Assignment #6

Rikiya Takehi
1W21CS04-6
Algorithms and Data Structures
23/05/2022

Problem Statement

Write a shell sort algorithm in java programming. Implement data, “testdata-sort-1.txt”, “testdata-sort-2.txt”, “testdata-sort-3.txt” and “testdata-sort-4.txt” for the shell sorting. For each test data, measure the time taken to sort the program in milliseconds.

Programs

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;
import java.util.Scanner;

public class ShellSort{
    static void shellSort(int[] a, int n){
        for(int h=n/2; h>0; h/=2){
            for(int i=h; i<n; i++){
                int tmp=a[i];
                int j;
                for(j=i-h; (j>=0)&&(a[j]>tmp);j=j-h){
                    a[j+h]=a[j];
                }
                a[j+h]=tmp;
            }
        }
    }

    public static void main(String[] args){
        try{
            Path file=Paths.get("./testdata-sort-2.txt");
            List<String>stringData=Files.readAllLines(file);
            int[] array=new int[stringData.size()];
            for(int i=0; i<array.length; i++){
                array[i]=Integer.parseInt(stringData.get(i));
            }
            Scanner stdIn = new Scanner(System.in);
            System.out.print("The number of elements?: 100000\n");
            int nx = 100000;
            int[] x = new int[nx];
            for (int i = 0; i < nx; i++) {
                System.out.print("x[" + i + "] : "+array[i]+"\n");
                x[i] = array[i];
            }

            System.out.println("Shell sort");

            long start = System.currentTimeMillis();
            shellSort(x, nx);

            System.out.println("Sorted array");
            for (int i = 0; i < nx; i++){
                System.out.println("x[" + i + "]=" + x[i]);
            }
            long end = System.currentTimeMillis();
            System.out.println("Execution time: " + (end - start));
        }catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Program explanation

Shell sort is a program where the program makes several sub-lists of all items, so that the insertion search could be used for small numbers of values. As the program moves on, each sub-list would have the following numbers of values: 2, 4, 8, 16, …, N (N is the number of values in the whole data). Thus, the number of sub-lists would decrease as shown: N/2, N/4, …, 2, 1. This program below, on line 10 states this.

```
for(int h=n/2; h>0; h/=2){
```

The rest of the programs (lines 11-18) are the exact same as the program for insertion search. Insertion search reads the data from the left side, and it swaps values whenever a value is larger than the value on its left. "tmp" on line 12 shows the leftmost value of the unsorted sequence. Lines 14-16 compares the numbers and swaps if the number on the left is greater.

In line 42, it records the exact time using the following code and states "start".

```
long start = System.currentTimeMillis();
```

In line 48, it records the time again using the following code and states "end". Then in line

```
long end = System.currentTimeMillis();
System.out.println("Execution time: " + (end - start));
```

49, it shows the time taken to complete the sorting by subtracting the time recorded as "end" by time recorded at "start".

Experimental settings

Input files:

"testdata-sort-1.txt", "testdata-sort-2.txt", "testdata-sort-3.txt", "testdata-sort-3.txt"

Results

Test Data	Time taken for shell sort (ms)	Time taken for insertion sort (ms)
Testdata sort 1	2	2
Testdata sort 2	239	1365
Testdata sort 3	1764	98083
Testdata sort 4	1943	1420

Discussion of the Results

The time taken to sort testdata-sort-1.txt was the exact same between shell sort and insertion sort. Perhaps one was faster than the other, but the difference is so small that it is difficult to measure.

For the time taken for sorting “testdata-sort-2.txt”, the shell sort was faster by 1126 milliseconds. The ratio was 239/1365 ≒ 0.175. The time taken for shell sort is 0.175 times the time taken for the insertion sort.

The time taken to sort “testdata-sort-3.txt”, the shell sort was faster by 96319 milliseconds. The ratio is 1764/98083≒0.0180. The time taken for shell sort is 0.018 times the time taken for the insertion sort in “testdata-sort-3.txt”.

For “testdata-sort-4.txt”, the shell sort was slower by 523 milliseconds. The ratio is 1943/1420≒1.37. Therefore, shell sort took more time than insertion sort by 1.37 times.

Since insertion sort can only move the value by one position, it takes a lot of time if the value needs to move from the end to the start. On the other hand, since shell sort takes values from all over the sequence to make a sub-list, values are able to move long positions in only one step. Therefore, logically thinking, if the values are placed completely randomly, it would be faster for the shell sort to sort out the sequence because the numbers are predicted to travel long distances to find its suitable positions. Probably, “testdata-sort-2” and “testdata-sort-3” are placed completely randomly. In contrast, “testdata-sort-4” is predicted to have minimal randomicity. “Testdata-sort-4” was probably already sorted out to a certain level.

Moreover, it would take significantly more time for insertion sort to sort large sequences, because the values have to travel significantly long positions. In contrast, the shell sort only needs to add a few more steps, as the sequence becomes larger. As in the case of “testdata-sort-2”, which has 100,000 data, the time taken for shell sort is 0.175 times that of insertion sort; in “testdata-sort-3”, which has 1,000,000 data, the time for shell sort is 0.018 times that of insertion sort. For the time taken between “testdata-sort-2” and “testdata-sort-3”, the shell sort’s ratio is only 1764/239=7.38, but the insertion sort is 98083/1365=71.9.