**Time Complexity**

Time complexity is a measure of the amount of time an algorithm takes to run as a function of the input size. It helps us analyze and compare algorithms based on their efficiency and scalability. By understanding the time complexity of an algorithm, we can predict how its performance will change as the input size grows.

The Big O notation is commonly used to express the time complexity of an algorithm. It provides an upper bound on the growth rate of an algorithm's runtime in terms of the input size. Let's discuss some commonly encountered time complexities and their meanings:

**O(1) - Constant Time:** An algorithm is said to have constant time complexity if it takes the same amount of time to execute, regardless of the input size. It is the most efficient time complexity. For example, accessing an element in an array by index or performing a simple arithmetic operation takes constant time.

**O(n) - Linear Time:** Linear time complexity means that the algorithm's runtime grows proportionally to the input size. It implies that each element of the input needs to be processed once. Examples include iterating through an array or a linked list.

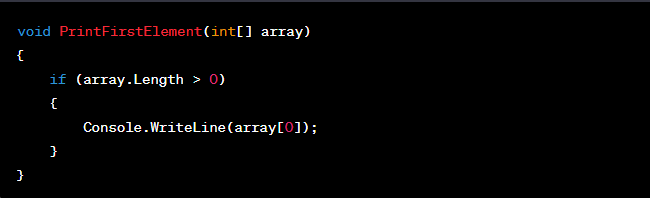
**O(n^2) - Quadratic Time:** Quadratic time complexity occurs when the runtime of an algorithm grows exponentially with the input size. This means that for every additional element in the input, the algorithm's runtime increases quadratically. Examples include nested loops where each loop iterates over the input.

**O(log n) - Logarithmic Time:** Algorithms with logarithmic time complexity often divide the input size in half at each step. They are efficient and commonly seen in binary search or certain divide-and-conquer algorithms. As the input size increases, the runtime of the algorithm grows slowly.

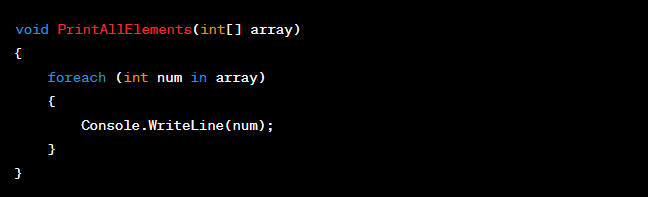
**O(n log n) - Linearithmic Time:** Linearithmic time complexity indicates that the algorithm's runtime grows linearly with the input size multiplied by the logarithm of the input size. Many efficient sorting algorithms, such as merge sort and quicksort, fall into this category.

**O(2^n) - Exponential Time:** Exponential time complexity signifies that the runtime of an algorithm grows exponentially with the input size. These algorithms are highly inefficient and become infeasible for large input sizes. Some brute-force algorithms or problems involving all combinations fall into this category.

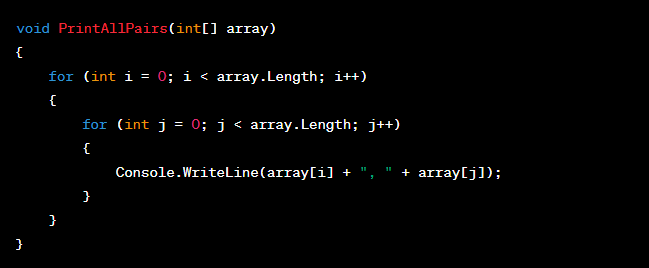
It's important to note that these are just a few examples, and there are many other time complexities that exist. When analyzing an algorithm's time complexity, we typically consider the *worst-case scenario*, as it provides an upper bound on the algorithm's runtime.

By analyzing and understanding the time complexity of an algorithm, we can make informed decisions when choosing the most efficient algorithm for a particular problem and optimize our code to improve performance.  
  
  
**Example 1:** Constant Time Complexity (O(1))  
  


In this example, the function **PrintFirstElement** prints the first element of an integer array. The code only performs a single operation, which is accessing the element at index 0. Regardless of the array size, the time taken to access the first element remains constant. Therefore, the time complexity of this code is **O(1) or constant time**.

**Example 2:** Linear Time Complexity (O(n))  
  


In this example, the function **PrintAllElements** prints all the elements of an integer array. It uses a foreach loop to iterate over each element and prints it. The time taken to execute this code is directly proportional to the size of the input array. Thus, the time complexity is **O(n) or linear time**, where n represents the size of the array.

**Example 3:** Quadratic Time Complexity (O(n^2))  
  


In this example, the function **PrintAllPairs** prints all possible pairs of elements in an integer array. It uses nested loops to iterate over the array and print the pairs. As the size of the array increases, the number of iterations in both loops grows quadratically. Therefore, the time complexity of this code is **O(n^2) or quadratic time**.

For more, please visit the following link:  
https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/