Kingdom of Saudi Arabia
Ministry of Education
Imam Abdulrahman bin Faisal University
Computer Science department

Algorithm Analysis and Design

Term 1 – 2022/2023

CIS Year 4, G1

Supervised By:

Dr. Azza A. Ali

Student Name	Student ID	Role
May M. AlOtaibi	2200004606	Leader
Bushra M. Alshehri	2200004242	Member
Jumana Y. Aljassim	2200006269	Member
Fida M. Alelou	2200003041	Member
Ghala M. Alkhaldi	2200003157	Member

List of Content

List of Figures

List of Tables

Introduction

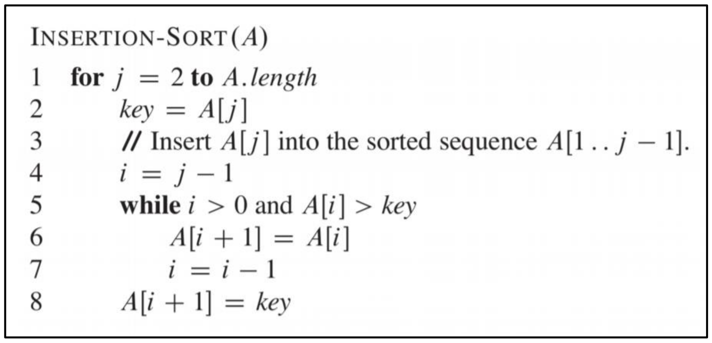
A code is first structured by writing an algorithm in a programming language, such as C++, Java, JavaScript, or C#. In algorithmic thinking, a problem is solved by a set of procedures. As well as providing clarity, the algorithm used by the programmer increases efficiency. There are many types of data structures that can be implemented in the code, including arrays, linked lists, stacks, and queues. Moreover, these structures can be used to perform a wide range of operations. For example, Traversal, Insertion, Deletion, Searching, Sorting, and Merging. There are several types of sorts that are used to sort data in distinct order such as increasing order, decreasing order, or random order. Which are: Insertion sort, heap sort, and merge sort. Sorting algorithms can be used to reduce the complexity of a problem and improve its efficiency [1].

1. Theoretical question

This section presents the Three sorts of explanation, algorithm, and time complexity for each case.

1.1 Insertion sort

Is a simple sorting algorithm that works like the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part. **Figure 1** shows the insertion algorithm [2].



```
INSERTION-SORT(A)
for $j = 2$ to $A.length$
    $key = A[j]$
    // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

Figure 1: Insertion Sort Algorithm

	Best Case	**Average Case**	**Worst Case**
Data Order	Data sorted increasingly.	Between best and worst case.	Data sorted decreasingly.
Time Complexity	$T(n) = \Theta(n)$	$T(n) = \Theta(n^2)$	$T(n) = \Theta(n^2)$

Table 1: Insertion Sort Algorithm's information

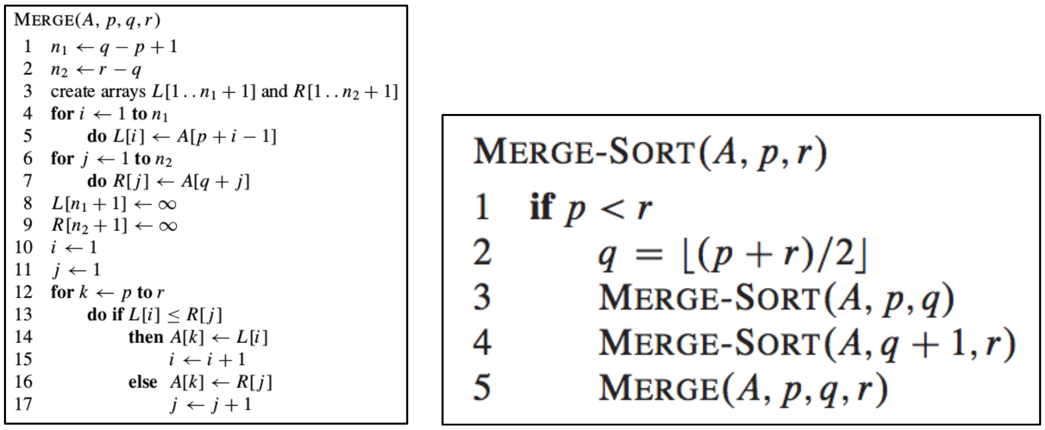
1.2. Merge sort

Is a type of data sorting. By merging arrays, an unsorted array can be divided into smaller arrays until one element remains in the array. Once each array has been sorted, merge them with each other and so on until you have the sorted array [5]. The algorithm of merge sort is shown in **Figure 2** below. Merge sort by using the Divide and Conquer approach. The concept of Divide and Conquer involves three steps [6]:

1. Divide: the problem into a number of subproblems.
2. Conquer: the subproblems by:

   - Solving the sub-problems recursively.
   - If the Sub-problem sizes are small enough, then solve the sub-problems in straightforward manner.

3. Combine: the solutions of the subproblems.



```
MERGE($A, p, q, r$)
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$
for $i \leftarrow 1$ to $n_1$
    do $L[i] \leftarrow A[p + i - 1]$
for $j \leftarrow 1$ to $n_2$
    do $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ to $r$
    do if $L[i] \le R[j]$
        then $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$
```

```
MERGE-SORT($A, p, r$)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT($A, p, q$)
    MERGE-SORT($A, q + 1, r$)
    MERGE($A, p, q, r$)
```

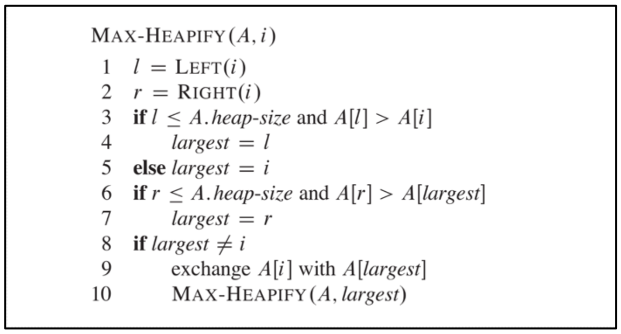
Figure 2: Merge Sort Algorithm

	Best Case	**Average Case**	**Worst Case**
Data Order	Data sorted increasingly.	Between best and worst case.	Data sorted decreasingly.
Time Complexity	$T(n) = \Theta(n \log n)$	$T(n) = \Theta(n \log n)$	$T(n) = \Theta(n \log n)$

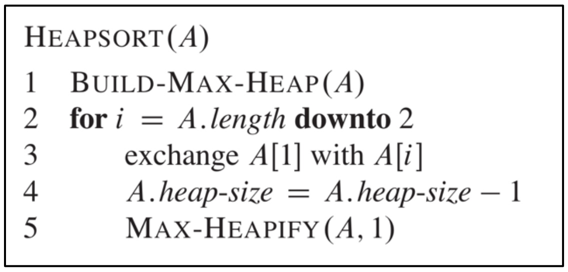
Table 2: Merge Sort Algorithm's information

1.3. Heap sort

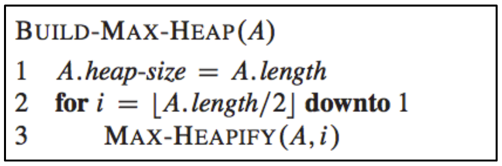
Is a comparison-based sorting technique based on Binary Heap data structure. It is like the selection sort where we first find the minimum element and place the minimum element at the beginning [3]. Repeat the same process for the remaining elements. It is one of the efficient sorting algorithms based on heap data structure [4].



```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```



```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```



```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

Figure 3: Heap Sort Algorithm

	Best Case	Average Case	Worst Case
Data Order	Data sorted increasingly.	Between best and worst case.	Data sorted decreasingly.
Time Complexity	$T(n) = \Theta(n \log n)$	$T(n) = \Theta(n \log n)$	$T(n) = \Theta(n \log n)$

Table 3: Heap Sort Algorithm's information

2. Data generation and experimental setup

This section is split into subsections that describe the features of the machine that is being used, the timing mechanism, the selection process for the input, and the input for each sort.

2.1. Description of the characteristics of the machine used

This section shows the characteristics of the device used in this experiment.

Random Access Memory (RAM)	Central processer unit (CPU)	Operating System (OS)	System Type
8 G	M1	MAC OS	64-bit

Table 4: characteristics of the device used in this experiment.

2.2. Timing mechanism

The ***System.nanoTime()*** function has been used in this project to return (ns), which was afterward divided by 1000 to get (μs) as the unit of time measurement for each sort's state. The time received from the code was then used to compare the sorting.

2.3. How many times did you repeat each experiment?

We have executed/run each algorithm a minimum four (04) times in order to evaluate the behavior of the algorithm effectively. We have noticed that most algorithms take more time when the inputs to the algorithm are increasing. Due to this reason, each algorithm was run for a different input size.

2.4. What times are reported?

We have taken the time as the average of the four obtained execution time. Mostly the difference in time execution was constantly increasing. Although the running time of the algorithm was correctly calculated, but we have calculated the average time for each algorithm. The average time shows a unique value which represents the running time in conciseness.

2.5. How did you select the inputs?

The input to the algorithm ranges from 100 to 100000. This range was selected because it demonstrates the impact of input size on running time for each algorithm in a straightforward manner. Since the running time for some algorithms became too long and was causing the program to delay, we terminated at the value of 100000.

2.6. Did you use the same inputs for all sorting algorithms?

In the situation of increasing, decreasing, and random order, this project will use the same number of inputs for all sorting algorithms like 100, 500, 1000, … . All sorting algorithms should produce results that are reliable and correct, which is why the same input is used for all of them. The same quantity of inputs was utilized to ensure a fair comparison between all the types.

3. The three sorts perform the best

The result of the experiment will be displayed in this question as the best, average, and worst cases of each sorting algorithm. X-axis represents input size, and y-axis represents median time (in microseconds) in the line graphs below.

3.1. Graph the best-case running time as a function of input size n for the three sorts (use the best case input you determined in each case in part 1)

Figure 4 below represents the best case of each sorting algorithm, namely insertion sort, heap sort, and merge sort. The best performance among the three sorts is the insertion sort.

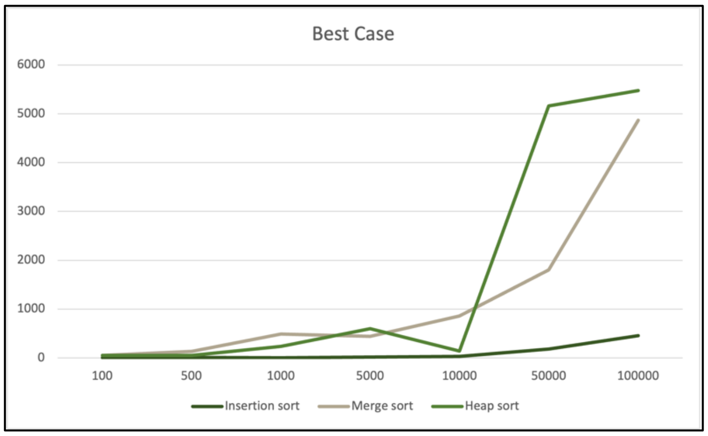


Figure 4: Best case of all sorting algorithm

3.2. Graph the worst-case running time as a function of input size n for the tree sorts (use the best case input you determined in each case in part 1)

Figure 5 below represents the worst case of each sorting algorithm, namely insertion sort, heap sort, and merge sort. The best performance among the three sorts is the merge sort.

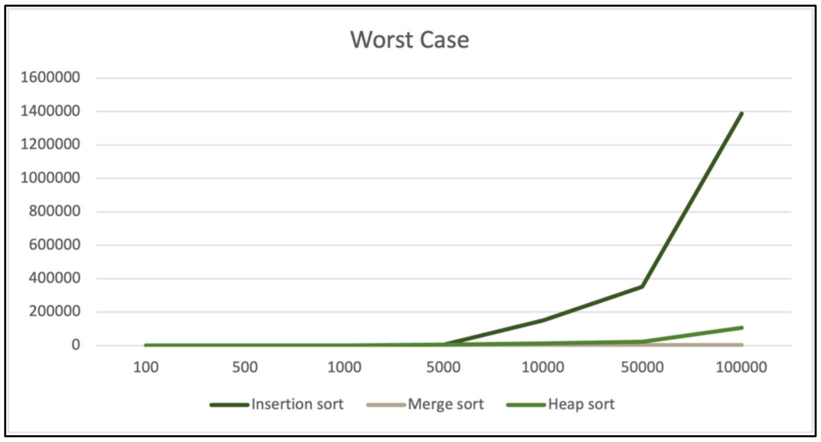


Figure 5: Worst case of all sorting algorithm

3.3. Graph the average case running time as a function of input size n for the three sorts

Figure 6 below represents the average case of each sorting algorithm, namely insertion sort, heap sort, and merge sort. The best performance among the three sorts is the merge sort.

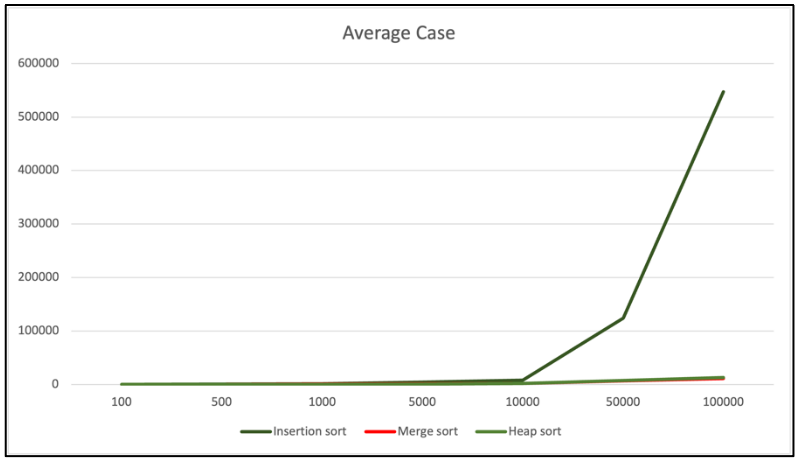


Figure 6: Average case of all sorting algorithm

3.4. Discussion of the best performance sorting algorithm

It is essential to consider large input sizes when analyzing the best running time. Since we always consider the worst case, we conclude that merging sort is the fastest sorting algorithm. Theoretically, heap sort is more efficient than merge sort because it doesn't require massive recursive calls and multiple arrays. On the other hand, merge sort is faster than heap sort if the input size is large. As a result, the merge sort is the most efficient. Despite the merge sort algorithm's efficiency, for small input sizes, the fastest algorithm may vary. Also, if the list were sorted, insertion sort would be the most efficient sorting algorithm.

4. To what extent does the best, average, and worst-case analyses (from class/textbook) of each sort agree with the experimental results?

Asymptotic order of theoretical running time is compared to asymptotic order of experimental running time by dividing theoretical running time by experimental running time in this question. Experimental results may differ from theoretical results in some cases. A number of factors could have an effect on the running time, such as CPU utilization, compiler speed, and programming language. Input size is represented on the x-axis, whereas the y-axis represents the quotient of theoretical and experimental run times.

4.1. For each sort, and for each case (best, average, and worst), determine whether the observed experimental running time is of the same order as predicted by the asymptotic analysis

4.1.1. Insertion sort

Cases	**Best Case (μs)**		**Average Case (μs)**		**Worst case (μs)**	
Input size (n)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)
100	100	3.084	10000	32.458	10000	80.67025
500	500	14.375	250000	586.333	250000	656.54775
1000	1000	3.417	1000000	1430.75	1000000	1229.53125
5000	5000	17.583	25000000	4598.666	25000000	5729.19825
10000	10000	35.125	100000000	8341.125	100000000	15117.479
50000	50000	184.541	2500000000	123926	2500000000	339167.552
100000	100000	456.792	10000000000	546964	10000000000	1356358.46

Table 5: Insertion Sort Algorithm's Time Complexity Case

Cases	**Best Case**	**Average Case**	**Worst case**
Input size (n)	P - × 100 T	P -× 100 T	P -× 100 T
100	3.084	0.32458	0.8067025
500	2.875	0.2345332	0.2626191
1000	0.3417	0.143075	0.12295313
5000	0.35166	0.018394664	0.02291679
10000	0.35125	8.3411×10^{-3}	0.01511748
50000	0.369082	4.95704×10^{-3}	0.0135667
100000	0.456792	5.46964×10^{-3}	0.01356358

Table 6: Error Ratio for the Insertion Sort Algorithm's cases

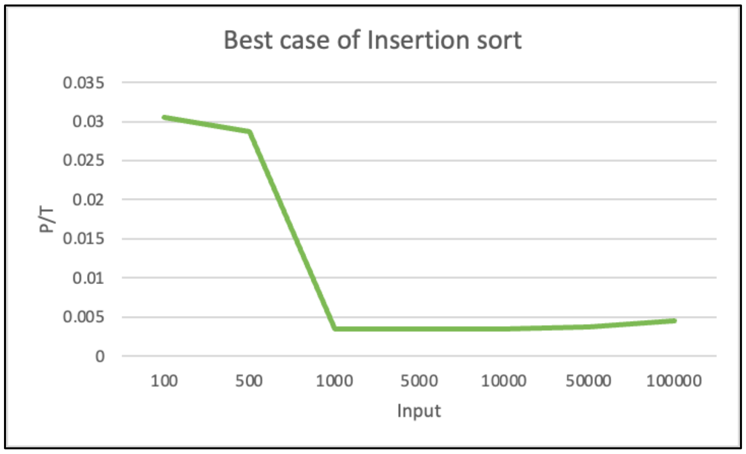


Figure 7: Best case of Insertion sort

The insertion sort best-case analysis is illustrated in **Figure 7**. Based on the line graph above, a horizontal line appeared at n0=1000 and P/T=0.003417. Based on these results, we can conclude that the experiment confirms the theory, which is Θ(n).

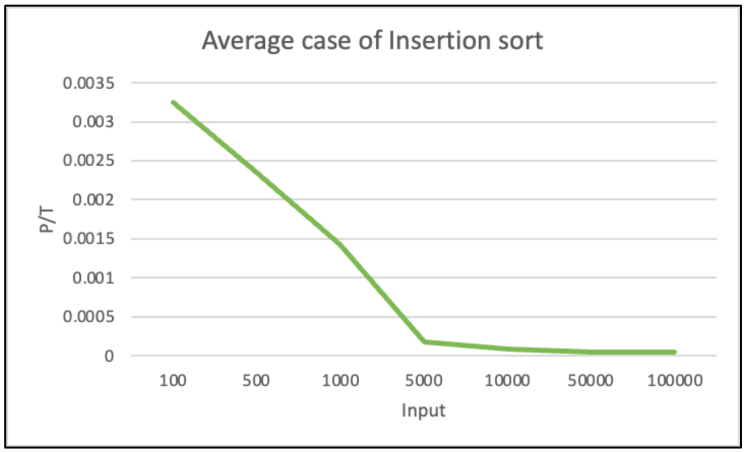


Figure 8: Average case of Insertion sort

The insertion sort best-case analysis is illustrated in **Figure 8**. Based on the line graph above, a horizontal line appeared at n0=50000 and P/T=4.95704$\times$ 10^{-5}. Based on these results, we can conclude that the experiment confirms the theory, which is $\Theta(n^2)$.

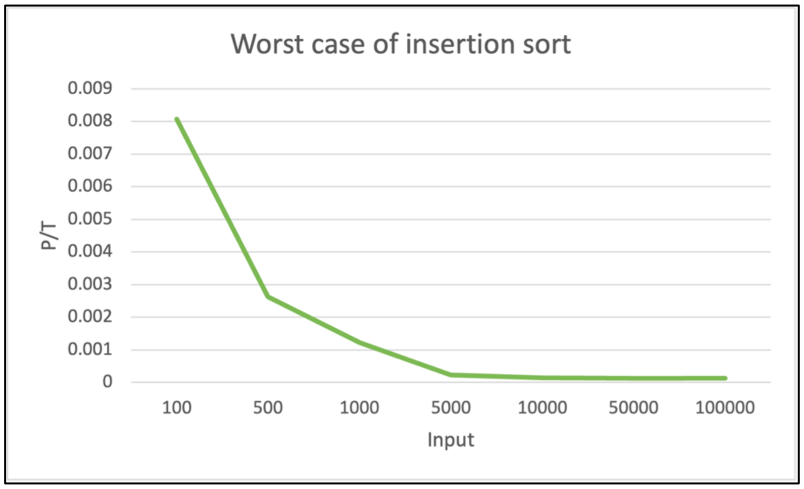


Figure 9: Worst case of Insertion sort

The insertion sort best-case analysis is illustrated in **Figure 9**. Based on the line graph above, a horizontal line appeared at n0=5000 and P/T= 1.401367$\times$ 10^{-4}. Based on these results, we can conclude that the experiment confirms the theory, which is $\Theta(n^2)$.

4.1.2. Merge sort

Cases	Best Case (μs)		Average Case (μs)		Worst case (μs)	
Input size (n)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)
100	200	52.542	200	50.083	200	43.541
500	1349.485002	134.584	1349.485002	289.541	1349.485002	32.417
1000	3000	491.542	3000	595.5	3000	74.375
5000	18494.85002	441.042	18494.85002	1223.125	18494.85002	376.452
10000	40000	859.542	40000	1956.083	40000	825.25
50000	234948.5002	1806.5	234948.5002	6806.542	234948.5002	1758
100000	500000	4869.291	500000	11091.292	500000	3110.459

Table 7: Merge Sort Algorithm's Running Time Case

Cases	Best Case	Average Case	Worst case
Input size (n)	P - × 100 T	P -× 100 T	P -× 100 T
100	26.271	25.0415	21.7705
500	9.973	21.455666	2.402175641
1000	16.385	19.85	2.479166667
5000	2.38467	6.613327	2.035442297
10000	2.148285	4.891355	2.063125
50000	0.768892	2.897035	0.748240837
100000	0.9738582	22.182584	0.622099

Table 8: Error Ratio for the Merge Sort Algorithm's cases

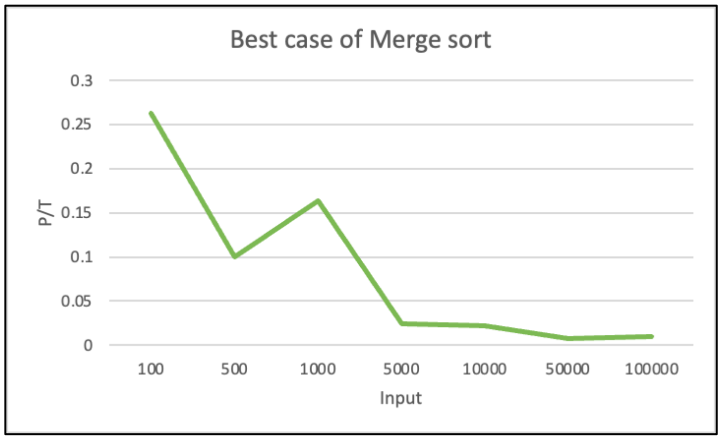


Figure 10: Best case of Merge sort

The Merge sort best-case analysis is illustrated in **Figure 10**. Based on the line graph above, a horizontal line appeared at n0=50000 and P/T=7.68892× 10^{-3}. Based on these results, we can conclude that the experiment confirms the theory, which is Θ(n lg n).

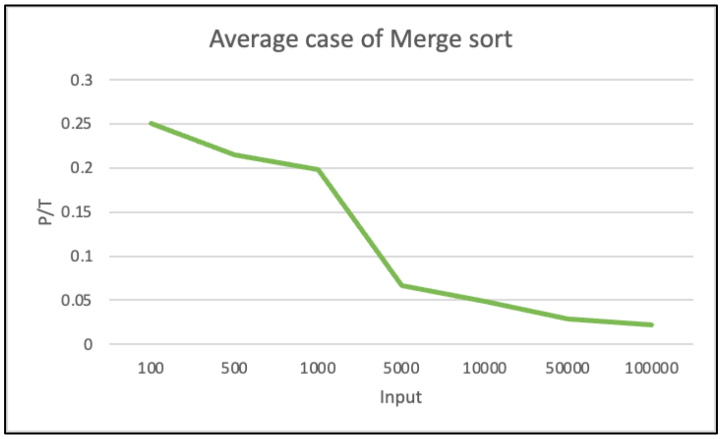


Figure 11: Average case of Merge sort

Figure 11 shows the merge sort average-case analysis. As shown in the line graph above, there is no clear horizontal line. Our findings support the theory, but the experiment contradicts it. Interestingly, the line graph almost becomes horizontal when the input size is 50000. As a result, if we increase the input size, a horizontal line might appear. Repeating the experiment more than three times might also have produced a horizontal line.

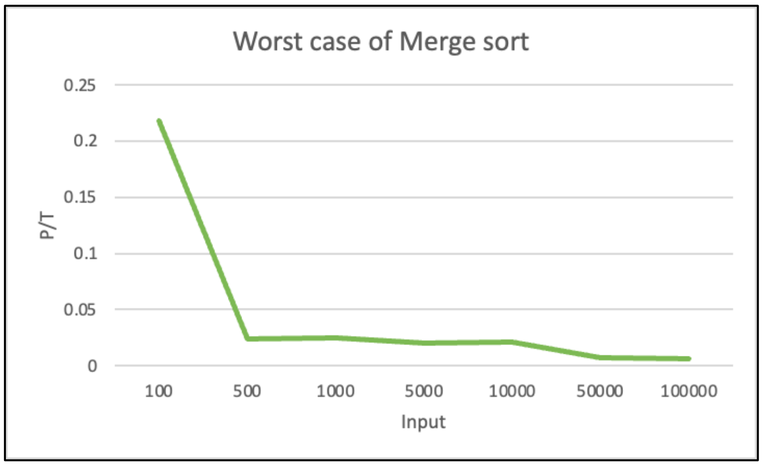


Figure 12: Worst case of Merge sort

The Merge sort worst-case analysis is illustrated in **Figure 12**. Based on the line graph above, a horizontal line appeared at n0=50000 and P/T=7.48240837× 10^{-3}. Based on these results, we can conclude that the experiment confirms the theory, which is Θ(n lg n).

4.1.3. Heap sort

Cases	**Best Case (μs)**		**Average Case (μs)**		**Worst case (μs)**	
Input size (n)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)
100	200	50.583	200	51.959	200	56.042
500	1349.485002	52.625	1349.485002	172.167	1349.485002	380.291
1000	3000	238.417	3000	442.583	3000	1182.125
5000	18494.85002	596.583	18494.85002	820.791	18494.85002	4579.834
10000	40000	142.041	40000	2016.209	40000	12462.583
50000	234948.5002	5157.834	234948.5002	7734.667	234948.5002	22931.45
100000	500000	5474.541	500000	13126.875	500000	105238.75

Table 9: Heap Sort Algorithm's Running Time Case

Cases	Best Case	Average Case	Worst case
Input size (n)	P - × 100 T	P -× 100 T	P -× 100 T
100	25.2915	25.97.95	28.021
500	3.899635781	12.757978	28.180454
1000	7.947233333	14.75276667	39.4041667
5000	3.225670927	4.437943531	24.7627528
10000	0.3551025	50.405225	31.1564575
50000	2.195304075	3.292069	9.76020276
100000	1.0949082	2.625375	21.04775

Table 10: Error Ratio for the Heap Sort Algorithm's cases

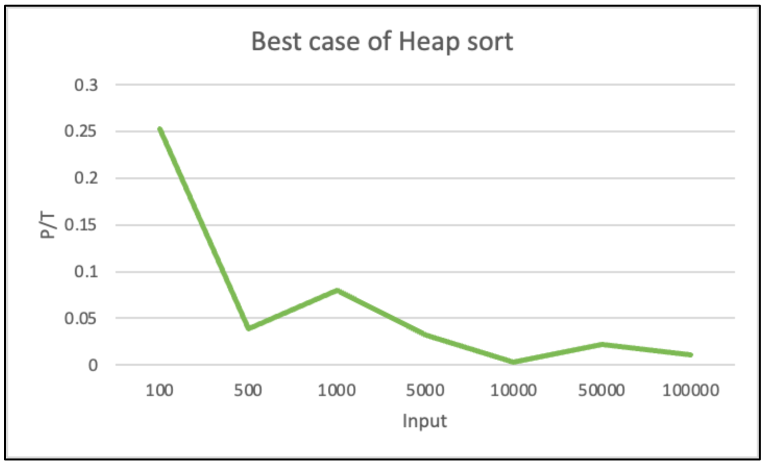


Figure 13: Best case of Heap sort

Figure 13 shows the heap sort best-case analysis. According to the line graph above, there is no clear horizontal line appearing. However, the difference between the experimental running time at the inputs 50000 and 100000 started to get closer compared to the difference between the other consecutive inputs. Therefore, a horizontal line might have appeared if we tried larger input sizes.

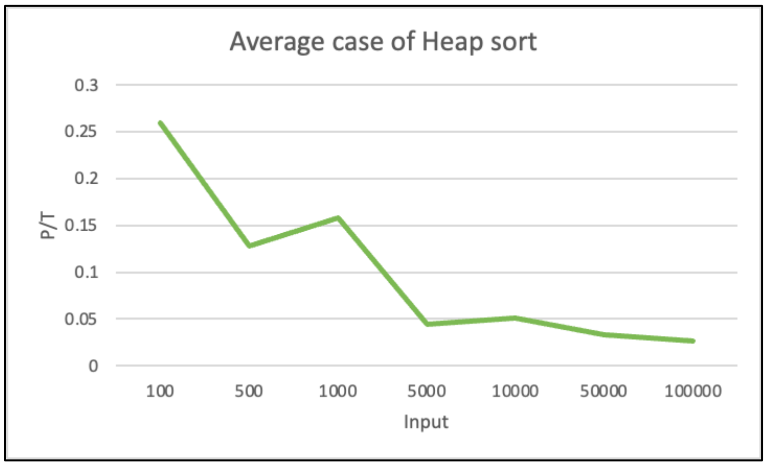


Figure 14: Average case of Heap sort

Figure 14 shows the heap average-case analysis. According to the line graph above, there is no clear horizontal line appearing. However, the difference between the experimental running time at the inputs 50000 and 100000 started to get closer compared to the difference between the other consecutive inputs. Therefore, a horizontal line might have appeared if we tried larger input sizes.

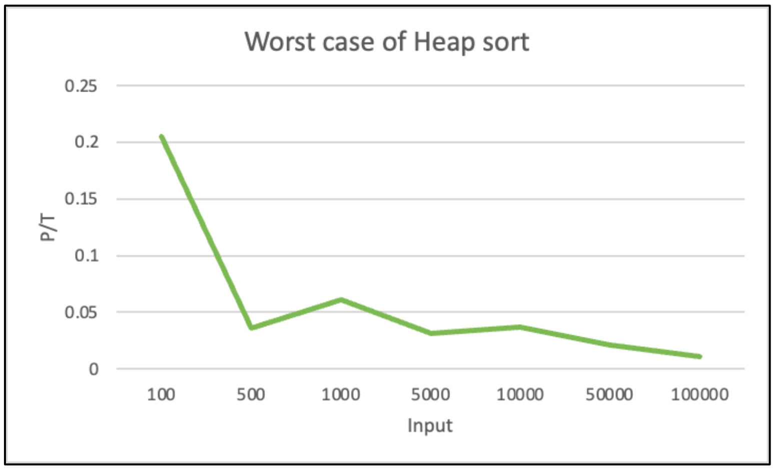


Figure 15: Worst case of Heap sort

Figure 15 shows the heap sort worst-case analysis. According to the line graph above, there is no clear horizontal line appearing. However, the difference between the experimental running time at the input 100000 started to get closer compared to the difference between the other consecutive inputs. Therefore, a horizontal line might have appeared if we tried larger input sizes.

4.2. Your determination should be backed up by your experiments and analysis and you must explain your reasoning. If you found the sort did not conform to the asymptotic analysis, you should try to understand why and provide an explanation

For each sort's case, the asymptotic theoretic analysis shows the maximum time it would take. Therefore, the experimental analysis might be slower than the theoretical analysis. This is due to factors like CPU speed.

5. For the comparison sorts, is the number of comparisons really a good predictor of the execution time?

For each sorting algorithm, we plotted a line graph to find the relationship between the number of comparisons and the running time. All three sorting algorithms were compared using the worst case, as it is the best indicator. Accordingly, we assumed a positive relationship between the number of comparisons and the running time before plotting the line graphs.

5.1. Insertion sort algorithm

According to the table below, we performed the following comparisons during the sorting process: using Equation ($n^2/2$). Based on the Insertion sort programmed code, we calculated the execution time. In Figure 16 Input size is represented on the x-axis, whereas the y-axis represents the number of comparisons. Figure 17 the Input size is represented on the x-axis, whereas the y-axis represents the execution time.

Input Size (n)	Number of comparisons	Execution Time (μs)
100	5000	79.167
500	125000	795.25
1000	500000	1217.330
5000	12500000	5898.959
10000	50000000	150242
50000	1250000000	350341.75
100000	5000000000	1388543.417

Table 11: Insertion Sort Algorithm

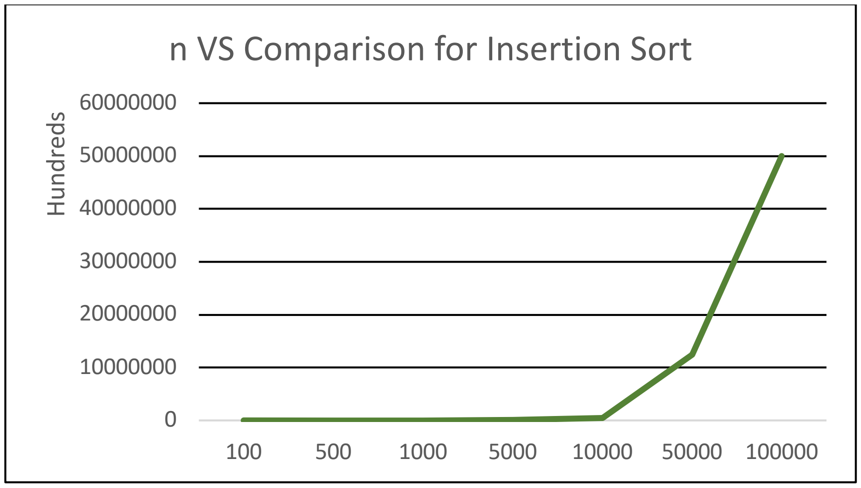


Figure 16: n Vs Comparison for Insertion Sort

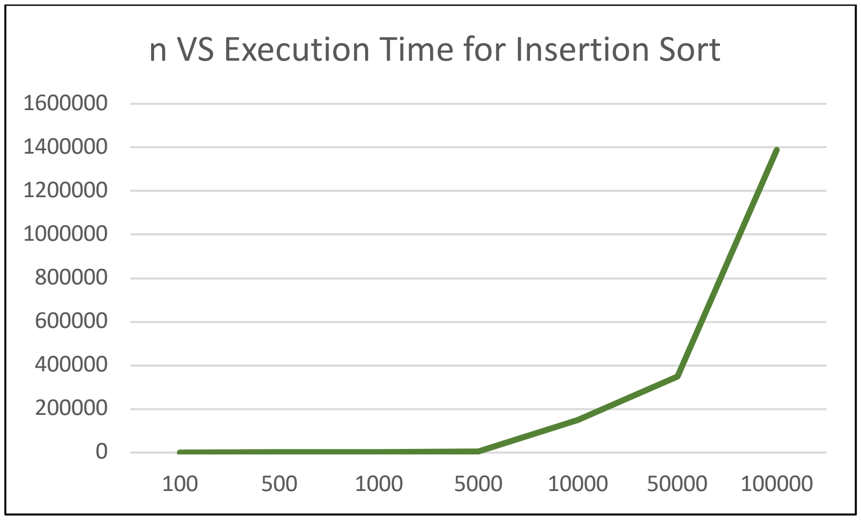


Figure 17: n Vs Execution Time for Insertion Sort

5.2. Merge sort algorithm

According to the table below, we performed the following comparisons during the sorting process: using Equation (n log n). Based on the Merge sort programmed code, we calculated the execution time. In Figure 18 Input size is represented on the x-axis, whereas the y-axis represents the number of comparisons. Figure 19 the Input size is represented on the x-axis, whereas the y-axis represents the execution time.

Input Size (n)	Number of comparisons	Execution Time (μs)
100	664.4	43.541
500	4483	32.417
1000	9965.8	74.375
5000	61438.6	376.452
10000	132877.1	825.25
50000	780482	1758
100000	3521928	3110.459

Table 12: Merge Sort Algorithm

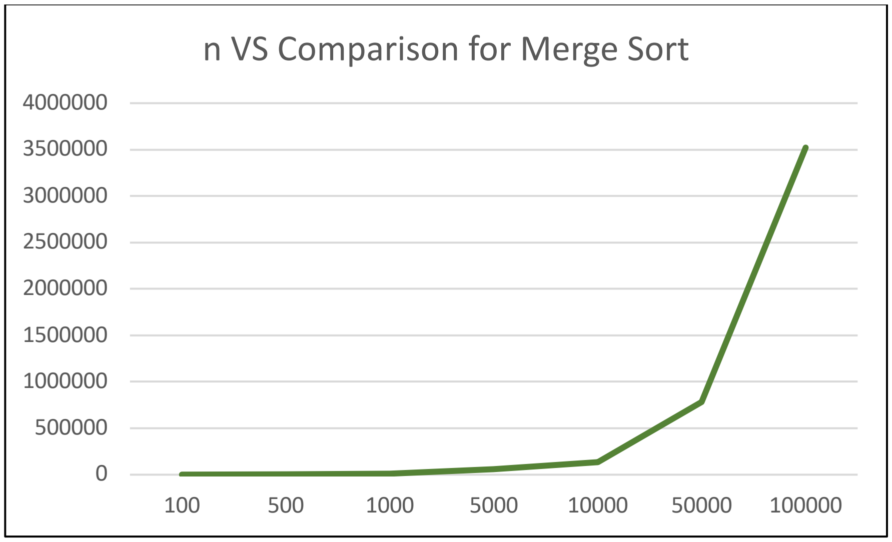


Figure 18: n Vs Comparison for Merge Sort

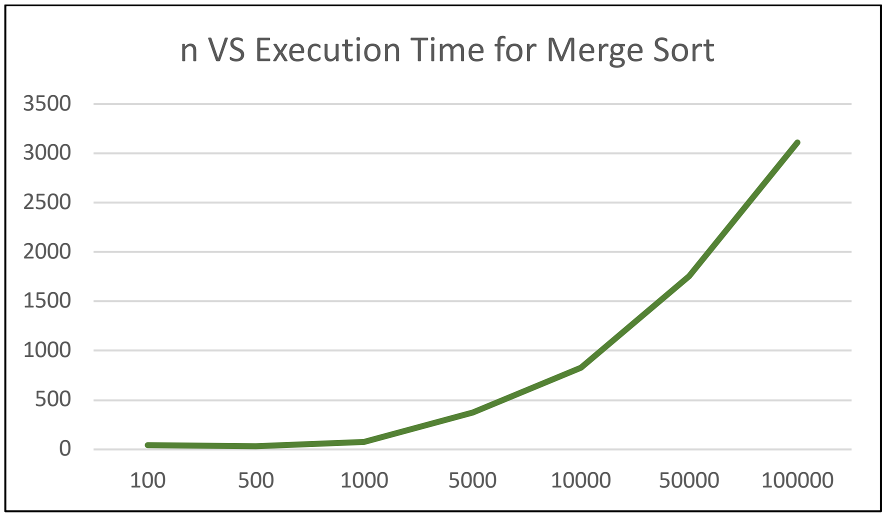


Figure 19: n Vs Execution Time for Merge Sort

5.3. Heap sort algorithm

According to the table below, we performed the following comparisons during the sorting process: using Equation (n log n). Based on the heap sort programmed code, we calculated the execution time. In Figure 20 Input size is represented on the x-axis, whereas the y-axis represents the number of comparisons. Figure 21 the Input size is represented on the x-axis, whereas the y-axis represents the execution time.

Input Size (n)	**Number of comparisons**	**Execution Time (μs)**
100	664.4	41.042
500	4483	48.291
1000	9965.8	182.125
5000	61438.6	579.834
10000	132877.1	1462.583
50000	780482	4931.459
100000	3521928	5238.75

Table 13: Heap Sort Algorithm

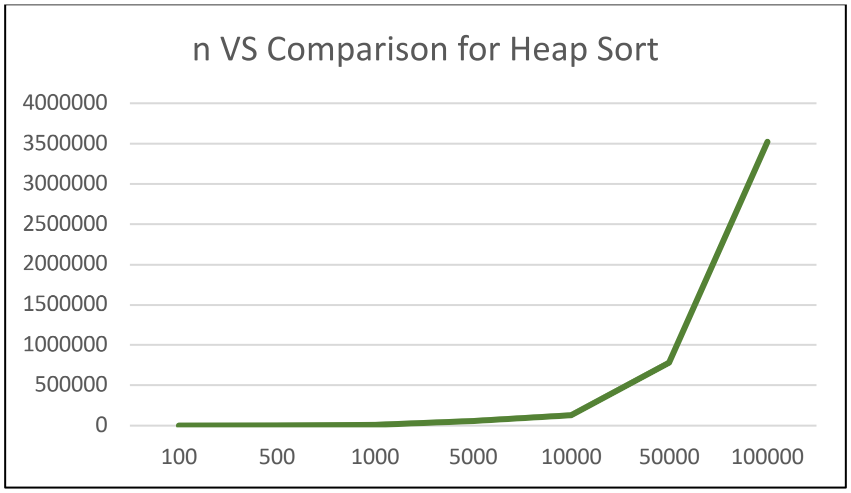


Figure 20: n Vs Comparison for Heap Sort

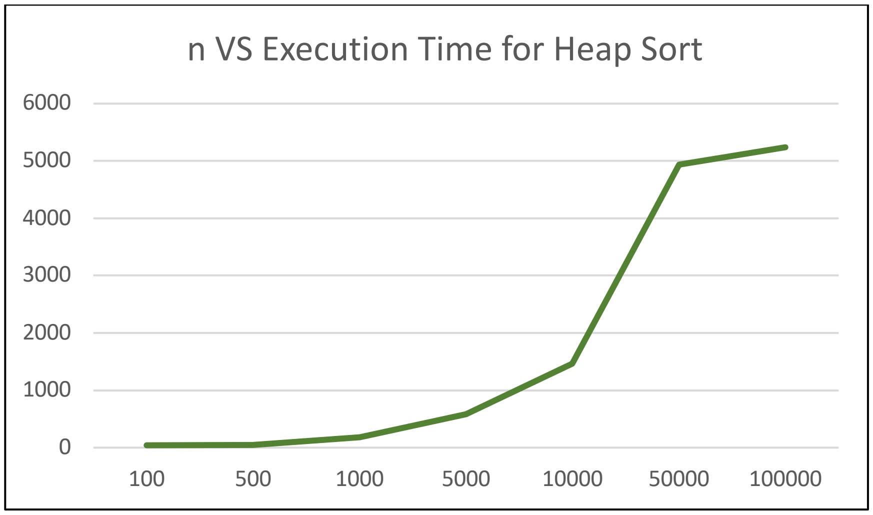


Figure 21: n Vs Execution Time for Heap Sort

According to the line graphs, it is shown that when the number of comparisons increases, the running time increases in all sorting algorithms. However, it is not easy to conclude the reliability of relying on the number of comparisons as a good predictor by only looking at the line graphs. Therefore, to support our conclusion, we have computed the correlation between the number of comparisons and time execution in each line graph. According to the results listed below, we can conclude that the number of comparisons is strongly correlated with the execution time in all three sorting algorithms. Therefore, the number of comparisons is a good predictor, especially in insertion and merge sort.

Insertion Sort: 0.9952 Strong positive correlation.
Merge Sort: 0.9349 Strong positive correlation.
Heap Sort: 0.8045 Strong positive correlation.

6. Design and analysis an improved Divide-and-conquer algorithm to compute a^n, where n is a natural number

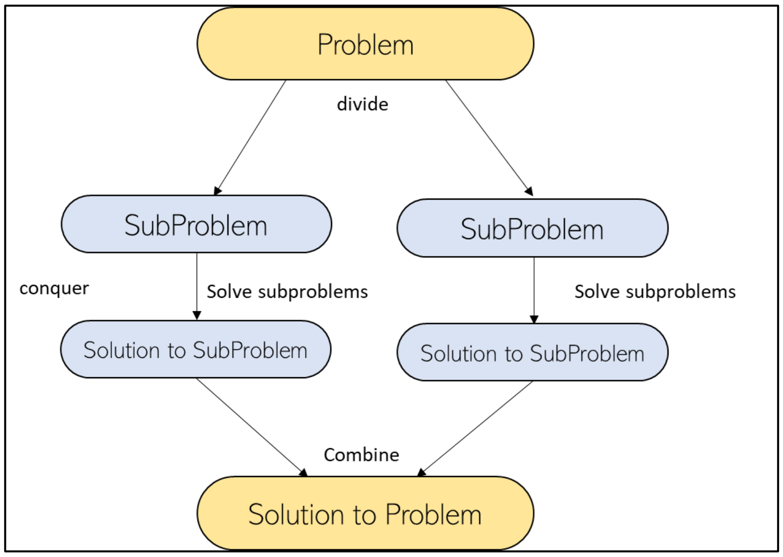


Figure 22: Divide-conquer techniques

6.1. Solve the problem in a brute-force manner

Brute force algorithm: A simple way to calculate power would multiply (a) exactly (n) times.

$a^n = (a * a * a\ldots * a)$

Alg: **FindPowOfA (a, n)**

Input: nonzero number (a) and nonnegative number (n)

```
Output: a^n                        cost        time
    1. Result=1                    c1          1
    2. For I → 1 to n              c2          n+1
    3. Do
Result → Result *a                 c3          n
    4. End for
    5. Return Result               c4          n
```

Time complexity: T(n) = c1(1)+c2(n+1) +c3(n)+c4(n) = O(n).

6.2. Efficient algorithm and time complexity

To determine the power of n, where n is a natural number, use this recursion algorithm (also recognized as a divide and conquer algorithm). Each time a, which is the basis of the logarithmic function, turns into a*a.

Alg: **FindPowOfA (a, n)**

Input: nonzero number (a) and nonnegative number (n)

Output: a^n	cost	time
1. If n=0, return 1	c1	1
2. If a=0, return 0	c2	1
3. If n is even, return FindPowOfA (a*a, n\2)	c3	log n
4. Else, return a* FindPowOfA (a*a, n\2)	c4	log n

Time complexity: T(n) = c1(1)+c2(1) +c3(log n) +c4(log n) = O(log n).

Divide-conquer is more efficient than the brute force algorithm.

Java code:

```
import java.util.Scanner;

public class DivideAndConquer
{
   static int FindPowOfA(int a, int n)
   {
      if(n = = 0){
         return 1;
      }
      // for avoid unnecessary recursive calls
      if(a = = 0){
         return 0;
      }

      if(n%2 = = 0){
         return FindPowOfA(a*a, n/2);
      } // if n is even

      else{
         return a * FindPowOfA(a*a, n/2);
      }

   }

         public static void main(String[] args) {

         Scanner scan = new Scanner (System.in);
         System.out.println("Enter number");
          int a = scan.nextInt();
         System.out.println("Enter natural number for power ");
         int n = scan.nextInt();

         System.out.println("The result is: "+ FindPowOfA(a,n));

   return;

         }
}
```

7. References

[1] *Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009).Introduction to algorithms. MIT press.*

[2] *Insertion sort. GeeksforGeeks. (2021, July 8). Retrieved November 27, 2021, from https://www.geeksforgeeks.org/insertion-sort/.*

[3] *Heapsort. GeeksforGeeks. (2021, November 9). Retrieved November 27, 2021, from https://www.geeksforgeeks.org/heap-sort/.*

[4] *InterviewBit. (n.d.). Heap sort algorithm. InterviewBit. Retrieved November 27, 2021, from https://www.interviewbit.com/tutorial/heap-sort-algorithm/.*

[5] *Wilkie, "Introduction to algorithms: Chapter Two, Merge Sort," Medium, 31- Jan-2017. [Online]. Available: https://medium.com/craft- academy/introduction-to-algorithms-chapter-two-merge-sort-edc7aba8d0d9. [Accessed: 26-Nov-2021].*

[6] *Wilkie, "Introduction to algorithms: Chapter Two, Merge Sort," Medium, 31- Jan-2017. [Online]. Available: https://medium.com/craft- academy/introduction-to-algorithms-chapter-two-merge-sort-edc7aba8d0d9. [Accessed: 26-Nov-2021].*