Big O notation

Selecting transcript lines in this section will navigate to timestamp in the video

- Different operations like inserting, accessing, and searching on different data structures take different amounts of computational time. To be able to compare these operations on various data structures independently of input, we use something called Big O Notation. Big O Notation is used to describe the performance or complexity of an algorithm. Let's think about the operations we've used so far on a race. We've inserted items, accessed items, searched for items, and more. For each of these, we can come up with its time complexity or how long it takes to run independently of input. As long as we know the item's index, accessing it within the array takes what we call O of 1 Time or constant time. This means that the algorithm executes in the same time or space regardless of the size of input. When we access an item by index, it does not matter if the array has a million items or just two items. It takes the same amount of computational time. This is also true for updating items in an array. We simply access the item at a given index and change its value using an assignment statement. No matter the size of the array, this always takes the same amount of time. This means for arrays, an update takes constant time. Now insertion is a little different. If your array is big enough and is not completely full, you can simply insert the item at the end of the array, taking O of 1 Time. However, if the array is full of items, you would have to copy everything to a new bigger array and then add your item. We say this would take O of n Time because the computational time taken will grow linearly and indirect proportion to the size of the input. In this case, each individual item would need to be moved to the new bigger array along with the new item we might want to add. This might not take much time for an array with three items when you're adding a fourth, but what if your array was a million items? Then we would be concerned. Although insertion could take O of 1 Time, we often used O Notation to represent the worst case scenario, which is, in this case, linear time. If we assume nothing about our data, searching also takes linear time. If you want to find a specific item and know nothing about the contents of the data or ordering, you would have to iterate through the entire array using a loop. In the best case, the first thing you compare the item to is the actual item, meaning the search takes O of 1 Time. In the worst case, you have to look through the entire array just to find out the item you were looking for wasn't even there. It takes O of n Time because we have to compare the item we are looking for to every item in the array, meaning if the size of the input increases, the computational time also increases in direct proportion to that. For deleting an element, if we have the index of the element we want to delete, then it takes constant time because it is similar to updating the value of that element. However, if we do not know the index of the element we want to delete, we have to search for the element and then delete it. In the worst case, the item we want to delete is not even in the array, so this means deletion time complexity is O of n. Now for sorting, there are lot of different algorithms out there like insertion sort, merge sort, radix sort, and more. Some have prerequisites about your data and others do not. We won't go too much into the different types of sorting algorithms, but the more information we have on the data stored, the more we can optimize how we sort that data. Insertion, searching, and sorting take a little more time, but if you plan on inserting once and accessing a bunch, then an array would be a good data structure to consider.