Title: Comparative Analysis of Sorting Algorithms

Term Paper of Data Structures and Algorithms Course (PC-CS301)

Submitted by

Name of Students	Examination Roll Nos.
Adrija Ghosh	**11600223018**
Joyita Bhawal	**11600223058**
Tanmay Dutta	**11600223133**

Under the joint supervisions of

Mrs. Rachita Ghoshhajra and Mr. Puspen Lahiri
Assistant Professors, Dept. of CSE, MCKVIE

Department of Computer Science & Engineering
MCKV Institute of Engineering
243, G.T. Road (N)
Liluah, Howrah – 711204

1. Abstract:

Sorting is a core operation in computer science, essential for optimizing numerous computational tasks. This paper presents a comparative analysis of three widely used sorting algorithms: Merge Sort, Quick Sort and Heap Sort. The objective is to examine their algorithms, underlying approaches, strengths, weaknesses, and the best case scenariost. Merge Sort employs a stable Divide and Conquer strategy to divide an array into smaller sub arrays and merge them in sorted order, providing a reliable but space-intensive solution. Quick Sort, also based on Divide and Conquer, partitions the array around a pivot, often offering faster performance but with a potential worst-case degradation. Heap Sort uses a binary heap structure to perform in-place sorting with consistent efficiency. Through an in-depth exploration of each algorithm's approach and practical use cases, this paper highlights the factors that influence their performance and guides the selection of the most appropriate algorithm depending on the specific problem at hand.

2. Introduction:

Merge Sort, Quick Sort, and Heap Sort are widely used sorting algorithms, each with distinct characteristics. Merge Sort employs a Divide and Conquer approach, recursively dividing the array into smaller subarrays and merging them, ensuring reliable sorting but requiring extra space. Quick Sort, also Divide and Conquer, partitions the array around a pivot element, offering fast sorting in most cases but potentially suffering from performance degradation depending on pivot selection. Heap Sort uses a binary heap to perform in-place sorting, avoiding Quick Sort's worst-case scenario but often slower due to additional overhead. Each algorithm is suited for different problem types and data sets.

3. Approach:

3.1 Merge Sort Approach Algorithms:

```
Algorithm Merge_Sort(low, high) {
   if (low < high) then{
      mid := ⌊(low + high) / 2⌋
      Merge_Sort(low, mid)
      Merge_Sort(mid + 1, high)
      Merge(low, mid, high)
   }
}
```

- **Working**:

- If low < high, the sub-array has more than one element, so it must be divided.
- The array a[low:high] is split into two parts:
  - **Left sub-array**: a[low:mid] **Right sub-array**: a[mid+1:high]
- **Recursive Calls**:
  - Call Merge_Sort(low, mid) to sort the left sub-array.
  - Call Merge_Sort(mid+1, high) to sort the right sub-array.
- **Merge Step**:
  - Call Merge(low, mid, high) to merge the two sorted sub-arrays into a single sorted array.

```
Algorithm Merge(low, mid, high) {
   h := low
   i := low
   j := mid + 1

   while ((h <= mid) and (j <= high)) do {
      if (a[h] <= a[j]) then  {
         b[i] := a[h]
         h := h + 1
      }
      else{
         b[i] := a[j]
         j := j + 1
      }
      i := i + 1
   }

   if (h > mid) then
      for k := j to high do  {
         b[i] := a[k]
         i := i + 1
      }
   else
      for k := h to mid do  {
         b[i] := a[k]
         i := i + 1
      }
   for k := low to high do
      a[k] := b[k]
}
```

Working of Merge Sort

- **Initialization**:
  - h starts from low to track the first sub-array a[low:mid].
  - j starts from mid+1 to track the second sub-array a[mid+1:high].
  - i starts from low to store the merged result in b[low:high].

- **Merge Process**:
    - While both sub-arrays have elements (h <= mid and j <= high), compare a[h] and a[j].
    - Place the smaller element in b[i] and increment the respective pointers (h or j).
- **Copy Remaining Elements**:
    - If the first sub-array is exhausted, copy the remaining elements of the second sub-array to b[].
    - If the second sub-array is exhausted, copy the remaining elements of the first sub-array to b[].
- **Copy Back**:
    - Copy all elements from b[low:high] back to the original array a[low:high].

3.2 Quick Sort Approach Algorithms:

```
Procedure Quicksort(Q, p, r) {
   if (p < r) then {
      q := Partition(Q, p, r)
      Quicksort(Q, p, q - 1)
      Quicksort(Q, q + 1, r)
   }
}
```

Working:

- If p < r, the sub-array has more than one element, so it must be partitioned.
- **Partition Step**:
    - Call Partition(Q, p, r) to divide Q[p:r] into two sub-arrays:
        1. **Left sub-array**: Elements less than or equal to the pivot.
        2. **Right sub-array**: Elements greater than the pivot.
    - Partition returns the index **q** where the pivot is placed.
- **Recursive Calls**:
    - Call Quicksort(Q, p, q-1) to sort the left sub-array.
    - Call Quicksort(Q, q+1, r) to sort the right sub-array.

```
Algorithm Partition(Q, p, r) {
   x := Q[p]
   i := p
   j := r
  while (Q[i] <= x)  i := i + 1
  while (Q[j] > x)  j := j - 1
  if (i < j) then Swap(Q[i], Q[j])
  else {
      Swap(Q[j], Q[p])
      return j
   }
}
```

- **Working**:
  - **Initialization**:
    - **x** is the pivot element, taken as the first element of the sub-array, Q[p].
    - **i** starts at **p** and scans the array from the left.
    - **j** starts at **r** and scans the array from the right.
  - **Partition Process**:
    - The **while loop** moves **i** to the right until Q[i] > x.
    - The **while loop** moves **j** to the left until Q[j] ≤ x.
    - If **i < j**, then swap Q[i] with Q[j] so that the smaller element is on the left and the larger element is on the right.
    - If **i ≥ j**, it means the partition is complete, and the pivot is placed in its correct position.
  - **Final Swap**:
    - Swap **Q[j]** with **Q[p]**, placing the pivot at its correct position.
    - Return **j**, which is the position of the pivot.

3.3 Heap Sort Approach Algorithms:

```
Algorithm MAX-HEAPIFY(A, i) {
   l := LEFT(i)
   r := RIGHT(i)
   if (l ≤ heap-size[A] and A[l] > A[i])  largest := l
   else  largest := i
   if (r ≤ heap-size[A] and A[r] > A[largest]) largest := r
   if (largest ≠ i)  {
      Swap(A[i], A[largest])
      MAX-HEAPIFY(A, largest)
   }
}  .
```

How it works:

- Find the indices of the left and right children of node **i**.
- Compare **A[i]** with its children.
- If one of the children is larger than **A[i]**, swap them and recursively call **MAX-HEAPIFY** on the child to restore the max-heap property.

```
Algorithm BUILD-MAX-HEAP(A)  {
   heap-size[A] := length[A]
   for i := ⌊length[A] / 2⌋ down to 1
      MAX-HEAPIFY(A, i) }
```

How it works:

- Initialize the heap size to the total number of elements in **A**.
- Start from the last non-leaf node **⌊length[A] / 2⌋** and call **MAX-HEAPIFY** on each node.
- This ensures that every subtree satisfies the max-heap property.

```
Algorithm HEAPSORT(A)  {
   BUILD-MAX-HEAP(A)
   for i := length[A] down to 2  {
      Swap(A[1], A[i])
      heap-size[A] := heap-size[A] - 1
      MAX-HEAPIFY(A, 1)
   }
}  .
```

How it works:

- Call **BUILD-MAX-HEAP(A)** to convert **A** into a max-heap.
- Repeatedly swap the root **A[1]** (largest element) with the last element **A[i]**.
- Decrease the heap size by 1 and restore the heap property using **MAX-HEAPIFY**.
- This process continues until the entire array is sorted in ascending order.

4. Example with Explanation:

4.1 Merge Sort

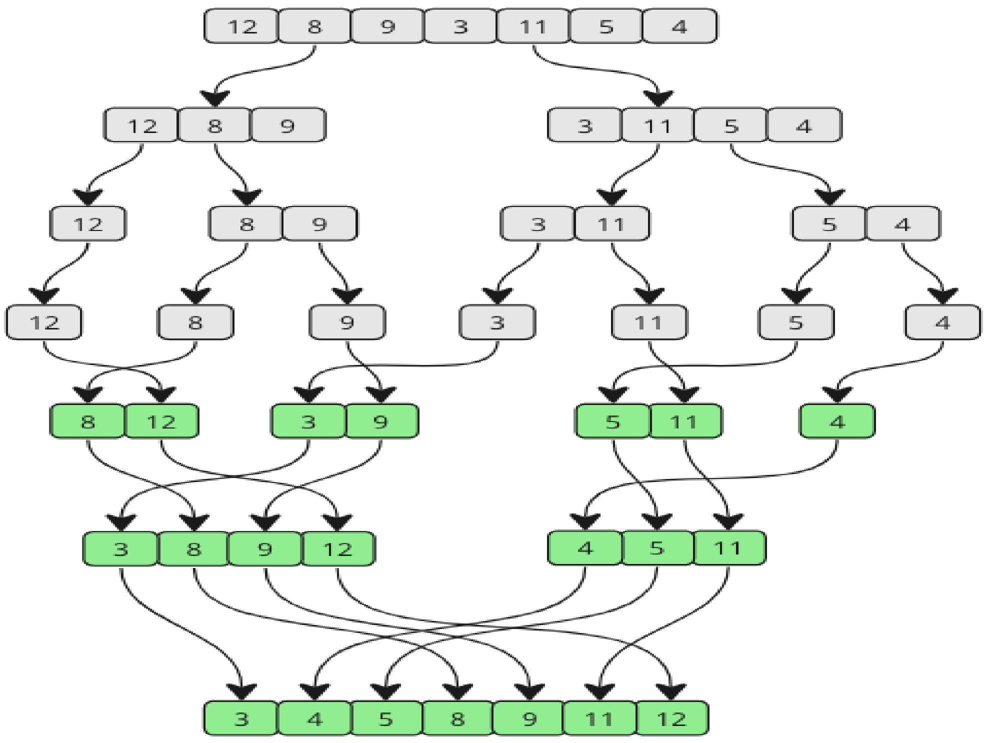
Let us understand merge sort approach with array a[]= [12, 8, 9, 3, 11, 5, 4]

1. **Divide the Array**:
   [12, 8, 9, 3, 11, 5, 4] → [12, 8, 9] | [3, 11, 5, 4]
2. **Sort Left Sub-array [12, 8, 9]**:
   [12] | [8, 9] → [8, 9]
   Merge [12] and [8, 9] → [8, 9, 12]
3. **Sort Right Sub-array [3, 11, 5, 4]**:
   [3, 11] → [3] | [11] → [3, 11]
   [5, 4] → [5] | [4] → [4, 5]
   Merge [3, 11] and [4, 5] → [3, 4, 5, 11]

4. **Merge [8, 9, 12] and [3, 4, 5, 11]**:
   Result → [3, 4, 5, 8, 9, 11, 12]

Final Sorted Array: [3, 4, 5, 8, 9, 11, 12]

Following is the pictorial representation of Merge Sort approach with each step clearly explained.



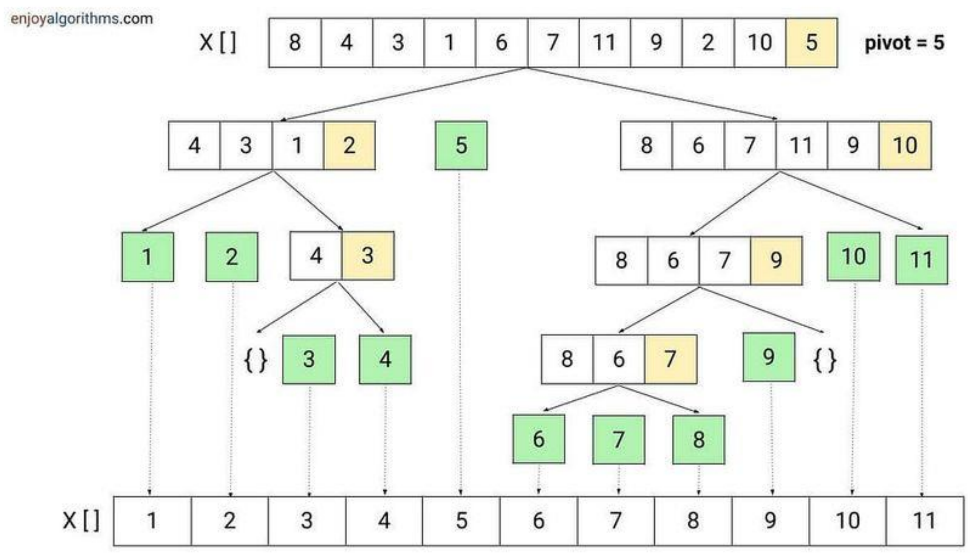
4.2 Quick Sort

Quick Sort Approach with Array: [8, 4, 3, 1, 6, 7, 11, 9, 2, 10, 5] (Pivot = 5)

1. **Partition around pivot 5**:
   Left: [4, 3, 1, 2] | Pivot: 5 | Right: [8, 6, 7, 11, 9, 10]
2. **Left Sub-array [4, 3, 1, 2] (Pivot = 2)**:
   Left: [1] | Pivot: 2 | Right: [4, 3]
   - Partition [4, 3] around pivot 3 → Left: None | Pivot: 3 | Right: [4]
3. **Right Sub-array [8, 6, 7, 11, 9, 10] (Pivot = 10)**:
   Left: [8, 6, 7, 9] | Pivot: 10 | Right: [11]
   - Partition [8, 6, 7, 9] around pivot 9 → Left: [8, 6, 7] | Pivot: 9 | Right: None
   - Partition [8, 6, 7] around pivot 7 → Left: [6] | Pivot: 7 | Right: [8]

Final Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

Following is the pictorial representation of the Quick Sort approach.



4.3 Heap Sort

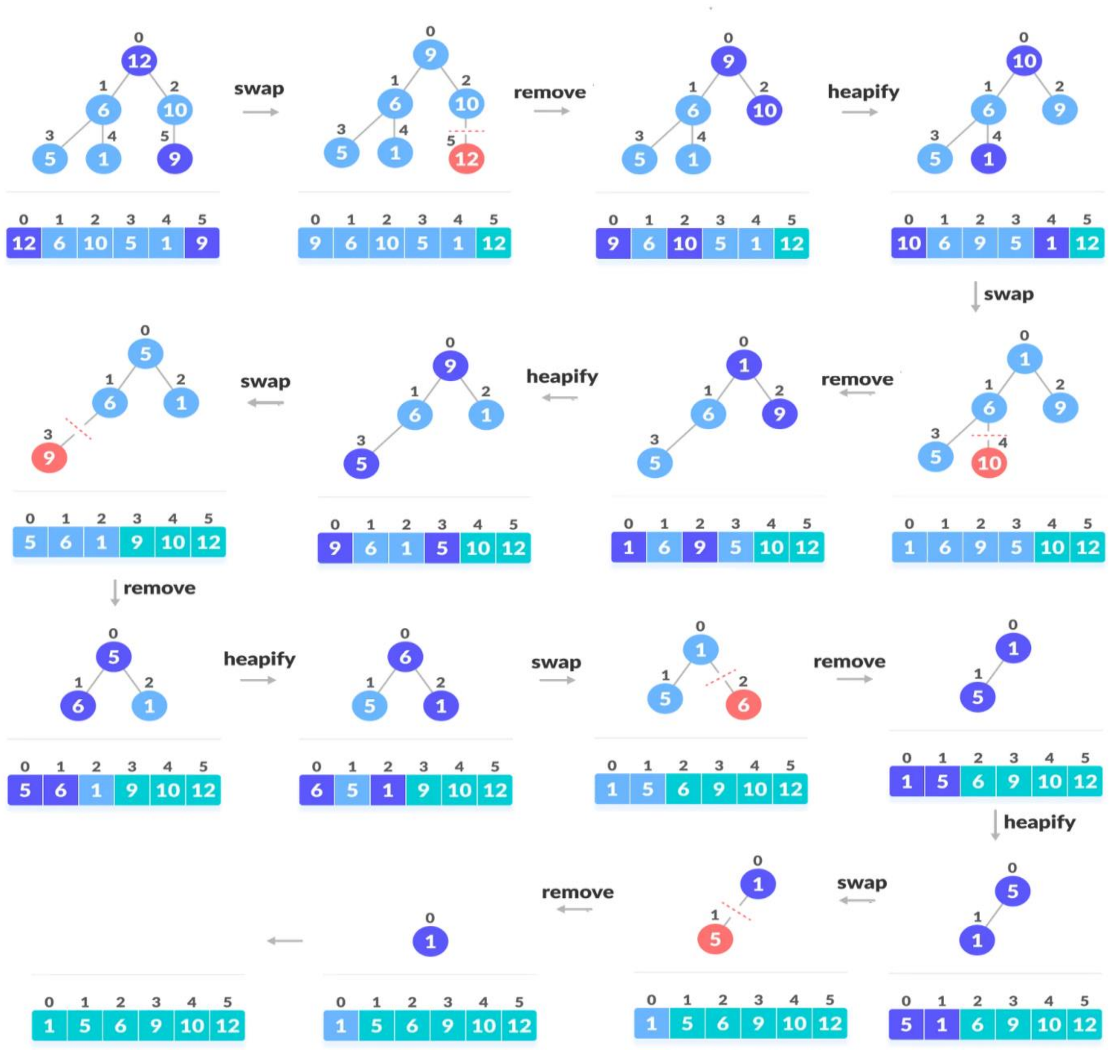
Heap Sort Approach with Array: [12, 6, 10, 5, 1, 9]

1. **Build a Max Heap**:
   Organize the array into a max heap where the largest element is at the root.
   Resulting Max Heap: [12, 6, 10, 5, 1, 9]
2. **Swap Root with Last Element**:
   Swap the root (largest element) with the last element in the heap.
   Array after swap: [9, 6, 10, 5, 1, 12]
3. **Remove Largest Element**:
   Remove the last element (now the largest).
   Array after removal: [9, 6, 10, 5, 1]
4. **Heapify Remaining Heap**:
   Restore the max heap property.
   Heapified array: [10, 6, 9, 5, 1]
5. **Repeat Steps 2-4**:
   - **Iteration 2**: Swap root (10) with last element (1), array: [1, 6, 9, 5, 10, 12]. Remove 10. Heapify: [9, 6, 1, 5].
   - **Iteration 3**: Swap root (9) with last element (5), array: [5, 6, 1, 9, 10, 12]. Remove 9. Heapify: [6, 5, 1].

- **Iteration 4**: Swap root (6) with last element (1), array: [1, 5, 6, 9, 10, 12]. Remove 6. Heapify: [5, 1].
- **Final Iteration**: Swap root (5) with last element (1), array: [1, 5, 6, 9, 10, 12]. Remove 5. Only 1 remains.

6. **Final Result**: The sorted array is: [1, 5, 6, 9, 10, 12]

Following is the pictorial representation of **Heap Sort.**



5. Result and Discussion:

Following is the comparison between Merge Sort, Quick Sort and Heap Sort in various criterias.

Criterion	**Merge Sort**	**Quick Sort**	**Heap Sort**
Algorithm Type	Divide and Conquer	Divide and Conquer	Selection Sort (Heap-based)
Time Complexity	Best: O(n log n)	Best: O(n log n)	Best: O(n log n)

	Worst: O(n log n)	Worst: O(n²)	Worst: O(n log n)
	Average: O(n log n)	Average: O(n log n)	Average: O(n log n)
Space Complexity	O(n) (extra array)	O(log n) (in-place)	O(1) (in-place)
Stability	**Stable**	Not Stable	Not Stable
In-place Sorting	**No** (requires extra space)	**Yes**	**Yes**
Partitioning	By splitting array	By pivoting around an element	By creating a heap (binary tree)
Best Use Case	Large datasets, Linked lists	Randomized/Unsorted data	When space is limited
Worst Use Case	When space is limited	Already sorted/reverse sorted	Requires frequent heapification
Recursive/Iterative	Recursive	Recursive	Can be iterative

6. Conclusion:

Different scenarios for different approaches**:**

Scenarios	**Best Choice**	**Reason**
Space is limited	Heap Sort	In-place sorting (O(1) extra space)
Linked lists	Merge Sort	Merge sort works directly on linked lists
Already sorted data	Merge Sort	Consistent O(n log n) time
Unsorted/random data	Quick Sort	Fastest in practical cases
Need stability	Merge Sort	Stable, unlike quick or heap sort
Need guaranteed time O(n log n)	Heap Sort / Merge Sort	Guaranteed O(n log n) worst case
Small datasets (in-place)	Quick Sort	In-place sorting with better cache locality
Large datasets	Merge Sort	Better due to consistent performance

To conclude, Merge Sort is stable and consistently O(nlog⁡n)O(n \log n)O(nlogn), but requires extra space. Quick Sort is fast on average but has a bad worst case. Heap Sort is space-efficient but not stable. Choice depends on stability, space, and time constraints.

7. References:

- https://www.w3schools.com/dsa/dsa_algo_mergesort.php
- https://www.enjoyalgorithms.com/blog/quick-sort-algorithm
- https://www.programiz.com/dsa/heap-sort
- Notes on Data Structure (SORTING)- Prepared by: Puspen Lahiri, Asst. Professor, CSE, MCKVIE