Candace Holcombe-Volke
Lab 4

Lab 4 – Sorting Application Analysis

This analysis of the Lab 4 Sorting Application will briefly review the design of the program, and discuss the impact of using iterative methods for the sorting algorithms and the implication of the recursive alternatives, the use of statically allocated data structures for storing and searching, and compare the effects of order in the data, file size, and gap size in Shell sort on the overall runtime of the program. Furthermore, this analysis will propose future tests that can be assessed on the data and the relative insights they might provide, as well as lessons learned through this experience.

The sorting application makes use to two methods of sorting data, Heap Sort and Shell Sort. The program is initially executed from the main() method which is called from the ReadFile class. The first command line argument is stored as the output file location, and all subsequent arguments are read in, one by one, as input file locations. main() passes each input file to readFile(), which interprets the data, assigning any value that contains a space to inputSize, which is used to determine the size of the storage array, and inserting all other values into the array via a loop. Once all input from the file has been stored, four copies of the array are created, so that the original data can be passed to each iteration of the sorting test. First, heap_sort() is called immediately after starting the System clock. heap_sort() uses methods makeHeap(), swap(), and reheapDown() (additional methods are used to maintain max heap order) to complete the first phase of building the max heap. After the heap is build, heap_sort() makes use of swap() and reheapDown() to arrange data in ascending order, finally printing with printArray(). Control returns to readFile() which stops the System clock, and converts the timeElapsed into microseconds, finally printing the data to both the console and the specified output file for files of input size 50.

readFile() continues by building an array that contains the first twenty Knuth gap values, and calling shell.sort(). readFile() maintains the timers, and subsequently calls each of the additional three Shell Sort tests, using, in addition to the Knuth sequence, a series of spaced out prime numbers, a series of spaced out even numbers, and Hibbard's sequence.[1] ShellSort's sort() function compares the inputSize value to the elements in each gap value array, finding the smallest element whose value is larger than the length of the input data array, and then starting with the gap value that is two indexes lower. sort() proceeds to perform a modified insertion sort using the gap values determined, comparing and swapping values until the array has been fully sorted.

Initially, sort duration was measured in milliseconds, however many sorts returned results of 0 milliseconds. Subsequently, the time measurement unit was changed to calculate microseconds, which create distinct values allowing for comparison of tests.

Both sort methods make use of array storage structures. As the input files contain the exact input size, the size of the required space is known at initiation. Additionally, the array-based structure does not require additional storage space to hold forward and next pointers. More importantly (and frequently), both Heap Sort and Shell Sort make use of frequent exchanges. The random access feature of the array allows smooth swaps without the need to allocate space for, and manage, additional pointers as would be required with a linked structure. Particularly for the Shell Sort, the array also allows the sorting function to "skip" directly to the next element of the subfile (the k value gap) without iterating through each node of a linked list. This saves n-k movements for every pass through the file, which can become quite

[1] Hibbard's sequence is defined by the formula 2^k-1.

considerable for larger files. The array structure is also particularly supportive for Heap Sort because it allows for finding child and parent nodes directly by calculating the address within the array.[2]

This sorting program makes use of iterative methods to organize the inputs, however recursion is an alternative option. A recursive Heap Sort method would make use of a heapify() method which would take in the input array, the size of the array, and an array index. It would then assess whether the tree was already in heap order. If it determined that a child node had a greater value than its parent (for a max heap implementation), the function would recursively call the heapify() method with only the subtree as argument. Once the smallest sub tree has been heapified, the program can extend back outward. This process may require additional exchanges with sub trees, as those exchanges would not happen on the initial pass "down". This could result in a higher cost as the heapification process may require multiple passes to compare against subtree elements. Conversely, the iteration process manages each swap in the initial pass, swapping elements when children are larger than their parents. It is performed inside of a for loop, which iterates through all internal nodes, with a nested while loop to continue swapping elements until they reach their rightful location.

The results of the multiple tests assessing the complexity and duration of the series of sorts show some interesting findings. (See Table 1 and Graphs 1-3.) Table 1 shows the observations recorded for each of the sort tests performed, with all times calculated in microseconds. Graph 1 shows the performance of each type of sort at each input size interval for sorted data; Graph 2 shows the performance for reverse order data; Graph 3 shows performance for random (non-sorted) data. (Please note: the axis are set relative to each sort – they are not equal across data orders.)

The first and most striking observation is the performance observed using Heap Sort. Heap Sort far outperforms all Shell Sorts for all input sizes measured when data is in random or reverse-sorted order. This is somewhat to be expected, particularly for the reverse-sorted data, as reverse-sorted data is already in max heap form. This allows the max heap to complete without requiring any exchanges to maintain max heap rules. While Heap Sort is not sensitive to data order, achieving O(n logn) for all sorts, it can perform more efficiently (however only on the order of coefficients and constants) in reverse-order scenarios.

Conversely, Heap Sort quite dramatically underperforms compared to all Shell Sorts for ascending ordered data. This holds true even for very small input sizes (observed values as low as 50). Heap Sort is less efficient with small files, and never regains an advantage over the Shell Sorts for file sizes as great as 20,000. This, too, seems intuitive after understanding the Heap Sort execution. As noted above, reverse-order data already complies with max heap rules. On the other hand, ascending order data is completely non-compliant to max heap rules, requiring that all elements be swapped during the build heap phase. Again, while this is not enough added complexity to change the overall performance, it does present a disadvantage compared with Shell Sorts on sorted data (discussed below).

While the graphs appear to show Heap Sort taking much longer to complete the ascending data sort than the other two sorts, it is important to note the values of the y-axis. All sorts complete the ascending sort in just about 3000 microseconds or less, compared with about 100 times as long for a Shell

[2] This is accomplished by making use of the formula: (i-1)/2 for parent; (2i)+1 for left child; (2i)+2 for right child

Sort with even gaps on reverse-sorted data. Heap Sort, in fact, takes longer to complete the random sort, however *comparatively*, Heap Sort is very inefficient for sorting ascending data.

In contrast to the Heap Sort performance, Shell Sort (in general) underperforms compared to Heap Sort when sorting reverse-order and random data but is extremely efficient when sorting ascending data. As we know, the performance of Shell Sort can vary based on the k-value gaps used for the sorting subfiles. In the tests performed, in addition to the required gap values, the Hibbard test used a gap set which included: [1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 8191, 16383, 32767, 65535, 131071, 262143, 524287, 1048575]. (Only the appropriate subsection was applied according to the size of the input file, based on the Knuth example.)

In the test observations, particularly in Graphs 2 and 3, we can begin to make easy comparisons about the performance of Shell Sort utilizing varying gap values. As would be expected, the even gap set, which is composed of diminishing factors of previously used gap values, performs worst in the reverse-ordered and random data sets. As each successive pass through the Shell Sort uses a small factor gap, it encounters a situation where it is performing redundant validations/comparisons. When the sort reaches to the gap value 10 and begins to perform the comparisons and swaps, one-third of the elements it will execute on have already been compared. This reduces its efficiency when compared to alternative gap increments, such as the prime values.

Interestingly however, this does not hold true for input files of sizes 5,000 and smaller. For those sorts on random and reverse-ordered data, the even gaps perform quite similarly, and in some cases even more efficiently, than the prime gap set. To understand this, we need to examine the gap values used.

Comparisons for file size 50:

Even gaps used: [10, 1]; Duration: 34 microseconds

Prime gaps used: [5, 1]; Duration: 62 microseconds

Here we have a scenario where the initial gap value for the prime set is half the size of the initial gap value for the even set. Effectively, this means that as the shell sort makes comparisons across the initial array, the prime set is comparing and swapping more data, and doing so closer together than the even gaps. This is likely more costly during the initial pass than the even function, but less costly in comparison when the gap value drops to 1 because more of the input array is already more “nearly sorted”.

While this is the general idea for Shell Sorts, it appears that the input size is too small and the gaps too limited (only two values used) for the prime set to gain an advantage.

Comparisons for file size 2000:

Even gaps used: [360, 120, 60, 30, 10, 1]; Duration: 2021 microseconds

Prime gaps used: [373, 149, 53, 17, 5, 1]; Duration: 1093 microseconds

By the time the input size grows to 2000, we start to see the prime set pull ahead. The more exacting gaps (no redundant values as seen by smaller factors of earlier gaps seen in the even set) shows that it is creating an advantage. The first three values used for the primes and even set are relatively close in value, however we see that the third-to-last and second-to-last vary quite a bit (with the even set using gaps about twice the value of the primes). Perhaps this is adding a bit of advantage because the larger sets have already passed through “spreading out” the nearly sorted data.

While the even set appears to perform most poorly among all observed Shell Sorts, determining the most efficient Shell Sort for reverse-ordered and random ordered data is not quite as clear from the results. Knuth sequence gaps perform pretty well for reverse-sorted data between input sizes of 10000 and 20000; however, it appears that the prime gaps, in addition to Hibbard gaps will be more efficient for larger sizes. This pattern shows again in random data, however Knuth is outdone by all except for the even gaps by approximately size 15000.

The Hibbard sequence seems to be the clear winner among the Shell Sorts for files larger than 10000 for all data orders that were tested. (Note: Heap Sort is, between the two sorts, by far more efficient.) Moreover, while it is not necessarily faster than all Shell Sorts of small size, it is certainly competitive, taking just a few microseconds longer – negligible at such small file sizes. One caveat is that it appears in both cases that as the files grow larger, the prime sequence will become the most efficient of the Shell Sorts.

I found this project to be very helpful in understanding the impact of an algorithm's complexity on the performance and runtime when making decisions about which sorts to use in a program. While even the largest input files in my experiments are small compared to what might be used by an enterprise, it is enough to demonstrate the importance of properly and conscientiously selecting sorting methods. This project afforded me my first opportunity to use the system clock in Java, which is a tool I found very helpful. I think this will be quite beneficial in future tests and comparisons of various tools. When I first began testing, I did not realize that I had some of my output commands within the start and stop time clocks, which was causing some skewed results. This was easy to fix after becoming aware that it was happening.

Additionally, using some selective print statements throughout each of the sorting methods is a great way to understand exactly how each sort works. In the early stages of this project when I was tweaking the sort codes to suit my needs, I used some additional print statements to see each of the iterations that the sorts were making, and how they affected the data. Additionally, I maintained two counters to see monitor the number of comparisons and exchanges that were taking place within each sort, which was a helpful tool to enhance my understanding of the sorting methods.

In the future I would like to run additional tests on the input files to understand the impact of some additional factors. If I had the time, I would like to see how much the sort duration is impacted by using recursive codes on each of the files. My assumption is that the times would be quite a bit longer and that the intermediate arrangements of the data would differ. I would like to know the threshold of what an ordinary computer can handle for recursive sorts. (During the matrix determinant calculation, I attempted to recursively find the determinant of a 12x12 matrix, but my computer could not handle the request.)

I also think it would be interesting to analyze insertion sorts compared with Shell Sorts, as the Shell Sort's main advantage is creating a more optimal environment in which insertion sort can execute more efficiently. It appears that the Shell Sorts are on their way to producing the worst-case results, n(logn)^2, however more tests are needed to confirm that. I would develop more tests with larger file sizes to see whether the Shell Sort is increasing or leveling off.

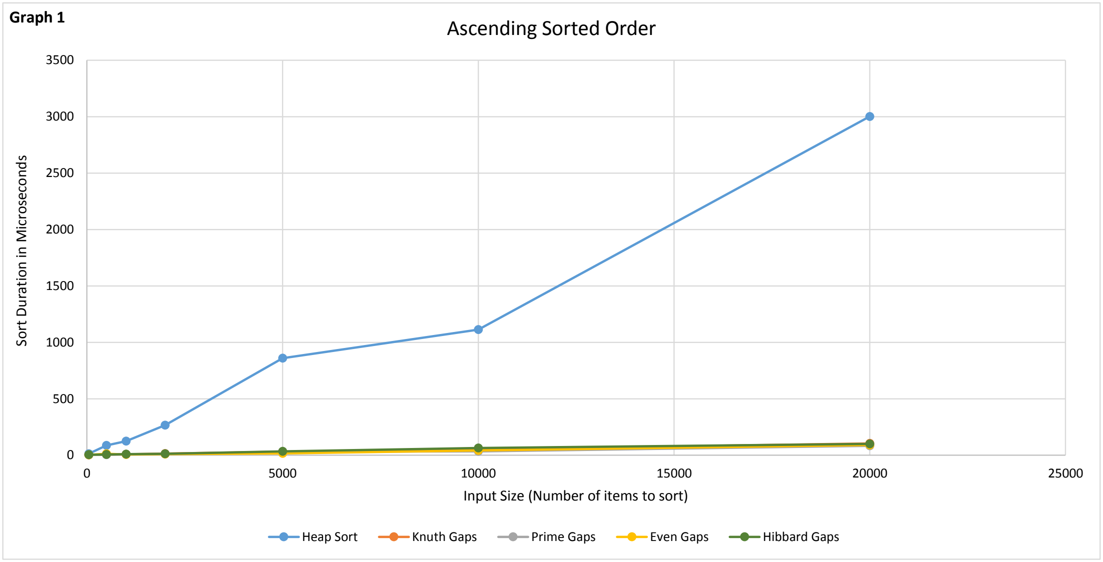
One additional change I would make in the future is to run the tests multiple times and average the times for each sort. This would allow me to account for any coincidental slow-down that may not be due to the algorithm, but perhaps some external resource drain that slows the processing.

Candace Holcombe-Volke
Lab 4
Table of Results

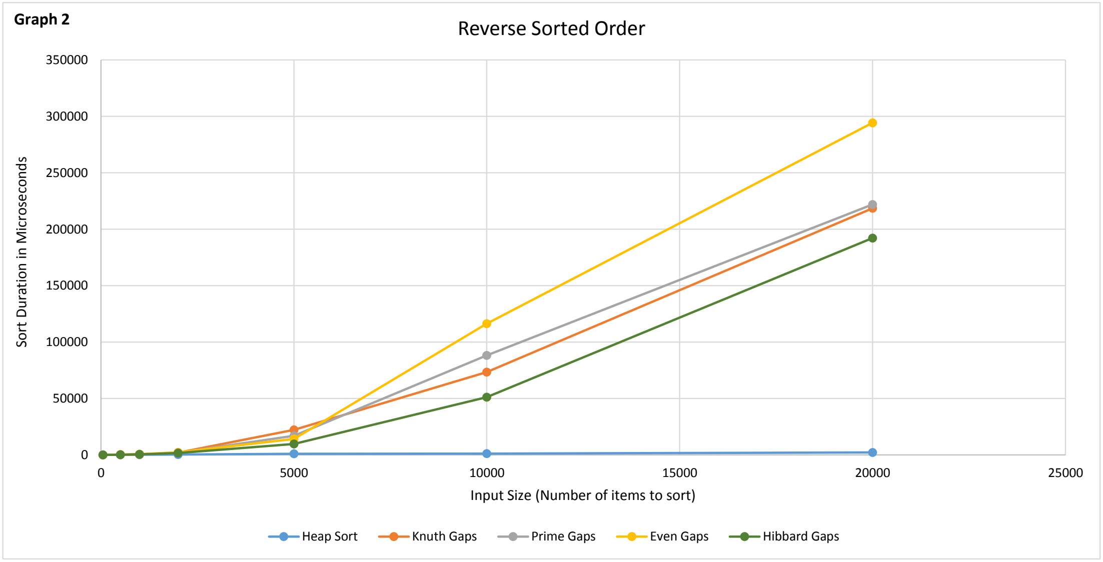
All Times are in microseconds

Direc-tion	**Size**	**Type**	**Time**	**Direc-tion**	**Size**	**Type**	**Time**	**Direc-tion**	**Size**	**Type**	**Time**
Ascending Sorted Order	**File Size 50**	Heap Sort	13	**Reverse Sorted Order**	**File Size 50**	Heap Sort	15	**Random Non-Sorted Order**	**File Size 50**	Heap Sort	87
		Knuth	3			Knuth	6			Knuth	66
		Primes	2			Primes	6			Primes	62
		Evens	3			Evens	7			Evens	34
		Hibbard	4			Hibbard	5			Hibbard	55
	File Size 500	Heap Sort	87		**File Size 500**	Heap Sort	81		**File Size 500**	Heap Sort	1112
		Knuth	6			Knuth	144			Knuth	3182
		Primes	5			Primes	139			Primes	1432
		Evens	13			Evens	275			Evens	392
		Hibbard	6			Hibbard	130			Hibbard	346
	File Size 1000	Heap Sort	125		**File Size 1000**	Heap Sort	123		**File Size 1000**	Heap Sort	855
		Knuth	7			Knuth	495			Knuth	292
		Primes	7			Primes	513			Primes	445
		Evens	6			Evens	623			Evens	500
		Hibbard	9			Hibbard	437			Hibbard	314
	File Size 2000	Heap Sort	265		**File Size 2000**	Heap Sort	351		**File Size 2000**	Heap Sort	513
		Knuth	14			Knuth	1910			Knuth	1096
		Primes	11			Primes	2033			Primes	1093
		Evens	10			Evens	2337			Evens	2021
		Hibbard	13			Hibbard	1591			Hibbard	963
	File Size 5000	Heap Sort	859		**File Size 5000**	Heap Sort	905		**File Size 5000**	Heap Sort	893
		Knuth	30			Knuth	22166			Knuth	6652
		Primes	31			Primes	16916			Primes	6387
		Evens	15			Evens	14052			Evens	7176
		Hibbard	34			Hibbard	9661			Hibbard	9527
	File Size 10,000	Heap Sort	1113		**File Size 10,000**	Heap Sort	1201		**File Size 10,000**	Heap Sort	1752
		Knuth	52			Knuth	73284			Knuth	28171
		Primes	34			Primes	88198			Primes	35715
		Evens	42			Evens	116262			Evens	37350
		Hibbard	64			Hibbard	51114			Hibbard	29931
	File Size 20,000	Heap Sort	3002		**File Size 20,000**	Heap Sort	2158		**File Size 20,000**	Heap Sort	3734
		Knuth	104			Knuth	218563			Knuth	122106
		Primes	83			Primes	221936			Primes	115744
		Evens	90			Evens	294318			Evens	138883
		Hibbard	100			Hibbard	192178			Hibbard	112756

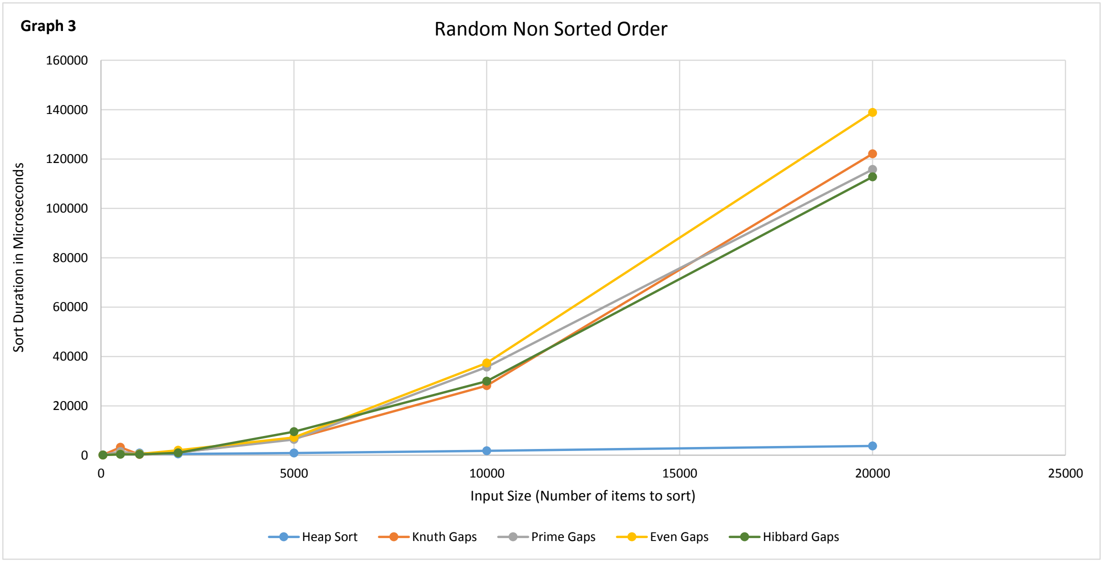
Graph 1



Graph 2



Graph 3



References

HeapSort.java. (2018). *Cs.fit.edu*. Retrieved 14 August 2018, from https://cs.fit.edu/~ryan/java/programs/sort/HeapSort.java

IDE | GeeksforGeeks | A computer science portal for geeks. (2018). *Ide.geeksforgeeks.org*. Retrieved 14 August 2018, from https://ide.geeksforgeeks.org/index.php

Shellsort & Algorithmic Comparisons. (2018). *Cs.wcupa.edu*. Retrieved 14 August 2018, from https://www.cs.wcupa.edu/rkline/ds/shell-comparison.html