

Assignment 2: Dynamic Programming

Forrest Yan Sun (UF ID: 47948015)

November 23, 2025

Problem 1: Weighted Approximate Common Substring

1. Algorithm Design

(1) Optimization Function Definition

Let $OPT(i, j)$ denote the **maximum score** of a common substring that **ends strictly** at index i of string $X = x_1x_2 \dots x_m$ and index j of string $Y = y_1y_2 \dots y_n$.

(2) Bellman Equation

The recurrence relation incorporates a local alignment strategy. If the extension of a previous substring results in a negative score due to a mismatch penalty, the score is reset to 0, effectively starting a new substring at the current position.

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

where the scoring function is defined as:

$$\text{score}(x_i, y_j) = \begin{cases} w_{x_i} & \text{if } x_i = y_j \quad (\text{Match}) \\ -\delta & \text{if } x_i \neq y_j \quad (\text{Mismatch}) \end{cases}$$

(3) Justification for Correctness

Since gaps are not allowed (as specified in the problem statement), a common substring ending at (i, j) must necessarily extend the substring ending at $(i-1, j-1)$. The inclusion of 0 in the maximization step ($\max(0, \dots)$) allows the algorithm to discard prefixes that have a negative cumulative score. This ensures that we always find the substring with the highest local score, effectively implementing a gapless variant of the Smith-Waterman algorithm.

(4) Extracting the Solution

The optimal value is not necessarily found at $OPT(m, n)$. Instead, the maximum score of the best substring is the global maximum in the DP table:

$$\text{MaxScore} = \max_{1 \leq i \leq m, 1 \leq j \leq n} OPT(i, j)$$

To recover the substring, we locate the indices (i^*, j^*) corresponding to this maximum score and trace back diagonally to $(i^* - 1, j^* - 1)$ until the value of OPT drops to 0.

2. Complexity Analysis

- **Time Complexity:** $O(mn)$. The algorithm fills a table of size $(m+1) \times (n+1)$. Each entry takes $O(1)$ time to compute as it only depends on one previous cell $(i-1, j-1)$.
- **Space Complexity:** $O(mn)$. We require a 2D array of size $(m+1) \times (n+1)$ to store the scores.

3. Experiments

Experimental Setup:

- **Scenario 1:** Constant weights ($w = 1$) and fixed penalty ($\delta = 10$). Verified on the example strings.
- **Scenario 2:** Variable weights based on real English letter frequencies from Cornell University Dept. of Math [1]. The penalty δ varied between the minimum and maximum weights (0.07 to 12.02). A fixed random seed was used to ensure reproducibility.

Results:

Table 1: Problem 1 Experimental Results (Scenario 2)

Delta	Max Score	Length	Substring (First 10 chars)
0.07	195.71	842	WPXBSIKKMF...
1.40	37.65	6	IQZEET
2.73	35.00	6	IQZEET
4.05	33.14	3	EET
5.38	33.14	3	EET
...
12.02	33.14	3	EET

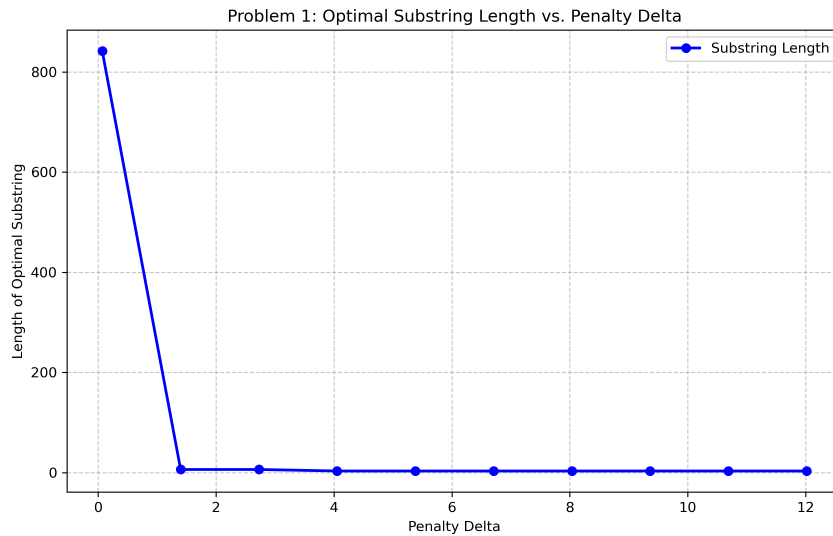


Figure 1: Optimal Substring Length vs. Penalty Delta. The sharp drop indicates a transition from noisy alignment to strict matching.

Analysis: As shown in Figure 1, when δ is very small (0.07), the penalty for mismatches is negligible, allowing the algorithm to extend the substring to length 842 (covering almost the entire string). As δ increases to 1.40, the length drops sharply to 6. For $\delta \geq 4.05$, the result stabilizes at

length 3 with the substring "EET". This represents a **perfect match** (no mismatches) with high-weight characters. Since perfect matches incur no penalty, their score remains constant regardless of how high δ becomes.

Problem 2: Largest Zero Sub-matrix

1. Algorithm Design

(1) Optimization Function Definition

Let $OPT(i, j)$ denote the **side length** of the largest square sub-matrix consisting entirely of zeros, whose **bottom-right corner** is located at position (i, j) in the matrix B .

(2) Bellman Equation

A square of zeros of size $k \times k$ ending at (i, j) can only exist if the overlapping squares at $(i - 1, j)$, $(i, j - 1)$, and $(i - 1, j - 1)$ allow it, and the current cell $B[i][j]$ is zero.

$$OPT(i, j) = \begin{cases} 0 & \text{if } B[i][j] \neq 0 \\ 1 + \min(OPT(i - 1, j), OPT(i, j - 1), OPT(i - 1, j - 1)) & \text{if } B[i][j] = 0 \end{cases}$$

(3) Correctness Justification

If $B[i][j] = 0$, the size of the square ending at (i, j) is constrained by the smallest square ending at its three neighbors (top, left, top-left). By taking the minimum of these neighbors plus one, we extend the square size valid in all three directions.

(4) Extracting the Solution

The size of the largest zero sub-matrix is simply the maximum value found in the DP table:
 $k_{max} = \max_{i,j} OPT(i, j)$.

2. Complexity Analysis

- **Time Complexity:** $O(mn)$. We iterate through the $m \times n$ matrix exactly once with constant-time operations per cell.
- **Space Complexity:** $O(mn)$. We use a 2D array for DP values.
- **Optimization:** The problem statement suggests conserving memory. Since the max square size for a 1000×1000 matrix can exceed 127, a **byte** is insufficient. We used a **short** (2 bytes) array for the DP table and a **byte** array for the input matrix to minimize footprint while preventing overflow.

3. Experiments

We simulated 5 datasets using random boolean values (0 and 1) with a fixed seed. The dataset sizes range from 10×10 to 1000×1000 .

Table 2: Problem 2 Experimental Data

Elements ($m \times n$)	Time (ms)	Memory (MB)	Max Square Size
100	0.0068	0.0003	2
1,000	0.0494	0.0029	3
10,000	0.4667	0.0286	4
100,000	3.2043	0.2861	4
1,000,000	5.3803	2.8610	4

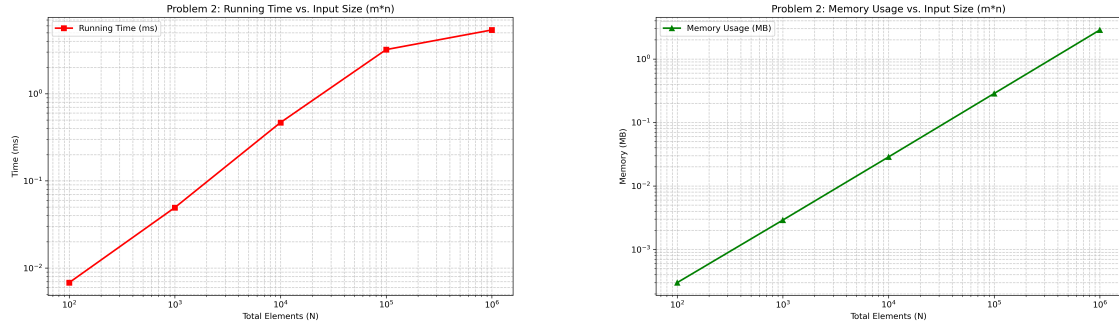


Figure 2: Left: Running Time (log-log scale). Right: Theoretical Memory Usage (log-log scale).

Results Analysis:

- **Running Time:** The time complexity is confirmed to be $O(mn)$. As shown in Figure 2 (Left), the relationship is linear. The slight deviation at larger inputs (e.g., 1M elements taking only 5.38ms) is attributed to JVM Just-In-Time (JIT) optimizations and cache locality, which improve throughput for large contiguous arrays.
- **Memory Usage:** Direct JVM memory measurement is noisy for small allocations. Therefore, we plotted the **theoretical allocated memory** ($Input_{byte} + DP_{short} = 3 \times m \times n$ bytes). Figure 2 (Right) shows a perfect linear relationship, verifying the $O(mn)$ space complexity.

References

- [1] Cornell University Dept. of Math. *Letter Frequencies*. Available at: <https://pi.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html>