Inha University in Tashkent, IUT

05/10/2020

Research project proposal

Multi-layer Radix sort

Student Name:	Student ID:	Section:
Enver Menadjiev	U1710019	003
Siyavushkhon Kholmatov	U1710263	003
Artyom Stitsyuk	U1710267	003

Computer Science and Engineering Department

Inha University in Tashkent

Abstract

In the contemporary world computational speed is the essential approach. Any computing device should be as fast as possible. However, how can we reach the high speed of the computing devices? It is achieved thanks to the rational algorithms. As a result, the creation of an ideal, or as close to it as possible algorithm is essential.

Algorithm is named sorting algorithm if it works by putting all elements of the list in a specific given order. [1] It has started to come up in the late 19th century. Initially, these methods were created only for numbers. Subsequently, sorting algorithms were adapted for other data types.

When it comes to Strings, all comparative algorithms give large time complexity. [2] Therefore, a non-comparative algorithm could be a nice approach to consider for sorting strings. [3] Radix sort is a non-comparative sorting algorithm. A distinctive feature of Radix sort is that it avoids comparison using creation and distribution of elements into different buckets according to their radix. This sorting method allows to sort large chunks of data using fair complexity. According to radix sort, a unique bucket for every letter is created. [3] Thus, if the language is English, 27 buckets are created only for letters and time complexity equals to O(27n). However, what if the task is to sort a sentence with words(strings) which contains symbols like dots, commas or spaces? Every symbol would require a new bucket and the complexity would increase drastically. Therefore, we decided to improve this algorithm with a new technique to reduce both time and space complexity.

The idea is to sort strings interpreting each character as ASCII. This becomes a two-layer radix sorting, which offers an opportunity to apply sorting several layers, recurrently with less time complexity. Since all letters and symbols are in the range of three-digit numbers by ASCII the time complexity will be constant. Consequently, with 10 buckets needed for every digit the time complexity equals to 30 * O(n) regardless of the language, the length of the sentence and appeared symbols.

Using this Radix sort as a counting method, can solve this problem in such a way that large chunks of any data (including strings) are sorted by the most rational algorithm.

The following algorithm is implemented in java language and is going to be explained below.

1. Introduction

It is needless to remind that in information technology science, sorting algorithms are considered to be one of the essential and most used techniques that puts all elements of the list in a specific needed order. [1] There is various methods and types are available to apply them in different kind of situations. As an example, we can consider Comparison and Non-Comparison types of sorting. Particularly, if we take a look at Non-Comparison sorting, we can release that there are different types of techniques available to use, such as Counting sort, Bucket Sort, Radix Sort and compare them. [4]

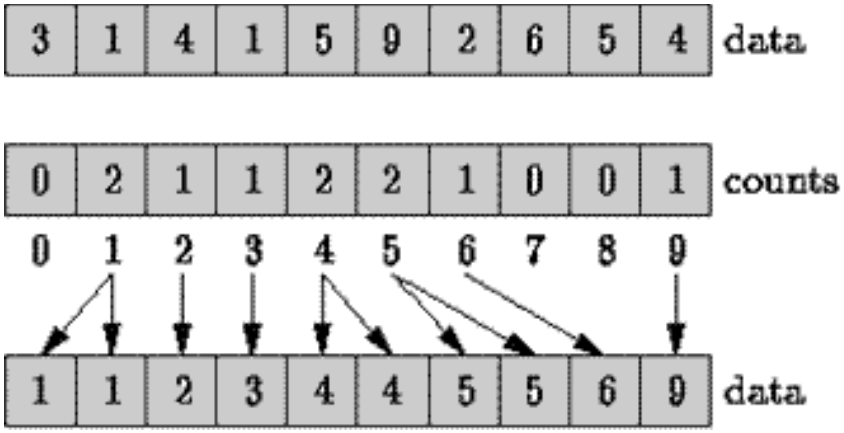

Figure 1. Radix sort

Operating principle of Radix sort is related to Bucket sort, which makes N iteration called passes for every digit of the maximum number from right to left direction using 10 buckets, from 0 to 9, as a result of digit range. So, step by step every piece of data is sorted into appropriate bucket and on the output there will be sorted list of data. If numbers in the list has more than one digit, this algorithm make an iteration for the first digit, and then repeat it for every next. [5]

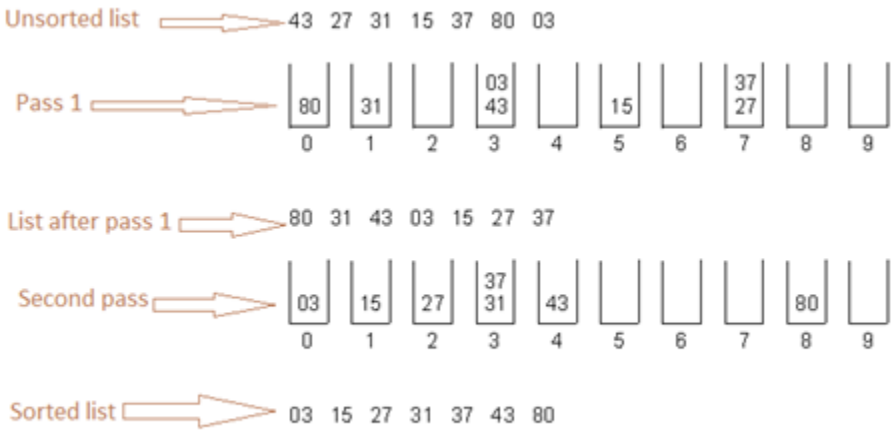

Figure 2. Radix sort, case with two digit numbers

```
Step 1: Find the largest number in Array as LARGE

Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE

Step 3: SET PASS = 0

Step 4: Repeat Step 5 while PASS <= NOP-1

Step 5:         SET I= and INITIALIZE buckets

Step 6:         Repeat Steps 7 to 9 while I<N-1

Step 7:               SET DIGIT = digit at PASS'th place in A[I]

Step 8:               Add A[I] to the bucket numbered DIGIT

Step 9:               INCEREMENT bucket count for bucket numbered DIGIT

                [END OF LOOP]

Step 10:        Collect the numbers in the bucket

        [END OF LOOP]

Step 11: END
```

Figure 3. Radix sort algorithm

Coming to main part of our topic, Radix sort allows to make sorting on array of characters, particularly on String data types. In Radix there are at least 26 (+1 for space) buckets that corresponds to the number of letters in the English language. Therefore, More than 41 buckets will be required to sort a string containing both alphabet letters and numbers with special characters, while in contrast there are only 10 buckets are used for only numerical sorting. As we can see, this will lead to more Space complexity compared to simple numbers. Even if in this algorithm there is no any limitations or gap, the main task and challenge for us is to reduce the number of used buckets to 10 for string sorting using characters with alphanumeric and keep algorithm in the fast way as it is in contrast to Comparison algorithms. How to achieve this?

2. Related works and discussions

This section will review the behavior of different algorithms with strings sorting techniques..

2.1. Quicksort and Merge sort

For a long period of time, quicksort was considered to be one of the fastest sorting algorithms for both single and multi-layer arrays. [6] Quicksort is known as one of the "Divide and Conquer" algorithms that takes a pivot element and makes partitions of the array of data around that selected pivot member.

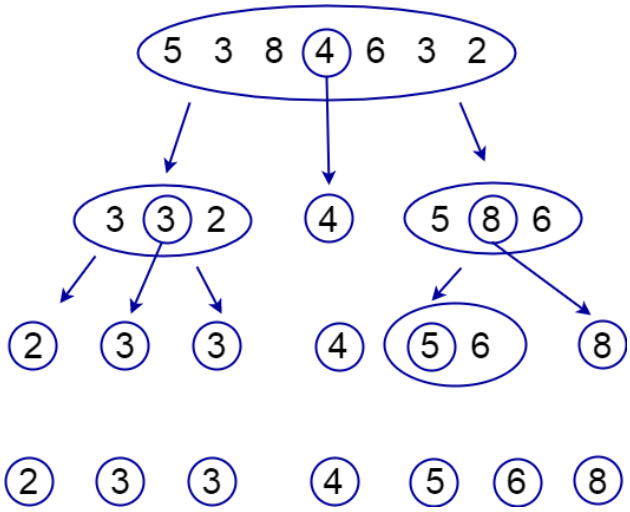

Figure 4. Quicksort

There is another algorithm that works according to the principle "Divide and Conquer" is merge sort. [7] The main goal of the algorithm is to divide a list into sublists until each of them will have only one element. Thus, in the simplest form, merge the sublist in the sorted order.

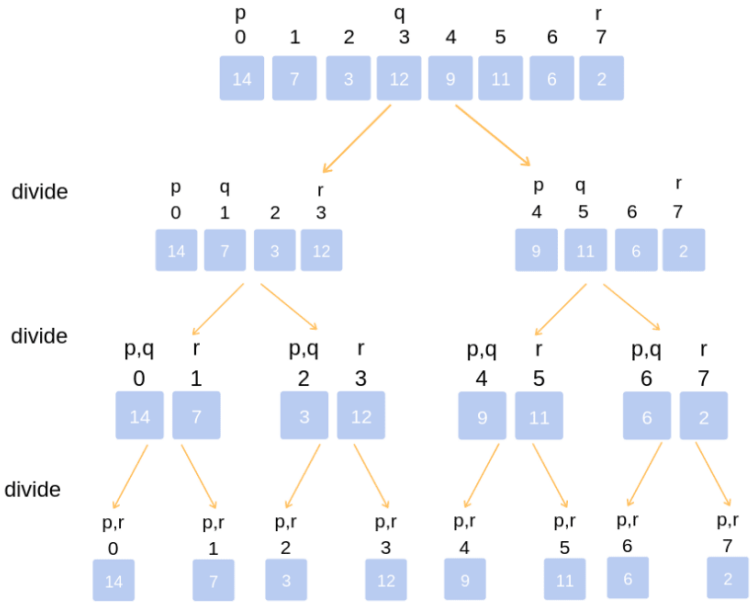

Figure 5. Merge sort

Disadvantages:

- Partition part of algorithm has a large memory consumption
- With increasing recursive operations, program slows down
- Worst case is O(N^2)

2.2. Radix sort and Bucket sort

Once computer devices have started to be developed in a massive production, their performance became the most important factor. To ameliorate their computational abilities, many various algorithms have been proposed. Since that time, a new type of sorting technique has been developed, based on buckets without any comparisons with key elements from arrays. Radix Sort and Bucket Sort algorithms are an example of this. Some researches have been conducted to identify and compare a performance of both Radix Sort and Quick Sort. The first demonstrated faster results with future wide applicability in many other aspects [3] [5]. Also, both Radix Sort and Bucket Sort were compared in different circumstances to check their performance. While the latter was faster with a small number of chunk data, it consumed much more memory for creating additional buckets to sort an array, explicitly increasing its space complexity [4].

Disadvantages:

- Increasing buckets number
- Depends on size of input
- Difficult solution for negative valuesf

These encouraged us to find out the golden mean for Radix Sort. Particularly, by making it faster and at the same time with less memory consumption. Since Radix Sort algorithm has already been proven as one of the fastest sorting algorithms with numbers and non-comparison technique, a main goal was to look at the sorting of a text data (sentences, strings, words). As a result of a variety in various language letters, the number of buckets has significantly increased. Some researchers studied the fast algorithms for sorting strings by combining both Radix Sort and Quicksort to get higher performance [2]. To overcome all of these constraints, we came up to the idea of Multi-Layer Radix Sort. It can sort any strings, words or even sentences recursively by using only 10 buckets which are essential for digits. In this way, we could eliminate any additional buckets, since all letters have their own correspondent in ASCII code and can be represented in digits only. Consequently, with these type of data Radix Sort performs much faster relatively to any other comparison-based algorithms, as it is already known as the fastest GPU sort technique [8] .

3. Proposed algorithm

3.1. Pseudocode

Program 1. Pseudocode of the proposed algorithm

```
MultiLayerRadixSort(string, delimiter){
	input = array = splitString(string, delimiter);
	size = getMaxSize(array);
	for(key in array){
		array[k] = normalizeStrings(array[k]); // Make strings equal in length
		array[k] = mapCharsToASCII(array[k]);  // Make an Array of ASCII codes
	}
	indices = extractIndices(array);
	for(i=0; i<size; i++){
		iterationLetters = extractLetter(array, i);
		indices = radixSort(iterationLetters, indices);
	}
	array = [];                                    // Flush temp array
```

```
	for(i=0; i<sizeOf(indices); i++){
		array[i] = input[indices[i]];		// Form a sorted array
	}

	return formString(array, delimiter);
}
radixSort(lettersArray, indices){
	exp = floor(log10(max($array))) + 1;
	for (i = 0; i < exp; i++) {
		buckets = [];
		for (k in indices){
			bucket = floor(lettersArray[indices[k]] / pow(10, i)) % 10;
			enqueue(buckets[bucket], indices[k]);
		}
		indices = [];

		for(k = 0; k < 10; k++){
			if(buckets[k] != NULL){
				enqueue(indices, buckets[k]);
			}
		}
	}
}
```

3.2. Java implementation

In order to achieve our goal for reducing the number of buckets in string sorting with Radix Sort, we have made different researchers and tests and finally find out the solution. The first version of solution, we have implemented using Java Programming language, which is as follows:

Program 2. Java implementation of the proposed algorithm

```java
public class Main {

    /**
     * Find the maximum number in given array
     * @param arr given array to search from
     * @return max number from array
     */
    private static int findMax(int[] arr) {
        int max = arr[0];
        for (int i : arr)
            if (i > max)
                max = i;
        return max;
    }

    /**
     * Getting the digit from number at any position in right to left ordering
```

```
     *
     * @param num      the given number
     * @param digitPos position of digit required to be cut
     * @return required digit at [digitPos] position
     */
    private static int getDigitAt(int num, int digitPos) {
        return (int) ((num / Math.pow(10, digitPos)) % 10);
    }

    /**
     * Sorting each pass of RadixSort
     *
     * @param arr       The array that passed to method
     * @param passLevel number of pass level in terms of 10^i from right to left,
     *                  (ex. 127, to get 2 -> passLevel=1, because
(arr[i]/10^passLevel)%10
     */
    private static int[] sortByBucket(int[] arr, int passLevel) {
        int[] finalArr = new int[arr.length], count = new int[10];
        Arrays.fill(count, 0);
        for (int num : arr) {
            count[getDigitAt(num, passLevel)]++;
        }
        for (int i = 1; i < count.length; i++)
            count[i] += count[i - 1];

        for (int i = arr.length - 1; i >= 0; i--) {
            int finalIndex = --count[getDigitAt(arr[i], passLevel)];
            finalArr[finalIndex] = arr[i];
        }
        return finalArr;
    }

    /**
     * Radix sort algorithm
     *
     * @param wordList given array to be sorted
     */
    private static int[] radixSort(List<List<Integer>> wordList) {
        int wordsCount = wordList.size();
        int[] wordsSize = new int[wordsCount];
        for (int i = 0; i < wordsCount; i++)
            wordsSize[i] = wordList.get(i).size();
        int maxLengthWord = findMax(wordsSize);
        int[][] filteredOrder = new int[2][wordsCount];
        for (int i = 0; i < wordList.size(); i++)
            filteredOrder[0][i] = i;
        for (int i = maxLengthWord - 1; i >= 0; i--) {
```

```
            int[] mask = new int[wordsCount], filteredMask;
            int k =0;
            for (int j : filteredOrder[0]) {
                int num = 0;
                if (wordList.get(j).size() > i)
                    num = wordList.get(j).get(i);
                mask[k++] = num;
            }
            int maxLetter = findMax(mask);
            int maxLetterLength = (int)(Math.log10(maxLetter) + 1);
            filteredMask = mask;
            filteredOrder[1] = mask;
            for (int j = 0; j < maxLetterLength; j++)
                filteredMask = sortByBucket(filteredMask, j);
            filteredOrder[0] = sortedIndexes(filteredOrder,filteredMask);
        }
        return filteredOrder[0];
    }

    private static int[] sortedIndexes(int[][] initialArray, int[] sortedArray) {
        int[] sortedIndexes = new int[initialArray[0].length];
        for (int i = 0; i < sortedArray.length; i++) {
            for (int j = 0; j < initialArray[1].length; j++) {
                if (sortedArray[i] == initialArray[1][j] && initialArray[1][j] != -1)
{
                    sortedIndexes[i] = initialArray[0][j];
                    initialArray[1][j] = -1;
                    break;
                }
            }
        }
        return sortedIndexes;
    }

    /**
     * Print the given array
     *
     * @param arr input array
     */
    private static void printArray(List<List<Integer>> arr) {
        for (List<Integer> word : arr) {
            for (Integer letter : word) {
                System.out.print((char)((int)letter)); //printing each letter of word
            }
            System.out.print(" "); //space between words
        }
    }
```

```
    public static void main(String[] args) {
        String str = "Student Learning Assignment, Multi-layer Radix Sort".trim();
        int wordsCount = (str.length() - str.replace(" ", "").length()) + 1;
        List<List<Integer>> wordList = new ArrayList<>(wordsCount);
        str += " "; //make sure at the end we have space to cut the last word
        StringBuilder tmpStr = new StringBuilder(str);
        while (tmpStr.toString().contains(" ")) {
            int firstWordAt = tmpStr.indexOf(" ");
            String firstWord = tmpStr.substring(0, firstWordAt);
            List<Integer> word = new ArrayList<>();
            for (int j = 0; j < firstWord.length(); j++)
                word.add((int) firstWord.charAt(j));
            wordList.add(word);
            tmpStr.delete(0, firstWordAt + 1);
        }
        System.out.println("Before Sort");
        printArray(wordList);

        int[] filteredOrder = radixSort(wordList);
        System.out.println("\nAfter Sort");
        for (int i : filteredOrder) {
            for (Integer letter : wordList.get(i))
                System.out.print((char)((int)letter)); //printing each letter of word
            System.out.print(" "); //space between words
        }
    }
}
```

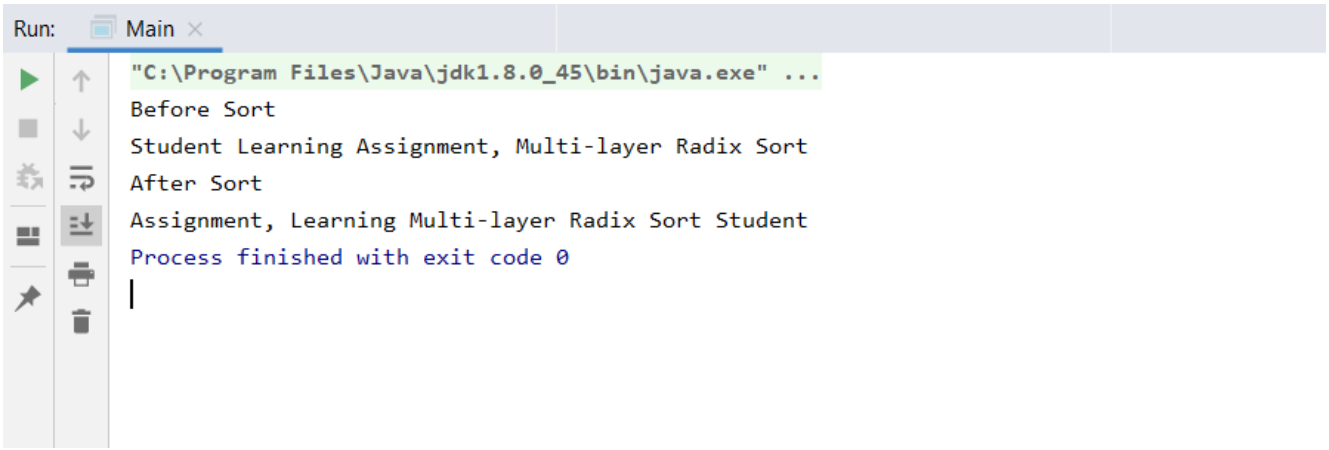

Figure 6. Example of sorting

Explanation:

Structure of algorithm divided into five local methods for sorting itself and one for printing array: findMax, getDigitAt, sortByBucket, radixSort, sortedIndexes and printArray respectively. The main concept of algorithm is, to make sorting of string characters in terms of ASCII code and process it as simple number using 10 buckets but in Multi-Layer manner.

In main function we assume that we are given the string that is required to be sorted alphabetically in ascending order. Firstly, we need to convert all the given string to ASCII code and store as simple Integer numbers. For this, we will use List<List<Integer>> wordlist which initially has wordCount empty elements. The concept of this structure is to store List of words which contains itself the list of letters respectively, so we can sort every level as matrix entities. After that, using StringBuilder and its pre-build methods, we cut the words, convert every letter to ASCII code as integer number and add them to wordlist in format of list of letters inside the list of words. Before applying sorting algorithm, we print the current array as how it is given and after that call *radixSort*(wordList) which will return the array of sorted and ordered indexes of the given string to print it out in correct way. Now, we look inside the main and core method – *radixSort,* there we find the number of words and the maximum length of word from the given string. This maximum length is required because we will make sorting from the right and take the last digits (letter) in normalized way from every row of words as a sub array of numbers and pass them to sortByBucket algorithm which will return the sorted list of the given last digits.

Example: if our wordlist contains list of ASCII words (row) where it has the list of letters (columns), we pass to sortByBucket the sub array of [82,90,69] (last digit of every row)
[65,79,82] // AOR
[65,77,90] // AMZ
[65,77,69] // AME

The result of sortByBucket is stored in filteredOrder two-dimensional array. filteredOrder[0] contains the indexes of passed digits to sortByBucket and filteredOrder[1] contains the digits themselves.

From above example: if [82,90,69] is passed, they are stored in filteredOrder[1] and their row indexes are stored in filteredOrder[0] as [0,1,2] respectively for every digit. filteredMask contains the sorted subarray from sortByBucket, such that [69,82,90].

When we have got the filteredMask,filteredOrder with its indexes, we pass them to sortedIndexes method in order to sort the filteredOrder[0] (indexes of the digits) according to filteredMask and keep its order.

From above example: if [69,82,90] was returned from sortByBucket to filteredMask and filteredOrder[0] contained [0,1,2], after the sortedIndexes method, our filteredOrder[0] will keep the order in terms of sorted digits, like: [2,1,0].

These steps will be repeatedly performed for all other digits in the given List of ASCII numbers. As a result, at the final stage, our filteredOrder[0] will contain the final order for the given string in alphabetically ascending order which is actually returned from radixSort method to the main function, where final result is printed out.

4. Performance

The algorithm utilizes radix sort approach, which is not a comparison-based sorting algorithm. All comparison based algorithms, such as quicksort [6], mergesort [7], heapsort [9] and others have O(n*log(n)) complexity. However, counting sort algorithm has a linear time complexity O(n+k) where k is the number until which the range is. Therefore, our algorithm is based especially on counting sort algorithm.

The Time Complexity of our algorithm is roughly O(32n), which is still a O(n), where n = number of letters, so let us prove this.

- If the number of words in a sentence equals to 1, it is already sorted that is why is returned. This action is performed to avoid $O(n^2)$ worst-case and get O(1) best-case running time.
- Getting the maximum size of the word in a sentence takes O(n) time to compare all the words.

- Mapping string to an array of ACII codes takes O(n) * O(1) running time, which equals to O(n) time complexity
- Radix sort takes O(n) * 3 * 10, where 3 is the number of iterations of the maximum exponent of 10, what allows to use ASCII codes up to 999 and 10 stands for the number of bucket checks. Therefore, we get O(n) complexity again.
- To conclude, the time complexity of the proposed algorithm is equal to:
  O(n) + O(n) + 30 * O(n) = 32 O(n) = **O(n)**, where n represent letters in a sentence
  The Space complexity of an algorithm is an O(n), where:
- 10 – number of buckets
- O(n) – number of indices (if all words consist of 1 letter)
- O(n) – temporary array for storing ASCII codes (this is an option, since all of the codes may be stored in initial locations and later decomposed to the input)
- To conclude, the time complexity of an algorithm is:
  10 + O(n) + O(n) + = 2 O(n) = **O(n)**, where n represent letters in a sentence

Complexity	**Best-case** (one-word case)	**Worst- average- case**
Time complexity	**O(1)**	**O(n)**
Space complexity	**O(1)**	**O(n)**

Table 1. Complexity

5. Conclusion

Most of data in databases are stored in varchar formats and are used to be linked or even selected, which takes time to be query processed. In order to improve complexity and reduce space of memory we decided to make a sorting of strings with the method of converting chars into ASCII code, which finally reduced number of buckets from 27 to 10. As a result, time complexity of the algorithm was reduced to O(n).

This approach may be used for solving following real-life problems:

- Sorting of real text using any random delimiter. For instance, to sort a text by sentences user can use this algorithm with delimiter "full stop".
- Another example for this may be multidimensional array of products that are grouped by categories. This algorithm offers an opportunity to sort this two level list in O(n) time.
- Finally, since ASCII includes all characters of all languages, this algorithm works regardless of language. There is no need to rebuild whole the algorithm for another language environment

It is our sincere belief that by creating new technological algorithms, we can revolutionize the way many developers operate nowadays. In this report, we have shared our own idea about the amelioration of Radix Sort, particularly with sorting of string data. To prove our conceptual ideas, one of the widespread programming languages as Java was used. We tend to believe that this algorithm might be useful in many other spheres to improve performance and speed for sorting and searching any data. Starting from small SQL databases to a considerable large ORM systems.

References

[1] G. Kocher and N. Agrawal, "Analysis and Review of Sorting Algorithms," *International Journal of Scientific Engineering and Research (IJSER),* vol. 2, no. 3, pp. 81-84, 2014.

[2] J. L. Bentley and R. Sedgewick, "Fast Algorithms for Sorting and Searching Strings," Bell Labs; Princeton University, Princeton, New Jersey, 1997.

[3] P. Mcllroy and K. Bostic, "Engineering Radix Sort," University of California at Berkeley, California, 1992.

[4] P. Horsmalahti, "Comparison of Bucket Sort and RADIX Sort," Tampere University of Technology, Finland, 2012.

[5] J. I. Davis, "A Fast Radix Sort," Wilfrid Laurier University, Ontario, Canada, 1992.

[6] C. A. R. Hoare, "Quicksort," *Computer Journal,* vol. 1, no. 5, pp. 10-15, 1962.

[7] L. Zheng, "Speeding up external mergesort," *IEEE Transactions on Knowledge and Data Engineering,* vol. 8, no. 2, pp. 322-332, 1996.

[8] N. Satish, M. Harris and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," IEEE, Rome, Italy, 2009.

[9] L. Wegner, "The external Heapsort," *IEEE Transactions on Software Engineering,* vol. 15, no. 7, pp. 917-925, 1989.