**Analyzing Performance of Arrays and Objects**

**The Big O of Objects:-**

**When to use Objects:**

* When you don’t’ need order
* When you need fast access / insertion and removal.

When we don’t need any ordering, objects are an excellent choice!

Insertion – **O(1)**Removal – **O(1)**Searching – **O(N)**Access – **O(1)**

In the upcoming section called *Hash Maps* where we actually learn a data structure that explains how objects work behind the scenes, how things are actually stored. And how the above algorithm in Object has constant or say Linear (for Searching) time complexity.

So no order but everything else, almost everything else is very quick. Quick mean to say about constant time for insertion, removal and accessing data.

**Note:**

* JavaScript is able to add something into an object, store a new piece of information in constant time. It’s also able to retrieve something in constant time. And we can also update something in constant time, which is really the same as retrieving or changing it. The same with removal. Accessing information with a key in Object is Constant Time.
* Although this is so fast for all basic operation. Since there Is no order in, So there’s no beginning of object, There’s no end. So it doesn’t matter where you insert because there is no where. That’s why we can’t insert at the beginning or in the middle or the end of the object. There’s no repercussions. We just add in using a key.

**Example:**let instructor = {

firstName: "Chandan",  
 isInstructor: true,   
 favoriteNumbers: [1,2,3,4]

}

**Some Object Methods:**

* Object.keys - **O(N)**  
  console.log(Object.keys(instructor));  
  **Output:**['firstName', 'isInstructor', 'favoriteNumbers']  
  This is over time because as the number of items in there grows, we’re going to have to visit every single thing once and add it to this array.  
  If we have 100 elements or 100 properties in our object, there’s 100 things, 100 operations we need to do. So it runs roughly in line with N it might be 2N, it might be 50N but still simplifies to O(N).
* Object.values - **O(N)**  
  console.log(Object.values(instructor));  
  **Output:**['Chandan', true, Array(4)]  
  It is same for values as happened with keys.
* Object.entries - **O(N)**  
  console.log(Object.entries(instructor));  
  **Output:**[Array(2), Array(2), Array(2)]  
  It is same for entries as happened with keys.
* hasOwnProperty - **O(1)**  
  console.log(instructor.hasOwnProperty("firstName"));  
  console.log(instructor.hasOwnProperty("lastName"));  
  **Output:**true  
  false  
  We pass in a property like first name and it just tells us it has a first name and return *true* or doesn’t have first name return as *false.* So that’s why this is constant time O(1).

If we’re able to access information in constant time, if we’ve the key firstName, we want the value. We should also be able to check if a key exists in pretty much the exact same time. So in summation, objects are really quick fro pretty much everything. However, there’s no order.

And as we’ll see coming up with arrays, arrays can be pretty fast for a lot of things, but the order also can slow them down depending on what we’re doing.

So objects are basic, they work very well and a lot of situations, key value pairs, all the main operations, inserting, accessing, updating, removing all Constant time. Searching is pretty rare.

**The Big O of Arrays:-** (Ordered Lists)

Ex:  
let name = [“Michael”, “Melissa”, “Andrea”];  
let values = [true, {}, [], 2, “awesome”];

So the big selling point of a race, of course, is that they are ordered there is an intrinsic ordering to the data, unlike an object where things just float around in a gelatinous mass.

**When to use Arrays:**

* When you need order
* When you need fast access / insertion and removal (sort of…)

If you’re just trying to store random data together, you could still use an array. But if you’re really trying to optimize for performance, there are other options.  
And even if you do need order, we’re going to see some other structures coming up, like a singly linked list and a doubly linked list that still in code order.

There is a linear it’s a list structure whether each element is in a particular spot and they’re all connected in an order, but they sometimes can perform better than arrays depending on what you need.

So keep in mind that, arrays are not the only order data structure on Earth. They’re just the only one that we get for free in JavaScript.

* Insertion - **It depends….**It really depends on where we’re inserting.   
  Let say if we want to insert a new value at the end of array using push() method, then this is going to take **O(1)**.  
  On the other hand if we want to insert new value at the beginning of the array using unshift() method, it will change the index number of all the previous values inside the array (means it will going to re-index again). So it will going to take **O(n)** runtime. Because the amount of time it takes roughly grows in proportion with the size of the array.  
  So, inserting at the beginning is problematic. The same goes for removing from the beginning.  
  That’s why push and pop always faster than unshift and shift.
* Removal - **It depends….**
* Searching - **O(N)**Fastest Searching we can do in an array is O(N). More discussed in Searching section.  
  Let say, If we’re talking about an unsorted array where there’s no order to the data. If I wanted to know if I had another 10000 names in there and I wanted to know if Robbie was in there, we have to check potentially every single element. So as a number of items grows in that array, so does the time potentially that it take to find that items. So search grows, **O(N)** it’s a linear**.**
* Access - **O(N)**It is exact same as we talked about with an object. Accessing is fast no matter where. If we’re talking about 10000 elements, accessing the middle element is just as fast as accessing the second element.

**Big O of Array Methods:**

* push - **O(1)**
* pop - **O(1)**
* shift - **O(N)**
* unshift - **O(N)**
* concat - **O(N)  
  Ex:**let name1 = ["Michael", "Melissa", "Andrea"];

let name2 = ["Chandan", "Kumar", "singh"];  
name1.concat(name2);  
**Ouput:**  
['Michael', 'Melissa', 'Andrea', 'Chandan', 'Kumar', 'singh']  
It’s basically just expressing the fact that as multiple given arrays grow, that we’re merging, so is the time that it’s going to take and it grows in proportion with the size, the total size of the new array.

* slice - **O(N)  
  Example:**let name2 = ["Chandan", "Kumar", "singh"];  
  name2.slice(0, 3);  
  **Ouput:**  
  ['Chandan', 'Kumar']Let say if we’re tyring to copy 10 elements versus 1000 elements from an array, the amount of time takes grows in proportion with that size. How large of a copy are, how many element we’re trying to copy?. So that’s O(N);
* splice - **O(N)  
  Example:**let name2 = ["Chandan", "Kumar", "singh"];  
  name2.splice(2,1,"King"); ***//replace 1 element at 2nd index***name2.splice(2,0,"Singh"); ***//just insert at 2nd index position***   
  **Ouput: *//only return array with the replaced element***  
  ['singh'] ***//singh replace from 2nd index****['Chandan', 'Kumar', 'King']* ***//new\_array***['singh'] ***//nothing replace from 2nd index****['Chandan', 'Kumar', ’Singh’, 'King']* ***//new\_array***  
  So, using splice we can insert at the beginning, middle or at the end of the array. But it general it is O(n) Because, Even if we’re inserting in the middle of the array, we just simplify that to O(n). It does meant shifting and indexing everything that comes after it.
* sort - **O(N \* log N)or O(nlogn)**As we know, The time it takes to sort an array is larger than just O(n).
* forEach/map/filter/reduce/etc. - **O(N)**Here, all of and those are looping over each element or doing something with each element, telling it, doing a Boolean test. It just involves acting on each element or with each element. So as the size of the arrays grows, so does the amount of time it takes.