CS 170 Homework 7

Due **Monday 10/21/2024, 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, write "none".

DP Solution Writing Guidelines:

When writing your solutions to DP problems only, please follow the **4-part solution** template below (unless specified otherwise):

1. Define a function $f(\cdot)$ in words. Make sure to include how many parameters there are and what they mean, and tell us what inputs you feed into f to get the answer to your problem.
2. Write the recurrence relation for f, as well as the "base cases".
3. Prove that the recurrence correctly solves the problem. In almost all cases, you will want to use induction to prove the correctness of a DP algorithm.
4. Analyze the runtime and space complexity of your final DP algorithm. Note that the top-down and bottom-up approaches to DP have the same runtime complexity; however, bottom-up can potentially yield a better space complexity.

2 Egg Drop

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

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

Hint: use ∞ to denote that it is impossible to find b.

(b) Consider dropping an egg at floor h 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 b in terms of $f(\cdot, \cdot)$, n, m, and/or h.

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

(c) The recurrence relation is

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

When we drop an egg at floor h, in the worst case, we need $\max\{f(h-1,m-1), f(n-h,m)\}$ drops. Then, the optimal strategy will choose the best of the n floors, so we need $\min_{h\in\{1\ldots n\}} \max\{f(h-1,m-1), f(n-h,m)\}$ 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 look approximately 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[h-1], curr[i-h])
                                for h in range(1, i+1)
                            ])

    return curr[n]
```

3 Counting Targets

We call a sequence of n integers $x_1, \ldots, x_n$ *valid* if each x_i is in $\{1, \ldots, m\}$.

(a) Give a dynamic programming-based algorithm that takes in n, m and "target" T as input and outputs the number of distinct valid sequences such that $x_1 + \cdots + x_n = T$. Your algorithm should run in time $O(m^2n^2)$. Note that you can assume $T \leq mn$ since no valid sequences sum to more than mn.

Please provide a 4-part solution.

(b) **(Extra Credit)** Give an algorithm for the problem in part (a) that runs in time $O(mn^2)$.

Please provide the subproblem definition, recurrence relation (including base cases), and the runtime/space complexity analyses. You do not need to provide a proof of correctness.

Solution:

(a) We use $f(s, i)$ to denote the number of sequences of length i with sum s. $f(s, i)$ is 0 when $i > 0$ and $s \leq 0$, and $f(s, 1)$ is 1 if $1 \leq s \leq m$. Otherwise it satisfies the recurrence:

$$f(s, i) = \sum_{j=1}^{m} f(s - j, i - 1)$$

There are a total of mn^2 subproblems since $T \leq mn$ and it takes $O(m)$ time to compute $f(s, i)$ from its subproblems, which leads to an $O(m^2n^2)$ DP algorithm. Our algorithm outputs $f(T, n)$.

(b) Manipulating the recurrence from the previous part, we get:

$$f(s, i) = \sum_{j=1}^{m} f(s - j, i - 1) = \sum_{j=0}^{m-1} f(s - 1 - j, i - 1) =$$

$$\left[\sum_{j=1}^{m} f(s - 1 - j, i - 1)\right] + f(s - 1, i - 1) - f(s - m - 1, i - 1) =$$

$$f(s - 1, i) + f(s - 1, i - 1) - f(s - m - 1, i - 1).$$

In other words, we're now exploiting the fact that the sums for $f(s, i)$ and $f(s - 1, i)$ differ by two terms, and so rather than recompute $f(s, i)$ from scratch we can just add/subtract these terms from $f(s - 1, i)$

Using this recurrence, there are still mn^2 subproblems, but it takes $O(1)$ time to compute $f(s, i)$ from its subproblems, and thus there is a $O(mn^2)$ time DP algorithm.

4 String Shuffling

Let x, y, and z be strings. We want to know if z can be obtained only from x and y by interleaving the characters from x and y such that the characters in x appear in order and the characters in y appear in order.

For example, if $x =$ **prasad** and $y =$ **SANJAM**, then it is true for $z =$ **praSANsadJAM**, but false for $z =$ **prasadSANJAMpog** (extra characters), $z =$ **prasSANJAad** (missing the final **M**), and $z =$ **prasadASNJAM** (out of order). How can we answer this query efficiently? Your answer must be able to efficiently deal with strings with lots of overlap, such as $x =$ **aaaaaaaaaab** and $y =$ **aaaaaaaac**.

(a) Design an efficient algorithm to solve the above problem, briefly justify its correctness, and analyze its runtime. You do *not* need to provide a space complexity analysis (you'll do this in the next part!).

(b) Consider a bottom-up approach to our DP algorithm in part (a). Naively if we want to keep track of every solved sub-problem, this requires $O(|x||y|)$ space (double check to see if you understand why this is the case). How can we reduce the amount of space our algorithm uses?

Solution:

(a) First, we note that we must have $|z| = |x| + |y|$, so we can assume this. Let $S(i, j)$ be true if and only if the first i characters of x and the first j characters of y can be interleaved to make the first $i + j$ characters of z. Then x and y can be interleaved to make z if and only if $S(|x|, |y|)$ is true.

For the recurrence, if $S(i, j)$ is true then either $z_{i+j} = x_i$, $z_{i+j} = y_j$, or both. In the first case it must be that the first $i - 1$ characters of x and the first j characters of y can be interleaved to make the first $i + j - 1$ characters of z; that is, $S(i - 1, j)$ must be true. In the second case $S(i, j - 1)$ must be true. In the third case we can have either $S(i - 1, j)$ or $S(i, j - 1)$ or both being true. This yields the recurrence:

$$S(i, j) = (S(i - 1, j) \wedge (x_i = z_{i+j})) \vee (S(i, j - 1) \wedge (y_j = z_{i+j}))$$

The base case is $S(0, 0) = T$; we also set $\forall i \in [0, |x|], S(i, -1) = F$ and $\forall j \in [0, |y|], S(-1, j) = F$. The running time is $O(|x||y|)$.

Somewhat naively if we'd like an iterative solution, we can keep track of the solutions to all subproblems with a 2D array where the entry at row i, column j is $S(i, j)$. If we iterate over this array row by row, going left to right, we'll always be able to fill in the next entry using values we've already computed.

Notice, however, that to compute any entry, we only really need the infomration in the previous row, and the current row we're filling out. Thus, rather than holding onto the entire table, we only need to store the current and previous row, reducing us from $O(m * n)$ space to $O(m)$ space.

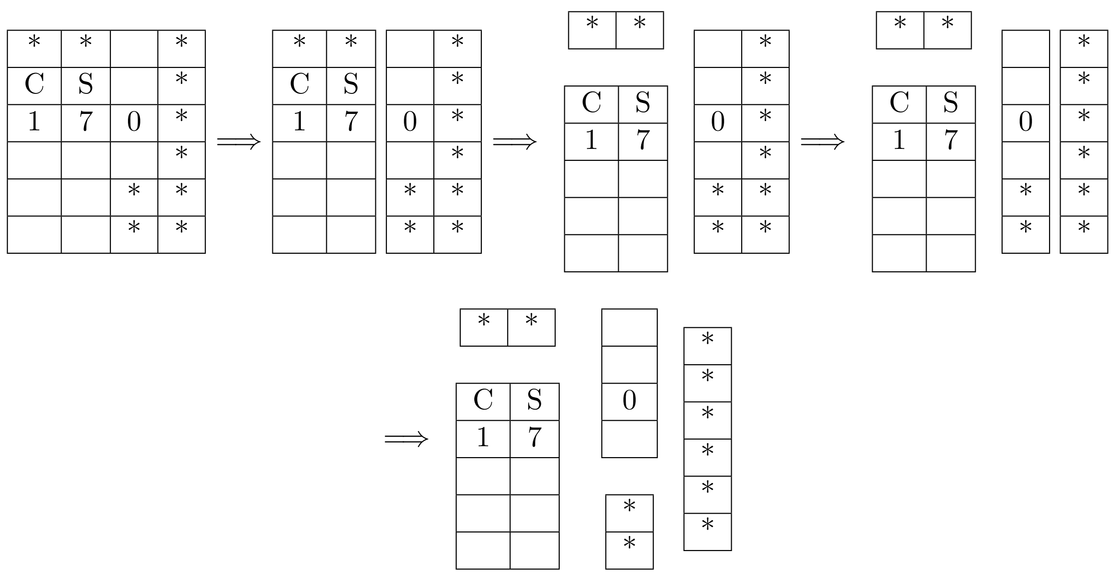
(b) We can keep track of the solutions to all subproblems with a 2D array of size $|x||y|$ where the entry at row i, column j is $S(i, j)$. If we iterate over this array row by row, going left to right, we'll always be able to fill in the next entry using values we've already computed.

Notice, however, that to compute any entry, we only really need the information in the previous row, and the current row we're filling out. Thus, rather than holding onto the entire table, we only need to store the current and previous row, reducing from $O(|x||y|)$ space to $O(\min(|x|, |y|))$ space.

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 $x \times y$ 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.)

Formally, the problem is as follows:

> **Input:** Dimensions of the paper $x \times y$ 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.

Design a DP-based algorithm to find the smallest number of cuts needed to separate all the bitten parts out in $O(x^3y^3)$ time. For **extra credit**, try to optimize your algorithm to achieve a runtime of $O((x + y)x^2y^2)$.

(a) Define your subproblem.

Hint: try making any arbitrary cut. What two subproblems do you now have? What parameters do you need to properly handle recursing on these two resulting subproblems?

(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 $P[i_1 \le i_2, j_1 \le j_2]$ 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: $O((x + y)x^2y^2)$ and $O(x^3y^3)$

We have $O(x^2y^2)$ total subproblems: $O(xy)$ possibilities for (i_1, j_1), and $O(mn)$ possibilities for (i_2, j_2). For each subproblem, we examine up to x possible choices for horizontal splits, and y 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 $O(x + y)$ 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 $O(xy)$ time, for a total subproblem time of $O(xy + (x + y)) = O(xy)$.

Thus, the overall (accepted) runtime is $O(x^2y^2) \cdot O(xy) = O(x^3y^3)$.

However, this $O(xy)$ factor per subproblem can be reduced to $O(x + y)$ (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 $O(x^3y^3)$ time, but using prefix sums allows us to do this in just $O(xy)$ 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 $O((x + y)x^2y^2)$.

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 $O((x+y)x^2y^2)$.

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

6 [Coding] More Advanced Dynamic Programming

For this week's homework, you'll implement implement some more dynamic programming algorithms. 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-fa24-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.

hw07

October 25, 2024

1 Dynamic Programming (Part 2!)

In this notebook, we'll implement the Dynamic Programming algorithms for Edit Distance and Traveling Salesperson that we saw in class.

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=3.11` * 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**

[31]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

[2]:
```
import otter
assert (otter.__version__ >= "5.5.0"), "Please reinstall the requirements and␣
 ↪restart your kernel."

grader = otter.Notebook("hw07.ipynb")
import numpy.random as random
from networkx import Graph, draw
import string
import pylev
import tqdm
import time
import pickle
```

```
from autograder_utils import validate_tour, handle_timeout, profile

test_data = pickle.load(open("public_data.pkl", "rb"))

rng_seed = 42
```

1.1 Q1: Edit Distance (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 Q1.0 (Optional): 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 Q1.1 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.

[33]:
```
node = (5,6) # SOLUTION
```

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

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)`.

[35]:
```
ans = [2,3] # SOLUTION
```

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

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)`.

[37]:
```
ans = [1,4] # SOLUTION
```

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

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)`.

[39]:
```
ans = [3,4] # SOLUTION
```

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

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`

[41]:
```
ans = [1,3] # SOLUTION
```

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

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`

[43]:
```
ans = [2] # SOLUTION
```

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

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.

1.1.2 Q1.1 Edit Distance (Graded)

Now, implement the `edit_distance` function itself! This is the only part of Q1 that will be graded.

[45]:
```
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.

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

1.1.3 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.

You can roughly split this task into two parts: finding the optimal edit distance and then reconstructing the best possible alignment. When debugging, might want to first ensure that your edit distances are correct before checking your alignment code. Additionally, when debugging the alignment, you might want to consider working through some small examples by hand - the examples from the textbook are a great place to start, as well as any examples that might invoke particular edge cases.

[46]:
```
rng = random.default_rng(rng_seed)

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 = list(string.ascii_lowercase)
MIN_WORD_SIZE = 250
MAX_WORD_SIZE = 500

for i in tqdm.tqdm(range(NUM_TRIALS)):
    word1_size, word2_size = rng.integers(MIN_WORD_SIZE, MAX_WORD_SIZE, size=2)
    word1 = ''.join(rng.choice(LETTERS, size=word1_size))
    word2 = ''.join(rng.choice(LETTERS, size=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 [00:08<00:00, 23.26it/s]
```

1.2 Q2) Traveling Salesperson DP

Now, we'll implement the dynamic programming algorithm for the traveling salesperson problem (TSP). A brute force solution will be hopelessly slow even for moderate-sized test cases, but we can use dynamic programming to get a solution in slightly more reasonable (but still exponential) time. For a refresher on the TSP algorithm, see Lecture 12 or https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf#page=20.

As with previous problems, we want you to return the actual tour, not the cost of the tour. We can once again apply the same procedure of backtracking through our subproblem array to reconstruct this tour.

1.2.1 Representing Subproblems

If we use a set as one of our subproblem parameters, we can't directly use a 2D array to store our subproblems. There are two common ways to work around this issue:

1. Subproblem Dictionary You could store subproblems in a dictionary, where the keys are tuples of the form `(S, i)`, where `i` represents the last city visited before returning home and `S` is the set of cities visited so far.

To make this work, you need to ensure that the keys are hashable. One way is using Python's built-in `frozenset` class for `S`. `frozenset` is a built-in type so you can use it without any additional imports, and works just like a normal set, except that it is immutable (and hashable). You can read more about `frozenset` here: https://docs.python.org/3/library/stdtypes.html#frozenset.

2. Bitmasking Instead of a hash set, we actually *can* still use a 2D array to store subproblems, where `S` is represented as an n-bit unsigned integer, and the i-th bit of `S` would be set to 1 if and only if the i-th city is part of the set of visited cities. Since `S` is an integer, we can use it to index into our 2D array.

The bitmasking approach tends to be about twice as fast and much more memory-efficient than the `frozenset` approach, but both approaches will pass the autograder if implemented correctly.

1.2.2 Implementation tip:

As with before, storing the entire tour at each step is too memory-intensive and will cause the autograder to fail. Instead, consider maintaining a separate dictionary or array which stores a smaller amount of information but can still help you reconstruct the tour (can the "shortest path in the DP DAG" idea help here?).

Be careful with indexing! The algorithm from the book assumes your cities are labeled $1, \ldots, n$ - if you are indexing into a Python list, will you need to adjust your indices?

Be careful with subproblem ordering! We need to ensure that whenever we go to solve a subproblem, all of the subproblems it depends on have already been solved.

The graph is not necessarily complete! If no tour is possible, return an empty list.

1.2.3 Graph helpers

Like the last homeworks, we use a weighted adjacency list to represent the graph. We'll use a similar format as before, except `graph[u]` is now a hashmap instead of a list of pairs. **For this assignment, graphs are undirected**, so if there is an (undirected) edge between nodes `u` and `v` with weight `w`, then `graph[u]` contains key `v` with value `w` and `graph[v]` contains key `u` with value `w`.

We provide the following code to help you test your implementation.

[3]:
```
def generate_adj_list(n, edge_list: list[tuple[int]]) -> list[dict[int, int]]:
    """
    args:
        n:int = number of nodes in the graph. The nodes are labelled with␣
↪integers 0 through n-1
```

```
        edge_list:list[tuple[int,int,int]] = edge list where each tuple (u,v,w)␣
↪represents the
        undirected and weighted edge (u,v,w) in the graph
    return:
        A list[dict[int, int]] representing the adjacency list
    """
    adj_list = [{} for _ in range(n)]
    for u, v, w in sorted(edge_list):
        adj_list[u][v] = w
        # undirected edges
        adj_list[v][u] = w
    return adj_list

def draw_graph(adj_list: list[dict[int, int]]):
    """Utility method for visualizing graphs

    args:
        adj_list: list[dict[int, int]] = adjacency list representation of the␣
↪graph generated by generate_adj_list
    """
    G = Graph()
    for u in range(len(adj_list)):
        for v, w in adj_list[u]:
            G.add_edge(u, v, weight=w)
    draw(G, with_labels=True)
```

[31]:
```
def tsp_dp(adj_list):
    """Compute the exact solution to the TSP using dynamic programming and␣
↪returns the optimal path.

    Args:
        dist_arr: Weighted undirected graph represented as an adjacency list.

    Returns:
        List[int]: A list of city indices representing the optimal path.
    """
    # BEGIN SOLUTION
    n = len(adj_list)
    if n == 0: return []

    MAX_DIST = n * max(w for neighbors in adj_list for w in neighbors.values())␣
↪+ 1

    dp = [[MAX_DIST] * n for _ in range(1 << n)]
    prev = [[MAX_DIST] * n for _ in range(1 << n)]

    def generate_subsets(n, k):
```

```
        def backtrack(start, curr, l):
            if l == k:
                yield curr
                return
            for i in range(start, n):
                curr |= (1 << i)
                yield from backtrack(i + 1, curr, l + 1)
                curr &= ~(1 << i)

        yield from backtrack(0, 0, 0)

    dp[1][0] = 0
    for size in range(2, n+1):
        for ss in generate_subsets(n-1, size-1):
            S = ss << 1 | 1
            for j in range(1, n):
                if not (S & (1 << j)): # ensure j is in the set
                    continue
                for k, w in adj_list[j].items():
                    if k == j or not (S & (1 << k)):
                        continue
                    if dp[S][j] >= dp[S ^ (1 << j)][k] + w:
                        dp[S][j] = dp[S ^ (1 << j)][k] + w
                        prev[S][j] = k

    # Backtracking to reconstruct tour
    tour = []
    S = (1 << n) - 1
    opt_dist, city = min((dp[S][j] + w, j) for j, w in adj_list[0].items())
    if opt_dist >= MAX_DIST:
        return []
    # print(prev)
    while S:
        # print(bin(S), city, prev[S][city])
        tour.append(city)
        S ^= (1 << city)
        city = prev[S | 1 << city][city]

    return tour
    # END SOLUTION

# BEGIN SOLUTION NO PROMPT
# Alternate solution using recursion and memoization and frozensets
def tsp_dp_alt(adj_list):
    n = len(adj_list)
    dp = {}
    prev = {}
```

```
    def tsp_helper(S, i):
        if (S, i) in dp:
            return dp[(S, i)]

        if S == frozenset():
            if i in adj_list[0]:
                return adj_list[0][i]
            return float('inf')

        min_cost = float('inf')
        prev_city = n + 1
        for city in S:
            if i not in adj_list[city]:
                continue
            cost = adj_list[i][city] + tsp_helper(S.difference({city}), city)
            min_cost, prev_city = min((min_cost, prev_city) , (cost, city))

        dp[(S, i)] = min_cost
        prev[(S, i)] = prev_city
        return min_cost

    best_distance = tsp_helper(frozenset(range(1, n)), 0)
    if best_distance == float('inf'):
        return []

    # Backtracking to reconstruct tour
    S = frozenset(range(1, n))
    city = 0  # start at the origin
    tour = [0]

    # print(prev)

    while S:
        city = prev[(S, city)]
        tour.append(city)
        S = S.difference({city})

    return tour
# END SOLUTION
```

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

1.2.4 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.

[22]:
```
# tests on very small cases
for adj_list, expected_distance in tqdm.tqdm(test_data['TSP-1']):
    result = tsp_dp(adj_list)

    if expected_distance < 0:
        # no tour is possible
        assert result == [], "You returned a tour when no tour is possible"
    else:
        assert set(result) == set(range(len(adj_list))), f"Your output does not␣
↪visit all cities"
        student_length = validate_tour(result, adj_list)
        assert student_length >= 0, f"Your output is not a valid tour"
        assert student_length == expected_distance, f"Your output is not a␣
↪minimum distance tour"
```

```
100%|| 20/20 [00:01<00:00, 13.76it/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)
```