Programming Interview Prep 101

Institute for Computational and Mathematical Engineering
Stanford University
Andreas Santucci
Fall 2020

Contents

1 Components of a Programming Interview

We assume you're preparing for a software engineering or data scientist job interview. There are many key differences, but we can also focus on common elements to help you prepare for both. Let's start by highlighting some of the aspects that you might cover in a few different roles (from Workera):

	ML Engineer	Data Scientist	Software Engineer
ML	✓	✓	X
Data Science	✓	✓	X
Mathematics	✓	✓	X
Algorithmic Coding	✓	✓	✓
Software Engineering	✓	X	✓

How is ML different from Data Science? Data science includes things like statistical hypothesis testing and AB testing that aren't considered cornerstones in machine learning. On the other hand, machine learning focuses more heavily on modeling architectures that allow for implicit interactions between features; think examples like gradient boosted decision trees or even (deep) neural networks.

What's the Point of an Interview? Don't be fooled into thinking that the point of an interview is to get the correct answer to every question; it's not! The purpose of an interview is to demonstrate to the interviewer that you are somebody they would want to work with. To be clear, analytical thinking ability is one component of that, but it doesn't complete the picture. Going into each interview, you should understand what the sub-goal is:

- Are you chatting with a **recruiter**? If so, they just want to hear about your (educational and work) background as well as your enthusiasm for working with the company and prospective team.
- Are you interviewing with a **prospective peer**? If so, they're going to be more interested in your analytical problem solving abilities and technical skills.
- Is your personality being probed in a **cross-functional interview** (e.g. interviewing with a product manager)? They may focus more on behavior questions.
- If you're interviewing with a **(hiring) manager**, be prepared to discuss what your motivations are for wanting to work with the company and team in further detail.

Asking Questions During an Interview Yes, it's important to ask good questions at the end of an interview, if you have time. But that's not what I want to talk about in this section. Here, I want to emphasize the importance of asking clarification questions *during* the technical interview itself. Sometimes, the interviewer intentionally underspecifies parts of the question in order to probe your understanding of the problem. Some classic examples:

- Whenever you're asked a programming question, always take a step back and *ask* if you can use standard library tools to solve it. It's *not* often the case that the interviewer wants to see you implement something from scratch (if they do, they'll specify this in the question).
- "Suppose I give you a *random* number...". You have to ask: a random number drawn from what distribution? Is it a discrete random variable, or continuous? What values can the random variable take on? With what probabilities?

- "Suppose you are tasked with the problem of figuring out whether to launch product A or product B for the company, how would you go about this?". Here, you can start to figure out what the interviewer wants with follow-up questions: e.g. are you allowed to run an A/B test? do you have access to historical data? are there internal reasons for preferring one product over another?

Asking questions isn't a sign of weakness: it's a sign that you understand ambiguities and nuances of the question being posed.

What do I do if I get stuck? If your interviewer is really adept, they should be assessing the boundaries of your knowledge. It is therefore natural for them to gradually guide you toward more and more difficult questions until you start to struggle. This is *expected* and a *natural* part of the interview process where a candidate's level of proficiency is determined. When you notice that you are stuck on a problem and you are doubting yourself, do the following two things (in order):

1. Take a deep breath and tell the interviewer, out loud, that you are genuinely challenged by the fascinating question they posed; buy yourself a few more moments to see if you can recollect your thoughts and try a new approach.
2. If the above fails, tell the interviewer that you would request a *gentle hint* for the problem. This is not a sign of failure: most problems you will solve in industry and academia are too challenging for you to solve on your own, so knowing when to ask for explicit help can be construed positively.

1.1 Behavioral Questions

Don't overlook these! They are easy to prepare for and can also make or break an interview: it's not enough to simply be an excellent coder, you need to show that you can work with others effectively. Example questions include:

- Tell me about the most challenging project you've worked on (or one that you are proud of).
- Tell me about a time you ran into difficulty working with team-members or a manager, and how did you manage the situation?
- Who do you look up to, and what are you doing each day to be more like them?
- Why do you want to work *here* in particular?

You should have canned answers for all of these questions and be prepared to deliver a miniature speech with enthusiasm. You should have a chance to focus on these in other aspects of Xtend Interview Prep, so we won't spend much time on them here; just understand that they are a critical aspect of the quantitative interview process.

1.2 What tech-stack do I need to know?

Most reasonable companies will let you interview with whatever language you are most comfortable with. If you get the job, they will trust that you can learn any tool or language that they require of you, if they so happen to have a strong preference.

2 Programming Questions

They're not going to be that difficult, you just need to make sure you get them *right*! If you have a 45 minute technical interview focused *exclusively* on programming, you should anticipate being asked one warmup question that should be solvable in a few minutes time and then a more meaty question that, once solved as initially posed, has options for follow-up questions to further gauge the quality of the candidate being interviewed.

2.1 Is a string a palindrome?

A palindrome is a string that reads the same backwards as it does forwards. There are efficient algorithms for checking whether a string is a palindrome, but you can also do it at a higher level using built-in string slicing.

```
def isPalindrome(str):
  str == str[::-1]
```

Follow-up: Implement from scratch OK, now your interviewer wonders if you can implement the algorithm yourself as a follow up. We can mimic the first solution we crafted above: simply create a reversed string and then check if it equals the input string.

```
def isPalindrome(str):
  reversed = ""
  for char in str:
    reversed = char + reversed
  return str == reversed
```

Computational Complexity Your interviewer asks you what is the storage requirement of your approach. Realize that for an input of m characters, you have to create a temporary (reversed) string also m characters, whence your storage requirement is $O(m)$. What about the time complexity of your program? Realize that we are iterating over the input character sequence, and so that costs $\mathcal{O}(m)$ work right there. What about the work done within the for-loop, how much does it cost to "add" two strings together? Since strings are immutable, what we're effectively doing is creating a new temporary string that contains characters copied from both input arguments to the addition operator, i.e. if we have two strings of length n_1 and n_2 then adding these strings together requires $\mathcal{O}(n_1 + n_2)$ operations of work. If our case, our expression is of the form `single_char + reversed_candidate` where reversed candidate is growing by one each iteration of the loop. So, on the first iteration the string-addition requires 1 operation, on the second iteration it requires 2 operations, and in general on the ith iteration it requires i operations to create the temporary copy. The total work across all for-loop iterations is $1 + 2 + 3 + ... + m = \frac{m(m+1)}{2} = \mathcal{O}(m^2)$. Unfortunately, this means that we have an $\mathcal{O}(m^2)$ algorithm. Is there a way to improve?

Follow-up: Improve the Complexity of your Algorithm Your interviewer asks you whether it's really necessary to reverse the entire string. Isn't there some other way to check if a string is a palindrome without creating an entire copy? This is straightforward: we simply iterate over the first 1/2 of the string and check character by character if each is equal to the corresponding character indexed from the end of the string.

```
def isPalindrome(str):
    for i in range(0, int(len(str)/2)):
        if str[i] != str[len(str)-i-1]:
            return False
    return True
```

Follow-up: computational complexity Now your interviewer wants to see if you understand computational complexity, so they ask you to analyze your algorithm. Realize that if our input string is m characters, then we iterate over $\lfloor m/2 \rfloor$ characters in the `for`-loop. Each iteration, we perform a constant-time comparison of two characters. In the worst case, our input is a palindrome and we don't `break` out of the loop early, whence the algorithm is $\mathcal{O}(m)$.

2.2 Find the missing number

Suppose you're given a listing of integers $\{1, 2, \ldots, n\}$ and you are told that exactly one integer is missing. The input numbers can appear in any order, and your job is to find the missing number. How can you solve this problem?

Brute Force We should always try to rattle off the simplest solution to the problem, if even only verbally, before considering how to solve the problem efficiently.

```
def findMissing(input, N):
  for i in range(1, N+1):
    if i not in input:
      return i
  return None
```

The problem with this approach is that if our `input` is simply a list of numbers, then the `in` operator requires linear time to scan through the input and search for a match. Since we end up nesting this $\mathcal{O}(n)$ operation inside of a for-loop over n elements, our algorithm is quadratic in runtime.

Using Counting and Algebra Recall that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. Here, we see how to develop a linear time algorithm: simply calculate the sum of the input values and compare it with $\sum_{i=1}^{n} i$: the difference between the two is exactly equal to the missing number. I.e.

```
def findMissing(input, N):
  complete_ttl = N*(N+1)/2
  ttl = 0
  for i in input:
    ttl += i
  return complete_ttl - ttl
```

This algorithm is $\mathcal{O}(n)$ since we have an arithmetic expression which requires three floating operations, and then we have a loop which requires $\mathcal{O}(n)$ iterations where each iteration requires one floating point operation (a single addition).

Follow up question: Overflow Your interviewer now mentions that some of the input values can be very large, and wonders if there are any drawbacks to the previous algorithm? If the inputs are so large, then our previous algorithm might fail when calculating either the complete or partial totals. I.e. it might be possible to represent *each* of the integers exactly, but their *sum* may be too large to store using an integer representation (note this isn't a problem in modern Python, but entertain the question).

```
def findMissing(input, N):
  found = set()
  for x in input:
    found.add(x)
  for i in range(1, N+1):
```

```
    if i not in found:
      return i
  return None
```

If we're willing to afford $\mathcal{O}(n)$ storage, we could easily create a hashmap that counts the number of occurences of each value in the input using one linear traversal, then iterate over values $\{1, 2, \ldots, n\}$ and check if each is in the set in constant time (requiring a total of $\mathcal{O}(n)$ additional operations). Therefore, we could get an algorithm that's linear in space and time without being susceptible to overflow.

2.3 Checking if a Program has Balanced Parentheses

You are given a computer program, and asked to check if it has a valid use of parentheses: i.e. each opening parenthesis is matched by a corresponding closing parenthesis. There can be square brackets, curly braces, or regular parentheses in the program.

```
open_list = ["[","{","("]
close_list = ["]","}",")"]

# Function to check parentheses
def isBalanced(program):
    stack = []
    for char in program:
        if char in open_list:
            stack.append(char)
        elif i in close_list:
            pos = close_list.index(char)
            if ((len(stack) > 0) and
                (open_list[pos] == stack[len(stack)-1])):
                stack.pop()
            else:
                return False
    if len(stack) == 0:
        return True
    else:
        return False
```

Complexity analysis Realize that if our input `program` has n characters, then we need to perform scan each character in order to validate the correctness of the program, i.e. our algorithm must have work lower bounded by $\Omega(n)$. In our particular implementation, within our `for`-loop, we are performing if-statements involving an `in` method-call, an append operation, a look-up via an index and a pop. The question is are these all constant time operations? The answer is yes! In particular, because we are using a list, `append` is $\mathcal{O}(1)$. Further, calling `in` on a list of size 3 and calling `close_list.index(char)` costs at most 3 operations (the number of different kinds of parentheses we can have), but critically this is independent of the input size of our computer program! Therefore the entire algorithm is $\mathcal{O}(n)$. The storage requirement is $\mathcal{O}(n)$ since in the worst case we could have a program with all open parentheses (or even a correct program with $n/2$ open parenthesis, which is still $\mathcal{O}(n)$).

2.4 Buying and Selling Stocks

Suppose you are given a input which describes a sequence of stock prices. Write an algorithm which outputs the maximum profit attainable. You may assume there are at least two prices in the input.

Brute force solution As always, we at least call out a simple brute force solution. It's worth verbally describing this to the interviewer briefly before moving on to a more efficient implementation, as in this way you will have secured partial credit for the problem.

```
def maxProfit(prices):
  max_profit = 0
  for i in range(len(prices)-1):
    for j in range(i+1, len(prices)):
      max_profit = max(prices[j] - prices[i], max_profit)
  return max_profit
```

This is a quadratic run-time solution: the outer-most loop executes $\mathcal{O}(n)$ times, the inner-most loop executes $(n-1)+(n-2)+\ldots+1 = \mathcal{O}(n^2)$ times, and finally invoking the `max` operation requires a constant time comparison between the two inputs. Therefore, the algorithm is $\mathcal{O}(n^2)$. We only require $\mathcal{O}(1)$ storage to represent the `max_profit` variable.

Efficient Solution We can do better than a quadratic time solution, however. We can take several passes over the input and achieve a linear time algorithm as follows.

```
def maxProfit(prices):
  running_min = [prices[0]]
  for i in range(1, len(prices)):
    running_min.append(min(running_min[i-1], prices[i]))
  max_profit_sell_on_day_i = []
  for i in range(len(prices)):
    max_profit_sell_on_day_i.append(prices[i] - running_min[i])
  max_profit = 0
  for val in max_profit_sell_on_day_i:
    max_profit = max(max_profit, val)
  return max_profit
```

Complexity Analysis There are three linear traversals that we make through our data. In the first linear traversal, we accumulate a running minimum. Realize that the append operation on a list is $\mathcal{O}(1)$ and taking a min is another constant time operation, and so we perform constant work within each iteration of the first `for`-loop. So, the first `for`-loop costs $\mathcal{O}(n)$ work. How much does it cost to compute the max profit if we sell on day i, for each of the days? Here, it's a very similar analysis, but instead of a `min` operation we have a subtraction; the resulting `for`-loop costs $\mathcal{O}(n)$ work as well. Lastly, calculating the `max` over an input costs $\mathcal{O}(n)$ work.[1]

3 SQL

Whether you're a software engineer or a data scientist, knowing how to query from large databases is an important skill to have.

[1] We remark that if we were lazy and sorted our `max_profit` list in order to find the global maximum profit, this would increase the complexity of the algorithm to $\mathcal{O}(n \log n)$.

3.1 Basic SQL

The basic structure of a SQL query is simple

```
SELECT
  ...
FROM
  ...
WHERE
  ...
GROUP BY
  ...
;
```

There are a few things to realize when you're learning how to program with SQL. Because SQL is a declarative language, it means that it isn't parsed from top-to-bottom like an imperative and procedural programming language.[2] Instead, all we know about a SQL program is that by the time it finishes executing, all of the statements will have been satisfied.

3.2 Selecting columns

Example Data Let's work with the following dataset. Suppose it's in our database with the name `sales`.

```
 salesperson_id |    name    |   city   | commission
----------------+------------+----------+------------
           5001 | James Hoog | New York |       0.15
           5002 | Nail Knite | Paris    |       0.13
           5005 | Pit Alex   | London   |       0.11
           5006 | Mc Lyon    | Paris    |       0.14
           5007 | Paul Adam  | Rome     |       0.13
           5003 | Lauson Hen | San Jose |       0.12
```

We can really easily select all of the columns from a dataset using the `*` wildcard operator. E.g.

```
SELECT * FROM sales;
```

If we want to select individual columns, we can simply list them out as part of the `SELECT` clause:

```
SELECT
  salesperson_id
  , name
FROM
  sales;
```

This selects the first two columns from the dataset. What if we want to rename a column? We can use an `AS` modifier within the `SELECT` clause:

[2] Python supports imperative, procedural programming for example, in which you lay out a sequence of instructions that get executed in the order that they appear in the program.

```
SELECT
  salesperson_id AS id
  , name
FROM
  sales;
```

Here, we chose to rename the first column as `salesperson` but left the second column with its original name.

3.3 Aggregating data

Let's now consider the following dataset, called `sales` again:

```
ord_no      purch_amt   ord_date    customer_id  salesperson_id
----------  ----------  ----------  -----------  -----------
70001       150.5       2012-10-05  3005         5002
70009       270.65      2012-09-10  3001         5005
70002       65.26       2012-10-05  3002         5001
...
```

Counting Observations How many rows are in our dataset?

```
SELECT
  COUNT(*)
FROM sales;
```

Equivalently, we could write:

```
SELECT
  COUNT(1)
FROM sales;
```

Boolean Counting How about simple Boolean counting? E.g. count the number of instances where the purchase amount was greater than $600?

```
SELECT
  COUNTIF(purch_amt > 600)
FROM sales;
```

Summation Example What's the total purchase price in the entire dataset?

```
SELECT
  SUM(purch_amt) AS ttl_purchase_price
FROM
  sales;
```

3.4 Group By Clause

What if we wanted to find the total purchase price for each salesperson? Remember the `GROUP BY` clause in the beginning when we showed off the SQL syntax structure? Let's use it!

```
SELECT
  salesperson_id
  , SUM(purch_amt) AS ttl_amt
FROM
  sales
GROUP BY
  salesperson_id;
```

If we wanted to be lazy, we could use syntactic sugar in the `GROUP BY` statement to refer to the first column in the select clause:

```
SELECT
  salesperson_id
  , SUM(purch_amt) AS ttl_amt
FROM
  sales
GROUP BY 1;
```

3.5 Filtering

Write a query to extract all observations related to the marketing department, supposing the table name is `department_info`.

```
+---------------+----------------------+------------+-------------+
| DEPARTMENT_ID | DEPARTMENT_NAME      | MANAGER_ID | LOCATION_ID |
+---------------+----------------------+------------+-------------+
|            10 | Administration       |        200 |        1700 |
|            20 | Marketing            |        201 |        1800 |
|            30 | Purchasing           |        114 |        1700 |
|            40 | Human Resources      |        203 |        2400 |
|            50 | Shipping             |        121 |        1500 |
|            60 | IT                   |        103 |        1400 |
|            70 | Public Relations     |        204 |        2700 |
-----------------------------------------------------------------
```

```
SELECT
  *
FROM
  department_info
WHERE
  DEPARTMENT_NAME = "Marketing";
```

3.6 Joins

Find the name of all reviewers who gave a NULL rating.

Table: reviewer.

```
 rev_id |            rev_name
--------+--------------------------------
   9001 | Righty Sock
   9002 | Jack Malvern
   9003 | Flagrant Baronessa
   9004 | Alec Shaw
   9005 |
   9006 | Victor Woeltjen
   9007 | Simon Wright
   9008 | Neal Wruck
   9009 | Paul Monks
   9010 | Mike Salvati
```

Sample table: rating.

```
 mov_id | rev_id | rev_stars | num_o_ratings
--------+--------+-----------+---------------
    901 |   9001 |      8.40 |        263575
    902 |   9002 |      7.90 |         20207
    903 |   9003 |      8.30 |        202778
    906 |   9005 |      8.20 |        484746
    924 |   9006 |      7.30 |
    908 |   9007 |      8.60 |        779489
    909 |   9008 |           |        227235
```

```
SELECT
  rev_name
FROM
  reviewer
  JOIN rating USING(rev_id)
WHERE
  rev_stars IS NULL;
```

4 Statistics and Machine Learning

These topics are becoming increasingly important these days. It's OK if you're not planning to go into an ML related field, but if you are these questions might be useful to consider?

- Explain the basic assumptions behind linear regression.
- Derive the loss function for logistic regression.
- Explain a few differences between logistic regression and Support Vector Machines when used in the context of classification?
- What is overfitting? How can you identify when you are overfitting, and what can you do to fix it?
- Can you identify when a model is underfitting? What can you do about this?

5 Resources

Workera This is a really cool site that lets you practice subcomponents of technical interviews. It was put together by Kian who is also an alumni of Stanford and an adjunct lecturer in the CS department. Check it out: workera

LeetCode If you just want to practice a bunch of problems, websites such as Leetcode are a great resource. Other mentions include codeforces, codechef, and topcoder. Focus on *easy* problems and getting correct and efficient solutions. In CME 212 we'll learn how to solve *medium* problems, and *hard* problems are simply a combination of 2-3 medium problems.

References

[1] w3resource.com Accessed Fall 2020

[2] five-essential-phone-screen-questions Accessed Fall 2020