Lab 4 Analysis - Noboru Hayashi

I. Implementation

In this lab activity, I chose a fixed-size array for storing input data, and iteratively runs 2 types of sorts: Shell Sort and Heap Sort, using index-accesses without creating any other data structure.

Also while testing the shell sort, I chose the following sets of gaps which are pre-defined as gap-sequences before sorting. After doing some research, the 4th type of the sequence is called worse case gap sequence by Shell, which leads to the worst case scenario for the shell sort, having O(n^2) time complexity.

```
int[] gaps_1 = { 1, 4, 13, 40, 121, 364, 1093, 3280, 9841, 29524 };
int[] gaps_2 = { 1, 5, 17, 53, 149, 373, 1121, 3371, 10111, 30341 };
int[] gaps_3 = { 1, 10, 30, 60, 120, 360, 1080, 3240, 9720, 29160 };
int[] gaps_4 = { 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2028, 4096, 8192, 16384, 32768
};
```

For implementing a heapify function for heap sorting, I chose in-place heapify by swapping elements using indices like below. Therefore no extra tree is needed for the sorting and that allows the program to save memory space.

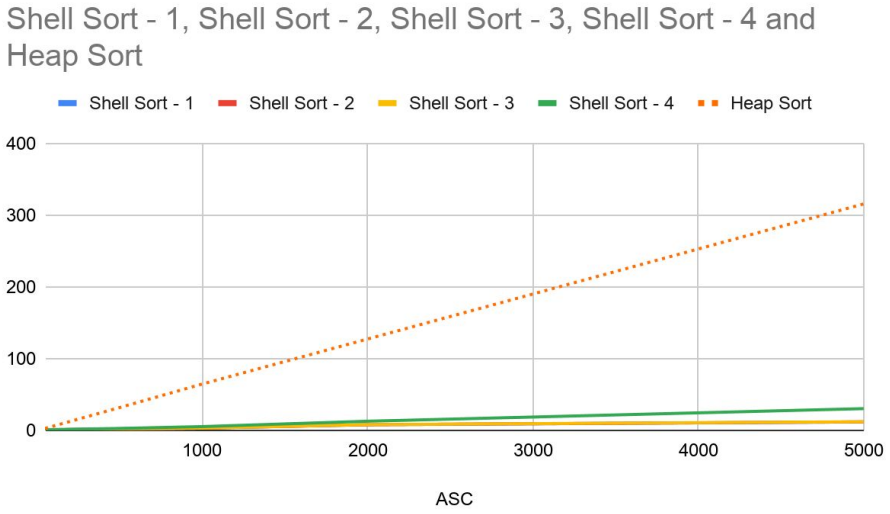
```
public static void heapify(int arr[], int rootIdx, int size) {
    int maxIdx = rootIdx;
    int lIdx = rootIdx * 2 + 1;
    int rIdx = rootIdx * 2 + 2;

    if (lIdx < size && arr[lIdx] > arr[maxIdx])
        maxIdx = lIdx;
    if (rIdx < size && arr[rIdx] > arr[maxIdx])
        maxIdx = rIdx;

    if (maxIdx != rootIdx) {
        int tmp = arr[maxIdx];
        arr[maxIdx] = arr[rootIdx];
        arr[rootIdx] = tmp;

        heapify(arr, maxIdx, size);
    }
}
```

II. Cost Analysis

After running a couple times with different sizes of the input files, I found the runtime is not stable, so I took the average time (in microsecond) of 5000 runs. The results are as below:

a) Ascending ordered data

ASC	Shell Sort - 1	Shell Sort - 2	Shell Sort - 3	Shell Sort - 4	Heap Sort
50	0.7002594	0.5922836	0.6821842	1.016045	2.7775
500	2.1213908	2.1779534	2.0931724	2.7666486	31.796005
1000	3.3518	3.1691518	3.1735826	5.3307044	64.628523
2000	7.9499052	7.968918	8.1069694	12.8182352	127.344379
5000	11.8544058	11.894706	12.1031064	30.3613932	315.594023



For ordered data, the shell sort reaches its best case: O(n) time complexity, while for Heap sort it has O(nlogn) complexity. So by comparing lines of time vs input size, I found all shell sort runtimes are quite smaller than one of Heap sort.

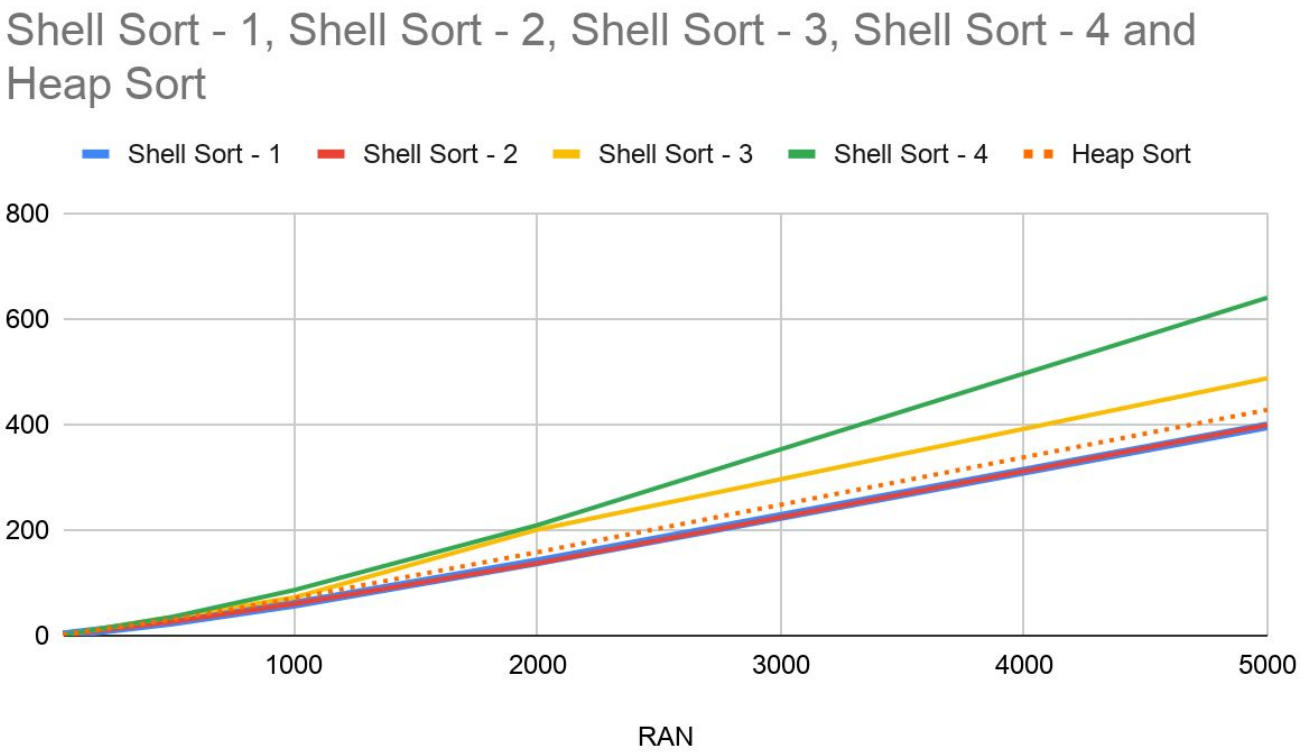
Also looking at the actual measured values of 4 types of shell sorts, the type 4 is less efficient than others. The reason would be gaps chosen for the 4th type are exponents of 2 and this would cause duplicate comparisons in sublists while iteratively choosing the gap value.

Besides that, another finding would be the time for shell sort is not actually linearly growing by the data size: For shell sort 1, 0.7 microseconds with 50 elements vs. 2.12 microseconds for the data 10 times larger. The possible reason would be there is some fixed overhead while running the sort containing in the total run time of the sort.

For space complexity, since shell sorts don't create any other array or list, so the cost would be O(1) for some helper constant values like gaps, temp values used for swapping. And same for heap sort, heapifying and sorting are achieved in-place on the original array, not creating any other data structure.

b) Randomly ordered data

RAN	Shell Sort - 1	Shell Sort - 2	Shell Sort - 3	Shell Sort - 4	Heap Sort
50	1.444295	1.4810712	1.350679	1.631745	1.9214606
500	25.791898	26.3517914	35.462102	35.4689588	30.1726264
1000	59.4918456	60.1298352	72.2924632	86.1648832	70.8042556
2000	139.8054552	137.141263	200.6561242	209.1769262	157.9326964
5000	397.301584	398.7826876	487.2008286	639.9023608	427.672043



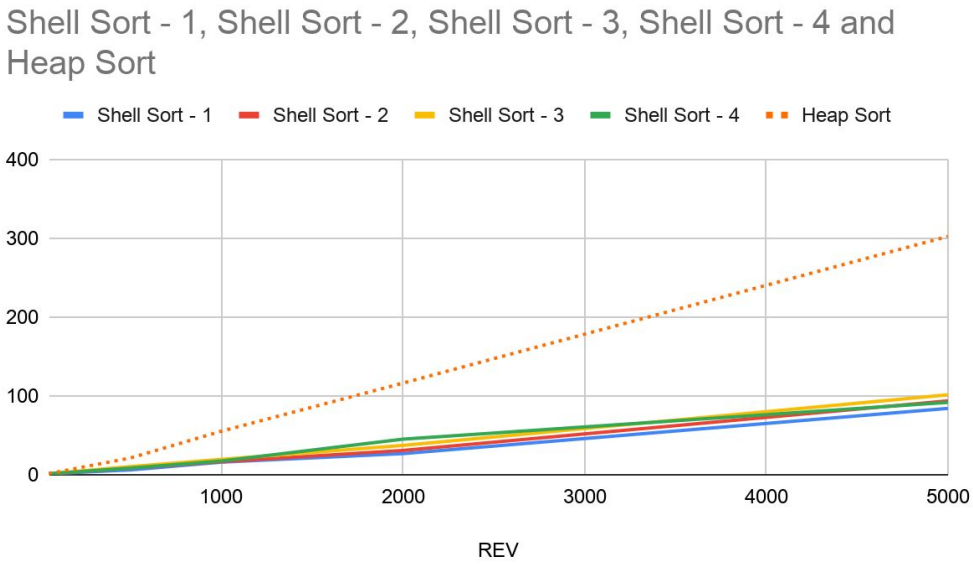
For randomly ordered data, the trends of the run times are similar for shell sorts and heap sort. As we can see the graph, the performance of the heap sort is better than shell sort with type 3 & 4 gaps.

Theoretically, the heap sort is performing with O(nlogn) complexity equally in best, average, worse cases.

And similar for the shell sorts, sorting random data would cost average time complexity O(nlogn). Comparing each gap sequence, I see the type 1(Knuth's sequence) has the best performance, while the type 4 (the worst-case gap sequence) has the worst.

c) Reverse Ordered Data

REV	Shell Sort - 1	Shell Sort - 2	Shell Sort - 3	Shell Sort - 4	Heap Sort
50	1.3288584	1.1458972	1.2760414	1.3082884	1.4213486
500	6.1431122	8.4797552	10.3885062	8.5086284	21.28888
1000	16.0081116	16.553259	19.522411	17.4489462	55.192681
2000	26.821997	30.8385426	37.2399646	45.1373428	116.2079016
5000	84.1643248	93.7048056	101.3929514	91.782346	302.2467192



For reversed order data input, the performance of the heap sort is similar as its time performance for ascending or random ordered data: O(nlogn).

Sorting reverse ordered data with shell sort costs less than using the heap sort, also more efficient than sorting random data by shell sort. So it seems after operating insertions sorts on sublists with gaps iteratively, the data becomes nearly sorted, and reaches the best case scenario for shell sort halfway.

III. Iteration & Recursion

This program accomplishes two types of sorting functionalities by iteratively processing insertion sorts on the sublist or heapify elements in the array.

If I had chosen to implement it in a recursive way, I possibly would start with allocating extra spaces to hold sublists of the array for insertion sorts operated within the shell sort, and recursively operating insertions sorts in sublists. For heap sort, generating a tree data is probably needed, so that the program can recursively use the tree as a heap and recursively parse it. In both situations, extra O(n) space is required for sorting.

IV. Enhancement

As I mentioned the section I, for shell sorting I chose the worst case gap sequence as the 4th type of the set of increments. There while sorting the random ordered data, shell sorting with this sequence is the least efficient among all types of sorts operated by the program.

V. Conclusion

In this lab assignment, I learned how to implement different types of sorting algorithms, and compare the time efficiency by clocking the system time and calculate the average duration of the sorting cost. In the future, I would like to examine more sorting algorithms's efficiency, and investigate asymptotic trends of the time complexity with the use of larger sizes of data.