

Data Structures and Algorithms

Searching and Sorting- The Heart of Problem Solving

Abhinav Kesarwani

B.Tech Student | Artificial Intelligence & Data Science

Gati Shakti Vishwavidyalaya (Ministry of Railways, Govt. of India)

[Linkedin](#)

Array as Data Structure

Operations on Array:

- Traversing
- Inserting
- Deleting
- Sorting
- Searching
- Merging

Complexity of Linear Search Algorithm

Linear search executes in $O(n)$ time where n is the number of elements in the array.

Best case of linear search is when VAL is equal to the first element of the array.

Worst case will happen when either VAL is not present in the array or it is equal to the last element of the array.

Binary Search

- Binary search is a searching algorithm that works efficiently with a sorted list.
- Analogy of a telephone directory or dictionary.
- Example:

```
int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Binary Search

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

Step 1: [INITIALIZE] SET BEG = lower_bound

 END = upper_bound, POS = - 1

Step 2: Repeat Steps 3 and 4 while BEG <= END

Step 3: SET MID = (BEG + END)/2

Step 4: IF A[MID] = VAL

 SET POS = MID

 PRINT POS

 Go to Step 6

 ELSE IF A[MID] > VAL

 SET END = MID - 1

 ELSE

 SET BEG = MID + 1

 [END OF IF]

 [END OF LOOP]

Step 5: IF POS = -1

 PRINT "VALUE IS NOT PRESENT IN THE ARRAY"

 [END OF IF]

Step 6: EXIT

Binary Search

- In the binary search algorithm, with each comparison, the size of the segment where search has to be made is reduced to half.
- In order to locate a particular value in the array, the total number of comparisons that will be made is given as

$$2^{f(n)} > n$$

or

$$f(n) = \log_2 n$$

Sorting

- The term sorting means arranging the elements of the array so that they are placed in some relevant order which may either be ascending order or descending order.
- If A is an array then the elements of A are arranged in sorted order (ascending order) in such a way that, $A[0] < A[1] < A[2] < \dots < A[N]$.
- For example, if we have an array that is declared and initialized as,

`int A[] = {21, 34, 11, 9, 1, 0, 22};`

Then the sorted array (ascending order) can be given as,

`A[] = {0, 1, 9, 11, 21, 22, 34};`

Bubble Sort

Technique:

- In Pass 1, $A[0]$ and $A[1]$ are compared, then $A[1]$ is compared with $A[2]$, $A[2]$ is compared with $A[3]$, and so on. Finally, $A[N-2]$ is compared with $A[N-1]$. **Pass 1 involves $n-1$ comparisons** and places the biggest element at the highest index of the array.
- In Pass 2, $A[0]$ and $A[1]$ are compared, then $A[1]$ is compared with $A[2]$, $A[2]$ is compared with $A[3]$, and so on. Finally, $A[N-3]$ is compared with $A[N-2]$. **Pass 2 involves $n-2$ comparisons** and places the second biggest element at the second highest index of the array.
- In Pass $n-1$, $A[0]$ and $A[1]$ are compared so that $A[0] < A[1]$. After this step, all the elements of the array are arranged in ascending order.

Bubble Sort

$A[] = \{30, 52, 29, 87, 63, 27, 19, 54\}$

Pass 1:

30, 52, 29, 87, 63, 27, 19, 54

30, 52, 29, 87, 63, 27, 19, 54

30, 52, 29, 87, 63, 27, 19, 54

30, 29, 52, 87, 63, 27, 19, 54

30, 29, 52, 87, 63, 27, 19, 54

30, 29, 52, 63, 87, 27, 19, 54

30, 29, 52, 63, 27, 87, 19, 54

30, 29, 52, 63, 27, 19, 87, 54

30, 29, 52, 63, 27, 19, 54, **87**

Bubble Sort

$A[] = \{30, 52, 29, 87, 63, 27, 19, 54\}$

Pass 1: 30, 29, 52, 63, 27, 19, 54, **87**

Pass 2: 29, 30, 52, 27, 19, 54, **63, 87**

Pass 3: 29, 30, 27, 19, 52, **54, 63, 87**

Pass 4: 29, 27, 19, 30, **52, 54, 63, 87**

Pass 5: 27, 19, 29, **30, 52, 54, 63, 87**

Pass 6: 19, 27, **29, 30, 52, 54, 63, 87**

Pass 7: 19, **27, 29, 30, 52, 54, 63, 87**

Algorithm for Bubble Sort

BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For $I = 0$ to $N-1$

Step 2: Repeat For $J = 0$ to $N - I$

Step 3: IF $A[J] > A[J + 1]$
 SWAP $A[J]$ and $A[J+1]$

 [END OF INNER LOOP]

 [END OF OUTER LOOP]

Step 4: EXIT

Complexity of Bubble Sort

- Complexity of any sorting algorithm depends upon the number of comparisons.
- In bubble sort, we have $N-1$ passes in total.
- In the first pass, $N-1$ comparisons are made to place the highest element in its correct position. In Pass 2, there are $N-2$ comparisons. Therefore, we need to calculate the total number of comparisons:

$$f(n) = (n - 1) + (n - 2) + (n - 3) + \dots + 3 + 2 + 1$$

$$f(n) = n(n - 1)/2$$

$$f(n) = n^2/2 + O(n)$$

$$\mathbf{f(n) = O(n^2)}$$

Properties of Sorting Algorithms

Stable: A sorting algorithm is stable if it does not change the order of elements with the same value.

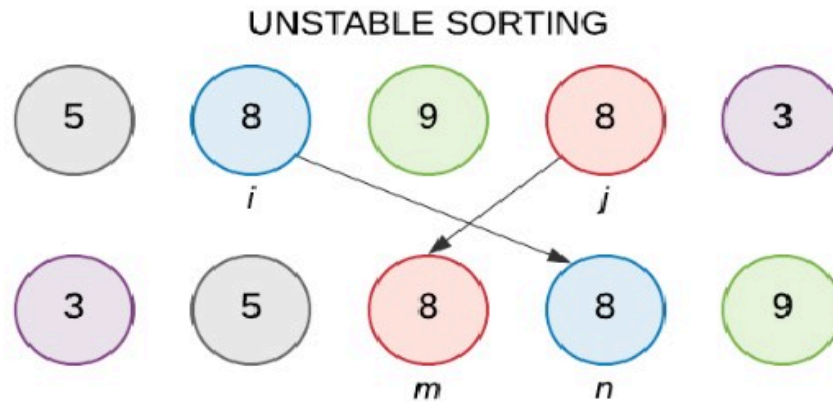
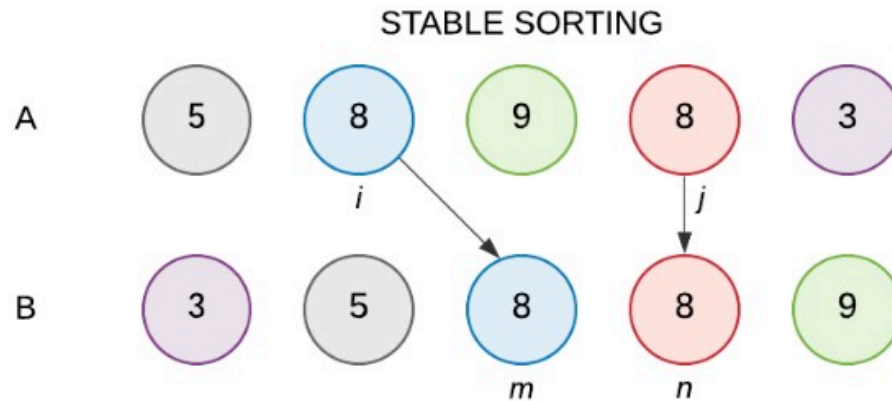
In-place:

An in-place sorting algorithm uses constant space for producing the output.

Online:

An online sorting algorithm is one that will work if the elements to be sorted are provided one at a time with the understanding that the algorithm must keep the sequence sorted as more and more elements are added in.

Stable and Unstable Sorting



Properties of Bubble Sort

- Stable Sorting
- Inplace Sorting
- Not Online

Insertion Sort

- Insertion sort is a very simple sorting algorithm, in which the sorted array (or list) is built one element at a time.
- Insertion sort works as follows:
 - 1.The array of values to be sorted is divided into two sets. One that stores sorted values and the other contains unsorted values.
 - 2.The sorting algorithm will proceed until there are elements in the unsorted set.
 - 3.Suppose there are n elements in the array. Initially the element with index 0 (assuming LB, Lower Bound = 0) is in the sorted set, rest all the elements are in the unsorted set
 - 4.The first element of the unsorted partition has array index 1 (if $LB = 0$)
 - 5.During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Insertion Sort

39	9	45	63	18	81	108	54	72	36
----	---	----	----	----	----	-----	----	----	----

39	9	45	63	18	81	108	54	72	36
----	---	----	----	----	----	-----	----	----	----

A[0] is the only element in sorted list

9	39	45	63	18	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 1)

9	39	45	63	18	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 2)

9	39	45	63	18	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 3)

9	18	39	45	63	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 4)

9	18	39	45	63	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 5)

9	18	39	45	63	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 6)

9	18	39	45	54	63	81	108	72	36
---	----	----	----	----	----	----	-----	----	----

(Pass 7)

9	18	39	45	54	63	72	81	108	36
---	----	----	----	----	----	----	----	-----	----

(Pass 8)

9	18	36	39	45	54	63	72	81	108
---	----	----	----	----	----	----	----	----	-----

(Pass 9)

 Sorted  Unsorted

Insertion Sort

- In Pass 1, $A[0]$ is the only element in the sorted set.
- In Pass 2, $A[1]$ will be placed either before or after $A[0]$, so that the array A is sorted.
- In Pass N , $A[N-1]$ will be placed in its proper place so that the array A is sorted.
- To insert the element $A[K]$ in the sorted list $A[0], A[1], \dots, A[K-1]$, we need to compare $A[K]$ with $A[K-1]$, then with $A[K-2]$, then with $A[K-3]$ until we meet an element $A[J]$ such that $A[J] \leq A[K]$.
- In order to insert $A[K]$ in its correct position, we need to move each element $A[K-1], A[K-2], \dots, A[J]$ by one position and then $A[K]$ is inserted at the $(J+1)$ th location.

Insertion Sort

INSERTION-SORT (ARR, N)

```
Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:     SET TEMP = ARR[K]
Step 3:     SET J = K - 1
Step 4:     Repeat while TEMP <= ARR[J]
                SET ARR[J + 1] = ARR[J]
                SET J = J - 1
                [END OF INNER LOOP]
Step 5:     SET ARR[J + 1] = TEMP
                [END OF LOOP]
Step 6: EXIT
```

Complexity of Insertion Sort

- **Best Case:** Array is already sorted, algorithm has a linear running time (i.e., $O(n)$) as during each iteration, the first element from unsorted set is compared only with the last element of the sorted set of the array.
- **Worst Case:** Array is sorted in reverse order. Here, the first element of the unsorted set has to ~~be compared with almost every element in the sorted set.~~ Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element. Therefore, in the worst case, insertion sort has a quadratic running time (i.e., $O(n^2)$).
- **Average Case:**, Algorithm will have to make at least $(K-1)/2$ comparisons. Thus, the average case also has a **quadratic running time**.

Advantages of Insertion Sort

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It requires less memory space (only $O(1)$ of additional memory space).
- It is said to be online, as it can sort a list as and when it receives new elements.

Selection Sort

Technique:

- First find the smallest value in the array and place it in the first position. Then find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted.
- In Pass 1, find the position POS of the smallest value in the array and then swap $ARR[POS]$ and $ARR[0]$. Thus, $ARR[0]$ is sorted.
- In pass 2, find the position POS of the smallest value in sub-array of $N-1$ elements. Swap $ARR[POS]$ with $ARR[1]$. Now, $A[0]$ and $A[1]$ is sorted.
- In pass $N-1$, find the position POS of the smaller of the elements $ARR[N-2]$ and $ARR[N-1]$. Swap $ARR[POS]$ and $ARR[N-2]$ so that $ARR[0], ARR[1], \dots, ARR[N-1]$ is sorted.

Selection Sort

Example:

39	9	81	45	90	27	72	18
----	---	----	----	----	----	----	----

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

Selection Sort

SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]

Step 2: [INITIALIZE] SET POS = K

Step 3: Repeat for J = K+1 to N-1

 IF SMALL > ARR[J]

 SET SMALL = ARR[J]

 SET POS = J

 [END OF IF]

 [END OF LOOP]

Step 4: RETURN POS

SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
 to N-1

Step 2: CALL SMALLEST(ARR, K, N, POS)

Step 3: SWAP A[K] with ARR[POS]

 [END OF LOOP]

Step 4: EXIT

Complexity of Selection Sort

- Selection sort is a sorting algorithm that is independent of the original order of the elements in the array.
- In pass 1, selecting the element with smallest value calls for scanning all ***n*** elements; thus, *n*-1 comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position.
- In pass 2, selecting the second smallest value requires scanning the remaining ***n*** - 1 elements and so on. Therefore,
- $(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1) / 2 = \mathbf{O(n^2)}$ comparisons

Advantages of Selection Sort

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

However, in case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

Merge Sort

- Merge sort is a sorting algorithm that uses the divide, conquer and combine algorithmic paradigm.

Divide means partitioning the n -element array to be sorted into two sub-arrays of $n/2$ elements in each sub-array. (If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A_1 and A_2 , each containing about half of the elements of A).

Conquer means sorting the two sub-arrays recursively using merge sort.

Combine means merging the two sorted sub-arrays of size $n/2$ each to produce the sorted array of n elements.

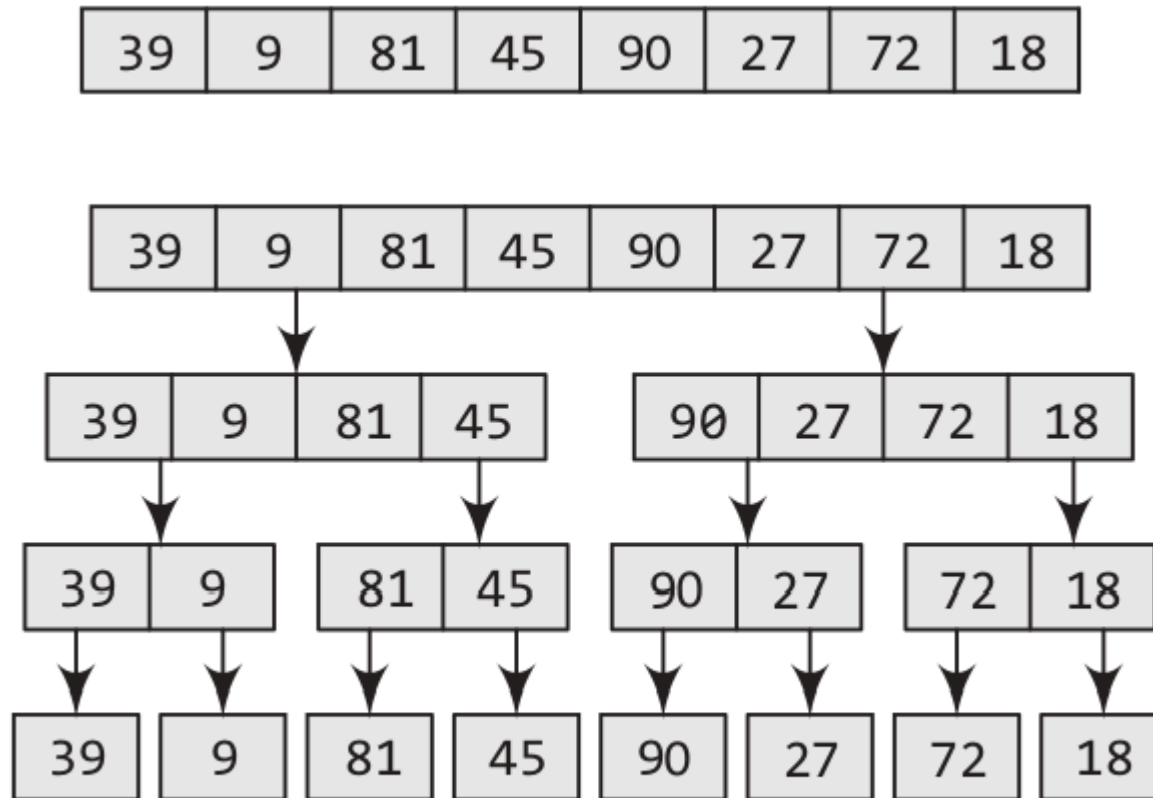
Merge Sort

Merge sort algorithms focuses on two main concepts to improve its performance (running time):

- A smaller list takes few steps and thus less time to sort than a large list.
- Less steps, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.
- The basic steps of a merge sort algorithm are as follows:
- If the array is of length 0 or 1, then it is already sorted. Otherwise:
- (Conceptually) divide the unsorted array into two sub- arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.

Merge Sort

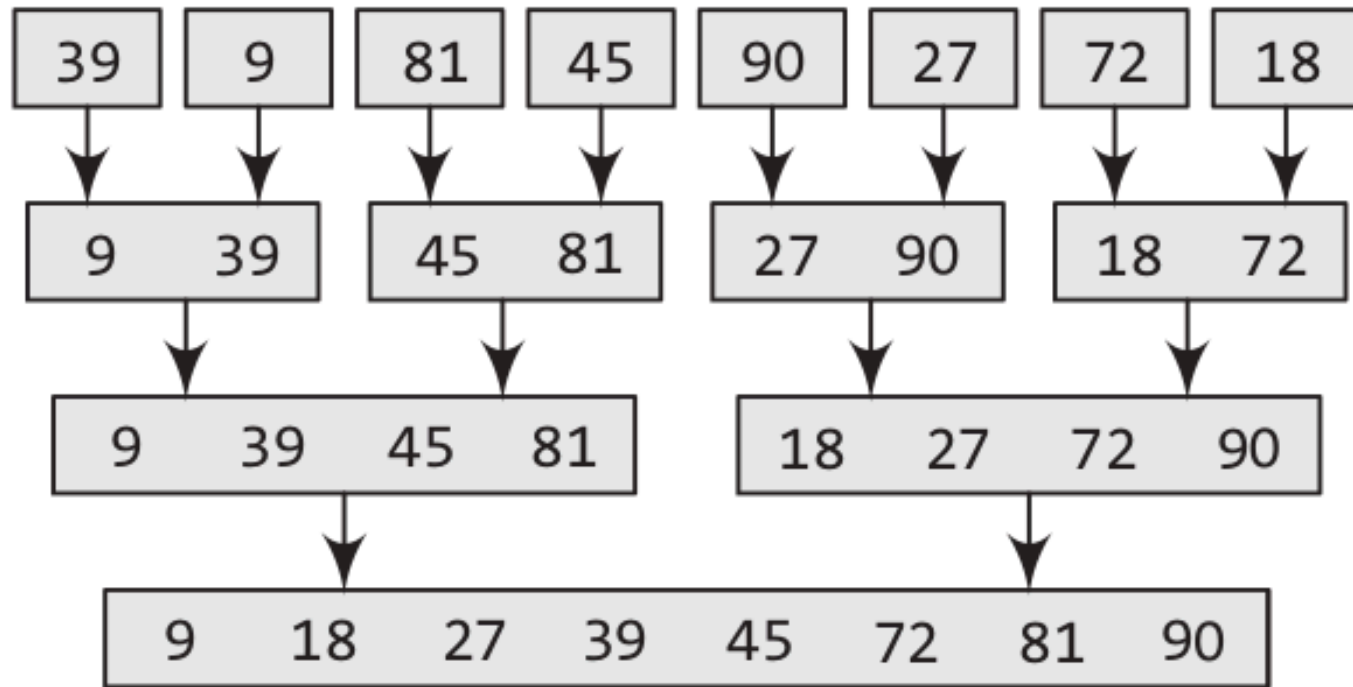
Example:



(Divide and Conquer the array)

Merge Sort

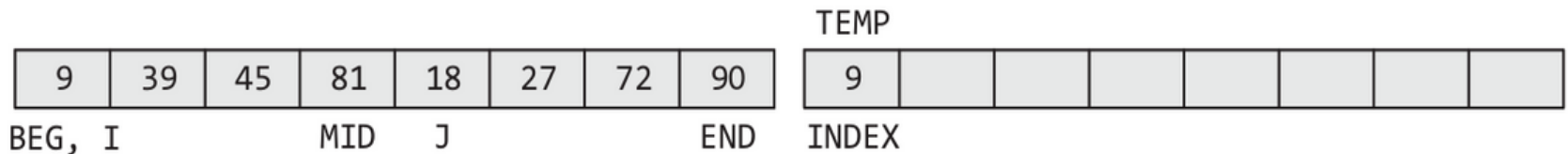
Example:



(Combine the elements to form a sorted array)

Merge Sort

- For the sake of understanding we have taken two sub-lists each containing four elements.
- The same concept can be utilized to merge 4 sub-lists containing two elements, and eight sub-lists having just one element.



Merge Sort

9	39	45	81	18	27	72	90	TEMP							
BEG	I		MID	J			END	9	18						
								INDEX							
9	39	45	81	18	27	72	90	9	18	27					
BEG	I		MID		J		END	INDEX							
9	39	45	81	18	27	72	90	9	18	27	39				
BEG	I		MID			J	END	INDEX							
9	39	45	81	18	27	72	90	9	18	27	39	45			
BEG		I	MID			J	END	INDEX							
9	39	45	81	18	27	72	90	9	18	27	39	45	72		
BEG			I, MID			J	END	INDEX							
9	39	45	81	18	27	72	90	9	18	27	39	45	72	81	
BEG			I, MID			J	END	INDEX							

When I is greater than MID, copy the remaining elements of the right sub-array in TEMP.

9	39	45	81	18	27	72	90	9	18	27	39	45	72	81	90
---	----	----	----	----	----	----	----	---	----	----	----	----	----	----	----

Merge Sort

```
MERGE_SORT(ARR, BEG, END)
```

```
Step 1: IF BEG < END
```

```
    SET MID = (BEG + END)/2
```

```
    CALL MERGE_SORT (ARR, BEG, MID)
```

```
    CALL MERGE_SORT (ARR, MID + 1, END)
```

```
    MERGE (ARR, BEG, MID, END)
```

```
    [END OF IF]
```

```
Step 2: END
```

Merge Sort

MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0

Step 2: Repeat while (I <= MID) AND (J<=END)

 IF ARR[I] < ARR[J]

 SET TEMP[INDEX] = ARR[I]

 SET I = I + 1

 ELSE

 SET TEMP[INDEX] = ARR[J]

 SET J = J + 1

 [END OF IF]

 SET INDEX = INDEX + 1

 [END OF LOOP]

Step 3: [Copy the remaining elements of right sub-array, if any]

 IF I > MID

 Repeat while J <= END

 SET TEMP[INDEX] = ARR[J]

 SET INDEX = INDEX + 1, SET J = J + 1

 [END OF LOOP]

 [Copy the remaining elements of left sub-array, if any]

 ELSE

 Repeat while I <= MID

 SET TEMP[INDEX] = ARR[I]

 SET INDEX = INDEX + 1, SET I = I + 1

 [END OF LOOP]

 [END OF IF]

Step 4: [Copy the contents of TEMP back to ARR] SET K=0

Step 5: Repeat while K < INDEX

 SET ARR[K] = TEMP[K]

 SET K = K + 1

 [END OF LOOP]

Step 6: END

Complexity of Merge Sort

- The running time of the merge sort algorithm in average case and worst case can be given as **$O(n \log n)$** .
- Although algorithm merge sort has an optimal time complexity but a major drawback of this algorithm is that it needs an additional space of $O(n)$ for the temporary array TEMP.

Quick Sort

- It works by using a divide-and-conquer strategy.
- The quick sort algorithm works as follows:
 - Select an element pivot from the array elements.
 - Re-arrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the **partition** operation.
 - Recursively sort the two sub-arrays thus obtained. (One with sub-list of lesser value than that of the pivot element and the other having higher value elements).

Quick Sort

Main task: find the pivot element, which will partition the array into two halves. To understand how we find the pivot element follow the steps given below. (we will take the first element in the array as pivot)

Quick sort works as follows:

1. Set the index of the first element in the array to **loc** and **left** variables. Also, set the index of the last element of the array to the **right** variable.

loc = 0, left = 0, and right = n-1 (where n is the no. of elements in the array).

2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc. **a[loc] should be less than a[right].**

(a) If that is the case, then simply continue comparing until right becomes equal to loc. Once $\text{right} = \text{loc}$, it means the pivot has been placed in its correct position.

(b) However, if at any point, we have **a[loc] > a[right]**, then interchange the two values and jump to Step 3.

(c) Set **loc = right.**

Quick Sort

3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by loc.

That is, **a[loc] should be greater than a[left]**.

(a) If that is the case, then simply continue comparing until left becomes equal to loc. Once $\text{left} = \text{loc}$, it means the pivot has been placed in its correct position.

(b) However, if at any point, we have **a[loc] < a[left]**, then interchange the two values and jump to Step 2.

(c) Set **loc = left**.

Quick Sort

Example:

27	10	36	18	25	45
----	----	----	----	----	----

We choose the first element as the pivot.
Set $loc = 0$, $left = 0$, and $right = 5$.

27	10	36	18	25	45
----	----	----	----	----	----

loc
 $left$ $right$



Scan from right to left. Since $a[loc] < a[right]$, decrease the value of $right$.

27	10	36	18	25	45
----	----	----	----	----	----

loc
 $left$ $right$



Start scanning from left to right. Since $a[loc] > a[left]$, increment the value of $left$.

25	10	36	18	27	45
----	----	----	----	----	----

$left$ $right$
 loc



Since $a[loc] > a[right]$, interchange the two values and set $loc = right$.

25	10	36	18	27	45
----	----	----	----	----	----

$left$ $right$
 loc

Quick Sort

Start scanning from left to right. Since $a[loc] > a[left]$, increment the value of left.

25	10	36	18	27	45
left			right		loc



Since $a[loc] < a[left]$, interchange the values and set $loc = left$.

25	10	27	18	36	45
left			right		loc



Scan from right to left. Since $a[loc] < a[right]$, decrement the value of right.

25	10	27	18	36	45
left			right	loc	



Since $a[loc] > a[right]$, interchange the two values and set $loc = right$.

25	10	18	27	36	45
left			right	loc	



Start scanning from left to right. Since $a[loc] > a[left]$, increment the value of left.

25	10	18	27	36	45
right			loc	left	

Quick Sort

QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)

 CALL PARTITION (ARR, BEG, END, LOC)

 CALL QUICKSORT(ARR, BEG, LOC - 1)

 CALL QUICKSORT(ARR, LOC + 1, END)

 [END OF IF]

Step 2: END

Quick Sort

PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0

Step 2: Repeat Steps 3 to 6 while FLAG = 0

Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT

 SET RIGHT = RIGHT - 1

 [END OF LOOP]

Step 4: IF LOC = RIGHT

 SET FLAG = 1

ELSE IF ARR[LOC] > ARR[RIGHT]

 SWAP ARR[LOC] with ARR[RIGHT]

 SET LOC = RIGHT

 [END OF IF]

Step 5: IF FLAG = 0

 Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT

 SET LEFT = LEFT + 1

 [END OF LOOP]

Step 6: IF LOC = LEFT

 SET FLAG = 1

ELSE IF ARR[LOC] < ARR[LEFT]

 SWAP ARR[LOC] with ARR[LEFT]

 SET LOC = LEFT

 [END OF IF]

 [END OF IF]

Step 7: [END OF LOOP]

Step 8: END

Complexity of Quick Sort

Average Case: Running time of quick sort can be given as **$O(n \log n)$** . The partitioning of the array which simply loops over the elements of the array once uses $O(n)$ time.

Best Case: Every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. At the most, only $\log n$ nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is $O(\log n)$. And because at each level, there can only be $O(n)$, the resultant time is given as **$O(n \log n)$** time. Practically, the efficiency of quick sort depends on the element which is chosen as the pivot.

Worst-case: Efficiency is given as **$O(n^2)$** . The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot.

Comparison of Sorting

ALGORITHM	Time COMPLEXITY			SPACE COMPLEXITY
	BEST CASE	AVERAGE CASE	WORST CASE	
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	$O(1)$

Comparison of Sorting

ALGORITHM	STABLE	IN-PLACE	ONLINE
Bubble sort	Yes	Yes	No
Insertion sort	Yes	Yes	Yes
Selection sort	No	Yes	No
Merge sort	Yes	No	No
Quick sort	No	Yes	No

Why Do We Need Searching and Sorting?

Scenario 1:

You want to find your friend's name in a contact list of 500 people.

→ Should you scroll one by one or search smartly?

Scenario 2:

You're buying a phone on Amazon. You sort by "Price: Low to High".

→ How does Amazon instantly sort thousands of results?

Scenario 3:

In a game leaderboard, players are ranked automatically.

→ What logic makes it happen?

Key Message:

Searching helps us find data.

Sorting helps us organize data.

Both are essential for making computers smart and efficient.


Why Do We Need Searching and Sorting?

Real-world Examples:

- Google Search (Search Algorithms)
- Online Shopping (Sorting by price, rating)
- Railway Reservation System (Finding available trains)
- YouTube (Search + trending list sorting)

Reference

This presentation is based on and adapted from lecture materials by Dr.Shweta Saharan

- Visualizations and animations from  Visualgo - An interactive visualizer for data structures and algorithms
- Content organized and customized by Abhinav Kesarwani, B.Tech AI&DS Student, GSV

*Thank
You*

