Big O Classes

Big O looks at the WORST case performance of an algorithm. Eg looking for an item in an array by iterating each element, best case is element is at index 0 and worst is last element. Big O only considers the latter case.

Big O is not a measure of how fast an algorithm runs in terms of time. It's a notation to depict how an algorithm performs on varying input sizes. A algorithm that runs in constant time may still run slower than a quadratic algorithm, because that constant time algorithm just takes a long time to run due to the heavy calculations it needs to do.

To get running time of algorithm: let's say our algorithm has time complexity of O(N). And each iteration we bench mark to take 1 millisecond. If our array size is 100, then our algorithm will take 100 millisecond to run. Also, if we change computers, the bench mark for one iteration will also change so our algorithm runs differently in terms of time depending on input size and computer hardware power. So time of algorithm varies with respect to these two attributes but time complexity always stays the same.

Constant O(1)
A function that returns the first element of an input array

Logarithmic O(logN)
Only parts of the input array needs to be looked at. These algorithms performs well for large input sizes since it only looks at parts of the input, meaning the other big part of the input data can be ignored. Binary search, finding an element in a sorted array is an example of a logarithmic operation.

Linear O(N)
Multiply each number in input array by 2. Need to loop through this array. Another example is finding an element in an unsorted array since u need to look at every element in the worst case where the element u want is at the end of the list.

Linearithmic O(NlogN)
Quicksort. There are O(logN) iterations needed to half the input array until all subarrays have a length of 1. During each of those iterations, you need to scan the entire subarray in order to rearrange the elements. Hence O(NlogN) complicity.

Quadratic O(N^2)
A double nested loop of a input array. Finding duplicates in an array is quadratic time. The naive approach anyways. Iterate over all elements, at each iteration, iterate over the other elements to compare if they are duplicates. Can be speed up to Linearithmic time by first sorting the array and then iterate over the sorted array comparing each element to the element after it. Sorting is Linearithmic time and the final loop is linear. Big O only looks at the largest complement so overall, this algorithm runs in Linearithmic time. This algorithm has space complexity of O(1) because the memory size is the same no matter the array size since we not storing extra variables if array size increases. Can even speed it up to linear time by iterating over each element and at each iteration store the number into a hash table and comparing if the number at the current iteration is in the hash table. This requires space complexity of O(N) since we end up storing all elements into the hash table. It can be observed that lowering the time complexity usually increases the space complexity. There are exceptions so it's not a hard rule.

Cubic O(N^3)
A triple nested loop of a input array

Polynomial O(N^C)
Quadratic and cubic are examples of polynomial time complexities.

The above classes are considered "fast". The following are considered "slow".

Exponential O(C^N)
Breaking a 10 letter password by brute force is O(10^N)

Factorial O(N!)
Generate all permutations of a list
Travelling salesman problem
4! = 4x3x2x1 = 24

O(N^N)
The slowest known time complexity.