Programming Assignment 2: Disaster Relief

Specification

(This assignment was partially inspired by Keith Schwarz's 2020 Nifty Assignment.)

Background

When natural disasters strike, governments, relief organizations, and even individual donors must often wrestle with how best to allocate available resources to help those who have been affected. This is generally a very complex decision, balancing countless logistical, economic, political, and other factors. One particular challenge is that different geographic areas can require different financial or other resources for relief, even if the populations of the areas are similar. (Or, put another way, the cost to help a single person after a disaster is not always constant.) Organizations sometimes have to make difficult decisions in the hope of helping as many people as possible with the available resources.

In this assignment, you will implement a system to determine how to allocate a budget of relief resources to help as many people as possible.

NOTE: While our simulation will focus on helping the greatest number of people for the least amount of money, this is an oversimplification of the problem of allocating resources in the wake of a disaster, and may not necessarily be the best approach.

Learning Objectives

By completing this assignment, students will demonstrate their ability to:

- Define a solution to a given problem using a recursive approach
- Write functionally correct recursive methods
- Produce clear and effective documentation to improve comprehension and maintainability of a method
- Write methods that are readable and maintainable, and that conform to provided guidelines for style and implementation

Required Methods

For this assignment, you will implement only a single method:

```java
public static Allocation allocateRelief(double budget, List<Region> sites)
```

This method takes a budget and a list of `Region` objects as parameter. The method will compute and return the allocation of resources that will result in the most people being helped with the given budget. If there is more than one allocation that will result in the most people being helped, the method will return the allocation that costs the least. If there is more than one allocation that will result in the most people being helped for the lowest cost, you may return any of these allocations.

For the purposes of our simulation, we will assume that providing relief to a region is *atomic*, meaning that either all people in the region are helped and the full cost is paid, or no relief is allocated to that region. We will not deal with the possibility of providing partial relief to a particular region.

You should implement your `allocateRelief` method where indicated in the provided `Client.java` file. You may also implement any additional helper methods you might like. (For example, you will likely want to implement a public-private pair in your algorithm.)

`Region` class

In our system, we will represent areas that may be allocated relief funds with the following `Region` class (comments and some methods are omitted here; see the full `Region` class in the coding challenge slide for these):

▼ Expand

```java
import java.util.*;

public class Region {
    private String name;
    private int population;
    private double baseCost;

    public Region(String name, int pop, double baseCost) {
        this.name = name;
        this.population = pop;
        this.baseCost = baseCost;
    }

    public int getPopulation() { return this.population; }

    public double getCost(int index) {
        return (1 + 0.1 * index) * this.baseCost;
    }

    public String toString() {
        return name + ": pop. " + population + ", base cost: $" + baseCost;
    }
}
```

`Allocation` class

We will represent a List of regions that will receive resources with the following `Allocation` class (comments and some methods are omitted here; see the full class in the coding challenge slide for these):

▼ Expand

```java
public class Allocation {

    private List<Region> regions;

    private Allocation(List<Region> regions) {
        this.regions = new ArrayList<>(regions);
    }

    public Allocation() {
        this(new ArrayList<>());
    }

    public List<Region> getRegions() {
        return new ArrayList<>(regions);
    }

    public Allocation withRegion(Region r) {
        if (regions.contains(r)) {
            throw new IllegalArgumentException("Allocation already contains region " + r);
        }
        List<Region> newRegions = new ArrayList<>(regions);
        newRegions.add(r);
        return new Allocation(newRegions);
    }

    public Allocation withoutRegion(Region r) {
        if (!regions.contains(r)) {
            throw new IllegalArgumentException("Alloction doesn't contain region " + r);
        }
        List<Region> newRegions = new ArrayList<>(regions);
        newRegions.remove(r);
        return new Allocation(newRegions);
    }

    public int size() {
        return regions.size();
    }

    public int totalPeople() {
        int total = 0;
        for (Region r : regions) {
            total += r.getPopulation();
        }
        return total;
    }

    public double totalCost() {
        double total = 0;
        for (int i = 0; i < regions.size(); i++) {
            total += regions.get(i).getCost(i);
        }
        return total;
    }

    public String toString() {
        return regions.toString();
    }
}
```

The two methods `withRegion` and `withoutRegion` can be used to add/remove a `Region` to/from an `Allocation`. **Notice that these methods *return a new* `Allocation` rather than modifying an existing `Allocation`**, similar to how `String` methods like `substring` or `toUpperCase` return a new `String` rather than modifying an existing one:

```
Allocation empty = new Allocation();
Region one = new Region("Region #1", 50, 500);

Allocation added = empty.withRegion(one);
Allocation removed = added.withoutRegion(one);
```

Make sure you write your implementation accordingly.

Client Program

We have provided a client program that will allow you to test your `allocateRelief` implementation. This client provides two methods that might be useful.

```
public static List<Region> createSimpleScenario()
```

- Manually creates a simple list of regions to represent a known scenario.
  - We have provided one example in the client code, and a few others in the examples below.

```
public static List<Region> createRandomScenario(int numRegions, int minPop, int maxPop, double minC
                                                double maxCostPer)
```

- Creates a scenario with `numRegions` regions by randomly choosing the population and cost of each region.
  - Populations will be chosen between `minPop` and `maxPop` (inclusive)
  - Costs will be generated by choosing a random value between `minCostPer` and

`maxCostPer` (inclusive) and multiplying that cost by the chosen population.

You can modify `createSimpleScenario` with different `Region` objects to test your implementation in scenarios of your own design, and/or you can generate random scenarios to try using `createRandomScenario`.

Click "Expand" below to see some example scenarios, their results, and visualizations of the decision trees.

▼ Expand

Example 1:

Input:

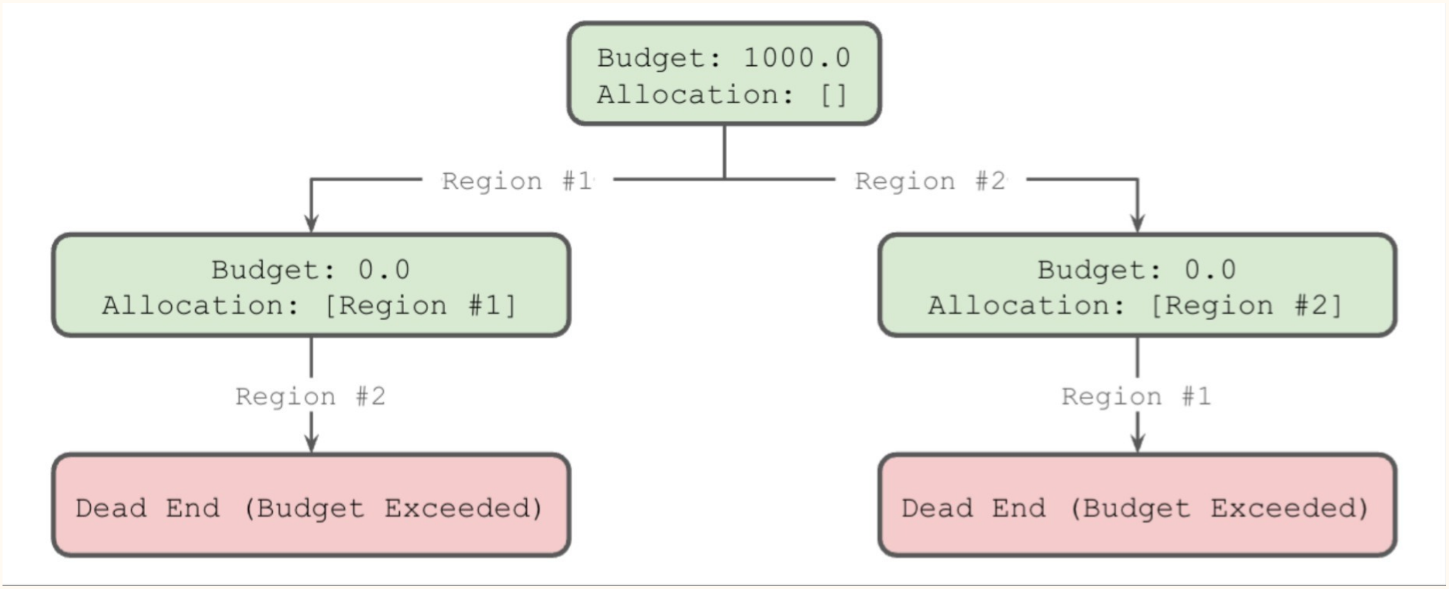
```
double budget = 1000;
```

```
public static List<Region> createSimpleScenario() {
    List<Region> result = new ArrayList<>();
    result.add(new Region("Region #1", 50, 1000));
    result.add(new Region("Region #2", 100, 1000));
    return result;
}
```

Output:

```
[Region #1: pop. 50, base cost: $1000.0, Region #2: pop. 100, base cost: $1000.0]
Result:
  [Region #2: pop. 100, base cost: $1000.0]
  People helped: 100
  Cost: $1000.00
  Unused budget: $0.00
```

Decision Tree:



▼ Expand

Example 2:

Input:

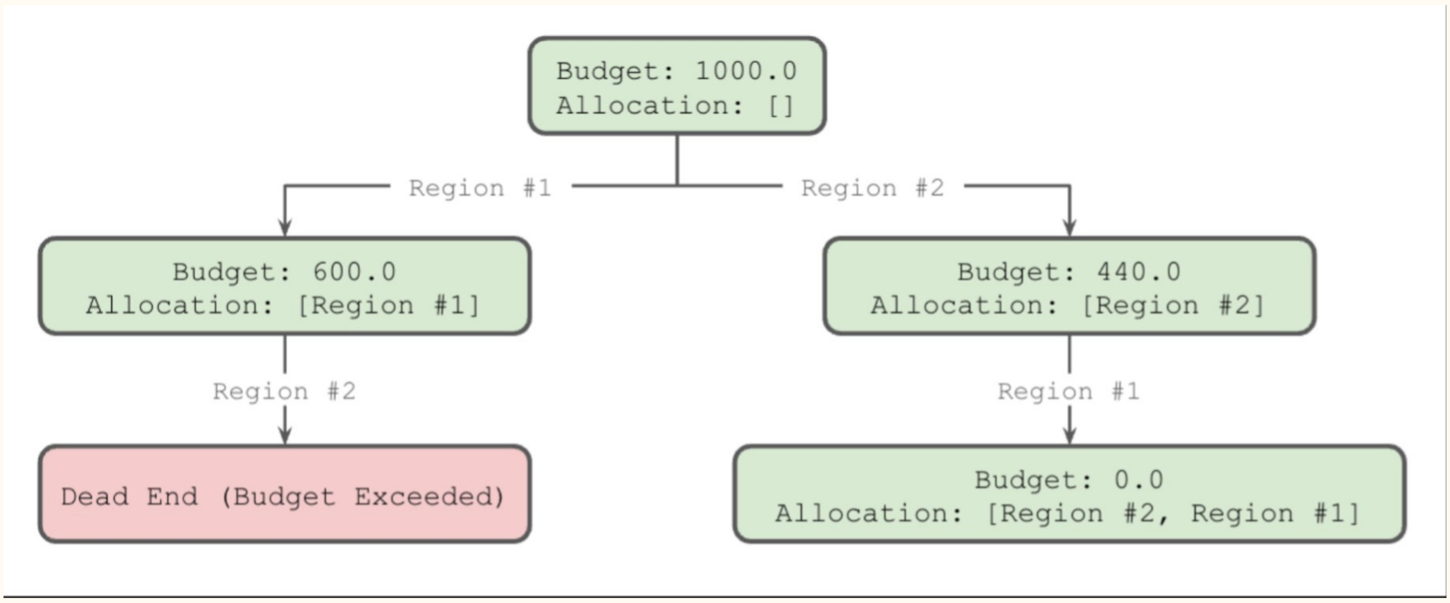
```
double budget = 1000;
```

```
public static List<Region> createSimpleScenario() {
    List<Region> result = new ArrayList<>();
    result.add(new Region("Region #1", 50, 400));
    result.add(new Region("Region #2", 50, 560));
    return result;
}
```

Output:

```
[Region #1: pop. 50, base cost: $400.0, Region #2: pop. 50, base cost: $560.0]
Result:
  [Region #2: pop. 50, base cost: $560.0, Region #1: pop. 50, base cost: $400.0]
  People helped: 100
  Cost: $1000.00
  Unused budget: $0.00
```

Decision Tree:



▼ Expand

Example 3:

Input:

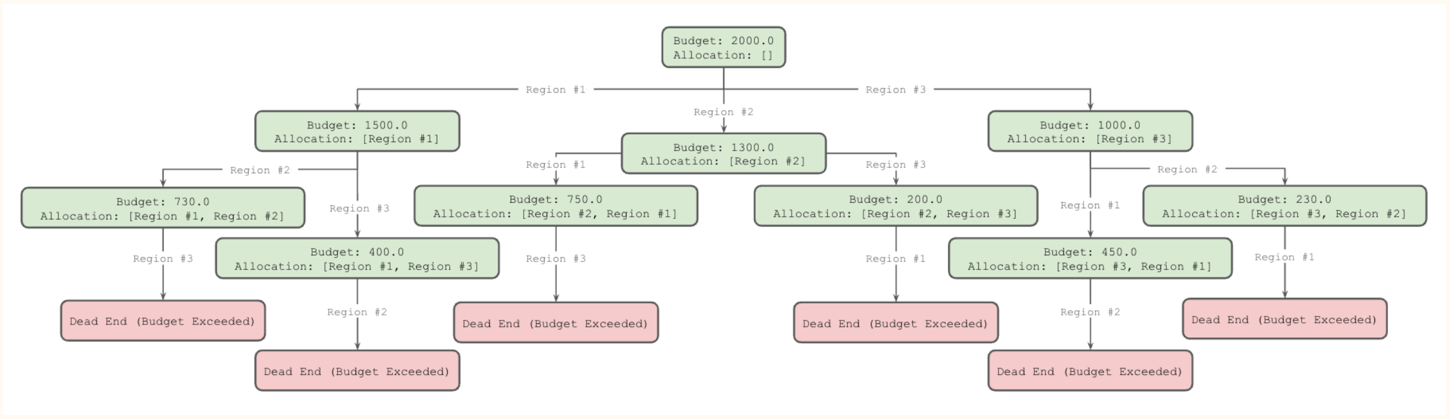
```
double budget = 2000;
```

```java
public static List<Region> createSimpleScenario() {
    List<Region> result = new ArrayList<>();
    result.add(new Region("Region #1", 50, 500));
    result.add(new Region("Region #2", 100, 700));
    result.add(new Region("Region #3", 60, 1000));
    return result;
}
```

Output:

```
[Region #1: pop. 50, base cost: $500.0, Region #2: pop. 100, base cost: $700.0, Region #3: pop.
Result:
  [Region #3: pop. 60, base cost: $1000.0, Region #2: pop. 100, base cost: $700.0]
  People helped: 160
  Cost: $1770.00
  Unused budget: $230.00
```

Decision Tree:


Budget: 2000.0
Allocation: []
Region #1
Region #2
Region #3
Budget: 1500.0
Allocation: [Region #1]
Budget: 1300.0
Allocation: [Region #2]
Budget: 1000.0
Allocation: [Region #3]
Region #2
Region #3
Region #1
Region #3
Region #2
Region #1
Budget: 730.0
Allocation: [Region #1, Region #2]
Budget: 400.0
Allocation: [Region #1, Region #3]
Budget: 750.0
Allocation: [Region #2, Region #1]
Budget: 200.0
Allocation: [Region #2, Region #3]
Budget: 450.0
Allocation: [Region #3, Region #1]
Budget: 230.0
Allocation: [Region #3, Region #2]
Region #3
Region #2
Region #3
Region #1
Region #2
Region #1
Dead End (Budget Exceeded)
Dead End (Budget Exceeded)
Dead End (Budget Exceeded)
Dead End (Budget Exceeded)
Dead End (Budget Exceeded)
Dead End (Budget Exceeded)

▼ Expand

Example 4 (same as Example 3, but with Region #3's population as 50 instead):

Input:

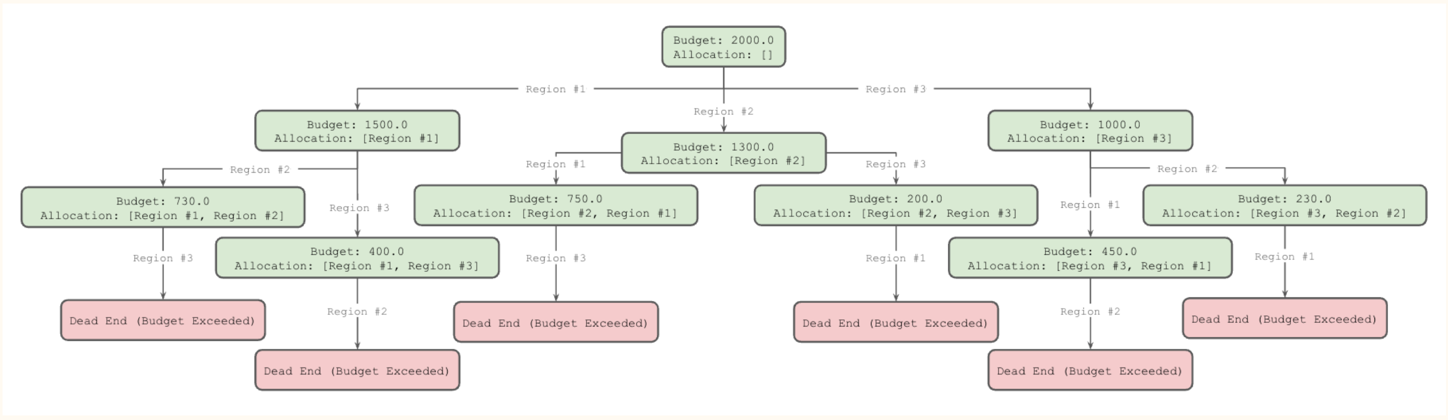
```java
double budget = 2000;
```

```java
public static List<Region> createSimpleScenario() {
    // Same regions as Example 2 but Location 3 has a population of 50
    List<Region> result = new ArrayList<>();
    result.add(new Region("Region #1", 50, 500));
    result.add(new Region("Region #2", 100, 700));
    result.add(new Region("Region #3", 50, 1000));
    return result;
}
```

Output:

```
[Region #1: pop. 50, base cost: $500.0, Region #2: pop. 100, base cost: $700.0, Region #3: pop.
Result:
  [Region #2: pop. 100, base cost: $700.0, Region #1: pop. 50, base cost: $500.0]
  People helped: 150
  Cost: $1250.00
  Unused budget: $750.00
```

Decision Tree:


Budget: 2000.0
Allocation: []
Region #1
Region #2
Region #3
Budget: 1500.0
Allocation: [Region #1]
Budget: 1300.0
Allocation: [Region #2]
Budget: 1000.0
Allocation: [Region #3]
Region #2
Region #3
Region #1
Region #3
Region #1
Region #2
Budget: 730.0
Allocation: [Region #1, Region #2]
Budget: 400.0
Allocation: [Region #1, Region #3]
Budget: 750.0
Allocation: [Region #2, Region #1]
Budget: 200.0
Allocation: [Region #2, Region #3]
Budget: 450.0
Allocation: [Region #3, Region #1]
Budget: 230.0
Allocation: [Region #3, Region #2]
Region #3
Region #2
Region #3
Region #1
Region #2
Region #1
Dead End (Budget Exceeded)
Dead End (Budget Exceeded)
Dead End (Budget Exceeded)
Dead End (Budget Exceeded)
Dead End (Budget Exceeded)
Dead End (Budget Exceeded)

Note: Each of the green nodes in the trees represent nodes that should be added to the result of `generateOptions` as a possible allocation.

Notice that the **ordering of regions in your allocation matters**. For example, a list ordered `[Region #1: pop. 100, base cost: $1000.0, Region #2: pop. 50, base cost: $200.0]` and a list ordered `[Region #2: pop. 50, base cost: $200.0, Location #1: pop. 100, base cost: $1000.0]` are different in their cost. Specifically, the later a region appears in an `Allocation`, the more it will cost to help people within that region. You can consider this being due to the situation growing more dire as time progresses. You should focus on the fact that **different orderings of the same regions are considered different allocations** because of this.

You don't need to worry about the details of how the cost computed; calling the `Allocation`'s `totalCost()` method will return the correct total for you. You do not need to directly call the `getCost()` method of `Region`.

You may create your own client programs if you like, and you may modify the provided client if you find it helpful. However, **your methods must work with the provided files without modification and must meet all requirements below**.

Testing Requirements

For this assignment, you'll be required to implement **four** total JUnit tests. The first three should cover the following cases:

▼ Expand

Case 1:

```
budget: 500
sites:
    name: Region #1, population: 100, base cost: 400
    name: Region #2, population 150, base cost 600
```

Think through this case and what the result should be, then expand to see the expected result:

▶ Expand

▼ Expand

Case 2:

```
budget: 500
sites:
    name: Region #1, population: 150, base cost: 400
    name: Region #2, population 100, base cost 450
```

Think through this case and what the result should be, then expand to see the expected result:

▶ Expand

▼ Expand

Case 3:

```
budget: 500
sites:
    name: Region #1, population: 150, base cost: 450
    name: Region #2, population 150, base cost 400
```

Think through this case and what the result should be, then expand to see the expected result:

▶ Expand

The fourth test should be a test case you come up with on your own. You are not required to look inside the `Region` class to determine how the cost is determined, therefore it is totally fine for this test case to be rather simplistic. **Our only requirement for this fourth test is that the inputted `sites` contains at least two regions.**

All four tests should be placed in their **own methods** within the provided `Testing.java` file. You're welcome to implement tests other than the ones outlined here, but doing so is not required.

Implementation Requirements

To earn a grade higher than N on the Behavior and Concepts dimensions of this assignment, **your algorithm must be implemented *recursively*. You will want to utilize the *public-private pair* technique discussed in class.** You are free to create any helper methods you like, but the core of your algorithm (specifically, building and potentially evaluating possible allocations of relief funds) must be recursive.

Additionally, for this assignment, you should follow the Code Quality guide when writing your code to ensure it is readable and maintainable. In particular, you should focus on the following requirements:

- Avoid recursing any more than you need to. Your method should not continue to explore a path when the current `Allocation` is no longer viable.
- Watch out for branches of an `if` / `else` statement that shares the same exact code. You should combine the conditionals and write the code only once.
- Make sure that all parameters within a method are used and necessary.
- You should comment your code following the Commenting Guide. You should write comments with basic info (a header comment at the top of your file), a class comment for your `Mondrian` class, and a comment for every method.
    - Make sure to avoid including *implementation details* in your comments. In particular, for your object class, a *client* should be able to understand how to use your object effectively by only reading your class and method comments, but your comments should maintain *abstraction* by avoiding implementation details.
    - Continuing with the previous point, keep in mind that the client should **not** be aware of what implementation strategy your class/methods utilize.
- All methods present in your class that are not listed in the specification must be private.

Disaster Relief

Download Starter Code

P2_DisasterRelief.zip

Remember that you're required to write four tests, each within their own method in `Testing.java`. More information on this requirement can be found in the spec.

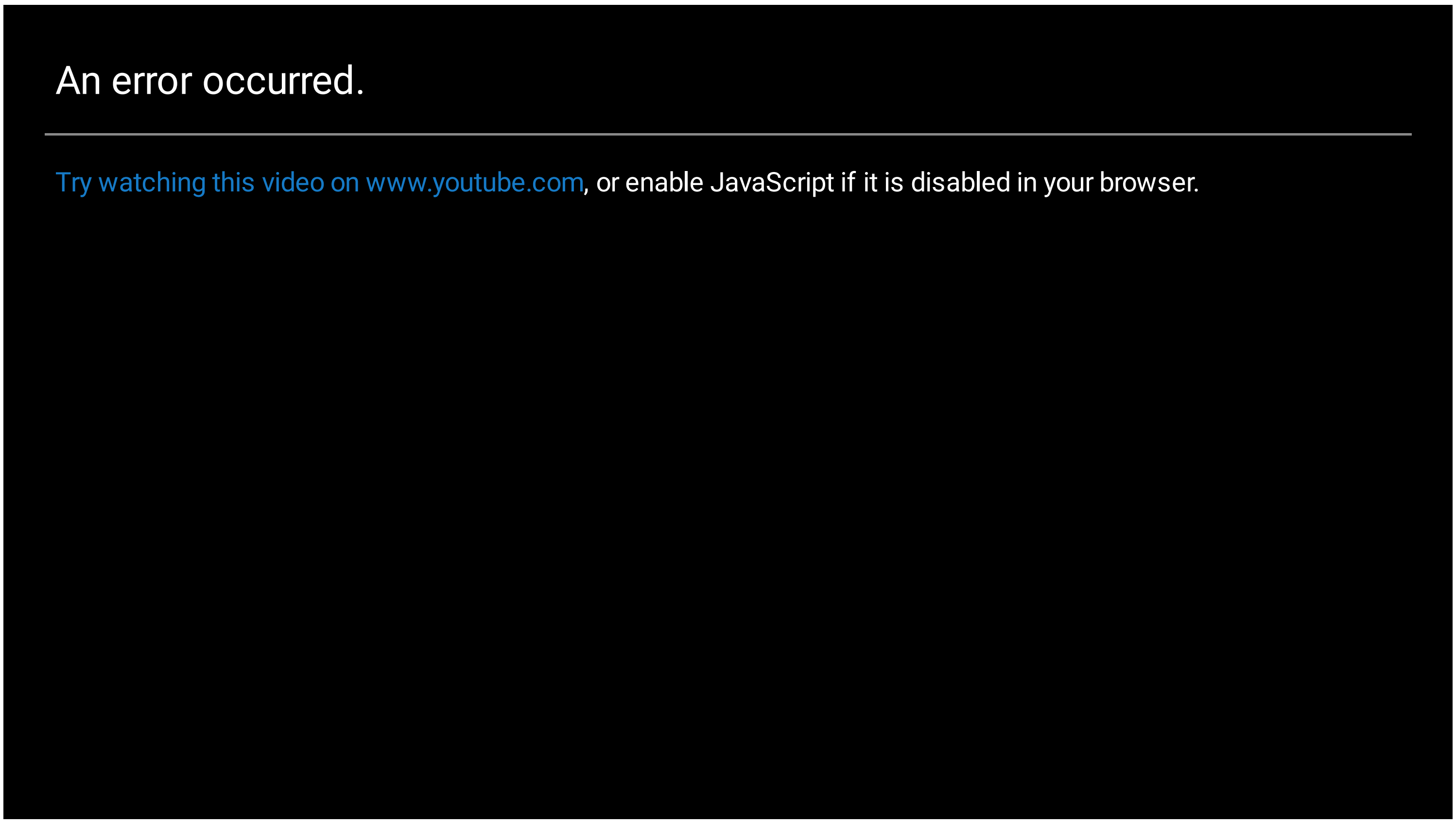
Reflection

The following questions will ask you practice **metacognition** to reflect on the topics covered on this assignment and your experience completing it. For each question, focus on your plan and/or process for working through the assignment along with the CS concepts. Think about things like how you organized your working time, what sorts of things tended to go wrong, and how you dealt with those errors or mistakes.

Please answer all questions.

Question 1

The first 3 questions will require you to reflect about potential benefits and drawbacks in employing algorithms to improve societal welfare. Start by watching a short segment of the following talk from UC Berkeley professor Rediet Abebe (2m 9s to 8m 57s):


An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

(https://youtu.be/h1NqpK4gDrM?t=129)

How do you think measurement challenges (such as data sparsity and inaccurate metrics) affect the effectiveness of algorithms in resource allocation for disaster relief?

No response

Question 2

What are the potential risks of relying on simple metrics (like income or population density) when allocating disaster relief resources, and how might these risks be mitigated?

No response

Question 3

Explain what income shocks are. How might similar ‘shocks’ or unforeseen events affect disaster relief efforts, and how could an algorithm be designed to handle these sudden needs?

No response

Question 4

Describe how you went about testing your implementation. What specific situations and/or test cases did you consider? Why were those cases important?

No response

Question 5

What skills did you learn and/or practice with working on this assignment?

No response

Question 6

What did you struggle with most on this assignment?

No response

Question 7

What questions do you still have about the concepts and skills you used in this assignment?

No response

Question 8

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

No response

Question 9

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

No response

Question 10

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

No response

🚀 Final Submission 🚀

🚀 Final Submission🚀

Fill out the box below and click "Submit" in the upper-right corner of the window to submit your work.

Question

I attest that the work I am about to submit is my own and was completed according to the course Academic Honesty and Collaboration policy. If I collaborated with any other students or utilized any outside resources, they are allowed and have been properly cited. If I have any concerns about this policy, I will reach out to the course staff to discuss *before* submitting.

(Type "yes" as your response.)

No response