0x04. Mock Interview

Mock interviews are multi-purpose:

- It helps you and the staff understand where you stand in terms of general knowledge
- It helps you and the staff understand where you stand in terms of technical skills
- It is a training for interviews

Mock interviews - How-to

Pairing is cross campuses, please make sure you sync with your interviewer/interviewee before starting the mock interview in Slack.

Tools for online Mock interview:

- Video conference: meet.google.com or Slack
- Whiteboard: www.webwhiteboard.com or awwapp.com
- Coding: coderpad.io or docs.google.com

Whiteboarding

⧖ Duration: 1 hour

What you should tell the candidate:

You have to talk non-stop and explain what you are thinking about. Only once you find a way to solve the problem should you start to write the code on the whiteboard.

You can choose any language of your choice for each solution.

What you should look for / make sure is covered / asked

- Does the candidate ask for clarification?
- If the candidate do not ask questions before trying to solve or code the problem, it's a bad sign
- Data structure: Does the candidate use a correct data structure to represent the data / return values / parameters?
- Does the code work for all possible edge cases? (empty data structure, etc...)
- Code:
- Good practices (commented, name of variables make sense, etc...)
- No errors in the code
- Can code fast

All in

You are playing to a new kind of board game: "Count me in".

The concept is easy:

- You receive multiple cards
- Each card contains a number
- You have to announce how many number (= a card) you have and have also the `number + 1` in your hand
- Each duplicate number/card are counted separately

The winner is the one with the biggest number of number/card having their "follower" in the hand.

(yes it's a random game, but perfect illustration of the algorithm)

If your hand is represented by a `list` and each number/card by `i`: Write an algorithm that returns the number of `i` with `i + 1` also in `list`.

Example 1:

```
list: [1,2,3]
result: 2
-> 1 has 2 in the list and 2 has 3 in the list
```

Example 2:

```
list: [1,3,2,3,5,0]
result: 3
-> 0 has 1 in the list; 1 has 2 in the list and 2 has 3 in the list
```

Example 3:

```
list: [0, 2, 6, 4]
result: 0
```

Example 4:

```
list: [2, 3, 2]
result: 2
-> 2 is present 2 times and both has 3 in list
```

What you don't tell the candidate, but answer her/him when she/he asks for clarification:

- All integers of `list` are positive
- `list` has only one element
- `list` length and integers are not bigger than 1000

Evaluation: 100%

Comment:

Extra instructions - hide

Search in Array

Iterate through each element, search in the list how many times (element + 1) is present.

Code implementation in Python:

```
def countIn(list: List[int]):
    nb = 0
    for i in list:
        if (i + 1) in list:
            nb += 1

        return nb
```

Follow-up questions:

- What is the time complexity of this algorithm? `O(n^2)` (`in` in Python has a complexity of `O(n)`)
- What is the space complexity of this algorithm? `O(1)`
- Can you find a faster solution?

Score: If this is the best solution the candidate can find: max 40%

Search in Set

Same as "Search in Array" but instead of searching in the list, searching on a Set copy of the list (find an element in a Set has complexity `O(1)`)

Code implementation in Python:

```
def countIn(list: List[int]):
    nb = 0
    set_list = set(list)

        for i in list:
        if (i + 1) in set_list:
            nb += 1

        return nb
```

Follow-up questions:

- What is the time complexity of this algorithm? `O(n)`
- What is the space complexity of this algorithm? `O(n)`
- Can you find a faster solution?

Score: If this is the best solution the candidate can find: max 60%

Search in a sorted list

Another way of changing the data storage to allow for more efficient searching is to sort it. Sorting has a time complexity of `O(n log(n))` , and searching for integers in a sorted array, using binary search, has a cost of `O(log(n))` .

This will give us a total time complexity of `O(n log(n))` .

However, we don't actually need to use binary search! If we iterate over the sorted `list` , then we know that if `i + 1` exists, it will be after all the copies of `i` .

Each copy of `i` should be counted if at least one copy of `i + 1` exists. Therefore, we can iterate down the sorted `list` , keeping track of how many times the current `i` has appeared. When we get to a different integer, we can check if it's `i + 1` , and if it is, then the number of `i` we saw should be added to count.

Code implementation in Python:

```
def countIn(list: List[int]):
    nb = 0
    list.sort()

        run_length = 0
    for i in range(len(list)):
        if list[i - 1] != list[i]:
            if list[i - 1] + 1 == list[i]:
                nb += run_length
            run_length = 0
        run_length += 1

        return nb
```

Follow-up questions:

- What is the time complexity of this algorithm? `O(n log(n))`
- What is the space complexity of this algorithm? `O(1)` (assuming `list.sort()` is optimized)

Score: If this is the best solution the candidate can find: max 100%

Skip | Submit