Garrett McNeill

8/1/2021 | CS5008

HW: #8

**Input Lists & Program Notes:**

My main program, sortCompare.c is built using a makefile.

*To build/compile the program, type:*

```
make sort
```

Once sort has been compiled, run the program "sort" and pass a single argument for the value for the size of the array to generate and sort.

The argument passed will generate an array of integers in the given size within our sortCompare.c main program. The array values are generated using the random number generator (randInt), and utilize the same seed to guarantee that the contents of the unsorted array are the same for all algorithms. The seed we use in our program is "12345."

*How to run pass array size as argument example:*

```
$  ./sort 100000
```

The above command will generate an array of size 100,000, and will run it for all sorting algorithms. The program only accepts a single argument as an int value. No additional parameter beyond this value is required to run the program.

**Analysis Summary:**

To compare two integers, you must load both into a register and compare them with each other. To store one array of elements from one to another, you must load elements one-by-one that you are copying and then store them in the new location. The preceding is unchangeable when performing a sorting operation in c, but with that constant – we can observe distinctly different results in sorting arrays of various sizes given different algorithms.

In our experiment, we looked at two different sorting operations: a) insertion sort, and b) shell sort. Within looking at a shell sort operation, we implemented and observed four different intervals to use in sorting our arrays.

To compare the efficiency of each sorting algorithm, we looked at the following indicators: i) runtime duration, ii) number of comparisons, and iii) number of moves. It is worth noting that instructions can take more than one processor clock, so time as a measure alone may not be the best indicator.

What we found by running all algorithms (insertion sort, and the five algorithmic intervals for shell sort), different results were returned for varying sizes of n.

Given an array size of n=25,000 elements, we found:

- Insertion sort to be the least efficient, with a greater number of moves and longer run time duration than any of the intervals for the shell sort algorithms.
- A similar result in both time and number of operations for shellSort interval 2, and interval 4.

Given an array size of n= 50,000 elements, we found:

- Insertion sort to be the least efficient.
- A better result in shellSort3 compared to shellSort1 in number of operations and time
- A similar result in both time and number of operations for shellSort interval 2, and interval 4.

Given an array size of n=100,000 elements, we found:

- Insertion sort to be significantly less efficient than any of the shellSort intervals
- A similar result in both time and number of operations for shellSort interval 2, and interval 4.

In aggregate, it is worth noting that the results seem to indicate that the larger the size of the interval, the less operations it will take to sort a larger array. shellSort algorithm #4 had the largest interval, but was slightly outperformed by shellSort algorithm #2.

In summary, this experiment yielded the observation that: i) shellSort is more efficient than insertion sort for an array of random integers. ii) given a large value for n, a shellSort algorithm with a larger starting interval will be more performant than a shellSort algorithm with a smaller starting interval.

*Note: In solving this homework, I looked at a tutorialspoint example for shellsort and utilized it as my "sanity check" for what a shellsort operation looked like. It was not perfect and I made some modifications to it as commented in the shellSort.c file. The interval from this example is "3^1 -1*".

```
Tutorialspoint reference URL:
https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_program_in_c.htm
```

**Program Output: n = 25,000**

```
- - - - Insertion Sort - - - -
Sorting an array of size: 25000
154925644 comparisons, 154950631 moves
time taken (sec) =   0.460000

- - - - Shell Sort 1 - - - -
Algo: { 1.72n^(1/3), 1 }
Sorting an array of size: 25000
4336470 comparisons, 4461408 moves
time taken (sec) =   0.010000

- - - - Shell Sort 2 - - - -
Algo: { 2^i - 1 | 1 <= i <= [log n] }
Sorting an array of size: 25000
557676 comparisons, 805640 moves
time taken (sec) =   0.010000

- - - - Shell Sort 3 - - - -
Algo: { 2^i | 1 <= i <= [log n] }
Sorting an array of size: 25000
2299720 comparisons, 2572673 moves
time taken (sec) =   0.010000

- - - - Shell Sort 4 - - - -
Algo: { (3^1 - 1) / 2  | 1 <= i <= t where t is the smallest integer
         such that (3^(t-2) -1) / 2 >= n }
Sorting an array of size: 25000
515439 comparisons, 725682 moves
time taken (sec) =   0.010000
```

**Program Output: n = 50,000**

```
- - - - Insertion Sort - - - -
Sorting an array of size: 50000
621340848 comparisons, 621390833 moves
time taken (sec) =   1.880000

- - - - Shell Sort 1 - - - -
Algo: { 1.72n^(1/3), 1 }
Sorting an array of size: 50000
14284697 comparisons, 14534622 moves
time taken (sec) =   0.050000

- - - - Shell Sort 2 - - - -
Algo: { 2^i - 1 | 1 <= i <= [log n] }
Sorting an array of size: 50000
1228911 comparisons, 1774828 moves
time taken (sec) =   0.010000

- - - - Shell Sort 3 - - - -
Algo: { 2^i | 1 <= i <= [log n] }
Sorting an array of size: 50000
5489815 comparisons, 6085720 moves
time taken (sec) =   0.030000

- - - - Shell Sort 4 - - - -
Algo: { (3^1 - 1) / 2  | 1 <= i <= t where t is the smallest integer
         such that (3^(t-2) -1) / 2 >= n }
Sorting an array of size: 50000
1297994 comparisons, 1753713 moves
time taken (sec) =   0.010000
```

**Program Output: n=100,000**

```
- - - - Insertion Sort - - - -
Sorting an array of size: 100000
2491999279 comparisons, 2492099263 moves
time taken (sec) =   7.370000

- - - - Shell Sort 1 - - - -
Algo: { 1.72n^(1/3), 1 }
Sorting an array of size: 100000
40821323 comparisons, 41321231 moves
time taken (sec) =   0.140000

- - - - Shell Sort 2 - - - -
Algo: { 2^i - 1 | 1 <= i <= [log n] }
Sorting an array of size: 100000
2828320 comparisons, 4020142 moves
time taken (sec) =   0.030000

- - - - Shell Sort 3 - - - -
Algo: { 2^i | 1 <= i <= [log n] }
Sorting an array of size: 100000
15932524 comparisons, 17224333 moves
time taken (sec) =   0.070000

- - - - Shell Sort 4 - - - -
Algo: { (3^1 - 1) / 2  | 1 <= i <= t where t is the smallest integer
         such that (3^(t-2) -1) / 2 >= n }
Sorting an array of size: 100000
3135069 comparisons, 4102215 moves
time taken (sec) =   0.030000
```