**SUMMARY**

Datapoints

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| M+N | Time in MS (Basic) | Time in MS  (Efficient) | Memory in KB  (Basic) | Memory in KB  (Efficient) |
| 16 | 0.225 | 0.304 | 11232 | 11456 |
| 64 | 0.876 | 1.997 | 11392 | 11308 |
| 128 | 1.921 | 4.001 | 11824 | 11328 |
| 256 | 7.482 | 14.518 | 11808 | 11600 |
| 384 | 15.436 | 34.092 | 11840 | 11152 |
| 512 | 27.265 | 56.961 | 12256 | 11472 |
| 768 | 65.708 | 132.140 | 13216 | 11648 |
| 1024 | 117.552 | 227.657 | 14368 | 11520 |
| 1280 | 182.664 | 357.392 | 15136 | 11424 |
| 1536 | 258.149 | 509.531 | 17008 | 11696 |
| 2048 | 476.079 | 908.688 | 20512 | 11488 |
| 2560 | 740.386 | 1447.897 | 25040 | 11936 |
| 3072 | 1059.279 | 2053.416 | 31248 | 11312 |
| 3584 | 1482.414 | 2818.944 | 33848 | 11744 |
| 3968 | 1794.422 | 3427.691 | 44304 | 11920 |

Insights

The above table shows time and memory comparisons with respect to input size (m+n) for two algorithms solving the Sequence Alignment problem.

Sequence Alignment problem is the problem of aligning two strings character by character such that the total penalty (cost) of alignment is minimum.

1. A penalty of delta occurs when a character is matched with a gap (i.e. a gap is inserted in the string)
2. A mismatch penalty occurs when there is a mismatch of characters of the string.

First method is the basic algorithm which uses Dynamic Programming. This creates a table of size m\*n and uses dynamic programming to calculate the minimum alignment cost. Its recurrence formula computes minimum over 3 cases:

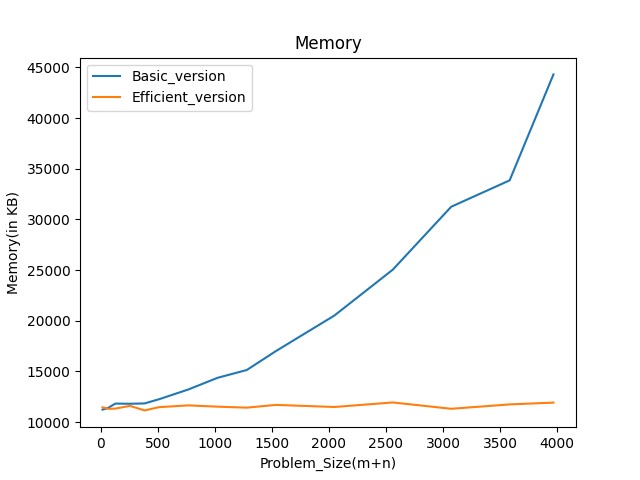
* 1. Mismatch between x[i] and y[j] for string X and string Y and incurring a mismatch penalty.
  2. Matching x[i] with gap in string Y and incurring a gap penalty.
  3. Matching y[j] with gap in string X and incurring a gap penalty.

The time complexity of this algorithm is O(m\*n) and the space complexity is O(m\*n). So as the input size increases the time taken and space utilized increases polynomially. This becomes a problem when dealing with large sizes of input which is common in real life applications like - DNA Sequencing where the DNA length is in billions.

The second algorithm is using Divide and Conquer along with Dynamic Programming to obtain a time complexity of O(m\*n) but saving heavily on space utilized. The space complexity is O(m+n) which is linear.

This efficient algorithm makes use of dynamic programming to obtain the optimal split point which can be used by divide and conquer to get optimal alignment.

Graph1 – Memory vs Problem Size (M+N)



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

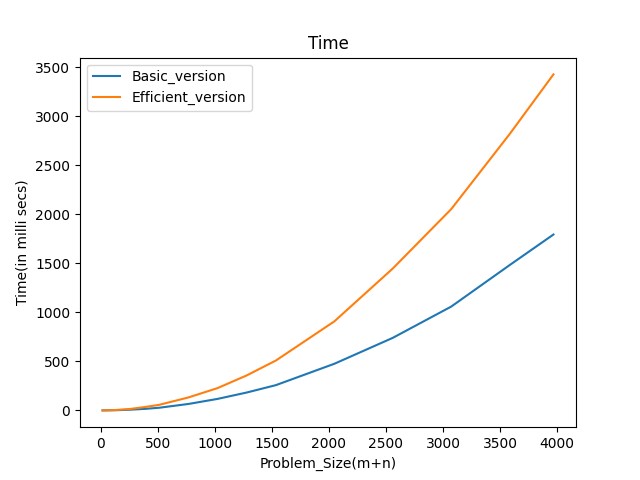
**Basic**: Polynomial **Efficient**: Linear

*Explanation:*

As seen from the graph, for the basic algorithm, as the problem size increases, the memory utilized increases polynomially which is expected as the theoretical space complexity is O(m\*n).

For the efficient algorithm, the memory utilization increases linearly as the input size increases (other than some minor outliers) which is also expected as the theoretical space complexity is O(m + n).

Graph2 – Time vs Problem Size (M+N)



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

**Basic**: Polynomial **Efficient**: Polynomial

*Explanation:*

For the basic algorithm, as the problem size increases the time complexity increases polynomially which is expected as the theoretical space complexity is O(m\*n).

For the efficient algorithm, as the problem size increases the time complexity increases polynomially which is expected as the theoretical space complexity is O(m\*n), but as seen from the graph the time taken is more than the basic algorithm, this is due to the fact that there is double the computation performed as the basic algorithm. This increased computation yields a significant reduction in the space utilized (from polynomial space to linear).