CMPS 2200 Assignment 5

In this assignment we'll look at the greedy and dynamic programming paradigms.

To make grading easier, please place all written solutions directly in `answers.md`, rather than scanning in handwritten work or editing this file.

All coding portions should go in `main.py` as usual.

Part 1: Making Change

The pandemic is over and you decide to take a much needed vacation. You arrive in a city called Geometrica, and head to the bank to exchange N dollars for local currency. In Geometrica they have a currency that is 1-1 with U.S. Dollars, but they only have coins. Moreover the coins are in denominations of powers of 2 (e.g., k denominations of values 2^0, 2^1, ..., 2^k). You wonder why they have such strange denominations. You think about it a while, and because you had such a good Algorithms instructor, you realize that there is a very clever reason.

1a) Given a N dollars, state a greedy algorithm for producing as few coins as possible that sum to N.

enter answer in `answers.md`

1b) Prove that this algorithm is optimal by proving the greedy choice and optimal substructure properties.

enter answer in `answers.md`

1c) What is the work and span of your algorithm?

enter answer in `answers.md`

Part 2: Making Change Again

You get tired of Geometrica and travel to the nearby town of Fortuito. While Fortuito also has a 1-1 exchange rate to the US Dollar, it has an even stranger system of currency where any given bank has a completely arbitrary set of denominations (k denominations of values $D_0, D_2, \ldots, D_k$). There is no guarantee that you can even make change. So you wonder, given N dollars is it possible to even make change? If so, how can it be done with as few coins as possible?

2a) You realize the greedy algorithm you devised above doesn't work in Fortuito. Give a simple counterexample that shows that the greedy algorithm does not produce the fewest number of coins.

enter answer in `answers.md`

2b) Since you paid attention in Algorithms class, you realize that while this problem does not have the greedy choice property it does have an optimal substructure property. State and prove this property.

enter answer in `answers.md`

2c) Use this optimal substructure property to design a dynamic programming algorithm for this problem. If you used top-down or bottom-up memoization to avoid recomputing solutions to subproblems, what is the work and span of your approach?

enter answer in `answers.md`

Part 3: Edit Distance

In class we proved an optimal substructure property for the **Edit Distance** problem. This allowed us to implement a simple recursive algorithm in Python that was horribly inefficient. We're going to implement a slightly different version of edit distance that includes substitutions, develop a top-down memoization scheme and then implement a way to visualize the optimal sequence of edits.

3a) The code for `MED` from the lecture notes is provided as a starting point in `main.py`. We will consider a slightly different version of the edit distance problem which allows for insertions, deletions and substitutions. We will assume that insertions, deletions and substitutions all have the same unit cost. State the optimal substructure property for this version of the edit distance problem and modify `MED` accordingly.

3b) Now implement `fast_MED`, a **top-down** memoized version of `MED`. Test your implementation code using `test_MED`.

3c) Now that you have implemented an efficient algorithm for computing edit distance, let's turn to the problem of identifying the actual edits between two sequences.

Notice that in the process of computing the optimal edit distance, we can also keep track of the actual sequence of edits to each position of S and T. Update your implementation of `fast_MED` to return the optimal edit distance as well as an *alignment* of the two strings which show the edits that yield this distance. An alignment just shows what changes are made to S to transform it to T. For example, suppose S=`relevant` and T=`elephant`. If insertion, deletion and substitution costs are all equal to 1, then the edit distance between S and T is 3 and an alignment of these two strings would look like this:

```
relev-ant
-elephant
```

Implement `fast_align_MED` to return the aligned versions of S and T, and test your code with `test_alignment`.