Part 1

Method 1

This method involves iterating through each ID value in the given list, directly comparing it with the target ID. If a match is found, the corresponding position is returned; otherwise, -1000 is returned. The worst-case scenario occurs when none of the target IDs are present in the list, resulting in each ID being compared once, leading to a total of $(m \cdot n)$ comparisons. Therefore, the worst-case time complexity is $O(m \cdot n)$

Method 2

For a pre-sorted list, the target ID is compared with the middle value. If they are equal, the corresponding position is returned; if not, the list is divided into two sub-lists based on the middle value, and the search continues recursively in the appropriate sub-list. For an unsorted list, the provided list of ID values is first divided into n sub-lists, which are then sorted and merged until a single sorted list is generated. The time complexity for sorting the list is $O(n \cdot \log n)$, and the time complexity for searching a single element in the sorted list using binary search is $O(\log n)$. Therefore, for m elements, the total time complexity becomes $O(m \cdot \log n)$. As a result, the worst-case asymptotic time complexity is $O(n \cdot \log n + m \cdot \log n)$.

Figure

The graphs show that the runtime measurements for test1 and test2 differ only slightly. For method1, sorting the list has minimal impact, so we will not differentiate between sorted and unsorted lists. In contrast, list sorting significantly affects method2; thus, we will refer to it as method2_sorted for sorted lists and method2_unsorted for unsorted lists. The images reveal that method2_sorted generally outperforms both method1 and method2_unsorted. This finding contrasts with theoretical analysis, possibly due to random data selection issues or internal storage concerns.

Figure 1

Figure 1 shows that increasing n with a fixed m leads to longer runtimes. The theoretical time complexity for method1 is $O(n)$, while method2_sorted is $O(n \cdot \log n)$ and method2_unsorted is $O(\log n)$. The graphs reflect these complexities: method1's runtime follows the $O(n)$ curve, method2_unsorted aligns with $O(\log n)$, and method2_sorted corresponds to $O(n \cdot \log n + \log n)$. Generally, $O(\log n)<O(n)<O(n \cdot \log n)$, indicating that method2_sorted performs better than method1, which in turn performs better than method2_unsorted. However, in special cases where $O(n)$ is large, $O(n)$ approaches $O(n \cdot \log n)$.

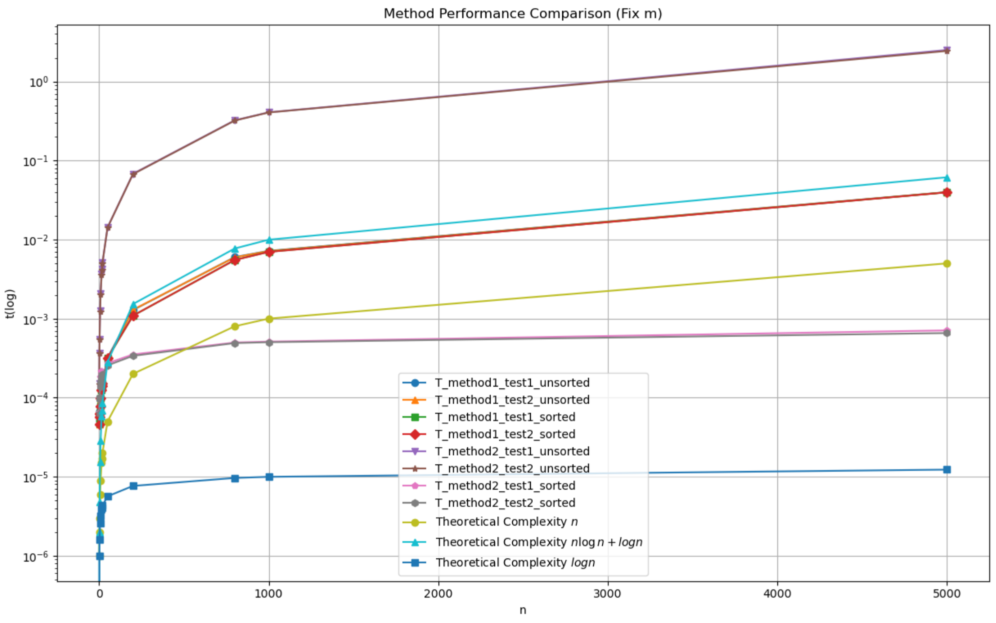


Figure 2

With a fixed value of n, the time complexity of method1 is $O(m)$, while the time complexity of method2_sorted is also $O(m)$, and the time complexity of method2_unsorted is likewise $O(m)$.For a given n, method2 generally outperforms method1, because under fixed n, the growth rate of $O(m \cdot \log n)$ is significantly slower than that of $O(m \cdot n)$. In most cases, when n is large and m increases, method2 exhibits higher search efficiency, leading to overall better performance compared to method1. In special cases, when n is very small, the simple linear search of method1 may perform slightly better due to the limited number of elements. Therefore, in general scenarios, as m increases, the performance of method2 typically surpasses that of method1.

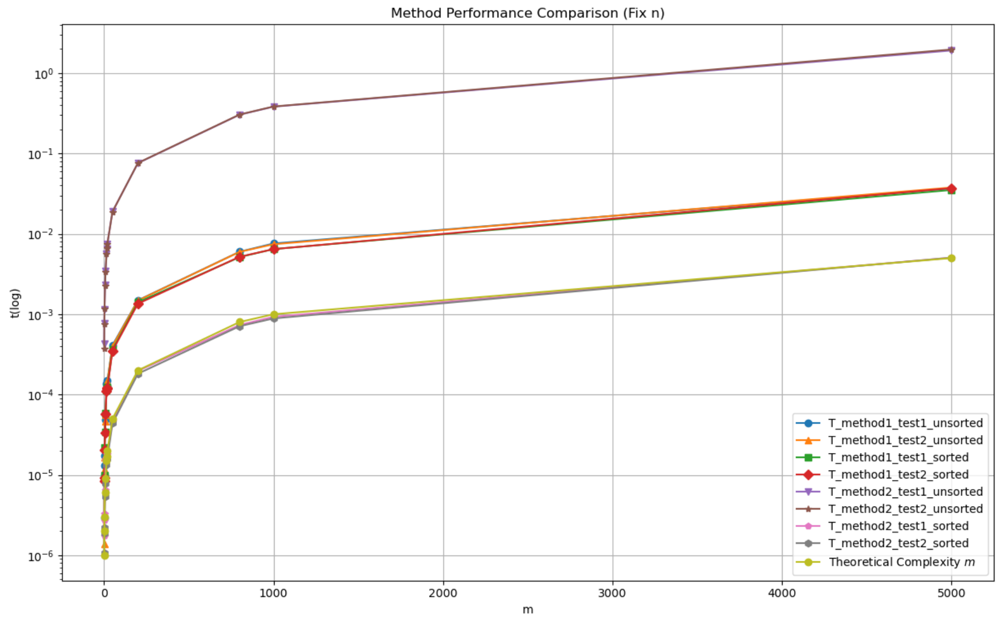
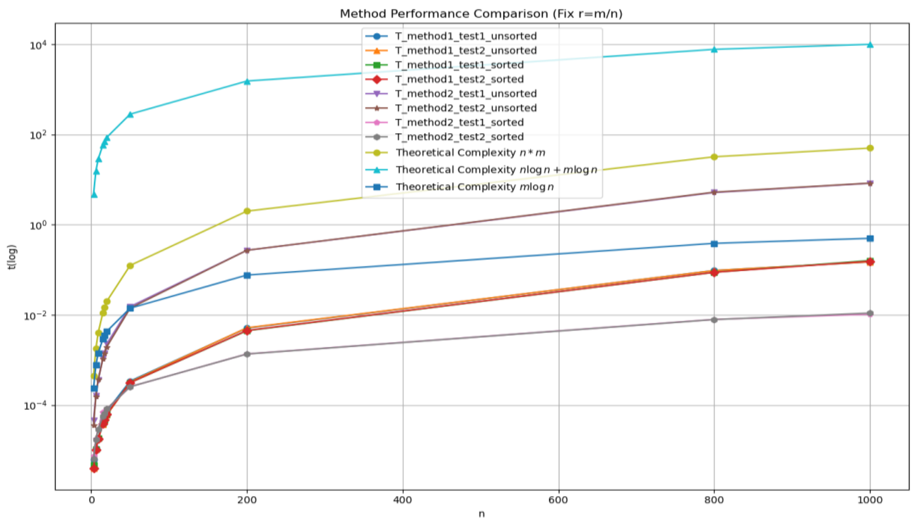


Figure 3

With a fixed value of $m = r \cdot n$, the time complexity of method1 is $O(n^2)$. The time complexity of method2_sorted is $O(n \cdot \log n)$, while the time complexity of method2_unsorted is $O(n \cdot \log n)$. In general, $O(n^2)$ is greater than $O(n \cdot \log n)$, so the theoretical performance of method2 is expected to be better than that of method1.



Part2

(a) Hash Map Construction

Use a hash function to calculate hash values for each substring of length ‘m’ in ‘A1’ and ‘A2’. A sliding window technique updates the hash value at each new position in constant time. Calculating hash values has a time complexity of $O(n)$. Each length ‘m’ substring is hashed in constant time, and the sliding window approach optimizes this by efficiently updating hash values, minimizing redundant calculations.

(b) Target Subsequence Match Search

The algorithm iterates over each pair ‘(sub1, sub2)’ in list ‘L’, calculating the hash value for each substring pair, which has has a time complexity of $O(m)$. It then checks if these hash values exist in ‘hashes_A1’ and ‘hashes_A1’ in constant time. If matching hash values are found, it verifies that ‘A2’ contains ‘sub2’ at the same position. Matching positions are stored in the result list ‘F’. Given that the length of ‘L’ is ‘l’, this step’s overall time complexity is $O(l \cdot m)$

In summary, the total time complexity of the algorithm is $O(n + l \cdot m)$. For large values of ‘n’, ‘l’, and ‘m’, this time complexity is efficient because:(i) The combination of hashing and the sliding window technique reduces redundant computations. (ii) Directly comparing hash values and positions enables fast filtering of matches, minimizing unnecessary character comparisons.