Data Structures & Algorithms Cheat Sheet (Theory Only)

---

1. Basic Terminologies in Data Structures & Algorithms

Data Structure: A way to organize and store data efficiently (e.g., Arrays, Linked Lists, Stacks, Queues).
Algorithm: A step-by-step procedure to solve a problem.
Time Complexity: Measures how the execution time of an algorithm grows with input size (n).
Space Complexity: Measures the amount of memory an algorithm requires.

---

2. Asymptotic Notations & Complexity Analysis

Asymptotic Notations describe the growth rate of an algorithm's time complexity.

Notation	Meaning	Example
O(n)	Worst-case (Upper bound)	Linear Search
Ω(n)	Best-case (Lower bound)	Insertion Sort (Best case)
Θ(n)	Average-case (Tight bound)	Merge Sort
O(1)	Constant Time Complexity	Accessing an array index
O(log n)	Logarithmic Time Complexity	Binary Search

Example Complexity Comparisons:

- **Best:** $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

---

3. Arrays & Operations

Array: A **fixed-size** data structure that stores elements of the same type.
Operations on Arrays:

- **Insertion:** Adding elements at a specific index.
- **Deletion:** Removing elements from an index.
- **Traversal:** Accessing all elements one by one.
- **Searching:** Finding an element in the array.
- **Sorting:** Arranging elements in increasing/decreasing order.

Example of Array Declaration in Python:

```
arr = [10, 20, 30, 40, 50]
print(arr[2])  # Output: 30
```

---

4. Searching Algorithms

(A) Linear Search (Sequential Search)

Searches element one by one from the start.
Time Complexity: O(n) (Worst case).
Example:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

arr = [10, 20, 30, 40]
```

```
print(linear_search(arr, 30))  # Output: 2
```

(B) Binary Search

Works only on sorted arrays by dividing the search space into halves.
Time Complexity: O(log n).
Example:

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

arr = [10, 20, 30, 40, 50]
print(binary_search(arr, 30))  # Output: 2
```

---

5. Sorting Algorithms

(A) Bubble Sort

Repeatedly **swaps adjacent elements** if they are in the wrong order.
Time Complexity: $O(n^2)$.
Example:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

arr = [5, 3, 8, 1, 2]
bubble_sort(arr)
print(arr)  # Output: [1, 2, 3, 5, 8]
```

(B) Selection Sort

Selects the minimum element and swaps it with the first element.
Time Complexity: $O(n^2)$.
Example:

```
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]
```

```
arr = [5, 3, 8, 1, 2]
selection_sort(arr)
print(arr)  # Output: [1, 2, 3, 5, 8]
```

(C) Insertion Sort

Builds the sorted array one element at a time.
Time Complexity: $O(n^2)$.
Efficient for small datasets.
Example:

```
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

arr = [5, 3, 8, 1, 2]
insertion_sort(arr)
print(arr)  # Output: [1, 2, 3, 5, 8]
```

(D) Heap Sort

Uses a **binary heap** to sort elements.
Time Complexity: $O(n \log n)$.
Example:

```
import heapq

def heap_sort(arr):
    heapq.heapify(arr)
    return [heapq.heappop(arr) for _ in range(len(arr))]

arr = [5, 3, 8, 1, 2]
sorted_arr = heap_sort(arr)
print(sorted_arr)  # Output: [1, 2, 3, 5, 8]
```

(E) Shell Sort

Improvement over insertion sort using gap sequences.
Time Complexity: $O(n \log n)$.
Example:

```
def shell_sort(arr):
    n = len(arr)
    gap = n // 2
    while gap > 0:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
```

```
arr = [5, 3, 8, 1, 2]
shell_sort(arr)
print(arr)  # Output: [1, 2, 3, 5, 8]
```

---

6. Performance & Comparison of Sorting Algorithms

Algorithm	Best Case	Worst Case	Average Case	Stable?	In-place?
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	Yes
Shell Sort	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log n)$	No	Yes

Bubble, Selection, and Insertion Sorts are simpler but inefficient for large datasets.
Heap Sort and Shell Sort are faster and efficient for large datasets.

Key Takeaways

Searching: Linear Search ($O(n)$) vs. Binary Search ($O(\log n)$).
Sorting: Bubble, Selection, Insertion ($O(n^2)$) vs. Heap, Shell ($O(n \log n)$).
Performance Comparison: Heap Sort is more efficient than simple sorting algorithms.

This **Data Structures & Algorithms Cheat Sheet** covers **terminologies, complexity analysis, searching, sorting, and performance comparison**. Let me know if you need further explanations!