Project Report:

Critical Thinking 5 – MSD RadixSort

Alejandro Ricciardi

Colorado State University Global

CSC400: Data Structures and Algorithms

Professor: Hubert Pensado

September 15, 2024

Project Report:

Critical Thinking 5 – MSD RadixSort

This documentation is part of the Critical Thinking 5 Assignment from CSC400: Data Structures and Algorithms at Colorado State University Global. This Project Report is an overview of the program's functionality and testing scenarios including console output screenshots. The program is coded in Java JDK-22 and named Critical Thinking 5 (MSD RadixSort).

The Assignment Direction:

String Objects

1. Implement the radix sort algorithm in Java. Analyze your algorithm in Big-Oh notation and provide the appropriate analysis.
2. Analyze the algorithm by documenting the steps taken when radix sort sorts the following array of string objects: joke book back dig desk word fish ward dish wit deed fast dog bend
3. Analyze the Big-O of your algorithm.

Ensure that your program has the required class and a test class. Compile and submit CTA 5, the Big-Oh evaluations, and screenshots of your program's execution and output in a single document. Also attach all appropriate source code. Combine all contents into a single zip file.

⚠️ My notes:

- Using Most Significant Digit (MSD) radix sort approach to sort the strings in alphabetic order.
- Implementing recursion in the MSD radix sort.
- Only lowercase words are allowed.

My Program Description:

MSD RadixSort is an implementation of a Most Significant Digit (MSD) radix sort in Java, which sorts an array of lowercase strings in alphabetical order by examining each character from left to right. The program uses recursion to sort subarrays (buckets) based on character positions.

Git Repository

I use GitHub as my Distributed Version Control System (DVCS), the following is a link to my GitHub, Omegapy.
My GitHub repository that is used to store this assignment is named My-Academics-Portfolio and the link to this specific assignment is: https://github.com/Omegapy/My-Academics-Portfolio/tree/main/Data-Structures-and-Algorithms-CSC400/Critical-Thinking-5

Classes Description:

- **MSDRadixSort Class**
  Sorts an array of strings using MSD radix sort algorithm with buckets and by utilizing an auxiliary array 'aux'.

- **MSDRadixSortTest Class**
  The main method runs all test cases for MSD Radix Sort.
  Test cases:
    1. Regular unsorted array
    2. Array with words of varying lengths
    3. Array with duplicates
    4. Empty array
    5. Array with one element
    6. Array with all identical elements
    7. Array with upper and lower case characters
    8. Array with special characters
    9. Display buckets during sorting

Radix Sort

“Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys” (GeeksforGeeks, 2024). Radix Sort does not compare elements directly, it sorts elements into buckets based on each digit or character value. It repeatedly sorts the elements by their significant digits or characters by using two different approaches, the Least Significant Digit (LSD) approach or the Most Significant Digit (MSD). With LSD the elements' digits or characters are sorted from the least significant to the most significant, from right to left. On the other hand, with MSD the elements' digits or characters are sorted from the most significant to the least significant, from left to right.

Buckets are a grouping of numbers or words based on their digit(s) or character(s) values. For instance, words starting with the letter ‘a’ can be grouped in a bucket, while words starting with the letter 'b' can be grouped in another bucket.

My Recursive MSD Radix Sort Approach Explain

For this assignment, based on the data to be sorted, I chose to use a recursive MSD radix sort because it is well-suited for sorting strings with variable lengths, the assignment data.

Radix Sort, using the MSD approach, sorts from left to right, making it well-suited for sorting strings, when strings are read from left to right and need to be sorted in alphabetical order. Additionally, recursion is also well suited for this assignment, it effectively handles data with varying string lengths, and it implements a stable sort, which is crucial for the Radix Sort to work properly.

Note that when using recursion, buckets containing a large amount of data may cause a 'StackOverflowError'. This can be handled by limiting the size of the buckets and splitting large buckets into two. For example, a large bucket containing words starting with the letter ‘a’ could be split into two buckets, one containing words starting with the letters ‘aa’ to ‘ao’ and another one containing words starting with the letters 'ap’ to ‘az’. For simplicity, this was not implemented in this assignment as is not required and the size of the data is small.

Analysis of the Sorting Process

Buckets in the Code

- **Step-1** - Bucket Creation and Size Determination:
  In the MSDRadixSort the private recursive method:
  `private static void msdRadixSort(String[] array, String[] aux, int low, int high, int digit)`
  The `int[] bucketCount = new int[R + 2];` array acts as the frequency count of characters at the current digit position, that is how many strings share the same character of characters at the current digit position (R), determining the sizes of the buckets.
  `R + 2` is used to allocate the count array with extra slots beyond just the 26 possible characters, the first extra slot `bucketCount[0]` handles strings that are shorter than the current digit position. For example, if the algorithm is sorting at the 4th character but the string has only 3 characters.
  The second extra slot `bucketCount[R + 1]` is used as padding. It ensures that indexing doesn't go out of bounds when sums are calculated. In step-1

- **Step-2** - Bucket Starting Positions:
  Here the counts are transformed into indices to determine the starting positions of each bucket in the auxiliary array `aux`.
  This is done by adding the counts together: `bucketCount[r + 1] += bucketCount[r];`.

- **Step-3** - Distributing Strings into Buckets:
  Here a for-loop that iterates over the string array from low to high places each string into its corresponding bucket in `aux`.
  `aux[bucketCount[c + 1]++] = array[i];` places the string `array[i]` into the correct bucket in `aux` and increments the count to point to the next position in the bucket.

- **Step-4** - Copying Buckets Back to the Original Array:
  Here the strings from the buckets in the `aux` array are copied back into the main string array.
  Note that while the strings within each bucket are not alphabetically sorted, the buckets themselves are ordered by digits, resulting in the main string array being partially sorted. For instance, all words starting with the letter 'a' are grouped in the first bucket, though they are not unsorted within it. Similarly, the second bucket contains words beginning with the letter 'b,' but these words are also unsorted within it, and so forth.

- **Step-5** - Recursive Sorting of Buckets:
  Here the `msdRadixSort` method is called recursively repeating steps 1 through 5 until `low >= high`. Note that the recursive call is nested into a for-loop creating a bucket array:

```
for (int r = 0; r < R; r++) { // 'a', r = 0; 'z', r = 25
        msdRadixSort(array, aux, low + bucketCount[r], low + bucketCount[r + 1] - 1, digit + 1);
}
```

  creating a bucket array for each possible character at the current digit. For each character bucket, the algorithm recursively sorts the strings based on the next digit (i.e., digit + 1).

Buckets Illustration During Sorting

Initial array:
["joke", "book", "back", "dig", "desk", "word", "fish", "ward", "dish", "wit", "deed", "fast", "dog", "bend"]

Table 1
First Level Buckets

Bucket (Character)	Strings
'b'	["book", "back", "bend"]
'd'	["dig", "desk", "dish", "deed", "dog"]
'f'	["fish", "fast"]
'j'	["joke"]
'w'	["word", "ward", "wit"]

Note: Based on the first character.

Second Level Buckets (Each First-Level Bucket, Based on Second Character):

For the 'b' bucket:

Bucket (Character)	Strings
'a'	["back"]
'e'	["bend"]
'o'	["book"]

And so on for other buckets.

Big-O Complexity Analysis

To recap, the Recursive MSD Radix Sort sorts the string one digit (or character) at a time. Its Radix (`R = 26`) is based on the 26 lowercase letters (a to z). The auxiliary array, `aux`, stores the string temporally during the sorting process, and each recursion processes one character at a time, starting from the most significant digit.

Time Complexity

- The recursive depth is the length of the longest string, let's call it *L*, therefore, the reduction of the recursion will go up to *L* depth.
- Step-1 computing frequency counts, this involves iterating through all *n* strings, thus the number of operations leads to an *O(n).*
- Step-2 transforming counts to indices, this is a constant-time operation due to radix (`R = 26`), so this leads to a complexity of *O(R)* or *O(26).*
- Step-3 distributing strings into buckets, since every string needs to be placed in a bucket, and it is *n* strings, thus the number of operations leads to an *O(n).*
- Step-4 coping string back to the original string array, since all the strings need to be copied back this also *O(n).*
- Step-5 recursively sorts each bucket, the overall complexity.

Final Complexity:
Since each step (1, 3, 4) is *O(n),* and the recursion depth is *L*, then the overall time complexity is *O(n × L).*

Time Complexity=O(n×L)
where:

- *n* is the number of strings.
- *L* is the length of the longest string.

Space Complexity:

- The algorithm uses an auxiliary array `aux` of size *n* to store the strings during sorting, therefore the space complexity is *O(n)* when storing in `aux`.
- The bucketCount array of size `R+2` (constant space for 28 letters) does not affect the overall complexity, so the total space complexity is *O(n).*

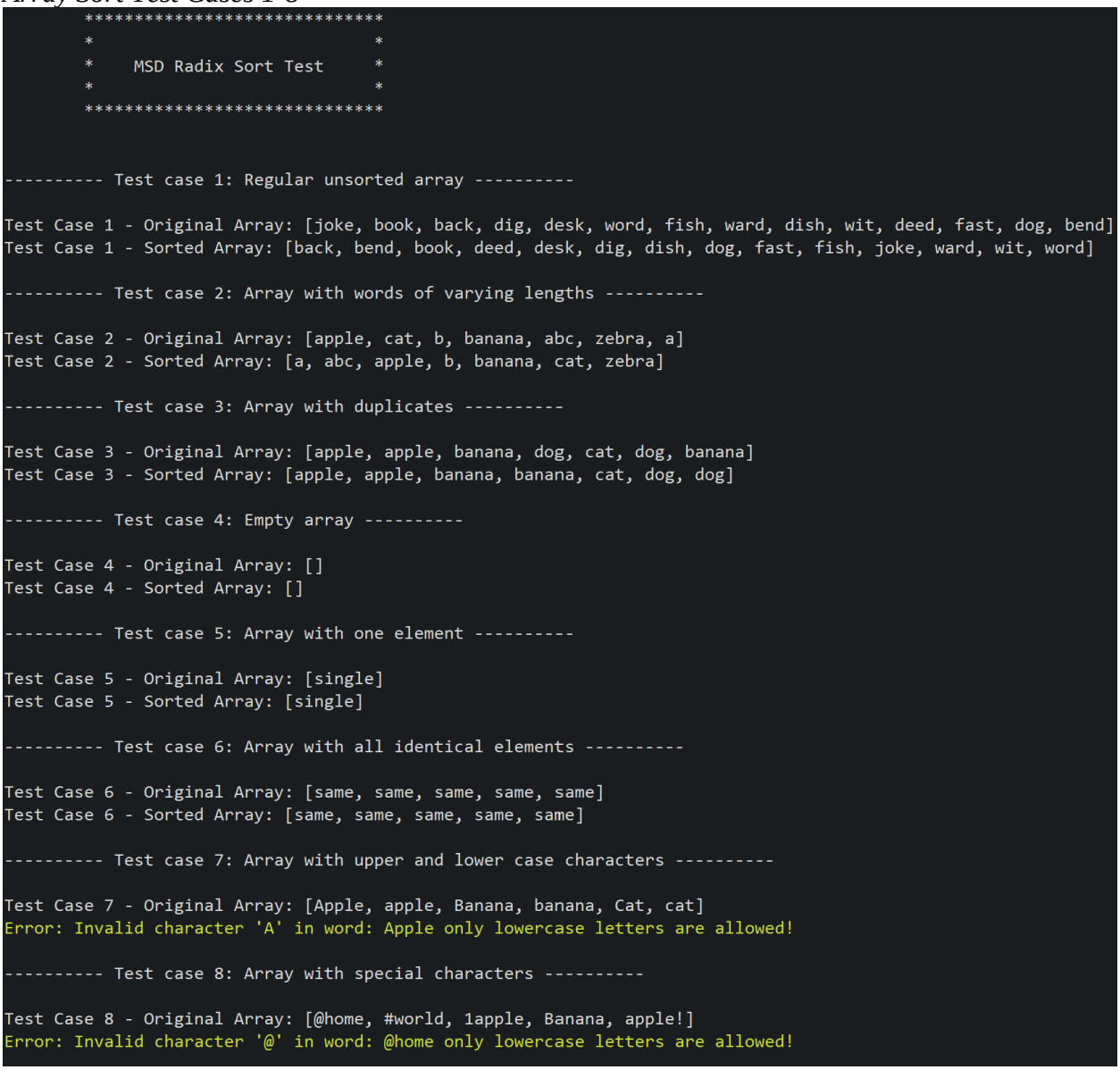
The Big-Ohs of the algorithm are
Time Complexity: *O(n×L)*
Space Complexity: *O(n)*

Screenshots

This section shows the results of the test performed on my Recursive MSD Radix Sort algorithms:

Figure 1

Array Sort Test Cases 1-8

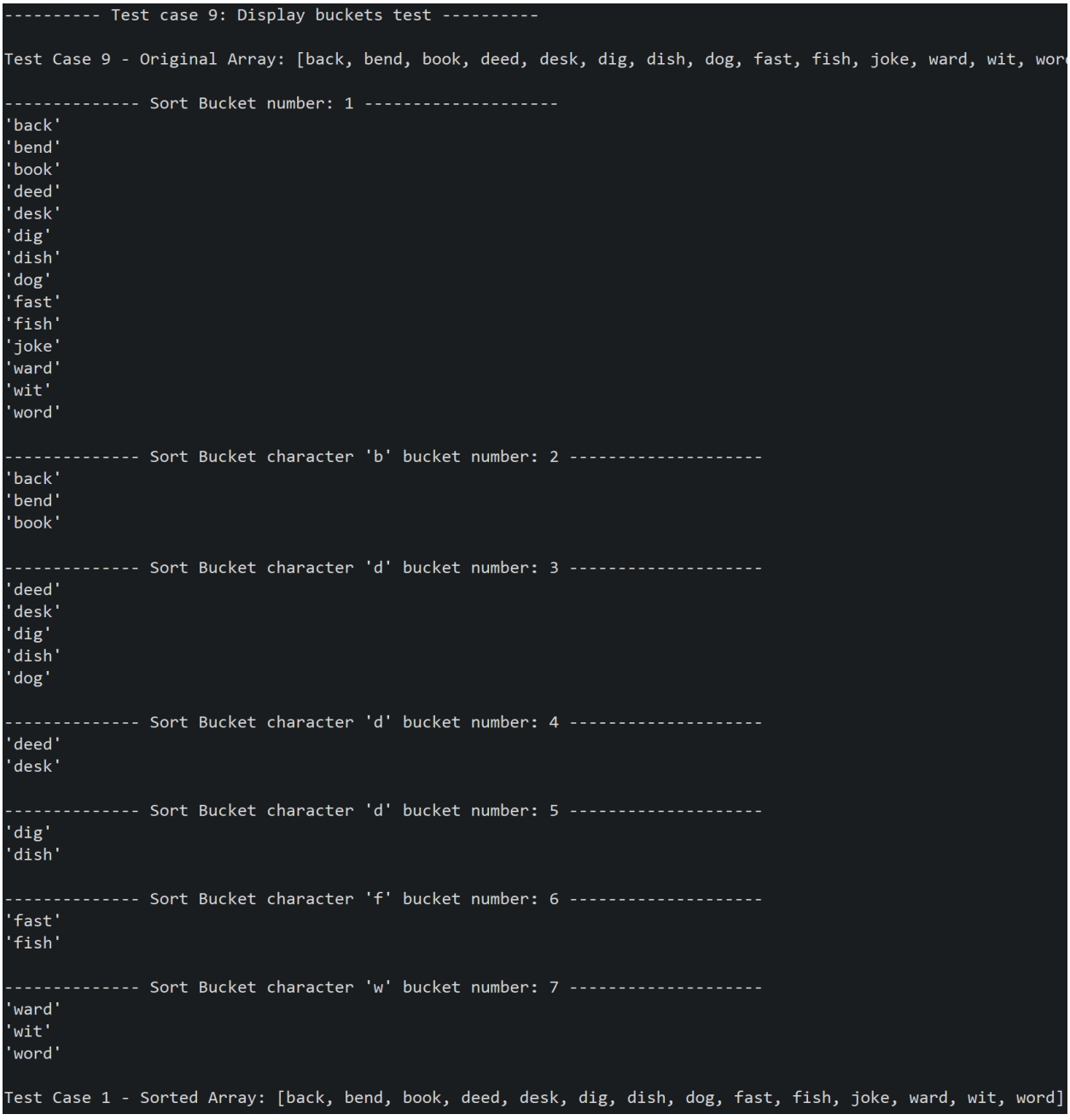


```
        ****************************
        *                          *
        *    MSD Radix Sort Test   *
        *                          *
        ****************************


---------- Test case 1: Regular unsorted array ----------

Test Case 1 - Original Array: [joke, book, back, dig, desk, word, fish, ward, dish, wit, deed, fast, dog, bend]
Test Case 1 - Sorted Array: [back, bend, book, deed, desk, dig, dish, dog, fast, fish, joke, ward, wit, word]

---------- Test case 2: Array with words of varying lengths ----------

Test Case 2 - Original Array: [apple, cat, b, banana, abc, zebra, a]
Test Case 2 - Sorted Array: [a, abc, apple, b, banana, cat, zebra]

---------- Test case 3: Array with duplicates ----------

Test Case 3 - Original Array: [apple, apple, banana, dog, cat, dog, banana]
Test Case 3 - Sorted Array: [apple, apple, banana, banana, cat, dog, dog]

---------- Test case 4: Empty array ----------

Test Case 4 - Original Array: []
Test Case 4 - Sorted Array: []

---------- Test case 5: Array with one element ----------

Test Case 5 - Original Array: [single]
Test Case 5 - Sorted Array: [single]

---------- Test case 6: Array with all identical elements ----------

Test Case 6 - Original Array: [same, same, same, same, same]
Test Case 6 - Sorted Array: [same, same, same, same, same]

---------- Test case 7: Array with upper and lower case characters ----------

Test Case 7 - Original Array: [Apple, apple, Banana, banana, Cat, cat]
Error: Invalid character 'A' in word: Apple only lowercase letters are allowed!

---------- Test case 8: Array with special characters ----------

Test Case 8 - Original Array: [@home, #world, 1apple, Banana, apple!]
Error: Invalid character '@' in word: @home only lowercase letters are allowed!
```

Continue next page

Figure 1

Bucket Sort Test Case 9



As shown in Figures 1 and 2 the program runs without any issues displaying the correct outputs as expected.

References

GeeksforGeeks (2024, August 29). Radix sort – Data structures and algorithms tutorials. GeeksforGeeks. https://www.geeksforgeeks.org/radix-sort/