CSCI3160: Regular Exercise Set 7

Prepared by Yufei Tao

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 $y[1 : m-1]$.

Solution. *Proof of the first bullet.* Let G be a correspondence graph induced by z (as defined in our lecture) and let e be the rightmost edge of G. If $z[k] \neq x[n]$ (and hence $z[k] \neq y[m]$), then e cannot be incident on $x[n]$, and e cannot be incident on $y[m]$. We can therefore add another edge to G by connecting $x[n]$ and $y[m]$. The new graph implies a common subsequence of x and y that is longer than z, giving a contradiction.

Proof of the second bullet. This is in fact a corollary of the first bullet. Suppose that $z[1 : k-1]$ is not an LCS of $x[1 : n-1]$ and $y[1 : m-1]$. Then, identify any LCS z' of $x[1 : n-1]$ and $y[1 : m-1]$, which is longer than z. Thus, $z' \circ x[n]$ (where $\circ$ is the "concatenation" operator) is an LCS of x and y. As z' is longer than z, we now have 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 $j \in [0, m]$. In the lecture, we already discussed how to calculate $opt(i, j)$ for all possible (i, j) pairs. Based on that discussion, explain an algorithm that can output an LCS of x and y in $O(nm)$ time.

Solution. Recall:

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

We will now apply the "piggyback technique" discussed in the lecture to generate an LCS. For this purpose, let us define

$$bestSub(i,j) = \begin{cases} nil & \text{if } i = 0 \text{ or } j = 0 \\ nil & \text{if } i, j > 0 \text{ and } x[i] = y[j] \\ \text{shrink x} & \text{if } i, j > 0,\ x[i] \neq y[j], \text{ and } opt(i-1, j) \geq opt(i, j-1) \\ \text{shrink y} & \text{if } i, j > 0,\ x[i] \neq y[j], \text{ and } opt(i-1, j) < opt(i, j-1) \end{cases}$$

After computing $opt(i, j)$ for all (i, j) pairs, we can compute each $bestSub(i, j)$ in constant time. The total time is $O(nm)$.

We can now construct an LCS z of x and y as follows. First, if x or y is the empty string, set z to the empty string. Second, if $x[n] = y[m]$, recursively obtain an LCS z' of $x[1 : n-1]$ and $y[1 : m-1]$ and then set $z = z' \circ x[n]$, where $\circ$ means concatenation. Finally, if $x[n] \neq y[m]$, we act differently according to $bestSub(n, m)$:

- If it is "shrink x", we recursively obtain an LCS z' of $x[1 : n-1]$ and y and then set $z = z'$.

- If it is "shrink y", we recursively obtain an LCS z' of x and $y[1 : m-1]$ and then set $z = z'$.

Problem 3 (Matrix-Chain Multiplication). The goal in this problem is to calculate $\boldsymbol{A}_1\boldsymbol{A}_2...\boldsymbol{A}_n$ 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. You are given an algorithm $\mathcal{A}$ that, given an $a \times b$ matrix $\boldsymbol{A}$ and a $b \times c$ matrix $\boldsymbol{B}$, can calculate $\boldsymbol{AB}$ in $O(abc)$ time. To calculate $\boldsymbol{A}_1\boldsymbol{A}_2...\boldsymbol{A}_n$, you can apply *parenthesization*, namely, convert the expression to $(\boldsymbol{A}_1...\boldsymbol{A}_i)(\boldsymbol{A}_{i+1}...\boldsymbol{A}_n)$ for some $i \in [1, n-1]$, and then parenthesize each of $\boldsymbol{A}_1...\boldsymbol{A}_i$ and $\boldsymbol{A}_{i+1}...\boldsymbol{A}_n$ 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 $(\boldsymbol{A}_1\boldsymbol{A}_2)(\boldsymbol{A}_3\boldsymbol{A}_4)$, you first calculate $\boldsymbol{B}_1 = \boldsymbol{A}_1\boldsymbol{A}_2$ and $\boldsymbol{B}_2 = \boldsymbol{A}_3\boldsymbol{A}_4$, and then calculate $\boldsymbol{B}_1\boldsymbol{B}_2$, all using $\mathcal{A}$. 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. Given i, j satisfying $1 \le i \le j \le n$, we define $cost(i, j)$ to be the smallest achievable cost for calculating $\boldsymbol{A}_i\boldsymbol{A}_{i+1}...\boldsymbol{A}_j$ with parenthesization. Our objective is to calculate $cost(1, n)$.

A key observation is that $\boldsymbol{B}_1 = \boldsymbol{A}_i...\boldsymbol{A}_k$ is an $a_i \times b_k$ matrix and $\boldsymbol{B}_2 = \boldsymbol{A}_{k+1}...\boldsymbol{A}_j$ is an $a_{k+1} \times b_j$ matrix (where $b_k = a_{k+1}$); so it takes $O(a_i b_k b_j)$ time to compute $\boldsymbol{B}_1\boldsymbol{B}_2$. This means that if we start with the parenthesization $(\boldsymbol{A}_i...\boldsymbol{A}_k)(\boldsymbol{A}_{k+1}...\boldsymbol{A}_j)$, the best achievable cost is $cost(i, k) + cost(k + 1, j) + O(a_i b_k b_j)$. This implies:

$$cost(i,j) = \begin{cases} O(1) & \text{if } i = j \\ \min_{k=i}^{j-1}(cost(i,k) + cost(k+1,j) + O(a_i b_k b_j)) & \text{if } i < j \end{cases}$$

Using dynamic programming, we can compute $cost(1, n)$ in $O(n^3)$ time. Using the "piggyback technique", we can produce an optimal parenthesization in $O(n^3)$ extra time.

Problem 4 (Longest Ascending 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. We say that B is an *end-aligned ascending subsequence* of A if $A[n]$ is the last integer in B. In the example given in the problem statement, $(5, 20, 30, 40, 44)$ is an end-aligned ascending subsequence of A, while $(10, 20, 30, 40, 50, 60, 70, 80)$ is not. Given an $i \in [1, n]$, we use $len(i)$ to denote the maximum length of all end-aligned ascending subsequences of $A[1 : i]$. In our example, $len(16) = 5$ because $(5, 20, 30, 40, 44)$ is a longest end-aligned ascending subsequence of A, but *len*(15) = 8 because (10, 20, 30, 40, 50, 60, 70, 80) is longest end-aligned ascending subsequence of $A[1:15]$.

Let B be an (arbitrary) longest end-aligned ascending subsequence of $A[1:i]$, and define k to be the length of B. There are two possibilities.

- $k = 1$. This implies that $A[j] > A[i]$ for all $j < i$.
- $k > 1$. In this case, let j be the integer such that $B[k-1] = A[j]$. Then, $B[1:k-1]$ must be an end-aligned longest subsequence of $A[1:j]$.

Given an $i \in [1, n]$, define $S(i) = \{j \mid j < i \text{ and } A[j] < A[i]\}$. The above discussion implies:

$$\mathit{len}(i) = 1 + \max_{j \in S(i)} \mathit{len}(j)$$

Using dynamic programming, we can compute $\mathit{len}(i)$ for all $i \in [1, n]$ in $O(n^2)$ time.

The maximum length of all ascending subsequences of A is

$$\max_{i=1}^{n} \mathit{len}(i).$$

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

Problem 5*. In this problem, we will revisit a regular exercise discussed before and derive a faster algorithm using dynamic programming.

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 $1 \le i \le j \le n$, define *subarray* $A[i:j]$ as the sequence $(A[i], A[i+1], ..., A[j])$, and the *weight* of $A[i:j]$ as $A[i] + A[i+1] + ... + A[j]$. For example, consider $A = (13, -3, -25, 20, -3, -16, -23, 18)$; $A[1:4]$ 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.

Remark: We solved the problem using divide-and-conquer in $O(n \log n)$ time before.

Solution. Given a subarray $A[i:j]$, we refer to j as the subarray's *ending position*. For each $k \in [1, n]$, define $\mathit{maxwght}(k)$ as the largest weight of all the subarrays whose ending positions are k. It holds that

$$\mathit{maxwght}(k) = \begin{cases} A[k] & \text{if } k = 1 \\ A[k] & \text{if } k > 1 \text{ and } \mathit{maxwght}(k-1) \le 0 \\ \mathit{maxwght}(k-1) + A[k] & \text{if } k > 1 \text{ and } \mathit{maxwght}(k-1) > 0 \end{cases}$$

The above obviously holds for $k = 1$. Next, we will prove its correctness for $k > 1$. Let $t \in [1, k]$ be an integer that maximizes the weight of $A[t:k]$.

Consider first the scenario where $\mathit{maxwght}(k-1) \le 0$. Suppose (for contradiction purposes) that $t < k$. Then, the weight of $A[t:k-1]$, which cannot exceed $\mathit{maxwght}(k-1)$, must be non-positive. Hence, the weight of $A[t:k]$ is at most $A[k:k]$. This implies that the weight of $A[t:k]$ — which is $\mathit{maxwght}(k)$ — must be exactly $A[k]$, establishing the second branch in the definition.

Finally, consider $\mathit{maxwght}(k-1) > 0$. Let t' be an integer such that the weight of $A[t':k-1]$ equals $\mathit{maxwght}(k-1)$. As $A[t':k]$ has a larger weight than $A[k:k]$, we can assert that $t < k$. Next, we argue that $A[t:k-1]$ and $A[t':k-1]$ must have the same weight, i.e., $\mathit{maxwght}(k-1)$. If this is not true, $A[t : k-1]$ has a lower weight than $A[t' : k-1]$, because of which $A[t : k]$ has a lower weight than $A[t' : k]$, contradicting the role of t. This establishes the third branch of the definition.

Using dynamic programming, we can calculate $maxwght(k)$ for all $k \in [1, n]$ in $O(n)$ time. The maximum weight of all the subarrays of A equals

$$\max_{k=1}^{n} maxwght(k)$$

which can also be obtained in $O(n)$ time. By resorting to the "piggyback" technique, we can obtain a subarray with the maximum weight in $O(n)$ extra time.