SUMMARY

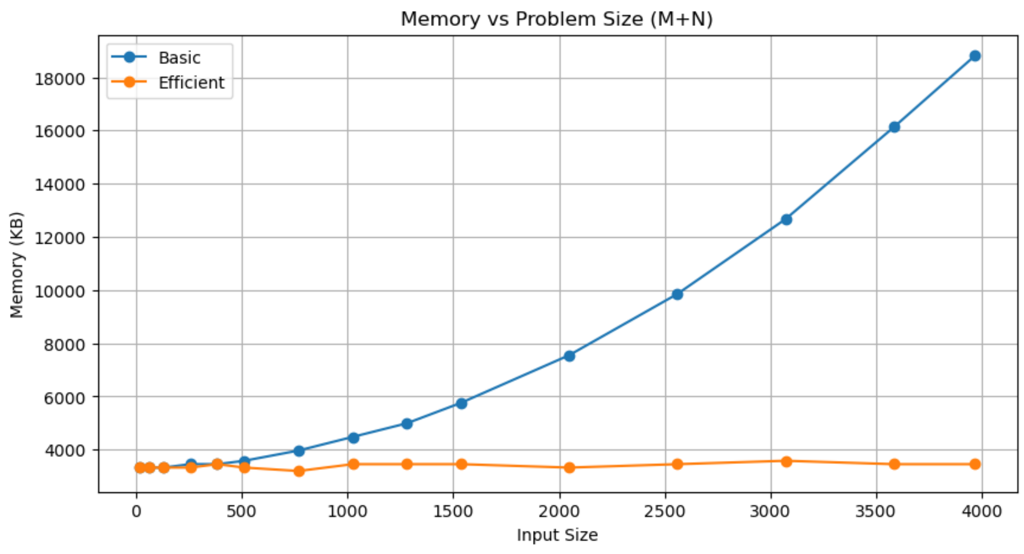
USC ID/s: 3772701387

Datapoints

M+N	Time in MS (Basic)	Time in MS (Efficient)	Memory in KB (Basic)	Memory in KB (Efficient)
16	0.016	0.054	3328	3328
64	0.064	0.181	3328	3328
128	0.198	0.599	3328	3328
256	0.755	2.326	3456	3328
384	1.989	4.163	3456	3456
512	2.778	7.321	3584	3328
768	7.046	16.566	3968	3200
1024	10.896	28.761	4480	3456
1280	17.171	49.052	4992	3456
1536	24.718	65.613	5760	3456
2048	42.323	123.346	7552	3328
2560	74.849	181.759	9856	3456
3072	105.36	250.536	12672	3584
3584	133.15	334.677	16128	3456
3968	190.57	439.651	18816	3456

Insights

Graph1 – Memory vs Problem Size (M+N)



Nature of the Graph (Logarithmic/ Linear/ Polynomial/ Exponential)

- Basic: O(MN) – Pseudo Polynomial
- Efficient: O(M+N) – Polynomial (Linear)

Explanation:

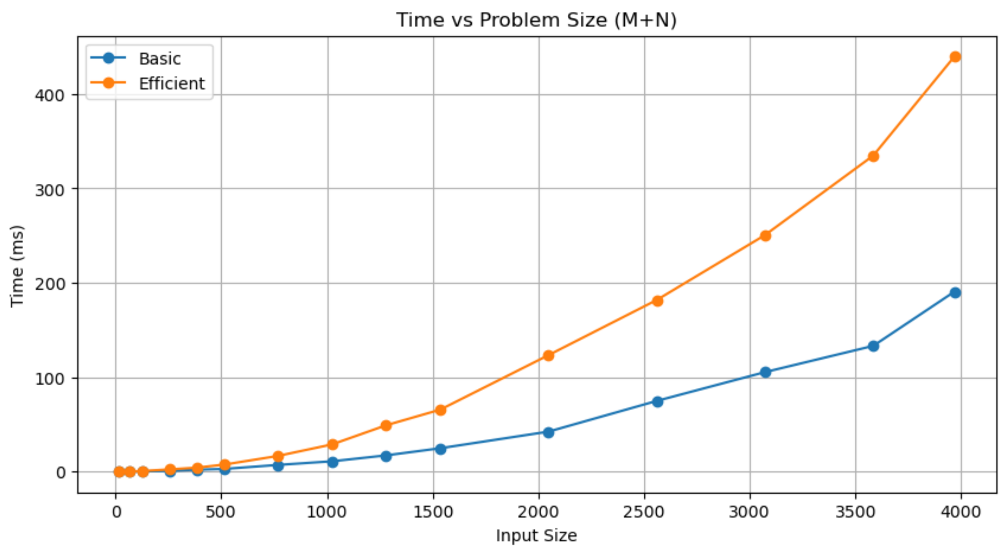
Base Algorithm Memory Use Analysis:

The primary memory requirement comes from storing the (M+1) x (N+1) dynamic programming table itself. Each cell stores a score (an integer). Additional memory is used for storing the input strings (O(M+N)) and the output aligned strings (which can be up to O(M+N) in length). However, the DP table is the dominant factor for memory usage, leading to an O(MN) space complexity.

Efficient Algorithm Memory Use Analysis:

The linear-space computations (NWScore) only need to store two rows (or columns) of the DP table at a time. If the DP rows are computed along the shorter dimension of the current subproblem, this uses O(min(M_current, N_current)) space. In the implementation, NWScore uses space proportional to the length of the Y string it's processing, which is O(N) for the largest calls. The recursion depth is typically O(log M) or O(log N). Each recursive call adds a small amount of memory to the call stack. Memory is also needed for the input strings (O(M+N)) and the output aligned strings (O(M+N)). The dominant factor for auxiliary space during computation is the O(min(M,N)) or O(N) for the DP rows. Combined with storing the strings themselves, the overall effective space complexity is O(M+N). That is the reason why Hirschberg’s Algorithm is more efficient than the Base DP Algorithm, which makes it the only feasible solution for tasks like DNA sequences matching (incredible long sequences).

Graph2 – Time vs Problem Size (M+N)



Nature of the Graph (Logarithmic/ Linear/ Polynomial/ Exponential)

- Basic: O(MN) – Pseudo Polynomial
- Efficient: O(MN) – Pseudo Polynomial

Explanation:

Base Algorithm Runtime Analysis:

The algorithm constructs a dynamic programming (DP) table of size approximately (M+1) x (N+1). Filling each cell in this table typically takes a constant amount of time (O(1)) – involving comparisons and additions based on values from adjacent cells and the scoring matrix. Therefore, the total time to fill the table is proportional to the number of cells, which is M * N. The backtracking step to reconstruct the optimal alignment path takes O(M+N) time as it traverses a path from one corner of the DP table to the other. Since O(MN) has higher order than O(M+N), the overall runtime complexity is dominated by the table filling, making it O(MN).

Efficient Algorithm Runtime Analysis:

In the Efficient Algorithm, it is implemented by Hirschberg's algorithm, which is a divide-and-conquer approach. In each step, it divides one of the strings (say, X of length M) into two halves. It then performs two linear-space computations (the NWScore function in the implementation) to find an optimal midpoint in the other string (Y of length N). Each of these linear-space computations typically takes O(MN/2) time if X is halved and Y's full length is processed. So, two such computations take O(MN) time for that recursive step. The problem is then recursively solved for two smaller subproblems. The recurrence relation for the time complexity is generally T(M,N) = 2 * T(M/2, Navg) + O(MN), which solves to O(MN).

While the asymptotic time complexity is the same as the basic algorithm, Hirschberg's algorithm often has a larger constant factor due to the overhead of recursion, string manipulations (even if optimized with indices), and the two passes in each step. This is why it appears slower in the graph.

Time Performance Comparison:

A key observation from the graph is that the Efficient algorithm consistently requires more execution time than the Basic algorithm across all tested problem sizes. While both algorithms share a similar asymptotic time complexity (O(MN)), as previous section mentioned, the Efficient algorithm likely incurs additional overhead. This overhead can stem from more complex control flow, such as recursive function calls inherent in divide-and-conquer strategies, and potentially more operations per unit of input compared to the straightforward iterative nature of the Basic algorithm. Thus, despite having the same theoretical growth rate, the constant factors affecting the actual runtime are larger for the Efficient algorithm, making it slower in practice for the observed range of inputs.

Summary

In conclusion, here is a classic time-space trade-off. When the input length is relatively small, the Base DP Algorithm has lower constant multipliers for the runtime. Besides, the Base DP Algorithm is easy to implement, which makes it more efficient. Conversely, the Efficient Algorithm will be a better choice, which makes an unsolvable task become solvable. The extra overhead runtime here is acceptable.

Contribution

3772701387: Complete the project independently.