ESSENTIAL PYTHON FOR DATA STRUCTURE COURSE

TRAINER : HASNAIN AHMAD

ICODE GURU

Sorting Algorithms

SORTING ALGORITHMS

- *1 : Bubble Sort*
- *2 : Insertion Sort*
- *3 : Selection Sort*
- *4 : Merge Sort*
- *5 : Quick Sort*
- *6 : Counting Sort*
- *7 : Tim Sort*
- *8 : Radix Sort*
- *9 : Bucket Sort*
- *10: Heap Sort*
- *11: TREE SORT*
- *12: SHELL SORT*

SELECTION SORT

- **Selection Sort**
- **Concept:** Repeatedly find the minimum element from the unsorted part and put it in the correct position.
- **Steps:**
  - Loop i = 0 → n-1
  - Find the minimum element in the remaining array
  - Swap it with arr[i]
- **Time Complexity:**
  - Best: $O(n^2)$
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- **Space:** $O(1)$
- **Stability:** ❌ Not Stable

BUBBLE SORT

1. **Bubble Sort**

- **Concept:** Compare adjacent elements and swap if they are in the wrong order. Repeat until the array is sorted .
- **Steps:**
  - Start from index 0
  - Compare arr[i] and arr[i+1]
  - Swap if needed
  - Largest element “bubbles up” to the end after each pass
- **Time Complexity:**
  - Best: $O(n)$ (if already sorted)
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- **Space:** $O(1)$ (in-place)
- **Stability:** ✅ Stable

INSERTION SORT

- **Insertion Sort**
- **Concept:** Build the final sorted array one element at a time by inserting elements into their correct position.
- **Steps:**
  - Start from index 1
  - Pick arr[i]
  - Insert it into the correct position among the previous elements
- **Time Complexity:**
  - Best: $O(n)$ (if already sorted)
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- **Space:** $O(1)$
- **Stability:** ✅ Stable

MERGE SORT

- **. Merge Sort**
- **Concept:** Divide & Conquer algorithm. Split array into halves, sort each half, then merge them.
- **Steps:**
  - Divide array into two halves
  - Recursively sort both halves
  - Merge the two sorted halves
- **Time Complexity:**
  - Best: O(n log n)
  - Average: O(n log n)
  - Worst: O(n log n)
- **Space:** O(n) (extra space for merging)
- **Stability:** ✅ Stable

QUICK SORT

- **Quick Sort**
- **Concept:** Choose a pivot, partition array into two parts (left < pivot, right > pivot), then recursively sort both sides.
- **Steps:**
  - Choose pivot (first, last, random, or median)
  - Partition elements around pivot
  - Recursively quick sort left and right parts
- **Time Complexity:**
  - Best: $O(n \log n)$
  - Average: $O(n \log n)$
  - Worst: $O(n^2)$ (bad pivot choices)
- **Space:** $O(\log n)$ (recursion stack)
- **Stability:** ❌ Not Stable

HEAP SORT

- **Heap Sort**
- **Concept:** Use a Binary Heap. Build a max-heap, then repeatedly remove the root (max element) and heapify again.
- **Steps:**
  - Build max-heap from array
  - Swap root with last element
  - Reduce heap size and heapify
- **Time Complexity:**
  - Best: O(n log n)
  - Average: O(n log n)
  - Worst: O(n log n)
- **Space:** O(1)
- **Stability:** ❌ Not Stable

SHELL SORT

- **Shell Sort**
- **Concept:** Optimized version of Insertion Sort. Compare elements far apart using a gap, reduce the gap until it becomes 1.
- **Time Complexity:**
  - Best: $O(n \log n)$
  - Average: $O(n (\log n)^2)$
  - Worst: $O(n (\log n)^2)$
- **Space:** $O(1)$
- **Stability:** ❌ Not Stable

COUNTING SORT

- **Counting Sort**
- **Concept:** Works for integers in a limited range. Count frequency of each element, then use prefix sums to place them in sorted order.
- **Time Complexity:**
  - Best: O(n + k)
  - Average: O(n + k)
  - Worst: O(n + k)
- **Space:** O(n + k)
- **Stability:** ✅ Stable

RADIX SORT

- **Radix Sort**
- **Concept:** Sort numbers digit by digit (least or most significant digit). Often uses Counting Sort inside.
- **Time Complexity:**
  - Best: O(nk)
  - Average: O(nk)
  - Worst: O(nk)
    (k = number of digits)
- **Space:** O(n + k)
- **Stability:** ✅ Stable

BUCKET SORT

- **Bucket Sort**
- **Concept:** Distribute elements into multiple buckets, sort each bucket individually, and then merge all buckets.
- **Time Complexity:**
  - Best: $O(n + k)$
  - Average: $O(n + k)$
  - Worst: $O(n^2)$ (if all elements go into one bucket)
- **Space:** $O(n + k)$
- **Stability:** Depends on sub-sort

TREE SORT

- **Tree Sort**
- **Concept:** Insert elements into a Binary Search Tree (BST), then do an in-order traversal.
- **Time Complexity:**
  - Best: $O(n \log n)$
  - Average: $O(n \log n)$
  - Worst: $O(n^2)$ (if tree becomes skewed)
- **Space:** $O(n)$ (for BST nodes)
- **Stability:** ❌ Not Stable

TIM SORT

Tim Sort

•**Concept:** A hybrid algorithm (Merge Sort + Insertion Sort). Used in Python (sorted(), .sort()) and Java.

•.

•**Time Complexity:**

- •Best: O(n)
- •Average: O(n log n)
- •Worst: O(n log n)

•**Space:** O(n)

•**Stability:** ✅ Stable

KEY POINTS

- ⚡ Key POINTS
- 1: Comparison-based algorithms: Bubble, Selection, Insertion, Merge, Quick, Heap, Shell, Tree.

  Lower bound: $\Omega(n \log n)$.
- 2 :Non-comparison-based algorithms: Counting, Radix, Bucket.

  Can be faster but only for integers / special cases.
- Stable algorithms: Bubble, Insertion, Merge, Counting, Radix, Tim.
- Not stable: Quick, Heap, Selection, Tree, Shell.

THANKS

TRAINER : HASNAIN AHMAD