

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 k passes, the k 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:

- Simple to understand and implement
- In-place algorithm.
- Very few swaps (only $n - 1$ swaps in total), which is useful for write-limited systems

Disadvantages of Selection Sort

- Inefficient time complexity: always $O(n^2)$, regardless of input order
- Not stable (equal elements may be reordered)
- Not adaptive: doesn't get faster on partially sorted data

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:

- Simple and intuitive algorithm
- In-place sorting
- Stable: preserves relative order of equal elements
- Adaptive: runs in $O(n)$ on nearly-sorted input
- Efficient for small arrays

Disadvantages of Insertion Sort

- Inefficient on large, randomly ordered lists: average & worst case $O(n^2)$
- Lots of shifting operations (though fewer swaps than many sorts)

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): $O(n \log(n))$
 - Average Case (The pivot divides the array into two parts, but not necessarily equal): $O(n \log(n))$
 - 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: $O(n \log(n))$**

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.

Counting Sort

Counting Sort is a non-comparison-based sorting algorithm that relies on counting the frequency of each unique value in the input. It is highly efficient when the range of input values is not significantly larger than the number of elements being sorted.

Algorithm Overview:

1. Create a count array: Determine the maximum value in the input array and initialize a count array of size $\max + 1$ with all values set to 0.
2. Count occurrences: For each element in the input array, increment the corresponding index in the count array.
3. Convert to cumulative count: Transform the count array into a cumulative count array, where each index now represents the position of that value in the final sorted array.
4. Build the output array: Traverse the input array in reverse, placing each element in its correct position in the output array using the cumulative counts.
5. Copy back: Transfer the sorted elements from the output array back to the input array.
 - **Time Complexity: $O(n + k)$** , where n is the array size and k is the range of input values.

Advantages of Counting Sort:

- Extremely fast when the input range is small relative to n
- Simple to implement
- Stable by design

Disadvantages of Counting Sort

- Inefficient when the input range is very large
- Not suitable for floating-point or negative values without modification
- Not in-place due to additional output array

Radix Sort

Radix Sort is a non-comparison-based sorting algorithm that sorts numbers by processing individual digits. It often uses Counting Sort as a subroutine for digit-wise sorting.

Algorithm Overview:

1. Find maximum digits: Determine the number of digits in the largest element in the input.
2. Sort by each digit: Starting from the least significant digit (LSD), sort all elements based on that digit. A stable sort such as Counting Sort is typically used.
3. Repeat for all digits: Continue digit-based sorting until the most significant digit (MSD) has been processed.
4. Completion: After the final digit pass, the array is fully sorted.
 - **Time Complexity: $O(n * k)$** , where k is the number of digits.

Advantages of Radix Sort:

- Faster than comparison-based algorithms for fixed-length integer keys
- Stable sorting
- Works well for uniformly-sized numeric or string data

Disadvantages of Radix Sort

- Requires a stable subroutine (e.g., Counting Sort)
- Not in-place
- Only works for fixed-length keys (numbers or fixed-length strings)
- Extra memory usage

Bucket Sort

Bucket Sort distributes elements into multiple “buckets,” sorts each bucket individually, then combines the results to form the final sorted array.

Algorithm Overview:

1. Create buckets: Initialize n empty buckets.
 2. Distribute elements: Place each element into a bucket based on a hash or mapping function.
 3. Sort buckets: Sort each bucket individually, typically using Insertion Sort.
 4. Concatenate: Combine all buckets in order to produce the final sorted array.
- **Time Complexity:**
 - Best Case (Evenly distributed data): $O(n + k)$
 - Worst Case (When all elements end up in a single bucket): $O(n^2)$

Advantages of Bucket Sort:

- Very fast for uniformly distributed data
- Stable
- Works well for floating-point values
- Highly parallelizable

Disadvantages of Bucket Sort:

- Performance heavily depends on data distribution
- Requires additional space for buckets
- Choosing the right number of buckets can be tricky

Comb Sort

Comb Sort is an improvement over Bubble Sort that eliminates small values near the end of the list (called “turtles”) by comparing elements separated by a shrinking gap.

Algorithm Overview:

1. Start with a large gap (typically n).
 2. In each pass, compare elements separated by this gap and swap if out of order.
 3. Reduce the gap by a shrink factor (commonly 1.3).
 4. Continue until the gap becomes 1, at which point the algorithm behaves like Bubble Sort.
- **Time Complexity:**
 - Best Case: $O(n \log(n))$
 - Worst Case: $O(n^2)$

Advantages of Comb Sort:

- Faster than Bubble Sort in practice
- Simple to implement
- Effectively removes “turtles” that slow Bubble Sort

Disadvantages of Comb Sort:

- Still has a quadratic worst-case complexity
- Not stable

Shell Sort

Shell Sort is a generalization of Insertion Sort that allows exchanging elements far apart by using a decreasing gap sequence.

Algorithm Overview:

1. Choose a gap value (often $n/2$) and perform a gapped insertion sort.
 2. Reduce the gap and repeat the process.
 3. Once the gap reduces to 1, perform a standard Insertion Sort.
- **Time Complexity:**
 - Best Case: $O(n \log(n))$
 - Worst Case: $O(n^2)$

Advantages of Shell Sort:

- Faster than simple quadratic sorts
- Easy to implement
- In-place algorithm

Disadvantages of Shell Sort:

- Efficiency strongly depends on gap sequence
- Not stable
- Hard to analyze theoretical complexity

Heap Sort

Heap Sort is a comparison-based sorting algorithm built on the Binary Heap data structure. It improves on Selection Sort by using a heap to efficiently find the maximum or minimum element in $O(\log n)$ time rather than $O(n)$.

Algorithm Overview:

1. Build a max-heap from the input array.
 2. Swap the root (maximum element) with the last element.
 3. Reduce the heap size and restore heap order using heapify.
 4. Repeat until all elements are extracted and the array becomes sorted.
- **Time Complexity: $O(n \log(n))$**

Advantages of Heap Sort:

- In-place sorting
- Not affected by input distribution

Disadvantages of Heap Sort:

- Higher constant factors compared to Merge Sort
- Not stable
- More complex to implement than simple sorts