Revision Notes: Introduction to Arrays, Time and Space Complexity

Class Outline

1. **Introduction to Arrays**: Basics and characteristics of arrays as data structures.
2. **Time and Space Complexity**: Understanding the importance and calculation of complexity for algorithms.
3. **Reversing and Rotating Arrays**: Algorithms and approaches for manipulating arrays.
4. **Dynamic Arrays**: Concept and internal working.
5. **Importance of Constraints**: How constraints affect algorithm efficiency.

---

1. Introduction to Arrays

Arrays are a basic data structure that store elements of the same type in contiguous memory locations, allowing efficient element retrieval using indices.

Key Characteristics:

- **Constant Time Access**: Accessing any element by its index is O(1) due to direct referencing in memory【4:12†transcript.txt】.
- **Fixed Size upon Declaration**: Once an array's size is set during declaration, it cannot be changed in static arrays【4:9†transcript.txt】.

Indexing:

- Arrays use zero-based indexing, meaning the first element is at index 0 and the last element (if n elements) is at index n-1【4:12†transcript.txt】.

---

2. Time and Space Complexity

Time Complexity:

- **O(n)**: Common for operations that require iteration through an array.
- **O(1)**: Operations like accessing an element by index【4:5†transcript.txt】.

Space Complexity:

- Refers to the amount of memory an algorithm uses relative to input size.
- **O(1)**: When space usage is constant regardless of input size 【4:6†transcript.txt】.
- **O(n)**: When additional space used scales linearly with input size, like creating an auxiliary array【4:8†transcript.txt】.

Big O Notation:

- Represented as O(f(n)), it abstracts the upper bound performance, ignoring constants and non-dominant terms【4:3†transcript.txt】 【4:18†transcript.txt】.

---

3. Reversing and Rotating Arrays

Reversing an Array:

- Use two-pointer technique: one starting from the beginning and the other from the end, swapping elements till they meet【4:15†transcript.txt】.

Rotating an Array:

- **Brute Force**: Involves moving the elements one by one, results in higher time complexity.
- **Optimized Approach**: Rotate using reverse operations. First reverse the entire array, reverse the first k elements, then reverse the last n-k elements【4:16†transcript.txt】 【4:17†transcript.txt】.

Algorithm steps:

1. Reverse the entire array.

Handling Rotations:

- If total rotations (k) exceed the array size (n), simplify using `k = k % n` 【4:17†transcript.txt】.

---

4. Dynamic Arrays

Unlike static arrays, dynamic arrays adjust their capacity automatically.

Key Features:

- **Variable Size**: Can grow as needed, often by doubling the capacity.
- **Costly Resizes**: Doubling size incurs O(n) cost as current elements are copied to a new larger array. However, most insertions are O(1) amortized 【4:1†transcript.txt】 【4:9†transcript.txt】.

Implementation in Languages:

- **Java**: `ArrayList`
- **C++**: `std::vector`
- **Python**: Built-in `list` 【4:0†typed.md】.

---

5. Importance of Constraints

Constraints give key insight into the feasible complexity for a given problem:

- **Understanding Limits**: Helps decide whether an O(n^2) or O(n log n) solution will run efficiently within the given input limits.
- **Avoiding Time Limit Exceeded (TLE)**: Choose appropriate algorithms based on input size limits 【4:3†transcript.txt】 【4:18†transcript.txt】.

Process Outline for Problem Solving:

2. **Develop Logic & Brute Force Solution**.
3. **Optimize Using Observations & Data Structures**.
4. **Ensure Feasibility with Constraints**【4:4†transcript.txt】【4:14†typed.md】.

Example:

For `N <= 10^5`, an O(N log N) solution might be feasible, while O(N^2) could lead to inefficiencies【4:19†typed.md】.

These notes summarize the key points from the class focused on array manipulations and algorithm complexity, underscoring their importance in software engineering. Always review the constraints and complexities while designing solutions.