Topic: Bubble Sort Algorithm

Lecture Notes

1 Introduction to Bubble Sort

Bubble Sort is one of the simplest sorting algorithms used in computer science. It is a **comparison-based** algorithm where each pair of adjacent elements is compared and swapped if they are not in the correct order.

It is often called a *sinking sort* because with each iteration, the largest element in the unsorted portion of the list ”bubbles” up to its correct position (usually the end of the list), much like bubbles rising to the surface of the water.

2 Bubble Sort Algorithm (Step-by-Step Procedure)

The procedure works as follows for sorting a list in ascending order:

1. Start with the first element (index 0).
2. Compare the current element with the next element.
3. If the current element is greater than the next element, **swap** them.
4. If the current element is less than the next element, move to the next pair.
5. Continue this until the end of the array is reached. This completes one **pass**.
6. Repeat the process for the remaining unsorted elements.

3 Pseudocode of Bubble Sort

Below is the standard pseudocode for the unoptimized version of Bubble Sort using 0-based indexing.

```
procedure bubbleSort(A : list of sortable items)
    n := length(A)

    // Outer loop for total passes
    // Ranges from 0 to n-2
    for i from 0 to n - 2 do

        // Inner loop for comparison
        // Ranges from 0 to n-2-i
        for j from 0 to n - 2 - i do

            // Swap if the element found is greater
            // than the next element
            if A[j] > A[j + 1] then
                swap(A[j], A[j + 1])
```

```
            end if

        end for
    end for
end procedure
```

Listing 1: Standard Bubble Sort Algorithm

4 Working of Bubble Sort with Example

Consider an unsorted array: **A = [5, 3, 4, 1, 2]**
Here, $n = 5$.

Pass 1 ($i = 0$)

The inner loop runs for j from 0 to $5 - 2 - 0 = 3$.

- $j = 0$: Compare (5, 3). $5 > 3$, swap. **[3, 5, 4, 1, 2]**
- $j = 1$: Compare (5, 4). $5 > 4$, swap. **[3, 4, 5, 1, 2]**
- $j = 2$: Compare (5, 1). $5 > 1$, swap. **[3, 4, 1, 5, 2]**
- $j = 3$: Compare (5, 2). $5 > 2$, swap. **[3, 4, 1, 2, 5]**

Result of Pass 1: The largest element (5) is fixed at the last position.

Pass 2 ($i = 1$)

The inner loop runs for j from 0 to $5 - 2 - 1 = 2$.

- $j = 0$: Compare (3, 4). $3 < 4$, no swap. **[3, 4, 1, 2, 5]**
- $j = 1$: Compare (4, 1). $4 > 1$, swap. **[3, 1, 4, 2, 5]**
- $j = 2$: Compare (4, 2). $4 > 2$, swap. **[3, 1, 2, 4, 5]**

Result of Pass 2: The second largest element (4) is fixed.

Pass 3 ($i = 2$)

The inner loop runs for j from 0 to $5 - 2 - 2 = 1$.

- $j = 0$: Compare (3, 1). $3 > 1$, swap. **[1, 3, 2, 4, 5]**
- $j = 1$: Compare (3, 2). $3 > 2$, swap. **[1, 2, 3, 4, 5]**

Result of Pass 3: The third largest element (3) is fixed.

Pass 4 ($i = 3$)

The inner loop runs for j from 0 to $5 - 2 - 3 = 0$.

- $j = 0$: Compare (1, 2). $1 < 2$, no swap. **[1, 2, 3, 4, 5]**

The array is now fully sorted.

5 Analysis of Bubble Sort

Beyond simple time complexity, analyzing the behavior of Bubble Sort reveals interesting properties regarding data movement and inversions.

1. Connection to Inversions

The number of swaps performed by Bubble Sort is mathematically equivalent to the number of **inversions** in the array. An inversion is defined as a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$.

- **Best Case:** 0 inversions (sorted array) $\rightarrow$ 0 swaps.
- **Worst Case:** $\frac{N(N-1)}{2}$ inversions (reverse sorted) $\rightarrow$ Maximum swaps.

Since adjacent swaps only remove one inversion at a time, Bubble Sort is often less efficient than algorithms that can remove multiple inversions per swap (like Quick Sort).

2. Rabbits and Turtles

There is a distinct asymmetry in how elements move:

- **"Rabbits" (Large elements at the beginning):** These move very quickly to their correct position. A large element can "bubble" all the way to the end in a single pass.
- **"Turtles" (Small elements at the end):** These move very slowly. A small element at the very end of the array moves only one position to the left during each pass. This "turtle" behavior is the primary reason why Bubble Sort is inefficient for data that is mostly sorted but has small elements near the end.

6 Time Complexity Analysis

The complexity is determined by the number of comparisons and swaps. In the standard algorithm, we have two nested loops.

$$\text{Total Comparisons} = (N-1) + (N-2) + \cdots + 1 = \frac{N(N-1)}{2} \approx \frac{N^2}{2}$$

1. **Worst Case Complexity:** $O(N^2)$
   Occurs when the array is sorted in reverse order. The algorithm performs the maximum number of comparisons and swaps.
2. **Average Case Complexity:** $O(N^2)$
   Occurs when the elements are in random order. The number of comparisons is still quadratic.
3. **Best Case Complexity:** $O(N)$ **(Optimized) /** $O(N^2)$ **(Standard)**
   For the standard algorithm, it is $O(N^2)$ because it blindly compares elements. However, for the optimized version (see Section 7), if the array is already sorted, it runs in $O(N)$ time.

7 Space Complexity and Stability

- **Space Complexity:** $O(1)$
  Bubble Sort is an **in-place** sorting algorithm. It does not require any extra memory structure (like an array) to sort; it only needs a single temporary variable for swapping.

- **Stability: Stable**
  A sorting algorithm is stable if it preserves the relative order of equal elements. In Bubble Sort, elements are swapped only if $A[j] > A[j+1]$. If elements are equal, no swap occurs, preserving their original order.

8 Optimized Bubble Sort (Early Termination)

The standard Bubble Sort always runs $O(N^2)$ even if the array is already sorted. We can optimize this by introducing a flag variable `swapped`.

Logic: If no swaps occur during a pass, the array is already sorted, and we can break the loop immediately.

```
procedure optimizedBubbleSort(A : list)
    n = length(A)
    swapped = true

    while swapped = true do
        swapped = false
        for i from 1 to n - 1 do
            if A[i-1] > A[i] then
                swap(A[i-1], A[i])
                swapped = true
            end if
        end for
        n = n - 1 // Reduce range
    end while
end procedure
```

Listing 2: Optimized Bubble Sort Pseudocode

9 Advantages and Disadvantages

Advantages:

- **Simplicity:** Easy to understand and implement.
- **No Overhead:** Requires no extra memory ($O(1)$ space).
- **Stability:** Does not change the relative order of identical elements.
- **Sorted Detection:** The optimized version can detect a sorted list in just one pass ($O(N)$).

Disadvantages:

- **Performance:** Very slow for large datasets ($O(N^2)$).
- **Excessive Swaps:** It performs more element swaps than Selection Sort, which can be costly if writing to memory is expensive.

10 Comparison with Other Sorting Algorithms

Algorithm	Best Case	Avg Case	Worst Case	Space	Stable?
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Yes
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$	No
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Yes
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$	Yes
Quick Sort	$O(N \log N)$	$O(N \log N)$	$O(N^2)$	$O(\log N)$	No

Table 1: Comparison of Bubble Sort with common sorting algorithms