CS 170 Homework 7

Due **3/11/2024, at 10:00 pm (grace period until 11:59pm)**

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".

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.

2 Not Too Much DP

(a) Given an array A with positive or negative integers (i.e. non-zero), we want to find the subarray (i.e. contiguous sequence of elements in the array) that creates the maximum product. We will use 1-dimensional DP to approach this problem where $dp[i]$ will return the maximum product and minimum product seen so far using elements up to (and including) i-th index from the array. Notice here we need to keep track of the minimum product as well; in case the next element in the array is negative, the minimum product might become the maximum product after new multiplication.

Given the DP subproblems below, perform the following:

0	1	2	3	4	5
$(-2, -2)$	$(3, -6)$	$(24, -12)$	$(48, -24)$	$(24, -48)$	$(96, -192)$

i. Recover the original array.

Solution: The original array will be [-2, 3, -4, 2, -1, 4]. $dp[0]$ tells us the first element of the array is -2. Then $dp[1]$ has minimum product = -6, which can only be obtained if the second element in the array is 3. Then we use similar reasoning to figure out the rest of the array. Notice throughout the process, some maximum product comes from multiplying a negative number to the previous minimum product.

ii. Identify the subarray that produces the maximum product.

Solution: [3, -4, 2, -1, 4]. Once the original array is recovered, you can figure out the subarray either by looking at the array or backtracking the way maximum product is built throughout the process. In this case we can see the overall maximum is 96, so we start the backtracking at $dp[5]$ and 4 should be included in the subarray. 96 is obtained by multiplying 4 to 24 which comes from $dp[4]$ so -1 should also be included. We go until $dp[0]$ where we realize the 3 does not come from $dp[0]$ so that means the element at index 0 is not included in the subarray.

(b) Given strings s_1, s_2, and s_3, find whether s_3 can be formed by an interleaving of s_1 and s_2. s_3 is said to be an interleaving of s_1 and s_2 if s_3 contains all of the characters of s_1 and s_2 and only those characters. Additionally, the order of the characters of s_1 and s_2 are preserved.

More formally, an interleaving of two strings a and b is a configuration where a and b are divided into n and m substrings respectively, such that:

- $a = a_1 + a_2 + \cdots + a_n$
- $b = b_1 + b_2 + \cdots + b_m$
- $|n - m| \leq 1$

The interleaving is

$$a_1 + b_1 + a_2 + b_2 + a_3 + b_3 + \cdots \text{ or } b_1 + a_1 + b_2 + a_2 + b_3 + a_3 + \cdots$$

Let l_1, l_2, l_3 be the lengths of s_1, s_2 and s_3 respectively. We use 2-dimensional DP to approach this problem where $dp[i][j] = True$ if the substring $s_3[i+j : l_3]$ is an interleaving of substrings $s_1[i : l_1]$ and $s_2[j : l_2]$, and False otherwise.

Assume for all subparts that s_1 correspond to rows and s_2 correspond to columns of the table.

(i) For this subpart, let s_1 = "cbadb", s_2 = "badda", s_3 = "cbbadadadb". Using those inputs, fill in the missing grids in the following table:

-	0	1	2	3	4	5
0	T	T	T	F	F	F
1	F	T	T	F	F	F
2	F	F	T	F	F	?
3	F	F	T	?	F	F
4	F	F	?	T	?	?
5	F	F	?	T	?	T

Notice here we included an extra row and column to account for cases where we used up one or both of the strings.

Solution:

-	0	1	2	3	4	5
0	T	T	T	F	F	F
1	F	T	T	F	F	F
2	F	F	T	F	F	F
3	F	F	T	T	F	F
4	F	F	F	T	F	F
5	F	F	T	T	T	T

(ii) For this subpart, you are given s_3 = "sponpdaens" and part of the DP table. Using those information, recover s_1 and s_2. (Note there might be multiple s_1, s_2 combinations that will produce the same table. If multiple combinations are possible, list out all of them.)

-	0	1	2	3	4	5
0	T	-	-	-	-	-
1	-	T	-	-	-	-
2	-	-	T	F	F	-
3	-	-	-	-	T	-
4	-	-	-	-	T	F
5	-	-	-	-	-	T

Solution: One of the key things here is for students to understand the subproblems and what they represent. For example, dp[4][5] represents whether it's possible to form the last character in s_3 using the last character in s_1. dp[5][5] = T tells us s_3 can be interleaved by s_1 and s_2. p[4][5] = F tells us that the last character of s_2 is s since s must come from either s_1 or s_2, if it doesn't come from

s_1, it must then come from s_2. Using similar logic, dp[4][4] = T tells us last character in s_1 is n. dp[3][4] = T shows second to last character in s_1 is e. dp[2][4] = F shows a is the second to last character in s_2. dp[2][3] = F shows third character in s_2 is d. dp[2][2] = T shows third character in s_1 is p. dp[1][1] and dp[0][0] can only tell us the first two characters in s_1/s_2 is either s / p or o / n, but with limited information we cannot be certain which one belongs to which. So we have 2 * 2 = 4 possible combinations of s_1 and s_2. And the 4 combinations will be:

1. s_1 = "sopen", s_2 = "pndas"
2. s_1 = "snpen", s_2 = "podas"
3. s_1 = "popen", s_2 = "sndas"
4. s_1 = "pnpen", s_2 = "sodas"

(iii) For this subpart, determine whether the following subtables are possible (subtable is simply a small part of the entire table). Give a brief justification/reasoning to your answer.

1.

T	T
T	T

Solution: Possible. As long as s_1 and s_2 share the same character and s_3 has that shared character repeated at least two times.

2.

T	F
F	T

Solution: Not possible. Because upper left and bottom right are both True means that the two characters involved can be interleaved successfully. Thus, the latter character must come from one of the two strings.

3.

T	F
T	F

Solution: Possible. Happens when two consecutive characters both come from the same string (either s_1 or s_2)

3 Egg Drop

You are given m identical eggs and an n story building. You need to figure out the highest floor $h \in \{0, 1, 2, \ldots n\}$ that you can drop an egg from without breaking it. Each egg will never break when dropped from floor h or lower, and always breaks if dropped from floor $h+1$ or higher. ($h = 0$ 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 h (regardless of the value of h).

(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 h 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 h 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 h.
- $f(n, 1) = n$, 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 $O(nm)$

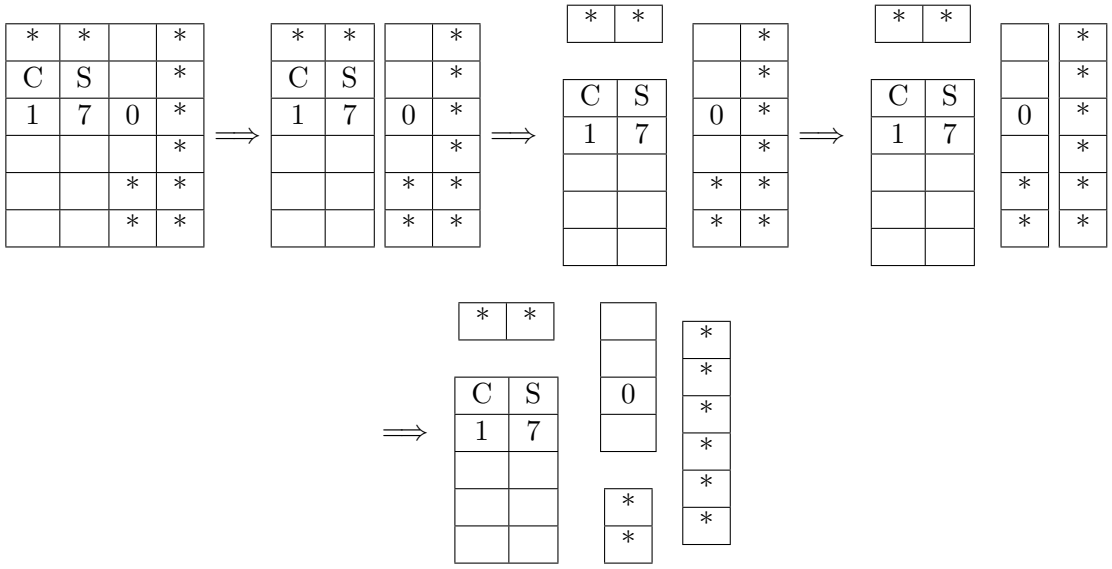
(g) Yes, it is possible! Notice that in our recurrence relation in part (c), we only need the values of $f(\cdot, m)$ and $f(\cdot, m-1)$. So we can just store the last two "columns" computed so far. The pseudocode for this would like as follows:

```
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]
```

4 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 $P[i, j]$ such that $P[i, j] = 1$ if and only if the ij^{th} square has holes bitten into it.
>
> **Goal:** Find the minimum number of cuts needed so that the $P[i, j]$ 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 $P[i_1 \leq i_2, j_1 \leq j_2]$ into pieces consisting either entirely of bitten pieces or clean pieces.

(b) **Recurrence Relation:**

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

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 $O(m^2n^2)$.

5 Knightmare

Give a dynamic programming algorithm to find the number of ways you can place knights on an L by H ($L < H$) chessboard such that no two knights can attack each other (there can be any number of knights on the board, including zero knights). Knights can move in a 2×1 shape pattern in any direction.

Provide a 4-part solution. Your algorithm's runtime should be $O(2^{3L}LH)$, and return your answer mod 1773.

Hint: if a knight is on row i, what rows on the chessboard can it affect?

Solution: The first part of this solution is the old 3-part format (with full explanation). The 4-part format is below it.

3-part solution:

Algorithm: We use length L bit strings to represent the configuration of rows of the chessboard (1 means there is a knight in the corresponding square and otherwise 0).

The main idea of the algorithm is as follows: we solve the subproblem of the number of valid configurations of $(h-1) \times L$ chessboard and use it to solve the $h \times L$ case. Note that as we iteratively incrementing h, a knight in the h-th rows can only affect configurations of rows $h+1$ and $h+2$. So we can denote $K(h, u, v)$ as the number of possible configurations of the first h rows with u being the $(h-1)$-th row and v being the h-th row, and then use dynamic programming to solve this problem.

Let a list of bitstrings be valid if placing the knights in the first row according to the first bitstring, in the second row according to the second bitstring, etc. doesn't cause two knights to attack each other. Then we have $K(2, u, v) = 1$ if u, v are valid and 0 otherwise for all u, v pairs.

For $K(h, v, w)$ we have:

$$K(h, v, w) = \sum_{u : u,v,w \text{ are valid}} K(h-1, u, v) \bmod 1773$$

Proof of Correctness: The only 2-row configuration of knights ending in row configurations u, v is the configuration u, v itself. So $K(2, v, w) = 1$ if u, v are valid. Otherwise, $K(2, v, w) = 0$.

For $h > 2$, the above recurrence is correct because for any valid h-row configuration ending in v, w, the first $h-1$ rows must be a valid configuration ending in u, v for some u, and for this same u, the last three rows u, v, w must be also valid configuration. Moreover, this correspondence between h-row and $(h-1)$-row configurations is bijective.

Runtime Analysis: To bound the total runtime, first note that for each row, we iterate over all possible configurations of knights in the row and for each such configuration, we perform a sum over all possible valid configurations of the previous two rows. The time to check if a configuration is valid is $O(L)$. Therefore, the time taken to compute the sub-problems for a single row is $O(2^{3L}L)$ which gives us an overall runtime of $O(2^{3L}LH)$ operations. Although there are a potentially exponential number of possible configurations (at most 2^{HL}), we are only interested in the answer mod 1773, which means that all of our numbers are constant size, and so arithmetic on these numbers is constant time. This gives us a final runtime of $O(2^{3L}LH)$.

4-part solution:

(a) **Subproblems:** $f(h, u, v)$ is the number of possible valid configurations mod 1773 of the first h rows with u being the configuration of the $(h-1)$-th row and v being the configuration of the h-th row.

(b) **Recurrence and Base Cases:** For $h > 2$, $f(h, v, w) = \sum_{u:u,v,w \text{ are valid}} f(h-1, u, v)$ mod 1773. For the base case, $f(2, u, v) = 1$ if u, v are valid and 0 otherwise for all u, v pairs. No other base cases are needed.

(c) **Proof of Correctness:** See proof of correctness in 3-part solution above.

(d) **Runtime and Space Complexity:** See runtime above to yield $O(2^{3L}LH)$. For space complexity, note that there are only $O(H2^{2L})$ subproblems (H rows, 2^L settings to the $(H-1)$th row, and 2^L settings to the Hth row). Each subproblem is a number of possible configurations mod 1773, bounded above by 1773, so our total space is $O(2^{2L}H)$. Note that since each subproblem only depends on subproblems of the previous row, we could reduce this by a factor of H to $O(2^{2L})$ by recycling space from earlier rows.

6 [Coding] Edit Distance

For this week's coding questions, we'll implement the Edit Distance algorithm you saw in lecture. There are two ways that you can access the notebook and complete the problems:

1. **On Datahub**: click here and navigate to the `hw07` folder.
2. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-sp24-coding`

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

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the Gradescope assignment titled "Homework 7 Coding Portion".
- *Getting Help:* Conceptual questions are always welcome on Edstem and office hours; *note that support for debugging help during OH will be limited.* If you need debugging help first try asking on the public Edstem 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 but nontrivial 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.

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)
```