# 11.4   Insertion sort[¶](https://www.hello-algo.com/en/chapter_sorting/insertion_sort/#114-insertion-sort)

Insertion sort is a simple sorting algorithm that works very much like the process of manually sorting a deck of cards.

Specifically, we select a pivot element from the unsorted interval, compare it with the elements in the sorted interval to its left, and insert the element into the correct position.

Figure 11-6 shows the process of inserting an element into an array. Assuming the pivot element is base, we need to move all elements between the target index and base one position to the right, then assign base to the target index.

11.4.1   Algorithm process[¶](https://www.hello-algo.com/en/chapter_sorting/insertion_sort/#1141-algorithm-process)

The overall process of insertion sort is shown in Figure 11-7.

1. Initially, the first element of the array is sorted.
2. The second element of the array is taken as base, and after inserting it into the correct position, **the first two elements of the array are sorted**.
3. The third element is taken as base, and after inserting it into the correct position, **the first three elements of the array are sorted**.
4. And so on, in the last round, the last element is taken as base, and after inserting it into the correct position, **all elements are sorted**.

def insertion\_sort(nums: list[int]):

"""Insertion sort"""

# Outer loop: sorted range is [0, i-1]

for i in range(1, len(nums)):

base = nums[i]

j = i - 1

# Inner loop: insert base into the correct position within the sorted range [0, i-1]

while j >= 0 and nums[j] > base:

nums[j + 1] = nums[j] # Move nums[j] to the right by one position

j -= 1

nums[j + 1] = base # Assign base to the correct position

11.4.2   Algorithm characteristics[¶](https://www.hello-algo.com/en/chapter_sorting/insertion_sort/#1142-algorithm-characteristics)

* **Time complexity is O(n2), adaptive sorting**: In the worst case, each insertion operation requires n−1, n−2, ..., 2, 1 loops, summing up to (n−1)n/2, thus the time complexity is O(n2). In the case of ordered data, the insertion operation will terminate early. When the input array is completely ordered, insertion sort achieves the best time complexity of O(n).
* **Space complexity is O(1), in-place sorting**: Pointers i and j use a constant amount of extra space.
* **Stable sorting**: During the insertion operation, we insert elements to the right of equal elements, not changing their order.

11.4.3   Advantages of insertion sort[¶](https://www.hello-algo.com/en/chapter_sorting/insertion_sort/#1143-advantages-of-insertion-sort)

The time complexity of insertion sort is O(n2), while the time complexity of quicksort, which we will study next, is O(nlog⁡n). Although insertion sort has a higher time complexity, **it is usually faster in cases of small data volumes**.

This conclusion is similar to that for linear and binary search. Algorithms like quicksort that have a time complexity of O(nlog⁡n) and are based on the divide-and-conquer strategy often involve more unit operations. In cases of small data volumes, the numerical values of n2 and nlog⁡n are close, and complexity does not dominate, with the number of unit operations per round playing a decisive role.

In fact, many programming languages (such as Java) use insertion sort in their built-in sorting functions. The general approach is: for long arrays, use sorting algorithms based on divide-and-conquer strategies, such as quicksort; for short arrays, use insertion sort directly.

Although bubble sort, selection sort, and insertion sort all have a time complexity of O(n2), in practice, **insertion sort is used significantly more frequently than bubble sort and selection sort**, mainly for the following reasons.

* Bubble sort is based on element swapping, which requires the use of a temporary variable, involving 3 unit operations; insertion sort is based on element assignment, requiring only 1 unit operation. Therefore, **the computational overhead of bubble sort is generally higher than that of insertion sort**.
* The time complexity of selection sort is always O(n2). **Given a set of partially ordered data, insertion sort is usually more efficient than selection sort**.
* Selection sort is unstable and cannot be applied to multi-level sorting.

11.5   Quick sort[¶](https://www.hello-algo.com/en/chapter_sorting/quick_sort/#115-quick-sort)

Quick sort is a sorting algorithm based on the divide and conquer strategy, known for its efficiency and wide application.

The core operation of quick sort is "pivot partitioning," aiming to: select an element from the array as the "pivot," move all elements smaller than the pivot to its left, and move elements greater than the pivot to its right. Specifically, the pivot partitioning process is illustrated in Figure 11-8.

1. Select the leftmost element of the array as the pivot, and initialize two pointers i and j at both ends of the array.
2. Set up a loop where each round uses i (j) to find the first element larger (smaller) than the pivot, then swap these two elements.
3. Repeat step 2. until i and j meet, finally swap the pivot to the boundary between the two sub-arrays.

After the pivot partitioning, the original array is divided into three parts: left sub-array, pivot, and right sub-array, satisfying "any element in the left sub-array ≤ pivot ≤ any element in the right sub-array." Therefore, we only need to sort these two sub-arrays next.

**Quick sort's divide and conquer strategy**

The essence of pivot partitioning is to simplify a longer array's sorting problem into two shorter arrays' sorting problems.

def partition(self, nums: list[int], left: int, right: int) -> int:

"""Partition"""

# Use nums[left] as the pivot

i, j = left, right

while i < j:

while i < j and nums[j] >= nums[left]:

j -= 1 # Search from right to left for the first element smaller than the pivot

while i < j and nums[i] <= nums[left]:

i += 1 # Search from left to right for the first element greater than the pivot

# Swap elements

nums[i], nums[j] = nums[j], nums[i]

# Swap the pivot to the boundary between the two subarrays

nums[i], nums[left] = nums[left], nums[i]

return i # Return the index of the pivot

11.5.1   Algorithm process[¶](https://www.hello-algo.com/en/chapter_sorting/quick_sort/#1151-algorithm-process)

The overall process of quick sort is shown in Figure 11-9.

1. First, perform a "pivot partitioning" on the original array to obtain the unsorted left and right sub-arrays.
2. Then, recursively perform "pivot partitioning" on both the left and right sub-arrays.
3. Continue recursively until the sub-array length reaches 1, thus completing the sorting of the entire array.

def quick\_sort(self, nums: list[int], left: int, right: int):

"""Quick sort"""

# Terminate recursion when subarray length is 1

if left >= right:

return

# Partition

pivot = self.partition(nums, left, right)

# Recursively process the left subarray and right subarray

self.quick\_sort(nums, left, pivot - 1)

self.quick\_sort(nums, pivot + 1, right)

11.5.2   Algorithm features[¶](https://www.hello-algo.com/en/chapter_sorting/quick_sort/#1152-algorithm-features)

* **Time complexity of O(nlog⁡n), non-adaptive sorting**: In average cases, the recursive levels of pivot partitioning are log⁡n, and the total number of loops per level is n, using O(nlog⁡n) time overall. In the worst case, each round of pivot partitioning divides an array of length n into two sub-arrays of lengths 0 and n−1, reaching n recursive levels, and using O(n2) time overall.
* **Space complexity of O(n), in-place sorting**: In completely reversed input arrays, reaching the worst recursion depth of n, using O(n) stack frame space. The sorting operation is performed on the original array without the aid of additional arrays.
* **Non-stable sorting**: In the final step of pivot partitioning, the pivot may be swapped to the right of equal elements.

11.5.3   Why is quick sort fast[¶](https://www.hello-algo.com/en/chapter_sorting/quick_sort/#1153-why-is-quick-sort-fast)

From its name, it is apparent that quick sort should have certain efficiency advantages. Although the average time complexity of quick sort is the same as "merge sort" and "heap sort," quick sort is generally more efficient, mainly for the following reasons.

* **Low probability of worst-case scenarios**: Although the worst time complexity of quick sort is O(n2), less stable than merge sort, in most cases, quick sort can operate under a time complexity of O(nlog⁡n).
* **High cache usage efficiency**: During the pivot partitioning operation, the system can load the entire sub-array into the cache, thus accessing elements more efficiently. In contrast, algorithms like "heap sort" need to access elements in a jumping manner, lacking this feature.
* **Small constant coefficient of complexity**: Among the mentioned algorithms, quick sort has the fewest total number of comparisons, assignments, and swaps. This is similar to why "insertion sort" is faster than "bubble sort."

11.5.4   Pivot optimization[¶](https://www.hello-algo.com/en/chapter_sorting/quick_sort/#1154-pivot-optimization)

**Quick sort's time efficiency may decrease under certain inputs**. For example, if the input array is completely reversed, since we select the leftmost element as the pivot, after the pivot partitioning, the pivot is swapped to the array's right end, causing the left sub-array length to be n−1 and the right sub-array length to be 0. If this recursion continues, each round of pivot partitioning will have a sub-array length of 0, and the divide and conquer strategy fails, degrading quick sort to a form similar to "bubble sort."

To avoid this situation, **we can optimize the strategy for selecting the pivot in the pivot partitioning**. For instance, we can randomly select an element as the pivot. However, if luck is not on our side, and we keep selecting suboptimal pivots, the efficiency is still not satisfactory.

It's important to note that programming languages usually generate "pseudo-random numbers". If we construct a specific test case for a pseudo-random number sequence, the efficiency of quick sort may still degrade.

For further improvement, we can select three candidate elements (usually the first, last, and midpoint elements of the array), **and use the median of these three candidate elements as the pivot**. This significantly increases the probability that the pivot is "neither too small nor too large". Of course, we can also select more candidate elements to further enhance the algorithm's robustness. Using this method significantly reduces the probability of time complexity degradation to O(n2).

Sample code is as follows:

def median\_three(self, nums: list[int], left: int, mid: int, right: int) -> int:

"""Select the median of three candidate elements"""

l, m, r = nums[left], nums[mid], nums[right]

if (l <= m <= r) or (r <= m <= l):

return mid # m is between l and r

if (m <= l <= r) or (r <= l <= m):

return left # l is between m and r

return right

def partition(self, nums: list[int], left: int, right: int) -> int:

"""Partition (median of three)"""

# Use nums[left] as the pivot

med = self.median\_three(nums, left, (left + right) // 2, right)

# Swap the median to the array's leftmost position

nums[left], nums[med] = nums[med], nums[left]

# Use nums[left] as the pivot

i, j = left, right

while i < j:

while i < j and nums[j] >= nums[left]:

j -= 1 # Search from right to left for the first element smaller than the pivot

while i < j and nums[i] <= nums[left]:

i += 1 # Search from left to right for the first element greater than the pivot

# Swap elements

nums[i], nums[j] = nums[j], nums[i]

# Swap the pivot to the boundary between the two subarrays

nums[i], nums[left] = nums[left], nums[i]

return i # Return the index of the pivot

11.5.5   Tail recursion optimization[¶](https://www.hello-algo.com/en/chapter_sorting/quick_sort/#1155-tail-recursion-optimization)

**Under certain inputs, quick sort may occupy more space**. For a completely ordered input array, assume the sub-array length in recursion is m, each round of pivot partitioning produces a left sub-array of length 0 and a right sub-array of length m−1, meaning the problem size reduced per recursive call is very small (only one element), and the height of the recursion tree can reach n−1, requiring O(n) stack frame space.

To prevent the accumulation of stack frame space, we can compare the lengths of the two sub-arrays after each round of pivot sorting, **and only recursively sort the shorter sub-array**. Since the length of the shorter sub-array will not exceed n/2, this method ensures that the recursion depth does not exceed log⁡n, thus optimizing the worst space complexity to O(log⁡n). The code is as follows:

def quick\_sort(self, nums: list[int], left: int, right: int):

"""Quick sort (tail recursion optimization)"""

# Terminate when subarray length is 1

while left < right:

# Partition operation

pivot = self.partition(nums, left, right)

# Perform quick sort on the shorter of the two subarrays

if pivot - left < right - pivot:

self.quick\_sort(nums, left, pivot - 1) # Recursively sort the left subarray

left = pivot + 1 # Remaining unsorted interval is [pivot + 1, right]

else:

self.quick\_sort(nums, pivot + 1, right) # Recursively sort the right subarray

right = pivot - 1 # Remaining unsorted interval is [left, pivot - 1]

11.6   Merge sort[¶](https://www.hello-algo.com/en/chapter_sorting/merge_sort/#116-merge-sort)

Merge sort is a sorting algorithm based on the divide-and-conquer strategy, involving the "divide" and "merge" phases shown in Figure 11-10.

1. **Divide phase**: Recursively split the array from the midpoint, transforming the sorting problem of a long array into that of shorter arrays.
2. **Merge phase**: Stop dividing when the length of the sub-array is 1, start merging, and continuously combine two shorter ordered arrays into one longer ordered array until the process is complete.

11.6.1   Algorithm workflow[¶](https://www.hello-algo.com/en/chapter_sorting/merge_sort/#1161-algorithm-workflow)

As shown in Figure 11-11, the "divide phase" recursively splits the array from the midpoint into two sub-arrays from top to bottom.

1. Calculate the midpoint mid, recursively divide the left sub-array (interval [left, mid]) and the right sub-array (interval [mid + 1, right]).
2. Continue with step 1. recursively until the sub-array interval length is 1 to stop.

The "merge phase" combines the left and right sub-arrays into a single ordered array from bottom to top. Note that merging starts with sub-arrays of length 1, and each sub-array is ordered during the merge phase.

It is observed that the order of recursion in merge sort is consistent with the post-order traversal of a binary tree.

* **Post-order traversal**: First recursively traverse the left subtree, then the right subtree, and finally handle the root node.
* **Merge sort**: First recursively handle the left sub-array, then the right sub-array, and finally perform the merge.

The implementation of merge sort is shown in the following code. Note that the interval to be merged in nums is [left, right], while the corresponding interval in tmp is [0, right - left].

def merge(nums: list[int], left: int, mid: int, right: int):

"""Merge left subarray and right subarray"""

# Left subarray interval is [left, mid], right subarray interval is [mid+1, right]

# Create a temporary array tmp to store the merged results

tmp = [0] \* (right - left + 1)

# Initialize the start indices of the left and right subarrays

i, j, k = left, mid + 1, 0

# While both subarrays still have elements, compare and copy the smaller element into the temporary array

while i <= mid and j <= right:

if nums[i] <= nums[j]:

tmp[k] = nums[i]

i += 1

else:

tmp[k] = nums[j]

j += 1

k += 1

# Copy the remaining elements of the left and right subarrays into the temporary array

while i <= mid:

tmp[k] = nums[i]

i += 1

k += 1

while j <= right:

tmp[k] = nums[j]

j += 1

k += 1

# Copy the elements from the temporary array tmp back to the original array nums at the corresponding interval

for k in range(0, len(tmp)):

nums[left + k] = tmp[k]

def merge\_sort(nums: list[int], left: int, right: int):

"""Merge sort"""

# Termination condition

if left >= right:

return # Terminate recursion when subarray length is 1

# Partition stage

mid = left + (right - left) // 2 # Calculate midpoint

merge\_sort(nums, left, mid) # Recursively process the left subarray

merge\_sort(nums, mid + 1, right) # Recursively process the right subarray

# Merge stage

merge(nums, left, mid, right)

11.6.2   Algorithm characteristics[¶](https://www.hello-algo.com/en/chapter_sorting/merge_sort/#1162-algorithm-characteristics)

* **Time complexity of O(nlog⁡n), non-adaptive sort**: The division creates a recursion tree of height log⁡n, with each layer merging a total of n operations, resulting in an overall time complexity of O(nlog⁡n).
* **Space complexity of O(n), non-in-place sort**: The recursion depth is log⁡n, using O(log⁡n) stack frame space. The merging operation requires auxiliary arrays, using an additional space of O(n).
* **Stable sort**: During the merging process, the order of equal elements remains unchanged.

11.6.3   Linked List sorting[¶](https://www.hello-algo.com/en/chapter_sorting/merge_sort/#1163-linked-list-sorting)

For linked lists, merge sort has significant advantages over other sorting algorithms, **optimizing the space complexity of the linked list sorting task to O(1)**.

* **Divide phase**: "Iteration" can be used instead of "recursion" to perform the linked list division work, thus saving the stack frame space used by recursion.
* **Merge phase**: In linked lists, node addition and deletion operations can be achieved by changing references (pointers), so no extra lists need to be created during the merge phase (combining two short ordered lists into one long ordered list).

Detailed implementation details are complex, and interested readers can consult related materials for learning.