# Analysis of Algorithms

## BLG 335E

# Project 2 Report

Kemal Tahir Bıcılıoğlu
bicilioglu21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 20/11/24

# 1. Implementation

## 1.1. Sort the Collection by Age Using Counting Sort

### 1.1.1. Technical Details

$\rightarrow$ Counting sort works by counting the occurrences of each value in the input and using that information to determine where each value should be placed in the sorted output. It starts by finding the smallest and largest values in the data to calculate the range. Then, it creates a count array where each index represents a possible value, and the array stores how many times each value appears. These counts are adjusted to show the final positions of the values in the sorted array. To maintain stability, the algorithm processes the input in reverse order, ensuring that items with the same value are placed in the order they appear, as the last positions indicated by the count array are used first. This makes counting sort both efficient and stable for small and continuous ranges of values. Here are the measured runtime values of the counting sort in ascending order by the age parameter using the provided datasets, along with a comparison to the standard sorting function:

| | **items_s** | **items_m** | **items_l** |
|---|---|---|---|
| **Counting Sort** | 1ms | 2ms | 6ms |
| **std::sort** | 4ms | 9ms | 23ms |

**Table 1.1:** Comparison of runtime measurements on input data (Different Size).

1. Counting Sort
   (a) Time Complexity
      i. Best Case: $\Omega(n + range)$
      ii. Average Case: $\theta(n + range)$
      iii. Worst Case: $\mathcal{O}(n + range)$
      iv. Space Complexity: $\mathcal{O}(n + range)$
   (b) Advantages:
      i. Fast. It has linear time running time complexity.
      ii. Stable Sorting. Counting sort preserves the relative order between the objects with the same key.
   (c) Disadvantages:
      i. Requires additional space proportional to the range of values.
      ii. Not suitable for non-integer or large-range sorting tasks.

### 1.1.2. Analyzing Results

$\rightarrow$ Counting sort is a fast and efficient algorithm, especially for sorting numbers or categories within a small and continuous range, as it works in linear time. The linear time complexity of counting sort can be easily observed from the running time measurements. It doesn't compare elements directly, making it different from many other sorting methods, and it can outperform them by avoiding the $\mathcal{O}(n \log n)$ time complexity of comparison-based algorithms. It also maintains the relative order of items with the same value (a property known as stable sorting) where secondary attributes need to stay in order. However, its use is limited because it relies heavily on the range of values, requiring a lot of memory if the range is large. This can lead to inefficiency, especially when there are gaps between the values, as memory is wasted on unused slots. It also struggles with very large datasets or data types like floating-point numbers, where other sorting methods may work better.

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

- Formula for the Rarity Score is given as the following:

$$\text{rarityScore}_i = (1 - P_i) \cdot \left(1 + \frac{\text{age}_i}{\text{maxAge}}\right)$$

$$P_i = \begin{cases} \frac{\text{countSimilar}_i}{\text{countTotal}_i}, & \text{if countTotal}_i > 0 \\ 0, & \text{otherwise} \end{cases}$$

where: - $P_i$ is the probability of similar items within the age window, - $\text{age}_i$ is the age of the current item, - maxAge is the maximum age among all items.

### 1.2.1. Technical Details

$\rightarrow$ A brute force approach to calculating the rarity scores involves using two nested loops. The outer loop iterates over each item, and the inner loop checks all other items to determine if their ages fall within the current item's age window. For each item in the window, the total count is incremented, and if the type and origin match, the similar count is also incremented. This method is straightforward but inefficient, as it results in $\Omega(n * ageWindow)$ time complexity even in best case scenario, making it unsuitable for larger datasets.

$\rightarrow$ In contrast, my implementation optimizes this process by leveraging a sliding window approach. Since many items share the same age, the age window often remains unchanged across consecutive iterations. Instead of recalculating the entire window for each item, I maintain two pointers (left and right) to represent the current window.

These pointers are adjusted only when the age changes, minimizing unnecessary updates. This allows the algorithm to reuse the existing window, avoiding the second loop and significantly reducing the computational overhead. As a result, the time complexity is reduced to $\theta(n)$ average time complexity, making the algorithm much faster and more efficient for datasets with repeated age values. However the worst case scenario is always $\mathcal{O}(n * ageWindow)$ due to the nature of the rarityScore calculation.

### 1.2.2. Defining&Analyzing Different Rarity Score Calculations

$\rightarrow$ The current rarity score calculation considers both the similarity of an item to others in its age window. This approach results in higher rarity scores for items with older ages, regardless of their actual rarity, because the age contributes to the score positively. While this may be suitable in certain contexts, it emphasizes age over the item's true rarity which can be replaced with another formula.

**Alternative Rarity Score Calculation:**

$$\text{rarityScore}_i = 1 - \frac{\text{countSimilar}_i}{\text{totalItems}}$$

calculates how unique an item is within the entire dataset. Here, $\text{countSimilar}_i$ represents the number of items that are similar to the current item based on specific criteria type and origin while totalItems is the total number of items in the dataset.

■ Another complex formula can be:

$$\text{rarityScore}_i = \frac{1}{1 + \frac{\text{countSimilar}_i}{\text{totalItems}}} \times \left(1 + \frac{\text{stdDev}_T}{\text{mean}_T}\right)$$

This formula for rarity score incorporates two main components: the frequency component and the variance component. The first term, $\frac{1}{1+\frac{\text{countSimilar}_i}{\text{totalItems}}}$, captures how often an item appears among similar items, reflecting its rarity. The second term, $\left(1 + \frac{\text{stdDev}_T}{\text{mean}_T}\right)$, adjusts for the diversity of item types within the entire set. This formula provides a more nuanced and reliable rarity score by considering both item frequency and type diversity, making it a better measure than simpler methods that only account for item counts or age.

## 1.3. Sort by Rarity Using Heap Sort

### 1.3.1. Technical Details

$\rightarrow$ Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure. It begins by building a max-heap (or min-heap, depending on the sorting order) from the input data. In a max-heap, each parent node is greater than or equal to

its children, while in a min-heap, each parent node is less than or equal to its children. The heap is built by calling the heapify function on each internal node, starting from the last non-leaf node down to the root.
Once the heap is constructed, the sorting process begins by repeatedly swapping the root element (which is the largest or smallest, depending on the heap type) with the last element in the heap. The size of the heap is then reduced by one, and the heapify operation is called to restore the heap property. This process continues until all elements are extracted from the heap and placed in their correct position in the sorted output.
Here are the measured runtime values of the heap sort in ascending order by the age parameter using the provided datasets, along with a comparison to the standard sorting function:

| | **items_s** | **items_m** | **items_l** |
|---|---|---|---|
| **Heap Sort** | 4ms | 8ms | 25ms |
| **std::sort** | 4ms | 9ms | 23ms |

**Table 1.2:** Comparison of runtime measurements on input data (Different Size).

1. Counting Sort
    (a) Time Complexity
        i. Best Case: $\Omega(n \log n)$
        ii. Average Case: $\theta(n \log n)$
        iii. Worst Case: $\mathcal{O}(n \log n)$
        iv. Space Complexity: $\mathcal{O}(1)$
    (b) Advantages:
        i. Time Complexity: Heap sort guarantees a time complexity of $\mathcal{O}(n \log n)$.
        ii. In-place Sorting: Heap sort is an in-place algorithm, meaning it does not require any additional memory.
        iii. Non-recursive, Heap sort can be implemented without recursion, unlike algorithms like quicksort or merge sort, making it easier to control stack depth and more suitable for environments with limited stack space.
    (c) Disadvantages:
        i. Not Stable: Heap sort is not a stable sorting algorithm, meaning that it does not preserve the relative order of equal elements.

### 1.3.2. Analyzing Results

$\rightarrow$ Heap sort is an efficient comparison-based sorting algorithm with a time complexity of $\mathcal{O}(n \log n)$ in all cases, making it more predictable than algorithms like quicksort, which can degrade to $\mathcal{O}(n^2)$ in the worst case. The time complexity of heap sort can be easily observed from the running time measurements. Heap sort operates by using a binary heap data structure, which allows it to repeatedly extract the largest (or smallest) element in logarithmic time. One of the key advantages of heap sort is that it is an in-place sorting algorithm, meaning it does not require additional memory beyond the input array, unlike algorithms such as merge sort. However, while heap sort is reliable and does not suffer from worst-case performance issues like quicksort, it has a higher constant overhead due to the repeated heapify operations. This can make it slower in practice for smaller datasets compared to other $\mathcal{O}(n \log n)$ algorithms, especially quicksort. Additionally, heap sort is not a stable sorting algorithm, meaning it does not preserve the relative order of equal elements. This may be a limitation in certain applications where stability is crucial.