CS 170 Homework 7

Due **10/17/2023, at 10:00 pm (grace period until 11:59pm)**

4-Part Solutions

For all (and only) dynamic programming problems in this class, we would like you to follow a 4-part solution format:

1. **Algorithm Description:** since dynamic programming algorithms can be difficult to explain, you should follow the template below to optimize clarity.
   (a) *Define your subproblem.* In words, define a function f so that the evaluation of f on a certain input gives the answer to the stated problem.

   You should clearly state how many parameters f has, what those parameters represent, what f evaluated on those parameters represents, and what inputs you should feed into f to get the answer to the stated problem.

   (b) *Provide your recurrence relation.* More precisely, give a recurrence relation showing how to compute f recursively, and make sure to provide base cases. If you need to use certain data structures to make computation of f faster, you should say so.

   (c) *Subproblem Ordering:* describe the order in which you should solve the subproblems to obtain the final answer.
2. **Proof of Correctness:** provide some inductive proof that shows why your DP algorithm computes the correct result.
3. **Runtime Analysis:** analyze the runtime of your algorithm.
4. **Space Analysis:** analyze the space/memory complexity of your algorithm.

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

2 Non-Prefix Code

As we have learned in lecture, the Huffman code satisfies the *Prefix Property*, which states that the bit string representing each symbol is not a prefix of the bit string representing any other symbol. One nice property of such codes is that, given a bit string, there is at most one way to decode it back to a sequence of symbols. However, this is not true anymore once we are working with codes that do not satisfy the Prefix Property. For example, consider the code that maps A to 1, B to 01 and C to 101. A bit string 101 can be interpreted in two ways: as C or as AB.

Your task is to, given a bit string s, determine whether it is possible to interpret s as a sequence of symbols. The mapping from symbols to bit strings of the code will be given to you as a dictionary d (e.g., in the example, $d = \{A : 1, B : 01, C : 101\}$); you may assume that you can access each symbol in the dictionary in constant time.

(a) Describe an algorithm that solves this problem in at most $O(nk\ell)$ where n is the length of the input bit string s, k is the number of symbols, and ℓ is an upper bound on the length of the bit strings representing symbols. **Please provide a 4-part solution**.

(b) **(Optional)** How can you modify the algorithm from part (a) to speed up the runtime to $O((n + k)\ell)$?

Solution:

Main Idea: We define our subproblems as follows: let $A[i]$ be a boolean variable representing whether $s[: i]$ can be represented by a sequence of encodings. We can then compute $A[i]$ using the values of $A[j], j < i$ via the following recurrence relation:

$$A[i] = \bigvee_{\substack{\text{any symbol } a \text{ whose encoding} \\ \text{matches the end of } s[: i]}} \text{(whether it is possible to interpret } s[: i] \text{ with } a \text{ as its last symbol)}$$

$$= \bigvee_{\substack{\text{any symbol } a \text{ whose encoding} \\ \text{matches the end of } s[: i]}} \text{(whether it is possible to interpret } s[: i - \text{length}(d[a])]$$

$$= \bigvee_{\substack{\text{symbol } a \text{ in } d \\ s[i-\text{length}(d[a])+1:i]=d[a]}} A[i - \text{length}(d[a])]$$

Note here that we set $A[0] = \text{TRUE}$. Our algorithm simply computes the above formula in a trivial manner.

Pseudocode:

```
procedure TRANSLATE(s):
    Create an array A of length n + 1 and initialize all entries to FALSE.
    Let A[0] = TRUE
    for i := 1 to n do
        for each symbol a in d do
            if i ≥ length(d[a]) and d[a] = s[i − length(d[a]) + 1 : i] then
                A[i] = A[i] ∨ A[i − length(d[a])]
```

return $A[n]$

Proof of Correctness: We can show this via a simple induction argument.

Base Case. When $i = 0$, we can interpret $s[:0]$ as just the empty string. Hence, $A[0] =$ TRUE.

Inductive Step. Suppose that $A[0], \ldots, A[i-1]$ contains the right value. We will show that the above recurrence relation gives the right value for $A[i]$. To do this, we partition interpretations of $s[:i]$ as a sequence of symbols $a_1 \ldots a_k$ based on the ending symbol a_k. For $a_k = a$, if the suffix of $s[:i]$ coincides with $d[a]$, every interpretation $a_1 \ldots a_k$ has a one-to-one correspondence with an interpretation $a_1 \ldots a_{k-1}$ of $s[:i - \text{length}(d[a])]$. From our inductive hypothesis, we know whether exactly whether $s[:i - \text{length}(d[a])]$ is interpretable based on $A[:i - \text{length}(d[a])]$. On the other hand, if the suffix of $s[:i]$ differs from $d[a]$, then there is no interpretation of $s[:i]$ ending with symbol a. Performing a logical or over all symbols a's implies that our recurrence relation yields the right value for $A[i]$.

Finally, note that our program below implements this recurrence in a straightforward way, so the output of our program is indeed $A[n]$, representing whether s is interpretable.

Runtime Analysis: There are n iterations of the outer for loop and k iterations of the inner for loop. Inside each of these loops, checking that the two strings are equal takes $O(\text{length}(d[a])) \leq O(\ell)$ time. Hence, the total running time is $O(nk\ell)$.

Note that it is possible to speed up the algorithm running time to $O((n+k)\ell)$ using a trie instead of reconstructing the string every time, but this is not required to receive full credit for the problem.

Space Complexity Analysis: We store n subproblems in $A[\cdot]$, each subproblem taking up constant space. Thus, the overall space complexity is $O(n)$.

Note that since any $A[i]$ only depends on the previous ℓ subproblems, you can optimize the space to $O(\ell)$.

3 Ideal Targets

You are given a directed acyclic graph $G = (V, E)$ with unweighted edges. Every vertex $v \in V$ has an integer score $s[v]$. For a vertex v, we say that a vertex u is an ideal target for v if:

1. It is possible to go from v to u.
2. $s[u]$ is maximized.

In other words, out of all vertices that v can reach, u (i.e. v's ideal target) is the one with the maximum score.

Given the scores for all vertices, describe a linear-time algorithm to find the ideal targets for every vertex v. (Note that v can be its own ideal target.)

Just provide the algorithm description and runtime analysis. Proof of correctness and space complexity analysis are not required.

Solution: We perform dynamic programming on the DAG in reverse topological order of vertices.

For each vertex v we calculate $dp[v]$ which denotes a tuple containing the ideal target of v and their score. For example, if x were the ideal target of v, then $dp[v] = (x, s[x])$.

These can be updated with the recurrence:

$$dp[v] = \max_{(v,u)\in E} dp[u] \tag{1}$$

Then, we further update $dp[v] = \max(dp[v], w[v])$. Note that all maxes here are taken with respect to the second element in $dp[\cdot]$.

Since this algorithm traverses through each vertex and edgfe a constant number of times, the overall runtime of this algorithm is $O(|V| + |E|)$.

4 Egg Drop

You are given m identical eggs and an n story building. You need to figure out the highest floor $\ell \in \{0, 1, 2, \ldots n\}$ that you can drop an egg from without breaking it. Each egg will never break when dropped from floor ℓ or lower, and always breaks if dropped from floor $\ell + 1$ or higher. ($\ell = 0$ means the egg always breaks). Once an egg breaks, you cannot use it any more. However, if an egg does not break, you can reuse it.

Let $f(n, m)$ be the minimum number of egg drops that are needed to find ℓ (regardless of the value of ℓ).

(a) Find $f(1, m)$, $f(0, m)$, $f(n, 1)$, and $f(n, 0)$. Briefly explain your answers.

(b) Consider dropping an egg at floor x when there are n floors and m eggs left. Then, it either breaks, or doesn't break. In either scenario, determine the minimum remaining number of egg drops that are needed to find ℓ in terms of $f(\cdot, \cdot)$, n, m, and/or x.

(c) Find a recurrence relation for $f(n, m)$.

Hint: whenever you drop an egg, call whichever of the egg breaking/not breaking leads to more drops the "worst-case event". Since we need to find ℓ regardless of its value, you should assume the worst-case event always happens.

(d) If we want to use dynamic programming to compute $f(n, m)$ given n and m, in what order do we solve the subproblems?

(e) Based on your responses to previous parts, analyze the runtime complexity of your DP algorithm.

(f) Analyze the space complexity of your DP algorithm.

(g) (**Extra Credit**) Is it possible to modify your algorithm above to use less space? If so, describe your modification and re-analyze the space complexity. If not, briefly justify.

Solution:

(a) We have that:

- $f(1, m) = 1$, since we can drop the egg from the single floor to determine if it breaks on that floor or not.
- $f(0, m) = 0$, since there is only one possible value for ℓ.
- $f(n, 1) = n$, since we only have one egg, so the only strategy is to drop it from every floor, starting from floor 1 and going up, until it breaks.
- $f(n, 0) = \infty$ for $n > 0$, since the problem is unsolvable if we have no eggs to drop.

(b) If the egg breaks, we only need to consider floors 1 to $x - 1$, and we have $m - 1$ eggs left since an egg broke, in which case we need $f(x - 1, m - 1)$ more drops. If the egg doesn't break, we only need to consider floors $x + 1$ to n, and there are m eggs left, so we need $f(n - x, m)$ more drops.

(c) The recurrence relation is

$$f(n,m) = 1 + \min_{x \in \{1 \ldots n\}} \max\{f(x-1, m-1), f(n-x, m)\}.$$

When we drop an egg at floor x, in the worst case, we need $\max\{f(x-1,m-1), f(n-x,m)\}$ drops. Then, the optimal strategy will choose the best of the n floors, so we need $\min_{x\in\{1\ldots n\}} \max\{f(x-1,m-1), f(n-x,m)\}$ more drops.

(d) We solve the subproblems in increasing order of m, n, i.e.:

```
for j in range(m+1):
    for i in range(n+1):
        solve f(i, j)
```

(e) We solve nm subproblems, each subproblem taking $O(n)$ time. Thus, the overall runtime is $O(n^2 m)$.

(f) The only thing we have to store is the DP array f, which contains nm elements. Thus, the overall space complexity is $O(nm)$

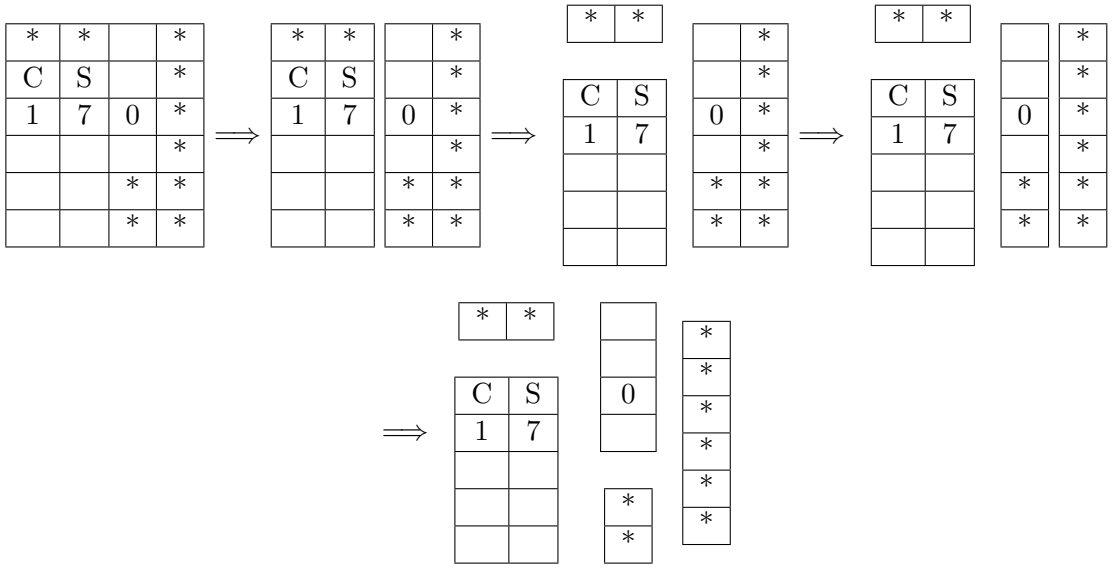
(g) Yes, it is possible! Notice that in our recurrence relation in part (c), we only need the values of $f(\cdot, m)$ and $f(\cdot, m-1)$. So we can just store the last two "columns" computed so far. The pseudocode for this would like as follows:

```
def eggdrop(n, m):
    if m == 0: # base case for m=0
        return float("inf")
    if n == 0:
        return 0

    curr = [i for i in range(n+1)] # base case for m=1

    for j in range(2, m+1):
        prev = copy(curr)

        for i in range(j+1):
            curr[i] = i
        for i in range(j+1, n+1):
            curr[i] = 1 + min([
                                max(prev[x-1], curr[i-x])
                                for x in range(1, i+1)
                            ])

    return curr[n]
```

5 My Dog Ate My Homework

One morning, you wake up to realize that your dog ate some of your CS 170 homework paper, which is an $m \times n$ rectangular grid of squares. Some of the squares have holes chewed through them, and you cannot use paper that has a hole in it. You would like to cut the paper into pieces so as to separate all the tattered squares from all the clean, un-bitten squares. You want to do this so that you can save as much as your work as possible.

For example, shown below is a 6×4 piece of paper where the bitten squares are marked with *. As shown in the picture, one can separate the bitten parts out in exactly four cuts.



(Each *cut* is either horizontal or vertical, and of one piece of paper at a time.)

Design a DP based algorithm to find the smallest number of cuts needed to separate all the bitten parts out. Formally, the problem is as follows:

> **Input:** Dimensions of the paper $m \times n$ and an array $A[i, j]$ such that $A[i, j] = 1$ if and only if the ij^{th} square has holes bitten into it.
>
> **Goal:** Find the minimum number of cuts needed so that the $A[i, j]$ values of each piece are either all 0 or all 1.

(a) Define your subproblem.

Hint: try making any arbitrary cut. What two subproblems do you now have?

(b) Write down the recurrence relation for your subproblems. A fully correct recurrence relation will always have the base cases specified.

(c) Describe the order in which we should solve the subproblems in your DP algorithm.

(d) What is the runtime complexity of your DP algorithm? Provide a justification.

(e) What is the space complexity of your algorithm? Provide a justification.

Solution:

(a) **Subproblem Definition:** We define $B[i_1, j_1, i_2, j_2]$ to be the minimum number of cuts needed to separate the sub-matrix $A[i_1 \leq i_2, j_1 \leq j_2]$ into pieces consisting either entirely of bitten pieces or clean pieces.

(b) **Recurrence Relation:**

$$B[i_1, j_1, i_2, j_2] = \min \begin{cases} 0, & \text{if all entries of } A[i_1 \dots i_2, j_1 \dots j_2] \text{ are equal} \\ 1 + B[i_1, j_1, i_1 + k, j_2] + B[i_1 + k + 1, j_1, i_2, j_2] & \text{for any } k \in \{1, \dots, i_2 - i_1\} \\ 1 + B[i_1, j_1, i_2, j_1 + k] + B[i_1, j_1 + k + 1, i_2, j_2] & \text{for any } k \in \{1, \dots, j_2 - j_1\} \end{cases}$$

Alternatively, you could have also encapsulated the 0 base case in all single-square pieces, and determined if a piece was pure via the merging, see below.

(c) **Subproblem Order:** we solve them in increasing order of $(j_1 - i_1 + 1)(j_2 - i_2 + 1)$. In other words, we solve all the smallest subproblems first (e.g. containing one square) and build our DP array up to our result $B[1, m, 1, n]$, which covers the entire paper.

(d) **Runtime Analysis:** Two answers are acceptable: $O((m+n)m^2n^2)$ and $O(m^3n^3)$

We have $O(m^2n^2)$ total subproblems: $O(mn)$ possibilities for (i_1, j_1), and $O(mn)$ possibilities for (i_2, j_2). For each subproblem, we examine up to m possible choices for horizontal splits, and n possible choices for vertical splits. A single split consideration will result in two smaller subproblems, which we can assume have already been solved, so we just need to find the best split, which takes $O(n + m)$ time.

In addition, for a subproblem, we also want to check the base case for if the piece is "pure" (contains only clean paper, or contains only bitten paper). Brute force checking this takes $O(mn)$ time, for a total subproblem time of $O(mn + (m + n)) \to O(mn)$.

However, this $O(mn)$ factor per subproblem can be reduced to $O(m + n)$ (this is not required to receive full points). We can precompute the purities of every single possible subrectangle and store it in a table. Brute-force performs the pre-computation in $O(m^3n^3)$ time, but using prefix sums allows us to do this in just $O(mn)$ time. So to solve our recurrence relation, if we can determine purity/impurity in $O(1)$ time (after doing some pre-computation), then we can reach an overall time of $O((m + n)m^2n^2)$.

Alternatively, we can initialize all min-cut values of single square pieces to be 0. Then, if it is possible to have some cut such that both resulting pieces have min-cut values of 0, and both resulting pieces are of the same type (clean-only or bitten-only, and we can take any sample of either and compare them), then we ourself are a pure piece. This would allow you to avoid the entire pre-computation business as mentioned before, and still achieve a runtime of $O((m + n)m^2n^2)$.

(e) **Space Complexity Analysis:** we have to store the entire DP array for our recurrence relation to work, so the space complexity is $O(m^2n^2)$.

6 Coding Questions

For this week's coding questions, we'll implement dynamic programming algorithms to solve two classic problems: **Longest Increasing Subsequence** and **Edit Distance**. There are two ways that you can access the notebook and complete the problems:

1. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   https://github.com/Berkeley-CS170/cs170-fa23-coding

   and navigate to the `hw07` folder. Refer to the `README.md` for local setup instructions.

2. **On Datahub**: Click here and navigate to the `hw07` folder if you prefer to complete this question on Berkeley DataHub.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the Gradescope assignment titled "Homework 7 Coding Portion".
- *OH/HWP Instructions:* Designated coding course staff will provide conceptual and debugging help during office hours and homework parties.
- *Edstem Instructions:* Conceptual questions are always welcome on the public thread. If you need debugging help first try asking on the public threads. To ensure others can help you, make sure to:

  1. Describe the steps you've taken to debug the issue prior to posting on Ed.
  2. Describe the specific error you're running into.
  3. Include a few small test cases, alongside both the output you expected to receive and your function's actual output.

  If staff tells you to make a private Ed post, make sure to include *all of the above items* plus your full function implementation. If you don't provide them, we will ask you to provide them.

- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.

lis_edit_dist

October 15, 2023

1 Dynamic Programming

In this notebook, we'll explore solving the Longest Increasing Subsequence problem and the Global Alignment problem using dynamic programming.

If you're using Datahub:

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

If you're running locally:

- 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**

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

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

grader = otter.Notebook("lis_edit_dist.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
```

1.0.1 Q1. Longest Increasing Subsequence

First implement the longest increasing subsequence. The algorithm is explained here https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf#page=3.

The algorithm discussed in lecture and the textbook only returns the length of the longest increasing subsequence. Here we want you to return the actual subsequence (the actual list of elements). To

find the actual subsequence, it may be useful to maintain an array seperate from the dp array which can be used to reconstruct the actual sequence.

[ ]:
```
def longest_increasing_subsequence (arr, n):
    """
    Return a list containing longest increasing subsequence of the array.
    If there are ties, return any one of them.

    args:
        arr:List[int] = an array of integers
        n:int = an int representing the length of arr

    return:
        List[int] Containing the longest increasing subsequence. Return the␣
 ↪actual
        elements, not the indices of the elements.
    """
    # BEGIN SOLUTION

    # compute the DP array
    dp = [1]*n
    prev = [i for i in range(n)]
    for i in range(n):
        for j in range(i):
            if arr[i] > arr[j] and dp[i] <= dp[j]:
                dp[i] = dp[j] + 1
                prev[i] = j

    # reconstruct the sequence
    end = dp.index(max(dp))
    sequence = []
    while prev[end] != end:
        sequence.append(arr[end])
        end = prev[end]
    sequence.append(arr[end])
    sequence.reverse()

    return sequence
    # END SOLUTION
```

Note: your solution should not take inordinate amounts of time to run. If it takes more than 10 seconds to run, it is too slow

We will also check submissions for hardcoded solutions.

Points: 2

[ ]:
```
grader.check("LIS")
```

1.0.2 Debugging

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.

[ ]:
```
def test_LIS(all_arrs, all_sols, longest_increasing_subsequence, tqdm, random):
    def check_subsequence(seq, arr):
        for i in range(len(seq) - 1):
            assert seq[i] < seq[i + 1], f"Your subsequence is not strictly␣
↪increasing: {seq}"

        index = 0
        matched = 0
        while matched < len(seq) and index < len(arr):
            if seq[matched] == arr[index]:
                matched += 1
            index += 1
        assert matched == len(seq), f"your list is not a valid subsequence of␣
↪the input list."
    assert all_arrs is not None
    assert all_sols is not None
    assert tqdm is not None

    for arr, sol in tqdm.tqdm(zip(all_arrs, all_sols), total=len(all_arrs)):
        student_sol = longest_increasing_subsequence(arr, len(arr))

        assert len(student_sol) == len(sol), f"""The length of your list␣
↪differs from the solution. Your list {student_sol}, the solution {sol}"""
        check_subsequence(student_sol, arr)
```

```
100%|
```

```
          | 358/358 [00:01<00:00, 323.26it/s]
```

1.1 Global Alignment

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.

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

[ ]:
```
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)`.

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

[ ]:
```
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)`.

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

[ ]:
```
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)`.

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

[ ]:
```
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`

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

[ ]:
```
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`

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

[ ]:
```
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)

[ ]:
```
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 in the
        tuple corresponds to x and the second to y. Use hypen's '-' to␣
↪represent gaps in each string.
    """
    # 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

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

1.2.1 Debugging

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.

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

    assert len(aligned) >= len(original), "your function returned a string␣
 ↪which is shorter than the input string!"
    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 produced by only adding gaps!"
        i += 1
        j += 1
    while j < len(aligned):
        assert aligned[j] == '-', "your function returned a string which cannot␣
 ↪be produced by only adding gaps!"
        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 same
        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}, but your
        strings have a distance of {dist}."""
finish = time.time()
assert finish - start < 60, "your solution timed out"
```

```
100%|
          | 200/200 [00:19<00:00, 10.02it/s]
```

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.

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