Key Point: Longest common subsequence

- model_name：ChatGPT
- model_version：ChatGPT 4o
- prompt：

> What is the time complexity of the longest common subsequence (LCS) using dynamic programming method, and how should it be calculated.

- response:

> The time complexity of the Longest Common Subsequence (LCS) problem using the dynamic programming (DP) method is O(m * n), where:
> - m is the length of the first sequence (string or list),
> - n is the length of the second sequence (string or list).
>
> **How to Calculate It:**
>
> 1. Problem Overview:
>   The LCS problem is about finding the longest subsequence (not necessarily contiguous) that appears in both strings or sequences in the same order.
> 2. Dynamic Programming Approach:
>   The key idea behind the dynamic programming approach is to break down the problem into smaller subproblems, solve each subproblem only once, and store the results to avoid redundant calculations. Here's how the process works:
>   - You create a 2D table dp where dp[i][j] will store the length of the LCS of the first i characters of string 1 and the first j characters of string 2.
>   - The table will have dimensions (m + 1) x (n + 1) to account for the case where one of the strings is of length 0 (empty string), in which case the LCS is also 0.
> 3. Filling the DP Table:
>   - Base Case: If either string is empty (i.e., i = 0 or j = 0), the LCS is 0. So, dp[i][0] = 0 and dp[0][j] = 0 for all valid i and j.
>   - Recurrence Relation: For each pair of characters X[i-1] and Y[j-1]:
>   - If X[i-1] == Y[j-1], then dp[i][j] = dp[i-1][j-1] + 1
>   - Otherwise, dp[i][j] = max(dp[i-1][j], dp[i][j-1])
>
> This step ensures that you consider all possible pairs of characters and compute the LCS length for increasing sizes of substrings.
>
> 4. Time Complexity:
>
> - You fill a table of size (m + 1) x (n + 1). Each entry dp[i][j] takes constant time, i.e., O(1) to compute.

> Therefore, the total time complexity is:O(mn).
>
> 5. Space Complexity:
> The space complexity is also O(mn) because you are maintaining a table of size (m+1) x (n+1) to store intermediate results.
>
> However, if you only need to keep track of the current and previous rows (since each entry only depends on the previous row and column), you can reduce the space complexity to O(min(m, n)) by using space optimization techniques.
>
> **Example:**
> For two strings:
>
> - String 1: "ABCBDAB"
> - String 2: "BDCABB"
>
> The DP table would be constructed to look something like this:
>
> | | | **B** | **D** | **C** | **A** | **B** | **B** |
> |---|---|---|---|---|---|---|---|
> | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
> | A | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
> | B | 0 | 1 | 1 | 1 | 1 | 2 | 2 |
> | C | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
> | B | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
> | D | 0 | 1 | 2 | 2 | 2 | 3 | 3 |
> | A | 0 | 1 | 2 | 2 | 3 | 3 | 3 |
> | B | 0 | 1 | 2 | 2 | 3 | 4 | 4 |
>
> The length of the LCS is found in dp[m][n], which in this case is 4.
>
> **Key Points:**
> Time complexity: O(m * n)
> Space complexity: O(m * n) (can be optimized to O(min(m, n)) with row/column optimization)

- prompt:

> Your answer is basically fine,
> some of positive feedback are:

> (1) The key idea behind the dynamic programming approach is to break down the problem into smaller subproblems, solve each subproblem only once, and store the results to avoid redundant calculations. -- this gives the reader a deeper understanding of dynamic programming, as well as a better understanding of time complexity calculations.
>
> (2) Your idea of using dynamic programming algorithm to solve LCS problem is basically accurate.
>
> some of negative feedback are:
>
> When calculating space complexity, you mentioned "you can reduce the space complexity to O(min(m, n)) by using space optimization techniques." But you did not introduce specific optimization techniques.
>
> Thus, please describe the row/column optimization you can use for LCS problems.

- response:

> Thank you for the feedback! You're right that I should have elaborated more on the specific optimization techniques for reducing space complexity in the LCS problem.
> **Space Optimization Techniques for LCS (Row/Column Optimization)**
> In the dynamic programming approach to solving the Longest Common Subsequence (LCS), you need a table of size (m + 1) x (n + 1) to store all intermediate results. However, it is possible to optimize the space complexity from O(m * n) to O(min(m, n)) by taking advantage of the fact that to compute dp[i][j], you only need access to the current row and the previous row.
>
> **The Core Idea of Space Optimization:**
>
> - The value of dp[i][j] depends only on dp[i-1][j-1] (previous row, previous column), dp[i-1][j] (previous row, same column), and dp[i][j-1] (same row, previous column).
> - This means that we can discard old rows (or columns) as we move through the table, keeping just enough information to calculate the next values.
>
> **Optimizing with Two Arrays (One for Current Row, One for Previous Row):**
> Instead of maintaining a full 2D table, you can use two 1D arrays to store the current and previous rows:
>
> 1. **Initialize two arrays:**
>   - previous[]: Stores values from the previous row.
>   - current[]: Stores values from the current row.
> 2. **Iterate through the table row by row:**
>   - For each row, calculate the values of dp[i][j] using the values stored in previous[]

(previous row) and current[] (current row).
- After calculating the values for the current row, swap previous[] and current[], so that current[] becomes the old row for the next iteration.

Steps for Row-Based Optimization:

1. **Initialization:**
    - Create two arrays of size n + 1 (where n is the length of the second string):

    ```
    previous = [0] * (n + 1)
    current = [0] * (n + 1)
    ```

2. **Iterate over the rows (first string):**
    - For each character of the first string (indexed by i), update current[] using the values from previous[] and the corresponding values from the second string.

    Example pseudocode:

    ```
    for i in range(1, m + 1):  # i is the current row
    for j in range(1, n + 1):  # j is the current column
        if X[i - 1] == Y[j - 1]:
            current[j] = previous[j - 1] + 1
        else:
            current[j] = max(previous[j], current[j - 1])
    # Swap previous and current
    previous, current = current, previous
    ```

3. **Final Result:**
    - The LCS length will be stored in previous[n] after all iterations, since after the last iteration previous[] holds the values from the final row.

Example of Optimized Space:
Let's say we have two strings:

- String 1: "ABCBDAB"
- String 2: "BDCABB"

Here's how the space optimization works:

1. **Initial Arrays:**

```
previous = [0, 0, 0, 0, 0, 0, 0, 0, 0]  # Size n+1
current = [0, 0, 0, 0, 0, 0, 0, 0, 0]   # Size n+1
```

2. **Iterating Through the Strings:**

> After processing, previous will hold the final row of the DP table (for example, the length of the LCS for the full strings), while current will hold intermediate results that get swapped at the end of each row calculation.
> Finally, previous[n] will hold the length of the LCS of the two strings.
>
> **Key Considerations:**
>
> - Time Complexity: The time complexity remains the same: O(m * n), since you're still performing the same number of computations as with the full DP table.
> - Space Complexity: The space complexity is reduced to O(min(m, n)), because you only need to keep two arrays, each of size n + 1 or m + 1, whichever is smaller.
>
> **Another Space Optimization: Column-Based Optimization**
> Instead of row-based optimization, you could also implement a column-based optimization where you maintain the current column and the previous column in a similar way, but this approach is less common since it typically results in using a larger array in cases where m > n.
>
> **Conclusion:**
> By using two 1D arrays (or two rows), you can optimize the space complexity of the LCS algorithm from O(m * n) to O(min(m, n)). This is a typical optimization technique used in dynamic programming problems when you only need to access a small portion of the DP table at any given time.

Key Point: Needleman-Wunsch Algorithm

- model_name：ChatGPT
- model_version：ChatGPT 4o
- prompt：

> How is the recursive formula of Needleman-Wunsch Algorithm defined?

- response:

> The Needleman-Wunsch algorithm is a dynamic programming algorithm used for global sequence alignment, and its recursive formula is designed to compute an optimal alignment between two sequences, typically protein or nucleotide sequences. It is based on a scoring system where matches, mismatches, and gaps are assigned specific values.
> **Problem Setup**
> Given two sequences:

- Sequence 1: $A = a_1, a_2, ..., a_m$
- Sequence 1: $B = b_1, b_2, ..., b_n$

The goal is to fill a matrix F such that F(i, j) represents the optimal alignment score for the subsequence A[1..i] aligned with B[1..j].

Scoring

- Match: A positive score (say +1).
- Mismatch: A negative score (say -1).
- Gap: A negative score for introducing a gap (say -2).

Initialization

The matrix F has dimensions (m+1)×(n+1), where:

- F(i,0) represents aligning the first i characters of A with an empty sequence B,
- F(0,j) represents aligning the first j characters of B with an empty sequence A.

Initial values are set as:

- F(i,0)=−2i for all i (penalizing gaps in sequence A),
- F(0,j)=−2j for all j (penalizing gaps in sequence B).

Recursive Formula

For each cell F(i,j) in the matrix (with 1≤i≤m and 1≤j≤n), the recursive formula is defined as:

$$F(i,j) = max \begin{cases} F(i-1,j-1) + s(a_i, b_j) & \text{match/mismatch} \\ F(i-1,j) + gap\_penalty & \text{gap in sequence B} \\ F(i,j-1) + gap\_penalty & \text{gap in sequence A} \end{cases}$$

Where:

- $s(a_i, b_j)$is the scoring function for aligning a_i(from sequence A) with b_j(from sequence B):
  - $s(a_i, b_j) = 1$ if $a_i = b_j$(match),
  - $s(a_i, b_j) = -1$ if $a_i \neq b_j$(mismatch)
- Gap penalty is a fixed negative value, say −2.

Explanation of the Formula:

1. Diagonal Move: The first option, $F(i-1, j-1) + s(a_j, b_j)$, corresponds to aligning the last characters of both sequences. If they match, it's a positive score; if they don't, it's a penalty.
2. Vertical Move: The second option, F(i−1,j)+gap penalty, corresponds to introducing a gap

> in sequence B, meaning you align a_i with a gap in B.
> 3. Horizontal Move: The third option, F(i,j−1)+gap penalty, corresponds to introducing a gap in sequence A, meaning you align b_j with a gap in A.
>
> The value of F(i,j) is the maximum of these three options, ensuring that the alignment that maximizes the score is chosen at each step.
> **Final Step**
> Once the matrix is filled, the optimal alignment score is located at F(m,n), which represents the alignment of the entire sequences A and B. The actual alignment can be traced back through the matrix by following the path of decisions that led to this score (this is often done in a separate traceback step).
> **Summary of the Recursive Formula:**
>
> $$F(i,j) = max(F(i-1,j-1)+s(a_i,b_j), F(i-i,j)+g, F(i,j-1)+g)$$
>
> This recursive approach ensures that the Needleman-Wunsch algorithm finds the globally optimal alignment between two sequences, considering all possible alignments with penalties for mismatches and gaps.

- prompt:

> Your answer is basically fine,
> some of positive feedback are:
>
> (1) The Needleman-Wunsch algorithm is a dynamic programming algorithm used for global sequence alignment, and its recursive formula is designed to compute an optimal alignment between two sequences, typically protein or nucleotide sequences. It is based on a scoring system where matches, mismatches, and gaps are assigned specific values.. -- this definition explains the Needleman-Wunsch algorithm and its related concepts, making it easier for the reader to understand the computation of recursive formulas.
>
> (2) Your calculation of the recursive formula is correct.
>
> some of negative feedback are:
>
> Your introduction to Scoring (Match, Mismatch, Gap) is too short and vague, thus making it difficult to understand recursive formulas
>
> Thus, please describe Scoring in the Needleman-Wunsch Algorithm.

- response:

> Thank you for the detailed feedback! You're absolutely right that the explanation of scoring

(match, mismatch, and gap penalties) could be expanded for clarity. Let me elaborate on these scoring components, as they are central to how the Needleman-Wunsch algorithm works.

Scoring in the Needleman-Wunsch Algorithm

In the context of sequence alignment, scoring refers to the values assigned to different types of events that occur when aligning two sequences. The goal is to maximize the total alignment score, so a good choice of scoring scheme can affect the quality of the alignment. The typical scoring system used in the Needleman-Wunsch algorithm includes three primary components:

1. Match Score: This is the score given when two characters (usually nucleotides or amino acids) from the two sequences are aligned with each other and they are the same. The match score is generally a positive value, indicating that a match is beneficial for the alignment.
2. Mismatch Penalty: This is the score given when two characters from the sequences do not match. The mismatch penalty is typically a negative value, reflecting the fact that mismatches are considered less desirable than matches. The magnitude of the mismatch penalty will influence how much mismatches are penalized in the alignment.
3. Gap Penalty: This is the score given when a gap (often represented by a hyphen or underscore in sequence representations) is introduced into one of the sequences. Gaps are used to account for insertions or deletions (indels) in the sequences. The gap penalty is generally a negative value, reflecting that introducing gaps reduces the alignment score. Some variations of the algorithm use different gap penalties for opening a gap (i.e., the first gap in a sequence) and extending an existing gap (i.e., adding additional gaps to the sequence).

These three scoring components are crucial for guiding the dynamic programming approach, as the algorithm seeks to maximize the alignment score by exploring all possible combinations of matching, mismatching, and introducing gaps.

1. **Match Score**
   The match score rewards aligning identical characters between the two sequences. This encourages the algorithm to find regions of similarity between the sequences.

- Example: If the two aligned characters are both "A" (e.g., from sequences A and B, respectively), the match score might be +1.
  - Typical Value: A positive value (e.g., +1 or +2).
- When to use: This score is applied when the characters in both sequences are the same (a match).

2. **Mismatch Penalty**

Mismatch penalties penalize situations where the aligned characters are different. The goal is to avoid mismatches unless the alignment is unavoidable due to gaps or sequence differences. The mismatch penalty is usually a negative value.

- Example: If one sequence has an "A" and the other has a "G" (i.e., $a_i = A$ and \$b_j = G), this would be considered a mismatch, and a penalty would be applied to the score.
    - Typical Value: A negative value (e.g., -1 or -2).
- When to use: This score is applied when the characters in both sequences do not match.

3. **Gap Penalty**
   Gap penalties are introduced when a gap is inserted into one of the sequences. Gaps are a necessary part of sequence alignment because biological sequences often evolve by inserting or deleting nucleotides or amino acids.
   There are typically two types of gap penalties:

- Gap Opening Penalty: The penalty for introducing a new gap. It is usually higher than the penalty for extending an existing gap.
- Gap Extension Penalty: The penalty for extending an already existing gap.

The gap opening penalty discourages introducing new gaps unless absolutely necessary, while the gap extension penalty makes extending existing gaps slightly less penalizing. However, in the basic Needleman-Wunsch algorithm, only a single constant gap penalty is often used, but more complex versions may apply different penalties for gap opening and gap extension.

- Example: If we align a sequence that has a gap at position i in sequence A with a character in sequence B, we incur a penalty.
    - Typical Value: A negative value (e.g., -2 or -3).
    - When to use: This score is applied when a character in one sequence aligns with a gap in the other sequence.
    - Gap Opening vs. Gap Extension: In more sophisticated models, the gap opening penalty could be −5, while the gap extension penalty might be −1. This reflects the higher cost of opening a gap than extending one.

Example Scoring Scheme:
Here’s an example of how these scoring components might be set up:

- Match score: +1
- Mismatch penalty: -1
- Gap penalty: -2 (applied for every gap)

This means:

> - If two characters are the same, they get a score of +1 (match).
> - If two characters differ, they get a score of -1 (mismatch).
> - If a gap is introduced in either sequence, it costs -2 points.
>
> **Putting It All Together:**
> Let’s visualize how these scoring components are applied in the recursive formula:
>
> - When a match occurs (e.g., $a_i = b_j$), the score will be +1, which encourages the alignment of similar characters.
> - When a mismatch occurs (e.g., $a_i \neq b_j$), the score will be -1, discouraging mismatches.
> - When a gap is introduced (either in sequence A or sequence B), the score will be -2, reflecting the penalty for inserting gaps.
>
> These scores influence the recursive formula by determining the cost of moving through the matrix and, ultimately, the total alignment score. The matrix is filled with these penalties and rewards to find the optimal global alignment of the two sequences.
>
> **Final Thoughts**
> In summary, the scoring system in the Needleman-Wunsch algorithm is designed to guide the dynamic programming process toward finding the optimal global alignment by assigning:
>
> - Positive scores for matching characters,
> - Negative scores for mismatches and gaps.
>
> This balance of rewards and penalties directly affects the structure and outcome of the final alignment, determining the degree of similarity between the sequences and how gaps and mismatches are handled in the optimal alignment.
>
> Let me know if you'd like more details or have other questions!

Key Point: Smith-Waterman algorithm

- model_name：ChatGPT
- model_version：ChatGPT 4o
- prompt：

> What is the difference in implementation between Smith-Waterman algorithm and Needleman-Wunsch Algorithm?

- response:

> The Smith-Waterman algorithm and the Needleman-Wunsch algorithm are both dynamic

programming techniques used in bioinformatics for sequence alignment. However, they differ in their approach, use cases, and implementation specifics. Here's a breakdown of the key differences in their implementation:

1. **Type of Sequence Alignment**
    - **Needleman-Wunsch Algorithm (Global Alignment):**
        - Purpose: Used for aligning two sequences globally (i.e., aligning the entire length of both sequences).
        - Initialization: The score matrix is initialized such that the first row and the first column represent cumulative gap penalties (i.e., no alignment between the start of one sequence and any other character in the other sequence).
        - Filling the Matrix: The matrix is filled by considering the alignment of characters across the entire length of both sequences.
        - End Condition: The algorithm forces a full alignment, aligning every character from the first sequence to every character of the second sequence.
    - **Smith-Waterman Algorithm (Local Alignment):**
        - Purpose: Used for aligning sub-sequences of the two sequences (i.e., finding the best local match or region of similarity).
        - Initialization: The first row and the first column are initialized to zero (no alignment or partial alignment penalties).
        - Filling the Matrix: Similar to Needleman-Wunsch but the score in each cell is constrained to never be negative (i.e., the minimum score is zero). This means the algorithm can "restart" a local alignment whenever a negative score is encountered.
        - End Condition: The final alignment is not forced to cover the entire sequence but focuses on the best matching subsequence. The best score can be anywhere in the matrix.
2. **Matrix Initialization**
    - **Needleman-Wunsch:**
        - The first row and first column are initialized with cumulative gap penalties.
        - Formula for initialization
          $M(i, 0) = -i * d$(gap penalty)
          $M(0, j) = -j * d$(gap penalty)
        - Where d is the gap penalty and i, j are the row and column indices respectively.
    - **Smith-Waterman:**
        - The first row and first column are initialized to zero.
        - Formula for initialization:
          $M(i, 0) = 0$ and $M(0, j) = 0$
3. **Filling the Matrix (Recurrence Relation)**
    Both algorithms use the same general recurrence relation for filling the matrix, but they

differ in their approach to scoring:

- Needleman-Wunsch: The matrix cell is filled by considering the three possible alignments (match, mismatch, gap), and the value at each cell is determined by the maximum score of these possibilities.

$$M(i,j) = max \begin{cases} M(i-1,j-1) + score(s_i, t_j) \\ M(i-1,j) + gap\_penalty \\ M(i,j-1) + gap\_penalty \end{cases}$$

- Smith-Waterman: The matrix cell is filled similarly, but with an additional constraint: the score is not allowed to go below zero. This allows the algorithm to start a new local alignment at any point in the matrix.

$$M(i,j) = max \begin{cases} 0 \\ M(i-1,j-1) + score(s_i, t_j) \\ M(i-1,j) + gap\_penalty \\ M(i,j-1) + gap\_penalty \end{cases}$$

4. **Final Score Calculation**
   - Needleman-Wunsch: The final alignment score is found at the bottom-right corner of the matrix (cell M(m,n), where m and n are the lengths of the two sequences).
   - Smith-Waterman: The final alignment score can be anywhere in the matrix, as the algorithm looks for the maximum score found in any cell. This corresponds to the best local alignment score in the matrix.
5. **Backtracking**
   - Needleman-Wunsch: The backtracking process starts at the bottom-right corner and traces back to the top-left corner of the matrix, following the path that leads to the highest score, thereby generating a global alignment.
   - Smith-Waterman: Backtracking also starts at the highest-scoring cell in the matrix, but only continues as long as the score is greater than zero. The backtracking will stop when it reaches a zero score, ensuring that only the highest-scoring local alignment is returned.
6. **Complexity**
   Both algorithms have time and space complexity of O(mn), where m and n are the lengths of the two sequences. However, the Smith-Waterman algorithm often requires more memory because it explores all possible local alignments, whereas Needleman-Wunsch deals with global alignment which doesn't have as much flexibility in terms of starting and ending points.

Summary of Key Differences:

> | Feature | Needleman-Wunsch (Global) | Smith-Waterman (Local) |
> |---|---|---|
> | Alignment Type | Global alignment (entire sequences aligned) | Local alignment (best subsequence match) |
> | Matrix Initialization | Cumulative gap penalties | Zero initialization |
> | Score Recurrence | Max of match/mismatch and gap penalties | Max of match/mismatch, gap penalties, or zero |
> | End Condition | Full alignment of both sequences | Best local alignment (best score anywhere) |
> | Backtracking | Starts from bottom-right, traces to top-left | Starts from highest score, traces to zero |
> | Use Case | When aligning entire sequences | When finding regions of local similarity |
>
> In short, Needleman-Wunsch is best suited for scenarios where the entire sequences need to be aligned, such as when comparing two full genes, while Smith-Waterman is more appropriate for identifying regions of local similarity, such as when trying to find conserved domains within larger sequences.

- prompt:

> Your answer is basically fine,
> some of positive feedback are:
>
> (1) Needleman-Wunsch is best suited for scenarios where the entire sequences need to be aligned, such as when comparing two full genes, while Smith-Waterman is more appropriate for identifying regions of local similarity, such as when trying to find conserved domains within larger sequences. -- this accurately summarizes the essential differences between the two algorithms and makes the reader's understanding of the two more profound.
>
> (2) The differences between the two algorithms are analyzed from many angles, and the analysis is more comprehensive.
>
> some of negative feedback are:
>
> The differences between the two algorithms are explained only from the principle, but there are no concrete examples, so it is still difficult to understand
>
> Thus, please Give an example and solve it with two algorithms.

- response:

> Thank you for the detailed feedback! I completely understand that an example would make the explanation clearer and more concrete. Let's walk through an example to illustrate both Needleman-Wunsch and Smith-Waterman algorithms step by step. We'll use the following simple sequences:
>
> - Sequence 1: GATTACA
> - Sequence 2: GCATGCU
>
> We will use the following scoring scheme:
>
> - Match score: +1
> - Mismatch penalty: −1
> - Gap penalty: −2
>
> **Step 1: Needleman-Wunsch (Global Alignment)**
> We'll start by constructing a global alignment for the two sequences, using the Needleman-Wunsch algorithm.
> **Initialization:**
> Create a matrix where the dimensions are (m+1) by (n+1), where m is the length of Sequence 1 and n is the length of Sequence 2. So, the matrix will be 8x8 (since both sequences have a length of 7).
> The first row and first column are initialized with gap penalties:
>
> $$\left\{\begin{array}{cccccccc} 0 & -2 & -4 & -6 & -8 & -10 & -12 & -14 \\ -2 & & & & & & & \\ -4 & & & & & & & \\ -6 & & & & & & & \\ -8 & & & & & & & \\ -10 & & & & & & & \\ -12 & & & & & & & \\ -14 & & & & & & & \end{array}\right\} \tag{1}$$
>
> **Filling the Matrix:**
> Now, we fill the matrix using the recurrence relation for Needleman-Wunsch:
>
> $$M(i,j) = max\begin{cases} M(i-1,j-1) + score(s_i,t_j) \\ M(i-1,j) + gap\_penalty \\ M(i,j-1) + gap\_penalty \end{cases}$$
>
> Where $score(s_i,t_j)$ is +1 for a match, -1 for a mismatch, and -2 for a gap.

After filling the entire matrix, it will look like this:

$$\left\{\begin{matrix} 0 & -2 & -4 & -6 & -8 & -10 & -12 & -14 \\ -2 & 1 & -1 & -3 & -5 & -7 & -9 & -11 \\ -4 & -1 & 2 & 0 & -2 & -4 & -6 & -8 \\ -6 & -3 & 0 & 3 & 1 & -1 & -3 & -5 \\ -8 & -5 & -2 & 1 & 4 & 2 & 0 & -2 \\ -10 & -7 & -4 & -1 & 2 & 5 & 3 & 1 \\ -12 & -9 & -6 & -3 & 0 & 3 & 6 & 4 \\ -14 & -11 & -8 & -5 & -2 & 1 & 4 & 7 \end{matrix}\right\} \tag{1}$$

Backtracking:

Now, we start from the bottom-right corner of the matrix and backtrack to the top-left corner to find the optimal alignment.

- From (7,7) (score 7), we trace back to (6,6), (5,5), (4,4), (3,3), (2,2), (1,1), (0,0). Every step follows the path that leads to the maximum score.
- This gives us the global alignment:

```
Sequence 1:   GATTACA
Sequence 2:   GCATGCU
Alignment:    G-ATTACA
              GCA-TGCU
```

This shows that the entire sequences are aligned with gaps where necessary.

Step 2: Smith-Waterman (Local Alignment)

Now, we apply the Smith-Waterman algorithm to find the best local alignment. This algorithm is used to find the best matching subsequences (not requiring full alignment of both sequences).

Initialization:

Again, we initialize the matrix in the same way as we did for Needleman-Wunsch, but the first row and column are initialized to 0 instead of gap penalties.

$$\left\{\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & & & & & & & \\ 0 & & & & & & & \\ 0 & & & & & & & \\ 0 & & & & & & & \\ 0 & & & & & & & \\ 0 & & & & & & & \\ 0 & & & & & & & \end{matrix}\right\} \tag{1}$$

Filling the Matrix:
Just like Needleman-Wunsch, we fill the matrix using the same recurrence relation. However, we add a 0 as the minimum score in each cell:

$$M(i,j) = max \begin{cases} 0 \\ M(i-1, j-1) + score(s_i, t_j) \\ M(i-1, j) + gap\_penalty \\ M(i, j-1) + gap\_penalty \end{cases}$$

After filling the matrix, we get the following result (here only a subset is shown for simplicity):

$$\left\{\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 4 & 3 & 2 & 1 \\ 0 & 0 & 0 & 1 & 3 & 5 & 4 & 3 \\ 0 & 0 & 0 & 0 & 2 & 4 & 6 & 5 \\ 0 & 0 & 0 & 0 & 1 & 3 & 5 & 7 \end{matrix}\right\} \tag{1}$$

Backtracking:
We backtrack from the highest score in the matrix, which is 7 at (7,7). Since this is the best score, we trace back to the cell where we got this score. The backtracking process stops when we hit a score of 0.
The best local alignment is:

```
Sequence 1:   TACA
Sequence 2:   TGCU
Alignment:    TACA
              TGCU
```

Conclusion:

- Needleman-Wunsch (Global Alignment): The entire sequences are aligned, with gaps introduced to ensure full alignment of both sequences:

> ```
> Sequence 1:  GATTACA
> Sequence 2:  GCATGCU
> Alignment:   G-ATTACA
>              GCA-TGCU
> ```
>
> - Smith-Waterman (Local Alignment): The best local alignment finds a subsequence match between part of Sequence 1 (TACA) and part of Sequence 2 (TGCU):
>
> ```
> Sequence 1:  TACA
> Sequence 2:  TGCU
> Alignment:   TACA
>              TGCU
> ```
>
> In this example, Needleman-Wunsch forces a global alignment of both sequences, while Smith-Waterman focuses on the best local alignment (subsequence matching), which is useful when there are regions of significant similarity in otherwise different sequences.

Key Point: Scoring a MSA

- model_name： ChatGPT
- model_version： ChatGPT 4o
- prompt：

> How are scoring functions (such as Sum of Pairs and Entropy based Score) selected and applied in MSA?

- response:

> Scoring functions in Multiple Sequence Alignment (MSA) are selected and applied based on the biological objective, data characteristics, and computational efficiency. Below is an explanation of two common scoring functions—Sum of Pairs (SP) score and Entropy-based score—and how they are selected and applied.
>
> 1. **Sum of Pairs (SP) Score**
>   **Definition:**
>   The Sum of Pairs score evaluates the alignment quality by summing substitution scores or penalties for all possible pairs of sequences at every aligned position.
>   **Formula:**
>   Given an alignment of N sequences and a scoring matrix (e.g., BLOSUM or PAM), the SP score is computed as:

$$SP = \sum_{i=1}^{L} \sum_{1 \leq k < j \leq N} S(x_{k,i}, x_{j,i})$$

where:

- L is the alignment length,
- $x_{k,i}$ and $x_{j,i}$ are the residues of sequences k and j at position i,
- $S(a, b)$ is the score from the substitution matrix for residues a and b.

Application:

- **Selection Criteria:** SP is favored when:
  - Pairwise sequence similarity is high (low divergence).
  - A substitution scoring matrix accurately reflects evolutionary relationships.
- **Advantages:**
  - Biologically interpretable: reflects evolutionary conservation.
  - Simplicity: straightforward to compute.
- **Disadvantages:**
  - Does not consider the position-specific or global context.
  - Ignores column-wise conservation metrics.
- **Use Cases:**
  - Progressive alignment methods like ClustalW use SP for pairwise alignment stages.
  - Appropriate when analyzing closely related sequences.

2. **Entropy-Based Score**

Definition:

This score measures column-wise sequence conservation in the alignment. High entropy indicates variability, while low entropy suggests conservation.

Formula:

The entropy H for a column i is:

$$H_i = -\sum_{i \in \mathcal{A}} p_{a,i} log_2 p_{a,i}$$

where:

- $\mathcal{A}$ is the set of possible residues,
- $p_{a,i}$ is the frequency of residue a in column i.

  The total entropy score for an alignment is:

$$H = \sum_{i=1}^{L} H_i$$

> **Application:**
>
> - **Selection Criteria:** Entropy-based scores are chosen when:
>   - Conservation of specific residues or regions is critical.
>   - Understanding functional or structural features is the focus.
> - **Advantages:**
>   - Highlights highly conserved regions, which often correspond to functional or structurally critical sites.
>   - Reflects sequence diversity at the column level.
> - **Disadvantages:**
>   - Sensitive to sequence length and alignment gaps.
>   - Does not directly account for evolutionary models.
> - **Use Cases:**
>   - Used in sequence motif discovery and structural alignment contexts.
>   - Useful for identifying conserved motifs in transcription factor binding sites or enzyme active sites.
>
> **Practical Selection and Application**
>
> 1. **Biological Objective:**
>   - High Conservation: Use entropy-based scores to highlight conserved regions in diverse sequences.
>   - Overall Similarity: Use SP scores to measure general sequence similarity.
> 2. **Nature of the Data:**
>   - For closely related sequences, SP score is more reliable since evolutionary relationships can be modeled with substitution matrices.
>   - For diverse sequences, entropy-based scoring helps identify conserved regions amidst variability.
> 3. **Computational Strategy:**
>   - SP scoring is computationally intensive for large datasets due to pairwise comparisons but is straightforward to compute.
>   - Entropy-based scoring is computationally efficient but may require normalization when comparing alignments of different lengths.
> 4. **Integration with MSA Algorithms:**
>   - SP scores are integral to progressive alignment methods and guide-tree generation.
>   - Entropy-based scores are often used post-alignment for analysis and validation, ensuring functional or structural relevance.
>
> By carefully aligning the scoring function with the biological goals and computational constraints, researchers can ensure meaningful and efficient MSA outcomes.

- prompt:

> Your answer is basically fine,
> some of positive feedback are:
>
> (1) The calculation formulas of the two scoring functions are accurate and have reasonable explanations.
>
> (2) The differences between the two scoring functions are analyzed from different angles, which makes the understanding of the two functions more profound.
>
> some of negative feedback are:
>
> Sum of Pairs and Entropy based Score is just a few examples of scoring functions I have given. You did not give the introduction of other scoring functions, which is a little one-sided.
>
> Thus, please give me some other scoring functions.

- response:

> Thank you for the feedback! You're absolutely right that Sum of Pairs and Entropy-based scores are just two examples of scoring functions used in MSA. There are several other scoring functions, each suited to different types of data and alignment objectives. Here's an expanded overview of some other commonly used scoring functions in Multiple Sequence Alignment:
>
> 1. **Log-Odds Score (or Log-Probability Score)**
>   **Definition:**
>   The log-odds score is a statistical measure that compares the likelihood of two residues being aligned by chance versus the likelihood of them being aligned due to evolutionary relationships.
>   **Formula:**
>   Given a substitution matrix $S(a, b)$ and the background probabilities of residues $p(a)$ and $p(b)$, the log-odds score for residues a and b is:
>
> $$Log - Odds(a, b) = log(\frac{S(a, b)}{p(a)p(b)})$$
>
>   Where:
>   - $S(a, b)$ is the score for aligning residues a and b,
>   - $p(a)$ and $p(b)$ are the probabilities of residues a and b appearing in the sequences (often taken from a large sequence database or a specific alignment).
>
>   **Application:**
>   - **Selection Criteria:** This score is commonly used in scoring matrices such as BLOSUM or PAM, which estimate evolutionary changes between residue pairs.

- **Advantages:**
    - Considers evolutionary likelihood, rather than just simple substitution penalties.
    - Effective for large datasets with varying levels of divergence.
- **Disadvantages:**
    - Requires accurate background residue frequencies to be effective.
- **Use Cases:**
    - Typically used in algorithms like BLAST and ClustalW.

2. **Position-Specific Scoring Matrix (PSSM)**
   Definition:
   A PSSM (also called a Position-Specific Scoring Profile or Position Weight Matrix, PWM) is a matrix used to score alignments by considering the frequency of each residue at every alignment position.
   Formula:
   For each position i in the alignment, the score is calculated as the sum of log-odds scores for each residue:

$$PSSM\_Score(i) = \sum_{a \in \mathcal{A}} p_{a,i} * log(\frac{p_{a,i}}{p(a)})$$

   Where:
   - $p_{a,i}$ is the observed frequency of residue a at position i in the alignment,
   - $p(a)$ is the background frequency of residue a.

   Application:
   - **Selection Criteria:** PSSMs are used when you need a position-specific scoring function that reflects residue frequency distributions across a set of sequences.
   - **Advantages:**
       - Allows for flexible modeling of conserved and variable residues across positions.
       - Good for aligning sequences of diverse lengths or varying levels of conservation.
   - **Disadvantages:**
       - Can be computationally expensive to build and apply, especially for large datasets.
   - **Use Cases:**
       - Motif discovery in regulatory sequences.
       - Used in PSI-BLAST, a variant of BLAST.

3. **BLOSUM and PAM Matrices**
   Definition:
   BLOSUM (Blocks Substitution Matrix) and PAM (Point Accepted Mutation) are families of

scoring matrices used to score sequence alignments. They provide substitution scores based on observed evolutionary changes in protein families.

- BLOSUM: The BLOSUM series is based on observed alignments of protein sequences that share a certain level of identity. For example, BLOSUM62 is based on sequences with ≥62% identity.
- PAM: The PAM matrices are based on the model of evolution, assuming a constant rate of mutations and calculating substitution frequencies over evolutionary time. PAM1 corresponds to one point mutation per 100 amino acid residues.

Formula:

Both BLOSUM and PAM matrices provide substitution scores, which are applied to each pair of aligned residues. For residues a and b, the score is directly given by:

$$S(a, b) = \text{score from the matrix}$$

Application:

- **Selection Criteria:**
  - BLOSUM matrices are used for local alignments where sequences have similar lengths and identities (e.g., in BLAST).
  - PAM matrices are better for global alignments of highly diverged sequences, reflecting evolutionary distance.
- **Advantages:**
  - BLOSUM matrices are widely used in practical sequence alignment tools.
  - PAM matrices reflect the evolutionary distance, which can be more suitable for older divergences.
- **Disadvantages:**
  - PAM matrices may not perform well for short sequences or those with high identity.
  - BLOSUM matrices may be too sensitive to short-range evolutionary events.
- **Use Cases:**
  - BLOSUM is used in BLAST and ClustalW for aligning protein sequences.
  - PAM is used in Needleman-Wunsch and Smith-Waterman for global sequence alignment.

4. **Gap Penalties (Opening and Extension)**

Definition:

Gap penalties are used to account for the introduction of gaps (insertions or deletions) into an alignment. These penalties typically consist of two parts:

- Gap opening penalty: The penalty for starting a gap.
- Gap extension penalty: The penalty for extending an already existing gap.

Formula:

The total gap penalty for a gap of length g is typically modeled as:

$$\text{Gap Penalty} = g_o + (g - 1) * g_e$$

Where:

- g_o is the gap opening penalty,
- g_e is the gap extension penalty.

Application:

- **Selection Criteria:**
  - Gap penalties are crucial for any sequence alignment, and different gap penalty strategies are chosen based on the alignment's goal (e.g., detecting mutations vs. aligning distantly related sequences).
- **Advantages:**
  - Adjustable to fine-tune alignment results.
  - Helps balance the alignment of long versus short gaps.
- **Disadvantages:**
  - Setting appropriate values for opening and extension penalties can be challenging and depends on the alignment's biological context.
- **Use Cases:**
  - Used in almost every alignment algorithm, including Smith-Waterman, Needleman-Wunsch, and ClustalW.

5. **Evolutionary-Based Scores**

Definition:

Evolutionary-based scoring methods use models of molecular evolution (such as the Jukes-Cantor or Kimura models) to score alignments based on expected evolutionary changes. These methods account for different mutation rates for different types of substitutions (e.g., transitions vs. transversions).

Formula:

The evolutionary score for a pair of residues a and b is based on the likelihood of their substitution given a particular evolutionary model. For example, the Jukes-Cantor model gives the probability of observing a mutation based on the number of substitutions per site.

For example, the Jukes-Cantor model's substitution probability can be expressed as:

$$P(a \to b) = \frac{1}{4}(1 + 3e^{-d})$$

Where:

- d is the number of substitutions per site (genetic distance).

> **Application:**
>
> - **Selection Criteria:** Used when it's necessary to account for evolutionary models of mutation and substitution.
> - **Advantages:**
>   - More accurate for distantly related sequences as they model mutation processes.
> - **Disadvantages:**
>   - More computationally complex compared to simple scoring matrices.
> - **Use Cases:**
>   - Used in PhyML or RAxML for phylogenetic tree reconstruction.
>
> **Conclusion**
> In addition to Sum of Pairs and Entropy-based scores, various other scoring functions (such as Log-Odds, PSSM, BLOSUM/PAM matrices, and Evolutionary-based scores) offer unique advantages depending on the type of sequence data and alignment goal. The selection of the scoring function often involves balancing biological accuracy with computational efficiency, as well as ensuring that the function is suitable for the evolutionary context of the sequences being analyzed.

Key Point: Dynamic programming for MSA

- model_name：ChatGPT
- model_version：ChatGPT 4o
- prompt：

> How to calculate the time complexity of Multiple sequence alignment using dynamic programming methods?

- response:

> **Time Complexity of Multiple Sequence Alignment (MSA) using Dynamic Programming**
> Multiple Sequence Alignment (MSA) is a fundamental problem in bioinformatics where the goal is to align three or more sequences of DNA, RNA, or protein to identify regions of similarity. The most common approaches for MSA involve dynamic programming (DP), especially when using algorithms like progressive alignment, as seen in tools like ClustalW, MAFFT, and MUSCLE.
> Let’s break down how to compute the time complexity for MSA using dynamic programming.
>
> 1. **Basic DP for Pairwise Sequence Alignment**
>   To understand MSA complexity, we first need to recall the time complexity for pairwise

sequence alignment using dynamic programming, typically done with the Needleman-Wunsch (for global alignment) or Smith-Waterman (for local alignment) algorithms.
For two sequences of lengths m and n, the dynamic programming table (DP table) will have m+1 rows and n+1 columns. The time complexity of filling this table is:
   - Time Complexity (pairwise alignment): O(m×n)
2. **Extending to Multiple Sequences**
   When aligning more than two sequences, the time complexity increases significantly. The most straightforward dynamic programming approach for multiple sequences is pairwise progressive alignment. Here's the general process:
   - You align two sequences first, then progressively align the third sequence to the result of the previous alignment, and so on.
   - The computational cost is dominated by the pairwise alignments at each step.

   Let's say you have k sequences, and each sequence i has length L_i. In the worst case, we need to align all k sequences pairwise.
3. **Time Complexity for Pairwise MSA**
   The most straightforward strategy for MSA using dynamic programming is pairwise progressive alignment:
   - Step 1: First, align the first two sequences. This takes $O(L_1 * L_2)$, where L_1 and L_2 are the lengths of the first two sequences.
   - Step 2: Then, align the result with the third sequence. Assuming the result of the first alignment has length L_{result}, aligning this result with the third sequence takes $O(L_{result} * L_3)$.
   - Step 3: Continue this process for all k sequences.

   For simplicity, assume each sequence is of approximately the same length L, so $L_1 \approx L_2 \approx L_3 \approx ... \approx L$. In this case, each progressive alignment takes time proportional to the product of the lengths of the sequences being aligned.
4. **Cumulative Time Complexity for Multiple Sequences**
   If we align k sequences with lengths L, the total time complexity is the sum of the costs of the pairwise alignments at each step. After aligning two sequences, the length of the resulting aligned sequence will grow to approximately the sum of the original sequence lengths, i.e., $L_1 + L_2$.
   For k sequences, this cumulative complexity can be approximated as:

$$O(L * L_1 + L * L_2 + ... + L * L_k) = O(k^2 * L^2)$$

   This assumes that each progressive alignment involves sequences of roughly the same length. If the sequence lengths vary, then the total complexity would be adjusted accordingly for each pairwise alignment.
5. **Refinement: Better Algorithms for MSA**

> There are more efficient algorithms to align multiple sequences, such as CLUSTALW, MAFFT, and MUSCLE, which often use guide trees and heuristic methods to avoid the full $O(k^2 * L^2)$ complexity. These algorithms generally aim to reduce the number of pairwise alignments performed, leveraging techniques like divide and conquer or clustering to reduce the search space.
> However, in the worst-case scenario where the dynamic programming approach is used without any optimization (e.g., a simple progressive alignment), the time complexity of MSA remains:
>
> $$O(k^2 * L^2)$$
>
> where:
>
> - k is the number of sequences,
> - L is the length of the sequences.
>
> 6. **Final Notes on Complexity**
>   - Space Complexity: The space complexity of the dynamic programming algorithm for pairwise alignment is typically O(m×n) for two sequences. For MSA, depending on the approach, space complexity can be proportional to the size of the DP table, which grows rapidly as more sequences are added.
>   - Heuristic Improvements: Many practical MSA tools don't use exact dynamic programming but rather employ heuristic methods (such as progressive alignment with a guide tree, as in ClustalW), which can significantly reduce the time complexity in practice.
>
> Thus, while the theoretical worst-case complexity of dynamic programming for MSA is $O(k^2 * L^2)$, many MSA methods implement optimizations to handle larger datasets efficiently.

- prompt:

> Your answer is basically fine,
> some of positive feedback are:
>
> (1) To understand MSA complexity, we first need to recall the time complexity for pairwise sequence alignment using dynamic programming, typically done with the Needleman-Wunsch (for global alignment) or Smith-Waterman (for local alignment) algorithms. --this introduces Basic DP for Pairwise Sequence Alignment, so that readers can have a basic understanding of DP method of MSA.
>
> (2) There are more efficient algorithms to align multiple sequences, such as CLUSTALW, MAFFT, and MUSCLE, which often use guide trees and heuristic methods to avoid the full

> $O(k^2 * L^2)$complexity. --this illustrates how you used pairwise comparisons for your previous calculations of time complexity, as well as other good algorithms for MSA.
>
> some of negative feedback are:
>
> In addition to the pairwise comparison method, other methods can be used. For example, For three sequences X, Y, Z, define dp[i, j, k] = score of optimum alignment among X[1..i], Y[1..j] and Z[1..k]. Devise recurrence relations for dp[i, j, k] and dp[i, j, k] is the maximum out of all 7 possibilities.
>
> Thus, please calculate the time complexity of my example algorithm.

- response:

> Thanks for the positive feedback! I understand your point regarding the need to explore the time complexity for multiple sequence alignment beyond just pairwise comparisons. Let’s break down the alternative method you've suggested, which considers three sequences using dynamic programming.
> **Approach: DP for Three Sequences**
> To align three sequences X, Y, and Z, we can define a 3D dynamic programming table where each entry dp[i,j,k] represents the optimal score for aligning the first i characters of sequence X, the first j characters of sequence Y, and the first k characters of sequence Z.
> **Step 1: Recurrence Relation**
> For the recurrence, we need to consider all possible ways of aligning the characters from X, Y, and Z at each step. There are 7 possible ways to match or gap the characters from X, Y, and Z:
>
> 1. Align X[i−1] with Y[j−1] and Z[k−1].
> 2. Align X[i−1] with Y[j−1] and gap Z.
> 3. Align X[i−1] with Z[k−1] and gap Y.
> 4. Align Y[j−1] with Z[k−1] and gap X.
> 5. Align X[i−1] and gap Y and gap Z.
> 6. Align Y[j−1] and gap X and gap Z.
> 7. Align Z[k−1] and gap X and gap Y.
>
> The recurrence relation for filling the table is:

$$dp[i,j,k] = max \begin{cases} dp[i-1,j-1,k-1] + score(X[i-1],Y[j-1],Z[k-1]), \\ dp[i-1,j-1,k] + gap\_penalty \\ dp[i-1,j,k-1] + gap\_penalty \\ dp[i,j-1,k-1] + gap\_penalty \\ dp[i-1,j,k] + gap\_penalty \\ dp[i,j-1,k] + gap\_penalty \\ dp[i,j,k-1] + gap\_penalty \end{cases}$$

Where score(X[i−1],Y[j−1],Z[k−1]) refers to the alignment score (usually 1 for a match, -1 for a mismatch) and the gap penalty is the penalty associated with introducing a gap in one of the sequences (commonly −1).

Step 2: Time Complexity of the Algorithm

Now, let’s calculate the time complexity based on the structure of the dynamic programming table.

- The DP table has dimensions m×n×p, where:
  - m is the length of sequence X,
  - n is the length of sequence Y,
  - p is the length of sequence Z.

Thus, the total number of entries in the DP table is O(m×n×p).

- For each entry, we compute the maximum of 7 values (each corresponding to one of the 7 possible ways to align characters from the three sequences). This requires constant time O(1) per entry.

Therefore, the time complexity for computing the entire table is:

$$O(m * n * p)$$

Step 3: Generalizing to k Sequences

The case for three sequences X, Y, and Z can be generalized to k sequences. The naive dynamic programming approach would extend the 3D table to k dimensions, with each dimension corresponding to one sequence.

For k sequences X_1, X_2, ..., X_k of lengths L_1, L_2, ..., L_k, we would need a k-dimensional DP table of size:

$$L_1 * L_2 * ... * L_k$$

> At each cell in the k-dimensional DP table, we would calculate the maximum of $2^k - 1$ possible alignments (as each sequence can either be aligned with another or have a gap). This results in a recurrence similar to the three-sequence case but generalized for k sequences.
> Thus, the time complexity of aligning k sequences with lengths L_1, L_2, ..., L_k using the naive k-dimensional dynamic programming approach is:
>
> $$O(L_1 * L_2 * ... * L_k)$$
>
> **Example: Time Complexity for 3 Sequences**
> If we specifically look at the case of three sequences X, Y, and Z, each of length L, the time complexity for filling the 3D table is:
>
> $$O(L^3)$$
>
> This is because we have a 3D DP table of size L×L×L, and each entry requires O(1) time to compute.
> **Final Time Complexity:**
>
> 1. For three sequences X, Y, and Z, the time complexity is:
>
> $$O(L^3)$$
>
> 2. For k sequences, the naive approach (using k-dimensional DP) results in:
>
> $$O(L_1 * L_2 * ... * L_k)$$
>
> **Conclusion**
> The dynamic programming approach for multiple sequence alignment can be generalized to multiple sequences, and the time complexity grows exponentially as the number of sequences increases. For
> k sequences, the complexity is $O(L_1 * L_2 * ... * L_k)$, and for three sequences, it is $O(L^3)$.
> This is a naive approach. Heuristic methods, like those used in CLUSTALW, MAFFT, and MUSCLE, typically reduce the time complexity by using progressive alignment strategies and guide trees.