**Table of Complexity Comparison:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Best Case** | **Average Case** | **Worst Case** | **Memory** | **Stable** | **Method Used** |
| [Quick Sort](http://www.geeksforgeeks.org/quick-sort/) | nlogn | nlogn | n^2 | Logn | No | Partitioning |
| [Merge Sort](http://www.geeksforgeeks.org/merge-sort/) | nlogn | nlogn | nlogn | n | Yes | Merging |
| [Heap Sort](https://www.geeksforgeeks.org/heap-sort/) | nlogn | nlogn | nlogn | 1 | No | Selection |
| [Insertion Sort](http://www.geeksforgeeks.org/insertion-sort/) | n | n^2 | n^2 | 1 | Yes | Insertion |
| [Selection Sort](http://www.geeksforgeeks.org/selection-sort/) | n^2 | n^2 | n^2 | 1 | No | Selection |
| [Bubble Sort](http://www.geeksforgeeks.org/bubble-sort/) | n | n^2 | n^2 | 1 | Yes | Exchanging |

**Selection Sort**

* Finding the **smallest/largest element** and then placing it at the right position by **swapping**.
* **Maximum number of swaps O(n)**, therefore **useful when memory writing is costly**.
* Works well with **small datasets** (and not on large datasets).
* Does not preserve the relative order of elements with equal keys which means it is **not stable**. But it can be made stable.
* As it does not require extra space, it is an **in-place** sorting algorithm.

**Stable Selection Sort**

* Input: 4a 5 3 2 4b 1
* Output (selection sort): 1 2 3 4b 4a 5
* (Stable Selection Sort): 1 2 3 4a 4b 5
* Selection sort can be made Stable if instead of swapping, the minimum element is placed in its position without swapping i.e. by placing the number in its position by pushing every element one step forward(shift all elements to left by 1).
* In simple terms use a technique like [insertion sort](https://www.geeksforgeeks.org/insertion-sort/) which means inserting element in its correct place.
* **Time Complexity: O(n2)**
* **Auxiliary Space: O(1)**

**Bubble Sort**

* **Swapping adjacent elements**
* **Not suitable for large datasets** due to large time complexity
* Time Complexity: **O(n2)**
* **Stable Sorting Algorithm**, elements with same key value maintain their relative order.
* **comparison-based sorting algorithm**, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.
* The **best case** occurs **when the array is already sorted**. So the **number of comparisons required is N-1 and the number of swaps required = 0**. Hence the **best case complexity is O(N).**
* The **worst-case** condition for bubble sort occurs when elements of the **array are arranged in decreasing order.**In the worst case, **the total number of iterations or passes required to sort a given array is (N-1).** where ‘N’ is the number of elements present in the array.
* Average Case Time Complexity: The number of comparisons is constant in Bubble Sort. So **in average case, there are O(N2) comparisons.** This is because irrespective of the arrangement of elements, the number of comparisons C(N) is same.
* What is the Boundary Case for Bubble sort?
* Bubble sort takes minimum time (Order of n) when elements are already sorted. Hence it is best to check if the array is already sorted or not beforehand, to avoid O(N2) time complexity.
* What is the maximum number of comparisons that can take place when a bubble sort algorithm is implemented?, suppose there are n elements in the array.
* (n\*(n-1))/2
* In computer graphics, it is popular for its capability to detect a tiny error (like a swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n).
* Example: It is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to the x-axis), and with incrementing y their order changes (two elements are swapped) only at intersections of two lines

**Recursive Bubble Sort**

* In bubble sort, in first pass, we move largest element to end. In second pass, we move second largest element to second last position and so on.
* **Recursion Idea**
* **Base Case: If array size is 1, return.**
* **Do One Pass of normal Bubble Sort. This pass fixes last element of current subarray.**
* **Recur for all elements except last of current subarray.**
* Time Complexity: **O(n\*n)**
* Auxiliary Space: **O(n)**
* **Recursive bubble sort** runs on **O(n) auxiliary space** complexity whereas **iterative bubble sort** runs on **O(1) auxiliary space** complexity.
* **Based on the number of comparisons** in each method, the **recursive bubble sort is better than the iterative bubble sort**, but the **time complexity** for both the methods is **same**.

Insertion Sort

Partially sorted best

Inplace

Stable

Online

**Merge Sort**

* **recursive algorithm** that continuously **splits the array in half until it cannot be further divided** i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.
* **Time Complexity:** **O(N log(N))**

**θ(Nlog(N))** in all cases best, average and worst as merge sort always divides the array into two halves and takes linear time to merge two halves.

* **Recurrence Relation** can be solved using Recurrence Tree or the Master Method

**T(n) = 2T(n/2) + θ(n)**

* **Auxiliary Space: O(N),** In merge sort all elements are copied into an auxiliary array. So, N auxiliary space is required for merge sort. Therefore, it is **not in-place sorting algorithm**.
* Used for **sorting large datasets, not good for small arrays**
* used in **external sorting**, where the data to be sorted is too large to fit into memory.
* Custom sorting: can be **adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.**
* Inversion Count Problem
* **Stable Sorting algorithm**, relative order of similar key elements not changed
* **parallelizable algorithm**, which means it **can be easily parallelized to take advantage of multiple processors or threads.**

**Quick Sort**

**Divide and conquer**

**Memory**

**Best O(1)**

**Worst O(n)**

**Average O(logN)**

**Large Datasets, not for small**

**Unstable**

The number of comparisons needed the worst case by the merge sort algorithm

M+n-1

running time that is least dependent on the initial ordering of the input?

Merge Sort

In Insertion sort if the array is already sorted then it takes O(n) and if it is reverse sorted then it takes O(n2) to sort the array. In Quick sort, if the array is already sorted or if it is reverse sorted then it takes O(n2).The best and worst case performance of Selection is O(n2) only. But if the array is already sorted then less swaps take place. In merge sort, time complexity is O(nlogn) for all the cases and performance is affected least on the order of input sequence. So, option (A) is correct.

Extremely large datasets: Quick Sort

Which of the following is true about merge sort?

|  |  |
| --- | --- |
|  | Merge Sort works better than quick sort if data is accessed from slow sequential memory. |
| Cross | Merge Sort is stable sort by nature |
|  | Merge sort outperforms heap sort in most of the practical situations |

Which sorting algorithm will take least time when all elements of input array are identical?

Insertion

Check merge last hyperlinks

In a modified merge sort, the input array is splitted at a position one-third of the length(N) of the array. Which of the following is the tightest upper bound on time complexity of this modified Merge Sort.

N(logN base 3/2)

**Counting Sort**

* sorting technique **based on keys between a specific range**
* works by **counting the number of objects having distinct key values (a kind of hashing)**
* Then do some **arithmetic operations to calculate the position of each object in the output sequence**.
* **Time Complexity: O(N + K)** where **N: number of elements** in the input array and **K: range of input.**Auxiliary Space: O(N + K)
* **makes assumptions about the data**, e.g., range of input, or input data will contain positive integers only
* **not a comparison-based algorithm**, it hashes the value in a temporary count array and uses them for sorting.
* uses a temporary array making it a **non-In[Place algorithm](https://www.geeksforgeeks.org/in-place-algorithm/).**
* efficient if the range of input data is not significantly greater than the number of objects to be sorted
* used as a **sub-routine to radix sort.**
* uses partial hashing to count the occurrence of the data object in O(1).
* **can be extended to work for negative inputs also.**
* **Stable Sorting algorithm**

**Inversion Count in Array**

Inversion Count: how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in reverse order, the inversion count is the maximum.

The task is to find the inversion count of arr[]. Where two elements arr[i] and arr[j] form an inversion if a[i] > a[j] and i < j.

Input: arr[] = {8,4,2,1}

Output: 6

Explanation: Given array has six inversions: (8,4), (4,2), (8,2), (8,1), (4,1), (2,1)

Input: arr[] = {1,20,6,4,5}

Output: 5

Explanation: Given array has five inversions: (20,6), (20,4), (20,5), (6,4), (6,5)

Traverse through the array, and for every index, find the number of smaller elements on its right side of the array. This can be done using a nested loop. Sum up the counts for all indices in the array and print the sum.

**static** **int** arr[] = **new** **int**[] { 1, 20, 6, 4, 5 };

**static** **int** getInvCount(**int** n)

    {

**int** inv\_count = 0;

**for** (**int** i = 0; i < n - 1; i++)

**for** (**int** j = i + 1; j < n; j++)

**if** (arr[i] > arr[j])

                    inv\_count++;

**return** inv\_count;

    }

Number of inversions are 5

**Time Complexity:** O(N2), nested loop

**Auxiliary Space:**O(1)