* Big O notation is a way to describe how well a computer program performs as the amount of data increases.

**o(n)** the time needed to complete the algorithm increases proportionally with the size of the input

**O(n^2)**

**O(n log n)**  curve slows down as the input size increases

**O(2^n)** not scalable… curve grows faster as input size increases

O(n^2) is slower compared to o(n)

O(n log n) is more efficient than o(n)

**O(n)**: This represents linear time complexity. It means that the time taken by the algorithm increases linearly with the size of the input. If the input size doubles, the time taken will also double.

**O(n^2)**: This represents quadratic time complexity. It's less efficient than linear time complexity, as the time taken grows at a faster rate with increasing input size. If the input size doubles, the time taken will quadruple.

**O(n log n)**: This time complexity is often associated with algorithms that use the divide-and-conquer approach, such as merge sort or quicksort. It falls between linear (O(n)) and quadratic (O(n^2)) time complexities. Although it's not as efficient as linear time, it's still more efficient than quadratic time.

**O(2^n)**: Algorithms with exponential time complexity are not scalable for larger inputs, as the time taken grows exponentially with input size. Even small increases in input size can result in significant increases in execution time.

**Comparing O(n^2) with O(n)**: O(n^2) is indeed slower compared to O(n), as the time taken by an algorithm with O(n^2) complexity grows at a faster rate than O(n) when input size increases.

**Comparing O(n log n) with O(n)**: O(n log n) is more efficient than O(n^2) but not as efficient as O(n). It is often used in algorithms that perform better than linear time but not quite as well as linear time. It serves as a balance between efficiency and complexity in many practical scenarios.





