Lab-2 Complexity Analysis

220001014 - Aviral Sharma

January 2024

1 Merge

1.1 A + B

Time Complexity Analysis

The provided code implements the standard merge algorithm. Let's denote:

- m: the number of elements in array a
- n: the number of elements in array b

The while loop in the `merge` function iterates $m + n$ times, and each iteration involves constant-time operations. Therefore, the time complexity of the merging step is $O(m + n)$.

Space Complexity Analysis

The space complexity of the code is determined by the usage of additional memory. The merged array `c` is created with a size of $m + n$, and its space complexity is $O(m + n)$.

In summary, the time complexity is $O(m \log m + n \log n)$, and the space complexity is $O(m + n)$.

1.2 $A \cup B$

Time Complexity Analysis

Finding the union in the `array_union` function involves iterating through both arrays once. The time complexity of this operation is $O(m + n)$.

Space Complexity Analysis

The space complexity of the algorithm is determined by the usage of additional memory:

Additional memory is used for the vector `ans`, which has the space complexity of $O(m + n)$.

Therefore, the space complexity of the algorithm is $O(m + n)$.

1.3 $A \cap B$

Time Complexity Analysis

Finding the intersection in the `array_union` function involves iterating through both arrays once. The time complexity of this operation is $O(m + n)$.

Therefore, the overall time complexity of the algorithm is $O(m \log m + n \log n)$.

Space Complexity Analysis

The space complexity of the algorithm is determined by the usage of additional memory:

Additional memory is used for the vector `ans`, which has the space complexity of $O(m + n)$.

Therefore, the space complexity of the algorithm is $O(m + n)$.

2 In-place merge

Time Complexity Analysis

The dominant part of the time complexity is determined by the merging step in the `merge` function. Let's denote:

- m: the number of elements in the first sorted part
- n: the total number of elements in both sorted parts

The while loop in the `merge` function iterates n times (or $n - m$ times) to merge the two sorted parts. Each iteration of the loop involves constant-time operations.

Therefore, the overall time complexity of the code is $O(n)$.

Space Complexity Analysis

The space complexity of the code is determined by the usage of additional memory. The algorithm is in-place, meaning it uses only a constant amount of extra memory for variables like i, j, k, and x.

Therefore, the space complexity of the code is $O(1)$ (constant space).

In summary, the time complexity is $O(n)$, and the space complexity is $O(1)$.

3 Merge Sort with in-place merge

Time Complexity Analysis

The main part of the code is the `mergeSort` function, which recursively divides the array into halves and merges them. Let's denote:

- n: the size of array a

The `mergeSort` function splits the array in half and recursively calls itself on each half. This process continues until the base case is reached (when r is not greater than l). The merging step in the `merge` function takes $O(n^2)$ time for each merge operation, as it is performed in-place.

The recurrence relation for the time complexity of merge sort is given by:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$$

Solving this recurrence relation gives a time complexity of $O(n^2 \log n)$.

Therefore, the overall time complexity of the code is $O(n^2 \log n)$.

Space Complexity Analysis

The space complexity is determined by the recursive calls in the `mergeSort` function. Since the algorithm uses recursion and does not use additional data structures for merging, the space complexity is dominated by the recursion stack.

The maximum depth of the recursion tree is $\log_2 n$, and at each level, the space complexity is constant (since merging is performed in-place). Hence, the overall space complexity is $O(\log n)$.

In conclusion, the time complexity of the code is $O(n^2 \log n)$, and the space complexity is $O(\log n)$.

4 Median of array

Time Complexity Analysis

The main part of the code is the `quickselect` function, which is a randomized version of the quicksort algorithm used to find the k-th smallest element in an array. Let's denote:

- n: the size of array a

The partitioning step in the `partition` function takes linear time, and in the worst case, the array is divided into two unequal halves.

The recurrence relation for the time complexity of quickselect is given by:

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

Solving this recurrence relation gives a time complexity of $O(n)$ on average, and in the worst case, it can be $O(n^2)$. However, the use of randomization in quickselect typically leads to an average-case time complexity of $O(n)$.

Therefore, the overall time complexity of the code is $O(n)$ on average.

Space Complexity Analysis

The space complexity is determined by the recursion stack in the `quickselect` function. In the worst case, the recursion depth can be $O(n)$, leading to a space complexity of $O(n)$. However, on average, the recursion depth is logarithmic, resulting in an average-case space complexity of $O(\log n)$.

In conclusion, the average time complexity of the code is $O(n)$, and the average space complexity is $O(\log n)$.

5 Finding first and second minima

Time Complexity Analysis

The provided code for finding the first and second minima in an array has a time complexity of $O(n)$, where n is the size of the array. This is because the code iterates through the array once to find the first and second minima.

In the worst case, the array is traversed twice, but the constant factor is relatively small. Therefore, the overall time complexity is $O(n)$.

Space Complexity Analysis

The space complexity is determined by the usage of additional memory to store the array and a constant amount of extra space for variables like `x`, `first_minima`, and `second_minima`.

Hence, the space complexity of the code is $O(1)$ due to the constant additional storage usage.

In conclusion, the time complexity of the code is $O(n)$, and the space complexity is also $O(1)$.

6 Quick Sort with pivot as almost the median

Time Complexity Analysis

The provided quicksort code with the pivot as almost the median utilizes the median-of-three strategy to choose a good pivot. The median-of-three strategy helps mitigate the chances of selecting a bad pivot and improves the algorithm's performance on partially sorted data.

Let's denote:

- n: the size of array a

The main operations in the code are the partitioning step in the `partition` function and the recursive calls in the `quicksort` function.

The recurrence relation for the time complexity of quicksort is given by:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Solving this recurrence relation gives a time complexity of $O(n \log n)$ in the average and worst case. The use of the median-of-three strategy helps in reducing the likelihood of worst-case scenarios.

Therefore, the overall time complexity of the code is $O(n \log n)$.

Space Complexity Analysis

The space complexity is determined by the recursion stack in the `quicksort` function. In the worst case, the recursion depth can be $O(n)$, leading to a space complexity of $O(n)$. However, on average, the recursion depth is logarithmic, resulting in an average-case space complexity of $O(\log n)$.

In conclusion, the average time complexity of the code is $O(n \log n)$, and the average space complexity is $O(\log n)$.