CCS592 Advanced Algorithms and Complexities

Assignment 1

Report

Group member:

Name	**No USM**
Chan Khai Shen	24101339

Table of Contents

1. Number sorting (Question 1) 5

1.1 Asking user to specify the fixed length of the numbers 5

1.2 Asking user to input the list of numbers 6

1.3 Array initialization 7

1.4 First pass 7

1.4.1 Implementation logic 8

1.4.2 Reorder (only if the numbers have 1 digit) 9

1.5 Implementation of subsequent passes using while loop 10

1.6 Second pass and all even number passes 11

1.7 Third pass and all odd number passes except the first pass 13

1.8 Reorder 14

1.9 Running the programme 15

2. Word sorting (Question 2) 16

2.1 Asking user to specify the maximum length of the words 16

2.2 Asking user to input the list of words 17

2.3 Array initialization 19

2.4 First pass 20

2.4.1 Implementation logic 21

2.4.2 Reorder (only if the maximum length of words is 1 character) 22

2.5 Implementation of subsequent passes using while loop 23

2.6 Second pass and all even number passes 24

2.7 Third pass and all odd number passes except the first pass 27

2.8 Reorder 29

2.9 Running the programme 30
3. Complexity analysis on the number sorting algorithm 31
3.1 Counting number of primitive operations 31
3.1.1 Array initialization 31
3.1.2 First pass 32
3.1.3 While loop for subsequent passes 35
3.1.4 Second pass and all even number passes 36
3.1.5 Third pass and all odd number passes except the first pass 38
3.1.6 Reorder 40
3.2 Generating random unsorted list of numbers 41
3.3 Conducting the experiment 42
3.4 Displaying the experiment result and plotting the graph 45
3.5 Running the programme 46
3.6 Determining time complexity based on the experiment result 47
4. Complexity analysis on the word sorting algorithm 49
4.1 Counting the number of primitive operations 49
4.1.1 Array initialization 49
4.1.2 First pass 50
4.1.3 While loop for subsequent passes 54
4.1.4 Second pass and all even number passes 55
4.1.5 Third pass and all odd number passes except the first pass 59
4.1.6 Reorder 63
4.2 Generating random unsorted list of words with random length 65
4.3 Conducting the experiment 67

4.4 Displaying the experiment result and plotting the graph 70

4.5 Running the programme 71

4.6 Determining time complexity based on the experiment result..... 71

1. Number sorting (Question 1)

This section is about the number sorting algorithm in question 1. The programme in this part is a Python programme that takes a list of positive integers of the same length and sorts the number using the algorithm specified in the question paper.

1.1 Asking user to specify the fixed length of the numbers

In the programme, the length of all numbers in the list must be fixed and the user is required to specify the length of the numbers beforehand. In the first step, the user is asked to input the length. Figure 1.1 shows the code for asking the user to specify a fixed length of the input numbers.

```python

# Asks the fixed length of all numbers (which will be used to check against the numbers inputted by the user)
# Note: All numbers in the list should be of the same length
while True:
    try:
        # Checks whether the length inputted is an integer
        LENGTH = int(input('Enter the fixed length of all numbers: '))

        # Checks whether the length inputted is smaller than 1
        if LENGTH < 1:
            print('The length must be at least 1.')
            continue

        break

    except:
        print('The length must be an integer.')
        continue
```

Figure 1.1 Ask user to specify the length of input number

As shown in Figure 1.1, the programme checks whether the user input is an integer by trying to convert the input string to an integer. If the conversion of the input string to an integer throws an error, then the user input is not an integer. The “except” block in the “try … except” statement catches the error and prints an error message to the user. After that, the programmes checks whether the integer is larger or equal to 1 by checking whether the integer is smaller than 1. If the integer is smaller than 1, then an error message will be printed. If any error message is printed, then the user will be asked to input again until he inputs an integer which is larger or equal to 1.

1.2 Asking user to input the list of numbers

In the next step, the user is asked to input every number in the list, one by one. Figure 1.2 shows to code for asking the user to input every number in the list.

```python
# Initialize an array to hold the numbers inputted by the user
numbers = []

# Asks the user to input the list of numbers to be sorted
# Note: All numbers must be positive integers with the fixed length as specified in the above input
while True:
    userInput = input('Enter "finish" if the list is finish or enter the next number in the list: ')

    # Ends the while loop if the list of numbers is finish
    if userInput.strip().lower() == 'finish':
        break

    # Checks whether the length of the input equals to the specified length
    if len(userInput) != LENGTH:
        print(f'The length of the input must be equal to the fixed length specified ({LENGTH}).')
        continue

    # Checks whether the input only consists of number
    onlyContainNumber = True

    for position in range(len(userInput)):
        if userInput[position] not in ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']:
            print('The input must only contain number.')
            onlyContainNumber = False
            break

    if not onlyContainNumber:
        continue

    numbers.append(userInput)
    print('The number is accepted.')

# Prints out the initial list of numbers (unsorted)
print('The list of numbers accepted: ')
for number in numbers:
    print(number)
```

Figure 1.2 Ask user to input the list of numbers

As shown in Figure 1.2, an array is initialized and every number inputted by the user is appended to the array. The user is asked to either input a number or input “finish” to indicate that the list is finish. If the user inputs “finish”, then the while loop will be ended. Else, the number will be checked against a few conditions. Firstly, the programme checks whether the input is of the fixed length specified beforehand. If the length of the input does not equal to the fixed length specified beforehand, then an error message will be printed. Secondly, the programme checks whether the input only contain numbers. If there is a non-number (not “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8” or “9”) occurred in the input string, then an error message will be printed. If the input string does not trigger any error, then the number will be accepted and appended to the list of numbers. In this case, the programme will print a message to notify that the

number inputted is accepted. After that, the user will be asked to input the next number. Th process described above will be repeated until the user inputs “finish”. After the list of number is finish, the programme will print out the initial, unsorted list of numbers in the sequence as inputted by the user.

Throughout the whole programme, all numbers are maintained in string form because this allows the programme to read the digit at specific location.

1.3 Array initialization

After that, the programme will sort the list of numbers in ascending order according to the algorithm described in the question paper. In the first step of the algorithm, two array is initialized. Figure 1.3 shows the code for initializing the arrays.

```

# Number sorting algorithm start

# Initialize start
# Initialize 2 arrays with 10 elements (which is also an array)
array1 = [[], [], [], [], [], [], [], [], [] ,[]]
array2 = [[], [], [], [], [], [], [], [], [], []]
# Initialize end

# Iteration start

```

Figure 1.3 Initialize array 1 and array 2

As shown in Figure 1.3, two arrays, namely array 1 and array 2, are initialized. These two arrays are identical. In these arrays, there are 10 elements, representing cell 0, cell 1, cell 2, cell 3, cell 4, cell 5, cell 6, cell 7, cell 8 and cell 9 in the description in the question paper respectively. In each element, there is an empty array which will be used to hold the numbers which are placed at that cell. Since there might be more than one number placed at one cell, so an array is a suitable data structure to hold the numbers in the sequence of the numbers are placed.

1.4 First pass

After array initialization, the next step is parsing the list of numbers in a few iterations to sort the numbers. The number of iterations is equal to the length of the

number. The sorting starts from the last digit in the numbers. Figure 1.4 shows the code for implementing the first pass (first iteration) of the sorting process.

```python

# Iteration start

# For the first pass, need to look at the last digit (position of length - 1)
position = LENGTH - 1

# First pass start
# Pass through all number in the list of numbers inputted
for number in numbers:
    # Append the number to the corresponding cell in array 1 based on the last digit of the number
    if number[position] == '0':
        array1[0].append(number)
    elif number[position] == '1':
        array1[1].append(number)
    elif number[position] == '2':
        array1[2].append(number)
    elif number[position] == '3':
        array1[3].append(number)
    elif number[position] == '4':
        array1[4].append(number)
    elif number[position] == '5':
        array1[5].append(number)
    elif number[position] == '6':
        array1[6].append(number)
    elif number[position] == '7':
        array1[7].append(number)
    elif number[position] == '8':
        array1[8].append(number)
    else:
        array1[9].append(number)
# First pass end

```

Figure 1.4 First pass

As shown in Figure 1.4, in the first pass, the position is initialized as the last digit of each number, which is “length – 1”. The for loop goes through the initial list of numbers for once. The numbers are passed in the sequence as they are inputted by the user. The programme reads from the array “numbers” which is the array that holds the list of numbers from user input.

1.4.1 Implementation logic

Every number is placed at the corresponding cell in array 1 according to its last digit. The logic is if the last digit is “0”, then the number is appended to the array in cell 0, else if the last digit is “1”, then the number is appended to the array in cell 1, else if the last digit is “2”, then the number is appended to the array in cell 2, else if the last digit is “3”, then the number is appended to the array in cell 3, else if the last digit is “4”, then the number is appended to the array in cell 4, else if the last digit is “5”, then

the number is appended to the array in cell 5, else if the last digit is “6”, then the number is appended to the array in cell 6, else if the last digit is “7”, then the number is appended to the array in cell 7, else if the last digit is “8”, then the number is appended to the array in cell 8, else the number is appended to the array in cell 9. If more than one numbers are appended to a single cell, then the sequence of the numbers in the array is according to the sequence of the number being passed.

1.4.2 Reorder (only if the numbers have 1 digit)

After the first pass, if the number has only one digit, then the number will be reordered and printed in the sorted sequence. Figure 1.5 shows the code for implementing reorder after the first pass.

```python

# Reorder the numbers in a sorted list after the first pass if the numbers has only 1 digit (last position is 0)
if position == 0:

    # Reorder start (if length == 1)
    sortedNumbers = []

    # The list of numbers is already sorted in ascending order by reading from the first cell to the last cell in
    # array 1
    for cellNumber in range(10):
        for number in array1[cellNumber]:
            sortedNumbers.append(number)
    # Reorder end (if length == 1)

    # Prints out the sorted list and then end
    print('The sorted list of numbers: ')
    for number in sortedNumbers:
        print(number)

else:
    # Continue for subsequent pass

```

Figure 1.5 Reorder after first pass

As shown in Figure 1.5, if the numbers only have one digit (the position of last digit is 0), then the list will be reordered and the numbers will be printed in sorted sequence. The numbers can be reordered in ascending order by reading the numbers in the sequence from cell 0 to cell 9 in array 1. The reordered, sorted list is held in an array named sortedNumbers and the list is printed to the user. If the numbers only have one digit, then the programme will end here, else if the number have more than one digit, then the numbers will not be reordered yet and the programme will run for the second pass.

1.5 Implementation of subsequent passes using while loop

Before each subsequent pass, the number of passes is incremented by 1 and the position is decremented by 1. Figure 1.6 shows the code for the incrementing number of passes and decrementing position.

```
else:
    # Continue for subsequent pass

    # Holds the number of passes for future reference
    numberOfPasses = 1

    while position > 0:
        # Increment number of passes (go to the next pass) and decrement position (move one position forward)
        numberOfPasses = numberOfPasses + 1
        position = position - 1
```

Figure 1.6 Increment number of passes and decrement position

As shown in Figure 1.6, before the second pass, the variable “numberOfPasses” is created and initiated as 1 because currently only the first pass is completed. While the position is larger than 0, the while loop, which runs the subsequent passes, will be run repeatedly. The position will be decremented in every iteration because in each subsequent pass, the position of the scanned digit is moved one digit to the front. The while loop will continue to run until the position becomes 0 which indicates the first digit. This follows the description in the question paper where the digits in each number are scanned one by one starting from the last digit to the first digit.

On the other hand, the number of passes is incremented before each subsequent pass. The code in the while loop is split into two parts. The number of passes is important because it is used to determine in the current pass whether the first part or the second part will be run. If the number of passes is divisible by 2 (the remainder after divided by 2 is equal to 0), then the first part of the code will be run, otherwise the second part of the code will be run. In other words, for passes that are divisible by 2, which are second pass, 4th pass, 6th pass, 8th pass and so on, the first part of the code will be run, whereas for the passes that are not divisible by 2, which are third pass, 5th pass, 7th pass, 9th pass and so on, the second part of the code will be run.

This is because in the first pass, the numbers are written to array 1, so in the second pass, the programme will read from array 1 and write to array 2. After that, in the third pass, array 1 will be re-initialized and the programme will read from array 2 and write to array 1. After that, in the fourth pass, array 2 will be re-initialized and the programme will read from array 1 and write to array 2. After that, in the fifth pass, array 1 will be re-initialized and the programme will read from array 2 and write to array 1.

It is observed that the passes can be grouped into 2 groups. The first group of passes, which is even number passes, always reads from array 1 and writes to array 2; whereas the second group of passes, which is odd number passes except the first pass, always reads from array 2 and writes to array 1. So, the code is split into two parts and the divisibility by 2 of the number of passes will determine which part of the code is run.

1.6 Second pass and all even number passes

In the second pass and even number passes, the programme re-initializes array 2 and reads from array 1 and writes to array 2. Figure 1.7 shows the code for running the second pass and all even number passes.

```python
        # For the second pass (or 4th pass, 6th pass, 8th pass, ...)
        # For all passes that read from array 1 and write to array 2
        if numberOfPasses % 2 == 0:
            # Re-initialize array 2 because array 2 should be cleared because this pass should write to array 2
            array2 = [[], [], [], [], [], [], [], [], [], []]

            # Second pass (or 4th pass, 6th pass, 8th pass, ...) start
            # Reads from the first cell to the last cell in array 1
            for cellNumber in range(10):
                # Reads every number in each cell in array 1
                for number in array1[cellNumber]:
                    # Appends the number to corresponding cell in array 2 based on the second last (or 4th last,
                    # 6th last, 8th last, ...) digit of the number
                    if number[position] == '0':
                        array2[0].append(number)
                    elif number[position] == '1':
                        array2[1].append(number)
                    elif number[position] == '2':
                        array2[2].append(number)
                    elif number[position] == '3':
                        array2[3].append(number)
                    elif number[position] == '4':
                        array2[4].append(number)
                    elif number[position] == '5':
                        array2[5].append(number)
                    elif number[position] == '6':
                        array2[6].append(number)
                    elif number[position] == '7':
                        array2[7].append(number)
                    elif number[position] == '8':
                        array2[8].append(number)
                    else:
                        array2[9].append(number)
            # Second pass (or 4th pass, 6th pass, 8th pass, ...) end
```

Figure 1.7 Second pass and all even number passes

As shown in Figure 1.7, the programme identifies the second pass and all even number passes by checking whether the remainder of the number of passes divide by 2 is equal to 0. If the remainder is equal to 0, then the current pass is an even number pass.

As shown in Figure 1.7, before the operation starts, array 2 is re-initialized to an array with 10 cells and each cell contains an empty array. This is because the previous pass writes to array 1 and the current pass reads from array 1. So, the information in array 2 is not read in the current pass, instead array 2 should be empty for the current pass to write to array 2.

After that, as shown in Figure 1.7, the numbers are read from array 1, starting from cell 0 to cell 9. In each cell, the numbers are read according to the sequence of the number in the array in each cell. Then, the number is placed at the corresponding cell in array 2 according to the digit at the current position. The logic is same as the logic used in the first pass which is described in section 1.4.1. If there are more than 1 number in a cell, then the numbers are appended to the array in the cell according

to the sequence of the numbers are read. This follows the description of the algorithm in the question paper.

1.7 Third pass and all odd number passes except the first pass

In the third pass and odd number passes except the first pass, the programme re-initializes array 1 and reads from array 2 and writes to array 1. Figure 1.8 shows the code for running the third pass and odd number passes except the first pass.

```
        # For the third pass (or 5th pass, 7th pass, 9th pass, ...)
        # For all passes that read from array 2 and write to array 1
        else:
            # Re-initialize array 1 because array 1 should be cleared because this pass should write to array 1
            array1 = [[], [], [], [], [], [], [], [], [], []]

            # Third pass (or 5th pass, 7th pass, 9th pass, ...) start
            # Reads from the first cell to the last cell in array 2
            for cellNumber in range(10):
                # Reads every number in each cell in array 2
                for number in array2[cellNumber]:
                    # Appends the number to the corresponding cell in array 1 based on the third last (or 5th last,
                    # 7th last, 9th last, ...) digit of the number
                    if number[position] == '0':
                        array1[0].append(number)
                    elif number[position] == '1':
                        array1[1].append(number)
                    elif number[position] == '2':
                        array1[2].append(number)
                    elif number[position] == '3':
                        array1[3].append(number)
                    elif number[position] == '4':
                        array1[4].append(number)
                    elif number[position] == '5':
                        array1[5].append(number)
                    elif number[position] == '6':
                        array1[6].append(number)
                    elif number[position] == '7':
                        array1[7].append(number)
                    elif number[position] == '8':
                        array1[8].append(number)
                    else:
                        array1[9].append(number)
            # Third pass (or 5th pass, 7th pass, 9th pass, ...) end

    # Iteration end

```

Figure 1.8 Third pass and all odd number passes except the first pass

As shown in Figure 1.8, before the operation starts, array 1 is re-initialized to an array with 10 cells and each cell contains an empty array. This is because the previous pass writes to array 2 and the current pass reads from array 2. So, the information in array 1 is not read in the current pass, instead array 1 should be empty for the current pass to write to array 1.

After that, as shown in Figure 1.8, the numbers are read from array 2, starting from cell 0 to cell 9. In each cell, the numbers are read according to the sequence of the number in the array in each cell. Then, the number is placed at the corresponding cell in array 1 according to the digit at the current position. The logic is same as the logic used in the first pass which is described in section 1.4.1. If there is more than 1 number in a cell, the number are appended to the array in the cell according to the sequence of the number are read. This follows the description of the algorithm in the question paper.

1.8 Reorder

The while loop will end when the “position” variable decrements until 0, which indicates that the position of the previously scanned digit reaches 0, which is the first digit. After the while loop is ended, the list of numbers will be reordered into a sorted list and displayed to the user. Figure 1.9 shows the code for reordering the list of numbers from either array 1 or array 2, depending on the “numberOfPasses” variable.

```python

    # Reorder start
    sortedNumbers = []

    # After the second pass (or 4th pass, 6th pass, 8th pass, ...)
    # After all passes that write to array 2
    if numberOfPasses % 2 == 0:
        # The list of numbers is already sorted in ascending order by reading from the first cell to the last cell
        # in array 2
        for cellNumber in range(10):
            for number in array2[cellNumber]:
                sortedNumbers.append(number)

    # After the third pass (or 5th pass, 7th pass, 9th pass, ...)
    # After all passes that write to array 1 (except the first pass)
    else:
        # The list of numbers is already sorted in ascending order by reading from the first cell to the last cell
        # in array 1
        for cellNumber in range(10):
            for number in array1[cellNumber]:
                sortedNumbers.append(number)
    # Reorder end

    # Prints out the sorted list and then end
    print('The sorted list of numbers: ')
    for number in sortedNumbers:
        print(number)

# Number sorting algorithm end

```

Figure 1.9 Reorder

As shown in Figure 1.9, during reorder, if the number of passes is divisible by 2 (which means the last pass is an even number pass), then the programme will read from array 2. Otherwise, the last pass is an odd number pass, so the programme will read from array 1. This is because in an even number pass, the programme writes to array 2, whereas in an odd number pass, the programme writes to array 1.

As shown in Figure 1.9, firstly, an array named “sortedNumbers” is initialized. This array will hold the sorted list of numbers. Every cell in the array (array 1 or array 2) is read, starting from cell 0 to cell 9. In each cell, every number is read according to the sequence of the number in the array in that cell. The numbers are appended to “sortedNumbers” array in the same sequence as the number is read. This produces a list of numbers which is sorted in ascending order. This follows the description of the algorithm in the question paper. Finally, the sorted list of numbers will be printed to the user.

1.9 Running the programme

Figure 1.10 shows the output of the programme running for the input of six 3-digit numbers, 988, 546, 786, 332, 501 and 765. The list of unsorted numbers is finally able to be sorted in ascending order by the programme.

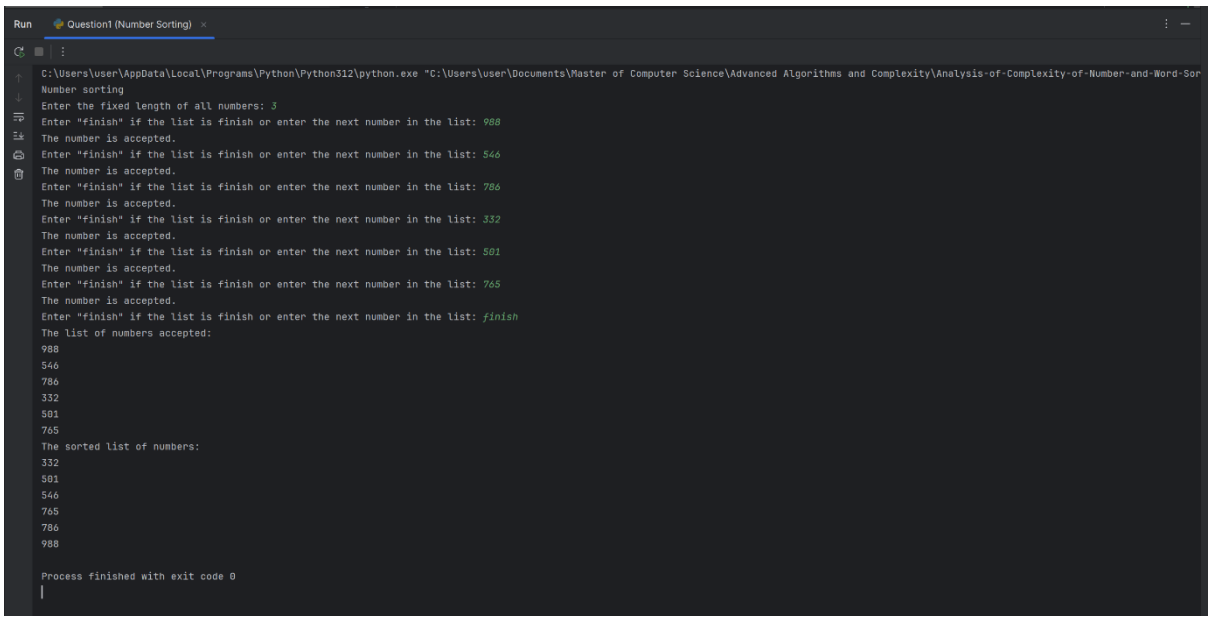

Figure 1.10 Running the programme

2. Word sorting (Question 2)

This section is about the word sorting algorithm in Question 2. The programme in this part is a Python programme that takes a list of words and sorts the list of words using a modified version of the algorithm described in the question paper. Every word should contain lower case characters only. Each word can have different lengths, but cannot exceed a maximum length which is specified beforehand by the user. The length of each word must be at least 1.

2.1 Asking user to specify the maximum length of the words

In the programme, the user is allowed to input words of different lengths. However, the user is asked to input a maximum length of the words beforehand. This maximum length is used later as a reference to determine the position of the last character of the longest word because the position of last character of the longest word is equal to “maximum length – 1”. This does not affect the ability to input words of different lengths because the user is free to input words with any length below the maximum length specified.

In the first step, the user is asked to input the maximum length of the words. Figure 2.1 shows the code for asking the user to specify the maximum length of words.

```python

print('Word sorting')

# Asks the maximum length of all words (which will be used to check against the words inputted by the user)
# Note: All words in the list should be shorter than or equal to the maximum length and longer than or equal to 1
while True:
    try:
        # Checks whether the length inputted is an integer
        MAXIMUM_LENGTH = int(input('Enter the maximum length of all words: '))

        # Checks whether the length inputted is smaller than 1
        if MAXIMUM_LENGTH < 1:
            print('The maximum length must be at least 1.')
            continue

        break

    except:
        print('The maximum length must be an integer.')
        continue

```

Figure 2.1 Ask user to specify maximum length of input words

As shown in Figure 2.1, the programme checks whether the maximum length inputted by the user is an integer by trying to convert the input string to an integer. If the conversion of the input string to an integer throws an error, then the user input is not an integer. The “except” block in the “try … except” statement catches the error and prints an error message to the user. After that, the programme checks whether the integer is larger or equal to 1 by checking whether the integer is smaller than 1. If the integer is smaller than 1, then an error message will be printed. If any error message is printed, then the user will be asked to input again until he inputs an integer which is larger or equal to 1.

2.2 Asking user to input the list of words

In the next step, the user is asked to insert a list of words, one by one. Figure 2.2 shows the code for asking the user to insert the unsorted list of words.

```python

# Initialize an array to hold the words inputted by the user
words = []

# Asks the user to input the list of words to be sorted
# Note: All words must consist of lower case characters only and of the length shorter than or equal to the
# maximum length as specified in the above input and longer than or equal to 1
while True:
    userInput = input('Enter "0" if the list is finish or enter the next word in the list: ')

    # Ends the while loop if the list of words is finish
    if userInput.strip() == '0':
        break

    # Checks whether the length of the input is shorter than or equal to the maximum length
    if len(userInput) > MAXIMUM_LENGTH:
        print(f'The length of the input must be shorter than or equal to to the maximum length ({MAXIMUM_LENGTH}).')
        continue

    # Checks whether the length of the input is longer than or equal to 1
    if len(userInput) < 1:
        print('The length of the input must be longer than or equal to 1.')
        continue

    # Checks whether the input only consists of lower case characters
    onlyContainLowerCharacter = True

    for position in range(len(userInput)):
        if userInput[position] not in ['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']:
            print('The input must only contain lower case characters.')
            onlyContainLowerCharacter = False
            break

    if not onlyContainLowerCharacter:
        continue

    words.append(userInput)
    print('The word is accepted.')

```

Figure 2.2 Ask user to input the list of words

As shown in Figure 2.2, an array named “words” is initialized and every number inputted by the user is appended to the array. The user is asked to either input a word or input “0” to indicate that the list is finish. If the user inputs “0”, then the while loop will be ended. Else, the word will be checked against a few conditions. Every word must have length which is shorter than or equal to the specified maximum length and longer than 1 and only contain lower case characters. Firstly, the programme checks whether the length of the input word is shorter than or equal to the maximum length specified beforehand. If the length of the input word is longer than the maximum length specified beforehand, then an error message will be printed. Secondly, the programme checks whether the length of the input word is longer than or equal to 1. If the length of the input word is shorter than 1, then an error message will be printed. Thirdly, the programme checks whether the input only contains lower case characters. If there is a non-lower-case-character (not “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” or “z”) occurred in the input string, then an error message will be printed. If the input string does not trigger any error, then the word will be accepted and appended to the list of words. In this case, the programme will print a message to notify that the word inputted is accepted. After that, the user will be asked to input the next word. The process described above will be repeated until the user inputs “0”. When user inputs “0”, the while loop is ended.

After the list of words is finish, the programme will print out the initial, unsorted list of words in the sequence as inputted by the user. Figure 2.3 shows the code for printing the initial, unsorted list of words.

```

# Prints out the initial list of words (unsorted)
print('The list of words accepted: ')
for word in words:
    print(word)

```

Figure 2.3 Print the initial, unsorted list of words

Throughout the whole programme, all words are maintained in string form because this allows the programme to read the character at a specific position.

2.3 Array initialization

In the programme, the list of words will be sorted by modifying the algorithm used to sort the list of numbers described in the question paper. Firstly, two arrays, namely array 1 and array 1, are initialized. Figure 2.4 shows the code for initializing array 1 and array 2.

```

# Word sorting algorithm start

# Initialize start
# Initialize 2 arrays with 27 elements (which is also an array)
array1 = [[], [], [], [], [], [], [], [], [] ,[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []]
array2 = [[], [], [], [], [], [], [], [], [], [], [] ,[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []]
# Initialize end

# Iteration start

```

Figure 2.4 Initialize array 1 and array 2

As shown in Figure 2.4, two arrays, namely array 1 and array 2, are initialized. These two arrays are identical. In these arrays, there are 27 elements. Cell 0 holds

the words that are shorter than the currently scanned position, whereas cell 1 to cell 26 represent “a” to “z” respectively. In each cell, there is an empty array which will be used to hold the words which are placed at that cell. There can be more than one words in one cell, so array is an appropriate data structure to hold the words in a cell.

2.4 First pass

After array initialization, the next step is parsing the list of the words in a few iterations to sort the list of words. The number of iterations is equal to the maximum length of the words. The sorting starts from the last character of the word. Figure 2.5 and Figure 2.6 show the code for implementing the first pass.

```python

# For the first pass, need to look at the last character (position of length - 1)
position = MAXIMUM_LENGTH - 1

# First pass start
# Pass through all words in the list of words inputted
for word in words:
    # Append the word to the corresponding cell in array 1 based on the last character of the word
    # If the word is shorter than the currently scanned position (length < position + 1), then append the word to
    # cell 0 (shorter strings are sorted before all longer strings)
    if len(word) < position + 1:
        array1[0].append(word)
    elif word[position] == 'a':
        array1[1].append(word)
    elif word[position] == 'b':
        array1[2].append(word)
    elif word[position] == 'c':
        array1[3].append(word)
    elif word[position] == 'd':
        array1[4].append(word)
    elif word[position] == 'e':
        array1[5].append(word)
    elif word[position] == 'f':
        array1[6].append(word)
    elif word[position] == 'g':
        array1[7].append(word)
    elif word[position] == 'h':
        array1[8].append(word)
    elif word[position] == 'i':
        array1[9].append(word)
    elif word[position] == 'j':
        array1[10].append(word)
    elif word[position] == 'k':
        array1[11].append(word)
    elif word[position] == 'l':
        array1[12].append(word)
```

Figure 2.5 First pass (part 1 of 2)

```
        elif word[position] == 'm':
            array1[13].append(word)
        elif word[position] == 'n':
            array1[14].append(word)
        elif word[position] == 'o':
            array1[15].append(word)
        elif word[position] == 'p':
            array1[16].append(word)
        elif word[position] == 'q':
            array1[17].append(word)
        elif word[position] == 'r':
            array1[18].append(word)
        elif word[position] == 's':
            array1[19].append(word)
        elif word[position] == 't':
            array1[20].append(word)
        elif word[position] == 'u':
            array1[21].append(word)
        elif word[position] == 'v':
            array1[22].append(word)
        elif word[position] == 'w':
            array1[23].append(word)
        elif word[position] == 'x':
            array1[24].append(word)
        elif word[position] == 'y':
            array1[25].append(word)
        else:
            array1[26].append(word)
    # First pass end
```

Figure 2.6 First pass (part 2 of 2)

As shown in Figure 2.5, in the first pass, the position is initialized as the last character of the longest word, which is “maximum_length – 1”. The for loop goes through the initial list of words for once. The words are passed in the sequence as they are inputted by the user. The programme reads from the array “words” which is the array that holds the list of words from user input.

2.4.1 Implementation logic

Every number is placed at the corresponding cell in array 1 according to its last digit. If the length of the word is shorter than the currently scanned position, then the word is placed at cell 0, otherwise the word is placed at cell 1 to cell 26. This is because in ordinary ascending order sorting, given two strings with the similar front part but different length, “for” and “fort”, the shorter string “for” will be sorted in front of the longer string “fort”. So, the word that is shorter than the currently scanned position is sorted at the first cell, which is in front of all other words. Table 2.1 shows how the word is allocated to cell 1 to cell 26 based on the character at the scanned position.

Table 2.1 Cell allocation for different characters scanned

Cell	Character scanned
0	Length is shorter than the currently scanned position
1	a
2	b
3	c
4	d
5	e
6	f
7	g
8	h
9	i
10	j
11	k
12	l
13	m
14	n
15	o
16	p
17	q
18	r
19	s
20	t
21	u
22	v
23	w
24	x
25	y
26	z

As shown in Table 2.1, the word is placed at the corresponding cell based on the character scanned. The order is based on ascending order of alphabet, starting from the first alphabet “a” to the last alphabet “z”. If more than one numbers are appended to a single cell, then the sequence of the words in the array is according to the sequence of the word being passed.

2.4.2 Reorder (only if the maximum length of words is 1 character)

If the maximum length of the words is 1, then after the first pass, the words will be reordered into a sorted list and printed out to the user. Figure 2.7 shows the code for reordering the words after the first pass.

```python

# Reorder the words in a sorted list after the first pass if the words has only 1 character (last position is 0)
if position == 0:

    # Reorder start (if length == 1)
    sortedWords = []

    # The list of words is already sorted in ascending order by reading from the first cell to the last cell in
    # array 1
    for cellNumber in range(27):
        for word in array1[cellNumber]:
            sortedWords.append(word)
    # Reorder end (if length == 1)

    # Prints out the sorted list and then end
    print('The sorted list of words: ')
    for word in sortedWords:
        print(word)

else:
    # Continue for subsequent pass

```

Figure 2.7 Reorder after first pass

As shown in Figure 2.7, if the maximum length of the words is 1 (the position of last character is 0), then the list will be reordered and the words will be printed in sorted sequence. The words can be reordered in ascending order by reading the words in the sequence from cell 0 to cell 26 in array 1. The reordered, sorted list is held in an array named “sortedWords” and the list is printed to the user. If the maximum length of the words is 1, then the programme will end here, else if the maximum length of the words is more than 1, then the words will not be reordered yet and the programme will run for the second pass.

2.5 Implementation of subsequent passes using while loop

Before each subsequent passes, the number of passes is incremented by 1 and the position is decremented by 1.

```python

else:
    # Continue for subsequent pass

    # Holds the number of passes for future reference
    numberOfPasses = 1

    while position > 0:
        # Increment number of passes (go to the next pass) and decrement position (move one position forward)
        numberOfPasses = numberOfPasses + 1
        position = position - 1

```

Figure 2.8 Increment number of passes and decrement position

As shown in Figure 2.8, before the second pass, the variable “numberOfPasses” is created and initiated as 1 because currently only the first pass is completed. While the position is larger than 0, the while loop, which runs the subsequent passes, will be run repeatedly. The position will be decremented in every iteration because in each subsequent pass, the position of the scanned character is moved one digit to the front. The while loop will continue to run until the position becomes 0 which indicates the first character. This is similar to the number sorting algorithm because the characters in each word are scanned one by one starting from the last character to the first character.

On the other hand, the number of passes is incremented before each subsequent pass. The code in the while loop is split into two parts. The number of passes is important because it is used to determine in the current pass whether the first part or the second part will be run. As explained in section 1.5, all passes can be grouped into 2 groups. The first group of passes, which is even number passes (divisible by 2), always reads from array 1 and writes to array 2; whereas the second group of passes, which is odd number passes except the first pass (not divisible by 2), always reads from array 2 and writes to array 1. So, the code is split into two parts and the divisibility by 2 of the number of passes will determine which part of the code is run.

2.6 Second pass and all even number passes

In the second pass and all even number passes, array 2 is re-initialized and the programme reads from array 1 and writes to array 2. Figure 2.9 and Figure 2.10 show the code for implementing the second pass and all even number passes.

```python

        # For the second pass (or 4th pass, 6th pass, 8th pass, ...)
        # For all passes that read from array 1 and write to array 2
        if numberOfPasses % 2 == 0:
            # Re-initialize array 2 because array 2 should be cleared because this pass should write to array 2
            array2 = [[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [] ,[], [], [],
                      [], [], []]

            # Second pass (or 4th pass, 6th pass, 8th pass, ...) start
            # Reads from the first cell to the last cell in array 1
            for cellNumber in range(27):
                # Reads every word in each cell in array 1
                for word in array1[cellNumber]:
                    # Appends the number to corresponding cell in array 2 based on the second last (or 4th last,
                    # 6th last, 8th last, ...) character of the word
                    # If the word is shorter than the currently scanned position (length < position + 1), then append
                    # the word to cell 0 (shorter strings are sorted before all longer strings)
                    if len(word) < position + 1:
                        array2[0].append(word)
                    elif word[position] == 'a':
                        array2[1].append(word)
                    elif word[position] == 'b':
                        array2[2].append(word)
                    elif word[position] == 'c':
                        array2[3].append(word)
                    elif word[position] == 'd':
                        array2[4].append(word)
                    elif word[position] == 'e':
                        array2[5].append(word)
                    elif word[position] == 'f':
                        array2[6].append(word)
                    elif word[position] == 'g':
                        array2[7].append(word)
                    elif word[position] == 'h':
                        array2[8].append(word)
                    elif word[position] == 'i':
                        array2[9].append(word)
                    elif word[position] == 'j':
                        array2[10].append(word)
```

Figure 2.9 Second pass and all even number passes (part 1of 2)

```
                elif word[position] == 'k':
                    array2[11].append(word)
                elif word[position] == 'l':
                    array2[12].append(word)
                elif word[position] == 'm':
                    array2[13].append(word)
                elif word[position] == 'n':
                    array2[14].append(word)
                elif word[position] == 'o':
                    array2[15].append(word)
                elif word[position] == 'p':
                    array2[16].append(word)
                elif word[position] == 'q':
                    array2[17].append(word)
                elif word[position] == 'r':
                    array2[18].append(word)
                elif word[position] == 's':
                    array2[19].append(word)
                elif word[position] == 't':
                    array2[20].append(word)
                elif word[position] == 'u':
                    array2[21].append(word)
                elif word[position] == 'v':
                    array2[22].append(word)
                elif word[position] == 'w':
                    array2[23].append(word)
                elif word[position] == 'x':
                    array2[24].append(word)
                elif word[position] == 'y':
                    array2[25].append(word)
                else:
                    array2[26].append(word)
            # Second pass (or 4th pass, 6th pass, 8th pass, ...) end
```

Figure 2.10 Second pass and all even number passes (part 2 of 2)

As shown in Figure 2.9, the programme identifies the second pass and all even number passes by checking whether the remainder of the number of passes divide by 2 is equal to 0. If the remainder is equal to 0, then the current pass is an even number pass.

As shown in Figure 2.9, before the operation starts, array 2 is re-initialized to an array with 27 cells and each cell contains an empty array. This is because the previous pass writes to array 1 and the current pass reads from array 1. So, the information in array 2 is not read in the current pass, instead array 2 should be empty for the current pass to write to array 2.

After that, as shown in Figure 2.9 and Figure 2.10, the words are read from array 1, starting from cell 0 to cell 26. In each cell, every word is read according to the sequence of the word in the array in each cell. Then, the word is placed at the corresponding cell in array 2 according to the character at the current position. If the length of the word is shorter than the current position, then the word is placed at cell 0. The logic is same as the logic used in the first pass which is described in section

2.4.1. If there are more than 1 word in a cell, then the words are appended to the array in the cell according to the sequence of the words are read. This is similar to the number sorting algorithm.

2.7 Third pass and all odd number passes except the first pass

In the third pass and all odd number passes except the first pass, array 1 is re-initialized and the programme reads from array 2 and writes to array 1. Figure 2.11 and Figure 2.12 show the code for the third pass and all odd number passes except the first pass.

```python

        # For the third pass (or 5th pass, 7th pass, 9th pass, ...)
        # For all passes that read from array 2 and write to array 1
        else:
            # Re-initialize array 1 because array 1 should be cleared because this pass should write to array 1
            array1 = [[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [],
                      [], [], []]

            # Third pass (or 5th pass, 7th pass, 9th pass, ...) start
            # Reads from the first cell to the last cell in array 2
            for cellNumber in range(27):
                # Reads every word in each cell in array 2
                for word in array2[cellNumber]:
                    # Appends the word to the corresponding cell in array 1 based on the third last (or 5th last,
                    # 7th last, 9th last, ...) character of the word
                    # If the word is shorter than the currently scanned position (length < position + 1), then append
                    # the word to cell 0 (shorter strings are sorted before all longer strings)
                    if len(word) < position + 1:
                        array1[0].append(word)
                    elif word[position] == 'a':
                        array1[1].append(word)
                    elif word[position] == 'b':
                        array1[2].append(word)
                    elif word[position] == 'c':
                        array1[3].append(word)
                    elif word[position] == 'd':
                        array1[4].append(word)
                    elif word[position] == 'e':
                        array1[5].append(word)
                    elif word[position] == 'f':
                        array1[6].append(word)
                    elif word[position] == 'g':
                        array1[7].append(word)
                    elif word[position] == 'h':
                        array1[8].append(word)
                    elif word[position] == 'i':
                        array1[9].append(word)
                    elif word[position] == 'j':
                        array1[10].append(word)
```

Figure 2.11 Third pass and all odd number passes except the first pass (part 1of 2)

```
                    elif word[position] == 'k':
                        array1[11].append(word)
                    elif word[position] == 'l':
                        array1[12].append(word)
                    elif word[position] == 'm':
                        array1[13].append(word)
                    elif word[position] == 'n':
                        array1[14].append(word)
                    elif word[position] == 'o':
                        array1[15].append(word)
                    elif word[position] == 'p':
                        array1[16].append(word)
                    elif word[position] == 'q':
                        array1[17].append(word)
                    elif word[position] == 'r':
                        array1[18].append(word)
                    elif word[position] == 's':
                        array1[19].append(word)
                    elif word[position] == 't':
                        array1[20].append(word)
                    elif word[position] == 'u':
                        array1[21].append(word)
                    elif word[position] == 'v':
                        array1[22].append(word)
                    elif word[position] == 'w':
                        array1[23].append(word)
                    elif word[position] == 'x':
                        array1[24].append(word)
                    elif word[position] == 'y':
                        array1[25].append(word)
                    else:
                        array1[26].append(word)
                # Third pass (or 5th pass, 7th pass, 9th pass, ...) end

        # Iteration end

```

Figure 2.12 Third pass and all odd number passes except the first pass (part 2 of 2)

As shown in Figure 2.11, before the operation starts, array 1 is re-initialized to an array with 27 cells and each cell contains an empty array. This is because the previous pass writes to array 2 and the current pass reads from array 2. So, the information in array 1 is not read in the current pass, instead array 1 should be empty for the current pass to write to array 1.

After that, as shown in Figure 2.11 and Figure 2.12, the words are read from array 2, starting from cell 0 to cell 26. In each cell, every word is read according to the sequence of the word in the array in each cell. Then, the word is placed at the corresponding cell in array 1 according to the character at the current position. If the length of the word is shorter than the current position, then the word is placed at cell 0. The logic is same as the logic used in the first pass which is described in section 2.4.1. If there are more than 1 word in a cell, then the words are appended to the array in the cell according to the sequence of the words are read. This is similar to the number sorting algorithm.

2.8 Reorder

The while loop will end when the “position” variable decrements until 0, which indicates that the position of the previously scanned character reaches 0, which is the first character. After the while loop is ended, the list of words will be reordered into a sorted list and displayed to the user. Figure 2.13 shows the code for reordering the list of words from either array 1 or array 2, depending on the “numberOfPasses” variable.

```
    # Reorder start
    sortedWords = []

    # After the second pass (or 4th pass, 6th pass, 8th pass, ...)
    # After all passes that write to array 2
    if numberOfPasses % 2 == 0:
        # The list of words is already sorted in ascending order by reading from the first cell to the last cell
        # in array 2
        for cellNumber in range(27):
            for word in array2[cellNumber]:
                sortedWords.append(word)

    # After the third pass (or 5th pass, 7th pass, 9th pass, ...)
    # After all passes that write to array 1 (except the first pass)
    else:
        # The list of words is already sorted in ascending order by reading from the first cell to the last cell
        # in array 1
        for cellNumber in range(27):
            for word in array1[cellNumber]:
                sortedWords.append(word)
    # Reorder end

    # Prints out the sorted list and then end
    print('The sorted list of numbers: ')
    for word in sortedWords:
        print(word)

# Word sorting algorithm end
```

Figure 2.13 Reorder

As shown in Figure 2.13, during reorder, if the number of passes is divisible by 2 (which means the last pass is an even number pass), then the programme will read from array 2. Otherwise, the last pass is an odd number pass, so the programme will read from array 1. This is because in an even number pass, the programme writes to array 2, whereas in an odd number pass, the programme writes to array 1.

As shown in Figure 2.13, firstly, an array named “sortedWords” is initialized. This array will hold the sorted list of words. Every cell in the array (array 1 or array 2) is read, starting from cell 0 to cell 26. In each cell, every word is read according to the sequence of the number in the array in that cell. The words are appended to

“sortedWords” array in the same sequence as the word is read. This produces a list of words which is sorted in ascending order. This is similar to the number sorting algorithm. Finally, the sorted list of words will be printed to the user.

2.9 Running the programme

Figure 1.10 shows the output of the programme running for the input of seven words with different lengths with maximum length of 3, ‘pot’, ‘lk’, ‘p’, ‘ifh’, ‘po’, ‘mx’ and ‘s’. The list of unsorted words is finally able to be sorted in ascending order by the programme.

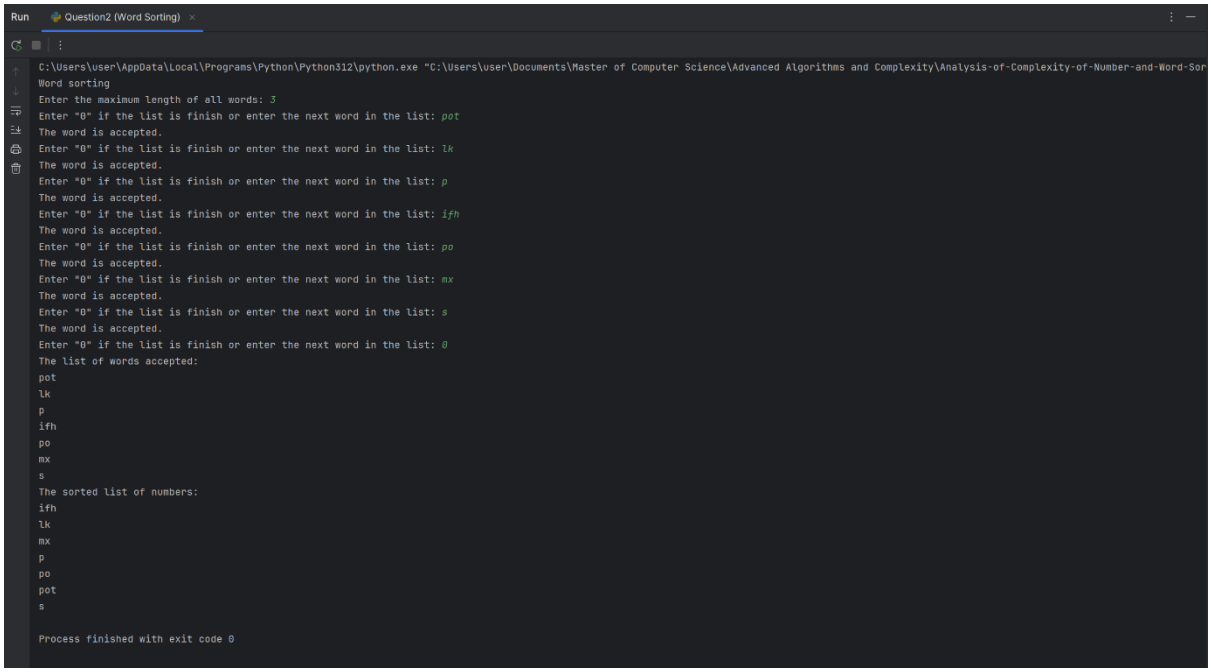

Figure 2.14 Running the programme

3. Complexity analysis on the number sorting algorithm

This section is about the time complexity analysis on the number sorting algorithm implemented in section 1. Counters are added to the number sorting algorithm to count the number of primitive operations. The number sorting algorithm is then run on different number of numbers and the number of primitive operations for each trial is recorded. Finally, a graph of number of primitive operations against number of numbers (N) is plotted to determine the time complexity of the number sorting algorithm.

3.1 Counting number of primitive operations

The code for implementing the number sorting algorithm is modified into a function that takes a list of unsorted numbers and the fixed length (number of digits) of the numbers as input and returns the number of primitive operations required to sort the list of numbers in ascending order. Compared to the code in section 1, the code in this section does not ask user to input the list of numbers and the fixed length (number of digits) of the numbers. Instead, the list of numbers and the length (number of digits) are passed as arguments of the function. The function will implement the number sorting algorithm exactly like what the code in section 1 does. Besides that, another modification is, in the function, to count the number of primitive operations of the number sorting algorithm, primitive operation counters are added to various parts of the code.

3.1.1 Array initialization

Figure 3.1 shows the code for implementing the function counting the number of primitive operations while implementing the number sorting algorithm. Figure 3.1 also shows the primitive operation counter for array initialization.

```python

# Complexity analysis of number sorting algorithm

# Function to count number of primitive operations in number sorting
# Takes 2 input, namely numbers and LENGTH
# numbers is the unsorted list of positive integer with fixed length (an array of type string), LENGTH is the fixed
# length (number of digits) of each number
# Returns the number of primitive operations required to sort the list of numbers
def count_primitive_operation_in_number_sorting(numbers, LENGTH):  10 usages  new *
    # Initialize primitive operation counter
    numberOfPrimitiveOperations = 0

    # Number sorting algorithm start

    # Initialize start
    # Initialize 2 arrays with 10 elements (which is also an array)
    array1 = [[], [], [], [], [], [], [], [], [], []]
    array2 = [[], [], [], [], [], [], [], [], [], []]
    numberOfPrimitiveOperations += 2    # 2 array initialization operations
    # Initialize end

```

Figure 3.1 Primitive operation counter for array initialization

As shown in Figure 3.1, the function "count_primitive_operation_in_number_sorting()" is used to count the number of primitive operations to sort the list of numbers inputted using the number sorting algorithm described in section 1. Besides the unsorted list of numbers, the function also takes length, which is the number of digits of each number in the list, as input. This length is important because it is used by the number sorting algorithm to determine the last position of each number since the last position is equal to length-1.

As shown in Figure 3.1, the number of primitive operations is initialized as 0 at the beginning of the function. During array initialization, two arrays, namely array 1 and array 2, are initialized. Since array initialization involves assignment of variable and there are two arrays, so the number of primitive operations is incremented by 2.

3.1.2 First pass

After array initialization, the next step is implementing the first pass. Figure 3.2 and Figure 3.3 show the primitive operation counters for the first pass.

```python

    # Iteration start

    # For the first pass, need to look at the last digit (position of length - 1)
    position = LENGTH - 1
    numberOfPrimitiveOperations += 2    # 1 subtraction operation and 1 assignment operation

    # First pass start
    # Pass through all number in the list of numbers inputted
    for number in numbers:
        # Append the number to the corresponding cell in array 1 based on the last digit of the number
        if number[position] == '0':
            array1[0].append(number)
            numberOfPrimitiveOperations += 2    # 1 comparison operation and 1 assignment operation
        elif number[position] == '1':
            array1[1].append(number)
            numberOfPrimitiveOperations += 3    # 2 comparison operations and 1 assignment operation
        elif number[position] == '2':
            array1[2].append(number)
            numberOfPrimitiveOperations += 4    # 3 comparison operations and 1 assignment operation
        elif number[position] == '3':
            array1[3].append(number)
            numberOfPrimitiveOperations += 5    # 4 comparison operations and 1 assignment operation
        elif number[position] == '4':
            array1[4].append(number)
            numberOfPrimitiveOperations += 6    # 5 comparison operations and 1 assignment operation
        elif number[position] == '5':
            array1[5].append(number)
            numberOfPrimitiveOperations += 7    # 6 comparison operations and 1 assignment operation
        elif number[position] == '6':
            array1[6].append(number)
            numberOfPrimitiveOperations += 8    # 7 comparison operations and 1 assignment operation
        elif number[position] == '7':
            array1[7].append(number)
            numberOfPrimitiveOperations += 9    # 8 comparison operations and 1 assignment operation
```

Figure 3.2 Primitive operation counters for first pass (part 1 of 2)

```python
        elif number[position] == '8':
            array1[8].append(number)
            numberOfPrimitiveOperations += 10   # 9 comparison operations and 1 assignment operation
        else:
            array1[9].append(number)
            numberOfPrimitiveOperations += 10   # 9 comparison operations and 1 assignment operation
    # First pass end

```

Figure 3.3 Primitive operation counters for first pass (part 2 of 2)

Before the first pass starts, the position is initialized as length-1. This involves 2 primitive operations, which is 1 subtraction operation and 1 variable assignment operation. So, as shown in Figure 3.2, the number of primitive operations is incremented by 2.

Firstly, the digit at the scanned position is compared to ‘0’ and if the comparison returns true, then the number is appended to the array at cell 0. So, if the digit at the scanned position is 0, then the number of primitive operations involved is 2, which is 1 comparison operation and 1 variable assignment operation. If the digit at the scanned position is not 0, then the digit will be compared to ‘1’ and if the comparison

returns true, then the number is appended to the array at cell 1. This involves 3 primitive operations, which is 2 comparison operations (compared with ‘0’ and with ‘1’) and 1 variable assignment operation. If the digit at the scanned position is not 1, then the digit will be compared with ‘2’. The comparison will continue by comparing the digits with 0 to 9, one by one, until the comparison returns true. When the comparison returns true, the number is appended to the corresponding cell. So, it can be observed that, going from 0 to 8, the number of comparison operation increases by 1 when going 1 digit forward. So, the incrementation of the number of primitive operations also increase by 1 when going 1 digit forward from 0 to 8. If the digit is not 0 to 8, then the digit is definitely 9. So, if the digit is 9, then no additional comparison is needed. So, the number of primitive operations when the digit is 8 is same as the number of primitive operations when the digit is 9, which is 10.

After the first pass, if the numbers only have one digit, then reorder is implemented and the process is ended. Figure 3.4 shows the primitive operation counters for reorder after the first pass (only if length is 1).

```
66
67      # Reorder the numbers in a sorted list after the first pass if the numbers has only 1 digit (last position is 0)
68      if position == 0:
69          numberOfPrimitiveOperations += 1    # 1 comparison operation
70
71          # Reorder start (if length == 1)
72          sortedNumbers = []
73          numberOfPrimitiveOperations += 1    # 1 assignment operation
74
75          # The list of numbers is already sorted in ascending order by reading from the first cell to the last cell in
76          # array 1
77          for cellNumber in range(10):
78              for number in array1[cellNumber]:
79                  sortedNumbers.append(number)
80                  numberOfPrimitiveOperations += 1    # 1 assignment operation
81          # Reorder end (if length == 1)
82
```

Figure 3.4 Primitive operation counters for reorder (only if length is 1)

The position is compared with 0 and if the position is equal to 0, then that means that the last position is also the first position, which means that the numbers only have 1 digit. This involves I comparison operation. So, as shown in Figure 3.4, the number of primitive operations is incremented by 1.

If the position is equal to 0, then the reorder process will be implemented. Firstly, an array is initialized to hold the sorted list of numbers. This involves 1 variable assignment operation. So, as shown in Figure 3.4, the number of primitive operations is incremented by 1.

After that, the numbers are appended to the sorted list of numbers, one by one. As shown in Figure 3.4, in the inner for loop, there is 1 variable assignment operation, which is the appendment of the number to the sorted list. So, the number of primitive operations is incremented by 1 in each iteration.

3.1.3 While loop for subsequent passes

If the position scanned in the first pass is not 0, then the number has more than 1 digit. In this case, the programme enters the else block, which runs a while loop until the position reaches 0. Figure 3.5 shows the primitive operation counter for the else block and the while loop.

```
    else:
        numberOfPrimitiveOperations += 1    # 1 comparison operation

        # Continue for subsequent pass

        # Holds the number of passes for future reference
        numberOfPasses = 1
        numberOfPrimitiveOperations += 1    # 1 assignment operation

        while position > 0:
            numberOfPrimitiveOperations += 1    # 1 comparison operation

            # Increment number of passes (go to the next pass) and decrement position (move one position forward)
            numberOfPasses = numberOfPasses + 1
            position = position - 1
            numberOfPrimitiveOperations += 4 # 2 assignment operation, 1 addition operation and 1 subtraction operation
```

Figure 3.5 Primitive operation counters for the while loop

In the if block, the position is compared with 0 and if the position is not equal to 0, then the programme enters the else block. So, if the programme entered the else block, there is already 1 comparison operation involved at the if statement. So, as shown in Figure 3.5, the number of primitive operations is incremented by 1.

Since currently only the first pass is implemented, so the number of passes is initialized as 1. This involves 1 variable assignment operation. So, as shown in Figure 3.5, the number of primitive operations is incremented by 1.

The condition of the while loop is the position is larger than 0, so as long as position is larger than 0, the code in the while loop will be repeated. So, for each iteration, there is 1 comparison operation involved. So, as shown in Figure 3.5, the number of primitive operations is incremented by 1 at the beginning of each iteration.

In the beginning of each iteration, the number of passes is incremented by 1. This involves 1 addition operation and 1 variable assignment operation. Other than that, the position is decremented by 1. This involves 1 subtraction operation and 1 variable assignment operation. So, in total, there are 4 primitive operations involved. So, as shown in Figure 3.5, the number of primitive operations is incremented by 4.

3.1.4 Second pass and all even number passes

In the while loop, the code is split into two parts. If it is the second pass or an even number pass, then the first part of the code will be run, otherwise the second part of the code will be run. Figure 3.6 and Figure 3.7 show the primitive operation counters for the second pass and all even number passes.

```python

# For the second pass (or 4th pass, 6th pass, 8th pass, ...)
# For all passes that read from array 1 and write to array 2
if numberOfPasses % 2 == 0:
    numberOfPrimitiveOperations += 2    # 1 modulus operation and 1 comparison operation

    # Re-initialize array 2 because array 2 should be cleared because this pass should write to array 2
    array2 = [[], [], [], [], [], [], [], [], [], []]
    numberOfPrimitiveOperations += 1    # 1 assignment operation

    # Second pass (or 4th pass, 6th pass, 8th pass, ...) start
    # Reads from the first cell to the last cell in array 1
    for cellNumber in range(10):
        # Reads every number in each cell in array 1
        for number in array1[cellNumber]:
            # Appends the number to corresponding cell in array 2 based on the second last (or 4th last,
            # 6th last, 8th last, ...) digit of the number
            if number[position] == '0':
                array2[0].append(number)
                numberOfPrimitiveOperations += 2    # 1 comparison operation and 1 assignment operation
            elif number[position] == '1':
                array2[1].append(number)
                numberOfPrimitiveOperations += 3    # 2 comparison operations and 1 assignment operation
            elif number[position] == '2':
                array2[2].append(number)
                numberOfPrimitiveOperations += 4    # 3 comparison operations and 1 assignment operation
            elif number[position] == '3':
                array2[3].append(number)
                numberOfPrimitiveOperations += 5    # 4 comparison operations and 1 assignment operation
            elif number[position] == '4':
                array2[4].append(number)
                numberOfPrimitiveOperations += 6    # 5 comparison operations and 1 assignment operation
            elif number[position] == '5':
                array2[5].append(number)
                numberOfPrimitiveOperations += 7    # 6 comparison operations and 1 assignment operation
            elif number[position] == '6':
                array2[6].append(number)
                numberOfPrimitiveOperations += 8    # 7 comparison operations and 1 assignment operation
```

Figure 3.6 Primitive operation counters for second pass and all even number passes (part 1 of 2)

```
                    elif number[position] == '7':
                        array2[7].append(number)
                        numberOfPrimitiveOperations += 9    # 8 comparison operations and 1 assignment operation
                    elif number[position] == '8':
                        array2[8].append(number)
                        numberOfPrimitiveOperations += 10   # 9 comparison operations and 1 assignment operation
                    else:
                        array2[9].append(number)
                        numberOfPrimitiveOperations += 10   # 9 comparison operations and 1 assignment operation
                # Second pass (or 4th pass, 6th pass, 8th pass, ...) end

```

Figure 3.7 Primitive operation counters for second pass and all even number passes (part 2 of 2)

The second pass and the even number passes are identified by the divisibility of the number of passes by 2. If the number of passes is divisible by 2, then the current pass is an even number pass. The divisibility of the number of passes by 2 is determined by comparing the remainder of the number of passes divided by 2 with 0. This involves 2 primitive operations, which is 1 modulus operation and 1 comparison operation. So, as shown in Figure 3.6, the number of primitive operations is incremented by 2.

In the second pass and all even number passes, the programme reads from array 1 and writes to array 2. The information in array 2 is not used and array 2 will be re-initialized to clear the information in array 2. The re-initialization of array 2 involves 1 variable assignment operation. So, as shown in Figure 3.6, the number of primitive operations is incremented by 1.

In the second pass and all even number passes, the numbers are read one by one from array 1 and appended to the array in the corresponding cell in array 2 based on the digit at the currently scanned position. This involves an if-elif-else loop implemented similarly to the first pass. As explained previously in section 3.1.2, the digit at the scanned position is compared to 0 to 8, one by one, so going from digit 0 to 8, the number of comparison operations increases by 1 when going 1 digit forward. Digit 9 has the same number of comparison operations as digit 8 because if the digit is not 0 to 8, then the digit must be 9, so no additional comparison is needed. Besides comparison operation, for every digit, there is also 1 variable assignment operation because the number is appended to the array in the corresponding cell in array 2. So, as shown in Figure 3.6 and Figure 3.7, for digit 0, the number of primitive operations is incremented by 2. Going from 0 to 8, the incrementation of the number of primitive operations increases by 1 when going 1 digit forward. For digit 9, the incrementation of primitive operation is 10, which is the same as digit 8.

3.1.5 Third pass and all odd number passes except the first pass

In the while loop, if the current pass is the third pass or an odd number pass after the third pass, then the programme will execute the code in the else block. Figure 3.8 and Figure 3.9 shows the primitive operation counters for the third pass and all odd number passes except the first pass.

```python

            # For the third pass (or 5th pass, 7th pass, 9th pass, ...)
            # For all passes that read from array 2 and write to array 1
            else:
                numberOfPrimitiveOperations += 2    # 1 modulus operation and 1 comparison operation

                # Re-initialize array 1 because array 1 should be cleared because this pass should write to array 1
                array1 = [[], [], [], [], [], [], [], [], [], []]
                numberOfPrimitiveOperations += 1    # 1 assignment operation

                # Third pass (or 5th pass, 7th pass, 9th pass, ...) start
                # Reads from the first cell to the last cell in array 2
                for cellNumber in range(10):
                    # Reads every number in each cell in array 2
                    for number in array2[cellNumber]:
                        # Appends the number to the corresponding cell in array 1 based on the third last (or 5th last,
                        # 7th last, 9th last, ...) digit of the number
                        if number[position] == '0':
                            array1[0].append(number)
                            numberOfPrimitiveOperations += 2    # 1 comparison operation and 1 assignment operation
                        elif number[position] == '1':
                            array1[1].append(number)
                            numberOfPrimitiveOperations += 3    # 2 comparison operations and 1 assignment operation
                        elif number[position] == '2':
                            array1[2].append(number)
                            numberOfPrimitiveOperations += 4    # 3 comparison operations and 1 assignment operation
                        elif number[position] == '3':
                            array1[3].append(number)
                            numberOfPrimitiveOperations += 5    # 4 comparison operations and 1 assignment operation
                        elif number[position] == '4':
                            array1[4].append(number)
                            numberOfPrimitiveOperations += 6    # 5 comparison operations and 1 assignment operation
                        elif number[position] == '5':
                            array1[5].append(number)
                            numberOfPrimitiveOperations += 7    # 6 comparison operations and 1 assignment operation
                        elif number[position] == '6':
                            array1[6].append(number)
                            numberOfPrimitiveOperations += 8    # 7 comparison operations and 1 assignment operation
```

Figure 3.8 Primitive operation counters for third pass and all odd number passes except the first pass (part 1 of 2)

```
                    elif number[position] == '7':
                        array1[7].append(number)
                        numberOfPrimitiveOperations += 9    # 8 comparison operations and 1 assignment operation
                    elif number[position] == '8':
                        array1[8].append(number)
                        numberOfPrimitiveOperations += 10   # 9 comparison operations and 1 assignment operation
                    else:
                        array1[9].append(number)
                        numberOfPrimitiveOperations += 10   # 9 comparison operations and 1 assignment operation
                # Third pass (or 5th pass, 7th pass, 9th pass, ...) end

            # Iteration end

```

Figure 3.9 Primitive operation counters for third pass and all odd number passes except the first pass (part 2 of 2)

When determining whether the current pass is an even number pass or an odd number pass, in the if statement, the programme checks the divisibility of the number of passes by 2. The programme does this by comparing the remainder of the number of passes divided by 2 with 0. If the remainder is equal to 0, then the programme executes the code in the if block, otherwise the programme executes the code in the else block. So, if the programme executes the else block, there is already 2 primitive operations involved, which is 1 modulus operation and 1 comparison operation. So, as shown in Figure 3.8, at the starting point of the else block, the number of primitive operations is incremented by 2.

In the third pass and all odd number passes (except the first pass), the programme reads from array 2 and writes to array 1. The information in array 1 is not used and array 1 will be re-initialized to clear the information in array 1. The re-initialization of array 1 involves 1 variable assignment operation. So, as shown in Figure 3.8, the number of primitive operations is incremented by 1.

In the third pass and all odd number passes (except the first pass), the numbers are read one by one from array 2 and appended to the array in the corresponding cell in array 1 based on the digit at the currently scanned position. This involves an if-elif-else loop implemented similarly to the first pass. As explained previously in section 3.1.2, the digit at the scanned position is compared to 0 to 8, one by one, so going from digit 0 to 8, the number of comparison operations increases by 1 when going 1 digit forward. Digit 9 has the same number of comparison operations as digit 8 because if the digit is not 0 to 8, then the digit must be 9, so no additional comparison is needed. Besides comparison operation, for every digit, there is also 1 variable assignment operation because the number is appended to the array in the corresponding cell in array 1. So, as shown in Figure 3.8 and Figure 3.9, for digit 0,

the number of primitive operations is incremented by 2. Going from 0 to 8, the incrementation of the number of primitive operations increases by 1 when going 1 digit forward. For digit 9, the incrementation of primitive operation is 10, which is the same as digit 8.

3.1.6 Reorder

After the while loop has ended, the list of numbers is reordered into a sorted list of numbers. Figure 3.10 shows the primitive operation counters for reorder.

```python

        # Reorder start
        sortedNumbers = []
        numberOfPrimitiveOperations += 1    # 1 assignment operation

        # After the second pass (or 4th pass, 6th pass, 8th pass, ...)
        # After all passes that write to array 2
        if numberOfPasses % 2 == 0:
            numberOfPrimitiveOperations += 2    # 1 modulus operation and 1 comparison operation

            # The list of numbers is already sorted in ascending order by reading from the first cell to the last cell
            # in array 2
            for cellNumber in range(10):
                for number in array2[cellNumber]:
                    sortedNumbers.append(number)
                    numberOfPrimitiveOperations += 1    # 1 assignment operation

        # After the third pass (or 5th pass, 7th pass, 9th pass, ...)
        # After all passes that write to array 1 (except the first pass)
        else:
            numberOfPrimitiveOperations += 2    # 1 modulus operation and 1 comparison operation

            # The list of numbers is already sorted in ascending order by reading from the first cell to the last cell
            # in array 1
            for cellNumber in range(10):
                for number in array1[cellNumber]:
                    sortedNumbers.append(number)
                    numberOfPrimitiveOperations += 1    # 1 assignment operation
        # Reorder end

    # Number sorting algorithm end

    return numberOfPrimitiveOperations

```

Figure 3.10 Primitive operation counters for reorder

Firstly, an empty array is initialized to hold the sorted list of numbers. This involves 1 variable assignment operation. So, as shown in Figure 3.10, the number of primitive operations is incremented by 1.

After that, the programme determines whether the last pass is an even number pass or an odd number pass by checking the divisibility of number of passes by 2. This involves 1 modulus operation and 1 comparison operation because the programme

compares the remainder of the number of passes divided by 2 with 0. If the remainder of the number of passes divided by 2 is equal to 0, then the programme executes the code in the if block, otherwise the programme executes the code in the else block. So, no matter the programme executes the code in the if block or the else block, the number of primitive operations involved is 2. So, as shown in Figure 3.10, for both the if block and the else block, the number of primitive operations is incremented by 2.

In the if block (last pass is even number pass), the numbers are read, cell by cell, from array 2, whereas in the else block (last pass is odd number pass), the numbers are read, cell by cel,l from array 1. In both cases, for every number, 1 variable assignment operation is involved because the number read is appended to the “sortedNumbers” array. So, as shown in Figure 3.10, in both cases, for each number, the number of primitive operations is incremented by 1.

Also shown in Figure 3.10, after executing reorder, the number of primitive operations is returned and the “count_primitive_operation_in_number_sorting()” function is ended.

3.2 Generating random unsorted list of numbers

To conduct the experiment to count the number of primitive operations of the number sorting algorithm for a few different values of N (number of numbers), a random unsorted list of numbers of specified number of numbers (N) needs to be generated and passed to the “count_primitive_operation_in_number_sorting()” function as the input. It is laborious to manually create an unsorted list of numbers because the number of numbers (N) in the experiment is large. So, as a solution, a function that can auto-generate random list of numbers is implemented. Figure 3.11 shows the code for the function mentioned.

```python

# Function to randomly generate a list of numbers with a fixed length
# Takes 2 inputs, N and LENGTH
# N is the number of numbers in the list, LENGTH is the fixed length (number of digits) in each number
# Returns a list of numbers (in an array of type string)
def generate_random_list_of_numbers(N, LENGTH):  10 usages  new *
    # Initialize an empty list of numbers
    numbers = []

    # Generate N numbers
    for _ in range(N):
        #Initialize a variable to store the number
        number = ''

        # Generate a number with LENGTH number of digits
        for _ in range(LENGTH):
            # Generate a random integer ranged from 0 to 9
            digit = random.randint( a: 0,  b: 9)

            # Convert the integer to string
            digitStr = str(digit)

            # Append the digit to number
            number = number + digitStr

        # Append the number to the main list
        numbers.append(number)

    return numbers

```

Figure 3.11 Generate random unsorted list of numbers

As shown in Figure 3.11, the “generate_random_list_of_numbers()” function takes N, which is the number of numbers in the list, and LENGTH, which is the fixed length (number of digits) of each number in the list, as input. The function returns a random unsorted list of numbers. All the numbers are in string form.

As shown in Figure 3.11, firstly, an empty array named “numbers” is initialized to hold the numbers generated. For each number, firstly, a string variable named “number”, containing an empty string is initialized to hold the generated number string. For each digit in each number, an integer between 0 and 9 is randomly generated and converted to string. Then, the digit string is appended to the string variable “number”. For each number, after the number of the desired length is generated, the number is appended to the “numbers” array. After every number in the list is generated, the “numbers” array is returned.

3.3 Conducting the experiment

The experiment is conducted with number of digits (or length) as the fixed variable, number of numbers (N) as the manipulated variable and number of primitive

operations as the observed variable. The experiment is conducted by running the "count_primitive_operation_in_number_sorting()" function with unsorted list of numbers of different number of numbers (N). The number of numbers used in the experiment are 50, 100, 150, 200, 250, 300, 350, 400, 450 and 500. Figure 3.12 shows the code for generating random unsorted list of numbers with different number of numbers (N).

```python

# Experiment start
# Number of digits (length) = 5 (Fixed variable)
LENGTH = 5

# Generate a random list of 50 numbers with 5 digits
numbers_N50 = generate_random_list_of_numbers( N: 50, LENGTH)

# Generate a random list of 100 numbers with 5 digits
numbers_N100 = generate_random_list_of_numbers( N: 100, LENGTH)

# Generate a random list of 150 numbers with 5 digits
numbers_N150 = generate_random_list_of_numbers( N: 150, LENGTH)

# Generate a random list of 200 numbers with 5 digits
numbers_N200 = generate_random_list_of_numbers( N: 200, LENGTH)

# Generate a random list of 250 numbers with 5 digits
numbers_N250 = generate_random_list_of_numbers( N: 250, LENGTH)

# Generate a random list of 300 numbers with 5 digits
numbers_N300 = generate_random_list_of_numbers( N: 300, LENGTH)

# Generate a random list of 350 numbers with 5 digits
numbers_N350 = generate_random_list_of_numbers( N: 350, LENGTH)

# Generate a random list of 400 numbers with 5 digits
numbers_N400 = generate_random_list_of_numbers( N: 400, LENGTH)

# Generate a random list of 450 numbers with 5 digits
numbers_N450 = generate_random_list_of_numbers( N: 450, LENGTH)

# Generate a random list of 500 numbers with 5 digits
numbers_N500 = generate_random_list_of_numbers( N: 500, LENGTH)

# Array of N (Manipulated variable)
array_N = [50, 100, 150, 200, 250, 300, 350, 400, 450, 500]

```

Figure 3.12 Generate random unsorted list of numbers with different number of numbers

As shown in Figure 3.12, the number of digits (length) is fixed as 5. This reflects the fixed variable, number of digits, which is not changed throughout the experiment. Then, by using the "generate_random_list_of_numbers()" function, the programme generates random unsorted lists of numbers with 50 numbers, 100 numbers, 150 numbers, 200 numbers, 250 numbers, 300 numbers, 350 numbers, 400 numbers, 450 numbers and 500 numbers. This reflects the manipulated variable, number of numbers, which is purposely changed throughout the experiment. The number of numbers (N)

is recorded in an array named “array_N” for the purpose of drawing the graph of number of primitive operations vs N.

After all lists of numbers is ready, the experiment is conducted to observe the number of primitive operations. Figure 3.13 and Figure 3.14 show the code for conducting the experiment.

```python

# Initialize array of number of primitive operations (Observed variable)
array_numberOfPrimitiveOperations = []

# Count the number of primitive operations for sorting 50 numbers
numberOfPrimitiveOperations_N50 = count_primitive_operation_in_number_sorting(numbers_N50, LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N50)

# Count the number of primitive operations for sorting 100 numbers
numberOfPrimitiveOperations_N100 = count_primitive_operation_in_number_sorting(numbers_N100, LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N100)

# Count the number of primitive operations for sorting 150 numbers
numberOfPrimitiveOperations_N150 = count_primitive_operation_in_number_sorting(numbers_N150, LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N150)

# Count the number of primitive operations for sorting 200 numbers
numberOfPrimitiveOperations_N200 = count_primitive_operation_in_number_sorting(numbers_N200, LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N200)

# Count the number of primitive operations for sorting 250 numbers
numberOfPrimitiveOperations_N250 = count_primitive_operation_in_number_sorting(numbers_N250, LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N250)

# Count the number of primitive operations for sorting 300 numbers
numberOfPrimitiveOperations_N300 = count_primitive_operation_in_number_sorting(numbers_N300, LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N300)

# Count the number of primitive operations for sorting 350 numbers
numberOfPrimitiveOperations_N350 = count_primitive_operation_in_number_sorting(numbers_N350, LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N350)

# Count the number of primitive operations for sorting 400 numbers
numberOfPrimitiveOperations_N400 = count_primitive_operation_in_number_sorting(numbers_N400, LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N400)

```

Figure 3.13 Conduct experiment (part 1 of 2)

```python

# Count the number of primitive operations for sorting 450 numbers
numberOfPrimitiveOperations_N450 = count_primitive_operation_in_number_sorting(numbers_N450, LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N450)

# Count the number of primitive operations for sorting 500 numbers
numberOfPrimitiveOperations_N500 = count_primitive_operation_in_number_sorting(numbers_N500, LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N500)
# Experiment end

```

Figure 3.14 Conduct experiment (part 2 of 2)

As shown in Figure 3.13, firstly, an empty array named “array_numberOfPrimitiveOperations” is initialized to hold the number of primitive operations recorded for each number of numbers (N) in the experiment. As shown in

Figure 3.13 and Figure 3.14, the array of 50 numbers, the array of 100 numbers, the array of 150 numbers, the array of 200 numbers, the array of 250 numbers, the array of 300 numbers, the array of 350 numbers, the array of 400 numbers, the array of 450 numbers and the array of 500 numbers are passed into the "count_primitive_operation_in_number_sorting()" function with the number of digits (length) fixed at 5. For each set of the experiment, the number of primitive operations is appended to the array "array_numberOfPrimitiveOperations".

3.4 Displaying the experiment result and plotting the graph

After the experiment is conducted, the experiment result is displayed by printing out the number of primitive operations for each N (number of numbers) and plotting a graph of number of primitive operations vs N. Figure 3.15 shows the code for displaying the experiment result and plotting the graph.

```python

# Print the fixed variable, the manipulated variable and the observed variable
print(f'Fixed variable: Number of digits (LENGTH): {LENGTH}')
print(f'Manipulated variable: Number of numbers (N): {array_N}')
print(f'Observed variable: Number of primitive operations: {array_numberOfPrimitiveOperations}')

# Plot the graph of number of primitive operations vs number of numbers (N)
# The manipulated variable (x)
N = np.array(array_N)

# The observed variable (y)
numberOfPrimitiveOperations = np.array(array_numberOfPrimitiveOperations)

# Plot the line
plt.plot( *args: N, numberOfPrimitiveOperations)

# Specify the label of the x-axis
plt.xlabel('N')

# Specify the label of the y-axis
plt.ylabel('Number of primitive operations')

# Specify the title of the graph
plt.title('Graph of Number of Primitive Operations vs N for Number Sorting Algorithm')

plt.show()

```

Figure 3.15 Display experiment result and plot graph

As shown in Figure 3.15, the data associated with the fixed variable, which is number of digits (length), the manipulated variable, which is number of numbers (N), and the observed variable, which is number of primitive operations, are printed to the user.

After that, as shown in Figure 3.15, the data of the manipulated variable (N) (in array form), and the data of the observed variable (number of primitive operations) (in array form) are converted into Numpy arrays. The Numpy arrays containing the manipulated variable data (number of numbers) (N) and observed variable data (number of primitive operations) is feed into the Matplotlib.pyplot function “plt.plot” to plot a line graph. The manipulated variable, N (number of numbers), is set as the x variable, whereas the observed variable, number of primitive operations, is set as the y variable. Then, before showing the graph, the x axis label of the graph is set as “N”, whereas the y axis label of the graph is set as “Number of primitive operations”, whereas the title of the graph is set as “Graph of Number of Primitive Operations vs N for Number Sorting Algorithm”.

3.5 Running the programme

Figure 3.16 shows the output of the programme.

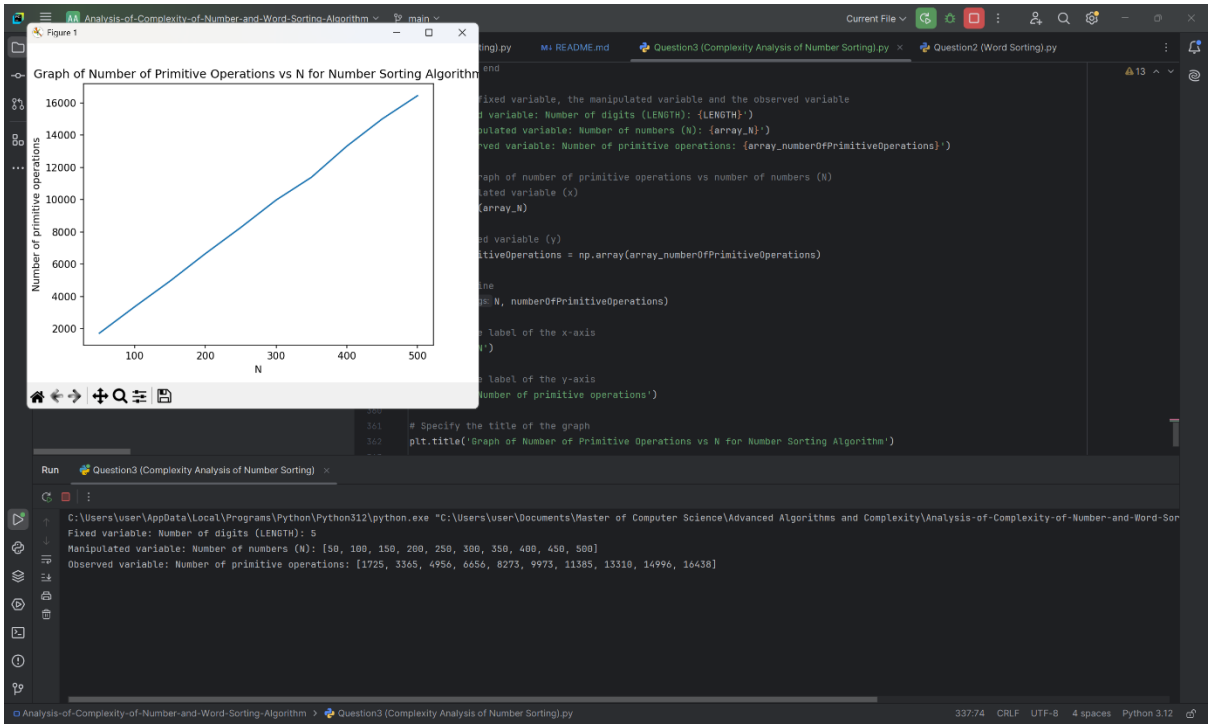

Figure 3.16 Programme output

As shown in Figure 3.16, as the output of the programme, the numbers of primitive operations for each N (number of numbers) are printed and a graph of number of primitive operations vs N is plotted.

3.6 Determining time complexity based on the experiment result

Table 3.1 records the experiment result by listing the number of primitive operations for each number of numbers (N).

Table 3.1 Number of primitive operations for different number of numbers (N)

Number of numbers (N)	Number of primitive operations
50	1725
100	3365
150	4956
200	6656
250	8273
300	9973
350	11385
400	13310
450	14996
500	16438

As shown in Table 3.1, it can be said that, the number of primitive operations increases as N (number of numbers) increases. Figure 3.17 is the graph of number of primitive operations vs N (number of numbers) plotted based on the experiment result as in Table 3.1.

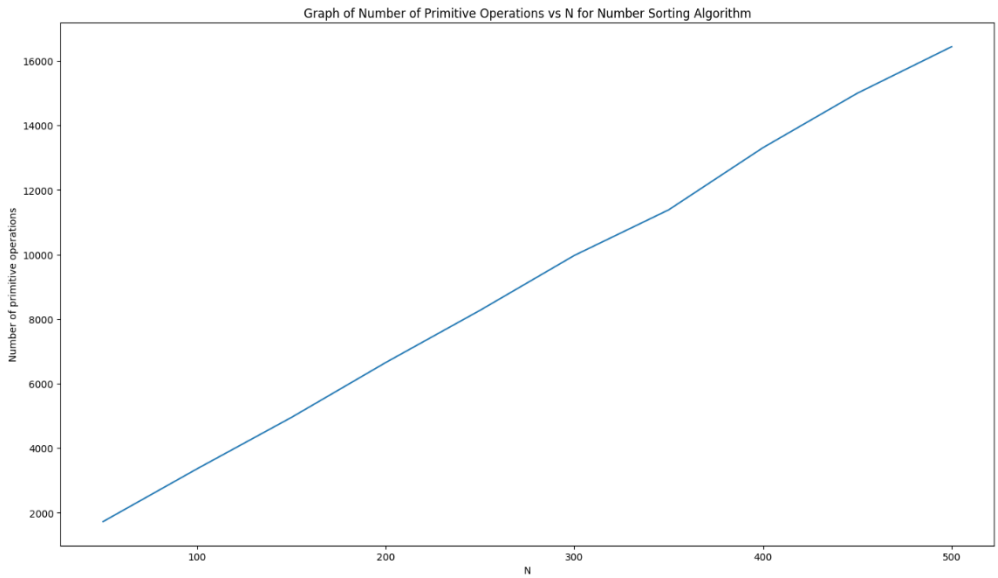

Figure 3.17 Graph of number of primitive operations vs N for number sorting algorithm

As shown in Figure 3.17, the graph of number of primitive operations vs N (number of numbers) is a line which is nearly straight. In the graph, the number of primitive operations increases steadily and linearly when N increases. This shows that the relationship between the number of primitive operations and N is linear. Since the relationship between the number of primitive operations and N is linear, it can be said that the time complexity of the number sorting algorithm is big-O O(N).

4. Complexity analysis on the word sorting algorithm

This section is about the time complexity analysis on the word sorting algorithm implemented in section 2. Counters are added to the word sorting algorithm to count the number of primitive operations. The word sorting algorithm is then run on different number of words and the number of primitive operations for each trial is recorded. Finally, a graph of number of primitive operations against number of words (N) is plotted to determine the time complexity of the word sorting algorithm.

4.1 Counting the number of primitive operations

The code for implementing the word sorting algorithm is modified into a function that takes a list of unsorted words and the maximum length (maximum number of characters) of the words as input and returns the number of primitive operations required to sort the list of words in ascending order. Compared to the code in section 2, the code in this section does not ask user to input the list of words and the maximum length (maximum number of characters) of the words. Instead, the list of words and the maximum length (maximum number of characters) are passed as arguments of the function. The function will implement the word sorting algorithm exactly like what the code in section 2 does. Besides that, another modification is, in the function, to count the number of primitive operations of the word sorting algorithm, primitive operation counters are added to various parts of the code.

4.1.1 Array initialization

Figure 4.1 shows the function to count the number of primitive operations when implementing the word sorting algorithm. Figure 4.1 also shows the primitive operation counter for array initialization.

```python

# Complexity analysis of word sorting algorithm

# Function to count number of primitive operation in word sorting
# Takes 2 input, namely words and MAXIMUM_LENGTH
# words is the unsorted list of words (lower case character string) with different lengths smaller than or equal to the
# maximum length, MAXIMUM_LENGTH is the maximum length (number of characters) of each word
# Returns the number of primitive operations required to sort the list of words
def count_primitive_operation_in_word_sorting(words, MAXIMUM_LENGTH):  10 usages  new *
    # Initialize primitive operation counter
    numberOfPrimitiveOperations = 0

    # Word sorting algorithm start

    # Initialize start
    # Initialize 2 arrays with 27 elements (which is also an array)
    array1 = [[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [],
              []]
    array2 = [[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [],
              []]
    numberOfPrimitiveOperations += 2    # 2 array initialization operations
    # Initialize end

```

Figure 4.1 Primitive operation counter for array initialization

As shown in Figure 4.1, the function "count_primitive_operation_in_word_sorting()" is used to count the number of primitive operations to sort the list of words inputted using the word sorting algorithm described in section 2. Besides the unsorted list of words, the function also takes maximum length, which is the maximum number of characters of the words in the list, as input. This maximum length is important because it is used by the word sorting algorithm to determine the last position of the longest word since the last position of the longest word is equal to maximum_length-1.

As shown in Figure 4.1, the number of primitive operations is initialized as 0 at the beginning of the function. During array initialization, two arrays, namely array 1 and array 2, are initialized. Since array initialization involves assignment of variable and there are two arrays, so the number of primitive operations is incremented by 2.

4.1.2 First pass

After array initialization, the next step is the first pass. Figure 4.2, Figure 4.3 and Figure 4.4 show the primitive operation counters for the first pass.

```python

    # Iteration start

    # For the first pass, need to look at the last character (position of length - 1)
    position = MAXIMUM_LENGTH - 1
    numberOfPrimitiveOperations += 2    # 1 subtraction operation and 1 assignment operation

    # First pass start
    # Pass through all words in the list of words inputted
    for word in words:
        # Append the word to the corresponding cell in array 1 based on the last character of the word
        # If the word is shorter than the currently scanned position (length < position + 1), then append the word to
        # cell 0 (shorter strings are sorted before all longer strings)
        if len(word) < position + 1:
            array1[0].append(word)
            numberOfPrimitiveOperations += 4    # 1 function call, 1 addition, 1 comparison, 1 assignment
        elif word[position] == 'a':
            array1[1].append(word)
            numberOfPrimitiveOperations += 5    # 1 function call, 1 addition, 2 comparisons, 1 assignment
        elif word[position] == 'b':
            array1[2].append(word)
            numberOfPrimitiveOperations += 6    # 1 function call, 1 addition, 3 comparisons, 1 assignment
        elif word[position] == 'c':
            array1[3].append(word)
            numberOfPrimitiveOperations += 7    # 1 function call,  1 addition, 4 comparisons, 1 assignment
        elif word[position] == 'd':
            array1[4].append(word)
            numberOfPrimitiveOperations += 8    # 1 function call, 1 addition, 5 comparisons, 1 assignment
        elif word[position] == 'e':
            array1[5].append(word)
            numberOfPrimitiveOperations += 9    # 1 function call, 1 addition, 6 comparisons, 1 assignment
        elif word[position] == 'f':
            array1[6].append(word)
            numberOfPrimitiveOperations += 10   # 1 function call, 1 addition, 7 comparisons, 1 assignment
        elif word[position] == 'g':
            array1[7].append(word)
            numberOfPrimitiveOperations += 11   # 1 function call, 1 addition, 8 comparisons, 1 assignment
```

Figure 4.2 Primitive operation counters for first pass (part 1 of 3)

```
        elif word[position] == 'h':
            array1[8].append(word)
            numberOfPrimitiveOperations += 12   # 1 function call, 1 addition, 9 comparisons, 1 assignment
        elif word[position] == 'i':
            array1[9].append(word)
            numberOfPrimitiveOperations += 13   # 1 function call, 1 addition, 10 comparisons, 1 assignment
        elif word[position] == 'j':
            array1[10].append(word)
            numberOfPrimitiveOperations += 14   # 1 function call, 1 addition, 11 comparisons, 1 assignment
        elif word[position] == 'k':
            array1[11].append(word)
            numberOfPrimitiveOperations += 15   # 1 function call, 1 addition, 12 comparisons, 1 assignment
        elif word[position] == 'l':
            array1[12].append(word)
            numberOfPrimitiveOperations += 16   # 1 function call, 1 addition, 13 comparisons, 1 assignment
        elif word[position] == 'm':
            array1[13].append(word)
            numberOfPrimitiveOperations += 17   # 1 function call, 1 addition, 14 comparisons, 1 assignment
        elif word[position] == 'n':
            array1[14].append(word)
            numberOfPrimitiveOperations += 18   # 1 function call, 1 addition, 15 comparisons, 1 assignment
        elif word[position] == 'o':
            array1[15].append(word)
            numberOfPrimitiveOperations += 19   # 1 function call, 1 addition, 16 comparisons, 1 assignment
        elif word[position] == 'p':
            array1[16].append(word)
            numberOfPrimitiveOperations += 20   # 1 function call, 1 addition, 17 comparisons, 1 assignment
        elif word[position] == 'q':
            array1[17].append(word)
            numberOfPrimitiveOperations += 21   # 1 function call, 1 addition, 18 comparisons, 1 assignment
        elif word[position] == 'r':
            array1[18].append(word)
            numberOfPrimitiveOperations += 22   # 1 function call, 1 addition, 19 comparisons, 1 assignment
        elif word[position] == 's':
            array1[19].append(word)
            numberOfPrimitiveOperations += 23   # 1 function call, 1 addition, 20 comparisons, 1 assignment
```

Figure 4.3 Primitive operation counters for first pass (part 2 of 3)

```
        elif word[position] == 't':
            array1[20].append(word)
            numberOfPrimitiveOperations += 24   # 1 function call, 1 addition, 21 comparisons, 1 assignment
        elif word[position] == 'u':
            array1[21].append(word)
            numberOfPrimitiveOperations += 25   # 1 function call, 1 addition, 22 comparisons, 1 assignment
        elif word[position] == 'v':
            array1[22].append(word)
            numberOfPrimitiveOperations += 26   # 1 function call, 1 addition, 23 comparisons, 1 assignment
        elif word[position] == 'w':
            array1[23].append(word)
            numberOfPrimitiveOperations += 27   # 1 function call, 1 addition, 24 comparisons, 1 assignment
        elif word[position] == 'x':
            array1[24].append(word)
            numberOfPrimitiveOperations += 28   # 1 function call, 1 addition, 25 comparisons, 1 assignment
        elif word[position] == 'y':
            array1[25].append(word)
            numberOfPrimitiveOperations += 29   # 1 function call, 1 addition, 26 comparisons, 1 assignment
        else:
            array1[26].append(word)
            numberOfPrimitiveOperations += 29   # 1 function call, 1 addition, 26 comparisons, 1 assignment
    # First pass end

```

Figure 4.4 Primitive operation counters for first pass (part 3 of 3)

Before the first pass starts, the position is initialized as maximum_length-1. This involves 2 primitive operations, which is 1 subtraction operation and 1 variable

assignment operation. So, as shown in Figure 4.2, the number of primitive operations is incremented by 2.

Firstly, if the length of the word is shorter than the scanned position, then the word is appended to the array at cell 0. Whether the length of the word is shorter than the scanned position is determined by getting the length of the word by calling len(word) and comparing the length of the word to the current position (position+1). This involves 3 primitive operations, which is 1 function call operation, 1 addition operation and 1 comparison operation. So, if the length of the word is shorter than the scanned position, then the number of primitive operations involved is 4, which is the 3 said primitive operations and 1 variable assignment operation.

Otherwise, if the length of the word is longer or equal to the scanned position, then the word is appended to the array at the corresponding cell based on the character at the scanned position. The character at the scanned position is compared to ‘a’ and if the comparison returns true, then the number is appended to the array at cell 1. So, if the character at the scanned position is ‘a’, then the number of primitive operations involved is 5, which is the 3 operations of the first comparison (length of word compared with position-1), 1 comparison operation (the character at the scanned position compared with ‘a’) and 1 variable assignment operation. If the character at the scanned position is not ‘a’, then the character will be compared to ‘b’ and if the comparison returns true, then the word is appended to the array at cell 2. This involves 6 primitive operations, which is 3 primitive operations when comparing the length of the word with position-1, 2 comparison operations (compared with ‘a’ and with ‘b’) and 1 variable assignment operation. If the character at the scanned position is not ‘b’, then the character will be compared with ‘c’. The comparison will continue by comparing the characters with ‘a’ to ‘z’, one by one, until the comparison returns true. When the comparison returns true, the character is appended to the corresponding cell as described in section 2.4.1. So, it can be observed that, going from ‘a’ to ‘y’, the number of comparison operation increases by 1 when going 1 character forward. So, the incrementation of the number of primitive operations also increase by 1 when going 1 digit forward from ‘a’ to ‘y’. If the character is not ‘a’ to ‘y’, then the digit is definitely ‘z’. So, if the character is ‘z’, then no additional comparison is needed. So, the number of primitive operations when the character is ‘y’ is same as the number of primitive operations when the character is ‘z’, which is 29.

After the first pass, if the words only have one character, then reorder is implemented and the process is ended. Figure 4.5 shows the primitive operation counters for reorder after the first pass (only if length is 1).

```
    # Reorder the words in a sorted list after the first pass if the words has only 1 character (last position is 0)
    if position == 0:
        numberOfPrimitiveOperations += 1    # 1 comparison operation

        # Reorder start (if length == 1)
        sortedWords = []
        numberOfPrimitiveOperations += 1    # 1 assignment operation

        # The list of words is already sorted in ascending order by reading from the first cell to the last cell in
        # array 1
        for cellNumber in range(27):
            for word in array1[cellNumber]:
                sortedWords.append(word)
                numberOfPrimitiveOperations += 1    # 1 assignment operation
        # Reorder end (if length == 1)
```

Figure 4.5 Primitive operation counters for reorder (only if length is 1)

The position is compared with 0 and if the position is equal to 0, then that means that the last position is also the first position, which means that the words only have 1 character. This involves I comparison operation. So, as shown in Figure 4.5, the number of primitive operations is incremented by 1.

If the position is equal to 0, then the reorder process will be implemented. Firstly, an array is initialized to hold the sorted list of words. This involves 1 variable assignment operation. So, as shown in Figure 4.5, the number of primitive operations is incremented by 1.

After that, the words are appended to the sorted list of words, one by one. As shown in Figure 4.5, in the inner for loop, there is 1 variable assignment operation, which is the appendment of the word to the sorted list. So, the number of primitive operations is incremented by 1 for each iteration.

4.1.3 While loop for subsequent passes

If the position scanned in the first pass is not 0, then the longest word has more than 1 character. In this case, the programme enters the else block, which runs a while loop until the position reaches 0. Figure 4.6 shows the primitive operation counters for the else block and the while loop.

```
else:
    numberOfPrimitiveOperations += 1    # 1 comparison operation
    # Continue for subsequent pass

    # Holds the number of passes for future reference
    numberOfPasses = 1
    numberOfPrimitiveOperations += 1    # 1 assignment operation

    while position > 0:
        numberOfPrimitiveOperations += 1    # 1 comparison operation

        # Increment number of passes (go to the next pass) and decrement position (move one position forward)
        numberOfPasses = numberOfPasses + 1
        position = position - 1
        numberOfPrimitiveOperations += 4 # 2 assignment operations, 1 addition operation, 1 subtraction operation
```

Figure 4.6 Primitive operation counters for the while loop

In the if block, the position is compared with 0 and if the position is not equal to 0, then the programme enters the else block. So, if the programme entered the else block, there is already 1 comparison operation involved at the if statement. So, as shown in Figure 4.6, the number of primitive operations is incremented by 1.

Since currently only the first pass is implemented, so the number of passes is initialized as 1. This involves 1 variable assignment operation. So, as shown in Figure 4.6, the number of primitive operations is incremented by 1.

The condition of the while loop is the position larger than 0, so as long as the position is larger than 0, the code in the while loop will be repeated. So, for each iteration, there is 1 comparison operation involved. So, as shown in Figure 4.6, the number of primitive operations is incremented by 1 for each iteration.

In the beginning of each iteration, the number of passes is incremented by 1. This involves 1 addition operation and 1 variable assignment operation. Other than that, the position is decremented by 1. This involves 1 subtraction operation and 1 variable assignment operation. So, in total, there are 4 primitive operations involved. So, as shown in Figure 4.6, the number of primitive operations is incremented by 4.

4.1.4 Second pass and all even number passes

In the while loop, the code is split into two parts. If it is the second pass or an even number pass, then the first part of the code will be run, otherwise the second part

of the code will be run. Figure 4.7, Figure 4.8 and Figure 4.9 show the primitive operation counters for the second pass and all even number passes.

```python

        # For the second pass (or 4th pass, 6th pass, 8th pass, ...)
        # For all passes that read from array 1 and write to array 2
        if numberOfPasses % 2 == 0:
            numberOfPrimitiveOperations += 2    # 1 modulus operation and 1 comparison operation

            # Re-initialize array 2 because array 2 should be cleared because this pass should write to array 2
            array2 = [[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [],
                      [], [], [], []]
            numberOfPrimitiveOperations += 1    # 1 assignment operation

            # Second pass (or 4th pass, 6th pass, 8th pass, ...) start
            # Reads from the first cell to the last cell in array 1
            for cellNumber in range(27):
                # Reads every word in each cell in array 1
                for word in array1[cellNumber]:
                    # Appends the number to corresponding cell in array 2 based on the second last (or 4th last,
                    # 6th last, 8th last, ...) character of the word
                    # If the word is shorter than the currently scanned position (length < position + 1), then
                    # append the word to cell 0 (shorter strings are sorted before all longer strings)
                    if len(word) < position + 1:
                        array2[0].append(word)
                        numberOfPrimitiveOperations += 4    # 1 function call,1 addition,1 comparison,1 assignment
                    elif word[position] == 'a':
                        array2[1].append(word)
                        numberOfPrimitiveOperations += 5    # 1 function call,1 addition,2 comparisons,1 assignment
                    elif word[position] == 'b':
                        array2[2].append(word)
                        numberOfPrimitiveOperations += 6    # 1 function call,1 addition,3 comparisons,1 assignment
                    elif word[position] == 'c':
                        array2[3].append(word)
                        numberOfPrimitiveOperations += 7    # 1 function call,1 addition,4 comparisons,1 assignment
                    elif word[position] == 'd':
                        array2[4].append(word)
                        numberOfPrimitiveOperations += 8    # 1 function call,1 addition,5 comparisons,1 assignment
```

Figure 4.7 Primitive operation counters for second pass and all even number passes (part 1 of 3)

```
                elif word[position] == 'e':
                    array2[5].append(word)
                    numberOfPrimitiveOperations += 9    # 1 function call,1 addition,6 comparisons,1 assignment
                elif word[position] == 'f':
                    array2[6].append(word)
                    numberOfPrimitiveOperations += 10   # 1 function call,1 addition,7 comparisons,1 assignment
                elif word[position] == 'g':
                    array2[7].append(word)
                    numberOfPrimitiveOperations += 11   # 1 function call,1 addition,8 comparisons,1 assignment
                elif word[position] == 'h':
                    array2[8].append(word)
                    numberOfPrimitiveOperations += 12   # 1 function call,1 addition,9 comparisons,1 assignment
                elif word[position] == 'i':
                    array2[9].append(word)
                    numberOfPrimitiveOperations += 13   # 1 function call,1 addition,10 comparisons,1 assignment
                elif word[position] == 'j':
                    array2[10].append(word)
                    numberOfPrimitiveOperations += 14   # 1 function call,1 addition,11 comparisons,1 assignment
                elif word[position] == 'k':
                    array2[11].append(word)
                    numberOfPrimitiveOperations += 15   # 1 function call,1 addition,12 comparisons,1 assignment
                elif word[position] == 'l':
                    array2[12].append(word)
                    numberOfPrimitiveOperations += 16   # 1 function call,1 addition,13 comparisons,1 assignment
                elif word[position] == 'm':
                    array2[13].append(word)
                    numberOfPrimitiveOperations += 17   # 1 function call,1 addition,14 comparisons,1 assignment
                elif word[position] == 'n':
                    array2[14].append(word)
                    numberOfPrimitiveOperations += 18   # 1 function call,1 addition,15 comparisons,1 assignment
                elif word[position] == 'o':
                    array2[15].append(word)
                    numberOfPrimitiveOperations += 19   # 1 function call,1 addition,16 comparisons,1 assignment
                elif word[position] == 'p':
                    array2[16].append(word)
                    numberOfPrimitiveOperations += 20   # 1 function call,1 addition,17 comparisons,1 assignment
```

Figure 4.8 Primitive operation counters for second pass and all even number passes (part 2 of 3)

```
                    elif word[position] == 'q':
                        array2[17].append(word)
                        numberOfPrimitiveOperations += 21   # 1 function call,1 addition,18 comparisons,1 assignment
                    elif word[position] == 'r':
                        array2[18].append(word)
                        numberOfPrimitiveOperations += 22   # 1 function call,1 addition,19 comparisons,1 assignment
                    elif word[position] == 's':
                        array2[19].append(word)
                        numberOfPrimitiveOperations += 23   # 1 function call,1 addition,20 comparisons,1 assignment
                    elif word[position] == 't':
                        array2[20].append(word)
                        numberOfPrimitiveOperations += 24   # 1 function call,1 addition,21 comparisons,1 assignment
                    elif word[position] == 'u':
                        array2[21].append(word)
                        numberOfPrimitiveOperations += 25   # 1 function call,1 addition,22 comparisons,1 assignment
                    elif word[position] == 'v':
                        array2[22].append(word)
                        numberOfPrimitiveOperations += 26   # 1 function call,1 addition,23 comparisons,1 assignment
                    elif word[position] == 'w':
                        array2[23].append(word)
                        numberOfPrimitiveOperations += 27   # 1 function call,1 addition,24 comparisons,1 assignment
                    elif word[position] == 'x':
                        array2[24].append(word)
                        numberOfPrimitiveOperations += 28   # 1 function call,1 addition,25 comparisons,1 assignment
                    elif word[position] == 'y':
                        array2[25].append(word)
                        numberOfPrimitiveOperations += 29   # 1 function call,1 addition,26 comparisons,1 assignment
                    else:
                        array2[26].append(word)
                        numberOfPrimitiveOperations += 29   # 1 function call,1 addition,26 comparisons,1 assignment
                # Second pass (or 4th pass, 6th pass, 8th pass, ...) end

```

Figure 4.9 Primitive operation counters for second pass and all even number passes (part 3 of 3)

The second pass and the even number passes are identified by the divisibility of the number of passes by 2. If the number of passes is divisible by 2, then the current pass is an even number pass. The divisibility of the number of passes by 2 is determined by comparing the remainder of the number of passes divided by 2 with 0. This involves 2 primitive operations, which is 1 modulus operation and 1 comparison operation. So, as shown in Figure 4.7, the number of primitive operations is incremented by 2.

In the second pass and all even number passes, the programme reads from array 1 and writes to array 2. The information in array 2 is not used and array 2 will be re-initialized to clear the information in array 2. The re-initialization of array 2 involves 1 variable assignment operation. So, as shown in Figure 4.7, the number of primitive operations is incremented by 1.

In the second pass and all even number passes, the words are read one by one from array 1 and appended to the array in the corresponding cell in array 2 based on the length of the word or the character at the currently scanned position. This involves an if-elif-else loop implemented similarly to the first pass. As explained

previously in section 4.1.2, the first if statement compares the length of the word with the currently scanned position (involves 3 primitive operations) and in the following elif statement, the character at the scanned position is compared to ‘a’ to ‘y’, one by one, so going from character ‘a’ to ‘y’, the number of comparison operations increases by 1 when going 1 character forward. Character ‘z’ has the same number of comparison operations as character ‘y’ because if the character is not ‘a’ to ‘y’, then the digit must be ‘z’, so no additional comparison is needed. Besides comparison operation, for every character, there is also 1 variable assignment operation because the number is appended to the array in the corresponding cell in array 2. So, as shown in Figure 4.7, Figure 4.8 and Figure 4.9, if the length of the word is shorter than the scanned position, then the number of primitive operations is incremented by 4, whereas for character ‘a’, the number of primitive operations is incremented by 5. Going from ‘a’ to ‘y’, the incrementation of the number of primitive operations increases by 1 when going 1 character forward. For character ‘z’, the incrementation of primitive operation is 29, which is the same as character ‘y’.

4.1.5 Third pass and all odd number passes except the first pass

In the while loop, if the current pass is the third pass or an odd number pass after the third pass, then the programme will execute the code in the else block. Figure 4.10, Figure 4.11 and Figure 4.12 shows the primitive operation counters for the third pass and all odd number passes except the first pass.

```python

            # For the third pass (or 5th pass, 7th pass, 9th pass, ...)
            # For all passes that read from array 2 and write to array 1
            else:
                numberOfPrimitiveOperations += 2    # 1 modulus operation and 1 comparison operation

                # Re-initialize array 1 because array 1 should be cleared because this pass should write to array 1
                array1 = [[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [],
                          [], [], [], []]
                numberOfPrimitiveOperations += 1    # 1 assignment operation

                # Third pass (or 5th pass, 7th pass, 9th pass, ...) start
                # Reads from the first cell to the last cell in array 2
                for cellNumber in range(27):
                    # Reads every word in each cell in array 2
                    for word in array2[cellNumber]:
                        # Appends the word to the corresponding cell in array 1 based on the third last (or 5th last,
                        # 7th last, 9th last, ...) character of the word
                        # If the word is shorter than the currently scanned position (length < position + 1), then
                        # append the word to cell 0 (shorter strings are sorted before all longer strings)
                        if len(word) < position + 1:
                            array1[0].append(word)
                            numberOfPrimitiveOperations += 4    # 1 function call,1 addition, 1 comparison,1 assignment
                        elif word[position] == 'a':
                            array1[1].append(word)
                            numberOfPrimitiveOperations += 5    # 1 function call,1 addition,2 comparisons,1 assignment
                        elif word[position] == 'b':
                            array1[2].append(word)
                            numberOfPrimitiveOperations += 6    # 1 function call,1 addition,3 comparisons,1 assignment
                        elif word[position] == 'c':
                            array1[3].append(word)
                            numberOfPrimitiveOperations += 7    # 1 function call,1 addition,4 comparisons,1 assignment
                        elif word[position] == 'd':
                            array1[4].append(word)
                            numberOfPrimitiveOperations += 8    # 1 function call,1 addition,5 comparisons,1 assignment
```

Figure 4.10 Primitive operation counters for third pass and all odd number passes except the first pass (part 1 of 3)

```
elif word[position] == 'e':
    array1[5].append(word)
    numberOfPrimitiveOperations += 9    # 1 function call,1 addition,6 comparisons,1 assignment
elif word[position] == 'f':
    array1[6].append(word)
    numberOfPrimitiveOperations += 10   # 1 function call,1 addition,7 comparisons,1 assignment
elif word[position] == 'g':
    array1[7].append(word)
    numberOfPrimitiveOperations += 11   # 1 addition,8 comparisons,1 assignment
elif word[position] == 'h':
    array1[8].append(word)
    numberOfPrimitiveOperations += 12   # 1 function call,1 addition,9 comparisons,1 assignment
elif word[position] == 'i':
    array1[9].append(word)
    numberOfPrimitiveOperations += 13   # 1 function call,1 addition,10 comparisons,1 assignment
elif word[position] == 'j':
    array1[10].append(word)
    numberOfPrimitiveOperations += 14   # 1 function call,1 addition,11 comparisons,1 assignment
elif word[position] == 'k':
    array1[11].append(word)
    numberOfPrimitiveOperations += 15   # 1 function call,1 addition,12 comparisons,1 assignment
elif word[position] == 'l':
    array1[12].append(word)
    numberOfPrimitiveOperations += 16   # 1 function call,1 addition,13 comparisons,1 assignment
elif word[position] == 'm':
    array1[13].append(word)
    numberOfPrimitiveOperations += 17   # 1 function call,1 addition,14 comparisons,1 assignment
elif word[position] == 'n':
    array1[14].append(word)
    numberOfPrimitiveOperations += 18   # 1 function call,1 addition,15 comparisons,1 assignment
elif word[position] == 'o':
    array1[15].append(word)
    numberOfPrimitiveOperations += 19   # 1 function call,1 addition,16 comparisons,1 assignment
elif word[position] == 'p':
    array1[16].append(word)
    numberOfPrimitiveOperations += 20   # 1 function call,1 addition,17 comparisons,1 assignment
```

Figure 4.11 Primitive operation counters for third pass and all odd number passes except the first pass (part 2 of 3)

```
                    elif word[position] == 'q':
                        array1[17].append(word)
                        numberOfPrimitiveOperations += 21   # 1 function call,1 addition,18 comparisons,1 assignment
                    elif word[position] == 'r':
                        array1[18].append(word)
                        numberOfPrimitiveOperations += 22   # 1 function call,1 addition,19 comparisons,1 assignment
                    elif word[position] == 's':
                        array1[19].append(word)
                        numberOfPrimitiveOperations += 23   # 1 function call,1 addition,20 comparisons,1 assignment
                    elif word[position] == 't':
                        array1[20].append(word)
                        numberOfPrimitiveOperations += 24   # 1 function call,1 addition,21 comparisons,1 assignment
                    elif word[position] == 'u':
                        array1[21].append(word)
                        numberOfPrimitiveOperations += 25   # 1 function call,1 addition,22 comparisons,1 assignment
                    elif word[position] == 'v':
                        array1[22].append(word)
                        numberOfPrimitiveOperations += 26   # 1 function call,1 addition,23 comparisons,1 assignment
                    elif word[position] == 'w':
                        array1[23].append(word)
                        numberOfPrimitiveOperations += 27   # 1 function call,1 addition,24 comparisons,1 assignment
                    elif word[position] == 'x':
                        array1[24].append(word)
                        numberOfPrimitiveOperations += 28   # 1 function call,1 addition,25 comparisons,1 assignment
                    elif word[position] == 'y':
                        array1[25].append(word)
                        numberOfPrimitiveOperations += 29   # 1 function call,1 addition,26 comparisons,1 assignment
                    else:
                        array1[26].append(word)
                        numberOfPrimitiveOperations += 29   # 1 function call,1 addition,26 comparisons,1 assignment
                # Third pass (or 5th pass, 7th pass, 9th pass, ...) end

        # Iteration end

```

Figure 4.12 Primitive operation counters for third pass and all odd number passes except the first pass (part 3 of 3)

When determining whether the current pass is an even number pass or an odd number pass, in the if statement, the programme checks the divisibility of the number of passes by 2. The programme does this by comparing the remainder of the number of passes divided by 2 with 0. If the remainder is equal to 0, then the programme executes the code in the if block, otherwise the programme executes the code in the else block. So, if the programme executes the else block, there is already 2 primitive operations involved, which is 1 modulus operation and 1 comparison operation. So, as shown in Figure 4.10, at the starting point of the else block, the number of primitive operations is incremented by 2.

In the third pass and all odd number passes (except the first pass), the programme reads from array 2 and writes to array 1. The information in array 1 is not used and array 1 will be re-initialized to clear the information in array 1. The re-initialization of array 1 involves 1 variable assignment operation. So, as shown in Figure 4.10, the number of primitive operations is incremented by 1.

In the second pass and all even number passes, the words are read one by one from array 2 and appended to the array in the corresponding cell in array 1 based on the length of the word or the character at the currently scanned position. This involves an if-elif-else loop implemented similarly to the first pass. As explained previously in section 4.1.2, the first if statement compares the length of the word with the currently scanned position (involves 3 primitive operations) and in the following elif statement, the character at the scanned position is compared to ‘a’ to ‘y’, one by one, so going from character ‘a’ to ‘y’, the number of comparison operations increases by 1 when going 1 character forward. Character ‘z’ has the same number of comparison operations as character ‘y’ because if the character is not ‘a’ to ‘y’, then the digit must be ‘z’, so no additional comparison is needed. Besides comparison operation, for every character, there is also 1 variable assignment operation because the number is appended to the array in the corresponding cell in array 2. So, as shown in Figure 4.10, Figure 4.11 and Figure 4.12, if the length of the word is shorter than the scanned position, then the number of primitive operations is incremented by 4, whereas for character ‘a’, the number of primitive operations is incremented by 5. Going from ‘a’ to ‘y’, the incrementation of the number of primitive operations increases by 1 when going 1 character forward. For character ‘z’, the incrementation of primitive operation is 29, which is the same as character ‘y’.

4.1.6 Reorder

After the while loop has ended, the list of words is reordered into a sorted list of words. Figure 4.13 shows the primitive operation counters for reorder.

```python

        # Reorder start
        sortedWords = []
        numberOfPrimitiveOperations += 1    # 1 assignment operation

        # After the second pass (or 4th pass, 6th pass, 8th pass, ...)
        # After all passes that write to array 2
        if numberOfPasses % 2 == 0:
            numberOfPrimitiveOperations += 2    # 1 modulus operation and 1 comparison operation

            # The list of words is already sorted in ascending order by reading from the first cell to the last cell
            # in array 2
            for cellNumber in range(27):
                for word in array2[cellNumber]:
                    sortedWords.append(word)
                    numberOfPrimitiveOperations += 1    # 1 assignment operation

        # After the third pass (or 5th pass, 7th pass, 9th pass, ...)
        # After all passes that write to array 1 (except the first pass)
        else:
            numberOfPrimitiveOperations += 2    # 1 modulus operation and 1 comparison operation

            # The list of words is already sorted in ascending order by reading from the first cell to the last cell
            # in array 1
            for cellNumber in range(27):
                for word in array1[cellNumber]:
                    sortedWords.append(word)
                    numberOfPrimitiveOperations += 1    # 1 assignment operation
        # Reorder end

    # Word sorting algorithm end

    return numberOfPrimitiveOperations

```

Figure 4.13 Primitive operation counters for reorder

Firstly, an empty array is initialized to hold the sorted list of words. This involves 1 variable assignment operation. So, as shown in Figure 4.13, the number of primitive operations is incremented by 1.

After that, the programme determines whether the last pass is an even number pass or an odd number pass by checking the divisibility of number of passes by 2. This involves 1 modulus operation and 1 comparison operation because the programme compares the remainder of the number of passes divided by 2 with 0. If the remainder of the number of passes divided by 2 is equal to 0, then the programme executes the code in the if block, otherwise the programme executes the code in the else block. So, no matter the programme executes the code in the if block or the else block, the number of primitive operations involved is 2. So, as shown in Figure 4.13, for both the if block and the else block, the number of primitive operations is incremented by 2.

In the if block (last pass is even number pass), the numbers are read, cell by cell, from array 2, whereas in the else block (last pass is odd number pass), the numbers are read, cell by cell, from array 1. In both cases, for every word, 1 variable

assignment operation is involved because the word read is appended to the “sortedWords” array. So, as shown in Figure 4.13, in both cases, for each word, the number of primitive operations is incremented by 1.

Also shown in Figure 3.10, after executing reorder, the number of primitive operations is returned and the “count_primitive_operation_in_word_sorting()” function is ended.

4.2 Generating random unsorted list of words with random length

To conduct the experiment to count the number of primitive operations of the word sorting algorithm for a few different values of N (number of words), a random unsorted list of words of specified number of words with random length shorter or equal to the maximum length needs to be generated and passed to the “count_primitive_operation_in_word_sorting()” function as the input. It is laborious to manually create an unsorted list of words because the number of words (N) in the experiment is large. So, as a solution, a function that can auto-generate random list of words is implemented. Figure 4.14 shows the code for the function mentioned.

```python

# Function to randomly generate a list of words shorter than or equal than a maximum length
# Takes 2 inputs, N and MAXIMUM_LENGTH
# N is the number of words in the list, MAXIMUM_LENGTH is the maximum length (number of characters) in each word
# Returns a list of words (an array of string with only lower case letter)
def generate_random_list_of_words(N, MAXIMUM_LENGTH):  10 usages  new *
    # Initialize an empty list of words
    words = []

    # An array of the permitted letters to choose from (used when string generation)
    # The permitted letters are all lower case letters only
    permittedLetters = ['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']

    # Generate N words
    for _ in range(N):
        # Randomly decide the length (number of characters) of the word
        length = random.randint( a: 1, MAXIMUM_LENGTH)

        # Initialize a variable to store the word
        word = ''

        # Generate a number with length number of characters
        for _ in range(length):
            # Generate a random number from 0 to 25 (decide which alphabet from permittedLetters is chosen)
            indexOfChosenLetter = random.randint( a: 0,  b: 25)

            # The character of the current position
            character = permittedLetters[indexOfChosenLetter]

            # Append the character to word
            word = word + character

        # Append the word to the main list
        words.append(word)

    return words

```

Figure 4.14 Generate random unsorted list of words with random length

As shown in Figure 4.14, the “generate_random_list_of_words()” function takes N, which is the number of words in the list, and MAXIMUM_LENGTH, which is the maximum length (maximum number of characters) of all words in the list, as input. The function returns a random unsorted list of words with random length which is shorter or equal to the maximum length inputted. All the words only contain lower case characters.

As shown in Figure 4.14, firstly, an empty array named “words” is initialized to hold the words generated. After that, an array named “permittedLetters” containing lower case character from ‘a’ to ‘z’ is declared. This array contains all the permitted characters. This array is used when creating the word and the usage will be explained later. For each word, firstly, a random integer which lies between 1 (inclusive) and the maximum length (inclusive) is generated. This integer is the length of the word. After that, a string variable named “word”, containing an empty string is initialized to hold the generated word string. For each character in each word, an integer between 0 and

25 is randomly generated. The integer is the index of the chosen letter in the array “permittedLetters”. For example, if the random index is 0, then the chosen letter is ‘a’, else if the random index is ‘24’, then the chosen letter is ‘y’. Then, the letter string is appended to the string variable “word”. For each word, after the word of the specified length is generated, the word is appended to the “words” array. After every word in the list is generated, the “words” array is returned.

4.3 Conducting the experiment

The experiment is conducted with maximum number of characters (maximum length) as the fixed variable, number of words (N) as the manipulated variable and number of primitive operations as the observed variable. The experiment is conducted by running the “count_primitive_operation_in_word_sorting()” function with unsorted list of words of different number of words (N). The number of words used in the experiment are 100, 200, 300, 400, 500, 600, 700, 800, 900 and 1000. Figure 4.15 shows the code for generating random unsorted list of words with different number of words (N).

```python

# Experiment start
# Maximum number of characters (maximum length) = 10 (Fixed variable)
MAXIMUM_LENGTH = 10

# Generate a random list of 100 words with maximum 10 characters
words_N100 = generate_random_list_of_words( N: 100, MAXIMUM_LENGTH)

# Generate a random list of 200 words with maximum 10 characters
words_N200 = generate_random_list_of_words( N: 200, MAXIMUM_LENGTH)

# Generate a random list of 300 words with maximum 10 characters
words_N300 = generate_random_list_of_words( N: 300, MAXIMUM_LENGTH)

# Generate a random list of 400 words with maximum 10 characters
words_N400 = generate_random_list_of_words( N: 400, MAXIMUM_LENGTH)

# Generate a random list of 500 words with maximum 10 characters
words_N500 = generate_random_list_of_words( N: 500, MAXIMUM_LENGTH)

# Generate a random list of 600 words with maximum 10 characters
words_N600 = generate_random_list_of_words( N: 600, MAXIMUM_LENGTH)

# Generate a random list of 700 words with maximum 10 characters
words_N700 = generate_random_list_of_words( N: 700, MAXIMUM_LENGTH)

# Generate a random list of 800 words with maximum 10 characters
words_N800 = generate_random_list_of_words( N: 800, MAXIMUM_LENGTH)

# Generate a random list of 900 words with maximum 10 characters
words_N900 = generate_random_list_of_words( N: 900, MAXIMUM_LENGTH)

# Generate a random list of 1000 words with maximum 10 characters
words_N1000 = generate_random_list_of_words( N: 1000, MAXIMUM_LENGTH)

# Array of N (Manipulated variable)
array_N = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]

```

Figure 4.15 Generate random unsorted list of words with different number of words

As shown in Figure 4.15, the maximum number of characters (maximum length) is fixed as 10. This reflects the fixed variable, maximum number of characters, which is not changed throughout the experiment. This means that the number of characters in each word in the list varies, but cannot exceed the maximum number of characters which is 10. Then, by using the “generate_random_list_of_words()” function, the programme generates random unsorted lists of words with 100 numbers, 200 numbers, 300 numbers, 400 numbers, 500 numbers, 600 numbers, 700 numbers, 800 numbers, 900 numbers and 1000 numbers. This reflects the manipulated variable, number of words, which is purposely changed throughout the experiment. The number of words (N) is recorded in an array named “array_N” for the purpose of drawing the graph of number of primitive operations vs N.

After all lists of words is ready, the experiment is conducted to observe the number of primitive operations. Figure 4.16 and Figure 4.17 show the code for conducting the experiment.

```python

# Initialize array of number of primitive operations (Observed variable)
array_numberOfPrimitiveOperations = []

# Count the number of primitive operations for sorting 100 words
numberOfPrimitiveOperations_N100 = count_primitive_operation_in_word_sorting(words_N100, MAXIMUM_LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N100)

# Count the number of primitive operations for sorting 200 words
numberOfPrimitiveOperations_N200 = count_primitive_operation_in_word_sorting(words_N200, MAXIMUM_LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N200)

# Count the number of primitive operations for sorting 300 words
numberOfPrimitiveOperations_N300 = count_primitive_operation_in_word_sorting(words_N300, MAXIMUM_LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N300)

# Count the number of primitive operations for sorting 400 words
numberOfPrimitiveOperations_N400 = count_primitive_operation_in_word_sorting(words_N400, MAXIMUM_LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N400)

# Count the number of primitive operations for sorting 500 words
numberOfPrimitiveOperations_N500 = count_primitive_operation_in_word_sorting(words_N500, MAXIMUM_LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N500)

# Count the number of primitive operations for sorting 600 words
numberOfPrimitiveOperations_N600 = count_primitive_operation_in_word_sorting(words_N600, MAXIMUM_LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N600)

# Count the number of primitive operations for sorting 700 words
numberOfPrimitiveOperations_N700 = count_primitive_operation_in_word_sorting(words_N700, MAXIMUM_LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N700)

# Count the number of primitive operations for sorting 800 words
numberOfPrimitiveOperations_N800 = count_primitive_operation_in_word_sorting(words_N800, MAXIMUM_LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N800)

```

Figure 4.16 Conduct experiment (part 1 of 2)

```python

# Count the number of primitive operations for sorting 900 words
numberOfPrimitiveOperations_N900 = count_primitive_operation_in_word_sorting(words_N900, MAXIMUM_LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N900)

# Count the number of primitive operations for sorting 1000 words
numberOfPrimitiveOperations_N1000 = count_primitive_operation_in_word_sorting(words_N1000, MAXIMUM_LENGTH)
array_numberOfPrimitiveOperations.append(numberOfPrimitiveOperations_N1000)
# Experiment end

```

Figure 4.17 Conduct experiment (part 2 of 2)

As shown in Figure 4.16, firstly, an empty array named “array_numberOfPrimitiveOperations” is initialized to hold the number of primitive operations recorded for each number of words (N) in the experiment. As shown in Figure 4.16 and Figure 4.17, the array of 100 words, the array of 200 words, the array of 300 words, the array of 400 words, the array of 500 words, the array of 600 words, the array of 700 words, the array of 800 words, the array of 900 words and the array of 1000 words are passed into the “count_primitive_operation_in_word_sorting()” function with the maximum number of characters (maximum length) fixed at 10. For

each set of the experiment, the number of primitive operations is appended to the array “array_numberOfPrimitiveOperations”.

4.4 Displaying the experiment result and plotting the graph

After the experiment is conducted, the experiment result is displayed by printing out the number of primitive operations for each N (number of words) and plotting a graph of number of primitive operations vs N. Figure 4.18 shows the code for displaying the experiment result and plotting the graph.

```
# Print the fixed variable, the manipulated variable and the observed variable
print(f'Fixed variable: Maximum number of characters (MAXIMUM_LENGTH): {MAXIMUM_LENGTH}')
print(f'Manipulated variable: Number of words (N): {array_N}')
print(f'Observed variable: Number of primitive operations: {array_numberOfPrimitiveOperations}')

# Plot the graph of number of primitive operations vs number of words (N)
# The manipulated variable (x)
N = np.array(array_N)

# The observed variable (y)
numberOfPrimitiveOperations = np.array(array_numberOfPrimitiveOperations)

# Plot the line
plt.plot( *args: N, numberOfPrimitiveOperations)

# Specify the label of the x-axis
plt.xlabel('N')

# Specify the label of the y-axis
plt.ylabel('Number of primitive operations')

# Specify the title of the graph
plt.title('Graph of Number of Primitive Operations vs N for Word Sorting Algorithm')

plt.show()
```

Figure 4.18 Display experiment result and plot graph

As shown in Figure 4.18, the data associated with the fixed variable, which is maximum number of characters (maximum length), the manipulated variable, which is number of words (N), and the observed variable, which is number of primitive operations, are printed to the user.

After that, as shown in Figure 4.18, the data of the manipulated variable (number of words) (N) (in array form), and the data of the observed variable (number of primitive operations) (in array form) are converted into Numpy arrays. The Numpy arrays containing the manipulated variable data (number of words) (N) and observed variable data (number of primitive operations) is fed into the Matplotlib.pyplot function

“plt.plot” to plot a line graph. The manipulated variable, N (number of words), is set as the x variable, whereas the observed variable, number of primitive operations, is set as the y variable. Then, before showing the graph, the x axis label of the graph is set as “N”, whereas the y axis label of the graph is set as “Number of primitive operations”, whereas the title of the graph is set as “Graph of Number of Primitive Operations vs N for Word Sorting Algorithm”.

4.5 Running the programme

Figure 4.19 shows the output of the programme.

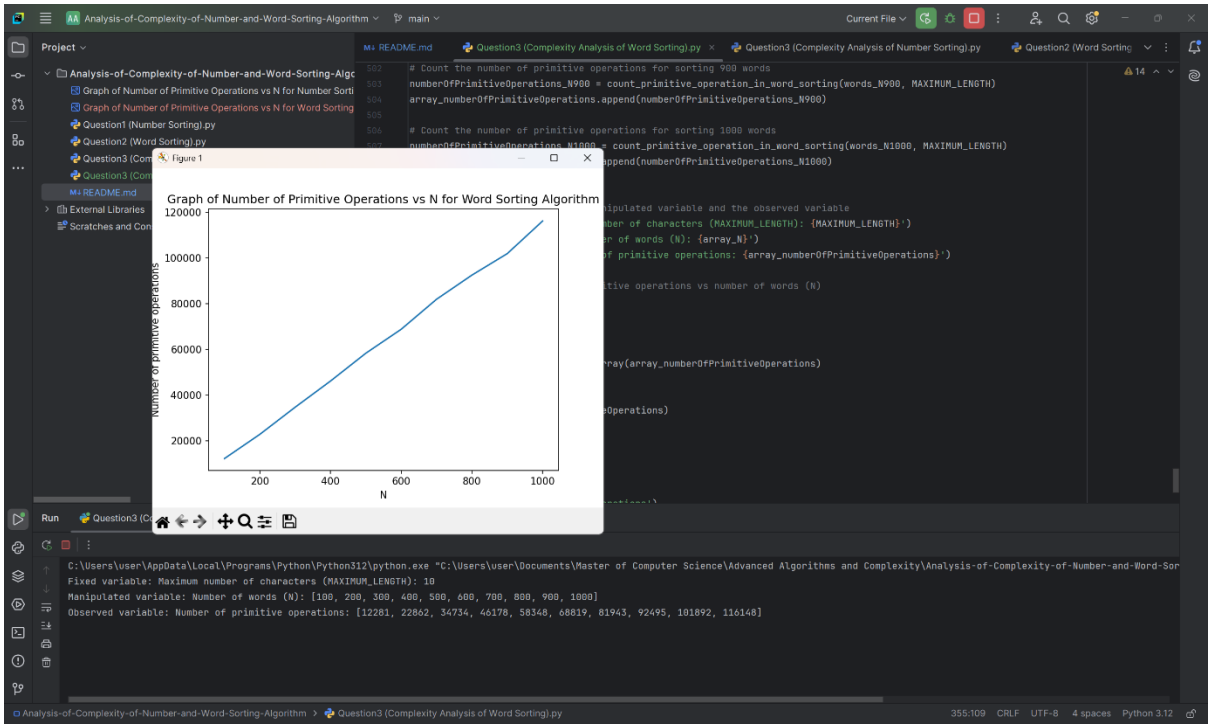

Figure 4.19 Programme output

As shown in Figure 4.19, as the output of the programme, the numbers of primitive operations for each N (number of words) are printed and a graph of number of primitive operations vs N is plotted.

4.6 Determining time complexity based on the experiment result

Table 4.1 records the experiment result by listing the number of primitive operations for each number of numbers (N).

Table 4.1 Number of primitive operations for different number of words (N)

Number of words (N)	Number of primitive operations
100	12281
200	22862
300	34734
400	46178
500	58348
600	68819
700	81943
800	92495
900	101892
1000	116148

As shown in Table 4.1, it can be said that, as N (number of words) increases, the number of primitive operations also increases. Figure 4.20 is the graph of number of primitive operations vs N (number of words) plotted based on the experiment result as in Table 4.1.

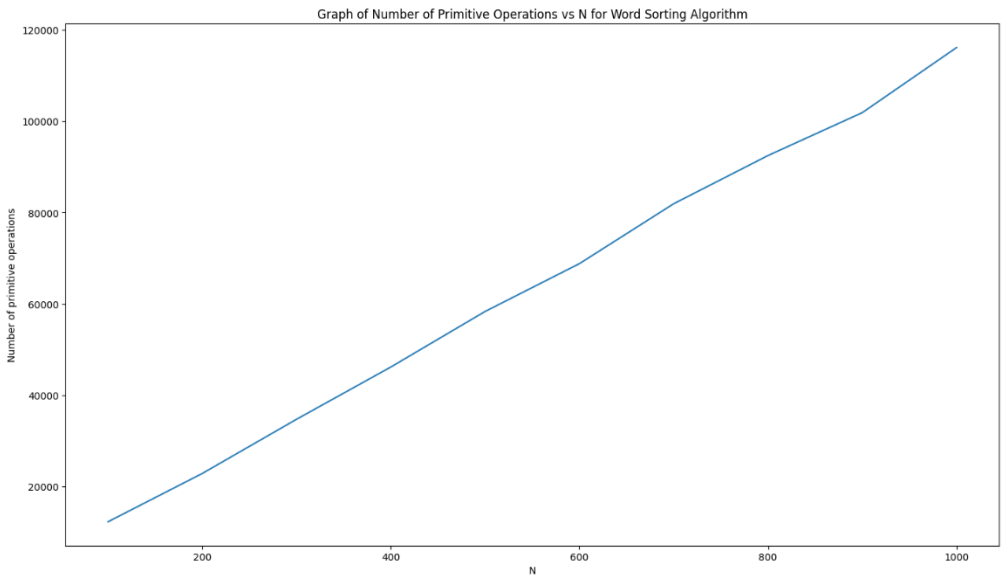

Figure 4.20 Graph of number of primitive operations vs N for word sorting algorithm

As shown in Figure 4.20, the graph of number of primitive operations vs N (number of words) is a line which is nearly straight. In the graph, the number of primitive operations increases steadily and linearly when N increases. This shows that the relationship between the number of primitive operations and N is linear. Since the

relationship between the number of primitive operations and N is linear, it can be said that the time complexity of the word sorting algorithm is big-O O(N).