**Detailed Revision Notes on Space Complexity and Array Rotations**

**1. Introduction to Space Complexity**

**Definition**

Space Complexity is the total amount of memory space that an algorithm or a computer program uses during its execution. The space complexity includes both the space required for the input to the algorithm and the auxiliary space required for the algorithm itself or output storage:

* **Input Space**: Space taken by the input variables.
* **Auxiliary Space**: Space taken by extra variables used in the algorithm.
* **Output Space**: Space used to store the resulting output.

**Big O Notation**

Similar to time complexity, space complexity can be expressed using Big O notation, which describes the upper bound of the space required in relation to the input size. For example:

* **O(1)**: Constant space.
* **O(n)**: Linear space dependent on the size of the input n.
* **O(n^2)**: Quadratic space, etc.

**Examples**

1. **Constant Space Complexity**
2. func(int N) { // 4 bytes
3. int x; // 4 bytes
4. int y; // 4 bytes
5. long z; // 8 bytes
6. }

**Space Complexity**: O(1) since variables take constant space irrespective of input.

1. **Linear Space Complexity**
2. func(int N) {
3. int arr[N]; // 4 \* N bytes
4. }

**Space Complexity**: O(n) because the space depends linearly on the input size.

1. **Quadratic Space Complexity**
2. func(int N) {
3. long l[N][N]; // 8 \* N \* N bytes
4. }

**Space Complexity**: O(n^2) since we're using a 2D array of size N x N.

**2. Calculating Space Complexity**

**Step-by-Step Process**

1. **Identify Variables and Data Structures:**
   * Recognize all the variables, arrays, or data structures used.
2. **Calculate Individual Space Requirements:**
   * Calculate the space each variable or structure will take.
3. **Sum Up the Total Space:**
   * Exclude input space unless explicitly required.
4. **Convert to Big O Notation:**
   * Identify the term with the highest growth rate and ignore lower-order terms and constants.

**Example Calculation**

Consider the function:

func(int N) { // 4 bytes

int arr[10]; // 40 bytes

int x = N; // 4 bytes

int y = x \* x; // 4 bytes

long z = x + y; // 8 bytes

int arr2[N]; // 4 \* N bytes

long l[N][N]; // 8 \* N \* N bytes

}

* **Space Calculation**:
  + int x: 4 bytes
  + int y: 4 bytes
  + long z: 8 bytes
  + int arr[10]: 40 bytes
  + int arr2[N]: 4 \* N bytes
  + long l[N][N]: 8 \* N \* N bytes
* **Total Space**: 56 + 4N + 8N^2
* **Big O Notation**: O(N^2)【8:11†source】【8:16†source】.

**3. Auxiliary Space Complexity**

Auxiliary space refers specifically to the extra space or temporary space used by an algorithm, excluding the space taken by the inputs:

* Example:
* function maxArr(int arr[], int N) {
* int ans = arr[0];
* for(int i = 1; i < N; i++) {
* ans = max(ans, arr[i]);
* }
* return ans;
* }

**Auxiliary Space**: O(1) because it uses a few extra variables regardless of input size【8:10†source】【8:12†source】.

**4. Space Complexity in Different Scenarios**

* **Loop Construct**:
  + When loops are nested, consider the auxiliary space inside the loop scope.
  + Space within the loop is reallocated on each iteration and does not accumulate.

**5. Understanding Array Rotations**

**Problem Statement**

For array rotations, we need to move elements in the array a certain number of times to the left or the right.

**Rotation by Reversing (Optimized Approach)**

1. **Reverse the Entire Array**:
   * This shifts elements around but keeps them orderly.
2. **Reverse the First k Elements**:
   * Corrects the order for elements that have come to the front.
3. **Reverse the Remaining Elements**:
   * Fixes the order for the remaining array elements that moved to the back.

**Example**

Original Array: [1, 2, 3, 4, 5, 6]

* **Rotate by 4**:
  + Reverse Entire Array: [6, 5, 4, 3, 2, 1]
  + Reverse First 4 Elements: [3, 4, 5, 6, 2, 1]
  + Reverse Remaining Elements: [3, 4, 5, 6, 1, 2]

**Code**

void rotate(int arr[], int n, int k) {

reverse(arr, 0, n-1); // reverse entire array

reverse(arr, 0, k-1); // reverse first k elements

reverse(arr, k, n-1); // reverse remaining elements

}

**Time Complexity**: O(n) **Space Complexity**: O(1) (since only constant additional space is used)【8:6†source】【8:5†source】【8:17†source】.

**6. Edge Cases in Array Rotations**

* If the number of rotations k is greater than array size n:
  + Perform k = k % n to reduce the number of rotations within bounds.
* **Handling Zero or Negative Rotations**:
  + Convert negative rotations to positive by adding n.

By understanding these concepts and validating through multiple examples and code snippets, we can effectively manage space complexity and optimize array manipulations for efficient algorithms.

Space Complexity:

Maximum space utilizes at any point in time by an algorithm.

Auxiliary Space:

Extra space used by the algorithm excluding input and output space.

Big O Notation:

Represents the upper bound of an algorithm’s time or space complexity.

Constant Space Complexity:

O (1), space complexity that doesn’t depend on input size.

Array Rotation:

Process of shifting elements of an array by k positions.

Time Complexity:

The computational complexity that describes the amount of time it takes to run an algorithm.

TLE:

Time limit exceed, an error indicating the algorithm exceeded maximum allowed time.

Reverse Function:

Function to reverse elements in a specific range within an array

N2 Time complexity:

Quadratic time complexity, worst case scenario of nested loops over a dataset.

Matrix:

2D array where elements are addressed using two indices.

Array indexing

Accessing elements in an array using their position starting from 0.

Pseudocode:

An informal description of the algorithm meant to represent its logic.

JSON Validation:

Ensure that the JSON data structure follows the correct format and syntax.

Key-Value Pairs:

The basic building blocks of JSON consist of key-value pairs, where keys are strings and values can be various data types.

Linear Time Complexity:

Operations that have a time complexity of O(n), indicating the time grows linearly with input size.

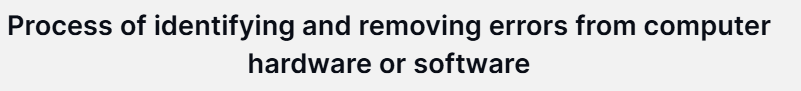
Quadratic Time Complexity:

Operations that take O(n2) time, typically associated with nested loops iterating over the same input.

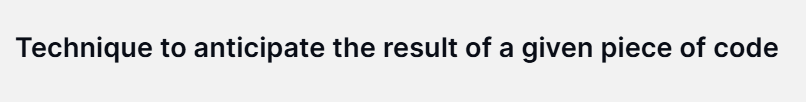
Logarithmic Time Complexity:

Operations that take O (Log n) time, often seen in algorithms that divide the problem size in half at each step.

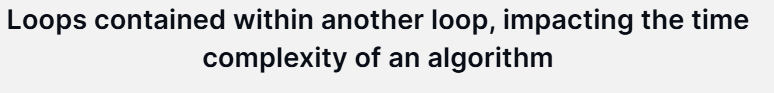
Debugging:



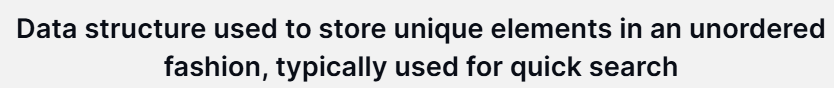
:

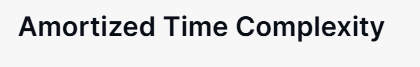


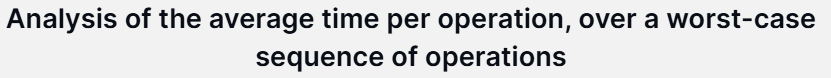












**Revision Notes for Intermediate DSA: Time Complexity, Space Complexity, Output Prediction & Debugging**

**Introduction**

In this session, we explored key concepts in software engineering related to the efficiency of algorithms and debugging techniques. The topics covered included time and space complexity, output prediction, and basic debugging strategies. The session was structured around solving practical problems and predicting outcomes for code snippets.

**1. Time Complexity Analysis**

**1.1 Constant Time Complexity (O(1))**

* **Example Code**:
* function constant():
* print("Hello, World!")
  + **Time Complexity**: O(1)
  + **Space Complexity**: O(1)

This indicates that the operation takes a fixed amount of time, regardless of the input size 【4:1†source】.

**1.2 Linear Time Complexity (O(N))**

* **Example Code**:
* function iterate(N):
* for i from 1 to N:
* print(i)
  + **Time Complexity**: O(N)
  + **Space Complexity**: O(1)

This represents a linear relationship between the input size and the operation time. As N increases, the time taken increases linearly 【4:1†source】.

**1.3 Quadratic Time Complexity (O(N²))**

* **Example Code**:
* function analyze(N):
* for i from 1 to N:
* for j from 1 to N:
* print(i, j)
  + **Time Complexity**: O(N²)
  + **Space Complexity**: O(1)

Involves nested iterations, where the execution time is proportional to the square of the input size 【4:1†source】.

**1.4 Logarithmic Time Complexity (O(Log N))**

* **Example Code**:
* function logScale(N):
* i = 1
* while i < N:
* print(i)
* i = i \* 2
  + **Time Complexity**: O(Log N)
  + **Space Complexity**: O(1)

This illustrates cases where the operations halve with each iteration, such as binary search 【4:1†source】.

**1.5 Effective Use of Big-O Notation**

* When faced with functions like O(2n + 100), constants and coefficients are ignored, reducing the expression to O(n), focusing on growth trends as the input size becomes very large【4:1†source】【4:11†source】.

**2. Space Complexity**

* Discussed mainly alongside time complexity, reminding that while the execution time might be critical, space usage should also be considered, especially for recursive algorithms or those using large data structures.

**3. Debugging Techniques**

**3.1 Identifying Loop Errors**

* **Infinite Loops**: Understanding the logic behind why a loop never ends is crucial. Properly incrementing loop counters or setting correct conditions will prevent this 【4:14†source】【4:18†source】.

**3.2 Conditional Errors**

* Syntax and logical errors surrounding if conditions can lead to incorrect flow paths within the code. Ensuring conditions evaluate correctly is essential for intended outcomes 【4:14†source】.

**4. Discussion on Assignments and Quizzes**

* The session involved interactive segments, including quizzes and assignments aimed at reinforcing the complexity concepts covered. These exercises were crafted to test understanding of complexity calculations and basic debugging abilities 【4:7†source】【4:15†source】.

**5. Optimizing Algorithms**

* Emphasized using data structures like hash sets to reduce complexity from quadratic to linear by eliminating nested loops when searching for elements within collections 【4:12†source】.

**Conclusion**

The session provided a comprehensive look at analysing and optimizing algorithms in terms of time and space complexity and honed debugging techniques essential for efficient coding practices. Understanding and applying these concepts is crucial for advanced software engineering and problem-solving in real-world scenarios.