Analysis of Algorithms

BLG 335E

Project 2 Report

Berra MUTLU
mutlub22@itu.edu.tr

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

1. Implementation

1.1. Sort the Collection by Age Using Counting Sort

Counting Sort is a non-comparative, integer-based sorting algorithm. It is particularly efficient when sorting datasets where the range of values (keys) is relatively small compared to the number of items being sorted. Counting Sort works by counting how many times each distinct value appears in the dataset, and then using these counts to calculate the position of each element in the final sorted array.

The Counting Sort algorithm is well-suited for sorting items based on specific attributes (like age, type, or origin), where the values for those attributes are constrained to a small range. Given the constraints of the dataset (integer-based attributes), Counting Sort is ideal because:

It can sort in `O(n + k)` time complexity, where n is the number of items and k is the range of values. This is linear in time complexity when k is much smaller than n, making it highly efficient when the range of values is not excessively large. It doesn't require comparisons between the elements, making it faster than comparison-based algorithms (like QuickSort or MergeSort) for datasets with a small range of values.

It only works for sorting integer values or categorical data where the range is known and relatively small. If the range of values is very large, Counting Sort can become inefficient and memory-intensive.

1.1.1. Implementation

- **Attribute Selection:** Finding the attribute to be sorted—either age, type, or origin—is the first step of the function. An if-else chain is used for this. The project assignment asked to implement the code for only age sort but I added other attributes as well.
- **Counting Occurrences:** The algorithm then uses an auxiliary count array to determine how many times each value of the chosen property occurs. A distinct value of the property (such as the quantity of objects of a specific age or kind) is associated with each index in the count array.
- **Cumulative Sum:** The algorithm determines the ultimate position of each item in the sorted output by computing the cumulative total of counts after counting the occurrences.
- **Placing Items:** Finally, the algorithm places each item at the correct position in the output array by using the cumulative counts and updates the count array to reflect the next occurrence of the same value.

	Small Data Set	Medium Data Set	Large Data Set
Counting Sort by Age	0.79 ms	1.56 ms	4.07 ms

Table 1.1: Comparison of counting sort by age on input data (Different Size).

The Counting Sort took 0.79 ms for small datasets. With timings of 1.56 ms for medium datasets and 4.07 ms for large datasets, the performance scaled reliably even as the dataset size rose. This behavior is consistent with Counting Sort's theoretical O(n + k) time complexity. The algorithm's time complexity is practically linear with respect to the number of items in the dataset (n), as the attribute's range (age) remains constant throughout dataset sizes.

Counting when sorting by discrete attributes, such as age, where there is a relatively small range of possible values (k), Sort is quite effective. The growing number of items (n) causes a slight rise in time for larger datasets, although this development is controlled and gradual.

In conclusion, counting Sort is an excellent choice for sorting by attributes like age, type, or origin, especially when the dataset contains categorical or bounded integer values.

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

The rarity score of an item is calculated based on its similarity to other items within an "age window" (a range of ages).

Formula:

`P = similarCount / totalCount`, where `similarCount` is the number of items within the age window that share the same type and origin, and `totalCount` is the number of items in the age window.

`rarityScore = (1 - P) * (1 + item.age / ageMax)` to reflect both the dissimilarity (via P) and the normalized age (via ageMax).

	Small Data Set	Medium Data Set	Large Data Set
Rarity Score for 50	773.08 ms	3029.28 ms	18252.8 ms

Table 1.2: Comparison of rarity score calculation on input data (Different Size).

The rarity score calculation involves iterating over all items in the dataset and comparing each item against every other item within its age window. The process is effectively a nested loop:

Outer Loop: Iterates through each item in the dataset (n iterations). **Inner Loop:** For each item, iterates through all other items to count the number of similar items within the specified age window. This results in up to n comparisons per item. This leads to an approximate time complexity of $O(n^2)$, where n is the number of items in the dataset.

For a small dataset, the number of items (n) is relatively low. The quadratic nature of the algorithm is not as pronounced, resulting in lower computation time. With a medium dataset, the number of items (n) increases. As n^2 grows quadratically, the total computation time increases significantly—approximately 4 times compared to the small dataset. As the size of the dataset grows, it is clear that operations $O(n^2)$ rise quadratically. The anticipated scaling behavior is reflected in the time measurement. For a large dataset, the quadratic growth dominates. The computation time increases drastically—approximately 6 times compared to the medium dataset.

1.2.1. Different Rarity Score Calculations:

- **Fixed-Sized Age Window:** This approach divides the age range into fixed-size bins, such as 0–50 and 51–100. The items in the same bin are used to calculate rarity. The percentage of items with the same category and origin in the same bin determines how rare each item is.

  `P = similarCount / totalCount`, Where `similarCount` is number of items in the same age bin with the same type and origin and `un`totalCot is total number of items in the bin. So the final formula is `rarityScore = (1 - P)`.

  It simplifies computation by focusing on pre-defined age ranges. Useful for datasets where items are naturally grouped by discrete age periods. On the other hand, items near bin boundaries may not fully represent their rarity compared to adjacent bins. Fixed bins may not adapt well to datasets with uneven age distributions.

- **Larger Age Window:** This approach extends the current age window to include more neighboring items. By taking into account more objects in the comparison, a bigger window size smoothes the rarity score.

  Similar to the original formula, but with an expanded age window, `min = age - 2 * ageWindow`, `max = age + 2 * ageWindow`. So, `rarityScore = (1 - P) * (1 + age / ageMax)`.

  It reduces the variation in rarity ratings brought on by little sample sizes. More appropriate for datasets that are sparse, with few entries per age range. On the contrary, over-smoothing might make truly uncommon goods less identifiable and increased computational cost due to the larger window size.

- **Rarity Without Age Dependency:** In this method, the age attribute is removed entirely from the rarity calculation. Rarity is determined solely by the frequency of items with the same type and origin.

  The formula is found by `P = similarCount / totalCount`, so `rarityScore = (1 - P)`. It may seem same as the fixed-sized age window approach but there is a big difference which is, in this approach we ignore the age attribute completely instead of giving it a fixed size.

  It restricts the calculation to categorical rarity, making it simpler. Performs well in datasets where the primary factors of interest are type and origin. But at the same time, it ignores temporal information, which could be critical in datasets where age provides meaningful context for rarity.

- **Rarest Element in Each Age Period:** This method identifies the rarest item in each age period and calculates the rarity of other items in the same period based on their similarity to the rarest item.

  For finding the formula, we should divide items into age periods, find the least frequent type and origin combination within each period. Finally, assign scores inversely proportional to their frequency.

  This method highlights exceptionally rare items within their temporal context and it is useful for datasets with distinct clusters of ages. Although, uneven rarity score distribution across bins could make comparisons difficult plus, it requires additional normalization to balance scores across periods.

Method	Strengths	Weaknesses
Fixed-Sized Age Window	Simple; aligns with pre-defined temporal bins.	Items near bin boundaries may not represent their neighbors well.
Larger Age Window	Smoother scores; better for sparse data.	Over-smoothing reduces distinctiveness of rare items; higher computational cost.
Rarity Without Age	Simplifies calculation; focuses purely on categorical rarity.	Ignores temporal context; may miss patterns tied to age.
Rarest Element in Each Period	Highlights exceptionally rare items within temporal contexts.	May require normalization across bins to ensure consistent scores.

Table 1.3: Comparative summary of rarity score calculation methods.

1.3. Sort by Rarity Using Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. A binary heap is a complete binary tree that satisfies the heap property: for a max-heap, the parent node's value is greater than or equal to that of its children; for a min-heap, the parent node's value is smaller than or equal to its children.

In this assignment, Heap Sort is used to sort the items by their **rarityScore**, either in ascending or descending order, depending on the descending flag passed to the function.

In contrast to algorithms like QuickSort, which can degenerate to $O(n^2)$ in the worst scenario, Heap Sort has a guaranteed time complexity of $O(n \log n)$ in the worst situation, which is why it was selected. For larger datasets, this ensures a consistent performance, which is crucial. Additionally, it is an in-place sorting algorithm, which means that, aside from a few auxiliary variables, it does not require additional space proportional to the input size. This makes it more space-efficient than techniques such as MergeSort.

1.3.1. Implementation

- **Heap Construction:** The algorithm starts by using the unsorted data to create a max-heap. In order to extract the maximum element (in ascending order), a max-heap makes sure that the greatest element is always at the heap's root.
- **Heapify Process:** After elements are extracted, the heap property is preserved using the heapify function. To preserve the heap structure, it compares the heap's root with its left and right children and switches them if needed.
- **Extract Elements:** The heap size is decreased by one when the max-heap is constructed, and the largest element (the root) is switched with the array's final element. The heap attribute is then restored by calling the heapify method. Until every element is sorted, this procedure keeps on.
- **Descending or Ascending Sorting:** Maintaining a max-heap is how the function manages ascending sorting. The heap is still built as a max-heap for descending sorting, but the extracted elements are inverted after extraction.

	Small Data Set	**Medium Data Set**	**Large Data Set**
Heap Sort by Rarity (Descending)	3.38 ms	7.30 ms	20.12 ms
Heap Sort by Rarity (Ascending)	3.20 ms	7.03 ms	18.33 ms

Table 1.4: Comparison of heap sort by rarity on input data (Different Size).

Sorting by rarity score in descending order took 3.38 ms, 7.30 ms, and 20.12 ms for small, medium, and large datasets, respectively. The observed performance aligns with the theoretical O(n log n) complexity of Heap Sort. As the dataset size grows, the logarithmic factor contributes more significantly to the runtime.

Heap Sort is slower than Counting Sort for small datasets because it involves additional comparisons and recursive heapify operations. The logarithmic component of Heap Sort's complexity causes the relative performance gap to expand as the dataset size grows. The algorithm is appropriate for situations where sorting must handle dynamic or floating-point data (such as rarityScore) due of its predictable behavior and uniform scaling.

Sorting by rarity score in ascending order performed similarly to the descending sort, with times of 3.20 ms, 7.03 ms, and 18.33 ms for small, medium, and large datasets, respectively. The difference between ascending and descending order is negligible because both operations involve the same heap construction and extraction process, only differing in the heap property (min-heap vs. max-heap).

Both ascending and descending Heap Sorts exhibit similar performance, validating the algorithm's flexibility for handling varying sorting requirements.

1.4. Conclusion

This project provided a hands-on opportunity to implement and analyze different sorting algorithms and rarity score calculations applied to an archaeological dataset. I developed a better grasp of the effectiveness and usefulness of both algorithms by employing Heap Sort for sorting by rarity and Counting Sort for age-based sorting, particularly in situations when data size and attribute range vary. The study also made it clearer to me how temporal complexity affects algorithm performance, especially when dealing with big datasets, and how crucial it is to maximize computational efficiency.

Additionally, the exploration of alternative rarity score calculation methods allowed me to develop a more nuanced approach to data analysis, learning how changes in the calculation formula can affect the outcome and performance. Overall, this project enhanced my skills in algorithm design, performance benchmarking, and data manipulation, while also reinforcing key concepts in computational complexity and optimization.