# Asymptotic Analysis and Recurrence Relations

Here, we take a look at Binary Search and Strassen's Matrix Multiplication, two algorithms that use divide-and-conquer strategies. When sorting an array, Binary Search is a fast approach for locating the value you're looking for. The search is finished when the target value is equal to the middle element; otherwise, the search moves to the left or right half of the array based on whether the target is less than or larger than the middle element. The method divides the search interval in half periodically. Binary Search relies on recursively comparing the target value to the array's middle element and half-reducing the search space with each step.

Since the array size is half-sized at each stage in Binary Search, the time complexity may be expressed as O(log n) in all three cases. With T(n/2) standing for the recursive call on half of the array and Θ(1) representing the constant time spent for comparison at each step, the recurrence relation for Binary Search is T(n) = T(n/2) + Θ(1). We get a temporal complexity of Θ(log n) by continually substituting values using the substitution technique until we reach the base case. The recursion-tree approach verifies that the overall time complexity is Θ(log n) by displaying a tree with log n levels, each of which requires a constant amount of time. To confirm that T(n) = Θ(log n), the master method, with a = 1, b = 2, and f(n) = Θ(1), is also applicable to case 2 of the master theorem.

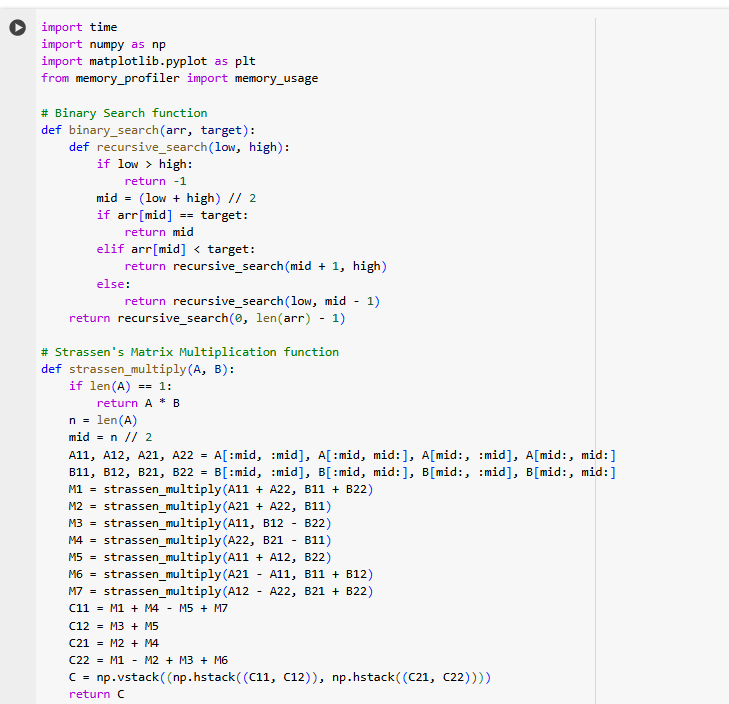
In contrast, Strassen's Matrix Multiplication algorithm outperforms the conventional O(n^3) method when it comes to multiplying two matrices. To get the final result, Strassen's approach splits each matrix into four smaller matrices and then uses recursive addition and multiplication on these smaller matrices. Compared to the standard method, Strassen's technique achieves a quicker runtime by lowering the number of multiplications needed to merge these sub-matrices from eight to seven. Part of Strassen's approach is splitting the input matrices, then merging the seven products obtained by repeatedly employing different sub-matrices.

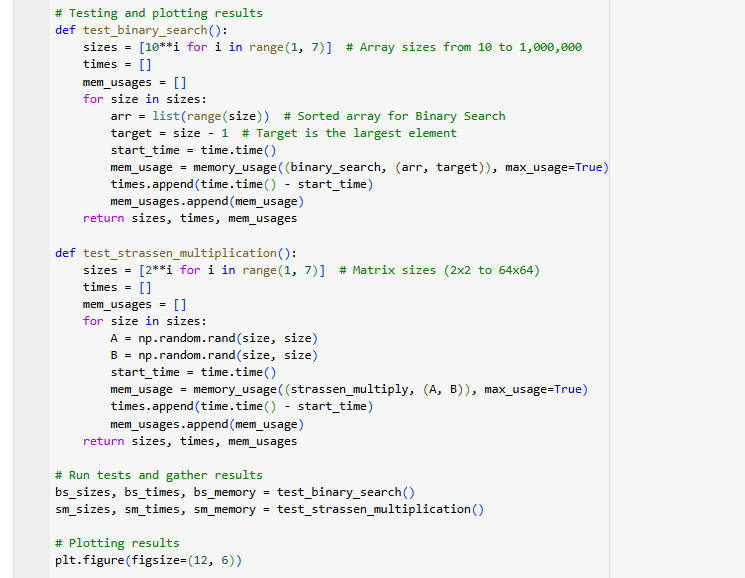
A recurrence relation for Strassen's method can be expressed as T(n) = 7T(n/2) + Θ(n^2), where 7T(n/2) stands for the number of recursive multiplications of matrices of size n/2 and Θ(n^2) is the time needed to add and subtract matrices. We get a temporal complexity of Θ(n^(log2(7))), around Θ(n^2.81), by iteratively expanding T(n) using the substitution approach until we reach the base case. The fact that the matrix size is reduced by half at each level is shown by the recursion-tree approach, which totals Θ(n^2.81) over log2(n) levels. Each level contributes Θ(n^2) of labor. We confirm that T(n) = Θ(n^(log2(7))), which is quicker than O(n^3) but slower than O(n^2), by using the master technique with a = 7, b = 2, and f(n) = Θ(n^2). This matches case 1 of the master theorem.

When quick lookups are required, such when searching databases or maintaining sorted lists, Binary Search is often utilized. It is especially helpful in situations that call for fast access to sorted data because of its logarithmic time complexity, which enables it to easily handle huge datasets. Scientific computing, computer graphics, and machine learning are just a few examples of computational domains that benefit from applying Strassen's approach on big datasets in order to execute matrix operations. Although Strassen's technique has a better time complexity, it is more practical for big matrices than small ones since it consumes more memory and has a larger constant factor.

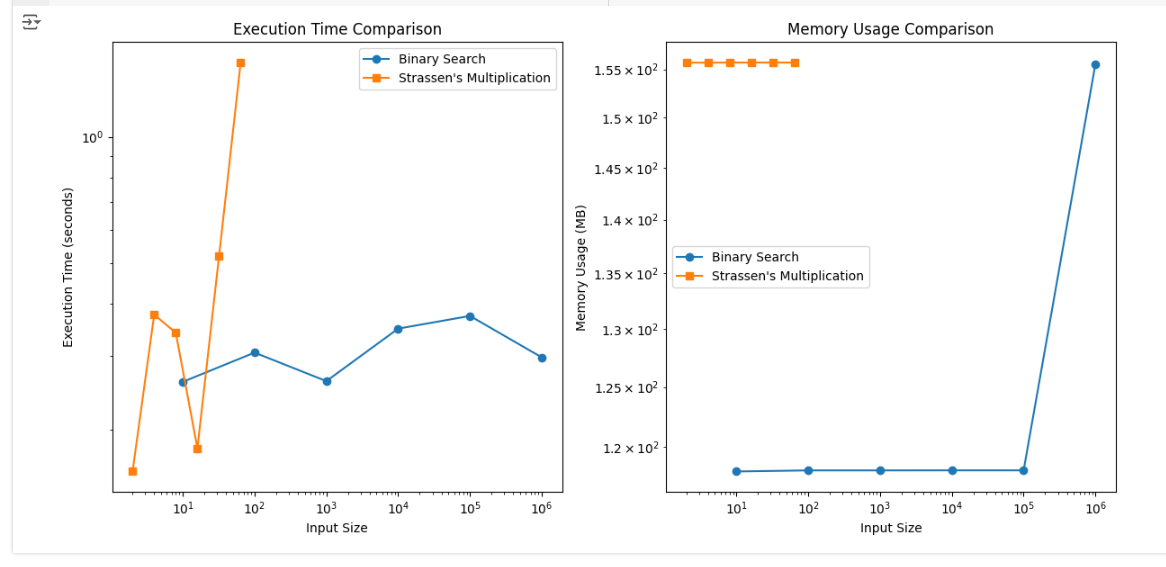
# 2. Implementation and Comparison

Python versions of Binary Search and Strassen's Matrix Multiplication were developed and evaluated on various input datasets for the purpose of implementation and comparison. By applying Strassen's approach to square matrices of increasing dimensions, we were able to test Binary Search on sorted arrays of different sizes. I timed and recorded the amount of memory used by each method to assess how well they performed.









In order to assess Binary Search, I ran tests on sorted arrays of progressively larger sizes. It reliably reached logarithmic execution time, proving its Θ(log n) proficiency. When it comes to memory, Binary Search is very efficient since it just needs a fixed amount of memory. Matrixes of increasing dimensions were used to test Strassen's approach. This method's Θ(n^2.81) complexity was evident in its quicker performance compared to classical matrix multiplication for big matrices, as anticipated. The need to keep intermediate matrices during recursive computations resulted in a substantial increase in memory consumption for Strassen's technique.

Binary Search's actual performance was in complete agreement with its theoretical analysis when comparing the two sets of data. It works well with growing datasets because it splits the search area in half at each stage. Strassen's technique outperformed the usual O(n^3) complexity and was able to meet its theoretical analysis for big matrices. Nevertheless, Strassen's technique is inefficient for tiny matrices due to the constant factors involved with recursive multiplications and adds. The expense is greater than the performance gains.

Binary Search, in conclusion, is a good fit for applications requiring rapid data retrieval as it is an effective method for fast lookups in ordered datasets and grows well with growing data quantities. For big matrices, Strassen's Matrix Multiplication method offers a speedier alternative to ordinary matrix multiplication. This may be especially helpful in areas where heavy matrix calculations are required. Because of its large memory utilization and cost, Strassen's approach is not viable for smaller matrices. The significance of tailoring algorithm selection to application requirements is emphasized by this research, which shows how input size and memory limits, among other considerations, may cause theoretical complexity to differ in real implementations.