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

System Structure

For this assignment, we will be modeling a collection of *regions* needing relief, each of which has a population of people needing help within it and a given cost for helping that region. All regions are connected to one another, meaning that we can travel from any region to another.

Our goal is to identify an *allocation*, which is a subset of regions, that **aids the most people** for a given budget. In the case of a **tie**, we will favor the allocation that has the **lowest cost**. To accomplish our goal, we will use the following two classes:

`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

```
public class Region {
    private String name;
    private int population;
    private double cost;

    public Region(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 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

```
import java.util.*;

public class Allocation {

    private Set<Region> regions;

    private Allocation(Set<Region> regions) {
        this.regions = new HashSet<>(regions);
    }
```

```
    public Allocation() {
        this.regions = new HashSet<>();
    }

    public Set<Region> getRegions() {
        return new HashSet<>(regions);
    }

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

    public Allocation withoutRegion(Region region) {
        if (!regions.contains(region)) {
            throw new IllegalArgumentException("Allocation doesn't contain region " + region);
        }
        Set<Region> newRegions = new HashSet<>(regions);
        newRegions.remove(region);
        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 (Region loc : regions) {
            total += loc.getCost();
        }
        return total;
    }

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

In particular, for this assignment, 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:

```
Region reg1 = new Region("Dusty Divot", 10, 500.0);
Region reg2 = new Region("The Nether", 0, 8.0)

Allocation example1 = new Allocation();
Allocation example2 = example1.withRegion(reg1);  // NOTE: this is a completely different Allocatio
                                                  //       from example1. In other words, they refe
                                                  //       two different Allocations

Allocation example3 = example1.withRegion(reg2);  // NOTE: example3 doesn't contain "Dusty Divot" a
                                                  //       its Regions; it only contains "The Nethe
```

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.

Required Methods

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

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

This method takes a budget and a list of `Region` objects as parameters. 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.

If `sites` is `null`, an `IllegalArgumentException` should be thrown.

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`.)

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 numLocs, int minPop, int maxPop, double minCostP
                                                 double maxCostPer)
```

- Creates a scenario with `numLocs` 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 (note that the diagrams do not show the process of choosing the "best" allocation).

▼ Expand

Example 1:

Input:

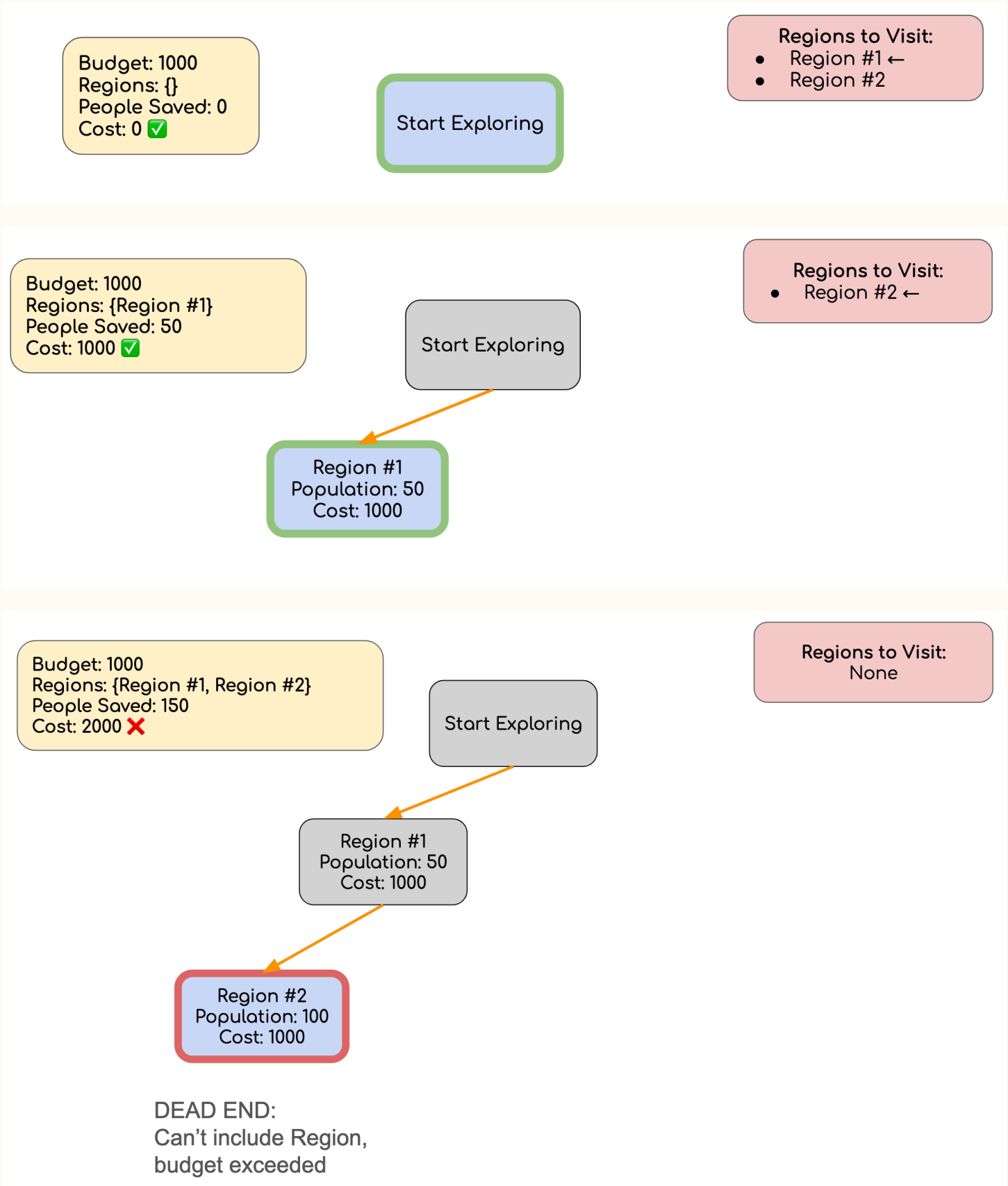
```
double budget = 1000;
```

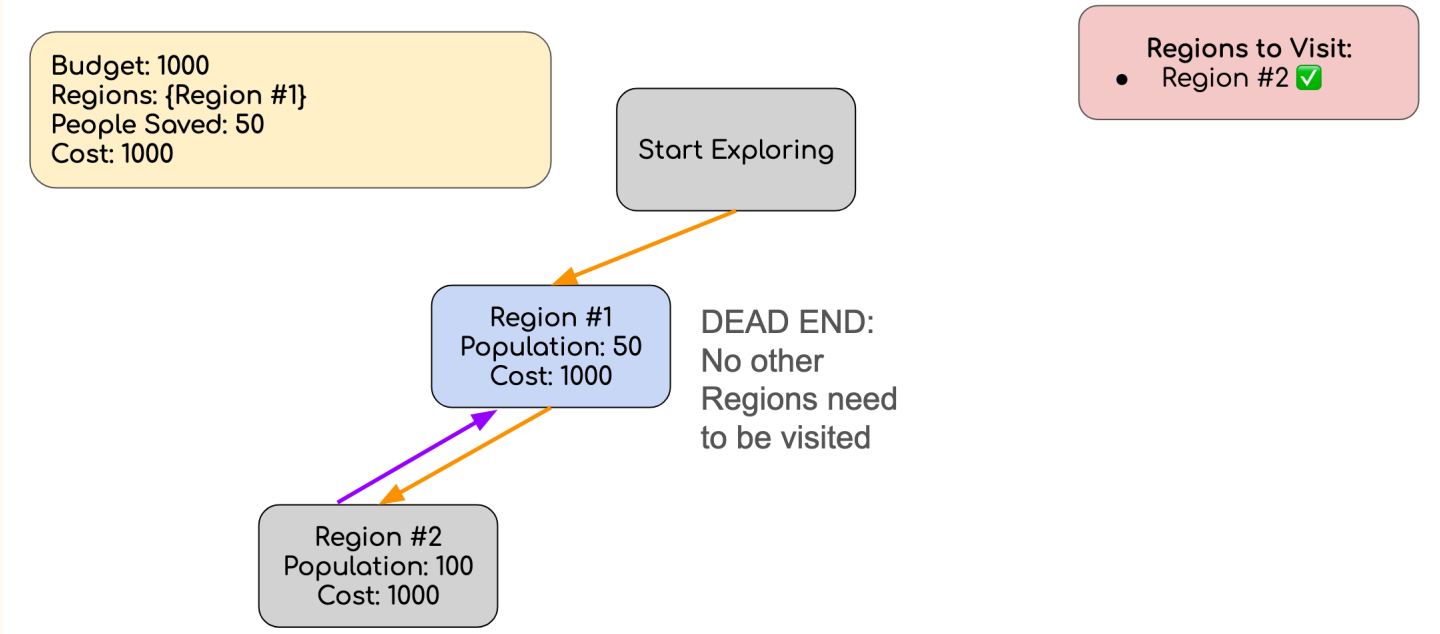
```
public static List<Region> createSimpleScenario() {
    List<Region> result = new ArrayList<>();
    result.add(new Region("Region #1", 50, 1000));
    result.add(new Region("Region #2", 100, 1000));
    return result;
}
```

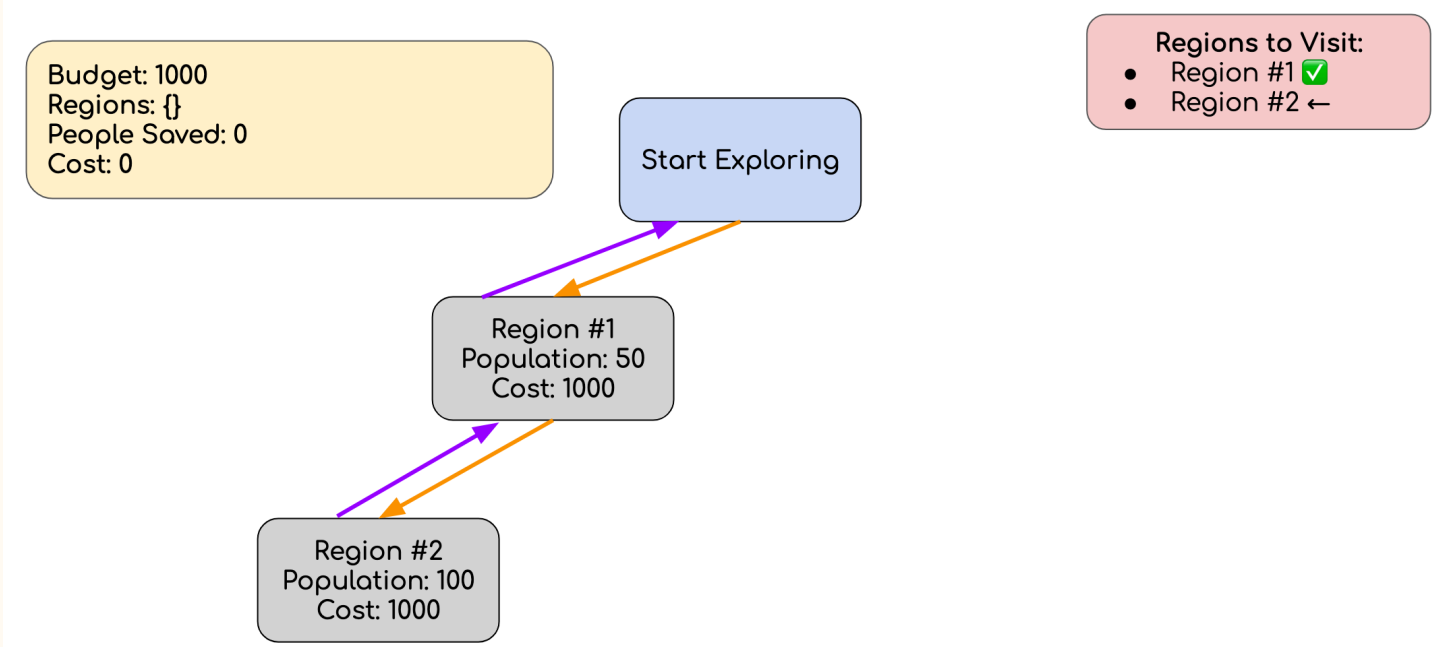
Output:

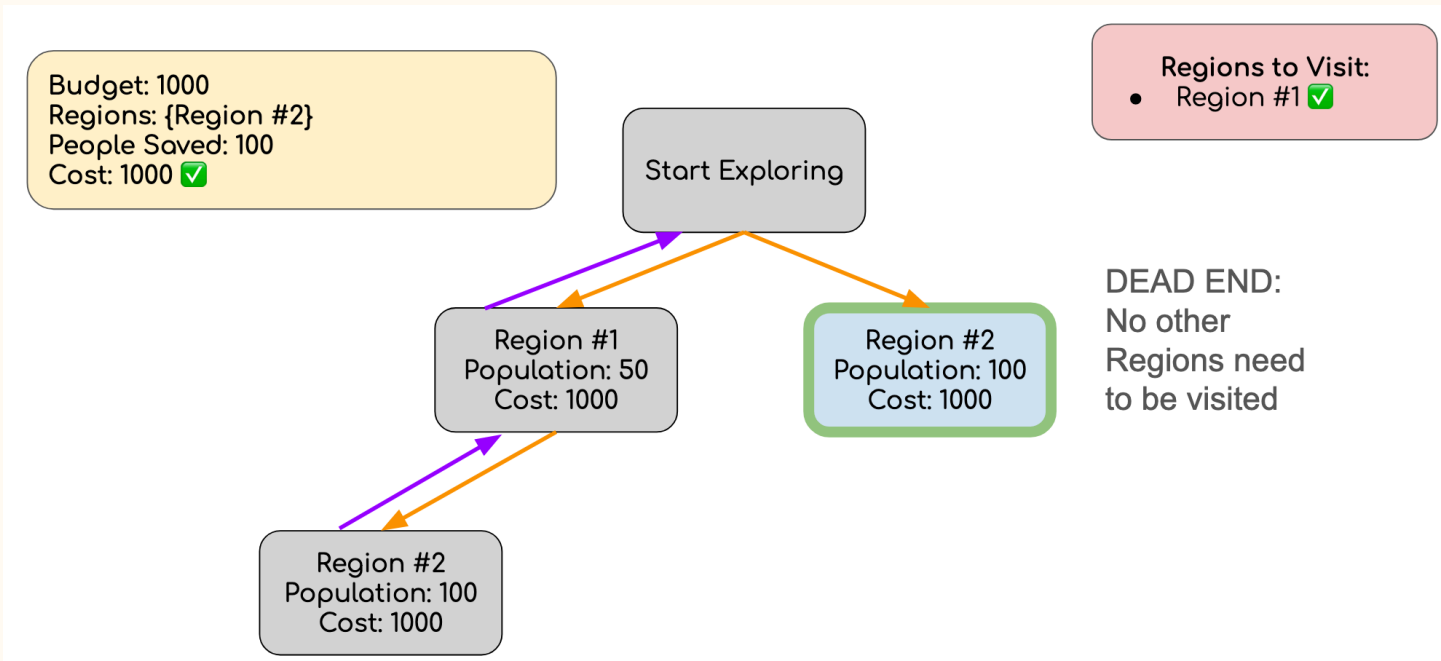
```
Result:
  [Region #2: pop. 100, cost: $1000.0]
  People saved: 100
  Cost: $1000.00
  Unused budget: $0.00
```

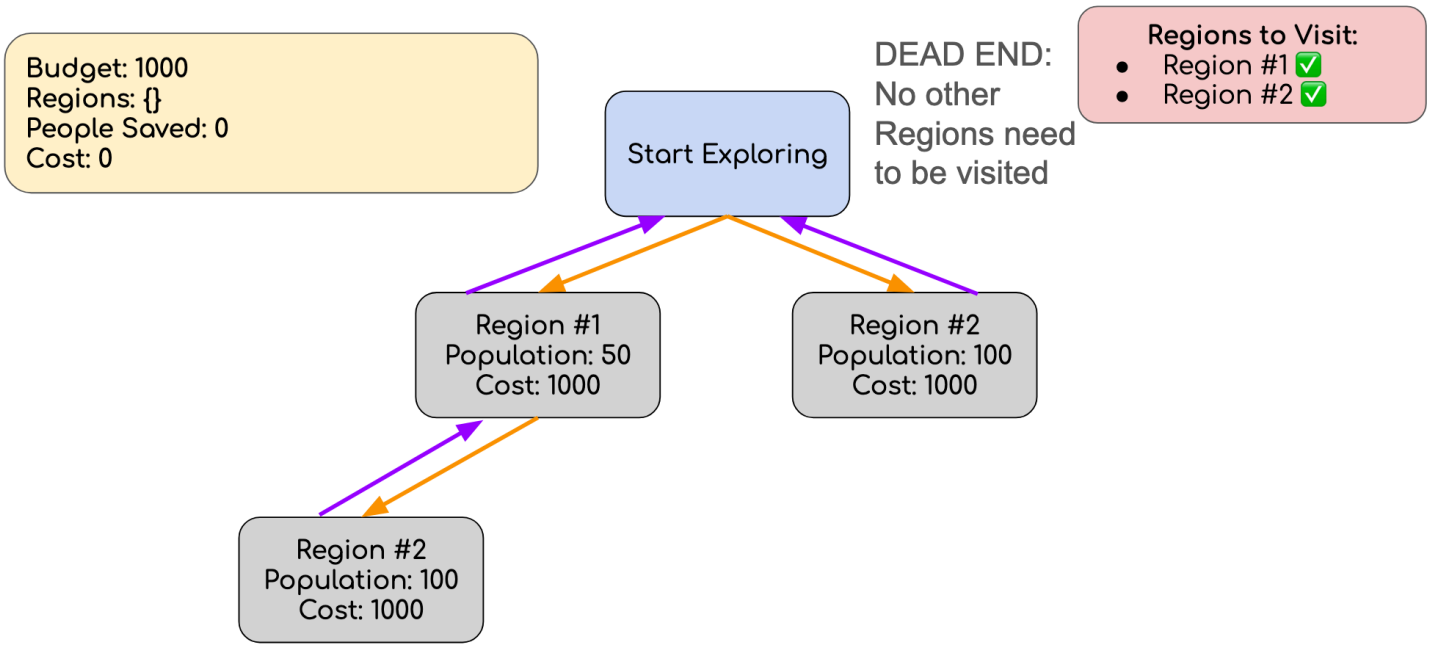
Decision Tree For Exploring All Regions:




Budget: 1000
Regions: {Region #1}
People Saved: 50
Cost: 1000
Start Exploring
Region #1
Population: 50
Cost: 1000
Region #2
Population: 100
Cost: 1000
DEAD END:
No other
Regions need
to be visited
Regions to Visit:
Region #2 ✅


Budget: 1000
Regions: {}
People Saved: 0
Cost: 0
Start Exploring
Region #1
Population: 50
Cost: 1000
Region #2
Population: 100
Cost: 1000
Regions to Visit:
Region #1 ✅
Region #2 ←


Budget: 1000
Regions: {Region #2}
People Saved: 100
Cost: 1000 ✅
Start Exploring
Region #1
Population: 50
Cost: 1000
Region #2
Population: 100
Cost: 1000
Region #2
Population: 100
Cost: 1000
DEAD END:
No other
Regions need
to be visited
Regions to Visit:
Region #1 ✅



▼ Expand

Example 2:

Input:

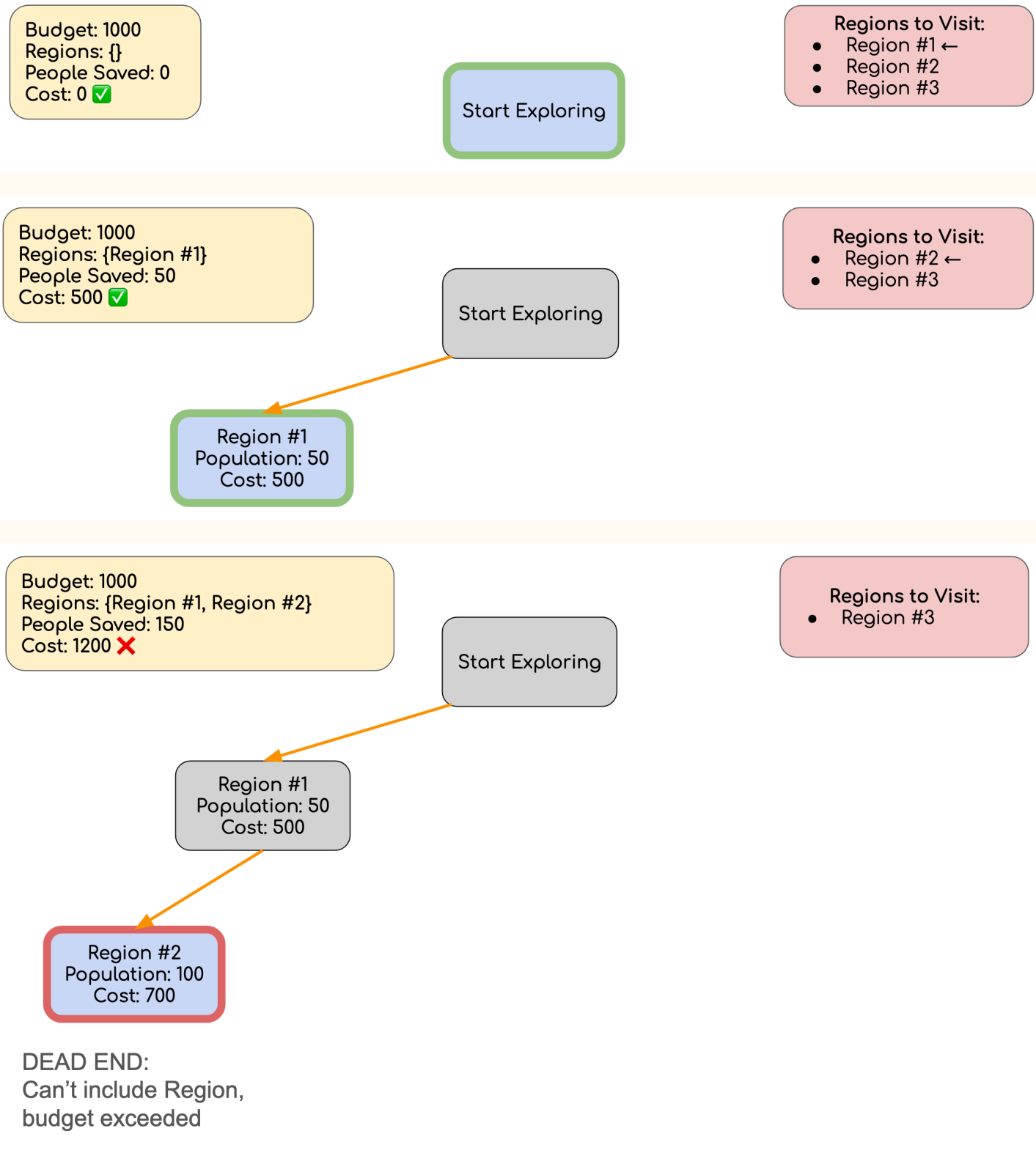
```
double budget = 1000;
```

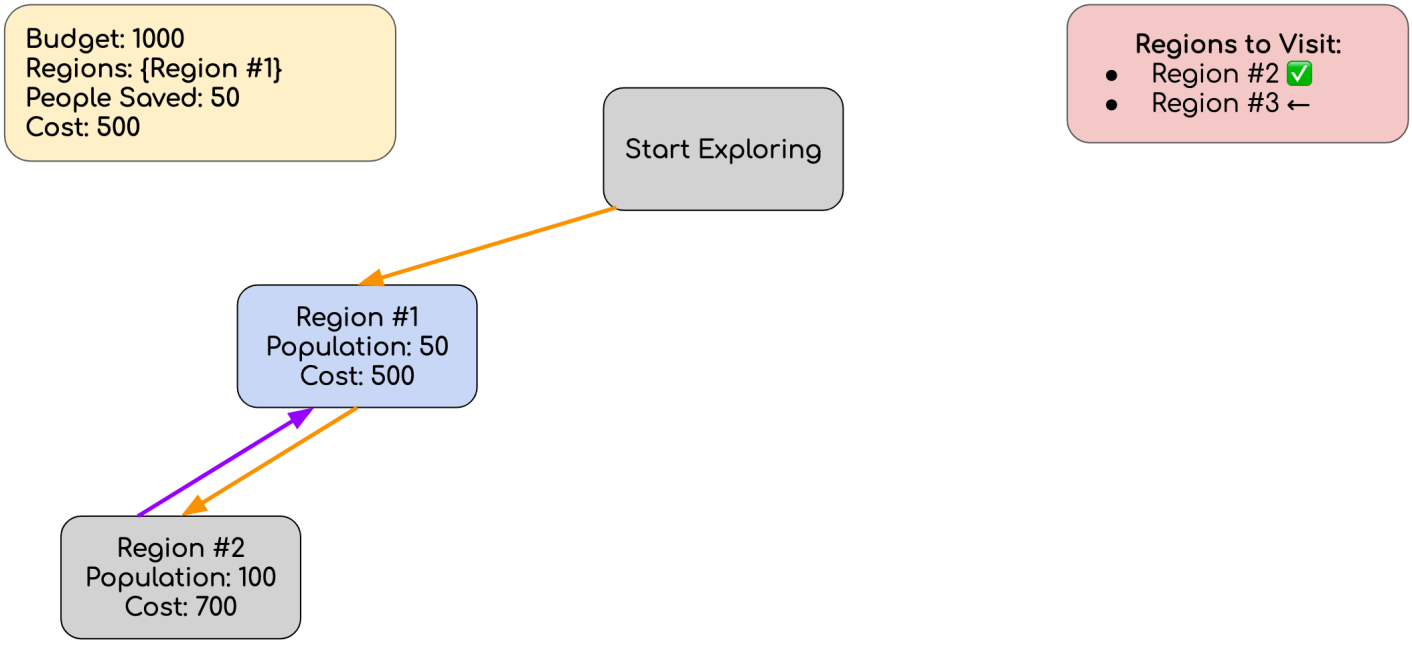
```
public static List<Region> createSimpleScenario() {
    List<Region> result = new ArrayList<>();
    result.add(new Region("Region #1", 50, 500));
    result.add(new Region("Region #2", 100, 700));
    result.add(new Region("Region #3", 60, 1000));
    return result;
}
```

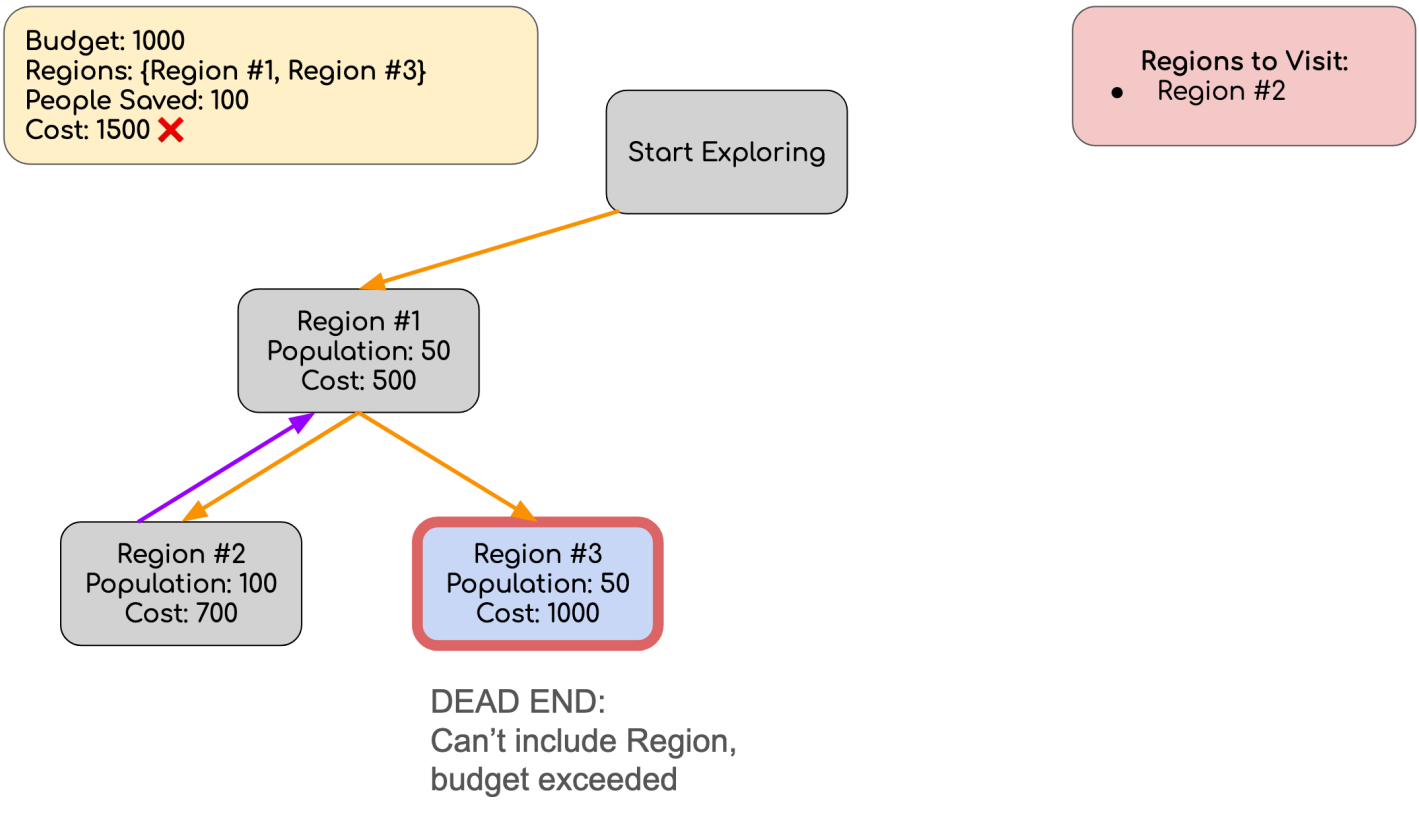
Output:

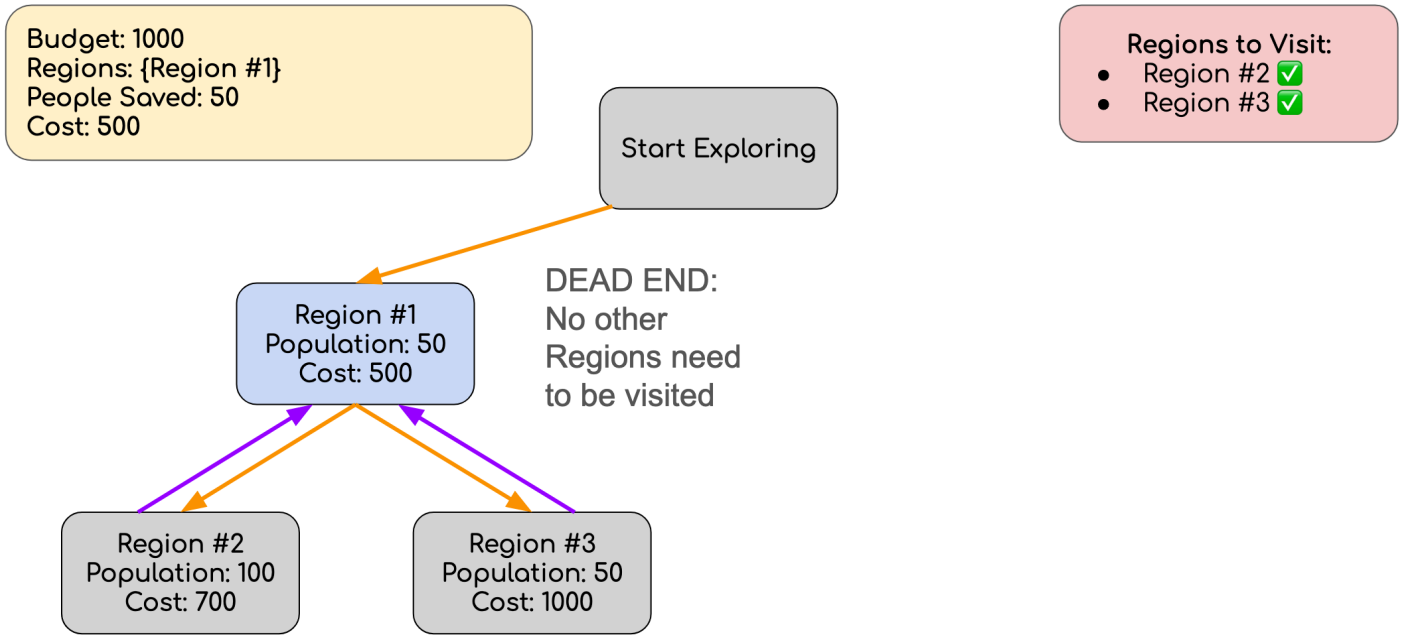
```
Result:
  [Region #2: pop. 100, cost: $700.0]
  People saved: 100
  Cost: $700.00
  Unused budget: $300.00
```

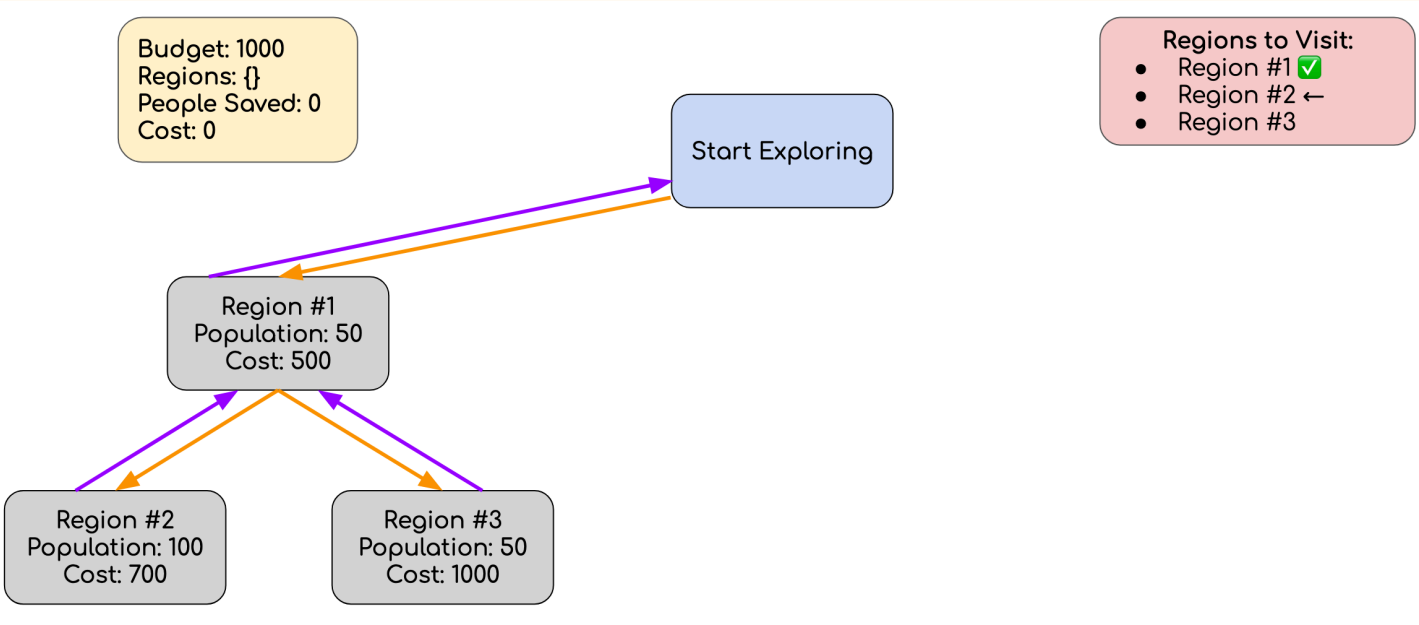
Decision Tree For Exploring All Regions:

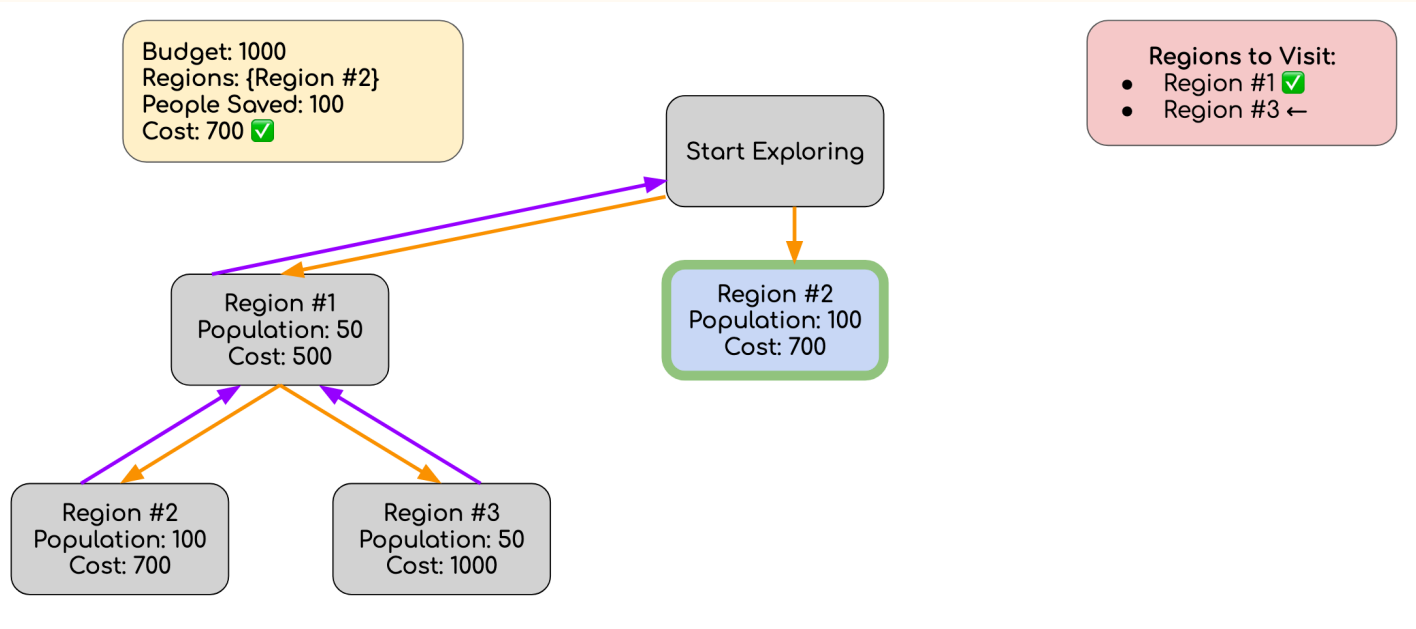

Budget: 1000
Regions: {}
People Saved: 0
Cost: 0 ✅
Start Exploring
Regions to Visit:
Region #1 ←
Region #2
Region #3
Budget: 1000
Regions: {Region #1}
People Saved: 50
Cost: 500 ✅
Start Exploring
Regions to Visit:
Region #2 ←
Region #3
Region #1
Population: 50
Cost: 500
Budget: 1000
Regions: {Region #1, Region #2}
People Saved: 150
Cost: 1200 ❌
Start Exploring
Regions to Visit:
Region #3
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
DEAD END:
Can't include Region,
budget exceeded

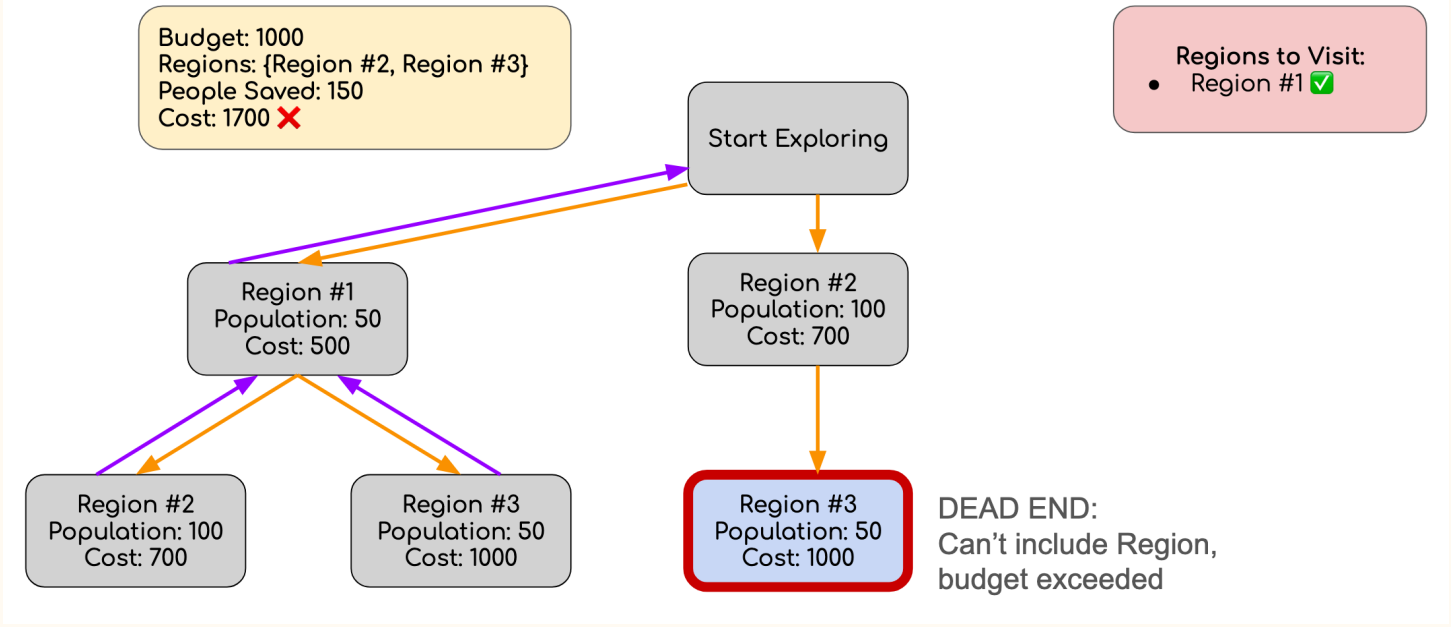

Budget: 1000
Regions: {Region #1}
People Saved: 50
Cost: 500
Start Exploring
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Regions to Visit:
Region #2 ✅
Region #3 ←

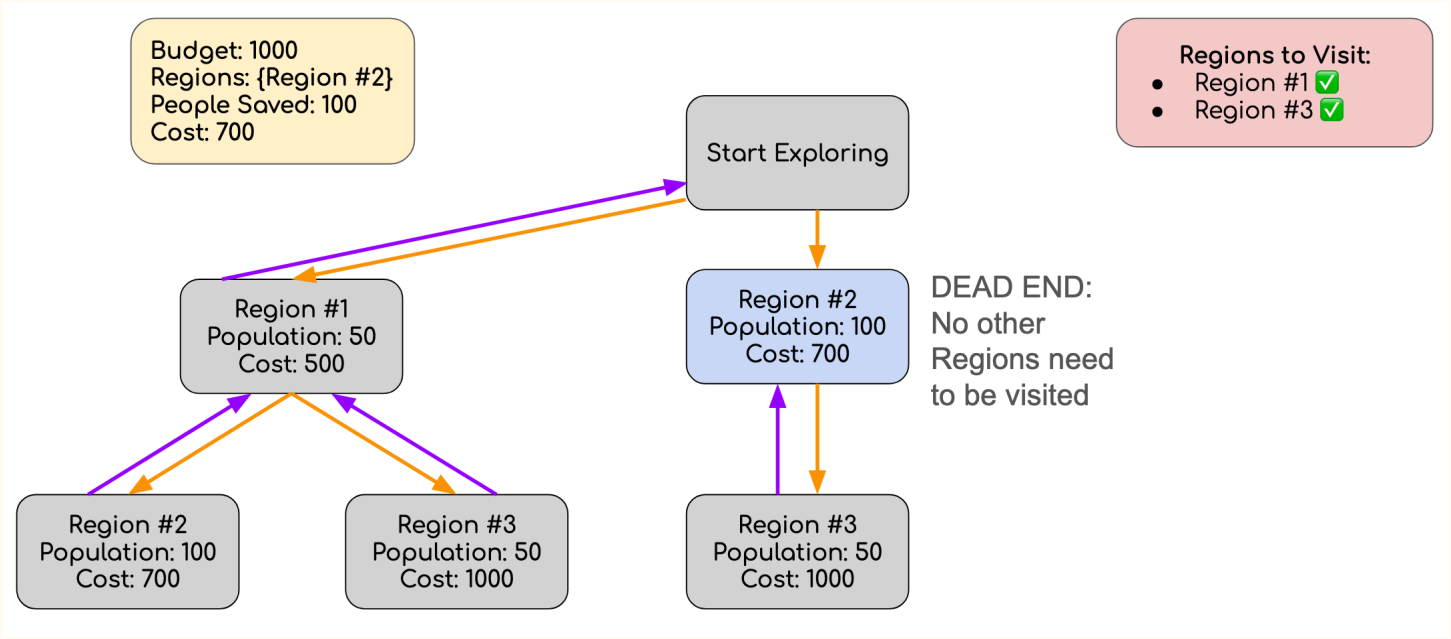

Budget: 1000
Regions: {Region #1, Region #3}
People Saved: 100
Cost: 1500 ❌
Start Exploring
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
DEAD END:
Can't include Region,
budget exceeded
Regions to Visit:
Region #2

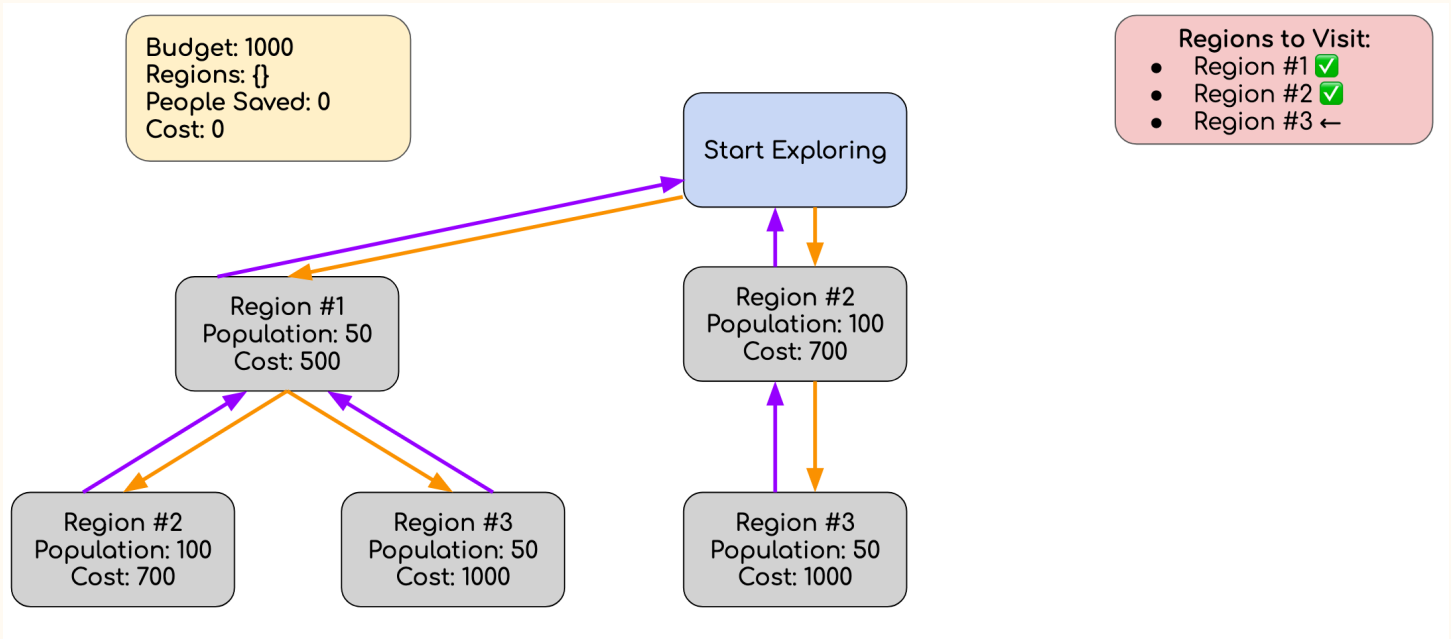

Budget: 1000
Regions: {Region #1}
People Saved: 50
Cost: 500
Regions to Visit:
Region #2 ✅
Region #3 ✅
Start Exploring
Region #1
Population: 50
Cost: 500
DEAD END:
No other
Regions need
to be visited
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000

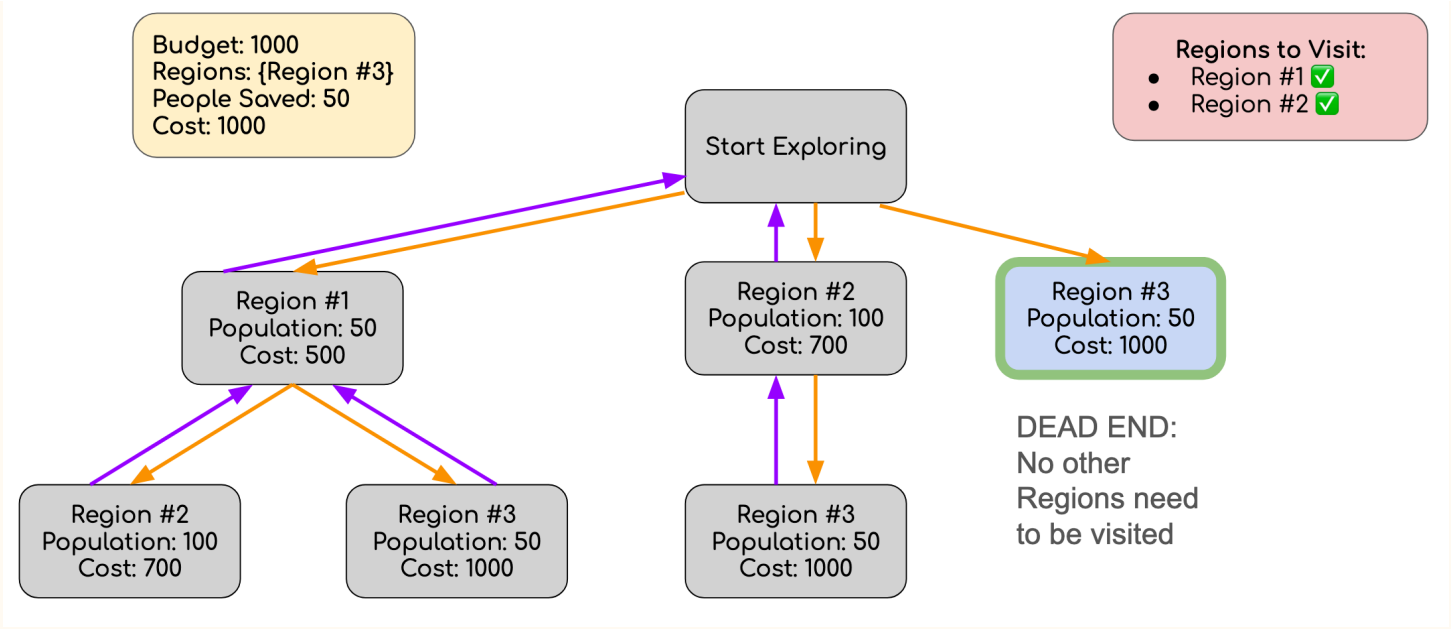

Budget: 1000
Regions: {}
People Saved: 0
Cost: 0
Regions to Visit:
Region #1 ✅
Region #2 ←
Region #3
Start Exploring
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000

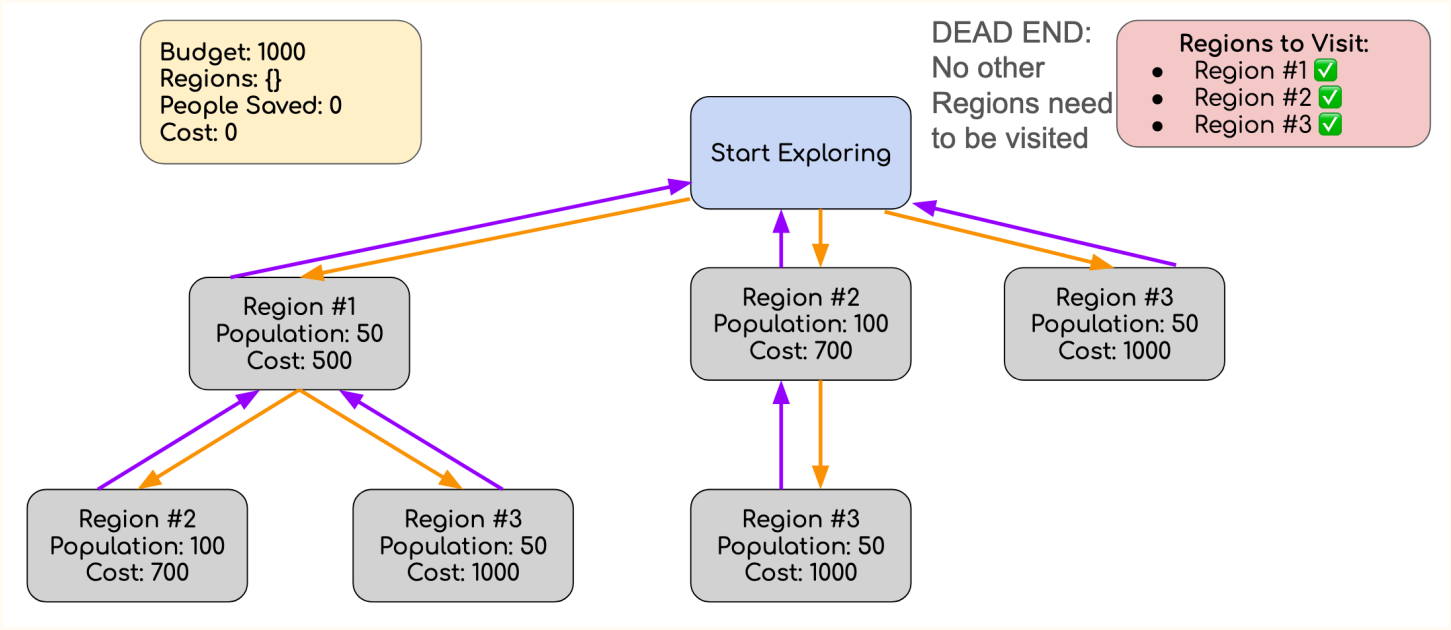

Budget: 1000
Regions: {Region #2}
People Saved: 100
Cost: 700 ✅
Regions to Visit:
Region #1 ✅
Region #3 ←
Start Exploring
Region #2
Population: 100
Cost: 700
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000


Budget: 1000
Regions: {Region #2, Region #3}
People Saved: 150
Cost: 1700 ❌
Regions to Visit:
• Region #1 ✅
Start Exploring
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
Region #3
Population: 50
Cost: 1000
DEAD END:
Can't include Region,
budget exceeded


Budget: 1000
Regions: {Region #2}
People Saved: 100
Cost: 700
Regions to Visit:
• Region #1 ✅
• Region #3 ✅
Start Exploring
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
DEAD END:
No other
Regions need
to be visited
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
Region #3
Population: 50
Cost: 1000


Budget: 1000
Regions: {}
People Saved: 0
Cost: 0
Regions to Visit:
• Region #1 ✅
• Region #2 ✅
• Region #3 ←
Start Exploring
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
Region #3
Population: 50
Cost: 1000





▼ Expand

Example 3:

Input:

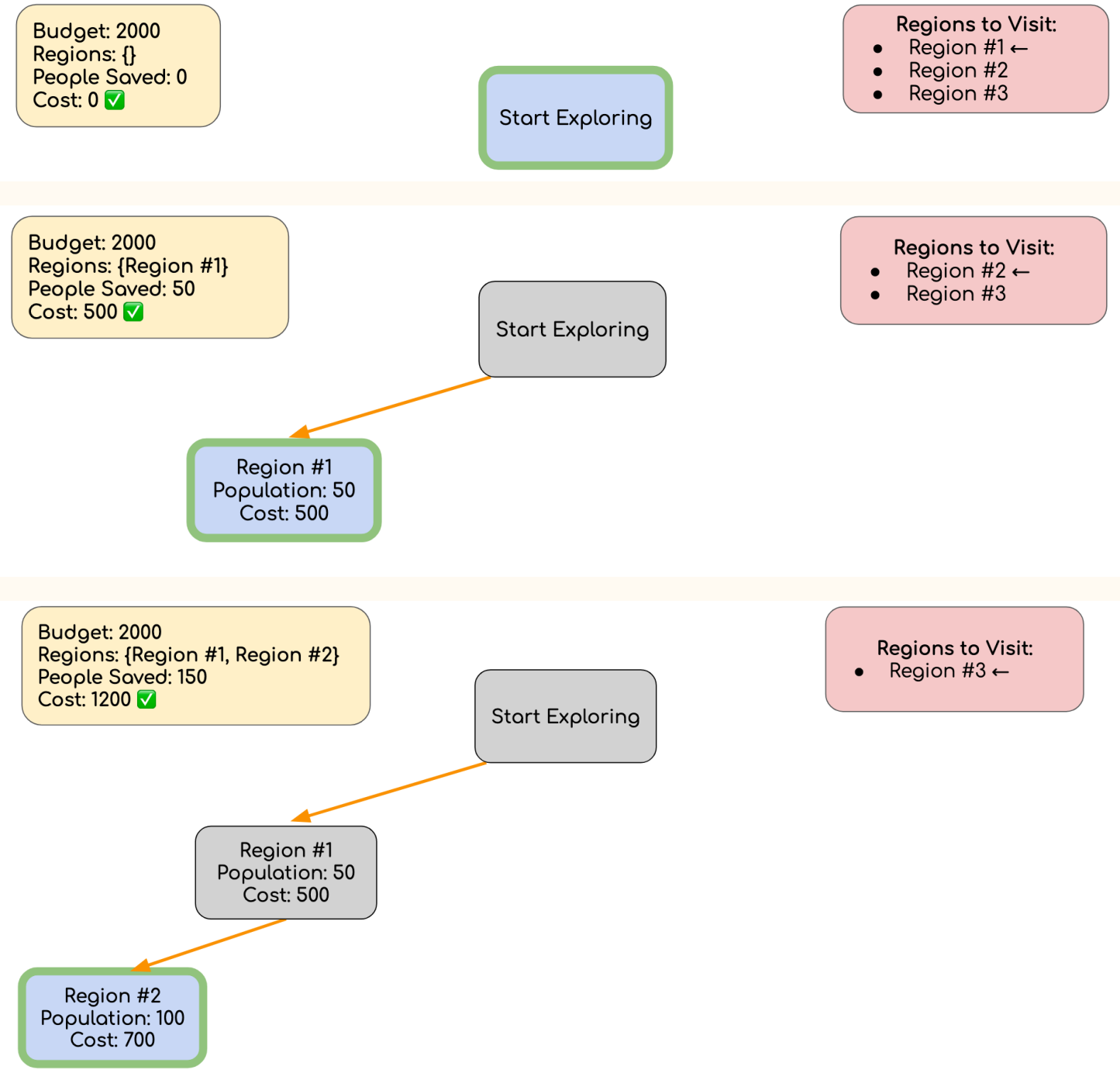
```
double budget = 2000;
```

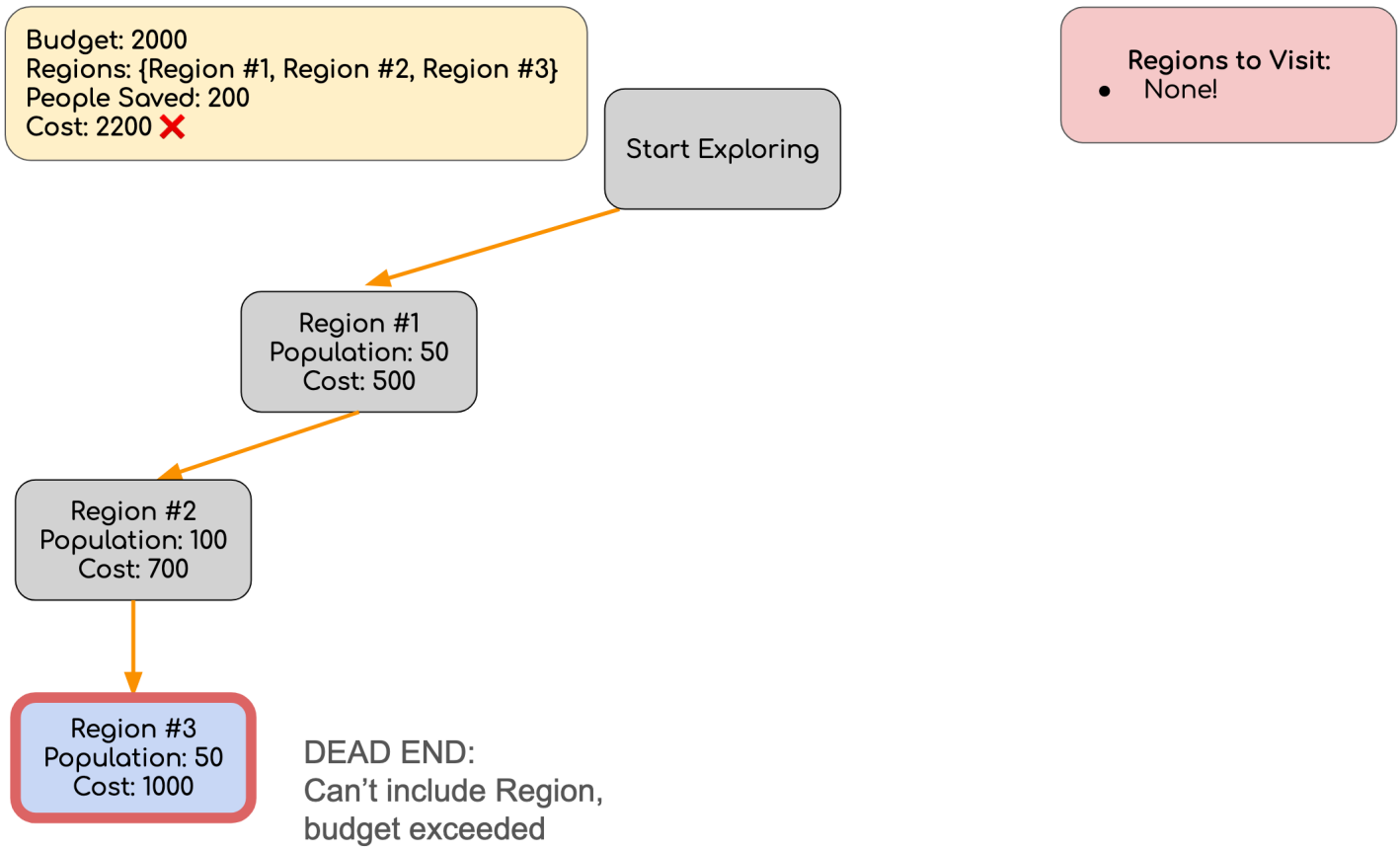
```
public static List<Region> createSimpleScenario() {
    // Sample Region as Example 2 but Region 3 has a population of 50
    List<Region> result = new ArrayList<>();
    result.add(new Region("Region #1", 50, 500));
    result.add(new Region("Region #2", 100, 700));
    result.add(new Region("Region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

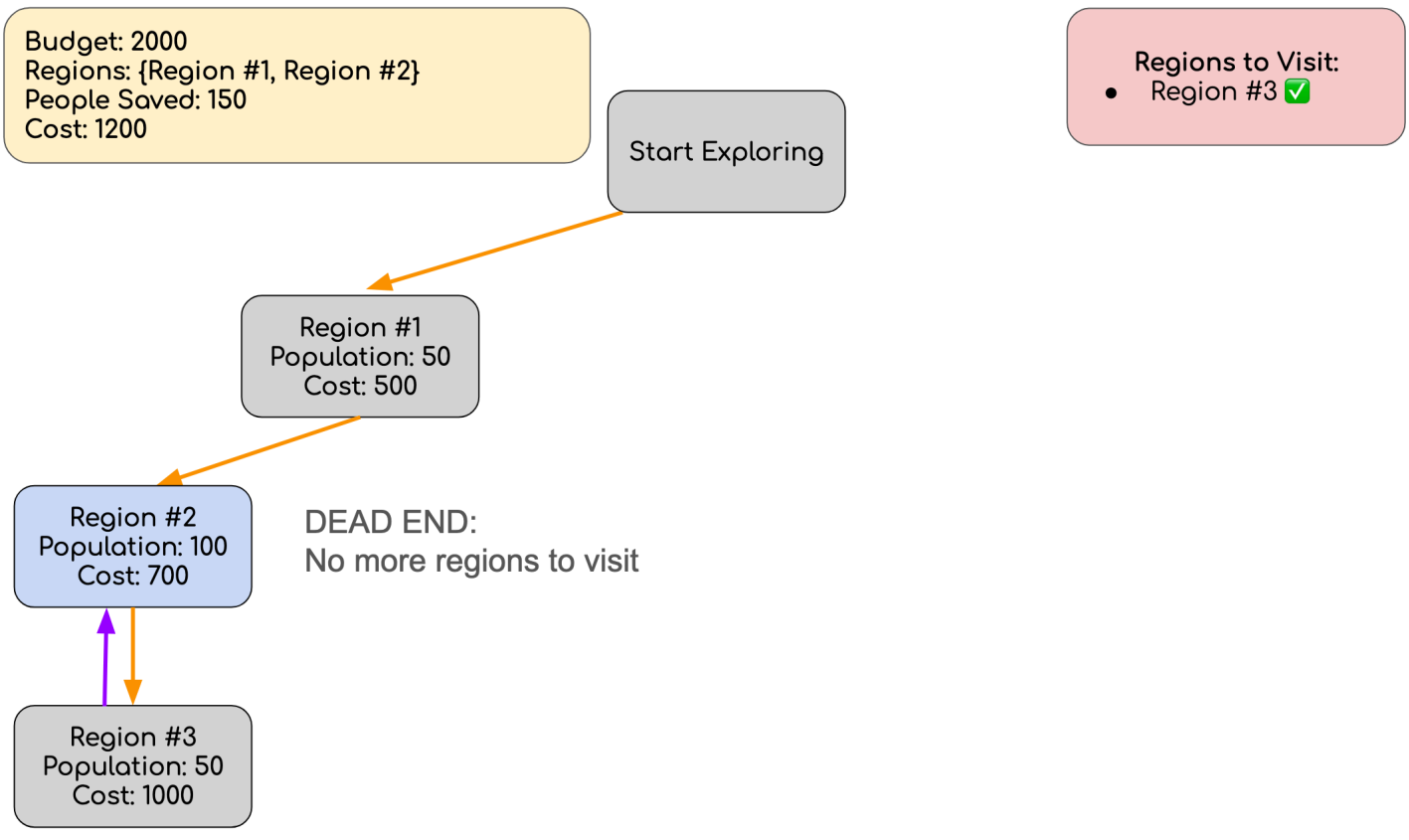
Output:

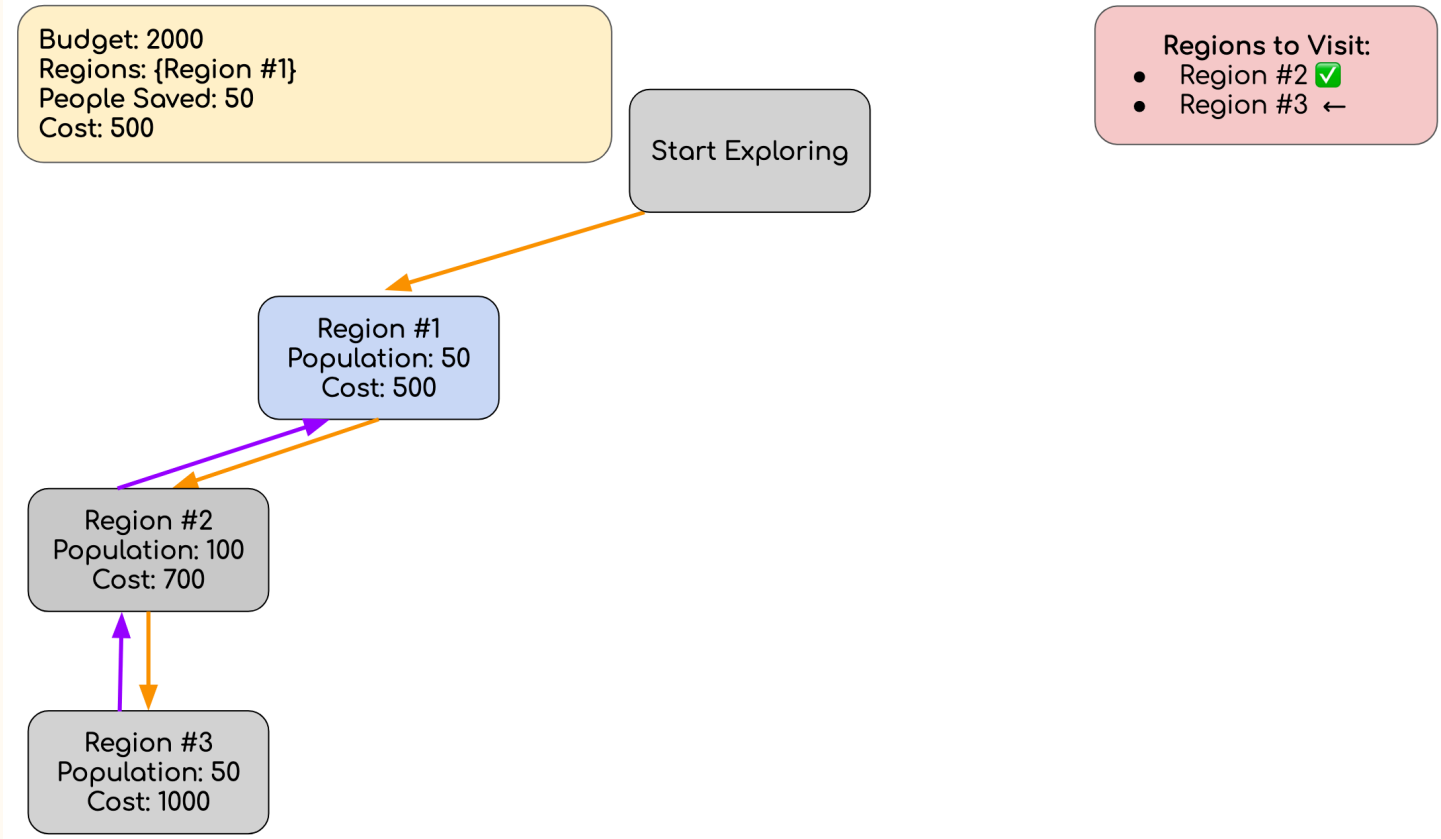
```
Result:
  [Region #2: pop. 100, cost: $700.0, Region #1: pop. 50, cost: $500.0]
  People saved: 150
  Cost: $1200.00
  Unused budget: $800.00
```

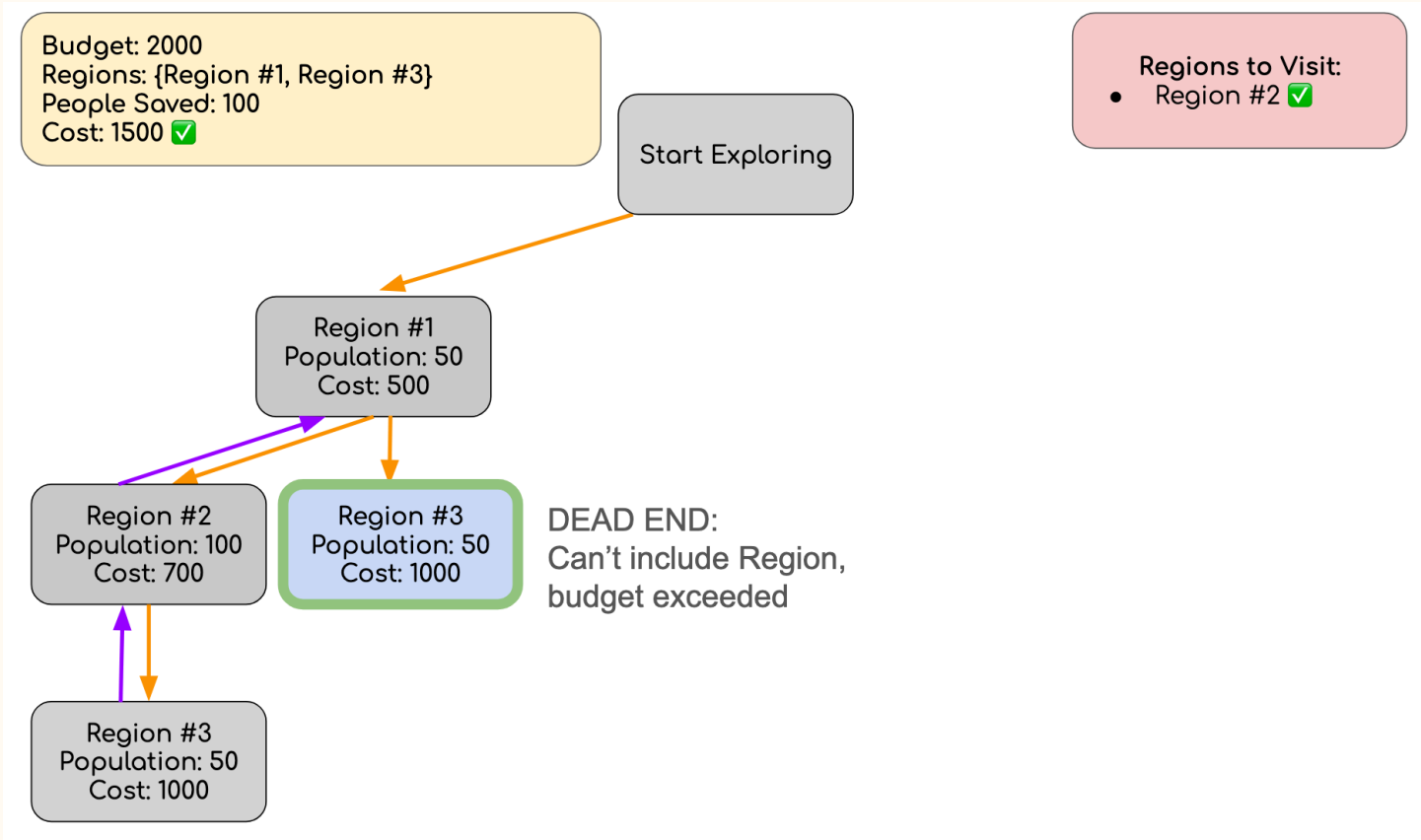
Decision Tree For Exploring All Regions:

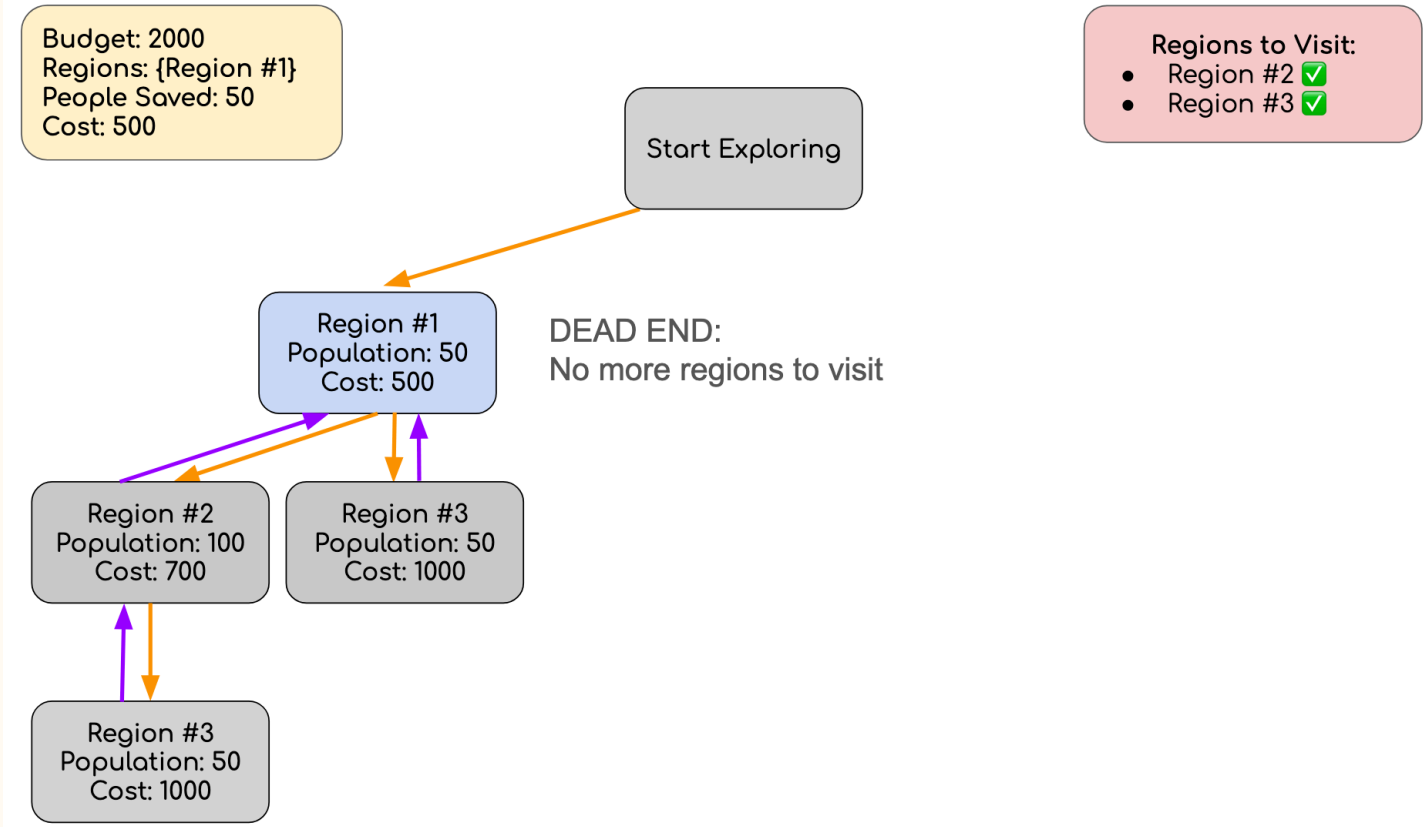


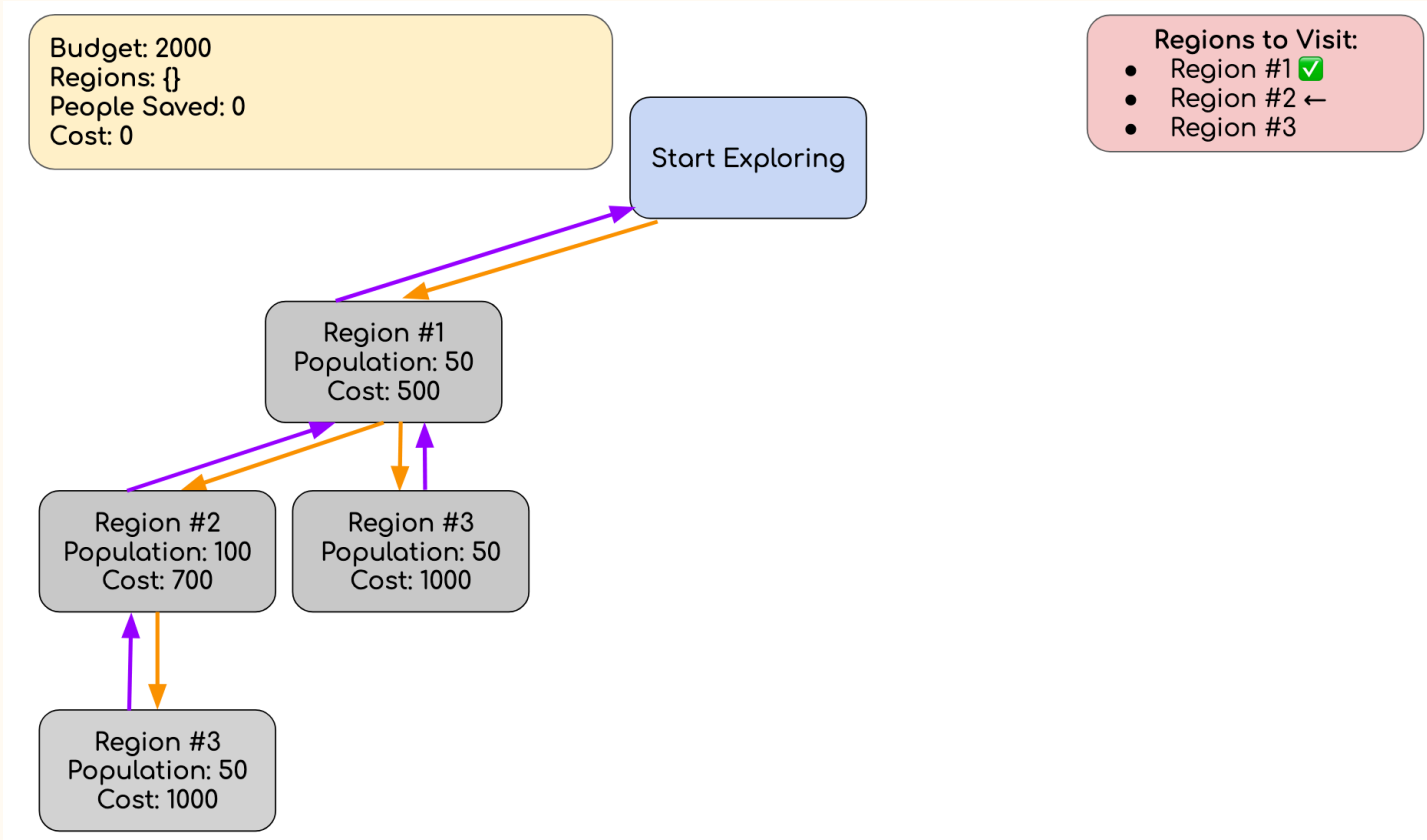

Budget: 2000
Regions: {Region #1, Region #2, Region #3}
People Saved: 200
Cost: 2200 ❌
Start Exploring
Regions to Visit:
None!
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
DEAD END:
Can't include Region,
budget exceeded

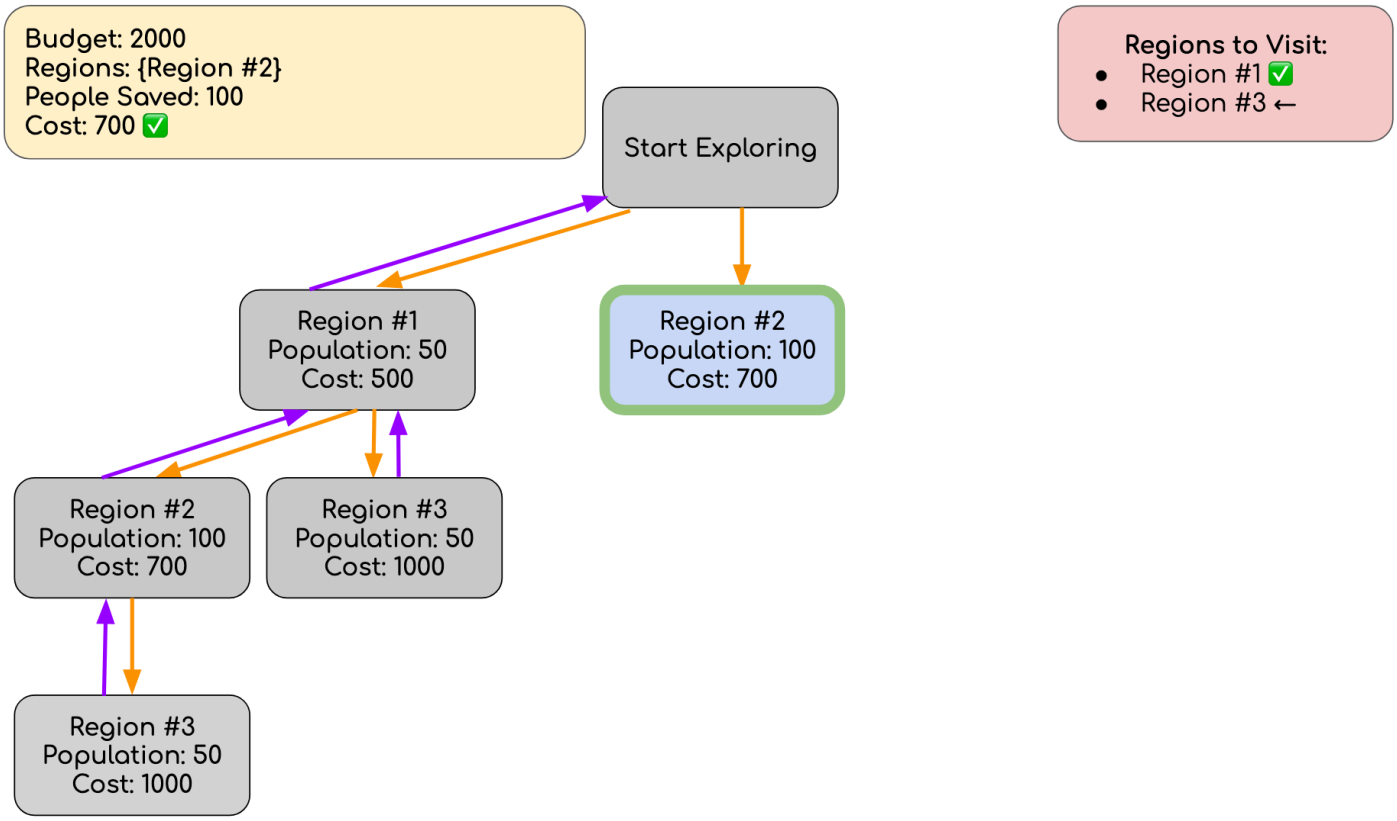

Budget: 2000
Regions: {Region #1, Region #2}
People Saved: 150
Cost: 1200
Start Exploring
Regions to Visit:
Region #3 ✅
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
DEAD END:
No more regions to visit
Region #3
Population: 50
Cost: 1000

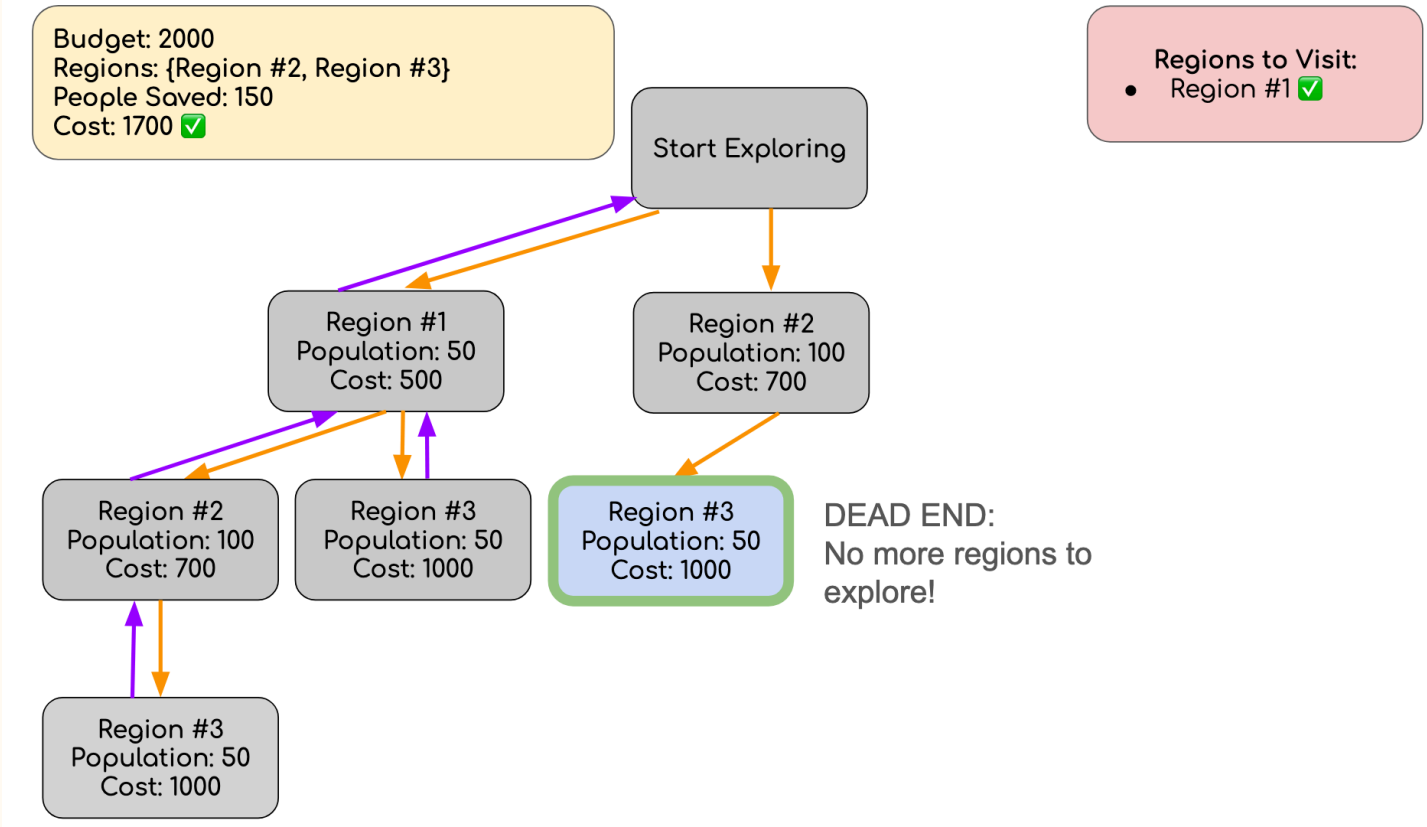

Budget: 2000
Regions: {Region #1}
People Saved: 50
Cost: 500
Start Exploring
Regions to Visit:
Region #2 ✅
Region #3 ←
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000

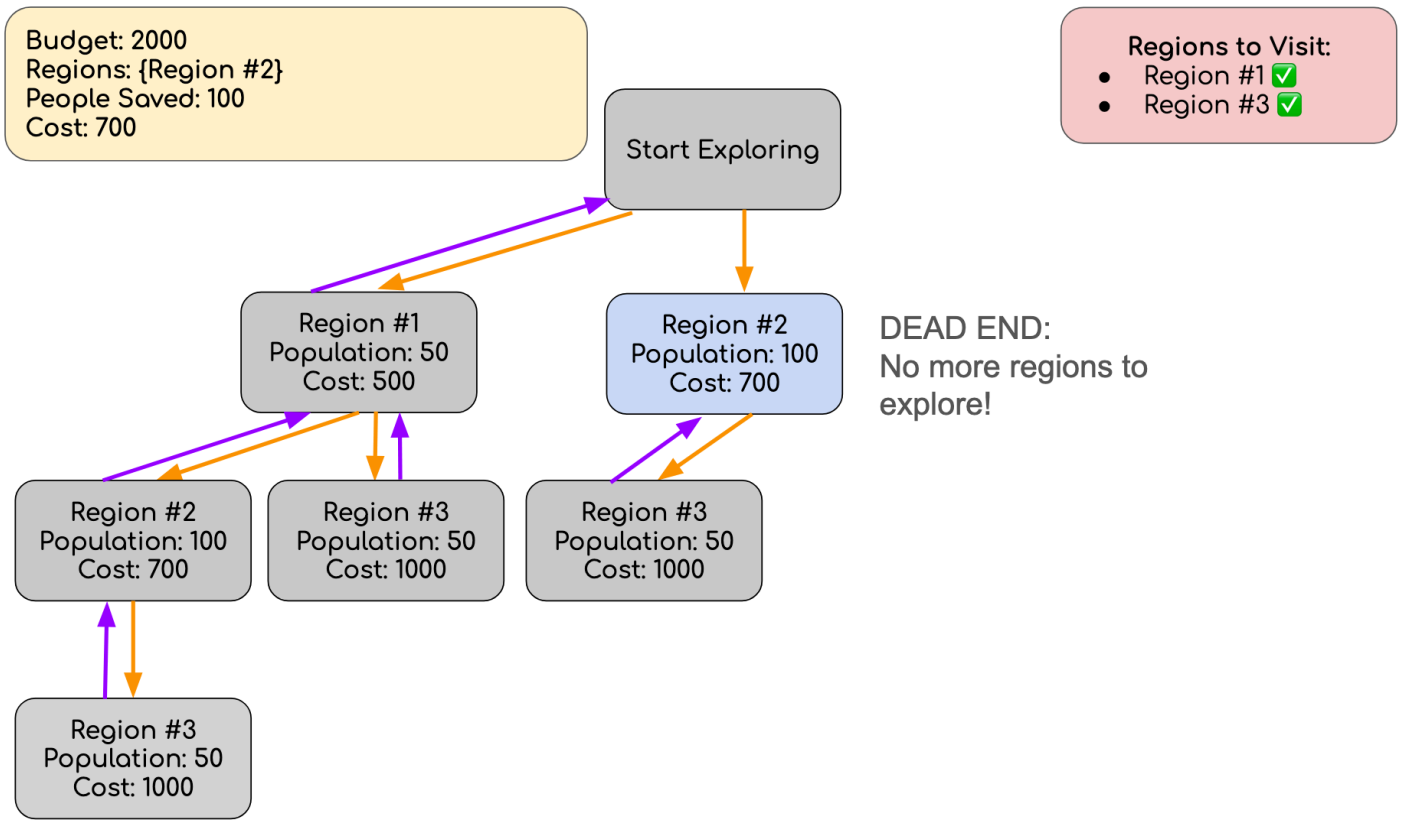

Budget: 2000
Regions: {Region #1, Region #3}
People Saved: 100
Cost: 1500 ✅
Start Exploring
Regions to Visit:
Region #2 ✅
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
DEAD END:
Can't include Region,
budget exceeded
Region #3
Population: 50
Cost: 1000

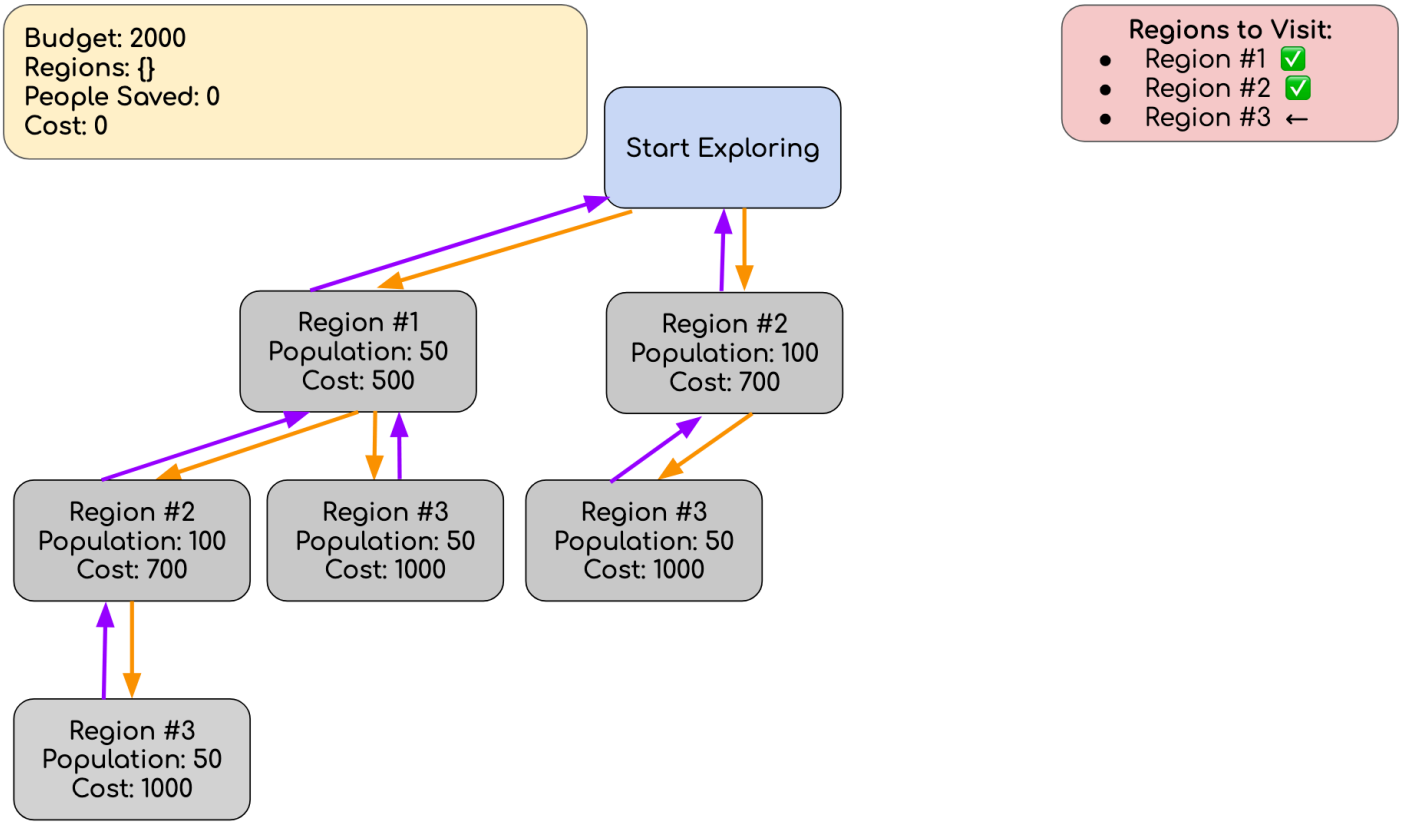

Budget: 2000
Regions: {Region #1}
People Saved: 50
Cost: 500
Regions to Visit:
Region #2 ✅
Region #3 ✅
Start Exploring
Region #1
Population: 50
Cost: 500
DEAD END:
No more regions to visit
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
Region #3
Population: 50
Cost: 1000

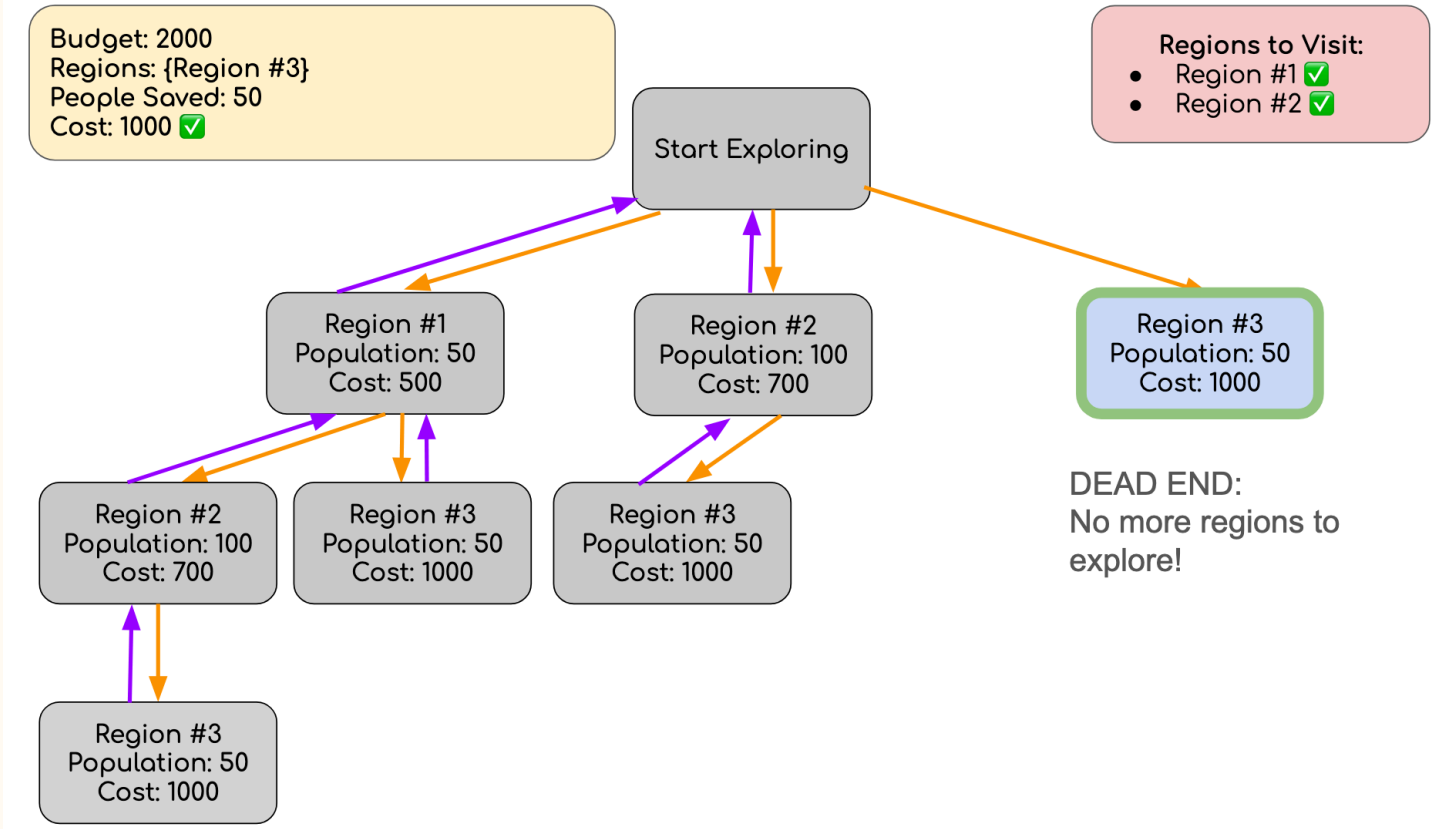

Budget: 2000
Regions: {}
People Saved: 0
Cost: 0
Regions to Visit:
Region #1 ✅
Region #2 ←
Region #3
Start Exploring
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
Region #3
Population: 50
Cost: 1000

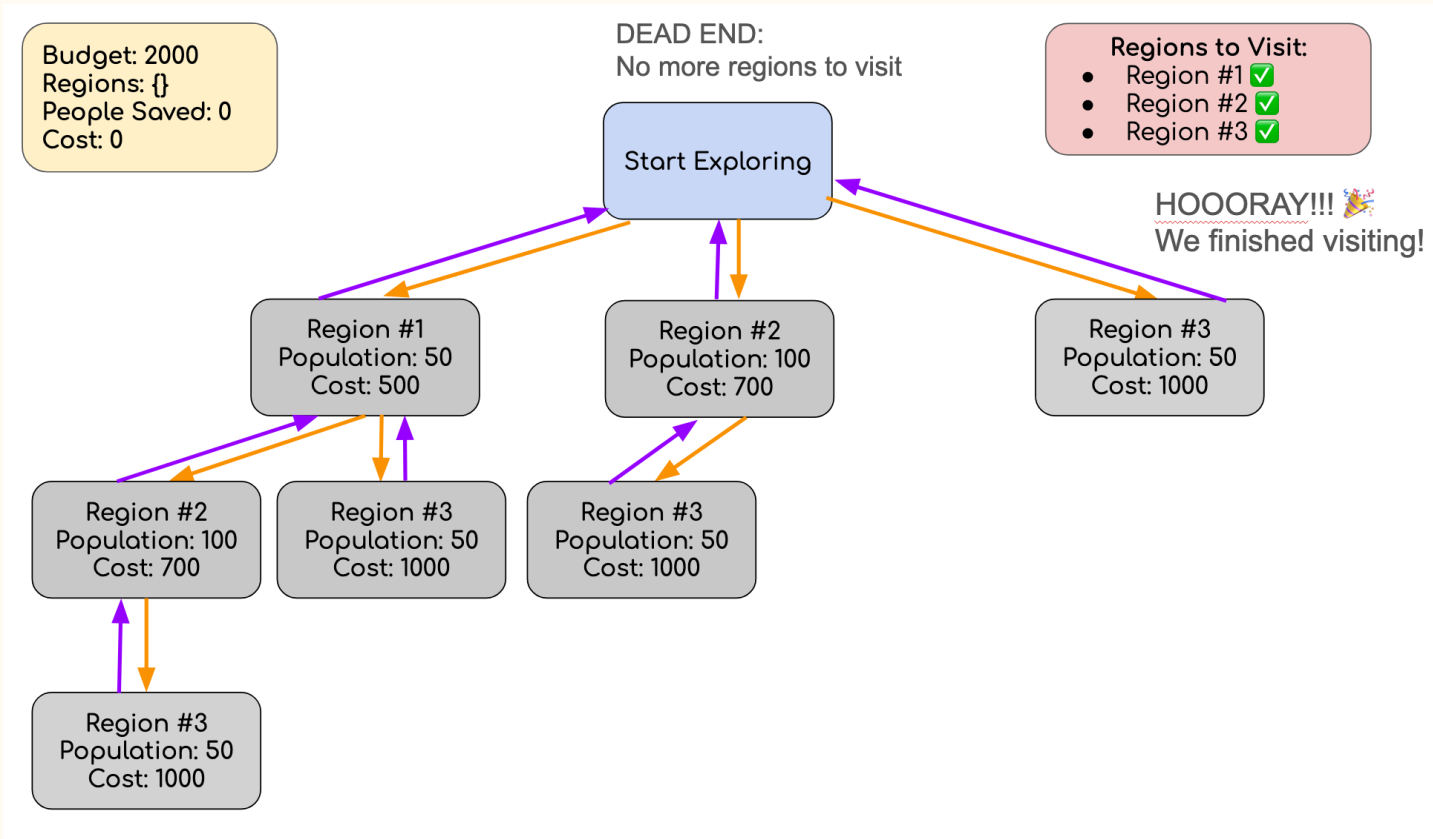

Budget: 2000
Regions: {Region #2}
People Saved: 100
Cost: 700 ✅
Regions to Visit:
Region #1 ✅
Region #3 ←
Start Exploring
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
Region #3
Population: 50
Cost: 1000


Budget: 2000
Regions: {Region #2, Region #3}
People Saved: 150
Cost: 1700 ✅
Regions to Visit:
Region #1 ✅
Start Exploring
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
Region #3
Population: 50
Cost: 1000
DEAD END:
No more regions to explore!
Region #3
Population: 50
Cost: 1000


Budget: 2000
Regions: {Region #2}
People Saved: 100
Cost: 700
Regions to Visit:
Region #1 ✅
Region #3 ✅
Start Exploring
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
DEAD END:
No more regions to explore!
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
Region #3
Population: 50
Cost: 1000
Region #3
Population: 50
Cost: 1000


Budget: 2000
Regions: {}
People Saved: 0
Cost: 0
Regions to Visit:
Region #1 ✅
Region #2 ✅
Region #3 ←
Start Exploring
Region #1
Population: 50
Cost: 500
Region #2
Population: 100
Cost: 700
Region #2
Population: 100
Cost: 700
Region #3
Population: 50
Cost: 1000
Region #3
Population: 50
Cost: 1000
Region #3
Population: 50
Cost: 1000





NOTE: The ordering of elements in your set does not matter. For example, a set containing `{Region #1: pop. 100, cost: $1000.0, Region #2: pop. 50, cost: $200.0}` and a set containing `{Region #2: pop. 50, cost: $200.0, Region #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 files without modification and must meet all requirements below**.

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.

The Scrabble Helper example from Lesson 11 will be helpful to you in completing this assignment.

Testing Requirements

TIP: If you want to call `allocateRelief` from outside `Client.java`, you can do `Client.allocateRelief`.

For this assignment, you'll be required to implement **three** total JUnit tests. Our requirements for each test are as follows:

- Test 1: A budget where you can allocate relief to all regions in sites.
- Test 2: A budget where you can allocate relief to none of the regions in sites.
- Test 3: A budget where you can allocate relief to some, but not all, of the regions in sites .
    - This one should contain at least 4 regions in sites, and at least 2 should be in the final allocation

All three 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.

For this assignment, we expect you to put effort into being thorough when writing tests. You should be using `assertEquals` with an expected `Set<Region>` or `Allocation` to compare your result.

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 different (redundant) orderings of regions. Since the `Allocation` class uses a `Set`, it does not make a distinction between adding `Region`s A, B, C versus B, C, A, so there will not be a difference in cost or results. If you'd like, you can design your solution to

avoid considering both of these orderings (considering only one of them). Both solutions that consider redundant orderings and solutions that avoid redundant orderings will be accepted. When making your decision, strive for simplicity above all else — do not add extra convoluted code to try to avoid or consider redundant orderings.

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 if 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.
    - 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.