Chapter 2

**Analyzing Performance**

**Of**

**Arrays & Objects**

* 1. **Built-In DATA STRUCTURES: Through the lens of Big-O**

Before we get to the advanced algorithms: *sorting/searching/trees* and other *data structures*, we first try to apply, *Big-O notation*, Time and space complexity, analyzing code performance to the stuff we already know. Such as: Javascript *arrays*, *objects*, built in *methods*, *loops*.

* What are the easy things, the fast things we can do to an array. What's the *method* that might be *slower* than you would expect. We're going to talk about how it performs.
* Our objectives :
* Understand how ***objects*** and ***arrays*** work through the lens of ***big-O***. Is there a *fast/slow* way to insert to an *array*.
* Explain why ***adding*** elements to the beginning of an ***array*** is costly, we'll talk about why that is and how it works and if there are alternatives.
* Then we'll compare and *contrast* the *runtime* for ***arrays*** and ***objects***, as well as *built in methods*, like the built in javascript: ***sort()*** for an array, ***forEach()*** method, ***Object.keys()***, ***Object.prototype.hasOwnProperty()***.

**2.2 Big-O of Objects**

Here is an object. An object is unordered data-structure, basically they are collection of key-value pairs.

let instructor = {

    firstName: "Kelly",

    islnstructor: true,

    favoriteNumbers: [1,2,3,4]

}

​

* When to use objects:
* When you *don't need order*
* When you need *fast access / insertion* and *removal*
* Big-O of Objects:
* Insertion = **O(1)**
* Removal = **O(1)**
* Searching = **O(N)**
* Access = **O(1)**

When you don't need any ordering, objects are an excellent choice!

* Objects work well when you don't need order, when you *want fast access* and insertion and removal. Almost everything else is very quick.
* Here quick means constant time for *insertion/removal* and *accessing* data. That's the fastest we can go.
* Later in the course have a section called Hash Maps where we actually learn a data structure that explains how *objects* work *behind the scenes* how things are actually stored.
* So what happens when we say ***instructor.firstName*** set to ***Kelly***, what does javascript actually do. A computer can't necessarily just access a place in memory called ***firstName***, there are some additional steps involved along the way. There's something called Hashing we talk about it *later* in this book.
* But for now all that you need to know is that *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
* you can also *update* something in *constant* *time* which is really the *same* as retrieving it (you're just changing it).
* Same with removal.
* No order: It's quick it's fast. For all the basic operations there is *no order* so there's *no beginning* of the object there's *no end*. So it *doesn't matter* where you *insert* because there is *no where*.
* You *can't insert* at the *beginning* or in the *middle* or the *end* of the *object* there's no repercussions. You just add in using a key.
* Later we'll talk about Hash Tables and Hash Maps where you'll understand understand how it is Constant time.
* Searching: Searching is **O(N)**, it's linear time.
* When we say searching it *doesn't mean* *looking for a key*. For example ***firstName*** because it is accessing, and *accessing information* with a key is *constant time*.
* Here searching means checking to see if a *given* piece of *information* is in a *value* *somewhere*.
* For example: if we wanted to know is ***true*** is stored in our object somewhere *we'd have to check*. Then we need to check *each key-value* one by one, first ***firstName*** then ***islnstructor*** and it eventually find ***true*** as a value.
* Hence if the number of *properties (key-value pair) increases*, the *checking (searching) time* also *increases*.
* Hence it is ***O(N)***, because the ***number of properties grows*** then the amount of ***searching time also grows***.
* Big-O of some Object Methods:

|  |  |
| --- | --- |
| *Object.keys* - **O(N)** | It is O(N),     * 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*. |
| *Object.values* - **O(N)** | * Same thing with ***values***, if we have a hundred elements or a hundred properties in our object there's 100 operations we need to do. |
| *Object.entries* - **O(N)** | * Also is the same for ***entries***. Technically entries is a little more work to compile the key and the value. But it all just simplifies to over and at the *end of the day* remember. |
| *hasOwnProperty*- **O(1)** | * ***hasOwnProperty*** is a little different.      * It just tells us it has/hasn't ***firstName*** and this is constant time. * If we're able to access *information* in *constant time*, then we should also be able to check if a ***key exists*** in ***pretty much the exact same time***. |

* Summery:
* Objects are really *quick* for pretty much *everything*.
* There's no order for Objects, and it makes them faster than arrays. But **arrays** are "**ordered**", 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 in a lot of situations.
* *Key-value pairs* operations like: ***inserting***, ***accessing***, ***updating***, ***removing*** all constant time.
* *Searching* is pretty rare but it is **O(N)**. It's linear and then the methods like ***keys***, ***values*** and ***entries*** where we're creating an array based off of the data inside the object are also **O(N)**.
* As the object *grows* the number of operations the *amount of time* it takes to *compile* those also grows.

**2.2 Big-O of Arrays**

*Arrays* are *ordered*, there is an *intrinsic ordering* to the data (unlike an object). In general, you use arrays when you need order.

|  |
| --- |
| * If you *don't need order* then you probably don't want to use an *array* if you're just trying to store *random data* together. * If you're really trying to optimize for performance there are other options. * And even if you do need order there are Singly Linked List and a Doubly Linked List that still encode order. It's a list structure where 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 arrays are not the only ordered data structure on Earth. They're just the only one that are built-in javascript data-structure. |

* *Arrays are often* very *useful* if we need *order* but it can come at a *cost* for some of the *operations*.

let names = ["Michael", "Melissa", "Andrea"];

let values = [true, {}, [], 2, "awesome"];

|  |  |
| --- | --- |
|  |  |

* Especially around *insertion* and *removal* it can *complicate things*.

|  |  |
| --- | --- |
| * Accessing: So accessing data in an array is very quick. Because it takes a ***single step*** to ***access*** an ***item*** of an ***array*** via its ***index***, or *add/remove* an *item* at the *end* of an array. | * The *complexity* for * *accessing*, * *pushing* or * *popping* a value in an array is ***O(1)***. |

|  |
| --- |
| * **Why does accessing an array element take constant time?** * Note that ***array***, ***linked list***, ***map***, ***set*** etc, everything that your program creates would be in ***RAM*** during your ***program execution***. That means RAM part is ruled out. * Array - The fundamental concept here is that the elements of an array are stored in Contigous Memory Locations, if ***a[0]*** is at a memory location ***2000***, then ***a[1]*** is guaranteed to be at location ***2002*** [ assuming ***2 bytes*** of storage ]. It is this gurantee that makes array access constant time. * In general, if I know the base address of array, i.e., ***a[0]***, I would get the ***address*** of an ***arbitrary element*** in ***array*** e.g., ***a[i]*** by simply doing address of **a[0] + i \* size;**   So, array look up / access is ***O(1)***.  This sort of math can be applied to any dimensional array, because the elements would be in a Continuous Memory Location.   * **Linked List:** Note however, that if you take **Linked List**, there is no guarantee that elements would be stored in this form. And that is why, Linked List access is ***not constant time***. It is dependent on the ***size*** of the ***linked list***. * So, ***linked list look up / access*** is **O(N)**. |

* So access time is **O(1)**, same for array and object.
* When you have a ***10000*** element in array and you ask for the ***9000th*** element, Javascript isn't going through every single element, it directly access that ***9000th*** element.
* You can *jump* *immediately* to the data. So it doesn't matter how *long* the *array* is. It doesn't matter if you're looking at the *last* item the *middle* item or the *first* item. It's *constant time*.
* Insertion and removal: Really depends on where we're inserting.
* ***Adding*** at the ***end*** is constant time ***O(1)***.



* Adding beginning is problematic: ***Adding*** an element at the ***start*** of the array is ***O(N)***. The problem comes when we try to insert at the beginning of an array. And the reason is the indices.



* In that case, our *indices* will totally *messed* *up*, we have to *re-index* every *single element* in the array. This is simple for a small array. But if we're talking about thousands and thousands of elements re-indexing every single one is not a trivial task.
* So if we were *inserting* at the *beginning* of an array we're talking about ***O(N)*** time because we have to do something once roughly once for every single element.
* Now that doesn't mean that it's exactly one operation for each element remember.
* And it could be of ***O(10N)***, ***O(2N)***and ***O(0.5N)*** it's just that the amount of time it takes roughly grows proportionally with the size of the array.
* The same goes for removing from the beginning. We have to *re-index* again just the *other direction* .
* *Adding* and *removing* from the *beginning* of an *array* is best to *avoid* if you can.
* But often it's *meaningful* to add to the *beginning* of an *array*.
* Methods: ***push*** and ***pop*** always faster than ***shift*** in ***unshift***. Unless of course it's an empty array in which case adding to the beginning or end is the same thing.
* Searching: If we're talking about an unsorted array where there's no order to the data.
* If we wanted to *search* an *element*, we have to check *potentially* every *single element*.
* So as the number of items *grows* in that array then the *time* also *grows* to find an item. Hence searching is **O(N)**.
* Big O of Array Operations:
* push - **O(1)**
* pop- **O(1)**
* shift- **O(N)**
* unshift- **O(N)**
* concat- **O(N)**
* slice- **O(N)**
* splice - **O(N)**
* sort - **O(N** \***log N)**
* forEach/map/filter/reduce/etc. **O(N)**
* ***push*** and ***pop*** are ***constant time*** adding and removing data to the end of the array of any array whether it's one item or 10000 or a million items is Constant time. It's easy there's no reindexing involved. We just ***put*** it at the ***end*** and give it a spot.
* ***unshift*** and ***shift***, on the other hand are a little more ***troublesome*** because we potentially have to *re-index* every element in the array.
* If you're adding to the beginning or removing from the beginning.
* ***concat, slice & splice:*** So all of these are ***O(N)*** linear time.
* Something that ***concat*** takes *multiple arrays* and *merges* them *together*. It's basically just *expressing* the fact that as these are *raised* *grow* that you'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* at the *end*.
* So *slice* is going to return a *copy* of *part of an array* or the entire thing if you want it to. If we try 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 or how many elements we're trying to copy. So that's O(N).
* ***splice*** is going to *remove* and *add* new *elements*. ***splice*** is a little different, we can insert at the beginning of an array, also at the very end of an array, we can replace given elements.
* Say we're inserting in the middle of the array, that means ***shifting*** and ***re-indexing*** everything that comes after it. So it is ***O(N)***.
* ***sort()*** *for Sorting:* **O(N** \***log N)**, it takes to sort an array is larger than ***O(N)***. We have to make comparisons and we have to move things around.
* It's not enough just to look at every element once, so it is non-linear and larger than ***O(N)***. It's a little more sometimes a lot more complicated.
* Also, forEach/map/filter/reduce/etc. are of ***O(N)***.
* Those are looping over elements or doing something with each element (eg. taking a Boolean test) it just involves acting on each element.
* So as the size of the array grows so does the amount of time it takes of it. Hence ***O(N)***.
* We can see almost everything that we could do with an array, all the built in methods (not all of them) but most ones listed here are of ***O(N)***.
* So in short:
* ***Objects*** are ***fast*** at pretty much everything but there's ***no order***.
* ***Arrays*** are ***great*** when you need an ***order*** but still be mindful that it's better if you can if you can do it to add and remove from the
* end and avoid adding and removing from the beginning because that starts this cascade effect.

So hopefully that makes sense how we get to these numbers ***O(N)***, ***O(1)*** or ***O(N\* log N)***, what the values mean.