

# Sorting

# Types of sorting

- Internal Sorting: If the data to be sorted is small and can be kept in main memory and sorting is performed, then it is called as internal sorting.
- External Sorting: If the data is large which can't be placed in main memory at a time, then the data that is currently being sorted is brought in main memory and rest is on secondary memory. This type of sorting is external sorting.
- **We are discussing only Internal Sorting as part of our syllabus.**

# Classification of Sorting Algorithm

- **By Number of Comparisons:** In this method, sorting algorithms are classified based on the number of comparisons.
- For comparison based sorting algorithms, best case behavior is  $O(n \log n)$  and worst case behavior is  $O(n^2)$ .
- Comparison-based sorting algorithms evaluate the elements of the list by key comparison operation and need at least  $O(n \log n)$  comparisons for most inputs.

- **By Number of Swaps:** In this method, sorting algorithms are categorized by the number of swaps.
- **By Memory Usage:** Some sorting algorithms are such, that they need  $O(1)$  or  $O(\log n)$  memory to create auxiliary locations for sorting the data temporarily.
- **By Recursion:** Sorting algorithms are either recursive [quick sort] or non-recursive [selection sort, and insertion sort], and there are some algorithms which use both (merge sort).

- **By Stability:** Sorting algorithm is stable if the relative order of original list is maintained in case of duplicate key values.
- **By Adaptability:** With a few sorting algorithms, the complexity changes based on pre-sortedness [quick sort]: pre-sortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive

# Bubble-Sort

- Bubble-Sort is the slowest algorithm for sorting, but it is heavily used, as it is easy to implement.
- In Bubble-Sort, we compare each pair of adjacent values.
- We want to sort values in increasing order so if the second value is less than the first value then we swap these two values. Otherwise, we will go to the next pair.
- We will have  $N$  number of passes to get the array completely sorted. After the first pass, the largest value will be in the rightmost position.

# Algorithm – Bubble Sort

- Traverse the array elements from left to right.
- Compare each element with its adjacent right.
- Swap the elements if left element is greater than right element.
- The largest element is moved to the rightmost end at first pass.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

Pass - 1

← swap →

Index	0	1	2	3	4	5
Values	40	20	50	60	30	10

← ← swap →

Index	0	1	2	3	4	5
Values	20	40	50	60	30	10

← swap →

Index	0	1	2	3	4	5
Values	20	40	50	30	60	10

Index	0	1	2	3	4	5
Values	20	40	50	30	10	60



## Pass - 2

← swap →

Index	0	1	2	3	4	<u>5</u>
Values	20	40	50	30	10	<u>60</u>

← swap →

Index	0	1	2	3	4	<u>5</u>
Values	20	40	30	50	10	<u>60</u>

Index	0	1	2	3	4	<u>5</u>
Values	20	40	30	10	50	<u>60</u>

### Pass - 3

← swap →

Index	0	1	2	3	<u>4</u>	<u>5</u>
Values	20	40	30	10	<u>50</u>	<u>60</u>

← swap →

Index	0	1	2	3	<u>4</u>	<u>5</u>
Values	20	30	40	10	<u>50</u>	<u>60</u>

Index	0	1	2	3	<u>4</u>	<u>5</u>
Values	20	30	10	40	<u>50</u>	<u>60</u>

## Pass - 4

← swap →

Index	0	1	2	<u>3</u>	<u>4</u>	<u>5</u>
Values	20	30	10	<u>40</u>	<u>50</u>	<u>60</u>

Index	0	1	2	<u>3</u>	<u>4</u>	<u>5</u>
Values	20	10	30	<u>40</u>	<u>50</u>	<u>60</u>

## Pass - 5

← swap →

Index	0	1	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>
Values	20	10	<u>30</u>	<u>40</u>	<u>50</u>	<u>60</u>

Index	<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>
Values	<u>10</u>	<u>20</u>	<u>30</u>	<u>40</u>	<u>50</u>	<u>60</u>

Tip: To improve the performance, we can keep a flag to check if swapping is needed or not. We can terminate the loop of swapping is not needed.

## **Bubble Sort: Performance**

Worst case complexity :  $O(n^2)$

Best case complexity (Improved version) :  $O(n)$

Average case complexity (Basic version) :  $O(n^2)$

Worst case space complexity :  $O(1)$  auxiliary

# Selection-Sort


- Selection sort is an in-place sorting algorithm.
- Selection sort works well for small files. It is used for sorting the files with very large values and small keys.
- Here selection is made based on keys and swaps are made only when required.
- Advantages • Easy to implement • In-place sort (requires no additional storage space)
- Disadvantages • Time complexity is  $O(n^2)$

# Algorithm – Selection Sort

1. Traverse the entire array, find the minimum value in the list.
2. Swap it with the value in the current position (index)
3. Repeat this process for all the elements until the entire array is sorted This algorithm is called selection sort since it repeatedly selects the smallest element.

Pass – 1

swap

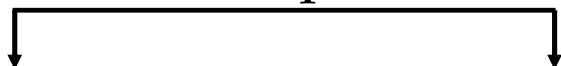


A horizontal arrow labeled 'swap' connects the top of the column for index 0 to the top of the column for index 3.

Index	0	1	2	3	4	5	6	7	8
Value	82	42	48	8	25	52	36	93	59

Pass – 2

swap



A horizontal arrow labeled 'swap' connects the top of the column for index 1 to the top of the column for index 4.

Index	0	1	2	3	4	5	6	7	8
Value	8	42	48	82	25	52	36	93	59



Pass – 3

swap



Index	0	1	2	3	4	5	6	7	8
Value	8	25	48	82	42	52	36	93	59

Pass – 4

swap



Index	0	1	2	3	4	5	6	7	8
Value	8	25	36	82	42	52	48	93	59

Pass – 5

swap

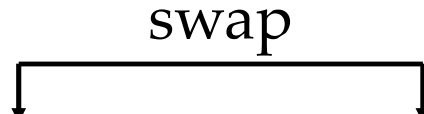


Index	0	1	2	3	4	5	6	7	8
Value	8	25	36	42	82	52	48	93	59

Pass – 6

Index	0	1	2	3	4	5	6	7	8
Value	8	25	36	42	48	52	82	93	59

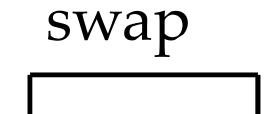
Pass – 7



A diagram showing a swap operation between index 6 and index 8. A horizontal line with downward-pointing arrows at both ends connects the two indices. The word "swap" is written above the line.

Index	0	1	2	3	4	5	6	7	8
Value	8	25	36	42	48	52	82	93	59

Pass – 8



A diagram showing a swap operation between index 7 and index 8. A horizontal line with downward-pointing arrows at both ends connects the two indices. The word "swap" is written above the line.

Index	0	1	2	3	4	5	6	7	8
Value	8	25	36	42	48	52	59	93	82

## Result

Index	0	1	2	3	4	5	6	7	8
Value	8	25	36	42	48	52	59	82	93

Performance of Selection Sort :

- Worst case complexity :  $O(n^2)$
- Best case complexity :  $O(n^2)$
- Average case complexity :  $O(n^2)$
- Worst case space complexity:  $O(1)$  auxiliar

# Insertion Sort

- It is a simple and efficient comparison sort.
- Each iteration removes an element from the input data and inserts it into the correct position in the list being sorted.
- The choice of the element being removed from the input is random and this process is repeated until all input elements have gone through.

## Advantages

- Simple implementation
- Efficient for small data
- Adaptive: If the input list is presorted [may not be completely] then insertions sort takes  $O(n + d)$ , where  $d$  is the number of inversions
- Practically more efficient than selection and bubble sorts, even though all of them have  $O(n^2)$  worst case complexity
- Stable: Maintains relative order of input data if the keys are same
- In-place: It requires only a constant amount  $O(1)$  of additional memory space

# Algorithm – Insertion Sort

- Start with second element of the array, compare it with the first element and if the second element is smaller then swap them.
- Move to the third element and compare it with the second element, then the first element and swap as necessary to insert it in the correct position among the first three elements.
- Continue this process, comparing each element with the all elements before it and swap to place it in the correct position among the sorted elements.
- Repeat until the entire array is sorted.

Index	0	1	2	3	4	5
Values	50	34	2	14	90	22

Index	0	1	2	3	4	5
Values	34	50	2	14	90	22

Index	0	1	2	3	4	5
Values	2	34	50	14	90	22



Index	0	1	2	3	4	5
Values	2	14	34	50	90	22

Index	0	1	2	3	4	5
Values	2	14	34	50	90	22

Index	0	1	2	3	4	5
Values	2	14	22	34	50	90

# Performance of insertion sort

- Worst case complexity :  $O(n^2)$
- Best case complexity :  $O(n^2)$
- Average case complexity :  $O(n^2)$
- Worst case space complexity:  $O(n^2)$   $O(1)$  auxiliary