NAIL A CODING INTERVIEW

Six-Step Mental Framework

By Grace Huang

Copyright © 2022-2023 Grace Huang

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the publisher's prior written permission.

The information in this book is provided for educational and informational purposes only. It is not intended as a substitute for professional advice. The author and publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Trademarks: All terms mentioned in this book known as trademarks or service marks have been appropriately capitalized. The author cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

First Edition: April 2022

Second Edition: September 2023

To the engineers whose code can make computers dance yet find themselves frozen at the interviews,

This book is dedicated to you.

INTRODUCTION

By now, having practiced hundreds of coding questions on HackerRank and LeetCode for weeks, you must feel quite confident about the upcoming interview.

After the interviewer gives the introduction and finally reveals the coding problem, a sense of nervousness starts to creep in within you. While you are internalizing the coding problem, for a split second, your mind gets lost. Now you don't know where to start.

Suddenly you also notice the 45-minute clock has been ticking, so your mind goes blank again. A well-prepared coding interview is off to a bad start.

I have been there, and I know that feeling.

I have interviewed hundreds of software engineer candidates while working for big tech companies and also hiring for my own company Roxy. I have interviewed new grads, industry veterans from Google, Amazon & Facebook, quantitative engineers from Hedge Funds, and indie entrepreneurs.

After meeting each of the stellar interview candidates, I often pondered on what makes them shine in the interviews. Their coding styles were very different. They preferred different languages. However, they had something in common — they seemed to follow a similar mental framework to solve problems within a limited amount of time.

From the interviews that didn't result in job offers, I also noticed some common themes or attributes.

After analyzing hundreds of interviews, I noticed that the same six steps, forming a mental framework, were often at work.

- Problem: Write down the problem statement.
- Clarifying: Clarify the problem space.

- Cases: Document the test cases.
- Algorithm: Describe and document the algorithm, including complexity analysis.
- Code: Start coding
- Test: Test the code with the test cases and fix any bugs

This mental framework can be applied to almost every coding problem.

This brought me to write this book.

End Goal

By the end of the book, the goal is for you to be familiar with the six-step mental framework and be able to apply it to any coding problem in an interview.

Structure

The book has two major parts: Introduction to the Six-Step Mental Framework and Examples.

Part 1

The first part of the book provides this framework and a set of specific, actionable techniques for each step.

In every step, I will utilize an example coding interview question—"convert a number to English words"—to illustrate how to apply this framework during a coding interview.

Part 2

The second part of the book offers two examples to illustrate how to use the Six-Step Mental Framework to solve coding problems in real-life interviews.

The examples are as follows:

- Code Problem 1: Arrays: Move Zeros to the Left (asked by Meta/ Facebook)
- Code Problem 2: Longest Path In The Matrix (asked by Alphabet/ Google)
- Code Problem 3: Find All the Possible Subsets (asked by Netflix)

Assumptions

This book assumes that you have a foundational knowledge of programming, a grasp of computer science fundamentals (including data structures, algorithms, and object-oriented programming), and prior coding experience.

However, it doesn't assume that you have necessarily undergone interviews conducted by hiring companies.

Programming Languages

This book uses JavaScript to solve coding problems, solely for the purpose of demonstrating the algorithms.

During actual interviews, you should feel free to use any programming language you are comfortable with.

Suggestions

Please don't hesitate to provide feedback if anything is unclear or if you spot any typos in this book.

You can contact me through any of the following methods:

Email: higracehuang@gmail.com

Twitter: https://twitter.com/imgracehuang

LinkedIn: https://www.linkedin.com/in/lghuang/

Ready? Let’s dive right in!

STEP 1: WRITE DOWN THE PROBLEM

Estimated time in interview: 1 minute

"I write to discover what I know."

— Flannery O'Connor

Why?

Some interviewers just verbally describe the question, while some paste the question on the whiteboard. In case they do not, you need to write it down yourself.

Why?

- Firstly, it helps you internalize what that problem is.
- Secondly, you can use it to “force” the interviewer to confirm your understanding.

This is the power of writing in communication!

If you are in a phone interview or via Zoom or Google Hangout, you can type down the whole question on the online shared editor (such as CollabEdit, or HackerRank). If you are doing it on the whiteboard, write it down on the board.

If the interviewer provides an example along with the problem, do write down the examples as well. Every little detail may be important to solve the problem.

In case what you’ve written down is different from what the interviewer wants you to do, they should let you know, before your solution deviates too much from the problem.

The last thing you want is when you realize you and the interviewer do not share the same understanding, 30 minutes have passed.

For whatever reason (you did not hear correctly or your interviewer's accent), it will eventually punish you by taking away your precious time for showcasing your strength.

Example: The English Number Conversion problem

Here is what you hear from the interviewer, and you can write down or type:

> Write a function to convert a given number to English words.
>
> For example, the input is 123, and the output is “one hundred twenty three”.

Then you can pose a question to the interviewer: "Is my understanding correct?"

Takeaways

- Listen attentively.
- Write down the interview coding question and examples.
- Confirm the question with the interviewer.

STEP 2: CLARIFY THE PROBLEM SPACE

Estimated time in interview: 5 minutes

"The most misleading assumptions are the ones you don't even know you're making."

— Douglas Adams

It is incredibly rare for interviewers to present you with a fully defined problem.

Correct, you read it right. The majority of problems they provide are intentionally incomplete and vague. Some might lack definitions for certain edge cases. The scope of the problem may be much bigger than it sounds.

The problem statement is often the bait in a trap. They require your curious mind to ask questions to complete them.

Interviewers want to assess:

- Whether you make assumptions easily
- Whether you have the technical intuition to ask good questions
- Whether you make good judgment

Assumptions

Avoid making assumptions. An assumption is "a thing that is accepted as true or as certain to happen, without proof". It is a common pitfall in any technical interview, and also the hardest thing to notice.

This tendency isn't your fault. As Yale neurobiology Professor Frank Han explained, the brain's vast neural network requires huge amounts of energy to keep it running[1]:

> "There are over one hundred billion cells in our brain and each of them makes over ten thousand connections with other brain cells. While the large number of possible combinations of cell connections allows for higher-ordered thinking, this is a big problem evolutionarily in terms of energy cost…Therefore, the brain has to encode things efficiently to save energy."

One way our brain saves energy is through making assumptions, i.e., drawing on our past experiences to find patterns in how the world works. In front of new situations, we apply these patterns—or assumptions—to them.

The problem with assumptions is that we apply our own past experiences to understand the situation. To avoid assumptions, we need to ask questions to bring you and the interviewer to the same page.

Clarifying Questions

Ask good questions. There is a difference between good questions and not-so-good questions in clarification.

Good clarifying questions themselves have information and directions. They show the interviewer that you are a careful thinker who considers a problem from all angles. This kind of question will make you take the lead in the interview.

Not-so-good questions are dry without much direction. They show the interviewer that you are still relying on the interviewer to drive.

---

[1] Frank Han, How the Brain Saves Energy: The Neural Thermostat, https://www.yalescientific.org/2010/09/how-the-brain-saves-energy-the-neural-thermostat/

Not-so-good Questions	Good Questions
“Do you have any other requirements?”	“Do we want to return exceptions when the input is null?”
	“How about the input is a negative number?”
	“Do we prefer speed or optimize for space?”

So, what kinds of questions can we ask?

We can ask millions of questions, but they may not be useful in solving the problem. Here are some general areas you can ask with some examples:

- Edge cases
    - Does the method accept null?
- The scale of the problem
    - How big can the integer be?
    - What if we have to calculate 1 million (or even larger) of the records?
- Implementation Limitations
    - Can we use the pop and push methods of the array?
    - What APIs are available for this problem?
- Implementation Objectives
    - Do you optimize for time or space?
    - What is our expectation in terms of time and space complexities?
- Usage of the solution
    - What is the purpose of the method?

- Do we want to turn it into a library?

The answers to the questions can provide you with a clearer understanding of the type of problem you are solving.

Good Judgment

Make good judgment. Good questions pave the way for good judgment. Interviewers often follow up with a question after yours, “What do you think it should return?”, or “Yeah, that's a good call. But I want to hear what you think?”

An experienced engineer may have faced this kind of question many times at work, and they often know what is the best way to deal with edge cases. They may consider maintainability, scalability, readability, and ease of debugging before making a decision.

How you made a judgment also shows your decision-making process. Do you make a decision with your gut feeling? Or do you consider all the possible options and compare the trade-offs before making a decision? Obviously, the latter shows maturity.

Why do all these matters in a coding interview? They mirror the attributes that your colleagues, including engineers, designers, and product managers, expect out of you in day-to-day work. They expect you to challenge their assumptions by asking good questions, and always making reasonable judgments when facing ambiguity.

Unfortunately, 5 out of 10 candidates that I have met do not clarify problems and jump to code immediately. The final solution will definitely be affected. Even though the candidate later realizes the missing cases and tries to patch them into the logic, the code may not look pretty or clean by then.

Example: the English Number Conversion problem

Let's continue with the same example question: "Convert a number to English words".

Here is a list of questions that a candidate can clarify:

What is the range of the input number? 0? 1–999? 1–999,999?
Can the input be negative?
Can the input be fractional?
Does the output have dashes? e.g. "twenty-three"
Does the output have "and" between hundred and the rest? e.g. "one hundred and thirty one"

If the interviewer has been waiting for your clarifying questions, they will provide concrete answers, for example:

Question	Answer
What is the range of the input number? 0? 1–999? 1–999,999?	The range will be 1 to 999.
Can the input be negative?	Only positive number.
Can the input be fractional?	Only whole numbers.
Does the output have dashes? e.g. "twenty-three"	No dashes.
Does the output have "and" between hundred and the rest? e.g. "one hundred and thirty one"	No "and" word or dashes.

Takeaways

- Avoid assumptions.
- Ask good clarifying questions.
- Make sound judgment.

STEP 3: WRITE DOWN TEST CASES

Estimated time in interview: 5 minutes

Why?

Wait! Six minutes have passed, yet we haven't started coding. However, it is not the time to code just yet.

Before even writing a single line of code, let's list all the test cases, including both common cases and corner/edge cases. Having test cases ready before coding provides a better picture than the actual product requirements about what you want to build. It materializes the problem space.

A benefit of this: it allows you to continue clarifying the problem as you discover more cases.

Example: the English Number Conversion problem

Here are all the possible types of test cases for the example question:

```
1...9 → one...nine

10...19 → ten...nineteen

20, 30,...90 → twenty, thirty,...ninety

21 → twenty one

100 → one hundred

101...109 → one hundred one...one hundred nine

110...119 → one hundred ten...one hundred nineteen

120 → one hundred twenty

123 → one hundred twenty three

200 → two hundred

999 → nine hundred ninety nine
```

First of all, to translate from a number to words, you need a dictionary.

From the test cases above, you can get an exhaustive list of unique words you need for this dictionary. As you enumerate all the possible cases, your mind starts to pick up some patterns.

In addition, here is a list of edge cases to be considered:

```
-1 → throws an exception

0 → throws an exception

1000 → throws an exception

1.1 → throws an exception
```

Some candidates don't test at all. Some candidates test in the end. Some candidates only test good cases. Only a few that I have met start with an exhaustive list of test cases before coding.

Takeaways

- Write down test cases, as thoroughly as possible
- Include both common use cases and edge cases

STEP 4: DESCRIBE AND WRITE DOWN THE ALGORITHM

Estimated time in interview: 15 minutes

> "An hour of planning can save you 10 hours of doing."
>
> — Dale Carnegie

Any step before this one is a preparation for this step. This step is to demonstrate your computer science fundamental skills.

Pick Data Structures

Data structure is the way data is stored which can help efficiently solve this problem.

Here are the common data structures:

- Arrays
- Linked List
- Stack
- Hash Map
- Tree
- Heap
- Graph

For each data structure, its unique features make a complex problem easier.

For one problem, you can use different data structures to solve it. But that does not mean you can choose any of them to solve. This is why you should ask interviewers what they would like the solution to optimize for, space, speed, or anything else.

Besides that, you can use more than one data structure to solve.

Describe Your Thinking

You may be asked to elaborate on how you would solve the problem, or describe the algorithm before coding.

Some interviewers may just watch what you will do — will you rush into code, or have a clear solution ready before coding?

From my observations of all the past candidates, the ones who immediately jumped to code often struggled to solve the problem in time, or had a working solution that was difficult to follow.

Only those who had formed a clear idea before the implementation managed to solve the problem with precise and readable code.

Based on the test cases, craft a solution with appropriate data structures and algorithms.

Interestingly, a book about why stellar engineers (often Russian) on Wall Street are so good at problem-solving in interviews also mentions a similar process.

> Russians had a reputation for being the best programmers on Wall Street, and Serge thought he knew why: They had been forced to learn to program computers without the luxury of endless computer time.
>
> Many years later, when he had plenty of computer time, Serge still wrote out new programs on paper before typing them into the machine. “In Russia, time on the computer was measured in minutes,” he said. “When you write a program, you are given a

> tiny time slot to make it work. Consequently, we learned to write the code in ways that minimized the amount of debugging. And so you had to think about it a lot before you committed it to paper. . . . The ready availability of computer time creates this mode of working where you just have an idea and type it and maybe erase it ten times. Good Russian programmers, they tend to have had that one experience at some time in the past — the experience of limited access to computer time."
>
> — Michael Lewis. Flash Boys: A Wall Street Revolt

Complexity Analysis

Complexity Analysis, also known as Big O analysis, is a way to measure runtime data storage your algorithm will require, and how fast your algorithm can run.

To understand how Big O analysis works, GeeksForGeeks provides the best explanation[2].

Not all optimal solutions demand minimal storage or the fastest runtime; often, there's a trade-off within the context.

Complexity analysis is in almost every coding interview. Interviewers either instruct you to perform it or expect you to have it in mind when designing your approach. Obviously, the latter shows your maturity the most.

If you use any of the common structures and understand the space and time complexity, the complexity of the entire solution is built upon them.

---

[2] How to do Big O Complexity Analysis https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

Example: the English Number Conversion problem

Before the number 20, all the words are unique words, such as "*one*", "*five*", "*ten*", and "*thirteen*".

Therefore, the most effective data structure is a *hashmap*, also called a *dictionary*. The time complexity for the *hashmap* is *O(1)*. Its space complexity is *O(n)*, however, the words in this dictionary are known, so the size of the *hashmap* is predefined and limited, so it can be considered as *O(1)*.

After 20, there is a pattern to combine unique words.

Here is the first draft of my algorithm:

1. Create a dictionary for all the unique words (one ... ten, eleven ... nineteen, twenty, thirty, forty, ... ninety)
2. Check the boundaries: <1 and > 999
3. Check simple unique words: <20
4. Check ones: if ones, look up the word in the dictionary; otherwise, do nothing
5. Check tens: if tens, look up the word in the dictionary; otherwise, do nothing
6. Check hundreds: if hundreds, look up the word in the dictionary, plus "<space>hundred"; otherwise, do nothing
7. Put all the words together

All roads lead to Rome — there are often multiple ways to solve a single problem.

I noticed that the solution above did not flow smoothly, because, in English, hundreds come before tens and ones.

I immediately noticed that I needed to do step #6, then #5 and #4, as follows:

1. Create a dictionary for all the unique words (one ... ten, eleven ... nineteen, twenty, thirty, forty, ... ninety)
2. Check the boundaries: <1 and >999
3. Check simple unique words: <20
4. Check hundreds: if hundreds, look up the word in the dictionary, plus "<space>hundred"; otherwise, do nothing
5. Check tens: if tens, look up the word in the dictionary; otherwise, do nothing
6. Check ones: if ones, look up the word in the dictionary; otherwise, do nothing
7. Put all the words together

Overall, the time complexity of this algorithm is *O(1)* and the space complexity is *O(1)*. It is a very efficient solution by far.

If the complexity goes over *O(1)*, I would find ways to optimize it.

Takeaways

- Choose appropriate data structures with reasons.
- Describe your thinking.
- Analyze the time and space complexity.
- Explain the system impact.
- Offer optimization if possible.

STEP 5: START CODING

Estimated time in interview: 5 minutes

Once you have completed all the previous steps, the actual coding becomes the easiest part.

Based on the algorithm you have come up with in Step 4, you translate it into code. This step should be relatively quick, as the work is a direct translation from human language to a programming language of your choice.

Maintaining a clear mindset also provides a boost of confidence.

Example: the English Number Conversion problem

It took me 5 minutes to type all the code below and get it running:

```
// Create a dictionary for all the unique words (one … ten,
eleven … nineteen, twenty, thirty, forty, … ninety)
var WORD_DICT = {
  0: "",
  1: "one",
  2: "two",
  3: "three",
  4: "four",
  5: "five",
  6: "six",
  7: "seven",
  8: "eight",
  9: "nine",
  10: "ten",
  11: "eleven",
  12: "twelve",
  13: "thirteen",
  14: "fourteen",
  15: "fifteen",
  16: "sixteen",
  17: "seventeen",
  18: "eighteen",
  19: "nineteen",
  20: "twenty",
```

```
  30: "thirty",
  40: "forty",
  50: "fifty",
  60: "sixty",
  70: "seventy",
  80: "eighty",
  90: "ninety",
  100: "hundred"
};

function convertNumberToWords(inputNumber) {
  var outputWords = []

  // Check the boundaries: <1 and > 999
  if (!inputNumber || inputNumber < 1 || inputNumber > 999)
{
    throw new Error("Invalid input");
  }

  // Check simple unique words: <20
  if (inputNumber < 20) {
    return WORD_DICT[inputNumber];
  }

  var hundredsPosition = Math.floor(inputNumber / 100);
  var tensPosition = hundredsPosition % 100;
  var onesPostion = hundredsPosition % 10;

  // Check hundreds: if hundreds, look up the word in
dictionary, plus "<space>hundred"; otherwise, do nothing
  if (hundredsPosition > 0) {
    outputWords.push(WORD_DICT[hundredsPosition],
WORD_DICT[100]);
  }

  // Check tens: if tens, look up the word in dictionary;
otherwise, do nothing
  if (tensPosition >= 2) {
    outputWords.push(WORD_DICT[10 * tensPosition]);
    // Check ones: if ones, look up the word in dictionary;
otherwise, do nothing
    if (onesPostion) {
      outputWords.push(WORD_DICT[onesPostion])
    }
  } else {
    outputWords.push(WORD_DICT[inputNumber -
hundredsPosition * 100]);
  }


  // Put all the words together
  return outputWords.join(" ");
}
```

Takeaways

- Use your most comfortable programming language.
- Fix bugs and syntax errors on your own.
- Refactor as you go.
- Finish the implementation in time.

STEP 6: TEST

Estimated time in interview: 5 minutes

"The man of science has learned to believe in justification, not by faith, but by verification."

— Thomas H. Huxley

Once you have completed the coding, it's time to test your code. How? Use the test cases you provided in Step 3. If any test case does not pass, fix it.

How can you ensure your test coverage is comprehensive? I have an effective technique: check every code branch. A code branch is where the code diverges and the results can be different based on the inputs. For example, *if*, *while*, *switch*, and *for* statements. If an *if* statement is used, two test cases will need to be added.

For example, in this if statement:

```
if (inputNumber < 20) {
  return true;
}

return false
```

There are 2 code branches around *if* statement. So we can easily come up with 2 test cases: *inputNumber* = 19, and *inputNumber* = 20.

As a best practice, since 20 is the boundary for this if statement, we can have 3 test cases, to cover the left of the boundary (19), the boundary (20), and the right of the boundary (21).

Example: the English Number Conversion problem

1...9 → one...nine	Part of the dictionary. It looks up the dictionary and returns early.

10...19 → ten...nineteen	Part of the dictionary. It looks up the dictionary and returns early.
20, 30,...90 → twenty, thirty,... ninety	Part of the dictionary. It looks up the dictionary and returns early.
21 → twenty one	It goes into the tensPosition and onesPosition checks.
100 → one hundred	The hundred position is parsed. It looks up the dictionary and returns early.
101...109 → one hundred one...one hundred nine	The hundred position is parsed. It looks up the dictionary for ones and returns.
110...119 → one hundred ten...one hundred nineteen	The hundred position is parsed. It looks up the dictionary for teens and returns.
120 → one hundred twenty	The hundred position is parsed. It looks up the dictionary for 20 and returns.
123 → one hundred twenty three	The hundred position is parsed. It looks up the dictionary for tens and ones and returns.
200 → two hundred	The hundred position is parsed and returned.
999 → nine hundred ninety nine	The hundred position is parsed. It looks up the dictionary for tens and ones and returns.
-1 → throws an exception	It returns an exception.
0 → throws an exception	It returns an exception.
1000 → throws an exception	It returns an exception.

Takeaways

- Verify the code with test cases
- Check all the code branches

CONCLUSION

Those are all six steps!

Established Process At Work

In case you are not aware yet, whether you can solve the coding problem is not the only thing that is evaluated, but your whole package of performance has been evaluated since the very beginning: communication, requirement gathering, data structures, algorithms, and testing.

They reflect the critical functions of an engineer day in and day out.

Most candidates can perform better if they have a process to follow, improve, and optimize. This Six-Step Mental Framework is to help you establish this process in your interview and day-to-day work.

The Case of Performing Under Pressure

Now, you may wonder whether this framework would work during interviews when you have to problem-solve under pressure. That's often the key reason why brilliant engineers perform well at work but perform poorly at interviews.

This phenomenon of underperformance under pressure is known as choking. The cognitive psychologist Sian Beilock explains in her book "Choke: What the Secrets of the Brain Reveal About Getting It Right When You Have To" that choking often happens because of the brain's response to stress, which can disrupt the cognitive processes required for skilled performance.

Some key factors contributing to choking are overthinking, stress, and

negative self-talk. As she suggests for preventing choking, chunking information is one of the five strategies (Practice Under Pressure, Cognitive Techniques, Chunking Information, Mental Rehearsal, Self-Awareness, Breathing and Relaxation Techniques).

The Six-Step Mental Framework is a way to chunk information into meaningful groups. This can reduce cognitive load and improve performance.

Practice, Practice, Practice

Now you can head to LeetCode, pick a random coding problem, and try again with this Six-Step Mental Framework.

If you are not sure yet, we can review this framework in the examples in the following chapters. The goal of the examples is to showcase the thought process when adopting the Six-Step Mental Framework in real-life interviews.

EXAMPLE PROBLEM #1: ARRAYS: MOVE ZEROS TO THE LEFT[3]

Step 1: Write Down The Problem

Given an integer array, move all elements that are 0 to the left while maintaining the order of other elements in the array.

Example input:

Index	0	1	2	3	4	5	6
Value	32	43	0	7	98	0	8

Output:

Index	0	1	2	3	4	5	6
Value	0	0	32	43	7	98	8

Step 2: Clarify The Problem Space

Here are the questions I would ask and the answers I may get from the interviewer:

Edge Cases

[3] Problem source: https://www.educative.io/blog/cracking-top-facebook-coding-interview-questions

Question	Answer
Can the elements be other types, such as strings?	No. Integer only
Can the elements be positive and negative?	Yes, they can be positive and negative
Can the elements be empty, such as null?	No. The array will be non-empty

Scale

Question	Answer
How large is the array? Can it be fit in memory on one machine?	It will be a limited size, for example, 100 at most.
How big can the integers be?	It can be just a regular int.

Objectives

Question	Answer
Do we optimize by time? Or space?	Time complexity is `O(n)` and space complexity is `O(1)`.
Can we use any functions like push and pop?	Yes, you can.

Step 3: Write Down Test Cases

Test 1: Happy Case

Input:

Index	0	1	2	3	4	5	6
Value	32	43	0	7	98	0	8

Output:

Index	0	1	2	3	4	5	6
Value	0	0	32	43	7	98	8

Test 2: Zero Case

Input:

Index	0	1	2	3	4	5	6
Value	0	0	0	0	0	0	0

Output:

Index	0	1	2	3	4	5	6
Value	0	0	0	0	0	0	0

Test 3: Non-Zero Case

Input:

Index	0	1	2	3	4	5	6
Value	32	43	45	343	43	34	43

Output:

Index	0	1	2	3	4	5	6
Value	32	43	45	343	43	34	43

Test 4: No Change Case

Input:

Index	0	1	2	3	4	5	6
Value	0	0	45	343	43	34	43

Output:

Index	0	1	2	3	4	5	6
Value	0	0	45	343	43	34	43

Test 5: Zeroes on the other side

Input:

Index	0	1	2	3	4	5	6
Value	23	23	45	343	43	0	0

Output:

Index	0	1	2	3	4	5	6
Value	0	0	23	23	45	343	43

Step 4: Describe and Write Down the Algorithm

I can think of several ways to solve this:

1. Pop the array as a stack, and insert the non-zero elements at the front. Keep track of the number of 0s, and put them back at the end.

2. Traverse the array, find all the 0s, and then move elements one by one.

While Approach #2 seems straightforward, keeping the indexes and moving accordingly seems very messy.

Approach #1 seems clever, as it offers a simple solution that meets the desired time complexity of O(n) and extra space complexity of O(1).

So the steps of Approach #1 are the following:

1. Pop the element from the far right of the array.
2. If the element is 0, increment the zero counter; if not, place it at the leftmost position.
3. Continue this process for iterations equal to the size of the array.
4. If the zero counter is greater than 0, add 0 to the leftmost position for a number of times equal to the zero counter value.
5. Return the modified array.

Check the algorithm above with all the test cases, and confirm they work.

Assess the complexity. Apart from the array, the extra space complexity remains *O(1)*. The time complexity is *O(n)*, where n represents the array size.

Step 5: Start Coding

```
function moveZeroesToLeft(inputArray) {
  var zeroCounter = 0;
  var arraySize = inputArray.length;

  for (var i = 0; i < arraySize; i++) {
    var poppedElement = inputArray.pop();

    if (poppedElement == 0) {
      zeroCounter++;
    } else {
      inputArray.unshift(poppedElement);
    }
  }

  for (var j = 0; j < zeroCounter; j++) {
    inputArray.unshift(0);
  }

  return inputArray;
}
```

Step 6: Test

```
console.log(moveZeroesToLeft([1,2,3,4,0,0]))
>[0, 0, 1, 2, 3, 4]

console.log(moveZeroesToLeft([1,2,3,4,5,6]))
>[1, 2, 3, 4, 5, 6]

console.log(moveZeroesToLeft([0,0,3,4,5,6]))
>[0, 0, 3, 4, 5, 6]

console.log(moveZeroesToLeft([1,2,0,4,0,6]))
>[0, 0, 1, 2, 4, 6]

console.log(moveZeroesToLeft([0,0,0,0,0,0]))
>[0, 0, 0, 0, 0, 0]

console.log(moveZeroesToLeft([0,0,0,0,-1,0]))
>[0, 0, 0, 0, 0, -1]
```

EXAMPLE PROBLEM #2: LONGEST PATH IN THE MATRIX[4]

Step 1: Write Down The Problem

Given a matrix with N rows and M columns, where each cell *m[i][j]* represents a value, you can move to *m[i+1][j]* if *m[i+1][j] > m[i][j]*, or to *m[i][j+1]* if *m[i][j+1] > m[i][j]*.

The task is to print the longest path length if we start from (0, 0).

Based on the above description, I draw down an example matrix to help my understanding of the problem.

2 (0,0)	23	2	4	5
3	43	12	2	4
5	2	4	43	5
1	3	3	4	87

I noticed that all possible paths lead towards the bottom-right corner. Whenever the anchor has moved, the same logic applies. So the problem could be solved using a recursive approach.

Step 2: Clarify The Problem Space

General

[4] Question source: https://igotanoffer.com/blogs/tech/google-software-engineer-interview#questions

Question	Answer
Can the input be a two-dimensional array?	Yes

Edge Cases

Question	Answer
What if m[i+1][j] = m[i][j] and m[i][j+1] = m[i][j]?	It should not move forward. It should just stop.
What if m[i+1][j] > m[i][j] and m[i][j+1] > m[i][j]?	It presents two routes.
Can any element be empty?	We can assume the matrix has non empty elements.
Can the number of the dimensions be more than 2?	No.

Scale

Question	Answer
How large can this matrix be? Can it be fit in memory on one machine?	Yes, it can fit into the memory

Objectives

Question	Answer
Do we optimize by time? Or space?	As efficient as possible.

Step 3: Write Down Test Cases

Test 1: Happy Case

Input:

```
[
  [1, 2, 3],
  [2, 3, 4]
]
```

Output: 3

Test 2: Empty Cases

Input:

```
[]
```

Output: 0

Input:

```
[[]]
```

Output: 0

Test 3: With Identical Elements

Input:

```
[
  [1, 1, 1],
  [1, 1, 1]
]
```

Output: 0

Test 4: Three plus Dimensional Array

Input:

```
[
  [[1, 2, 3]],
  [[2, 3, 4]]
]
```

Output: (throws an error)

Test 5: Irregular Matrix

Input:

```
[
  [1, 2, 3],
```

```
  [2, 3, 4],
  [100, 300]
]
```

Output: 3

Step 4: Describe and Write Down the Algorithm

The input will be an array of arrays containing integers, and the output will be an integer.

The output starts with 0.

1. Set the element at *inputArray[0][0]* as an anchor.
2. Compare the value at *inputArray[1][0]* with the value at *inputArray[0][0]*. If *inputArray[1][0] > inputArray[0][0]*, set the element at *inputArray[1][0]* as the anchor, and increment the output by 1. If the size of *inputArray* is greater than 1, repeat step #2.
3. Alternatively, compare the value at *inputArray[0][1]* with the value at *inputArray[0][0]*. If *inputArray[0][1] > inputArray[0][0]*, set the element at *inputArray[0][1]* as the anchor, and increment the output by 1. If the size of *inputArray[0]* is greater than 1, repeat step #2.
4. If neither condition in step #2 or #3 is met, return the output value.

Step 5: Start Coding

First, I started with the version based on the algorithm I had above.

```
function getLongestPathLength(inputArray) {
  var allLengths = [];

  getPathLength(inputArray, 0, 0, 0, allLengths);

  return Math.max(allLengths);
}
```

```
function isValidMatrixElement(inputArray, x, y) {
  if (inputArray[x] === undefined) {
    return false;
  }

  if (inputArray[x][y] === undefined) {
    return false;
  }

  return true;
}

function getPathLength(inputArray, currentX, currentY,
currentLength, allLengths) {
  if (!isValidMatrixElement(inputArray, currentX,
currentY)) {
    return currentLength;
  }

  var ySize = inputArray[currentX].length
  var xSize = inputArray.length
  var anchorValue = inputArray[currentX][currentY];

  if (xSize > currentX &&
    isValidMatrixElement(inputArray, currentX + 1,
currentY) &&
    inputArray[currentX + 1][currentY] > anchorValue) {

    return getPathLength(inputArray, currentX + 1,
currentY, currentLength + 1, allLengths);

  } else if (ySize > currentY &&
    isValidMatrixElement(inputArray, currentX, currentY +
1) &&
    inputArray[currentX][currentY + 1] > anchorValue) {
    return getPathLength(inputArray, currentX, currentY +
1, currentLength + 1, allLengths);

  } else {
    allLengths.push(currentLength);
    return currentLength;
  }
}

console.log(getLongestPathLength([]));

console.log(getLongestPathLength([
  []
]));

console.log(getLongestPathLength([
  [1, 1],
  [1, 1]
]));
```

```
console.log(getLongestPathLength([
  [1, 2],
  [1, 2]
]));

console.log(getLongestPathLength([
  [1, 2, 3],
  [1, 2, 3]
]));

console.log(getLongestPathLength([
  [1, 2, 3],
  [2, 3, 4]
]));

console.log(getLongestPathLength([
  [1, 2, 3],
  [2, 3, 4],
  [100, 300, 400]
]));
```

By testing with the cases above, I found some of my code that was broken and fixed it accordingly.

In the initial solution, the time complexity is O(NM), where N and M represent the size of either dimension. The space complexity is also O(NM) because, for each length, it needs to be stored in the array.

Then I noticed that I could optimize by eliminating the array *allLengths*. This adjustment reduces the space complexity to O(1), in addition to the input array.

```
function getLongestPathLength(inputArray) {

  return _getLongestPathLength(inputArray, 0, 0, 0, 0);
}

function isValidMatrixElement(inputArray, x, y) {
  if (inputArray[x] === undefined) {
    return false;
  }

  if (inputArray[x][y] === undefined) {
    return false;
  }
```

```
  if (typeof inputArray[x][y] !== 'number') {
    throw 'invalid element entry, expect number, but got '
+ typeof inputArray[x][y] + ': ' + inputArray[x][y];
  }

  return true;
}

function _getLongestPathLength(inputArray, currentX,
currentY, currentLength, maxLength) {
  if (!isValidMatrixElement(inputArray, currentX,
currentY)) {
    return currentLength;
  }

  var ySize = inputArray[currentX].length
  var xSize = inputArray.length
  var anchorValue = inputArray[currentX][currentY];

  if (xSize > currentX &&
    isValidMatrixElement(inputArray, currentX + 1,
currentY) &&
    inputArray[currentX + 1][currentY] > anchorValue) {

    return _getLongestPathLength(inputArray, currentX + 1,
currentY, currentLength + 1, maxLength);

  } else if (ySize > currentY &&
    isValidMatrixElement(inputArray, currentX, currentY +
1) &&
    inputArray[currentX][currentY + 1] > anchorValue) {
    return _getLongestPathLength(inputArray, currentX,
currentY + 1, currentLength + 1, maxLength);

  } else {
    return Math.max(currentLength, maxLength);
  }
}

console.log(getLongestPathLength([]));

console.log(getLongestPathLength([
  []
]));

console.log(getLongestPathLength([
  [1, 1],
  [1, 1]
]));

console.log(getLongestPathLength([
  [1, 2],
  [1, 2]
]));
```

```
console.log(getLongestPathLength([
  [1, 2, 3],
  [1, 2, 3]
]));

console.log(getLongestPathLength([
  [1, 2, 3],
  [2, 3, 4]
]));

console.log(getLongestPathLength([
  [1, 2, 3],
  [2, 3, 4],
  [100, 300, 400]
]));

console.log(getLongestPathLength([
  [1, 2, 3],
  [2, 3, 4],
  [100, 300]
]));

console.log(getLongestPathLength([
  [
    [1, 2, 3]
  ],
  [
    [2, 3, 4]
  ]
]));
```

Step 6: Test

```
console.log(getLongestPathLength([]));
```

```
0
```

```
console.log(getLongestPathLength([
  []
]));
```

```
0
```

```
console.log(getLongestPathLength([
  [1, 1],
  [1, 1]
]));
```

```
0
```

```
console.log(getLongestPathLength([
  [1, 2],
  [1, 2]
]));

1

console.log(getLongestPathLength([
  [1, 2, 3],
  [1, 2, 3]
]));

2

console.log(getLongestPathLength([
  [1, 2, 3],
  [2, 3, 4]
]));

3

console.log(getLongestPathLength([
  [1, 2, 3],
  [2, 3, 4],
  [100, 300, 400]
]));

4

console.log(getLongestPathLength([
  [1, 2, 3],
  [2, 3, 4],
  [100, 300]
]));

3

console.log(getLongestPathLength([
  [[1, 2, 3]],
  [[2, 3, 4]]
]));

Uncaught invalid element entry, expect number, but got
object: 1,2,3
```

EXAMPLE PROBLEM #3: FIND ALL THE POSSIBLE SUBSETS[5]

Step 1: Write Down The Problem

Given a set of integers, find all the possible subsets.

For example,

Input

2	3	4

Output

	2	3	2,3	4	2,4	3,4	2,3,4

Step 2: Clarify The Problem Space

General

Question	Answer
Does the order of elements in the subsets matter? In other words, should [1, 2] and [2, 1] be considered as different subsets or the same?	No. [1,2] and [2,1] should be considered as the same subset.

[5] https://www.codinginterview.com/netflix-interview-questions

Question	Answer
What is the format of the input data? Is it a list, array, or some other data structure?	It is an array.
What is the expected format of the output? Should the subsets be represented as lists, arrays, or some other data structure?	An array of arrays should be okay.

Edge Cases

Question	Answer
Should the empty set be included as a valid subset?	Yes.
Can the input contain duplicate elements?	Yes, the input can have duplicate elements. But the output has no duplicate elements. Input: `[1, 1]` Output: `[[], [1]]`
What happens when the input set contains only one element?	Input: `[1]` Output: `[[], [1]]`

Scale

Question	Answer
Do you expect this algorithm to work efficiently for large input sets, or is it primarily designed for smaller sets?	Work for small sets for now.

Objectives

Question	Answer
Are there any constraints on the time or space complexity of the solution?	As efficient as possible.

Step 3: Write Down Test Cases

Test 1: Happy Case

Input: [1, 2]

Output: [[], [1], [2], [1, 2]]

Test 2: Empty Set

Input: []

Output: [[]]

Test 3: Single Element Set

Input: [42]

Output: [[], [42]]

Test 4: Set with Duplicates

Input: [1, 2, 2]

Output: [[], [1], [2], [1, 2], [2, 2], [1, 2, 2]]

Test 5: Set with Repeated Elements

Input: [1, 1, 2, 2]

Output: [[], [1], [2], [1, 2], [1, 1], [2, 2], [1, 1, 2], [1, 2, 2], [1, 1, 2, 2]]

Test 6: Set with Repeated Elements, Not Ordered

Input: [2, 1, 1, 2]

Output: [[], [1], [2], [1, 2], [1, 1], [2, 2], [1, 1, 2], [1, 2, 2], [1, 1, 2, 2]]

Step 4: Describe and Write Down the Algorithm

Based on the test cases, I noticed the recursive nature of the outputs as well as the need to backtrack to avoid duplicates.

The algorithm I'm using to find all possible subsets of a given set of integers is based on a recursive approach with backtracking.

Here's a step-by-step description of the algorithm:

1. Sort the input array.
2. Initialize an empty array to store subsets.
3. Define a recursive function to backtrack.
4. In the function, if the current index is equal to the array length, add the current subset to the result.
5. Iterate through the array elements starting from the current index.
    1. Include the element in the current subset, recursively call the function with an updated index, and remove the last element from the subset.
6. Remove the last element from the subset after each recursive call.
7. Start the recursion with an initial index of 0 and an empty subset.
8. Return the result array containing all subsets.

Now assess the complexity.

The Time Complexity is exponential, $O(2^n)$, with n representing the number of elements in the input set. This is because, for each element

in the input set, there are two choices: either include it in the current subset or exclude it. This leads to a branching factor of 2^n, where n is the number of elements in the input set. In the worst case, you will explore all possible subsets, resulting in 2^n subsets.

The Space Complexity is linear, $O(n)$, with n as the number of elements in the input set. In the worst case, when you include all elements in the current subset, the maximum depth of the recursion is n.

Step 5: Start Coding

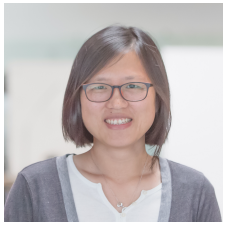
```
function findSubsets(nums) {
  const subsets = [];

  function backtrack(start, currentSubset) {
    subsets.push(currentSubset.slice()); // Clone the
current subset and add it to the result

    for (let i = start; i < nums.length; i++) {
      if (i > start && nums[i] === nums[i - 1]) {
        // Skip duplicate elements to avoid duplicate
subsets
        continue;
      }

      currentSubset.push(nums[i]);
      backtrack(i + 1, currentSubset); // Recursively
generate subsets
      currentSubset.pop(); // Backtrack by removing the
last element
    }
  }

  nums.sort((a, b) => a - b); // Sort the input to handle
duplicates

  backtrack(0, []); // Start the recursion
  return subsets;
}
```

Step 6: Test

```
console.log("Test 1: Happy Case");
console.log(findSubsets([1, 2]));
```

```
// Output: [[], [1], [2], [1, 2]]

console.log("Test 2: Empty Set");
console.log(findSubsets([]));
// Output: [[]]

console.log("Test 3: Single Element Set");
console.log(findSubsets([42]));
// Output: [[], [42]]

console.log("Test 4: Set with Duplicates");
console.log(findSubsets([1, 2, 2]));
// Output: [[], [1], [2], [1, 2], [2, 2], [1, 2, 2]]

console.log("Test 5: Set with Repeated Elements");
console.log(findSubsets([1, 1, 2, 2]));
// Output: [[], [1], [2], [1, 2], [1, 1], [2, 2], [1, 1,
2], [1, 2, 2], [1, 1, 2, 2]]

console.log("Test 6: Set with Repeated Elements, Not
Ordered");
console.log(findSubsets([2, 1, 1, 2]));
// Output: [[], [1], [2], [1, 2], [1, 1], [2, 2], [1, 1,
2], [1, 2, 2], [1, 1, 2, 2]]
```

ABOUT AUTHOR



Grace Huang was a software engineer at several big tech companies, including Amazon, and Bloomberg. Grace co-founded a hardware / AI company, Roxy. The product line was later acquired and the team joined Twitter. Since leaving Twitter, Grace has been focusing on writing and teaching.

Other technical books that Grace wrote:

- Build macOS Apps With SwiftUI: A Practical Learning Guide (https://amzn.to/40PUpzu)
- Dynamic Trio: Building Web Applications with React, Next.js & Tailwind (https://amzn.to/3sHbnDV)
- Code Reviews In Tech: The Missing Guide (https://gracehuang.gumroad.com/l/codereviews)
- A Practical Guide to Writing a Software Technical Design Document (https://gracehuang.gumroad.com/l/mqmUt)

You can reach Grace at @imgracehuang on Twitter.