**CS 412**

**ALGORITHMS: DESIGN & ANALYSIS**

**Final Project Report**

**Section: L2**

**Instructor: Dr. Shahid Hussain**

**Group Members:**

**Muhammad Sabihul Hasan (04387)**

**Hasan Naseem (05102)**

**Saad Ahmed Khan (04293)**

1. **Introduction/ Problem Statement**

Sorting is an integral part of computer science and is almost used in every field. A sorting algorithm rearranges a list of elements into some order, usually a numerical or lexicographical order. The area of sorting has attracted a great deal of research mainly due to the complexity of solving it efficiently. Many algorithms have been created that serve the purpose of sorting.

The sorting algorithms we have covered in our course can sort a structure of strings or integers after comparing each value with another value in the data structure followed by rearranging of those values. However, we can also sort or arrange items without comparing with them with each other.

There are some sorting algorithms that perform sorting without comparing the elements rather by making a certain assumption about the data they are going to sort. The process is known as non-comparison sorting and algorithms are known as the non-comparison-based sorting algorithms.

More formally sorting algorithms have been divided into two categories based on their functionalities; comparison based and non-comparison-based sorting and as their names suggest the most basic difference between the two is that the in the former, elements of a list are compared to achieve a sorted sequence while in the latter no comparison is done.

Our report focuses on the latter; in particular three algorithms that follow the non-comparison-based sorting approach. These three algorithms are **Count sort**, **Radix Sort** and **Bucket Sort.**

1. **Non-Comparison Based Sorting**

Non-Comparison based sorting algorithms follow a unique approach in sorting a given sequence. They make special assumptions about the input sequence. The assumptions vary across the different algorithms. However, they perform better in terms of their time complexities as compared to comparison-based algorithms.

Looking at the performance of comparison-based sorting algorithm, we will realize that Bubble sort, Selection Sort, and Insertion sort takes around *O(n2)* time to sort *n* items. While heap sort, quick sort, and merge sort takes *O(nlogn)* time on their best case and around *O(n2)* on their worst case.

Hence, we need a non-comparison-based sorting algorithm which allows you to sort elements in linear time i.e., *O(n).*

1. **Counting Sort**

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing), then performing some arithmetic to calculate the position of each object in the output sequence.

The special conditions required for counting sort are:

1. The values to be sorted are integers in some range min to max, characterized by some range; *k* = (max – min) + 1
2. *N* >= *k* (*n* is the number of values to be sorted)

The following steps are carried out by the counting sort algorithm:

Step 1:

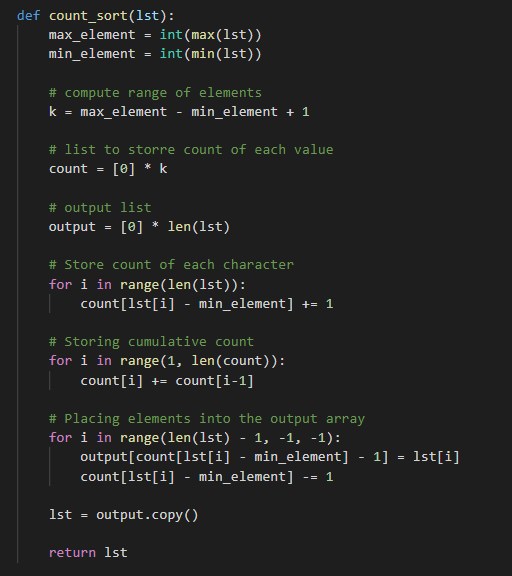
Store the count of each unique object in a set, in another set.

Step 2:

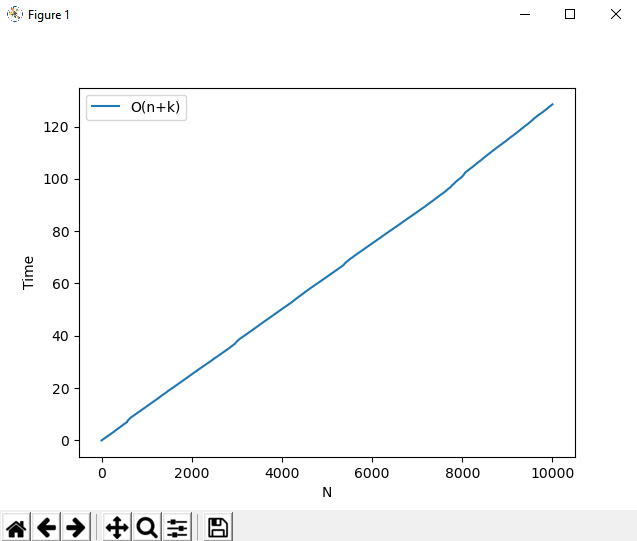
Modify the set such that each element at each index stores the sum of values at previous index.

Step 3:

Output each object from the set followed by decreasing its count by 1.

The following snippet shows our python implementation of the counting sort algorithm:

**Empirical Analysis**

****

**Figure 1: Time vs N for Counting Sort**

In order to visualize the running time complexity of the algorithm, the algorithm was run ***N*** times and the total computational time required each time is recorded for every iteration.

For every iteration:

* A list of size equal to the value of the iteration is initialized with random integer values.
* The randomly initialized list is sorted using the algorithm, and the total running time is recorded.

A graph is plotted for each list of size (1 to ***N***) along with the total time required to sort those lists.

For higher values of ***N***, the graph clearly shows a linear function which depicts that the algorithm correctly sorts the elements in linear time.

The overall time complexity of the algorithm is **O(*N + k*)**

Since, according to the algorithm the initial list is traversed in **O(*N*)** time, the count storing list is traversed in **O(*k*)**, and the resulting sorted list is also computed in **O(*N*)** time.

The values of ***N*** and ***k*** have been clarified in the input constraints of the algorithm.

1. **Radix Sort**

Radix sort avoids comparison by creating and distributing elements according to their radix. For elements with more than one significant digit, this process is repeated for each digit, while preserving the ordering of the prior step, until all digits have been considered.

Radix sort can be applied to data that can be sorted lexicographically. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.

Radix sort uses counting sort as a sub-routine to sort.

The input sequence has to satisfy three conditions in this case:

1. Each individual value to be sorted has length ***d***
2. In these subsequences, each item is itself which falls within a range ***k*** i.e., min to max (the data type of this item could vary from being a string; from a - z to being an integer; from 0 - 9).
3. ***N*** >= ***k*** + 1

The following steps are carried out by the radix sort algorithm:

Step 1:

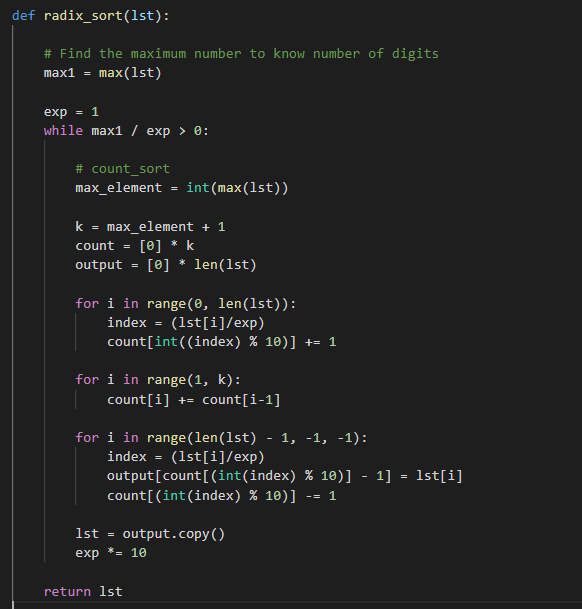
Find the largest element in the set Let **d** be the number of digits of the largest element calculated because we have to go through all the significant places of all elements.

Step 2:

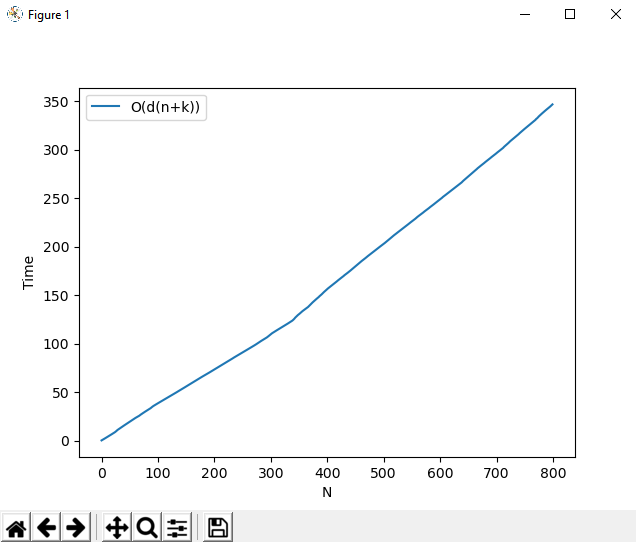
Now, go through each significant place one by one.

Step 3:

Use any stable sorting technique to sort the digits at each significant place. We have used counting sort for this

The following snippet shows our python implementation of the radix sort algorithm:

**Empirical Analysis**



**Figure 2: Time vs N for Radix Sort**

In order to visualize the running time complexity of the algorithm, the algorithm was run ***N*** times and the total computational time required each time is recorded for every iteration.

For every iteration:

* A list of size equal to the value of the iteration is initialized with random integer values.
* The randomly initialized list is sorted using the algorithm, and the total running time is recorded.

A graph is plotted for each list of size (1 to ***N***) along with the total time required to sort those lists.

For higher values of ***N***, the graph clearly shows a linear function which depicts that the algorithm correctly sorts the elements in linear time.

For the radix sort that uses counting sort as an intermediate stable sort, the time complexity is ***O(d(N + k))*** for the worst case and ***O(N + k)*** as the best case.

The values of ***N***, ***k*** and ***d*** have been clarified in the input constraints of the algorithm.

Here, ***d*** will compute to be the total number of cycles in the worst case and ***O(N + k)*** is the time complexity of counting sort.

Each position in the sequence is worked upon once which means there is an outer loop that executes so the outermost loop must take at least ***O(k)*** time. Processing each subsequence has a time complexity of ***O(N)*** has there are ***N*** elements in each subsequence

Hence overall time complexity is ***O(N + k).***

1. **Bucket Sort**

Bucket Sort is a sorting technique that sorts the elements by first dividing the elements into several groups called buckets. The elements inside each bucket are sorted using any of the suitable sorting algorithms or recursively calling the same algorithm.

Several buckets are created. Each bucket is filled with a specific range of elements.

The elements inside the bucket are sorted using any other algorithm. Finally, the elements of the bucket are gathered to get the sorted array.

This algorithm divides the input sequence into several groups called buckets such that;

1. the values to be sorted are evenly distributed in some range min to max
2. It is possible to divide the range into ***k*** equal parts, each of size ***N/k***

The following steps are carried out by the bucket sort algorithm:

Step1:

Set up an array of initially empty "buckets".

Step 2:

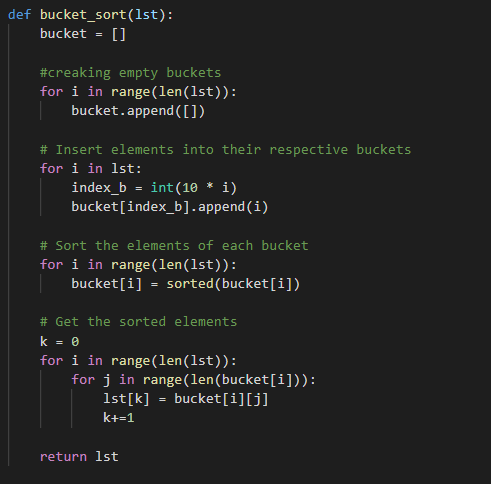
Go over the original array, putting each object in its bucket.

Step 3:

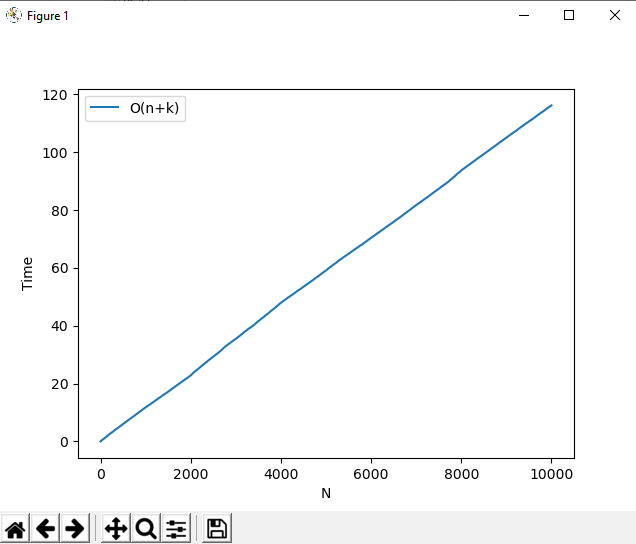
Sort each non-empty bucket.

Step 4:

Visit the buckets in order and put all elements back into the original array.

The following snippet shows our python implementation of the bucket sort algorithm:

**Empirical Analysis**

****

**Figure 3: Time vs N for Bucket Sort**

In order to visualize the running time complexity of the algorithm, the algorithm was run ***N*** times and the total computational time required each time is recorded for every iteration.

For every iteration:

* A list of size equal to the value of the iteration is initialized with random integer values.
* The randomly initialized list is sorted using the algorithm, and the total running time is recorded.

A graph is plotted for each list of size (1 to ***N***) along with the total time required to sort those lists.

For higher values of ***N***, the graph clearly shows a linear function which depicts that the algorithm correctly sorts the elements in linear time.

Well, there are ***k*** different buckets, so the outermost loop must take at least ***O(k)*** time, because we have to access each bucket to check whether it is empty or full. The inner loop will execute a total of ***O(N)*** times overall, because there are a total of ***N*** elements distributed across the ***k*** buckets. From this, we get the ***O(N + k)*** as the total runtime.

The reason that this is important is that it means that if you try doing a bucket sort with a huge number of buckets then the runtime will be dominated by the time required to scan over all the buckets looking for the buckets that were actually used, even if most of them are empty.

1. **Conclusion**

The above algorithms perform better in most cases when it comes to time complexities as compared to comparison-based sorting algorithms. Count sort's time complexity for the best, average and worst case is O(n + k). This is because the algorithm performs n + k iterations regardless of the input size. Radix sort has a time complexity of O(d(n + k)) using count sort as the intermediate sort. When it comes to bucket sort, the worst-case time complexity is no better than that of most comparison-based algorithms; O(n^2). But its best-case time complexity matches that of Count sort and Radix sort algorithms. Hence it can be seen that non comparison-based algorithms perform better than comparison-based algorithms however certain they need to meet certain requirements.