


The Novel Multi-Layered Approach to Enhance the Sorting Performance of Healthcare Analysis

Ashish Seth, INHA University, South Korea

 <https://orcid.org/0000-0003-1580-897X>

ABSTRACT

Emergence of big data in today's world leads to new challenges for sorting strategies to analyze the data in a better way. For most of the analyzing technique, sorting is considered as an implicit attribute of the technique used. The availability of huge data has changed the way data is analyzed across industries. Healthcare is one of the notable areas where data analytics is making big changes. An efficient analysis has the potential to reduce costs of treatment and improve the quality of life in general. Healthcare industries are collecting massive amounts of data and look for the best strategies to use these numbers. This research proposes a novel non-comparison-based approach to sort a large data that can further be utilized by any big data analytical technique for various analyses.

KEYWORDS

Analysis, Non-Comparison Sorting, Radix-Sort, Space-Complexity, Time-Complexity

INTRODUCTION

For any effective analysis, the efficient sorting technique is required implicitly. In the contemporary world computational speed is the essentially important. Any computing device should be as fast as possible. However, how can we reach the high speed of the computing devices? Due to presence of rational algorithms it is achieved. As a result, the creation of an ideal, or as close to it as possible algorithm is essential, Seth.A. et.al (2021).

Sorting is a fundamental and well-studied problem that has been studied extensively. Sorting algorithm is an algorithm that puts elements of a list in a certain order. 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. There exists lot of sorting algorithms in terms of computational complexity. The performance of any algorithms depends on several factors that must be taken in consideration such as time complexity, stability, memory space, Kharabsheh. K.S. et.al (2013). To compare various sorting strategies, Goodrich M. et. al. (2010) and Sipser M. (1996)

DOI: 10.4018/IJRQEH.289178

This article published as an Open Access article distributed under the terms of the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/4.0/>) which permits unrestricted use, distribution, and production in any medium, provided the author of the original work and original publication source are properly credited.

mentioned several factors that must be taken into consideration, primarily time complexity; the time complexity refers to the amount of time taken by an algorithm to produce result.

When it comes to Strings, all comparative algorithms give large time complexity. Therefore, a non-comparative algorithm could be a nice approach to consider for sorting strings. Radix sort is a non-comparative sorting algorithm which avoids comparison by creating and distributing elements into 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. 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. In our work we have proposed an improved 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.

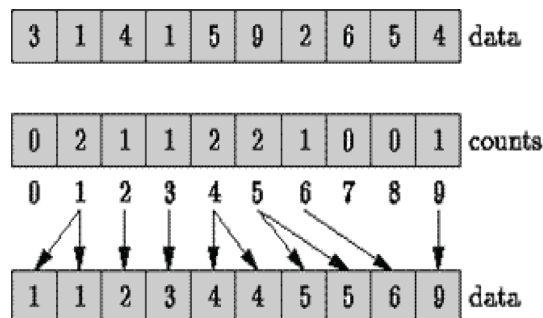
Using this Radix sort as a counting method, can solve this problem in such a way that large chunks of any data (including strings) are sorted by the most rational algorithm. The algorithm is implemented in java and php languages and will be discussed in later sections.

LIMITATIONS OF AN EXISTING APPROACH

With general radix sort approach, the problem is that algorithm must examine every element of every item being sorted. On the other hand, comparison-based sorts techniques skip a fair number of sub-elements (digits, characters, etc.). Non-comparison based sorting algorithms make assumptions about the input. To ensure linear time complexity input elements are expected within a range of constant length whereas in comparison based methods no such assumption about the input is desired. Moreover, non-comparison based methods adds extra memory cost and lacking in generality of the input. Other behavior of comparison-based method is that they need to call a comparator on input elements a whole bunch of times and this makes them inherently slower.

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 elements of a list in certain correct order. 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 etc.

Figure 1. Bucket sort example



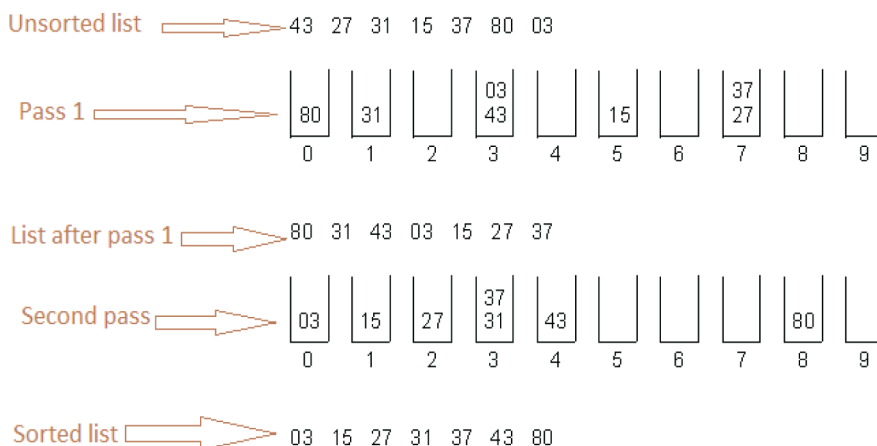
Operating principle of Radix sort is related to Bucket sort, which makes N iteration called *passes* for every digit of the maximum number from right to left direction using 10 buckets, from 0 to 9, as a result of digit range.

ALGORITHM FOR RADIX SORT

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 \leq 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: INCREMENT bucket count for bucket numbered DIGIT
 [END OF LOOP]
 Step 10: Collect the numbers in the bucket
 [END OF LOOP]
 Step 11: END

Radix sort allows to make sorting on array of characters, particularly on String data types. To achieve this goal, Radix will use 27 buckets to sort strings containing only characters, and at least 40 buckets to sort a string containing alphanumeric characters rather than usual 10 buckets to sort only numbers. 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.

Figure 2. Radix sort example



Radix sort example

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]);
            }
        }
    }
}
```

Algorithm Implementation Using Java

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 is implemented using Java programming language, which is as follows:

```
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  
10i from right to left,  
    *(ex. 127, to get 2 -> passLevel=1, because  
    (arr[i]/10passLevel)%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) {

```

```

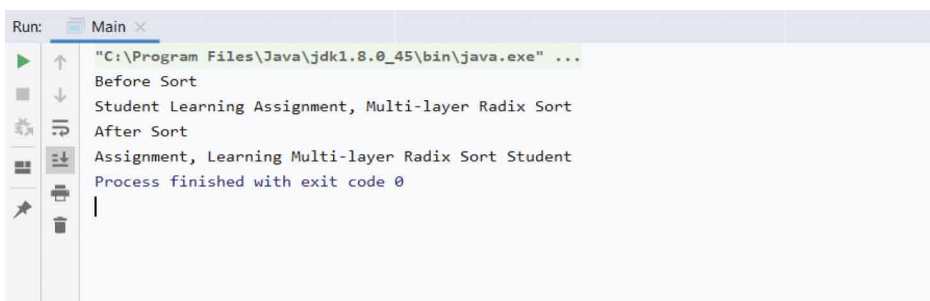
        for (Integer letter: word) {
            System.out.print((char)((int)letter)); //printing each
            letter of word
        }
        System.out.print(" "); //space between words
    }
}

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 3. Sorting of characters string using Radix Sort with 10 buckets



```

Run: Main x
"C:\Program Files\Java\jdk1.8.0_45\bin\java.exe" ...
Before Sort
Student Learning Assignment, Multi-layer Radix Sort
After Sort
Assignment, Learning Multi-layer Radix Sort Student
Process finished with exit code 0

```

Working of Proposed Algorithm

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.

PHP Module of Proposed Algorithm

The proposed algorithm consists of several parts, which are made as modules and may be substituted if needed for the alternatives or removed if the input is already formatted. We have produced several implementations of the algorithm in PHP and Java. PHP was used as a pseudocode to understand the algorithm itself, read and analyze the code easily. Algorithm may be modified to be multi-layer, but

for investigation purposes, two layers are used. The example chosen to examine algorithm is sorting words in a sentence using only ten buckets, which was achieved by interpreting the “string” as an array of “characters”, which were mapped to the ASCII code.

For the sake of simplicity, let us consider the implementation in PHP. According to this, algorithm structure may be divided into several parts:

- Normalizing the input
- Modifying indices
- Sorting via adapted Radix Sort approach

```
function multiLayerRadixSort(string $input): string
{
    /** Input Normalization */
    $input = explode(' ', $input);           // ["INHA",
    "University", "In", "Tashkent"]
    if(count($input) == 1)
        return $input[0];
    $size = getMaxSize($input);             // 10

    /** String to int mapping (ASCII codes) */
    $tempArray = array_map(function ($item) use ($size) {
        return mapStrToInt($item, $size);
    }, $input);
    $indices = array_keys($tempArray);

    /** Sorting an array */
    for ($i = $size - 1; $i >= 0; $i--) {
        $indices = radixSort($tempArray, $i, $indices);
    }
    /** Formatting the output */
    return implode(' ', array_map(function($item) use($input){
        return $input[$item];
    }, $indices));
}
```

Let us consider as an example an input sentence “*INHA University In Tashkent*”. The first step normalizes the input, what stands for the preparation of input “string” for the further manipulations. First and foremost, using built-in **explode** function, which takes as parameters delimiter and the string, we get an array of strings which were divided by the whitespace.

Delimiter can be any character by the choice of the user, what makes an algorithm multipurpose. If the number of words, equals to one, the sentence is automatically sorted, which is why it will be automatically returned. After this step our input looks like: [“*INHA*”, “*University*”, “*In*”, “*Tashkent*”]

Next line invokes **getMaxSize** function, which simply returns maximum “string” size among all the words in a sentence. Since word “*University*” has 10 letter in it, function returns *10*.

```
function getMaxSize(array $input): int
{
    return max(array_map(function ($item) {
```

```
}, $input));
    return strlen((string)$item);
}
```

String to int mapping block makes a certain manipulation on an array, where each word is mapped to the array of integers corresponding to the ASCII code of a character. Function **mapStrToInt** is presented below.

```
function mapStrToInt(string $input, int $size): array
{
    $empty = array_fill(0, $size - strlen($input), 0);
    $input = array_map(function ($char) {
        return ord($char);
    }, str_split($input, 1));

    array_push($input, ...$empty);
    return $input;
}
```

Function parameters are input string and the maximum size of the word in a sentence. Firstly, the function creates an empty array and fills it with zeros. Afterwards, by applying **ord** function to each character it maps every word to an array of corresponding integers. Lastly, two arrays are combined with each other, such that an array with zeros is in after the reasonable text. This technique allows us to have all words of the same length without losing the weight of the characters. The **tempArray** after this step may be depicted like:

```
[
0 => [73, 78, 72, 65, 0, 0, 0, 0, 0, 0], // INHA
1 => [85, 110, 105, 118, 101, 114, 115, 105, 116, 121], // University
2 => [73, 110, 0, 0, 0, 0, 0, 0, 0, 0], // In
3 => [84, 97, 115, 104, 107, 101, 110, 116, 0, 0] // Tashkent
]
```

The next step is to save the indices of the modified word arrays, what later would allow us to sort the sentence without sorting the characters inside the words. This may simply be made by the **array_keys** function, which returns the keys of the **tempArray**, which are the indices of the words.

Afterwards, every word is processed by invoking **radixSort** method, which is described below.

```
function radixSort(array $input, int $levelIndex, array $indices): array
{
    $array = array_map(function ($item) use ($levelIndex) {
        return $item[$levelIndex];
    }, $input);

    $exp = floor(log10(max($array))) + 1;
    return countingSort($array, $exp, $indices);
}
```

The function takes **input** array, **levelIndex** and an array of **indices** as parameters and returns modified indices as an output. **levelIndex** variable represents current letter processed, what allows to produce an array of the current letters out of the whole input (all words in the sentence). The forwarding step checks the maximum power of ten in the ASCII codes provided. The function returns indices modified in the **countingSort** function (words stored by the current letter – **levelIndex**).

```
function counting(array $input, int $exp, array $indices): array
{
    for ($i = 0; $i < $exp; $i++) {
        $buckets = [];

        foreach($indices as $index){
            $bucket = floor($input[$index] / pow(10, $i)) % 10;
            $buckets[$bucket] = $buckets[$bucket] ?? [];
            array_push($buckets[$bucket], $index);
        }

        $indices = [];
        for($k = 0; $k < 10; $k++){
            if(isset($buckets[$k])){
                array_push($indices,...$buckets[$k]);
            }
        }
    }
    return $indices;
}
```

The **countingSort** function takes **input** array, **exponent** and indices as an input and return modified **indices** back to the **radixSort** and the **multiLayerRadixSort** functions. The first step is creation of empty buckets. Subsequently, we store indices in the buckets, where the keys of the buckets array correspond to the result of modulus operation on the input digits. This step allows us to store the position of the whole word, while sorting its letter. This is achieved by storing the index of the word, but not the value of the character. Consequently, indices are flushed and refilled according their position in the buckets. These steps are repeated until the numbers (ASCII codes) are stored. This function returns indices sorted by the current letter.

The last step is storing the words in a sentence according to their updated indices.

```
/** formatting the output */

return implode(' ', array_map(function($item) use($input){
    return $input[$item];
}, $indices));
```

The input and the output code with the actual I/O of the program are depicted below:

```
$input = "INHA University In Tashkent";
$output = multiLayerRadixSort($input);
print_r("Input:\n$input\nOutput:\n$output");
```

Console Output

Input:
INHA University In Tashkent
Output:
INHA In Tashkent University

ANALYSIS OF PROPOSED ALGORITHM

The algorithm utilizes radix sort approach, which is not a comparison-based sorting algorithm. Comparison based algorithms (Merge Sort, Heap Sort, Quick-Sort, etc) are $O(n \cdot \log(n))$. Counting sort is a linear time sorting algorithm that sort in $O(n+k)$ time when elements are in range from 1 to k, what is used in the Radix sorting technique.

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 ASCII 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 an algorithm is:

$$O(n) + O(n) + 30 * O(n) = 32 O(n) = O(n)$$

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.

Table 1. Complexity of layered radix sort

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

CONCLUSION

A non-comparative sorting algorithm avoids comparison by creating and distributing elements into buckets according to their radix. Mostly 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 our proposed algorithm made a sorting of strings with the method of converting chars to ascii codes, which reduced number of buckets from 27 to 10 and most importantly, time complexity is improved. This approach may be used for solving following real-life problems:

- Sorting of real text using any random delimiter. For instance, to sort the 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.

REFERENCES

- Cook, C., & Kim, D. (2011). Best sorting algorithm for nearly sorted lists. *ACM Communications*, 23(11), 620–624. doi:10.1145/359024.359026
- Goodrich, M., & Tamassia, R. (2010). *Data Structures and Algorithms in Java*. John Wiley & Sons.
- Kharabsheh, K. S., AlTurani, I. M., & Zanoon, N. I. (2013). Review on Sorting Algorithms A Comparative Study. *International Journal of Computer Science and Security*, 7(3).
- Seth, A., & Seth, K. (2021). Optimal Composition of Services for Intelligent Systems using TOPSIS. *International Journal of Information Retrieval Research*, 11(3), 49–64. doi:10.4018/IJIRR.2021070104
- Sipser, M. (1996). *Introduction to the Theory of Computation*. Thomson.

Ashish Seth is a Consultant, Researcher and Teacher. He is a Professor at School of Global Converge Studies, Inha University, Incheon, South Korea and is presently deputed at Inha University in Tashkent. He has more than 18 years of research and teaching experience. He worked at various universities in India and abroad at various academic positions and responsibilities. He is senior member IEEE and ACM distinguished speaker. He is an active member of International societies like IEEE, ACM, CSI, IACSIT, IAENG, etc. He is also serving as an editor, reviewer for some journals. He finds interest in reading and writing articles on emerging technologies.