Homework 2

Due on April 16th, 2021

In this assignment, you will come up with and implement two dynamic programming algorithms.
Key instructions:

- 1. Please indicate your name at the top of each assignment(in comments for each piece of code you submit) and cite any references you used (including articles, books, code, websites, and personal communications). If you're not sure whether to cite a source, err on the side of caution and cite it. Remember not to plagiarize: all solutions must be written by yourself.
- 2. Please use **Python 3** for this assignment, the grading of this assignment will be done in a Python 3.7.5 environment.
- 4. Please do NOT use any additional imports, only write your code where you see `TODO: YOUR CODE HERE`, and change your return value accordingly.
- 5. Please modify and submit the following files:
  - `diffing.py`
  - `respacing.py`
- 6. Reminder on Late Policy: Each student has a total of one slip day that may be used without penalty for homework.

1 Diffing with costs

1.1 Problem statement

We saw in lecture that 'diffing' two strings can be accomplished with dynamic programming. In this problem we will solve a more general version. We would like to align two strings, `s` and `t`, in a way to produce a *minimum-cost* alignment.
To produce an alignment on two strings `s` and `t`, we insert the special character '-' some number of times into each string to produce `align_s` and `align_t` so that:

- `align_s` and `align_t` have the same length, and
- There is no `i` such that `align_s[i]` and `align_t[i]` are both '-'.

The cost of an alignment is given by a cost function, which we will call `cost`. The cost of an alignment is the sum over all `i` of `cost(align_s[i], align_t[i])`. The cost of aligning a letter with itself is always 0.
For example, given this cost table:

		Letter from s			
		a	b	c	-
Letter from t	a	0	1	5	3
	b	5	0	5	3
	c	3	4	0	3
	-	2	2	1	·

The *minimum-cost* alignment between string `s =` 'acb' and `t =` 'baa' is given by:

```
align_s:  -acb
align_t:  ba-a
```

With a total cost of 5, which is lower than any other alignment.
This problem arises, for example, in DNA sequencing; given two strands of DNA, there are many sequences of mutations (insertions, deletions, etc.) which would have transformed one to the other; we would like to find the most probable. We know that certain mutations are more likely than other, and these probabilities are reflected in the cost table(the pair with higher likelihood has a lower cost).

1.2 TODO

In `diffing.py`:

fill_cell . You should fill in `fill_cell(table, i, j, s, t, cost)`, which will be called once per cell. `table` is the table as filled in so far; use its `get` method to find these values. Your function should not mutate the table directly, but rather return a `DiffingTableCell`. `cost` is a function which encodes a cost function: it takes two characters and returns their cost. Each call to `fill_cell` should call it no more than four times. `fill_cell` should run in $O(1)$ time.

cell_ordering . You should fill in `cell_ordering(n,m)`, which will be called once. n and m are the lengths of the strings s and t. `cell_ordering` should return a list of two-tuples; these are the coordinates of cells in the order you would like `fill_cell` to be called. No cell should be repeated, but you do not need to use all cells.

diff_from_table . You should fill in `diff_from_table(s,t, table)`. `s` is the string being respaced, and `table` is the table that is already filled in. `diff_from_table` should run in $O(n + m)$ time, and make no calls to `cost`.

Once you have finished implementing these 3 functions, you can do a quick sanity check by running the `main` given at the bottom of `diffing.py` which puts all the pieces together with our dynamic programming framework. See if the result is coherent with the example we gave you in this document.

```
python3 diffing.py
```

Test harness To help you concentrate more on the actual algorithms part, we have also provided you a test harness `test_diffing.py` that runs tests on your `diffing.py` implementation using `diffing_tests.txt` as the input testfile. It is hence your responsibility to make sure your code works before submission. However, the test harness and the test cases we gave you are **non-exhaustive**, meaning *passing these tests alone does not necessarily guarantee a perfect score*.

2 Respacing

2.1 Problem statement

The respacing problem is to put spaces back into a string that has lost them, given a dictionary. For example, given the string "itwasthebestoftimes" and an English dictionary, we would like to reconstruct the original sentence: "it was the best of times".

2.2 TODO

In `respacing.py`:

fill_cell . You should fill in `fill_cell(T, i, j, string, is_word)`, which will be called once per cell. T is the table as filled in so far; use its `get` method to find these values. Your function should not mutate T directly, but rather return a `RespaceTableCell`. `is_word` is a function which encodes a dictionary: given a string, it returns true or false, according to whether the string is a word. Each call to `fill_cell` should call it no more than once. `fill_cell` should run in $O(N)$ time.

cell_ordering . You should fill in `cell_ordering(N)`, which will be called once. N is the length of the string to be respaced. `cell_ordering` should return a list of two-tuples; these are the coordinates of cells in the order you would like `fill_cell` to be called. No cell should be repeated, but you do not need to use all cells.

respace_from_table . You should fill in `respace_from_table(s, table)`. `s` is the string being respaced, and `table` is the table that is already filled in. `respace_from_table` should run in $O(N^2)$ time, and make no calls to `is_word`.

Just like the first problem, an example for putting these all together with our dynamic programming framework is given in the `main` at the bottom of `respacing.py`

```
python3 respacing.py
```

Test harness Also we have also provided you a **non-exhaustive** test harness `test_respace.py` that runs tests on your `respacing.py` implementation using `respace_tests.txt` as the input testfile. *Passing these tests alone does not necessarily guarantee a perfect score.*

3 Dynamic Programming Framework

We have provided the skeleton code for you, in `dynamic_programming.py`, `diffing.py`, and `respacing.py`. It is designed to let you just think about the algorithm and worry less about the software engineering stuff, while also ensuring that the algorithm you design is in fact a dynamic program. You will write some code to fill in a dynamic programming table, and then to compute the final answer from this table.
The table for respacing is $(N+1) \times (N+1)$, where N is the length of the word to be respaced. Each cell may contain a single boolean and a single integer.
The table for diffing is $(|s|+1) \times (|t|+1)$, where s and t are the strings to be diffed and $|s|$ stands for the length of string s. Each cell may contain a single integer and two characters (a character is a length-1 string).

4 Testing and logistics

4.1 Outside resources

We are asking you to come up with new algorithms using dynamic programming. We expect that finding an appropriate optimal substructure will be the bulk of the work – the code is fairly short once you have found one. **You should NOT use or seek any resources that describe how to solve these problems, even in pseudocode.**

4.2 Testing

In addition to the provided tests, we encourage you to write additional ones.
You can assume that all strings (words in respacing, input to respacing, and strings in diffing) consist of lowercase letters a,b,c,...z, have length at least 1, and are not None.