# Introduction to Binary Search and Complexity Analysis with Python

## Part 1 of "Data Structures and Algorithms in Python"

Data Structures and Algorithms in Python is beginner-friendly introduction to common data structures (linked lists, stacks, queues, graphs) and algorithms (search, sorting, recursion, dynamic programming) in Python, designed to help you prepare for coding interviews and assessments. Check out the full series here:

1. Binary Search and Complexity Analysis
2. Python Classes and Linked Lists
3. Arrays, Stacks, Queues and Strings (coming soon)
4. Binary Search Trees and Hash Tables (coming soon)
5. Insertion Sort, Merge Sort and Divide-and-Conquer (coming soon)
6. Quicksort, Partitions and Average-case Complexity (coming soon)
7. Recursion, Backtracking and Dynamic Programming (coming soon)
8. Knapsack, Subsequence and Matrix Problems (coming soon)
9. Graphs, Breadth-First Search and Depth-First Search (coming soon)
10. Shortest Paths, Spanning Trees & Topological Sorting (coming soon)
11. Disjoint Sets and the Union Find Algorithm (coming soon)
12. Interview Questions, Tips & Practical Advice (coming soon)

Earn a verified certificate of accomplishment for this course by signing up here: http://pythondsa.com .

Ask questions, get help & participate in discussions on the community forum: https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/78

### Prerequisites

This course assumes very little background in programming and mathematics, and you can learn the required concepts here:

- Basic programming with Python (variables, data types, loops, functions etc.)
- Some high school mathematics (polynomials, vectors, matrices and probability)
- No prior knowledge of data structures or algorithms is required

We'll cover any additional mathematical and theoretical concepts we need as we go along.

## How to Run the Code

The best way to learn the material is to execute the code and experiment with it yourself. This tutorial is an executable Jupyter notebook. You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

### Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on Google Colab or Kaggle to use these platforms.

### Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up Python, download the notebook and install the required libraries. We recommend using the Conda distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

> **Jupyter Notebooks**: This notebook is made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Try executing the cells below:

```python
# Import a library module
import math
```

```python
# Use a function from the library
math.sqrt(49)
```

```
7.0
```

## Problem

This course takes a coding-focused approach towards learning. In each notebook, we'll focus on solving one problem, and learn the techniques, algorithms, and data structures to devise an *efficient* solution. We will then generalize the technique and apply it to other problems.

In this notebook, we focus on solving the following problem:

> **QUESTION 1:** 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.

This may seem like a simple problem, especially if you're familiar with the concept of *binary search*, but the strategy and technique we learning here will be widely applicable, and we'll soon use it to solve harder problems.

## Why You Should Learn Data Structures and Algorithms

Whether you're pursuing a career in software development or data science, it's almost certain that you'll be asked to solve programming problems like *reversing a linked list* or *balancing a binary tree* in a technical interview or coding assessment.

It's well known, however, that you will almost never face these problems in your job as a software developer. So it's reasonable to wonder why such problems are asked in interviews and coding assessments. Solving programming problems demonstrates the following traits:

1. You can **think about a problem systematically** and solve it systematically step-by-step.
2. You can **envision different inputs, outputs, and edge cases** for programs you write.
3. You can **communicate your ideas clearly** to co-workers and incorporate their suggestions.
4. Most importantly, you can **convert your thoughts and ideas into working code** that's also readable.

It's not your knowledge of specific data structures or algorithms that's tested in an interview, but your approach towards the problem. You may fail to solve the problem and still clear the interview or vice versa. In this course, you will learn the skills to both solve problems and clear interviews successfully.

## The Method

Upon reading the problem, you may get some ideas on how to solve it and your first instinct might be to start writing code. This is not the optimal strategy and you may end up spending a longer time trying to solve the problem due to coding errors, or may not be able to solve it at all.

Here's a systematic strategy we'll apply for solving problems:

1. State the problem clearly. Identify the input & output formats.
2. Come up with some example inputs & outputs. Try to cover all edge cases.
3. Come up with a correct solution for the problem. State it in plain English.
4. Implement the solution and test it using example inputs. Fix bugs, if any.
5. Analyze the algorithm's complexity and identify inefficiencies, if any.
6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

*"Applying the right technique"* is where the knowledge of common data structures and algorithms comes in handy.

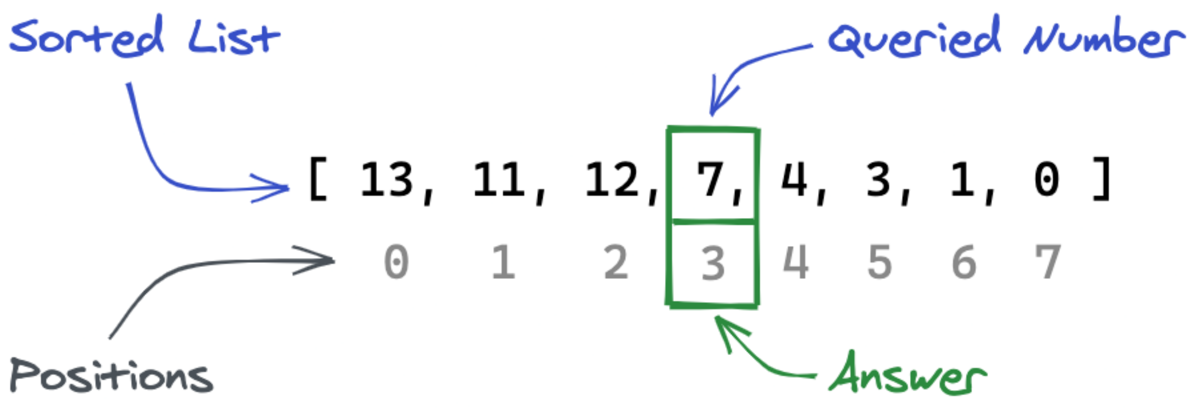
Use this template for solving problems by applying this method: [https://jovian.ai/aakashns/python-problem-solving-template](https://jovian.ai/aakashns/python-problem-solving-template)

## Solution

### 1. State the problem clearly. Identify the input & output formats.

You will often encounter detailed word problems in coding challenges and interviews. The first step is to state the problem clearly and precisely in abstract terms.

In this case, for instance, we can represent the sequence of cards as a list of numbers. Turning over a specific card is equivalent to accessing the value of the number at the corresponding position the list.



The problem can now be stated as follows:

## Problem

> We need to write a program to find the position of a given number in a list of numbers arranged in decreasing order. We also need to minimize the number of times we access elements from the list.

## Input

1. `cards`: A list of numbers sorted in decreasing order. E.g. `[13, 11, 10, 7, 4, 3, 1, 0]`
2. `query`: A number, whose position in the array is to be determined. E.g. `7`

## Output

3. `position`: The position of `query` in the list `cards`. E.g. `3` in the above case (counting from `0`)

Based on the above, we can now create the signature of our function:

```python
def locate_card(cards, query):
    pass
```

**Tips**:

- Name your function appropriately and think carefully about the signature
- Discuss the problem with the interviewer if you are unsure how to frame it in abstract terms
- Use descriptive variable names, otherwise you may forget what a variable represents

## 2. Come up with some example inputs & outputs. Try to cover all edge cases.

Before we start implementing our function, it would be useful to come up with some example inputs and outputs which we can use later to test out problem. We'll refer to them as *test cases*.

Here's the test case described in the example above.

```
cards = [13, 11, 10, 7, 4, 3, 1, 0]
query = 7
output = 3
```

We can test our function by passing the inputs into function and comparing the result with the expected output.

```
result = locate_card(cards, query)
print(result)
```

```
None
```

```
result == output
```

```
False
```

Obviously, the two result does not match the output as we have not yet implemented the function.

We'll represent our test cases as dictionaries to make it easier to test them once we write implement our function. For example, the above test case can be represented as follows:

```
test = {
    'input': {
        'cards': [13, 11, 10, 7, 4, 3, 1, 0],
        'query': 7
    },
    'output': 3
}
```

The function can now be tested as follows.

```
locate_card(**test['input']) == test['output']
```

```
False
```

Our function should be able to handle any set of valid inputs we pass into it. Here's a list of some possible variations we might encounter:

1. The number `query` occurs somewhere in the middle of the list `cards`.
2. `query` is the first element in `cards`.
3. `query` is the last element in `cards`.
4. The list `cards` contains just one element, which is `query`.
5. The list `cards` does not contain number `query`.
6. The list `cards` is empty.
7. The list `cards` contains repeating numbers.
8. The number `query` occurs at more than one position in `cards`.
9. (can you think of any more variations?)

> **Edge Cases**: It's likely that you didn't think of all of the above cases when you read the problem for the first time. Some of these (like the empty array or `query` not occurring in `cards`) are called *edge cases*, as they represent rare or extreme examples.

While edge cases may not occur frequently, your programs should be able to handle all edge cases, otherwise they may fail in unexpected ways. Let's create some more test cases for the variations listed above. We'll store all our test cases in an list for easier testing.

```
tests = []
```

```
# query occurs in the middle
tests.append(test)

tests.append({
    'input': {
        'cards': [13, 11, 10, 7, 4, 3, 1, 0],
        'query': 1
    },
    'output': 6
})
```

```
# query is the first element
tests.append({
    'input': {
        'cards': [4, 2, 1, -1],
        'query': 4
    },
    'output': 0
})
```

```
# query is the last element
tests.append({
    'input': {
        'cards': [3, -1, -9, -127],
        'query': -127
    },
    'output': 3
})
```

```
# cards contains just one element, query
tests.append({
    'input': {
        'cards': [6],
        'query': 6
    },
    'output': 0
})
```

The problem statement does not specify what to do if the list `cards` does not contain the number `query`.

1. Read the problem statement again, carefully.
2. Look through the examples provided with the problem.
3. Ask the interviewer/platform for a clarification.
4. Make a reasonable assumption, state it and move forward.

We will assume that our function will return `-1` in case `cards` does not contain `query`.

```
# cards does not contain query
tests.append({
    'input': {
        'cards': [9, 7, 5, 2, -9],
        'query': 4
    },
    'output': -1
})
```

```
# cards is empty
tests.append({
    'input': {
        'cards': [],
        'query': 7
    },
    'output': -1
})
```

```
# numbers can repeat in cards
tests.append({
    'input': {
        'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0],
        'query': 3
    },
    'output': 7
})
```

In the case where `query` occurs multiple times in `cards`, we'll expect our function to return the first occurrence of `query`.

While it may also be acceptable for the function to return any position where `query` occurs within the list, it would be slightly more difficult to test the function, as the output is non-deterministic.

```
# query occurs multiple times
tests.append({
    'input': {
        'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0],
        'query': 6
    },
    'output': 2
})
```

Let's look at the full set of test cases we have created so far.

```
tests
```

```
[{'input': {'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}, 'output': 3},
 {'input': {'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}, 'output': 6},
 {'input': {'cards': [4, 2, 1, -1], 'query': 4}, 'output': 0},
 {'input': {'cards': [3, -1, -9, -127], 'query': -127}, 'output': 3},
 {'input': {'cards': [6], 'query': 6}, 'output': 0},
 {'input': {'cards': [9, 7, 5, 2, -9], 'query': 4}, 'output': -1},
 {'input': {'cards': [], 'query': 7}, 'output': -1},
 {'input': {'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 3},
  'output': 7},
 {'input': {'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0],
   'query': 6},
  'output': 2}]
```

Great, now we have a fairly exhaustive set of test cases to evaluate our function.

Creating test cases beforehand allows you to identify different variations and edge cases in advance so that can make sure to handle them while writing code. Sometimes, you may start out confused, but the solution will reveal itself as you try to come up with interesting test cases.

**Tip:** Don't stress it if you can't come up with an exhaustive list of test cases though. You can come back to this section and add more test cases as you discover them. Coming up with good test cases is a skill that takes practice.

## 3. Come up with a correct solution for the problem. State it in plain English.

Our first goal should always be to come up with a *correct* solution to the problem, which may necessarily be the most *efficient* solution. The simplest or most obvious solution to a problem, which generally involves checking all possible answers is called the *brute force* solution.

In this problem, coming up with a correct solution is quite easy: Bob can simply turn over cards in order one by one, till he find a card with the given number on it. Here's how we might implement it:

1. Create a variable `position` with the value 0.
2. Check whether the number at index `position` in `card` equals `query`.
3. If it does, `position` is the answer and can be returned from the function
4. If not, increment the value of `position` by 1, and repeat steps 2 to 5 till we reach the last position.
5. If the number was not found, return `-1`.

> **Linear Search Algorithm**: Congratulations, we've just written our first *algorithm*! An algorithm is simply a list of statements which can be converted into code and executed by a computer on different sets of inputs. This particular algorithm is called linear search, since it involves searching through a list in a linear fashion i.e. element after element.

**Tip:** Always try to express (speak or write) the algorithm in your own words before you start coding. It can be as brief or detailed as you require it to be. Writing is a great tool for thinking clearly. It's likely that you will find some parts of the solution difficult to express, which suggests that you are probably unable to think about it clearly. The more clearly you are able to express your thoughts, the easier it will be for you to turn into code.

## 4. Implement the solution and test it using example inputs. Fix bugs, if any.

Phew! We are finally ready to implement our solution. All the work we've done so far will definitely come in handy, as we now exactly what we want our function to do, and we have an easy way of testing it on a variety of inputs.

Here's a first attempt at implementing the function.

```
def locate_card(cards, query):
    # Create a variable position with the value 0
    position = 0

    # Set up a loop for repetition
    while True:

        # Check if element at the current position matche the query
        if cards[position] == query:

            # Answer found! Return and exit..
            return position

        # Increment the position
        position += 1

        # Check if we have reached the end of the array
        if position == len(cards):

            # Number not found, return -1
            return -1
```

Let's test out the function with the first test case

```
test
```

```
{'input': {'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}, 'output': 3}
```

```
result = locate_card(test['input']['cards'], test['input']['query'])
result
```

```
3
```

```
result == output
```

```
True
```

Yay! The result matches the output.

To help you test your functions easily the `jovian` Python library provides a helper function `evalute_test_case`. Apart from checking whether the function produces the expected result, it also displays the input, expected output, actual output from the function, and the execution time of the function.

```
!pip install jovian --upgrade --quiet
```

```
from jovian.pythondsa import evaluate_test_case
```

```
evaluate_test_case(locate_card, test)
```

```
Input:
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}

Expected Output:
3


Actual Output:
3

Execution Time:
0.004 ms

Test Result:
PASSED


(3, True, 0.004)
```

While it may seem like we have a working solution based on the above test, we can't be sure about it until we test the function with all the test cases.

We can use the `evaluate_test_cases` (plural) function from the `jovian` library to test our function on all the test cases with a single line of code.

```
from jovian.pythondsa import evaluate_test_cases
```

```
evaluate_test_cases(locate_card, tests)
```

```
TEST CASE #0

Input:
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}
```

```
Expected Output:
3


Actual Output:
3

Execution Time:
0.004 ms

Test Result:
PASSED


TEST CASE #1

Input:
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}

Expected Output:
6


Actual Output:
6

Execution Time:
0.003 ms

Test Result:
PASSED


TEST CASE #2

Input:
{'cards': [4, 2, 1, -1], 'query': 4}

Expected Output:
0


Actual Output:
```

```
0

Execution Time:
0.001 ms

Test Result:
PASSED


TEST CASE #3

Input:
{'cards': [3, -1, -9, -127], 'query': -127}

Expected Output:
3


Actual Output:
3

Execution Time:
0.004 ms

Test Result:
PASSED


TEST CASE #4

Input:
{'cards': [6], 'query': 6}

Expected Output:
0


Actual Output:
0

Execution Time:
0.005 ms
```

```
Test Result:
PASSED


TEST CASE #5

Input:
{'cards': [9, 7, 5, 2, -9], 'query': 4}

Expected Output:
-1


Actual Output:
-1

Execution Time:
0.004 ms

Test Result:
PASSED


TEST CASE #6
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-27-2111dddaa84c> in <module>
----> 1 evaluate_test_cases(locate_card, tests)

~/miniconda3/envs/pythondsa/lib/python3.6/site-packages/jovian/pythondsa/__init__.py in
evaluate_test_cases(function, test_cases, error_only)
     69         if not error_only:
     70             print("\n\033[1mTEST CASE #{}\033[0m".format(i))
---> 71         result = evaluate_test_case(function, test_case, display=False)
     72         results.append(result)
     73         if error_only and not result[1]:

~/miniconda3/envs/pythondsa/lib/python3.6/site-packages/jovian/pythondsa/__init__.py in
evaluate_test_case(function, test_case, display)
     50
     51     start = timer()
---> 52     actual_output = function(**inputs)
     53     end = timer()
     54

<ipython-input-19-9ed30c367c36> in locate_card(cards, query)
```

```
      7
      8         # Check if element at the current position matche the query
----> 9         if cards[position] == query:
     10
     11             # Answer found! Return and exit..

IndexError: list index out of range
```

Oh no! Looks like our function encountered an error in the sixth test case. The error message suggests that we're trying to access an index outside the range of valid indices in the list. Looks like the list `cards` is empty in this case, and may be the root of the problem.

Let's add some `print` statements within our function to print the inputs and the value of the `position` variable in each loop.

```python
def locate_card(cards, query):
    position = 0

    print('cards:', cards)
    print('query:', query)

    while True:
        print('position:', position)

        if cards[position] == query:
            return position

        position += 1
        if position == len(cards):
            return -1
```

```python
cards6 = tests[6]['input']['cards']
query6 = tests[6]['input']['query']

locate_card(cards6, query6)
```

```
cards: []
query: 7
position: 0
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-29-5f727cb3c2c3> in <module>
      2 query6 = tests[6]['input']['query']
      3
----> 4 locate_card(cards6, query6)

<ipython-input-28-aabbbc74d5cf> in locate_card(cards, query)
      8         print('position:', position)
      9
---> 10         if cards[position] == query:
```

```
              return position

IndexError: list index out of range
```

Clearly, since `cards` is empty, it's not possible to access the element at index 0. To fix this, we can check whether we've reached the end of the array before trying to access an element from it. In fact, this can be terminating condition for the `while` loop itself.

```
def locate_card(cards, query):
    position = 0
    while position < len(cards):
        if cards[position] == query:
            return position
        position += 1
    return -1
```

Let's test the failing case again.

```
tests[6]
```

```
{'input': {'cards': [], 'query': 7}, 'output': -1}
```

```
locate_card(cards6, query6)
```

```
-1
```

The result now matches the expected output. Do you now see the benefit of listing test cases beforehand? Without a good set of test cases, we may never have discovered this error in our function.

Let's verify that all the other test cases pass too.

```
evaluate_test_cases(locate_card, tests)
```

```

TEST CASE #0

Input:
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}

Expected Output:
3


Actual Output:
3

Execution Time:
```

```
0.004 ms

Test Result:
PASSED


TEST CASE #1

Input:
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}

Expected Output:
6


Actual Output:
6

Execution Time:
0.006 ms

Test Result:
PASSED


TEST CASE #2

Input:
{'cards': [4, 2, 1, -1], 'query': 4}

Expected Output:
0


Actual Output:
0

Execution Time:
0.002 ms

Test Result:
PASSED
```

```
TEST CASE #3

Input:
{'cards': [3, -1, -9, -127], 'query': -127}

Expected Output:
3


Actual Output:
3

Execution Time:
0.003 ms

Test Result:
PASSED


TEST CASE #4

Input:
{'cards': [6], 'query': 6}

Expected Output:
0


Actual Output:
0

Execution Time:
0.002 ms

Test Result:
PASSED


TEST CASE #5

Input:
{'cards': [9, 7, 5, 2, -9], 'query': 4}
```

```
Expected Output:
-1


Actual Output:
-1

Execution Time:
0.003 ms

Test Result:
PASSED


TEST CASE #6

Input:
{'cards': [], 'query': 7}

Expected Output:
-1


Actual Output:
-1

Execution Time:
0.004 ms

Test Result:
PASSED


TEST CASE #7

Input:
{'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 3}

Expected Output:
7
```

```
Actual Output:
7

Execution Time:
0.004 ms

Test Result:
PASSED


TEST CASE #8

Input:
{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}

Expected Output:
2


Actual Output:
2

Execution Time:
0.002 ms

Test Result:
PASSED


SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0
```

```
[(3, True, 0.004),
 (6, True, 0.006),
 (0, True, 0.002),
 (3, True, 0.003),
 (0, True, 0.002),
 (-1, True, 0.003),
 (-1, True, 0.004),
 (7, True, 0.004),
 (2, True, 0.002)]
```

Our code passes all the test cases. Of course, there might be some other edge cases we haven't thought of which may cause the function to fail. Can you think of any?

**Tip**: In a real interview or coding assessment, you can skip the step of implementing and testing the brute force solution in the interest of time. It's generally quite easy to figure out the complexity of the brute for solution from the plain English description.

## 5. Analyze the algorithm's complexity and identify inefficiencies, if any.

Recall this statement from original question: *"Alice challenges Bob to pick out the card containing a given number by **turning over as few cards as possible**."* We restated this requirement as: *"Minimize the number of times we access elements from the list `cards` "*

Before we can minimize the number, we need a way to measure it. Since we access a list element once in every iteration, for a list of size `N` we access the elements from the list up to `N` times. Thus, Bob may need to overturn up to `N` cards in the worst case, to find the required card.

Suppose he is only allowed to overturn 1 card per minute, it may take him 30 minutes to find the required card if 30 cards are laid out on the table. Is this the best he can do? Is a way for Bob to arrive at the answer by turning over just 5 cards, instead of 30?

The field of study concerned with finding the amount of time, space or other resources required to complete the execution of computer programs is called *the analysis of algorithms*. And the process of figuring out the best algorithm to solve a given problem is called *algorithm design and optimization*.

## Complexity and Big O Notation

> **Complexity** of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size e.g. `N`. Unless otherwise stated, the term *complexity* always refers to the worst-case complexity (i.e. the highest possible time/space taken by the program/algorithm to process an input).

In the case of linear search:

1. The *time complexity* of the algorithm is `cN` for some fixed constant `c` that depends on the number of operations we perform in each iteration and the time taken to execute a statement. Time complexity is sometimes also called the *running time* of the algorithm.
2. The *space complexity* is some constant `c'` (independent of `N`), since we just need a single variable `position` to iterate through the array, and it occupies a constant space in the computer's memory (RAM).

> **Big O Notation**: Worst-case complexity is often expressed using the Big O notation. In the Big O, we drop fixed constants and lower powers of variables to capture the trend of relationship between the size of the input and the complexity of the algorithm i.e. if the complexity of the algorithm is `cN^3 + dN^2 + eN + f`, in the Big O notation it is expressed as **O(N^3)**

Thus, the time complexity of linear search is **O(N)** and its space complexity is **O(1)**.

## Save and upload your work to Jovian

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. Jovian offers an easy way of saving and sharing your Jupyter notebooks online.

```
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit(project='python-binary-search', environment=None)
```

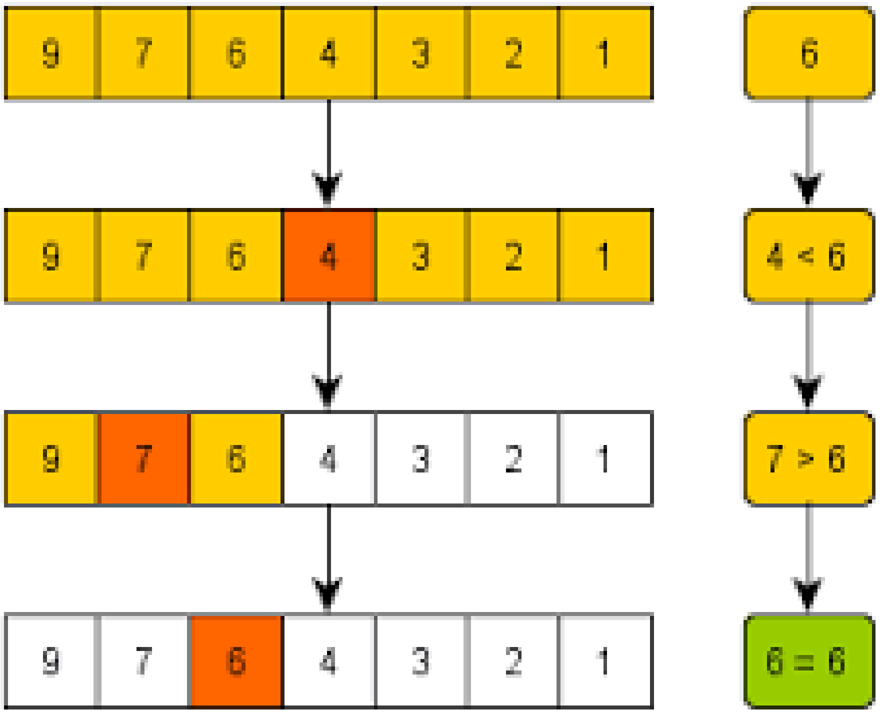
```
[jovian] Attempting to save notebook..
```

## 6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

At the moment, we're simply going over cards one by one, and not even utilizing the face that they're sorted. This is called a *brute force* approach.

It would be great if Bob could somehow guess the card at the first attempt, but with all the cards turned over it's simply impossible to guess the right card.

The next best idea would be to pick a random card, and use the fact that the list is sorted, to determine whether the target card lies to the left or right of it. In fact, if we pick the middle card, we can reduce the number of additional cards to be tested to half the size of the list. Then, we can simply repeat the process with each half. This technique is called binary search. Here's a visual explanation of the technique:



## 7. Come up with a correct solution for the problem. State it in plain English.

Here's how binary search can be applied to our problem:

1. Find the middle element of the list.
2. If it matches queried number, return the middle position as the answer.
3. If it is less than the queried number, then search the first half of the list
4. If it is greater than the queried number, then search the second half of the list
5. If no more elements remain, return -1.

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
[jovian] Updating notebook "aakashns/python-binary-search" on https://jovian.ai/
[jovian] Uploading notebook..
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/aakashns/python-binary-search
```

```
'https://jovian.ai/aakashns/python-binary-search'
```

## 8. Implement the solution and test it using example inputs. Fix bugs, if any.

Here's an implementation of binary search for solving our problem. We also print the relevant variables in each iteration of the `while` loop.

```
def locate_card(cards, query):
    lo, hi = 0, len(cards) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        mid_number = cards[mid]

        print("lo:", lo, ", hi:", hi, ", mid:", mid, ", mid_number:", mid_number)

        if mid_number == query:
            return mid
        elif mid_number < query:
            hi = mid - 1
        elif mid_number > query:
            lo = mid + 1

    return -1
```

Let's test it out using the test cases.

```
evaluate_test_cases(locate_card, tests)
```

```

TEST CASE #0
lo: 0 , hi: 7 , mid: 3 , mid_number: 7

Input:
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}

Expected Output:
3


Actual Output:
3

Execution Time:
0.884 ms

Test Result:
PASSED



TEST CASE #1
lo: 0 , hi: 7 , mid: 3 , mid_number: 7
lo: 4 , hi: 7 , mid: 5 , mid_number: 3
lo: 6 , hi: 7 , mid: 6 , mid_number: 1

Input:
```

```
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}

Expected Output:
6


Actual Output:
6

Execution Time:
0.541 ms

Test Result:
PASSED


TEST CASE #2
lo: 0 , hi: 3 , mid: 1 , mid_number: 2
lo: 0 , hi: 0 , mid: 0 , mid_number: 4

Input:
{'cards': [4, 2, 1, -1], 'query': 4}

Expected Output:
0


Actual Output:
0

Execution Time:
0.489 ms

Test Result:
PASSED


TEST CASE #3
lo: 0 , hi: 3 , mid: 1 , mid_number: -1
lo: 2 , hi: 3 , mid: 2 , mid_number: -9
lo: 3 , hi: 3 , mid: 3 , mid_number: -127

Input:
```

```
{'cards': [3, -1, -9, -127], 'query': -127}

Expected Output:
3


Actual Output:
3

Execution Time:
0.727 ms

Test Result:
PASSED


TEST CASE #4
lo: 0 , hi: 0 , mid: 0 , mid_number: 6

Input:
{'cards': [6], 'query': 6}

Expected Output:
0


Actual Output:
0

Execution Time:
0.12 ms

Test Result:
PASSED


TEST CASE #5
lo: 0 , hi: 4 , mid: 2 , mid_number: 5
lo: 3 , hi: 4 , mid: 3 , mid_number: 2

Input:
{'cards': [9, 7, 5, 2, -9], 'query': 4}
```

```
Expected Output:
-1


Actual Output:
-1

Execution Time:
0.382 ms

Test Result:
PASSED


TEST CASE #6

Input:
{'cards': [], 'query': 7}

Expected Output:
-1


Actual Output:
-1

Execution Time:
0.002 ms

Test Result:
PASSED


TEST CASE #7
lo: 0 , hi: 13 , mid: 6 , mid_number: 6
lo: 7 , hi: 13 , mid: 10 , mid_number: 2
lo: 7 , hi: 9 , mid: 8 , mid_number: 2
lo: 7 , hi: 7 , mid: 7 , mid_number: 3

Input:
{'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 3}

Expected Output:
```

```
7


Actual Output:
7

Execution Time:
0.626 ms

Test Result:
PASSED



TEST CASE #8
lo: 0 , hi: 14 , mid: 7 , mid_number: 6

Input:
{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}

Expected Output:
2


Actual Output:
7

Execution Time:
0.106 ms

Test Result:
FAILED



SUMMARY

TOTAL: 9, PASSED: 8, FAILED: 1
[(3, True, 0.884),
 (6, True, 0.541),
 (0, True, 0.489),
 (3, True, 0.727),
 (0, True, 0.12),
 (-1, True, 0.382),
 (-1, True, 0.002),
 (7, True, 0.626),
 (7, False, 0.106)]
```

Looks like it passed 8 out of 9 tests! Let's look at the failed test.

```
evaluate_test_case(locate_card, tests[8])
```

```
Input:
{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}

Expected Output:
2

lo: 0 , hi: 14 , mid: 7 , mid_number: 6

Actual Output:
7

Execution Time:
0.341 ms

Test Result:
FAILED


(7, False, 0.341)
```

Seems like our function returned the position `7`. Let's check what lies at this position in the input list.

```
cards8 = tests[8]['input']['cards']
query8 = tests[8]['input']['cards']
```

```
query8[7]
```

```
6
```

Seems like we did locate a 6 in the array, it's just that it wasn't the first 6. As you can guess, this is because in binary search, we don't go over indices in a linear order.

So how do we fix it?

When we find that `cards[mid]` is equal to `query`, we need to check whether it is the first occurrence of `query` in the list i.e the number that comes before it.

```
[8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0]
```

To make it easier, we'll define a helper function called `test_location`, which will take the list `cards`, the `query and mid` as inputs.

```python
def test_location(cards, query, mid):
    mid_number = cards[mid]
    print("mid:", mid, ", mid_number:", mid_number)
    if mid_number == query:
        if mid-1 >= 0 and cards[mid-1] == query:
            return 'left'
        else:
            return 'found'
    elif mid_number < query:
        return 'left'
    else:
        return 'right'

def locate_card(cards, query):
    lo, hi = 0, len(cards) - 1

    while lo <= hi:
        print("lo:", lo, ", hi:", hi)
        mid = (lo + hi) // 2
        result = test_location(cards, query, mid)

        if result == 'found':
            return mid
        elif result == 'left':
            hi = mid - 1
        elif result == 'right':
            lo = mid + 1
    return -1
```

```python
evaluate_test_case(locate_card, tests[8])
```

```
Input:
{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}

Expected Output:
2

lo: 0 , hi: 14
mid: 7 , mid_number: 6
lo: 0 , hi: 6
mid: 3 , mid_number: 6
lo: 0 , hi: 2
mid: 1 , mid_number: 8
lo: 2 , hi: 2
mid: 2 , mid_number: 6
```

```
Actual Output:
2

Execution Time:
0.969 ms

Test Result:
PASSED


(2, True, 0.969)
```

```
evaluate_test_cases(locate_card, tests)
```

```
TEST CASE #0
lo: 0 , hi: 7
mid: 3 , mid_number: 7

Input:
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}

Expected Output:
3


Actual Output:
3

Execution Time:
0.224 ms

Test Result:
PASSED


TEST CASE #1
lo: 0 , hi: 7
mid: 3 , mid_number: 7
lo: 4 , hi: 7
mid: 5 , mid_number: 3
lo: 6 , hi: 7
mid: 6 , mid_number: 1
```

```
Input:
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}

Expected Output:
6


Actual Output:
6

Execution Time:
0.581 ms

Test Result:
PASSED


TEST CASE #2
lo: 0 , hi: 3
mid: 1 , mid_number: 2
lo: 0 , hi: 0
mid: 0 , mid_number: 4

Input:
{'cards': [4, 2, 1, -1], 'query': 4}

Expected Output:
0


Actual Output:
0

Execution Time:
0.313 ms

Test Result:
PASSED


TEST CASE #3
lo: 0 , hi: 3
mid: 1 , mid_number: -1
```

```
lo: 2 , hi: 3
mid: 2 , mid_number: -9
lo: 3 , hi: 3
mid: 3 , mid_number: -127

Input:
{'cards': [3, -1, -9, -127], 'query': -127}

Expected Output:
3


Actual Output:
3

Execution Time:
47.893 ms

Test Result:
PASSED


TEST CASE #4
lo: 0 , hi: 0
mid: 0 , mid_number: 6

Input:
{'cards': [6], 'query': 6}

Expected Output:
0


Actual Output:
0

Execution Time:
0.13 ms

Test Result:
PASSED
```

```
TEST CASE #5
lo: 0 , hi: 4
mid: 2 , mid_number: 5
lo: 3 , hi: 4
mid: 3 , mid_number: 2

Input:
{'cards': [9, 7, 5, 2, -9], 'query': 4}

Expected Output:
-1


Actual Output:
-1

Execution Time:
0.319 ms

Test Result:
PASSED


TEST CASE #6

Input:
{'cards': [], 'query': 7}

Expected Output:
-1


Actual Output:
-1

Execution Time:
0.002 ms

Test Result:
PASSED


TEST CASE #7
```

```
lo: 0 , hi: 13
mid: 6 , mid_number: 6
lo: 7 , hi: 13
mid: 10 , mid_number: 2
lo: 7 , hi: 9
mid: 8 , mid_number: 2
lo: 7 , hi: 7
mid: 7 , mid_number: 3

Input:
{'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 3}

Expected Output:
7


Actual Output:
7

Execution Time:
0.684 ms

Test Result:
PASSED


TEST CASE #8
lo: 0 , hi: 14
mid: 7 , mid_number: 6
lo: 0 , hi: 6
mid: 3 , mid_number: 6
lo: 0 , hi: 2
mid: 1 , mid_number: 8
lo: 2 , hi: 2
mid: 2 , mid_number: 6

Input:
{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}

Expected Output:
2
```

```
Actual Output:
2

Execution Time:
0.539 ms

Test Result:
PASSED



SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0
```

```
[(3, True, 0.224),
 (6, True, 0.581),
 (0, True, 0.313),
 (3, True, 47.893),
 (0, True, 0.13),
 (-1, True, 0.319),
 (-1, True, 0.002),
 (7, True, 0.684),
 (2, True, 0.539)]
```

In fact, once we have written out the algorithm, we may want to add a few more test cases:

1. The number lies in first half of the array.
2. The number lies in the second half of the array.

Here is the final code for the algorithm (without the `print` statements):

```python
def test_location(cards, query, mid):
    if cards[mid] == query:
        if mid-1 >= 0 and cards[mid-1] == query:
            return 'left'
        else:
            return 'found'
    elif cards[mid] < query:
        return 'left'
    else:
        return 'right'

def locate_card(cards, query):
    lo, hi = 0, len(cards) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        result = test_location(cards, query, mid)
        if result == 'found':
            return mid
        elif result == 'left':
            hi = mid - 1
        elif result == 'right':
            lo = mid + 1
    return -1
```

Try creating a few more test cases to test the algorithm more extensively.

Let's save our work before continuing.

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
[jovian] Updating notebook "aakashns/python-binary-search" on https://jovian.ai/
[jovian] Uploading notebook..
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/aakashns/python-binary-search
```

```
'https://jovian.ai/aakashns/python-binary-search'
```

## 9. Analyze the algorithm's complexity and identify inefficiencies, if any.

Once again, let's try to count the number of iterations in the algorithm. If we start out with an array of N elements, then each time the size of the array reduces to half for the next iteration, until we are left with just 1 element.

Initial length - `N`

Iteration 1 - `N/2`

Iteration 2 - `N/4` i.e. `N/2^2`

Iteration 3 - `N/8` i.e. `N/2^3`

...

Iteration k - `N/2^k`

Since the final length of the array is 1, we can find the

```
N/2^k = 1
```

Rearranging the terms, we get

```
N = 2^k
```

Taking the logarithm

```
k = log N
```

Where `log` refers to log to the base 2. Therefore, our algorithm has the time complexity **O(log N)**. This fact is often stated as: binary search *runs* in logarithmic time. You can verify that the space complexity of binary search is **O(1)**.

## Binary Search vs. Linear Search

```
def locate_card_linear(cards, query):
    position = 0
    while position < len(cards):
        if cards[position] == query:
            return position
        position += 1
    return -1
```

```
large_test = {
    'input': {
        'cards': list(range(10000000, 0, -1)),
        'query': 2
    },
    'output': 9999998

}
```

```
result, passed, runtime = evaluate_test_case(locate_card_linear, large_test, display=Fa

print("Result: {}\nPassed: {}\nExecution Time: {} ms".format(result, passed, runtime))
```

```
Result: 9999998
Passed: True
Execution Time: 1103.3 ms
```

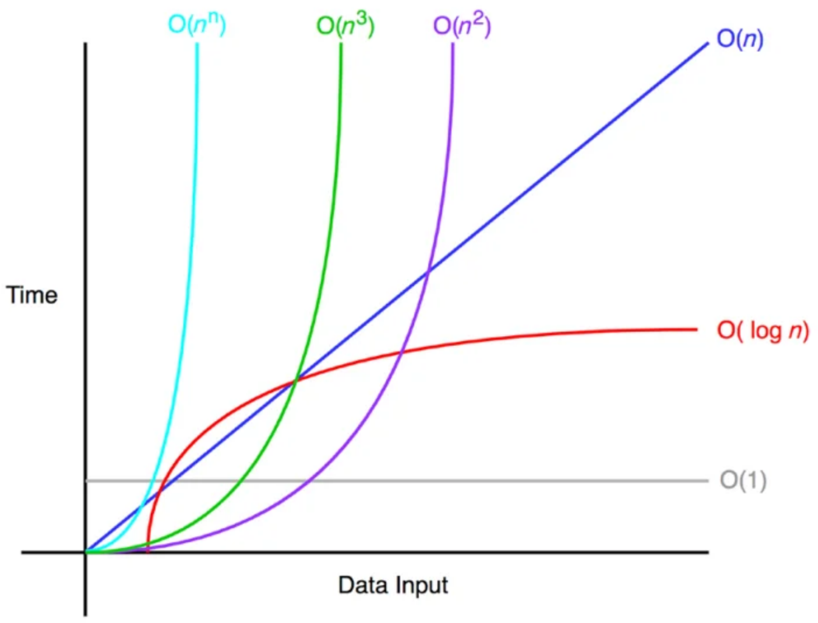
```
result, passed, runtime = evaluate_test_case(locate_card, large_test, display=False)

print("Result: {}\nPassed: {}\nExecution Time: {} ms".format(result, passed, runtime))
```

```
Result: 9999998
Passed: True
Execution Time: 0.019 ms
```

The binary search version is over 55,000 times faster than the linear search version.

Furthermore, as the size of the input grows larger, the difference only gets bigger. For a list 10 times, the size, linear search would run for 10 times longer, whereas binary search would only require 3 additional operations! (can you verify this?) That's the real difference between the complexities **O(N)** and **O(log N)**.

Another way to look at it is that binary search runs `c * N / log N` times faster than linear search, for some fixed constant `c`. Since `log N` grows very slowly compared to `N`, the difference gets larger with the size of the input. Here's a graph showing how the comparing common functions for running time of algorithms (source):



Do you see now why we ignore constants and lower order terms while expressing the complexity using the Big O notation?

## Generic Binary Search

Here is the general strategy behind binary search, which is applicable to a variety of problems:

1. Come up with a condition to determine whether the answer lies before, after or at a given position
2. Retrieve the midpoint and the middle element of the list.
3. If it is the answer, return the middle position as the answer.
4. If answer lies before it, repeat the search with the first half of the list
5. If the answer lies after it, repeat the search with the second half of the list.

Here is the generic algorithm for binary search, implemented in Python:

```python
def binary_search(lo, hi, condition):
    """TODO - add docs"""
    while lo <= hi:
        mid = (lo + hi) // 2
        result = condition(mid)
        if result == 'found':
            return mid
        elif result == 'left':
            hi = mid - 1
        else:
            lo = mid + 1
    return -1
```

The worst-case complexity or running time of binary search is **O(log N)**, provided the complexity of the condition used to determine whether the answer lies before, after or at a given position is **O(1)**.

Note that `binary_search` accepts a function `condition` as an argument. Python allows passing functions as arguments to other functions, unlike C++ and Java.

We can now rewrite the `locate_card` function more succinctly using the `binary_search` function.

```python
def locate_card(cards, query):

    def condition(mid):
        if cards[mid] == query:
            if mid > 0 and cards[mid-1] == query:
                return 'left'
            else:
                return 'found'
        elif cards[mid] < query:
            return 'left'
        else:
            return 'right'

    return binary_search(0, len(cards) - 1, condition)
```

Note here that we have defined a function within a function, another handy feature in Python. And the inner function can access the variables within the outer function.

```python
evaluate_test_cases(locate_card, tests)
```

```
TEST CASE #0

Input:
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 7}

Expected Output:
3


Actual Output:
3

Execution Time:
0.006 ms

Test Result:
PASSED


TEST CASE #1

Input:
{'cards': [13, 11, 10, 7, 4, 3, 1, 0], 'query': 1}
```

```
Expected Output:
6


Actual Output:
6

Execution Time:
0.006 ms

Test Result:
PASSED


TEST CASE #2

Input:
{'cards': [4, 2, 1, -1], 'query': 4}

Expected Output:
0


Actual Output:
0

Execution Time:
0.005 ms

Test Result:
PASSED


TEST CASE #3

Input:
{'cards': [3, -1, -9, -127], 'query': -127}

Expected Output:
3
```

```
Actual Output:
3

Execution Time:
0.008 ms

Test Result:
PASSED


TEST CASE #4

Input:
{'cards': [6], 'query': 6}

Expected Output:
0


Actual Output:
0

Execution Time:
0.003 ms

Test Result:
PASSED


TEST CASE #5

Input:
{'cards': [9, 7, 5, 2, -9], 'query': 4}

Expected Output:
-1


Actual Output:
-1

Execution Time:
0.004 ms
```

```
Test Result:
PASSED


TEST CASE #6

Input:
{'cards': [], 'query': 7}

Expected Output:
-1


Actual Output:
-1

Execution Time:
0.005 ms

Test Result:
PASSED


TEST CASE #7

Input:
{'cards': [8, 8, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 3}

Expected Output:
7


Actual Output:
7

Execution Time:
0.006 ms

Test Result:
PASSED
```

```
TEST CASE #8

Input:
{'cards': [8, 8, 6, 6, 6, 6, 6, 6, 3, 2, 2, 2, 0, 0, 0], 'query': 6}

Expected Output:
2


Actual Output:
2

Execution Time:
0.005 ms

Test Result:
PASSED


SUMMARY

TOTAL: 9, PASSED: 9, FAILED: 0
```

```
[(3, True, 0.006),
 (6, True, 0.006),
 (0, True, 0.005),
 (3, True, 0.008),
 (0, True, 0.003),
 (-1, True, 0.004),
 (-1, True, 0.005),
 (7, True, 0.006),
 (2, True, 0.005)]
```

The `binary_search` function can now be used to solve other problems too. It is a tested piece of logic.

> **Question**: Given an array of integers nums sorted in ascending order, find the starting and ending position of a given number.

This differs from the problem in only two significant ways:

1. The numbers are sorted in increasing order.
2. We are looking for both the increasing order and the decreasing order.

Here's the full code for solving the question, obtained by making minor modifications to our previous function:

```python
def first_position(nums, target):
    def condition(mid):
        if nums[mid] == target:
            if mid > 0 and nums[mid-1] == target:
                return 'left'
            return 'found'
        elif nums[mid] < target:
            return 'right'
        else:
            return 'left'
    return binary_search(0, len(nums)-1, condition)

def last_position(nums, target):
    def condition(mid):
        if nums[mid] == target:
            if mid < len(nums)-1 and nums[mid+1] == target:
                return 'right'
            return 'found'
        elif nums[mid] < target:
            return 'right'
        else:
            return 'left'
    return binary_search(0, len(nums)-1, condition)

def first_and_last_position(nums, target):
    return first_position(nums, target), last_position(nums, target)
```

We can test our solution by making a submission here: https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/

## The Method - Revisited

Here's a systematic strategy we've applied for solving the problem:

1. State the problem clearly. Identify the input & output formats.
2. Come up with some example inputs & outputs. Try to cover all edge cases.
3. Come up with a correct solution for the problem. State it in plain English.
4. Implement the solution and test it using example inputs. Fix bugs, if any.
5. Analyze the algorithm's complexity and identify inefficiencies, if any.
6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

Use this template for solving problems using this method: https://jovian.ai/aakashns/python-problem-solving-template

This seemingly obvious strategy will help you solve almost any programming problem you will face in an interview or coding assessment.

The objective of this course is to rewire your brain to think using this method, by applying it over and over to different types of problems. This is a course about thinking about problems systematically and turning those thoughts into code.

## Problems for Practice

Here are some resources to learn more and find problems to practice.

- Assignment on Binary Search: https://jovian.ai/aakashns/python-binary-search-assignment
- Binary Search Problems on LeetCode: https://leetcode.com/problems/binary-search/
- Binary Search Problems on GeeksForGeeks: https://www.geeksforgeeks.org/binary-search/
- Binary Search Problems on Codeforces: https://codeforces.com/problemset?tags=binary+search

Use this template for solving problems: https://jovian.ai/aakashns/python-problem-solving-template

Start a discussion on the forum: https://jovian.ai/forum/c/data-structures-and-algorithms-in-python/lesson-1-binary-search-linked-lists-and-complex/81

Try to solve at least 5-10 problems over the week to master binary search.

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
[jovian] Updating notebook "aakashns/python-binary-search" on https://jovian.ai/
[jovian] Uploading notebook..
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/aakashns/python-binary-search
```

```
'https://jovian.ai/aakashns/python-binary-search'
```