Informatics 2 – Introduction to Algorithms and Data Structures

Lab Sheet 5: Edit distance in Python

week 2, semester 2: 3rd-7th 2025

In this Lab Sheet, we will step through the implementation of the Edit distance dynamic programming algorithm (covered in lecture 20). We know that attendance for the labs in week 2 was pretty quiet - so please feel free to also work on the Labsheet 4 this week.

Most of the basic concepts you need to develop implementations have been introduced in earlier Labs - how to define arrays (including multi-dimensional arrays), print functionality, and timing-of execution using `timeit`. Please ask your demonstrator if you are struggling with anything.

1 Edit Distance

In this section we will experiment with a variety of implementations of *edit distance*. To this end, please start a new Python file called `editdistance.py` to store your implementations of these functions.

1.1 Array-based Implementation

The first implementation we will attempt is the straightforward dynamic programming algorithm we developed in class, where we fill all entries within two $(m+1) \times (n+1)$ arrays/tables d and a in order of increasing row index i (and within that row, in order of increasing column index j). You should define this function as

```
def edit_dyn(s, t):
```

1.2 Printing the optimal alignment

Lecture slides 18 contain a discussion of how we can use the arrays d and a to actually *find* an alignment that meets the edit distance. Extend your code for `edit_dyn` to implement the ideas there to display the/a optimal alignment for the input sequences. Here are some sample outputs we would expect to see:

```
>>> editdistance.edit_dyn("house", "home")
```

```
h o u s e
h o - m e
```

(or maybe `h o m - e` on the bottom)

```
>>> editdistance.edit_dyn("biddable", "routinely")
- b i d d a b l e
r o u t i n e l y

>>> editdistance.edit_dyn("biddable", "inability")
b i d d a b - l - - e
- i - n a b i l i t y
```

For this task, it may help to consider the discussion/details at the end of lecture 18.

Be aware that for getting the alignment to appear on the screen, we can avoid the automatic "newline" by specifying a particular `end` character, for example: `print(x, end = ' ')`

An alternative way to approach the task of building the optimal alignment is to define *two* recursive functions inside `edit_dyn`, one function to build the alignment of string s (the string s, with embedded '-' characters) from table a, and the other to build the alignment of string t. Then after those two functions have returned the results, we can just print the padded version of s followed by printing the padded version of t.

1.3 "Memoization" as an alternative

We could alternatively have written a recursive algorithm for edit distance, and then used memoization to eradicate the repeated re-computations. In this case the memo needs to work with two arguments :

- If it finds the solution for `s,t` already stored within the memo, it returns that value.
- Alternatively, if the value has not been computed yet, it makes the necessary computations (possibly including recursive calls) to compute the solution, but then adds this solution to the memo for `s,t` before `return`-ing the answer.

For experimenting with the memoization, we will just focus on computing the edit distance (and not worry about computing/displaying the alignment afterwards). You will/should start by implementing a naïve recursive version of this algorithm (named `edit_rec`, say).

After that is done, there are a number of ways of achieving memoization in Python.

- We can follow the same approach as we did with the memoization of `fib` in Lecture 18, except this time defining a 2-parameter version of `memoize` (to handle the fact that edit distance takes two parameters, s and t).
- There is a higher-order function called `lru_cache` from the `functools` module which can be used to take care of the memoization for you. To use this "decorator" function, you just need to *import* the `functools` module at the beginning of your Python file, and then to apply the decorator immediately before the definition of your recursive function:

```
@functools.lru_cache(maxsize=None)
def edit_rec(s,t):
```

You should set up some experiments to compare the running-times of `edit_dyn` and `edit_rec` against your memoized version of the recursive implementation, using `timeit` to evaluate the times. You should consider reasonably long pairs of strings (say 15 characters or more, for each) to see the effect of the memoization.

Mary Cryan, 2nd February 2025