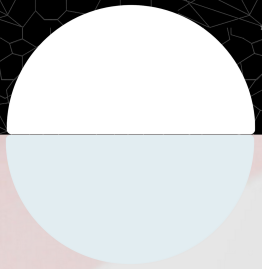
GuideBook

# How to Solve Algorithm Problems?

Make Coding Interview Preparation Less Painful



WALEED KHAMIES

# How to Solve Algorithm Problems

## Make Coding Interview Preparation Less Painful

WALEED KHAMIES

This book is for sale at
http://leanpub.com/how-to-solve-algorithm-problems-book

This version was published on 2023-05-21 ISBN 978-1-7390105-1-5




# Tweet This Book!

Please help WALEED KHAMIES by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Exciting news! Just got my copy of ”How to Solve Algorithm Problems” by @khamiesw! I can’t wait to dive into the strategies and techniques shared in this essential guide. Join me on this coding journey and grab your copy today! Happy coding! #TechBooks #NewRelease

The suggested hashtag for this book is #how-to-solve-algorithm-problems.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#how-to-solve-algorithm-problems

*To my lovely parents, Ahmed, and Salha,*

*No words can fully describe the sincere gratitude and love I have for everything you have done for us. I won't be here without you, so thank you!*

*To the gang, my brothers,*

*Who are facing immense danger and hardship due to the ongoing military conflict in Sudan between the RSF militia and the Sudanese army. Please hold on, things will get better soon.*

*To my only land, Sudan,*

*I lose a piece of my heart every single day. You have been through a lot, so much pain and suffering, with rampant corruption and violence. But I believe that we will overcome these challenges and emerge stronger than before.*

*To my beloved reader,*

*I hope this guide serves as a valuable resource throughout your career. May it empower you to achieve your goals and reach new heights of success.*

# Table of Contents

**GuideBook** . . . . . **i**
How to Solve Algorithms Problems? . . . . . i
Make Coding Interview Preparation Less Painful . . . . . i

**Preface** . . . . . **iii**

**How to Solve Algorithm Problems** . . . . . **1**
How Should You Read This Guide? . . . . . 2
Notion Template: Learn Algorithms By Doing . . . . . 3
What Is The Goal Of This Guide? . . . . . 3
What Should You Expect After Completing This Guide? . . . . . 4

**Learning Cycle & Interview Assessment Factors** . . . . . **7**
2.1 Understanding the Learning Cycle . . . . . 7
2.1.1 Beginner Phase . . . . . 8
2.1.2 Experienced Phase . . . . . 8
2.1.3 Senior Phase . . . . . 9
2.2 Interview Assessment Factors . . . . . 11
2.2.1 Understanding of Algorithms and Data Structures . . . . . 11
2.2.2 Problem-Solving Skills . . . . . 14
2.2.3 Attention To Detail . . . . . 14
2.2.4 Code Efficiency . . . . . 15
2.2.5 Time Complexity Analysis . . . . . 15
2.2.6 Modular Code . . . . . 16
2.2.7 Debugging . . . . . 17
2.2.8 Communication . . . . . 19

**Solving Algorithm Problems** . . . . . **24**

3.1 Steps to Solve Algorithm Problems . . . . . 24
3.1.1 Understand the Problem . . . . . 24
3.1.2 Formalize the Problem . . . . . 25
3.1.3 Repeat Reading the Question Yourself . . . . . 25
3.1.4 Bring Input Examples . . . . . 25
3.1.5 Develop a Brute-Force Solution . . . . . 26
3.1.6 Analyze Time and Space Complexities For the Brute-Force Solution . . . . . 26
3.1.7 Optimize The Brute-Force Solution . . . . . 26
3.1.8 Analyze Time and Space Complexities For the Optimized Solution . . . . . 27
3.2 KSum Family Problems . . . . . 28
3.2.1 The 2Sum Problem . . . . . 28
3.2.2 The 3Sum Problem . . . . . 34
3.2.3 KSum . . . . . 40

**4.1. FGCC Framework . . . . . 48**
4.1.1 What is FGCC? . . . . . 48
4.1.2 A Mental Framework . . . . . 49
4.1.3 Steps To Apply FGCC Framework . . . . . 49

**4.2. FGCC In Practice . . . . . 51**
4.2.1 Introduction to Backtracking Technique . . . . . 51
4.2.1.1 Finding Permutations . . . . . 51
Solution . . . . . 52
4.2.1.2 Finding Combinations . . . . . 54
4.2.1.3 Letter Combinations of a Phone Number . . . . . 56
4.2.2 The Pillars of the FGCC Framework . . . . . 59

**5. Top #3 Algorithm Techniques . . . . . 65**
5.1 Two Pointers . . . . . 65
5.1.1 Code Example . . . . . 66
5.1.2 Usage . . . . . 68
5.1.3 Data Structures . . . . . 68
5.2 Breadth-First Search (BFS) . . . . . 68
5.2.1 Code Example . . . . . 69
5.2.2 Usage . . . . . 70

5.2.3 Data Structures . . . . . 71
5.3 Depth-First Search (DFS) . . . . . 71
5.3.1 Usage . . . . . 74
5.3.2 Data Structures . . . . . 74

**6. Supplements** . . . . . **77**
6.1 Detect a Linked List Cycle . . . . . 78
6.1.1 Problem Description . . . . . 78
6.1.2 I/O Examples . . . . . 78
6.1.3 Solution . . . . . 79
6.1.4 Complexity Analysis . . . . . 80
6.2 Remove the Nth Node From the End of a Linked List . . . . . 80
6.2.1 Problem Description . . . . . 81
6.2.2 I/O Examples . . . . . 81
6.2.3 Solution . . . . . 82
6.2.4 Complexity Analysis . . . . . 83
6.3 Swapping Linked List Node Pairs . . . . . 84
6.3.1 Problem Description . . . . . 84
6.3.2 I/O Examples . . . . . 84
6.3.3 Solution . . . . . 86
6.3.4 Complexity Analysis . . . . . 90
6.4 Validate Binary Search Tree . . . . . 91
6.4.1 Problem Description . . . . . 91
6.4.2 I/O Examples . . . . . 92
6.4.3 Solution . . . . . 93
6.4.4 Complexity Analysis . . . . . 95
6.5 Same Binary Tree . . . . . 96
6.5.1 Problem Description . . . . . 96
6.5.2 I/O Examples . . . . . 96
6.5.3 Solution . . . . . 97
6.5.4 Complexity Analysis . . . . . 98
6.6 Symmetric Binary Tree . . . . . 99
6.6.1 Problem Description . . . . . 99
6.6.2 I/O Examples . . . . . 100
6.6.3 Solution . . . . . 101
6.6.4 Complexity Analysis . . . . . 101

**References** . . . . . **104**

**Acknowledgments** . . . . . **105**

**About the Author** . . . . . **106**

# GuideBook

## How to Solve Algorithms Problems?

### Make Coding Interview Preparation Less Painful

#### by Waleed Khamies

This book is also for sale on the following platforms:

**Leanpub**: https://leanpub.com/how-to-solve-algorithm-problems-book

**Khamies' Store**: https://shop.waleedkhamies.com/b/hsap

This version was published on 2023-05-21



www.waleedkhamies.com, info@waleedkhamies.com

*Book Cover By : Author on Canva.*

"" Clear Mind → Better Judgement → Better Outcome.""

NAVAL RAVIKANT

# Preface

During interview preparation, things can be foggy and stressful, and sometimes you may question whether you are qualified for the job. Does that sound familiar? If so, you are not alone! We have all been there.

At times, I become frustrated and disheartened, and on other days, I even question whether it is worth continuing with these interviews. Despite putting in significant effort to prepare, I often find that the outcome does not match my expectations or reflect my abilities. Later, I realized that it was because of my poor preparation and lack of balance between effectiveness and efficiency.

So, I began to focus on problems that were relevant to the position I was applying for. Additionally, I prioritized these problems based on their importance, starting with the most critical ones, and it worked like a charm!

However, during my preparation journey, I encountered many online resources. I conduct an extensive research looking for good materials that could provide me with two important things: saving my preparation time by helping me focus on important problems and techniques, and providing me with an in-depth explanation while being practical and concise.

I spent several months going from reading blog posts to taking online courses and even watching videos from that algorithm expert on YouTube who happens to be from India. However, the information was scattered, and there were things I should have learned and things I should not have because they were too advanced and less likely to be brought up during interviews. That is why I decided to write this guide. A guide that saves you time while making you focus on things that matter.

I feel obliged to pass these lessons on to you and make your journey a lot easier. You do not have to waste the same time that I invested in these interviews. Yes, I have learned a lot of things during my preparation journey,

but you can also use this time to learn other important things to you, so please enjoy this guide!

If you find this guide helpful and want to learn more about coding interviews, please check out my free series on **Medium**[1] where I discuss in-depth some families of algorithm patterns that interviewers love to bring up during coding interviews.

To stay up to date with the latest news regarding this book or want to hear more information about algorithms and data science, feel free to join my free newsletter: **Khamies' Data Diaries**[2] on Substack. My Substack community is where you can connect with me to share your thoughts and opinions.

Waleed Khamies, April 30, 2023.

[1] https://bit.ly/uacifds-list
[2] https://khamiesw.substack.com/

# 1 | INTRODUCTION

PHOTO BY JAN TINNEBERG ON UNSPLASH

# HOW TO SOLVE ALGORITHM PROBLEMS?

# How to Solve Algorithm Problems

In today's technical interviews, coding interviews have become an essential component, as they provide interviewers with a better understanding of a candidate's programming abilities. Many roles such as Software Engineer, Machine Learning Engineer, Data Scientist, or Research Scientist use coding interviews as part of their hiring pipeline. However, preparing for these coding interviews is not an easy journey. You could spend months doing the preparations, solving Leetcode problems, spend your whole evenings practicing, and skip some weekends and socials, but you still fail your interviews. Why?

It is not about doing the effort, lack of preparation, or time management, but about your effectiveness in doing these preparations. Being able to have a systemic approach to preparing for coding interviews is essential to successfully pass your coding interviews. This systemic approach includes the type of algorithm problems that you solve, the approach to solving these problems, the time spent to come up with an optimized solution, and the way of communicating to your interviewers.

Also, acing your coding interviews includes your ability to handle the randomness of the hiring pipeline. The tech hiring pipeline is not ideal and has a lot of blind spots, as you can unexpectedly get rejected because there is someone in the hiring team who does not like you, even though you are technically solid! That happens because of two main reasons:

1. **Confirmation Bias**: Which may occur during interviews when interviewers give more weight to certain negative aspects they observe in you, which confirm their pre-existing beliefs, and subsequently reject you based on those observations.

2. **Cultural Bias**: As part of the hiring process, most companies conduct behavioral interviews to assess whether you are a good fit for their organizational culture. Company culture encompasses shared values and beliefs among employees and can be global in scope.

To address these issues, we designed this practical guide to help you in the preparation process and how you successfully navigate coding interviews. First, we provided an in-depth explanation of the main factors used in assessing potential candidates. Then, we explained how to approach any algorithm problem using a systematic approach that simplifies the whole process.

Chapter 2 will cover the stages of learning that every problem solver must undergo. It will also outline the factors commonly used by interviewers to evaluate candidates. Moving on to Chapter 3, we will delve into the fundamental steps necessary for effectively solving any algorithm problem. Concrete examples will be provided to demonstrate how to approach these algorithm problems efficiently.

In Chapter 4, a basic framework will be presented, designed to aid candidates in their preparation. Algorithm problems will be provided to demonstrate how the framework works in practice. Lastly, in Chapter 5, we will highlight key algorithm techniques that can be applied to many algorithmic problems. Additionally, a problem-solving checklist tailored to beginners will be provided to help them remain on track during their coding preparation.

# How Should You Read This Guide?

This guide is designed to be practical and self-contained, which means you can benefit the most from it by practicing solving algorithm problems. If you forget some of the steps mentioned in this book or need a refresher, you can always revisit the guide multiple times. Additionally, to make the learning process more engaging, we have included a notion template that can help you remember the steps and cultivate the right mindset. This template contains some of the most introductory and important algorithm problems. These problems were chosen carefully to assist you in developing your muscles while ensuring you practice solving important problems.

# Notion Template: Learn Algorithms By Doing

The goal of this notion template is to make you practice solving algorithm problems in the proper way. You will find in this template:

- **50 Problems**: A set of problems with different levels of difficulty to ensure you apply the techniques that are mentioned in this guide.
- **A Progress Tracker**: To keep you on track and not procrast
- **Resources**: To provide you with information and guidance about solving those problems.

**Scan this QR code to access the notion practice template.**

# What Is The Goal Of This Guide?

The goal of this guide is to help you to:

1. Improve your approach to solving algorithm problems.
2. Learn how to write efficient code that will impress your interviewer.
3. Adopt a better preparation framework that will make the journey enjoyable and fun.
4. Familiarize yourself with the three most crucial algorithm techniques used in problem-solving that will enable you to solve 80% of algorithm problems.

# What Should You Expect After Completing This Guide?

After reading this guide, and completing the practice exercises, you will be able to:

1. Reduce the time for solving algorithm problems.
2. Reduce the stress during coding interviews by following a systematic approach.
3. Have a better impression on your coding interviews.

# 2 | LEARNING CYCLE & ASSESSMENT FACTORS

PHOTO BY AARON BURDEN ON UNSPLASH

# LEARNING CYCLE & ASSESSMENT FACTORS

# Learning Cycle & Interview Assessment Factors

This chapter introduces the three phases of the data structures and algorithm learning cycle and provides insights on how to navigate each phase effectively. The reader will learn about the skills required for each phase and the focus areas during the preparation journey.

Additionally, the chapter highlights the interview assessment factors that interviewers consider when evaluating candidates. Understanding these factors is vital in preparing for interviews and increasing the chances of success. The chapter discusses these factors in detail and offers tips on how to address them positively during the interview process.

By the end of this chapter, the reader will have a clear understanding of these learning cycle phases and how to navigate them effectively. Moreover, they will have insights into the interview assessment factors and how to address them positively during the interview process.

---

## 2.1 Understanding the Learning Cycle

There are three phases that everyone who studies data structures and algorithms has to go through. Each phase represents the learner's skill level during a specific time of their coding preparation journey. In this section, we will address these phases and the things that should be focused on during them.

## 2.1.1 Beginner Phase

### 2.1.1.1 General Characteristics

- You have a good knowledge of programming and are comfortable using a programming language.
- You can solve easy algorithm problems, but you sometimes struggle to solve them. Usually, it takes more than 15 minutes to bring an optimized solution.

### 2.1.1.2 Focus Points

- Review how to read inputs and write outputs using your favorite programming language.
- Review the basic operations of each standard data structure in your favorite programming language.
- Practice solving only easy algorithm problems; 50 questions are enough.
- Focus more on solving array and linked list problems.

**It is better to make sure that those 50 questions are covering all important data structures and algorithms. Here is an example of such a good split that you can apply when you are practicing algorithms:**

1. **Array: 20 problems**
2. **Linked List: 10 problems**
3. **Binary Tree: 10 problems**
4. **String: 10 problems**

## 2.1.2 Experienced Phase

### 2.1.2.1 General Characteristics

- You feel very comfortable solving easy algorithm questions. You do not spend more than 10 minutes to bring an optimized solution.

- Your knowledge of algorithm techniques is excellent.
- You can solve medium-level problems, but it takes you more than 20 minutes to come up with an optimized solution.
- Sometimes, you have some struggles with knowing when to apply algorithm techniques properly.

#### 2.1.2.2 Focus Points

- Focus on solving medium-difficulty algorithm problems.
- Focus on recognizing repetitive patterns and algorithm techniques when tackling algorithm problems.
- Group algorithm problems by similar patterns.
- Convert each pattern group to a solution template you can use if one pattern pops up again in other problems.

**It is better to make sure that these 150 medium algorithm questions are focusing on some data structures and algorithm categories than others. Here is an example of such good distribution:**

1. **Array: 60 problems**
2. **Linked List: 40 problems**
3. **Binary Tree: 40 problems**
4. **String: 40 problems**
5. **Graph: 20 problems**

### 2.1.3 Senior Phase

#### 2.1.3.1 General Characterstics

- You feel comfortable using different algorithm techniques, and data structures.
- You can easily transform a problem into an optimized solution in less than 30 mins.
- The only thing you struggle with is how to stack multiple techniques upon each other to bring a creative solution.

### 2.1.3.2 Focus Points

- Practice solving hard algorithms problems. 100 should be enough in this phase.
- Communicate the solution patterns with other people. This will benefit you by helping you expand your horizon to solve problems in different ways. And it will help others to learn from you.

**In this phase, it is better to focus more on graph and binary tree problems. Here is an example of such good distribution:**

1. **Array: 10 problems**
2. **Linked List: 10 problems**
3. **Binary Tree: 20 problems**
4. **String: 20 problems**
5. **Graph: 40 problems**

**In the notion practicing template, you will find that problems were organized based on their difficulty. There are four groups of problems:**

1. **Starters Level Problems: 10 problems to warm up you.**
2. **Beginner Level Problems: 20 easy problems.**
3. **Medium Level Problems: 20 medium problems.**

**Make sure to practice solving all these problems. They were selected with careful consideration based on their level of difficulty, and their appearance frequency in coding interviews.**

# 2.2 Interview Assessment Factors

If you are preparing for a job interview, it is important to understand the assessment factors that interviewers use to evaluate candidates. By knowing what interviewers are looking for, you can tailor your responses and behaviors to make a positive impression and increase your chances of getting hired. In this section, we will discuss the key assessment factors used by interviewers and provide tips for addressing them effectively during the interview.

---

## 2.2.1 Understanding of Algorithms and Data Structures

Interviewers look for how proficient the candidate is in algorithms and data structures. The more you know about algorithm techniques, the more chance to satisfy this factor.

Usually, the interviewer will test this factor by keeping an eye on you after you come up with a brute-force solution to see if you can optimize this code to run much faster. Let us take this example, an interviewer may ask the following question:

**Write a program that finds the index of a target number in a sorted array.**

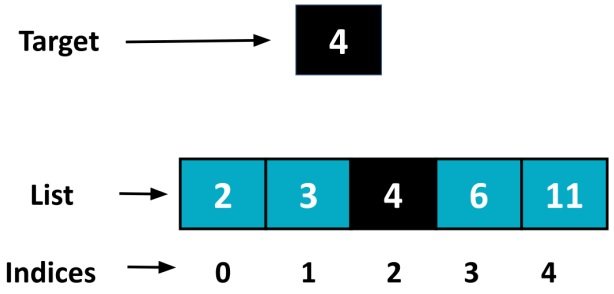



**Figure 2.1: Finding a target number in a sorted array.**

A beginner candidate may answer this question by bringing up a quick brute-force solution. He could say that to find the target number, we need to iterate over the array and check each element if it is equal to the target number. Here is the python solution:

**The brute-force solution of finding a target number in a given array**

```
1  def find_target(target, array):
2    n = len(array)
3    for i in range(n):
4        if array[i] = target: return i
5    return -1 # not found
```

Even though this solution is correct, it is not efficient, because it has ***O(n)*** time complexity, which implies that if finding a target in an array with size 1000 takes us 1 second, an array with a size = 10^6 will take around 1000 seconds. (~16 mintues)

However, an experienced candidate will know this solution can be optimized further. Since the array is sorted, he could propose using the binary search algorithm to find the target number. He knows that the binary search algorithm takes ***O(logn)***, which is faster compared to the ***O(n)*** linear time. (~20 seconds)

Next you can see a sample code of the binary search solution. It may seem longer than the brute-force solution, but it is more efficient, and faster:

```
def binary_search(arr, target):
    # Set the left and right indices of the search range
    left, right = 0, len(arr) - 1

    # Continue searching while the left
    # index is less than or equal to the right index
    while left <= right:
        # Calculate the midpoint of the search range
        pivot = (left + right) // 2
        # If the target is found at the
        # pivot index, return the index
        if arr[pivot] == target:
            return pivot

        # If the target is less than the element
        # at the pivot index, search the left half
        elif target < arr[pivot]:
            right = pivot - 1

        # If the target is greater than the
        # element at the pivot index, search the right half
        else:
            left = pivot + 1

    # If the target is not found, return -1
    return -1
```

From the code, we can notice that code uses recursive function calls to iterate over the array. Using recurisve function calls to iterate over arrays is one of the core technique to apply non-linear traversal to a collection. In chapter 5, we will cover some important techniques that will help you in solving complex algorithm problems.

Also, we should mention that this logrithmic time complexity is a result of the shrinking in size that happens to the array at each recursive call. As the size of the array is divided by two each time the recursive function is called.

## 2.2.2 Problem-Solving Skills

Interviewers will try to evaluate your thought process on solving algorithm problems. It is a big mistake if you start solving an algorithm problem by coding it right away. First, you have to clarify the question then bring some examples to walk your interviewer through your solution, and finally optimize your solution by handling code bottlenecks. As an example, in the last question, it is better to start by asking these questions:

```
Question 1: How big is the input size?

Question 2: Are there NULL values in the array?
```

Also, writing some I/O examples will help you to fully grasp the problem:

```
Example 1:
Input: array [ 1, 2, 4], target = 4,
Output: 2

Example 2:
Input: array [ -1,2,5,6,7], target = 6
Output: 3
```

Later, we will discuss these steps extensively by approaching some algorithm problems using concrete examples.

## 2.2.3 Attention To Detail

Being able to have good attention to detail is important when solving algorithms problems during coding interviews. If we returned to our last

example, we could see that our experience programmer used the binary search algorithm to solve the problem, but why?

That is because there is an important piece of information that is mentioned in the problem description, the word "**sorted**". An experienced programmer will know that by mentioning that the array is sorted he is able to use the binary search algorithm to optimize the code. That is because the binary search algorithm only works when an array is sorted.

Also in low-level languages like C/C++, compared to Python, there is a need to consider the data types used in the implementation. For example, in binary search, using a large data type for the pivot index calculation may cause overflow errors for very large arrays. Similarly, using a floating-point data type may introduce rounding errors and lead to incorrect results.

## 2.2.4 Code Efficiency

Interviewers will pay attention to you when you start coding and defining your variables. They will keep an eye on how you use different data structures. As an example, if you use an array to store elements, but your question have a constraint of having a fast item lookup, then you will fail your interview instantly!

An experienced programmers will know that using a Hashmap is the ideal choice in this situation, as its search operation has ***O(1)*** time complexity which it is much faster than the search operation in an array with ***O(n)*** time complexity.

Also, you have to pay attention to the corner cases in the inputs, i.e., what should your code do if it encounters an empty array. we can use `if len(array) == 0: return -1` to handle a such corner case.

## 2.2.5 Time Complexity Analysis

A crucial part of clearing coding interviews is to provide time and space complexity analysis for your suggested solutions. This is one of the crucial steps, in fact, you can not build efficient code without mastering this part.

**Time Complexity**
: It is the amount of time that an algorithm needs to solve a problem. It is a measure of how long aa program takes to run as the size of the data being processed increases.

**Space Complexity**
: It is the amount of memory space that needs to solve a problem. It is a measure of how much a program takes from the memory as the size of the data being processed increases.

In the next chapter, we will discuss some problems and estimate their time and space complexities.

## 2.2.6 Modular Code

One of the key factors that interviewers try to asses their candidates is their ability to write a modular code. A modular code is breaking down a large program into smaller, self-contained modules or functions, each responsible for a specific task. Using a function or class is one of the ways of writing a modular code.

Writing a modular code will not only show the interviewer that you are an experienced programmer, but also signify that this person has an organized thought process. Also, one of the practical benefits of writing a modular code that it will let your code to be tested during the unit tests phase. The following examples explain using modular codes in practice:

**Bad Code: Not Using Modularization**

```
n = len(array)
    for i in range(n):
        if array[i] = target: return i
```

**Clean Code : Using Modularization**

```
  def find_target(target, array):
    n = len(array)
    for i in range(n):
        if array[i] = target: return i
    return -1 # not found
```

## 2.2.7 Debugging

Solving the problem is not just the end goal of the interview, but knowing how to debug the code if there are errors is also an important factor. You have to make sure that your code is bug-free from semantic and syntax and runtime errors.

**Syntax Error: Passing A List To The (range) Function**

```
for i in range(array):
#       ... Do This ...
```

This is an example of a syntax error, which is a common type of error during interviews. Even though an interviewer does not put much weight on the syntax errors if you missed one or two, it is really important when evaluating candidates and trying to select just one of them.

As we see, the error is in the "range (array)" sentence, it should be corrected like this:

**Clean Code: Passing The List Size To The (range) Function**

```
for i in range(len(array)):
#       ... Do This ...
```

Also, here is an example of a runtime error. In the earlier binary search example, the function was required to be called for the left and right sections of the array from its center. However, an error was made while passing all the elements to the left of the array center, including the middle element (pivot). Such an error would lead to a malfunction of the algorithm during runtime.

### Runtime Error: Dividing By A Single Forward Slash (/)

See in the next binary search code, how dividing by "/" instead of "//" will produce Python's **TypeError: "TypeError: list indices must be integers or slices, not float"** error.

```
def binary_search(arr, target):

    left = 0
    right = len(arr) - 1

    while left <= right:
        # Calculate the midpoint of the search range
        pivot = (left + right) / 2 # <- this will assign a float
                                   # value to the pivot which
                                   # will cause an index
                                   # array later.
        if arr[pivot] == target:
            return pivot

        elif target < arr[pivot]:
            right = pivot - 1

        else:
```

```
            left = pivot + 1

    return -1
```

**Clean Code: Dividing By Double Forward Slash (//)**

```
def binary_search(arr, target):

    left = 0
    right = len(arr) - 1

    while left <= right:

        pivot = (left + right) // 2 # <- this will fix the issue.

        if arr[pivot] == target:
            return pivot

        elif target < arr[pivot]:
            right = pivot - 1
        else:
            left = pivot + 1


    return -1
```

## 2.2.8 Communication

This is by far the most important factor in this list. If you can not communicate your solution to the interviewer, then what is the point of solving the problem in the first place? During interviews, it is essential to make sure to stay connected with your interviewer by communicating in detail what steps

you are willing to do, and why you are doing them, and let the "How" when you code it.

Apply effective communication tells the interviewer you are open to asking for help and making sure to not stuck to deliver your work on time. Using our binary search problem, here is an example of how apply effective communication during a coding interview:

1. Listen carefully to the problem statement and make sure you understand wha is being asked. Ask the interviewer any clarifying questions if necessary.

> **Interviewer**: "Given a sorted array of integers, write a function to search for a specific integer in the array. If the integer is found, return its index. If not, return -1."
>
> **You**: "Okay, so just to confirm, you want me to write a function that takes in a sorted array of integers and a specific integer to search for, and returns the index of that integer if it's found, and -1 if it's not?"

2. Explain your thought process and approach out loud to the interviewer as you work through the problem.

> **You**: "Okay, so since the array is already sorted, I can use the binary search algorithm to find the index of the integer. I'll start by setting the low index to 0, the high index to the length of the array minus 1, and the mid index to the floor of (low + high) divided by 2."

3. Use clear and concise language to explain your ideas and solutions.

> **You**: "So, I'll start by checking if the value at the mid index is equal to the target integer. If it is, I'll return the mid index. If it's less than the target integer, I'll update the low index to be mid + 1 and repeat. If it's greater than the target integer, I'll update the high index to be mid - 1 and repeat."

4. Practice active listening by paying attention to the interviewer's feedback and responding appropriately.

> **Interviewer**: "That sounds good so far. Can you walk me through an example to make sure I understand how it works?"
>
> **You**: "Sure, let's say we have the array `[1, 3, 5, 7, 9]` and we want to search for the integer `5`. We'll start with `low = 0, high = 4, and mid = 2`. Since the value at mid is less than the target integer, we update low to be `mid + 1`, which gives us `low = 3`. Then we update mid to be the floor of (low + high) divided by 2, which gives us `mid = 3`. Now the value at mid is greater than the target integer, so we update high to be `mid - 1`, which gives us `high = 2`. Since low is greater than high, we know the integer is not in the array, so we return `-1`."

# 3 | SOLVING ALGORITHM PROBLEMS

PHOTO BY MAGNEZIS MAGNESTIC ON UNSPLASH

# SOLVING ALGORITHM PROBLEMS

# Solving Algorithm Problems

This chapter discusses the key steps involved in solving algorithm problems. The first section covers essential steps, including understanding algorithms and data structures, problem-solving skills, attention to detail, code efficiency, time complexity analysis, modular code, debugging, and communication.

The second section focuses on applying these factors to solve problems related to the KSum Family. The chapter explores three critical problems in this family, providing step-by-step solutions that emphasize the key concepts discussed in the first section.

After completing this chapter, readers will have a better understanding of how to approach algorithm problems and solve them efficiently.

---

## 3.1 Steps to Solve Algorithm Problems

When attempting to solve an algorithm problem, it is important to adopt a systematic approach that assists in developing a robust solution. This means creating a solution that takes into account various corner cases and is efficient in terms of speed and memory usage. To achieve this, here are the steps to follow when solving an algorithm problem:

### 3.1.1 Understand the Problem

The first step in solving any problem is to understand it. You cannot solve something that you do not understand, and as they say: "Reading the question is half of the answer." It is important to pay attention to the details when encountering an algorithm problem.

## 3.1.2 Formalize the Problem

In this step, the job is to convert the problem information to a single question. This question will be in an input-output format, where one specifies what the input to the problem is and what the expected output is when solving the problem.

## 3.1.3 Repeat Reading the Question Yourself

Repeating a question several times guarantees that one will not miss any hidden information between the lines. In the binary search example, knowing the array is sorted helped in developing an extremely efficient algorithm.

## 3.1.4 Bring Input Examples

After understanding and formalizing the problem as a question, it is time to bring a handful of examples. These examples will serve as the expected inputs and outputs of the developed algorithm. The number of these input examples depends on the person, but it is better to have three examples, each one of them serving a specific goal, as follows:

### 3.1.4.1 Example 1: An Empty-Case Input

The algorithm will expect to receive an empty input such as an empty string, an empty list, or a number with a null value. Bringing this type of input includes these corner cases in the algorithm design process.

### 3.1.4.2 Example 2: A Medium-Case Input

This type of input example is dedicated to testing the algorithm in its most general flow. In other words, these are the inputs that the algorithm will usually deal with. There will be a concrete example of this type of input in the next section.

#### 3.1.4.3 Example 3: A Corner-Case Input

The algorithm will expect to handle some special input examples that the general flow of the algorithm will not expect to see frequently. Examples of such corner cases include:

- Duplicated values in an array when the algorithm should expect to receive unique values.
- Negative inputs when only the algorithm should expect to receive positive inputs.

### 3.1.5 Develop a Brute-Force Solution

Now, it is time to develop a quick, dirty, and not practical solution for the problem. In this stage of solving the problem, there is no need to write an efficient code; only a code that works is required.

### 3.1.6 Analyze Time and Space Complexities For the Brute-Force Solution

After developing the brute-force solution, one has to analyze the time and space complexities. The first reason for this step is that it will tell the interviewer that one knows how much the algorithm will cost in terms of time and space. The second reason is that it will assist in optimizing the code in later stages.

### 3.1.7 Optimize The Brute-Force Solution

This step is the difference between a beginner candidate and an experienced candidate. The interviewer will try to see if you could optimize the brute-force solution and produce a better result. In this stage, you have to go over your brute-force solution line-by-line and look for operations that take too much time and space if the input size becomes very big.

### 3.1.8 Analyze Time and Space Complexities For the Optimized Solution

Again, you have to estimate the time and space complexities of your optimized solution. If you reached this stage, congratulations! That means you have passed your technical interview.

# 3.2 KSum Family Problems

In this section, we will delve into the process of solving algorithmic problems while considering the factors discussed earlier. We will closely examine three essential problems that belong to the KSum Family[1], and walk through their solutions while highlighting these principles.

**KSum** is one of the most asked questions among interviewers, because It contains multiple solution patterns that you can see among almost any other algorithm problem. Also, it allows the interviewers to easily extend their questions from one type of KSum problem to another version of the KSum problem. For these reasons, KSum represents a good example to study closely these assessement factors.

---

## 3.2.1 The 2Sum Problem

### 3.2.1.1 Problem Description

**We are given a list of unique numbers and want to find the index of a number that matches a given target.**

[1] https://bit.ly/uacifds-ksum

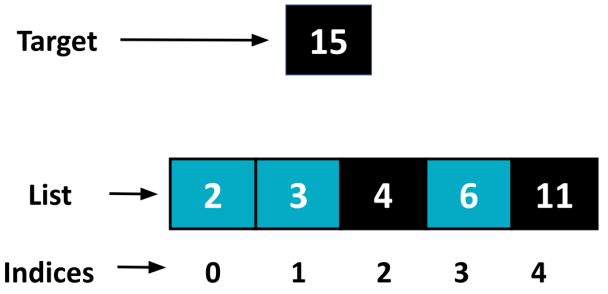



Figure 3.1: 2Sum Problem.

### 3.2.1.2 Problem Understanding

We have a list of integers that have duplicates, and we are interested to find k numbers from this list whose sum will be equal to a target.

Important Keywords: *1. non-unique integers*, *2. unique combination sets.*

### 3.2.1.3 I/O Examples

**In/Out Examples.**

```
Input: list of integers and a target.
Output: A collection of unique sets, where
each set has length = k.

Examples:
---------

Example 1: (Empty-Case Example)
Input: array = [ ], k = 4, target = 0
Output: [ ]

Example 2: (Medium-Case Example)
Input: array = [ 2, 4, 6,9,15 ], k = 2, target = 15
Output: [ [ 6,9] ]

Example 3: (Corner-Case Example)
Input: array = [1, 3, 5, 6, 7, 8, 10, 2, 2, 13], k = 4, target = 15
output: [[1, 2, 2, 10], [1, 2, 5, 7],
        [1, 3, 5, 6], [2, 2, 3, 8],
        [2, 2, 3, 6, 1]
```

### 3.2.1.4 Brute-Force Solution

To solve this problem, we see that we have access to a target. This target represents the sum of two numbers from the given array. Then, a simple solution to this problem will be by making two loop variables **(i, j)**.

Then, we look for a combination of two numbers that add up to the target value. In the following code, we can see the implementation of this algorithm.

**The Brute-force solution of 2Sum problem.**

```python
def two_sum_brute(nums, target):

    n = len(nums)

    for i in range(n):
        for j in range(i + 1, n):  # we start the second loop
                                   # from j = i+1.
            if nums[i] + nums[j] == target:
                return [i, j]
    return []
```

### 3.2.1.5 Complexity Analysis

#### Time Complexity

The time complexity of this solution will be ***O(n²)*** because we used two nested loops, which will result in this time complexity in the worst-case scenario.

#### Space Complexity

However, space complexity is ***O(1)*** which is great, because this solution will always use constant space regarding the size of the input. As we can see, ***O(n²)*** is not a good time complexity, especially if you want to run this code on an array with millions of numbers. In other words, If the algorithm takes 1 second to process 10 numbers, then that means it will take 100,000 seconds (~28 hours) to process 1 million numbers.

### 3.2.1.6 Optimized Solution

To optimize the brute-force code, we need to examine the algorithm line-by-line to see what operations take more resources. Below, we can see a table that outlines the time complexity of the main operations involved in the brute-force solution for the 2Sum problem.

| ID | Operation | Time Complexity |
|---|---|---|
| 1 | **n = len(nums)** | **O(1)** |
| 2 | **for i in range(n):** | **O(n)** |
| 3 | **for j in range(i+1, n):** | **O(n)** |
| 4 | **if nums [i] + nums [j] == target:** | **O(1)** |
| 5 | **return [i,j]** | **O(1)** |

From the table, we can see clearly that our algorithm has a bottleneck because of operations 2, and 3, because in the worst-case senario, operation 3 will have ***O(n.n) = O(n²)*** time complexity.

**How we can remove this bottleneck and optimize these operations?**

One of the most useful data structures that programmers like to use is the hashmap data structure. Hashmap has ***O(1)*** time complexity for the search operation of almost any element inside it. Then, we can work around the inefficiency of the brute-force solution by relying on space complexity.

### Optimization Steps

- Store all the numbers inside a hashmap.
- Loop over the list items, and for each element, we will query our hashmap using a key with a value equal to ***target -nums[i]***, where i is the loop variable, and nums is the list of numbers.

Here is the optimized version of the code:

**The optimized solution of 2Sum problem.**

```
def twoSum_optimized(nums, target):

    mapper = {}
    for (i, e) in enumerate(nums):

       # Store the numbers inside the hashmap,
       # where the keys are the numbers,
       # and the values are the coressponding indicies.

        mapper[e] = i

    for i in range(len(nums)):
        b = target - nums[i]  # get the key value

        if b in mapper and mapper[b] != i:

            # if the key is existed in the hashmap and the its
            # value
            # does not equal to the index of the second number,
            #  we return
            # the indicies.

            return (i, mapper[b])
```

### 3.2.1.7 Complexity Analysis

#### Time Complexity

The time complexity of this solution will be ***O(n)*** because we used only one loop variable to iterate over the array elements.

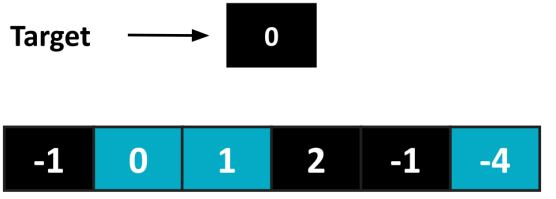
#### Space Complexity

But, space complexity is ***O(n)*** because we rely on the hashmap to increase the time efficiency of our algorithm.

## 3.2.2 The 3Sum Problem

### 3.2.2.1 Problem Description

We are given a list of non-unique integers, and we want to find three numbers that add up to zero. In such a way, the triplets should be all unique.




A Figure Illustrates 3Sum Problem.

### 3.2.2.2 Problem Understanding

We have a list of integers that are non-unique, and we are interested to find three numbers from this list whose sum will be equal to zero.

Important Keywords: *1. non-unique integers*, *2. unique triplets.*

### 3.2.2.3 I/O Examples

**In/Out Examples.**

```
Input: list of integers and a target.

Output: A collection of unique triplets.

Examples:
--------

Example 1: (Empty-Case Example)

Input: array = [ ], target = 0
Output: [ ]

Example 2: (Medium-Case Example)

Input: array = [2,3,4,-1,-2], target = 0
Output: [(-1,-2, 3)]

Example 3: (Corner-Case Example)

Input: array = [-1,0,1,2,-1,-4], target = 0
output: [ [-1,-1,2], [-1,0,1] ]
not [ [-1,-1,21, [-1,0,1], [1,0,-1] ]
```

### 3.2.2.4 Brute-Force Solution

The intuitive approach to solving the problem is making three loops variables **(i, j, k)**. Then, we look for a combination of three numbers that add up to zero.

In the following code lines, notice how the sorting we did initially allowed the Python set "result" to recognize duplicate combinations.

**The Brute-force solution of 3Sum problem.**

```
def three_sum_brute(nums):

    nums = sorted(nums)  # sort nums.
    n = len(nums)
    target = 0
    result = set()

    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):

                combination_sum = nums[i] + nums[j] + nums[k]
                if combination_sum == target:
                    result.add((nums[i], nums[j], nums[k]))

    return result
```

### 3.2.2.5 Complexity Analysis

#### Time Complexity

The time complexity of this solution will be $O(n^3)$ because we used three nested loops. Notice that this is one of the worst time complexities, and the need for the optimization step is mandatory in this case.

#### Space Complexity

Space complexity is $O(n)$ which is fair enough because the code will use a linear space.

### 3.2.2.6 Optimized Solution

The next table outlines the time complexity of the main operations involved in the brute-force solution for the 3Sum problem. Let us examine our brute-force solution line-by-line to see what operations take more resources.

| ID | Operation | Time Complexity |
|---|---|---|
| 1 | **nums = sorted(nums)** | **O(nlogn)** |
| 2 | **n = len(nums)** | **O(1)** |
| 3 | **for i in range(n):** | **O(n)** |
| 4 | **for j in range(i+1, n):** | **O(n)** |
| 5 | **for k in range(j+1, n):** | **O(n)** |
| 6 | **if combination_sum == target:** | **O(1)** |
| 7 | **result.add((nums[i], nums[j], nums[k]))** | **O(1)** |

From the previous table, we can see clearly that our algorithm has a bottleneck because of operations 4 and 5. Because of the triple for-loops, the overall time complexity for this solution cost will cost ***O(n.n.n) = O(n³)***.

**How can we speed up this naive solution, and make it robust in terms of time and memory?**

One of the most useful algorithm techniques to solve algorithm problems is Two Pointers. We are going to solve this problem as follows:

1. Sort the array of integers in increasing order.
2. Create a set to hold the combinations of triplets called "result."
3. While looping over the array using a variable **(i)**:
    - Initialize two pointers, a left pointer with a value equal to i+1 and a right pointer with a value equal to n-1, where n = array size.
    - Make a nested loop and traverse the array using the two pointers. Because we are looking for nums[left] + nums[right] + nums[i]= 0, we will set our target = -nums[i].
    - If nums[left] + nums[right] = target, we will add this to the set of results.

- If nums[left] + nums[right] < target, that means we need to increase the left pointer to increase the value of this summation.
- Finally, if nums[left] + nums[right] > target, it means we need to decrease the right pointer to decrease the value of the summation and push it toward zero.
- In the end, we return the "result."

Here is how we can code this optimized version of the 3Sum problem:

**The optimized solution of 2Sum problem.**

```
def three_sum_optimized(nums):

    nums = sorted(nums)
    n = len(nums)
    result = set()

    for i in range(n):

        left = i + 1
        right = n - 1
        target = 0 - nums[i]

        while left < right:

            combination_sum = nums[left] + nums[right]
            if combination_sum == target:
                result.add((nums[i], nums[left], nums[right]))
                left += 1
                right -= 1
            elif combination_sum < target:

                left += 1
            else:
                right -= 1

    return result
```

### 3.2.2.7 Complexity Analysis

#### Time Complexity

We used two main operations, Sorting ***O(nlogn)*** and Searching ***O(n²)*** (two nested loops). However, the time complexity is still ***O(n²)***, because ***O(nlogn)*** + ***O(n²)*** = ***O(n²)***.

#### Space Complexity

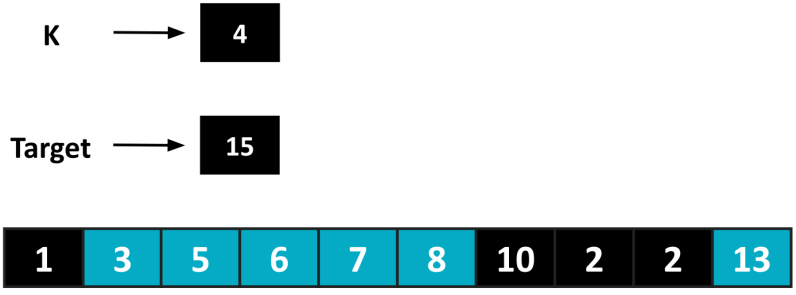
Space complexity is ***O(n)*** because we used the Python set **result** to store the triplets.

## 3.2.3 KSum

### 3.2.3.1 Problem Description

Given a list of non-unique integers and a target, we want to find any set of $K$ numbers from this list that add up to that target, where those sets are all unique.

$$a_1 + a_2 + a_3 + ... + a_k = target$$




A Figure Illustrates KSum Problem.

### 3.2.3.2 Problem Understanding

We have a list of integers that have duplicates, and we are interested to find $K$ numbers from this list whose sum will be equal to a target.

Important Keywords: *1. **non-unique integers**, 2. **unique combination sets**.*

#### 3.2.3.3 I/O Examples

**In/Out Examples.**

```
Input: list of integers and a target.
Output: A collection of unique sets, where
each set has length = k.

Examples:
---------

Example 1: (Empty-Case Example)

Input: array = [ ], k = 4, target = 0

Output: [[ ]]

Example 2: (Medium-Case Example)

Input: array = [ 2, 4, 6,9,15 ], k = 2, target = 15

output: [[ 6,9 ]]

Example 3: (corner-Case Example)

Input: array = [1,3,5,6,7,8,10, 2, 2,13], k = 4, target = 15

Output: [[1, 2, 2, 10], [1, 2, 5, 7],
        [1, 3, 5, 6], [2, 2, 3, 8], [2, 2, 5, 6]]
```

#### 3.2.3.4 Solution

As you can see in 3Sum, even though we have duplicates in the array, our code should handle such a situation by keeping all the sets unique. Also, we provide different ***K*** values. This is important later when you test your code against different KSum problems.

This problem is the general form of the 2Sum problem. To be able to infer the KSum solution from our previous solutions, we need to understand a very important thing first. Any KSum problem can be solved with this simple idea:

**"Reduce KSum to 2Sum problem, then solve it."**

First, let us define some terms:

$$t_n = \text{The uppermost n degree target}$$

$$K = \text{The degree of KSum problem}$$

Mathematically, the 2Sum problem could be formalized as:

$$a_1 + a_2 = t_1$$

While 3Sum is formalized as:

$$a_1 + a_2 + a_3 = t_2$$

And 4Sum is formalized as:

$$a_1 + a_2 + a_3 + a_4 = t_3$$

Then to solve 4Sum, we can rearrange these equations as follows:

$$t_2 = t_3 - a_4$$

$$t_1 = t_2 - a_3$$

$$a_1 = t_1 - a_2$$

**Then, here is the summary of the KSum algorithm**:

- Starting with the uppermost degree target

$$K = n$$

$$t = t_n$$

- Recursively, calculate the upper-degree targets (i.e $t_3$, $t_2$ ) and the missing numbers (i.e. $a_4$, $a_3$).
- When $K = 2$, call the 2Sum function to calculate the last missing numbers ( $a_1$, $a_2$ ).

To code **KSum** in Python, we defined two functions, we will introduce two functions. The first function handles the 2Sum operation, while the second function handles the KSum operation.

**The 2Sum function:**

**2Sum using Two Pointers technique**

```
def two_sum_2pointers(
    nums,
    target,
    path,
    result,
    ):

    left = 0
    right = len(nums) - 1

    while left < right:

        if nums[left] + nums[right] == target:
            result.append(path + [nums[left], nums[right]])

      # ----------- To filter the duplicate cases ---------

            while left < right and nums[left] == nums[left + 1]:
                left += 1
            while left < right and nums[right] == nums[right - 1]:
                right -= 1

      # -----------------  End of Filtering ----------------
```

```
24
25              left += 1
26              right -= 1
27
28          if nums[left] + nums[right] < target:
29              left += 1
30
31          if nums[left] + nums[right] > target:
32              right -= 1
```

As you can see, we use the Two Pointers technique to implement this version of 2Sum. In chapter 5, we will learn more about this technique in detail.

**The KSum function:**

**KSum using Two Pointers technique**

```
1  def k_sum(
2      nums,
3      k,
4      target,
5      path,
6      result,
7      ):
8      # stop early if the number of summation variables is less
9      # than 2 or greater than the length of input array (nums).
10     if len(nums) < k or k < 2:
11         return
12
13     if k == 2:  # Call 2Sum_2pointers function when degree (k) = 2
14         two_sum_2pointers(nums, target, path, result)
15     else:
16
17     # recursively reduce k (degree) value and estimate
18     # upper-degree targets (target-nums[i]) till we reach
19     # k =2 to solve it as a two_sum problem.
20
```

```
        for i in range(len(nums)):
        # remove the duplicates in nums.
            if i == 0 or i > 0 and nums[i - 1] != nums[i]:
                k_sum(nums[i + 1:], k - 1, target - nums[i], path
                      + [nums[i]], result)
```

The ***k_sum*** function keeps calculating the required numbers by reducing the value of k by one till ***K = 2***. Then it calls the ***two_sum_2pointers*** function to calculate the last two missing numbers.

### 3.2.3.5 Complexity Analysis

#### Time Complexity

The time complexity of this solution will be:

$$O(n^{k-1})$$

because we used recursive function calls to solve for the k elements.

#### Space Complexity

In general, the code will have a space complexity equal to ***O(n)*** which represents the space needed to store the results. ( ignoring the function stack memory and the path variable)

# 4 | FGCC FRAMEWORK

PHOTO BY SHARON PITTAWAY ON UNSPLASH ON UNSPLASH

# FGCC FRAMEWORK

# 4.1. FGCC Framework

In this chapter, we introduce the FGCC Framework, a powerful tool for simplifying coding interview preparation. FGCC stands for "Focus, Group, Convert, and Communicate," and it aims to enhance the effectiveness and enjoyment of the preparation process. It is based on the idea of leveraging existing solutions as a baseline and then customizing them to solve algorithm problems. By following this framework, you can approach algorithm problems with efficiency and effectiveness.

Throughout this chapter, we examine three problems that demonstrate the power of the FGCC framework. By understanding and applying these patterns, you can solve a wide range of algorithm problems with ease. The FGCC framework provides a structured and systematic approach to coding interview preparation, allowing you to make the most of your time and excel in your interviews.

---

## 4.1.1 What is FGCC?

FGCC stands for "Focus, Group, Convert, and Communicate". It is a framework that I developed during my preparation for coding interviews. The goal of the framework is to simplify the preparation for coding interviews by making them more enjoyable and effective. The intuitive idea of FGCC comes from my work as an applied machine learning scientist, where I often face the challenge of training machine learning (ML) algorithms with limited examples, particularly in projects involving Natural Language Processing (NLP) and BERT.

**Machine Learning Algorithm**
is a group of weights that are updated during a training process using one or more datasets.

**Machine Learning Model**
is a trained machine learning algorithm.

## 4.1.2 A Mental Framework

The basic idea is that you do not have to start training your machine learning algorithm from scratch. You can use a baseline model trained on public datasets (e.g., Wikipedia articles) and then fine-tune it (warm start). An example of such a baseline model is GPT-3, and ChatGPT is a model built upon the GPT-3 baseline.

Applying the same analogy to our preparation for coding interviews, it can be summarized as follows:

1. First, establish a mental baseline by reviewing other people's solutions to various algorithm problems.
2. Second, develop your own customized model to solve any algorithm problem that may arise during interviews.

## 4.1.3 Steps To Apply FGCC Framework

1. **Focus (F)**: Focus on recognizing repetitive patterns when tackling algorithm problems.
2. **Group (G)**: Group those problems with similar patterns.
3. **Convert (C)**: Convert each pattern group into a solution template that can be reused when similar patterns appear in other problems.
4. **Communicate (C)**: Share these patterns with others.

The steps of Focus (F) and Group (G) help build the mental baseline, while Convert (C) and Communicate (C) contribute to the creation of a tailored baseline. Later, we will discuss important algorithm templates that are necessary for solving algorithm problems.

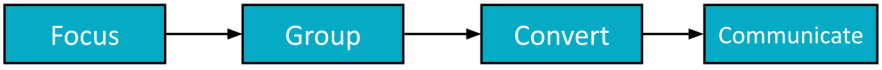



**Figure 4.1: The FGCC Framework Components.**

# 4.2. FGCC In Practice

## 4.2.1 Introduction to Backtracking Technique

During coding preparation, you may come across several algorithm problems. Initially, you might notice that these problems are different from each other and do not follow any specific pattern. This may lead you to think about developing a completely new solution for each problem. However, after spending some time practicing, you will start to notice an interesting thing: these problems share one or more common elements. They exhibit patterns either in the problem definition or in the solution approach. Once you begin recognizing these patterns, the process of solving other problems will be shortened, as you can reuse these patterns to build and invent new solutions.

To get an idea of how the FGCC framework works, we will discuss three problems and see how they share a very interesting pattern that can be used to solve similar problems.

---

### 4.2.1.1 Finding Permutations

Finding permutations is one of the most popular algorithm problems among interviewers. The problem is formalized as follows:

**We are given a list of integers and want to find all the possible permutations that can be formed from this list**.

Here is a quick example of how the expected inputs, and outputs for this problem:

```
Input: nums = [1,2,3,4]
Output: [[1, 2, 3, 4],[1, 2, 4, 3],
        [1, 3, 2, 4],[1, 3, 4, 2],
        [1, 4, 2, 3],[1, 4, 3, 2],
        [2, 1, 3, 4], [2, 1, 4, 3],
        [2, 3, 1, 4],[2, 3, 4, 1],
        [2, 4, 1, 3],[2, 4, 3, 1],
        [3, 1, 2, 4], [3, 1, 4, 2],
        [3, 2, 1, 4], [3, 2, 4, 1],
        [3, 4, 1, 2], [3, 4, 2, 1],
        [4, 1, 2, 3], [4, 1, 3, 2],
        [4, 2, 1, 3],[4, 2, 3, 1],
        [4, 3, 1, 2],[4, 3, 2, 1]]
```

## Solution

In math class, we learned that the number of possible permutations of n objects is calculated with this formula:

$$P_k^n = \frac{n!}{(n-k)!}$$

In this problem, k = n. The easiest way to think about this problem is by visualizing permutations as a tree, where each path in the tree represents a valid permutation. To solve this problem, we traverse the list of integers and visit all the paths in the tree. In other words, we are interested in performing a tree traversal on the list of integers to find all the valid paths that represent our desired permutations.

Let us take the previous example, suppose that we are given the list nums = [1,2,3,4], and we aim to find all the possible permutations from this list. If we apply the tree analogy to this example, each subtree will represent a permutation set, and each tree path will represent a possible permutation. The figure below represents the permutation set that starts with the number '1'.

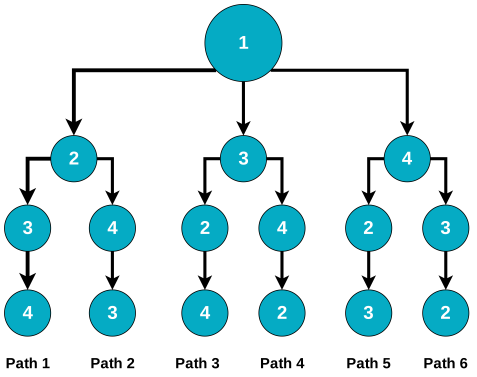



**Figure 4.2: The permutations of an array num = (1,2,3,4).**

If we traverse this tree using the Depth-First Search algorithm (DFS), the paths will be as follows: [1,2,3,4], then [1,2,4,3], then [1,3,2,4], then [1,3,4,2], then [1,4,2,3], and finally [1,4,3,2], which are considered valid permutations. The following code demonstrates the use of a "loop" operator to traverse the list of numbers horizontally and a "recursive function call" to traverse them vertically. First, we move one step in the horizontal direction, taking the next element in the list as a root (i.e., 1 in our example) to the next subtree and store it in the "path" variable. Second, we proceed in the vertical direction to build this next subtree. Here is the full code:

```
def permute_dfs(nums):
    def dfs(nums, path, result):
        # A base condition to stop the recursion.
        if len(nums) <= 0:
        # When the there are no more items in the
        # list that belong to each node, the
        # recursion will stop.
            result.append(path)
            return

        # loop over the nodes that belong to each level
        for i in range(len(nums)):
        # Call dfs function, while saving the
        # nodes of each path in a list called path.
```

```
            dfs(nums[0:i] + nums[i + 1 :], path + [nums[i]], result)

    result = []
    path = []

    dfs(nums, path, result)

    return result
```

You can find more explanation to this problem in this article[1].

### 4.2.1.2 Finding Combinations

This problem is similar to the permutation problem, but instead of considering all the permutations as outputs, we focus on generating subsets with pre-defined lengths. Here is the problem definition:

**We are given a list of integers and we want to find all the possible combinations of k numbers, where k represents the number of chosen objects from the list.**

Here is a quick example of how the expected inputs, and outputs for this problem:

```
Input: nums = [1,2,3,4], k = 2

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

The solution will be similar to the permutation solution, except that we will implement backtracking when the length of our path equals k.

**Backtracking is a technique used to halt the execution of a recursive algorithm and consider partial solutions due to certain constraints.**

[1] https://blog.waleedkhamies.com/uacifds-unfold-permutation-family-problems-38da375236e3#0a93

Backtracking often goes hand in hand with recursion. However, to apply the backtracking technique, we need three components:

1. **Goal (Base Condition)**: We require a goal that stops our recursion.
2. **Constraints**: We need a set of constraints that prevent us from considering certain solution paths.
3. **Options**: We need a set of options to choose from.

Now, let us return to our previous example. Our goal is to stop when we have found all possible combinations. Simultaneously, we impose a constraint by terminating our search when we reach a specific path length (len(path) = k).

At each step, we have one option available, which we select by moving vertically and taking the next sublist to construct the subsequent subtree. The following figure illustrates all combinations for this example:

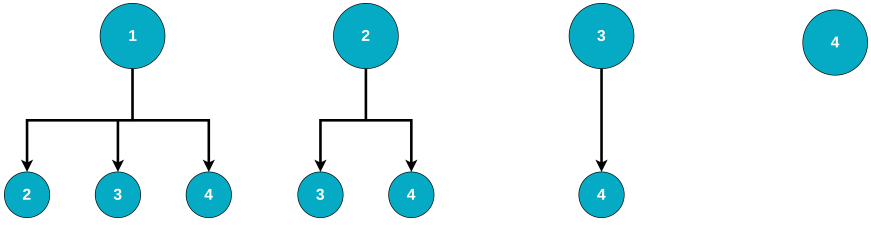



**Figure 4.3: The combinations of an array num = (1,2,3,4) where k =2.**

Here is the combination code sample:

**Applying DSF + backtracking to find all the combinations in a list of numbers.**

```
def combine(n, k):
    def dfs(nums, path, result):
        # base condition: just return
        # paths that have length k.
        if len(path) == k:
            result.append(path)
            return

        for i in range(len(nums)):

            dfs(nums[i + 1 :], path + [nums[i]], result)
```

```
12
13      nums = [i for i in range(1, n + 1)]
14
15      result = []
16
17      path = []
18
19      dfs(nums, path, result)
20
21      return result
```

### 4.2.1.3 Letter Combinations of a Phone Number

Before we explain FGCC pillars in these examples, we will discuss one more algorithm problem.The "Letter Combinations of a Phone Number" problem is a coding challenge that involves the generation of all possible letter combinations derived from a given string of digits representing a phone number. In this problem, each digit on the phone keypad corresponds to a specific set of letters. The objective is to systematically explore and generate all feasible combinations of letters by appropriately pressing the phone keypad buttons in the given order.

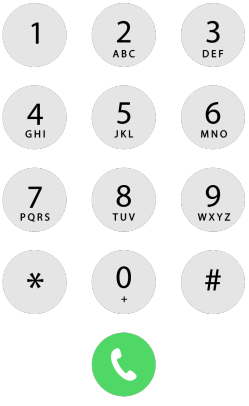



**Figure 4.4: Phone Keypad.**

**Given that we can access a hashmap that maps each number to a string. Find all the letter combinations we can make if we are provided with a phone number.**

Let us see a quick example of the expected inputs, and outputs for this problem:

**I/O Example.**

```
1 Input: digits = “23”
2 Output: [“ad”,”ae”,”af”,”bd”,”be”,”bf”,”cd”,”ce”,”cf”]
```

As we have noticed, the problem description mentions an interesting aspect, the term “combinations.” This indicates that we can utilize our solution for the previous problem to solve this one.

The backtracking technique can be applied to find these combinations by following these steps:

1. Create an integer variable called “index” to keep track of the position in the digits list.
2. Create a string variable called “path” to track each valid path that the algorithm explores.
3. Create a list called “result” to collect the combinations.
4. Call the function “lettersCombinations_dfs,” which constructs a tree and stores each path in the result list.

Here is the Python code:

**Applying DFS + backtracking to find all the letter combinations of a phone number.**

```
def letterCombinations(digits):
    hashmap = {
        "1": "None",
        "2": ["a", "b", "c"],
        "3": ["d", "e", "f"],
        "4": ["g", "h", "i"],
        "5": ["j", "k", "l"],
        "6": ["m", "n", "o"],
        "7": ["p", "q", "r", "s"],
        "8": ["t", "u", "v"],
        "9": ["w", "x", "y", "z"],
            }

    def dfs(digits, index, dic, path, result):

        if index >= len(digits):
            result.append(path)
            return

        string = dic[str(digits[index])]

        for ch in string:
            dfs(digits, index + 1, dic, path + ch, result)

    if len(digits) <= 0:
        return []

    result = []
    index = 0
    path = ""
    dfs(digits, index, hashmap, path, result)

    return result
```

## 4.2.2 The Pillars of the FGCC Framework

Now, let us explore the main pillars of the FGCC framework and understand how each pillar can be applied to our problem-solving process.

### 4.2.2.1 Focus (F)

**Focus on recognizing repetitive patterns when tackling algorithm problems**.

From the problems discussed earlier, we observed a recurring pattern. This pattern can be identified in two aspects: the problem definition and the solution.

#### 1. Problem Formulation Pattern

All the previous problems shared a common structure in their problem formulation, which includes the phrase "Find all the ..." followed by additional details. Recognizing this pattern is key to identifying problems that can be solved using the backtracking technique.

#### 2. Solution Pattern

Upon reviewing the previous code blocks, you will notice certain lines that repeat a pattern. Let us find that for each previous problem:

#### A. Finding Permutations

**DFS + Backtracking in permutation problem.**

```
#               ...
    if len(nums) =< 0: # A base condition
        result.append(path)
        return

    # loop over the nodes that belong to each level
    for i in range(len(nums)):
        dfs(nums[0:1] + nums[i + 1 :] , path + [nums[i]], result)
#               ...
```

ctually, the permutation problem is a good example of how the DFS technique is applied to non-graph problems.

### B. Finding Combinations

```
#               ...
    if len(path) = k: # base condition:
    # just return paths that
    # have length k.
        result.append(path)
        return

    for i in range(len(nums)):

    dfs(nums[i + 1 :], path + [nums[i]], result)
#               ...
```

### C. Letter Combinations of a Phone Number

```
#               ...
    if index = len(digits):
        result.append(path)
        return

    string = dic[str(digits[index])]

    for ch in string:
        dfs(digits, index + 1, dic, path + ch, result)
#               ...
```

#### 4.2.2.2 Group (G) & Convert (C)

**Group the problems by similar patterns.**

**Convert each pattern group to a solution template you can use if one pattern pops up again in other problems.**

The pattern we have noticed in the previous problems can be generalized to represent the backbone of the backtracking technique. Here is what the generic code looks like:

**DFS + Backtracking code pattern.**

```
#               ...
    if "Stop Condition": # A base condition
        result.append(path)
        return

    # loop over the nodes that belong to each level
    for i in range(len(collection)):
        dfs(collection, path + [collection[i]], result)
#               ...
```

#### 4.2.2.3 Communication

**Communicate those patterns with other people**.

The final pillar of the FGCC framework is communication. It is important to communicate the solution templates and patterns with others. This not only benefits the larger community but also has several personal advantages:

1. It improves your communication skills.
2. It enhances your ability to organize your thoughts and ask insightful questions.
3. It helps you develop your writing skills.

By sharing your knowledge with others, you are not only contributing to the community but also benefiting yourself in various ways.

# 5 | TOP #3 ALGORITHM TECHNIQUES

# TOP #3
# ALGORITHM TECHNIQUES

# 5. Top #3 Algorithm Techniques

There are three algorithm techniques that you can use to tackle a wide range of algorithm challenges. With these techniques, you can traverse different data structures in a linear and non-linear way, which provides you with more capability in finding creative solutions for complex problems. These techniques include:

- Two Pointers Approach.
- Breadth-First Search Approach (BFS).
- Depth-First Search Approach (DFS).

In this section, we will discuss them briefly and highlight some of the important aspects that you should take into account when you apply them.

---

## 5.1 Two Pointers

The two-pointer approach is considered one of the most popular techniques when you have a search operation in a list (sorted or not) or a collection. The basic idea of this approach is to use two variables instead of one loop variable (pointer) to traverse a list or collection. As you can see from the next image, we defined two variables, i and j, and set them to point at the beginning and the end of the array.

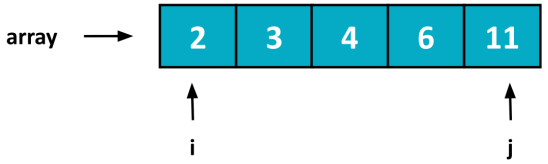



**Figure 5.1: Applying Two Pointers (i, j) on array data structure.**

You can set the pointers to point to any location in the array, but the most important thing is to make sure that selecting these initial locations will help you make solving the problem a lot easier than choosing other locations.

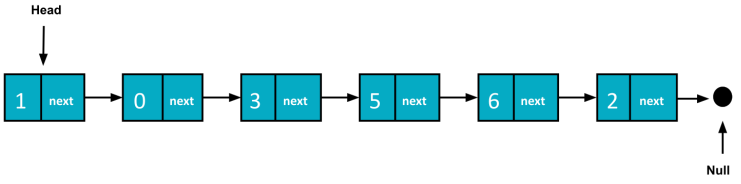



**Figure 5.2: Applying Two Pointers (slow, fast) on linked list data structure.**

## 5.1.1 Code Example

As an example, if an interviewer asks you to write a program that reverses an array, it would be a lot easier to solve the problem if you set two pointers to point to the beginning and the end of the array, compared to other locations.

Additionally, some problems may require you to apply preprocessing to the inputs before using the Two Pointers technique. One of the best examples in

this case is the 2Sum problem that we have encountered in earlier chapters. You can solve the 2Sum problem using the Two Pointers technique; however, you need to first sort the array before applying it. Here is an example of how we can solve 2Sum using this technique:

**Solving the 2Sum problem using Two Pointers Technique.**

```
def two_sum_2pointers(
    nums,
    target,
    path,
    result,
    ):

    left = 0
    right = len(nums) - 1

    while left < right:

        if nums[left] + nums[right] == target:
            result.append(path + [nums[left], nums[right]])

      # ----------- To filter the duplicate cases ---------

            while left < right and nums[left] == nums[left + 1]:
                left += 1
            while left < right and nums[right] == nums[right - 1]:
                right -= 1

      # -----------------  End of Filtering ----------------

            left += 1
            right -= 1

        if nums[left] + nums[right] < target:
            left += 1

```

```
31          if nums[left] + nums[right] > target:
32              right -= 1
```

In this example, the `left` and `right` variables represent the two pointers. As you can see, we set the `left` and `right` pointers to point at the beginning and the end of the array, respectively. Then, we compare the target with the sum of the numbers associated with the two pointers. If the sum is less than the target, we increase the `left` pointer and move to the next number. If the sum is not less than the target, we decrease the `right` pointer and move it to the previous number.

### 5.1.2 Usage

The Two Pointers technique is mainly used when you want to speed up certain operations on an array. It is commonly utilized in conditional traversal problems where you need to iterate over an array based on certain conditions, such as finding a target (e.g., 2Sum problem) or finding the largest container area (container with the most water problem).

### 5.1.3 Data Structures

The Two Pointers technique is often used with the following data structures:

- Array.
- LinkedList.

## 5.2 Breadth-First Search (BFS)

Breadth-first search (BFS) is an algorithm used to traverse a graph or tree data structure. It explores all the neighboring vertices or nodes before moving on to the next level. The algorithm starts from the root node and traverses the tree level by level, exploring all the neighboring nodes before moving on to the next level.

Let us consider an example to understand how breadth-first search works. Suppose you want to find the shortest path from your home to your office in downtown Montreal. You can utilize the BFS algorithm, where your home serves as the root node, and the neighboring nodes represent the streets that connect your home to your office. Here is a figure illustrating this example:

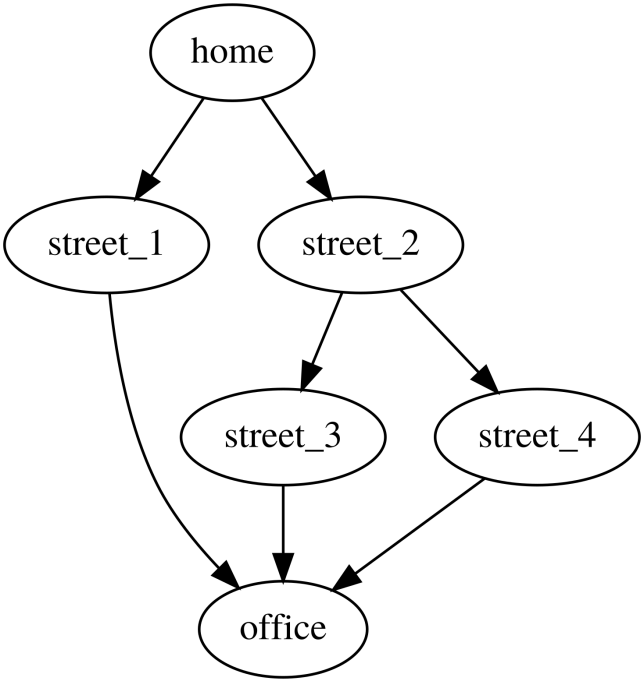



**Figure 5.3: Finding the shortest path between the home and the office.**

Then, by exploring all the neighboring streets first and then moving on to the next level, you will eventually find the shortest path to your office.

## 5.2.1 Code Example

Let us move on to the coding part of BFS. The BFS algorithm can be implemented using a queue data structure, where the nodes are added to the queue as they are visited. In the following code block, we define our graph as a Python dictionary, where each key represents a node, and its value represents the street that is close to or branching from that node:

```
graph = {
'home': {'street_1', 'street_2'},
'street_1': {'office'},
'street_2': {'street_3', 'street_4'},
'street_3': {'office'},
'street_4': {'office'},

'office': set()
```

Now, here is a simple code that applies the BFS technique to the previous graph:

```
def BFS(graph, home, office):
    queue = [(home, [home])]
    while queue:
        (vertex, path) == queue.pop(0)
        for neighbor in graph[vertex]:
            if neighbor not in path:
                if neighbor == office:
                    return path + [neighbor]
                else:
                    queue.append((neighbor, path + [neighbor]))

            return None
```

As you can see, our queue holds a collection of a parent node and the current path. We skip adding the node to the path if we have already seen it, and we return the path variable if we reach our target, the office.

## 5.2.2 Usage

As mentioned earlier, breadth-first search is used when you want to traverse a tree or graph. It guarantees finding the shortest path from the root node to any other node.

### 5.2.3 Data Structures

The BFS algorithm is usually used with the following data structures:

- Tree.
- Graph.
- Matrix.

Next, we will explore another version of a traversal algorithm that is commonly used with trees and graphs. For more information about this technique, you can check out this article[1].

## 5.3 Depth-First Search (DFS)

Depth-first search (DFS) is a popular algorithmic technique used for traversing graphs and trees. It is particularly useful for exploring collections in a non-linear manner. The DFS algorithm begins at the root node of a tree and explores each branch as deeply as possible before backtracking. It employs a stack data structure to keep track of visited nodes and those yet to be visited. The LIFO (Last-In-First-Out) behavior of the stack enables efficient traversal of nodes.

To illustrate this concept, let us consider a scenario where you are searching for your phone within your house. Your house comprises three rooms: the living room, bedroom, and kitchen. We can represent this as a graph in Python as follows:

[1] https://www.educative.io/answers/how-to-implement-a-breadth-first-search-in-python

```
graph = {

"living room': {'sofa', 'coffee table', 'rug', 'bedroom'},
'sofa': {'remote', 'phone'},

'coffee table': {'phone charsger'},

'rug': set(),

'bedroom': {'phone', 'pillow'},

'phone': set(),

"phone charger': set(),

"pillow': set(),

'kitchen': {'fridsge', 'pantry', 'living room'},
'fridge': set(),

"pantxy': {'phone charger'}
```

And here is the visual illustration of this graph:

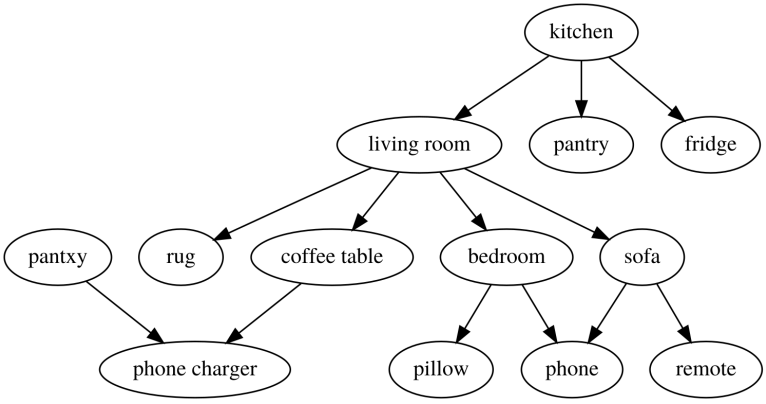



**Figure 5.4: The graph representation of the house.**

Now, to find the missing phone, we can apply the DFS algorithm to this simple graph. Here is the sample code:

```
def dfs(current_room, searched_room, visited, rooms):
    visited.add(current_room)

    if current_room == searched_room:
        return True

    for neighbor in rooms[current_room]:

        if neighbor not in visited:

            if dfs(neighbor, searched_room, visited, rooms):
                return True

    return False
```

DFS will traverse through all the rooms, starting from the living room, and visit all the items in the living room. It will then continue exploring the associated components with each item, such as the bedroom with its phone and pillow.

### 5.3.1 Usage

The Depth-First Search algorithm is commonly used when you need to perform a non-linear search in a collection, such as graphs or trees. It allows you to explore deeply into the structure before backtracking.

### 5.3.2 Data Structures

The DFS algorithm is typically used with the following data structures:

- Tree
- Graph
- Matrix

For more information about this technique, you can refer to this article[2] and this article[3].

[2] https://brilliant.org/wiki/depth-first-search-dfs/
[3] https://blog.waleedkhamies.com/unfold-binary-tree-family-problems-part-1-2-e986049a1fb2

# 6 | SUPPLEMENTS

# 6. Supplements

In this supplemental chapter, we will explore some problems related to linked lists and binary trees. The goal of these problems is to deepen your understanding of the concepts and techniques discussed in this book while providing guidance on how to apply these techniques to the practice problems available on the Notion template.

We will discuss the following problems:

1. **Detect a Linked List Cycle**: To identify whether a linked list contains a cycle using the fast-slow pointer technique.
2. **Remove the Nth Node From the End of a Linked List**: To remove the nth node from the end of a linked list. We will discuss different approaches using two pointers and explain their trade-offs.
3. **Swapping Linked List Node Pairs**: To swap adjacent nodes in a linked list. We will explore strategies and discuss implementation details.
4. **Validate Binary Tree**: To check if a binary tree satisfies specific properties, like the binary search tree (BST) property. We will discuss algorithms and analyze their time and space complexities.
5. **Same Binary Tree**: To determine if two binary trees are identical. We will explore recursive and iterative approaches to compare binary trees for structural and value equality.
6. **Symmetric Binary Tree**: To check if a binary tree is symmetric. We will discuss techniques to determine symmetry and efficient problem-solving strategies.

Throughout this chapter, we will provide explanations of the problems and their optimized solutions. We will go step-by-step, highlighting some important techniques and providing tips for solving similar problems.

By the end of this chapter, you will have the opportunity to practice these problems and enhance your understanding of linked lists and binary trees.

---

# 6.1 Detect a Linked List Cycle

The Linked List Cycle problem presents the challenge of identifying whether a given linked list contains a cycle. When a cycle exists in a linked list, one of the nodes points back to a previous node, forming an internal loop. The goal is to determine whether such a cycle exists in the given linked list or not.

## 6.1.1 Problem Description

**Given a head of a linked list that may have a cycle, we have to check if it contains a cycle or not.**

## 6.1.2 I/O Examples

**Input 1:** ***head***

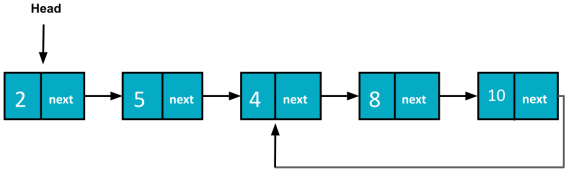



**Figure 6.1: Example 1.**

**Output: True**

**Input 2:** ***head***

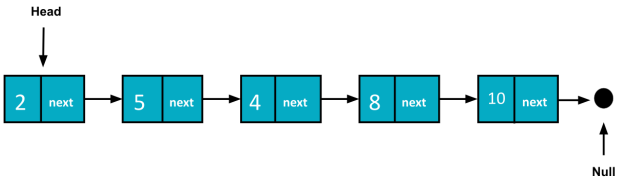



**Figure 6.2: Example 2.**

**Output: False**

## 6.1.3 Solution

The solution to this problem is very intuitive to us. Imagine that you are passing through a street, and you suspect that you have visited this street before. The easiest way to know if this is true or false is by making a sign—let us say—in one of the street buildings. Then, if you see this sign again, you can confirm that you are stuck in a loop.

Similarly, to check if a linked list has a loop or not, we will use the **Man-Sign** analogy. The **sign** here will be a pointer called ***slow***, and the walking **man** will be a pointer called ***fast***. The fast-pointer will move with steps equal to twice the steps of the slow-pointer. Then, if there is a cycle, the fast-pointer will meet the slow-pointer in one of the nodes. Here is the full code:

```python
def list_has_cycle(head):
    if not head:
        return False

    fast_p = head.next
    slow_p = head

    while fast_p:

        if slow_p == fast_p:
            return True
        fast_p = fast_p.next.next
        slow_p = slow_p.next

    return False
```

### 6.1.4 Complexity Analysis

#### Time Complexity

The time complexity of this solution will be ***O(n)***, because in the worst-case scenario the cycle will be between the first and the last nodes.

#### Space Complexity

The space complexity will be ***O(1)*** as there is no extra space used.

## 6.2 Remove the Nth Node From the End of a Linked List

The goal of this problem is to remove the ***nth*** node from end of list. This requires efficiently identifying and removing a node that is ***n*** positions away from the end of a given linked list.

## 6.2.1 Problem Description

Given the head of a linked list, and a number ***n*** that represents the index of the node that should be deleted from the end of the list. Remove the node, and return the head of the linked list.

## 6.2.2 I/O Examples

Input : ***head***, n=3

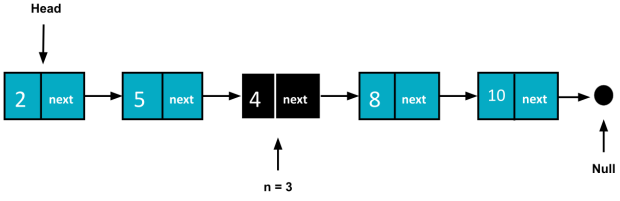



Figure 6.3: Input Sample.

Output: ***head***

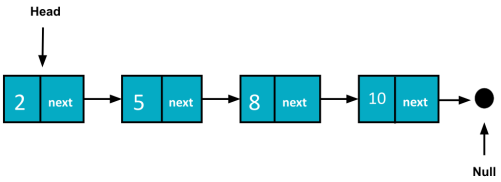



**Figure 6.4: Output Sample.**

## 6.2.3 Solution

To solve this problem, we will define two pointers, slow, and fast. Then, the idea is to make the distance between the two pointers equal to ***n***. That can be achieved as follows:

1. Initialize two pointers *slow*, and *fast* and set them to point to the linked list head.
2. Move the fast pointer *n* steps.
3. Then, while the *fast* pointer does not point to *Null*, move the slow, and the fast pointers one step at each time.
4. Then, if the fast pointer reaches the end of the list, where it points to nothing (*Null*), we remove the *nth* element.

Because the slow pointer at this time step will be before the ***nth*** node (at the node with ***n = n-1***). Here is the full code:

```python
def remove_nth_node(head, n):

    i_slow = head
    j_fast = head

    # first we will move the fast pointer
    # so that the distance between
    # j_fast - j_slow = n

    for _ in range(n):
        j_fast = j_fast.next

    if not j_fast:  # return the next node to
    # the slow pointer, if the the number
    # of nodes are less than n.
        return i_slow.next
    while j_fast.next:
        j_fast = j_fast.next
        i_slow = i_slow.next

    i_slow.next = i_slow.next.next

    return head
```

## 6.2.4 Complexity Analysis

### Time Complexity

The time complexity will be ***O(n)***, because in the worst-case scenario, we must remove the node with ***n = 1***.

### Space Complexity

The space complexity will be ***O(1)***, because there is no extra space used during the iteration process.

# 6.3 Swapping Linked List Node Pairs

Imagine that you are waiting in a line in one of ***Tim Horton's*** restaurants. This day, the restaurant has a special offer for customers who come for a pickup or dine-in.

So, they asked their customers who stand in the waiting line to change their places, in a way that any two neighbor customers should swap their locations. In exchange, any swapped pairs will get a free drink each.

## 6.3.1 Problem Description

**Given the head of a linked list, swap any adjacent node pair.**

## 6.3.2 I/O Examples

**Input 1:** ***head***

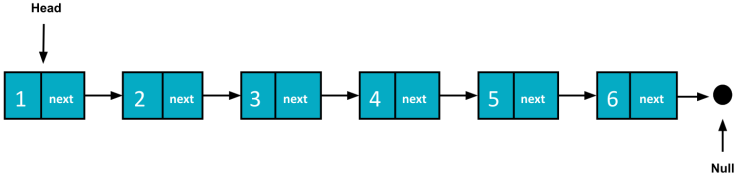



**Figure 6.6: Input Sample.**

**Output:** ***head***

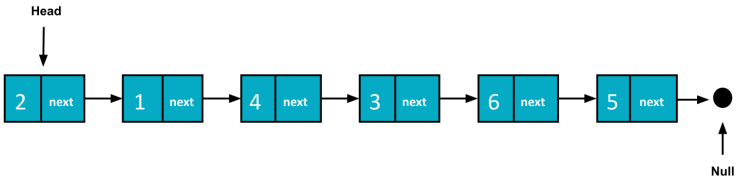



Figure 6.7: Output Sample.

## Input 2: *head*

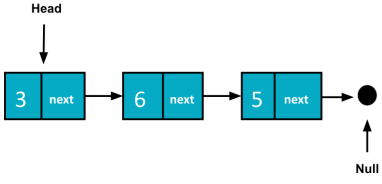



Figure 6.8: Input Sample.

## Output: *head*

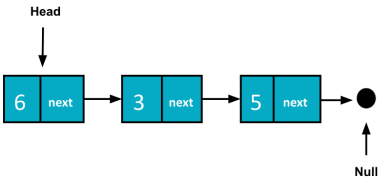



**Figure 6.9: Output Sample.**

## 6.3.3 Solution

To help our friends in the line to do the swapping, let us take the previous example and try to reverse it back to its normal state as follows:

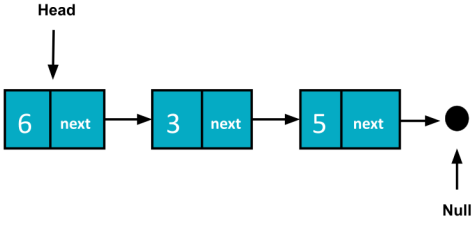



**Figure 6.10: Swaping node pairs.**

To swap **node(6)** and **node(3)**, we will:

1. Create a dummy node and set its next pointer to the head of the linked list.
2. Initialize three pointers ***prev***, ***current***, and ***nextp***.
3. Set the ***prev*** pointer to point to the dummy node.

4. Set the ***current*** to point to node(6) (the linked list head).
5. Set the ***nextp*** pointer to point to node(3).

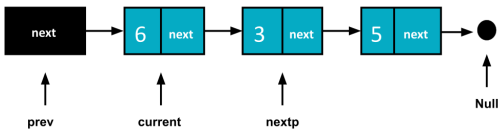



**Figure 6.11: Creating three pointers. (prev_p, current_p, and next_p)**

6. Change the ***current*** pointer to point to **node(5)**.

   ***current.next = nextp.next***

7. Change the ***prev*** pointer to point to **node(3)**.

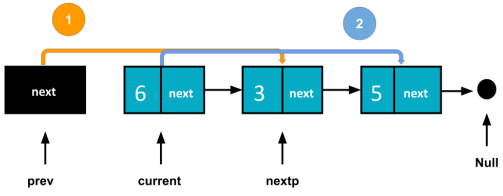



**Figure 6.12: Swapping nodes 6, and 3.**

   ***prev.next = nextp***

8. Update the next pointer of the node referenced by the ***nextp*** pointer to point to **node(3)**.

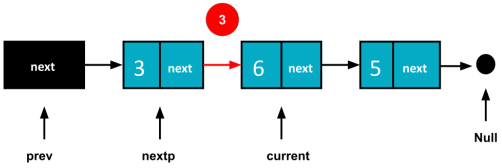



**Figure 6.13: Setting the next pointer of the node that is pointed by the next_p pointer to point to node(3).**

***nextp.next = current***

9. Move the ***prev*** pointer to **node(6)**

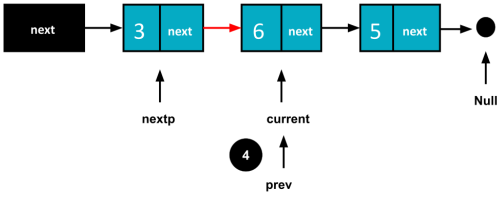



**Figure 6.14: Moving the prev_p pointer to node(6).**

***prev = nextp.next***

10. Move the ***current*** pointer to **node(5)**

***current = current.next***

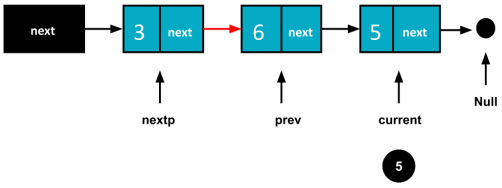



**Figure 6.15: Moving the current_p pointer to node(5)**

11. Finally, move the ***next*** pointer to the terminal (**Null**).

    ***nextp = current.next***

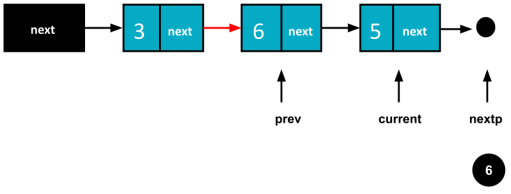



**Figure 6.16: Moving the next_p pointer to the terminal (Nulll).**

Next is the full code of the algorithm:

```
def swap_node_pairs(head):

    if not head or not head.next:
        return head

    dummy_node = ListNode()
    dummy_node.next = head
    prev = dummy_node
    current = head
    nextp = head.next

    while prev.next and prev.next.next:
        # finish here
        nextp = current.next
        # code logically starts from here
        current.next = nextp.next
        nextp.next = current
        prev.next = j

        prev = nextp.next
        current = current.next

    return dummy_node.next
```

## 6.3.4 Complexity Analysis

### Time Complexity

The time complexity will be ***O(n)***, because we will iterate over all the array.

### Space Complexity

The space complexity will be ***O(1)***, because there is no extra space used.

# 6.4 Validate Binary Search Tree

This problem belongs to the tree data structure. In this problem, we have to validate if this binary tree is considered a binary search tree or not. A binary tree is considered a binary search tree if:

1. The left subtree of the parent node has nodes with key values less than the parent node key's value. In other words, all the nodes under the left subtree of a given parent node are less than the parent node.
2. The right subtree of the parent node has nodes with key values that are bigger than the parent node key's value.
3. The left subtree and right subtree should also be binary search trees.

See how those subtrees under the root node follow these rules.

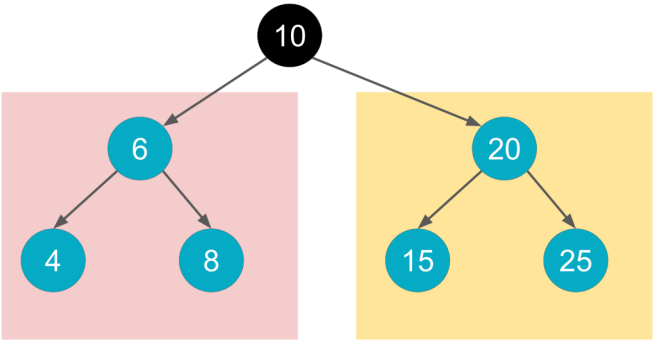



**Figure 6.17: Binary Search Tree.**

## 6.4.1 Problem Description

**Given the root of a binary tree, validate if the binary tree is a binary search tree or not.**

## 6.4.2 I/O Examples

**Input 1:** ***root***

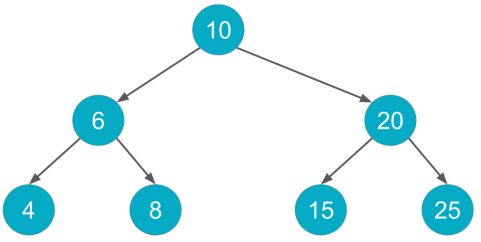



**Figure 6.18: Input Sample 1.**

**Output: True**

**Input 2:** ***root***

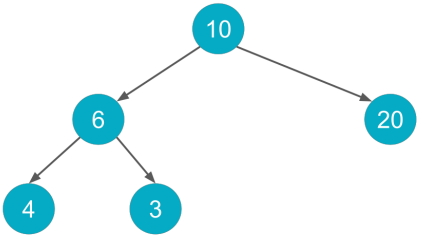



**Figure 6.19: Input Sample 2.**

**Output: False.** Because the node with key value 3 has a key value that is less than its parent.

**Input 3:** ***root***

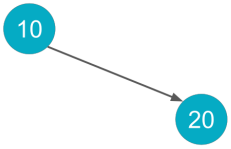



**Figure 6.20: Input Sample 3.**

**Output: True**

## 6.4.3 Solution

Based on the definition of a binary search tree, we can observe an interesting pattern in the order of the tree nodes. Let us take the following binary tree as an example:

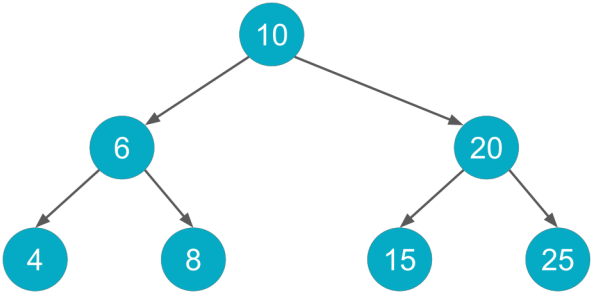



**Figure 6.21: Binary Tree.**

If we unfold the right subtree of the root node and compare the nodes' values, we will have the following pattern:

***node(4) < node(6) < node(8)***

Then, if we apply that to the right subtree, we will have:

***node(4) < node(6) < node(8) < node(15) < node(20) < node(25)***

Now, let us include the root node to fit this order:

***node(4) < node(6) < node(8) < node(10) < node(15) < node(20) < node(25)***

This pattern could be summarized as:

***left child < parent node < right child***

This pattern is similar to the binary tree inorder traversal approach. Then, we can say that a binary tree is not a valid binary tree if:

***left child ≥ parent node***

***OR***

***right child ≤ parent node***

**Then, the full algorithm to solve this problem will be as follows**

1. Traverse a binary tree in an inorder traversal fashion.
2. Store the nodes' values inside a list ***num***.
3. Check if there are invalid node pairs, by comparing the list values:

***A pair is invalid if num[i] ≥ num[i+1]***.

Now, let us see the full code:

```
def isvalid_BT(root):
    def Inorder(root, bag):

        if not root:
            return []
        if not root.left and not root.right:
            return bag.append(root.val)
        if not root.left:
            bag.append(root.val)
            return Inorder(root.right, bag)
        if not root.right:
            Inorder(root.left, bag)
            return bag.append(root.val)

        Inorder(root.left, bag)
        bag.append(root.val)
        Inorder(root.right, bag)

    bag = []
    Inorder(root, bag)

    for index in range(len(bag) - 1):
        if bag[index] >= bag[index + 1]:
            return False

    return True
```

## 6.4.4 Complexity Analysis

### Time Complexity

The time complexity for this solution is ***O(n)***, because every node will be visited one time.

### Space Complexity

However, the space complexity will be ***O(n)***, as we use the ***num*** list to store the nodes' values for the linear checkup operation.

# 6.5 Same Binary Tree

In this problem, our job is to check if two binary trees are the same or not.

## 6.5.1 Problem Description

**Given the roots of two binary trees, $root_1$ and $root_2$ we have to check if the two trees are similar to each other or not.**

## 6.5.2 I/O Examples

**Input: $root_1$ , $root_2$**

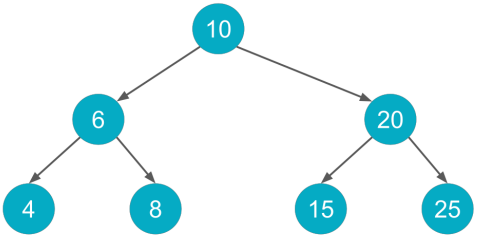



**Figure 6.22: Input Sample.**

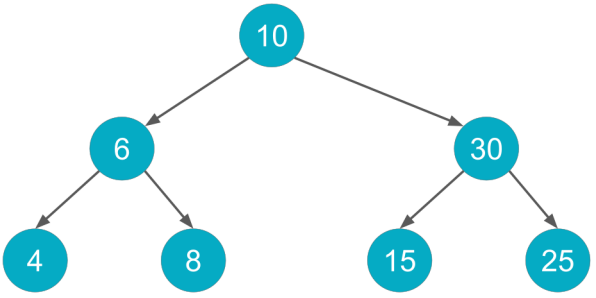



**Figure 6.23: Output Sample.**

**Output**: **False**, The parent nodes of the right subtrees in the both trees, **(20, 30)** are not the same.

## 6.5.3 Solution

A simple solution to this problem will be to traverse both binary trees simultaneously and check if the values of each corresponding pair are the same as follows:

1. Check if both trees end at the same level.
2. Check if there is dissimilarity in the binary trees' height.
3. Check if there is an unmatching among key values of the binary tree.

Here is the full code put together below :

```
def is_same_BT(root1, root2):
    def check(root1, root2):

        if not root1 and not root2:  # check for
                                     # the end of both trees.
            return True
        if not root1 or not root2:  # Check for
                                    #the heights' similarity.
            return False

        if root1.val != root2.val:  # check for
                                    # the key values' similarity.
            return False

        return check(root1.left, root2.left) and check(root1.right, roo\
t2.right)

    return check(root1, root2)
```

## 6.5.4 Complexity Analysis

### Time Complexity

The time complexity for this solution is ***O(n+m)***, where ***n = number of nodes in the first tree***, and ***m = number of nodes in the second tree***. In this case, every node in each tree will be visited one time. In the worst-case scenario we will be having two different trees which will yield this time complexity.

### Space Complexity

The space complexity will be ***O(n)*** which equals the size of the stack that needed to be allocated due to the recursive calls.

# 6.6 Symmetric Binary Tree

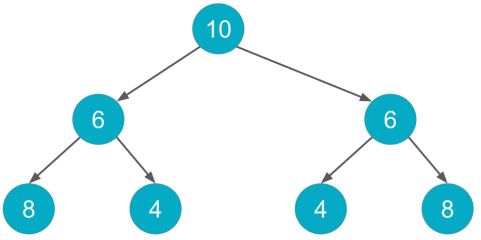



Figure 6.24: Binary Tree.

**We can say this is a symmetric binary tree if:**

1. The left child of the left subtree equals the right child of the right subtree.(Condition 1)

   ***left_child_node(7) of left_parent_node(8) = right_child_node(7) of right_parent_node(8)***

2. The right child of the left subtree equals the left child of the right subtree.(Condition 2)

   ***right_child_node(5) of left_parent_node(8) = left_child_node(5) of right_parent_node(8)***

In other words, if we draw a vertical line dividing the tree into two halves, both halves should mirror each other.

## 6.6.1 Problem Description

**Given the *root* of a binary tree, determine if it is symmetric binary tree or not.**

## 6.6.2 I/O Examples

**Input 1:** ***root***

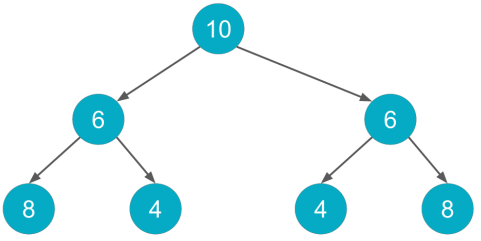



**Figure 6.25: Input Sample 1.**

**Output: True**

**Input 2:** ***root***

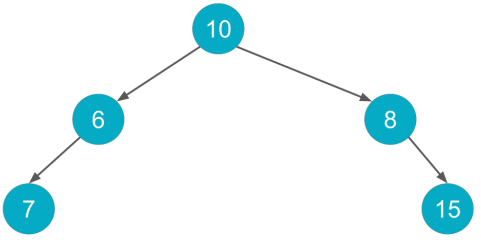



**Figure 6.26: Input Sample 2.**

**Output: False. node(15) != node(7).**

### 6.6.3 Solution

Based on the definition above, our algorithm can be developed simply. All we need is to check if the tree follows conditions (1), and (2). Let us see the full code below:

```
def is_symmetric_BT(root):
    def check(left, right):

        if not left and not right:
            return True
        if not left or not right:
            return False
        # Check the similarity of
        # the nodes' values (base condition).
        if left.val != right.val:
            return False
        # Check condition (1), and condition (2).
        return check(left.left, right.right)\
        and check(left.right, right.left)

    return check(root.left, root.right)
```

### 6.6.4 Complexity Analysis

#### Time Complexity

The time complexity for this solution is ***O(n)***, because every node will be visited one time.

#### Space Complexity

And the space complexity will be ***O(n)*** which it equals to the size of the stack that needed to be allocated due to the recursive calls.

# 9 | CHECKLIST

# Problem-Solving

## CHECKLIST

- [ ] Read the problem description carefully.
- [ ] Write down any constraints and special keywords in the problem description.
- [ ] Convert the problem description to an input-output question.
- [ ] Write down three examples.
- [ ] Develop a brute-force solution.
- [ ] Write down time and space complexities.
- [ ] Look for bottle-necks on your brute-force solution.
- [ ] Optimize using some algorithm techniques.
- [ ] Write down time and space complexities.

# References


1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," MIT Press, 2009. [Online]. Available: https://mitpress.mit.edu/books/introduction-algorithms-third-edition.
2. S. S. Skiena, "The Algorithm Design Manual," Springer Science & Business Media, 2008. [Online]. Available: https://www.springer.com/gp/book/9781848000698.
3. A. Bhargava, "Grokking Algorithms," Manning Publications, 2016. [Online]. Available: https://www.manning.com/books/grokking-algorithms.
4. G. Lakhani and M. McDowell, "Cracking the Coding Interview: 189 Programming Questions and Solutions," CareerCup, 2015. [Online]. Available: http://www.crackingthecodinginterview.com/.
5. W. Khamies, "Unfold KSum Family Problems," Medium, 2023. [Online]. Available: https://blog.waleedkhamies.com/uacifds-unfold-ksum-family-problems-f27221357c92.
6. W. Khamies, "Unfold Combinatorics Family Problems," Medium, 2023. [Online]. Available: https://blog.waleedkhamies.com/uacifds-unfold-permutation-family-problems-38da375236e3#0a93.
7. Educative, "How to Implement a Breadth-First Search in Python," [Online]. Available: https://www.educative.io/answers/how-to-implement-a-breadth-first-search-in-python.
8. Brilliant, "Depth-First Search (DFS)," [Online]. Available: https://brilliant.org/wiki/depth-first-search-dfs/.
9. W. Khamies, "Unfold Binary Tree Family Problems," Medium, 2023. [Online]. Available: https://blog.waleedkhamies.com/unfold-binary-tree-family-problems-part-1-2-e986049a1fb2.

# Acknowledgments

*I am grateful to my supervisors for their support and mentorship, my friends for their encouragement, and my family for their unconditional love. I also appreciate the readers for their valuable feedback and support. Thank you all for being part of my journey.*

# About the Author

Waleed Khamies is an applied machine learning scientist with a deep passion for exploring the realm of semantic representations in multimodal data. With a strong background in research and engineering, Waleed has gained invaluable experience as a research engineer intern at Mila and a research associate intern at Brown University's Robotics Lab.

Holding a master's degree in Mathematical Sciences/Machine Learning from AMMI and a bachelor's degree in Electrical and Electronics Engineering from UofK, Waleed is recognized for his expertise in reinforcement learning and the development of cutting-edge deep learning models for robotics applications. His work has been published in reputable workshops, and he actively shares his knowledge and insights through his engaging blog and newsletter.

For more information about his work and interests, please visit his website at waleedkhamies.com.