# Data Structures and Algorithms By Umar Saad

# Table of contents

# Strategy for solving a problem

1. State the problem clearly
2. Come up with some example in/outputs, trying to cover all edge cases
3. Come up with a solution
4. Implement the solution using 3. and fix bugs if any
5. Understand the complexity and identify inefficiencies
6. Fix the inefficiency, repeat steps 3-6

The problem at hand:

Alice has some cards with numbers written on them. She arranges the cards in decreasing order, and lays them out face down in a sequence on a table. She challenges Bob to pick out the card containing a given number by turning over as few cards as possible.Write a function to help Bob locate the card.

## 1. Stating the problem clearly

Essentially the problem is we need to find the specific card’s location **(index)** which holds the target **(target)** from a list of numbers **(nums)** that are in decreasing order with the least amount of card checks possible.

## 2a. Analysing the inputs and Outputs

Inputs

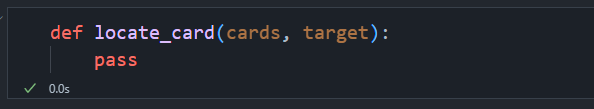
* List of the card: **Nums**
* The number we want to find: **Target**

Outputs

* The index of the card which holds target: **Position**

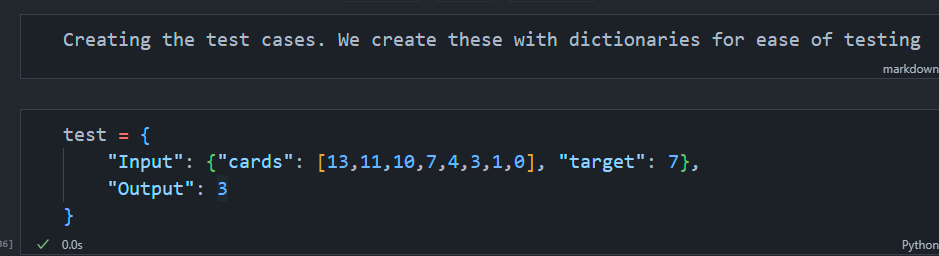
2b Now we can frame the function for out solution

* Ensure we use proper naming conventions for functions and variables
* Ensure we understand the question (ask)



## 2c. Creating Example input and outputs (Test Cases)

Now create the test case expected inputs and outputs as a dictionary

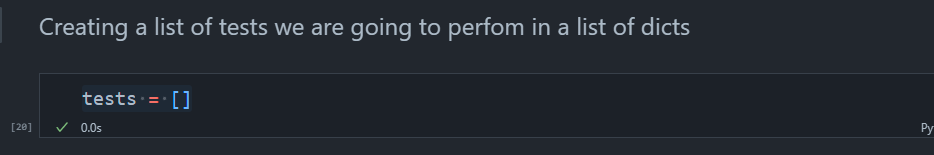


However our function should be able to handle ANY set of valid inputs, some of the potential edge cases are:

* Target is in the beginning, middle or end
* The list only contains one element, which is target
* The list doesn't contain target
* The list contains repeating numbers
* The target appears more than once

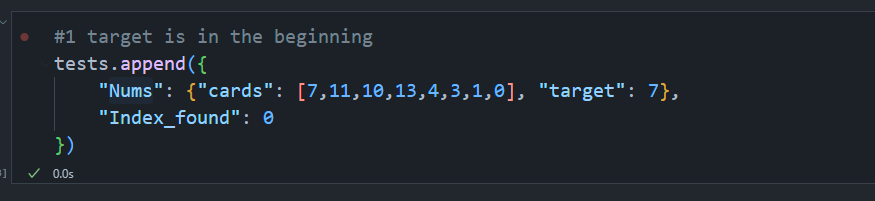
But ensure you ask about edge cases that are not written

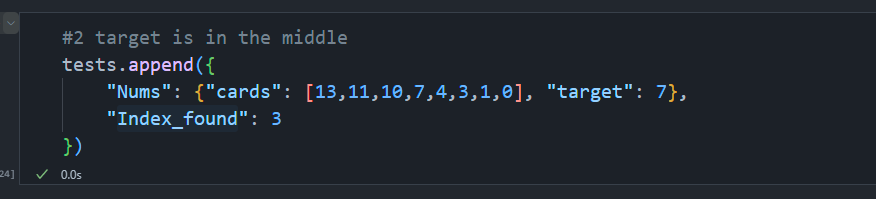
To solve these test cases we create a list called tests which holds each test for the edge cases

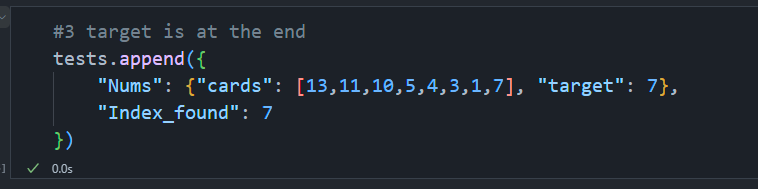


And then we append our edge cases we mentioned above

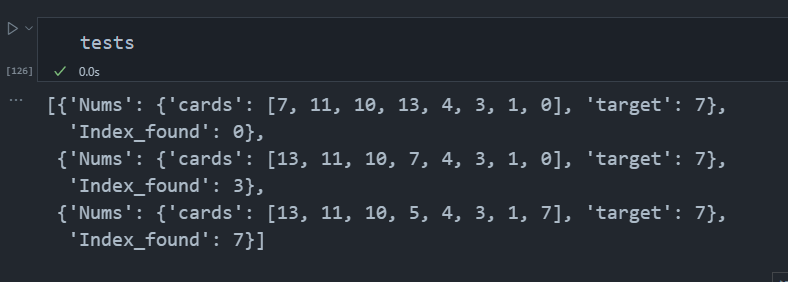
* Target is in the beginning, middle or end







So now we can see our tests list (used 3 examples of edge cases)



## 3. Think of a solution

To think of solution we do NOT start with code, first start by explaining the solution in PLAIN English.

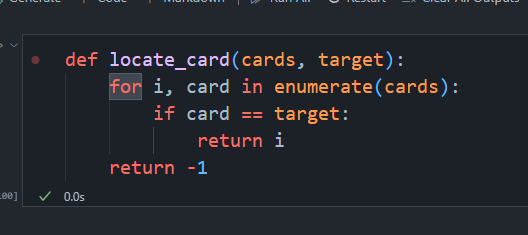
* First aim for correctness

This first solution doesn't have to be the BEST solution of the most Efficient. For now we just need to solve the problem.

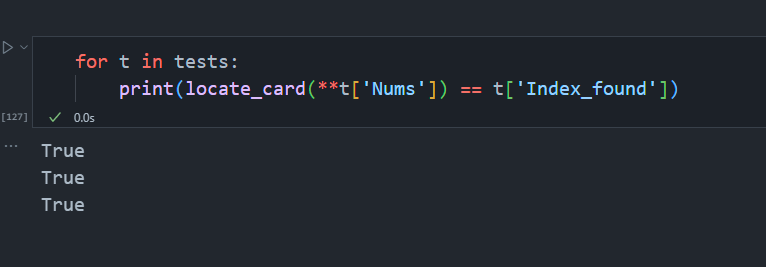
*In plain English:*

*Thinking of the problem at hand, the easy way to solve this is by looping through each card until I find the target, brute force. Loop through with an index and value and find the card with the value and return the index of that number. If the number isn't found we can return -1.*

## 4. The solution we have come up with is a linear search algorithm/

**

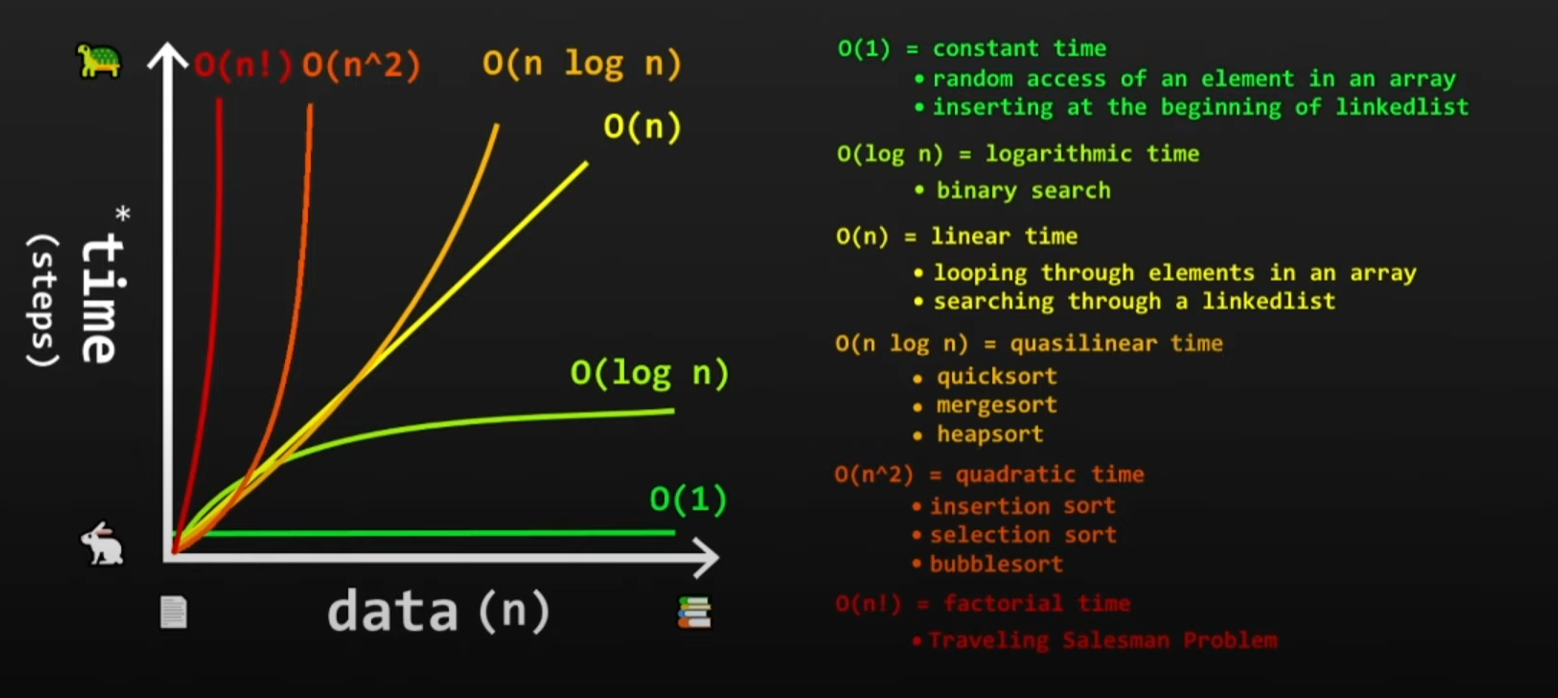
*And as we can see our 3 test cases have passed as they have found the target as we check if the input.*

**

* After correctness, aim for efficiency

## 5. Complexity & Big O notation:

Now analyse the complexity and efficiency of our current solution. *Right now were Turning each card 1 by 1 using enumerate. Since size of the list is N meaning that we access the list N times. So in the worst case we need to access N times (length of list, last index is target) to find target.*

[](https://www.youtube.com/watch?v=XMUe3zFhM5c)

*Complexity meaning the amount of time/space required by an algorithm for an input of a given size (N). Complexity will always refer to the worst case complexity for algorithm at hand for the specific problem. E.g. The highest time/space taken.*

*The time complexity for our solution (linear search search) is cN. (c) is the constant which is the number of operations and time per operation we do per iteration, in our case (checking if the number in the index is the target).*

*HOWEVER in big O we dont use constants (c), cN = O(N), N is always the input*

* **O(n)**

**If checking each item takes 3 seconds, it doesn’t matter if (n) is 3 seconds or 5 or 1million seconds or operations per item, the pattern of growth is still linear as the list gets bigger.**

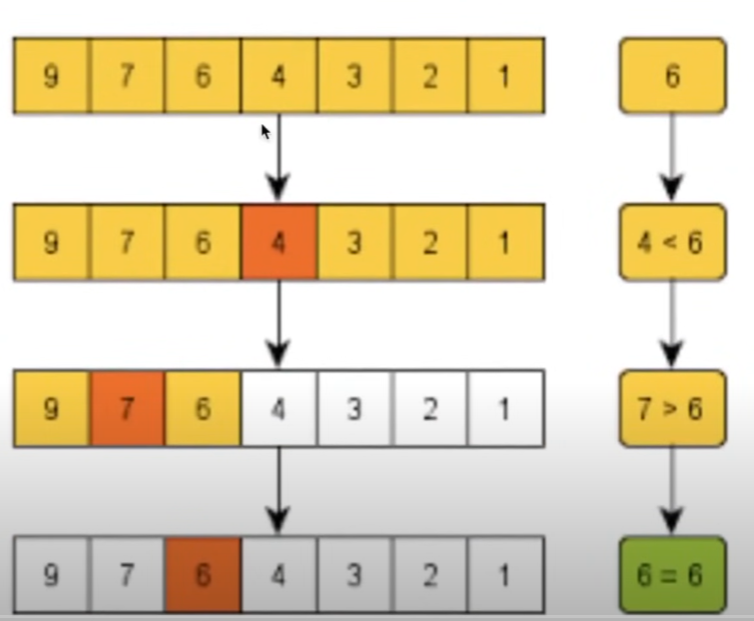
* **O(n2):**

*The space complexity for our solution (linear search search) is O(1) meaning constant no growth in space) because we are only holding the index of each element as we go, the index is always being changed so we dont require any more space, only one variable in storage (Index).*

* *Overcoming the inefficiency with our linear search (brute force) solution using big O*

Currently we know that our list is sorted which we can use to our advantage. For example if our target was the card with the number 7, lets pick the middle card, if the card is 5 that means all the cards before 5 cannot be 7 since the list is sorted, resulting in half the list not even needing to be checked. We can now check the list after the middle card.

This algorithm is a **binary search. A seen below in the example we are trying to find 6 instead of 7, and we keep halving the length of the list until till we find 6.**

[](https://www.youtube.com/watch?v=XMUe3zFhM5c)

As mentioned above now that we have finished step 6 as mentioned we now need to to go back and re-do steps 3-6.

3b. Come up with a solution in plain English

Above we agreed to use a binary search. In our problem they way we will sue the search is:

1. Find the middle element of the list
2. If the middle element is the target, return the element
3. If it is less than the target, then search the first half
4. If it is greater, then search the second half
5. If it is not found, return -1

4. Now we have to implement our new solution

#notes

#stopped at 1 hour 6 minutes

Add the test cases into binary-search Jupiter notebook to refine skills on test casing

Work more and understand binary search