Ryan Treves CS 106B Final Project Autumn 2020

Problem Description:

In the U.S., Independent System Operators (ISOs) help balance supply of electricity from power plants (or 'generators') with demand for electricity from utilities that serve customers. This is a challenging task, as demand fluctuates both from year to year and from hour to hour. At the same time, deciding which generators to use involves weighing economic and environmental values of diverse stakeholders.

In this problem, we need to design a function that will assist an ISO in determining which power generators to bring online. Each generator can be represented as a struct with three variables:

```
struct Generator {
    int capacity;
    int emissions;
    int cost;
};
```

Where `capacity` is the power output of the generator, `emissions` are the number of greenhouse gas emissions associated with running the generator, and `cost` is the cost of running the generator.

Your task is to create a function

```
Vector<Generator> buildPortfolio(Vector<Generator> generators, string optimize, int demand)
```

that, given an unsorted vector of generators, an optimization variable, and a level of demand, uses **recursive backtracking** to return a vector containing the subset of generators for the ISO to bring online such that:

a) Demand is met or exceeded
b) The optimization variable is minimized

The optimization variable can be either "`budget`", in which case **`buildPortfolio`** should return the cheapest combination of generators possible to meet demand; or "`emissions`", in which case it should return the combination of generators with the lowest total emissions.

You may assume that the total available generation capacity will always exceed demand, and demand will always be nonnegative. You may use one or more helper functions.

Solution A Code:

```
void portfolioHelper(Vector<Generator>& generators, int demand, string optimize, int current, int& best, Vector<Generator>& currentPortfolio, Vector<Generator>& bestPortfolio) {
    // Base cases
    if (demand <= 0 || generators.isEmpty()) {
        if (current != 0 && (best == -1 || best > current)) {
            best = current;
            bestPortfolio = currentPortfolio;
```

```
        }
        return;
    }

    // Examine the first generator
    Generator gen = generators.remove(0);

    // Recurse through remaining generators without adding it
    portfolioHelper(generators, demand, optimize, current, best, currentPortfolio,
bestPortfolio);

    // Add either cost or emissions to current total
    if (optimize == "budget") {
        current += gen.cost;
    }
    else {
        current += gen.emissions;
    }
    currentPortfolio.add(gen);

    // Recurse through remaining generators with adding it
    portfolioHelper(generators, demand - gen.capacity, optimize, current, best,
currentPortfolio, bestPortfolio);
    currentPortfolio.remove(currentPortfolio.size() - 1);
    generators.add(gen);
}

Vector<Generator> buildPortfolio(Vector<Generator> generators, string optimize, int
demand) {
    int best = -1;
    Vector<Generator> currentPortfolio = {};
    Vector<Generator> bestPortfolio = {};
    portfolioHelper(generators, demand, optimize, 0, best, currentPortfolio,
bestPortfolio);
    return bestPortfolio;
}
```

Solution A has $O(2^n)$.

Solution B Code: *Solution B is identical, with the exception of those lines highlighted in yellow*

```
void portfolioHelperPrunes(Vector<Generator>& generators, int demand, string
optimize, int current, int& best, Vector<Generator>& currentPortfolio,
Vector<Generator>& bestPortfolio) {
    // Pruning
    if (best != -1 && current > best) {
        return;
    }

    // Base cases
    if (demand <= 0 || generators.isEmpty()) {
        if (current != 0 && (best == -1 || best > current)) {
```

```
            best = current;
            bestPortfolio = currentPortfolio;
        }
        return;
    }

    // Pull off the last generator
    Generator gen = generators.remove(generators.size()-1);

    // Recurse through remaining generators without adding it
    portfolioHelper(generators, demand, optimize, current, best, currentPortfolio,
bestPortfolio);

    // Add either cost or emissions to current total
    if (optimize == "budget") {
        current += gen.cost;
    }
    else {
        current += gen.emissions;
    }
    currentPortfolio.add(gen);

    // Recurse through remaining generators with adding it
    portfolioHelper(generators, demand - gen.capacity, optimize, current, best,
currentPortfolio, bestPortfolio);
    currentPortfolio.remove(currentPortfolio.size() - 1);
    generators.add(gen);
}

Vector<Generator> buildPortfolioPrunes(Vector<Generator> generators, string
optimize, int demand) {
    int best = -1;
    Vector<Generator> currentPortfolio = {};
    Vector<Generator> bestPortfolio = {};
    portfolioHelperPrunes(generators, demand, optimize, 0, best, currentPortfolio,
bestPortfolio);
    return bestPortfolio;
}
```

Solution B has $O(2^n)$ at worst but O(n) at best.

Solution A vs. B

There are a number of approaches to solving this problem: some students might construct two helper functions (one for `optimize = "budget"`, one for `optimize = "emissions"`), some might build a non-void helper function that itself returns the best portfolio, others might use more arms-length recursion. Here, we highlight a typical Solution A, and a similar but higher performance Solution B. In contrast with Solution A, which processes the vector of generators from front to back, Solution B processes back to front: this yields efficiency gains due to the O(1) performance of removing the last element of a vector, as opposed to O(n) for removing the first element of a vector. Furthermore, Solution B uses pruning to truncate any decision

trees which prematurely exceed the existing optimal solution, meaning that overall performance may be significantly better than $O(2^n)$ depending on the input vector. Overall, Solution A is sufficient for very small scale cases, only Solution B can efficiently handle cases of ~20-25 generators, and neither solution is viable for cases any larger. An extension to this problem could involve coding an alternative solution capable of handling 100s or even 1000s of generators using a `PriorityQueue`.

Test cases: The following test cases were selected:

- 0 generators available, demand = 0
- 1 generator available, demand = 0
- Two generators available (small case)
- Five generators available (larger case)

Separate time trials were conducted to test performance. Adding time trials to a pre-prepared test suite could help guide students towards higher-performance solutions.

Motivation

This problem tests a student's ability to code a recursive backtracking algorithm and return a 'best' solution. It demands facility with passing by reference vs. passing by value, managing many parameters, and working with an unfamiliar object type (`Generator`). However, more importantly, it tests a student's ability to read through instructions that suggest a complex problem *("a unique struct with three variables? Two optimizing conditions? 3+ parameters??"*) and deduce that a solution will actually be fairly simple. While this may make it a great problem to test comprehension and level of comfort with recursive backtracking, the apparent complexity likely means it wouldn't serve as a particularly good first recursive backtracking problem.

I was personally motivated to design this problem in order to bridge my academic focii of climate change & environmental policy with the fun I had problem-solving in CS106B. As I conceived of this problem, I quickly realized how real-life application would necessitate multivariate optimization (beyong just cost and emissions – what about reliability? Energy efficiency? Politics?) and probably a more robust Generator *class* in order to incorporate nuance. However, I still think this was a useful exercise!

Common misperceptions

A number of possible pitfalls may arise as students work on this problem:

- Buggy base case: the following buggy base case code may look correct; however, it fails to properly handle the potential solution of an empty portfolio:

```
if (demand <= 0 || generators.isEmpty()) {
  if (best == -1 || best > current) {
      best = current;
      bestPortfolio = currentPortfolio;
  }
  return;
}
```

- Not re-adding each generator back into the complete Vector of Generators (i.e. omitting the last line of the helper function)
- Passing in variables by value instead of by reference: if `bestPortfolio` or `currentPortfolio` are passed in by value, for example, the algorithm is buggy. This can be difficult to resolve if the student is unaware of the issue.
- Attempting to solve the problem without a helper function: it is very difficult if not impossible to implement a recursive solution without a helper function, given the function & parameter specifications.

Conceptual Mastery

This problem addresses three key learning goals:

- I am excited to use programming to solve real-world problems I encounter outside class.
- I can break down complex problems into smaller subproblems by applying my algorithmic reasoning and recursive problem-solving skills.
- I can evaluate design tradeoffs when creating data structures and algorithms or utilizing them to implement technological solutions.

The difficulty for me in designing the problem was choosing a challenging but manageable function specification. For example, requiring the function to simply return the lowest emissions or cost total but not the best portfolio was too simple, while requiring the function to return the best portfolio in addition to those lowest totals would have likely involved a third helper function – too much.

In addition, producing a rigorous test suite was challenging given that the `Generator` struct isn't a class, and thus cannot respond to operands such as "`==`" or be printed using `cout`. If the scope of this assignment were to produce a problem suitable for a full Assignment, I would have added preceding steps involving constructing a Generator class with class variables and class functions. This would have facilitated more simple testing.