Analysis and Comparison of Sorting Algorithms (Insertion, Merge, and Heap) Using Java

Khaznah Alhajri[1], Wala Alsinan[1], Sahar Almuhaishi[1], Fatimah Alhmood[1], Narjis AlJumaia[1], and Azza.A.A[1]*

2190004752@iau.edu.sa, 2190004429@iau.edu.sa, 2200005060@iau.edu.sa, 2200004124@iau.edu.sa, 2190000716@iau.edu.sa, aaaali@iau.edu.sa

[1]Computer Science Department, College of Science and Humanities Imam Abdulrahman Bin Faisal University, P.O.Box 31961, Jubail, Saudi Arabia

Abstract
Sorting is an important data structure in many applications in the real world. Several sorting algorithms are currently in use for searching and other operations. Sorting algorithms rearrange the elements of an array or list based on the elements' comparison operators. The comparison operator is used in the accurate data structure to establish the new order of elements. This report analyzes and compares the time complexity and running time theoretically and experimentally of insertion, merge, and heap sort algorithms. Java language is used by the NetBeans tool to implement the code of the algorithms. The results show that when dealing with sorted elements, insertion sort has a faster running time than merge and heap algorithms. When it comes to dealing with a large number of elements, it is better to use the merge sort. For the number of comparisons for each algorithm, the insertion sort has the highest number of comparisons.

Keywords
Insertion sort, Merge sort, Heap sort, and Sorting algorithms.

I. INTRODUCTION

In real-world applications, it is necessary to arrange the data in a sorted order to perform searching and other operation efficiently such as particular records in the database, roll numbers in the merit list, a particular page in a book, and others. All this would have been a mess if the data was kept unsorted [12]. Fortunately, there is an algorithm called a sorting algorithm, it takes a list of items as input data, performs specific operations on those lists, and delivers an ordered list as output. The use of algorithms did not begin with the introduction of computers, people use them while they are solving problems. We can describe algorithms as a finite sequence of rules which describes and analyze the algorithms [8]. In this report, three sorting algorithms are discussed to check the performance and comparison of all these algorithms based on time complexity and running time. Time complexity is based on the amount that the computer time takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform [7]. The number of primitive operations or "steps" executed by an algorithm on a specific input determines its running time. It is preferable to define the concept of step as machine-independently as possible [7]. An analysis is made for each algorithm by finding the best case, worst case, and average case. We check how much processing time is taken by all three sorting algorithms and compared them and finding which sorting algorithm takes less time to sort the elements from 1000 to 200,000. If any algorithm takes less processing time it means that it sorts the element faster than others [5]. The main role of the sort algorithm is to operate in the largest data set [11]. The main function of sorting is to organize and filter the largest amount of data. The performance of the database depends on the type of sort algorithm that is used [12]. The choice of algorithm accuracy depends on the most important factors: user's hardware, software available, and comfort of use of the database [12]. The sorting algorithms that will be included in this report are insertion sort, merge sort and heap sort.

This paper is structured as follows: Section II provides the literature review. Section III presents the background of the three algorithms. In Section IV the details about methodology and experimental setup have been provided; followed by results in Section V. Finally, the conclusion is in Section VI.

II. BACKGROUND

A. Insertion Sort Algorithm

Insertion sort is an incremental algorithm that inserts items into the proper place. The first element in the left

Manuscript received December 5, 2022
Manuscript revised December 20, 2022
https://doi.org/10.22937/IJCSNS.2022.22.12.25

hand will be considered as sorted. Then the second element will be compared to the first element. If the first element is greater than the second, the first element is placed on the right side; otherwise, nothing occurs. Similarly, all unsorted elements will be taken and placed in their proper place from the smallest element to the largest element [6]. The running time of INSERTION-SORT on an input of n values, we sum the products of the cost and times columns. The performance analysis of insertion sort in three cases, which are:

- **Running time in best case** happens when all elements are already ordered, the complexity time can be calculated as $T(n) = O(n)$.
- **Running time in worst case** when elements are arranged in reverse, and the complexity time can be calculated as $T(n) = O(n^2)$.
- **Running time in average case** is often roughly as bad as the worst case. Half the elements are sorted and the other half of the elements are unsorted. , The complexity time can be calculated as $T(n) = O(n^2)$.

The insertion sort is simple and has a good running time in the best case. However, insertion sort has a long running time in worst and average cases.

B. Merge Sort Algorithm

Merge sort is in place order and follows the divide and conquer approach. The Merge Sort Algorithm is an inplace order recursive algorithm. The array of size n is divided into the largest number of log n subarrays and merging them into a single array takes $O(n)$ time. The time complexity of the Merge sort is $O(nlogn)$ in all three cases. The relation of Merge sort time complexity: $T(n) = 2T(n/2)+O(n)$. Merge sort has three steps. First, dividing problems into sub-problems. Second, conquer the subproblems by solving them recursively. Third, combine the solution of these sub-problems [13]. The running time of each step can be expressed as:

• **Divide:** The division step. Computing the middle of the subarray, takes constant time. Thus, $D(n) = O(1)$.

• **Conquer:** Recursively solve two subproblems of size $n = 2$, which contributes $2T(n/2)$ to the running time. • **Combine:** The merge procedure on an n-element subarray takes time O(n), and so $C(n) = O(n)$ [7].

Merge sort is faster in larger lists because it does not run over the entire list many times. In addition, the merge sort has a consistent running time of ($nlogn$) in all three cases. On the other hand, Merge sort is slower than the other sort algorithms for smaller data sets and requires more memory space to store the sub-lists. That means it takes up more space [3].

C. Heap Sort Algorithm

Heap sort is an improved sort algorithm of selection sort. This is performed on the heap data and the heap is basically the complete binary tree [2]. It is also a comparison-based sorting technique based on the Binary Heap data structure. The heap sort algorithm is in place order and can max heap (the root is the largest element and bigger than its children)or min heap (the root is the smallest element and is smaller than its children) [13]. The complexity of heap sort is $O(nlogn)$ for all the cases. Because the time complexity of building a heap is $O(n)$ and $n-1$ call heapify that takes $O(logn)$ and the complete time complexity is $O(nlogn)$ [2]. it will work like this, First, create a heap from the input array, Second it will visualize the array with the correct property of binary tree by using heapify (iterate each node), Finally apply heap sort(for all tree violations) all of them inside Build function [4]. The advantages of heap sort are optimized performance, efficiency, and accuracy are a few of the best qualities of this algorithm. The algorithm is also highly consistent with very low memory usage. No extra memory space is required to work, unlike the Merge Sort or recursive Quick Sort. However, heap sort is considered unstable, expensive, and not very efficient when working with highly complex data [10].

III. METHODOLOGY

A. *Experimental Setup*

This section presents the used machine and platform.

1) ***Used Machine:*** MacBook Pro. The startup disk is Macintosh HD. The operating system is iOS and the software version is macOS Monterey. The processor is a Quad-Core Intel Core i5 with a speed of 1.4 GHz and 8 GB memory.

2) ***Used Tools:***

- **Apache NetBeans** is used to run Java language code. NetBeans is a Java-integrated development environment (IDE). NetBeans enables the development of applications from a set of modular software components known as modules. NetBeans is available for Windows, macOS, Linux, and Solaris [9].
- **Microsoft Excel** is used to analyze the results. Excel is the industry-leading spreadsheet software program, a powerful data visualization, and analysis tool [1].

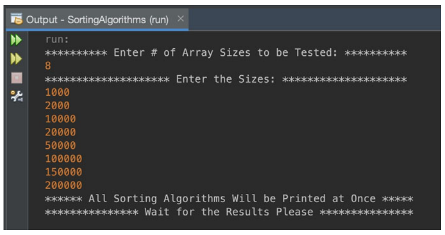
B. *Data Generation*

This section explains how the data was generated in our program.

1) ***Algorithms inputs selection:*** The inputs were selected depending on the entered array size by the user as shown in Figure 1 below, then all algorithms will be tested for all the sizes entered for three cases (best, average, and worst). This way was used to minimize the time consumed for entering each size separately. The array types can be in three orders:

- **Increasing (Best Case):** Use the same array after sorting. This array is considered the best case.
- **Random (Average Case):** Generate an array of unsorted elements using a random method of package "java.util.Random".
- **Decreasing (Worst Case):** Generate a reversed array with decreasing sorted elements.

Manuscript received December 5, 2022
Manuscript revised December 20, 2022
https://doi.org/10.22937/IJCSNS.2022.22.12.25



```
run:
********** Enter # of Array Sizes to be Tested: **********
8
******************** Enter the Sizes: ********************
1000
2000
10000
20000
50000
100000
150000
200000
****** All Sorting Algorithms Will be Printed at Once *****
*************** Wait for the Results Please ***************
```

Fig. 1. Program main screen

The random method will is used to display different numbers from 0 to 2000. Every time the code runs, it will generate different array elements due to using the random method. In the increasing case, the program will use the same array of the random, but after being sorted. While in the decreasing, a for loop is used to sort elements decreasingly.

2) ***Timing Mechanism:*** The program uses the same array in every algorithm to find the running time. The n tested sizes are 1000, 2000, 10000, 20000, 50000, 100000, 150000, and 200000. To find the execution time, nanoTime() method in Java was used. The method works by taking the start time and end time of the system in the following format: *long (object for start or end) = System.nanoTime()*. Then, subtract the start time from the end time as follow: *long (object name to save the results) = end - start*. Finally, the results in section IV will be shown in milliseconds.

IV. RESULTS

A. *Performance of Three Sorts*

This section compares the best performance of three sorts in terms of the number of comparisons in the worst case and running time in different cases, which are, the best case, the average case, and the worst case.

1) ***Best Case Running Time::*** Table I and Figure 2 show the running time of three sort algorithms in the increasing array based on the number of elements and the used algorithm. The results show that as the number of elements increases, the execution time also increases. However, the insertions sort has the lowest running time in this case.

TABLE I

COMPARISON OF ALGORITHMS IN BEST CASE

Best Case

# of Elements	*Insertion*	*Merge*	*Heap*
1000	33.503	965.819	952.486
2000	24.516	2073.397	1175.025
10000	123.022	539.335	5265.575
20000	245.264	1169.774	1008.147
50000	629.535	21382.793	25952.665
100000	1271.088	40686.591	53284.744
150000	1715.798	64203.761	83897.952
200000	2412.414	117437.992	130566.874

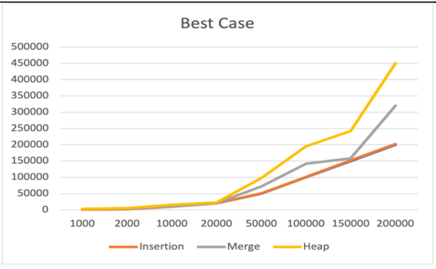


Fig. 2. Best case of three sorts

2) ***Average Case Running Time:*** Table II and Figure 3 show the running time of three sort algorithms in the random array elements based on the number of elements and the used algorithm. The results show that as the number of elements increases, the execution time of insertion sort also increases, which agrees with the theory that the complexity of insertion sort in average and worst cases is O(n^2). However, the insertions sort has the highest running time in this case. The merge sort has less running time than the heap sort.

TABLE II

COMPARISON OF ALGORITHMS IN AVERAGE CASE

Average Case

# of Elements	*Insertion*	*Merge*	*Heap*
1000	754.874	1122.511	978.721
2000	329.914	2424.892	120.403
10000	7598.451	5916.253	550.802
20000	34739.784	14353.147	10799.151
50000	220007.479	24870.058	28464.242
100000	868804.217	4954.337	57776.533
150000	1960932.746	67756.309	98784.478
200000	393903.125	109421.212	134257.488

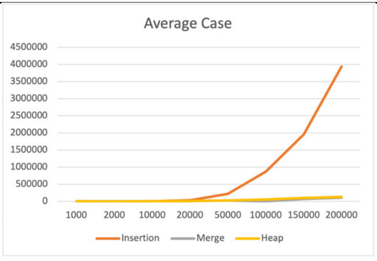


Fig. 3. Average case of three sorts

3) ***Worst Case Running Time:*** Figure 4 and Table III show the running time of three sorting algorithms in the decreasing array based on the number of elements and the used algorithm. The results show that as the number of elements increases, the execution time of insertion sort also increases, which agrees with the theory that the complexity of insertion sort in average and worst cases is O(n^2). However, the insertions sort has the highest running time in this case. The merge sort has less running time than the heap sort (same as the average case).

TABLE III

COMPARISON OF ALGORITHMS IN WORST CASE

Worst Case

# of Elements	*Insertion*	*Merge*	*Heap*
1000	1926.143	1599.623	1203.553
2000	7298.100	2122.927	1248.841
10000	7911.923	8165.13	5574.589
20000	33122.914	12371.737	1055.474
50000	224438.002	25566.416	27903.156
100000	870811.548	50668.821	57986.197
150000	1992570.014	67747.504	92434.554
200000	3634577.868	103713.9	137524.641

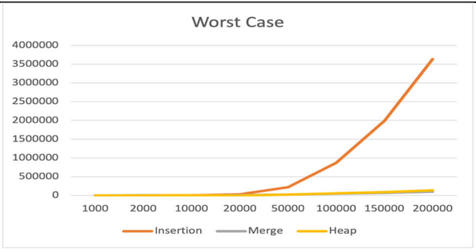


Fig. 4. Worst case of three sorts

4) ***Number of Elements vs Number of Comparisons vs Running Time****:* A comparison has been done based on the number of elements, the number of comparisons, and the running time of each algorithm in the worst case. Figure 5 illustrates the number of comparisons and running time based on the number of elements (n). Where CI and RTI are the comparison and running time of insertion sort, CM and RTM are the comparison and running time of merge sort, and CH and RTH are the comparison and running time of heap sort. The number of comparisons can be calculated as follows:

- Number of comparisons in insertion sort = $n^2/2$.
- Number of comparisons in merge sort = *logn*.
- Number of comparisons in heap sort = *nlogn*.

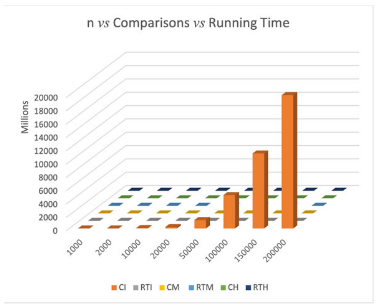


Fig. 5. # of Elements vs # of Comparisons vs Running Time

5) ***Discussion of the best performance sorting algorithm****:* The running time of all the existing algorithms are listed in Tables I, II, and III and shown in Figures 2, 3, 4, and 5. The running time of each algorithm is given in terms of milliseconds. The number of elements gradually increased, and the corresponding running time is separately recorded by running the algorithms. As shown in the previously mentioned tables and figures, as the number of elements increases, the running time and number of comparisons also increase for all algorithms. However, the algorithms using the incremental list, which is the best case, achieved relatively less execution time. Furthermore, in the best case, the performance of the insertion sort becomes faster when the list is sorted and has a minimum number of elements, which is more efficient than the heap and merge sorts even when the list elements number increases. In the average and worst cases, the performance of the merge sort is faster than the insertion and heap sorts when they have a large number of elements because the merge sort algorithm uses the divide and conquer technique with running time O(nlogn). For the number of comparisons, the insertion sort has the highest number of comparisons as the number of elements increases. However, the merge and heap sorting algorithms have the same number of comparisons because of the same previously mentioned comparison count formula.

*B. **Theoretical VS Experimental Results Comparison***

In this section, experimental results of all cases in all sorting algorithms with their expected theoretical result are compared.

1) ***Insertion Sort****:* The results of each case are shown Theoretically and experimentally in Table IV. The results of the insertion sort experimentally agree with the theoretical analysis of insertion sort, which is O(n) for the best case and O(n^2) for the average and worst cases. Figure 6, 7, and 8 illustrate the line graph on insertion in all cases.

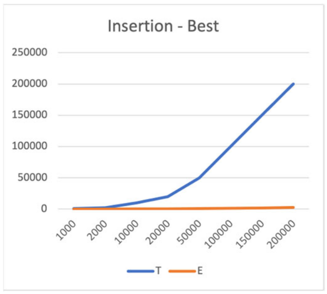


Fig. 6. Best case of insertion sort

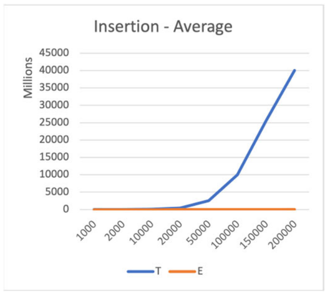


Fig. 7. Average case of insertion sort

Manuscript received December 5, 2022
Manuscript revised December 20, 2022
https://doi.org/10.22937/IJCSNS.2022.22.12.25

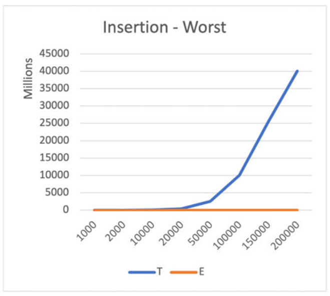


Fig. 8. Worst case of insertion sort

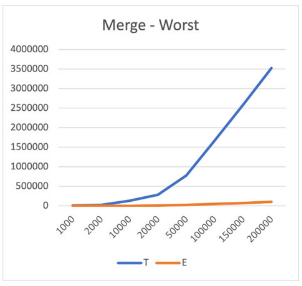


Fig. 11. Worst case of Merge sort

TABLE IV

THEORETICAL(T) VS EXPERIMENTAL(E) RESULTS OF INSERTION SORT

n	Best		Average		Worst	
	T	E	T	E	T	E
1000	1000	33.503	1000000	754.874	1000000	1926.143
2000	2000	24.516	4000000	329.914	4000000	72.98100
10000	10000	123.022	100000000	7598.451	100000000	7911.923
20000	20000	245.264	400000000	34739.784	400000000	33122.914
50000	50000	629.535	2500000000	220007.479	2500000000	224438.002
100000	100000	1271.088	10000000000	868804.217	10000000000	870811.548
150000	150000	1715.798	25500000000	1960932.746	25500000000	1992570.014
200000	200000	2412.414	40000000000	3939031.25	40000000000	3634577.868

2) ***Merge Sort****:* The results of each case in merge sort are shown Theoretically and experimentally in Table V. The results of the merge sort experimentally agree with the theoretical analysis of merge sort, which is O(n log n) in all cases. Figure 9, 10, and 11 illustrate the merge in all cases.

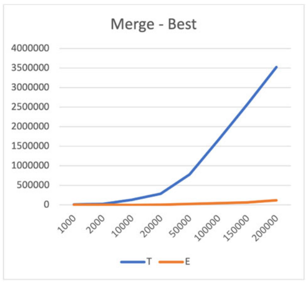


Fig. 9. Best case of Merge sort

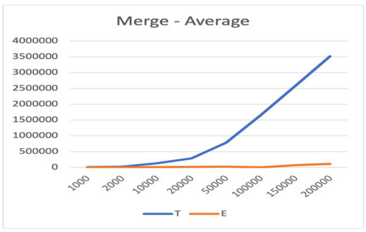


Fig. 10. Average case of Merge sort

3) ***Heap Sort****:* The results of each case in heap sort are shown Theoretically and experimentally in Table VI. The results of the heap sort experimentally agree with the theoretical analysis of heap sort, which is O(n log n) in all cases. Figure 12, 13, and 14 illustrate the heap in all cases.

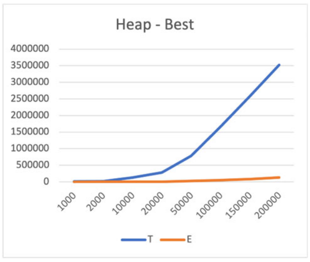


Fig. 12. Best case of heap sort

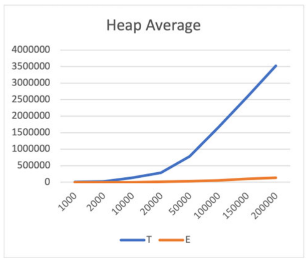


Fig. 13. Average case of heap sort

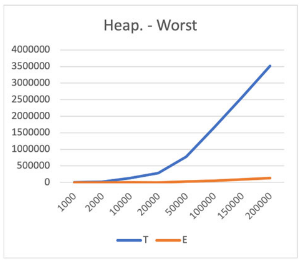


Fig. 14. Worst case of heap sort

operations. Sorting algorithms rearrange the elements of an array or list based on the elements' comparison operators. The comparison operator is used in the accurate data structure to establish the new order of elements. This paper analyzed and compared the time complexity and running time theoretically and experimentally of insertion, merge, and heap sort algorithms. Java language was used by the NetBeans tool to implement the code of the algorithms. Microsoft Excel was used to present the experimental results figures. The results show that when dealing with sorted elements, insertion sort operates in a faster running time than merge and heap algorithms. When it comes to dealing with a large number of elements, it is better to use the merge sort. For the number of comparisons for each algorithm, the insertion sort has the highest number of comparisons.

TABLE V

THEORETICAL(T) VS EXPERIMENTAL(E) RESULTS OF MERGE SORT

n	Best		Average		Worst	
-	T	E	T	E	T	E
1000	9965.784285	965.819	9965.784285	1122.511	9965.784285	1599.623
2000	21931.56857	2073.397	21931.56857	2424.892	21931.56857	2122.927
10000	132877.1238	539.335	132877.1238	5916.253	132877.1238	8165.13
20000	285754.2476	1169.774	285754.2476	14353.147	285754.2476	12371.737
50000	780482.0237	21382.793	780482.0237	24870.058	780482.0237	25566.416
100000	1660964.047	40686.591	1660964.047	4954.337	1660964.047	50668.821
150000	2579190.446	64203.761	2579190.446	67756.309	2579190.446	67747.504
200000	3521928.095	117437.992	3521928.095	109421.212	3521928.095	1037.139

TABLE VI

THEORETICAL(T) VS EXPERIMENTAL(E) RESULTS OF HEAP SORT

n	Best		Average		Worst	
-	T	E	T	E	T	E
1000	9965.784285	952.486	9965.784285	978.721	9965.784285	1203.553
2000	21931.56857	1175.025	21931.56857	120.403	21931.56857	1248.841
10000	132877.1238	5265.575	132877.1238	550.802	132877.1238	5574.589
20000	285754.2476	1008.147	285754.2476	10799.151	285754.2476	1055.474
50000	780482.0237	25952.665	780482.0237	28464.242	780482.0237	27903.156
100000	1660964.047	53284.744	1660964.047	57776.533	1660964.047	57986.197
150000	2579190.446	83897.952	2579190.446	98784.478	2579190.446	92434.554
200000	3521928.095	130566.874	3521928.095	134257.488	3521928.095	137524.641

V. CONCLUSION

Sorting is an important data structure in many applications in the real world. Several sorting algorithms are currently in use for searching and other

Manuscript received December 5, 2022
Manuscript revised December 20, 2022
https://doi.org/10.22937/IJCSNS.2022.22.12.25

REFERENCES

[1] Microsoft excel spreadsheet software: Microsoft 365.

[2] Humaira Ali, Haque Nawaz, and Abdullah Maitlo. Performance analysis of heap sort and insertion sort algorithm. *International Journal*, 9(5), 2021.

[3] Ayush Arora. Sorting algorithms-properties/pros/cons/comparisons, Dec 2019.

[4] Marco Benini and Federico Gobbo. Algorithms and their explanations. In *Conference on Computability in Europe*, pages 32–41. Springer, 2014.

[5] Ashutosh Bharadwaj and Shailendra Mishra. Comparison of sorting algorithms based on input sequences. *International Journal of Computer Applications*, 78(14), 2013.

[6] Miss Pooja K Chhatwani. Insertion sort with its enhancement. *International Journal of Computer Science and Mobile Computing*, 3(3):801–806, 2014.

[7] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[8] Jehad Hammad. A comparative study between various sorting algorithms. *International Journal of Computer Science and Network Security (IJCSNS)*, 15(3):11, 2015.

[9] Patrick Keegan, Ludovic Champenois, and Gregory Crawley. Netbeans™ ide field guide, 2006.

[10] Christos Levcopoulos and Ola Petersson. Adaptive heapsort. *Journal of Algorithms*, 14(3):395–413, 1993.

[11] Zbigniew Marszałek. Performance test on triple heap sort algorithm. *Technical Sciences/University of Warmia and Mazury in Olsztyn*, 2017.

[12] Smita Paira, Sourabh Chandra, Sk Safikul Alam, and Partha Sarthi Dey. A review report on divide and conquer sorting algorithm. *IEEE Kolkata Section*, pages 978–993, 2014.

[13] Russel Schaffer and Robert Sedgewick. The analysis of heapsort.
Journal of Algorithms, 15(1):76–100, 1993.