**Wafa Anam**

**CPSC 319**

**Login: wafa.anam**

**T03**

**TA: MD Mamunur Rashid**

**Assignment 1**

**Introduction**

In order to observe the performance of different sorting algorithms with various input types, selection, insertion, merge, and quick sorting algorithms were implemented in Java with the purpose of sorting in ascending order. The code was then run with an integer array input in ascending, descending, and random orders. This was done in the Java Eclipse IDE. The number of inputs was increased by factors of 10, ranging from 10 to 1000000. The difference in the system’s nanotime was then outputted to the console to measure how long each sorting algorithm took to complete its performance. The source code is presented in the appendix.

**Observations**

Table 1. The time in seconds needed for selection sort to complete its function for integer arrays of n inputs, organized in different orders.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Order of array | n | | | | | |
| 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
| Ascending | 4.2286E-5 | 1.24806E-4 | 0.007197287 | 0.683484832 | 68.72458503 | 7502.57977181 |
| Descending | 4.4749E-5 | 1.69556E-4 | 0.007726071 | 0.744242807 | 73.151534273 | 7277.67010267 |
| Random | 6.2813E-5 | 1.51081E-4 | 0.007668593 | 0.686997876 | 69.105856073 | 7051.20446395 |

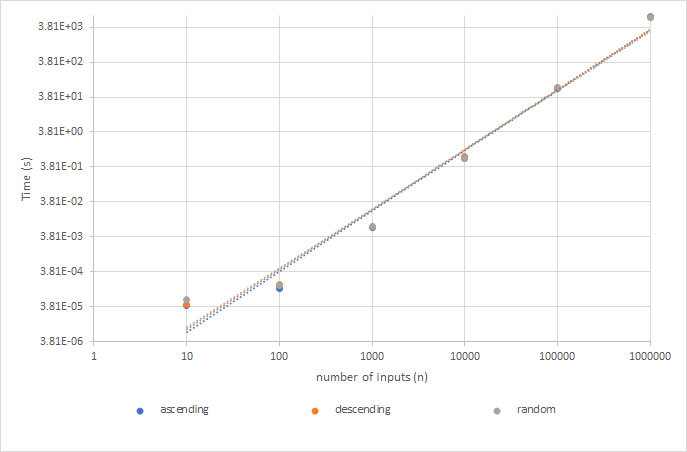


Figure 1. The time taken for each order of array input of varying sizes to be sorted to ascending order by selection sort. Horizontal axis is represented by a logarithmic scale.

Table 2. The time in seconds needed for insertion sort to complete its function for integer arrays of n inputs, organized in different orders.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Order of array | n | | | | | |
| 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
| Ascending | 4.2697E-5 | 6.6919E-5 | 6.8561E-5 | 3.65797E-4 | 0.00309634 | 0.028859351 |
| Descending | 7.3898E-5 | 1.62166E-4 | 0.010240256 | 0.967138645 | 95.380722142 | 11261.742844076 |
| Random | 4.8034E-5 | 1.04279E-4 | 0.00517001 | 0.481986255 | 48.11478956 | 6558.020128674 |

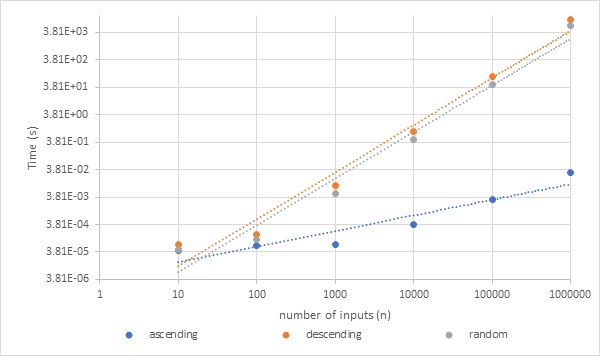


Figure 2. The time taken for each order of array input of varying sizes to be sorted to ascending by insertion sort. Horizontal axis is represented by a logarithmic scale.

Table 3. The time in seconds needed for merge sort to complete its function for integer arrays of n inputs, organized in different orders.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Order of array | n | | | | | |
| 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
| Ascending | 5.9119E-5 | 1.29733E-4 | 6.60569E-4 | 0.007166907 | 0.081020486 | 0.813689583 |
| Descending | 5.6245E-5 | 1.23985E-4 | 7.18046E-4 | 0.007368895 | 0.077661397 | 0.788330139 |
| Random | 5.9529E-5 | 1.41638E-4 | 8.17808E-4 | 0.008095562 | 0.090227808 | 0.962147222 |

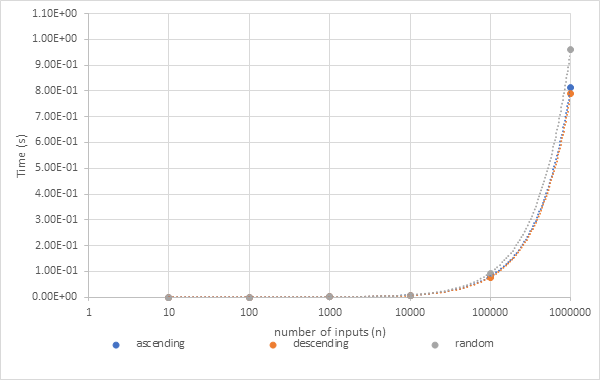


Figure 3. The time taken for each order of array input of varying sizes to be sorted to ascending by merge sort. Horizontal axis is represented by a logarithmic scale. Line of best fit is linear.

Table 4. The time in seconds needed for quick sort to complete its function for integer arrays of n inputs, organized in different orders.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Order of array | n | | | | | |
| 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
| Ascending | 4.6802E-5 | 7.4308E-5 | 2.31959E-4 | 0.00199895 | 0.023720543 | 0.224095864 |
| Descending | 4.5571E-5 | 9.2373E-5 | 2.36885E-4 | 0.002214897 | 0.025153760 | 0.243259754 |
| Random | 5.0498E-5 | 9.1141E-5 | 3.61691E-4 | 0.003911274 | 0.045402723 | 0.522698899 |

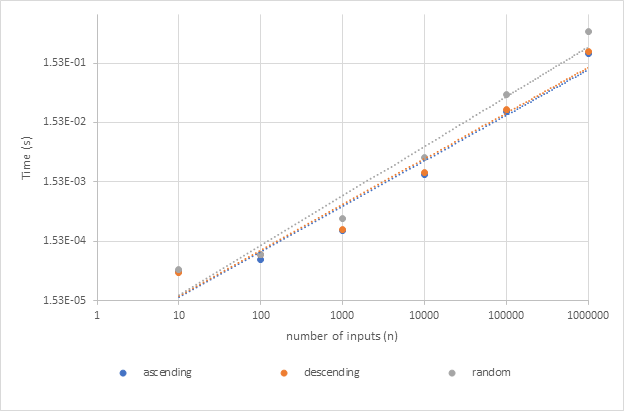
****

Figure 4. The time taken for each order of array input of varying sizes to be sorted to ascending by quick sort. Horizontal axis is represented by a logarithmic scale.

**Analysis**

Data Analysis

From the timing results above, it is seen that the shortest time for sorting the largest arrays comes from quicksort, with all sorting occurring in less than a second. The longest timing comes from selection sort, at over 7500 seconds. For arrays up to a size of 100 elements, selection sort and insertion sort perform very similarly to the other algorithms. Past this however, their timings begin to increase significantly above those of merge and quick sorts’. The exception is with an ascending array for insertion sort however, which beats even quick sort with a large input.

Based on the trendlines that best fit figures 1-4, merge and quick sorts were much closer to linear growth than selection and insertion sorts, which fit better with a power trendline. Furthermore, selection, merge and quick sort were not very apparently affected by input array order, whereas in insertion sort, it made a significant difference.

Complexity Analyses:

1. Selection Sort

**public** **static** **void** selectionSort(**int**[] arr) //operations:

{

**for**(**int** i = 0; i < arr.length-1; i++) //2n+1

{

**int** min = i; //1(n-1)

**for**(**int** j = i+1; j < arr.length; j++) //2n+2

**if**(arr[j] < arr[min]) //3n

min = j; //n

**int** temp = arr[min]; //2(n-1)

arr[min] = arr[i]; //3(n-1)

arr[i] = temp; //2(n-1)

}

}

Outer loop: operations = 2n+1 🡪 O(n)

Inside loop: operations = 8n-8 🡪 O(n)

Inner loop: operations = 5n+2 🡪 O(n)

Overall: O(n) + O(n) O(n) 🡪 O(n2)

Since the algorithm works by comparing each element of the array to the rest of the items, until it reaches the end, the number of comparisons is the same for arrays of the same length, regardless of their starting order. This results in the O(n2) characterization seen by most of the red and green instructions. The O(n) portion for the swaps is dropped as a lesser term. This algorithm therefore should perform independent of input order.

1. Insertion Sort

**public** **static** **void** insertionSort(**int**[] arr) //operations:

{

**for**(**int** i = 1, j; i < arr.length; i++) //2n+1

{

**int** temp = arr[i]; //2(n-1)

**for**(j = i; j > 0 && temp < arr[j-1]; j--) //n(n+1)

arr[j] = arr[j-1]; //-n+n(n+1)/2

arr[j] = temp; //2(n-1)

}

}

Outer loop: operations = 2n+1 🡪 O(n)

Inside loop: operations = 4n-4 🡪 O(n)

Inner loop: operations = n2+(n2/2)+(n/2) 🡪 O(n2)

Overall: O(n) + O(n) +O(n2) 🡪 O(n2)

This calculation was done for the worst-case scenario which for this algorithm happens to occur when the initial data is in reverse order. This is because the algorithm works by comparing items at the beginning of the array with items to their left. Once a smaller item is found, the item of interest is inserted, and the remaining items to the right are shifted. If the data is in reverse order then, each value would have to be compared to every value to their left, but starting at the left, so the maximum number of compares and shifts would occur for each value. This worst case is represented by the O(n2) characterization found above. If the array is in ascending order however, no shifts, and only n comparisons would occur since each item would only need to be compared immediately with their left. In this best case then, the algorithm can be characterized as O(n).

1. Merge Sort

**public** **static** **void** mergeSort(**int**[] arr, **int** first, **int** last)

{ //operations:

**if**(first < last) //n

{

**int** mid = (first + last)/2; //n-1

*mergeSort*(arr, first, mid);

*mergeSort*(arr, mid+1, last);

*merge*(arr, first, mid, mid+1, last);

}

}

operations = 2n-1 🡪 O(n)

**public** **static** **void** merge(**int**[] arr, **int** first\_1, **int** last\_1, **int** first\_2, **int** last\_2)

{

**int** size = last\_2 - first\_1 + 1; //operations:

**int**[] temp = **new** **int**[size]; //1

**int** i = 0, j = 0, k = 0; //3

**for**(i = 0, j = first\_1, k = first\_2; i < size && j <=last\_1 && k <= last\_2; i++) //4+2n

{

**if**(arr[j] < arr[k]) //3n

temp[i] = arr[j++];

**else**

temp[i] = arr[k++]; //3n

}

**while**(i < size)

{

**if**(j > last\_1)

temp[i] = arr[k++];

**else**

temp[i] = arr[j++];

i++;

}

**for**(i = first\_1, j = 0; i <= last\_2; i++, j++)

arr[i] = temp[j];

}

Overall: O(n)

The merge function here is O(n). Since the size of the array merge receives is halved on each successive call to it from mergeSort, the steps are halved too making it a O() function. Using the multiplication rule, the overall characterization is O(n). This holds for any order of input since no matter what, the array is continuously halved until only one element remains, then remerged. This concept is illustrated in figure 4.

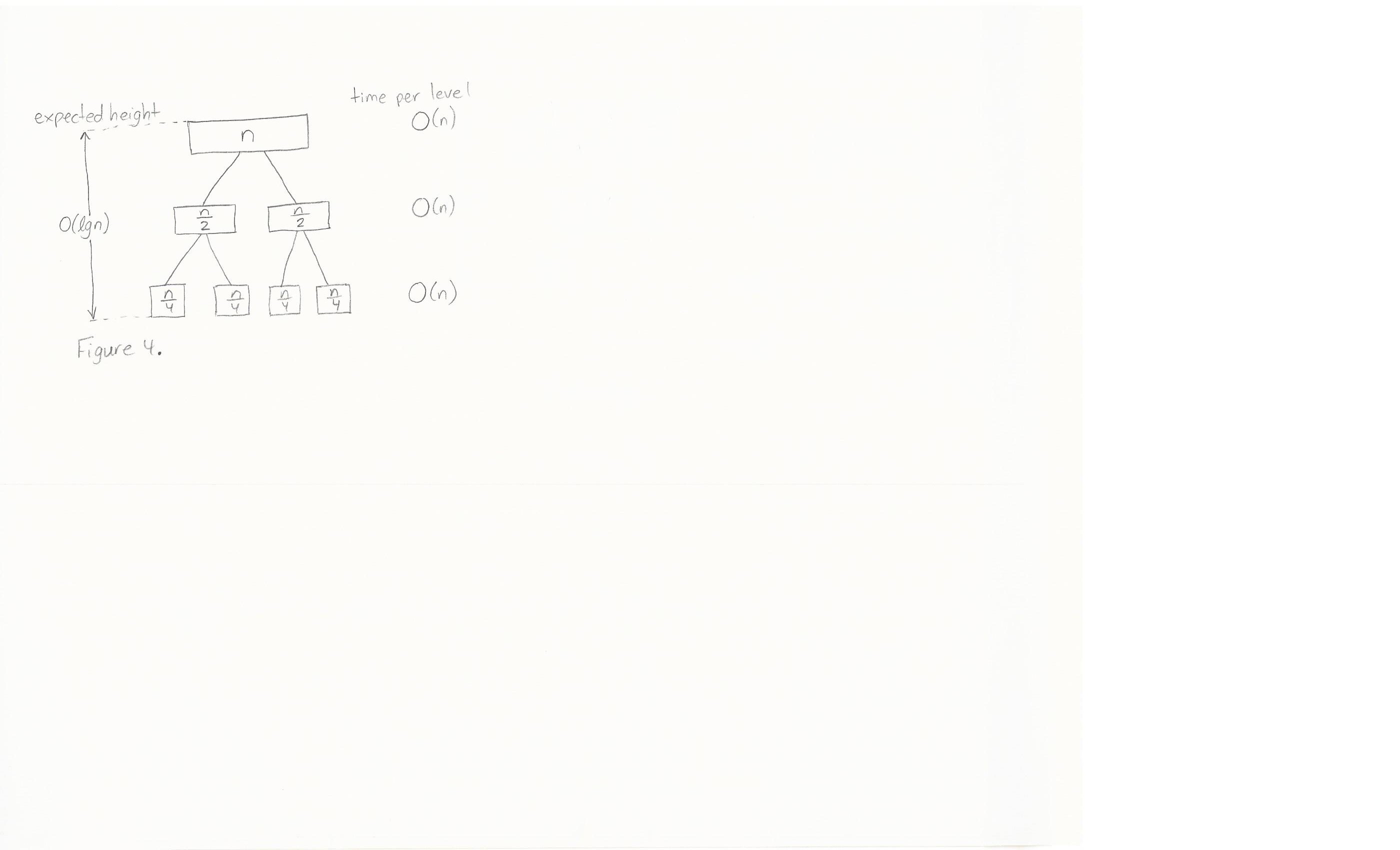


Figure 4. representation of best, worst, and average input cases for implemented merge sort.

1. Quick Sort

A complexity of quick sort is difficult to do with counting. Three functions were used for the implementation of this sort (2 quicksort functions and a swap function) and they can be found in the appendix. The analysis is instead represented by the tree diagrams of figure 5. This algorithm Follows a O(n) characterization at its average and best cases, when the subarrays created are equal or relatively similar in size. The worst case however, occurs when each pivot happens to be the smallest or largest item in the subarray. This results in a O(n2) characterization. With the implementation of the sort in the appendix, this worst case is unlikely to occur since the pivot is selected from the centre each time.

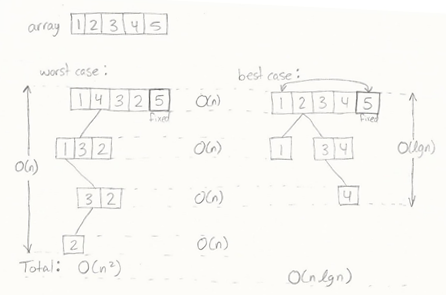


Figure 5. Best and worst-case sketch for implemented quick sort using a sample 5 element array

**Discussion**

The collected data matches the complexity analyses in most cases. The only case that is not represented by the empirical values is for the worst-case scenario of quick sort. Since the implemented code selected the pivot point at the centre, only a very specific random array (such as in figure 5) would lead to subarrays of the largest possible size. However, since the data inputted for the tests were ascending, descending, and random order, it was very unlikely that the specific random case would occur. Thus, the empirical results do not address quick sort’s weakness which can cause it to behave with O(n2) characteristics, much like insertion and selection sorts.

For insertion sort, the behaviour of the algorithm for ordered inputs is seen in both the empirical and analytical results. In figure 2, the trendline for the ascending data falls significantly below both other trends. The shape also differs. The fastest sorts also occur for arrays of all sizes with insertion sort and input arrays in ascending order. This is supported by the fact that at its best, insertion sort follows a O(n) characterization, which beats even quick sort’s best-case O(n).

The two algorithms characterized as O(n2) for their worst case performed similarly with respect to trends. However, insertion sort’s performance relied heavily on input order whereas the empirical data for selection sort is similar for all input orders. This makes sense with the complexity analysis result where it was determined that selection sort is dependent on the number of comparisons it performs, which is the same regardless of input order.

For their average and best cases, merge and quick sort fell under the same big-O classification. Quick sort did better for all tested input sizes, but by an amount of time that is probably not observable by humans. Furthermore, their growth followed a similar pattern for increasing inputs as expected. Again, the main expected difference between the two the dependency on order of quick sort versus the independency of merge sort but it was not detected by the tests run.

**Conclusion**

The timing analyses of the algorithms reveals that there is no ultimate best algorithm but rather that each algorithm may apply to a different situation. When sorting small amounts of data, it would most likely be appropriate to implement insertion or selection sort. This is because their implementations are quite straightforward and easy to understand. Furthermore, the timing is short enough for up to 1000 elements, as seen in tables 1 and 2. For larger amounts of data however, quick or merge sort become more appropriate since they performed the best with large arrays, on average. If there is concern that the array may be in ascending or descending order however, quick sort should be avoided for implementations that select their pivot at either end. If the input is known to be sorted or nearly sorted, insertion sort is the best option as it begins to depend only on the number of inputs. Finally, merge sort would be a poor choice if memory/space is a concern since the merge function requires for a new array to be created rather than an in-place sort.

**References**

Drozdek, A. (2009). Data structures and algorithms in Java. Singapore: Cengage Learning.

Rashid, MD. (2018). CPSC 319: Tutorial (Winter 2018). Available: <https://pages.cpsc.ucalgary.ca/~mdmamunur.rashid1/>

**Appendix**

Source Code:

**import** java.io.PrintWriter;

/\*\*

\* Provides methods to create an array of integers of specified size and order and sorts array

\* with specified sorting algorithm.

\* Prints sorted array to text file and prints time required to execute algorithm

\* as standard output.

\*

\* **@author** Wafa Anam

\* **@version** 1.0

\* **@since** January 18, 2018

\*/

**public** **class** Assign1 {

/\*\*

\* Prints an array of integers to a text file, one element per line.

\* **@param** array is the array of integers to be printed

\* **@param** file is the name of the desired text file

\* implementation borrowed from Rashid, 2018

\*/

**public** **static** **void** printIntArrayToText(**int**[] array, String file)

{

PrintWriter pr = **null**;

**try** {

pr = **new** PrintWriter(file);

**for** (**int** i = 0; i < array.length; i++) {

pr.println(array[i]);

}

} **catch** (Exception e) {

e.printStackTrace();

System.***out***.println("No such file exists.");

} **finally** {

**if** (pr != **null**) {

pr.close();

} **else** {

System.***out***.println("PrintWriter not open");

}

}

}

/\*\*

\* Constructs an array of integers of the specified size in ascending order.

\* Values range from 1 to size.

\* **@param** size is the size of the array and must be greater than zero.

\* **@return** an array of integers arranged in ascending order

\*/

**public** **static** **int**[] ascendingArray(**int** size)

{

**int**[] arr = **new** **int**[size];

**for** (**int** i = 0; i < size; i++) {

arr[i] = i + 1;

}

**return** arr;

}

/\*\*

\* Constructs an array of integers of the specified size in descending order.

\* Values range from 1 to the size of the array.

\* **@param** size is the size of the array and must be greater than zero.

\* **@return** an array of integers arranged in descending order

\*/

**public** **static** **int**[] descendingArray(**int** size)

{

**int**[] arr = **new** **int**[size];

**for** (**int** i = 0; i < size; i++) {

arr[i] = size - i;

}

**return** arr;

}

/\*\*

\* Constructs an array of integers of the specified size in a random order.

\* Values range from 1 to the size of the array, and may repeat.

\* **@param** size is the size of the array and must be greater than zero.

\* **@return** an array of integers arranged in a random order

\* Code segments adapted from Rashid, 2018

\*/

**public** **static** **int**[] randomArray(**int** size)

{

**int**[] arr = **new** **int**[size];

**for** (**int** i = 0; i < size; i++) {

arr[i] = (**int**) (Math.*random*() \* size);

}

**return** arr;

}

/\*\*

\* Sorts an array of integers in ascending order.

\* **@param** arr the array of integers to be sorted

\* Implementation from Drozdek, 2009

\*/

**public** **static** **void** selectionSort(**int**[] arr)

{

**for**(**int** i = 0; i < arr.length-1; i++)

{

**int** min = i;

**for**(**int** j = i+1; j < arr.length; j++)

**if**(arr[j] < arr[min])

min = j;

**int** temp = arr[min];

arr[min] = arr[i];

arr[i] = temp;

}

}

/\*\*

\* Sorts an array of integers in ascending order.

\* **@param** arr the array of integers to be sorted

\* Implementation from Drozdek, 2009

\*/

**public** **static** **void** insertionSort(**int**[] arr)

{

**for**(**int** i = 1, j; i < arr.length; i++)

{

**int** temp = arr[i];

**for**(j = i; j > 0 && temp < arr[j-1]; j--)

arr[j] = arr[j-1];

arr[j] = temp;

}

}

/\*\*

\* Sorts an array of integers in ascending order.

\* Recursively breaks array into single elements and calls

\* upon merge function to order.

\* **@param** arr the array of integers to be sorted

\* **@param** first the index of the first element in the array

\* **@param** last the index of the last element in the array

\* Implementation from Drozdek, 2009

\*/

**public** **static** **void** mergeSort(**int**[] arr, **int** first, **int** last)

{

**if**(first < last)

{

**int** mid = (first + last)/2;

*mergeSort*(arr, first, mid);

*mergeSort*(arr, mid+1, last);

*merge*(arr, first, mid, mid+1, last);

}

}

/\*\*

\* Merges two array segments into one array in ascending order

\* **@param** arr the array to be rearranged

\* **@param** first\_1 the index of the first element of the first array segment

\* **@param** last\_1 the index of the last element of the first array segment

\* **@param** first\_2 the index of the first element of the second array segment

\* **@param** last\_2 the index of the last element of the second array segment

\*/

**public** **static** **void** merge(**int**[] arr, **int** first\_1, **int** last\_1, **int** first\_2, **int** last\_2)

{

**int** size = last\_2 - first\_1 + 1;

**int**[] temp = **new** **int**[size];

**int** i = 0, j = 0, k = 0;

**for**(i = 0, j = first\_1, k = first\_2; i < size && j <= last\_1 && k <= last\_2; i++)

{

**if**(arr[j] < arr[k])

temp[i] = arr[j++];

**else**

temp[i] = arr[k++];

}

**while**(i < size)

{

**if**(j > last\_1)

temp[i] = arr[k++];

**else**

temp[i] = arr[j++];

i++;

}

**for**(i = first\_1, j = 0; i <= last\_2; i++, j++)

arr[i] = temp[j];

}

/\*\*

\* Moves largest array element to the end of an array of integers

\* and calls upon quickSort to sort it in ascending order

\* **@param** arr the array of integers to be sorted

\* Implementation from Drozdek, 2009

\*/

**public** **static** **void** quickSort(**int**[] arr)

{

**if**(arr.length < 2)

**return**;

**int** max = 0;

**for**(**int** i = 0; i < arr.length; i++)

**if**(arr[max] < arr[i])

max = i;

*swap*(arr, arr.length - 1, max);

*quicksort*(arr, 0, arr.length -2);

}

/\*\*

\* Swaps elements of an integer array in place recursively

\* for a final ascending order result.

\* **@param** arr the integer array to be sorted

\* **@param** first the index of the first element of the array

\* **@param** last the index of the last element of the array

\* Implementation from Drozdek, 2009

\*/

**public** **static** **void** quicksort(**int**[] arr, **int** first, **int** last)

{

**int** lower = first + 1, upper = last;

*swap*(arr, first, (first+last)/2);

**int** bound = arr[first];

**while**(lower <= upper)

{

**while**(arr[lower] < bound)

lower++;

**while**(arr[upper] > bound)

upper--;

**if**(lower < upper)

*swap*(arr, lower++, upper--);

**else**

lower++;

}

*swap*(arr, upper, first);

**if**(first < upper-1)

*quicksort*(arr, first, upper - 1);

**if**(upper+1 < last)

*quicksort*(arr, upper + 1, last);

}

/\*\*

\* Switches the values at one index of an integer array with the value at another index.

\* **@param** arr is the integer array and must not be empty

\* **@param** first is the index of one of the elements of the desired swap

\* **@param** second the index of the second element to be swapped

\* Both parameters first and second must be less than the size of the array

\* and greater than zero.

\*/

**public** **static** **void** swap(**int**[] arr, **int** first, **int** second)

{

**int** temp = arr[first];

arr[first] = arr[second];

arr[second] = temp;

}

**public** **static** **void** main(String[] args)

{

**int** valid = 1, size = 0;

String order = **null**, algorithm = **null**, file = **null**;

**int**[] array = **null**;

**if**(args.length == 4)

{

order = args[0];

size = Integer.*parseInt*(args[1]);

algorithm = args[2];

file = args[3];

**if**(!order.equalsIgnoreCase("ascending")&&!order.equalsIgnoreCase("descending")&&

!order.equalsIgnoreCase("random"))

{

System.***out***.println("Invalid input for order");

valid = 0;

}

**if**(size < 0)

{

System.***out***.println("Invalid input for size");

valid = 0;

}

**if**(!algorithm.equalsIgnoreCase("selection")&&!algorithm.equalsIgnoreCase("insertion")&&

!algorithm.equalsIgnoreCase("merge")&&!algorithm.equalsIgnoreCase("quick"))

{

System.***out***.println("Invalid input for algorithm");

valid = 0;

}

}

**else** {

System.***out***.println("Four arguments are required");

valid = 0;

}

**if**(valid == 0)

{

System.***out***.println("program terminating");

System.*exit*(0);

}

**if**(order.equalsIgnoreCase("ascending"))

array = *ascendingArray*(size);

**else** **if**(order.equalsIgnoreCase("descending"))

array = *descendingArray*(size);

**else**

array = *randomArray*(size);

*printIntArrayToText*(array, "before");

**long** startTime = 0;

**if**(algorithm.equalsIgnoreCase("selection"))

{ startTime = System.*nanoTime*();

*selectionSort*(array);

}

**else** **if**(algorithm.equalsIgnoreCase("insertion"))

{ startTime = System.*nanoTime*();

*insertionSort*(array);

}

**else** **if**(algorithm.equalsIgnoreCase("merge"))

{ startTime = System.*nanoTime*();

*mergeSort*(array, 0, size-1);

}

**else**

{ startTime = System.*nanoTime*();

*quickSort*(array);

}

System.***out***.println("The time elapsed to complete " + algorithm + "Sort for a(n) " + order +

" array of "+ size +" elements was " +

((**double**)(System.*nanoTime*()- startTime))/1000000000 + " seconds.");

*printIntArrayToText*(array, file);

}

}