Analysis of Algorithms

BLG 335E

Project 2 Report

Ece Nil Kombak 820220330
kombak22@itu.edu.tr

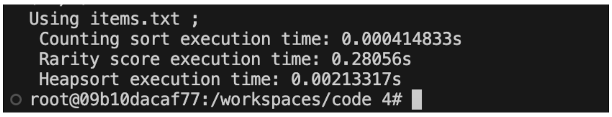
Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 25.11.2024

1. Implementation

1.1. Sort the Collection by Age Using Counting Sort

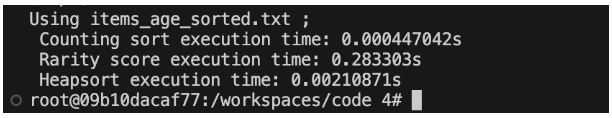
Counting sort is a simple sorting algorithm that works by counting how many times each value appears and using that information to place the values in correct order. It is very fast for sorting numbers in a small range but may not work well for large ranges or non-integer values.

My approach on counting sort is that I first loop through the items to count how many times each value appears and store it in a count array. Then, I update the count array to find the positions for each value. Finally, I loop backward through the items to place them in the output array, adjusting for ascending or descending order.



```
 Using items.txt ;
  Counting sort execution time: 0.000414833s
  Rarity score execution time: 0.28056s
  Heapsort execution time: 0.00213317s
○ root@09b10dacaf77:/workspaces/code 4#
```

Figure 1.1: Counting Sort with `items.txt`

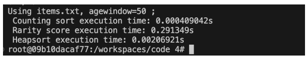


```
 Using items_age_sorted.txt ;
  Counting sort execution time: 0.000447042s
  Rarity score execution time: 0.283303s
  Heapsort execution time: 0.00210871s
○ root@09b10dacaf77:/workspaces/code 4#
```

Figure 1.2: Counting Sort with `items_age_sorted.txt`

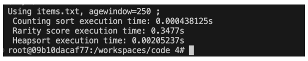
By benchmarking my algorithm on sorted and unsorted data, you can see the results in Figure-1.1 and Figure-1.2 when the data is already sorted, counting sort works faster because the counts are already in the right order so fewer adjustments are needed when placing items in the output array. The time complexity is still $O(n + k)$, but in practice it is quicker since there is less movement of data.

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



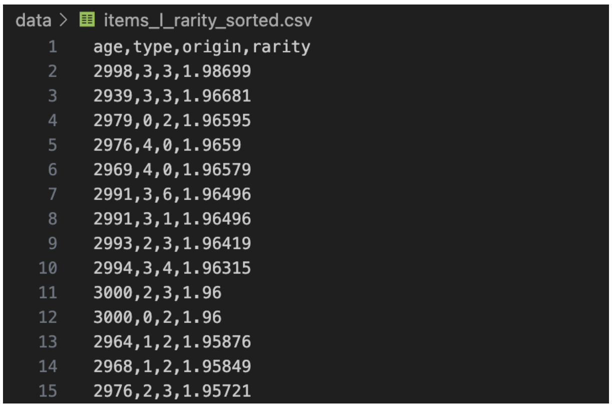
```
Using items.txt, agewindow=50 ;
 Counting sort execution time: 0.000409042s
 Rarity score execution time: 0.291349s
 Heapsort execution time: 0.00206921s
root@09b10dacaf77:/workspaces/code 4#
```

Figure 1.3: Execution time, age window 50



```
Using items.txt, agewindow=250 ;
 Counting sort execution time: 0.000438125s
 Rarity score execution time: 0.3477s
 Heapsort execution time: 0.00205237s
root@09b10dacaf77:/workspaces/code 4#
```

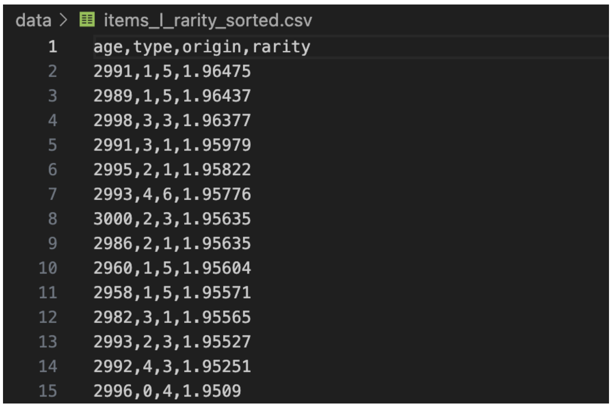
Figure 1.4: Execution time, age window 250



data > items_l_rarity_sorted.csv

```
age,type,origin,rarity
2998,3,3,1.98699
2939,3,3,1.96681
2979,0,2,1.96595
2976,4,0,1.9659
2969,4,0,1.96579
2991,3,6,1.96496
2991,3,1,1.96496
2993,2,3,1.96419
2994,3,4,1.96315
3000,2,3,1.96
3000,0,2,1.96
2964,1,2,1.95876
2968,1,2,1.95849
2976,2,3,1.95721
```

Figure 1.5: Rarity Score, age window 50



data > items_l_rarity_sorted.csv

```
age,type,origin,rarity
2991,1,5,1.96475
2989,1,5,1.96437
2998,3,3,1.96377
2991,3,1,1.95979
2995,2,1,1.95822
2993,4,6,1.95776
3000,2,3,1.95635
2986,2,1,1.95635
2960,1,5,1.95604
2958,1,5,1.95571
2982,3,1,1.95565
2993,2,3,1.95527
2992,4,3,1.95251
2996,0,4,1.9509
```

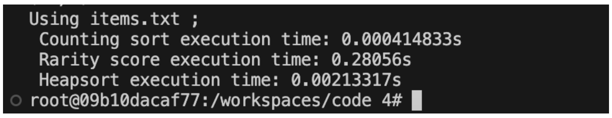
Figure 1.6: Rarity Score, age window 250

In my rarity score calculation, I use an age window to group items and calculate how rare an item is based on the probability of finding similar items within that window. When the age window is small, like 50, the groups are tighter and unique items are easier to spot, so the rarity scores tend to be higher as we can observe from Figure-1.5. When the age window is increased to 250, the groups get bigger therefore more items are included and the rarity scores become lower, Figure-1.6, because the probability of finding similar items increases. Execution time is not significantly affected by the age window size.

1.3. Sort by Rarity Using Heap Sort

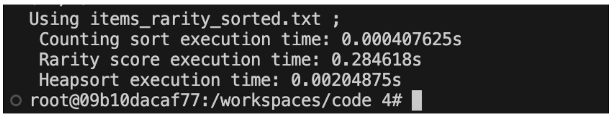
Heap sort is a sorting algorithm that organizes data using a special tree structure, a heap. It repeatedly removes the largest or smallest value from the heap and adds it to the sorted list. It is efficient and good for handling large datasets.

My approach to heap sort is that I first build a max-heap by calling the heapify function on all non-leaf nodes. Then, I repeatedly swap the root with the last element, shrink the heap, and re-heapify. The time complexity is $O(nlogn)$ because building the heap takes $O(n)$, and each re-heapify operation for n elements takes $O(logn)$.



```
Using items.txt ;
 Counting sort execution time: 0.000414833s
 Rarity score execution time: 0.28056s
 Heapsort execution time: 0.00213317s
root@09b10dacaf77:/workspaces/code 4#
```

Figure 1.7: Heap Sort with `items.txt`



```
Using items_rarity_sorted.txt ;
 Counting sort execution time: 0.000407625s
 Rarity score execution time: 0.284618s
 Heapsort execution time: 0.00204875s
root@09b10dacaf77:/workspaces/code 4#
```

Figure 1.8: Heap Sort with `items_rarity_sorted.txt`

By benchmarking my algorithm on sorted and unsorted data, as you can see from Figure-1.7 and Figure-1.8, I found that heapsort performs similarly in both cases, with only a tiny slowdown on unsorted list. This is because the algorithm's time complexity remains $O(nlogn)$ for both heap construction and sorting, regardless of the input order.

1.4. Analyzing Counting Sort, Rarity Score and Heap Sort with Small and Large data

```
Using items_l.csv ;
 Counting sort execution time: 0.00142471s
 Rarity score execution time: 6.27658s
 Heapsort execution time: 0.0130489s
root@09b10dacaf77:/workspaces/code 4#
```

Figure 1.9: Large Dataset

```
Using items_s.csv ;
 Counting sort execution time: 0.000474041s
 Rarity score execution time: 0.293153s
 Heapsort execution time: 0.00265033s
root@09b10dacaf77:/workspaces/code 4#
```

Figure 1.10: Small Daaset

Counting sort performs well on both small and large datasets with only slight change in execution time. Its time complexity is $O(n + k)$, making it efficient regardless of dataset size, as it does not rely on comparisons but scales with the range of values.

Rarity Score algorithm's execution time for sorting by rarity score increases significantly, from 0.293153s for the small dataset to 6.27658s for the large one. This shows that the algorithm used here might be less efficient, worse than $O(nlogn)$time complexity, which causes slower performance as the dataset size grows.

Heap sort shows consistent and efficient performance with either of the datasets regardless of its size. With complexity of $O(nlogn)$, it handles large datasets well, and the slight increase in time reflects its logarithmic scaling.