Algorithm and Data Structures

Assignment 6: Report

Name: Maeda Taishin

Student ID: 1W21CF15

Date: 23/05/2023

Shell Sort Algorithm

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;
public class ShellS {
    Run | Debug
    public static void main(String[] args){
        try{
            Path file = Paths.get (first:"testdata-sort-1.txt");
            List<String> stringData = Files.readAllLines(file);

            int[] arry = new int[stringData.size()];
            for (int i = 0; i<arry.length; i++){
                arry[i] = Integer.parseInt(stringData.get(i));
            }
            int n = arry.length;
            long start = System.currentTimeMillis();
            shellS(arry, n);
            long end = System.currentTimeMillis();
            System.out.println(x:"Sorted Array: ");
            printArry(arry);
            System.out.println("Execution time: " + (end - start));

        }catch (IOException e){
            e.printStackTrace();
        }
    }
    private static void shellS(int[] arry, int n){
        for(int h=n/2; h>0; h/=2){
            for(int i=h; i<n; i++){
                int tmp = arry[i];
                int j;
                for(j=i-h; j>=0 && arry[j]>tmp; j-=h){
                    arry[j+h] = arry[j];
                }
                arry[j+h] = tmp;
            }
        }
    }
    private static void printArry(int[] arry){
        for(int i=0; i<arry.length; i++){
            System.out.println("x[" + i + "]=" + arry[i]);
        }
        System.out.println();
    }
}

```

Code Explanation:

Main function:

Line: 9

Initially, the main function will be described. Talking about reading a text file, the idea is that we first assumed the text file, “testdata-search.txt” in this case, is the same directory

Line: 10

Next, we needed to read all lines in the files provided and store data in a list if string.

Lines: 12-15

Then, we needed to convert the list that is recognized as a string in this case to an integer array. In order to do that, several Java libraries are needed to be stored at the beginning of the code (Lines: 1-5). Lastly, the code inside the main function will be placed inside the try-catch exception which allows us to define blocks of codes to be tested or executed if an error occurs (Lines: 8 and 24-25). After that, a variable called "n" is created to represent the size of the array, as the program starts counting the first element as the zeroth position. The function in this program is the sorting function using the Shell sort algorithm which will be discussed later.

Lines: 17&19

Furthermore, several syntaxes are included to measure the execution time of the program after calling the select sorting function. For instance, the data type "long" is used since it contains the minimum value of -2 to the power of 63 and a maximum value of, 2 to the power of 63, minus 1, so the program can return the time in milliseconds when calculating the time interval. The program is designed to record the current time right before entering the function in line 17 and record again after the function is called and executed. Next, the amount of time spent will be calculated by subtracting the "end" and "start" which are the time after and before respectively and printing it out in line 22 of this program.

Sorting function:

Moving on to the explanation about the function, in this program, the array "arry" and the array's length "n" are passed to the Shell sort function called "shellS". Generally speaking, the Shell sort is an upgraded version of the insertion sort algorithm. The idea is to swap far elements in the unsorted array instead of comparing contiguous items. This can be done by distributing the initial lists (array) into sub-lists (array) with an interval of "h". And then, the method of insertion is applied to sort those sub-lists (array). Note that, "h" will be a variable created to indicate the span between items and the process of sorting will keep repeating until the interval "h" becomes 1.

Line: 29

A for-loop is defined, and "h" is initialized to equal n/2, which is half of the array's length. The value of "h" will decrease by half every time the loop is entered as long as it is greater than zero.

Line: 30

Then, a nested loop is defined which keeps running from i=h until “i” is smaller than “n” in order to search a current element in the position “i”th.

Lines: 31-32

Next, a temporary variable “tmp” is created to store that current element “arry[i]” in line 31, and the variable “j” is also created in line 32.

Line: 33

Another loop is defined which runs from j=i-h by decrementing the value of “j” by “h” to the condition when “j” is greater or equal to 0 and “arry[j]” which is the element in the “j”th position is greater than the temporarily stored value “array[i]”.

Line: 34

The action inside the loop is the action of shifting elements in the array and swapping the element at the “j+h”th position with the “j”th position.

Line: 36

Finally, the temporary variable “tmp” is inserted at the element at “j+h”th position (arry[i+h]) which is done outside the loop (line: 33)

For the comment codes (lines: 20-21 and 40-45):

I just used them to make sure that the code actually executes the sorted array by using the testdata-sort-1.txt file. Since it contains the smallest amount of data (100) I was able to make the program print it out to see and confirm the outcomes.

Execution Time:

test-data-sort-1.txt

```
x[94]=74
x[95]=76
x[96]=76
x[97]=77
x[98]=78
x[99]=79

Execution time: 0
chawinwaimaleongora-ek@Chawins-MacBook-Air Code %
```

For the first data file, it says that it took 0 milliseconds to sort. However, it is not exactly 0, but the program can compute extremely fast for the small array. Therefore, I changed the time measurement to nanoseconds, and the time taken is 23625 nanoseconds which is equal to 0.023625 milliseconds.

```
17          long start = System.nanoTime();
18          shellS(arry, n);
19          long end = System.nanoTime();
```

```
Execution time: 23625
chawinwaimaleongora-ek@Chawins-MacB
```

test-data-sort-2.txt

```
Execution time: 15
chawinwaimaleongora-ek@Chawins-Mac
```

For the second data file, it took 15 milliseconds to sort.

test-data-sort-3.txt

```
Execution time: 164
chawinwaimaleongora-ek@Chawins-MacBo
```

For the third data file, it took 164 milliseconds to sort.

test-data-sort-4.txt

```
Execution time: 19
chawinwaimaleongora-ek@Chawins-MacBo
```

For the fourth data file, it took 19 milliseconds to sort.

Discussion on the performance by comparing with insertion sort:

Algorithm	test-data-sort-1.txt	test-data-sort-2.txt	test-data-sort-3.txt	test-data-sort-4.txt
Insertion	0.035334	2062	205779	4
Shell	0.023625	15	164	19

Table 1. Execution time in milliseconds for Insertion sort and Shell sort algorithms

As can be seen from the execution results in Table 1, the shell sort algorithm took significantly less time to execute than the insertion sort algorithm. However, the shell sort algorithm took more time to sort for the test-data-sort-4.txt which will be discussed later. The idea of swapping non-consecutive elements by classifying them into larger interval which reduces the gaps between the elements and their position and then sorting with a shorter interval can potentially reduce the number shifting in a huge amount of data. For the first 3 data files, the element inside is randomized, so applying the idea of distributing into intervals can make the program execute faster than the insertion sort algorithm. However, elements in the last file data (test-data-sort-4.txt) are almost being sorted from its initial stage and only a few swaps are required. Thus, sorting with larger intervals require more unnecessary comparisons which results in taking more time to execute than the insertion sort algorithm.

Discussing time complexity, the best-case complexity is $O(n\ log\ n)$ as when the data is already being sorted, the inside nested loop will not have to shift anything. For the average-case complexity, the time complexity is $O(n\ log\ n^2)$ as when the elements are arranged not properly neither ascending nor descending. However, things will be different when elements are required to be arranged in reverse order. For example, the condition happens when the elements are arranged in descending order, and it is necessary to be sorted in ascending order. Hence, the situation will be the worst-case complexity as the innermost loop always iterates until “j” become less than 0 with the time

complexity of $O(n^2)$. In addition, there are several other gap sequences that will lead to a better worst-case time complexity proposed after the first version made by Donald L. Shell in 1959 as follows:

1. Knuth Sequence: 0, 1, 4, 13, 40, 121, 364, ….

   Defined as $(\frac{3^k-1}{2})$ with a time complexity of $O(n^{\frac{3}{2}})$.

2. Hibbard Sequence: 0, 1, 3, 7, 15, 31, ….

   Defined as $2^k - 1$ with a time complexity of $O(n^{\frac{3}{2}})$.

3. Sedgwick Sequence: 1, 8, 23, 77, 281, 1073, 4193, ….

   Defined as $4^k + 3 \cdot 2^{k-1} + 1$ with a time complexity of $O(n^{\frac{4}{3}})$

Note that, k is the iteration number.