Why Data Structures Matter ?

There are multiple measures of code quality, one measure is code maintainability, another aspect is code efficiency.

`Version 1`

```
def print_numbers_version_one():
  number = 2
  while number <= 100:
    if number % 2 == 0:
      print(number)
      number += 1
```

`Version 2`

```
def print_numbers_version_two():
  number = 2
  while number <= 100:
    print(number)
    number += 2
```

Here `Version 2` runs faster than `Version 1`. This is because Version 1 ends up looping 100 times, while Version 2 only loops 50 times. The first version then, takes twice as many steps as the second version.

`Data Structures` refers to how data is organized. Data organization significantly impact ***how fast your code runs***.

The Array: The Foundational Data Structure

Array is a list of data elements.

```
array = [1, 2, 3, 4]
```

The `length` of the array is how many data elements the array holds. Here it is 4. The `index` of an array is the number that identifies where a piece of data lives inside the array.

Data Structure Operations

- `Read`: Reading refers to looking something up at a particular spot within the data structure.
- `Search`: Searching refers to looking for a particular value within a data structure.
- `Insert`: Insertion refers to adding a new value to our data structure.
- `Delete`: Deletion refers to removing a value from our data structure.

Measuring Speed

We can measure the speed of an operation in terms of how many `computational steps` it takes.

If A takes 5 steps and B takes 500 steps, then A will always be faster on all hardware. Measuring the speed of an operation is also known as measuring its time complexity.

Reading

Reading operation looks up what value is contained at a particular index inside the array.

A computer can read from an array in just one step. This is because the computer has the ability to jump to any particular index in the array.

```
array = [1, 2, 3, 4]
```

Element at `index 0` will be 1, at `index 1` will be 2 and so on.

When a program declares an array, it allocates a contiguous set of empty cells for use in the program. Each cell's memory address is one number greater than the previous cell's address.

```
                [1,   2,   3,   4]
Memory Address:  100, 101, 102, 103
Index         :  0    1    2    3
```

- A computer can jump to any memory address in one step.
- Whenever a computer allocates an array, it also makes note at which memory address the array begins.

If we asked the computer to find the value at index 3, the computer would simply take the memory address at index 0 and add 3.

Let's calculate for index 2.

- Array starts at index 0, with address 100.
- Adding 2 to the address we get $100 + 2 = 102$
- Then computer jumps to address 102 and fetches the value.

Searching

searching an array means looking to see whether a particular value exists within an array and if so, at which index it's located.

A computer has immediate access to all of its memory addresses, but it has no idea offhand what values are contained at each memory address.

To search for a element within the array, the computer has no choice but to inspect each cell one at a time.

```
array = [1, 2, 3, 4]
```

Here we are searching for `4`.

- Computer starts at index 0, it checks whether the element at index 0 is equal to `4`.
- If yes it stops and we know the index or else it moves to next index and performs the same.
- This is repeated until `4` is found.

so in total it took four steps, this basic search operation—in which the computer checks each cell one at a time—is known as `linear search`.

For an array of five cells, the maximum number of steps linear search would take is five. For an array of 500 cells, the maximum number of steps linear search would take is 500.

Another way of saying this is that for N cells in an array, linear search would take a maximum of N steps. In this context, N is just a variable that can be replaced by any number.

Reading is more efficient than Searching.

Insertion

The efficiency of the insertion depends on the position where this operations should take place.

For example, to insert at end of the array, it takes just one step.

```
array = [1, 2, 3, 4, 5]
```

Because the computer initially allocated only five cells in memory for the array, and now we're adding a sixth element, the computer may have to allocate additional cells toward this array.

To insert a new piece of data at the beginning or in the middle of an array, we need to shift pieces of data to make room for what we're inserting, leading to additional steps.

Here if we want to insert `8` at index `2`.

- 5 is moved to right.
- 4 is moved to right.
- 3 is moved to right.
- Now there is an empty space at index `2`.
- Now `8` is inserted at index `2`.

The worst-case scenario for insertion into an array, is when we insert data at the beginning of the array. This is because when inserting at the beginning of the array, we have to move all the other values one cell to the right.

We can say that insertion in a worst-case scenario can take `N + 1` steps for an array containing `N` elements. This is because we need to shift all N elements over, and then finally execute the actual insertion step.

Deletion

Deletion from an array is the process of eliminating the value at a particular index.

```
array = [1, 2, 3, 4, 5]
```

Here if we need to delete `2`, it just takes one step, but there will be gap in the array.

- We will shift 3 to the left.
- 4 to the left.
- 5 to the left.

For an array containing N elements, the maximum number of steps that deletion would take is N steps.

Sets: How a Single Rule Can Affect Efficiency

A set is a data structure that does not allow duplicate values to be contained within it.

```
set_list = {1, 2}
```

Here if we try to insert 2, computer wouldn't allow it as 2 is already present in the array.

`Reading` from a set is exactly the same as reading from an array.

`Searching` and `Deletions` are identical to array, both takes up to N steps.

In `Insertion`, the computer first needs to determine that this value doesn't already exist in this set, the computer will first need to search the set to see whether the value we want to insert is already there. Only if the set does not yet contain our new value will the computer allow the insertion to take place.

Every insertion into a set first requires a search. insertion into the end of a set will take up to `N + 1` steps for N elements.

In the worst-case scenario, where we're inserting a value at the beginning of a set, the computer needs to search N cells to ensure that the set doesn't already contain that value, another N steps to shift all the data to the right, and another final step to insert the new value. That's a total of `2N + 1` steps.