FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

ALGORITHM ANALYSIS &

DESIGN PROJECT #1

REPORT

FURKAN SELİM SALİHOĞLU

1921221007

CONTENTS

1. Mathematical Analysis of Each Algorithm and Experimental Results

1.1. Selection Sort

Selection sort is a sorting algorithm that sorts the elements of an array in ascending order. It works by iterating over the elements with the smallest element. This process continues until the end of the array and the elements are sorted in ascending order.

1.1.1. Pseudo Code

The selection sort algorithm is an array by repeatedly finding the minimum element from the unsorted part and putting it at the beginning. Selection sort pseudo code:

1. n ← length[A]
2. for i ← 0 to n-1 do:
3. smallest ← i
4. for j ← i+1 to n do:
5. if A[j] ← A[smallest] then do:
6. smallest ← j
7. temp ← A[smallest]
8. A[smallest] ← A[i]
9. A[i] ← temp

How Selection Sort Works:

1. Start from the first element of the array
2. Mark the current element as the smallest
3. Iterate over the next elements and find the smallest
4. Swap the current element with the smallest
5. Go to the next element and repeat step 2
6. Continue until the end of the array and the sorting is complete

This pseudo code explains how the selection sort algorithm works. This algorithm iterates over each element of the array to find the smallest element and swaps the current element with it. This process continues until the end of the array and elements are sorted in ascending order. Selection sort is an effective sorting algorithm for small arrays, but it could be slow for large arrays.

1.1.2. Complexity Analysis

If we are given n elements, then in the first pass, it will do n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3, and so on. Thus, the total number of comparisons can be found by;

$$(n-1)+(n-2)+\cdots+1$$

$$Sum = \frac{n\,(n-1)}{2} = n^2$$

Calculating with Master Theorem:

$$C(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} = \sum_{i=0}(n-i-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Therefore, the selection sort algorithm encompasses a time complexity of $O(n^2)$ and a space complexity of O(1) because it necessitates some extra memory space for the temp variable for swapping.

Time Complexities:

- **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of $O(n^2)$ for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is $O(n^2)$, in which the existing elements are in jumbled order, i.e, neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also $O(n^2)$, which occurs

when we sort the descending order of an array into the ascending order.

In the selection sort algorithm, the time complexity is $O(n^2)$, in all three cases. This is because, in each step, we are required to find minimum elements so that it can be placed in the correct position. Once we trace the complete array, we will get out the minimum element.

1.1.3. Experimental Design

When designing an experimental selection sort, I first examined the output of three different array types by creating a random array, an array in the form of 1, 2, …, N, and an array in the form of N, N-1, …, 1. I determined the sizes of these arrays to be 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, and 1000000 and recorded the output from these arrays. Since computers are very fast, they can successfully sort the first array with a size of 5000 in less than 0 milliseconds. Afterward, the results quickly increase. Our graph based on the results obtained is shown in Figure 1, Figure 2, and Figure 3.

Figure 1: Selection Sort Algorithm random array results.

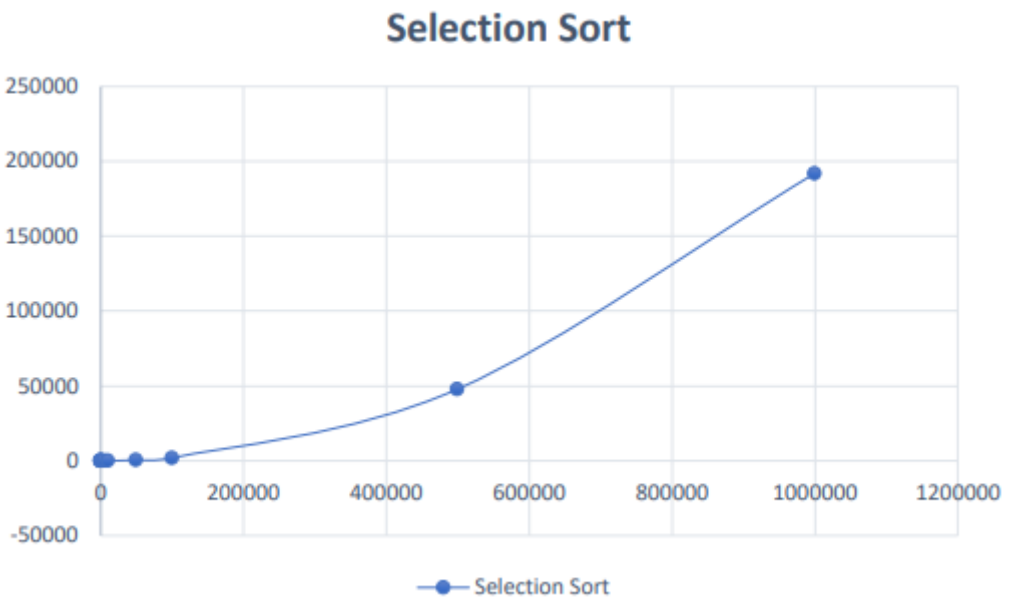

Figure 2: Selection Sort Algorithm sorted array results

Figure 3: Selection Sort Algorithm descending sorted array results.

1.2.Insertion Sort

1.2.1. Pseudo Code

1. n ← length[A]
2. for i ← 1 to n do:
3. key ← A[i]
4. j ← i – 1
5. while j >= 0 and A[j] > key do:
6. A[j + 1] ← A[j]
7. j ← j – 1
8. A[j + 1] ← key

How Insertion Sort Works

1. Start from the first element of the array
2. Insert the current element into the sorted portion
3. Go to the next element and repeat from step 2
4. Continue until the end of the array and the sorting is complete.

This pseudo code explains how the insertion sort algorithm works. This algorithm iterates over the elements of the array and inserts each element into its correct position in the sorted portion of the array. This process continues until the end of the array and the elements are sorted in ascending order. Insertion sort is an effective sorting algorithm for small arrays, but it could be slow for large arrays. It is also efficient for partially sorted arrays, as it can take advantage of the existing order to sort the array more quickly.

1.2.2. Complexity Analysis

If we are given n elements, then in the first pass, it will make n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3, and so on. Thus, the total number of comparisons can be found by;

$$(n-1)+(n-2)+\cdots+1$$

$$Sum = \frac{n\,(n-1)}{2} = n^2$$

Calculating with Master Theorem:

$$C_B(n) = \sum_{i=1}^{n-1}\sum_{j=0}^{i-1} = n-1 \in \Theta(n)$$

$$C_W(n) = \sum_{i=1}^{n-1}\sum_{j=0}^{i-1} 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Therefore, the insertion sort algorithm encompasses a time complexity of O(n2) and a space complexity of O(1) because it necessitates some extra memory space for a key variable to perform swaps.

Time Complexities:

- **Best Case Complexity:** The insertion sort algorithm has a best-case time complexity of O(n) for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still.
- **Average Case Complexity:** The average-case time complexity for the insertion sort algorithm is O(n2), which is incurred when the existing elements are in jumbled order, i.e., neither in ascending order nor in descending order.
- **Worst Case Complexity:** The worst-case time complexity is also O(n2), which occurs when we sort the ascending order of an array into descending order.

1.2.3. Experimental Design

The experimental design for insertion sort is the same as the one described in 1.1.3 for comparing the results correctly. The results are given in Figure 4, Figure 5, and Figure 6.

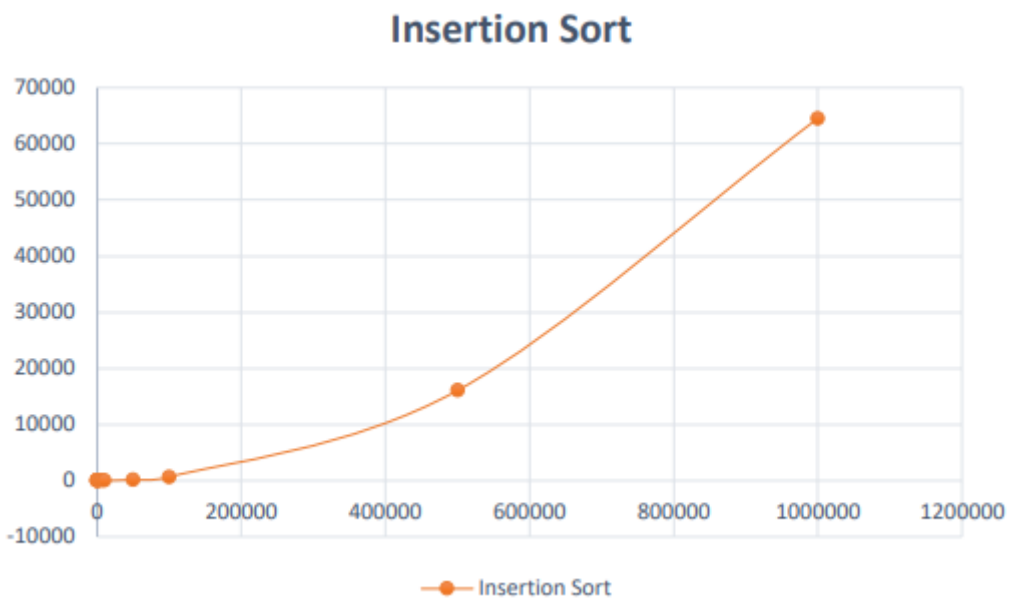

Figure 4: Insertion Sort Algorithm random sorted array results

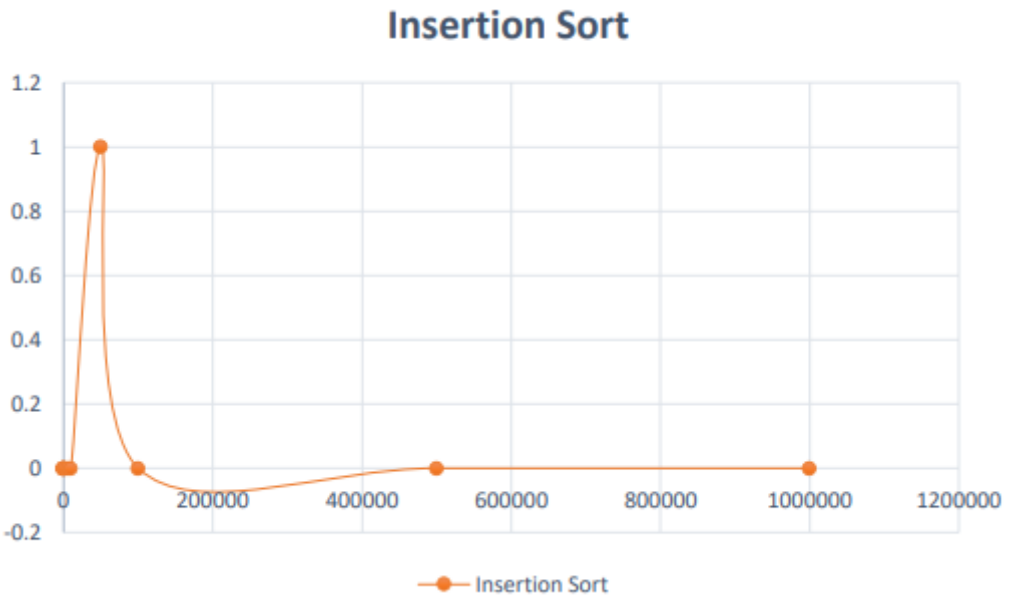

Figure 5: Insertion Sort Algorithm sorted array results.

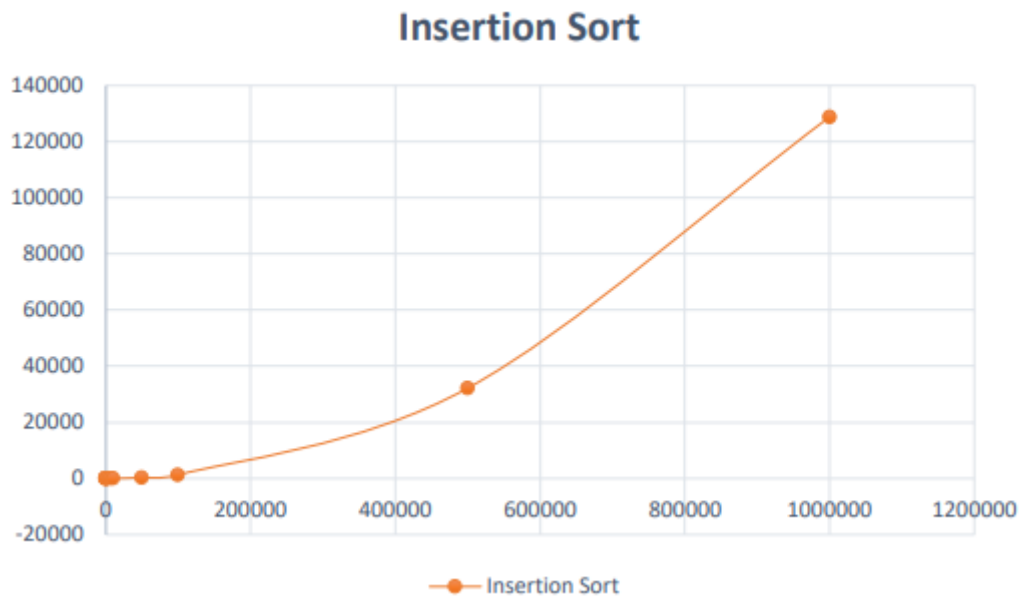

Figure 6: Insertion Sort Algorithm descending sorted array results.

1.3. Shell Sort

1.3.1. Pseudo Code

1.function shellSort(arr):

2.n = length(arr)

3.gap = n / 2

4.while gap > 0:

5.for i from gap to n - 1:

6.temp = arr[i]

7.j = i

8.while j >= gap and arr[j - gap] > temp:

9.arr[j] = arr[j - gap]

10.j = j - gap

11.arr[j] = temp

12.gap = gap / 2

How Shell Sort work?

Shell Sort works by sorting sublists of elements separated by a specific gap (increment). The basic idea is to start with a large gap and gradually reduce it until the gap becomes 1. At that point, the algorithm performs a final pass using the standard insertion sort on nearly sorted elements.

1.3.2. Complexity Analysis

Shellsort, also known as Shell sort or Shell's method, is an in-place comparison sort algorithm that generalizes the insertion sort algorithm. It was proposed by Donald Shell in 1959. Shellsort improves on the time complexity of the insertion sort by allowing the exchange of elements that are far apart. The basic idea behind Shellsort is to first sort sublists of elements that are h apart, where h is called the "gap" or "increment." The algorithm starts with a large gap and reduces it in each iteration until the gap becomes 1. The final pass with a gap of 1 is equivalent to the standard insertion sort, but by that time, the array has been partially sorted, making the overall sorting process more efficient. The computational complexity of Shellsort depends on the choice of the gap sequence. There are different gap sequences that can be used, and the performance of the algorithm is influenced by the selected sequence. Some popular gap sequences include the original sequence proposed by Donald Shell (n/2, n/4, ..., 1), the Pratt sequence, and the Sedgewick sequence, among others. Let's denote n as the number of elements in the array.

$$O(\sum_{k=1}^{t/2} Nh_k + \sum_{k=t/2+1}^{t} N^2/h_k) = O(N\sum_{k=1}^{t/2} h_k + N^2 \sum_{k=t/2+1}^{t} 1/h_k)$$

$$= O(Nh_{t/2}) + O(\frac{N^2}{h_{t/2}}) = O(N^{3/2}) \qquad Since\, h_{t/2} = \Theta(\sqrt{N})$$

Time Complexities:

- **Best Case Complexity:** The worst-case time complexity of Shellsort depends on the gap sequence used. In the worst case, when using the original gap sequence (n/2, n/4, ..., 1), the time complexity is often expressed as O(n^2). However, with certain gap sequences like the one proposed by Sedgewick, the worst-case time complexity can be reduced to O(n^(4/3)).
- **Average Case Complexity:** The average-case time complexity of Shellsort is not easy to analyze precisely, and it depends on the specific gap sequence. Some gap sequences yield better average-case performance than others. On average, Shellsort often performs better than the quadratic sorting algorithms like insertion sort but may not be as efficient as more advanced algorithms like quicksort or merge sort.
- **Worst Case Complexity:** The best-case time complexity of Shellsort also depends on the gap sequence. With certain gap sequences, Shellsort can achieve a best-case time

complexity of O(n log^2 n).

1.3.3. Experimental Design

The experimental design for insertion sort is the same as the one described in 1.1.3 for comparing the results correctly. The results are given in Figure 7, Figure 8, and Figure 9.

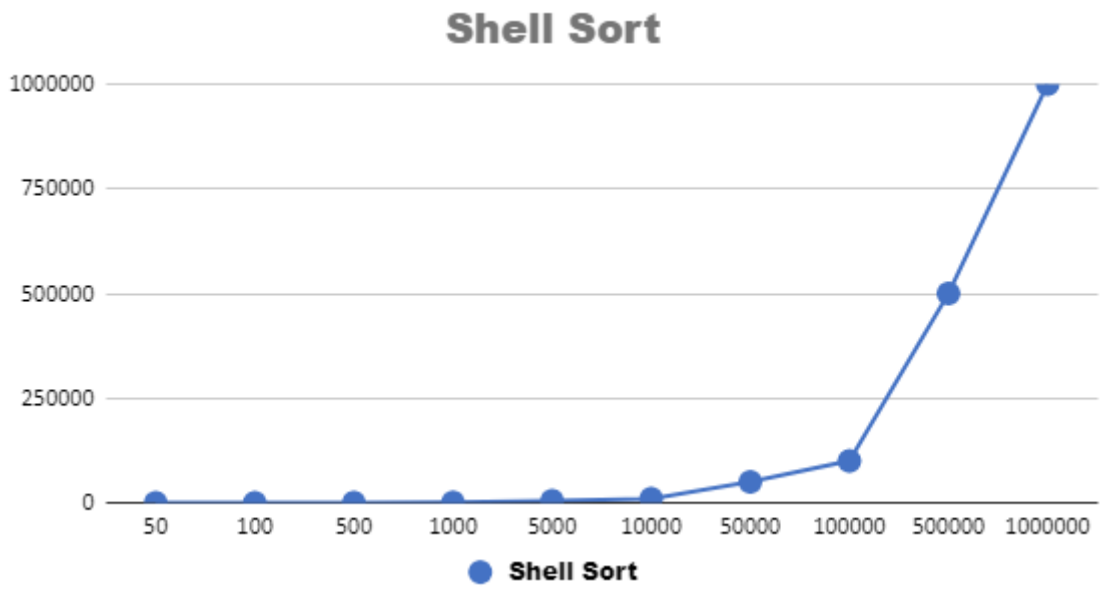

Figure 7: Shell Sort Algorithm random sorted array results.

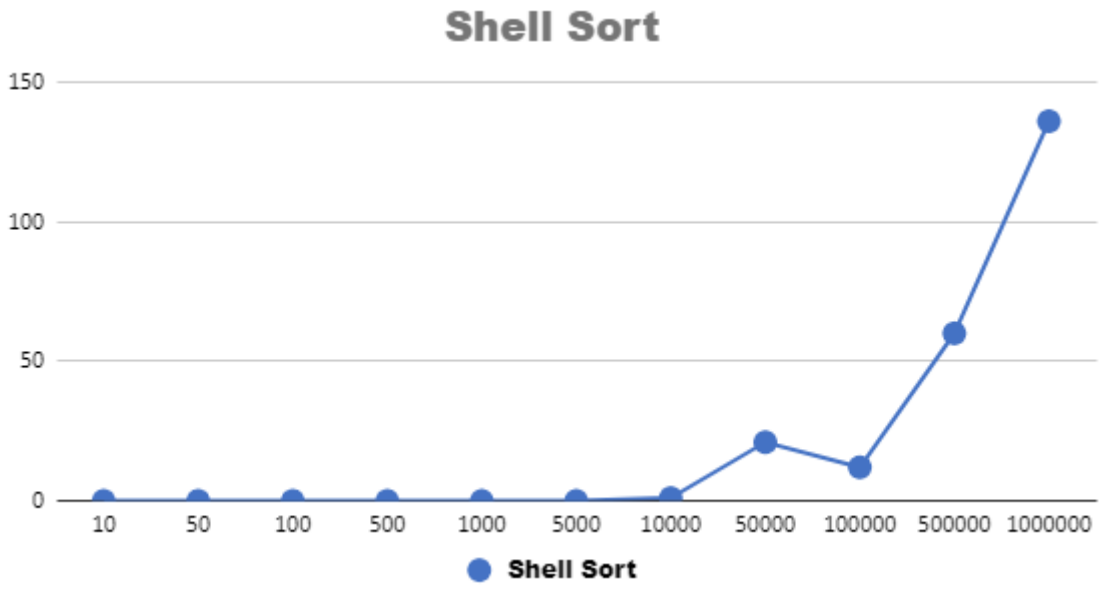

Figure 8: Shell Sort Algorithm sorted array results.

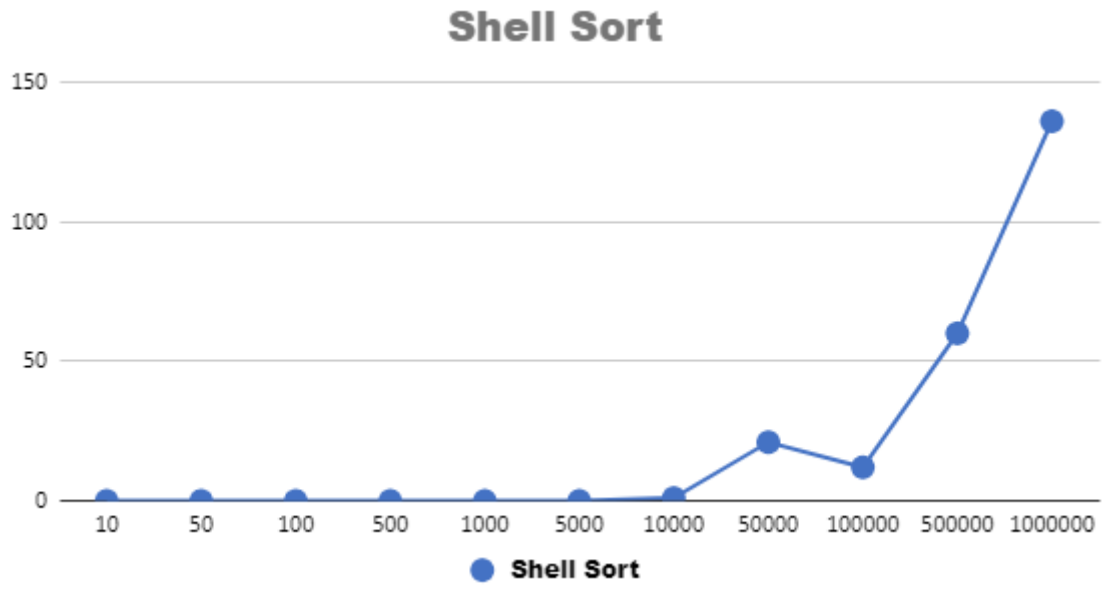

Figure 9: Shell Sort Algorithm descending sorted array results.

1.4.Merge Sort

1.4.1. Pseudo Code

- **Pseudo Code of sort (A, l, r)**

1. if l < r then do:

2. m ← l + (r - l) / 2

3. sort (A, l, m)

4. sort (A, m + 1, r)

5. merge (A, l, m, r)

- **Pseudo Code of merge (A, l, m, r)**

1. n1 = m – l + 1

2. n2 = r – m

3. L[] ← length[n1]

4. R[] ← length[n2]

5. for i ← 0 to n1 do:

6. L[i] ← A[l + i]

7. for j ← 0 to n2 do: 10

8. R[j] ← A[m + 1 + j]

9. i ← 0

10. j ← 0

11. k ← l

12. while i < n1 and j < n2 do:

13. if L[i] ≤ R[j] then do:

14. A[k] ← L[j]

15. i ← i + 1

16. else do:

17. A[k] ← R[j]

18. j ← j + 1

19. k ← k + 1

20. while i < n1 do:

21. A[k] ← L[i]

22. i ← i + 1

23. k ← k + 1

24. while j < n2 do:

25. [k] ← R[j]

26. j ← j + 1

27. k ← k + 1

How Merge Sort Works:

1. Split the array into two halves
2. Split the left and right halves into two again
3. Repeat the splitting until the size of the array is 1
4. Merge and sort the array in order

The merge sort algorithm iteratively divides an array into equals until we achieve an atomic value. In case if there are an odd number of elements in an array, then one of the halves will have more elements than the other half. After dividing an array into two subarrays, we will notice that it did not hamper the order of elements as they were in the original array. After now, we will further divide these two arrays into other halves. Again, we will divide these arrays until we achieve an atomic value, i.e., a value that cannot be further divided. Next, we will merge them back in the same way as they were broken down. For each list, we will first compare the elements and then combine them to form a new sorted list. In the next iteration, we will compare the lists of two data values and merge them back into a list of found data values, all placed in a sorted manner.

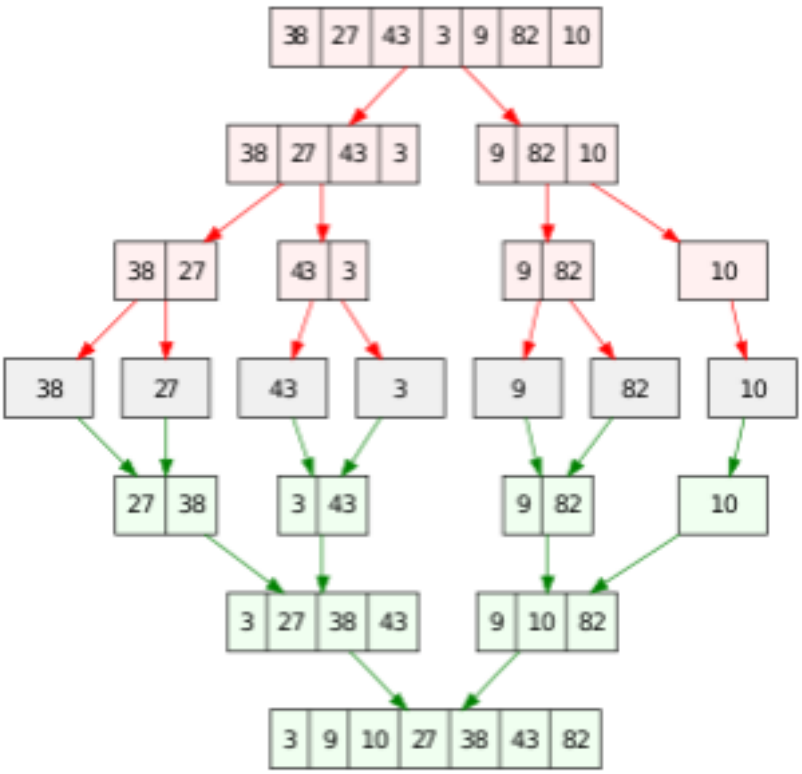

Figure 10: An example of how the merge sort algorithm works

1.4.2. Complexity Analysis

Let T(n) be the total time taken by the Merge Sort Algorithm.

- Sorting two halves will take at the most t 2*T*n/2 time.
- When we merge the sorted list, we come up with a total ‘n-1’ comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be;

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

But we ignore ‘-1’ because the element will take some time to be copied in merge lists.

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$$

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$

$$= 2^2T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n$$

$$T(n) = 2^2T\left(\frac{n}{2^2}\right) + 2n$$

$$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$$

$$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$$

$$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + n + 2n$$

$$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in$$

$$\frac{n}{2^i} = 1 \; and \; T\left(\frac{n}{2^i}\right) = 0$$

$$n = 2^i$$

$$logn = log_2 i$$

$$logn = ilog2$$

$$\frac{logn}{log2} = i$$

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in$$

$$= 2^i \, x \, 0 + log_2 n.n$$

$$T(n) = nlogn$$

Calculating with Master Theorem:

$$T(n) = 2T\left(\frac{n}{2}\right) + f(n)$$

$$f(n) \in \Theta(n)$$

$$d = 1, a = 2, b = 2$$

$$a = b^d$$

$$T(n) \in \Theta(nlogn)$$

Time Complexities

- **Best Case Complexity:** The merge sort algorithm has a best-case time complexity of O(nlogn) for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the merge sort algorithm is O(nlogn), which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also O(nlogn), which occurs when we sort the descending order of an array into the ascending order.

1.4.3. Experimental Design

The experimental design for merge sort is the same as the one described in 1.1.3 for comparing the results correctly. The results are given in Figure 11, Figure 12, and Figure 13.

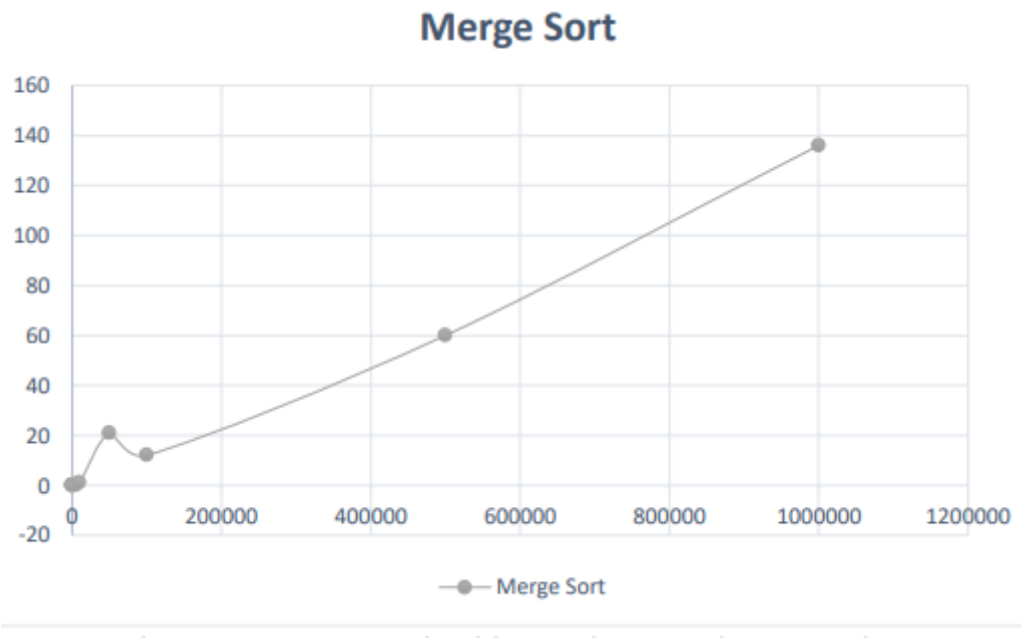

Figure 11: Merge Sort Algorithm random sorted array results.

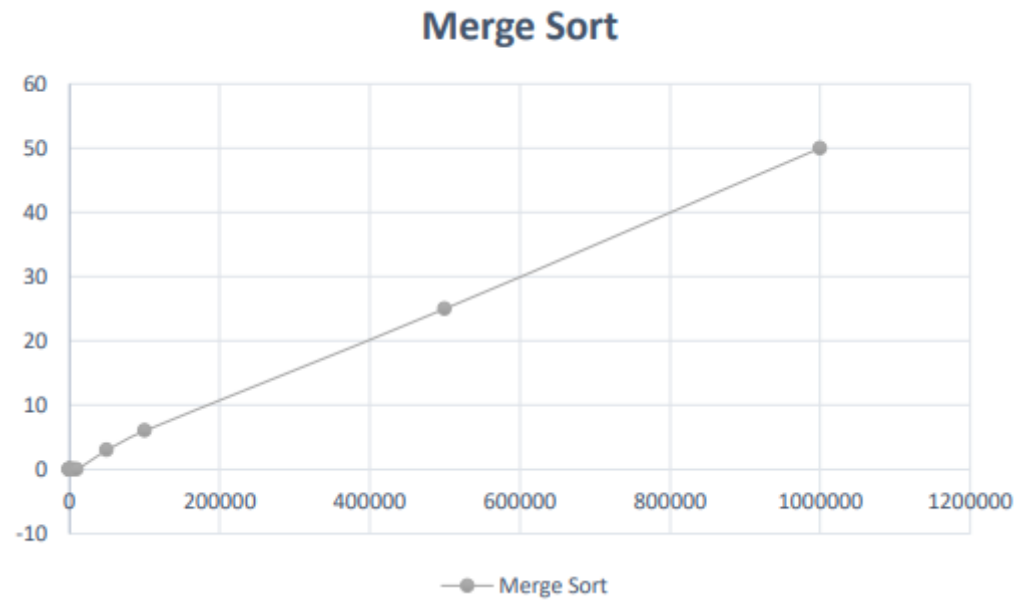

Figure 12: Merge Sort Algorithm sorted array results.

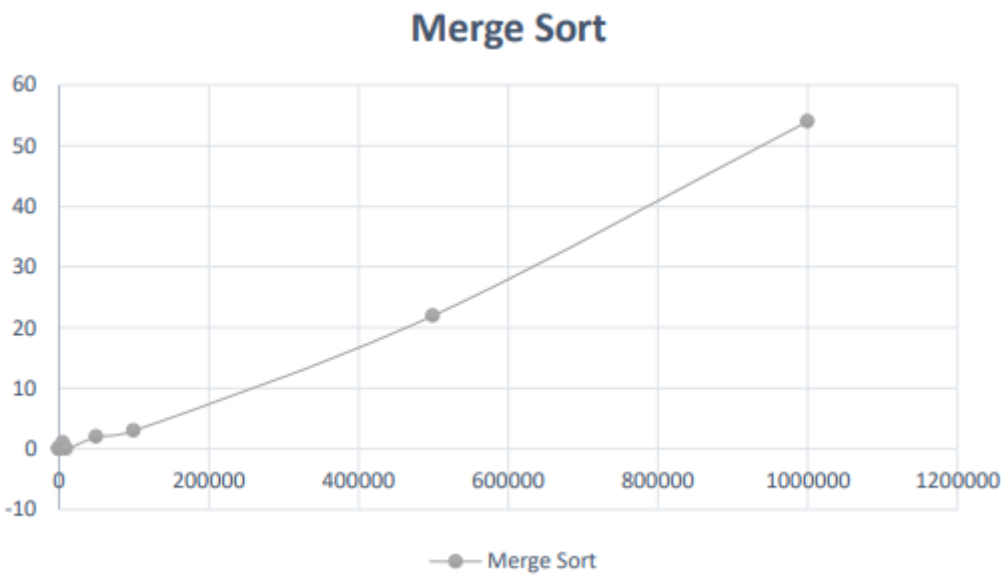

Figure 13: Merge Sort Algorithm descending sorted array results.

1.5. 3-way Merge Sort

1.5.1. Pseudo Code

- proceduremergeSort3(arr):

1. if length(arr)>1:
2. mid1=length(arr)//3
3. mid2=2*(length(arr)//3)+1
4. left=arr[0:mid1]
5. middle=arr[mid1:mid2]
6. right=arr[mid2:]
7. mergeSort3(left)
8. mergeSort3(middle)
9. mergeSort3(right)
10. merge(arr,left,middle,right)

- procedure merge(arr,left,middle,right):

1. i=j=k=0
2. while i<length(left)andj<length(middle)andk<length(right):
3. if left[i]<=middle[j]andleft[i]<=right[k]:
4. arr[i+j+k]=left[i]
5. i+=1
6. elif middle[j]<=left[i]andmiddle[j]<=right[k]:
7. arr[i+j+k]=middle[j]
8. j+=1
9. else:
10. arr[i+j+k]=right[k]
11. k+=1
12. while i<length(left):
13.arr[i+j+k]=left[i]
14. i+=1
15. while j<length(middle):
16. arr[i+j+k]=middle[j]
17. j+=1
18. while k<length(right):
19. arr[i+j+k]=right[k]
20. k+=1

1.5.2. Complexity Analysis

The time complexity of the 3-way merge sort algorithm can be analyzed based on the number of comparisons and the number of elements in the input array.

Let's denote n as the number of elements in the array.

1. **Divide (Splitting):** The array is divided into three parts in each recursive call. The splitting step takes constant time, and the number of recursive calls until the base case is reached is log3(n). Therefore, the overall time complexity for the splitting step is O(log3(n)).
2. **Merge (Merging):** In each merge step, all n elements of the array are considered. The merging process involves comparing and rearranging elements. The worst-case time complexity for the merging step is O(n) since each element needs to be compared and

placed in the correct position.

3. **Overall Time Complexity:** Combining the splitting and merging steps, the overall time complexity of the 3-way merge sort is O(nlog3(n)).

The space complexity of the algorithm is also important:

1. **Recursive Call Stack:** The depth of the recursive call stack is log3(n), so the space complexity due to recursive calls is O(log3(n)).
2. **Auxiliary Space**: The additional space required for creating temporary arrays during merging is O(n).
3. **Overall Space Complexity:** Considering both the recursive call stack and auxiliary space, the overall space complexity is O(n+log3(n)). In summary, the 3-way merge sort algorithm has a time complexity of O(n.log3(n)) and a space complexity of O(n+log3(n)).

1.6. Lomuto Quick Sort

1.6.1. Pseudo Code

- Pseudo Code of Swap(A[], p1, p2)

1. temp ← A[p1]

2. A[p1] ← A[p2]

3. A[p2] ← temp

- Pseudo Code of partition (A[], low, high)

1. pivot ← A[high]

2. i ← low – 1

3. for j ← low to high-1 do:

4. if A[j] <= pivot then do:

5. i += 1

6. Swap(A[], i, j)

7. Swap(A[], i+1, high)

8. return i+1

- Pseudo Code of sort(A[], low, high)

1. if low < high then do:

2. pi ← partition(A[], low, high)

3. sort(A[], low, pi-1)

4. low = pi + 1

5. else then do:

6. sort (A[], pi+1, high)

7. high ← pi

How Lomuto Quicksort Works

This algorithm works by assuming the pivot element as the last element. If any other element is given as a pivot element then swap it first with the last element. Now initialize two variables i as low and j also low, iterate over the array and increment i when A[j] <= pivot and swap A[i] with A[j] otherwise increment only j. After coming out from the loop swap A[i] with A[high]. This i stores the pivot element.

1.6.2. Complexity Analysis

- **Best Case Complexity:** Calculating with Master Theorem:

$$C_B(n) = 2C_B\left(\frac{n}{2}\right) + n$$

a = 2, b = 2, d = 1

Θ(nlogn)

- **Worst Case Complexity:** Calculating with Master Theorem:

$$C_w(n) = C_w(n-1) + n$$

$$C_w(n) = (n+1) + n + n - 1 + \cdots + 3$$

$$C_w \in \Theta(n^2)$$

1.6.3. Experimental Design

The experimental design for Lomuto quicksort is the same as the one described in 1.1.3 for comparing the results correctly. The results are given in Figure 14, Figure 15, and Figure 16.

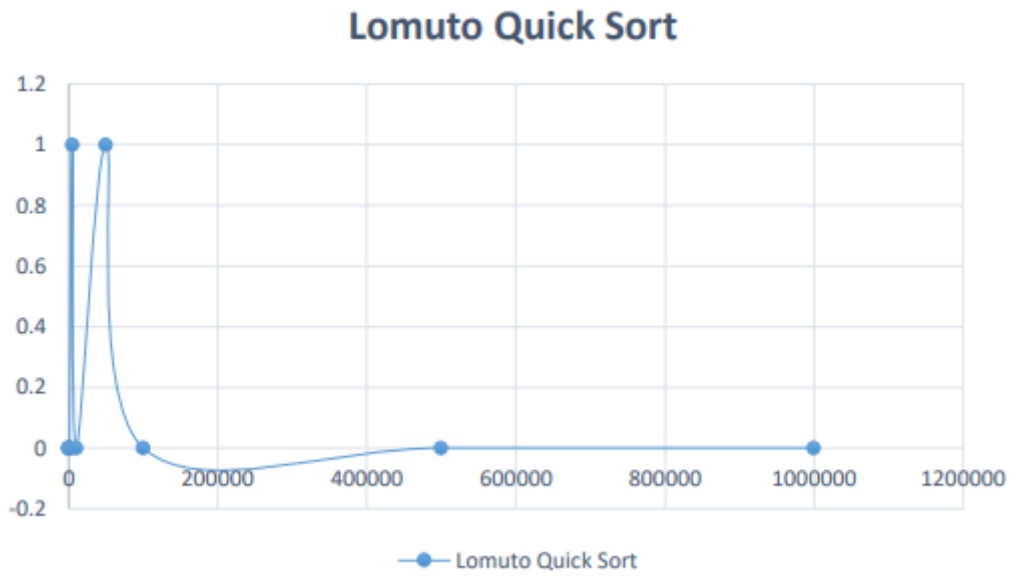

Figure 14: Lomuto Quicksort Algorithm random sorted array results.

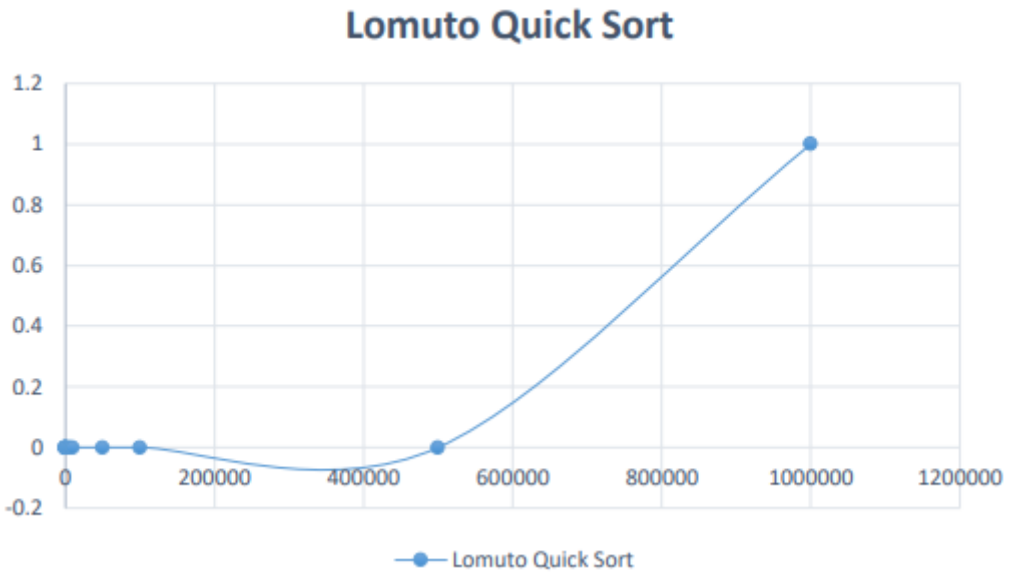

Figure 15: Lomuto Quick Sort Algorithm sorted array results.

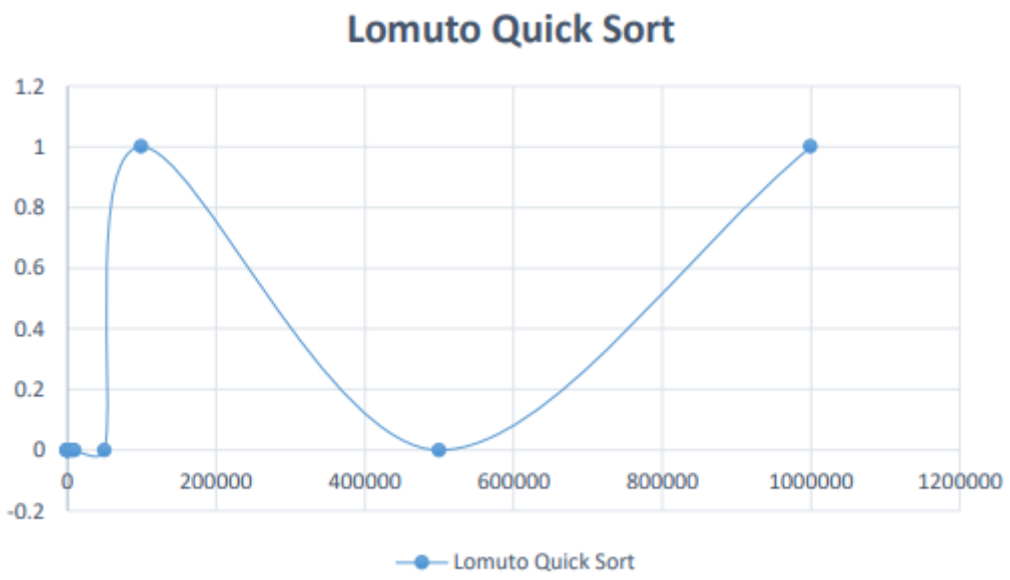

Figure 16: Lomuto Quick Sort Algorithm descending sorted array results.

1.7. Hoare Quick Sort

1.7.1. Pseudo Code

- Psuedo Code of partition (A[], low, high)

1. pivot A[low]
2. i low - 1
3. j high + 1
4. while true then do:
5. do:
6. i += 1
7. while(A[i] < pivot)
8. do:
9. j -= 1
10. while(A[j] > pivot)
11. if i >= j then do:
12. return j
13. temp A[i]
14. A[i] A[j]
15. A[j] temp

- Pseudo Code of sort(A[], low, high)

1. if low < high then do:

2. pi partition(A[], low, high)

3. if (pi - low) <= (high – (pi + 1)) then do:

4. sort(A[], low, pi)

5. low pi + 1

6. else then do:

7. sort(A[], pi + 1, high)

8. high pi

How Hoare Quicksort Works

Hoare's Partition Scheme works by initializing two indexes that start at two ends, the two indexes move toward each other until an inversion is (A smaller value on the left side and a greater value on the right side) found. When an inversion is found, two values are swapped and the process is repeated.

1.7.2. Complexity Analysis

- **Best Case Complexity:** Calculating with Master Theorem:

$$C_B(n) = 2C_B\left(\frac{n}{2}\right) + n$$

$$a = 2, b = 2, d = 1$$

$$\Theta(nlogn)$$

- **Worst Case Complexity:** Calculating with Master Theorem:

$$C_w(n) = C_w(n-1) + n$$

$$C_w(n) = (n+1) + n + n - 1 + \cdots + 3$$

$$C_w \in \Theta(n^2)$$

1.7.3. Experimental Design

The experimental design for merge sort is the same as the one described in 1.1.3 for comparing the results correctly. The results are given in Figure 17, Figure 18, and Figure 19.

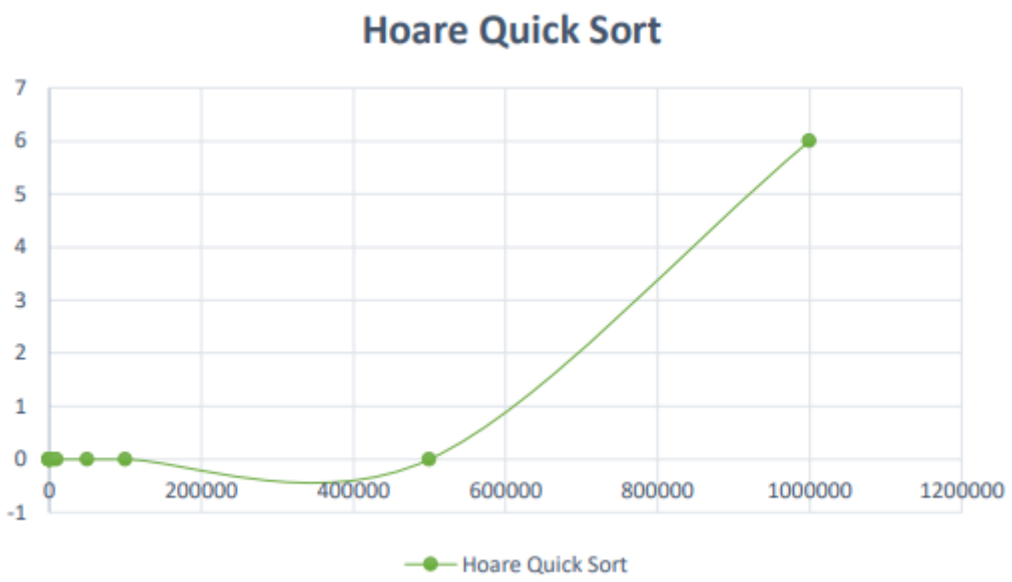

Figure 17: Hoare Quick Sort Algorithm random sorted array results

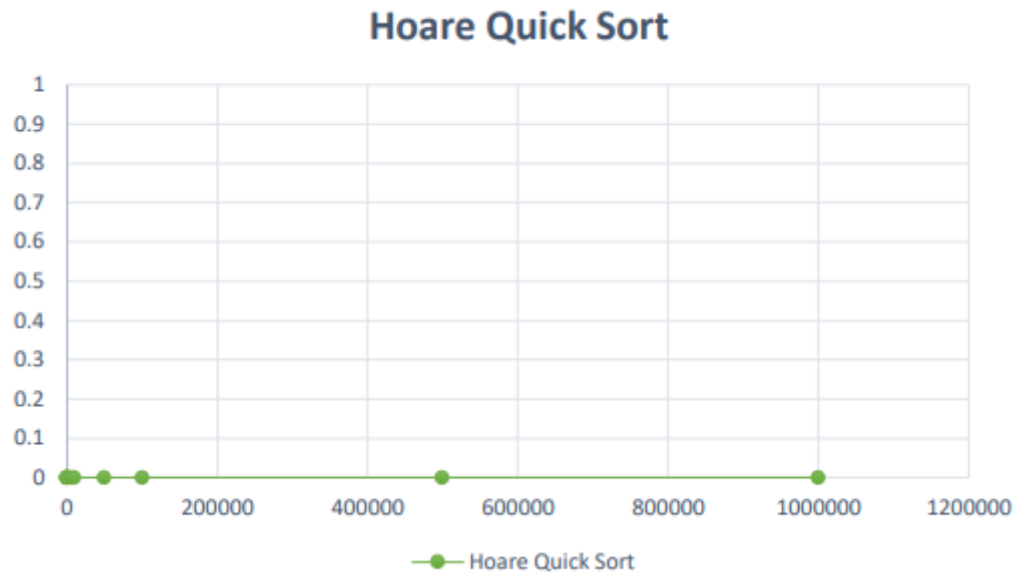

Figure 18: Hoare Quick Sort Algorithm sorted array results

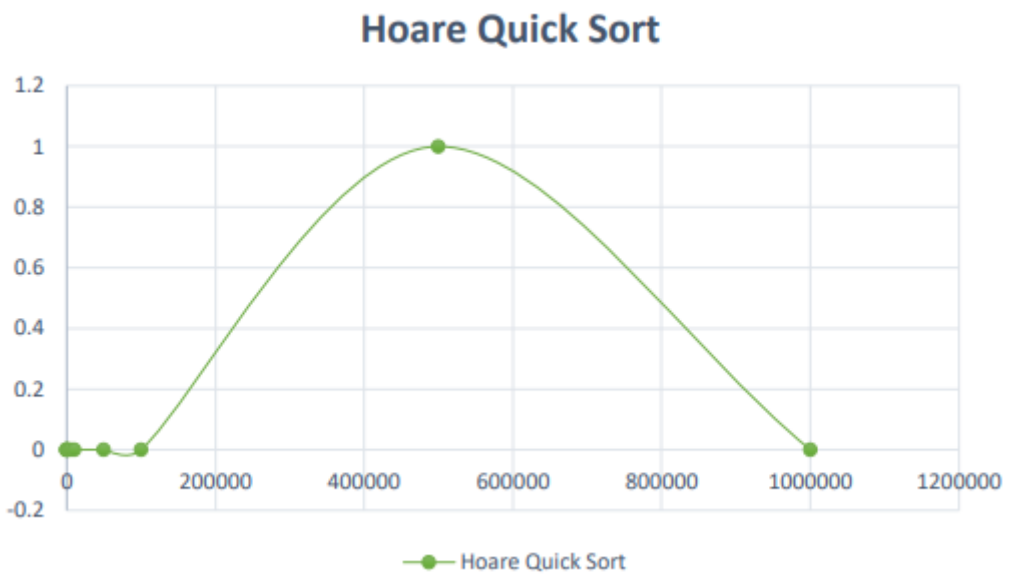

Figure 19: Hoare Quick Sort Algorithm descending sorted array results.

1.8. Heap Sort

1.8.1. Pseudo Code

- Pseudo Code of sort (A[])

1. n length[A]
2. for i n/2 -1 to 0 do:
3. heapify(A[], n, i)
4. for i n -1 to 0 do:
5. temp A[0]
6. A[0] A[i]
7. A[i] temp
8. heapify(A, i, 0)

- Pseudo Code of heapify(A, n, i)

1. largest i
2. l 2 * i + 1
3. r 2 * i + 2
4. if l < n and A[r] > A[largest] then do:
5. largest l
6. if r < n and A[r] > A[largest] then do:
7. largest r

8. if largest != i then do:

9. swap A[i]

10. A[i] A[largest]

11. A[largest] swap

12. heapify(A[], n, largest)

How Heap Sort Works

1. First convert the given array into a heap structure.
2. Take the largest element (the root node) of the array and move it to the end of the array.
3. Delete the last element of the array and rebuild the heap structure.
4. Repeat steps 2 and 3 for the length of the array.

The Heap Sort algorithm inserts all elements into a heap then swaps the first element with the last element and reduces the heap size by one because the last element is already at its final location in the sorted array. Then we heapify the first element because swapping has broken the max heap property. We continue this process until the heap size remains one. Hereafter swapping, it may not satisfy the heap property. So, we need to adjust the location of the element to maintain heap property. This process is called heapify. Here we recursively fix the children until all of them satisfy the heap property.

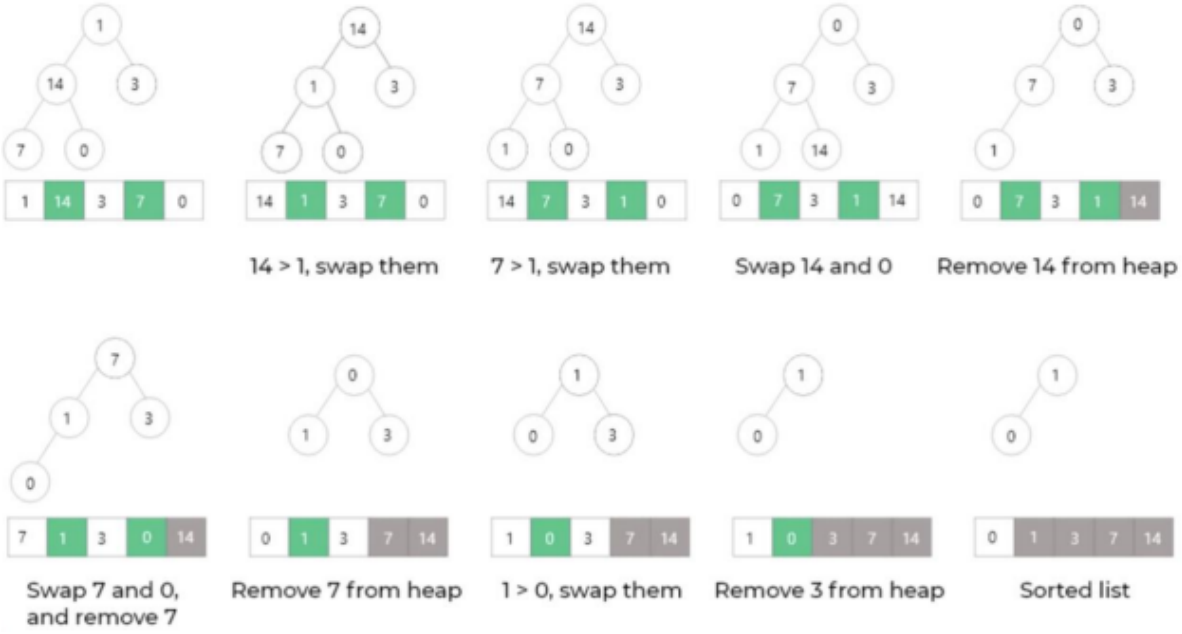

Figure. 20: An example of how the heap sort algorithm works

1.8.2. Complexity Analysis

To find the complexity of the heap sort algorithm, we need to examine the operations performed by the code and determine how many times each operation is repeated. The operations of the heap sort algorithm can be divided into two main parts: Build heap and heap sorting. Build heap for a given list of n keys:

$$C(n) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2\big(n - log_2(n+1)\big) \in \Theta(n)$$

Repeat operation of root removal n-1 times (fix heap):

$$C(n) = \sum_{i=1}^{n-1} 2log_2 i \in \Theta(nlogn)$$

Time Complexities

- **Best Case Complexity:** The heap sort algorithm has a best-case time complexity of O(nlogn).
- **Worst Case Complexity:** The heap sort algorithm has a worst-case time complexity of O(nlogn).
- **Average Case Complexity:** The heap sort algorithm has an average-case time complexity of O(nlogn).

1.8.3. Experimental Design

The experimental design for merge sort is the same as the one described in 1.1.3 for comparing the results correctly. The results are given in Figure 21, Figure 22, and Figure 23.

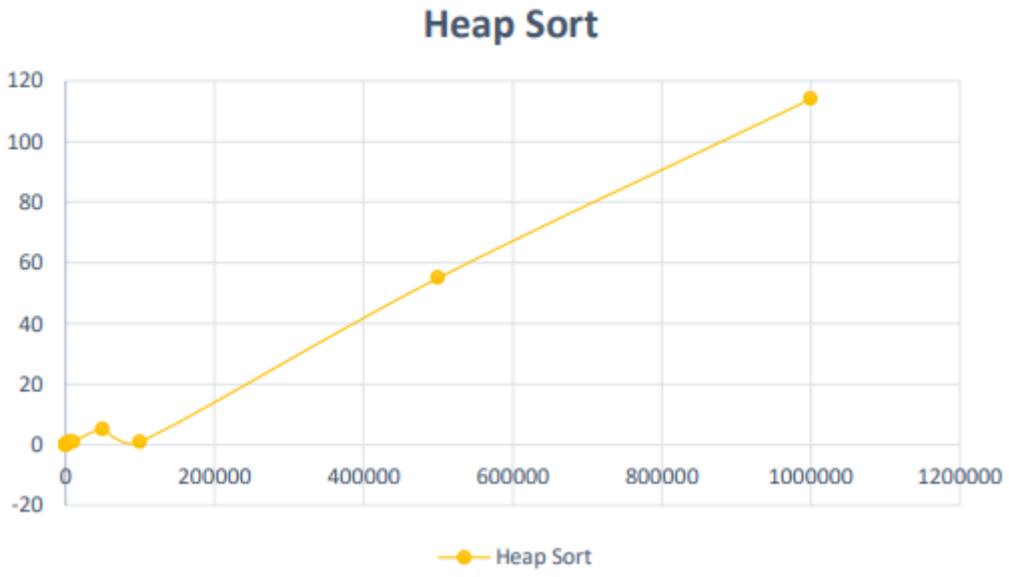

Figure 21: Heap Sort Algorithm random sorted array results.

Figure 22: Heap Sort Algorithm sorted array results.

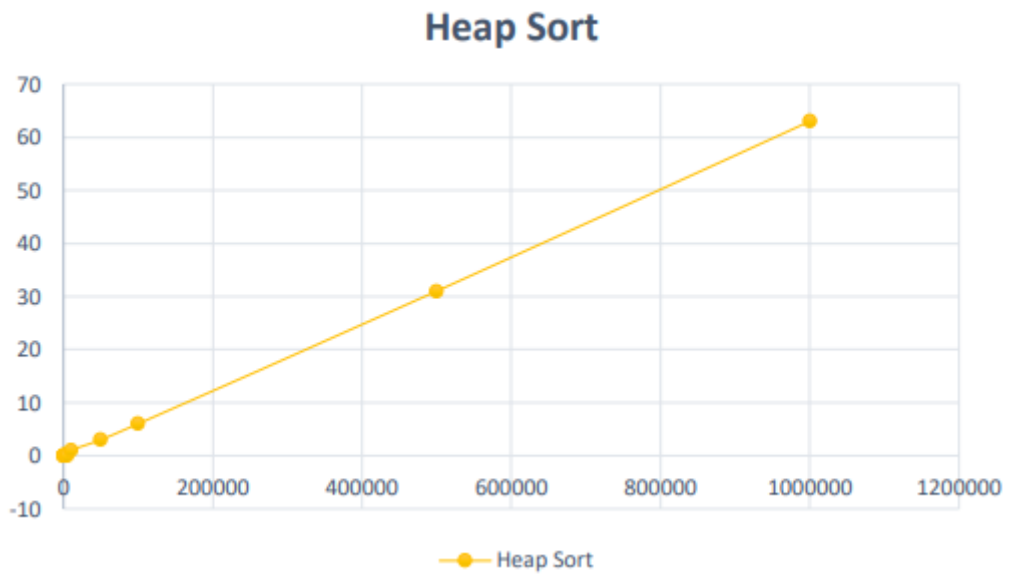

Figure 23: Heap Sort Algorithm descending sorted array results.

2.Results and Discussion

Selection Sort:

- Performance: Quadratic time complexity (O(n^2)).
- Observations: Inefficient for larger arrays but relatively better on partially or fully sorted arrays.

Insertion Sort:

- Performance: Quadratic time complexity (O(n^2)).
- Observations: Efficient for nearly sorted arrays, struggles with descending sorted arrays.

Shell Sort:

- Performance: Depends on gap sequence, generally improved over insertion sort.
- Observations: Demonstrates reasonable efficiency on random arrays.

Merge Sort:

- Performance: Consistently O(nlogn) time complexity.
- Observations: Performs well across different array types and sizes, reliable for sorting.

3-way Merge Sort:

- Performance: O(nlogn) time complexity (similar to regular merge sort).
- Observations: Further experiments needed for a comprehensive evaluation.

Lomuto Quick Sort:

- Performance: O(n^2) time complexity in the worst case.
- Observations: Performs better on already sorted arrays but not the most efficient for general cases.

Hoare Quick Sort:

- Performance: O(n^2) time complexity in the worst case.
- Observations: Similar characteristics to Lomuto quicksort.

Heap Sort:

- Performance: Consistently O(nlogn) time complexity.
- Observations: Performs well across different array types and sizes, a reliable choice for sorting.

In summary, the choice of a sorting algorithm depends on the specific characteristics of the data. Merge sort and heap sort are generally efficient, while quicksort variants may be suitable for specific scenarios. The experimental results provide insights into each algorithm's strengths and weaknesses, aiding in informed decision-making for sorting tasks.