Iteration Calculation:

Dividing a number by 2 until it reaches 1, calculating the steps as log base 2 of the number.

Algorithm comparison:

Evaluating efficiency using criteria like iterations to find better performing algorithms under certain conditions.

Time Complexity:

A computational complexity to estimate the time taken by an algorithm as the input size grows.

Big O Notation:

Describe the upper bound of the time complexity of an algorithm.

TLE (Time Limit Exceeded):

Occurs when an algorithm exceeds the maximum time limit set for program execution.

Logarithmic Function:

Inverse of exponential function, e.g., log base 2 of a number to find how many times it can be divided by 2 to reach 1.

Number of Iterations:

The repeated steps required by an algorithm, often estimated using logarithms.

Instruction per Iteration:

Estimating execution time by counting steps per loop iteration, leading to constraints.

Time Complexity Analysis:

Involves calculating complexity using higher order terms and ignoring constants and lower terms.

Exponential and logarithms:

Logarithms as the inverse function to exponentiation

Constant Time Complexity:

Algorithm’s execution time does not change with input size, denoted as O(1).

Lower order term:

In time complexity analysis, lower order terms are neglected for large input size estimation.

**Revision Notes: Software Engineering Class**

**Key Topics Covered**

1. **Approaching a Problem**  
   The process of tackling a software problem effectively involves several important steps:
   * **Understand the Problem Statement**: Thoroughly read the problem statement and the constraints provided.
   * **Formulate an Idea**: Develop a logical plan or algorithm based on your understanding. This involves thinking about the pseudo code and the loops you will need to use.
   * **Determine Time Complexity**: Assess whether the operations will complete within the expected time frame, aiming for less than *108108* iterations for maximum input size.
   * **Implementation**: If the developed approach meets the constraints, proceed with coding【4:0†source】【4:2†source】.
2. **Time Complexity**
   * The aim is to ensure that the solution remains efficient: typically, it should perform less than *108108* operations.
   * Common complexities inspected involve linear (*O(n)O(n)*), logarithmic (*O(log⁡n)O(logn)*), and quadratic (*O(n2)O(n2)*) times.
3. **Space Complexity**
   * Space complexity is about how much memory your algorithm uses at its peak during execution.
   * It is expressed in Big O notation, e.g., *O(1)O(1)* for constant space and *O(n)O(n)* for space proportional to input size【4:1†source】【4:2†source】【4:10†source】.
4. **Arrays in Software Engineering**
   * **Definition**: Arrays are collections of elements that are stored contiguously in memory. They are a primary data structure used frequently in programming.
   * **Indexing**: Starts from 0, meaning the first element is accessible via index 0, and complexity of accessing an element at a specific index is *O(1)O(1)*.
   * **Operations on Arrays**:
     + **Reversing Array**: Specific strategies can optimize this operation, like reversing parts and then recombining them efficiently【4:6†source】【4:11†source】【4:19†source】.

**Detailed Concepts**

**Problem Solving and Complexity Analysis**

* **Factors Problem Example**: Demonstrates deriving the time complexity by outlining the loop through numbers 1 to n to count factors. Opting for a loop from 1 to *nn​* greatly reduces the complexity to withstand higher limits (e.g., n can be *109109*)【4:0†source】【4:15†source】.

**Array Handling Techniques**

* **Printing Elements**: Achieved by traversing from start to end, resulting in *O(n)O(n)* time complexity.
* **Rotating Elements in the Array**: Utilize an efficient algorithm that performs in *O(n)O(n)* time and uses no extra space beyond the original array. This involves strategic reverse operations【4:6†source】【4:18†source】.

**Optimizing Space Usage**

* Reuse of space can significantly reduce unnecessary memory use. For instance, when dealing with segments of data, leveraging a smaller structure that gets reused is more efficient than storing everything simultaneously【4:8†source】.