**Big O Notation**

- used for defining time complexity and space complexity

- used to determinate which algorithm is the best

**Time is not reliable.**

- different machines will record different times

- So, you can’t say which one is best just with execution time

- speed measurements may not be precise enough

- its ok to verify which one is best just by the time, but we need a reliable formula to say that, that’s the Big O!

**Big O**

- count the number of simple operations the computer must perform

- count the numbers of operations that the algorithm has, and if they are **1** or **n** times executed

- when has a n operation, the number of operations grows proportionally with n

- Big O Notation is a way to formalize operation counting

- It allows us to talk formally about how the runtime of an algorithm grows as the input grow

- way to describe the relation between the input size and the function runtime grows

- We say that an algorithm is O(f(n)) if the number of simples operations the computer has to do is eventually less than a constant time f(n), as n increases

- f(n) could be linear (f(n) = n) | O(n)

-- if the number of operations, grows as the n grows

- f(n) could be quadratic (f(n) = n²) | O(n²)

-- if the number of operations, quadratic grows as n grows

- f(n) could be constant (f(n) = 1) | O(1)

-- if the number of operations is constant as n grows

- If you have 2 separates loop inside a function, will be O(2n), then in this case the function will be O(n)

- But if you have a nested loop in a function, (a loop inside a loop), will be O(n \* n), then in this case the function will be O(n²)

- But if the nested loop has a fixed loop value, for example j < 5, the notation will be O(n) because the growing executions would depend only on the external loop

**Rules to simplifying Big O expressions.**

- Rules of thumb

- these rules are consequences of the definition of Big O Notation

**- Constants and smaller terms don’t matter**

**- O(2n + 50) -> Wrong!**

**- O(n) -> Right!**

**- O(500) -> Wrong!**

**- O(1) -> Right!**

**- O(2n² + 5n + 12) -> Wrong!**

**- O(n²) -> Right!**

**- Arithmetic operations are constants**

- operations like + or – or \* or / will always be constant

- variable assignment is constant

- accessing elements in an array or object by his index or key is constant

- in a loop, the complexity is the length of the loop times the complexity of whatever happens inside of the loop

**Space complexity**

- how much additional memory do we need to allocate in order to run the code in our algorithm

- sometimes you’ll hear the term auxiliary space complexity to refer to space required by the algorithm, not including space taken up by the inputs

**- Rules of thumb**

- Most primitives (Booleans numbers, undefined, null) are constant space, O(1)

- String require O(n) space (where n is the string length)

- Reference types are generally O(n), where n is the length (for arrays) or the number of key (for object)

**Logs**

- log2(8) = 3 **->** 2³ = 8

- log2(value) = exponent **->** 2^exponent = value

- O(log2 n) = O(log n)

- O(log n) is much better than O(n)

Diagrama

Descrição gerada automaticamente

**Recap**

- To analyze the performance of an algorithm, we use Big O Notation

- Big O Notation can give us a high level understanding of the time or space complexity of an algorithm

- Big O Notation doesn’t care about precision, only about general trends (Linear, quadratic, constant)

- The time or space complexity depends only on the algorithm, not the hardware used to run the algorithm

- Big O Notation is everywhere

**Problem Solving Approach**

**Objectives**

- Define what an algorithm is

- Devise a plan to solve algorithms

- Compare and contrast problem solving patterns

**What is an algorithm?**

- A process or set of steps to accomplish a certain task

**How do you improve in problem solving**

- Devise a plan for solving problems

- Master common problem solving patterns

**Problem solving strategies(steps).**

**-** understand the problem

- explore concrete examples

- break it down

- solve/simplify

- look back and refactor

**Step 1: Understand the Problem**

- Restate the problem with your own words

- What are the inputs that go into the problem

- What are the outputs that should come from the solution

- Do you have enough information just with the inputs

- how should I label the important pieces of data that are a part of the problem

**Step 2: Concrete Examples**

- User Stories

- Unit Tests

- Start with simple examples (with inputs and outputs)

- Progress to more complex examples

- explore examples with empty inputs

- explore examples with invalid inputs

- explore many inputs to understand the problem and his business rules

**Step 3: Break it down.**

- explicitly write each step of the problem, like a guide

- write each step of the problem like comments

- break it down in steps

- see the process

**Step 4: Solve/Simplify**

- Solve the problem, and if you can’t, solve a simpler problem

- write code and solve what you know how to do

**- Simplify**

- Find the core difficulty in what you’re trying to do

- Temporarily ignore that difficulty

- write a simplified solution

- then incorporate that difficulty back in

- don’t stay stuck in 1 part only, if you stay, just skip that, and come back latter

- let the things that you don’t know to the end, do all what you know to do first

**Step 5: Look Back and Refactor**

**- Refactoring questions**

- Can you check the results?

- Can you derive the results?

- Can you understand it immediately?

- Can you use the result or method for some other problem?

- Can you improve the performance of your solution?  
- Can you think of other ways to refactor?

- How have other peoples solved this problem?

**Recap**

- Understand the problem (asks question and examples for the interviewer to clarify)

- Let the interviewer know where you’re going, next steps before you code

- Solve / Simplify if you can’t solve, don’t get stuck in 1 problem for too long

- Look back and refactor the code

**Problem Solving Patterns**

- Have different approaches and patterns to write code

**Patterns**

**- Frequency counters**

- This pattern uses objects or sets to collect values/frequencies of values

- This can often avoid the need for nested loops or O(n²) operations with arrays/strings

- its indexing the values of the iterable value(array or string) just looping him 1 time, instead of do a nested loop. Changing the Big O Notation from O(n²) to O(n)

- In this case, you put the key as the value, and the value you put how many times he appears.

- Example: [1, 2, 3, 1] -> {‘1’: 2, ‘2’: 1, ‘3’: 1}

**- Multiple Pointers**

- Create pointers or values that correspond to an index or position and move towards the beginning, middle or end based on a certain condition

- Very efficient for solving problems with minimal space complexity as well

- Create pointers in the beginning and end of the iterable object, and move towards to find pars

- Or create pointers next to each other to find unique values

**- Sliding Window**

- When you want to find a subset of data inside a larger set of data(array, string…)

**- Divide and Conquer**

- Divide a data set into smaller chunks and then repeating the process with a subset of data

- divide the problem into smaller pieces and repeat the process calling itself passing a smaller piece of data

**Recursion**

- Take a problem and divide into smaller pieces

- a function that calls itself

- Every time the function calls itself, must be a different input

- To puro recursive function, for arrays use methods like spread operator, concat or slice to not mutate the original arr. For strings, use methods like slice, substr or substring to make copies of the string. For objects, to make copies, use methods like Object.assign or the spread operator.

**- Call Stack**

- The order of the called function’s executions

- the order of execution goes from the top to the bottom, all new functions called goes to the top and must be executed first to the others below continues to get executed

**- Base Case**

- The condition when the recursion ends

- Mandatory verification

**- Regular Pitfalls**

- No or Wrong Base Case

- Returning the wrong thing. The “return” is very important, you always must return something.

**- Helper Method Recursion**

- Create a helper function inside the original function and then call itself

- It’s an outer function that is not recursive that calls a inner function that is recursive

- create a variable in the outside function to add the values from the inner function

**Searching Algorithms**

**- Linear Search**

- check every element, one at a time of array searching to the value passed as param

- used for an unordered array

- methods that use linear search: indexOf, includes, find, findIndex

- Big O Complexitie: O(n)

**- Binary Search**

- Only works on a sorted list

- You can eliminate half of the remaining elements at a time

- Take the middle of the array, and check if the element that you are looking for is greater or lower than the element of the middle of the array

- Then you eliminate the half that you know that the element it’s not there

- Use the divide and Conquer method (Recursion)

- Big O Complexitie: O(log n)

**Sorting**

- Sorting is the process of reorganizing in order items of any kind of data collection

- In different kinds of collections, some sorting algorithms will be better than others

**- Bubble Sort**

- Used when you know the data is nearly sorted

- compare a value and the next on the list, if the value is bigger than the next to him, swap then

- Time complexitie: O(n²)

**- Quick Sort**

- Use the divide and conquer method(recursion)

- Select an element, called “pivot” and compare every other elements in the array to se if the elements are bigger or smaller than the pivot, then you create 2 array with biggests elements and smaller elements an call the function recursively

**Data Structures**

**Singly Linked List**

- A data structure that it stores whatever sort of data

- Its ordered

- Consists of elements that has no indices and have just a reference to the next item

- Every element points to the next one

- Consists of nodes that has a value and a pointer to another node or null

- Every linked list has a **head** (first element), a **tail** (last element), and a **length**

Diagrama

Descrição gerada automaticamente

**- Lists**

- Do not have indexes

- Connected via nodes with a next pointer

- Random access is not allowed (must start from the head)

**- Arrays**

- Indexed in order

- Insertion and deletion can be expensive

- Can quickly be accessed at a specific index

- Lists are quicker in insertion or deletion, and arrays are quicker in accessing

**- Singly Linked List Big O Notation**

- Insertion -> O(1)

- Removal -> O(1) or O(n)

- Searching -> O(n)

- Access -> O(n)

**- Recap**

- Singly Linked Lists are an excellent alternative to arrays when insertion and deletion at the beginning are frequently required.

- Arrays contain a built-in index whereas Linked List do not have.

- The ideia of a list data structure that consists of nodes is the foundation for other data structures like stacks and queues.

**Doubly Linked List**

- Almost identical to Singly Linked List, expect every node has another pointer to the previous node

- No indexing too

- Uses more memory, but has more flexibility

Diagrama

Descrição gerada automaticamente