Due October 20 (11:59 p.m.)

Instructions

- **Answers should be written in the boxes by modifying the provided Latex template or some other method answers are clearly marked and legible. Submit via Gradescope.**
- **Honors questions are optional. They will not count towards your grade in the course. However you are encouraged to submit your solutions to these problems to receive feedback on your attempts.**
- **You must enter the names of your collaborators or other sources (including use of LLMs) as a response to Question 0. Do NOT leave this blank; if you worked on the homework entirely on your own, please write "None" here. Even though collaborations in groups of up to 3 people are encouraged, you are required to write your own solution.**
- **For the late submission policy, see the website.**
- Acknowledgment - Thank you to Amir, Daji, Oded, Peter and Rotem for help in setting the problem set.

1-0 List all your collaborators and sources: ($-\infty$ points if left blank)

tutor, tx

Problem 6-1 – Heaviest Common Subsequence (35 Points)

In this question, we consider a variant of the LCS problem called the Heaviest Common Subsequence (HCS). Here every letter σ of the alphabet is associated with a weight $w(\sigma) \in \mathbb{Z}^+$. Our goal is to find a *heaviest* (rather than the longest) common sub-sequence of two given strings $X[1, \ldots, n], Y[1, \ldots, m]$, that is, a common sub-sequence $\sigma_1 \ldots \sigma_k$ of X, Y that maximizes the weight $\sum_{i=1}^{k} w(\sigma_i)$.

1. (10 points) Let $H[i, j]$ denote the weight of the HCS of $X[1, \ldots, i], Y[1, \ldots, j]$. Complete the blank below to obtain a recurrence for $H[i, j]$.

Solution: Going to use the same recurrence given by the LCS problem but add the weight for each letter when they match. Then $H[i, j]$ is going to be the max total weight of a common subsequence of X and Y.

If $X[i] = Y[j]$, then any optimal common subsequence ending with that given letter has a weight that's equal to the optimal weight for $X[1..i-1], Y[1..j-1]$ plus the weight of the matched symbol. So: $H[i-1, j-1] + w(X[i])$. We're going to take the max of either subproblem because we need to exclude either $X[i]$ or $Y[j]$. So for base cases $H[0, j] = 0, (0 \leq j \leq m)$ and $H[i, 0] = 0, (0 \leq i \leq n)$, the recurrence will be:

$$\begin{cases} H[i-1, j-1] + w(X[i]) & \text{if } X[i] = Y[j] \\ \max(H[i-1, j], H[i, j-1]) & \text{if } X[i] \neq Y[j] \end{cases}$$

2. (13 points) Consider the following strings X, Y and weights w:

$X = [A, L, G, O, S]$ $\quad$ $Y = [F, L, A, G]$

Letter	A	F	G	L	O	S
w	1	2	3	3	2	3

.

Similar to the LCS algorithm that we saw in class, use the HCS recurrence you wrote in part 1 to fill out the following table H for input X, Y, w, computing the value of $H[i, j]$ for each $i = 0, \ldots, 5$ and $j = 0, \ldots, 4$. Leave some empty space in each cell to add arrows in the next part.

Solution:

i \ j	0	1	2	3	4
0	0	0	0	0	0
1	0	0 ↑	0 ↑	1 ↖	1←
2	0	0 ↑	3↖	3 ←	3←
3	0	0↑	3↑	3↑	6 ↖
4	0	0↑	3↑	3↑	6↑
5	0	0↑	3↑	3↑	6↑

3. (12 points) Add to your table from item 2 arrows (↑, ←, ↖) indicating how each cell was updated by your algorithm (just like we did in class for LCS). Use the arrows to find an HCS for this example.

Solution: (Write your HCS here, fill in the arrows above):

Follow arrows to bottom right box, given HCS of 6, subsequence of letters LG. (Recording matches in reverse, first G then L).

Problem 6-2 – Greedy Hermit Crabs (35 Points)

On the beach there are n hermit crabs of sizes $A[1, \ldots, n]$, and m shells of sizes $B[1, \ldots, m]$. We want to match as many crabs as possible with shells that can fit them: crab i can fit into shell j if $A[i] \leq B[j]$. The output should be an array $S[1, \ldots, n]$, where $S[i]$ is the index of the shell into which crab i is placed, or $*$ if we did not match crab i with a shell.

1. (15 points) Show that there is an optimal solution where the smallest crab is assigned the smallest shell that can fit them (assuming there is such a shell).

   (Hint: suppose you are given an assignment of crabs to shells that does not have this property. Can you modify it to obtain an assignment that does, and is at least as good?)

   Solution: So let's say that A[1] is the smallest crab and that there exists any shell that can fit this crab, meaning there is an optimal matching where the crab's assigned the smallest shell. Let k be the smallest-index shell, assuming $A[i] \leq B[k]$. The base case would be that the samllest crab is already assigned to the smallest shell, in which case, we're done. Otherwise, in the optimal soln, there are two options: either the smallest shell is unassigned or it's assigned to some other crab. If it's the first option, we can just assign crab 1 to this shell, leaving the other matches unchanged. But this'll increase the number of matches by 1, which'll change how optimal M is, so it can't happen. If the second option is the case, then the assumption is that some crab i fits in this smallest shell. Also assuming that the smallest crab is assigned to some shell j. This means that the shell currently assigned to this crab has to be greater than the smallest shell, $j > k$ and $A[1] \leq B[k] \leq B[j]$. We just have to swap the two assignments of shells, assigning crab 1 (smallest crab) to shell k and crab i to shell j. Therefore, in the optimal solution, every matching can be modified to an optimal matching where the samllest crab is assigned to the smallest shell.

2. (15 points) Now suppose that the arrays A, B are sorted in non-decreasing order. Describe a greedy algorithm that computes an optimal assignment of crabs to shells.

   Solution:

```
for i = 1..n: S[i] (* here)
i = 1, ptr for crabs
j = 1, ptr for shells


while i <= n and j <= m:
        if A[i] <= B[j]:
                S[i] = j, assigning shell j to crab i
               i +=1
                j = j + 1, shell j used, advance both
        else: j = j + 1, meaning shell too small for crab i and to move onto next shell
return S
```

3. (5 points) What is the asymptotic runtime of your algorithm, assuming again that A, B are already sorted?

Solution: Each pointer i and j only moving forward through crabs and shells so the loop does at most $n + m$ iterations. This results in a runtime of $O(n + m)$.

Honors problem: (***) Recall the Huffman Encoding problem shown in class - Prove Kraft's inequality, which asserts that for any collection of lengths $\ell_1, \ldots, \ell_n$, there is a (binary) prefix code where the encodings have lengths $\ell_1, \ldots, \ell_n$ if and only if $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$.

Solution:

Problem 6-3 – Fence Painting with Tom Sawyer (30 points)

Tom wants to paint a fence made out of N boards of different lengths given in an array $L[1, 2, \ldots, N]$, where $L[i]$ is the length of the i-th board. He has convinced $K \leq N$ equally talented neighborhood kids to paint contiguous sections of the fence, *in parallel*: each kid $p \in \{1, \ldots, K\}$ works on a contiguous section $s_p, s_p + 1 \ldots, f_p$. Each kid's section starts immediately after the preceding kid's. We could have $f_p = s_p - 1$, in which case kid p does not paint any boards. For example, if the fence consists of $N = 7$ boards and we have $K = 3$ kids, one option is to assign boards $\{1, 2, 3, 4\}$ to kid 1, board $\{5\}$ to kid 2, and boards $\{6, 7\}$ to kid 3. Another option is to assign no boards to kid 1, boards $\{1, 2, 3\}$ to kid 2, and boards $\{4, 5, 6, 7\}$ to kid 3.

The time T_p required for kid $p \in \{1, \ldots, K\}$ to complete their work is equal to the *total length* of the boards assigned to kid p. Since the kids work in parallel, the work is finished as soon as *all* the kids are finished. Our goal is to have the fence painted as soon as possible, that is, to minimize $\max_{p \in \{1,\ldots,K\}} T_p$. (In the first example above, the first kid will take $T_1 = L[1] + L[2] + L[3] + L[4]$ time, the second will take $T_2 = L[5]$ time, and the last will take $T_3 = L[6] + L[7]$ time. The time required to paint the entire fence will be $\max(T_1, T_2, T_3)$.)

In this problem we develop a dynamic programming algorithm to find the optimal way to paint the fence. We will use the following subproblems: for $i \in \{0, \ldots, N\}$ and $j \in \{1, \ldots, K\}$, let $m[i, j]$ denote the optimal time to paint boards $1, \ldots, i$ using j kids.

1. (10 points) Write a recurrence for $m[i, j]$. Include the base cases for the recurrence.

   (**Hint:** when $j > 1$, what are the possibilities for the work we assign to the last kid, j? And for each such possibility, how long will the fence take to paint if we use the other kids optimally?)

Solution:

$$\min_{0 \le k \le i} (\max(T(i-k), j-1), \sum_{m=-i-k}^{i} S[i] - S[r-1])$$

Base cases:
m[0,j] = 0 for all j >= 1
m[i,1] = $S[i] = \sum_{p=1}^{i} L[p]$ for all i >= 1

2. (10 points) Describe an algorithm that is given the board lengths $L[1, \ldots, N]$ and the number of kids K, and finds the optimal time to paint the entire fence. The running time of your algorithm should be $O(N^3K)$.

Solution:
```
S[0] = 0
for i = 1 to N:
    S[i] = S[i-1] + L[i] (prefix sums)
for j = 1 to K:
    m[0][j] = 0 (initialize table and parent ptr for reconstructing
for i = 1 to N:
    m[i][1] = S[i]
    parent[i][1] = 1 (last block starts at 1 when only 1 kid)
filling DP table now..
for j = 2 to K:
    for i = 1 to N:                 best = infinity
        bestr = 1
        for r = 1 to i:
            cost = max(m[r-1][j-1], S[i] - S[r-1])
            if cost < best:
                best = cost and bestr = r
        m[i][j] = best
        parent[i][j] = bestr
return m, parent, need parent for reconstructing partition, would have to follow parent backwards
```

3. (10 points) Explain how to improve your algorithm from the previous part, so that its running time is only $O(N^2K)$.

Solution: The prev algo runs asymptotically in $O(N^3K)$. To optimize down to $O(N^2K)$, we would have to calculate the prefix sums before comparing them. This specific operation should result in constant time instead of adding that extra N factor to the running time. Afte this, the triple loop in the pseudocode becomes constant per r, making the runtime $O(N^2K)$. Essentially, in the recurrence with min and max, we want to calculate $\sum_{t=1}^{1} L[t]$ so that the length of boards r is $len(r, i) = S[i] - S[r-1]$. Only afterward do we need to follow through with the rest of

the recurrence, except without the summation operation this time. This will reduce the N factor by one, resulting in a running time of $O(N^2K)$.

Honors Problem - Binge Watching Greedily (0 Points, **)

Garfield wants to watch TV **non-stop** for the entire time period $[S, F)$. He has a list of n TV shows (each on a different channel), where the i-th show runs for the time period $[s_i, f_i)$, and the union of all $[s_i, f_i)$ fully covers the entire time period $[S, F)$

Garfield doesn't mind switching in the middle of a show he is watching, but is very lazy to switch TV channels, so he wants to find the smallest set of TV shows that he can watch, and still stay occupied for the entire period $[S, F)$. Your goal is to design an efficient $O(n \log n)$-time greedy algorithm to help Garfield.

1. Describe your greedy algorithm in plain English.

Solution:

2. Describe how to implement your algorithm in $O(n \log n)$ time. Prove the correctness of your algorithm and the bound on its run time. (Hint: show that the output of your algorithm is never worse than *any* optimal solution.[1])

Solution:

[1] For a given list of n TV shows, there could be multiple optimal solutions. Your greedy algorithm only needs to return one of them.