**Contents**

[**Recommended courses** 1](#_Toc165313245)

[**Recommended books** 1](#_Toc165313246)

[**Big O** 1](#_Toc165313247)

[**Big O categories** 2](#_Toc165313248)

[**Array** 3](#_Toc165313249)

[**Definition** 3](#_Toc165313250)

[**Getting at specific index** 3](#_Toc165313251)

[**Insertion at specific index** 3](#_Toc165313252)

[**Deletion at specific index** 4](#_Toc165313253)

# **Recommended courses**

* **The Last Algorithms Course You’ll Need** - *The Primeagen*

# **Recommended books**

[**A common-sense guide to data structures and algorithms : level up your core programming skills**](https://sait.primo.exlibrisgroup.com/discovery/fulldisplay?docid=alma991003030600606261&context=L&vid=01SAIT_INST:DEFAULT&lang=en&search_scope=MyInst_and_CI&adaptor=Local%20Search%20Engine&tab=Everything&query=any%2Ccontains%2CA%20Common-Sense%20Guide%20to%20Data%20Structures%20and%20Algorithms%2C%20Second%20Edition%3A%20Level%20Up%20Your%20Core%20Programming%20Skills&offset=0) **-** Wengrow, Jay, author. 2020; 2nd edition

[**Introduction to algorithms**](https://sait.primo.exlibrisgroup.com/discovery/fulldisplay?docid=alma991003077836706261&context=L&vid=01SAIT_INST:DEFAULT&lang=en&search_scope=MyInst_and_CI&adaptor=Local%20Search%20Engine&tab=Everything&query=any%2Ccontains%2C%22the%20introduction%20to%20algorithms%22&offset=0) **-** Cormen, Thomas H., author.; Leiserson, Charles Eric, author.; Rivest, Ronald L., author.; Stein, Clifford, author. 2022; Fourth edition.

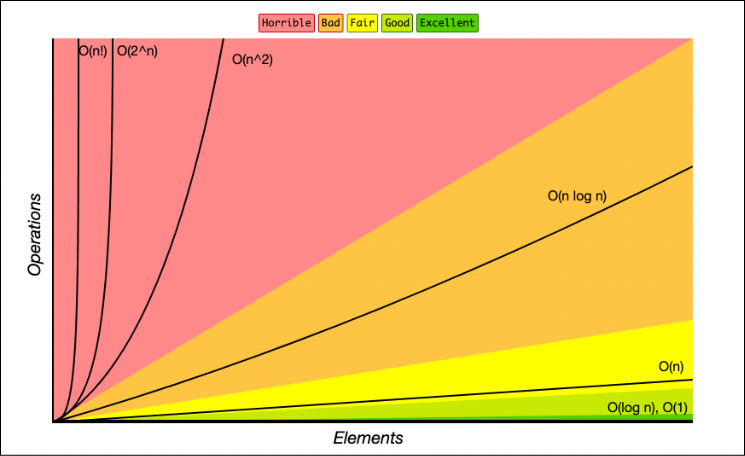
# **Big O**

General knowledge about how much computation and memory is required as we increase the input.

* growth with respect to input (computation and memory);
* always measure by the worst case scenario;
* constants are dropped.

observation: it is not a real trade off between computation time and memory, as it takes time to create memory.

## **Big O categories**



* **O(1)**: it always does the same amount of operations (effectively instant), independently of the input size.
* **log(n)**: the same as O(1), but you only need to look at one point at a time.
* **O(n)**: linear growth. → **loops**.

For everyone more input unit, there is one more loop that it has to do.

If we have 100 unrelated loops, we will not consider the algorithm as O(100N), because it is still linear and not relevant when compared to something else that grows exponentially. So, we always drop the constants, and just categorize as O(n).

* **O(n^2)**: for every single character, we go over every single character (like calculating the area of a square, like multiplying matrixes). The same goes for O(n^3). → **nested loops**.
* **O(n.log(n))**: for every time you go over the input, you need to search over half of the amount of the input/space.
* **O(n!) and O(2^n)**: are algorithms that cannot effectively run in computers.

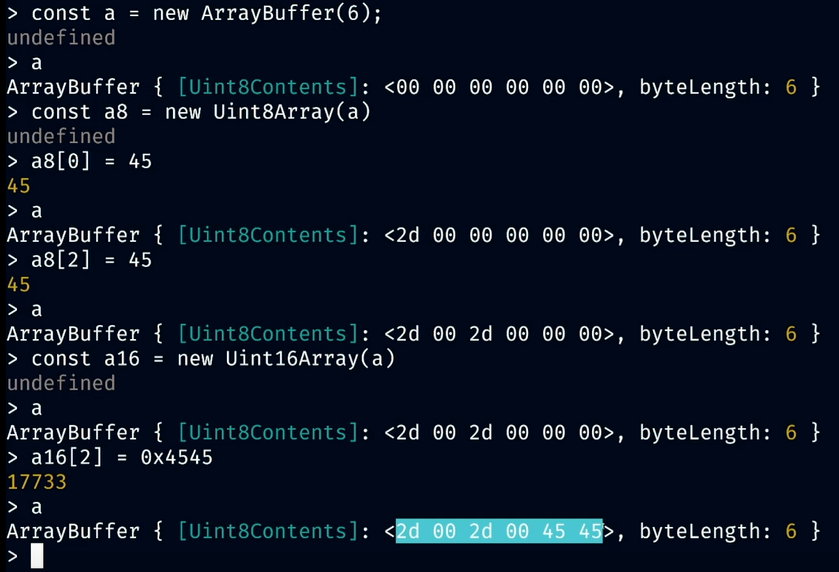
# **Array**

## **Definition**

An **array** is a **continuous (unbreaking) memory space** in which contains a certain amounts of bytes (0s-1s), so the memory understands it as a single type in a row. It allows us to walk through the data.

### **Getting at specific index**

At first we looked at it as 8 bits units, then as 16 bits units, and we can walk at 8 and 16 bits units at a time, respectively. We are simply using the length of the array to walk through it.



### **Insertion at specific index**

It does not insert something, it overwrites it. You don’t reallocate memory.

You go to a memory address, you add the width of the type (bytes) multiplied by the offset.

What would happen if we had 4 bytes (32 bits) allocated for a name attribute, and right after 4 bytes (32 bits) for the email address, but then we try to expand the name memory allocation? It would mess up with what comes after, the email address.

### **Deletion at specific index**

You don’t actually delete something out of continuous memory. You can overwrite to 0, for instance, or to null. It is basically a way for us and the program to interpret that there is not something there.

→ Arrays have a fixed size in continuous memory chunks.

→ You cannot grow a static/traditional array, and there is no insert, push, or pop (although it can be implemented).

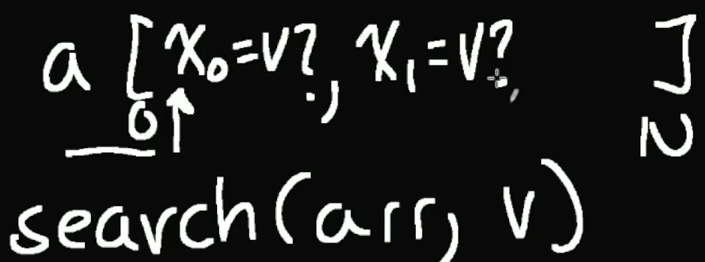
You can't grow it, you can reallocate it, meaning you're going to have to create a new array. Take your old array, write it into the new one, you now have a bigger array. You now have more room but nonetheless, it is these fixed sizes.

***How big is the array then that you instantiate?***

*So, that is part of optimizing. So, like in Rust, if I'm not mistaken, if you create a new vector, it's gonna create a effectively a memory buffer with the size 5 underneath. And so, as you push and pop, it has 5 units for you to be able to play and now is that efficient? Well, that's the problem. How big do you need to create your underlying buffer to be able to effectively use that space without having to re allocate too much or without not using it? Right, cuz we could all create 10,000 units and be like, hey, look, we'll never have to reallocate again. But if 10,000 units are big units, you're gonna use a lot of memory, just to have that in there.*

# **Search**

### **Linear search**



We go to the item at position 0, ask it if that value is the one that you are looking for. If it is not, it asks the one in the next position.

*What is the Big O of a linear search?*

*The worst scenario is if the value is not in the array, so it would have to check all values. As the input grows, so the time it takes equivalently, or linearly. 🡪 O(N).*