CMPS 2200 Assignment 5

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

As with previous assignments, your code implementations will go in `main.py`. Please add your written answers to `answers.md` which you can convert to a PDF using `convert.sh`. Alternatively, you may scan and upload written answers to a file names `answers.pdf`.

Part 1: Fixed-Length vs. Variable-Length Codes

In class we looked at the Huffman coding algorithm for data compresssion. Let's implement the algorithm and look at its empirical performance on a dataset of 5 text files (`alice29.txt`, `asyoulik.txt`, `f1.txt`, `fields.c`, `grammar.lsp`).

1a) We have implemented a means to compute character frequencies in a text file with the function `get_frequencies` in `main.py`. Implement `fixed_length_cost` to compute the costs for fixed length encodings for each text file.

1b) Complete the implementation of Huffman coding in `make_huffman_tree`. Note that we manipulate binary trees in the priority queue using the object `TreeNode`. Moreover, once the tree is constructed, we must compute the actual encodings by traversing the Huffman tree that has been constructed. To do this, complete the implementation of `get_code`, which is a typical recursive binary tree traversal. That is, given a tree node, we recursively visit the left and right subtrees, appending a `0` or `1` to the encoding in each direction as appropriate. If we visit a leaf of the tree (which represents a character in the alphabet) we store the collected encoding for that character in `code`.

1c) Now implement `huffman_cost` to compute the cost of a Huffman encoding for a character set with given frequencies, and test your implementation of Huffman coding using `test_huffman_simple`.

1d) Test your Huffman coding on the 5 given text files using `analyze_files`. Fill out a table containing of the encoding cost of each file for both fixed-length and Huffman encodings. Fill out a final column which gives the ratio of Huffman coding cost to fixed-length coding cost. You may modify `analyze_files`. Do you see a consistent trend? If so, what is it?

enter answer in `answers.md`

1e) Suppose that we used Huffman coding on a document with alphabet Σ in which every character had the same frequency. What is the expected cost of a Huffman encoding for the document? Is it consistent across documents?

enter answer in `answers.md`

Part 2: 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.

2a) Given a N dollars, state a greedy algorithm for producing as few coins as possible that sum to N.

enter answer in `answers.md`

2b) Prove that this algorithm is optimal by proving the greedy choice and optimal substructure properties.

enter answer in `answers.md`

2c) What is the work and span of your algorithm?

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`.