Sorting Techniques

Sorting is one of the fundamental operations in computer science.

---

1. Bubble Sort

Overview: Repeatedly compares adjacent elements and swaps them if they are in the wrong order. It "bubbles" the largest element to the end of the list.

- **Time Complexity**:
  - Best case: O(n) (if already sorted).
  - Worst case: $O(n^2)$.
- **Space Complexity**: O(1) (in-place).
- **Best for**: Educational purposes or extremely small datasets.
- **Why**:
  - Easy to implement and understand.
- **Limitations**:
  - Time complexity: $O(n^2)$ for all cases.
- Highly inefficient for large datasets.

Python Implementation

```
def bubble_sort(arr):
  n = len(arr)
  for i in range(n):
    swapped = False
    for j in range(0, n - i - 1):  # Last i elements are already sorted
      if arr[j] > arr[j + 1]:
        arr[j], arr[j + 1] = arr[j + 1], arr[j]
        swapped = True
    if not swapped:  # Break if no swaps were made (array is sorted)
      break
  return arr
```

```
print(bubble_sort([5, 3, 8, 6, 2]))
```

---

2. Selection Sort

Overview: Repeatedly finds the smallest (or largest) element in the unsorted part of the list and places it in its correct position.

- **Time Complexity**: $O(n^2)$ (always, as comparisons don't depend on order).
- **Space Complexity**:$O(1)$ (in-place).

Python Implementation

```
def selection_sort(arr):
  n = len(arr)
  for i in range(n):
    min_index = i
    for j in range(i + 1, n):
      if arr[j] < arr[min_index]:
        min_index = j
    arr[i], arr[min_index] = arr[min_index], arr[i]
  return arr


print(selection_sort([5, 3, 8, 6, 2]))
```

---

3. Insertion Sort

Overview: Builds the sorted array one element at a time by picking elements and inserting them into their correct position.

- **Time Complexity**:
  - Best case: $O(n)$ (if already sorted).
  - Worst case: $O(n^2)$.
- **Space Complexity**: $O(1)$ (in-place).
- **Best for**: Small or nearly sorted datasets.
- **Why**:

   - Simple and efficient for small input sizes (O(n) for nearly sorted data).
   - In-place and stable.
- **Limitations**:
   - Time complexity: $O(n^2)$ for large, unsorted datasets.

Python Implementation

```
def insertion_sort(arr):
  for i in range(1, len(arr)):
    key = arr[i]
    j = i - 1
    while j >= 0 and key < arr[j]:
      arr[j + 1] = arr[j]
      j -= 1
    arr[j + 1] = key
  return arr


print(insertion_sort([5, 3, 8, 6, 2]))
```

---

4. Merge Sort

Overview: A divide-and-conquer algorithm that splits the array into halves, sorts each half recursively, and then merges them.

- **Time Complexity**: O(n log n) (always).
- **Space Complexity**: O(n) (due to temporary arrays).
- **Best for**: Large datasets requiring stable sorting or when memory is not a concern.
- **Why**:
   - Always O(n logn) time complexity.
   - Stable (maintains relative order of equal elements).
- **Limitations**:
   - Requires extra memory (O(n)) for temporary arrays.

Python Implementation

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

print(merge_sort([5, 3, 8, 6, 2]))
```

5. Quick Sort

Overview: A divide-and-conquer algorithm that selects a "pivot" and partitions the array such that all smaller elements are on one side and larger elements on the other.

- **Time Complexity**:
  - Best/Average case: O(n log n).
  - Worst case: $O(n^2)$ (if pivot is poorly chosen, e.g., smallest or largest element).
- **Space Complexity**: O(log n) (due to recursion).
- **Best for**: Large datasets in general-purpose applications.
- **Why**:
  - Average time complexity: O(n logn).
  - In-place (requires little additional memory).
  - Efficient for random datasets.
- **Limitations**:
  - Worst-case time complexity: $O(n^2)$ (can be mitigated with randomized pivot selection).

Python Implementation

```
def quick_sort(arr):
  if len(arr) <= 1:
    return arr

  pivot = arr[len(arr) // 2]
  left = [x for x in arr if x < pivot]
  middle = [x for x in arr if x == pivot]
  right = [x for x in arr if x > pivot]

  return quick_sort(left) + middle + quick_sort(right)

print(quick_sort([5, 3, 8, 6, 2]))
```

---

6. Heap Sort

Overview: Builds a binary heap and repeatedly extracts the maximum (or minimum) element.

- **Time Complexity**: O(n log n) (always).
- **Space Complexity**: O(1) (in-place).
- **Best for**: Memory-constrained systems where in-place sorting is required.
- **Why**:
  - Time complexity: O(n logn) (worst-case, best-case, and average-case).
  - In-place and does not require additional memory.
- **Limitations**:
  - Not stable (relative order of equal elements is not preserved).

  Slower than Quick Sort in practice.

Python Implementation

```
def heapify(arr, n, i):
  largest = i
  left = 2 * i + 1
  right = 2 * i + 2

  if left < n and arr[left] > arr[largest]:
    largest = left
  if right < n and arr[right] > arr[largest]:
    largest = right
  if largest != i:
    arr[i], arr[largest] = arr[largest], arr[i]
    heapify(arr, n, largest)

def heap_sort(arr):
  n = len(arr)

  for i in range(n // 2 - 1, -1, -1):
```

```
      heapify(arr, n, i)


   for i in range(n - 1, 0, -1):
      arr[i], arr[0] = arr[0], arr[i]
      heapify(arr, i, 0)


   return arr


print(heap_sort([5, 3, 8, 6, 2]))
```

---

7. Counting Sort

- **Best for**: Sorting integers or datasets with a small range of values.
- **Why**:
  - Time complexity: O(n+k)O(n + k), where kk is the range of input.
  - Very fast for small ranges.
- **Limitations**:
  - Requires additional memory (O(k)O(k)).
  - Only works for discrete keys (e.g., integers).

---

8. Radix Sort

- **Best for**: Large datasets with integers or strings.
- **Why**:
  - Time complexity: O(nk), where kk is the number of digits or characters.
  - Can be faster than O(n logn) for specific datasets.
- **Limitations**:
  - Requires extra memory for intermediate buckets.
  - Not comparison-based.

Comparison of Sorting Techniques

Algorithm	Time Complexity (Best)	Time Complexity (Worst)	Space Complexity	Stable?
Bubble Sort	O(n)	O(n^2)	O(1)	Yes
Selection Sort	O(n^2)	O(n^2)	O(1)	No
Insertion Sort	O(n)	O(n^2)	O(1)	Yes
Merge Sort	O(n log n)	O(n log n)	O(n)	Yes
Quick Sort	O(n log n)	O(n^2)	O(log n)	No
Heap Sort	O(n log n)	O(n log n)	O(1)O(1)	No

The "best" sorting technique depends on the context, including factors like the size of the dataset, memory constraints, and whether the data is already partially sorted.

Best Sorting Algorithm by Use Case

Use Case	Recommended Algorithm
Small datasets	**Insertion Sort**
Large, general-purpose datasets	**Quick Sort**
Stable sorting required	**Merge Sort**
Memory-constrained systems	**Heap Sort**
Integer datasets with small range	**Counting Sort**
Sorting integers/strings (large range)	**Radix Sort**
Nearly sorted datasets	**Insertion Sort**
Educating beginners	**Bubble Sort**