Problem 1. Let x and y be two strings of length n and m, respectively. Suppose that $x[n] = y[m]$. Prove: the following are true for any LCS z of x and y.

- Let k be the length of z. It holds that $z[k] = x[n] = y[m]$.
- $z[1 : k-1]$ is an LCS of $x[1 : n-1]$ and $y[1 : m-1]$.

Proof.

- Suppose $z[k] \neq x[n]$ (i.e. $z[k] \neq y[m]$), then we can always construct a longer LCS by concatenating z with $x[n]$, which is longer than z, a contradiction.
- Suppose $z[1 : k-1]$ is not an LCS of $x[1 : n-1]$ and $y[1 : m-1]$, we can identify an LCS z' of $x[1 : n-1]$ and $y[1 : m-1]$ with length k longer than $z[1 : k-1]$. Thus, by concatenating z' with $x[n]$ and $y[m]$, we have obtained a longer LCS with length $k+1$, which is a contradiction.

□

Problem 2. Let x be a string of length n, and y a string of length m. Define $opt(i, j)$ to be the length of an LCS of $x[1 : i]$ and $y[1 : j]$ for $i \in [0, n]$ and $j \in [0, m]$. Explain an algorithm that can output an LCS of x and y in $O(nm)$ time.

Solution. Recall that

$$f(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ 1 + f(i-1, j-1), & \text{if } i, j > 0 \text{ and } x[i] = y[j], \\ \max\{f(i, j-1), f(i-1, j)\}, & \text{otherwise.} \end{cases}$$

By computing all the $f(i, j)$ ($1 \leq i \leq n, 1 \leq j \leq m$) in row-major order, we can achieve an $O(nm)$ time complexity.

We can apply the piggyback technique by defining

$$best(i,j) = \begin{cases} \text{nil}, & \text{if } i = 0 \text{ or } j = 0, \\ (i-1, j-1), & \text{if } i, j > 0 \text{ and } x[i] = y[j], \\ \arg\max\{f(i, j-1), f(i-1, j)\}, & \text{otherwise.} \end{cases}$$

By referring to the table storing all the $f(i, j)$'s, we can compute $best(i, j)$ in $O(nm)$ time. We are ready to construct the LCS z of x and y. First, set z to the empty string if either x or y is an empty string. Second, if $x[n] = y[m]$, we recursively obtain an LCS z' of $x[1 : n-1]$ and $y[1 : m-1]$ and set z as the concatenation of z' and $x[n]$. Otherwise, we recursively obtain z' by computing the LCS of $x[1 : n-1]$ and $y[1 : m]$ if $best(i, j) = (i-1, j)$, or the LCS of $x[1 : n]$ and $y[1 : m-1]$, and then set $z = z'$.

Problem 3. **(Matrix-Chain Multiplication)** The goal in this problem to calculate $\boldsymbol{A}_1\boldsymbol{A}_2\ldots\boldsymbol{A}_n$ where $\boldsymbol{A}_i$ is an $a_i \times b_i$ matrix for $i \in [1,n]$. This implies that $b_{i-1} = a_i$ for $i \in [2,n]$, and the final result is an $a_1 \times b_n$ matrix. In $O(abc)$ time, we can compute the matrix product $\boldsymbol{AB}$ using algorithm $\mathcal{A}$, where $\boldsymbol{A}$ is an $a \times b$ matrix and $\boldsymbol{B}$ is a $b \times c$ matrix. To calculate $\boldsymbol{A}_1\boldsymbol{A}_2\ldots\boldsymbol{A}_n$, you can apply parenthesization, namely, convert the expression to $(\boldsymbol{A}_1\boldsymbol{A}_2\ldots\boldsymbol{A}_i)(\boldsymbol{A}_{i+1}\ldots\boldsymbol{A}_n)$ for some $i \in [1, n-1]$, and then parenthesize each of $\boldsymbol{A}_1\ldots\boldsymbol{A}_i$ and $\boldsymbol{A}_{i+1}\ldots\boldsymbol{A}_n$ recursively. A fully parenthesized product is

- either a single matrix, or
- the product of two fully parenthesized products.

For example, if $n = 4$, then $(\boldsymbol{A}_1\boldsymbol{A}_2)(\boldsymbol{A}_3\boldsymbol{A}_4)$ and $((\boldsymbol{A}_1\boldsymbol{A}_2)\boldsymbol{A}_3)\boldsymbol{A}_4$ are fully parenthesized, but $\boldsymbol{A}_1(\boldsymbol{A}_2\boldsymbol{A}_3\boldsymbol{A}_4)$ is not. Each fully parenthesized product has a computation cost under $\mathcal{A}$; e.g., given $(\boldsymbol{A}_1\boldsymbol{A}_2)(\boldsymbol{A}_3\boldsymbol{A}_4)$, the algorithm first calculate $\boldsymbol{B} = \boldsymbol{A}_1\boldsymbol{A}_2$ and $\boldsymbol{B}_2 = \boldsymbol{A}_3\boldsymbol{A}_4$, and then calculate $\boldsymbol{B}_1\boldsymbol{B}_2$. The cost of the fully parenthesized product is the total cost of the three pairwise matrix multiplications. Design an algorithm to find in $O(n^3)$ time a fully parenthesized product with the smallest cost.

Solution. Let $f(i,j)$ be the smallest cost of computing a fully parenthesized product of $\boldsymbol{A}_i\boldsymbol{A}_{i+1}\ldots\boldsymbol{A}_j$. Then, for every $i \le j$,

$$f(i,j) = \begin{cases} 0, & \text{if } i = j, \\ \min_{k=i}^{j-1}(a_i b_k b_j + f(i,k) + f(k+1,j)), & \text{otherwise.} \end{cases}$$

Key observation: $\boldsymbol{B}_1 = \boldsymbol{A}_i\ldots\boldsymbol{A}_k$ is an $a_i \times b_k$ matrix and $\boldsymbol{B}_1 = \boldsymbol{A}_{k+1}\ldots\boldsymbol{A}_j$ is an $a_{k+1} \times b_j$ matrix. Hence, the computational cost of calculating $\boldsymbol{B}_1\boldsymbol{B}_2$ is $O(a_i b_k b_j)$. We can let $k = i, i+1, \ldots, j-1$ to determine the best way of computing the product.

Using dynamic programming, we can compute $f(1,n)$ in $O(n^3)$ time. Namely, in round i we compute all $f(a, a+i)$ where $1 \le a \le n-i$, in $O(n^2)$ time per round.

To use the the piggyback technique, we can define $bestSub(i,j)$ to store k that minimizes $a_i b_k b_j + f(i,k) + f(k+1,j)$, in $O(n^3)$ time. Afterwards, we can generate an optimal parenthesization in $O(n^2)$ extra time. (Consider $g(i,j) = O(1) + \max_{k=i}^{j-1} g(i,k) + g(k+1,j)$.)

Problem 4. **(Longest Increasing Subsequence)** Let A be a sequence of n distinct integers. A sequence B of integers is a subsequence of A if it satisfies one of the following conditions:

- $A = B$, or
- we can convert A to B by repeatedly deleting integers.

The subsequence B is ascending if its integers are arranged in ascending order. Design an algorithm to find an ascending subsequence of A with the maximum length. Your algorithm should run in $O(n^2)$ time. For example, if $A = (10, 5, 20, 17, 3, 30, 25, 40, 50, 60, 24, 55, 70, 58, 80, 44)$, then a longest ascending sequence is $(10, 20, 30, 40, 50, 60, 70, 80)$.

Solution. Define $f(i)$ as the length of the longest possible ascending subsequence of $A[1 : i]$ that ends with the element $A[i]$. Furthermore, let $S(i)$ be a set of index satisfying $\{k : k < i \text{ and } A[k] < A[i]\}$.

$$f(i) = \max\{1, 1 + \max_{k \in S(i)} f(k)\}.$$

Observation: When $f(i) = 1$, it must be that $S(i) = \emptyset$ since all $f(k) \geq 1$; Otherwise, there is some ascending subsequence ending with $A[j]$ that has $A[i] < A[j]$ and forms a longer ascending subsequence ending with $A[i]$.

Using dynamic programming, we can compute $f(i)$ for all $i \in [1, n]$ in $O(n^2)$ time. The maximum length of ascending subsequence of A is thus the maximum length between all longest asending subsequence that ends with $A[1], A[2], \ldots, A[n]$.

By the piggyback technique, we can produce a longest asending subsequence of A in $O(n^2)$ extra time.

Problem 5. Let A be an array of n integers (A is not necessarily sorted). Each integer in A may be positive or negative. Given i, j satisfying $1 \leq i \leq j \leq n$, define subarray $A[i : j]$ as the sequence $(A[i], A[i + 1], \ldots, A[j])$, and the weight of $A[i : j]$ as $A[i] + A[i + 1] + \ldots + A[j]$. For example, consider $A = (13, -3, -25, 20, -3, -6, -23, 18)$; $A[1 : 4]$ has weight 5, while $A[2 : 4]$ has weight -8. Design an algorithm to find a subarray of A with the largest weight in $O(n)$ time.

Solution. Define $f(i)$ to be the weight of the largest subarray that ends at i. Then,

$$f(i) = \begin{cases} A[1], & \text{if } i = 1, \\ \max\{A[i], f(i-1) + A[i]\}, & \text{otherwise.} \end{cases}$$

Proof of correctness: It is obviously true for $i = 1$. Assume that $f(i - 1) \leq 0$, then the weight of $A[t : i - 1]$ for any $t \leq i - 1$ cannot exceed $f(i - 1)$. Hence, the weight of $A[t : i]$ is at most $A[i : i]$. Thus, $f(i)$ is exactly $A[i]$ as we can take $A[k : k]$ as the subarray. Next, assume that $f(k - 1) > 0$ by taking $A[t : k - 1]$ as the subarray. Then, suppose that $A[t' : k]$ obtains a larger weight than $A[t : k]$, it must be that $A[t' : k - 1]$ has a larger weight than $A[t : k - 1]$ by subtracting $A[k]$ from both subarrays. This is a contradiction.

Using dynamic programming, we can obtain $f(i)$ for all $i \in [1, n]$ in $O(n)$ time. The maximum weight of all subarrays of A is then the maximum weight of all subarrays ending at $i = 1, 2, \ldots, n$, which is

$$\max_{i=1}^{n} f(i),$$

and obtainable in an extra $O(n)$ time.

Using the piggyback technique, we can obtain the subarray that sums up to the optimal weight in $O(n)$ time.

Problem 6. Let x be a string of length n, and y a string of length m. Define $opt(i, j)$ to be the length of an LCS of $x[1:i]$ and $y[1:j]$ for $i \in [0, n]$ and $j \in [0, m]$. Compute the values of all possible (i, j) for $x = \texttt{10010101}$ and $y = \texttt{010110110}$.

Solution.

$opt(i,j)$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1
2	0	1	1	2	2	2	2	2	2	2
3	0	1	2	2	2	2	3	3	3	3
4	0	1	2	2	3	3	3	4	4	4
5	0	1	2	3	3	3	4	4	4	5
6	0	1	2	3	4	4	4	5	5	5
7	0	1	2	3	4	4	5	5	5	6
8	0	1	2	3	4	5	5	6	6	6

Problem 7. Find an LCS of x and y, where $x = \texttt{10010101}$ and $y = \texttt{010110110}$.

Solution. We first compute $best(i, j)$ to identify the decision process happened for the computation of LCS for every substring $x[1:i]$ and $y[1:j]$.

$best(i,j)$	0	1	2	3	4	5	6	7	8	9
0	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil
1	nil	(1,0)	(0,1)	(1,2)	(0,3)	(0,4)	(1,5)	(0,6)	(0,7)	(1,8)
2	**nil**	(1,0)	(2,1)	(1,2)	(2,3)	(2,4)	(1,5)	(2,6)	(2,7)	(1,8)
3	nil	**(2,0)**	(3,1)	(2,2)	(3,3)	(3,4)	(2,5)	(3,6)	(3,7)	(2,8)
4	nil	(3,1)	**(3,1)**	(4,2)	(3,3)	(3,4)	(4,5)	(3,6)	(3,7)	(4,8)
5	nil	(4,0)	(4,2)	**(4,2)**	**(5,3)**	(5,4)	(4,5)	(5,6)	(5,7)	(4,8)
6	nil	(5,1)	(5,1)	(5,3)	(5,3)	**(5,4)**	(6,5)	(5,6)	(5,7)	(6,8)
7	nil	(6,0)	(6,2)	(6,2)	(6,4)	(7,4)	**(6,5)**	**(7,6)**	(7,7)	(6,8)
8	nil	(7,1)	(7,1)	(7,3)	(7,3)	(7,4)	(8,5)	(7,6)	**(7,7)**	**(8,8)**

The LCS of x and y is thus $\texttt{001011}$ $((2,0) \longrightarrow (3,1) \longrightarrow (4,2) \longrightarrow (5,4) \longrightarrow (6,5) \longrightarrow (8,8))$, each pair denotes that $x[i] = y[j]$ is matched to construct the optimal LCS, and the path that reaches $f(8,9)$ is shown in blue.

Problem 8. Given a string s of length n, stored in an array of characters, we call $s[i : j]$ a substring of s, for all pairs of i, j satisfying $1 \leq i \leq j \leq n$. Let x be a string of length n, and y a string of length m. Design an algorithm to find a longest common substring of x and y in $O(nm)$ time.

Solution. Define $f(i, j)$ to be the longest common substring of x and y that ends at i and j, respectively. Then,

$$f(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ 0, & \text{if } i, j > 0 \text{ and } A[i] \neq B[j], \\ 1 + f(i-1, j-1), & \text{if } i, j > 0 \text{ and } A[i] = B[j]. \end{cases}$$

Proof of correctness: It is obviously true for $i = 0$ or $j = 0$ as $x[1 : 0]$ and $y[1 : 0]$ is empty. Now consider an optimal substring by the match of $x[t_1 : i]$ and $y[t_2 : j]$ that is overlooked by our algorithm. Apparently, when $x[i] \neq y[j]$, no common substring with ending with x at i and y at j exists as it would violate the definition of common substring, then it must be that such $x[t_1 : i]$ and $y[t_2 : j]$ does not exist. Otherwise, assume that $x[i] = y[j]$. It must be that $x[t_1 : i-1]$ and $y[t_2 : j-1]$ forms a better solution for $f(i-1, j-1)$ that should have been considered by $f(i-1, j-1)$, which is a contradiction.

Problem 9. Let M be an $n \times n$ matrix where each cell $M[i, j]$ stores a distinct integer, for all $i \in [1, n]$ and $j \in [1, n]$. Define a path of length $\ell \geq 1$ to be a sequence of ℓ cells $M[i_1, j_1], M[i_2, j_2], \ldots, M[i_\ell, j_\ell]$ satisfying both conditions below:

- for each $k \in [2, n]$, $M[i_{k-1}, j_{k-1}]$ and $M[i_k, j_k]$ are neighboring cells (this means the former cell is above, below, to the left of, or to the right of the latter cell);
- for each $k \in [2, n]$, $M[i_{k-1}, j_{k-1}] < M[i_k, j_k]$.

Design an algorithm that finds a path of the maximum length in $O(n^2 \log n)$ time.
(Hint 1: Find the length of longest paths starting from each cell.)
(Hint 2: To choose a topological order, sort all the cells.)

Solution. Define $f(i, j)$ to be the longest path that ends at cell (i, j). Furthermore, denote $S(i, j)$ to be the set of legal neighbours for cell (i, j) so that any $(i', j') \in S$ satisfies $|i-i'|+|j-j'| = 1$ and $M[i', j'] < M[i, j]$. Then,

$$f(i,j) = \begin{cases} 0, & \text{if } S(i,j) = \emptyset, \\ 1 + \max_{(i',j') \in S(i,j)} \{f(i', j')\}, & \text{otherwise.} \end{cases}$$

Proof of correctness: It is apparently true for $S(i, j) = \emptyset$, as no edge flows into cell (i, j) forming a path longer than 1 while being constrained to end at (i, j). Now, assume that there is a path of

Obtain a list of cells with their coordinates and sort the list in ascending order in $O(n^2 \log n^2) = O(n^2 \log n)$ time. Then, when we calculate $f(i, j)$, all the possible subproblems $f(i', j')$'s must have been solved since any $(i', j') \in S$ must have $M[i', j'] < M[i, j]$.

Therefore, the longest path is the maximum amongst the longest paths ending with all (i, j)'s, computable in $O(n^2)$ time, which is

$$\max_{i=1}^{n} \max_{j=1}^{n} f(i, j).$$

Using the piggyback technique, we can obtain the longest path with an extra $O(n^2)$ time.

Problem 10. Improve the running time of the solution derived in Problem 9 to $O(n^2)$. (Hint: What is the dependency graph among the cells?)

Solution. We only need to consider a better way for the calculation of $f(i, j)$.

The dependency graph among the cells must form a DAG. Consider each cell as a vertex and every edge $\{(i, j), (i', j')\}$ indicates that we can walk from (i, j) to (i', j') as they are neighbours and $M[i, j] < M[i', j']$. The graph has n^2 vertices, and at most $4n$ edges since each vertex has at most 4 neighbours, and thus bounding the size of $S(i, j)$.

We can then build a DAG G and compute its topological order in $O(|V| + |E|)$ time. Then, we can compute all $f(i, j)$'s in $O(|V| + |E|)$ time as we have resolved all the dependencies.