Bubble Sort

Minor in AI, IIT Ropar

17th March, 2025

1 Introduction

Imagine you're tasked with organizing a vast library of books by author. Traditionally, this would be a labor-intensive task, but what if technology could simplify the process? This case study explores how the bubble sort algorithm can be used to efficiently sort data, making it a foundational tool in programming.

2 Background: Why Bubble Sort Matters

Bubble sort is a simple yet educational algorithm that has become famous due to its ease of understanding and implementation. Despite being inefficient compared to other sorting algorithms, studying bubble sort helps in grasping worst-case scenarios and improving algorithm design.

3 How Bubble Sort Works: A Step-by-Step Guide

Let's dive into a practical example to illustrate how bubble sort works:
Example Array: $64, 34, 25, 90, 12$

Step-by-Step Process

Pass 1

1. **Compare 64 and 34**: Since $64 > 34$, swap them. - Array becomes: $[34, 64, 25, 90, 12]$

2. **Compare 64 and 25**: Since $64 > 25$, swap them. - Array becomes: $[34, 25, 64, 90, 12]$
3. **Compare 64 and 90**: Since $64 < 90$, no swap. - Array remains: $[34, 25, 64, 90, 12]$
4. **Compare 90 and 12**: Since $90 > 12$, swap them. - Array becomes: $[34, 25, 64, 12, 90]$

Pass 2

1. **Compare 34 and 25**: Since $34 > 25$, swap them. - Array becomes: $[25, 34, 64, 12, 90]$
2. **Compare 34 and 64**: Since $34 < 64$, no swap. - Array remains: $[25, 34, 64, 12, 90]$
3. **Compare 64 and 12**: Since $64 > 12$, swap them. - Array becomes: $[25, 34, 12, 64, 90]$
4. **Compare 64 and 90**: Since $64 < 90$, no swap. - Array remains: $[25, 34, 12, 64, 90]$

Pass 3

1. **Compare 25 and 34**: Since $25 < 34$, no swap. - Array remains: $[25, 34, 12, 64, 90]$
2. **Compare 34 and 12**: Since $34 > 12$, swap them. - Array becomes: $[25, 12, 34, 64, 90]$
3. **Compare 34 and 64**: Since $34 < 64$, no swap. - Array remains: $[25, 12, 34, 64, 90]$
4. **Compare 64 and 90**: Since $64 < 90$, no swap. - Array remains: $[25, 12, 34, 64, 90]$

Pass 4

1. **Compare 25 and 12**: Since $25 > 12$, swap them. - Array becomes: $[12, 25, 34, 64, 90]$
2. **Compare 25 and 34**: Since $25 < 34$, no swap. - Array remains: $[12, 25, 34, 64, 90]$
3. **Compare 34 and 64**: Since $34 < 64$, no swap. - Array remains: $[12, 25, 34, 64, 90]$
4. **Compare 64 and 90**: Since $64 < 90$, no swap. - Array remains: $[12, 25, 34, 64, 90]$

Pass 5

1. **Compare 12 and 25**: Since $12 < 25$, no swap. - Array remains: $[12, 25, 34, 64, 90]$
2. **Compare 25 and 34**: Since $25 < 34$, no swap. - Array remains: $[12, 25, 34, 64, 90]$
3. **Compare 34 and 64**: Since $34 < 64$, no swap. - Array remains: $[12, 25, 34, 64, 90]$
4. **Compare 64 and 90**: Since $64 < 90$, no swap. - Array remains: $[12, 25, 34, 64, 90]$

Final Sorted Array

$[12, 25, 34, 64, 90]$

4 Bubble Sort Code

```
def bubble_sort(array):
    n = len(array)
    for i in range(n):
        # Start looking at each item of the list one by one,
        # comparing it with its adjacent value
        for j in range(0, n-i-1):
            # If we find an element that is greater than its adjacent element
            if array[j] > array[j+1]:
                # Swap them
                array[j], array[j+1] = array[j+1], array[j]
    return array

arr = [64, 34, 25, 90, 12]
print("Before sorting: ", arr)
arr = bubble_sort(arr)
print("After sorting: ", arr)
# Output
# Before sorting: [64, 34, 25, 90, 12]
# After sorting: [12, 25, 34, 64, 90]
```

The above code works exactly as the step by step process mentioned above.

5 Optimization: Making Bubble Sort More Efficient

- Basic Optimization: Introduce a flag to stop passes if no swaps occur, reducing unnecessary iterations.
- Impact: Reduces comparisons but maintains $O(n^2)$ time complexity in the worst case.

```
def bubble_sort_modified(array):
    n = len(array)
    for i in range(n):
        # Create a flag that will allow the function to terminate early
        # if there's nothing left to sort
        swapped = False
        # Start looking at each item of the list one by one,
        # comparing it with its adjacent value
        for j in range(0, n-i-1):
            # If we find an element that is greater than its adjacent element
            if array[j] > array[j+1]:
                # Swap them
                array[j], array[j+1] = array[j+1], array[j]
                # Set the flag to True so we'll loop again after this iteration
                swapped = True
        # If no two elements were swapped in inner loop, the list is sorted
        if not swapped:
            break
    return array

arr = [64, 34, 25, 90, 12]
print("Before sorting: ", arr)
arr = bubble_sort_modified(arr)
print("After sorting: ", arr)
# Output
# Before sorting: [64, 34, 25, 90, 12]
# After sorting: [12, 25, 34, 64, 90]
```

In the above code just a little modification to the original bubble sort is done. Here, we have added a swapped variable, if for a pass no swaps were taken place then swapped variable has value False, which in turn would lead to breaking out of the loop and algorithm won't iterate to further as the array has become sorted.

6 Case Analysis: Understanding Performance

- Best Case: Not applicable traditionally, but with optimization, it can be linear if the array is already sorted.
- Worst Case: $O(n^2)$ when the array is reverse sorted.
- Average Case: Generally considered the same as the worst case due to the consistent number of comparisons.

7 Comparison with Other Algorithms

- Selection Sort: Similar time complexity but reduces the number of swaps.
- Python's Sorting Algorithm (Timsort): A hybrid of merge sort and insertion sort, offering better efficiency.

8 Key Takeaways:

- Importance of Intuition: Understanding how algorithms work is crucial for effective programming.
- Efficiency Analysis : Focus on reducing the number of basic operations to improve algorithm efficiency.
- Practical vs. Theoretical Efficiency : There can be a difference between theoretical analysis and real-world performance due to hardware and compiler optimizations.

9 Future Directions: Building on Bubble Sort

- Improving Efficiency: Techniques like divide and conquer can lead to more efficient algorithms.
- Real-World Applications: Understanding algorithmic efficiency is critical for handling large datasets in AI and ML projects.

10 Conclusion

Bubble sort is a foundational algorithm that, despite its inefficiency, provides valuable insights into the principles of sorting and algorithm design. By understanding how bubble sort works and its limitations, developers can better appreciate the importance of efficient algorithms in real-world applications.