SUMMARY

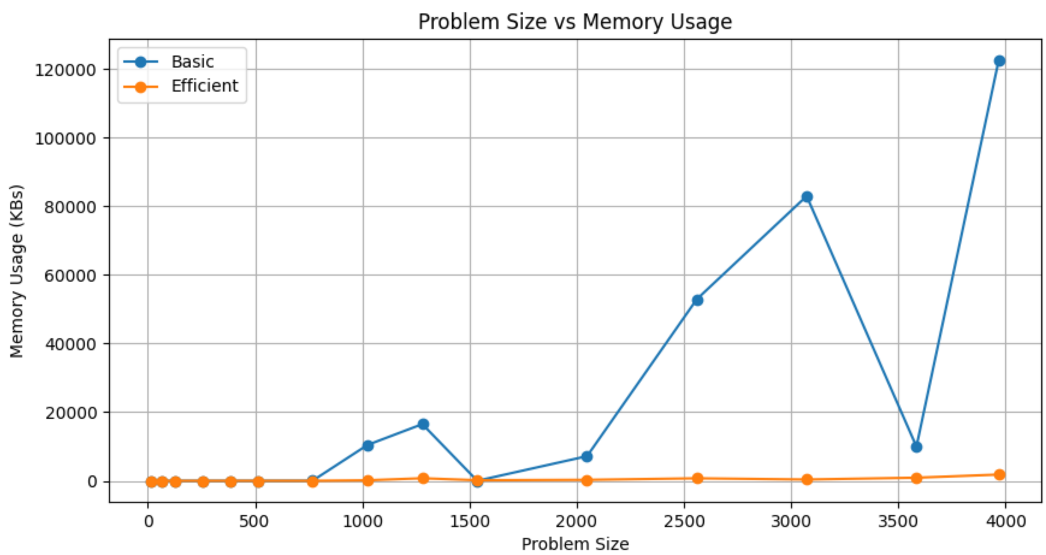
USC ID/s -
Matthew Schulz: 9011355532, *Ryan Marr*: 3454675882, *Vedant Raval*: 4328073870

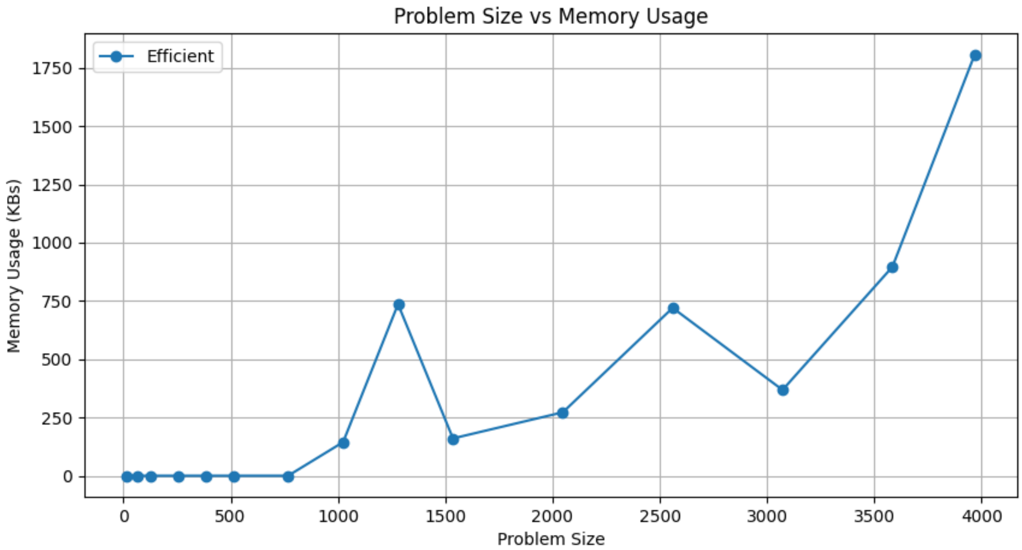
M+N	Time in MS (Basic)	Time in MS (Efficient)	Memory in KB (Basic)	Memory in KB (Efficient)
16	0.18	0.73	0	0
64	0.81	3.76	0	0
128	2.07	9.61	0	0
256	5.56	21.53	0	0
384	11.32	44.22	0	0
512	18.26	80.70	0	0
768	40.18	159.81	16	0
1024	72.22	254.73	10432	144
1280	109.61	366.44	16512	736
1536	152.69	502.22	0	160
2048	269.05	859.57	7168	272
2560	417.30	1293.68	52928	720
3072	587.65	1778.68	82976	368
3584	794.58	2413.93	9952	896
3968	1003.77	2940.67	122720	1808

Datapoints

Insights

Graph1 – Memory vs Problem Size (M+N)





Note: As the growth rates of the memory plots for the two methods were different, it was difficult to show an accurate plot for the Efficient algorithm along with the Basic algorithm. Hence added a separate graph depicting the growth rate of the Efficient Algorithm.

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

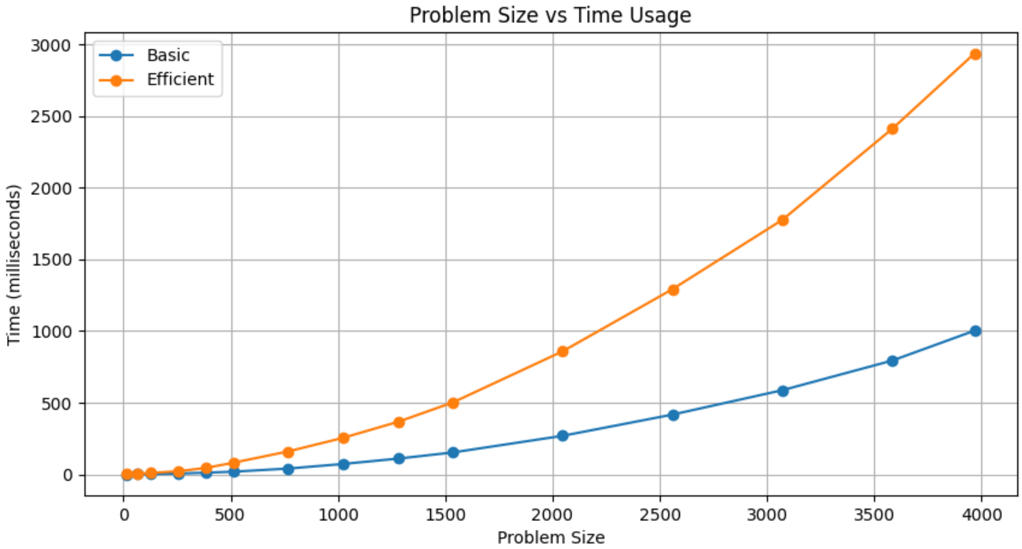
Basic: Polynomial

Efficient: Linear

Explanation:

- Given the way of computing memory usage using *psutil*, we expect to observe outliers in the memory plots for the two methods.
- But if we disregard those outliers and look at the rate in which the memory usage in general grows with the problem size for the two methods then we can clearly see that the memory used for the *Basic* algorithm is so much more as compared to the memory used for the *Efficient* algorithm for higher values of Problem Size that the memories for *Efficient* algorithm seem to be growing at a nearly constant rate as compared to the *Basic* algorithm. It is quite easily evident that the *Efficient* algorithm grows at a polynomial rate.
- Though, if we zoom in to the curve observed for the *Efficient* algorithm, then we can see that it grows at a linear rate in general (disregarding the outliers).
- This difference lies in the core idea of utilizing the approach of Divide and Conquer for the *Efficient* algorithm as opposed to using a 2D matrix of size O(mn) in the *Basic* algorithm, where the length of string X is m and the length of string Y is n.
- In the *Efficient* algorithm - for two equal halves of X, X_L and X_R, we use a Dynamic Programming based approach to get unequal splits of Y, Y_L and Y_R, such that the total cost of alignment between (X_L, Y_L) and (X_R, Y_R) is minimized. Since here we are solving the problem of finding an optimal split-point in Y at each Divide and Conquer step, we only need array(s) of size O(m + n) as opposed to storing a 2D matrix of size O(mn) in the *Basic* algorithm corresponding to all the possible sub-strings $X_{1,2,\dots,k}$ and $Y_{1,2,\dots,j}$, for k ≤ m and j ≤n.
- This major difference between these two algorithms is why we observe a polynomial graph for the *Basic* algorithm and Linear graph for the *Efficient* algorithm.

- **Graph2 – Time vs Problem Size (M+N)**



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

Basic: Polynomial

Efficient: Polynomial

Explanation:

- Given the task of computing the optimal alignment cost between two input strings of X and Y, the *Efficient* algorithm essentially breaks the Y string into two sub-parts Y_L and Y_R by choosing a split point in the string Y which minimizes the sum of alignment costs between two equal halves of X and the two unequally broken sub-strings of Y. This can be done using Dynamic programming.
- Suppose the length of X is m and the length of Y is n. Then the above operation could be done in C*(mn) operations, which is of the same order as the time taken for the *Basic* algorithm.
- Once we have obtained the sub-strings X_L, X_R, Y_L, and Y_R - the *Efficient* algorithm would now proceed with the pairs of (X_L, Y_L) and (X_R, Y_R) and repeat the same steps. In this second level, the algorithm would take C*(mn)/2. Similarly, the third level would require C*(mn)/4 steps and so on…
- We know that the sum of the sequence - $1 + 1/2 + 1/2^2 + \ldots$ is a constant and thus the *Efficient* algorithm would still require O(mn) operations.
- Thus, the graph for both the algorithms would be expected to be Polynomial in nature and it is what we observe indeed.

Contribution

(Please mention what each member did if you think everyone in the group does not have an equal contribution, otherwise, write “Equal Contribution”)
9011355532: Equal Contribution
3454675882: Equal Contribution
4328073870: Equal Contribution