SCHOOL OF COMPUTER SCIENCES UNIVERSITI SAINS MALAYSIA

CPT212 : DESIGN & ANALYSIS OF ALGORITHMS

Semester II, Academic Session: 2024/2025

Report

Lecturer | Assoc Prof. Dr. Tan Tien Ping

Assignment 1

Principles of Analysis of Algorithms and Sorting Methods

Prepared By:

Name	Matric Number
Goh Shu Ying	**22305493**
Saw Yee Xuan	**22305488**
Arwen Laung Ai Wen	**22303449**

Date of Submission: 11th May 2025

Table of Contents

List of Figures

Question 1: Radix Sort Numbers

Code

```
import java.util.Arrays;

public class RadixSort {

    public static void main(String[] args) {
        // Predefined input
        int[] numbers = { 275, 87, 426, 61,409,170, 677, 503 };

        System.out.println("Example: " + Arrays.toString(numbers));
        radixSort(numbers);

        // Print out the final sorted array numbers
        System.out.print("-".repeat(80));
        System.out.print("\nFinal Sorted list: ");
        for (int i = 0; i < numbers.length; i++) {
            System.out.print(numbers[i] + " ");
        }
    }

    public static void radixSort(int[] array) {
        // Define 2D array as buckets for 10 digits
        int[][] digitBuckets = new int[10][array.length];

        // Define array for a count for each digit bucket
        int[] bucketCount = new int[10];

        // Initialise the maximum value of the array as first element of
        the array
        int max = array[0];
        // Find the maximum value of the array
        for (int i = 1; i < array.length; i++) {
          if (array[i] > max) {
            max = array[i];
          }
        }
        // Get the number of digits of the maximum value of the array
        int maxDigits = String.valueOf(max).length();

        // Start by initialising divisor to 1
        int divisor = 1;

        // Iterate through each digit from LSD to MSD
        for (int pass = 1; pass <= maxDigits; pass++) {
        // Place elements into the buckets
            for (int index = 0; index < array.length; index++) {
                // Get the digit starting from LSD
```

```
            int digit = (array[index] / divisor) % 10;
            // Place the number into the correct digit bucket and
            increase its bucketCount
            digitBuckets[digit][bucketCount[digit]++] =
            array[index];
        }

        // Display current pass in clean format
        System.out.println("\nPass " + pass);
        printBuckets(digitBuckets, bucketCount);
        System.out.println("");

        // Print current pass sorted list
        System.out.print("Pass " + pass + " Sorted List: ");
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < bucketCount[i]; j++) {
                System.out.print(digitBuckets[i][j] + " ");
            }
        }
        System.out.println("");

        // Collect from buckets back to array or, flatten the 2D
        array to a single array
        int index = 0;
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < bucketCount[i]; j++) {
                array[index++] = digitBuckets[i][j];
            }
        }

        // Reset bucket counts
        for (int i = 0; i < bucketCount.length; i++) {
            bucketCount[i] = 0;
        }

        // Move to next significant digit
        divisor *= 10;
    }
}

// Helper function to print the buckets
public static void printBuckets(int[][] buckets, int[] count) {
    int maxNumber = 0;
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < count[i]; j++) {
            if (buckets[i][j] > maxNumber) {
                maxNumber = buckets[i][j];
            }
        }
    }
```

```
        // Determine digit width
        int digitWidth = String.valueOf(maxNumber).length();
        String format = " %" + digitWidth + "d |";

        // Print out the header row
        System.out.print("| Digit Bucket |");
        for (int i = 0; i < 10; i++) {
            System.out.printf("%" + digitWidth + "s  |", i);
        }
        System.out.println();

        // Find maximum number of rows to print
        int maxRows = 0; // Initialise maxRows to 0
        // Iterate through each element in count of each digit bucket to
        find the bucket with the highest number of count
        for (int index = 0; index < count.length; index++) {
            if (count[index] > maxRows) {
                maxRows = count[index]; // If the count in the digit
                bucket is higher than previous digit bucket count, then
                update the current digit bucket count as maxRows
            }
        }

        // Print out each consecutive row after header with format
        for (int row = 0; row < maxRows; row++) {
            System.out.print("|              |");
            for (int col = 0; col < 10; col++) {
                if (row < count[col]) {
                    System.out.printf(format, buckets[col][row]);
                } else {
                    System.out.printf(" %" + digitWidth + "s |", "");
                }
            }
            System.out.println();
        }
    }
}
```

Explanation

The Java program begins by initialising a fixed array of non-negative integers and implementing the `radixSort` method to sort them in ascending order. In the `main` method, the array `{275, 87, 426, 61,409,170, 677, 503}` is printed to the console, then passed to `radixSort`.

In `radixSort`, the algorithm first sets up a two-dimensional array `digitBuckets[10][n]`: One row for each possible digit 0–9, and a one-dimensional `bucketCount[10]` array to track how many numbers are placed in each bucket during a pass. It then loops through the array to find its maximum value and determine the number of digits in that maximum, referred to as `maxDigits`. This determines the number of passes the sort will take, one for each digit placed from the least significant (units) to the most significant (e.g., ten-thousands).

Each of the `maxDigits` passes follows the same pattern. First, the algorithm extracts the current digit of each number by dividing by a `divisor` (initially 1, then 10, 100, etc.) and then taking the remainder modulo 10. It places each number into the corresponding bucket and increments that bucket's counter. It then calls `printBuckets` to print out the bucketed contents in a table, showing exactly which numbers are in which digit class. Next, it "collects" the buckets back into the original array in order by taking all the numbers out of bucket 0 first, then bucket 1, and so on up to bucket 9. Thus preserving the relative order of numbers with identical digits. Finally, it resets all bucket counts to zero and multiplies the divisor by 10 to focus on the next more significant digit.

Output

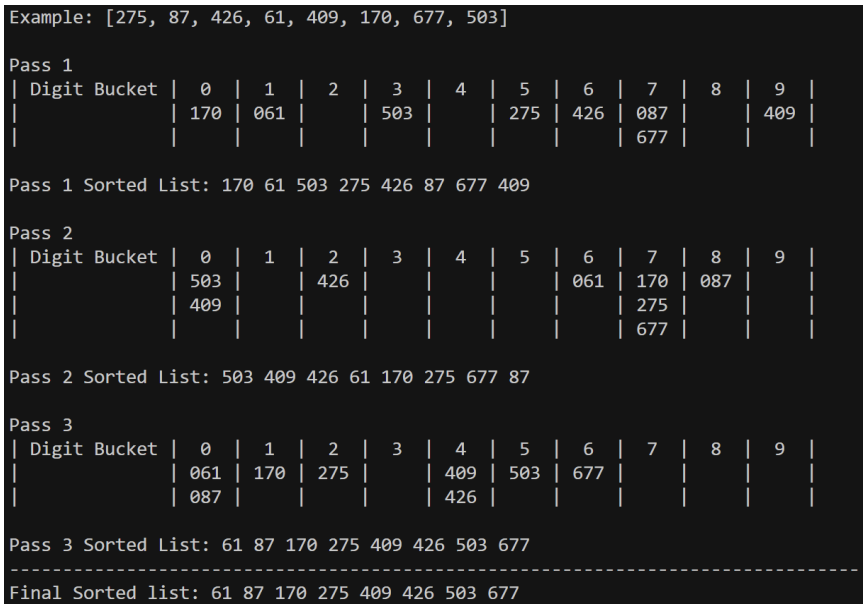

Figure 1.1 Output of RadixSort.java with Maximum Digit of 3

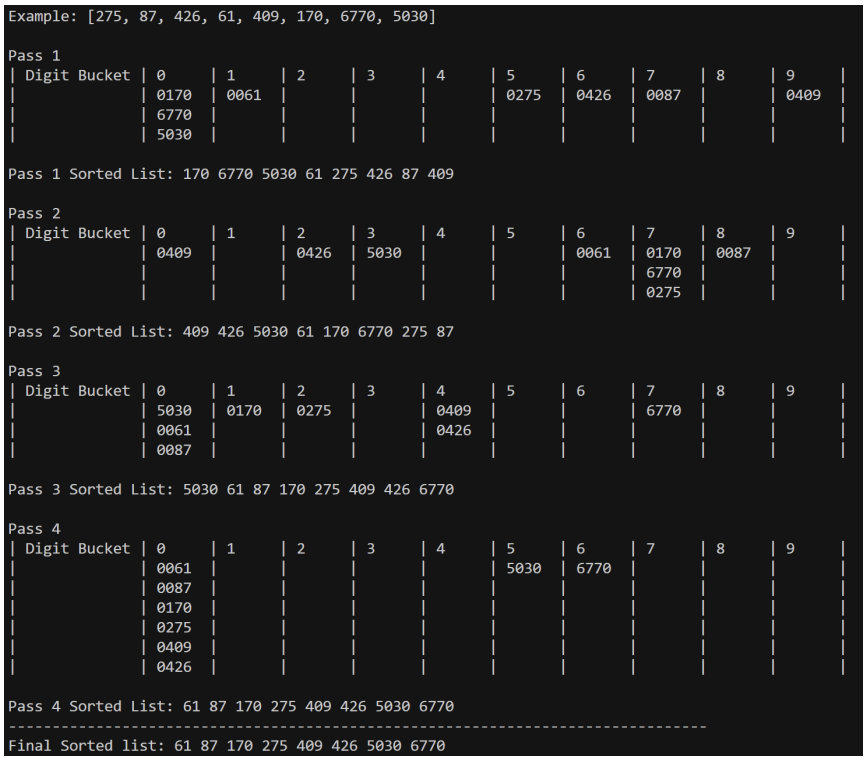

Figure 1.2 Output of RadixSort.java with Maximum Digit of 4

Question 2: Radix Sort Strings

Code

```
import java.util.Arrays;

public class RadixSortWords {

  public static void main(String[] args) {
    // Predefined input
    String[] words = { "hello", "hell", "cat", "fly", "jump", "cupcake", "cup",
    "cake" };

    System.out.println("Example: " + Arrays.toString(words));

    // Sort the words by calling the predefined method
    radixSortWords(words);

    // Print out the final sorted array words
    System.out.print("Sorted Words: ");
    for (int i = 0; i < words.length; i++) {
      System.out.print(words[i] + " ");
    }
  }

  public static void radixSortWords(String[] arr) {
    // Create two 2D arrays
    String[][] Array1 = new String [26][]; // Outer array
    // Inner array
    for (int i = 0; i < 26; i++) {
      Array1[i] = new String[arr.length];
    }
    String[][] Array2 = new String [26][]; // Outer array
    // Inner array
    for (int i = 0; i < 26; i++) {
      Array2[i] = new String[arr.length];
    }

    // Create two arrays for each alphabet bucket in each 2D array to track \
    number of counts in each bucket
    int[] count1 = new int [26];
    int[] count2 = new int [26];

    // Print out initialisation buckets
    System.out.println("\n1. Initialisation:");
    System.out.println("Array 1:");
    // Print the buckets in format by calling predefined method
    printBuckets(Array1, count1, "Array 1");
    System.out.println("Array 2:");
    printBuckets(Array2, count2, "Array2");
```

```
// Finds the longest string length
int maxLength = 0;
for (int i = 0; i < arr.length; i++) {
  maxLength = Math.max(maxLength, arr[i].length());
}

// Print the maximum length of word
System.out.println("Maximum Length of Word: "+ maxLength);
System.out.println("-".repeat(100) + "\n");

// Print the iterations
System.out.println("2. Iteration: ");

// Start iterating from the LSF, right most alphabet of the word
for (int pass = maxLength - 1; pass >= 0; pass-- ) {
  if ((maxLength - pass - 1) % 2 == 0) {
    // Distribute elements to alphabet buckets
    for (int i = 0; i < arr.length; i++) {
      int letterBucket = getCharIndex(arr[i], pass);
      Array1[letterBucket][count1[letterBucket]++] = arr[i];
    }

    System.out.println("Iteration of character at position " + pass + ": ");
    printBuckets(Array1, count1, "Array1");

    // Collect elements from buckets back into array
    int index = 0;
    for (int i = 0; i < 26; i++) {
      for (int j = 0; j < count1[i]; j++) {
        arr[index++] = Array1[i][j];
      }
      count1[i] = 0;
    }

    // Print the updated array after collecting from alphabet buckets
    System.out.print("Current Array 1 List: ");
    for (int i = 0; i < arr.length; i++) {
      System.out.print(arr[i] + " ");
    }
    System.out.println("\n");
  }
  else {
    // Distribute elements to alphabet buckets
    for (int i = 0; i < arr.length; i++) {
      int letterBucket = getCharIndex(arr[i], pass);
      Array2[letterBucket][count2[letterBucket]++] = arr[i];
    }

    System.out.println("Iteration at character at position " + pass +": ");
```

```
        printBuckets(Array2, count2, "Array2");

        // Collect elements from buckets back into array
        int index = 0;
        for (int i = 0; i < 26; i++) {
          for (int j = 0; j < count2[i]; j++) {
            arr[index++] = Array2[i][j];
          }
          count2[i] = 0;
        }

        // Print the updated array after collecting from buckets
        System.out.print("Current Array 2 List: ");
        for (String word : arr) {
          System.out.print(word + " ");
        }
        System.out.println("\n");
      }
      System.out.println("-".repeat(100) + "\n");
    }
  }

  // Get the position of the letter in the alphabet (a = 0, b = 1, ..., z = 25)
  // Pads short words by treating missing characters as 'a' (index 0)
  public static int getCharIndex(String word, int position) {
    if (position < word.length()) {
      return word.charAt(position) - 'a';
    }
    else {
      return 0;
    }
  }

  // Helper function to print the alphabet buckets in format, print array
  contents vertically (one bucket per line)
  private static void printBuckets(String[][] array, int[] count, String
  arrayName) {
    System.out.println("\n---" + arrayName + " Alphabet Buckets ---");

    for (int i = 0; i < 26; i++) {
      char bucketLabel = (char) ('a' + i);
      System.out.print(bucketLabel + ": ");

      for (int j = 0; j < count[i]; j++) {
        System.out.print(array[i][j] + " ");
      }

      System.out.println();
    }
    System.out.println();
```

```
  }
}
```

Explanation

This program is a modified version of the original radix sort that was used for sorting numbers. Instead of sorting digits, it has been adapted to sort words made up of lowercase letters [a–z]. While the original code used a single 2D array of digit buckets to sort numbers based on their digits, this modified version uses two 2D arrays to alternately store words during passes. This allows for more flexibility when dealing with strings of varying lengths, ensuring that words are properly sorted by each character position.

Radix sort works by taking each element in the list into a specific bucket based on a character at a given position (Rastogi, 2023). The sorting process starts from the rightmost character to the left. Each word is placed in a "bucket" corresponding to the current character. These buckets are based on the alphabet (a-z), where 'a' corresponds to bucket 0, 'b' corresponds to bucket 1, and so on. If a word is shorter than the current character position being evaluated, it is treated as having an 'a' in that position to keep the sort consistent even for words of varying lengths.

The input to this sorting algorithm is a predefined list of words: "hello", "hell", "cat", "fly", "jump", "cupcake", "cup" and "cake". These words serve as the dataset to demonstrate how the radix sort algorithm works. Since words can have different lengths, the program first finds the longest word in the list, which determines the number of sorting passes, ensuring that even the longest word is fully processed.

After each pass, the contents of the buckets and the updated array are printed to show the sorting progress. This iterative process continues until all character positions have been processed, resulting in a final array of words sorted in alphabetical order.

Output

```
Example: [hello, hell, cat, fly, jump, cupcake, cup, cake]

1. Initialisation:
Array 1:

---Array 1 Alphabet Buckets ---
a:
b:
c:
d:
e:
f:
g:
h:
i:
j:
k:
l:
m:
n:
o:
p:
q:
r:
s:
t:
u:
v:
w:
x:
y:
z:
```

```
Array 2:

---Array2 Alphabet Buckets ---
a:
b:
c:
d:
e:
f:
g:
h:
i:
j:
k:
l:
m:
n:
o:
p:
q:
r:
s:
t:
u:
v:
w:
x:
y:
z:

Maximum Length of Word: 7
-----------------------------------------------------------------------------------------------
```

```
2. Iteration:
Iteration of character at position 6:

---Array1 Alphabet Buckets ---
a: hello hell cat fly jump cup cake
b:
c:
d:
e: cupcake
f:
g:
h:
i:
j:
k:
l:
m:
n:
o:
p:
q:
r:
s:
t:
u:
v:
w:
x:
y:
z:

Current Array 1 List: hello hell cat fly jump cup cake cupcake

-----------------------------------------------------------------------------------------------
```

```
Iteration at character at position 5:

---Array2 Alphabet Buckets ---
a: hello hell cat fly jump cup cake
b:
c:
d:
e:
f:
g:
h:
i:
j:
k: cupcake
l:
m:
n:
o:
p:
q:
r:
s:
t:
u:
v:
w:
x:
y:
z:

Current Array 2 List: hello hell cat fly jump cup cake cupcake

-----------------------------------------------------------------------------------------------
```

```
Iteration of character at position 4:

---Array1 Alphabet Buckets ---
a: hell cat fly jump cup cake cupcake
b:
c:
d:
e:
f:
g:
h:
i:
j:
k:
l:
m:
n:
o: hello
p:
q:
r:
s:
t:
u:
v:
w:
x:
y:
z:

Current Array 1 List: hell cat fly jump cup cake cupcake hello

-----------------------------------------------------------------------------------------------
```

```
Iteration at character at position 3:

---Array2 Alphabet Buckets ---
a: cat fly cup
b:
c: cupcake
d:
e: cake
f:
g:
h:
i:
j:
k:
l: hell hello
m:
n:
o:
p: jump
q:
r:
s:
t:
u:
v:
w:
x:
y:
z:

Current Array 2 List: cat fly cup cupcake cake hell hello jump

-----------------------------------------------------------------------------------------------
```

```
Iteration of character at position 2:

---Array1 Alphabet Buckets ---
a:
b:
c:
d:
e:
f:
g:
h:
i:
j:
k: cake
l: hell hello
m: jump
n:
o:
p: cup cupcake
q:
r:
s:
t: cat
u:
v:
w:
x:
y: fly
z:

Current Array 1 List: cake hell hello jump cup cupcake cat fly

-----------------------------------------------------------------------------------------------
```

```
Iteration at character at position 1:

---Array2 Alphabet Buckets ---
a: cake cat
b:
c:
d:
e: hell hello
f:
g:
h:
i:
j:
k:
l: fly
m:
n:
o:
p:
q:
r:
s:
t:
u: jump cup cupcake
v:
w:
x:
y:
z:

Current Array 2 List: cake cat hell hello fly jump cup cupcake

-----------------------------------------------------------------------------------------------
```

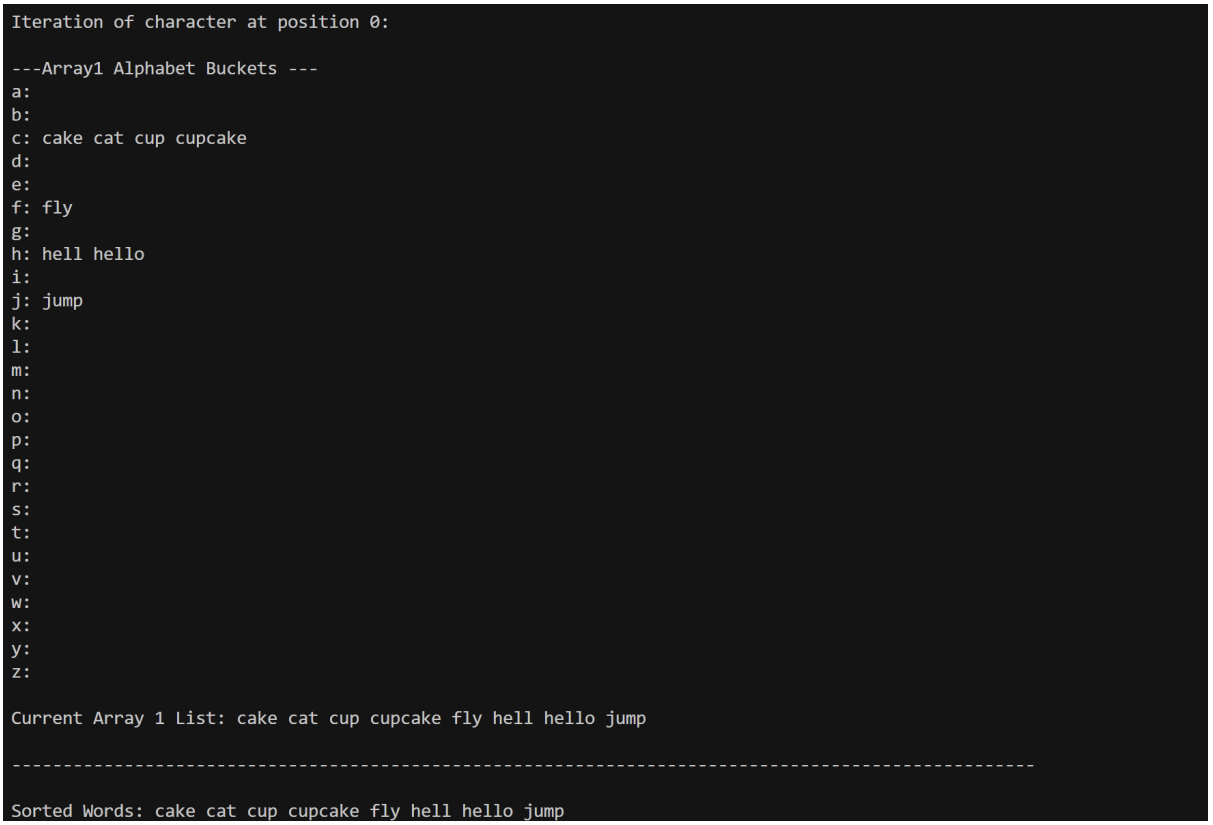

Figure 1.3 Output of RadixSortWords.java

Question 3: Radix Sort Complexity Analysis

Code

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public class RadixSort_Analysis {
    // Counter for primitive operations
    private static int operationCount = 0;

    public static void main(String[] args) throws IOException {
        // Create an array that contains sizes of input
        int[] inputSizes = {100, 500, 1000, 5000, 10000, 15000};

        // Run multiple trials for each size of input to get average
        int numberTrials = 10;

        // Create a CSV file to store experiment results for graph plotting
        purpose
        FileWriter csvResults = new FileWriter("RadixSort_Analysis.csv");
        csvResults.append("Array
        Size,Operations,Operations/InputSize,Operations/
        (MaxDigit*InputSize)\n"); // First row of the CSV file

        // Print table header to display in the terminal
        System.out.println("-".repeat(98));
        System.out.printf("| %-10s | %-18s | %-21s | %-32s |\n", "Input
        Size", "Average Operations", "Operations/Input Size",
        "Operations/(Input Size * Max Digits)");
        System.out.println("|" + "-".repeat(12) + "|" +
        "-".repeat(20) + "|" + "-".repeat(23) +
        "|" + "-".repeat(38) + "|");

        // For each size of input in array inputSizes...
        for (int i = 0; i < inputSizes.length; i++) {
            int totalOperations = 0;
            int totalDigits = 0;

            // Perform 10 times of sorting trials on this size of input
            for (int trial = 0; trial < numberTrials; trial++) {
                // Generate random array of this size, with the inputs
```

```
            ranging from 1 to 999999
            int[] array = generateRandomArray(inputSizes[i], 1,
            99999999); // Generate numbers up to 6 digits

            // Find maximum value in this array
            int max = array[0];

            // Find maximum value in this array
            for (int j = 1; j < array.length; j++) {
                if (array[j] > max) {
                    max = array[j];
                }
            }

            // Determine the maximum digits (k) of the maximum value of
            the array
            int maxDigits = String.valueOf(max).length();

            // Accumulate the total maximum digits of the arrays of the
            trials
            totalDigits += maxDigits;

            // Reset static class-level variable operationCount
            everytime before sorting next array
            operationCount = 0;

            // Sort and count operations
            radixSort(array);

            // Accumulate the total operations of the trials
            totalOperations += operationCount;
        }

        // Calculate averages of operations and digits
        int averageOperations = totalOperations / numberTrials;
        double averageDigits = (double) totalDigits / numberTrials;

        // Write results to CSV file
        csvResults.append(String.format("%d,%d,%.2f,%.2f\n",
            inputSizes[i], averageOperations, (double) averageOperations
            / inputSizes[i],
            (double) averageOperations / (inputSizes[i] *
            averageDigits)));
```

```
            // Printing each row with aligned columns
            System.out.printf("| %-10d | %-18d | %-21.2f | %-36.2f |\n",
            inputSizes[i],
            averageOperations,
            (double) averageOperations / inputSizes[i],
            (double) (averageOperations / (inputSizes[i] * averageDigits)));
        }

        System.out.println("-".repeat(98));
        System.out.println("");

        // Force any unsaved data to be written immediately to the file
        csvResults.flush();

        // Close CSV file and release resources
        csvResults.close();

        System.out.println("Data saved to RadixSort_Analysis.csv");
    }

    public static void radixSort(int[] array) {
        int[][] digitBuckets = new int[10][array.length];
        operationCount++; // Assignment to digitBuckets

        int[] bucketCount = new int[10];
        operationCount++; // Assignment to bucketCount

        int max = array[0];
        operationCount += 2; // Assignment of max, array lookup for array[0]

        for (int i = 1; i < array.length; i++) {
            operationCount += 2; // Comparison of array[i] > max, array lookup
            for array[i]
            if (array[i] > max) {
                max = array[i];
                operationCount++; // Assignment of max
            }
            operationCount += 3; // Comparison of i < array.length,
            increment of i, assignment of i
        }
        operationCount += 2; // Initialisation of i, last comparison of i <
        array.length
```

```
        int maxDigits = String.valueOf(max).length();
        operationCount++; // Assignment of maxDigits

        int divisor = 1;
        operationCount++; // Assignment of divisor

        for (int pass = 1; pass <= maxDigits; pass++) {
            for (int index = 0; index < array.length; index++) {
                int digit = (array[index] / divisor) % 10;
                operationCount += 4; // Array lookup for array[index],
                division, modulus, assignment to digit
                digitBuckets[digit][bucketCount[digit]++] = array[index];
                operationCount += 6; // Array lookups for array[index],
                bucketCount[digit], digitBuckets[digit],
                digitBuckets[digit][bucketCount[digit]],
                increment of bucketCount[digit], assignment of
                digitBuckets[digit][bucketCount[digit]++]
                operationCount += 3; // Comparison of index < array.length,
                increment of index, assignment of index
            }
            operationCount += 2; // Initialisation of index, last comparison
            of index < array.length

            int index = 0;
            operationCount++; // Assignment to index

            for (int i = 0; i < 10; i++) {
                operationCount += 3; // Comparison of i < 10, increment of
                i, assignment of i

                for (int j = 0; j < bucketCount[i]; j++) {
                    array[index++] = digitBuckets[i][j];
                    operationCount += 9; // Comparison of j <
                    bucketCount[i], increment of j, assignment of j,
                    array lookups for bucketCount[i], digitBuckets[i],
                    digitBuckets[i][j], array[index],
                    increment of array[index++], assignment of
                    array[index++]
                }
                operationCount += 2; // Initialisation of j, last comparison
                of j < bucketCount[i]
            }
```

```
            operationCount += 2; // Initialisation of i, last comparison of
            i < 10

            for (int i = 0; i < bucketCount.length; i++) {
                bucketCount[i] = 0;
                operationCount += 5; // Comparison of i <
                bucketCount.length, increment of i, assignment of i, array
                lookup for bucketCount[i], assignment of bucketCount[0]
            }
            operationCount += 2; // Initialisation of i, last comparison of
            i < bucketCount.length

            divisor *= 10;
            operationCount += 2; // Multiplication, assignment of divisor

            operationCount += 3; // Comparison of pass <= maxDigits,
            increment of pass, assignment of pass
        }
        operationCount += 2; // Initialisation of pass, last comparison of
        pass <= maxDigits
    }

    // Method to generate random array of specified size
    public static int[] generateRandomArray(int size, int min, int max) {
        int[] array = new int[size];
        Random random = new Random();

        for (int i = 0; i < size; i++) {
            array[i] = random.nextInt(max - min + 1) + min;
        }

        return array;
    }
}
```

RadixSort_Analysis.java

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;
```

```
public class RadixSortWords_Analysis {
  // Counter for primitive operations
  private static long operationCount = 0;

  public static void main(String[] args) throws IOException {
    // Create an array that contains sizes of input
    int[] inputSizes = {100, 500, 1000, 5000, 10000, 15000};

    // Run multiple trials for each size of input to get average
    int numberTrials = 10;

    // Create a CSV file to store experiment results for graph plotting purpose
    FileWriter csvResults = new FileWriter("RadixSortWords_Analysis.csv");
    csvResults.append("Array
    Size,Operations,Operations/InputSize,Operations/
    (MaxAlphabets*InputSize)\n"); // First row of the CSV file

    // Print table header to display in the terminal
    System.out.println("-".repeat(101));
    System.out.printf("| %-10s | %-18s | %-21s | %-36s |\n", "Input Size",
    "Average Operations", "Operations/Input Size", "Operations/(Input Size * Max
     Alphabets)");
    System.out.println("|" + "-".repeat(12) + "|" + "-".repeat(20) + "|" +
    "-".repeat(23) + "|" + "-".repeat(41) + "|");


    // For each size of input in array inputSizes...
    for (int i = 0; i < inputSizes.length; i++) {
      long totalOperations = 0;
      int totalMaxLength = 0;

      // Perform 10 times of sorting trials on this size of input
      for (int trial = 0; trial < numberTrials; trial++) {
          // Generate random array of this size, with the inputs ranging from 2
          to 10 letters
          String[] wordArray = generateRandomWordArray(inputSizes[i], 2, 8); //
          Generate words up to 10 letters

          // Find maximum word length in this array
          int maxLength = findMaxLength(wordArray);

          // Accumulate the total maximum digits of the arrays of the trials
          totalMaxLength += maxLength;
```

```
            // Reset static class-level variable operationCount everytime before
            sorting next array
            operationCount = 0;

            // Sort and count operations
            radixSortWords(wordArray);

            // Accumulate the total operations of the trials
            totalOperations += operationCount;
        }

        // Calculate averages of operations and digits
        long averageOperations = totalOperations / numberTrials;
        double averageAlphabets = (double) totalMaxLength / numberTrials;

        // Write results to CSV file
        csvResults.append(String.format("%d,%d,%.2f,%.2f\n",
            inputSizes[i], averageOperations, (double) averageOperations /
            inputSizes[i],
            (double) averageOperations / (inputSizes[i] * averageAlphabets)));

        // Printing each row with aligned columns
        System.out.printf("| %-10d | %-18d | %-21.2f | %-39.2f |\n",
        inputSizes[i],
        averageOperations,
        (double) averageOperations / inputSizes[i],
        (double) (averageOperations / (inputSizes[i] * averageAlphabets)));
      }

      System.out.println("-".repeat(101));
      System.out.println("");

      // Force any unsaved data to be written immediately to the file
      csvResults.flush();

      // Close CSV file and release resources
      csvResults.close();

      System.out.println("Data saved to RadixSortWords_Analysis.csv");
    }

    public static void radixSortWords(String[] arr) {
```

```
    String[][] Array1 = new String [26][arr.length];
    operationCount++; // Assignment to Array1

    String[][] Array2 = new String [26][arr.length];
    operationCount++; // Assignment to Array2

    int[] count1 = new int [26];
    operationCount++; // Assignment to count1

    int[] count2 = new int [26];
    operationCount++; // Assignment to count2

    int maxLength = findMaxLength(arr);
    operationCount += 2; // Method call of findMaxLength(), assignment of
    maxLength

    for (int pass = maxLength - 1; pass >= 0; pass-- ) {
      operationCount += 4; // 2 subtractions, modulus, comparison
      if ((maxLength - pass - 1) % 2 == 0) {
        for (int i = 0; i < arr.length; i++) {
          int letterBucket = getCharIndex(arr[i], pass);
          operationCount += 3; // Array lookup for arr[i], method call of
          getCharIndex(), assignment of letterBucket
          Array1[letterBucket][count1[letterBucket]++] = arr[i];
          operationCount += 6; // Array lookups for count1[letterBucket],
          Array1[letterBucket], Array1[letterBucket][count1[letterBucket]],
          arr[i], assignment of Array1[letterBucket][count1[letterBucket]++],
          increment of count1[letterBucket]
          operationCount += 3; // Comparison of i < arr.length, increment of i,
          assignment of i
        }
        operationCount += 2; // Initialisation of i, last comparison of i <
        arr.length

        int index = 0;
        operationCount++; // Assignment of index
        for (int i = 0; i < 26; i++) {
          for (int j = 0; j < count1[i]; j++) {
            arr[index++] = Array1[i][j];
            operationCount += 8; // Comparison of j < count1[i], increment of j,
            assignment of j, array lookups for Array1[i], Array1[i][j],
            arr[index], increment of arr[index++], assignment of arr[index++]
          }
```

```
        operationCount += 2; // Initialisation of j, last comparison of j <
        count1[i]
        count1[i] = 0;
        operationCount += 5; // Comparison of i < 26, increment of i,
        assignment of i, array lookup for count1[i], assignment of count1[i]
      }
      operationCount += 2; // Initialisation of i, last comparison of i < 26
    }
    else {
      for (int i = 0; i < arr.length; i++) {
        int letterBucket = getCharIndex(arr[i], pass);
        operationCount += 3; // Array lookup for arr[i], method call of
        getCharIndex(), assingment of letterBucket
        Array2[letterBucket][count2[letterBucket]++] = arr[i];
        operationCount += 6; // Array lookups for arr[i],
        count2[letterBucket], Array2[letterBucket],
        Array2[letterBucket][count2[letterBucket]++], assignment of
        Array2[letterBucket][count2[letterBucket]++], increment of
        count2[letterBucket]
        operationCount += 3; // Comparison of r < arr.length, increment of i,
        assignment of i
      }
      operationCount += 2; // Initialisation of i, last comparison of i <
      arr.length

      int index = 0;
      operationCount++; // Assignment of index
      for (int i = 0; i < 26; i++) {
        for (int j = 0; j < count2[i]; j++) {
          arr[index++] = Array2[i][j];
          operationCount += 8; // Comparison of j < count2[i], increment of j,
          assignment of j, array lookups for Array2[i], Array2[i][j],
          arr[index], increment of arr[index++], assignment of arr[index++]
        }
        operationCount += 2; // Initialisation of j, last comparison of j <
        count2[i]
        count2[i] = 0;
        operationCount += 5; // Comparison of i < 26, increment of i,
        assignment of i, array lookup for count2[i], assignment of count2[i]
      }
      operationCount += 2; // Initialisation of i, last comparison of i < 26
    }
    operationCount += 3; // Comparison of pass >= 0, decrement of pass,
```

```
      assignment of pass
    }
    operationCount += 2; // Initialisation of pass, decrement of pass
  }

  // Method to generate random word array of specified size
  public static String[] generateRandomWordArray(int size, int minLength, int
  maxLength) {
    String[] array = new String[size];
    Random random = new Random();

    // Lower characters to generate random words
    char[] alphabet = "abcdefghijklmnopqrstuvwxyz".toCharArray();

    for (int i = 0; i < size; i++) {
      // Determine random length for this word
      int wordLength = random.nextInt(maxLength - minLength + 1) + minLength;

      // Create a char array for the word
      char[] wordChars = new char[wordLength];

      // Fill with random letters
      for (int j = 0; j < wordLength; j++) {
          wordChars[j] = alphabet[random.nextInt(26)];
      }

      // Convert to string and add to array
      array[i] = new String(wordChars);
    }

    return array;
  }

  public static int findMaxLength(String[] arr) {
    int maxLength = 0;
    operationCount++;
    for (int i = 0; i < arr.length; i++) {
      operationCount += 2; // Array lookup for arr[i],comparison of
      arr[i].length() > maxLength
      if (arr[i].length() > maxLength) {
          maxLength = arr[i].length();
          operationCount += 2; // Array lookup for arr[i], assignment of
          maxLength
```

```
        }
        operationCount += 3; // Comparison of i < arr.length, increment of i,
        assignment of i
      }
      operationCount += 3; // Initialisation of i, last comparison of i <
      arr.length, return
      return maxLength;
    }

    // Get the position of the letter in the alphabet (a = 0, b = 1, ..., z = 25)
    // Pads short words by treating missing characters as 'a' (index 0)
    public static int getCharIndex(String word, int position) {
      operationCount++; // Comparison of position < word.length()
      if (position < word.length()) {
        operationCount++; // Return
        return word.charAt(position) - 'a';
      }
      else {
        operationCount++; // Return
        return 0;
      }
    }
}
```

RadixSortWords_Analysis.java

Explanation

Complexity analyses are performed on both radix sort algorithms for numbers and strings. The Java programs *RadixSort_Analysis.java* and *RadixSortWords_Analysis.java* implement operation counters to track the number of different primitive operations executed in the radix sort method during sorting. The primitive operations include:

- **Assignment**: Increased when variables are initialised or assigned values.
- **Arithmetic**: Incremented for operations such as addition, multiplication, subtraction, division or modulo.
- **Array lookup**: Increased when accessing an element in an array.
- **Method call**: Raised when a self-defined method in the program is called.
- **Return**: Increased whenever a value is returned from a method. (Chiarulli, n.d.)

An experiment is conducted to compare the complexity analyses on both algorithms. The lengths of input numbers are set to a minimum of 1 digit to a maximum of 8 digits, and the lengths of

input strings are set to a minimum of 2 letters to a maximum of 8 letters as well to maintain the consistency and ability to compare with each other. The programs utilise self-defined methods that generate random numbers, `generateRandomArray()` and strings, `generateRandomWordArray()`. The analysis programs run 10 trials on 10 arrays generated for each input size to increase the accuracy of the results. Then, the results are written and stored in CSV files *RadixSort_Analysis.csv* and *RadixSortWords_Analysis.csv* respectively to be used to plot the graphs.

The results of the experiment indicate that sorting strings requires a higher number of primitive operations compared to sorting numbers. This is because strings use more possible characters (a–z) than numbers (0–9), leading to a larger number of buckets in each pass of the string radix sort. So, there is a method defined in the sorting string algorithm, which is `getCharIndex()` to obtain the index of each character in alphabetical order, adding to the overall operation count.

The following shows the output of each sorting algorithm program which summarises the input size, average operations, operations per input and the constant factor as the normalisation of number of operations by input size and maximum digit and length of string.

Output

```
-------------------------------------------------------------------------------------------------
| Input Size | Average Operations | Operations/Input Size | Operations/(Input Size * Max Digits) |
|------------|--------------------|-----------------------|--------------------------------------|
| 100        | 19005              | 190.05                | 23.76                                |
| 500        | 91406              | 182.81                | 22.85                                |
| 1000       | 181906             | 181.91                | 22.74                                |
| 5000       | 905908             | 181.18                | 22.65                                |
| 10000      | 1810910            | 181.09                | 22.64                                |
| 15000      | 2715911            | 181.06                | 22.63                                |
-------------------------------------------------------------------------------------------------

Data saved to RadixSort_Analysis.csv
```

Figure 1.4 Output of RadixSort_Analysis.java

```
-------------------------------------------------------------------------------------------------
| Input Size | Average Operations | Operations/Input Size | Operations/(Input Size * Max Alphabets) |
|------------|--------------------|-----------------------|-----------------------------------------|
| 100        | 19668              | 196.68                | 24.59                                   |
| 500        | 92071              | 184.14                | 23.02                                   |
| 1000       | 182570             | 182.57                | 22.82                                   |
| 5000       | 906569             | 181.31                | 22.66                                   |
| 10000      | 1811569            | 181.16                | 22.64                                   |
| 15000      | 2716569            | 181.10                | 22.64                                   |
-------------------------------------------------------------------------------------------------

Data saved to RadixSortWords_Analysis.csv
```

Figure 1.5 Output of RadixSortWords_Analysis.java

Input size refers to the number of inputs per each sorting process. As the input size increases, it is observed that as the number of primitive operations increases, the number of primitive operations

needed per number input is smaller than string input, meaning sorting string requires more operations than sorting number. Take the input size 100 as example, the operations needed per number input is 190.05, which is smaller than operations needed per string input, that is 196.68. The last columns which show the constant factors indicate that both algorithms implementations are scaling well, and that the value of sorting number algorithm is smaller than sorting string algorithm, which makes sense because average operations of sorting number is smaller than average operations of sorting strings, provided that input sizes and maximum lengths of number and string have been set the same in both programs, so the constant factor of sorting number algorithm is slightly smaller than sorting string algorithm.

Based on the results of the experiment, two graphs of the number of primitive operations against the number of inputs are plotted for radix sort number and radix sort string algorithms in Jupyter notebook.

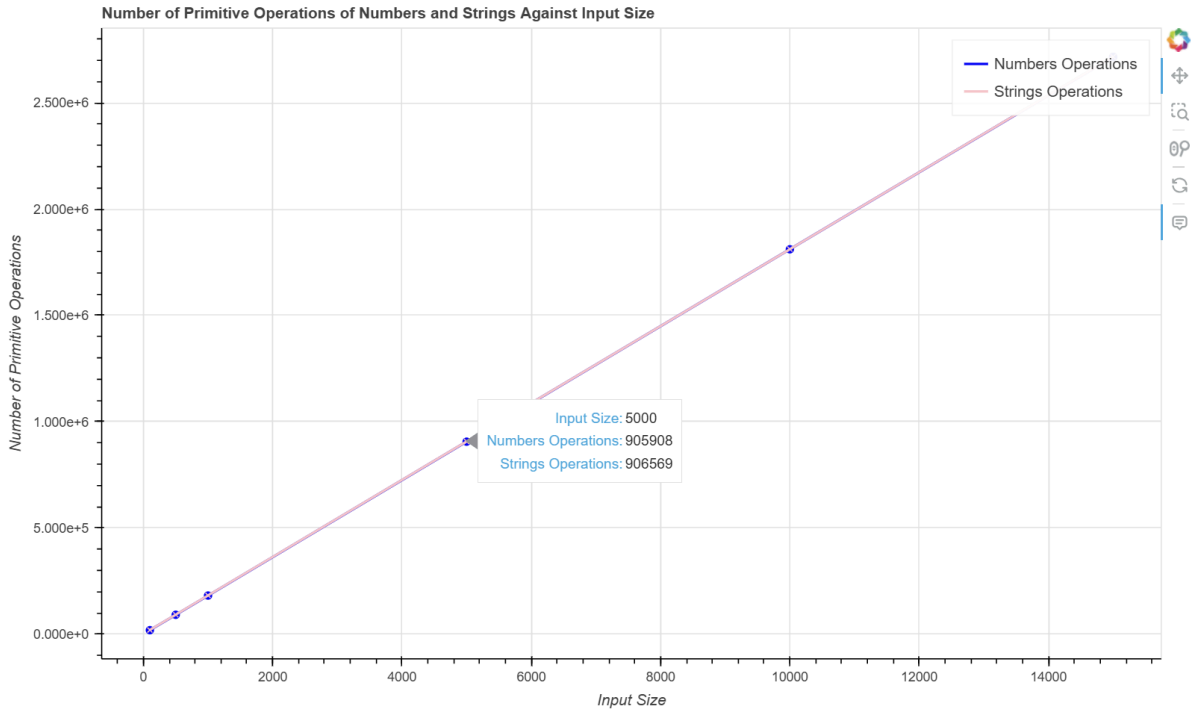

Figure 1.6 Graph of Number of Primitive Operations of Radix Sort Number and Radix Sort String Against Input Size

To calculate the big-Oh notation of the radix sort algorithm, first we need to understand the concept of radix sort. Radix Sort processes each digit from least significant to most significant. In the analysis programs, the maximum digits and the maximum length of string are set to 8, which means in the worst case scenario, there will be 8 passes over the entire list of input.

Based on the graph shown in *Figure 1.6*, as the input size increases, it is proven that the number of primitive operations increases proportionally as well, with the same maximum number of digits (for numbers) or maximum string length (for string) across all the trials and for all input sizes.

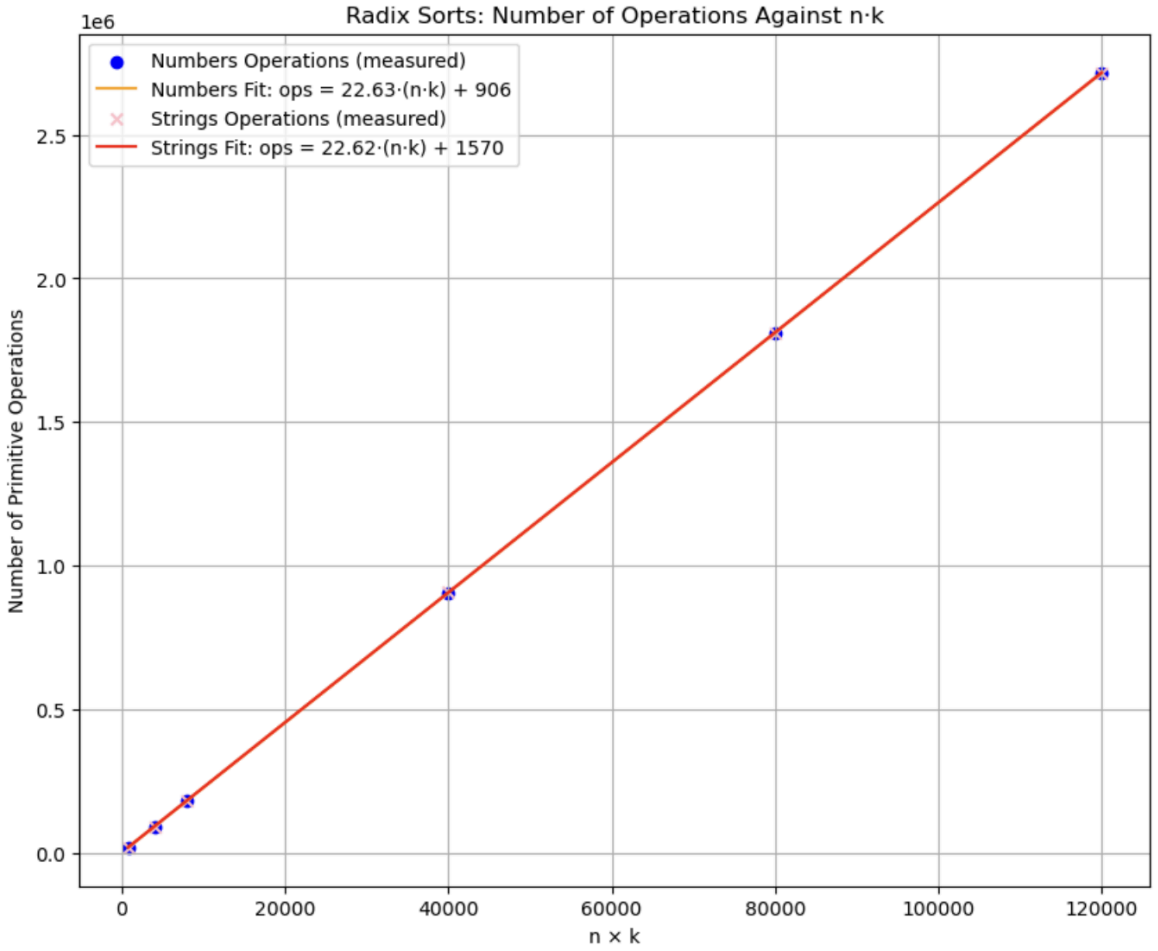

Figure 1.6 Graph of Number of Primitive Operations of Radix Sort Numbers and Strings Against the Product of Input Size and Maximum Key Length (n·k)

To consolidate both dimensions *n* and *k* into a single linear relationship, we also plot the number of primitive operations against the product of the number of inputs (*n*) and the maximum key length (*k*), which represents the total work expected from radix sort. As seen from *Figure 1.6*, the points lie almost perfectly along a straight line, confirming that the algorithm's complexity grows linearly with *n·k*. The regression line equations, `ops=22.63·(n·k)+906` and `ops=22.62·(n·k)+1570` show the fitted line has slope ≈ 22.6, indicating about 22.6 primitive operations per unit of *n·k* and intercepts ≈ 906 and 1568, which are negligible for large inputs. This validates *O(n·k)* by showing a small constant overhead and a consistent growth rate.

Therefore, we can deduce that the big-Oh notation of the radix sort algorithm is $O(n \cdot k)$, where n is the number of inputs and k is the maximum number of digits (for numbers) or the maximum string length (for strings) (Watson, 2022).

Files and Source Codes

This assignment is submitted in a zipped folder that contains the following files:

- RadixSort.java
- RadixSortWords.java
- RadixSort_Analysis.java
- RadixSortWords_Analysis.java
- RadixSort_Analysis.csv
- RadixSortWords_Analysis.csv
- Question 3 Graph.ipynb
- Graph of Operations Against Input Size.html
- README.md
- CPT212 Assignment 1.pdf

The source code of the programs are uploaded into a public GitHub repository and can be accessed via the following link:

https://github.com/ArwenLaung/CPT212-Assignment1.git

References

Chiarulli, C. (n.d.). *Primitive operations*. Chris@Machine. https://www.chiarulli.me/DSAndAlgos/02-Primitive_Operations/

Rastogi, A. (2023, May 8). *How to use radix sort to sort strings*. *Medium*. https://medium.com/@akshat28vivek/sorting-strings-with-radix-sort-an-overview-2e82436da586

Watson, W. (2022, January 6). Putting the rad in radix sort - nerd for tech - medium. Medium. https://medium.com/nerd-for-tech/putting-the-rad-in-radix-sort-d7c3be4fdbdf