1 Hw07 Coding: Dynamic Programming

In this notebook, we'll work through the Global Alignment (Edit Distance) problem using dynamic programming and will see how backtracing can be used to reconstruct solutions from the DP table.

1.0.1 If you're using Datahub:

- Run the cell below **and restart the kernel if needed**

1.0.2 If you're running locally:

You'll need to perform some extra setup. #### First-time setup * Install Anaconda following the instructions here: https://www.anaconda.com/products/distribution * Create a conda environment: `conda create -n cs170 python=3.8` * Activate the environment: `conda activate cs170` * See for more details on creating conda environments https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html * Install pip: `conda install pip` * Install jupyter: `conda install jupyter`

Every time you want to work

- Make sure you've activated the conda environment: `conda activate cs170`
- Launch jupyter: `jupyter notebook` or `jupyter lab`
- Run the cell below **and restart the kernel if needed**

In [1]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

In [ ]:
```
import otter
assert (otter.__version__ >= "4.4.1"), "Please reinstall the requirements and restart your kern

grader = otter.Notebook("hw07.ipynb")
import numpy.random as random
import string
import random
import pylev
import tqdm
import time
from inputs1 import all_arrs, all_sols

rng_seed = 42
```

1.1 1. Global Alignment (Edit Distance)

The edit distance problem finds the minimal number of insertions, deletions and substitutions of characters required to transform one word into another. A big application of this problem is finding the global alignment between two strings, which is often used in computational biology.

As described in the textbook https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf#page=6.

> A natural measure of the distance between two strings is the extent to which they can be aligned, or matched up. Technically, an alignment is simply a way of writing the strings one above the other. For instance, here are two possible alignments of SNOWY and SUNNY:
>
> S−NOWY | −SNOW−Y SUNN−Y | SUN−−NY Cost: 3 | Cost: 5
>
> The "−" indicates a "gap"; any number of these can be placed in either string. The cost of an alignment is the number of columns in which the letters differ. And the edit distance between two strings is the cost of their best possible alignment.

In this problem, you will implement an algorithm to compute the alignment between two strings x and y, specifically, your algorithm should return the global alignment (as shown above), not just an integer value denoting the edit distance.

1.1.1 The following section will walk you through how to implement this algorithm.

This section contains ungraded multiple choice questions to test your understanding. If you like, you may skip to the last question which is the only graded question. Inputs: - x:string = length n string - y:string = length m string

Algorithm Sketch: 1. Compute the dp subproblems as described in class and the textbook 2. Using the memoized subproblems from step 1, reconstructing the optimal global alignment

Step 1 can be computed by simply implementing the pseudocode described in the textbook.

Step 2 can be computed using an approach called backtracking which we walk through here. Recall that all DP have underlying DAG's where nodes represent subproblems. See https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf#page=9. On this DAG, the DP algorithm finds the shortest path from `(0,0)` to `(n,m)`. The length of the shortest path is our edit distance, and the edges in the path correspond to the global alignment. In our back tracking algorithm, we start at `(n,m)` and reconstruct the shortest path to `(0,0)`. Since we start with `(n,m)` and end at `(0,0)`, we are back tracking the computations we did in step 1, hence the name.

Sanity Check (ungraded): Suppose we computed the DP matrix on the strings `x` and `y` want to find the edit distance between the first `5` characters of `x` and the first `6` characters of `y`. On the underlying DAG, this corresponds to the shortest path from `(0,0)` to which node? Give your answer as a tuple containing 2 integers.

```
In [ ]: node = (5,6) # SOLUTION
```

```
In [ ]: grader.check("s1")
```

Now suppose that we have a way to reconstruct this shortest path, we need to convert the edges on this path into the actual alignment.

Sanity Check (ungraded): Suppose that our algorithm backtracks to node `(i,j)` and determines that the edge `(i-1,j)->(i,j)` is in this shortest path. So far, the algorithm computed 2 strings `x_align` and `y_align` based on the path from `(i,j)` to `(n,m)`. These correspond to an alignment of the substrings `x[i:n]` and `y[j:n]`. Given this new edge, what characters should you add to `x_align` and `y_align`? Input your answer choice as list of ints (ie `ans = [1]` or `ans = [1,2]`), where each int represents one of the following choices:

1. add a gap to the start of `x_align`
2. add a gap to the start of `y_align`
3. add `x[i-1]` to the start of `x_align`
4. add `y[j-1]` to the start of `y_align`

Hint: a character must be added to both strings since at each step, `len(x_align) == len(y_align)`.

```
In [ ]: ans = [2,3] # SOLUTION
```

```
In [ ]: grader.check("s2")
```

Sanity Check (ungraded): Suppose that our algorithm backtracks to node `(i,j)` and determines that the edge `(i,j-1)->(i,j)` is in this shortest path. So far, the algorithm computed 2 strings `x_align` and `y_align` based on the path from `(i,j)` to `(n,m)`. These correspond to an alignment of the substrings `x[i:n]` and `y[j:n]`. Given this new edge, what characters should you add to `x_align` and `y_align`? Input your answer choice as list of ints (ie `ans = [1]` or `ans = [1,2]`), where each int represents one of the following choices:

1. add a gap to the start of `x_align`
2. add a gap to the start of `y_align`
3. add `x[i-1]` to the start of `x_align`

4. add `y[j-1]` to the start of `y_align`

Hint: a character must be added to both strings since at each step, `len(x_align) == len(y_align)`.

```
In [ ]: ans = [1,4] # SOLUTION
```

```
In [ ]: grader.check("s3")
```

Sanity Check (ungraded): Suppose that our algorithm backtracks to node `(i,j)` and determines that the edge `(i-1,j-1)->(i,j)` is in this shortest path. So far, the algorithm computed 2 strings `x_align` and `y_align` based on the path from `(i,j)` to `(n,m)`. These correspond to an alignment of the substrings `x[i:n]` and `y[j:n]`. Given this new edge, what characters should you add to `x_align` and `y_align`? Input your answer choice as list of ints (ie `ans = [1]` or `ans = [1,2]`), where each int represents one of the following four choices:

1. add a gap to the start of `x_align`
2. add a gap to the start of `y_align`
3. add `x[i-1]` to the start of `x_align`
4. add `y[j-1]` to the start of `y_align`

Hint: a character must be added to both strings since at each step, `len(x_align) == len(y_align)`.

```
In [ ]: ans = [3,4] # SOLUTION
```

```
In [ ]: grader.check("s4")
```

Since we know how to translate edges into global alignment, we now want to reconstruct the actual path from `(n,m)` to `(0,0)`. If an edge `(a,b)->(c,d)` is part of the shortest path, this means the subproblem `(a,b)` was used to compute the solution to `(c,d)`. For the edit distance problem, we know that the subproblem `(i,j)` is computed from the either `(i-1,j)`, `(i,j-1)`, or `(i-1,j-1)`. Therefore, if the node `(i,j)` is visited in the shortest path, then one of the edges `(i-1,j)->(i,j)`, `(i,j-1)->(i,j)`, or `(i-1,j-1)->(i,j)` is in the shortest path.

We can figure out the correct edge based on the values in the dp matrix. Recall the recurrence of edit distance: `dp[i][j] = min(dp[i - 1][j] + 1, dp[i][j - 1] + 1, dp[i - 1][j - 1] + diff(i,j))`. This means that at least one of the three values `dp[i - 1][j] + 1`, `dp[i][j - 1] + 1`, or `dp[i - 1][j - 1] + diff(i,j)` must equal `dp[i][j]`. If the value equals `dp[i][j]`, then that is a possible previous subproblem; otherwise, it is not. If there are multiple possible previous problems, you may back track to any one of them.

Sanity Check (ungraded): Suppose you know that `dp[i][j] = 5` and following values in the DP matrix. Which subproblems could be used to compute `dp[i][j]`? Input your answer choice as list of ints (ie `ans = [1]` or `ans = [1,2]`) 1. `dp[i-1][j] = 4` 2. `dp[i][j-1] = 5` 3. `dp[i-1][j-1] = 5, diff(i,j) = 0`

```
In [ ]: ans = [1,3] # SOLUTION
```

```
In [ ]: grader.check("s5")
```

Sanity Check (ungraded): Suppose you know that `dp[i][j] = 9` and following values in the DP matrix. Which subproblems could be used to compute `dp[i][j]`? Input your answer choice as list of ints (ie `ans = [1]` or `ans = [1,2]`) 1. `dp[i-1][j] = 9` 2. `dp[i][j-1] = 8` 3. `dp[i-1][j-1] = 9, diff(i,j) = 1`

```
In [ ]: ans = [2] # SOLUTION
```

```
In [ ]: grader.check("s6")
```

Following this logic, we start at `(n,m)` and repeatedly find the previous node until we reach `(0,0)`. Each time we backtrack one step, we update the alignment based on the edge we took.

1.2 Edit Distance (graded)

```
In [ ]: def edit_distance(x, y):
            """
            args:
                x:string = the first word.
                y:string = The second word.

            return:
                Tuple[String,String] = the optimum global alignment between x and y. The first string i
                tuple corresponds to x and the second to y. Use hypen's '-' to represent gaps in each s
            """
            # BEGIN SOLUTION
            n = len(x)
            m = len(y)
            dp = []
            def diff(i,j):
                return 1 if x[i - 1] != y[j - 1] else 0

            # base cases
            for i in range(n + 1):
                dp.append([-1] * (m + 1))
            for i in range(n + 1):
```

```
    dp[i][0] = i
for j in range(m + 1):
    dp[0][j] = j

# compute recurrence
for i in range(1, n + 1):
    for j in range(1, m + 1):
        by_deletion = dp[i - 1][j] + 1
        by_insertion = dp[i][j - 1] + 1
        by_substitution = dp[i - 1][j - 1] + diff(i,j)
        dp[i][j] = min(by_deletion, by_insertion, by_substitution)

# Backtrace to compute the optimum alignment:
x_align, y_align = "", ""
i,j = n,m
while (i, j) != (0,0):
    deletion = dp[i-1][j] + 1 if i > 0 else float("inf")
    insertion = dp[i][j-1] + 1 if j > 0 else float("inf")
    substitution = (dp[i-1][j-1] + diff(i,j)) if i > 0 and j > 0 else float("inf")
    moves = [
        (deletion,(i-1,j)),
        (insertion,(i,j-1)),
        (substitution,(i-1,j-1)),
    ]
    prev_i, prev_j  = min(moves)[1]
    x_align += x[i-1] if prev_i == i-1 else '-'
    y_align += y[j-1] if prev_j == j-1 else '-'
    i,j = prev_i, prev_j

return x_align[::-1], y_align[::-1]
# END SOLUTION
```

Note: your solution should not take inordinate amounts of time to run. If it takes more than 60 seconds to run, it is too slow. The staff solution takes 20 seconds on average.

Points: 6

```
In [ ]: grader.check("edit_distance")
```

1.2.1 Debugging

Below, we provide some tests to help you debug your code. Some good test cases to try your algorithm on are:

1. Small, handmade test cases, where you can compute the answer by hand to compare against your algorithm's output.

2. Randomly generated test cases, to test your algorithm's correctness on a wider range of inputs.

Below, we provide some small test cases to get started. You can compare your output against the ground-truth answers from DPV: https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf#page=6

Note that there may be multiple valid solutions!

In [ ]:
```
case_1_1 = 'snowy'
case_1_2 = 'sunny'

align_1_1,align_1_2 = edit_distance(case_1_1, case_1_2)

print(f"case 1: {align_1_1} {align_1_2}")
```

In [ ]:
```
case_2_1 = 'polynomial'
case_2_2 = 'exponential'

align_2_1,align_2_2 = edit_distance(case_2_1, case_2_2)

print(f"case 2: {align_2_1} {align_2_2}")
```

A simplified verion of the otter tests are pasted here for your convenience. Feel free to add whatever print statements or assertions you'd like when debugging.

In [ ]:
```
start = time.time()
def check_word(original, aligned):
    ''' checks that the string `aligned` is obtained by only adding gaps to the string `origina

    assert len(aligned) >= len(original), "your function returned a string which is shorter thar
    i,j = 0,0
    while i < len(original) and j < len(aligned):
        while aligned[j] == '-' and j < len(aligned):
            j += 1
        assert original[i] == aligned[j], "your function returned a string which cannot be produ
        i += 1
        j += 1
    while j < len(aligned):
        assert aligned[j] == '-', "your function returned a string which cannot be produced by
        j += 1

NUM_TRIALS = 200
LETTERS = string.ascii_lowercase
MIN_WORD_SIZE = 250
MAX_WORD_SIZE = 500

letters = string.ascii_lowercase
```

```
for i in tqdm.tqdm(range(NUM_TRIALS)):
    word1_size = random.randint(MIN_WORD_SIZE, MAX_WORD_SIZE)
    word2_size = random.randint(MIN_WORD_SIZE, MAX_WORD_SIZE)
    word1 = ''.join(random.choice(letters) for i in range(word1_size))
    word2 = ''.join(random.choice(letters) for i in range(word2_size))
    align1,align2 = edit_distance(word1, word2)

    assert len(align1) == len(align2), f"""a global alignment requires the two strings to be the
        length, your functions returns two strings of length {len(align1)} and {len(align2)}!"""

    check_word(word1, align1)
    check_word(word2, align2)

    dist = 0
    for a,b in zip(align1,align2):
        if a != b:
            dist += 1
    staff_distance = pylev.levenshtein(word1, word2)
    assert staff_distance == dist, f"""the inputs have an edit distance of {staff_distance}, bu
        strings have a distance of {dist}."""
finish = time.time()
assert finish - start < 60, "your solution timed out"
```

```
100%|                                                                                    | 200/
```

In addition to these test cases, it may be helpful to create some additional test cases on your own. Feel free to add additional tests below:

In [ ]:
```
# TODO: Your tests here!
```

1.3 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

In [ ]:
```
grader.export(pdf=False, force_save=True, run_tests=True)
```