Sorts

By Mani Abedii

View Full DSA Repository

A clear, compact guide covering essential data structures and algorithms, designed for easy understanding and quick learning.

Sorting algorithms are techniques used to arrange data elements in a specific order. Some of the most commonly used sorting algorithms include:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Quick Sort
5. Merge Sort
6. Counting Sort
7. Radix Sort
8. Bucket Sort
9. Comb Sort
10. Shell Sort
11. Heap Sort

These algorithms vary in terms of efficiency, speed, and memory usage. Selecting the right sorting algorithm depends on factors such as the size and type of the data, as well as the desired output. While some algorithms perform well on small datasets, others are optimized for handling large and complex data efficiently.

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.

Algorithm Overview:

1. The array is sorted through a sequence of iterative passes. After the completion of the first pass, the largest element is moved to the final position of the array, which is its correct sorted position. Similarly, after the second pass, the second-largest element is placed in the second-to-last position, and this process continues for subsequent passes.
2. During each pass, only the unsorted portion of the array is considered—that is, the elements that have not yet reached their correct positions. After kkk passes, the kkk largest elements will have been placed in their respective final positions at the end of the array.
3. In each pass, the algorithm traverses the remaining unsorted elements, comparing each pair of adjacent elements. Whenever a larger element precedes a smaller one, the two are swapped. This ensures that, by the end of the pass, the largest element among the remaining unsorted elements is moved to its correct position.

- **Time Complexity: O(n^2)**

Advantages of Bubble Sort:

- **Simplicity of implementation:** Conceptually simple and easy to understand
- **In-place sorting:** No additional memory required beyond a few auxiliary variables

Disadvantages of Bubble Sort

- **Inefficient time complexity:** Worst-case and average-case time complexity of O(n^2)
- **Dependence on comparisons:** As a comparison-based algorithm, Bubble Sort relies on pairwise element comparisons to determine order, limiting its performance and adaptability compared to more advanced sorting algorithms.

Selection Sort

Selection Sort is a comparison-based sorting algorithm that sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element. This process is repeated until all elements are placed in their correct positions and the array is fully sorted.

Algorithm Overview:

1. First, the smallest element in the array is identified and swapped with the first element, placing it in its correct position.
2. Next, the smallest element among the remaining unsorted elements (i.e., the second smallest overall) is found and swapped with the second element.
3. This process is repeated until all elements have been placed in their correct positions and the array is fully sorted.

- **Time Complexity: O(n^2)**

Advantages of Selection Sort:

- **Simplicity of implementation:** Simple and easy to understand and implement
- **In-place sorting:** Requires only a constant O(1) extra memory space
- **Fewer swaps:** Requires smaller number of swaps compared to many other sorting algorithms.

Disadvantages of Selection Sort

- **Inefficient time complexity:** Worst-case and average-case time complexity of O(n^2).
- **Dependence on comparisons:** Does not maintain the relative order of equal elements which means it is not stable.

Insertion Sort

Insertion Sort is a simple sorting algorithm that iteratively inserts each element from the unsorted portion of the list into its appropriate position within the sorted portion, gradually building the sorted list one element at a time.

Algorithm Overview:

1. Divide the array: Conceptually separate the array into two parts — a *sorted portion* (initially containing only the first element) and an *unsorted portion* (containing all remaining elements).
2. Pick an element: Begin with the first element of the unsorted portion and determine its correct position within the sorted portion.
3. Shift elements: If the selected element is smaller than any elements in the sorted portion, shift those larger elements one position to the right to create space.
4. Insert the element: Place the selected element into its correct position within the sorted portion.
5. Repeat: Continue this process for each element in the unsorted portion until the entire array becomes sorted.

- **Time Complexity:**
  - Best Case (The list is already sorted): O(n)
  - Average Case (The list is randomly ordered): O(n^2)
  - Worst Case (The list is in reverse order): O(n^2)

Advantages of Insertion Sort:

- **Simplicity of implementation:** Simple and easy to understand and implement
- **In-place sorting:** Requires only a constant O(1) extra memory space
- **Fewer swaps:** Requires smaller number of swaps compared to many other sorting algorithms.

Disadvantages of Insertion Sort

- **Inefficient time complexity:** Worst-case and average-case time complexity of O(n^2).
- **Dependence on comparisons:** Does not maintain the relative order of equal elements which means it is not stable.

Quick Sort

Quick Sort is a divide-and-conquer sorting algorithm that selects a pivot element and partitions the array around it, placing the pivot in its correct sorted position. The algorithm then recursively sorts the resulting subarrays.

Algorithm Overview:

1. Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary.
2. Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, then we obtain the index of the pivot.
3. Recursive Call: Recursively, apply the same process to the two partitioned sub-arrays (left & right of the pivot).
4. Base Case: The recursion stops when there is only one element left in the subarray as a single element is already sorted.

- **Time Complexity:**
    - Best Case (The pivot element divides the array into two equal halves): O(nlog(n))
    - Average Case (The pivot divides the array into two parts, but not necessarily equal): O(nlog(n))
    - Worst Case (The smallest or largest element is always chosen as the pivot): $O(n^2)$

Choice of Pivot:

There are many different choices for picking pivots:

- **Approach 1:** Always pick the first (or last) element as the pivot.
    - This method is simple but can lead to the worst-case performance when the input array is already sorted.
- **Approach 2:** Choose a random element as the pivot.
    - This approach is generally preferred because it avoids predictable patterns that could trigger the worst-case scenario.
- **Approach 3:** Choose the median element as the pivot.
    - This option is theoretically ideal, as selecting the true median guarantees perfectly balanced partitions. Although the median can be found in linear time, the associated constant factors make this method slower in practice.

Advantages of Quick Sort:

- It follows the divide-and-conquer paradigm, which simplifies problem solving and implementation.
- It is highly efficient on large datasets.
- It has low overhead, requiring only a small amount of additional memory.
- It is among the fastest general-purpose sorting algorithms when stability is not a requirement.

Disadvantages of Quick Sort

- Its worst-case time complexity is $O(n^2)$, which occurs when pivot selection is poor.
- It is generally less effective for very small datasets.
- It is not a stable sorting algorithm.

Merge Sort

Merge Sort is a sorting algorithm that applies the *divide-and-conquer* strategy. It recursively divides the input array into smaller subarrays, sorts each subarray, and then merges the sorted subarrays to produce the final ordered array.

In essence, the algorithm splits the array into two halves, sorts each half independently, and merges them. This procedure continues until the entire array is fully sorted.

Algorithm Overview:

1. Divide: Recursively split the array into two halves until no further division is possible.
2. Conquer: Apply the Merge Sort algorithm to sort each resulting subarray independently.
3. Merge: Combine the sorted subarrays into a single sorted array, continuing the process until all elements have been merged in order.

- **Time Complexity: O(nlog(n))**

Advantages of Merge Sort:

- It is a stable sorting algorithm
- It provides guaranteed worst-case performance, maintaining good efficiency even on large or random datasets.
- It is straightforward to implement.
- Its structure makes it naturally parallelizable, allowing efficient use of multi-core systems.

Disadvantages of Merge Sort

- It has high space complexity, requiring additional memory proportional to the size of the input.
- It is not an in-place algorithm, since extra storage is needed for merging.
- In practice, it is often slower than Quick Sort on average.