Algorithms

Regarding the code, we don't learn anything new in this project. Except to master the management of chained lists or arrays and the structures. However, the most important concept that we discover in push swap is the algorithms.

What's an algorithm ?

An algorithm is a set of steps for solving a specific problem or completing a task.

Think of it like a recipe that outlines how to make a cake from scratch. Each step is clearly defined and must be followed in a specific order to achieve the desired outcome. Another example would be solving a Rubik's Cube. When you see certain shapes, you have to execute certain instructions.

Algorithms can be applied to a wide range of problems, from simple arithmetic operations to complex machine learning models.

Learning algorithms, especially in the context of push_swap, is useful for several reasons:

- **Problem-solving skills**: Algorithms help develop problem-solving skills as it requires one to understand the problem, come up with a solution and then implement it.
- **Understanding of data structures**: The "push_swap" problem involves sorting data, so understanding algorithms helps in understanding different data structures like stacks and arrays and how to manipulate them.
- **Improved efficiency**: Understanding algorithms helps improve the efficiency of code by reducing the number of operations required to solve a problem.
- **Preparation for technical interviews**: Knowledge of algorithms is often tested in technical interviews as it provides insight into a candidate's problem-solving skills and understanding of basic computer science concepts.

A problem can always have several solutions. Some solutions are more efficient than others. This means that one algorithm can be more efficient and optimized than another. This is called algorithmic complexity.

Algorithm complexity

Algorithm complexity refers to the amount of resources (such as time or memory) required for an algorithm to solve a problem. It provides a way to measure and compare the efficiency of different algorithms.

There are two main types of algorithm complexity:

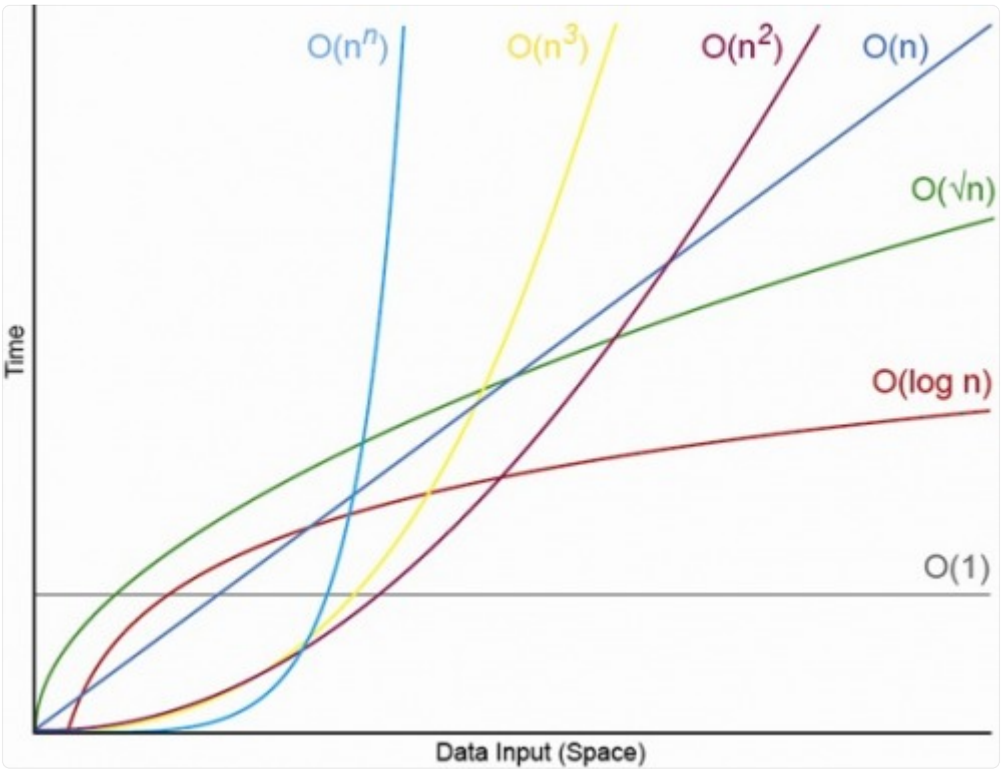
1. **Time complexity**: measures the amount of time an algorithm takes to solve a problem.
2. **Space complexity**: measures the amount of memory an algorithm takes to solve a problem.

Measurement of algorithmic complexity

The complexity is usually expressed using Big O Notation, which provides an upper bound on the resources required by the algorithm. The most classical approach is therefore to calculate the worst-case computation time. Let's take an example:

Let's say you have an algorithm that takes 100 milliseconds to solve a problem with an input size of 10. If you double the size of the input, it might take maximum 200 milliseconds. If you have an input size of 30, it might take max. 300 milliseconds. Etc. This means the time complexity of the algorithm is O(n), where n is the size of the input.

Here is a diagram that compares several levels of complexity/big O notations with each other:



Algorithm comparisons

1. O(1) represents a constant time complexity, which means that the time required by the algorithm does not depend on the size of the input. For example, accessing an element in an array using an index is O(1).
2. O(n) represents a linear time complexity, which means that the time required by the algorithm grows linearly with the size of the input. For example, a simple linear search algorithm is O(n).
3. O(log n) represents a logarithmic time complexity, which means that the time required by the algorithm grows logarithmically with the size of the input. For example, a binary search algorithm is O(log n).
4. O(n^2) represents a quadratic time complexity, which means that the time required by the algorithm grows exponentially with the size of the input. For example, a simple bubble sort algorithm is O(n^2).

It's important to note that Big O Notation provides an upper bound on the growth rate of an algorithm and is not an exact measure of the resources required by the algorithm. The actual resources required by an algorithm can be different from the estimate provided by Big O Notation.

Known sorting algorithms

To achieve the push swap project, you will need to choose a sorting algorithm that will work with the intructions you are allowed to use and that will make the fewest possible moves.

Below is an example of sorting algorithms and their complexity, according to three different methods for the time complexity and with one for the space complexity.

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Complexity of sorting algorithms. Source: Big-O Cheat Sheet, 2016.

Now it's up to you to choose which algorithm you will use! You can of course try to create your own if you can :-)

Good luck with push_swap ;)

Previous
push_swap

Next
Building the thing

Last updated 14 days ago