Analysis of Algorithms

BLG 335E

Project 2 Report

Alper Tutum
tutum21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 22/11/2024

1. Implementation

1.1. Sort the Collection by Age Using Counting Sort

First of all, a counting sort algorithm is implemented for sorting the items list by age, either in ascending or descending order depending on a boolean input. Counting sort is a stable, non-comparison based sorting algorithm with a time complexity of $\Theta(n + k)$ where n is the size of the array to be sorted and k is the range of values in the array. In the given scenario it makes sense to use counting sort for sorting by age since k in this case is not a large number and can be found fairly quickly. Assuming that k isn't known prior to the running of counting sort, it is required to find the greatest value of age in the given dataset. This process takes $\Theta(n)$ as one loop through the array is needed.

After k is obtained, an auxiliary array is needed for counting how many of each age value is found within the original item array. This process again has a time complexity of $\Theta(n)$. The next step is to add the value in the previous index to the current one starting with index 1. Since this step requires going through the auxiliary array with size k in a loop, it has a time complexity of $\Theta(k)$. The process results in an auxiliary array which holds indexes for the next step.

The final step is to create an output array and fill it with the proper values of the original array by using the index values in the auxiliary array. This is done by iterating through n items of the input array starting from the end and placing the items in the output array in the corresponding index which is found in the auxiliary array. The index should then be decremented by 1 in order for the next iterations to work properly. This whole process has $\Theta(n)$ time complexity.

The counting sort algorithm was used on three different datasets with different sizes: 9530 items (s), 18904 items (m), 46675 items (l). The k value for each dataset also varied; 3000 for s, 5000 for m, 10000 for l. The linear behavior of counting sort's time complexity was observed in the running times for each dataset:

- s: 423 μs
- m: 685 μs
- l: 1.45 ms

It can be concluded that counting sort is efficient for datasets with k value small relative to n, and works in linear time.

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

The function to calculate rarity scores of each item takes an array of items and an age window integer as inputs. First the upper and lower bounds for each age value are

defined by adding and subtracting the current item's age value and the age window. Inside the inner loop every item in the array is checked whether they are in the current item's range, if they are $countTotal$ is incremented by 1. After satisfying this condition, the current item in the outer loop and the inner loop are compared. If they have both the same origin and type, $countSimilar$ is incremented.

The actual calculation of the rarity score takes place after the inner loop is finished. The rarity score is calculated with the given formula:

$$R = (1 - P)(1 + age/age_{max})$$

If $countTotal$ is greater than 0 probability (P) is equal to $countSimilar/countTotal$, otherwise it is 0. This calculation is done for every element of the input array and the rarity values are assigned to them.

Since there is one loop which iterates n times nested inside another loop that iterates n times, the time complexity of the $calculateRarityScores()$ function can be given as $\Theta(n^2)$. The running time results of the function for each dataset is as follows:

- s: 0.273 s
- m: 0.989 s
- l: 5.87 s

These results show that the efficiency of this algorithm drops drastically as the size of the input data increases. This is expected due to the $\Theta(n^2)$ nature of the function's time complexity.

Alternative methods for calculating rarity score can be taken into consideration. For example if the age window is kept as a constant (eg. 30) rather than passed as variable argument, the rarity scores would provide more consistent output. Likewise, a method to divide the items to age periods and comparing the rarity within each period could work out to achieve more precise scores. If the rarity score is desired to be less sensitive against changes in small periods, the age window could be defined as a constant with a larger value such as 100 or 200.

1.3. Sort by Rarity Using Heap Sort

Heap sort is an in-place sorting algorithm (unlike counting sort) which has $O(n * logn)$ time complexity. The in-place property of this algorithm makes it efficient for sorting large datasets while also maintaining a reasonable running time.

In the given scenario heap sort takes in an items array which is sorted by age and its rarity scores calculated, then returns the same array but now ordered by rarity in any order based on the $descending$ parameter. The first step of heap sort is to build the heap. This requires a heapify subroutine. Heapify works by determining the left and right index points of a given pivot in the array then finding which item on those indexes has the

greatest value (rarity for our case). If the largest value isn't at the pivot, heapify is called recursively, this time the index with the largest value passed as the pivot. The mentioned process is specifically called max-heapify as it functions to heapify for a max-heap, if largest is replaced by smallest a min-heapify function can be obtained. With the heapify function in hand, a heap can be built by looping from the middle index of the array down to 0 and calling heapify.

After the heap has been built, heap sort can be concluded by looping from the last index of the array down to 1, exchanging the root item (index 0) with the item in the current index then calling heapify to keep the heap property. Given that building the heap has a tight upper bound of $O(n)$ and heapify $O(logn)$, it can be seen that since heapify is being called n times in the final loop, the time complexity of the entire heap sort procedure is $O(n * logn)$.

The heap sort algorithm was performed on s, m and l which are also mentioned in the previous parts. The results are:

- s: 1.97 ms
- m: 4.19 ms
- l: 11.7 ms

The results mostly match with the expected running time scaling behavior of heap sort's $O(n*logn)$ time complexity. Also it can be seen that counting sort in the given conditions is much faster than heap sort.