INSTRUCTIONS:

1. Submit your solution to Gradescope by the due time; no late submissions will be accepted.
2. Type your solution (except figures; figures can be hand-drawn and then scanned); no hand-written solutions will be accepted.

Problem 1 (10 points).

Let T be the dynamic programming table when computing the edit distance between two strings. Prove that along any diagonal of T the values are non-decreasing.

Problem 2 (10 points).

Let S be a set of reads, each of which is of length at most L. Prove that the shortest common superstring (SCS) of S does not contain a repeat of length $2L-1$.

Problem 3 (10 points).

Given two strings s and t, design an algorithm to find the optimal suffix-prefix alignment (Lecture 19). Specifically, the algorithm should find a suffix of s, a prefix of t, and an alignment between the suffix and the prefix such that its score is maximized. You are given the score matrix $score(a,b)$, $a,b \in \Sigma$, which gives the score when letter a is aligned to letter b. You may assume the unit gap score, i.e., the score of a single gap "-" is a fixed given parameter g, $g < 0$. Your algorithm should run in $O(|s| \cdot |t|)$ time.

Solution. Define $OPT(i,j)$ as the maximum score of aligning *some prefix* of $s[1 \cdots i]$ and $t[1 \cdots j]$. The recurrence will be:

$$OPT(i,j) = \max \begin{cases} OPT(i-1,j-1) + score(s[i],t[j]) \\ OPT(i-1,j) + g \\ OPT(i,j-1) + g \end{cases}$$

Be careful with the initialization step. Note that $OPT(i,j)$ compare a suffix of $s[1 \cdots i]$ with $t[1 \cdots j]$. Henc, we should initialize $OPT(i,0) = 0$, for any $0 \le i \le |s|$, as we can always choose the empty suffix of $s[1 \cdots i]$ to compare with t, which is empty in this case. We initialize $OPT(0,j) = j \cdot g$. The entire table can be filled up using above recurrence. Be careful again about the termination step. Notice that t can stop any where, as we align a suffix of s to a *prefix* of t. So we should pick $\max_{0 \le j \le |t|} OPT(|s|,j)$, which gives the optimal score. The actual alignment can be obtained with trace-back.

Common issues found in grading: (a), incorrect or missing initialization; deducting 20% points. (b), finding maximum thoughout the entire table or missing the termination step; deducting 20% points.

Problem 4 (10 points).

You run an ice cream business, and you want to place some advertisements in your local newspaper. There are two kinds of ads you can run, Type-C and Type-W, and you've noticed that Type-C works best on cold days (by promoting the good taste of your ice cream) and Type-W works best on warm days (by mentioning how cold and refreshing your ice cream is). Depending on the weather and which ad you run, you see a certain amount of increased profit that day: on a cold day the profit will be \$75 if running a Type-C ad and \$50 if running a Type-W ad, on that day; on a warm day the profit will be \$50 if running a Type-C ad and \$100 if running a Type-W ad, on that day. You have committed to running an ad every day. The cost of placing either a Type-C or Type-W ad is \$10 per day. But the newspaper charges you a fee of \$25 every time you change which ad you are running. You are given a (perfectly correct) weather prediction for the next n days. Design a dynamic programming algorithm to select which ad to run on each of the next n days to maximize your total profit. For examples, for an input being WWWCCCWCWCWCW, where C/W indicates a cold/warm day, you should output WWWCCCWWWWWWW, where C/W indicates running a Type-C/typw-W ad, with a total profit of \$895. Your algorithm should run in $O(n)$ time.

Solution. Define $FC(i)$ as the maximum profit up to day-i while day-i runs a Type-C ad; define $FW(i)$ as the maximum profit up to day-i while day-i runs a Type-W ad. We have the following recurrence:
If day-i is cold, then:
$FC(i) = \max\{FC(i-1), FW(i-1) - 25\} + 75 - 10$, and
$FW(i) = \max\{FW(i-1), FC(i-1) - 25\} + 50 - 10$.
If day-i is warm, then:
$FC(i) = \max\{FC(i-1), FW(i-1) - 25\} + 50 - 10$, and
$FW(i) = \max\{FW(i-1), FC(i-1) - 25\} + 100 - 10$.

We initialize $FC(0) = FW(0) = 0$, and fill up two arrays FC and FW with above recurrence. Finally, we pick the larger one between $FC(n)$ and $FW(n)$. The actual choices for each day can be obtained through tracing back. The running time is certainly $O(n)$ as calculating each $FC(i)$ or $FW(i)$ takes constant time.

Problem 5 (10 points).

Consider the (global) edit distance problem. Often there are multiple optimal alignments (i.e., all have the same, minimized edit distance). Given two sequences s and t, design an algorithm to compute the number of distinct optimal alignments. Your algorithm should run in $O(|s| \cdot |t|)$ time. *Hint:* the number of distinct optimal alignments can be obtained by computing the number of optimal traceback paths.

Problem 6 (10 points).

You are given an undirected graph $G = (V, E)$ and an integer k; assume that $|E| \geq k$. You aim to remove k edges from G such that in the resulting graph the number of connected components is maximized. Formulate this problem as an ILP. *Hint:* the main idea can be borrowed from Lecture 21.

Solution. Assume $|V| = n$, $|E| = m$, $V = \{v_1, v_2, \cdots, v_n\}$. For each edge $e \in E$, we use binary variable x_e to indicate whether e will be kept ($x_e = 1$) or removed ($x_e = 0$). We add a constraint $\sum_{e \in E} x_e = m - k$ to make sure k edges will be removed. Then, we add a variable y_i to represent the label of v_i. We use constraint $y_i \geq 1$ and $y_i \leq i$, $1 \leq i \leq n$, to set a distinct upper bound for different vertices, as we did in Lecture 21. For any edge $e = (v_i, v_j)$ we add constraint $y_i \leq y_j + i \cdot (1 - x_e)$ and $y_j \leq y_i + j \cdot (1 - x_e)$ to guarantee that, if edge e gets kept, i.e., $x_e = 1$, then v_i and v_j will have the same label, i.e., $y_i = y_j$.

Think about any connected component (after removing k edges): all vertices in it will have the same label because the connected vertices by edges will have the same label and such equality will propogate to the entire component. Therefore, *at most* one vertex in each component can reach its upper bound, since all vertices have distinct upper bounds. This gives a way to count the number of connected components: by counting the number of vertices whose label reaches its upper bound! Now, introduce binary variable z_i for vertex v_i to indicate whether the label for v_i can reach its upper bound, i.e., whether $y_i = i$; we use contraints: $i \cdot z_i \leq y_i$ to implement that (you can see $z_i = 1$ only if $y_i = i$). Finally, we set the objective function to be $\max \sum_{i=1}^{n} z_i$, which is to maximize the number of vertices whose label reaches its upper bound, which is equivalent to maximize the number of components.