Analysis of Algorithms

BLG 335E

Project 2 Report

Feyza Nur Keleş
kelesf22@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: DATE HERE

1. Implementation

1.1. Sort the Collection by Age Using Counting Sort

1.1.1. Theoretical Background

Counting Sort is a non-comparative sorting algorithm that operates by counting the occurrences of each element in the dataset and calculating their positions in the sorted array. It works efficiently when the range of the data (i.e., the age of items) is not significantly larger than the number of elements to be sorted.

1.2. Technical Details

The Counting Sort algorithm is designed to sort items based on the "age" attribute. The steps of the algorithm are as follows:

- **Initialization:** First, the algorithm checks if the attribute to be sorted is "age". If not, the function simply returns the original list of items. The maximum value of the "age" attribute is then determined by the `getMax()` function.
- **Counting Occurrences:** A count array of size max_val $+ 1$ is initialized to zero. The algorithm iterates through the list of items, counting the occurrences of each age and storing these counts in the count array.
- **Cumulative Count Calculation:** The count array is then modified to store the cumulative count. This step helps in determining the correct position of each item in the sorted array.
- **Placing Items in Sorted Order:** The items are placed into a new array called `sorted`. Each item is placed in its correct position based on the cumulative count array.
- **Reversing the Array for Descending Order:** If the sorting order is descending, the `sorted` array is reversed manually by swapping elements from both ends.

The algorithm operates with a time complexity of $O(n + k)$, where n is the number of items and k is the range of the "age" values. This allows the algorithm to sort large datasets in linear time, provided the range of ages is not too large. For ascending order sorting, the array is built in increasing order, while for descending order, the array is reversed.

1.2.1. Execution Time for Counting Sort on Different Datasets

Counting Sort has a time complexity of $O(n + k)$, where n is the number of elements and k is the range of input values. It is a linear time sorting algorithm, making it much

Dataset	Execution Time (Microseconds)
Large Dataset	1046
Medium Dataset	548
Small Dataset	222
Reverse Sorted Large Dataset	1732
Already Sorted Large Dataset	1732
All Ages Are Equal and 0 Large Dataset	1218 microseconds

Table 1.1: Execution Time for Counting Sort on Various Datasets

faster than comparison-based algorithms like Heap Sort for datasets with small integer ranges. As the dataset size increases, the execution time increases linearly with the number of elements, but if the range of the values k is large, performance can degrade.

1.2.1.1. Efficiency with Varying Inputs

For the large dataset, the execution time is 1046 microseconds. Since Counting Sort operates in linear time with respect to n, this is a reasonable result for this dataset size. When the dataset size is reduced to medium and small, the execution time drops to 548 and 222 microseconds, respectively, which is expected given the linear time complexity. For the reverse-sorted and already sorted datasets, the execution time remains almost the same, which is characteristic of Counting Sort as it does not depend on the initial order of the data.

1.2.1.2. Best Case Scenario

The best-case scenario occurs when the input data is all the same. In this case, counting sort has a time complexity of $O(n)$ because while n is the number of elements, k was the range of the input data which is 1 in this case since all the datas are the same therefore it would be n+1 which can be shown as $O(n)$ complexity.

1.2.1.3. Worst Case Scenario

The worst-case scenario arises when the input data is sorted in reverse order. While Counting Sort's time complexity remains $O(n + k)$, the actual execution time increases as more operations are required to position each element.

1.2.2. Analyzing Results

The execution times for various scenarios with Counting Sort shows the following:

- For the large dataset, the execution time is 1046 microseconds, which is efficient for Counting Sort's linear time complexity.

- As the dataset size decreases, the execution time also decreases, with the medium dataset taking 548 microseconds and the small dataset taking 222 microseconds.
- The reverse-sorted and already sorted datasets show no change in execution time, confirming that Counting Sort is not sensitive to input order.
- The dataset with all ages equal and set to 0 shows a slightly higher time of 1218 which is the best case scenario because execution time doesn't depend on the order but the amount of different elements.

Based on the results, counting sort performs best when all the input values are the same. This is because the algorithm only needs to update a single count and the range k is minimal, making the process very efficient. On the other hand, the worst case occurs with reverse-sorted data. Even though counting sort doesn't depend on the order of elements, the larger range k increases the size of the count array, which adds overhead. The results show that the algorithm is more sensitive to the range of input values (k) rather than the initial order of the data. This highlights why inputs with a small range are more optimal for counting sort.

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

1.3.1. Rarity Calculation Using Age Windows

To calculate rarity, we look at other items within a certain "near-age window" (e.g., ±50 years). For each item, we calculate a probability P of how common the item's 'type' and 'origin' are in this window.

The rarity score R for each item is calculated using the formula:

$$R = (1 - P)\left(1 + \frac{\text{age}}{\text{agemax}}\right)$$

where P is the probability, calculated as:

$$P = \frac{\text{countSimilar}}{\text{countTotal}} \quad \text{if countTotal} > 0$$

and $P = 0$ otherwise.

1.3.1.1. Alternative Rarity Score Calculations

- **Fixed-size Age Window**: This approach uses a constant age window (e.g., ±50 years) to compare an item to other items. It ensures consistency but may not account for varying item distributions across time.
- **Dynamic Age Window**: Instead of using a fixed window size, this approach dynamically adjusts the window based on the distribution of item ages. This can provide more accurate rarity scores, especially for datasets with uneven distributions.
- **Rarest Element in Each Age Period**: Here, the rarity score is based on finding the rarest element in each age period, without directly using the age of the items. This formula captures the uniqueness of an item in a given period, rather than a global context.
- **Larger Age Windows**: Larger age windows (e.g., ±100 years) are used to capture broader historical contexts. This approach allows for a more comprehensive analysis but may reduce the sensitivity of the rarity score.

1.4. Sort by Rarity Using Heap Sort

1.4.1. Theoretical Background

Heap Sort is a comparison-based sorting algorithm that utilizes a binary heap to sort items. It repeatedly extracts the root element (the largest or smallest element) and reorders the heap. Heap Sort runs in $O(n \log n)$ time, making it more efficient for large datasets compared to simpler algorithms like Bubble Sort.

Heap Sort has a time complexity of $\mathbf{O(n \log n)}$, even in the best case, because;

1.4.1.1. Heap Construction

Building the heap requires iterating through all the nodes and potentially "heapifying" each one. This step generally has a time complexity of:

$$O(n)$$

This is because:

- Heapifying a node takes $O(\log n)$, but most nodes are near the bottom of the tree and require fewer operations.
- When all values are equal, the number of adjustments (comparisons and swaps) is minimized. However, the algorithm still examines all nodes, so the complexity remains $O(n)$.

1.4.1.2. Sorting Phase

Extracting the maximum element (root) from the heap and re-heapifying takes $O(\log n)$ for each of the n elements. Therefore, this step requires:

$$n \times O(\log n) = O(n \log n)$$

Therefore the total time complexity is the sum of the heap construction and sorting phases becomes this even for the best case:

$$O(n) + O(n \log n) = O(n \log n)$$

1.4.2. Technical Details

We used Heap Sort to sort items based on their ‘rarityScore‘. The heap is built by iterating through the array, and then the root element (maximum or minimum) is repeatedly swapped with the last element in the heap, followed by heapifying the remaining elements.

1.4.3. Execution Time for Heap Sort on Different Datasets

Dataset	Execution Time (Microseconds)
Large Dataset	17462
Medium Dataset	6366
Small Dataset	2974
Reverse Sorted Large Dataset	14826
Already Sorted Dataset	15240
All Rarities Are equal and 0 Large Dataset	1375

Table 1.2: Execution Time for Heap Sort on Various Datasets

1.4.4. Benchmarking Heap Sort Algorithm

Heap Sort shows an expected $O(n \log n)$ time complexity across varying dataset sizes. As the dataset size increases, the execution time increases logarithmically. The performance is independent of the dataset order, as Heap Sort does not exploit pre-sorted or reverse-sorted data. The algorithm is effective for medium and small datasets, but its performance becomes less optimal for larger datasets compared to algorithms like Counting Sort, which operates in linear time.

1.4.4.1. Efficiency with Varying Inputs

The efficiency of Heap Sort varies with the structure and distribution of the input data. While its worst-case, best-case, and average-case asymptotic complexity are all $O(n \log n)$, the actual runtime can differ based on factors like the degree of sorting or uniformity in the data.

1.4.4.2. Best Case Scenario

The best case for Heap Sort is when all elements in the dataset are equal. In this case:

- Minimal adjustments are required during both the heap construction and sorting phases because the heap property is inherently satisfied.
- From the observed timings, when all values are equal (e.g., "All Rarities Are Equal and 0 Large Dataset"), the runtime is significantly reduced compared to other cases. For instance:

  Time taken: 1375 ms (much lower than other cases).

- Despite fewer operations, the asymptotic complexity remains $O(n \log n)$ since the algorithm still iterates through all nodes and calls heapify function and does comparison operation.

1.4.4.3. Worst Case Scenario

The worst-case scenario for Heap Sort typically involves highly disordered input, which requires the maximum number of adjustments during heap construction and sorting. From the observations:

- The "Reverse Sorted Large Dataset" takes the longest time, e.g.:

$$\text{Time taken: } 14826 \text{ ms.}$$

- This is due to the significant number of swaps and comparisons required to maintain the heap property at each stage.

1.4.4.4. Other Observations

Intermediate cases like the "Large Dataset" (17462) and "Already Sorted Dataset" (15240) suggest that Heap Sort performs moderately well on partially sorted or random data. However:

- The performance on "Already Sorted Dataset" indicates that it does not exploit existing order, unlike some other algorithms like Merge Sort or Insertion Sort.
- The performance difference between these datasets highlights that Heap Sort is input-insensitive in terms of complexity but not necessarily runtime efficiency.

1.4.5. Analyzing Results

- For the large dataset, the execution time (17462 microseconds) is consistent with the expected performance of Heap Sort at $O(n \log n)$.
- Medium and small datasets take significantly less time (6366 and 2974 microseconds, respectively), demonstrating Heap Sort's scalability.
- The reverse-sorted and already sorted datasets show similar times, reaffirming Heap Sort's insensitivity to input order.
- The dataset with all rarities equal and set to 0 executes faster (1375 microseconds), though the size remains the dominant factor.
- The performance on larger datasets is slower compared to linear time algorithms like Counting Sort, making Heap Sort less optimal for very large data.

In conclusion, while Heap Sort maintains predictable performance across various dataset configurations, it is not the most efficient choice for large datasets compared to algorithms like Counting Sort, which are optimized for specific input types or sizes.