**Bubble Sort**

**Bubble sort** is a sorting algorithm that iteratively compares two adjacent elements and swaps them if they are not in desired order. At the end of each iteration, an element of the array moves to the desired sorted position in the array. This way list gets sorted from the one end right to the other.

The algorithm works and just like the movement of air bubbles in the water that rise up to the surface. Therefore, it is called a bubble sort.

45

-9

11

-2

0

-9

45

11

0

-2

0

-2

45

11

-9

45

-2

0

11

-2

45

0

11

-9

0

-2

0

-9

11

45

-2

0

-9

11

45

-2

-9

0

11

-9

Iteration 0

45

Iteration 1

45

-9

11

-2

-9

11

-2

0

45

-9

11

-2

0

-2

0

-9

45

Iteration 2

-9

11

**Time Complexities**

**Worst Case Complexity: O(n2)** If we want to sort in ascending order and the array is in descending order then the worst case occurs.

Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.  
  
Thus, the total number of comparisons = n\*(n-1) ~ n2

**Best Case Complexity: O(n)** If the array is already sorted, then there is no need for sorting.

**Average Case Complexity: O(n2)** It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

**Space Complexity**

**Space complexity: O(1**) if an extra variable is used for swapping (minimal optimized space complexity). **Space complexity: O(2)** for some poor implementations that need two extra variables.

**Selection Sort**

**Selection sort** is a sorting that selects the smallest / biggest (depending upon sorted order) element from an unsorted sub-list in each iteration and places that element at the beginning of the sub-list list.

New sub-list is selected at each iteration and smallest / biggest element is selected from the sub-list, which ultimately goes to relevant sorted position in the mother list.

Iteration 2

Iteration 1

-9

-2

0

11

45

0

-9

11

0

-2

45

0

Iteration 0

-9

11

45

-2

11

0

0

45

-2

11

0

45

-9

sublist

sublist

sublist

45

11

0

-2

min element indx : 0

min element indx : 3

min element indx : 4

-2

-9

Updated list

Updated list

45

0

11

-2

-9

Updated list

45

0

-9

-2

11

-2

0

-9

45

11

**Time Complexities**

The time complexity of the selection sort is the same in all cases. At every step, you have to find the minimum element and put it in the right place. The minimum element is not known until the end of the array is not reached.

**Worst Case Complexity: O(n2)** If we want to sort in ascending order and the array is in descending order then the worst case occurs.

Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.  
  
Thus, the total number of comparisons = n\*(n-1) ~ n2

**Best Case Complexity: O(n2)** If the array is already sorted, still same number of swaps are attempted

**Average Case Complexity: O(n2)** It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

**Space Complexity**

**Space complexity: O(1**) if an extra variable is used for minimum index.

**Insertion Sort**

**Insertion sort** is a sorting algorithm that divides the array in two sections: sorted and unsorted. In each iteration, new element is inserted in sorted section from unsorted section. Inside the sorted section, the new element has to find its optimal position to keep the sorted section sorted.

Insertion sort works similarly as we sort cards in our hand in a card game. We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

Iteration 2

Iteration 1

Iteration 0

11

-9

45

11

0

-2

45

0

-2

-9

11

45

0

-2

0

-9

11

45

0

-2

45

-2

-9

11

0

45

-2

45

-2

11

0

45

-2

-2

-9

11

0

45

-2

sublist

sublist

sublist

next element

next element

next element

Updated list

Updated list

Updated list

**Time Complexities**

**Worst Case Complexity: O(n2)** Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.  
  
Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made. Thus, the total number of comparisons = n\*(n-1) ~ n2

**Best Case Complexity: O(n)** When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.

**Average Case Complexity: O(n2)** It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

**Space Complexity**

**Space complexity: O(1**) if an extra variable is used for minimum index.

**Merge Sort**

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm. In Divide and Conquer Principle, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.

For sorting in each iteration, merge sort divides the input array into two equal subarrays, calls itself recursively for the two subarrays, sorts the subarrays and finally merges the two sorted halves.

-2

45

0

11

-9

-9

11

0

45

-2

-9

11

0

45

-2

11

-9

45

-2

45

-2

45

0

-2

-9

-2

0

11

45

**Time Complexities**

No matter what is the case, array is pre-sorted or really jumbled same number of steps will always be performed.

**Worst Case Complexity: O(n\*log n)**

**Best Case Complexity: O(n\*log n)**

**Average Case Complexity: O(n\*log n)**

**Space Complexity**

**Space complexity: O(n**) because an extra array is used to produce sorted array

**Quick Sort**

Another Divide and conquer algorithm. The intuitive idea behind quick sort is it picks an element as the pivot from a given array of elements and then partitions the array around the **pivot element**.

The array is partitioned in a fashion such that all elements less than the pivot are in the left subarray while all elements strictly greater than the pivot element are stored in the right subarray. Quicksort function is invoked again for the two subarrays created above and steps are repeated.

90

10

30

40

10

30

40

500

**Partition around 50**

**Partition around 80**

90

80

700

50

40

90

30

80

10

**Partition around 70**

**Partition around 40**

30

10

**Partition around 30**

10

**Partition Logic**

**Repeat as long as i <= j**

Move in this direction long as arr[ i ] < pivot

**i**

700

50

40

90

30

80

10

Move in this direction long as arr[ j ] > pivot

**j**

**pivot**

**Once the movement of i and j stops, swap arr[ i ] and arr[ j ]**

**Swap arr[ i ] and arr[ pivot ]**

**return i**

**Time Complexities**

**Worst Case Complexity: O(n2)**  It occurs when the pivot element picked is either the greatest or the smallest element. This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 elements. Thus, quicksort is called only on this sub-array.

However, the quicksort algorithm has better performance for scattered pivots.  
  
Thus, the total number of worst comparisons = n\*(n-1) ~ n2

**Best Case Complexity: O(n\*log n)** It occurs when the pivot element is always the middle element or near to the middle element.

**Average Case Complexity: O(n\*log n)** It occurs when pivot element is not extreme ends of sorted array plus it's not even from the middle of sorted array

**Space Complexity**

**Space complexity: O(log n)** for in-place replacement logic

**Counting Sort**

**Counting sort** is a sorting algorithm that sorts the elements of an integer array containing multiple occurrences of same elements. Sorting is done by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

Assuming array is of length n and consists of k unique elements.

If array is of positive integers, auxiliary array could be array where index represents main array's element and auxiliary element is number of occurrences of that main array's element. This auxiliary array could later be concerted to "cumulative" count array. Which is ultimately used for sorting.

5

1

3

0

1

3

1

0

3

4

4

5

0

1

2

3

index

5

1

1

2

3

0

3

count

max

In case of negative integer array, auxiliary array relations could be more challenging as now index could NOT represent main array's element anymore.

2

1

-3

5

1

3

-1

0

3

4

-3

5

3

4

5

Index for

for

1

2

-3

-2

-1

0

max

min

2

1

1

2

1

1

0

1

1

count

**Time Complexities**

Assuming array is of length **n** and consists of **k** unique elements.

**Worst Case Complexity: O(n + k)**

**Best Case Complexity: O(n + k)**

**Average Case Complexity: O(n + k)**

**Space Complexity**

**Space complexity:** Space complexity is **O(n + k)** or **O(max)** for in-place replacement logic.

**Radix Sort**

Radix sort is a sorting algorithm that does sorting without actual element level comparison. Instead it is a bucket-based sorting (scatter and gather approach) by comparing face values of unit’s, 10’s or 100’s place digits.

400

Place values

Face values

4

2

1

20

1

4

2

1

least to most significant

This type of sorting can be used for array of positive integers, array can have mixed element set containing 1 to n digit integers. This logic can also be used for alphabetical sorting.

Radix Sort distributes the elements into buckets based on each digit’s value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, (or most significant to the least significant for strings) Radix Sort achieves the final sorted order.

170

45

75

90

802

24

02

66

075

170

090

066

045

024

002

802

002

024

045

066

045

090

170

802

Sorting based on 10s digit

Sorting based on 100s digit

170

45

75

90

802

24

2

66

Sorting based on unit digit

**Time Complexity O(d \* (n + b))**

**Space Complexity O(n + b)**

d is the number of digits,

n is the number of elements

b is the base of the number system being used

**Bucket Sort**

Bucket sort is an algorithm that again uses scatter and gather sorting approach.

Where a bucket structure is created based on range of array elements. Then array elements are conditionally distributed into their bucket compartments. Then elements inside each bucket are sorted using basic sorting logic. Then the whole bucket structure is merged back to form sorted array.

12

24

1

13

19

17

8

21

9

11



Gather

**21 24**

**21 24**

Sort

Scatter

20-25

15-20

10-15

5-10

0-5

1

8

9

11

12

13

17

19

21

24

**17 19**

**11 12 13**

**8 9**

**1**

**11 13 12**

**17 19**

**9 8**

**1**

0-5

5-10

20-25

10-15

15-20

**Time Complexities**

**Worst Case Complexity:** When array elements are not well distributed, some buckets may end up having more elements than others. Worst case is one bucket having all elements.

In addition to poor distribution, if elements are in reverse order to sorted order, worst case happens. Then the worst-case complexity depends upon which algorithm is used.

Worst case for quick sort **O(n2)**

Worst case for merge sort **O(n \* log n)**

**Best Case Complexity:** It occurs when the elements are uniformly distributed in the buckets with a nearly equal number of elements in each bucket. Let that number be k.

The complexity becomes even better if the elements inside the buckets are already sorted.

**O(n)** is the complexity for making the buckets and **O(k)** is the complexity for sorting the elements of the bucket using algorithms having linear time complexity at the best case. Best case complexity **O(n + k)**

**Average Case Complexity: O(n + k)** It occurs when the elements of the array are in jumbled order and well distributed. Not the best case, but not even the case when majority of elements are in one bucket.

**Space Complexity**

**Space complexity: O( n + k**) where n is the number of elements and k is the number of buckets.