Sorting Algorithms 1

Introduction to Sorting

Sorting is the process of arranging data in a particular order (ascending or descending).

Why is sorting important?

- Faster searching (e.g., Binary Search)
- Efficient data presentation
- Useful in data analysis and reporting

Types of sorting algorithms:

- Comparison-based (Bubble, Merge, Quick)
- Non-comparison-based (Counting, Radix, Bucket)

Sorting Algorithms

- Quicksort
- Merge sort
- Bubble sort
- Insertion sort
- Selection sort
- Timsort

Quick Sort

Quick sort, also known as partition exchange sort.

Quicksort algorithm is based on the divide-and conquer class of algorithms, similar to the merge sort algorithm, where we break (divide) a problem into smaller chunks that are much simpler to solve, and further, the final results are obtained by combining the outputs of smaller problems (conquer).

The pivot can be:

- Any element at random.
- The first or last element.
- Middle element.

How quicksort algorithm works:

1. We start by choosing a pivot element with which all the data elements are to be compared, and at the end of the first iteration, this pivot element will be placed in its correct position in the list. In order to place the pivot element in its correct position, we use two pointers, a left pointer, and a right pointer. This process is as follows:

Cont.

a. The left pointer initially points to the value at index 1, and the right pointer points to the value at the last index. The main idea here is to move the data items that are on the wrong side of the pivot element. So, we start with the left pointer, moving in a left-to-right direction until we reach a position where the data item in the list has a greater value than the pivot element.

b. Similarly, we move the right pointer toward the left until we find a data item less than the pivot element.

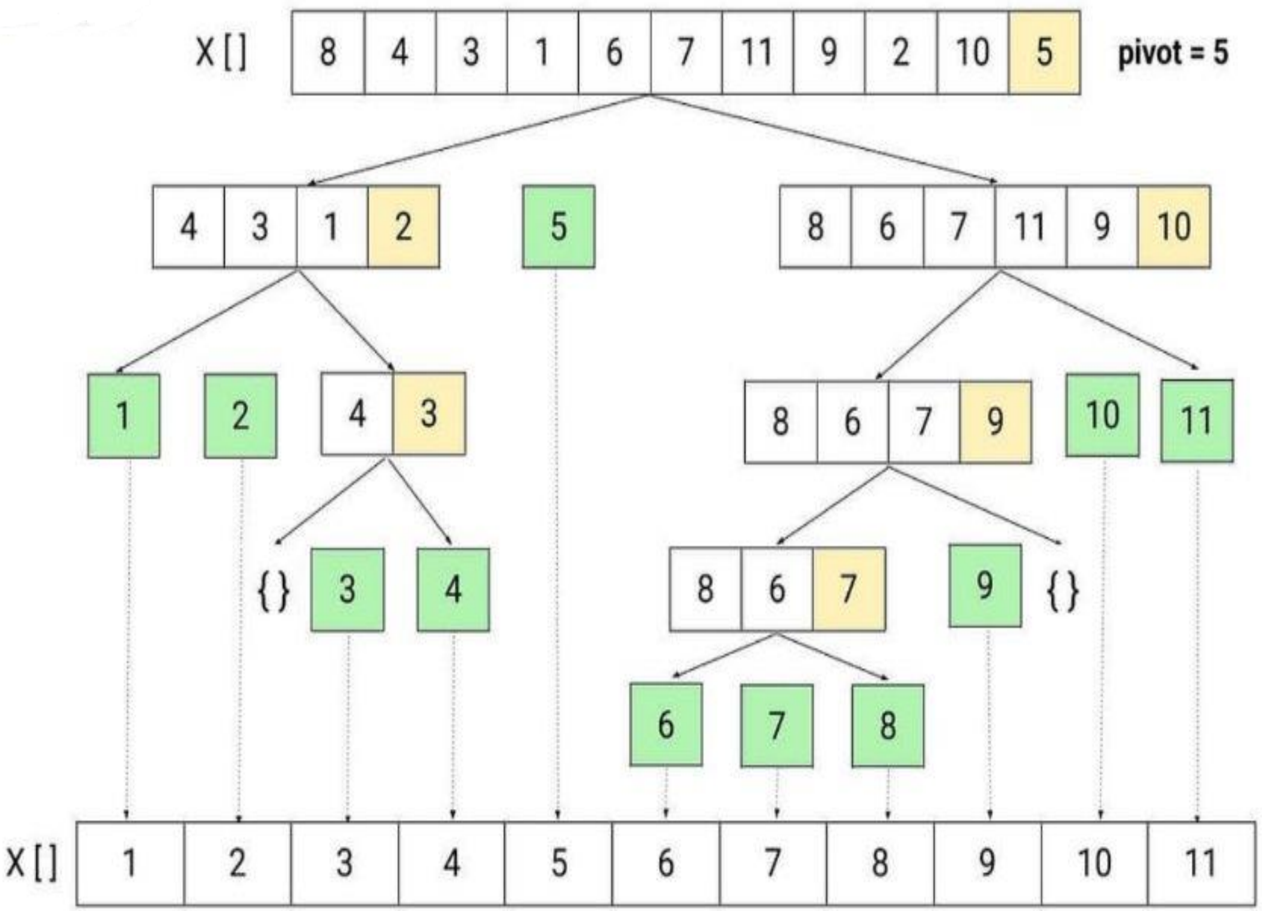
c. Next, we swap these two values indicated by the left and right pointers.

d. We repeat the same process until both pointers cross each other, in other words, until the right pointer index indicates a value less than that of the left pointer index.

2. After each iteration described in step 1, the pivot element will be placed at its correct position in the list, and the original list will be divided into two unordered sublists, left and right. We follow the same process (as described in step 1) for both these left and right sublists until each of the sublists contains a single element.

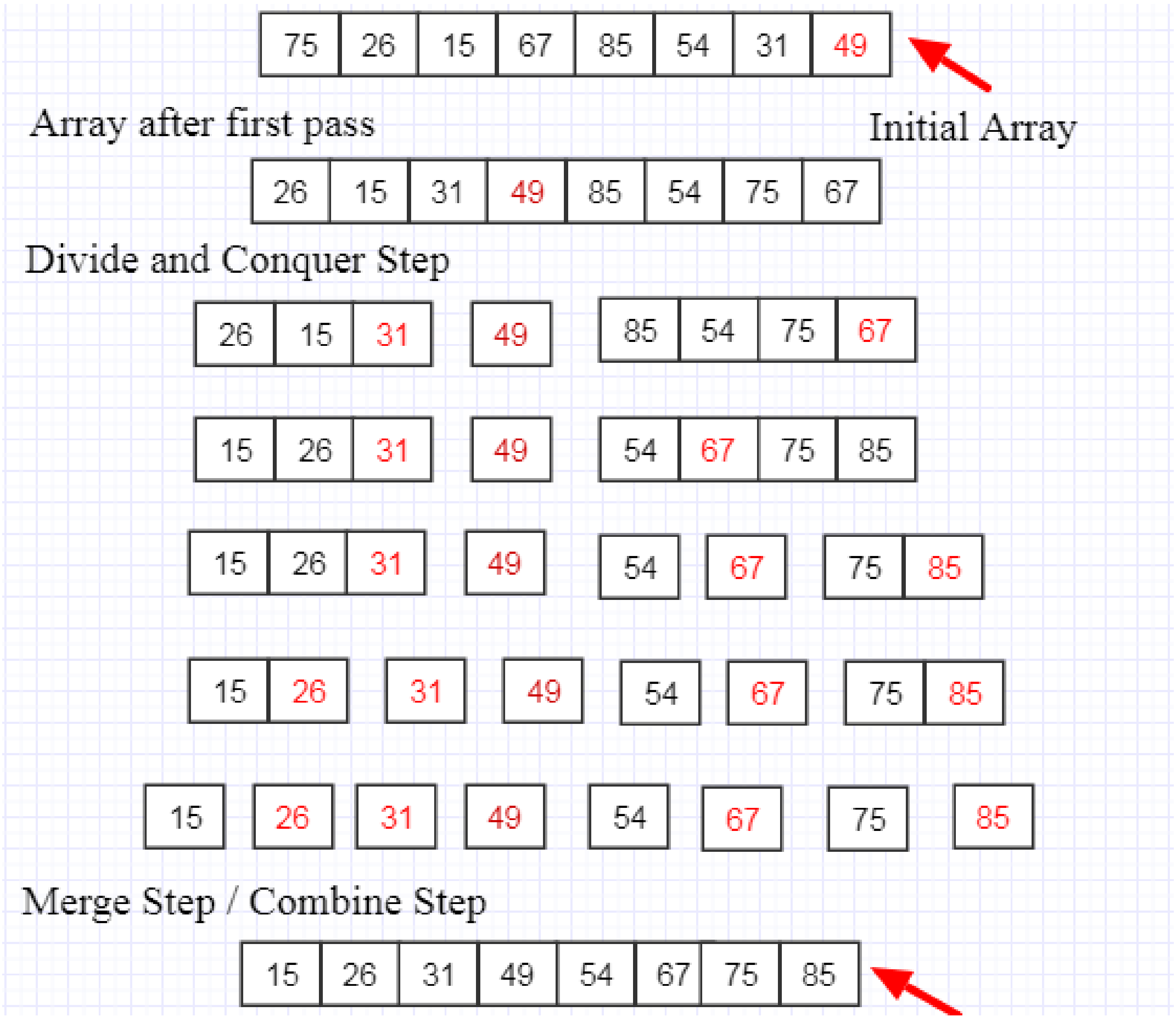
3. Finally, all the elements will be placed at their correct positions, which will give the sorted list as an output.

Quicksort



X[]
8 4 3 1 6 7 11 9 2 10 5
pivot = 5
4 3 1 2
5
8 6 7 11 9 10
1
2
4 3
8 6 7 9
10
11
{}
3
4
8 6 7
9
{}
6
7
8
X[]
1 2 3 4 5 6 7 8 9 10 11

Quicksort



75 26 15 67 85 54 31 49
Initial Array
Array after first pass
26 15 31 49 85 54 75 67
Divide and Conquer Step
26 15 31 49 85 54 75 67
15 26 31 49 54 67 75 85
15 26 31 49 54 67 75 85
15 26 31 49 54 67 75 85
15 26 31 49 54 67 75 85
Merge Step / Combine Step
15 26 31 49 54 67 75 85

Time complexity

Case	Time Complexity	Explanation
Best Case	$O(n \log n)$	The pivot divides the array into two **equal halves** each time. Log(n) levels of recursion × n comparisons.
Average Case	$O(n \log n)$	On average, the pivot divides the array reasonably well (not perfectly, but not extremely skewed).
Worst Case	$O(n^2)$	Happens when the pivot is the smallest or largest element repeatedly (highly unbalanced splits), such as in sorted arrays without randomization.

Space Complexity

Type	Space Complexity	Explanation
Auxiliary Space	O(log n) on average	Due to recursive calls on the call stack for each partition. Balanced partitions result in log(n) levels of recursion.
Worst Case	O(n)	In the worst case (highly unbalanced), recursion depth could go up to n.
In-Place Sorting	Yes	Quick Sort **does not require extra space** for sorting – it works within the array using swaps.

Quick sort Pseudocode

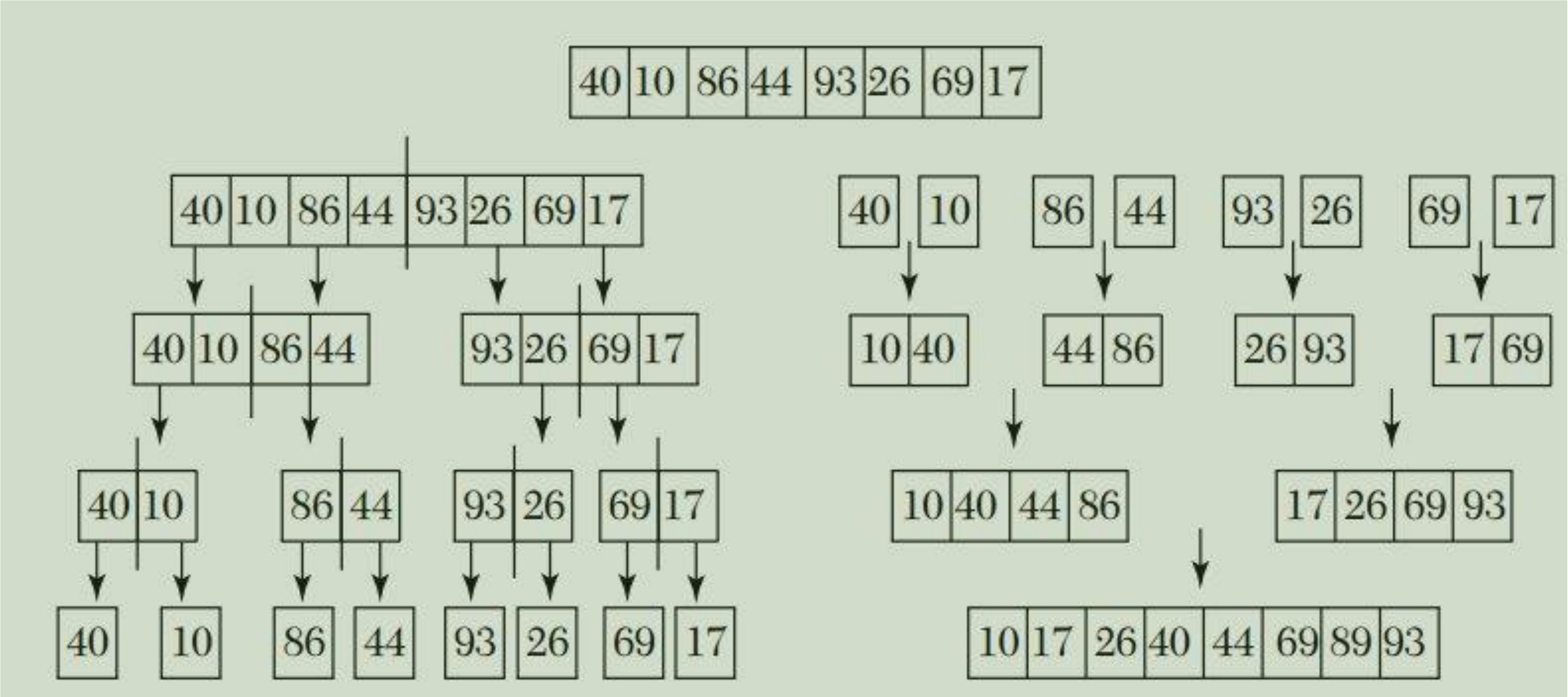
```
function QUICKSORT(ARRAY, START, END)
    # base case size <= 1
    if START >= END then
        return
    end if
    PIVOTINDEX = PARTITION(ARRAY, START, END)
    QUICKSORT(ARRAY, START, PIVOTINDEX – 1)
    QUICKSORT(ARRAY, PIVOTINDEX + 1, END)
end function
```

cont

```
function PARTITION(ARRAY, START, END)
  PIVOTVALUE = ARRAY[END]
  PIVOTINDEX = START
  loop INDEX from START to END
    if ARRAY[INDEX] <= PIVOTVALUE
      TEMP = ARRAY[INDEX]
      ARRAY[INDEX] = ARRAY[PIVOTINDEX]
      ARRAY[PIVOTINDEX] = TEMP
      PIVOTINDEX = PIVOTINDEX + 1
    end if
  end loop
  return PIVOTINDEX – 1
```

Merge Sort

Merge sort is a sorting method which follows the divide and conquer approach. The divide and conquer approach is a very good approach in which divide means partitioning the array having n elements into two subarrays of n/2 elements each. However, if there are no elements present in the list/array or if an array contains only one element, then it is already sorted. However, if an array has more elements, then it is divided into two sub-arrays containing equal elements in them. Conquer is the process of sorting the two sub-arrays recursively using merge sort. Finally, the two sub-arrays are merged into one single sorted array.


40 10 86 44 93 26 69 17
40 10 86 44 93 26 69 17
40 10 86 44
93 26 69 17
40 10
86 44
93 26
69 17
40
10
86
44
93
26
69
17
40
10
86
44
93
26
69
17
10 40
44 86
26 93
17 69
10 40 44 86
17 26 69 93
10 17 26 40 44 69 89 93

```
function MERGE(ARRAY, START, HALF, END)
   TEMPARRAY = new array[END – START + 1]
   INDEX1 = START
   INDEX2 = HALF + 1
   NEWINDEX = 0
   loop while INDEX1 <= HALF and INDEX2 <= END
      if ARRAY[INDEX1] < ARRAY[INDEX2] then
         TEMPARRAY[NEWINDEX] = ARRAY[INDEX1]
         INDEX1 = INDEX1 + 1
      else
         TEMPARRAY[NEWINDEX] = ARRAY[INDEX2]
         INDEX2 = INDEX2 + 1
      end if

   loop while INDEX1 <= HALF
      TEMPARRAY[NEWINDEX] = ARRAY[INDEX1]
      INDEX1 = INDEX1 + 1
      NEWINDEX = NEWINDEX + 1
   end loop
   loop while INDEX2 <= END
      TEMPARRAY[NEWINDEX] = ARRAY[INDEX2]
      INDEX2 = INDEX2 + 1
      NEWINDEX = NEWINDEX + 1
   end loop
   loop INDEX from 0 to size of TEMPARRAY – 1
      ARRAY[START + INDEX] =
```

Time Complexity

Case	Time Complexity	Explanation
Best Case	O(n log n)	Even if the array is already sorted, merge sort still divides and merges every element.
Average Case	O(n log n)	Recursively divides the array into halves (log n levels), and at each level it performs O(n) merging.
Worst Case	O(n log n)	Same as average and best, because it always does the same number of operations regardless of input order.

Space Complexity

Type	Space Complexity	Explanation
Auxiliary Space	O(n)	Merge Sort creates temporary arrays to merge the two halves.
In-place	No	It **requires additional space** for merging.
Recursive Stack	O(log n)	Due to recursive function calls, though this is minor compared to the auxiliary array space.