10 Algorithms To Solve Before your Python Coding Interview

In this article I present and share the solution for a number of basic algorithms that recurrently appear in FAANG interviews.

AnBento Follow
Jul 30 · 7 min read ★

Photo by Headway on Unsplash

Why Practicing Algorithms Is Key?

If you are relatively new to Python and plan to start interviewing for top companies (among which FAANG) listen to this: you need to start practicing algorithms right now.

Don't be naive like I was when I first started solving them. Despite I thought that cracking a couple of algorithms every now and then was fun, I never spent too much time to practice and even less time to implement a faster or more efficient solution. Between myself, I was thinking that at the end of the day solving algorithms all day long was a bit too nerdy, it didn't really have a practical use in the real daily work environment and it would not have brought much to my pocket in the longer term.

> "Knowing how to solve algorithms will give you a competitive advantage during the job search process"

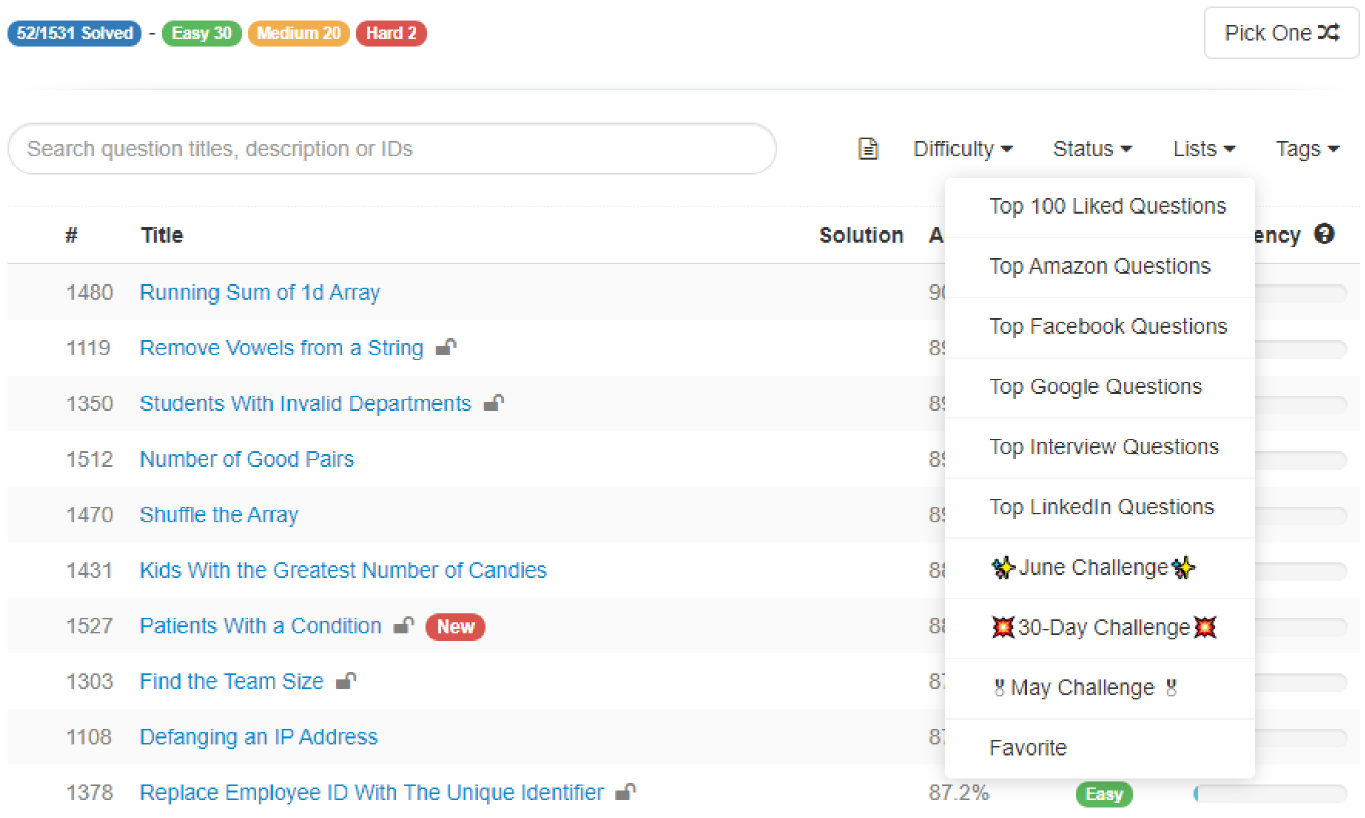
Well…I was wrong (at least partially): I still think that spending too much time on algorithms without focusing on other skills is not enough to make you land your dream job, but I understood that since complex problems present themselves in every day work as a programmer, big companies had to find a standardized process to gather insights on the candidate's problem solving and attention to detail skills. This means that knowing how to solve algorithms will give you a competitive advantage during the job search process as even less famous companies tend to adopt similar evaluation methods.

There Is An Entire World Out There

Pretty soon after I started solving algorithms more consistently, I found out that there are plenty of resources out there to practice, learn the most efficient strategies to solve them and get mentally ready for interviews (***HackerRank**, **LeetCode***, ***CodingBat*** and ***GeeksForGeeks*** are just few examples).

Together with practicing the top interview questions, these websites often group algorithms by company, embed active blogs where people share detailed summaries of their interview experience and sometimes even offer mock interview questions as part of premium plans.

For example, ***LeetCode*** let you filter top interview questions by specific companies and by frequency. You can also choose the level of difficulty (Easy, Medium and Hard) you feel comfortable with:



52/1531 Solved - Easy 30 Medium 20 Hard 2

Pick One

Search question titles, description or IDs

Difficulty Status Lists Tags

- Top 100 Liked Questions
- Top Amazon Questions
- Top Facebook Questions
- Top Google Questions
- Top Interview Questions
- Top LinkedIn Questions
- ✨June Challenge✨
- 💥30-Day Challenge💥
- 🎖 May Challenge 🎖
- Favorite

#	Title	Solution	A	ency
1480	Running Sum of 1d Array		9	
1119	Remove Vowels from a String		8	
1350	Students With Invalid Departments		8	
1512	Number of Good Pairs		8	
1470	Shuffle the Array		8	
1431	Kids With the Greatest Number of Candies		8	
1527	Patients With a Condition New		8	
1303	Find the Team Size		8	
1108	Defanging an IP Address		8	
1378	Replace Employee ID With The Unique Identifier		87.2% Easy	

Source: https://leetcode.com/problemset/all/

There are hundreds of different algorithmic problems out there, meaning that being able to recognize the common patterns and code an efficient solution in less then 10 mins will require a lot of time and dedication.

> "Don't be disappointed if you really struggle to solve them at first , this is completely normal"

Don't be disappointed if you really struggle to solve them at first, this is completely normal. Even more experienced Python programmers would find many algorithms challenging to solve in a short time without an adequate training.

Also don't be disappointed if your interview doesn't go as you expected and you just started solving algorithms. There are people that prepare for months solving a few problems every day and rehearse them regularly before they are able to nail an interview.

To help you in your training process, below I have selected 10 algorithms (mainly around **String Manipulation** and **Arrays**) that I have seen appearing again and again in phone coding interviews. The level of these problems is mainly ***easy*** so consider them as good starting point.

Please note that the solution I shared for each problem is just one of the many potential solutions that could be implemented and often a BF ("Brute Force") one. Therefore feel free to code your own version of the algorithm, trying to find the right balance between runtime and employed memory.

Strings Manipulation

1. Reverse Integer

```python
# Given an integer, return the integer with reversed digits.
# Note: The integer could be either positive or negative.

def solution(x):
    string = str(x)

    if string[0] == '-':
        return int('-'+string[:0:-1])
    else:
        return int(string[::-1])

print(solution(-231))
print(solution(345))
```

reverse_int.py hosted with ❤ by GitHub view raw

```
Output:
-132
543
```

A warm-up algorithm, that will help you practicing your slicing skills. In effect the only tricky bit is to make sure you are taking into account the case when the integer is negative. I have seen this problem presented in many different ways but it usually is the starting point for more complex requests.

2. Average Words Length

```
# For a given sentence, return the average word length.
# Note: Remember to remove punctuation first.

sentence1 = "Hi all, my name is Tom...I am originally from Australia."
sentence2 = "I need to work very hard to learn more about algorithms in Python!"

def solution(sentence):
    for p in "!?',;.":
        sentence = sentence.replace(p, '')
    words = sentence.split()
    return round(sum(len(word) for word in words)/len(words),2)

print(solution(sentence1))
print(solution(sentence2))
```

avg_words_length.py hosted with ❤ by GitHub view raw

```
Output:
4.2
4.08
```

Algorithms that require you to apply some simple calculations using strings are very common, therefore it is important to get familiar with methods like **`.replace()`** and **`.split()`** that in this case helped me removing the unwanted characters and create a list of words, the length of which can be easily measured and summed.

3. Add Strings

```
# Given two non-negative integers num1 and num2 represented as string, return the sum of
# You must not use any built-in BigInteger library or convert the inputs to integer dire

#Notes:
```

```python
#Both num1 and num2 contains only digits 0-9.
#Both num1 and num2 does not contain any leading zero.

num1 = '364'
num2 = '1836'

# Approach 1:
def solution(num1,num2):
    eval(num1) + eval(num2)
    return str(eval(num1) + eval(num2))

print(solution(num1,num2))

#Approach2
#Given a string of length one, the ord() function returns an integer representing the Ur
#when the argument is a unicode object, or the value of the byte when the argument is ar

def solution(num1, num2):
    n1, n2 = 0, 0
    m1, m2 = 10**(len(num1)-1), 10**(len(num2)-1)

    for i in num1:
        n1 += (ord(i) - ord("0")) * m1
        m1 = m1//10

    for i in num2:
        n2 += (ord(i) - ord("0")) * m2
        m2 = m2//10

    return str(n1 + n2)
print(solution(num1, num2))
```

add_strings.py hosted with ❤ by GitHub view raw

```
Output:
2200
2200
```

I find both approaches equally sharp: the first one for its brevity and the intuition of using the **`eval( )`** method to dynamically evaluate string-based inputs and the second one for the smart use of the **`ord( )`** function to re-build the two strings as actual numbers trough the Unicode code points of their characters. If I really had to chose in between the two, I would probably go for the second approach as it looks more complex at first but it often comes handy in solving "Medium" and "Hard" algorithms that require more advanced string manipulation and calculations.

4. First Unique Character

```python
# Given a string, find the first non-repeating character in it and return its index.
# If it doesn't exist, return -1. # Note: all the input strings are already lowercase.

#Approach 1
def solution(s):
    frequency = {}
    for i in s:
        if i not in frequency:
            frequency[i] = 1
        else:
            frequency[i] +=1
    for i in range(len(s)):
        if frequency[s[i]] == 1:
            return i
    return -1

print(solution('alphabet'))
print(solution('barbados'))
print(solution('crunchy'))

print('###')

#Approach 2
import collections

def solution(s):
    # build hash map : character and how often it appears
    count = collections.Counter(s) # <-- gives back a dictionary with words occurrence c
                                        #Counter({'l': 1, 'e': 3, 't': 1, 'c': 1, 'o':
    # find the index
    for idx, ch in enumerate(s):
        if count[ch] == 1:
            return idx
    return -1

print(solution('alphabet'))
print(solution('barbados'))
print(solution('crunchy'))
```

first_unique_char.py hosted with ❤ by GitHub view raw

```
Output:
1
2
1
###
1
2
1
```

Also in this case, two potential solutions are provided and I guess that, if you are pretty new to algorithms, the first approach looks a bit more familiar as it builds as simple counter starting from an empty dictionary.

However understanding the second approach will help you much more in the longer term and this is because in this algorithm I simply used **`collection.Counter(s)`** instead of building a chars counter myself and replaced **`range(len(s))`** with **`enumerate(s)`** , a function that can help you identify the index more elegantly.

5. Valid Palindrome

```python
# Given a non-empty string s, you may delete at most one character. Judge whether you ca
# The string will only contain lowercase characters a-z.

s = 'radkar'
def solution(s):
    for i in range(len(s)):
        t = s[:i] + s[i+1:]
        if t == t[::-1]: return True

    return s == s[::-1]

solution(s)
```

valid_palindrome.py hosted with ❤ by GitHub view raw

```
Output:
True
```

The *"Valid Palindrome"* problem is a real classic and you will probably find it repeatedly under many different flavors. In this case, the task is to check weather by removing at most one character, the string matches with its reversed counterpart. When ***s = 'radkar'*** the function returns **`True`** as by excluding the 'k' we obtain the word *'radar'* that is a palindrome.

Arrays

6. Monotonic Array

```python
# Given an array of integers, determine whether the array is monotonic or not.
A = [6, 5, 4, 4]
B = [1,1,1,3,3,4,3,2,4,2]
C = [1,1,2,3,7]

def solution(nums):
    return (all(nums[i] <= nums[i + 1] for i in range(len(nums) - 1)) or
            all(nums[i] >= nums[i + 1] for i in range(len(nums) - 1)))

print(solution(A))
print(solution(B))
print(solution(C))
```

monotonic_array.py hosted with ❤ by GitHub view raw

```
Output:
True
False
True
```

This is another very frequently asked problem and the solution provided above is pretty elegant as it can be written as a one-liner. *An array is monotonic if and only if it is monotone increasing, or monotone decreasing* and in order to assess it, the algorithm above takes advantage of the `all()` function that returns `True` if all items in an iterable are true, otherwise it returns `False`. If the iterable object is empty, the `all()` function also returns `True`.

7. Move Zeroes

```python
#Given an array nums, write a function to move all zeroes to the end of it while maintai
#the non-zero elements.

array1 = [0,1,0,3,12]
array2 = [1,7,0,0,8,0,10,12,0,4]

def solution(nums):
    for i in nums:
        if 0 in nums:
            nums.remove(0)
            nums.append(0)
    return nums

solution(array1)
solution(array2)
```

move_zeroes.py hosted with ❤ by GitHub view raw

```
Output:
[1, 3, 12, 0, 0]
[1, 7, 8, 10, 12, 4, 0, 0, 0, 0]
```

When you work with arrays, the `.remove()` and `.append()` methods are precious allies. In this problem I have used them to first remove each zero that belongs to the original array and then append it at the end to the same array.

8. Fill The Blanks

```python
# Given an array containing None values fill in the None values with most recent
# non None value in the array
array1 = [1,None,2,3,None,None,5,None]

def solution(array):
    valid = 0
    res = []
    for i in nums:
        if i is not None:
            res.append(i)
            valid = i
        else:
            res.append(valid)
    return res

solution(array1)
```

fill_the_blanks.py hosted with ❤ by GitHub view raw

```
Output:
[1, 1, 2, 3, 3, 3, 5, 5]
```

I was asked to solve this problem a couple of times in real interviews, both times the solution had to include edge cases (that I omitted here for simplicity). On paper, this an easy algorithm to build but you need to have clear in mind what you want to achieve with the for loop and if statement and be comfortable working with `None` values.

9. Matched & Mismatched Words

```
vo sentences, return an array that has the words that appear in one sentence and not
er and an array with the words in common.

l = 'We are really pleased to meet you in our city'
2 = 'The city was hit by a really heavy storm'

tion(sentence1, sentence2):
 = set(sentence1.split())
 = set(sentence2.split())

rn sorted(list(set1^set2)), sorted(list(set1&set2)) # ^ A.symmetric_difference(B), & A.intersec

lution(sentence1, sentence2))
```

match_words.py hosted with ❤ by GitHub view raw

```
Output:
(['The','We','a','are','by','heavy','hit','in','meet','our',
    'pleased','storm','to','was','you'],
 ['city', 'really'])
```

The problem is fairly intuitive but the algorithm takes advantage of a few very common set operations like `set()` , `intersection()` or `&` and `symmetric_difference()`or `^` that are extremely useful to make your solution more elegant. If it is the first time you encounter them, make sure to check this page.

10. Prime Numbers Array

```python
# Given k numbers which are less than n, return the set of prime number among them
# Note: The task is to write a program to print all Prime numbers in an Interval.
# Definition: A prime number is a natural number greater than 1 that has no positive div

n = 35
def solution(n):
    prime_nums = []
    for num in range(n):
        if num > 1: # all prime numbers are greater than 1
            for i in range(2, num):
                if (num % i) == 0: # if the modulus == 0 is means that the number can be
                    break
            else:
                prime_nums.append(num)
    return prime_nums
solution(n)
```

prime_numbers.py hosted with ❤ by GitHub view raw

```
Output:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

I wanted to close this section with another classic problem. A solution can be found pretty easily looping trough `range(n)` if you are familiar with both the prime numbers definition and the ***modulus operation***.

Conclusion

In this article I shared the solution of 10 Python algorithms that are frequently asked problems in coding interview rounds. If you are preparing an interview with a well-known tech Company this article is a good starting point to get familiar with common algorithmic patterns and then move to more complex questions. Also note that *the exercises presented in this post (together with their solutions) are slight reinterpretations of problems available on Leetcode and GeekForGeeks. I am far from being an expert in the field therefore the solutions I presented are just indicative ones.*

You may also like:

8 Popular SQL Window Functions Replicated In Python

A tutorial on how to take advantage of Pandas in business analytics to efficiently replicate the most used SQL Window…

towardsdatascience.com

15 Git Commands To Master Before Your Very First Project

The last Git tutorial you are going to need to master version control at the command line.

levelup.gitconnected.com

Airflow: How To Refresh Stock Data While You Sleep — Part 1

In this first tutorial on Apache Airflow, learn how to build a data pipeline to automatically extract, transform and…

towardsdatascience.com

Sign up for The Daily Pick

By Towards Data Science

Data Science Interview Python Algorithms Data Engineering