Analysis of Algorithms

BLG 335E

Project 2 Report

Ali Emre Kaya
kayaemr21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 20.11.2024

1. Implementation

1.1. Sort the Collection by Age Using Counting Sort

Counting Sort is a non-comparison-based sorting algorithm that requires two auxiliary arrays besides the original input array, which we call **items**. The first auxiliary array, **C**, stores the count of occurrences of each value in **items**, and its size is determined by the maximum value in **items**. To populate **C**, we traverse the **items** array, counting the occurrences of each value. After this, **C** is updated to store prefix sums, which indicate the final positions where each value should be placed. Before proceeding with the sorting, the second auxiliary array, **sorted**, is created with the same size as **items**.

The sorting process begins by traversing the **items** array in reverse order to ensure stability, as **C** holds the last indexes for each value. For each value in **items**, its position is determined using **C**, and the value is placed into the **sorted** array. After placing the value, the corresponding count in **C** is decremented to reflect the next available position. This process continues until all elements are placed in **sorted**, producing an array sorted in ascending order. If descending order is desired, a simple $\frac{N}{2}$ swap operation suffices.

The time complexity of Counting Sort is $O(N + \max(\textbf{items}))$, where $O(N)$ comes from traversing the **items** array twice (once to count occurrences and once to place the elements in the **sorted** array), and $O(\max(\textbf{items}))$ comes from initializing and updating the **C** array, because of the O is asymptotic upper bound time complexity represent like O(N). The space complexity is also $O(N + \max(\textbf{items}))$, as the **C** array requires $\max(\textbf{items}) + 1$ space to store the counts and prefix sums, while the **sorted** array requires N space to hold the sorted result, because of the O is asymptotic upper bound space complexity represent like O(N). These complexities make Counting Sort efficient for arrays where the range of values (maximum value) is not significantly larger than the size of the array.

When analyzing the running times in microseconds for three different dataset sizes, the following pattern was observed: the dataset with 40k rows took **3346 microseconds**, the dataset with 20k rows took **1276 microseconds**, and the dataset with 10k rows took **619 microseconds**. These measurements provide strong evidence that the **Counting Sort** algorithm exhibits **linear time complexity** of $O(n)$, where n represents the number of elements in the dataset.

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

To calculate rarity scores, the first step is to find the maximum age in the array. This is achieved using a function, `getMax`, which traverses the array and returns the maximum age value. The time complexity of `getMax` is $O(N)$, where N is the size of the array.

For each item in the array, another traversal is performed to calculate the variables `totalCount` and `similarCount`, which takes $O(2 \cdot \text{ageWindow})$ time for each item due to the specified range. Afterward, the rarity score is calculated using the given rarity function. Overall, calculating the rarity score for all items involves running `getMax` in $O(N)$ and evaluating the rarity function for all items, which takes $O(N \cdot \text{ageWindow} \cdot 2)$. As the asymptotic upper bound dominates, the total time complexity of the algorithm is $O(N \cdot \text{ageWindow})$.

Additionally, when calculating the rarity for different dataset sizes for age window variable defined as 50, the following times were recorded: **966 microseconds** for the large dataset, **392 microseconds** for the medium dataset, and **200 microseconds** for the small dataset.

1.3. Sort by Rarity Using Heap Sort

The heap sort algorithm relies on the recursive `heapify` function to build and maintain the heap structure. The `heapSortByRarity` function orchestrates the sorting process by repeatedly calling `heapify` and ensures that the array is sorted by `rarityScore`. To simplify the relationship between parent and child nodes in the heap, a dummy item is inserted at the beginning of the array, effectively converting it into a 1-indexed structure. This eliminates the complexity of calculating parent-child relationships in a 0-indexed array. After sorting is complete, the dummy item is removed.

The `heapify` function starts by assuming that the current node, indexed by `i`, is the parent node. It calculates the indices of the left and right children as $2i$ and $2i + 1$, respectively. Depending on the `descending` parameter, the function determines whether to maintain a min-heap or max-heap structure. For a min-heap (`descending = true`), the function selects the smallest value among the parent and its children. For a max-heap (`descending = false`), it selects the largest value. If a child violates the heap property, it is swapped with the parent, and the `heapify` function is recursively called on the affected child to ensure the heap property is maintained throughout the subtree.

The `heapSortByRarity` function begins by constructing the heap. This is achieved by calling `heapify` on all non-leaf nodes in reverse order, starting from $n/2$ down to 1. Once the heap is built, the sorting phase begins. The largest (or smallest) item in the heap, located at the root, is swapped with the last unsorted item in the array. The size of the heap is reduced by one, and `heapify` is called on the root to restore the heap property. This process repeats until the heap is empty, resulting in a sorted array. Finally, the dummy item is removed from the array to restore its original structure.

The time complexity of `heapify` is $O(\log N)$, as it operates along the height of the tree. Building the heap requires $O(N)$, as each call to `heapify` on non-leaf nodes takes progressively less time. Sorting involves $O(N \log N)$, as each of the N elements is swapped and re-heapified. Therefore, the overall time complexity of heap sort is $O(N \log N)$. And space complexity of heap sort is O(1). This approach ensures an

efficient and stable sorting mechanism with clear parent-child relationships facilitated by the use of a dummy item.

When examining the performance measurements of **Heap Sort**, we observe that it took **3162 microseconds** to sort a non-sorted list and **2458 microseconds** to sort an already sorted list. These results indicate that **Counting Sort** is faster because it's performance for same datasets is that **1163 microseconds** to sort a non-sorted list and **870 microseconds** to sort an already sorted list (both count sort measurements measured sorting by age), as Heap Sort operates with a time complexity of $O(n \log n)$ and Count Sort is $O(n)$. However, it is important to note that **Counting Sort** cannot handle **floating-point numbers**. In cases where sorting data such as floating-point numbers is required, there is no faster method than $O(n \log n)$.