CMSC 451: Lecture 9
DP: Longest Common Subsequence and Edit Distance

Strings: In this lecture we continue our study of dynamic programming algorithms. One important area of algorithm design is the study of algorithms for character strings. Finding patterns or similarities within strings is fundamental to various applications, ranging from document analysis to computational genomics. We study two widely studied measures of string similarity, longest common subsequence and edit distance. Today, we will consider efficient DP solutions to these problems.

Longest Common Subsequence: Consider two character sequences, that is, *strings*,

$$X = \langle x_1, x_2, \ldots, x_m \rangle \qquad \text{and} \qquad Z = \langle z_1, z_2, \ldots, z_k \rangle,$$

where x_i and z_j are elements over some given *alphabet*, Σ. (For example $\Sigma = \{a, b, c, \ldots, z\}$ or $\Sigma = \{A, G, C, T\}$.) Let $|X|$ denote the number of characters in X.

We say that Z is a *subsequence* of X its characters all appear in order in X. More formally, there is a strictly increasing sequence of k indices $\langle i_1, i_2, \ldots, i_k \rangle$ $(1 \le i_1 < i_2 < \ldots < i_k \le n)$ such that $Z = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_k} \rangle$ (see Fig. 2).

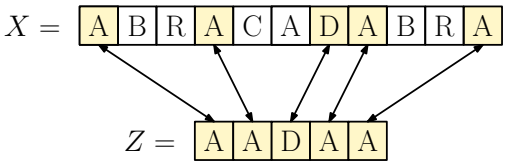

Fig. 1: The string $Z = \langle \text{AADAA} \rangle$ is a subsequence of $X = \langle \text{ABRACADABRA} \rangle$.

Given two strings X and Y, the *longest common subsequence* of X and Y is a longest sequence Z that is a subsequence of both X and Y. For example, let $X = \langle \text{ABRACADABRA} \rangle$ and let $Y = \langle \text{YABBADABBADOO} \rangle$. Then the longest common subsequence is $Z = \langle \text{ABADABA} \rangle$ (see Fig. 2).

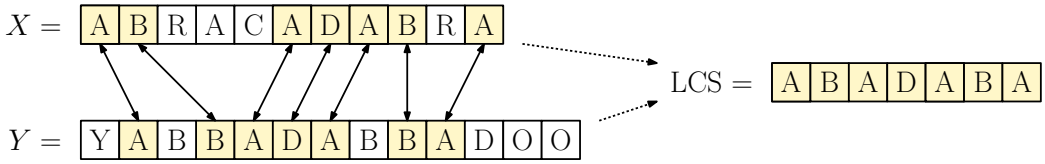

Fig. 2: An example of the LCS of two strings X and Y.

The *Longest Common Subsequence Problem* (LCS) is the following. Given two sequences $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ determine the length of their longest common subsequence, and more generally the sequence itself. Note that the subsequence is not necessarily unique. For example the LCS of $\langle \text{ABC} \rangle$ and $\langle \text{BAC} \rangle$ is either $\langle \text{AC} \rangle$ or $\langle \text{BC} \rangle$.

DP Formulation for LCS: The simple brute-force solution to the problem would be to try all possible subsequences from one string, and search for matches in the other string, but this is hopelessly inefficient, since there are an exponential number of possible subsequences.

Instead, we will derive a dynamic programming solution. In typical DP fashion, we decompose the problem into subproblems, which can be solved recursively. There are many ways to do this for strings, but it turns out for this problem that considering all pairs of *prefixes* will suffice for us. Given $0 \le i \le |X|$, the ith *prefix* of X, denoted X_i, is the initial substring length i, that is, $X_i = \langle x_1, \ldots, x_i \rangle$. Define $X_0 = \langle\rangle$ to be the empty sequence.

The idea will be to compute the longest common subsequence for every possible pair of prefixes. For $0 \le i \le |X|$ and $\text{lcs}(i, j)$ denote the length of the longest common subsequence of X_i and Y_j. For example, in the above case we have $X_5 = \langle\text{ABRAC}\rangle$ and $Y_6 = \langle\text{YABBAD}\rangle$. Their longest common subsequence is $\langle\text{ABA}\rangle$. Thus, $\text{lcs}(5, 6) = 3$.

Let us start by deriving a recursive formulation for computing $\text{lcs}(i, j)$. Later, we will consider how to implement this recursion efficiently.

Basis: If either sequence is empty, then the longest common subsequence is clearly empty. Therefore, $\text{lcs}(i, 0) = \text{lcs}(j, 0) = 0$.

Last characters match: Suppose $x_i = y_j$. For concreteness, let's imagine that this letter is 'A'. Since both strings end in 'A', it is easy to see that the LCS *must* also end in 'A'.[1] Also, there is no harm in assuming that the last two characters of both strings will be matched with each other in forming the LCS. (Matching the last 'A' of one string to an earlier instance of 'A' of the other can only limit our future options.)

Since the 'A' is the last character of the LCS, we can remove it from both strings and continue to find a common subsequence of the prefixes X_{i-1} and Y_{j-1}. Since the removal of the last character has no impact on this subproblem, we should solve it optimally. Therefore, the length of the final LCS is $\text{lcs}(X_{i-1}, Y_{j-1}) + 1$ (see Fig. 3). This provides us with the following rule:

$$\text{if } (x_i = y_j) \quad \text{then} \quad \text{lcs}(i, j) = \text{lcs}(i - 1, j - 1) + 1$$

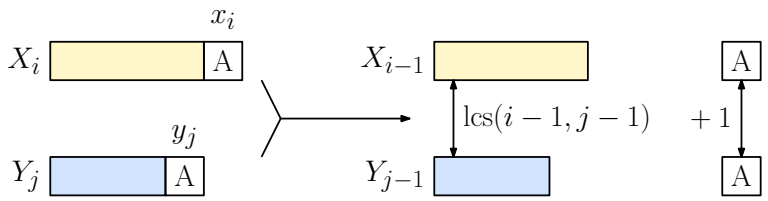

Fig. 3: LCS of two strings, where $x_i = y_j$.

For example, suppose that $X_i = \langle\text{ABCA}\rangle$ and let $Y_j = \langle\text{DACA}\rangle$. We match the final 'A' characters, compute the LCS length of $X_{i-1} = \langle\text{ABC}\rangle$ and $Y_{j-1} = \langle\text{DAC}\rangle$, which is $\langle\text{AC}\rangle$. We then 'A' back, which yields the final LCS of $\langle\text{ACA}\rangle$.

Last characters do not match: Suppose that $x_i \ne y_j$. In this case x_i and y_j cannot both be in the LCS (since they would have to be the last character of the LCS). Thus either x_i is *not* part of the LCS, or y_j is *not* part of the LCS (and possibly *both* are not part of the LCS). Let's consider these two options.

[1] We will leave the formal proof as an exercise, but intuitively this is proved by contradiction. If the LCS did not end in 'A', then we could make it longer by adding 'A' to its end.

x_i is not in the LCS: Since we know that x_i is out, we can remove the last character from X_i, which leaves us with X_{i-1}. We continue to compute the LCS of X_{i-1} and Y_j, which is given by $\text{lcs}(i-1, j)$.

y_j is not in the LCS: Since we know that y_j is out, we can remove the last character from Y_j, which leaves us with Y_{j-1}. We continue to compute the LCS of of X_i and Y_{j-1}, which is given by $\text{lcs}(i, j-1)$.

At this point it may be tempting to try to make a "smart" choice. By analyzing the last few characters of X_i and Y_j, perhaps we can figure out which character is best to discard. However, this approach is doomed to failure (and you are strongly encouraged to think about this, since it is a common point of confusion). Remember the DP selection principle: *When given a set of feasible options to choose from, try them all and take the best.*

Let's not try to be smart. Consider both options, and see which one provides the better result.

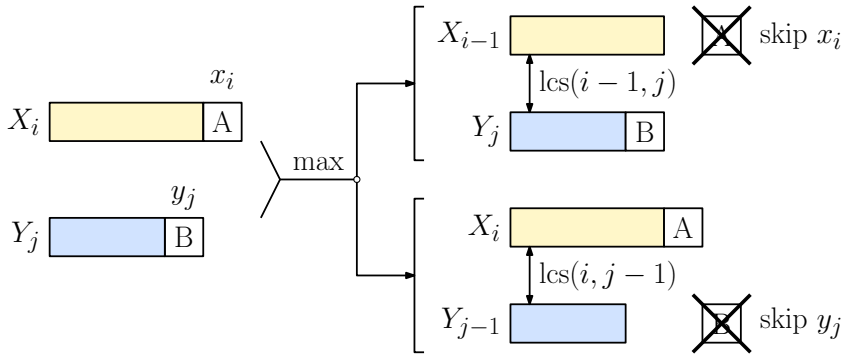

Fig. 4: LCS of two strings, where $x_i \neq y_j$.

We compute both options and take the one that gives us the longer LCS (see Fig. 4). (Hey, did we forget Option 3, where *neither* symbol is in the LCS? Yes, this can happen, but these two rules suffice to handle this. Try it out and you'll see.) Thus, we have the following rule:

$$\text{if } (x_i \neq y_j) \text{ then } \text{lcs}(i, j) = \max(\text{lcs}(i-1, j), \text{lcs}(i, j-1))$$

Combining these observations we have the following recursive *DP formulation*:

$$\text{lcs}(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \text{lcs}(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(\text{lcs}(i-1, j), \text{lcs}(i, j-1)) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Memoized implementation: The principal source of the inefficiency in a naive implementation of the recursive rule is that it makes repeated calls to $\text{lcs}(i, j)$ for the same values of i and j. To avoid this, it creates a 2-dimensional array $\text{lcs}[0..m, 0..n]$, where $m = |X|$ and $n = |Y|$. We initialize its elements to -1, which indicates that the entry is currently undefined. The memoized version first checks whether the requested value has already been computed, and

if so, it just returns the cached value. Otherwise, it invokes the recursive rule to compute it. Our objective is to compute the LCS of the entire strings of lengths m and n, so the initial call is memo-lcs(m, n).

Because we will eventually want to construct the final LCS, we will also add some "hooks" to our code to record our decisions. We create a parallel *hook table*, $H[0..n, 0..m]$, which stores three possible values.

$+$: Add $x_i (= y_j)$ to the end of the LCS. (Represented by the symbol '$\nwarrow$'.)

X : Do not include x_i to the LCS. (Represented by the symbol '$\uparrow$'.)

Y : Do not include y_j to the LCS. (Represented by the symbol '$\leftarrow$'.)

The algorithm is presented in the code block below. See Fig. 5(a) for an example. (We'll discuss the H-table later.)

Memoized LCS with Hooks

```
memo-lcs(i,j) {                                  // memoized implentation of LCS
    if (lcs[i,j] == -1) {                        // undefined?
        if (i == 0 || j == 0) {                  // basis case
            lcs[i,j] = 0
        } else if (x[i] == y[j]) {               // last characters match
            lcs[i,j] = memo-lcs(i-1, j-1) + 1
            H[i,j] = '+'
        } else {                                 // last chars don't match
            skipX = memo-lcs(i-1, j)             // length if we skip X
            skipY = memo-lcs(i, j-1)             // length if we skip Y
            if (skipX >= skipY)                  // better to skip X
                lcs[i,j] = skipX; H[i,j] = 'X'
            else                                 // better to skip Y
                lcs[i,j] = skipY; H[i,j] = 'Y'
        }
    }
    return lcs[i,j]                              // return lcs length
}
```

Correctness follows from the correctness of the DP formulation. The running time is $O(mn)$. To see this, observe that there are $(m + 1)(n + 1) = O(mn)$ entries in the table. The body of each recursive call runs in $O(1)$ time. Each call either returns immediately or fills in one more entry in the tables. Thus, the total time is proportional to the total number of table entries, which is $O(mn)$.

Extracting the LCS: Next, let us see how to use our hooks to extract the final LCS. We will start at the end with $H[m, n]$ and trace the optimal recursion path back to $H[0, 0]$. If $H[i, j] = +$, this means that $x_i = y_j$, and we are putting this common character into the LCS. We add this character to the LCS, and continue with $H[i-1, j-1]$. If $H[i, j] = X$, this means that we are skipping character x_i, and continuing with $H[i-1, j]$. Finally, if $H[i, j] = Y$, this means that we are skipping character y_j, and continuing with $H[i, j-1]$. The code is presented below. An example of the trace-back is shown in Fig. 5(b).

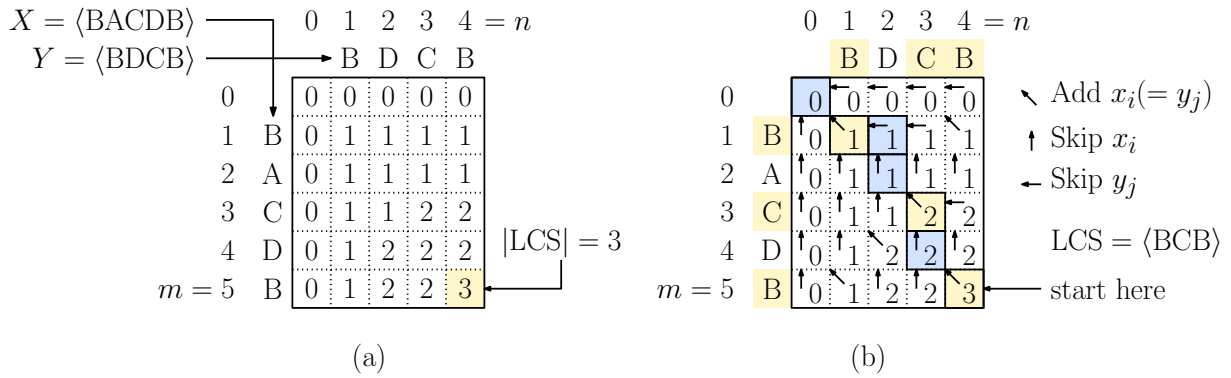

Fig. 5: (a) Contents of the lcs array for the input sequences $X = \langle \text{BACDB} \rangle$ and $Y = \langle \text{BCDB} \rangle$. The numeric table entries are the values of $\text{lcs}[i, j]$. (b) Illustrates the H-table and the extraction of the final sequence.

Extracting the LCS using the Hints

```
get-lcs-sequence() {                          // get the LCS sequence
    LCS = empty
    i = m; j = n                              // start at lower right
    while(i != 0 or j != 0)                   // loop until i == j == 0
        switch H[i,j]
            case '+' ->                       // add x[i] (= y[j])
                prepend x[i] to LCS; i--; j--;
            case 'X' ->                       // skip x[i]
                i--
            case 'Y' ->                       // skip y[j]
                j--
    return LCS
}
```

Bottom-up implementation: (Optional) The alternative to memoization is to just create the lcs table in a bottom-up manner, working from smaller entries to larger entries. By the recursive rules, in order to compute $\text{lcs}[i, j]$, we need to have already computed $\text{lcs}[i-1, j-1]$, $\text{lcs}[i-1, j]$, and $\text{lcs}[i, j-1]$. Thus, we can compute the entries row-by-row or column-by-column in increasing order. See the code block below and Fig. 5(a). The running time and space used by the algorithm are both clearly $O(mn)$.

Edit Distance: A more widely used measure of string similarity than LCS is the edit-distance. This is widely used in the field of computational genomics, when analyzing the similarity of DNA/RNA sequences.

Given two strings $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$, the edit distance is the minimum number of primitive operations needed to convert X into Y. Primitive operations include things like inserting a character, deleting a character, changing the value of a character, or swapping two adjacent characters. Generally, we may apply weights to these choices (e.g., favoring insertion over deletion). Let's keep this simple by focusing on just three operations: insert, delete, and change in the unweighted case. (For example, in Fig. 6) we show that

Bottom-up Longest Common Subsequence

```
bottom-up-lcs() {                               // bottom-up implementation of LCS
    lcs = new array [0..m, 0..n]
    for (i = 0 to m) lcs[i,0] = 0               // basis cases
    for (j = 0 to n) lcs[0,j] = 0
    for (i = 1 to m) {                          // fill rest of table
        for (j = 1 to n) {
            if (x[i] == y[j])                   // take x[i] (= y[j]) for LCS
                lcs[i,j] = lcs[i-1, j-1] + 1
            else
                lcs[i,j] = max(lcs[i-1, j], lcs[i, j-1])
        }
    }
    return lcs[m, n]                            // final lcs length
}
```

the X and be converted Y through 9 edit operations.) The minimum number of insertions, deletions, and changes to convert one string to another is called the *Levenshtein distance* between these strings. It is named for the Soviet mathematician Vladimir Levenshtein, who invented way back in 1965.

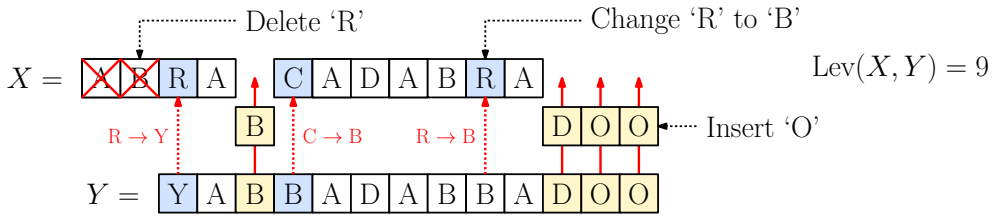

Fig. 6: Levenshtein distance for $X = \langle \text{ABRACADABRA} \rangle$ and $Y = \langle \text{YABBADABBADOO} \rangle$.

Let's develop a DP formulation for this problem. We will follow a pattern similar to the LCS problem. For $0 \leq i \leq m$ and $0 \leq j \leq n$, let $\text{Lev}(i, j)$ denote the Levenshtein distance between the prefixes $X_i = \langle x_1, \ldots, x_i \rangle$ and $Y_i = \langle y_1, \ldots, y_j \rangle$. Let's explore the various cases.

Basis: If either sequence is empty, then the edit distance is equal to the number of characters in the other string. If X_i is empty, then we need to insert all j characters of Y_j. If Y_j is empty, then we need to delete all i characters of X_i. Thus, we have following rules:

$$\begin{aligned} &\text{if } i = 0 \text{ then } \text{Lev}(i, j) = j \\ &\text{if } j = 0 \text{ then } \text{Lev}(i, j) = i \end{aligned}$$

Last characters match: If $x_i = y_j$, then we should go ahead and match these characters. (It costs us nothing to do so, and if we were to hold out to match one of these with an earlier instance of the same character, this would only limit our future options.) This does not incur any increase in the edit distance, and what remains is to match the remaining prefixes, X_{i-1} and Y_{j-1}. Since the removal of the last character has no impact on this subproblem, we should solve it optimally. Therefore, the Leveshtein distance is $\text{Lev}(X_{i-1}, Y_{j-1})$ (see Fig. 3). This provides us with the following rule:

$$\text{if } (x_i = y_j) \quad \text{then} \quad \text{Lev}(i,j) = \text{Lev}(i-1, j-1)$$

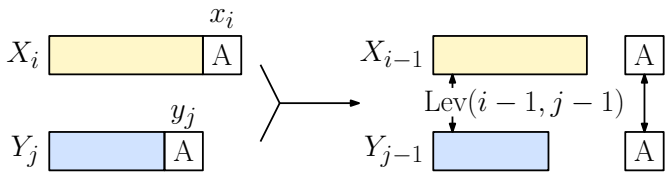

Fig. 7: LCS of two strings, where $x_i = y_j$.

For example, suppose that $X_i = \langle \text{ABCA} \rangle$ and let $Y_j = \langle \text{DACA} \rangle$. We match the final 'A' characters, compute the LCS length of $X_{i-1} = \langle \text{ABC} \rangle$ and $Y_{j-1} = \langle \text{DAC} \rangle$, which is $\langle \text{AC} \rangle$. We then 'A' back, which yields the final LCS of $\langle \text{ACA} \rangle$.

Last characters do not match: If the last character do not match, that is, $x_i \neq y_j$. We know that some edit operation will be needed, but which? There are three options (see Fig. 8).

Insert y_j at the end of X_i: This increases the distance by +1. After doing so, the character y_j has been accounted for. What remains is to compute the distance between X_i with the remainder, Y_{j-1}. In this case, $\text{Lev}(i,j) = 1 + \text{Lev}(i, j-1)$.

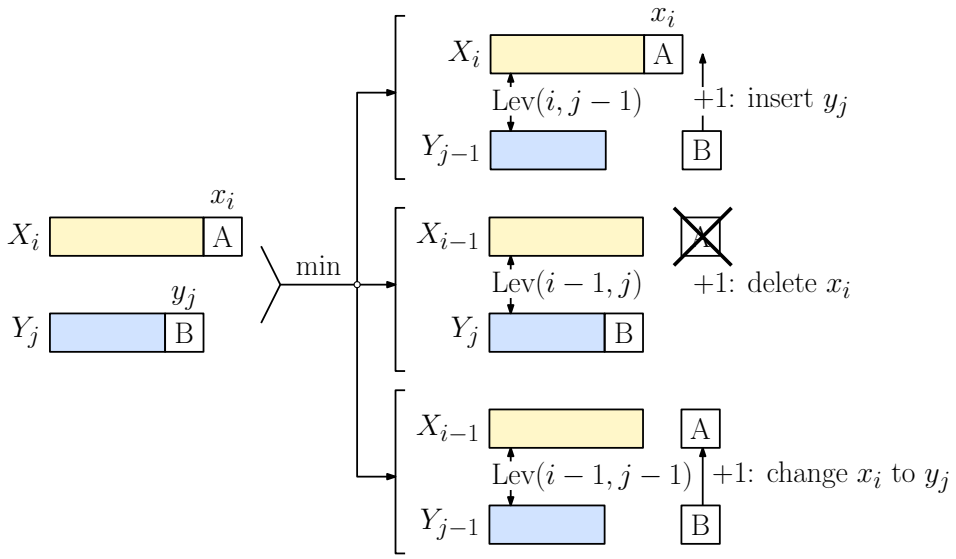

Fig. 8: LCS of two strings, where $x_i = y_j$.

Delete x_i: This increases the distance by +1. After doing so, the character x_i has been accounted for. What remains is to compute the distance between the remainder, X_{i-1}, with Y_j. In this case, $\text{Lev}(i,j) = 1 + \text{Lev}(i-1, j)$.

Change x_i into y_j: This increases the distance by +1. After doing so, both the characters x_i and y_j have been accounted for. What remains is to compute the distance between the remainders, X_{i-1} and Y_{j-1}. In this case, $\text{Lev}(i,j) = 1 + \text{Lev}(i-1, j-1)$.

At this point it may be tempting to try to make a "smart" choice. But, in customary DP fashion, we do not attempt to determine which action is best. We just try them all and take the best, that is, the one that achieves the minimum value. Thus, we have the rule:

$$\text{if } (x_i \neq y_j) \text{ then } \text{Lev}(i,j) = 1 + \min(\text{Lev}(i,j-1), \text{Lev}(i-1,j), \text{Lev}(i-1,j-1))$$

In summary, we have the following recursive *DP formulation* for the Levenshtein distance:

$$\text{Lev}(i,j) = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \text{Lev}(i-1,j-1) & \text{if } \min(i,j) > 0 \text{ and } x_i = y_j, \\ 1 + \min \begin{pmatrix} \text{Lev}(i,j-1), \\ \text{Lev}(i-1,j), \\ \text{Lev}(i-1,j-1) \end{pmatrix} & \text{if } \min(i,j) > 0 \text{ and } x_i \neq y_j. \end{cases}$$

We will leave the implementation (whether memoized or bottom-up) as an exercise. Both are quite similar in structure to the LCS code. The same is true for adding the necessary "hooks" (match, insert, delete, or change). As with LCS, the running time is $O(mn)$.

Summary: We have presented DP algorithms for two problems in string similarity, longest common subsequence (LCS) and the edit or Levenshtein distance. Both algorithms run in time that is proportional to the product of the lengths of the two strings. Needless to say, this is unacceptably slow in many applications where string sizes can be large.

Can we do better? There are near linear-time algorithms for LCS (see Wikipedia). There are many tricks and heuristics for speeding up edit distance in practice. Unfortunately, there is pretty strong evidence that in the worst case, you cannot do much better for the Levenshtein distance. It has been proved that the Levenshtein distance for two strings of length n cannot be computed in time $O(n^{2-\varepsilon})$, for any $\varepsilon > 0$, unless the Strong Exponential Time Hypothesis (SETH, for short) is false. It is beyond the scope of this lecture to introduce SETH is, but it is widely held to be true.