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.

Important 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. For one discussion of this problem, see this talk from UC Berkeley professor Rediet Abebe.

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:

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

This method takes a budget and a list of `Location` 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 location is *atomic*, meaning that either all people in the location are helped and the full cost is paid, or no relief is allocated to that location. We will not deal with the possibility of providing partial relief to a particular location.

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 for `allocateRelief`.)

`Location` class

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

```java
import java.util.*;

public class Location {
    private String name;
    private int population;
    private double cost;

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

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

    public double getCost() { return this.cost; }

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

`Allocation` class

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

```
import java.util.*;
public class Allocation {

    private Set<Location> locations;

    private Allocation(Set<Location> locations) {
        this.locations = new HashSet<Location>(locations);
    }

    public Allocation() {
        this(new HashSet<Location>());
    }

    public Set<Location> getLocations() {
        return new HashSet<Location>(locations);
    }

    public Allocation withLoc(Location loc) {
        if (locations.contains(loc)) {
            throw new IllegalArgumentException("Allocation already contains location " + loc);
        }
        Set<Location> new_locations = new HashSet<Location>(locations);
        new_locations.add(loc);
        return new Allocation(new_locations);
    }

    public Allocation withoutLoc(Location loc) {
        if (!locations.contains(loc)) {
            throw new IllegalArgumentException("Allocation does not contain location " + loc);
        }
        Set<Location> new_locations = new HashSet<Location>(locations);
        new_locations.remove(loc);
        return new Allocation(new_locations);
    }

    public int totalPeople() {
        int total = 0;
        for (Location loc : locations) {
            total += loc.getPopulation();
        }
        return total;
    }

    public double totalCost() {
        double total = 0;
        for (Location loc : locations) {
            total += loc.getCost();
        }
        return total;
    }
}
```

The two methods `withLoc` and `withoutLoc` can be used to add and remove (respectively) a `Location` 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. Make sure you write your code 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<Location> createSimpleScenario()`

- Manually creates a simple list of locations 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 createRandomScenario(int numLocs, int minPop, int maxPop, double minCostPer, double maxCostPer)`

- Creates a scenario with `numLocs` locations by randomly choosing the population and cost of each location.
  - 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 `Location` 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<Location> createSimpleScenario() {
    List<Location> result = new ArrayList<>();
    result.add(new Location("Location #1", 50, 1000));
    result.add(new Location("Location #2", 100, 1000));
    return result;
}
```

Output:

```
Result:
  [Location #2: pop. 100, cost: $1000.0]
  People saved: 100
  Cost: $1000.00
  Unused budget: $0.00
```

Decision Tree:

decisiontree1.pdf

▼ Expand

Example 2:

Input:

```
double budget = 2000;
```

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

Output:

```
Result:
  [Location #3: pop. 60, cost: $1000.0, Location #2: pop. 100, cost: $700.0]
  People saved: 160
  Cost: $1700.00
  Unused budget: $300.00
```

Decision Tree:

decisiontree2.pdf

▼ Expand

Example 3:

Input:

```
double budget = 2000;
```

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

Output:

```
Result:
  [Location #2: pop. 100, cost: $700.0, Location #1: pop. 50, cost: $500.0]
  People saved: 150
  Cost: $1200.00
  Unused budget: $800.00
```

Decision Tree:

decisiontree3.pdf

Note: The dashed lines in the decision tree represent the best set of `Location` objects that is being returned.

Note: The ordering of elements in your set does not matter. For example, a set containing `{Location #1: pop. 100, cost: $1000.0, Location #2: pop. 50, cost: $200.0}` and a set containing `{Location #2: pop. 50, cost: $200.0, Location #1: pop. 100, cost: $1000.0}` are identical.

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

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 evaluating possible allocations of relief funds) must be recursive.

You are **not** required to avoid trying permutations. Since the `Allocation` class does not make a distinction between adding `Location`s A, B, C versus B, C, A, there will not be a difference in cost or results. Both solutions that intentionally avoid trying permutations, and solutions that try permutations are OK.

Development Strategy

We recommend you start by developing a version of the `allocateRelief` method that simply prints all possible allocations within the specified budget. This will be easier than trying to find the optimal allocation and will produce much of the code necessary for the final version. Then, once you have successfully implemented this version, you can modify the code to find and return the allocation that helps the most people as described above.

There is an ***OPTIONAL*** slide to help develop such a method. It is ***not*** *required* nor is it *graded*. Please make sure to transfer whatever work was completed on the slide into the actual Disaster Relief Code Challenge slide!

The Scrabble Helper example from Lesson 12 will be helpful to you in completing this assignment.

OPTIONAL – Generate Options

This code slide does not have a description.

Disaster Relief

This code slide does not have a description.

Testing Comprehension + Spec

In this assignment, you will not only design but also implement your very own unit tests! Before we talk specifically about testing in this assignment, here are some resources for a refresher on jUnit and writing unit tests:

- Testing Comprehension in Mini Git
- Testing Section
- jUnit Cheat Sheet

For this assignment, you will write three different unit tests, that will test different cases for your `allocateRelief` method.

Some of these inputs do not have obvious or intuitive outputs... this is what the spec is for!

Question 1

If our parameters look like:

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

Which `Location`s should our `Allocation` contain?

◯ Location #1

◯ Location #2

◯ Location #1, Location #2

◯ No Location

Question 2

If our parameters look like:

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

Which `Location`s should our `Allocation` contain?

- ○ Location #1
- ○ Location #2
- ○ Location #1, Location #2
- ○ No Location

Question 3

If our parameters look like:

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

Which `Location`s should our `Allocation` contain?

- ○ Location #1
- ○ Location #2
- ○ Location #1, Location #2
- ○ No Location

Question 4

Now, you will implement unit tests covering two of the above cases! On the "Disaster Relief" tab, under the file browser, you should find a file titled "Testing.java". We have provided you with the test for the first case. Fill in the implementation for the other two cases. Then, when you press "Check," you should see your beautiful tests! How do you feel right now?

- ☐ Amazing
- ☐ Incredible
- ☐ Astounding
- ☐ Magnificent

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

Describe one other strategy that could have been used to choose an allocation instead of "help the most people." How would using that strategy have changed your implementation?

No response

Question 2

Do you think *any* algorithmic approach, whether the one you implemented, the one you described above, or another, should be used to determine how to allocate relief funds in the wake of a disaster? Why or why not?

No response

Question 3

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 4

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

No response

Question 5

What did you struggle with most on this assignment?

No response

Question 6

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

No response

Question 7

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

No response

Question 8

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 9

[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