**Bubble Sort**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

**Example:**  
**First Pass:**  
( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, the algorithm compares the first two elements and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**  
( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**  
( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** )

#### ****Bubble Sort Complexity****

Bubble sort is a simple sorting algorithm.  This sorting algorithm is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

This algorithm is not suitable for large data sets as its average and worst-case complexity are of Ο(n2) where **n** is the number of items.

#### ****Selection Sort****

The selection sort algorithm sorts an array by repeatedly finding the minimum element considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

1. The subarray is already sorted.
2. The remaining subarray is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

#### ****Selection Sort Complexity****

Selection sort is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end.  Initially, the sorted part is empty and the unsorted part is the entire list.  The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array.  This process continues moving unsorted array boundary by one element to the right. This algorithm is not suitable for large data sets as it is average and worst-case complexities are of Ο(n2), where **n** is the number of items.

#### ****Insertion Sort****

Insertion sort is an in-place comparison-based sorting algorithm.  Here, a sub-list is maintained which is always sorted.  For example, the lower part of an array is maintained to be sorted.  An element that is to be inserted in this sorted sub-list has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list in the same array.

#### ****Insertion Sort Complexity****

This algorithm is not suitable for large data sets as its average and worst-case complexity are of Ο(n2), where n is the number of items

#### ****Shell Sort****

Shell sort is a highly efficient sorting algorithm and is based on the insertion sort algorithm.  This algorithm avoids large shifts as in the case of insertion sort if the smaller value is to the far right and has to be moved to the far left.  It uses insertion sort on widely spread elements, first to sort them and then sorts the less widely spaced elements. The spacing is termed as an **interval**.  The interval is calculated based on Knuth's formula as −



#### ****Shell Sort Complexity****

This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is Ο(n), where n is the number of items. And the worst-case space complexity is O(n).

#### ****Heap Sort****

**Heapsort** is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort.  Like selection sort, heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a heap data structure to more quickly find the largest element in each step.

#### ****Heap Sort Complexity****

Although somewhat slower in practice on most machines than quicksort, it has the advantage of a more favorable worst-case O(n log n) runtime.

#### ****How Heap Sort Works****

Supposed { 6, 5, 3, 1, 8, 7, 2, 4 } are the elements of a list.  In building the heap, larger nodes don't stay below smaller node parents. They are swapped with parents, and then recursively checked if another swap is needed, to keep larger numbers above smaller numbers on the heap binary tree.

#### ****Merge Sort****

Merge Sort is a divide and conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.  It uses a merge function to merge two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

#### ****Merge Sort Complexity****

The time complexity of Merge Sort is O(nLogn) in all 3 cases; worst, average, and best as merge sort always divides the array into two halves and takes linear time to merge two halves.

#### ****How the Merge Sort Works****

Merge sort organize array elements by recursively dividing the set into two halves until the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.

**Quick Sort**

Quicksort is a divide and conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick the first element as a pivot.
2. Always pick the last element as the pivot (implemented below)
3. Pick a random element as a pivot.
4. Pick median as a pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

**Quick Sort Complexity**

*Worst Case:* The worst case occurs when the partition process always picks the greatest or smallest element as pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order.

T(n) = T(0) + T(n-1) + (n) which is equivalent to T(n) = T(n-1) + (n)

= (n2).

**Best Case:** The best case occurs when the partition process always picks the middle element as a pivot. Following is recurrence for the best case.

T(n) = 2T(n/2) + (n)

= (nLogn)

**Average Case:**  
We can get an idea of average case by considering the case when partition, for instance,  O(n/9) elements in one set and O(9n/10) elements in another set.

T(n) = T(n/9) + T(9n/10) + (n)

= O(nLogn)

#### ****Bin/Bucket Sort****

Bin, also referred to as Bucket Sort runs in linear time on average. Like Counting Sort, bucket Sort is fast because it considers something about the input. Bucket Sort considers that the input is generated by a random process that distributes elements uniformly over the intervalμ=[0,1].

**Bin/Bucket Sort Algorithm**

1. Partition μ into n non-overlapping intervals called buckets.
2. Puts each input number into its buckets
3. Sort each bucket using a simple algorithm, e.g. Insertion Sort and then
4. Concatenate the sorted lists.

 Bucket Sort considers that the input is an n element array A and that each element A [i] in the array satisfies 0≤A [i] <1.

The code depends upon an auxiliary array B [0....n-1] of linked lists (buckets) and considers that there is a mechanism for maintaining such lists.

#### ****Radix Sort****

Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order.  In the radix sort algorithm, a list of integer numbers is sorted based on the digits of individual numbers.  Sorting is performed from the least significant digit to the most significant digit.

Radix sort algorithm requires the number of passes that are equal to the number of digits present in the largest number among the list of numbers.

#### ****Queue****

 A queue is an n ordered list in which all insertions take place at one end, the**rear**, while all deletions take place at the other end, the **front**.  It is an example of a linear data structure.

A queue has a **First-In, First-Out (FIFO)**structure where elements can only be added to the rear of the queue and removed from the front of the queue.  It has two main operations, **enqueue** for insertion and **dequeue** for deletion.

#### ****Queue Restrictions****

The restrictions on queue imply that the first element which is inserted into the queue will be the first one to be removed.  Thus A is the first letter to be removed, and queues are known as **First In First Out (FIFO)** lists.

A **LinkedList** is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers.  A linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

There are three types of LinkedList:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

A **Singly Linked List** is the most common form of a Linked List where each node contains a data field and a **single** pointer to the next node in the list.

The reference to the **first** node in the list is called the **HEAD** of the list. The pointer/reference/link field contained in the node is used to traverse to the next node and to its next node and so on till we reach a node that points to NULL. This is the **last** node in the list.  Also, a singly linked list can only be traversed in one and only one direction i.e. from head to the last node. There is no way to traverse from the last node back to the head. The following is an example of a singly linked list with 5 nodes.

**Inserting Nodes Into a Singly Linked List**

There are three cases to consider for inserting a node into a singly linked list. Adding a node to the :

* Beginning of the list.
* End of the list.
* Specified position in the list.

**Inserting Nodes at the Beginning of the List**

To insert a new node at the beginning of the list the following algorithm is used :

* Assign the reference of the HEAD to the new node’s next field.
* Make the new node as the HEAD of the list.

**Inserting a Node at the End of the List**

To insert a node at the end of the list the following algorithm is used.

* Traverse the list until we find the last node.
* The new node’s reference is assigned to the last node’s next field.

**Inserting a Node at a Specified Position in the List**

To insert a node at a specified position in the list the following algorithm is used.

* Traverse (position – 1) times or till the end of the list is reached and maintain previous and current references.
* Assign the reference of the new node to the prev node’s next field.
* Assign the cur node’s reference to the new node’s next field.

**Deleting Nodes From a Singly Linked List**

Deleting a node from a singly linked list can be a little complex as the node to be deleted can be the first node, the last node, or a node in the middle of the list. Let us discuss each case.

* **First Node:** If the node to be deleted is the first node itself, we assign the reference of the next of the HEAD node to the HEAD node.
* **Last Node or Any Other Node:** To delete any other node in the list, we traverse the list keeping track of the previous and current nodes in the list until we find the node to be deleted with the required data field or we reach the end of the list i.e. NULL without finding the data element in the list.

If the node is found, we assign the reference of the next field of the current node to the previous node’s next.

#### ****Doubly Linked List****

A **Doubly Linked List** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes.

**Difference Between Singly Linked List and Doubly Linked List**

Both types of lists contain a pointer to the next node, as well as a data field to represent the actual value stored in the node.  The only difference between Doubly LinkedList and SinglyLinkedList is that the Doubly LinkedList also contains a pointer to the previous node, not just the next node.

A Doubly LinkedList must contain three variables:

* data variable
* next node variable
* previous node variable

#### ****Circular Linked List****

A**Circular Linked List** is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.