Insertion Sort

Insertion Sort

- **Core Idea**: Build the final sorted array one item at a time.

- **Analogy**: How most people sort a hand of playing cards. You hold the sorted cards in one hand and pick up new cards one by one, inserting them into their correct position.

The Algorithm: Step-by-Step

The algorithm maintains a sorted sub-array at the beginning of the list. For each new element, it is "inserted" into its correct place within that sorted part.

1. Start with the first element (a sorted sub-array of size 1).
2. Take the next element (*tmp*).
3. Shift all elements in the sorted sub-array that are greater than *tmp* one position to the right.
4. Insert *tmp* into the created opening.
5. Repeat for all remaining elements.

Visualizing Insertion Sort

Let's sort the array *[5, 2, 4, 6, 1, 3]*. The | separates the sorted sub-array from the rest.

- **Initial**: *[5 | 2, 4, 6, 1, 3]*
- **After inserting 2**: *[2, 5 | 4, 6, 1, 3]*
- **After inserting 4**: *[2, 4, 5 | 6, 1, 3]*
- **After inserting 6**: *[2, 4, 5, 6 | 1, 3] (6 is already in place)*
- **After inserting 1**: *[1, 2, 4, 5, 6 | 3]*
- **After inserting 3**: *[1, 2, 3, 4, 5, 6] (Done)*

A More Efficient Implementation

- Instead of repeatedly swapping elements (which takes three assignments), we can store the element to be inserted in a temporary variable (tmp) and shift larger elements to the right (one assignment). This is significantly faster.

```
def insertion_sort(array, n):
    for k in range(1, n):
        tmp = array[k]
        j = k - 1
        # Shift elements greater than tmp to the right
        while j >= 0 and array[j] > tmp:
            array[j + 1] = array[j]
            j -= 1
        # Insert tmp into its correct position
        array[j + 1] = tmp
```

Performance Analysis

- **Space Complexity**: *Theta(1)*. The sort happens **in-place**, requiring only a few extra variables.

- **Time Complexity**: Highly dependent on the initial order of the data. We analyze it in three cases: Best, Worst, and Average.

Time Complexity: Best Case

- **Input**: An already sorted array (e.g., *[10, 20, 30, 40]*).

- **Behavior**: When picking the next element *tmp*, the *while* loop condition (*array[j] > tmp*) is immediately false. No shifting ever occurs.

- **Runtime**: The outer loop runs $n-1$ times, performing a single comparison each time. The total runtime is *Theta(n)*.

Time Complexity: Worst Case

- **Input**: A reverse-sorted array (e.g., *[40, 30, 20, 10]*).

- **Behavior**: Each new element *tmp* is the smallest seen so far and must be shifted all the way to the beginning of the array. The *k-th* element requires k comparisons and shifts.

- **Runtime**: The total number of inner loop operations is approximately *1+2+dots+(n−1)=fracn(n−1)2*. The total runtime is *Theta(*n^2*)*.

Time Complexity: Average Case & Inversions

- **Input**: A randomly shuffled array.
- **Behavior**: On average, each new element tmp will be inserted into the middle of the already sorted sub-array. This still requires, on average, *k/2* shifts for the *k-th* element.
- **Runtime**: The total runtime is still *Theta*(n^2).
- **A More Precise View**: The number of shifts is exactly equal to the number of inversions *(d)* in the array. This gives a more accurate runtime formula: *Theta(n+d*).

When is Insertion Sort Useful?

Despite its *Theta*(n^2) average case, Insertion Sort is very practical in specific scenarios:

1. **For Small Arrays**: Due to its simplicity and low overhead, it's often faster than complex algorithms like Quicksort for small n (e.g., $n < 16$).
2. **For "Nearly Sorted" Arrays**: If an array has very few inversions (i.e., d is close to n), its runtime approaches *Theta(n)*. This makes it an excellent choice for data that is already mostly in order.
3. **As a Building Block**: Because of these strengths, it's used as a component in sophisticated hybrid algorithms like **Timsort** (used by Python and Java) and **Introsort**.

Summary: Insertion Sort

- **Algorithm**: Inserts elements one by one into a growing sorted sub-array.
- **Space**
- T**ime Complexity**:
    - **Best**: *Theta(n)* (Adaptive for nearly sorted data).
    - **Average**: *Theta(*n^2*)*.
    - **Worst**: *Theta(*n^3*)*.
- **Key Feature**: Simple, efficient for small lists, and highly effective on data that is already mostly sorted.