem is adapted from Leetcode: https://leetcode.com/problems/redundant-connection/solution/

4 Threads

4.1 For most of the programming assignments in CS 61B, we would write some code in Java, compile it with the `javac` command, then execute it with the `java` command. To truly understand threads, let's dive deeper into the compilation and execution stages.

The Java compiler is invoked by the `javac` command and turns human-written Java code into an *executable* file written in Java bytecode. The executable is the `sample.class` file that results from running `javac sample.java` in the command line.

This executable is a Java program. When we later run this executable (with `java sample.class`), our operating system creates a new process to run the program. By definition, a *program* is the result of compiling code and a *process* is a program currently in execution.

A process that executes a Java program consists of multiple threads. A thread is an independent execution sequence of code. If you have taken CS 61A, one way to think about it is that each thread has its own environment diagram. Another way to think about it is that each thread executes its own chunk of code.

One of the reasons why Java is considered a "high-level" programming language is because each of these threads (within a running Java program) has its own specialized task. For instance, one thread executes the code that we have written (i.e. the `main()` function), another thread frees up unused memory (i.e. garbage collection), another thread may update the display, etc.

Threads within the same process share the same memory. This is extremely useful in the age of parallel computing since it allows us to take advantage of multi-core processors. However, there lies danger in the concurrent access of shared memory (i.e. race conditions). With great power comes great responsibility, and CS 61C and CS 162 will teach you methods to write code that is *thread-safe*.

4.2 Let's explore how we can write multi-threaded Java programs. Due to COVID-19, CSM would like to start a mask donation and distribution program for the needy. Here is the single-threaded version:

```java
public class MaskDonationDrive {
    public static int mask_count = 0;
    public static int getCount() { return mask_count; }
    public static void changeCount(int m) { mask_count += m; }

    public static void main(String[] args) {
        while (true) {
            Scanner scanner = new Scanner(System.in);
            System.out.println("How many masks would you like to donate?");

            // Suspends execution until user specifies an integer at the command-line.
            int donated_masks = scanner.nextInt();

            System.out.println("Thank you for donating " + donated_masks + " masks!");
            MaskDonationDrive.changeCount(donated_masks);

            int mask_count = MaskDonationDrive.getCount();
            if (mask_count > 0) {
                System.out.println("Donated " + mask_count + " masks to the needy!");
                MaskDonationDrive.changeCount(-1 * mask_count);
            }
        }
    }
}
```

Fill in the skeleton code below for the multi-threaded version of the program.
One thread will constantly receive donations and update the mask count.
The other thread will distribute donations to the needy every 5 seconds.

```java
public class MaskDonationDrive {
    public static int mask_count = 0;

    public static int getCount() { return mask_count; }

    public static void changeCount(int m) { mask_count += m; }

    public static void main(String[] args) {
        Thread donator = new Thread(new Donator());
        Thread distributor = new Thread(new Distributor());

        _____________________________________________________;
        _____________________________________________________;
    }
```

```
}

public class Donator extends MaskDonationDrive implements Runnable {
    public void run() {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("How many masks would you like to donate?");

            _____________________________________________________;
            _____________________________________________________;
            _____________________________________________________;
            _____________________________________________________;
        }
    }
}

public class Distributor extends MaskDonationDrive implements Runnable {
    public void run() {
        while (true) {
            Thread.sleep(5000); // Suspends execution of the thread for 5 seconds.
            _____________________________________________________;
            _____________________________________________________;
            _____________________________________________________;
            _____________________________________________________;
            }
        }
    }
}
```