Description:

MSS1 **-** In the first Maximum Subsequence Sum, it can be shown that T(n) = O(n^3). In this, we have three nested loops which the integers run through the receive the sum. However, this is not the most effective way, as seen in the future that the three loops can be combined the make the process more efficient.

MSS2 - In the second Maximum Subsequence Sum, it can be shown that T(n) = O(n^2). This is a quadratic function, the difference between MSS1 and MSS2 is that the for loops are slightly compressed, as it is trimmed from 3 down to 2. This is an average case scenario, yet not necessarily the best case to calculate the sum.

MSS3 - Here, we try to make an even quicker way to calculate the sum in a more efficient way. In the third Maximum Subsequence Sum, it can be shown that T(n) = O(n\*log(n)). This is the first introduction to logarithmic functions within calculating the sum, but it can be seen through various graphs that this is an efficient way. It uses a positive number for the prefix of the optimal subsequence, as obviously it can’t use a negative number to start with at a logarithmic function.

MSS4 - In the fourth Maximum Subsequence Sum, it can be shown that T(n) = O(n). This is the absolute fastest way to calculate the sums. The text file full of integers would be placed into an array and would be placed in order, as the loop goes through each element of the array and adds it to the value prior, and is proved compared to the other three that this is the absolute fastest way to calculate the sum.

Complexity Analysis:

A piece of paper with writing

Description automatically generated with medium confidence

A piece of paper with writing on it

Description automatically generated with medium confidence

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Algorithm Runtime Comparison:

Table 1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | Execution Time (nanoseconds) | | | |
|  |  | Alg. MSS1 | Alg. MSS2 | Alg. MSS3 | Alg. MSS4 |
| Number of Elements | 10 | 3900 | 2849 | 4191 | 3892 |
| 20 | 7577 | 4530 | 6936 | 5020 |
| 50 | 43124 | 19196 | 25916 | 8820 |
| 100 | 197843 | 127661 | 58750 | 13277 |
| 1000 | 44500201 | 430313 | 281678 | 19952 |
| 5000 | 5000323528 | 3303368 | 4267222 | 91938 |
| 10000 | 40148200535 | 38241074 | 30722097 | 187434 |

Graph 1:

Graph 2:

Graph 3:

Graph 4:

Graph 5:

Conclusion: Overall, our algorithms performed as was expected. As visible in Table 1, it is clear that algorithm MSS4 is our most efficient algorithm, with the execution time for every input size being much faster than any other algorithm. Theoretically, we were expecting to see an O(n3) time complexity for MSS1, O(n2) for MSS2, O(n\*log(n)) for MSS3, and O(n) for MSS4. From graphs 3,4, and 5, we can see that the trend of our actual data is similar to the theoretical curves. If more data was collected for each respective input size, we believe that there would be a stronger correlation between the experimental data and the theoretical trendlines. Furthermore, some deviation from the theoretical values can be expected due to the program execution being potentially (and randomly) interrupted by lower-level computer tasks. The data in Graph 1 also confirms the theory as by our final input size of 10000, MSS4 < MSS3 < MSS2 < MSS1. It is important to specify that this set of inequalities does not always hold throughout our data. When looking at MSS2 and MSS3, we can see from Table 1 that MSS2 has faster execution times from input size of 10 to input size of 5000. It is not until input size of 10000 that MSS3 is faster than MSS2. This shouldn’t be considered as an error in our data because a quadratic function can be smaller than a n\*log(n) function (up to a finite value of n) depending on the constants involved.