Polynomial running time refers to an algorithm's efficiency, where the time it takes to complete increases proportionally to a polynomial function of the input size. This means that if the input size is 'n', the algorithm's runtime can be described as O(n^k), where 'k' is a constant. Algorithms with polynomial running times are generally considered efficient and feasible for practical use, especially when compared to algorithms with exponential running times.

Here's a more detailed explanation:

Polynomial Function:

A polynomial function is an expression involving only non-negative integer powers of the input size 'n', such as n^2, n^3, or even n^100.

Big O Notation:

The notation O(n^k) is used to describe the upper bound of an algorithm's running time. It essentially means that the algorithm's runtime will not exceed a certain polynomial of the input size.

Examples:

Linear Time (O(n)): An algorithm that processes each element of the input once, such as iterating through a list.

Quadratic Time ($O(n^2)$): An algorithm that compares each element of the input with every other element, such as some sorting algorithms.

Cubic Time ($O(n^3)$): An algorithm with three nested loops, each iterating through the input size.

Tractability:

Polynomial time algorithms are often referred to as tractable or feasible because they can solve problems within a reasonable amount of time, even for large inputs.

Contrast with Exponential Time:

Algorithms with exponential running times (e.g., O(2^n)) become impractical very quickly as the input size increases, making them unsuitable for large datasets.

An exponential running time refers to an algorithm where the execution time increases exponentially with the size of the input. This means that for each additional unit of input, the time required to compute the result roughly doubles or multiplies by a larger factor. Algorithms with exponential running time are generally considered inefficient for large inputs.

Here's a more detailed explanation:

Key Characteristics:

Rapid Growth:

The most distinguishing feature is the rapid increase in computation time as the input grows.

Big O Notation:

Exponential running time is often represented using Big O notation, such as O(2^n), where 'n' is the size of the input. This indicates that the number of operations grows exponentially with the input size.

Examples:
Problems like the Traveling Salesperson Problem (TSP) and some graph algorithms often have exponential time complexity.

Contrast with Other Time Complexities:

Polynomial Time:

In contrast, polynomial time algorithms (e.g., O(n^2), O(n^3)) have running times that increase at a polynomial rate, which is generally more manageable for larger inputs.

Linear Time:
Linear time algorithms (e.g., O(n)) have a running time that increases proportionally with the input size, which is the most efficient case for most algorithms.

Why Exponential Time is Problematic:

Scalability Issues:

Algorithms with exponential running times become impractical very quickly as the input size increases, making them unsuitable for handling large datasets.

Computational Limits:
The computational resources required (time, memory) grow exponentially, making it infeasible to find solutions within a reasonable timeframe.

Example:

Consider an algorithm with O(2^n) time complexity. If it takes 1 second to process an input of size 10, it might take 2 seconds for size 11, 4 seconds for size 12, and so on. For a larger input, the time required can become extremely long.

In summary, exponential running time signifies a computationally intensive algorithm where even small increases in input size can lead to drastically longer processing times, making it a significant concern for practical applications.

Logarithmic running time, denoted as O(log n), describes an algorithm's efficiency where the execution time grows proportionally to the logarithm of the input size (n). This means that as the input size increases, the algorithm's runtime increases very slowly, making it highly efficient for large datasets.

Key characteristics of logarithmic running time:

Efficiency:

Logarithmic algorithms are considered very efficient, especially when dealing with large inputs, because the number of operations required grows much slower than the input size itself.

Halving or reducing the search space:

Algorithms with logarithmic time complexity often work by repeatedly dividing the problem size or the search space by a constant factor in each step.

Common examples:

Binary Search: In a sorted array, binary search repeatedly halves the search interval until the target element is found or determined to be absent.

Operations on Balanced Binary Search Trees: Searching, insertion, and deletion operations in data structures like AVL trees or Red-Black trees typically exhibit logarithmic time complexity because they maintain a balanced structure that ensures the height of the tree remains logarithmic with respect to the number of nodes.

Understanding the "log" in O(log n):

While the base of the logarithm can vary, in the context of algorithm analysis, it is typically assumed to be base 2 ($\log_2 n$). This reflects the common scenario where an algorithm divides the problem in half at each step. For example, if an algorithm takes $\log_2 n$ steps:

For an input size of n=8, it would take $\log_2 8 = 3$ steps.

For an input size of n=16, it would take $\log_2 16 = 4$ steps.

This slow growth rate is what makes logarithmic algorithms highly desirable for performance-critical applications.