Lab-2 Complexity Analysis

220001014 - Aviral Sharma

January 2024

1 Matrix multiplication using Divide and Conquer

Time Complexity Analysis

The time complexity of the Strassen algorithm can be analyzed as follows:

- The recursive divide-and-conquer approach divides the matrices into submatrices of size $n/2 \times n/2$. This division is performed recursively until the base case is reached, which has a size of 1.
- At each level of recursion, a total of 7 multiplications of submatrices are performed, each involving matrices of size $n/2 \times n/2$. This results in a total of $O(7^{\log_2 n})$ multiplications.
- Additionally, there are a total of 18 additions or subtractions of submatrices at each level of recursion.
- Therefore, the time complexity of the Strassen algorithm is $O(n^{\log_2 7})$, which is approximately $O(n^{2.81})$.

Space Complexity Analysis

The space complexity of the Strassen algorithm can be analyzed as follows:

- The space complexity primarily arises from the recursion stack during the recursive calls, which has a depth of $\log_2 n$.
- Additionally, temporary matrices are created for storing intermediate results during the multiplication process. However, these temporary matrices are reused and do not contribute significantly to the overall space complexity.
- Therefore, the space complexity of the Strassen algorithm is $O(\log n)$.

2 Maximum subarray sum

2.1 Using Divide and Conquer

Time Complexity Analysis

- The algorithm recursively divides the input array into halves until it reaches subarrays of size 1.
- At each level of recursion, the algorithm performs a constant number of operations to calculate the maximum subarray sum crossing the midpoint.
- Therefore, the time complexity of the algorithm can be expressed by the recurrence relation $T(n) = 2T(n/2) + O(n)$, where n is the size of the input array.
- Using the Master theorem, the time complexity of the algorithm is $O(n \log n)$.

Space Complexity Analysis

- The space complexity primarily arises from the recursion stack during the recursive calls, which has a depth of $\log_2 n$.
- Additionally, the algorithm uses a constant amount of extra space for variables and temporary storage.
- Therefore, the space complexity of the algorithm is $O(\log n)$.

2.2 Using O(n) method

Time Complexity Analysis

- The code iterates through the input array once using a single loop.
- Within each iteration, the code updates the current sum and the maximum sum using Kadane's algorithm, which has a time complexity of $O(1)$ per iteration.
- Therefore, the time complexity of the algorithm is $O(n)$, where n is the size of the input array.

Space Complexity Analysis

- The code uses a fixed amount of extra space for variables such as x, a, $maximum$, $curr\_sum$, and n.
- Additionally, the code does not use any dynamic memory allocation or recursion.
- Therefore, the space complexity of the algorithm is $O(1)$, indicating constant space usage regardless of the input size.