Analysis of Algorithms

BLG 335E

Project 1 Report

EMRE YAZICI
150220063@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 23.11.2024

1. Implementation

	small	medium	large
Counting Sort	0.2180 ms	0.4410 ms	1.3400 ms
Rarity Calculation	12.418 ms	38.411 ms	90.986 ms
Heap Sort	2.4480 ms	6.0960 ms	15.496 ms

Table 1.1: Comparison of our functions on input data (Different Sizes).

1.1. Sort the Collection by Age Using Counting Sort

First, we identify the maximum age in our dataset (O(n)). Next, we create a count array of length $(\text{maxAge} + 1)$ where each element is initialized to zero. This array will include all numbers from 0 to maxAge as indices. We then iterate through the dataset linearly and, for each item, increment the count array element at the index corresponding to the item's age. Iterating through the entire dataset is an $O(n)$ operation. Afterward, we process the count array to create a cumulative array. If sorting in descending order, we iterate through the array from end to start, or if sorting in ascending order, from start to end. During this process, we add each element to the previous one sequentially. This results in a cumulative array, and the operation has a complexity of $O(\text{maxAge})$. Finally, we linearly traverse the dataset again and place each item into the output array at the index specified by count[age - 1]. After placing an item, we decrement the count for that age in the count array. Filling the output array is an $O(n)$ operation. In total, the algorithm has a complexity of $O(n + \text{maxAge})$. Since we know maxAge is not significantly large in our dataset, this approach effectively allows us to perform sorting in $O(n)$ time.

1.2. Calculate Rarity Scores Using Age Windows (with a Probability Twist)

When calculating the rarity score, we examine the items within the age $\pm$ ageWindow range for each input item. Within this range, we calculate the probability of finding similar items by dividing the number of items with the same *type* and *origin* as the input item by the total number of items in the range. Using this probability, we compute the rarity score with the formula:

$$(1 - \text{probability}) \times \left(1 + \frac{\text{item.age}}{\text{maxAge}}\right).$$

To optimize the process of finding items within the ageWindow, we leverage the fact that the dataset is sorted by age. Instead of iterating through the entire dataset for each item, we construct a cumulative count array (similar to counting sort). This allows us to determine the range of indices for the desired age range directly in $O(1)$ time as

$[\text{count[min-1]}, \text{count[max]} - 1]$. Consequently, calculating the rarity score for each item is reduced from $O(n)$ to $O(2 \times \text{ageWindow})$.

The total complexity involves:

1. Creating the count array, which is $O(n + \text{maxAge})$.
2. Calculating the score for each of the n items in $O(1)$, resulting in $O(n)$ overall.

Thus, the total complexity is $O(n + \text{maxAge})$. Since maxAge is known to be a small number in our dataset, the operations are effectively performed in $O(n)$ time. The fact that the dataset is sorted helps reduce the complexity from $O(n^2)$ to $O(n)$, enabling efficient computation.

1.3. Sort by Rarity Using Heap Sort

First, we organize our array to conform to heap properties. This is achieved by performing the *heapify* operation on all non-leaf nodes. During this process, any *heapify* operation can occur from the maximum height down to the lowest level, resulting in a maximum complexity of $O(\log n)$ per operation. Since we iterate over all non-leaf nodes using a loop, the heap construction process has a complexity of $O(n)$. Therefore, the process of organizing the array according to heap rules has a total complexity of $O(n \log n)$. Depending on the desired sorting order, we use a max-heap for ascending and a min-heap for descending order. Once the heap is constructed, we enter a recursive process. Initially, the root (the first element) is swapped with the last element. Then, considering the array as if its size is reduced by one, a *heapify* operation is performed. This process continues until we are left with a heap of size 1. Since this recursive process involves n steps, each with a complexity of $O(\log n)$, the sorting step has a complexity of $O(n \log n)$. In total, the algorithm achieves a complexity of $O(n \log n)$.