SUMMARY

Wei-Cheng Wang 3351037624

Che-Wei Hsu 8686190570

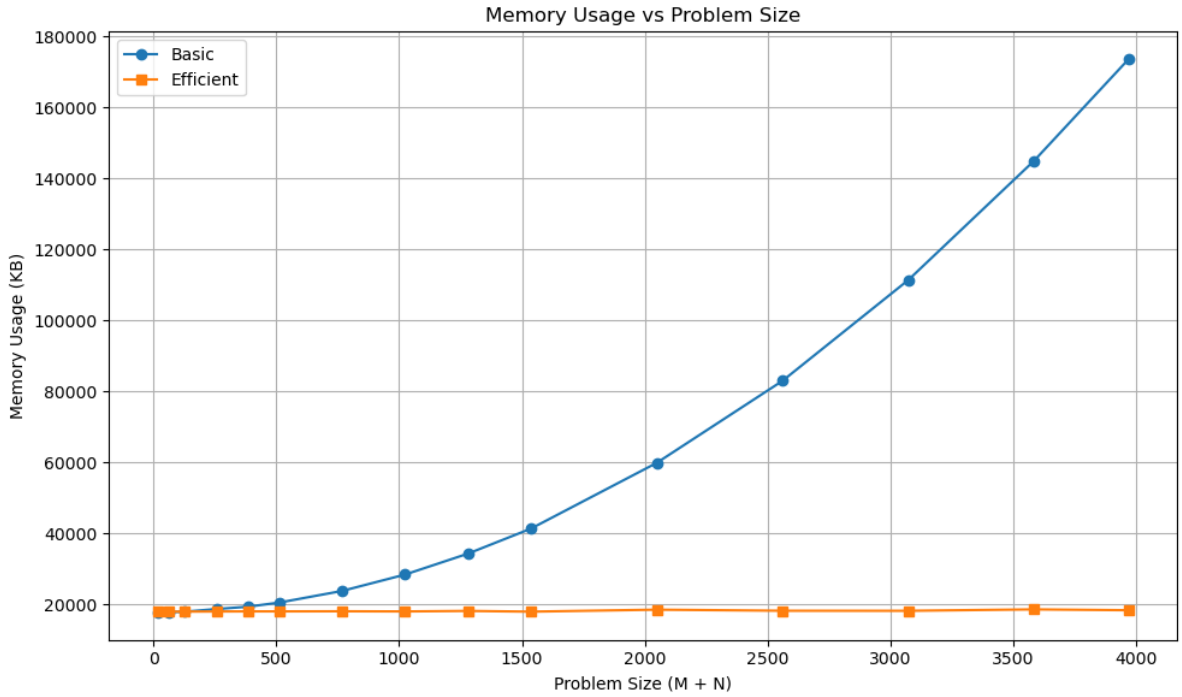
Wen-Yi Chuang 5738862390

Datapoints

M+N	Time in MS (Basic)	Time in MS (Efficient)	Memory in KB (Basic)	Memory in KB (Efficient)
16	0.01597	0.039101	17856	18016.0
64	0.15903	0.378847	17888	18016.0
128	0.61798	1.388073	18016	18032.0
256	2.40397	5.363941	18752	18064.0
384	5.48792	12.20727	19408	18096.0
512	9.81569	22.15099	20576	18096.0
768	24.4880	49.30806	23856	18128.0
1024	43.9911	90.05427	28464	18080.0
1280	68.1493	145.2150	34352	18224.0
1536	102.023	209.7318	41392	18016.0
2048	185.535	399.9739	59904	18576.0
2560	293.003	610.6942	83008	18272.0
3072	410.657	823.0700	111392	18256.0
3584	603.755	1220.636	144976	18656.0
3968	698.582	1558.127	173536	18416.0

Insights

Graph1 - Memory vs Problem Size $(M + N)$



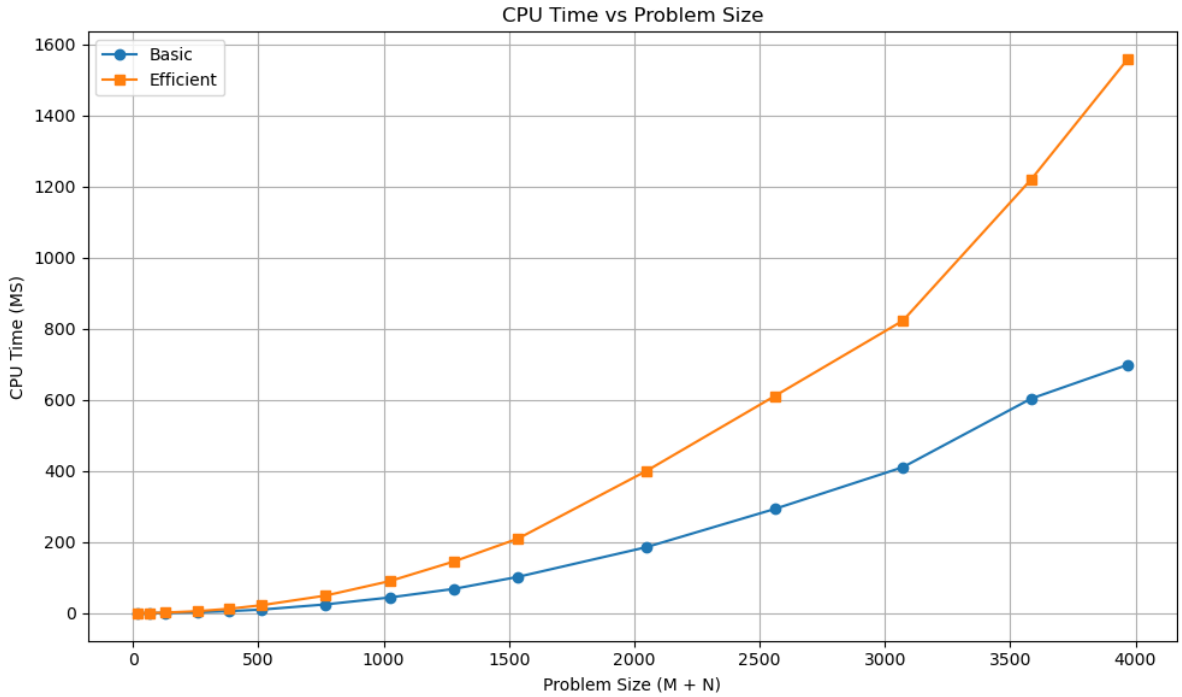
Basic →$O(mn)$ space
The basic algorithm requires a $(m + 1) \times (n + 1)$ dp table storing all the alignment cost from the bottom-up pass. Therefore, as the problems size $(m + n)$ grow, the $m \times n$ table grows in polynomial.

From the graph above, when $m + n$ is around 2000, the memory usage is 60000KB. If we double the problem size to 4000, the memory usage grows tripling to around 180000KB. In theory, it should be a four times growth if we only considered the increase in table size. However, if we consider the overhead cost in the comparison, the increase is around 2-3 times when we double the size of the input.

Efficient → $O(n)$ space
The efficient version applies the divide-and-conquer algorithm. We first divide the string X in half(X_L, X_R), compute the alignment cost between X_L and each of the substrings of Y (forward pass), and between X_R and every substring of Y (backward pass). After finding the optimal split point for Y by minimizing the total alignment cost, in the combine step, we recursively assign the two subproblems. Each subproblems will pass up the optimal solution to its parent and concatenate these results give us the final optimal alignment. Therefore, we only need to keep two rows of $n + 1$ during the whole process, resulting in memory usage linear in n

From the graph above, no matter the problem size, the memory usage stays around 20000KB

Graph2 - Time vs Problem Size $(M + N)$



Basic → $O(mn)$ time
The basic algorithm builds the $(m + 1) \times (n + 1)$ table exactly once (each of the cell in the table can be calculated in constant time), then does the top-down pass in linear time. Therefore, the CPU time grows in polynomial time

When doubling both X and Y (for example, the problem size grows from 2000 to 4000), the time complexity grows roughly $O(2m \times 2n)$. From the graph, we can see the CPU time increase from 200ms to nearly 800ms, which is approximately four times growth.

Efficient → $O(mn \log m)$ time
The efficient version requires splitting X in half, running the forward and backward pass to locate the best split point which takes a total of $2cmn$ CPU time in every recursion level. Since we are splitting the problem half each time in the divide step, there are $\log_2 m$ levels in total. Hence, the total CPU time grows in $O(mn \log m)$ (in polynomial time).

From the graph above, when the problem size is doubled from 2000 to 4000, there is also approximately a four times growth in CPU time.

In addition, we can see that when the problem size is 2000, the CPU time of the efficient version is twice more than the basic algorithm. Which illustrates the $2cmn$ CPU time difference in computing the forward and backward pass for the alignment cost in the efficient version compared to the basic algorithm.

To sum up, the efficient algorithm provides better space complexity compared to basic algorithm which is $O(n)$ and $O(mn)$ respectively. However, the time complexity of the efficient algorithm requires more CPU time by doing the one more pass of calculation for each log m levels, resulting in $O(mn \log m)$ time which is slower compared to the $O(mn)$ of the basic algorithm.

Contribution

<3351037624>: <Equal Contribution>
<8686190570>: <Equal Contribution>
<5738862390>: <Equal Contribution>