Insertion Sort

bluesky: @leonlonsdale.dev
discord: https://discord.gg/dhrdFh98UA

How it works

- **Insertion Sort** is a simple sorting algorithm that builds the sorted list one item at a time.
- It works by iterating through the list, starting from the second element.
- For each element, the algorithm compares it with the elements before it (in the sorted portion of the list).
- The current element is then placed in its correct position by shifting larger elements to the right.
- This process creates two sub-arrays:
  - **Left sub-array**: Sorted portion.
  - **Right sub-array**: Unsorted portion.
- The algorithm repeats this process until the entire list is sorted.
- The iteration starts from index 1 because the element at index 0 is already considered sorted at the beginning.

Pseudocode

1. Start at the second element in the list (index 1), as the element at index 0 is already considered sorted.
2. For each element from index 1 to the end of the list:
   - Set the current element as the "sort number."
   - Set a pointer `position` to the element just before the current element (i.e., `position = i - 1`).
   - While `position` is valid (i.e., greater than or equal to 0) and the element at `position` is greater than the "sort number":
     - Shift the element at `position` one position to the right (i.e., `arr[position + 1] = arr[position]`).
     - Decrement `position` by 1.
   - Once the correct position is found (where `arr[position]` is less than or equal to the "sort number"), place the "sort number" at `arr[position + 1]`.
3. Repeat this process for each element in the list until the entire list is sorted.
4. Return the sorted list.

Example Code

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        sort_number = i
        position = i - 1

        while position >= 0 and arr[position] > sort_number:
            arr[position + 1] = arr[position]
            position -= 1

        arr[position + 1] = sort_number

    return arr
```

Complexity

Time Complexity

Worst Case

The worst case occurs when the list is in reversed order.

1. **Outer Loop:** The outer loop will always iterate over the length of the list, which is $O(n)$.

2. **While Loop:** The while loop will run for each element in the list, as every value needs to be compared and shifted. This is also $O(n)$.

Since the while loop runs $O(n)$ times for each iteration of the outer loop, the worst-case time complexity is $O(n^2)$.

Best Case

The best case occurs when the list is already sorted. In this case:

1. **Outer Loop:** The outer loop still iterates over the length of the list, so it's $O(n)$.
2. **While Loop:** The while loop will not be triggered because the elements are already in the correct order. Thus, the inner loop runs in constant time, $O(1)$.

Overall, the best-case time complexity is $O(n)$.

Space Complexity

Insertion Sort uses a constant amount of extra space, as it only needs a few variables (like `sort_number` and `position`) to keep track of the current element and its position. These variables do not depend on the size of the input list.