**SORTING ALGORITHMS**

**Bubble sort:**

In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting. Consider an array A of n elements whose elements are to be sorted by using Bubble sort.

1. In Pass 1, A[0] is compared with A[1], A[1] is compared with A[2], A[2] is compared with A[3] and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.
2. In Pass 2, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list.
3. In pass n-1, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of this pass. The smallest element of the list is placed at the first index of the list.

|  |  |
| --- | --- |
| Space Complexity | O(1) |
| Worst case running time | O(n2) |
| Average case running time | O(n) |
| Best case running time | O(n2) |

**Selection sort:**

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

Consider an array A whose elements are to be sorted.

Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k-1], A[k-2], and so on until we find an element A[j] such that, A[j]<=A[k].

All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best Case** | **Average Case** | **Worst Case** |
| Time | Ω(n) | θ(n2) | o(n2) |
| Space |  |  | o(1) |

**Quick Sort:**

Quick sort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting of an array of n elements. This algorithm follows divide and conquer approach.

1. Set the first index of the array to left and loc variable. Set the last index of the array to right variable. i.e.

left = 0, loc = 0, en d = n - 1, where n is the length of the array.

1. Start from the right of the array and scan the complete array from right to beginning comparing each element of the array with the element pointed by loc.

Ensure that, a[loc] is less than a[right].

* 1. If this is the case, then continue with the comparison until right becomes equal to the loc.
  2. If a[loc] > a[right], then swap the two values. And go to step 3.
  3. Set, loc = right

1. start from element pointed by left and compare each element in its way with the element pointed by the variable loc. Ensure that a[loc] > a[left]
   1. if this is the case, then continue with the comparison until loc becomes equal to left.
   2. [loc] < a[right], then swap the two values and go to step 2.
   3. Set, loc = left.

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best Case** | **Average Case** | **Worst Case** |
| Time Complexity | O(n) for 3 way partition or O(n log n) simple partition | O(n log n) | O(n2) |
| Space Complexity |  |  | O(log n) |

**Merge sort:**

Merge sort is the algorithm which follows divide and conquer approach. Consider an array A of n number of elements.

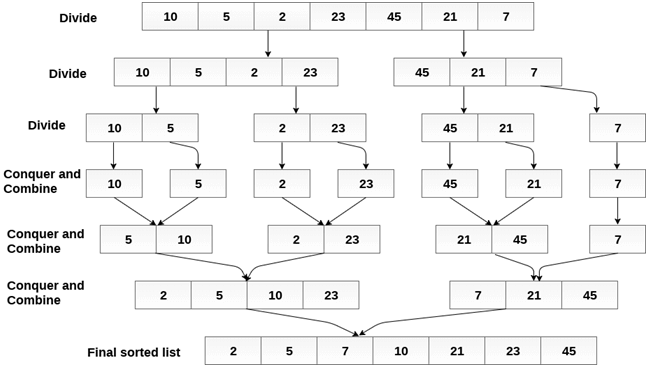
1. If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
2. Conquer means sort the two sub-arrays recursively using the merge sort.
3. Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

The main idea behind merge sort is that, the short list takes less time to be sorted.

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best case** | **Average Case** | **Worst Case** |
| Time Complexity | O(n log n) | O(n log n) | O(n log n) |
| Space Complexity |  |  | O(n) |

Consider the following array of 7 elements. Sort the array by using merge sort.

A = {10, 5, 2, 23, 45, 21, 7}



Selection Sort:

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.

The array with n elements is sorted by using n-1 pass of selection sort algorithm.

* In 1st pass, smallest element of the array is to be found along with its index **pos**. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
* In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
* In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].

**Example:**

**A = {10, 2, 3, 90, 43, 56}.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Pass** | **Pos** | **A[0]** | **A[1]** | **A[2]** | **A[3]** | **A[4]** | **A[5]** |
| 1 | 1 | 2 | 10 | 3 | 90 | 43 | 56 |
| 2 | 2 | 2 | 3 | 10 | 90 | 43 | 56 |
| 3 | 2 | 2 | 3 | 10 | 90 | 43 | 56 |
| 4 | 4 | 2 | 3 | 10 | 43 | 90 | 56 |
| 5 | 5 | **2** | **3** | **10** | **43** | **56** | **90** |

Sorted A = {2, 3, 10, 43, 56, 90}

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best Case** | **Average Case** | **Worst Case** |
| Time | Ω(n) | θ(n2) | o(n2) |
| Space |  |  | o(1) |

**Heap Sort:**

Heap sort processes the elements by creating the min heap or max heap using the elements of the given array. Min heap or max heap represents the ordering of the array in which root element represents the minimum or maximum element of the array. At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.

The heap sort basically recursively performs two main operations.

* Build a heap H, using the elements of Array.
* Repeatedly delete the root element of the heap formed in phase 1.

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best Case** | **Average Case** | **Worst case** |
| Time Complexity | Ω(n log (n)) | θ(n log (n)) | O(n log (n)) |
| Space Complexity |  |  | O(1) |