Assignment 6 — Dynamic Programming

Due: November 27th

Dynamic programming optimizes the idea of recursive problem solving. By sacrificing space complexity for time complexity, it can more efficiently tackle problems that require solving multiple interdependent subproblems and testing different combinations of subsolutions.

Deliverables:

GitHub Classroom: `https://classroom.github.com/a/JV6nyTT0`

Required Files: `compile.sh`, `run.sh`

Optional Files: `*.c`, `*.h`, `*.py`, `*.java`, `*.js`, `*.json`, `*.ts`, `*.clj`, `*.kt`, `*.jl`, `*.rs`

Part 1: Sequence Alignment

Consider the following everyday, real-world problem[1]: suppose that you are given two strings, a known word and an unknown word. You are asked to determine the likelihood that the unknown word is a misspelling of the known word. For example, you might argue that "profssir" is an understandable typo of "professor":

```
p r o f e s s o r
p r o f - s s i r
```

Here, the words differ only in a missing 'e' and an 'o' replaced by the 'i' that is next to it on many keyboards. The 'e' is thus *aligned* with a *gap*, the 'o' with the 'i', and all other characters with their matches. The alignment can then be assessed by a *scoring function* δ, which assigns integer scores to aligned pairs[2].

Part 2: A Greedy Approach

In real life, one might subconsciously tackle this problem with a straightforward, greedy approach:

GreedyAlignment$(x = x_0x_1 \dots x_{n-1}, y = y_0y_1 \dots y_{m-1}, \delta : \Sigma \times \Sigma \to \mathbb{Z})$

Input: A string x of length n, a string y of length m, and a scoring function δ
Output: The maximum score of aligning x with y

1: **let** i, j, and k be 0
2: **while** $i < n$ **and** $j < m$ **do**
3: **if** $\delta(x_i, y_j) > \delta(x_i, \texttt{-})$ **and** $\delta(x_i, y_j) > \delta(\texttt{-}, y_j)$ **then**
4: **let** i be $i + 1$, j be $j + 1$, and k be $k + \delta(x_i, y_j)$
5: **else if** $\delta(x_i, \texttt{-}) > \delta(\texttt{-}, y_j)$ **then**
6: **let** i be $i + 1$ and k be $k + \delta(x_i, \texttt{-})$
7: **else**
8: **let** j be $j + 1$ and k be $k + \delta(\texttt{-}, y_j)$
9: **return** $k + (\sum_i^{n-1} \delta(x_i, \texttt{-})) + (\sum_j^{m-1} \delta(\texttt{-}, y_j))$

However, greedy algorithms such as GreedyAlignment are unsound. Prove the following lemmas:

Lemma: There exist strings x, y and a scoring function δ such that GreedyAlignment produces a lower scoring alignment than optimal.

Lemma: There exist strings x, y and a scoring function δ such that GreedyAlignment aligns every character with a mismatched character, even though matching characters exist (and would score higher).

[1] This problem is also common in bioinformatics, where aligning DNA sequences can infer a gene's mutations or functions.
[2] For example, $\delta(\texttt{e}, \texttt{-})$ should be higher than $\delta(\texttt{e}, \texttt{m})$; it is unlikely that 'e' is replaced by 'm' on the other side of the keyboard.

Part 3: A Dynamic Programming Approach

In your programming language of choice (see Assignment 1), design and implement a dynamic programming algorithm to align strings. To help you get started, note the following observations:

- Given an instance of the problem, there are two primary ways to make the problem smaller: reduce the length of the first string, or reduce the length of the second string.
- Given a pair of characters in those strings, three possibilities exist: either they should be aligned with each other, the first should be aligned with a gap, or the second should be aligned with a gap.

Each input will consist of two non-empty strings followed by a symmetric, integer matrix enumerating a scoring function's possible outputs. You may assume that both strings will consist of lowercase English letters.

For example, the above strings, along with a simple scoring function, could be represented as:

```
professor
profssir
. e f i o p r s -
e 2 0 0 0 0 1 1 0
f 0 2 0 0 0 1 0 0
i 0 0 2 1 0 0 0 0
o 0 0 1 2 1 0 0 0
p 0 0 0 1 2 0 0 0
r 1 1 0 0 0 2 0 0
s 1 0 0 0 0 0 2 0
- 0 0 0 0 0 0 0 0
```

Your program must accept as a command line argument the name of a file containing two strings and a matrix as described above, then print to `stdout` the highest scoring alignment according to the following format:

- The aligned strings must appear on their own lines, with the first string before the second string.
- The aligned characters must be space-separated, and '`-`' characters must be used to represent gaps.

For example:

```
>$ ./compile.sh
>$ ./run.sh in1.txt
Alignment with score 15:
p r o f e s s o r
p r o f - s s i r
```

You may further assume that there will exist exactly one optimal alignment, such that output as described above will be unique. Your program will be tested using `diff`, so its printed output must match *exactly*.

Part 4: Submission

The following files are required and must be pushed to your GitHub Classroom repository by the deadline:

- `report.pdf` — Definition, base cases, formula, and solution for a dynamic programming algorithm to align strings, along with proofs of the lemmas given in Part 2 and analysis of complexity.
- `compile.sh` — A Bash script to compile your submission (even if it does nothing), as specified.
- `run.sh` — A Bash script to run your submission, as specified.

The following files are optional:

- `*.c`, `*.h`, `*.py`, `*.java`, `*.js`, `*.json`, `*.ts`, `*.clj`, `*.kt`, `*.jl`, `*.rs` — The source code of a working program to align strings, as specified.

Any files other than these will be ignored.