DATA STRUCTURE ALGORITHMS AND APPLICATIONS (CT- 159)

Lecture – 4
Elementary Sorting Techniques:
Bubble Sort, Insertion Sort, Selection Sort

Instructor: Engr. Nasr Kamal

Department of Computer Science and Information Technology

Sorting

Sorting in data structures refers to the process of arranging data in a specific order, typically in ascending or descending order. This is a fundamental operation in computer science and has numerous applications, including:

1. **Data Organization**: Sorting data helps in organizing it in a way that makes it easier to access and process. For example, sorting a list of names alphabetically can make searching for a specific name faster.
2. **Searching Efficiency**: Many searching algorithms, such as binary search, require the data to be sorted. Sorted data can significantly improve search efficiency compared to unsorted data.
3. **Data Analysis**: Sorted data is crucial for various analytical operations, such as finding the median, calculating percentiles, or generating frequency distributions.
4. **Simplifying Algorithms**: Certain algorithms, like merge operations or certain types of dynamic programming solutions, become simpler and more efficient when applied to sorted data.

Types of Sorting Algorithms

Sorting algorithms can be classified into different categories based on their methods and efficiency:

Comparison-Based Sorting Algorithms:

- **Bubble Sort**: Compares adjacent elements and swaps them if necessary.
- **Selection Sort**: Selects the smallest (or largest) element and places it in its correct position.
- **Insertion Sort**: Builds the sorted list one item at a time by inserting elements into their correct position.
- **Merge Sort**: Divides the list into halves, sorts each half, and then merges them.
- **Quick Sort**: Partitions the list around a pivot element and recursively sorts the partitions.
- **Heap Sort**: Uses a heap data structure to sort elements.

Bubble Sort

- Bubble Sort is one of the simplest and most intuitive sorting algorithms.
- It repeatedly steps through a list, compares adjacent elements, and swaps them if they are in the wrong order.
- This process is repeated until the list is sorted

How Bubble Sort Works

1. **Compare Adjacent Elements**: Start at the beginning of the list and compare the first two elements.
2. **Swap if Necessary**: If the first element is greater than the second element (for ascending order), swap them.
3. **Move to Next Pair**: Move to the next pair of adjacent elements and repeat the comparison and swap if necessary.
4. **Pass through the List**: Continue this process until you reach the end of the list. After the first pass, the largest element will have "bubbled up" to its correct position at the end of the list.
5. **Repeat**: Repeat the process for the remaining elements (excluding the last sorted elements) until the list is completely sorted.

Pseudocode of Bubble Sort

```
function bubbleSort(arr):
    n = length of arr
    for i from 0 to n-1:
    swapped = false
    for j from 0 to n-i-2:
        if arr[j] > arr[j+1]:
        swap arr[j] and arr[j+1]
        swapped = true
    if not swapped:
        break
```

.

Example

Consider an array [5, 3, 8, 4, 2].

First Pass:

Compare 5 and 3 → Swap → [3, 5, 8, 4, 2]

Compare 5 and 8 → No Swap → [3, 5, 8, 4, 2]

Compare 8 and 4 → Swap → [3, 5, 4, 8, 2]

Compare 8 and 2 → Swap → [3, 5, 4, 2, 8]

Second Pass:

Compare 3 and 5 → No Swap → [3, 5, 4, 2, 8]

Compare 5 and 4 → Swap → [3, 4, 5, 2, 8]

Compare 5 and 2 → Swap → [3, 4, 2, 5, 8]

Third Pass:

Compare 3 and 4 → No Swap → [3, 4, 2, 5, 8]

Compare 4 and 2 → Swap → [3, 2, 4, 5, 8]

Fourth Pass:

Compare 3 and 2 → Swap → [2, 3, 4, 5, 8]

No swaps are needed in the final pass, so the algorithm stops. T

he sorted array is [2, 3, 4, 5, 8].

Time Complexity

- **Best Case**: O(n) with optimization (already sorted list).
- **Average Case**: O(n^2) (random order).
- **Worst Case**: O(n^2) (reverse order).

While bubble sort is simple and easy to understand, its quadratic time complexity makes it inefficient for large datasets compared to more advanced sorting algorithms like quicksort, mergesort, or heapsort, which have better average and worst-case time complexities.

Insertion Sort

- Insertion sort is a simple and intuitive sorting algorithm that builds the final sorted array one item at a time.
- It's much like sorting playing cards in your hands—taking one card at a time and inserting it into its proper position among the already sorted cards.

How Insertion Sort Works

1. **Start with the second element** of the list (since a single-element list is already sorted).
2. **Compare this element** with the elements before it (which are already sorted).
3. **Shift elements** that are greater than the current element one position to the right.
4. **Insert the current element** into its correct position in the sorted portion of the list.
5. **Repeat** this process for each element in the list.

Pseudocode of Insertion Sort

```
function insertionSort(arr):
    for i from 1 to length of arr – 1:
    key = arr[i]
    j = i – 1
    while j >= 0 and arr[j] > key:
        arr[j + 1] = arr[j]
        j = j – 1
    arr[j + 1] = key
```

Example

Let's sort the array `[5, 3, 8, 4, 2]` using insertion sort:

Initial Array: `[5, 3, 8, 4, 2]`

Iteration 1 (`i = 1`, `key = 3`):

Compare `3` with `5` (which is at index `0`). Since `3 < 5`, shift `5` to the right.

Array after shifting: `[5, 5, 8, 4, 2]`

Insert `3` into the correct position: `[3, 5, 8, 4, 2]`

Iteration 2 (`i = 2`, `key = 8`):

Compare `8` with `5`. Since `8 > 5`, no shifting is needed.

Array remains: `[3, 5, 8, 4, 2]`

Example

Iteration 3 (`i = 3`, `key = 4`):

Compare `4` with `8`. Since `4 < 8`, shift `8` to the right.

Array after shifting: `[3, 5, 8, 8, 2]`

Compare `4` with `5`. Since `4 < 5`, shift `5` to the right.

Array after shifting: `[3, 5, 5, 8, 2]`

Insert `4` into the correct position: `[3, 4, 5, 8, 2]`

Iteration 4 (`i = 4`, `key = 2`):

Compare `2` with `8`. Since `2 < 8`, shift `8` to the right.

Array after shifting: `[3, 4, 5, 8, 8]`

Compare `2` with `5`. Since `2 < 5`, shift `5` to the right.

Array after shifting: `[3, 4, 5, 5, 8]`

Example

Iteration 4 (`i = 4`, `key = 2`):

Compare `2` with `8`. Since `2 < 8`, shift `8` to the right.

Array after shifting: `[3, 4, 5, 8, 8]`

Compare `2` with `5`. Since `2 < 5`, shift `5` to the right.

Array after shifting: `[3, 4, 5, 5, 8]`

Compare `2` with `4`. Since `2 < 4`, shift `4` to the right.

Array after shifting: `[3, 4, 4, 5, 8]`

Compare `2` with `3`. Since `2 < 3`, shift `3` to the right.

Array after shifting: `[3, 3, 4, 5, 8]`

Insert `2` into the correct position: `[2, 3, 4, 5, 8]`

The final sorted array is `[2, 3, 4, 5, 8]`.

Time Complexity

Best Case: O(n) when the array is already sorted.

Average Case: O(n^2) when the array is in random order.

Worst Case: O(n^2) when the array is sorted in reverse order.

Insertion sort is efficient for small datasets or nearly sorted data, but less suitable for large datasets compared to more advanced sorting algorithms.

Selection Sort

- Selection sort is a straightforward comparison-based sorting algorithm that works by repeatedly selecting the smallest (or largest, depending on sorting order) element from the unsorted portion of the list and moving it to the beginning (or end) of the sorted portion.

- It has a simple implementation but is less effecient compared to more advanced sorting algorithms.

How Selection Sort Works

1. **Start with the first element** of the list and assume it to be the minimum.
2. **Compare this element** with the other elements in the list to find the smallest element.
3. **Swap** the smallest element found with the first element of the list.
4. Move to the next position in the list and repeat the process for the remaining unsorted portion.
5. **Continue** until the entire list is sorted.

Pseudocode of Selection Sort

```
function selectionSort(arr):
    n = length of arr
    for i from 0 to n-1:
      min_index = i
      for j from i+1 to n-1:
            if arr[j] < arr[min_index]:
                  min_index = j
      swap arr[i] and arr[min_index]
```

Example

Consider an array [64, 25, 12, 22, 11]. We will sort it using selection sort:

1. **Initial Array**: [64, 25, 12, 22, 11]
2. **First Pass (i = 0)**:
    - Assume the minimum element is at index 0 (value 64).
    - Compare 64 with elements at indices 1 to 4:
        - Compare 64 with 25. Minimum is 25.
        - Compare 25 with 12. Minimum is 12.
        - Compare 12 with 22. Minimum is 12.
        - Compare 12 with 11. Minimum is 11.
    - Swap 64 with 11.
    - Array after the swap: [11, 25, 12, 22, 64]

Example

Second Pass (i = 1):

- Assume the minimum element is at index 1 (value 25).
- Compare 25 with elements at indices 2 to 4:
  - Compare 25 with 12. Minimum is 12.
  - Compare 12 with 22. Minimum is 12.
  - Compare 12 with 64. Minimum is 12.
- Swap 25 with 12.
- Array after the swap: [11, 12, 25, 22, 64]

Third Pass (i = 2):

- Assume the minimum element is at index 2 (value 25).
- Compare 25 with elements at indices 3 to 4:
  - Compare 25 with 22. Minimum is 22.
  - Compare 22 with 64. Minimum is 22.
- Swap 25 with 22.
- Array after the swap: [11, 12, 22, 25, 64]

Example

Fourth Pass (i = 3):

- Assume the minimum element is at index 3 (value 25).
- Compare 25 with the element at index 4 (value 64). Minimum is 25.
- No swap needed as 25 is already in place.
- Array remains: [11, 12, 22, 25, 64]

The array is now fully sorted: [11, 12, 22, 25, 64].

Time Complexity

- **Best Case**: O(n^2) when the array is already sorted.
- **Average Case**: O(n^2) for random order.
- **Worst Case**: O(n^2) when the array is sorted in reverse order.

Selection sort is easy to implement and understand, but it is inefficient for large lists because of its O(n^2) time complexity. It performs well for small lists or nearly sorted data, and it is notable for its simplicity and the fact that it performs a minimal number of swaps (at most n−1swaps).