**MOTIVATION**

Understanding algorithm design is fundamental to understanding the basic operations of computer science and software development. Though algorithms exist outside of digital computing, the age of modern computing has heralded a time wherein algorithms have become an integral part of the technology with which we interact every day. Resultantly, algorithms are now synonymous with their software implementations, and as Cormen et al. introduce in the third edition of *Introduction to Algorithms*, algorithms are seen as “tools for solving well-specified computational problems (5).” Such tools can be harnessed to effectively utilize finite computational resources and thereby solve complex problems, which without efficient algorithms would otherwise require inordinate amounts of labor or whose humanly computation would become irrelevant upon completion.

Aware that algorithms play a fundamental role in computer science and have innumerable practical applications in software development, computer science and software development curriculums consider an understanding of algorithms to be a foundational pillar in academic and professional training.

However, the study of algorithms is broad and unwieldly, as grappling with its many intricacies and applications is not possible within the time constraints of an academic term. Therefore, in an attempt to keep the study of algorithms tangible and germane, the current canonical approach in algorithm pedagogy is to introduce algorithmic concepts through various sorting algorithms.

But learning sorting algorithms is not merely a novice exercise in algorithm analysis; rather, an understanding of the function, applications, advantages, and disadvantages of various sorting algorithms finds many practical applications in computer science and software development.

Sorting is something that is universally done both in and outside of digital computing; it reduces problem complexity and reduces the search time for elements in a collection. Once a collection is sorted, it is often much more useful because of its increased access speed and organization.

One sorting algorithm is *Insertion Sort*. Because of its simple implementation and relative efficiency when compared to other sorting algorithms of quadratic time complexity, an analysis of Insertion Sort serves as an ample introduction to sorting and to algorithmic studies as a whole.

However, Insertion Sort is in fact one of the least time efficient sorting algorithms. Other algorithms like *Merge Sort, Heap Sort,* and *Quick Sort* all tout time complexities that are far superior to Insertion Sort’s.

This paper seeks to understand how each of the aforementioned sorting algorithms works by analyzing each algorithm’s pseudocode, correctly implementing and documenting the algorithm in the Java programming language, and analyzing an experiment using the Java implementation.

Graphing the data collected from the experimentation will provide an effective visual representation of the differences in each sorting algorithm’s time complexity and performance.

**BACKGROUND**

Previously mentioned was that Insertion Sort is simple and efficient. According to Cormen et al. in *Introduction to Algorithms,* the algorithm “works the way many people sort a hand of playing cards (17).” In this way, elements are sorted in ascending order from left to right by inserting them into their correct position after comparing the element to other elements, right to left. This simple procedure makes visualizing the algorithm easy for anyone who has sorted cards and implementing the algorithm quite straightforward in a programming language.

Insertion Sort is also an efficient algorithm for sorting a small number of elements. Additionally, when data is nearly sorted, the algorithm proves to be quite fast. Insertion Sort also sorts in place, meaning it carries no overhead like space consuming divide-and-conquer algorithms.

Despite carrying the above advantages, Insertion Sort is known to slow as the number of elements *n* in a collection grows large. This fact is observed by many authors in this area, notably Cormen et al. in *Introduction to Algorithms* or Jon Bently in *Programming Pearls*.

When the sorting of large datasets is required, it is important to examine how a certain sorting algorithm will perform when needing to sort the dataset. Despite being regarded as simple and efficient for small datasets, Insertion Sort’s performance on large datasets is superceded by Merge Sort, Heap Sort, and Quick Sort.

Pseudocode analysis and implementation experimentation allow for the precise calculation of exactly how long sorting a collection of *n* elements will take for each respective sorting algorithm. *Time complexity* expresses the quantifiable amount of time taken by an algorithm to run as a function of the number of elements in a collection. *Big-O notation* is used to describe the worst case running time of an algorithm. To do this, only the *n* term of the largest order is considered, because as *n* grows large, the largest order is all that matters. In *Big-O* notation, the time complexity for Insertion Sort is the quadratic O(n2); for *n* elements in a collection, the sorting time will be *n2.* Comparatively, the time complexities for Merge Sort and Heap Sort are both O(n log2n), whereas Quick Sort shares Insertion Sort’s O(n2).

However, Big-O describes an algorithm’s time complexity in the respective worst case scenario. For Insertion Sort, the worst case scenario is when a collection of elements is sorted in descending order. Insertion Sort’s sorting time drastically changes depending on the input, where an *average case* of random elements takes less time than O(n2), and the *best case* of presorted ascending elements makes comparisons in linear time O(n).

Sorting algorithms like Merge Sort and Heap Sort will always have a time complexity of O(n log2n), regardless of how their inputs are ordered. Therefore, neither of these two sorting algorithms have best, worst, and average cases, per se; rather, each case sorts with the same time complexity.

Quick Sort, like Insertion Sort, features a definite best, worst, and average case. However, Quick Sort performs sorting operations much faster for its best and average cases at a relatively quick O(n log2n).

This paper observes the various cases of Insertion Sort, Merge Sort, Heap Sort, and Quick Sort.

**PROCEDURE**

*Insertion Sort*

Insertion Sort follows the steps presented in the pseudocode below, adapted from Cormen et al.

**INSERTION-SORT(A)**

1 for j = 2 to A.length

2 key = A[j]

3 i = j - 1

4 while i > 0 and A[i] > key

5 A[i+1] = A[i]

6 i = i - 1

7 A[i+1] = key

The algorithm begins by receiving a passed array of comparable elements A. On the first line, the second element of the array is assigned to variable j, and a **for loop** increments j by one until it reaches the length of the array. Within this **for loop** follow lines two through seven, for each iteration of the loop.

The element at variable j is assigned to the variable key, and the variable i is assigned the index one less than j. After the assignments take place on lines two and three, the algorithm begins the inner loop.

While i is at least the first index in the array A and while its element is greater than the key, shift the elements to the right. Finally, the element at i+1 is set as the key.

This algorithm makes use of two loops and therefore two loop invariants. The outer loop’s (for loop’s) invariant reads as follows;

With each iteration of the **for loop** of lines one through eight, the subarray *A[1…j – 1]* consists of the elements originally in *A[1…j – 1],* but in sorted order.

**Initialization:** At the **for loop**’s first iteration, there is only one element in the array at *A[1].* Because this is not only the original element, but also the only element, so the array is trivially sorted. The invariant holds prior to the **for loop**’s first iteration.

**Maintenance:** The **for loop** places the element at *A[j]* into its correct position within *A[1…j – 1].* Thus, the subarray *A[1…j]* consists of the elements originally in *A[1…j],* but in sorted order. The invariant holds as *j* is incremented.

**Termination:** The **for loop** terminates when *j > A.length* = *n*. Because the invariant holds after the completion of each iteration, it holds after the final iteration and at termination—the subarray *A[1…n]* contains all of the elements of the original array, but in sorted order.

There is also an inner loop (while loop) invariant that finds itself within the *Maintenance* property of the outer **for loop**.

At the beginning of the while loop, elements of the subarray *A[i…j]* are greater than or equal to the key.

**Initialization:** At the initialization of the while loop, *i = j – 1, A[i] > key*, and *A[j] = key*.

**Maintenance:** The while loop’s invariant is maintained with the reassignment. *A[i+1] = A[i]* moves the element in A which is greater than the key into the element at index *i+1*.

**Termination:** The inner while loop’s invariant remains true even after termination. Before line seven is executed, where the key is assigned to *A[i+1], A[1…i]* is sorted and at most equal to key, and *A[i…j]* is sorted and at least equal to the key.

Knowing that the algorithm is correct, it is worthwhile to calculate Insertion Sort’s time complexity.

INSERTION-SORT(A) Cost Time

1 for j = 2 to A.length c1 n

2 key = A[j] c2  n-1

3 i = j – 1 c3 n-1

4 while i > 0 and A[i] > key c4

5 A[i+1] = A[i] c5

6 i = i – 1 c6

7 A[i+1] = key c7 n-1

= n2

The time complexity is indeed the same O(n2) that is presented in literature.

*Merge Sort*

Merge Sort follows the steps presented in the pseudocode below, adapted from Cormen et al.

**MERGE(A,p,q,r)**

1 n = q - p + 1

2 m = r - q

3 let L[1...n+1] and R[1...m+1] be new arrays

4 for i = 1 to n

5 L[i] = A[p+i-1]

6 for j = 1 to m

7 R[j] = A[q+j]

8 L[n+1] = INFINITY

9 R[m+1] = INFINITY

10 i = 1

11 j = 1

12 for k = p to r

13 if L[i] <= R[j]

14 A[k] = L[i]

15 i = i + 1

16 else A[k] = R[j]

17 j = j + 1

MERGE makes use of a loop and therefore a loop invariant. The loop invariant for MERGE reads as follows;

At the start of each iteration of the **for loop**of lines twelve through seventeen, the subarray *A[p…k-1]* contains the *k – p* smallest elements of *L[1…n+1]* and *R[1…m+1]*, in sorted order. Moreover, *L[i]* and *R[j]* are the smallest elements of their arrays that have not been copied back into *A*.

**Initialization:** Prior to the first iteration of the loop, *k = p*, so the subarray *A[p…k-1]* is empty. This empty subarray contains the *k – p = 0* smallest elements of *L* and *R*, and since *I = j = 1,* both *L[i]* and *R[j]* are the smallest elements of their arrays that have not been copied back into *A*.

**Maintenance:** When *L[i] <= R[j]*, *L[i]* is the smallest element not copied into *A*. *A[p…k-1]* contains the *k — p* smallest elements. Line fourteen copies *L[i]* into *A[k]*, so the subarray *A[p…k]* contains the *k – p + 1* smallest elements.

**Termination:** At the close of the loop, *k = r + 1*. By the loop invariant, the subarray *A[p…k-1]*, which is *A[p…r]*, contains the *k – p = r – p + 1* smallest elements of *L[1…n+1]* and *R[1…m+1]*, in sorted order. The arrays *L* and *R* together contain *n + m + 2 = r – p + 3* elements. Every element but the largest two have been copied into *A*.

**MERGE-SORT(A,p,r)**

1 if p < r

2 q = floor((p+r)/2)

3 MERGE-SORT(A,p,q)

4 MERGE-SORT(A,q+1,r)

5 MERGE-SORT(A,p,q,r)

*Heap Sort*

Heap Sort follows the steps among the three methods presented in the pseudocode below. This pseudocode is adapted from Cormen et al.

**MAX-HEAPIFY(A,i)**

1 l = left(i)

2 r = right(i)

3 if l <= A.heapsize and A[l] > A[i]

4 largest = l

5 else largest = i

6 if r <= A.heap-size and A[r] > A[largest]

7 largest = r

8 if largest != i

9 exchange A[i] with A[largest]

10 MAX-HEAPIFY(A,largest)

The max-heap property states that for every node *i,* the key in the parent of *i* is greater than or equal to the key in *i*, except for when *i* is the root of the binary tree. In order to maintain the max-heap property, the method MAX-HEAPIFY is called. At each step, the largest child of a node *i* is determined, and the index of that node is stored as *largest*. When the parent is largest, then the max-heap property already exists and the procedure terminates. If one of the children is largest, then the parent and that child are exchanged, and MAX-HEAPIFY is recursively called.

**BUILD-MAX-HEAP(A)**

1 A.heapsize = A.length

2 for i = ⌊A.length/2⌋ downto 1

3 MAX-HEAPIFY(A,i)

BUILD-MAX-HEAP makes use of a loop and therefore a loop invariant. The loop invariant for BUILD-MAX-HEAP reads as follows;

At the start of each iteration of the **for loop**of lines two and three, each node *i+1, i+2,…,n* is the root of a max-heap.

**Initialization:** Prior to the first iteration of the loop, *i =* ⌊n/2⌋. Each node ⌊n/2⌋+1, ⌊n/2⌋+2,…,n is a leaf and is thus the root of a trivial max-heap.

**Maintenance:** Up to this point, there is max-heap property is maintained. The children of node *i* are numbered higher than *i*. MAX-HEAPIFY preserves the loop invariant.

**Termination:** At termination, *i = 0.* By the loop invariant, that each node is a leaf, each node *1,2,…,n* is the root of a max-heap.

**HEAPSORT(A)**

1 BUILD-MAX-HEAP(A)

2 for i = A.length downto 2

3 exchange A[1] with A[i]

4 A.heapsize = A.heapsize - 1

5 MAX-HEAPIFY(A,1)

HEAPSORT makes use of a loop and therefore a loop invariant. The loop invariant for HEAPSORT reads as follows;

At the start of each iteration of the **for loop** on line two, the subarray *A[1…i]* is a max-heap containing the *i* smallest elements of *A*, and the subarray *A[i+1…n]* containing the *n-i­* largest elements of *A* sorted.

**Initialization:** Initially, *i = n.* By line 1, *A[1…n]* is a max-heap containing the *n* smallest elements of A. At the start, the subarray *A[i+1…n]* is empty, and the loop invariant holds trivially.

**Maintenance:** Before an iteration, *A[1*…*i]* is a max-heap containing the *i* smallest elements of *A,* and the subarray *A[i+1…n]* contains the *n-i* largest elements of *A* sorted. This holds by the loop invariant.

By exchanging *A[1]* with *A[i]* preserves the order of the elements in *A[i…n]*. The elements of *A[1…i-1]* are all smaller than *A[i…n]*.

Once *A.heapsize* is decremented, the method call to MAX-HEAPIFY(A,1) makes *A[1…i-1]* a max-heap. This restores the loop invariant.

**Termination:** When *i = 1*, *A[1…1]* is a max-heap containing the smallest element of *A*, and the subarray *A[2…n]* contains the *n-1* largest elements of *A* sorted.

*Quick Sort*

Quick Sort follows the steps presented in the pseudocode below.

**PARTITION(A,p,r)**

1 x = A[r]

2 i = p - 1

3 for j = p to r - 1

4 if A[j] <= x

5 i = i+1

6 exchange A[i] and A[j]

7 exchange A[i+1] and A[r]

8 return i + 1

PARTITION makes use of a loop and therefore a loop invariant. The loop invariant for PARTITION reads as follows;

At the beginning of each iteration of the **for loop** in PARTITION, *A[p…i]* is less than or equivalent to the pivot *x, A[i+1…j-1]* is greater than the pivot *x*, *A[j…r-1] remains unexamined, and A[r]* is equivalent to the pivot *x.*

**Initialization:** Prior to the first iteration, *x* is pivot, and *A[p...1]* and *A[i+1...j-1]* are both empty arrays.

**Maintenance:** At each iteration, if *A[j] ≤ x*, then *A[j]* and *A[i+1]* are exchanged. *i* and *j* are incremented. If *A[j] > x*, then only *j* is incremented.

**Termination:** At the completion of the loop, *j = r*, so all elements in the array *A* are partitioned into one of three cases;

*A[p...i] ≤ x A[i+1…x-1] > x A[x] = x*

**QUICKSORT(A,p,r)**

1 if p < r

2 q = PARTITION(A,p,r)

3 QUICKSORT(A,p,q-1)

4 QUICKSORT(A,q+1,r)

**SIMULATION TESTING**

To ensure that experimental results can be trusted, it is necessary to test the mechanisms that will complete the full testing.

An empty array and two arrays of twenty-five elements—unsorted and sorted—were used as test input for each algorithm. The actual results matched the expected results, as seen in **Table 1** below.

|  |  |  |
| --- | --- | --- |
| Input | Expected Results | Actual Results |
| Empty Array | Nothing returned | Nothing returned |
| Random array of 10 | Sorted array | Sorted array |
| Sorted array of 10 | Sorted array (same) | Sorted array (same) |

*Table 1: Test Results*

Additionally, assert statements are included in the Java implementation to prove that each loop invariant holds true. Dr. Howser, do you actually read these reports? Circle this sentence…

***Problems Encountered***

The program design from a previous, successful experiment performed only on Insertion Sort was copied and expanded upon for the first implementation of the program.

The original program design called for Boolean flags to be set within the main Java class. These flags were to act as switches, where the specific sorting algorithms and cases to be tested could be toggled on and off. However, by adding in an additional three sorting algorithms, the previously simple design that allowed for experimental flexibility became much more complicated.

Accounting for some 1,300 different runtime permutations—for example, specifying what data should be written to which file and when—proved to be extremely difficult. Eventually, this approach was scrapped in favor of a less flexible but much simpler hard-coded approach. Because the program was entirely rewritten, the programmer required additional time to code and test the program before actual experimental data could be collected.

Other problems included;

1. challenges implementing the Merge Sort and Heap Sort pseudocode in Java
2. accidentally passing sorted arrays
3. calculating algorithm times using milliseconds, rather than nanoseconds.

Each problem was resolved after thorough testing revealed the program’s faults.

**EXPERIMENTAL ANALYSIS: METHODOLOGY AND RESULTS**

Each sorting algorithm was implemented in the object oriented Java programming language. Using a series of Java classes to facilitate input generation, testing, and data collection, an experiment was conducted to confirm if the implementation in Java would yield the same results as the theoretical models. The complete Java classes can be found in the Appendix. The table below provides brief summaries of each classes’ function.

|  |  |
| --- | --- |
| Class | Purpose |
| MainApp | Holds the main() method, which runs algorithm analyses, as well as the testing conditions. The main() method increments a dataset size and runs sorts on each dataset. |
| DataGenerator | Creates a new array and fills it with random data. |
| AlgorithmTester | Holds the main testing method, as well as some utility methods. |
| TestResult | Encapsulates data from testing as a single (x,y) point. |
| Sorter | An interface for the sorter classes. |
| InsertionSort | The Java implementation of the Insertion Sort algorithm. |
| MergeSort | The Java implementation of the Merge Sort algorithm. |
| HeapSort | The Java implementation of the Heap Sort algorithm. |
| QuickSort | The Java implementation of the Quick Sort algorithm. |
| Stopwatch | Serves as a timer to calculate the millisecond duration of performing insertion sort on an array. |
| MultiFileWriter | Serves as wrapper for multiple FileWriters, directing input to the correct FileWriter, which in turn, writes experiment data to a comma separated value (CSV) file. |

*Table 3: Classes developed for this experiment*

***Methodology***

The experimentation was conducted under the following conditions;

*initial n = 0 maximum n = 100,000 max. value = 1000 n increment = 1000 trials = 5*

Under the above conditions, each experiment was conducted on arrays of size n = 0 to n = 100,000. For each experiment, *n* was incremented by one thousand, underwent five trials, and the mean sort time was calculated from the ten trials and printed to a .csv file.

For the average case, the input consisted of arrays with randomly generated unsorted integers ranging in value from 0 to 999. For the best and worst cases, the arrays consisted of randomly generated integers ranging in value from 0 to 999, sorted in ascending and descending order, respectively.

Specifically, Project1 creates a DataGenerator object. Within a **for loop**, the DataGenerator object repeatedly creates random arrays that are fed into an AlgorithmTester object. The AlgorithmTester object takes a sorter as a parameter—the passed sorter is the sorter used for a particular trial. Trial conditions are passed as parameters to a testAlgorithm() function, which returns a mean sort time to a TestResult object, eventually printed by the MultiFileWriter to a .csv file.

Within the AlgorithmTester object, the Sorter object completes sorts *t* times, where *t* is the number of trials—in this experiment, five. Each sort writes a time, calculated via the Stopwatch object instantiated in each Sorter object, to an ArrayList of times. The mean of the times is returned as the TestResult object.

***Results***

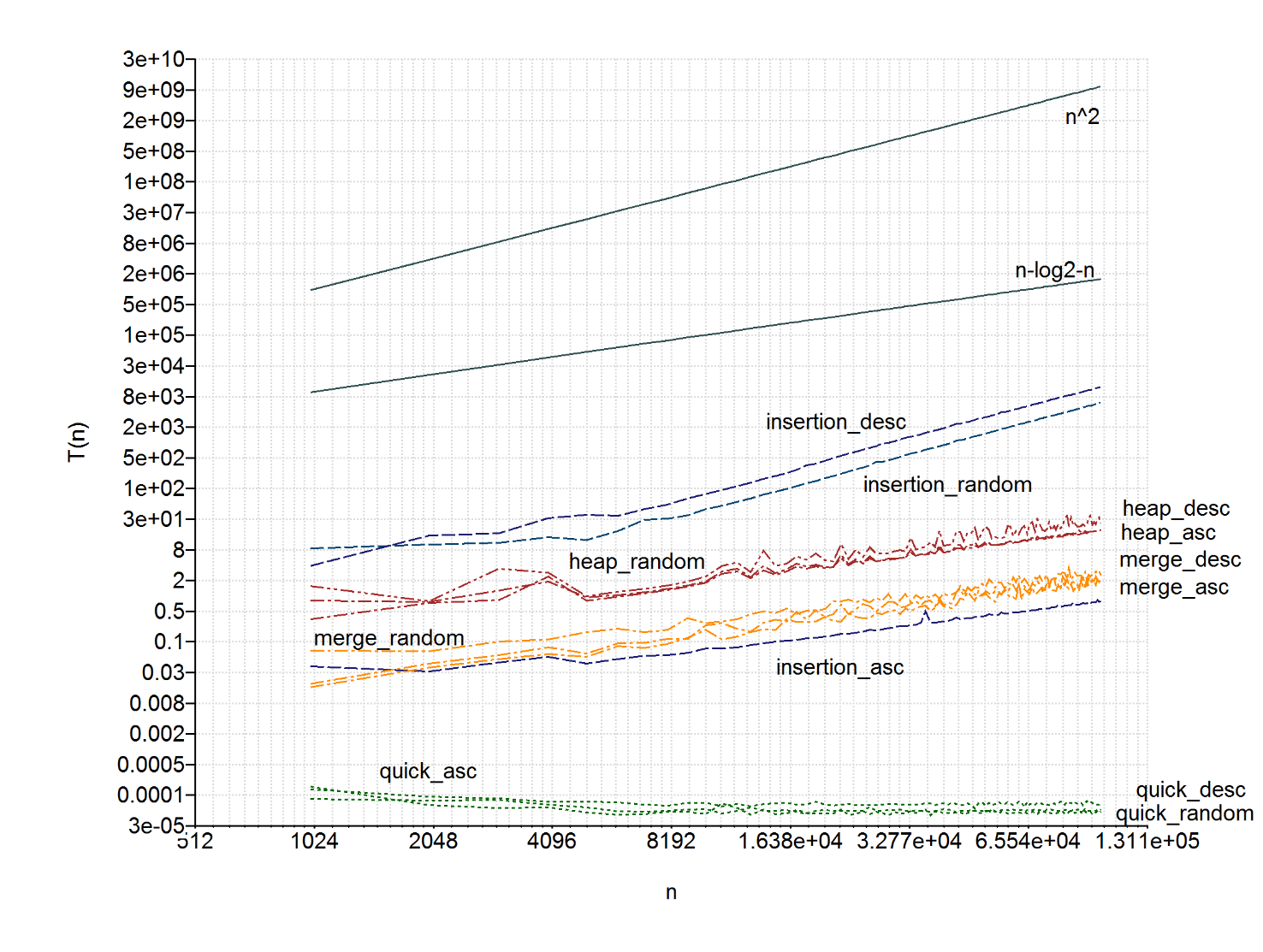
The hypotheses and actual results can be found in the table below;

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Hypothesis | | Actual Result |
| Insertion Sort: Average case (random data) | As *n* grows large, so does the mean time to sort an array of *n* random, unsorted integers. | As *n* grew large, so did the mean time to sort an array of *n* random, unsorted integers. | |
| Insertion Sort: Best case (ascendingly sorted) | As *n* grows large, the mean time to sort an array of *n* random, ascendingly sorted integers remains the same. | As *n* grew large, the mean time to sort an array of *n* random, ascendingly sorted integers remained the same. | |
| Insertion Sort: Worst case (descendingly sorted) | As *n* grows large, the mean time to sort an array of *n* random, descendingly sorted integers remains the same. | As *n* grew large, the mean time to sort an array of *n* random, descendingly sorted integers grew. | |
| Merge Sort: Normal case (random data) | As *n* grows large, so does the mean time to sort an array of *n* random, unsorted integers. The mean time will not grow quickly. | As *n* grew large, so did the mean time to sort an array of *n* random, unsorted integers, however slowly. | |
| Merge Sort: Normal case (ascendingly sorted) | As *n* grows large, so does the mean time to sort an array of *n* random, ascendingly integers. The mean time will not grow quickly. | As *n* grew large, so did the mean time to sort an array of *n* random, ascendingly sorted integers, however slowly. | |
| Merge Sort: Normal case (descendingly sorted) | As *n* grows large, so does the mean time to sort an array of *n* random, descendingly integers. The mean time will not grow quickly. | As *n* grew large, so did the mean time to sort an array of *n* random, descendingly sorted integers, however slowly. | |
| Heap Sort: Normal case (random data) | As *n* grows large, so does the mean time to sort an array of *n* random, unsorted integers. The mean time will not grow quickly. | As *n* grew large, so did the mean time to sort an array of *n* random, unsorted integers. | |
| Heap Sort: Normal case (ascendingly sorted) | As *n* grows large, so does the mean time to sort an array of *n* random, ascendingly sorted integers. The mean time will not grow quickly. | As *n* grew large, so did the mean time to sort an array of *n* random, ascendingly sorted integers, however slowly. | |
| Heap Sort: Normal case (descendingly sorted) | As *n* grows large, so does the mean time to sort an array of *n* random, descendingly sorted integers. The mean time will not grow quickly. | As *n* grew large, so did the mean time to sort an array of *n* random, decendingly sorted integers, however slowly. | |
| Quick Sort: Average/Best case  (random pivot) | As *n* grows large, so does the mean time to sort an array of *n* random, unsorted integers. The mean time will not grow quickly. The constant factor of Quick Sort will outperform the other sorting algorithms. | As *n* grew large, so did the mean time to sort an array of *n* random, unsorted integers. | |
| Quick Sort: Worst case  (random pivot) | As *n* grows large, so does the mean time to sort an array of *n* random, unsorted integers, at the same rate as Insertion Sort. | Untested. | |

*Table 2: Table of expected and actual results*

As observed in **Table 2**, the experimental results confirm the hypotheses.

**Figure 1** displays the results of the twelve experiments, with and graphed for comparison.



*Figure 1: Experiment times for Insertion Sort, Merge Sort, Heap Sort, and Quick Sort*

***Analysis***

The hypothesized results are reflected in the actual results, as graphed in Figure 1.

Figure 1 shows that the best case for Insertion Sort, an ascendingly sorted array of integers, is scanned in linear time, that the worst case, a descendingly sorted array of integers, is sorted as O(n2), and that the average case is similar to the worst case.

For the two sorting algorithms that theoretically lack best, worst, and average cases, Figure 1 shows that Merge Sort and Heap Sort perform similarly regardless of the order of the input elements. In both Merge Sort and Heap Sort, an ascendingly sorted dataset marginally outperforms a descendingly sorted dataset, which in turn marginally outperforms a randomly sorted dataset. Figure 1 also demonstrates that although Merge Sort and Heap Sort share time complexities of O(n log2n), Merge Sort outperforms Heap Sort in the actual, quantifiable time taken to sort elements.

Figure 1 shows that Quick Sort, does not follow its theoretical time complexity of O(n log2n). Rather, its time complexity appears to closer resemble O(n). Quick Sort greatly outperforms the other three sorting algorithms that were tested, as its constant factor is much smaller.

**CONCLUSION**

Through the testing procedure, the hypotheses—that the Java implementation of each sorting algorithm would fit the theoretical time complexities—were mostly confirmed. Although the results from the Quick Sort experimentation exhibited an apparent time complexity of O(n), the overall n2 or n log2n trends are clearly visible. Should this experiment be repeated, it would be best to increase the maximum size of n and to limit the number of programs that have access to the test environment’s CPU in order to limit erratic cycles and mitigate the occurrence of outliers. Further experimentation should also reduce the dataset increment and test for dataset sizes far greater than 100,000.

***References***

Cormen et al.:

T.H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein., *Introduction to Algorithms, 3rd Edition.* MIT press Cambridge, 2001.