Insertion Sort Algorithm

Lecture Notes

1) Introduction

Insertion Sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, it provides several advantages for small or nearly sorted datasets.

Analogy: Insertion sort works similarly to the way you sort playing cards in your hands. Imagine you are holding a hand of cards:

- You start with an empty left hand and the cards face down on the table.
- You remove one card at a time from the table and insert it into the correct position in the left hand.
- To find the correct position for a new card, you compare it with the cards already in the hand, moving from right to left.
- At all times, the cards held in the left hand are sorted.

2) Algorithm Steps

The algorithm divides the array into two parts: a **sorted subarray** (initially just the first element) and an **unsorted subarray** (the rest of the array).

1. Iterate from the second element (index 1) to the end of the array.
2. Select the current element and store it in a temporary variable (often called `key`).
3. Compare the `key` with the elements in the sorted subarray (indices less than the current index).
4. Shift all elements in the sorted subarray that are greater than the `key` to one position to the right.
5. Insert the `key` into its correct position.
6. Repeat until the array is fully sorted.

3) Pseudocode

Below is the standard pseudocode implementation using 0-based indexing.

```
procedure insertionSort(A: list of sortable items)
    n = length(A)
    for i from 1 to n-1 do
        key = A[i]
        j = i - 1

        // Move elements of A[0..i-1], that are greater than key,
        // to one position ahead of their current position
        while j >= 0 and A[j] > key do
            A[j + 1] = A[j]
            j = j - 1
        end while

        A[j + 1] = key
    end for
end procedure
```

4) Working with Example

Let us trace the execution of the algorithm on the array $A = [5, 3, 4, 1, 2]$.

Initial Array: $[5, 3, 4, 1, 2]$

- The element at index 0 (5) is considered the sorted part.

Pass 1 ($i = 1$, key=3):

- Compare 3 with 5. Since $5 > 3$, shift 5 to the right.
- Array state: $[5, 5, 4, 1, 2]$
- Insert 3 at index 0.
- **Result:** $[3, 5, 4, 1, 2]$

Pass 2 ($i = 2$, key=4):

- Compare 4 with 5. Since $5 > 4$, shift 5.
- Compare 4 with 3. Since $3 < 4$, stop shifting.
- Insert 4 at index 1.
- **Result:** $[3, 4, 5, 1, 2]$

Pass 3 ($i = 3$, key=1):

- Compare 1 with $5, 4, 3$. All are greater than 1.
- Shift $5, 4, 3$ to the right.
- Insert 1 at index 0.
- **Result:** $[1, 3, 4, 5, 2]$

Pass 4 ($i = 4$, key=2):

- Compare 2 with $5, 4, 3$. All are greater. Shift them.
- Compare 2 with 1. $1 < 2$, stop shifting.
- Insert 2 at index 1.
- **Result:** $[1, 2, 3, 4, 5]$

5) Analysis of Insertion Sort

5.1 Adaptive Nature

Insertion Sort is **adaptive**, which means that it becomes efficient for data sets that are already substantially sorted. The time complexity decreases significantly as the number of inversions (pairs of elements that are out of order) decreases. If the array is already sorted, the inner while loop never executes shifts, resulting in $O(N)$ time.

5.2 Online Nature

The algorithm is **online**. It can sort a list as it receives it. If we add a new element to an already sorted list, we only need one pass of insertion logic to place the new element correctly. This is useful for systems receiving real-time data streams.

6) Time Complexity Analysis

- **Best Case Complexity:** $O(N)$
  Occurs when the array is already sorted. The algorithm simply compares each element to its predecessor and performs no swaps.
- **Average Case Complexity:** $O(N^2)$
  Occurs when the array elements are in random order.
- **Worst Case Complexity:** $O(N^2)$
  Occurs when the array is sorted in reverse order. Every element has to be compared with every other element in the sorted subarray and shifted.

7) Space Complexity and Stability

- **Space Complexity:** $O(1)$
  It is an in-place sorting algorithm, requiring only a constant amount of additional memory space (for the `key` and iterator variables).
- **Stability:** Stable
  Insertion sort is stable. It does not change the relative order of elements with equal keys.

8) Optimized Variant: Binary Insertion Sort

A common optimization is **Binary Insertion Sort**. In standard Insertion Sort, we use linear search to find the correct position for the $\texttt{key}$ in the sorted subarray. Binary Insertion Sort uses **Binary Search** to find this position.

- This reduces the number of comparisons from $O(N)$ to $O(\log N)$ per iteration.
- However, the number of swaps/shifts required to move elements remains $O(N)$.
- Consequently, the overall worst-case time complexity remains $O(N^2)$, but it performs fewer comparisons.

9) Advantages and Disadvantages

Advantages:

- Simple implementation.
- Efficient for small data sets.
- Adaptive (fast for nearly sorted data).
- Stable and In-place.

Disadvantages:

- Inefficient for large data sets compared to $O(N \log N)$ algorithms.
- High number of writes/shifts in the worst case.

10) Comparison with Other Algorithms

Table 1 compares Insertion Sort with other common sorting algorithms.

Table 1: Comparison of Sorting Algorithms

Algorithm	Best Case	Average Case	Worst Case	Stable?
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	Yes
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	Yes
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	No
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	Yes
Quick Sort	$O(N \log N)$	$O(N \log N)$	$O(N^2)$	No