SUMMARY

USC ID/s:

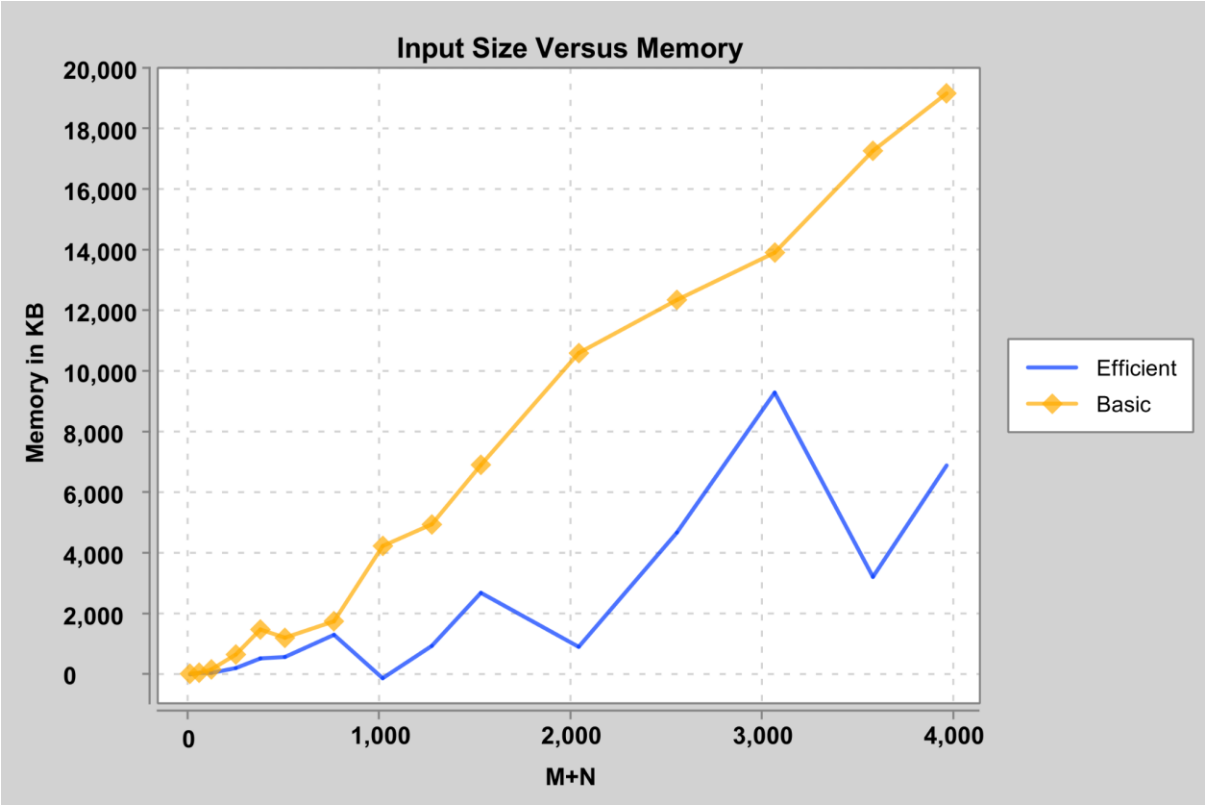
Ernesto Pimentel, 9004-1961-90
Shujie Chen, 7181-3025-74
Rahul Katwala, 4962-8819-81

Datapoints

M+N	Time in MS (Basic)	Time in MS (Efficient)	Memory in KB (Basic)	Memory in KB (Efficient)
16	0.090	0.030	20.000	0.0
64	0.460	0.456	56.000	50.338
128	1.568	1.203	164.000	50.334
256	5.940	4.192	656.000	209.114
384	13.503	7.366	1480.000	527.918
512	23.330	12.310	1204.000	578.232
768	56.521	31.888	1752.00	1308.023
1024	100.320	71.715	4236.000	-130.065
1280	161.319	126.582	4944.000	939.634
1536	231.664	240.421	6912.000	2694.293
2048	416.436	505.945	10596.000	904.446
2560	661.930	894.296	12352.000	4680.616
3072	936.525	1647.566	13912.000	9297.5768
3584	1344.172	2878.134	17268.000	3211.290
3968	1587.693	3693.95	19164.000	6887.955

Insights

Graph1 – Memory vs Problem Size (M+N)



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

Basic: Polynomial $\Theta(mn)$

Efficient: Linear $\Theta(m+n)$

Explanation:

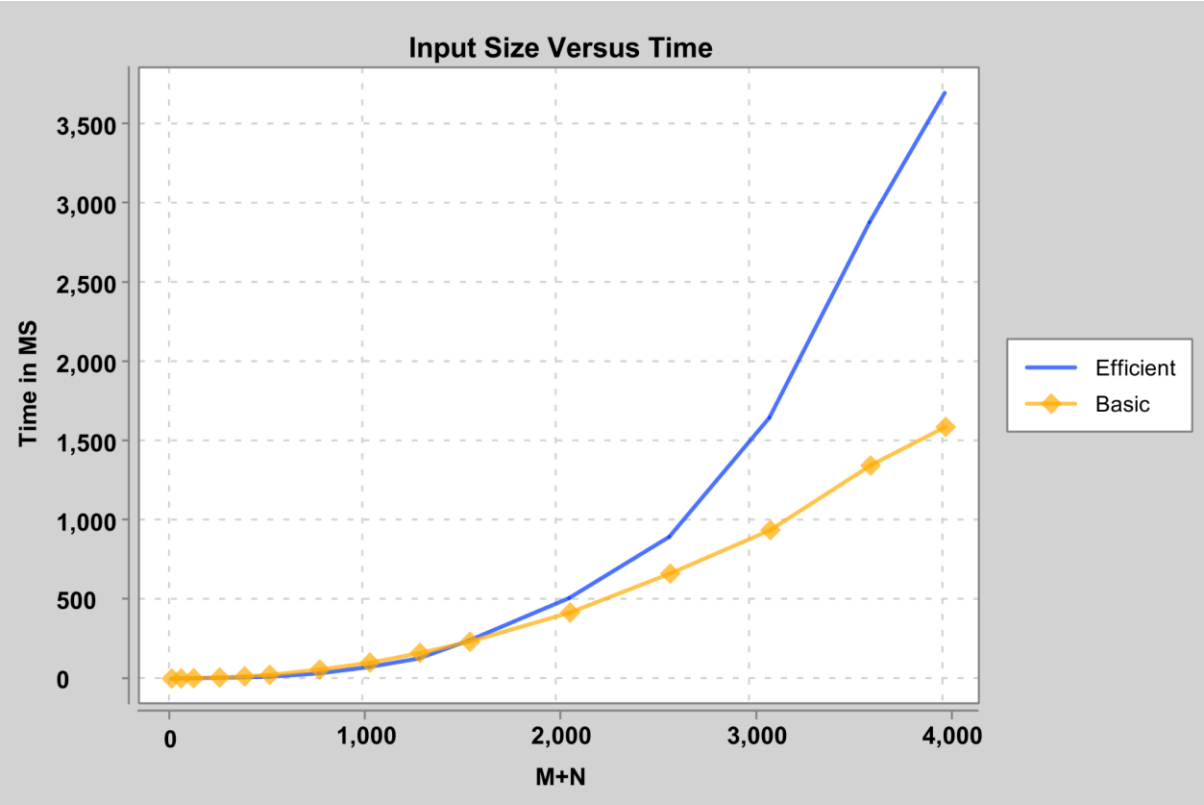
The basic algorithm is straightforward on why it has a polynomial graph. In this algorithm we need to fill out an m by n table in order to find the optimal solution. The graph shows a quadratic nature which is a polynomial function. It is also evident that this polynomial function is growing at a much faster rate than the efficient solution and the gap seems to grow as the input gets larger.

In contrast, the efficient algorithm shows a different nature. The graph doesn't show a perfectly smooth linear line but the trend is definitely linear. The efficient algorithm is always bounded by some constant which is the sum of the input size m+n.

This is because we no longer utilize an m x n table to memorize min-costs for all pairs of substrings, but rather only store the last 2 columns (m x 2) to support the recurrence relation. This means the memory cost of a min-cost calculation is bounded by the length of string x. Then in the divide-and-conquer phase, the total cost comes to $\Theta(m)$, since lengths of all generated substrings sum up to x.length() = m, regardless of how we split x. In addition, the string y also occupies y.length() = n memory space.

The line looks very jagged, but we suspect it has to do something with the Java garbage collector. However, it is still clear the line demonstrates that it performs much more memory-efficiently than the basic algorithm.

Graph2 – Time vs Problem Size (M+N)



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

Basic: Polynomial $\Theta(mn)$

Efficient: Polynomial $\Theta(mn)$

Explanation:

For the basic algorithm we are using dynamic programming for string alignment. In this algorithm, we are filling a DP table of size m by n. It will take $\Theta(mn)$ to fill up a table of m by n. Each recurrence call the algorithm does, it takes constant time, so the graph shows us what a quadratic curve would look like.

For the efficient algorithm, we are using dynamic programming in the bottom-up phase as well, but using divide-and-conquer to recursively find the alignment in the top-down phase. As per discussion for the basic algorithm, denote the time cost of computing min cost for a (m+n)-size problem as Cmn, where C is a constant. At the ith level of the recursion tree, where the problem size is $\left(\frac{1}{2}\right)^i$ of that at the root. Thus the total time cost of this level would be $\left(\frac{1}{2}\right)^i \cdot Cmn$. The total time cost of the recursion tree comes to:

$$\sum_{i=0}^{\log n} \left(\frac{1}{2}\right)^i \cdot Cmn \approx 2Cmn$$

Therefore, the efficient algorithm exhibits the same time complexity magnitude as the basic version, but should cost approximately twice as much time as the latter.

Contribution

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