00_Getting Started

- Template
- Sorting before Two Pointer
- BIG O Breakdown

Template

```
def solve_problem(inputs):
    # Step 1: Understand the Problem
    # - Parse inputs and outputs.
    # - Clarify constraints (e.g., time, space).
    # - Identify edge cases.

    # Step 2: Plan and Design
    # - Think about the brute-force approach.
    # - Optimize: Can you use dynamic programming, divide & conquer, etc.?
    # - Choose the appropriate data structures (e.g., arrays, hashmaps, heaps).

    # Step 3: Implement the Solution
    # - Use helper functions for modularity.
    # - Write clear, well-commented code.

    def helper_function(args):
        # Optional: For recursion, BFS, DFS, etc.
        pass

    # Main logic
    result = None  # Initialize result or output variable.

    # Example logic
    # for num in inputs:
    #     result += num  # Or other computations.

    return result

# Driver code (for testing locally)
if __name__ == "__main__":
    inputs = []  # Replace with example test cases.
    print(solve_problem(inputs))
```

Sorting before Two Pointer

The **total time complexity** is derived by analyzing each step of the algorithm and summing their individual complexities. Here's a detailed breakdown:

Steps of the Algorithm

1. **Sorting the Array (`arr.sort()`):**
   - The `sort()` function sorts the array in ascending order.
   - Sorting an array of size `n` takes **O(n log n)** using efficient sorting algorithms like Timsort (used in Python).
2. **Two-Pointer Traversal (the `while` loop):**
   - After sorting, the two-pointer technique is applied.
   - The two pointers (`left` and `right`) traverse the array at most once. In each iteration:
     - Either `left` is incremented, or `right` is decremented.
     - This ensures that the loop runs for at most **O(n)** iterations.

Combining the Steps

- Sorting takes **O(n log n)**.
- The two-pointer traversal takes **O(n)**.
- Since these steps are performed sequentially (not nested), their complexities are **added**:
  Total Time Complexity=O(nlogn)+O(n)\text{Total Time Complexity} = O(n \log n) + O(n)
  Total Time Complexity=O(nlogn)+O(n)

Simplifying the Complexity

- In Big O notation, only the **dominant term** matters as `n` grows large.
  - **O(n log n)** dominates **O(n)** because logarithmic growth adds a significant factor to linear growth.
- Therefore, the total complexity simplifies to: O(nlogn)O(n \log n)O(nlogn)

Why Approximation?

The symbol `≈` in **O(n log n) + O(n) ≈ O(n log n)** indicates that:

- The **O(n)** term is negligible compared to **O(n log n)** for large `n`.
- So the effective time complexity is considered **O(n log n)**.

Practical Example

Let’s assume `n = 1,000,000`:

- **Sorting:** nlogn=1,000,000·log21,000,000≈20,000,000n \log n = 1,000,000 \cdot \log_2 1,000,000 \approx 20,000,000nlogn=1,000,000·log21,000,000≈20,000,000 operations.
- **Traversal:** O(n)=1,000,000O(n) = 1,000,000O(n)=1,000,000 operations.
- Clearly, the sorting step dominates.

BIG O Breakdown

What is Big O Notation?

Big O notation is a mathematical way to describe the **time complexity** or **space complexity** of an algorithm. It represents the worst-case growth rate of an algorithm as the input size nn increases, helping us understand how scalable an algorithm is.

Key Characteristics of Big O:

1. **Focus on Growth**: Big O focuses on how the number of operations grows with the size of the input (nn).
   - Example: An algorithm that performs 2n+52n + 5 operations is O(n)O(n) because as nn grows, the constant 55 and coefficient 22 become negligible.
2. **Worst-Case Analysis**: It assumes the largest number of operations the algorithm might perform for the input size nn.
3. **Ignore Constants and Lower-Order Terms**:
   - Constants don't matter in Big O because they don't scale with nn.
   - Example: O(2n)=O(n)O(2n) = O(n), O(n+10)=O(n)O(n + 10) = O(n).
4. **Only the Dominant Term Counts**:
   - If an algorithm has multiple terms like O(n2+n)O(n^2 + n), the term with the fastest growth rate dominates. So, O(n2+n)=O(n2)O(n^2 + n) = O(n^2).

Common Big O Complexities

Complexity	Name	Description
O(1)O(1)	Constant	Takes the same amount of time regardless of input size.
O(logn)O(\log n)	Logarithmic	Reduces the problem size by half at every step.
O(n)O(n)	Linear	Time grows directly proportional to the input size.

Complexity	Name	Description
O(nlogn)O(n \log n)	Linearithmic	Common in efficient sorting algorithms like Merge Sort.
O(n2)O(n^2)	Quadratic	Common in nested loops, grows rapidly with input size.
O(2n)O(2^n)	Exponential	Doubles operations for each increment in input size.
O(n!)O(n!)	Factorial	Infeasible for even moderately large input sizes.

How to Calculate Big O

To calculate Big O, analyze the algorithm step by step, focusing on loops, function calls, and recursive depth.

---

1. Loops

- A loop that runs nn times is O(n)O(n).
- Nested loops multiply their complexities.
  - Example:

```
for i in range(n):        # O(n)
    for j in range(n):    # O(n)
        print(i, j)       # O(1)
```

Total = O(n)·O(n)=O(n2)O(n) \cdot O(n) = O(n^2).

---

2. Sequential Steps

- If an algorithm has multiple parts, add their complexities.
  - Example:

```
for i in range(n):    # O(n)
    print(i)          # O(1)


for j in range(m):    # O(m)
```

```
    print(j)        # O(1)
```

    Total = O(n)+O(m)O(n) + O(m).
- If n=mn = m, the total is O(n)+O(n)=O(2n)=O(n)O(n) + O(n) = O(2n) = O(n).

---

3. Function Calls

- If a function is called recursively, analyze how many times it is called and the work done in each call.
  - Example (Binary Search):

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

    - Binary search splits the array into halves, so its complexity is O(logn)O(\log n).

---

4. Recurrence Relations

- Recursive algorithms often have recurrence relations.
  - Example (Merge Sort):
    - Merge Sort divides the array into halves and merges them back.
    - Relation: T(n)=2T(n/2)+O(n)T(n) = 2T(n/2) + O(n) (two recursive calls + merge operation).
    - Complexity: O(nlogn)O(n \log n).

Practical Examples

Example 1: Single Loop

```
for i in range(n):
    print(i)  # O(1)
```

- Total = O(n)O(n).

Example 2: Nested Loop

```
for i in range(n):
    for j in range(n):
        print(i, j)  # O(1)
```

- Total = O(n)·O(n)=O(n2)O(n) \cdot O(n) = O(n^2).

Example 3: Sorting and Traversal

```
arr.sort()  # O(n log n)
for i in arr:
    print(i)  # O(n)
```

- Total = O(nlogn)+O(n)=O(nlogn)O(n \log n) + O(n) = O(n \log n).

Example 4: Binary Search

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

- Complexity: O(logn)O(\log n), because the array size is halved in each recursive call.

Tips for Calculating Big O

1. **Focus on Loops:** Count how many times each loop runs.
2. **Break Down Steps:** Analyze each segment of the algorithm separately.
3. **Remove Constants:** Ignore constants and lower-order terms.
4. **Understand Recursive Depth:** For recursive algorithms, determine how the input size shrinks with each call.

With practice, determining Big O becomes intuitive!