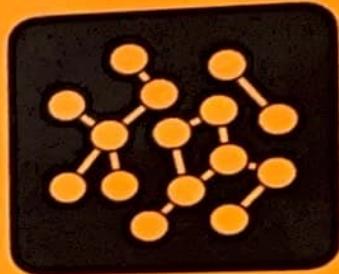
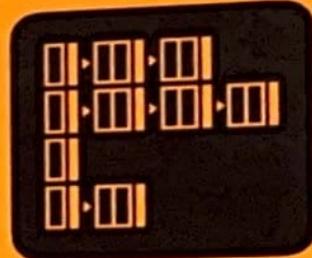
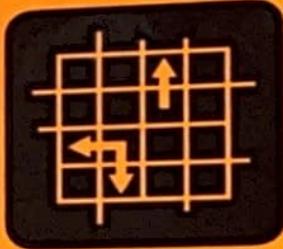
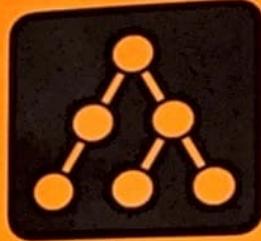


CODING INTERVIEW PATTERNS

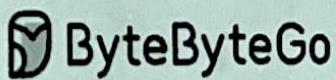
Nail Your Next Coding Interview





Coding Interview Patterns

Alex Xu | Shaun Gunawardane



Coding Interview Patterns

Copyright ©2024 ByteByteGo. All rights reserved. Published by ByteByteGo Inc.

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review.

Technical Editor: Daryl Lonnnon, Dennis Sell, Raymond Guo, Cristian Dugacicu

Copy Editor: Dominic Gover

Graphic Design: Guanxu Chen, Mengqiao Zhao

Join the community

We created a members-only Discord group. It is designed for community discussions on the following topics:

- Finding mock interview buddies.
- General chat with community members.

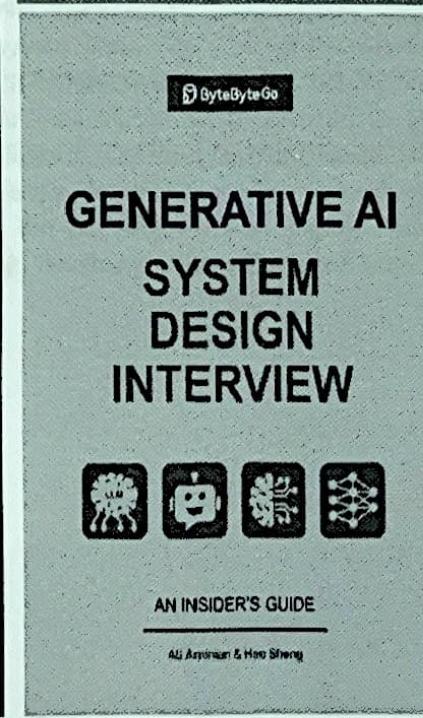
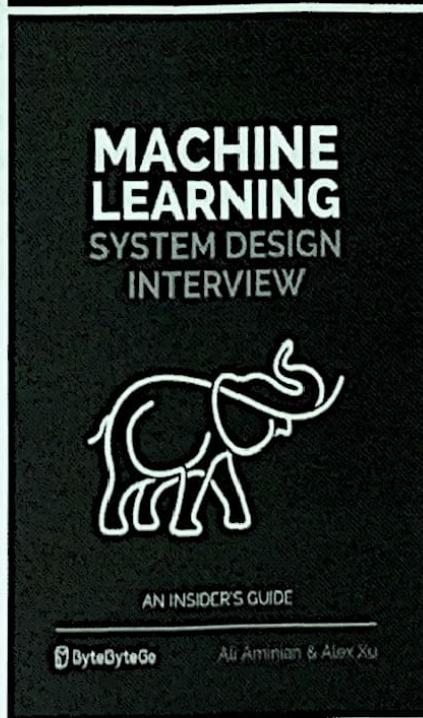
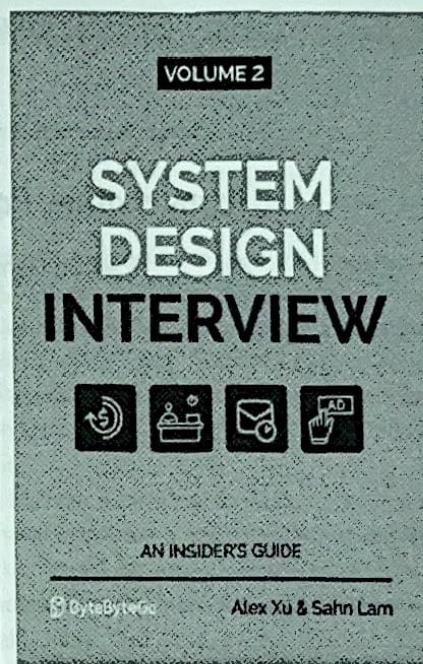
Come join us and introduce yourself to the community today! Use the link below or scan the barcode:

bit.ly/coding-patterns-discord



Other Books By ByteByteGo

The following books are available in paperback on Amazon:



To Julia.

Alex Xu

To my mother, Ravi, and Destiny.

Shaun Gunawardane

Acknowledgment

For HackerRank:

A special thanks to HackerRank: Some of the problems in this book are from HackerRank, included with the approval of CEO Vivek Ravisankar.

You can check out HackerRank here: <https://www.hackerrank.com>

For the reviewers:

This book has been written with significant input and reviews from over 100 engineers and managers. Thank you all, so much: Chandrasekhar A, Aadesh Aachaliya, Samin Ahsan, Mayank Ahuja, Rupen Anjaria, Hassan Sadeed Ali, Bhagawan Das Armani, Sauman Arshad, Guruprasad Bagade, Anton Balaniuc, Gaurav Bhardwaj, Pranav Bhasker, Andrii Bui, David Cattarin, Vinay Chandragiri, Farid Chowdhury, Chuluunsuren Damdinsuren, Ridip De, Cristian Dugaciu, Anil Kivilcim Eray, Sathish Reddy Gathpa, Alexander Golubkov, Yan Gou, Sunil Gudivada, Gabriel Guerra, Raymond Guo, Niwesh Gupta, Abhaar Gupta, Saurabh Gupta, Girish H Kiran, Siarhei Helis , Sally Iskhodzhanova, Vrishab Jain, Parveen Jain, Ankur Jain, Mainak Jana, Zyaad Jaunnoo, Satyam Jha, Tau Jin, Ravindra Kadiyala, Grégoire Karydes, Nilesh Khonde, Vikram Krishnamurthy, Yeun-Yuan (Jessie) Kuo, Boris Lavva, Yifan Li, Yiwei Li , Xue Cong Li, Terry Lim, Badri LM, Daryl Lonnon, Nikola Lukic, Naidu M, Raj Kumar Mahto, Sidharth Malhotra, Ayush Malviya, Aastha Mehta, Maximiliano Micciullo, Jitendra Mohanty, Massimo Monticelli, Mahidhar Mullapudi, Priyanka Nagpal, Tyron Naidoo, Prathamesh Naik, Aditya Narayan, Mu Niu, Vivek Notani, Rajesh Pandey, Feng Peng, Kartik Periseta, Bibhu Prasad Panigrahi, Adeel Qayyum, Satyen Rai , Sankar Ramanathan,, Manoel Ramon, George V. Reilly, Swarup Sahoo, Sumanth Sathyanarayana , Dennis Sell, Raj Shah, Sarthak Sharma, Vaibhav Sharma, Aishwarya Singh, Manu Sinha, Manish Sonal, Shivendra Srivastava, Janet Sung, Andrés Gutiérrez Ramírez, Abdus Sayef Reyadh, Jim Tang, Vaibhavi Tiwari, Daniel Tseng, Mehmet Şevki Ütebay, Ritikesh Vali, Siddharth Verma, Horatiu Eugen Vlad, Krishnakanth Vuppala-pati, Haoyu Wang, Xinwei Wang , Larbi Wiran, Yan Wu, Angela Xiang, Lingqian Xie, Hunter Xu, Chang You, Jiashen (Jason) Zhang, Kaiying Zhang, Tao Zhang, Yuheng Zhao, and Xiao Zhu.

Contents

Preface	v
Introduction	1
Coding Interviews: What to Expect	1
What do Interviewers Look For?	2
How to Approach a Problem in an Interview	3
Effective Communication in Interviews	4
Chapter 1 Two Pointers	7
Introduction to Two Pointers	7
Pair Sum - Sorted	10
Triplet Sum	14
Is Palindrome Valid	20
Largest Container	24
Chapter 2 Hash Maps and Sets	29
Introduction to Hash Maps and Sets	29
Pair Sum - Unsorted	32
Verify Sudoku Board	35
Zero Striping	40
Chapter 3 Linked Lists	47
Introduction to Linked Lists	47
Linked List Reversal	50
Remove the K th Last Node From a Linked List	56
Linked List Intersection	60
LRU Cache	63
Chapter 4 Fast and Slow Pointers	71
Introduction to Fast and Slow Pointers	71
Linked List Loop	73
Linked List Midpoint	77
Happy Number	80
Chapter 5 Sliding Windows	83
Introduction to Sliding Windows	83
Substring Anagrams	86
Longest Substring With Unique Characters	90
Longest Uniform Substring After Replacements	95
Chapter 6 Binary Search	101
Introduction to Binary Search	101
Find the Insertion Index	105

First and Last Occurrences of a Number	110
Cutting Wood	116
Find the Target in a Rotated Sorted Array	122
Chapter 7 Stacks	127
Introduction to Stacks	127
Valid Parenthesis Expression	130
Next Largest Number to the Right	133
Evaluate Expression	137
Chapter 8 Heaps	143
Introduction to Heaps	143
K Most Frequent Strings	145
Combine Sorted Linked Lists	151
Median of an Integer Stream	155
Chapter 9 Intervals	159
Introduction to Intervals	159
Merge Overlapping Intervals	161
Identify All Interval Overlaps	166
Largest Overlap of Intervals	170
Chapter 10 Prefix Sums	175
Introduction to Prefix Sums	175
Sum Between Range	177
K-Sum Subarrays	180
Product Array Without Current Element	184
Chapter 11 Trees	189
Introduction to Trees	189
Invert Binary Tree	193
Balanced Binary Tree Validation	197
Rightmost Nodes of a Binary Tree	200
Widest Binary Tree Level	204
Binary Search Tree Validation	207
Lowest Common Ancestor	212
Build a Binary Tree From Preorder and Inorder Traversals	216
Maximum Sum of a Continuous Path in a Binary Tree	221
Chapter 12 Tries	227
Introduction to Tries	227
Design a Trie	230
Insert and Search Words with Wildcards	235
Find All Words on a Board	239
Chapter 13 Graphs	247
Introduction to Graphs	247
Graph Deep Copy	250
Count Islands	253
Matrix Infection	258
Bipartite Graph Validation	264
Longest Increasing Path	268
Shortest Transformation Sequence	272
Merging Communities	280

Prerequisites	287
Chapter 14 Backtracking	293
Introduction to Backtracking	293
Find All Permutations	298
Find All Subsets	302
N Queens	305
Chapter 15 Dynamic Programming	309
Introduction to Dynamic Programming	309
Climbing Stairs	311
Minimum Coin Combination	316
Matrix Pathways	321
Neighborhood Burglary	325
Longest Common Subsequence	329
Longest Palindrome in a String	334
Maximum Subarray Sum	339
0/1 Knapsack	345
Chapter 16 Greedy	351
Introduction to Greedy Algorithms	351
Jump to the End	354
Gas Stations	358
Candies	364
Chapter 17 Sort and Search	371
Introduction to Sort and Search	371
Sort Linked List	373
Sort Array	379
K th Largest Integer	386
Chapter 18 Bit Manipulation	393
Introduction to Bit Manipulation	393
Hamming Weights of Integers	396
Lonely Integer	399
Swap Odd and Even Bits	401
Chapter 19 Math and Geometry	405
Introduction to Math and Geometry	405
Spiral Traversal	406
Reverse 32-Bit Integer	411
Maximum Collinear Points	415
Afterword	421

How to Use This Book

This book is structured to help you learn and master new topics. You can start with any chapter that interests you, or follow the chapters sequentially. If you prefer, you can approach the problems in a different order. That's perfectly fine. The chapters are designed to be self-contained, so you can jump between them without losing context.

Preface

Have you ever felt the crushing disappointment of being in a coding interview and realizing you have no idea how to find a cycle in a graph? Or maybe you've found yourself second-guessing whether to use `<` or `<=` in a binary search algorithm? Don't worry, you're not alone.

The sheer volume of coding problems available online can be overwhelming, making it difficult to know where to start, what to focus on, and how to tackle complex subjects. This barrier is faced by both novices and seasoned engineers.

Enter "Coding Interview Patterns." This book is designed to be your comprehensive guide to mastering coding interviews. We cut through the noise to bring you a curated collection of essential data structures and algorithms, all presented in a way that's quick to learn and easy to understand. Our goal is to help you not only solve the problems at hand, but also to recognize and apply the underlying patterns to new challenges with confidence.

Each chapter guides you through relevant problems step-by-step, in the way you would approach them in high-pressure interviews. We break down each problem, provide illustrations, and dive deep into the reasoning and intuition behind each solution.

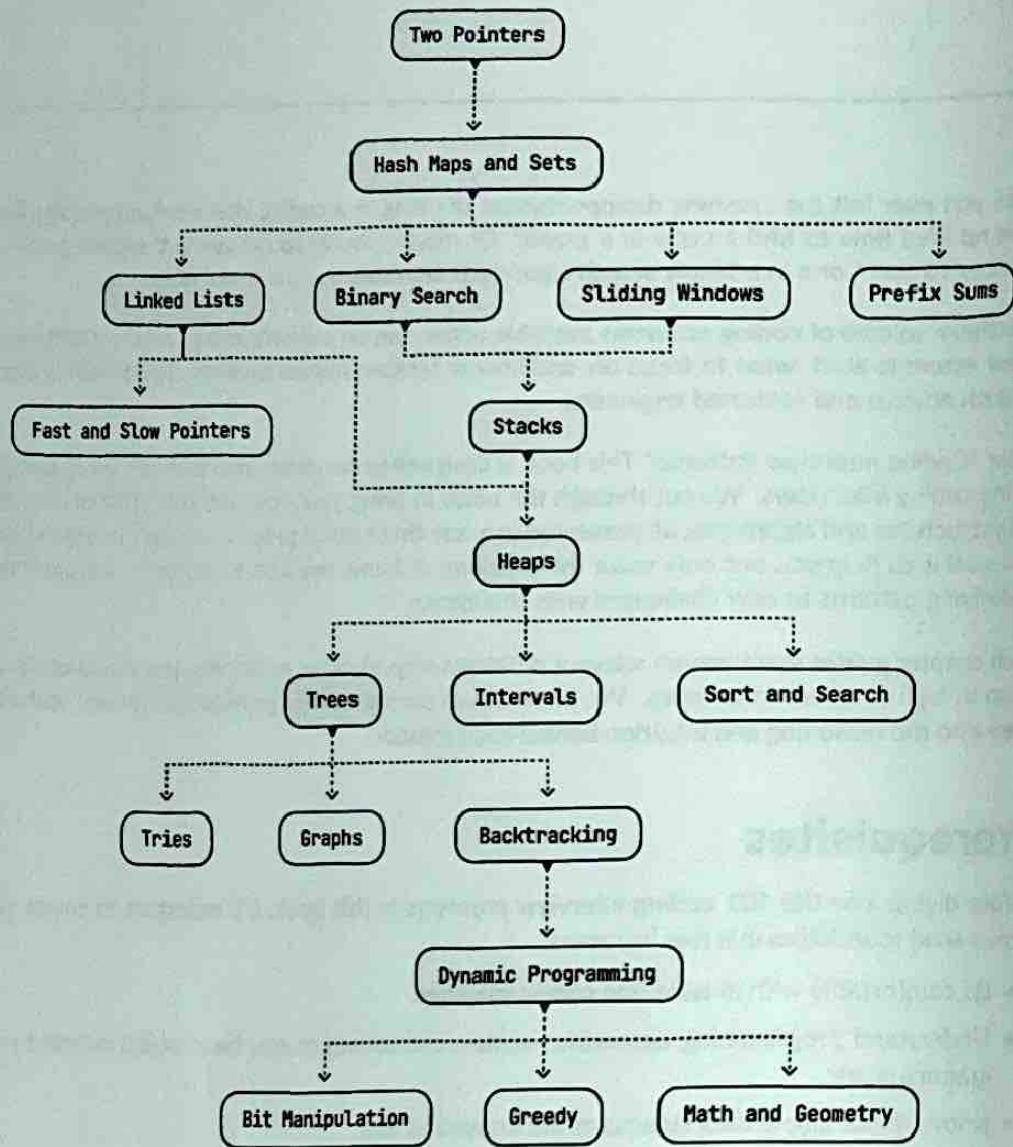
Prerequisites

Before diving into the 101 coding interview problems in this book, it's important to ensure you have a solid foundation in a few key areas:

- Be comfortable with at least one coding language.
- Understand programming essentials: syntax, control structures, basic object-oriented programming, etc.
- Know about simple data structures like arrays and lists.
- Be familiar with algebra, including logarithms and exponents.
- Understand the basics of recursion.
- Understand how Big O notation and complexity analysis work.

How to Use This Book

To ensure a structured learning experience and address dependencies between chapters, we've put considerable effort into developing a suggested order for tackling each chapter. Of course, if you prefer to approach the problems in a different order, that's perfectly fine, as dependencies are generally minimal.



Bonus Learning Material

Due to the space limitations of this book, we aren't able to feature every problem and solution. Therefore, we've compiled a PDF with bonus problems and their solutions as a supplementary resource.

To receive the bonus PDF, sign up for our Coding Interview Patterns newsletter by using the link below or scanning the QR code.

bit.ly/coding-patterns-pdf



uction

Practice, Practice, Practice

The only way to truly excel in coding interviews is with consistent practice. We highly recommend visiting our website to practice all 101 problems. Engaging with these problems on our platform is going to reinforce your learning, and better prepare you for high-stakes coding interviews:

bit.ly/run-code

Solutions Repository

You should invest ample time attempting to solve the problems on your own. However, if you get stuck or want to review the code in your favorite code editor, we've created a GitHub repository for easy access:

bit.ly/coding-patterns-code

Types of Coding Interviews

Introduction

The tech industry's eternal need for talent has led to evolving hiring practices. Initially, major tech companies like Microsoft relied on brain teasers to evaluate candidates' problem-solving abilities. However, these puzzles often fell short in assessing a candidate's true competency, experience, and programming skills.

Recognizing these limitations, the industry pivoted towards coding interviews. This shift acknowledged the diverse nature of software engineering, where professionals possess varied and distinct knowledge bases. Even though coding interviews aren't perfect, they provide a more inclusive approach, allowing a wider range of candidates to showcase their programming prowess, while effectively evaluating their problem-solving skills.

Coding Interviews: What to Expect

During a coding interview, you are presented with various programming challenges designed to test your knowledge of data structures and algorithms, as well as your ability to write clean, efficient code. Interviewers aren't just looking for the correct solution, they're interested in understanding how you approach the problem, think through different solutions, and communicate your reasoning clearly.

It's important to avoid bugs, but don't stress too much about minor mistakes. In fact, identifying and fixing bugs on the fly can be a positive sign, showing your debugging skills and attention to detail.

Additionally, interviewers want to hear your thought process as you work through the problem. Make sure to explain your decisions, discuss any trade-offs, and provide a narrative of your approach as you code. The coding language you use in an interview may vary depending on the role, so be sure to confirm your language preference with your recruiter beforehand.

Types of Coding Interviews

Coding assessments come in various formats, each with its own set of expectations and challenges. Understanding these formats will help you prepare more effectively.

Online assessment	This is a timed, remote test to evaluate your skills and knowledge via various coding challenges or multiple-choice questions.
Phone screen	A phone screen interview is typically a brief initial assessment, usually lasting 30-60 minutes. This step in the hiring process serves multiple purposes: <ul style="list-style-type: none">• The interviewer can evaluate your understanding of the company and the role.• The interviewer may assess how well you fit the team's culture.• Depending on the position, it can include technical questions, or even coding challenges.
Take-home assignment	This is a coding task or project to be completed in your own time. It allows you to work at your own pace and demonstrate your problem-solving skills. These assignments usually require a well-structured coding style, thorough testing, and clear documentation.
On-site/final day interviews	These interviews consist of a series of technical and sometimes behavioral interviews conducted at the company's location, or virtually. The coding interview portion of these interviews involves solving coding problems on a whiteboard, or using an online coding tool. The emphasis is on both the correctness and efficiency of your solutions, and your communication skills during the interview.

What do Interviewers Look For?

Positive Signals

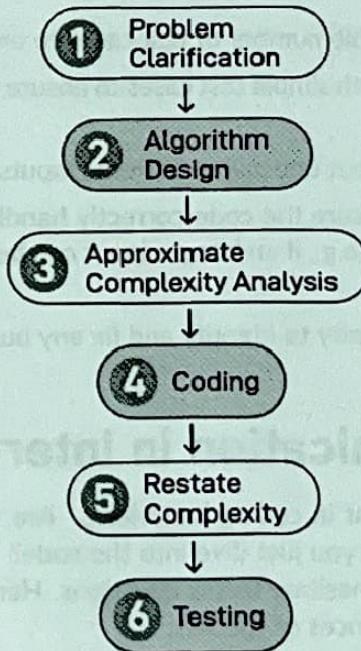
- **Strong problem-solving skills:** Demonstrate a solid grasp of data structures and algorithms. It's essential to know when and how to use them to solve complex problems efficiently.
- **Effective communication:** Clearly explain your reasoning, discuss trade-offs, and ensure the interviewer understands your approach and thought process.
- **Interaction and collaboration:** Engage with the interviewer, ask clarifying questions, and collaborate to refine your solution. This shows your ability to work well in a team.
- **Clean, efficient code:** Writing code that's not only correct, but is also clean, readable, and efficient is a big plus.
- **Ability to optimize:** Identify opportunities to optimize your solution's time and space complexity.
- **Adaptability and flexibility:** Be open to feedback and willing to consider alternative solutions. Interviewers value candidates who can pivot and adapt their approach when necessary.
- **Testing and debugging skills:** Proactively test your code and effectively debug errors. Commit to delivering a high-quality solution.

Negative Signals

- **Weak fundamentals:** A poor understanding of basic data structures, algorithms, and programming fundamentals can hinder your ability to solve problems efficiently.
- **Lack of clarity:** Struggling to articulate your thought process, or explain your approach can be a red flag. Interviewers need to understand your reasoning in order to assess your problem-solving skills accurately.
- **Overconfidence:** Displaying arrogance or overconfidence can be off-putting to interviewers. It's important to show humility and a willingness to learn.
- **Rigidity:** Being inflexible and resistant to feedback, or refusing to consider alternative approaches can be a red flag. Interviewers value candidates who can adapt and think critically.
- **Ignoring constraints:** Failing to consider problem constraints or edge cases can indicate a lack of thoroughness.
- **Poor time management:** Spending too long on one aspect of the problem, or not completing the solution within the allotted time can be a significant drawback.

How to Approach a Problem in an Interview

Having a framework for answering coding interview problems is more important than many may realize. An unstructured answer makes your solution difficult to follow. Here is our recommended guide for tackling coding problems.



1. Clarify the problem and collect constraints.

- Restate the problem in your own words. This ensures you and your interviewer are on the same page.
- Ask clarifying questions to gather all necessary constraints and edge cases.

2. Design your algorithm.

- It's useful to start with a brute force or naive solution. A simple solution is better than no solution. More optimal solutions can sometimes be built up from the brute force or

naive solution. If it's straightforward, describing it verbally without coding it up might be sufficient.

- Consider and discuss multiple possible solutions. Talk through these options, decide which one will work best, and explain your choice.
- If the algorithm is complex, consider using pseudocode. This makes your strategy clear and can serve as a high-level guide when writing the actual code.

3. Provide approximate time and space complexity estimates.

- Provide an initial complexity analysis of your approach to demonstrate its effectiveness and discuss it with the interviewer.
- At this stage, an approximation is usually acceptable for more complicated algorithms. A more thorough analysis can be done once the code is written.

4. Write clean, readable code.

- Use meaningful variable names when possible.
- Making use of helper functions can help structure your code and avoid repetition.
- Designing custom classes for specific objects can improve readability (e.g., representing intervals with an Interval class with start and end attributes).

5. Restate the complexity.

- Clearly state the time and space complexity of your code.
- Highlight any differences between this and the initially stated complexity. Explain why these differences exist.

6. Test your code.

- Consider using a reasonable number of test cases to ensure your code is correct.
- Test simple cases: start with simple test cases to ensure the basic functionality of the code works.
- Test edge cases: think about unusual or extreme inputs that might break your code.
- Test boundary values: ensure the code correctly handles values at the lower and upper limits of your input range (e.g., if an integer input can be between 0 and $2^{31} - 1$, test both 0 and $2^{31} - 1$).
- Use testing as an opportunity to identify and fix any bugs in your code.

Effective Communication in Interviews

Communication is a crucial element in coding interviews. Are you asking for requirements and clarification when necessary, or do you just dive into the code? Remember, the coding interview should be a conversation, so don't hesitate to ask questions. Here are some tips on how effective communication can boost your chances of success:

- **Think out loud.** Run through your thought processes with the interviewer. It's often better for them to see how you leverage observations and tackle the problem from various angles, rather than to arrive at the optimal solution without any discussion.
- **Ensure the interviewer is following along.** This is crucial because it allows you to clarify any confusion they might have about your approach, correct misunderstandings, and highlight any flaws in your approach, if you're on the wrong track.
- **Discuss alternative approaches.** When faced with multiple ways to approach a problem -

such as choosing between different data structures – let the interviewer know your options. Discuss pros and cons of each, and explain your final choice.

- **Getting unstuck.** If you find yourself stuck, explain where you are in your thought process, what options you've considered, and why you think there's a dead end. This allows the interviewer to understand your situation and possibly provide guidance.
- **Thinking time.** Sometimes, it's useful to take a moment to think about a problem in a period of silence. Always let the interviewer know if you need a minute to think about it. This clarifies that you're processing and tackling the problem, and are not just stuck and lost for words.

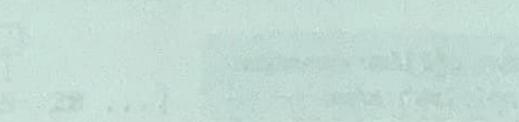
In addition to the interview tips in this introduction, you will find many more in the upcoming chapters. With all that said, let's dive straight in and master the coding interview patterns that will take your interview performance to the next level!

Introduction to Two Pointers

In this chapter, we'll learn about two-pointer solutions where we can move from start to end in two pointers, and just like a painter's brush, a variable that represents the current object being painted. In this chapter, we'll learn how to use two pointers to solve common problems involving arrays and strings.

[... 30 5 1 20 ...]

Introducing a second pointer opens a new world of possibilities. Most importantly, this can help reduce our time complexity. With two pointers, we can traverse the elements in linear time, which is much faster than the previous approach.

[... 30 5 1 20 ...] 

Let's start by looking at some common problems involving two pointers and their applications with arrays and strings. As the code snippets below, we can see how two pointers can be used to solve many two-element-related problems.

[... 14 5 1 20 ...]
[... 30 5 1 20 ...]

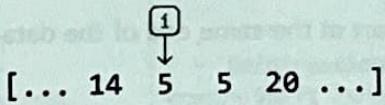
When approaching two-pointer problems, it's important to understand the problem requirements and constraints. An array or string has elements with predictable dynamics in terms of size and order. However, we must also consider whether the values are unique or not, and whether they are sorted or unsorted. For example, consider a problem that requires us to find two numbers in an array that add up to a target value. We must consider the following constraints:

1. The array contains unique elements.
2. The array is sorted in ascending order.
3. The array has a length of at least 2.

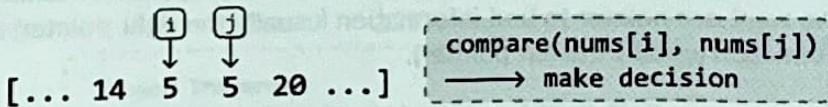
Two Pointers

Introduction to Two Pointers

As the name implies, a two-pointer pattern refers to an algorithm that utilizes two pointers. But what is a pointer? It's a variable that represents an index or position within a data structure, like an array or linked list. Many algorithms just use a single pointer to attain or keep track of a single element:



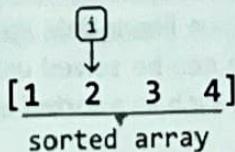
Introducing a second pointer opens a new world of possibilities. Most importantly, we can now make **comparisons**. With pointers at two different positions, we can compare the elements at those positions and make decisions based on the comparison:



In many cases, such comparisons are made using two nested for-loops, which takes $O(n^2)$ time, where n denotes the length of the data structure. In the code snippet below, *i* and *j* are two pointers used to compare every two elements of an array:

```
for i in range(n):
    for j in range(i + 1, n):
        compare(nums[i], nums[j])
```

Often, this approach does not take advantage of **predictable dynamics** that might exist in a data structure. An example of a data structure with predictable dynamics is a sorted array: when we move a pointer in a sorted array, we can predict whether the value being moved to is greater or smaller. For example, moving a pointer to the right in an ascending array guarantees we're moving to a value greater than or equal to the current one:



prediction: if $\text{nums}[i] == 2$, then $\text{nums}[i + 1] \geq 2$

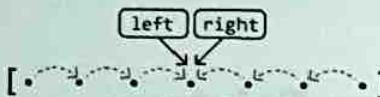
As you can see, data structures with predictable dynamics let us move pointers in a logical way. Taking advantage of this predictability can lead to improved time and space complexity, which we'll illustrate with real interview problems in this chapter.

Two-pointer Strategies

Two-pointer algorithms usually take only $O(n)$ time by eliminating the need for nested for-loops. There are three main strategies for using two pointers.

Inward traversal

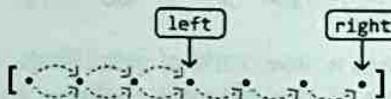
This approach has pointers starting at opposite ends of the data structure and moving inward toward each other:



The pointers move toward the center, adjusting their positions based on comparisons, until a certain condition is met, or they meet/cross each other. This is ideal for problems where we need to compare elements from different ends of a data structure.

Unidirectional traversal

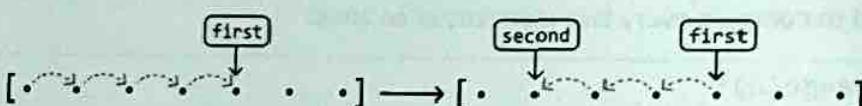
In this approach, both pointers start at the same end of the data structure (usually the beginning) and move in the same direction:



These pointers generally serve two different but supplementary purposes. A common application of this is when we want one pointer to find information (usually the right pointer) and another to keep track of information (usually the left pointer).

Staged traversal

In this approach, we traverse with one pointer, and when it lands on an element that meets a certain condition, we traverse with the second pointer:



Similar to unidirectional traversal, both pointers serve different purposes. Here, the first pointer is used to search for something, and once found, a second pointer finds additional information concerning the value at the first pointer.

We discuss all of these techniques in detail throughout the problems in this chapter.

When To Use Two Pointers?

A two-pointer algorithm usually requires a linear data structure, such as an array or linked list. Otherwise, an indication that a problem can be solved using the two-pointer algorithm, is when the input follows a predictable dynamic, such as a sorted array.

Predictable dynamics can take many forms. Take, for instance, a palindromic string. Its symmetrical pattern allows us to logically move two pointers toward the center. As you work through the

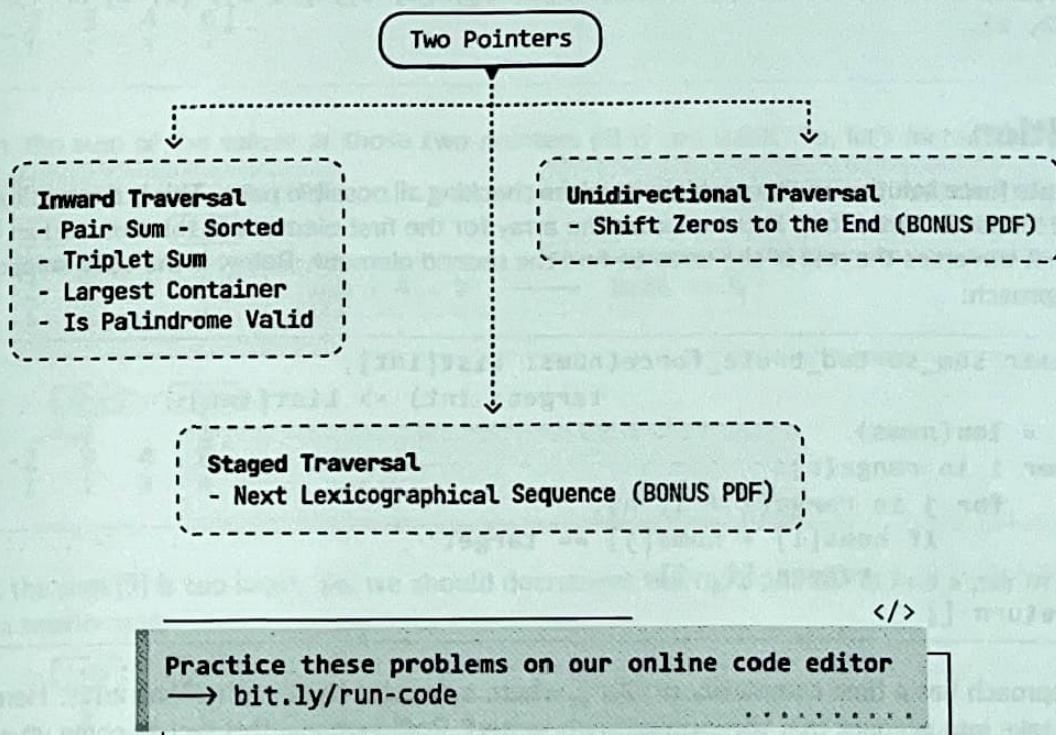
problems in this chapter, you'll learn to recognize these predictable dynamics more easily.

Another potential indicator that a problem can be solved using two pointers is if the problem asks for a pair of values or a result that can be generated from two values.

Real-world Example

Garbage collection algorithms: In memory compaction – which is a key part of garbage collection – the goal is to free up contiguous memory space by eliminating gaps left by deallocated (aka dead) objects. A two-pointer technique helps achieve this efficiently: a 'scan' pointer traverses the heap to identify live objects, while a 'free' pointer keeps track of the next available space to where live objects should be relocated. As the 'scan' pointer moves, it skips over dead objects and shifts live objects to the position indicated by the 'free' pointer, compacting the memory by grouping all live objects together and freeing up continuous blocks of memory.

Chapter Outline



To receive the bonus PDF, sign up for our Coding Interview Patterns newsletter by using the link below:

bit.ly/coding-patterns-pdf

The two-pointer pattern is very versatile and, consequently, quite broad. As such, we want to cover more specialized variants of this algorithm in separate chapters, such as *Fast and Slow Pointers* and *Sliding Windows*.

Pair Sum – Sorted

Given an array of integers sorted in ascending order and a target value, return the indexes of any pair of numbers in the array that sum to the target. The order of the indexes in the result doesn't matter. If no pair is found, return an empty array.

Example 1:

Input: `nums = [-5, -2, 3, 4, 6]`, `target = 7`

Output: `[2, 3]`

Explanation: `nums[2] + nums[3] = 3 + 4 = 7`

Example 2:

Input: `nums = [1, 1, 1]`, `target = 2`

Output: `[0, 1]`

Explanation: other valid outputs could be `[1, 0]`, `[0, 2]`, `[2, 0]`, `[1, 2]` or `[2, 1]`.

Intuition

The brute force solution to this problem involves checking all possible pairs. This is done using two nested loops: an outer loop that traverses the array for the first element of the pair, and an inner loop that traverses the rest of the array to find the second element. Below is the code snippet for this approach:

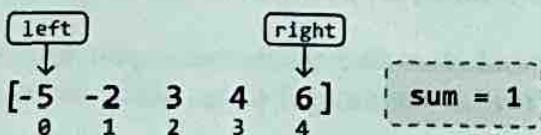
```
def pair_sum_sorted_brute_force(nums: List[int],
                                  target: int) -> List[int]:
    n = len(nums)
    for i in range(n):
        for j in range(i + 1, n):
            if nums[i] + nums[j] == target:
                return [i, j]
    return []
```

This approach has a time complexity of $O(n^2)$, where n denotes the length of the array. Here, we do not take into account that the input array is sorted. Could we use this fact to come up with a more efficient solution?

A two-pointer approach is worth considering here because a sorted array allows us to move the pointers in a logical way. Let's see how this works in the example below:

`[-5 -2 3 4 6], target = 7`

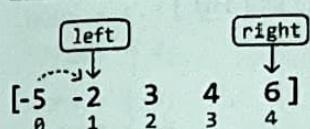
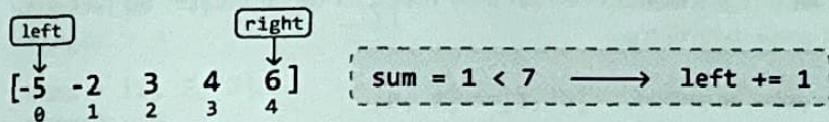
A good place to start is by looking at the smallest and largest values: the first and last elements, respectively. The sum of these two values is 1.



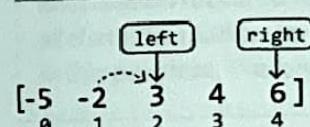
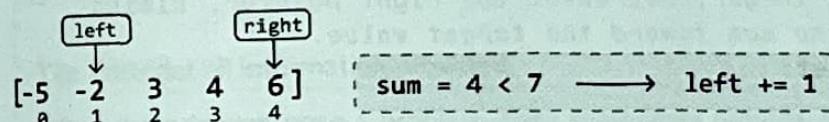
Since 1 is less than the target, we need to move one of the pointers to find a new pair with a larger sum.

- **Left pointer:** The left pointer will always point to a value less than or equal to the value at the right pointer because the array is sorted. Incrementing it would result in a sum greater than or equal to the current sum of 1.
- **Right pointer:** Decrementing the right pointer would result in a sum that's less than or equal to 1.

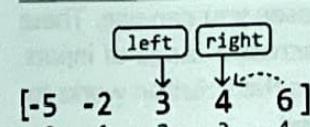
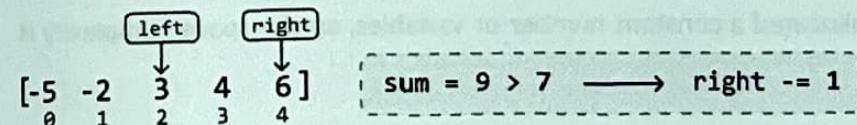
Therefore, we should increment the left pointer to find a larger sum:



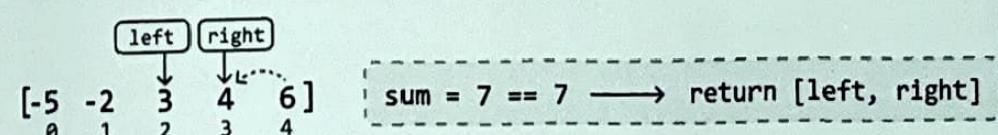
Again, the sum of the values at those two pointers (4) is too small. So, let's increment the left pointer:



Now, the sum (9) is too large. So, we should decrement the right pointer to find a pair of values with a smaller sum:



Finally, we found two numbers that yield a sum equal to the target. Let's return their indexes:



Above, we've demonstrated a two-pointer algorithm using inward traversal. Let's summarize this logic. For any pair of values at `left` and `right`:

- If their sum is less than the target, increment `left`, aiming to increase the sum toward the target value.
- If their sum is greater than the target, decrement `right`, aiming to decrease the sum toward the target value.
- If their sum is equal to the target value, return `[left, right]`.

We can stop moving the `left` and `right` pointers when they meet, as this indicates no pair summing to the target was found.

Implementation

```
def pair_sum_sorted(nums: List[int], target: int) -> List[int]:  
    left, right = 0, len(nums) - 1  
    while left < right:  
        sum = nums[left] + nums[right]  
        # If the sum is smaller, increment the left pointer, aiming  
        # to increase the sum toward the target value.  
        if sum < target:  
            left += 1  
        # If the sum is larger, decrement the right pointer, aiming  
        # to decrease the sum toward the target value.  
        elif sum > target:  
            right -= 1  
        # If the target pair is found, return its indexes.  
        else:  
            return [left, right]  
    return []
```

Complexity Analysis

Time complexity: The time complexity of `pair_sum_sorted` is $O(n)$ because we perform approximately n iterations using the two-pointer technique in the worst case.

Space complexity: We only allocated a constant number of variables, so the space complexity is $O(1)$.

Test Cases

In addition to the examples already discussed, here are some other test cases you can use. These extra test cases cover different contexts to ensure the code works well across a range of inputs. Testing is important because it helps identify mistakes in your code, ensures the solution works for uncommon inputs, and brings attention to cases you might have overlooked.

Input	Expected output	Description
nums = [] target = 0	[]	Tests an empty array.
nums = [1] target = 1	[]	Tests an array with just one element.
nums = [2, 3] target = 5	[0, 1]	Tests a two-element array that contains a pair that sums to the target.
nums = [2, 4] target = 5	[]	Tests a two-element array that doesn't contain a pair that sums to the target.
nums = [2, 2, 3] target = 5	[0, 2] or [1, 2]	Testing an array with duplicate values.
nums = [-1, 2, 3] target = 2	[0, 2]	Tests if the algorithm works with a negative number in the target pair.
nums = [-3, -2, -1] target = -5	[0, 1]	Tests if the algorithm works with both numbers of the target pair being negative.

Interview Tip

Tip: Consider all information provided.



When interviewers pose a problem, they sometimes provide only the minimum amount of information required for you to start solving it. Consequently, it's crucial to thoroughly evaluate all that information to determine which details are essential for solving the problem efficiently. In this problem, the key to arriving at the optimal solution is recognizing that the input is sorted.

Triplet Sum

Given an array of integers, return all triplets $[a, b, c]$ such that $a + b + c = 0$. The solution must not contain duplicate triplets (e.g., $[1, 2, 3]$ and $[2, 3, 1]$ are considered duplicate triplets). If no such triplets are found, return an empty array.

Each triplet can be arranged in any order, and the output can be returned in any order.

Example:

Input: $\text{nums} = [0, -1, 2, -3, 1]$
Output: $[-3, 1, 2], [-1, 0, 1]$

Intuition

A brute force solution involves checking every possible triplet in the array to see if they sum to zero. This can be done using three nested loops, iterating through each combination of three elements.

Duplicate triplets can be avoided by sorting each triplet, which ensures identical triplets with different representations (e.g., $[1, 3, 2]$ and $[3, 2, 1]$) are ordered consistently ($[1, 2, 3]$). Once sorted, we can add these triplets to a hash set. This way, if the same triplet is encountered again, the hash set will only keep one instance. Below is the code snippet for this approach:

```
def triplet_sum_brute_force(nums: List[int]) -> List[List[int]]:
    n = len(nums)
    # Use a hash set to ensure we don't add duplicate triplets.
    triplets = set()
    # Iterate through the indexes of all triplets.
    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                if nums[i] + nums[j] + nums[k] == 0:
                    # Sort the triplet before including it in the
                    # hash set.
                    triplet = tuple(
                        sorted([nums[i], nums[j], nums[k]]))
                    triplets.add(triplet)
    return [list(triplet) for triplet in triplets]
```

This solution is quite inefficient with a time complexity of $O(n^3)$, where n denotes the length of the input array. How can we do better?

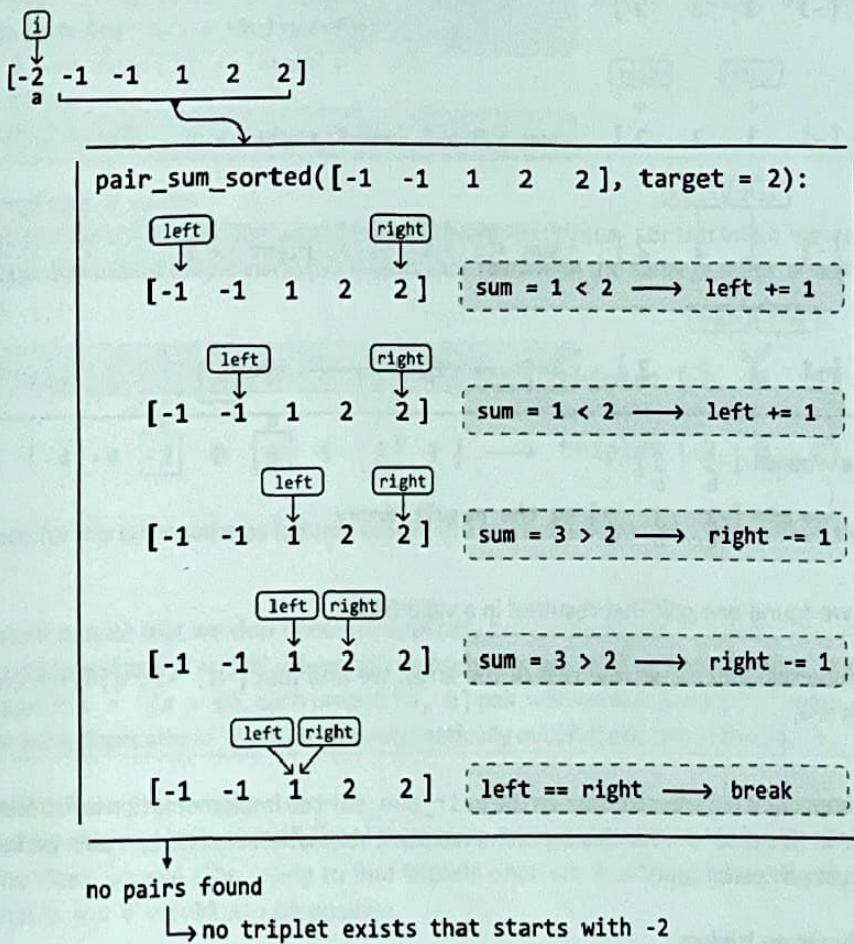
Let's see if we can find at least one triplet that sums to 0. Notice that if we fix one of the numbers in a triplet, the problem can be reduced to finding the other two. This leads to the following observation:

For any triplet $[a, b, c]$, if we fix 'a', we can focus on finding a pair $[b, c]$ that sums to ' $-a$ ' ($a + b + c = 0 \rightarrow b + c = -a$).

Sound familiar? That's because the problem of finding a pair of numbers that sum to a target has already been addressed by *Pair Sum - Sorted*. However, we can only use that algorithm on a sorted array. So, the first thing we should do is sort the input. Consider the following example:

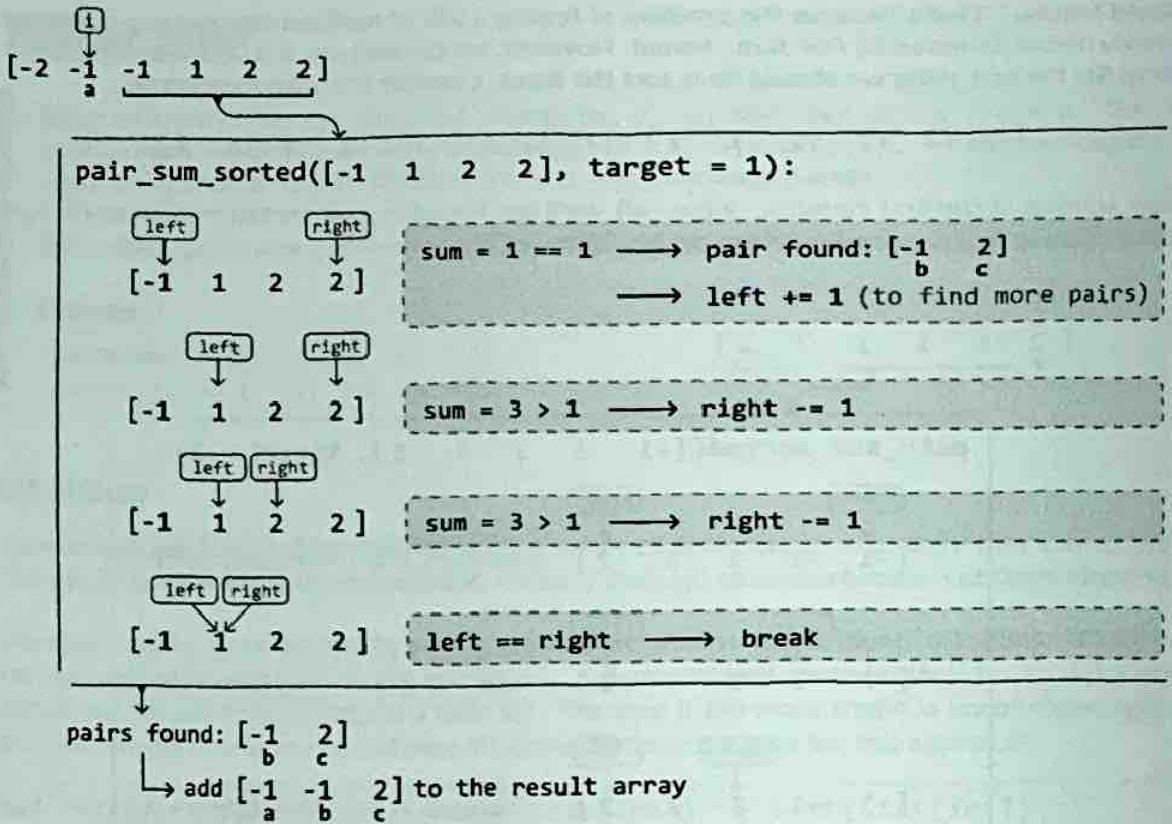
$[-1 \ 2 \ -2 \ 1 \ -1 \ 2]$ $\xrightarrow{\text{sort}}$ $[-2 \ -1 \ -1 \ 1 \ 2 \ 2]$

Now, starting at the first element, -2 (i.e., ' a '), we'll use the *pair_sum_sorted* method on the rest of the array to find a pair whose sum equals 2 (i.e., ' $-a$ ')



As you can see, when we called *pair_sum_sorted*, we did not find a pair with a sum of 2 . This indicates there are no triplets starting with -2 that add up to 0 .

So, let's increment our main pointer, i , and try again.



This time, we found one pair that resulted in a valid triplet.

If we continue this process for the rest of the array, we find that `[-1, -1, 2]` is the only triplet whose sum is 0.

There's an important difference between the `pair_sum_sorted` implementation in *Pair Sum - Sorted* and the one in this problem: for this problem, we don't stop when we find one pair, we keep going until all target pairs are found.

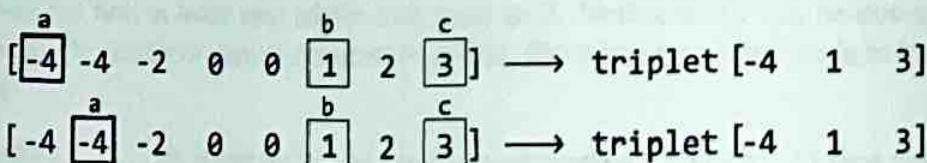
Handling duplicate triplets

Something we previously glossed over is how to avoid adding duplicate triplets. There are two cases in which this happens. Consider the example below:

`[-4, -4, -2, 0, 0, 1, 2, 3]`

Case 1: duplicate 'a' values

The first instance where duplicates may occur is when seeking pairs for triplets that start with the same 'a' value:



Since `pair_sum_sorted` would look for pairs that sum to `'-a'` in both instances, we'd naturally end

up with the same pairs and, hence, the same triplets.

To avoid picking the same 'a' value, we keep increasing i (where $\text{nums}[i]$ represents the value 'a') until it reaches a different number from the previous one. We do this before we start looking for pairs using the `pair_sum_sorted` method. This logic works because the array is sorted, meaning equal numbers are next to each other. The code snippet for checking duplicate 'a' values looks like this:

```
# To prevent duplicate triplets, ensure 'a' is not a repeat of the
# previous element in the sorted array.
if i > 0 and nums[i] == nums[i - 1]:
    continue
... Find triplets ...
```

Case 2: duplicate 'b' values

As for the second case, consider what happens during `pair_sum_sorted` when we encounter a similar issue. For a fixed target value ('-a'), pairs that start with the same number 'b' will always be the same:

[-4 -4	a	b	c	3]	→ triplet [-2 0 2]
[-4 -4	a	b	c	3]	→ triplet [-2 0 2]

The remedy for this is the same as before: ensure the current 'b' value isn't the same as the previous value.

It's important to note that we don't need to explicitly handle duplicate 'c' values. The adjustments made to avoid duplicate 'a' and 'b' values ensure each pair $[a, b]$ is unique. Since 'c' is determined by the equation $c = -(a + b)$, each unique $[a, b]$ pair will result in a unique 'c' value. Therefore, by just avoiding duplicates in 'a' and 'b', we automatically avoid duplicates in the $[a, b, c]$ triplets.

Optimization

An interesting observation is that triplets that sum to 0 cannot be formed using positive numbers alone. Therefore, we can stop trying to find triplets once we reach a positive 'a' value since this implies that 'b' and 'c' would also be positive.

Implementation

From the above intuition, we know we need to slightly modify the `pair_sum_sorted` function to avoid duplicate triplets. We also need to pass in a start value to indicate the beginning of the subarray on which we want to perform the pair-sum algorithm. Otherwise, the two-pointer logic remains nearly identical to that of *Pair Sum - Sorted*.

```
def triplet_sum(nums: List[int]) -> List[List[int]]:
    triplets = []
    nums.sort()
    for i in range(len(nums)):
        # Optimization: triplets consisting of only positive numbers
        # will never sum to 0.
```

```

if nums[i] > 0:
    break
# To avoid duplicate triplets, skip 'a' if it's the same as
# the previous number.
if i > 0 and nums[i] == nums[i - 1]:
    continue
# Find all pairs that sum to a target of '-a' (-nums[i]).
pairs = pair_sum_sorted_all_pairs(nums, i + 1, -nums[i])
for pair in pairs:
    triplets.append([nums[i]] + pair)
return triplets

def pair_sum_sorted_all_pairs(nums: List[int],
                               start: int, target: int) -> List[int]:
    pairs = []
    left, right = start, len(nums) - 1
    while left < right:
        sum = nums[left] + nums[right]
        if sum == target:
            pairs.append([nums[left], nums[right]])
            left += 1
            # To avoid duplicate '[b, c]' pairs, skip 'b' if it's the
            # same as the previous number.
            while left < right and nums[left] == nums[left - 1]:
                left += 1
        elif sum < target:
            left += 1
        else:
            right -= 1
    return pairs

```

Complexity Analysis

Time complexity: The time complexity of `triplet_sum` is $O(n^2)$. Here's why:

- We first sort the array, which takes $O(n \log(n))$ time.
- Then, for each of the n elements in the array, we call `pair_sum_sorted_all_pairs` at most once, which runs in $O(n)$ time.

Therefore, the overall time complexity is $O(n \log(n)) + O(n^2) = O(n^2)$.

Space complexity: The space complexity is $O(n)$ due to the space taken up by Python's sorting algorithm. It's important to note that this complexity does not include the output array `triplets` because we're only concerned with the additional space used by the algorithm, not the space needed for the output itself.

If the interviewer asks what the space complexity would be if we included the output array, it would be $O(n^2)$. This is because the `pair_sum_sorted_all_pairs` function, in the worst case, can add approximately n pairs to the output. Since this function is called approximately n times, the overall space complexity would be $O(n^2)$.

Test Cases

In addition to the examples already covered in this explanation, below are some others to consider when testing your code.

Input	Expected output	Description
<code>nums = []</code>	<code>[]</code>	Tests an empty array.
<code>nums = [0]</code>	<code>[0]</code>	Tests a single-element array.
<code>nums = [1, -1]</code>	<code>[]</code>	Tests a two-element array.
<code>nums = [0, 0, 0]</code>	<code>[0, 0, 0]</code>	Tests an array where all three of its values are the same.
<code>nums = [1, 0, 1]</code>	<code>[]</code>	Tests an array with no triplets that sum to 0.
<code>nums = [0, 0, 1, -1, 1, -1]</code>	<code>[-1, 0, 1]</code>	Tests an array with duplicate triplets.

Is Palindrome Valid

A palindrome is a sequence of characters that reads the same forward and backward.

Given a string, determine if it's a palindrome after removing all non-alphanumeric characters. A character is alphanumeric if it's either a letter or a number.

Example 1:

Input: s = "a dog! a panic in a pagoda."
Output: True

Example 2:

Input: s = "abc123"
Output: False

Constraints:

The string may include a combination of lowercase English letters, numbers, spaces, and punctuations.

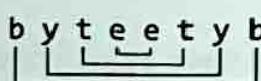
Intuition

Identifying palindromes

A string is a palindrome if it remains identical when read from left to right or right to left. In other words, if we reverse the string, it should still read the same, disregarding spaces and punctuation:

"racecar" "racecar"

An important observation is that if a string is a palindrome, the first character would be the same as the last, the second character would be the same as the second-to-last, etc:



A palindrome of odd length is different because it has a middle character. In this case, the middle character can be ignored since it has no "mirror" character elsewhere in the string.



Palindromes provides an ideal scenario for using two pointers (`left` and `right`). By initially setting the pointers at the beginning and end of the string, we can compare the characters at these positions. Ignoring non-alphanumeric characters for the moment, the logic can be summarized as follows:

- If the alphanumeric characters at `left` and `right` are the same, move both pointers inward to process the next pair of characters.
- If not, the string is not a palindrome: return false.

If we successfully compare all character pairs without returning false, the string is a palindrome, and we should return true.

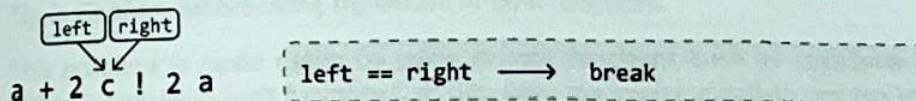
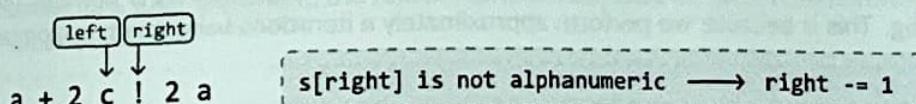
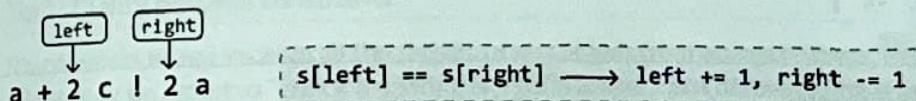
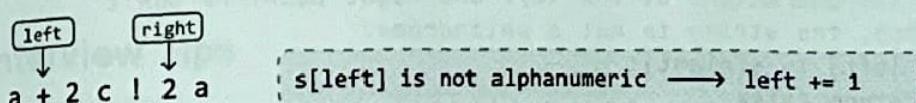
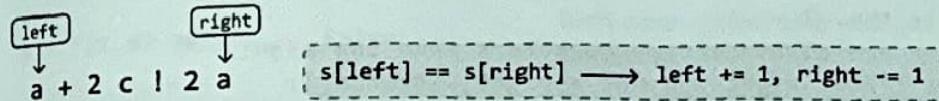
Processing non-alphanumeric characters

Now, let's explore how to find palindromes that include non-alphanumeric characters.

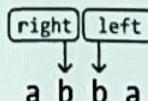
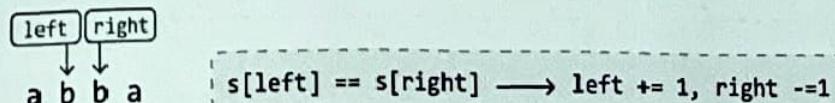
Since non-alphanumeric characters don't affect whether a string is a palindrome, we should skip them. This can be achieved with the following approach, which ensures the left and right pointers are adjusted to focus only on alphanumeric characters:

- Increment `left` until the character it points to is alphanumeric.
- Decrement `right` until the character it points to is alphanumeric.

With this in mind, let's check if the string below is a palindrome using all the information we know so far:



As shown above, when the left and right pointers meet, it signals our exit condition. When these pointers meet, we've reached the middle character of the palindrome, at which point we can exit the loop since the middle character doesn't need to be evaluated. However, we need to keep in mind that exiting when `left` equals `right` won't always be sufficient as an exit condition. For example, if the number of alphanumeric characters is even, the pointers won't meet. This can be observed below:



Therefore, we need to ensure we exit the loop when `left` equals `right`, or when `left` passes `right`. In other words, the algorithm continues while `left` is less than `right`:

```
while left < right:
```

Implementation

In Python, we can use the inbuilt `isalnum` method to check if a character is alphanumeric.

```
def is_palindrome_valid(s: str) -> bool:
    left, right = 0, len(s) - 1
    while left < right:
        # Skip non-alphanumeric characters from the left.
        while left < right and not s[left].isalnum():
            left += 1
        # Skip non-alphanumeric characters from the right.
        while left < right and not s[right].isalnum():
            right -= 1
        # If the characters at the left and right pointers don't
        # match, the string is not a palindrome.
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True
```

Complexity Analysis

Time complexity: The time complexity of `is_palindrome_valid` is $O(n)$, where n denotes the length of the string. This is because we perform approximately n iterations using the two-pointer technique.

Space complexity: We only allocated a constant number of variables, so the space complexity is $O(1)$.

Test Cases

In addition to the examples discussed, below are more examples to consider when testing your code.

Input	Expected output	Description
s = ""	True	Tests an empty string.
s = "a"	True	Tests a single-character string.
s = "aa"	True	Tests a palindrome with two characters.
s = "ab"	False	Tests a non-palindrome with two characters.
s = "!, (?)"	True	Tests a string with no alphanumeric characters.
s = "12.02.2021"	True	Tests a palindrome with punctuation and numbers.
s = "21.02.2021"	False	Tests a non-palindrome with punctuation and numbers.
s = "hello, world!"	False	Tests a non-palindrome with punctuation.

Interview Tips



Tip 1: Clarify problem constraints.

It's common to not receive all the details of a problem from an interviewer. For example, you might only be asked to "check if a string is a palindrome." But before diving into a solution, it's important to clarify details with the interviewer, such as the presence of non-alphanumeric characters, their treatment, the role of numbers, the case sensitivity of letters, and other relevant details.



Tip 2: Confirm before using significant in-built functions.

This problem is made easier by using in-built functions such as `.isalnum` (or equivalent). Before using an in-built function that simplifies the implementation, ask the interviewer if it's okay to use it, or if they would prefer you implement it yourself.

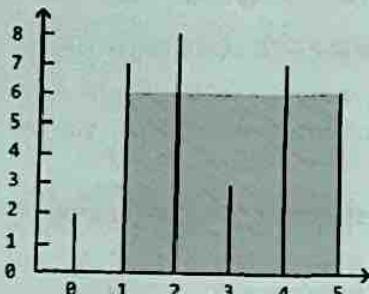
The interviewer will most likely allow the use of an in-built function, or ask you to implement it as an exercise for later in the interview. If you use an in-built function, make sure you understand its time and space complexity.

Remember that interviewers are looking for team players, and this shows them you're considerate of their preferences and can adapt your approach based on the requirements.

Largest Container

You are given an array of numbers, each representing the height of a vertical line on a graph. A container can be formed with any pair of these lines, along with the x-axis of the graph. Return the amount of water which the largest container can hold.

Example:

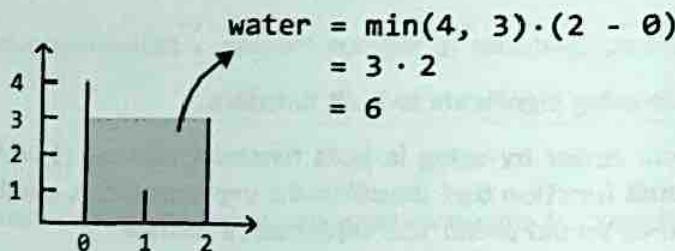


Input: heights = [2, 7, 8, 3, 7, 6]

Output: 24

Intuition

If we have two vertical lines, `heights[i]` and `heights[j]`, the amount of water that can be contained between these two lines is $\min(\text{heights}[i], \text{heights}[j]) * (j - i)$, where $j - i$ represents the width of the container. We take the minimum height because filling water above this height would result in overflow.



In other words, the area of the container depends on two things:

- The width of the rectangle.
- The height of the rectangle, as dictated by the shorter of the two lines.

The brute force approach to this problem involves checking all pairs of lines, and returning the largest area found between each pair:

```
def largest_container_brute_force(heights: List[int]) -> int:
    n = len(heights)
    max_water = 0
    # Find the maximum amount of water stored between all pairs of
    # lines.
    for i in range(n):
        for j in range(i + 1, n):
```

```

        water = min(heights[i], heights[j]) * (j - i)
        max_water = max(max_water, water)
    return max_water

```

Searching through all possible pairs of values takes $O(n^2)$ time, where n denotes the length of the array. Let's look for a more efficient solution.

We would like both the height and width to be as large as possible to have the largest container.

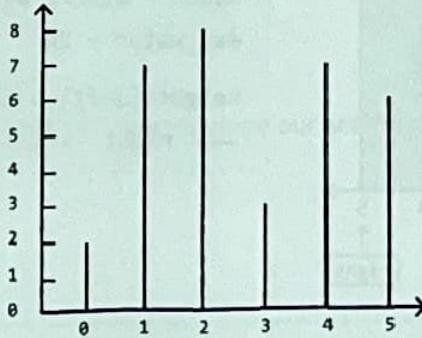
It's not immediately obvious how to find the container with the largest height, as the heights of the lines in the array don't follow a clear pattern. However, we do know the container with the maximum width: the one starting at index 0 and ending at index $n - 1$.

So, we could start by maximizing the width by setting a pointer at each end of the array. Then, we can gradually reduce the width by moving these two pointers inward, hoping to find a container with a larger height that potentially yields a larger area. This suggests we can use the two-pointer pattern to solve the problem.

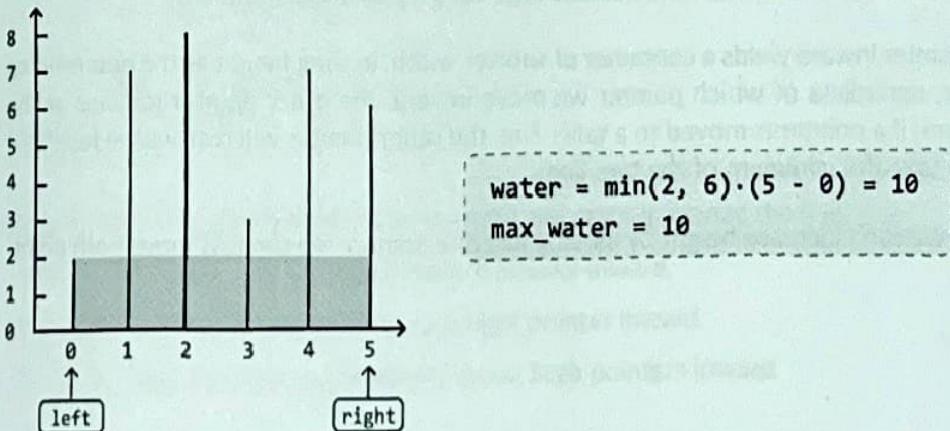
Moving a pointer inward means shifting either the left pointer to the right, or the right pointer to the left, effectively narrowing the gap between them.

Consider the following example:

`heights = [2 7 8 3 7 6]:`

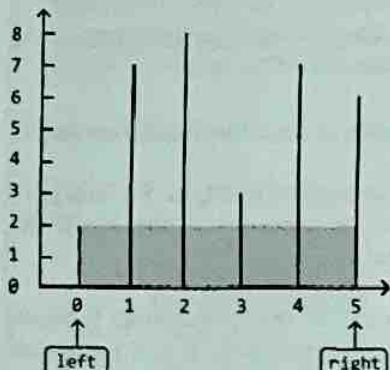


The widest container can store an area of water equal to 10. Since this is the largest container we've found so far, let's set `max_water` to 10.



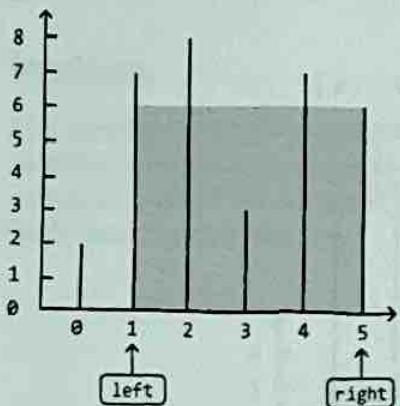
How should we proceed? Moving either pointer inward yields a container with a shorter width.

This leaves height as the determining factor. In this case, the left line is shorter than the right line, which means that the left line limits the water's height. Therefore, to find a larger container, let's move the left pointer inward:



```
heights[left] < heights[right]  
→ left += 1
```

The current container can hold 24 units of water, the largest amount so far. So, let's update `max_water` to 24. Here, the right line is shorter, limiting the water's height. To find a larger container, move the right pointer inward:

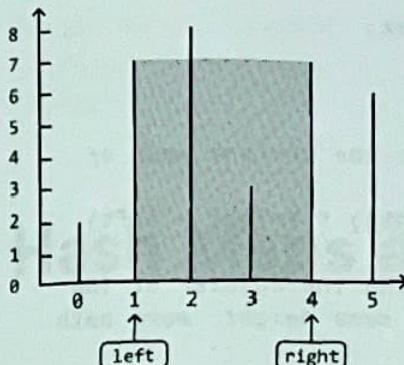


```
water = min(7, 6) · (5 - 1) = 24  
max_water = 24  
heights[left] > heights[right]  
→ right -= 1
```

After this, we encounter a situation where the height of the left and right lines are equal. In this situation, which pointer should we move inward? Well, regardless of which one, the next container is guaranteed to store less water than the current one. Let's try to understand why.

Moving either pointer inward yields a container of shorter width, leaving height as the determining factor. However, regardless of which pointer we move inward, the other pointer remains at the same line. So, even if a pointer is moved to a taller line, the other pointer will restrict the height of the water, as we take the minimum of the two lines.

Therefore, since we can't increase height by moving just one pointer, we can just move both pointers inward:



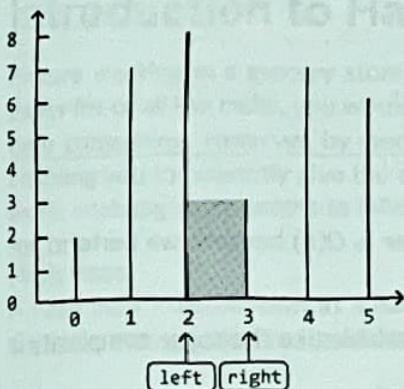
```

water = min(7, 1) · (4 - 1) = 21
max_water = 24

heights[left] == heights[right]
→ left += 1 and right -= 1

```

Now, the right line is limiting the height of the water. So, we move the right pointer inward:



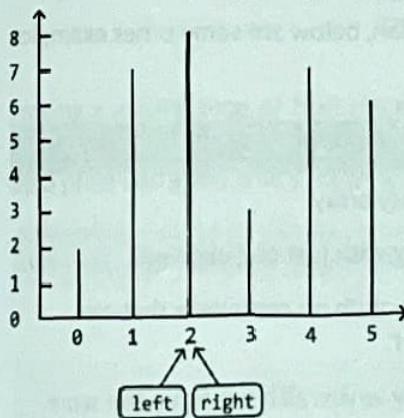
```

water = min(8, 3) · (3 - 2) = 3
max_water = 24

heights[left] > heights[right]
→ right -= 1

```

Finally, the left and right pointers meet. We can conclude our search here and return max_water:



```
left == right → break
```

Based on the decisions taken in the example, we can summarize the logic:

1. If the left line is smaller, move the left pointer inward.
2. If the right line is smaller, move the right pointer inward.
3. If both lines have the same height, move both pointers inward.

Implementation

```

def largest_container(heights: List[int]) -> int:
    max_water = 0
    left, right = 0, len(heights) - 1
    while (left < right):
        # Calculate the water contained between the current pair of
        # lines.
        water = min(heights[left], heights[right]) * (right - left)
        max_water = max(max_water, water)
        # Move the pointers inward, always moving the pointer at the
        # shorter line. If both lines have the same height, move both
        # pointers inward.
        if (heights[left] < heights[right]):
            left += 1
        elif (heights[left] > heights[right]):
            right -= 1
        else:
            left += 1
            right -= 1
    return max_water

```

Complexity Analysis

Time complexity: The time complexity of `largest_container` is $O(n)$ because we perform approximately n iterations using the two-pointer technique.

Space complexity: We only allocated a constant number of variables, so the space complexity is $O(1)$.

Test Cases

In addition to the examples discussed throughout this explanation, below are some other examples to consider when testing your code.

Input	Expected output	Description
<code>heights = []</code>	0	Tests an empty array.
<code>heights = [1]</code>	0	Tests an array with just one element.
<code>heights = [0, 1, 0]</code>	0	Tests an array with no containers that can contain water.
<code>heights = [3, 3, 3, 3]</code>	9	Tests an array where all heights are the same.
<code>heights = [1, 2, 3]</code>	2	Tests an array with strictly increasing heights.
<code>heights = [3, 2, 1]</code>	2	Tests an array with strictly decreasing heights.

Hash Maps and Sets

Introduction to Hash Maps and Sets

Picture working in a grocery store. A customer asks for the price of a fruit. If you just have a paper list of all the fruits, you would need to look through it to find the specific fruit, which can be time-consuming. However, by memorizing the list, you can know the prices of all fruits instantly, enabling you to promptly give the correct price to the customer. This is similar to how hash maps work, enabling quick access to information.

Hash maps

A hash map – also known as a hash table or dictionary, depending on the language – is a data structure that pairs keys with values. Let's illustrate this using the example of fruits and their prices:

Fruit	Price
apple	5.00
bananas	4.50
carrot	2.75

Having a mental map of fruit prices is conceptually similar to being able to instantly access fruit prices using a hash map, where the fruit name is the key, and its price is the value. When we look up a price using the fruit's name as the key, the hash map will immediately return its price:

hashmap	
apple	5.00
bananas	4.50
carrot	2.75

key value

Hash maps are incredibly efficient for lookups, insertions, and deletions, as they typically perform these operations in constant time: $O(1)$. They're one of the most versatile, widely-used data structures in computer science, used for tasks such as counting the frequency of elements, caching data, and more.

Properties of hash maps

- Data is stored in the form of key-value pairs.
- Hash maps don't store duplicates. Every key in a hash map is unique, ensuring each value can

be distinctly identified and accessed.

- Hash maps are unordered data structures, meaning keys are not stored in any specific order.

Time complexity breakdown

Below, n denotes the number of entries in the hash map.

Operation	Average case	Worst case	Description
Insert	$O(1)$	$O(n)$	Add a key-value pair to the hash map.
Access	$O(1)$	$O(n)$	Find or retrieve an element.
Delete	$O(1)$	$O(n)$	Delete a key-value pair.

In coding interviews, we generally consider hash map operations to have a fast average time complexity of $O(1)$, as opposed to their worst-case complexities. This is based on the assumption that an efficient hash function minimizes collisions [1]. However, in the worst-case scenario where a poorly optimized hash function results in frequent collisions, the time complexity can deteriorate to $O(n)$, necessitating a linear search through all entries.

Hash sets

Hash sets are a simpler form of hash maps. Instead of storing key-value pairs, they store only the keys. Using the grocery store analogy, a hash set is like having a mental checklist of fruits without their prices. It's useful for quickly checking the presence or absence of an item, like checking whether a particular fruit is in stock.

When to use hash maps or sets

Common use cases of hash maps include implementing dictionaries, counting frequencies, storing key-value pairs, and handling scenarios requiring quick lookups.

Common use cases of hash sets include storing unique elements, marking elements as used or visited, and checking for duplicates.

In the description of a problem, pay attention to keywords like "frequency", "unique", "map", "dictionary", or "fast lookup", because these often indicate that hash maps or sets could be useful.

Essential concepts for mastering hash maps and sets

This chapter discusses the practical uses of hash maps and sets. For a more complete understanding of how they work and why they're so efficient, please explore topics that go beyond the scope of this chapter, such as:

- Hash functions: explore the intricacies of how keys are mapped to specific values in a hash table [2].
- Collision and collision-handling techniques: understand methods like chaining, or open addressing for resolving hash collisions [1].
- Load factors and rehashing: these make it easier to understand how hash tables grow and change size [3].

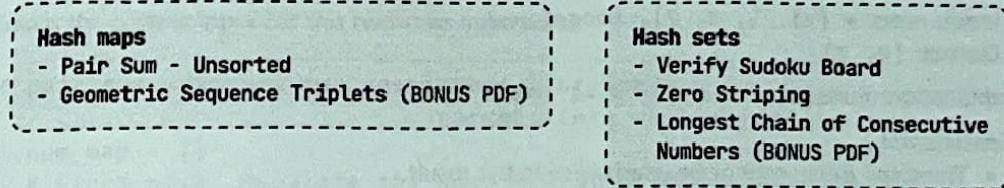
Real-world Example

Web browser cache: Hash maps and sets are used everywhere in real-world systems. A classic example of hash maps in action is in caching systems within web browsers. When you visit a website, your browser stores data such as images, HTML, and CSS files in a cache so it can load

much faster on future visits.

Chapter Outline

Hash Maps and Sets



Practice these problems on our online code editor
→ bit.ly/run-code

</>

To receive the bonus PDF, sign up for our Coding Interview Patterns newsletter by using the link below:

bit.ly/coding-patterns-pdf

References

- [1] Hash collisions: https://en.wikipedia.org/wiki/Hash_collision
- [2] Hash functions: https://en.wikipedia.org/wiki/Hash_function
- [3] Load factor and rehashing: <https://www.scaler.com/topics/data-structures/load-factor-and-rehashing/>

Pair Sum - Unsorted

Given an array of integers, return the indexes of any two numbers that add up to a target. The order of the indexes in the result doesn't matter. If no pair is found, return an empty array.

Example:

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

Output: `[0, 2]`

Explanation: `nums[0] + nums[2] = -1 + 4 = 3`

Constraints:

- The same index cannot be used twice in the result.

Intuition

A brute force approach is to iterate through every possible pair in the array to see if their sum is equal to the target. This is the same as the brute force solution described in the *Pair Sum - Sorted* problem, which has a time complexity of $O(n^2)$, where n is the length of the array. We could also sort the array and then perform the two-pointer algorithm used in *Pair Sum - Sorted*, which would take $O(n \log(n))$ time due to sorting. Let's see if we can find an even faster solution.

Complement

We're asked to find a pair (x, y) such that $x + y == \text{target}$. In this equation, there are two unknowns: x and y . An important observation is that if we know one of these numbers, we can easily calculate what the other number should be.

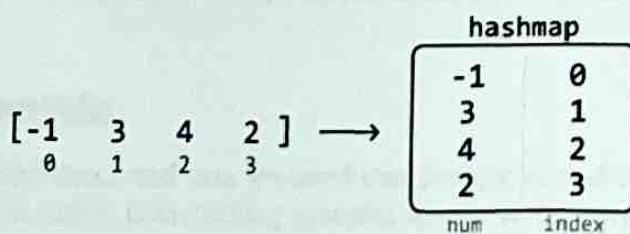
For each number x in `nums`, we need to find another number y such that $x + y = \text{target}$, or in other words, $y = \text{target} - x$. We can call this number the **complement** of x .

Keep in mind, we need to return the indexes of the pair of numbers, not the pair itself. So, we'll need a way to find a number's complement as well as its index.

One way we could do this is to loop through the array to find each number's complement and corresponding index. But this takes $O(n^2)$ time since we'd need to do a linear traversal to search for each number's complement. Instead, we'd like an efficient way to determine the index of any number in the array without needing to search the array. Is there a data structure that can help with this?

Hash map

A hash map works great because we can store and look up values in $O(1)$ time. Each number and its index can be stored in the hash map as key-value pairs:



This allows us to retrieve the index of any number's complement efficiently. Notice that duplicate numbers don't need to be considered here since only one valid pair needs to be found.

The most intuitive way to incorporate a hash map is to:

1. In the first pass, populate the hash map with each number and its corresponding index.
2. In the second pass, scan through the array to check if each number's complement exists in the hash map. If it does, we can return the indexes of that number and its complement.

Below is the code snippet for this two-pass approach:

```
def pair_sum_unsorted_two_pass(nums: List[int],
                                target: int) -> List[int]:
    num_map = {}
    # First pass: Populate the hash map with each number and its
    # index.
    for i, num in enumerate(nums):
        num_map[num] = i
    # Second pass: Check for each number's complement in the hash map.
    for i, num in enumerate(nums):
        complement = target - num
        if complement in num_map and num_map[complement] != i:
            return [i, num_map[complement]]
    return []
```

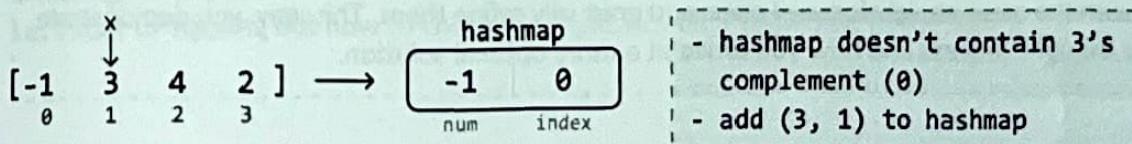
This algorithm requires two passes. Is it possible to do this in only one? A one-pass solution implies that we would need to populate the hash map while searching for complements. Is this possible? Consider the example below:

$[-1 \quad 3 \quad 4 \quad 2]$, target = 3
 0 1 2 3

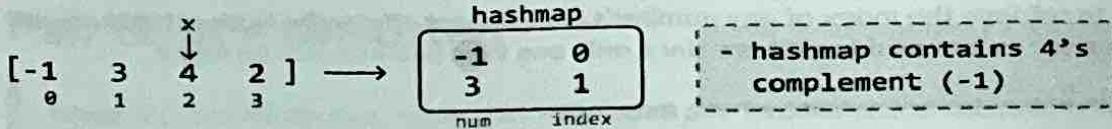
Start at index 0. Its complement would be $3 - (-1) = 4$. Does our hash map have 4 in it? No, it's empty at the moment. So, let's add -1 and its index to the hash map:



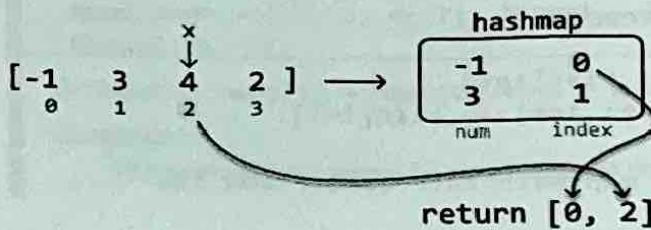
Next, let's look at index 1. Its complement (0) does not exist in the hash map. So, just add 3 and its index to the hash map:



At index 2, we notice 4's complement (-1) exists in the hash map. This means we found a pair that sums to the target:



Now, we can return the indexes of the two values. Fetch the index of 4 from the input array and the index of its complement from the hash map:



Implementation

```
def pair_sum_unsorted(nums: List[int], target: int) -> List[int]:
    hashmap = {}
    for i, x in enumerate(nums):
        if target - x in hashmap:
            return [hashmap[target - x], i]
        hashmap[x] = i
    return []
```

Complexity Analysis

Time complexity: The time complexity of `pair_sum_unsorted` is $O(n)$ because we iterate through each element in the `nums` array once and perform constant-time hash map operations during each iteration.

Space complexity: The space complexity is $O(n)$ since the hash map can grow up to n in size.

Interview Tip

Tip: Iterate through solutions.



Don't always jump straight to the most optimal or clever solution, as this won't give the interviewer much insight into your problem-solving process. Consider multiple approaches, starting with the more straightforward ones, and gradually refine them. This way, you demonstrate your thought process and how you arrive at a more optimal solution.

Verify Sudoku Board

Given a partially completed 9×9 Sudoku board, determine if the current state of the board adheres to the rules of the game:

- Each row and column must contain unique numbers between 1 and 9, or be empty (represented as 0).
- Each of the nine 3×3 subgrids that compose the grid must contain unique numbers between 1 and 9, or be empty.

Note: You are asked to determine whether the current state of the board is valid given these rules, not whether the board is solvable.

Example:

3	6		5	8	4			
5	2							
8	7					3	1	
1	2	5			3	2		
9		8	6	3				5
5			9		6			
3			8	2	5			
1						7	4	
5	2	6						

two 2s in the same row
two 8s in the same column ➔ return False

Output: False

Constraints:

- Assume each integer on the board falls in the range of $[0, 9]$.

Intuition

Our primary objective is to check every row, column, and each of the nine 3×3 subgrids, for any duplicate numbers. Let's first discuss the mechanism for finding duplicate elements, then look into how we can apply this to rows, columns, and subgrids.

Checking for duplicates

Let's start by figuring out how to check for duplicates on a single row of the board:

A 9x9 grid of numbers. The grid is as follows:

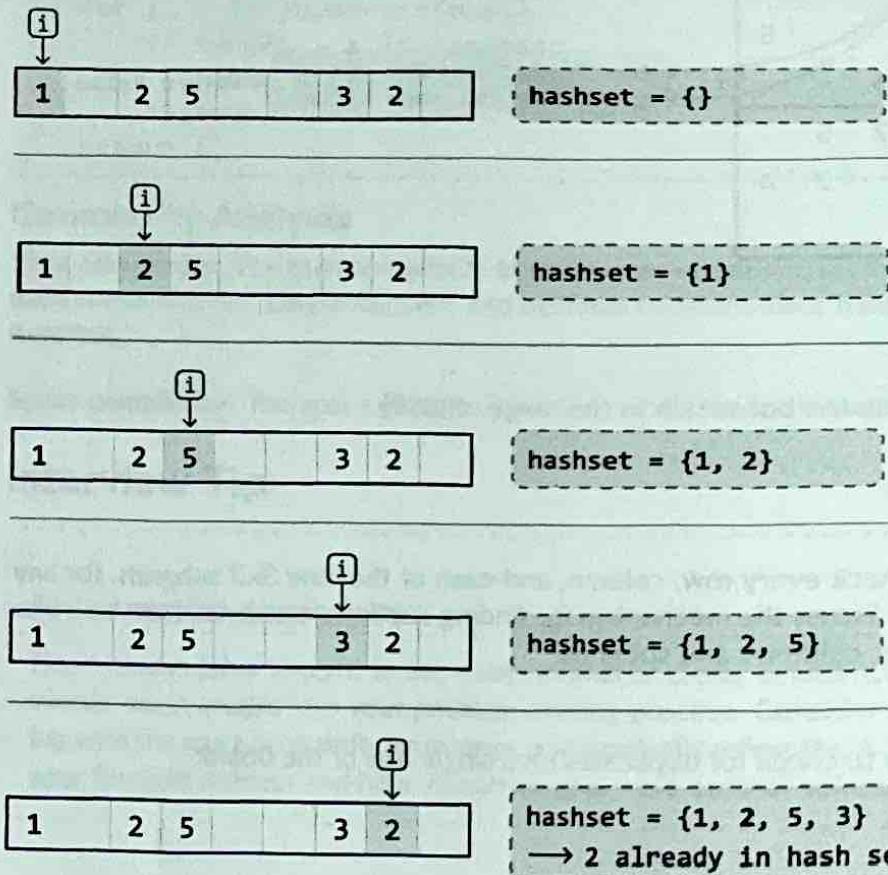
3	6		5	8	4			
5	2							
	8	7				3	1	
1		2	5		3	2		
9			8	6	3			5
	5			9		6		
	3				8	2	5	
1						7	4	
	5	2	6					

The horizontal arrow indicates a transformation to a row-major representation of the array:

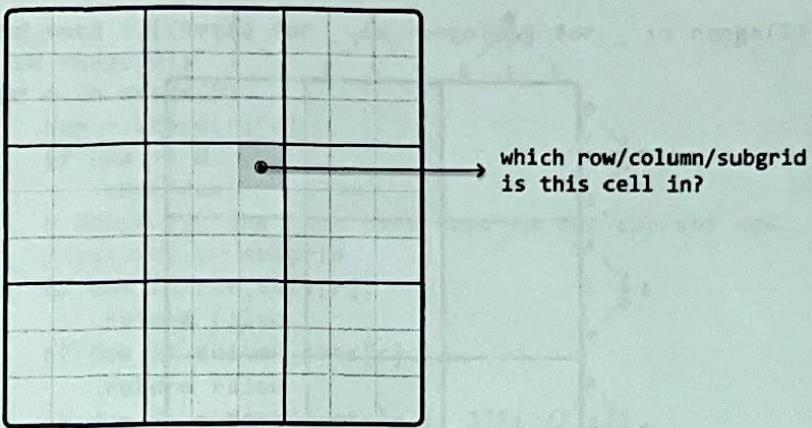
1	2	5			3	2		
---	---	---	--	--	---	---	--	--

A naive way to do this is to take each number in the row and search the row to see if that number appears again. Performing a linear search for each number would result in a time complexity of $O(n^2)$ to check all numbers in one row, which is quite time consuming.

We can use a hash set to improve the time complexity. By using a hash set, we can keep track of which numbers were previously visited as we iterate through the row. When we encounter a new number, we can check if it's already in the set in $O(1)$ time. If it is, then it's a duplicate:



If we had a hash set for each of the 9 rows, we could keep track of duplicates in each row separately. We can do this for columns and subgrids as well, with one hash set for each column, and one hash set for each subgrid. The challenge here is determining which hash sets correspond to each cell's row, column, and subgrid, so we know which hash sets to reference:



So, let's discuss how we can identify a cell's row, column, or subgrid.

Identifying rows and columns

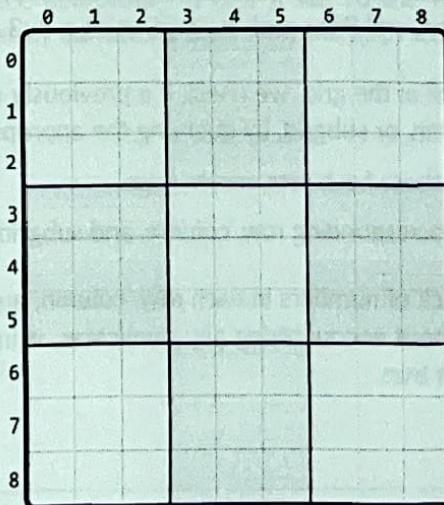
Identifying rows is straightforward because each row has an index. The same applies to columns. Therefore, we can create an array of 9 hash sets for each row, allowing us to access a row's hash set directly by its index. Similarly, we can set up an array of hash sets for each column.

```
row_sets = [hashset(), hashset(), ..., hashset()]
           0          1          8
col_sets = [hashset(), hashset(), ..., hashset()]
           0          1          8
```

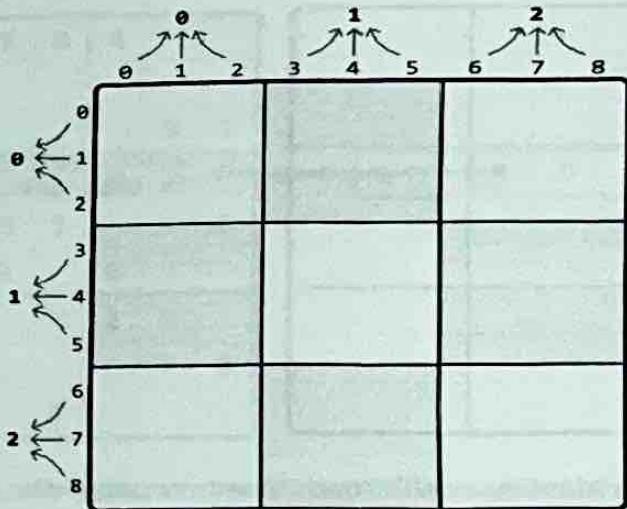
Identifying subgrids

Subgrids pose an interesting challenge because we can't immediately identify which subgrid a cell belongs to, unlike the straightforward index-based identification for rows and columns.

That said, as with rows and columns, there are still only 9 subgrids. If we visualize the subgrids, we can see them displayed in a 3×3 grid:



What we'd like is a way to index each of these subgrids as if indexing a 3×3 matrix. To do this, we require a method to convert the indexes ranging from 0 to 8 to the corresponding adjusted indexes from 0 to 2, as illustrated below:



Since we got these adjusted indexes from shrinking a 9×9 grid to a 3×3 grid – which is effectively dividing the number of rows and columns by 3 – we can get the new subgrid row and column indexes by dividing by 3 (using integer division), as well:

$$\begin{array}{ccccccc}
 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
 \downarrow & \downarrow \\
 \div 3 & & & \div 3 & & \div 3 & & \div 3 & \\
 =0 & & & =1 & & =2 & & &
 \end{array}$$

With these modified indexes, we can organize nine hash sets within a 3×3 table, one for each subgrid. Each cell in this table represents the corresponding subgrid in the above 3×3 representation. So, we can access the hash set of a subgrid at any cell by using our adjusted indexes (i.e., `subgrid_sets[r // 3][c // 3]`) (the `//` operator performs integer division).

One-pass Sudoku verification

We now have everything needed for a one-pass solution. We can start by initializing hash sets, 9 for each row, 9 for each column, and 9 for each subgrid, using a 3×3 array.

As we iterate through each cell in the grid, we check if a previously encountered number already exists in the current row, column, or subgrid, by querying the appropriate hash sets:

- If the number is in any of these hash sets, return false.
- Otherwise, add it to the corresponding row, column, and subgrid hash sets.

This process helps us keep track of numbers in each row, column, and subgrid. If we successfully iterate through the board without encountering any duplicates, it indicates the Sudoku board is valid. Therefore, we can return true.

Implementation

```

def verify_sudoku_board(board: List[List[int]]) -> bool:
    # Create hash sets for each row, column, and subgrid to keep
    # track of numbers previously seen on any given row, column, or
    # subgrid.
    row_sets = [set() for _ in range(9)]
    column_sets = [set() for _ in range(9)]

```

```

subgrid_sets = [[set() for _ in range(3)] for _ in range(3)]
for r in range(9):
    for c in range(9):
        num = board[r][c]
        if num == 0:
            continue
        # Check if 'num' has been seen in the current row,
        # column, or subgrid.
        if num in row_sets[r]:
            return False
        if num in column_sets[c]:
            return False
        if num in subgrid_sets[r // 3][c // 3]:
            return False
        # If we passed the above checks, mark this value as seen
        # by adding it to its corresponding hash sets.
        row_sets[r].add(num)
        column_sets[c].add(num)
        subgrid_sets[r // 3][c // 3].add(num)
return True

```

Complexity Analysis

In this problem, the length of the board is fixed at 9, effectively reducing all approaches to a time and space complexity of $O(1)$. However, to better understand the efficiency of our algorithm in a broader context, let's use n to denote the board's length, allowing us to evaluate the algorithm's performance against arbitrary board sizes.

Time complexity: The time complexity of `verify_sudoku_board` is $O(n^2)$ because we iterate through each cell in the board once, and perform constant-time hash set operations.

Space complexity: The space complexity is $O(n^2)$ due to the `row_sets`, `column_sets`, and `subgrid_sets` arrays. Each array contains n hash sets, and each hash set is capable of growing to a size of n .

Implementation - Hash Sets

Zero Striping

For each zero in an $m \times n$ matrix, set its entire row and column to zero in place.

Example:

	0	1	2	3	4	5
0	1	2	3	4	5	
1	6	0	8	9	10	
2	11	12	13	14	15	
3	16	17	18	19	0	



	0	1	2	3	4	
0	1	0	3	4	0	
1	0	0	0	0	0	
2	11	0	13	14	0	
3	0	0	0	0	0	

Intuition – Hash Sets

A brute-force solution involves recording the positions of all 0s initially in the matrix and, for each of these 0s, iterating over their row and column to set them to zero. However, imagine an input array that's filled with many zeros. In the worst case, iterating over every row and column for each zero will take $O(m \cdot n \cdot (m + n))$, where $m \cdot n$ denotes the number of 0s, and $(m + n)$ represents the total number of cells in a row and column combined. This approach is quite inefficient, so let's look for a better solution.

Imagine any cell in the matrix. After the matrix is transformed, this cell will either retain its original value, or become zero. Is there a way to tell if a cell is going to become zero?

The key observation is that if a cell is in a row or column containing a zero, that cell will become zero.

We could search a cell's row and column to check if they contain a zero, meaning each search would take $O(m + n)$ time. But it would be more efficient if we had a way to check this in constant time, and this is where hash sets would be useful. If we create two hash sets – one to track all the rows containing a zero and another to track all the columns containing a zero – we can determine if a specific cell's row or column contains a zero in $O(1)$ time.

With these hash sets created, the next step is to populate them. As we iterate through the matrix, when encountering a cell containing zero, we:

- Add its row index to the row hash set (`zero_rows`).
- Add its column index to the column hash set (`zero_cols`).

	0	1	2	3	4
0	1	2	3	4	5
1	6	0	8	9	10
2	11	12	13	14	15
3	16	17	18	19	0

O → 1 O → 3

O zero_rows = {1, 3}
 □ zero_cols = {1, 4}

Next, we identify the cells whose row or column indexes are present in the respective hash sets, and change their values to zero. Let's look at how this works with a few examples:

	0	1	2	3	4
0	1	2	3	4	5
1	6	0	8	9	10
2	11	12	13	14	15
3	16	17	18	19	0

zero_rows = {1, 3}
 zero_cols = {1, 4}

cell(1, 2):
 row 1 is in zero_rows
 → set to 0

	0	1	2	3	4
0	1	2	3	4	5
1	6	0	0	9	10
2	11	12	13	14	15
3	16	17	18	19	0

	0	1	2	3	4
0	1	2	3	4	5
1	6	0	8	9	10
2	11	12	13	14	15
3	16	17	18	19	0

zero_rows = {1, 3}
 zero_cols = {1, 4}

cell(2, 4):
 col 4 is in zero_cols
 → set to 0

	0	1	2	3	4
0	1	2	3	4	5
1	6	0	8	9	10
2	11	12	13	14	0
3	16	17	18	19	0

	0	1	2	3	4
0	1	2	3	4	5
1	6	0	8	9	10
2	11	12	13	14	15
3	16	17	18	19	0

zero_rows = {1, 3}
 zero_cols = {1, 4}

cell(2, 2):
 neither row 2 or col 2 are in the hash sets
 → don't set to 0

This provides a general strategy:

1. In one pass of the matrix, identify each cell containing a zero and add its row and column indexes to the zero_rows and zero_cols hash sets, respectively.
2. In a second pass, set any cell to zero if its row index is in zero_rows or its column index is in zero_cols.

Implementation – Hash Sets

```
def zero_striping_hash_sets(matrix: List[List[int]]) -> None:
    if not matrix or not matrix[0]:
        return
    m, n = len(matrix), len(matrix[0])
    zero_rows, zero_cols = set(), set()
    # Pass 1: Traverse through the matrix to identify the rows and
    # columns containing zeros and store their indexes in the
    # appropriate hash sets.
    for r in range(m):
        for c in range(n):
            if matrix[r][c] == 0:
                zero_rows.add(r)
                zero_cols.add(c)
    # Pass 2: Set any cell in the matrix to zero if its row index is
    # in 'zero_rows' or its column index is in 'zero_cols'.
    for r in range(m):
        for c in range(n):
            if r in zero_rows or c in zero_cols:
                matrix[r][c] = 0
```

Complexity Analysis

Time complexity: The time complexity of `zero_striping_hash_sets` is $O(m \cdot n)$ because we perform two passes over the matrix and perform constant-time operations in each pass.

Space complexity: The space complexity is $O(m + n)$ due to the growth of the hash sets used to track zeros: one hash set scales with the number of rows, and the other scales with the number of columns. In the worst case, every row and column has a zero.

Intuition – In-place Zero Tracking

The previous solution was time efficient mainly due to the use of hash sets. However, this came at the cost of extra space used to store the hash set values. Is there an alternate way to keep track of which rows and columns contain a zero?

A key observation is that if a row or column contains a zero, all the cells in that row or column will be eventually replaced by zero. Therefore, there's no need to preserve the values in these rows or columns.

A strategy we can try is to use the **first row and column (row 0 and column 0)** as markers to track which rows and columns contain zeros.

To understand how this would work, consider the example below. For rows, we can use the first column to mark the rows that contain a zero. Specifically, this means if any cell in a row is zero, we set the corresponding cell in the first column to zero. This zero in the first column serves as a marker to indicate the entire row should eventually be set to zeros.

	0	1	2	3	4
0	1	2	3	4	5
1	6	0	8	9	10
2	11	12	13	14	15
3	16	17	18	19	0

use the first column to mark rows that have zeros

	0	1	2	3	4
0	1	2	3	4	5
1	0 ← 0	8	9	10	
2	11	12	13	14	15
3	0 ← 17	18	19	0	

Similarly to how we marked rows, we can mark columns containing zeros using the first row:

	0	1	2	3	4
0	1	2	3	4	5
1	6	0	8	9	10
2	11	12	13	14	15
3	16	17	18	19	0

use the first row to mark columns that have zeros

	0	1	2	3	4
0	1	0	3	4	0
1	6	0	8	9	10
2	11	12	13	14	15
3	16	17	18	19	0

To set markers for the first row and first column, we can begin searching the rest of the matrix, excluding the first row and first column, for any zero-valued cells. Let's refer to this part of the matrix as the 'submatrix.' When we find a zero, we set the corresponding cell in the first row and column to zero. Scanning every cell in the submatrix for zeros allows us to set markers in the first row and first column:

	0	1	2	3	4
0	1	0	3	4	0
1	0 ← 0	8	9	10	
2	11	12	13	14	15
3	0 ← 17	18	19	0	

Now, we should start converting cells in the submatrix to zeros based on their corresponding markers. We can assess any cell in the submatrix by checking:

- Whether its corresponding marker in the first column is zero.
- Whether its corresponding marker in the first row is zero.

If either of these conditions are met, we should set that cell's value to zero, as shown below:

	0	1	2	3	4
0	1	0	3	4	0
1	0	0	8	9	10
2	11	12	13	14	15
3	0	17	18	19	0

cell(1, 2):
The corresponding marker
at the first column is 0
→ set to 0

	0	1	2	3	4
0	1	0	3	4	0
1	0	0	0	9	10
2	11	12	13	14	15
3	0	17	18	19	0

To update this submatrix, we can iterate from the second row and column and update cell values based on the logic we just mentioned:

	0	1	2	3	4
0	1	0	3	4	0
1	0	0	0	0	10
2	11	12	13	14	15
3	0	17	18	19	0

	0	1	2	3	4
0	1	0	3	4	0
1	0	0	0	0	0
2	11	0	13	14	0
3	0	0	0	0	0

Handling zeros in the first row and column

After completing the previous step, there's just one issue to address. What if the first row or column originally had a zero, like in the example below?

	0	1	2	3	4
0	1	3	3	0	5
1	6	0	8	9	10
2	11	12	13	14	15
3	16	17	18	19	0

Here, we can't distinguish which zero in the first row was originally present, or resulted from being used as a marker. This means we won't know if the first row should be zeroed:

	0	1	2	3	4
0	1	0	3	0	0
1	6	0	8	9	10
2	11	12	13	14	15
3	16	17	18	19	0

→ can't tell which zero is a marker or was originally there

The remedy for this is to flag whether a zero exists in the first row or first column before using them as markers.

	0	1	2	3	4
0	1	0	3	0	0
1	6	0	8	9	10
2	11	12	13	14	15
3	16	17	18	19	0

first_row_has_zero = True
 first_col_has_zero = False

Once we've filled the first row and column with markers, as shown in matrix X below, and set the appropriate cell values in the submatrix to zero, as shown in matrix Y, we then evaluate the first row and column separately. The first row was initially marked as containing a zero, so we convert all cells in the first row to zero (matrix Z). The first column was not flagged for having a zero initially, so it remains unaltered at this step:

	0	1	2	3	4
0	1	0	3	0	0
1	0	0	8	9	10
2	11	12	13	14	15
3	0	17	18	19	0

→

	0	1	2	3	4
0	1	0	3	0	0
1	0	0	0	0	0
2	11	0	13	0	0
3	0	0	0	0	0

→

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	11	0	13	0	0
3	0	0	0	0	0

X **Y** **Z**

In-place zero-marking strategy

Let's summarize the above approach into the following steps:

1. Use a flag to indicate if the first row initially contains any zero.
2. Use a flag to indicate if the first column initially contains any zero.
3. Traverse the submatrix, setting zeros in the first row and column to serve as markers for rows and columns that contain zeros.
4. Apply zeros based on markers: iterate through the submatrix that starts from the second row and second column. For each cell, check if its corresponding marker in the first row or column is marked with a zero. If so, set that element to zero.
5. If the first row was initially marked as containing a zero, set all elements in the first row to zero.
6. If the first column was initially marked as having a zero, set all elements in the first column to zero.

Implementation – In-place Zero Tracking

```
def zero_striping(matrix: List[List[int]]) -> None:
    if not matrix or not matrix[0]:
        return
    m, n = len(matrix), len(matrix[0])
    # Check if the first row initially contains a zero.
    first_row_has_zero = False
    for c in range(n):
        if matrix[0][c] == 0:
            first_row_has_zero = True
            break
    for r in range(1, m):
        for c in range(1, n):
            if matrix[r][c] == 0:
                matrix[r][c] = matrix[0][c]
    if first_row_has_zero:
        for c in range(n):
            matrix[0][c] = 0
```

```

# Check if the first column initially contains a zero.
first_col_has_zero = False
for r in range(m):
    if matrix[r][0] == 0:
        first_col_has_zero = True
        break
# Use the first row and column as markers. If an element in the
# submatrix is zero, mark its corresponding row and column in the
# first row and column as 0.
for r in range(1, m):
    for c in range(1, n):
        if matrix[r][c] == 0:
            matrix[0][c] = 0
            matrix[r][0] = 0
# Update the submatrix using the markers in the first row and
# column.
for r in range(1, m):
    for c in range(1, n):
        if matrix[0][c] == 0 or matrix[r][0] == 0:
            matrix[r][c] = 0
# If the first row had a zero initially, set all elements in the
# first row to zero.
if first_row_has_zero:
    for c in range(n):
        matrix[0][c] = 0
# If the first column had a zero initially, set all elements in
# the first column to zero.
if first_col_has_zero:
    for r in range(m):
        matrix[r][0] = 0

```

Complexity Analysis

Time complexity: The time complexity of `zero_striping` is $O(m \cdot n)$. Here's why:

- Checking the first row for zeros takes $O(m)$ time, and checking the first column takes $O(n)$ time.
- Then, we perform two passes of the entire matrix, one to mark 0s and another to update the matrix based on those markers. Each pass takes $O(m \cdot n)$ time.
- Finally, we iterate through the first row and first column up to once each, which takes $O(m)$ and $O(n)$ time, respectively.

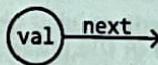
Therefore, the overall time complexity is $O(m) + O(n) + O(m \cdot n) = O(m \cdot n)$.

Space complexity: The space complexity is $O(1)$ because we use the first row and column as markers to track which rows and columns contain zeros, instead of using auxiliary data structures.

Linked Lists

Introduction to Linked Lists

A linked list is a data structure consisting of a sequence of nodes, where each node is linked to the next. A node in a linked list has two main components: the data it stores (`val`) and a reference to the next node (`next`) in the sequence:

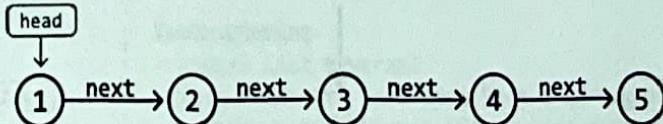


We define a node using the `ListNode` class, as below:

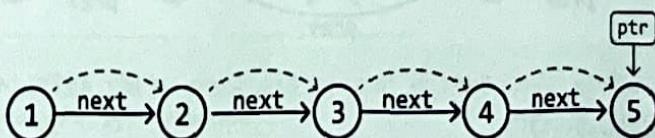
```
class ListNode:
    def __init__(self, val: int, next: ListNode):
        self.val = val
        self.next = next
```

Singly linked list

The simplest form of a linked list is a singly linked list, where each node points to the next node in the linked list, and the last node points to nothing (null), indicating the end of the linked list. The start of the linked list is called the 'head,' which is generally the only node we initially have immediate access to:

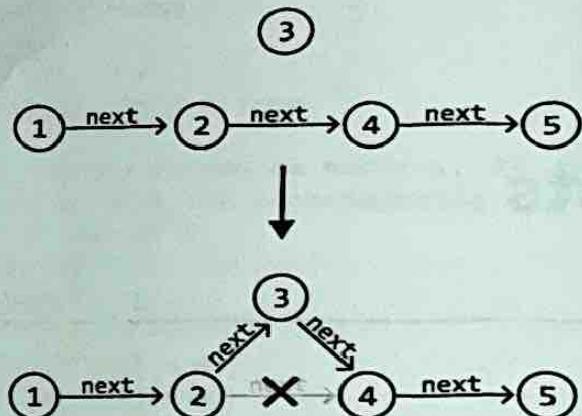


To access the other nodes in a linked list, we would need to traverse it starting at the head.



Singly linked lists can be used to store a collection of data. One of their main benefits lies in their dynamic sizing capability, since they can grow or shrink in size flexibly, unlike arrays which are fixed in size. Additionally, singly linked lists excel in scenarios requiring frequent insertions and deletions, as these operations can be performed more efficiently than in arrays, which need to shift elements to perform insertion or deletion.

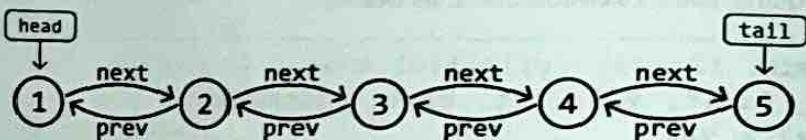
example - insert node 3 between nodes 2 and 4:



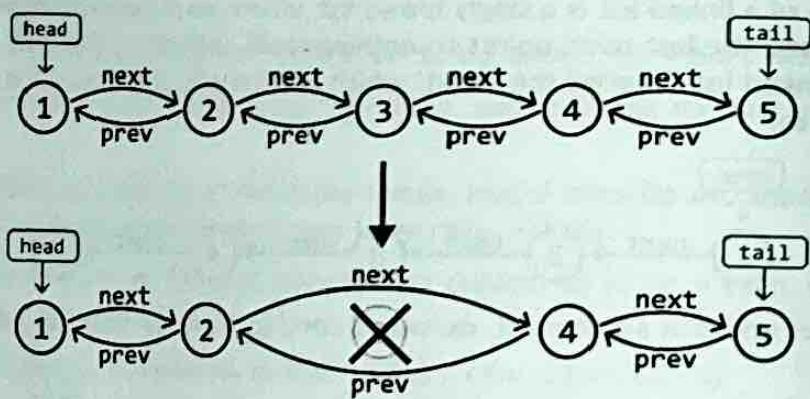
The efficiency of these operations comes at the cost of the inability to perform random access, as nodes can't be accessed by indexes like in an array. This trade-off may be acceptable in many use cases where the benefits of dynamic sizing and the efficiency of insertion/deletion outweigh the main performance benefits of random access.

Doubly linked list

A doubly linked list is an extended version of the linked list where each node contains two references: one to the next node (next), and one to the previous node (prev). In most implementations, doubly linked lists have immediate access to both the head node and the tail node.

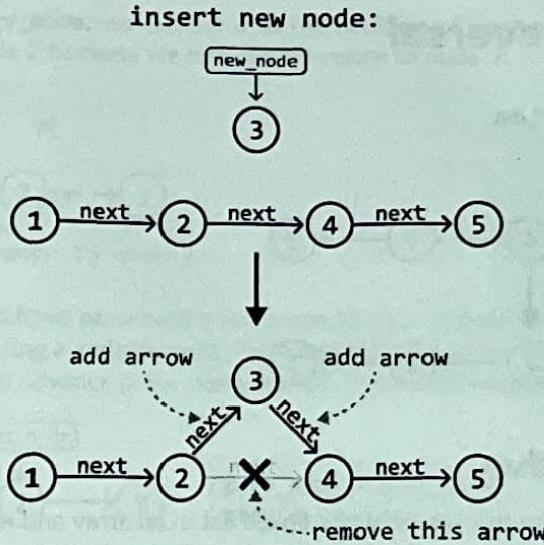


A big advantage of doubly linked list is that it allows for **bidirectional traversal**. Additionally, deleting nodes in a doubly linked list is generally more straightforward because we have references to both the next and previous nodes:



Pointer Manipulation

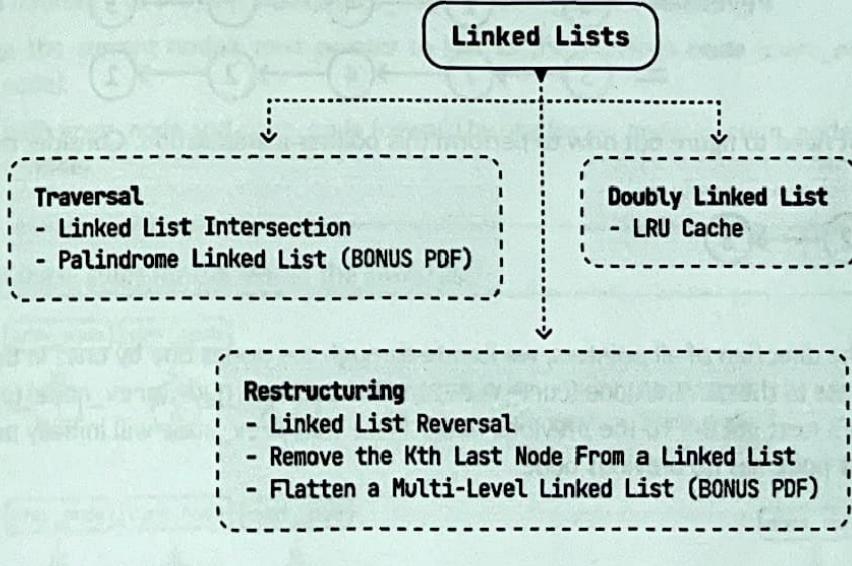
Many linked list interview problems require traversing or restructuring a linked list. Understanding and being proficient at pointer manipulation is essential to solving these problems. A useful tip is to visualize pointers as arrows that point from one node to another, and observe how these arrows should be moved to reflect the structural change. For example, this is how we would visualize a node insertion:



Real-world Example

Music Playlist: Music player applications often use linked lists to implement playlists, particularly doubly linked lists, where each song node links to the next and previous songs. This structure enables efficient addition, removal, and reordering of songs because only the pointers between nodes need to be updated, rather than moving the song data in memory.

Chapter Outline



Practice these problems on our online code editor
[→ bit.ly/run-code](http://bit.ly/run-code)

</>

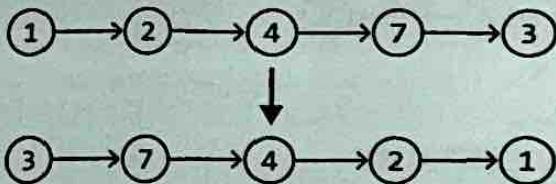
To receive the bonus PDF, sign up for our Coding Interview Patterns newsletter by using the link below:

bit.ly/coding-patterns-pdf

Linked List Reversal

Reverse a singly linked list.

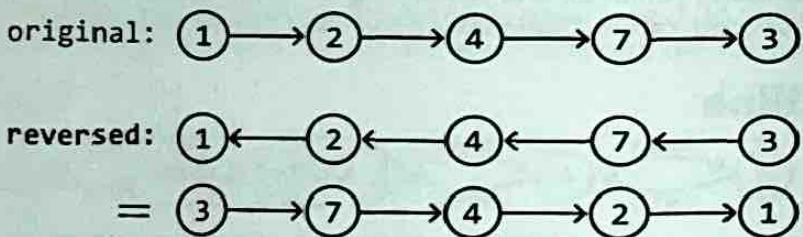
Example:



Intuition - Iterative

A naive strategy is to store the values of the linked list in an array and reconstruct the linked list by traversing the array in reverse order. However, this solution does not reverse the original linked list; it just creates a new one. Could we try performing the reversal in place?

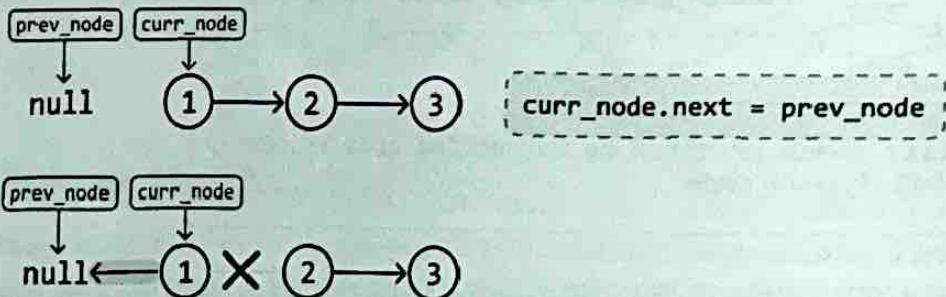
Let's think about the problem in terms of pointer manipulation. The key observation here is that if we "flip" the direction of the pointers, we're effectively reversing the linked list:



Now, we just need to figure out how to perform this pointer manipulation. Consider the example below:

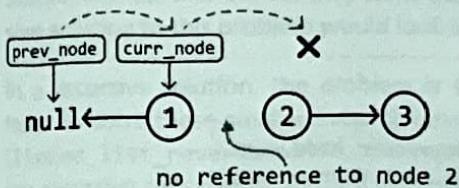


To reverse the direction of all pointers, we iterate through the nodes one by one. In this process, we need access to the current node (`curr_node`) and the previous node (`prev_node`) to adjust the current node's next pointer to the previous node. Note that `prev_node` will initially point at null since the first node has no previous node:

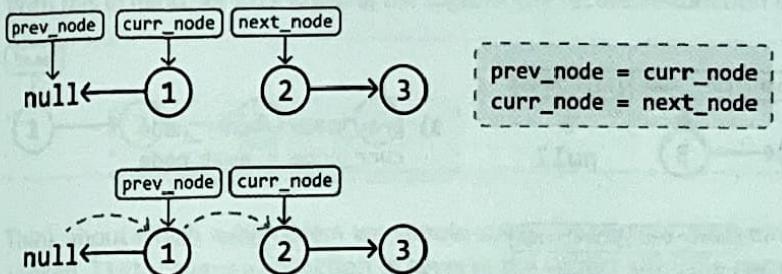


To reverse the next pointer, we'll need a way to shift the `curr_node` and `prev_node` pointers one

node over. To shift `prev_node`, we can set it to the position of `curr_node`. However, we can't move `curr_node` to node 2 because we lost our reference to node 2:



This suggests we should have preserved a reference to node 2 before reversing the `curr_node`. This can be done by creating a variable `next_node` and setting it to `curr_node.next`. Let's assume we did this. Now, we can advance `prev_node` and `curr_node` forward by one:

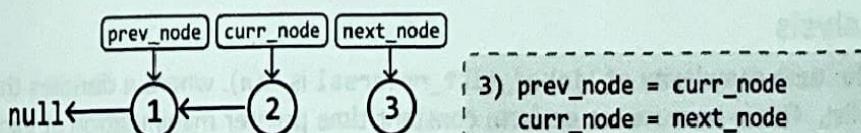
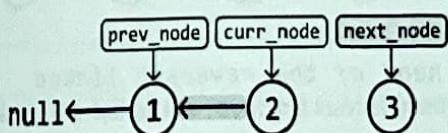
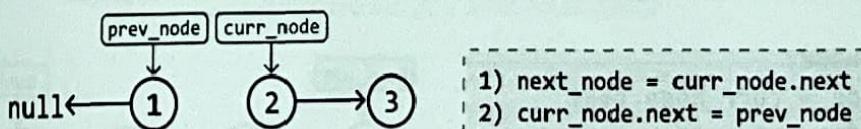


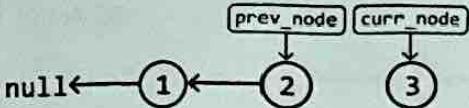
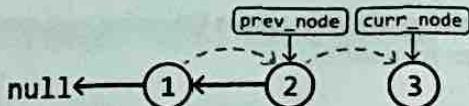
Note, we don't need to shift `next_node`, as it can be set by `curr_node.next` in the next iteration.

We can summarize this logic in three steps. At each node in the linked list:

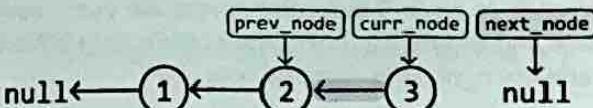
- 1) Save a reference to the next node (`next_node = curr_node.next`).
- 2) Change the current node's next pointer to link to the previous node (`curr_node.next = prev_node`).
- 3) Move both `prev_node` and `curr_node` forward by one (`prev_node = curr_node`, `curr_node = next_node`).

Let's repeat these steps for the rest of the linked list:

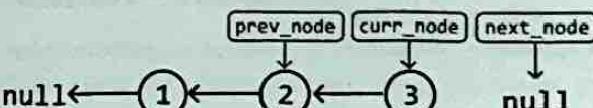




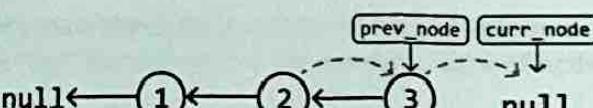
1) `next_node = curr_node.next`
2) `curr_node.next = prev_node`



`next_node`



3) `prev_node = curr_node`
`curr_node = next_node`



We can stop the reversal when `curr_node` becomes null, indicating there are no more nodes to reverse.

The final step is to return the head of the reversed linked list, which is pointed to by `prev_node` once `curr_node` becomes null.

Implementation – Iterative

```
def linked_list_reversal(head: ListNode) -> ListNode:
    curr_node, prev_node = head, None
    # Reverse the direction of each node's pointer until 'curr_node'
    # is null.
    while curr_node:
        next_node = curr_node.next
        curr_node.next = prev_node
        prev_node = curr_node
        curr_node = next_node
    # 'prev_node' will be pointing at the head of the reversed linked
    # list.
    return prev_node
```

Complexity Analysis

Time complexity: The time complexity of `linked_list_reversal` is $O(n)$, where n denotes the length of the linked list. This is because we perform constant-time pointer manipulation at each node of the linked list.

Space complexity: The space complexity is $O(1)$.

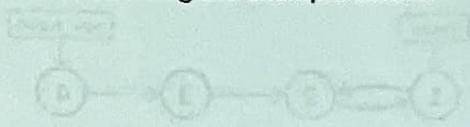
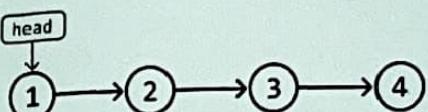
Intuition – Recursive

Sometimes, the interviewer may want the problem solved using recursion. Let's see what a recursive solution to this problem would look like.

In a recursive solution, the problem is solved by solving smaller instances of the same problem. To solve these smaller subproblems, we would need to use the linked list reversal function (`linked_list_reversal`) in its own implementation. The process of solving smaller problems using recursive calls continues until the smallest version of the problem is solved.

The smallest version of this problem involves reversing a linked list of size 0 or 1. These are linked lists which are inherently the same as their reverse. So, these can be our **base cases**.

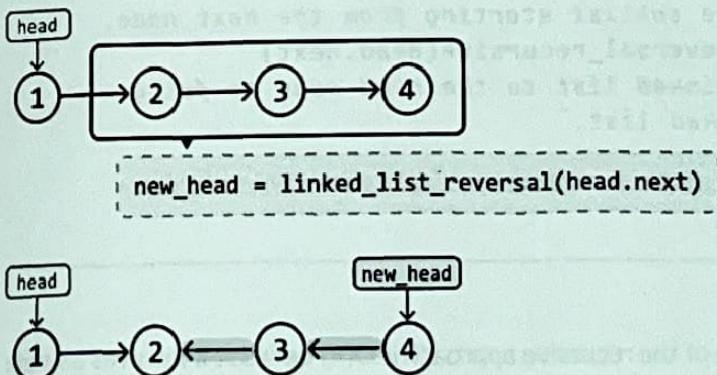
With this in mind, let's try crafting the logic of the recursive function using the example below:



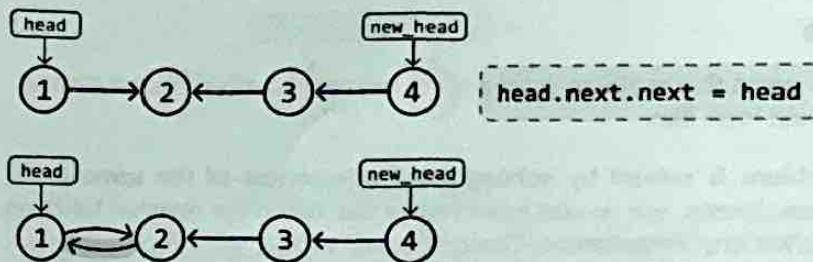
Think about which subproblem we should solve. To reverse the entire linked list, we can use our `linked_list_reversal` function to reverse the sublist after the current node. This way, we only need to focus on reversing the pointer of the current node. Let's see how this works.

As mentioned, let's first recursively call `linked_list_reversal` on the sublist starting at `head.next`. Let's assume this recursive call reverses this sublist and returns its head as intended.

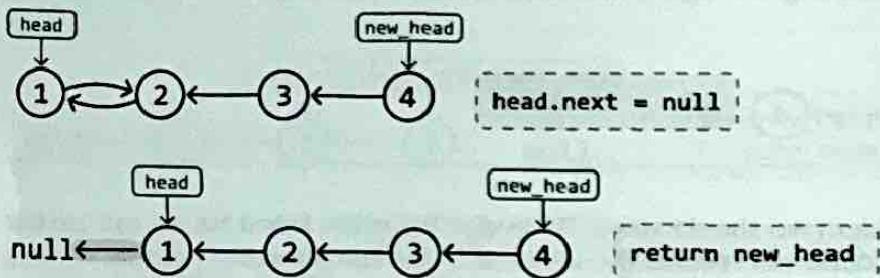
When designing a recursive function, assume any recursive call to that function will behave as intended, even if the function hasn't been fully implemented.



Next, the tail of the reversed sublist (node 2) needs to point to node 1. We can reference node 2 using `head.next`. So, all we need to do is set `head.next.next` to `head` as illustrated below:



The linked list is almost fully reversed now, but we still have node 1 pointing to node 2. To remove this link, just set `head.next` to `null`. Then, we can return `new_head`, which is the head of the reversed list:



Implementation – Recursive

```
def linked_list_reversal_recursive(head: ListNode) -> ListNode:
    # Base cases.
    if (not head) or (not head.next):
        return head
    # Recursively reverse the sublist starting from the next node.
    new_head = linked_list_reversal_recursive(head.next)
    # Connect the reversed linked list to the head node to fully
    # reverse the entire linked list.
    head.next.next = head
    head.next = None
    return new_head
```

Complexity Analysis

Time complexity: The time complexity of the recursive approach is $O(n)$ because it involves a single recursive traversal through the linked list, visiting each node exactly once.

Space complexity: The space complexity is $O(n)$ due to the stack space taken up by the recursive call stack, which can grow n levels deep because n recursive calls are made.

Interview Tip



Tip: Visualize pointer manipulations.

Often, it can be tricky to figure out exactly what to do when dealing with linked list manipulation. Drawing pointers as arrows between nodes can be quite helpful. By observing how these arrows should be reoriented to represent changes in the linked list's structure, we can deduce the necessary pointer manipulation logic. This approach also helps identify which nodes we need references to when making these changes.

To illustrate this, let's say we want to reverse the linked list shown below. We know that the head pointer needs to point to node 2, so we'll start by drawing an arrow from the head pointer to node 2.

However, there's no explicit edge case to consider here since all the "head nodes" in the list are *pointed to elements* — we can't therefore know if we're about to perform a *normal* reversal or an *inverted* reversal. So instead of this, we can make a decision to always follow the *head* pointer and start our traversal from there.

Visualizing Reversal

Let's base our manipulation strategy off of this. If we start our traversal from the head pointer and move right, we can immediately see that node 2 is about to undergo either end of insertion or a yield of the current node. (This requires all nodes except node 2 to be *normal* elements and all others must consist of a *tail* or *head* node.) Let's now try implementing our reasoning with this diagram:

First, observe the current pointer `prev = null`. It's a pointer pointing to the previous node.

Next, we'll move to node 2. Since `prev` is `null`, we'll set `prev = node 2`.

Finally, we'll move to node 3. Since `prev` is now node 2, we'll set `prev = node 3`.

At this point, we've reached the end of the list, so we'll set `prev = null` and return the new head pointer.

As you can see, this is a very simple way to visualize pointer manipulations.

Now, consider what's written just above this section: *"A self-taught developer who has had no help or training will likely have trouble with this."* Well, if we take a step back, we can realize that most of the time, we don't actually need to think about the nodes themselves or how they relate to one another. Most of the time, we just need to remember that a *normal* node is a node that points to the next node in sequence, and an *inverted* node is a node that points to the previous node in sequence. This is why we can ignore the nodes themselves and focus on the pointers.

Indeed, it's often the case that a *normal* node is a pointer to the *next* node in the sequence of the list, and an *inverted* node is a pointer to the *previous* node in the sequence. This is why we can ignore the nodes themselves and focus on the pointers.

But what if we want to implement a *linked list*? One of the best ways to do this is to use *linked lists*. A *linked list* is a data structure that consists of a series of nodes, each containing data and a reference to the next node in the sequence. These nodes are typically represented as objects with properties like `value` and `next`.

For example, consider the following code snippet:

```
class Node {  
    constructor(value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

In this code, we define a *Node* class that takes a *value* as a parameter and initializes its *next* pointer to `null`.

Now, let's say we want to create a linked list with three nodes: `n1`, `n2`, and `n3`. We can do this by creating three instances of the *Node* class and setting their *value* properties to `1`, `2`, and `3` respectively:

```
n1 = new Node(1);  
n2 = new Node(2);  
n3 = new Node(3);
```

Next, we'll set the *next* pointer of `n1` to `n2`, the *next* pointer of `n2` to `n3`, and the *next* pointer of `n3` to `null`:

```
n1.next = n2;  
n2.next = n3;  
n3.next = null;
```

Finally, we'll set the *head* pointer to `n1` and return the list:

```
head = n1;  
return head;
```

And that's it! We've created a linked list with three nodes: `n1`, `n2`, and `n3`.

Now, let's say we want to print out the values of all the nodes in the list. We can do this by using a *for* loop:

```
for (let node = head; node !== null; node = node.next) {  
    console.log(node.value);  
}
```

When we run this code, we'll see the output: `1`, `2`, and `3`.

That's it! We've created a linked list with three nodes: `n1`, `n2`, and `n3`.

Now, let's say we want to add a new node to the list. We can do this by creating a new *Node* instance and setting its *value* property to the desired value:

```
const newNode = new Node(4);
```

Next, we'll set the *next* pointer of `n3` to `newNode`:

```
n3.next = newNode;
```

Finally, we'll return the list:

```
return head;
```

And that's it! We've added a new node to the linked list.

Now, let's say we want to remove a node from the list. We can do this by using a *for* loop:

```
for (let node = head; node !== null; node = node.next) {  
    if (node.value === 2) {  
        node.next = node.next.next;  
    }  
}
```

When we run this code, we'll see the output: `1`, `3`, and `4`.

That's it! We've removed the node with value `2` from the linked list.

Now, let's say we want to insert a new node at a specific position in the list. We can do this by creating a new *Node* instance and setting its *value* property to the desired value:

```
const newNode = new Node(5);
```

Next, we'll find the node before the position where we want to insert the new node:

```
const prevNode = head;
```

Then, we'll set the *next* pointer of `prevNode` to `newNode`:

```
prevNode.next = newNode;
```

Finally, we'll return the list:

```
return head;
```

And that's it! We've inserted a new node at the specified position in the linked list.

Now, let's say we want to reverse the linked list. We can do this by using a *for* loop:

```
for (let node = head; node !== null; node = node.next) {  
    const temp = node.next;  
    node.next = prevNode;  
    prevNode = node;  
}
```

When we run this code, we'll see the output: `4`, `3`, and `1`.

That's it! We've reversed the linked list.

Now, let's say we want to delete the entire linked list. We can do this by setting the *head* pointer to `null`:

```
head = null;
```

Finally, we'll return the list:

```
return head;
```

And that's it! We've deleted the entire linked list.

Now, let's say we want to copy the entire linked list. We can do this by using a *for* loop:

```
const copiedList = new Node(head.value);
```

Then, we'll set the *head* pointer of the copied list to the *next* node of the original list:

```
head = copiedList.next;
```

Finally, we'll return the copied list:

```
return copiedList;
```

And that's it! We've copied the entire linked list.

Now, let's say we want to merge two linked lists. We can do this by using a *for* loop:

```
for (let node = head1; node !== null; node = node.next) {  
    const temp = node.next;  
    node.next = head2;  
    head2 = node;  
}
```

When we run this code, we'll see the output: `1`, `2`, `3`, `4`, and `5`.

That's it! We've merged the two linked lists.

Now, let's say we want to search for a specific value in the linked list. We can do this by using a *for* loop:

```
for (let node = head; node !== null; node = node.next) {  
    if (node.value === targetValue) {  
        return true;  
    }  
}
```

When we run this code, we'll see the output: `true` if the value is found, and `false` if it's not.

That's it! We've searched for a specific value in the linked list.

Now, let's say we want to sort the linked list. We can do this by using a *for* loop:

```
for (let node = head; node !== null; node = node.next) {  
    const temp = node.next;  
    node.next = prevNode;  
    prevNode = node;  
}
```

When we run this code, we'll see the output: `1`, `2`, `3`, `4`, and `5`.

That's it! We've sorted the linked list.

Now, let's say we want to find the length of the linked list. We can do this by using a *for* loop:

```
let length = 0;  
for (let node = head; node !== null; node = node.next) {  
    length++;  
}
```

When we run this code, we'll see the output: `5`.

That's it! We've found the length of the linked list.

Now, let's say we want to find the middle element of the linked list. We can do this by using a *for* loop:

```
let middle = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count % 2 === 0) {  
        middle = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `3`.

That's it! We've found the middle element of the linked list.

Now, let's say we want to find the k-th element of the linked list. We can do this by using a *for* loop:

```
let kthElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `4`.

That's it! We've found the k-th element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

```
let kthToLastElement = null;  
let count = 0;  
for (let node = head; node !== null; node = node.next) {  
    if (count === k) {  
        kthToLastElement = node;  
    }  
    count++;  
}
```

When we run this code, we'll see the output: `2`.

That's it! We've found the k-th to last element of the linked list.

Now, let's say we want to find the k-th to last element of the linked list. We can do this by using a *for* loop:

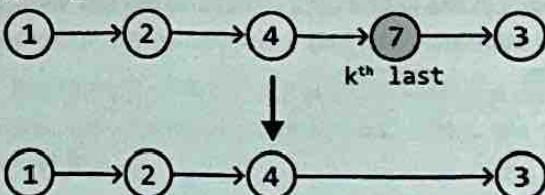
```
let kthToLastElement = null;  
let count = 0;  
for (let node
```

Remove the Kth Last Node From a Linked List

Return the head of a singly linked list after removing the kth node from the end of it.

Example:

k = 2



Constraints:

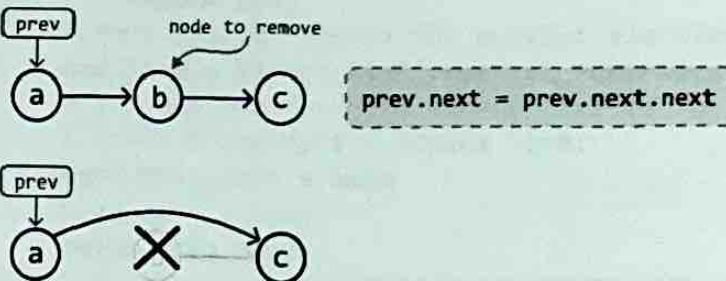
- The linked list contains at least one node.

Intuition

We can divide this problem into two objectives:

1. Find the position of the kth last node.
2. Remove this node.

Let's first understand how node removal works. Consider the example below, where we need to remove node b. To do this, we need access to the node preceding it (node a), so we can redirect the pointer of node a to skip over node b. This ensures node b is no longer reachable through linked list traversal:

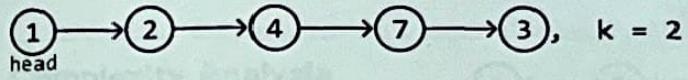


This indicates we need to find the node directly before the kth last node in order to remove it.

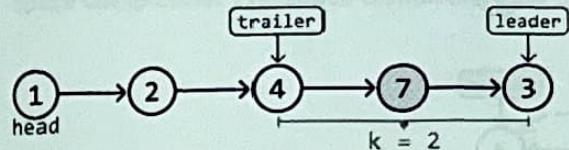
A naive solution to this problem is to first obtain the length of the linked list (n) by traversing it. Then, use this length to determine the number of steps required to arrive at the node before the kth last node, which is just n - k - 1 steps. This solution involves two for-loops, but is there a cleaner way to approach this problem?

The challenge with navigating a singly linked list in a single for-loop is that as we traverse, it's hard to tell how far we are from the final node. The only way we'd know this is when we reach the final node itself, since its next node is null. How can we make use of this information?

Consider using two pointers instead of one. Could we create a scenario where, by the time one pointer reaches the end of the linked list, another pointer is positioned before the kth last node? Let's explore this logic using the following example:

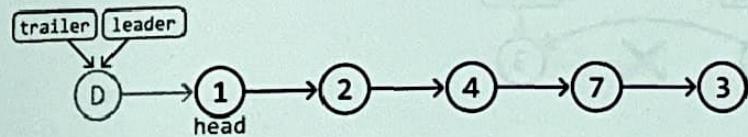


We denote the first pointer as leader and the pointer that follows it as trailer. When the leader pointer reaches the last node of the linked list, we want the trailer pointer to end up at node 4 (the node right before the k^{th} last node) to prepare for deletion. In other words, the leader should be k nodes in front of the trailer when the leader reaches the last node.



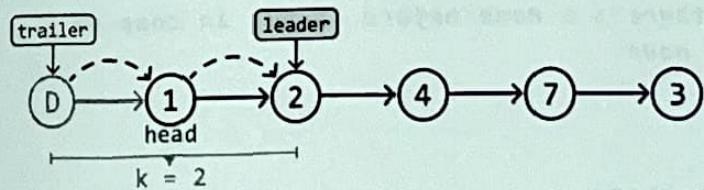
To achieve this, we can start by advancing the leader pointer through the linked list for k steps. When the leader pointer is k nodes ahead of the trailer, we can advance both pointers together until the leader reaches the last node. This process will be explained in more detail soon.

However, there's an important edge case to consider first: what if the head itself is the node we need to remove? In this case, there's no node before the head, so we cannot perform the removal, as mentioned earlier. To circumvent this, we can create a dummy node, place it before the head node, and start our traversal from there.

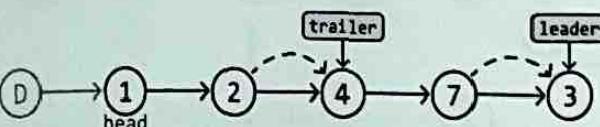
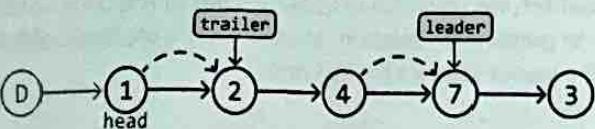
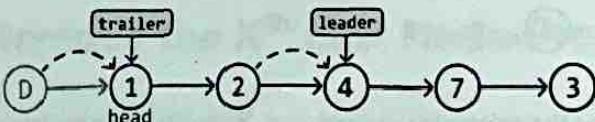


Let's now try incorporating our strategy into the example.

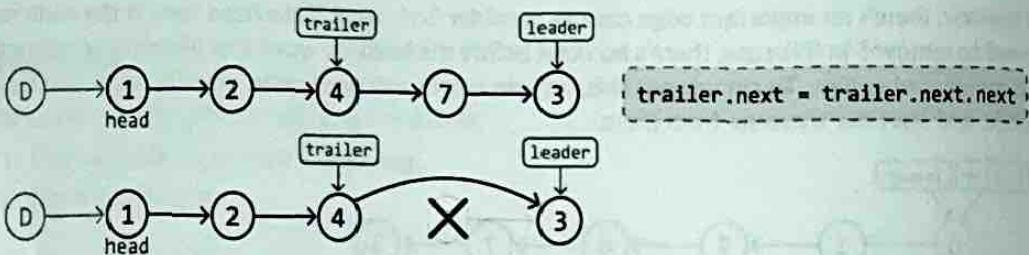
First, advance the leader pointer k (2) times so it's k nodes ahead of the trailer pointer:



With the leader k nodes ahead, we can move both the trailer and leader pointers until the leader reaches the last node:



With the trailer pointer at the ideal position, we can remove node 7:



After this removal, we just return `dummy.next`, which points at the head of the modified linked list.

Implementation

```
def remove_kth_last_node(head: ListNode, k: int) -> ListNode:
    # A dummy node to ensure there's a node before 'head' in case we
    # need to remove the head node.
    dummy = ListNode(-1)
    dummy.next = head
    trailer = leader = dummy
    # Advance 'leader' k steps ahead.
    for _ in range(k):
        leader = leader.next
    # If k is larger than the length of the linked list, no node
    # needs to be removed.
    if not leader:
        return head
    # Move 'leader' to the end of the linked list, keeping 'trailer'
    # k nodes behind.
    while leader.next:
        leader = leader.next
        trailer = trailer.next
    # Remove the kth node from the end.
    trailer.next = trailer.next.next
    return dummy.next
```

Complexity Analysis

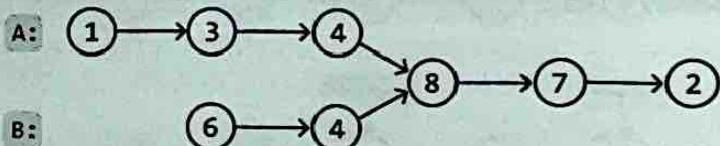
Time complexity: The time complexity of `remove_kth_last_node` is $O(n)$. This is because the algorithm first traverses at most n nodes of the linked list, and then two pointers traverse the linked list at most once each.

Space complexity: The space complexity is $O(1)$.

Linked List Intersection

Return the node where two singly linked lists intersect. If the linked lists don't intersect, return null.

Example:



Output: Node 8

Intuition

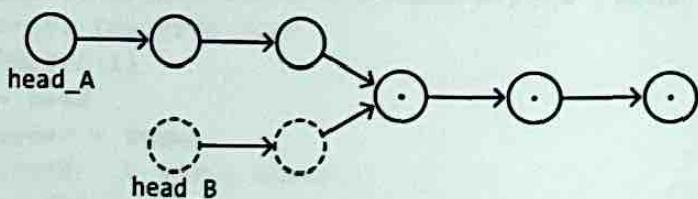
Let's first understand what an intersection between two linked lists is.

An intersection occurs when two linked lists converge at a shared node and, from that point onwards, share all subsequent nodes.

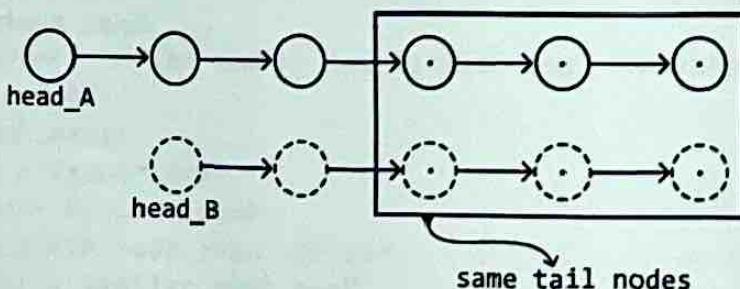
Note, this intersection has nothing to do with the values of the nodes.

A naive approach is to use a hash set. We can traverse the first linked list once and store each node in a hash set. Next, we traverse the second linked list until we find the first node that exists in the hash set, signifying the intersection point since it's the first node shared between the two linked lists. This approach solves the problem linearly, but can we find a solution that uses constant space?

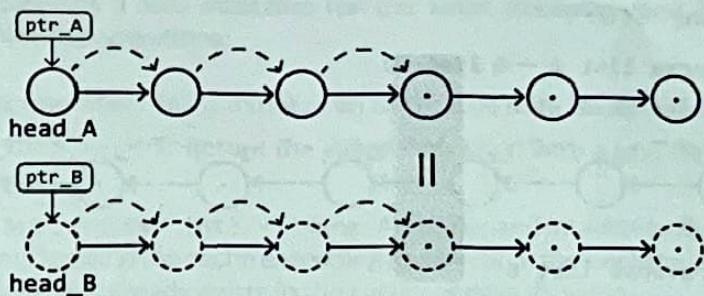
Consider the following example:



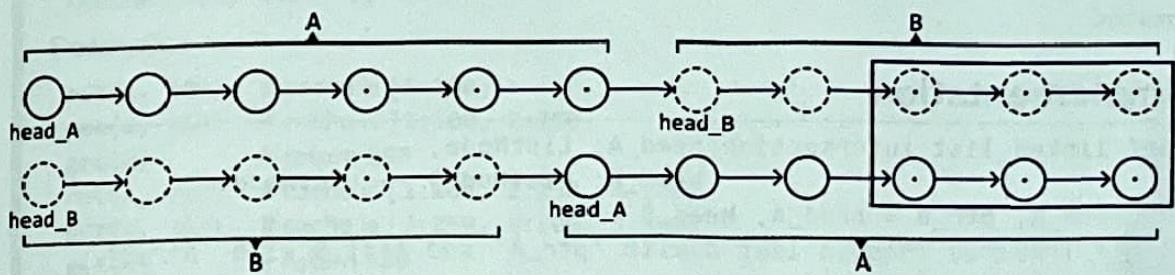
Treating these as two separate linked lists can get confusing with the above visualization. Instead, let's visualize the input as two linked lists to help us think about the problem more clearly. Note that the tail nodes are still shared between the two linked lists, we're just visualizing them separately:



Notice that this problem is easier to solve if the two linked lists are of equal length. This is because the intersection node can be found at the same position from the heads of both linked lists. So, we're guaranteed to reach the intersection node at the same time.

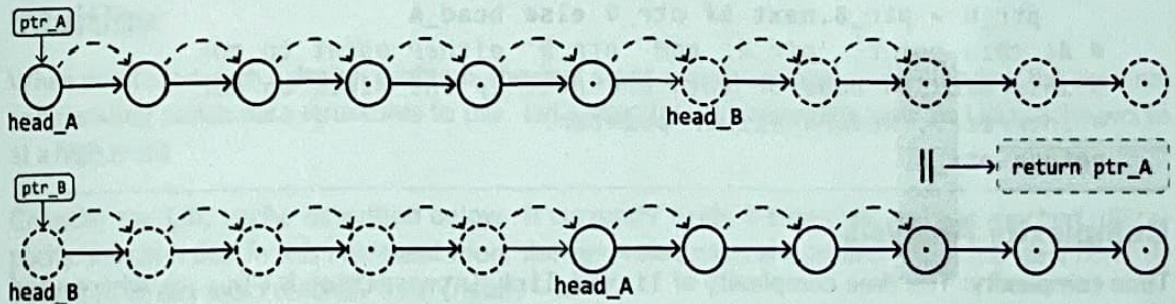


Could we somehow replicate this behavior when dealing with linked lists of varying lengths? The key observation is that, while two linked lists 'list A' and 'list B' may have different lengths, 'list A → list B' has the same length as 'list B → list A' (where '→' represents the connection of two linked lists). Conveniently, these combined linked lists also share the same tail nodes:

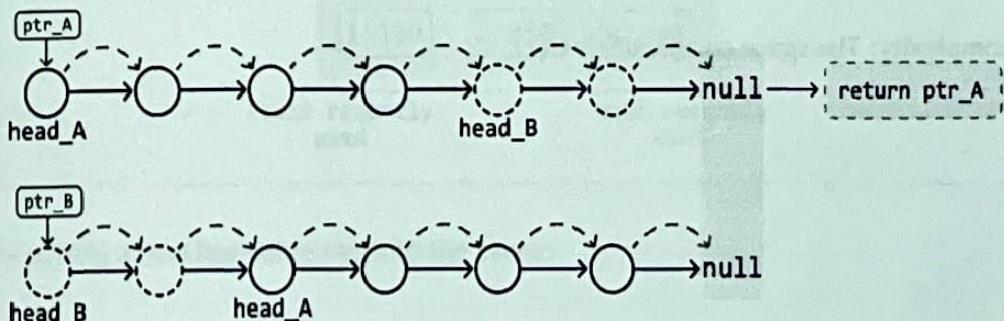


We've now set up a scenario where we have two combined linked lists of the same length, which share the same tail nodes. By traversing these combined linked lists, we'll eventually reach the intersection node simultaneously on both linked lists (if one exists).

To do this, we can traverse both combined linked lists with two pointers, and stop once the nodes at both pointers are the same. This node would be the intersection node:



If no intersection exists, both pointers will end up stopping at null nodes:



Traversing combined linked lists

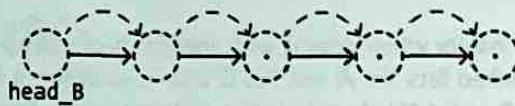
An important observation is that to traverse through 'list A → list B', we don't actually need to connect these two linked lists together. Instead, we can traverse 'list A' and, upon reaching its end, continue by traversing 'list B':

traverse list A → list B:

1. traverse list A



2. traverse list B



This technique allows us to traverse the entire sequence of both linked lists as if they were connected.

Implementation

```
def linked_list_intersection(head_A: ListNode,
                                head_B: ListNode) -> ListNode:
    ptr_A, ptr_B = head_A, head_B
    # Traverse through list A with 'ptr_A' and list B with 'ptr_B'
    # until they meet.
    while ptr_A != ptr_B:
        # Traverse list A -> list B by first traversing 'ptr_A' and
        # then, upon reaching the end of list A, continue the
        # traversal from the head of list B.
        ptr_A = ptr_A.next if ptr_A else head_B
        # Simultaneously, traverse list B -> list A.
        ptr_B = ptr_B.next if ptr_B else head_A
    # At this point, 'ptr_A' and 'ptr_B' either point to the
    # intersection node or both are null if the lists do not
    # intersect. Return either pointer.
    return ptr_A
```

Complexity Analysis

Time complexity: The time complexity of `linked_list_intersection` is $O(n + m)$, where n and m denote the lengths of list A and B, respectively. This is because pointers linearly traverse both linked lists sequentially.

Space complexity: The space complexity is $O(1)$.

LRU Cache

Design and implement a data structure for the Least Recently Used (LRU) cache that supports the following operations:

- **LRUCache(capacity: int)**: Initialize an LRU cache with the specified capacity.
 - **get(key: int) -> int**: Return the value associated with a key. Return -1 if the key doesn't exist.
 - **put(key: int, value: int) -> None**: Add a key and its value to the cache. If adding the key would result in the cache exceeding its size capacity, evict the least recently used element. If the key already exists in the cache, update its value.

Example:

Input: [put(1, 100), put(2, 250), get(2), put(4, 300), put(3, 200),
get(4), get(1)], capacity = 3
Output: [250, 300, -1]

Explanation:

```

put(1, 100)    # cache is [1:100]
put(2, 250)    # cache is [1:100, 2:250]
get(2)         # return 250
put(4, 300)    # cache is [1:100, 2:250, 4:300]
put(3, 200)    # cache is [2:250, 4:300, 3:200]
get(4)         # return 300
get(1)         # key 1 was evicted when adding key 3 due to the capacity: return -1

```

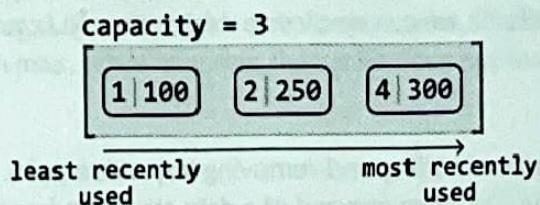
Constraints:

- All keys and values are positive integers.
 - The cache capacity is positive.

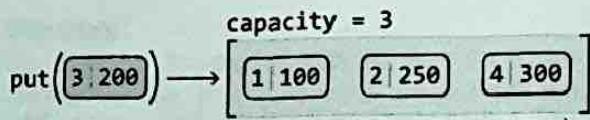
Intuition

When presented with a design problem, the first steps usually involve understanding the problem and deciding which data structures to use. Let's start by understanding how an LRU cache works at a high level.

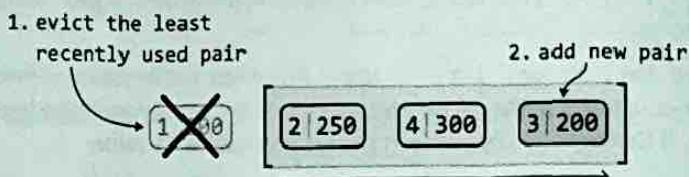
Consider the LRU cache described below. It currently holds 3 elements and has reached full capacity. Assume that in this representation, the key-value pairs are ordered from the least recently used (left) to the most recently used (right):



Let's try putting a new key-value pair into the cache:



This new pair would effectively be the most recent in the cache, so we know it should be added at the most-recently-used end of the cache. Since the cache is currently at maximum capacity, we need to make room for the new pair by first evicting the least recently used pair:

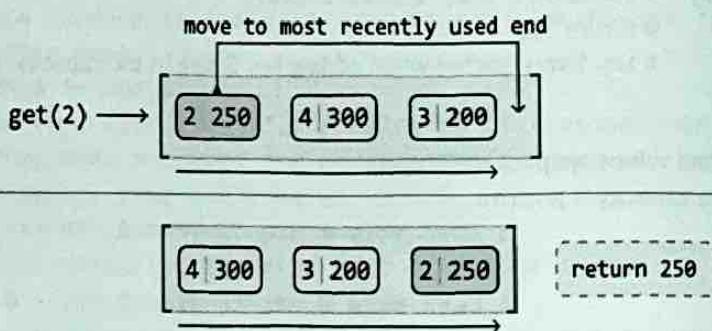


From this high-level overview, we can summarize operations we need to implement the put function:

Operation 1: Remove a key-value pair from the least recently used end of the cache.

Operation 2: Add a key-value pair to the most recently used end of the cache.

Now, let's try retrieving a value from this example cache. If we perform get(2), we expect it to return 250. Accessing this pair would effectively make it the most recently used pair. So, we should move it to the most recently used end of the cache:



From this example, we identified two key operations for the get function:

Operation 3: Move a key-value pair to the most recent end of the cache.

Operation 4: Access a value using its key.

We've now narrowed the design down to the four main operations listed above. These will help us identify which data structures we can employ to design the LRU cache.

Choosing data structures

Operations 1 and 2

The first two operations involve adding and removing key-value pairs. Specifically, we need the ability to remove a key-value pair from one end of a data structure (representing the least recently used end) and add a key-value pair to the other.

Which data structure allows us to efficiently add or remove an element from it? A suitable data structure for these operations is a linked list, particularly because we can add and remove a node

in constant time if we have a reference to that node. But should we use a singly or doubly linked list?

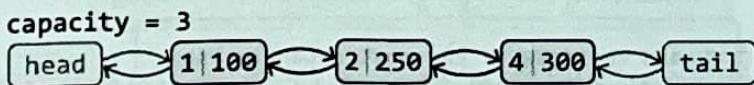
Singly vs. doubly linked list

Adding or removing a node from the head of a linked list takes $O(1)$ time, whether it's a singly or doubly linked list. However, removing the tail node from a singly linked list takes $O(n)$ time, even with a reference to the tail, because we need to traverse the list to access the node before the tail. In contrast, a doubly linked list allows an $O(1)$ time removal of the tail because each node has a reference to its previous node, enabling direct access without traversal. So, let's choose the **doubly linked list**.

An important feature we need is the ability to access both ends of the doubly linked list when adding or removing nodes. With this in mind, let's establish some definitions:

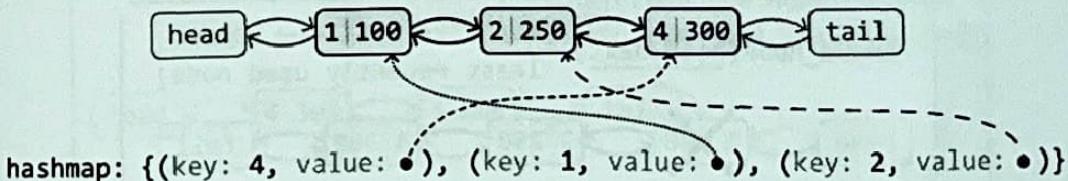
- The **tail** of the linked list signifies the most frequently used node.
- The **head** of the linked list signifies the least recently used node.

To reference the ends of the linked list, we can establish **head** and **tail** nodes, where **head** points to the least recently used node, and **tail** points to the most recently used node:



Operations 3 and 4

Operation 3 indicates that we'll need to be able to move a node to the most recently-used end of the cache, and that this node doesn't necessarily need to be at the head or tail of the linked list. If this node was somewhere in the middle, we'd need to traverse the linked list to find it. Is there a way we could access this node in $O(1)$ time? Since this node is associated with a key, we could use a **hash map** to store key-node pairs. This allows us to access a node by its key in constant time. The diagram below illustrates how the hash map's values are references to nodes in the linked list:

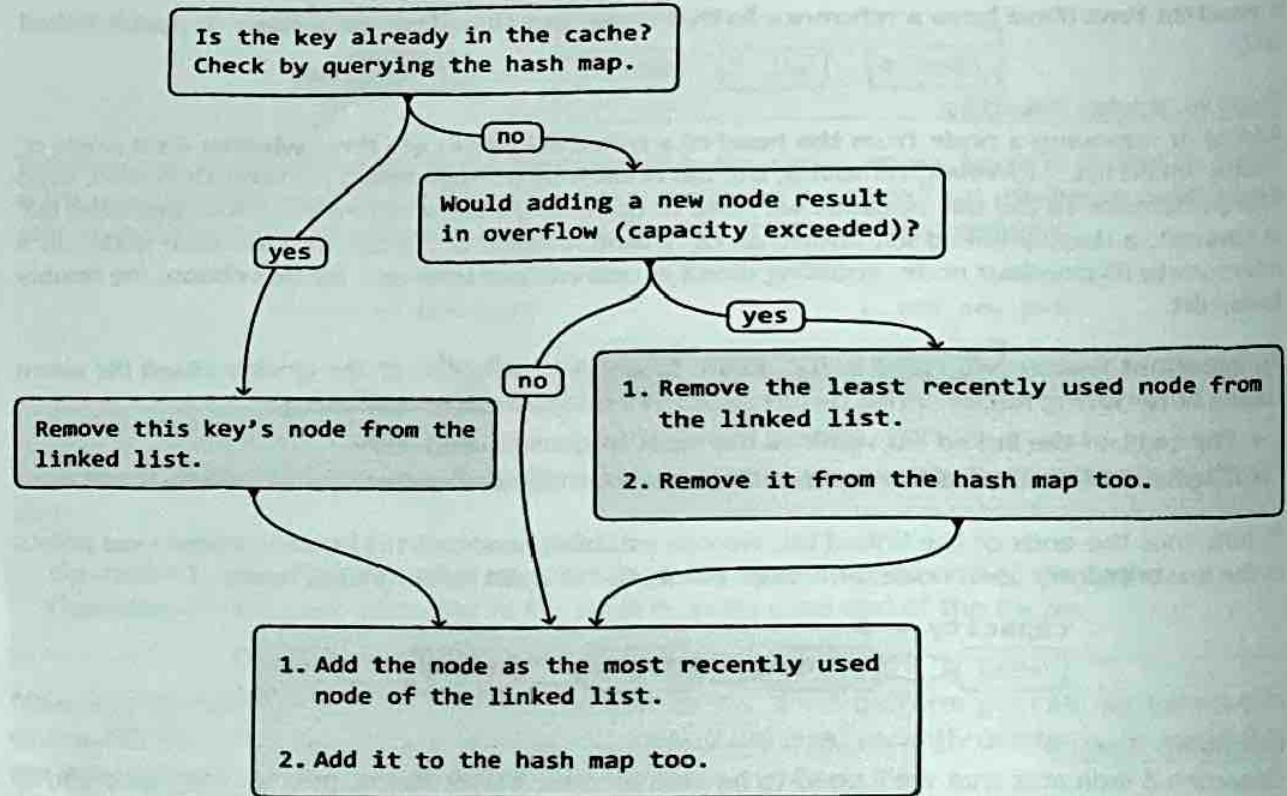


Using a hash map also addresses operation 4 regarding efficient access to values from their keys.

Now we've decided on using a doubly linked list and a hash map to represent the LRU cache, let's examine how the put and get functions would be implemented.

put(key: int, val: int) -> None:

Below is the flow for adding a new key-value pair to the cache. This involves correctly updating the linked list and the hash map, while ensuring the cache does not exceed its capacity:



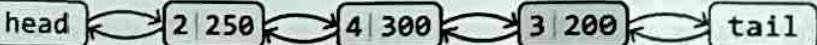
To better understand how to add a new node to a doubly linked list that's at maximum capacity, check out the following example:

`put([3 200]):`

`remove_node(head.next)` (cache is full → remove least recently used node)



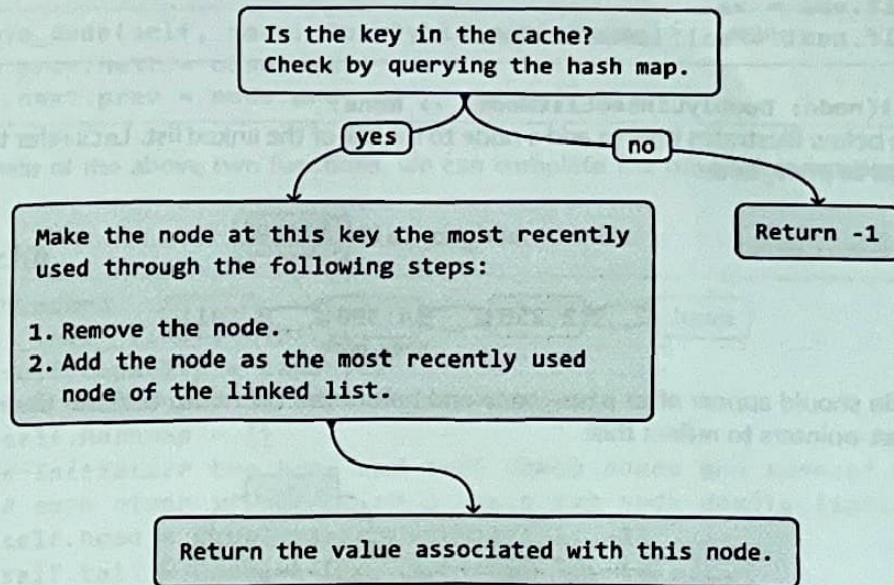
`add_to_tail([3 200])`



As you can see, we'll need a function to remove a node (`remove_node`), as well as a function to add a node to the tail of the linked list (`add_to_tail`). We discuss these functions in more detail in the implementation section.

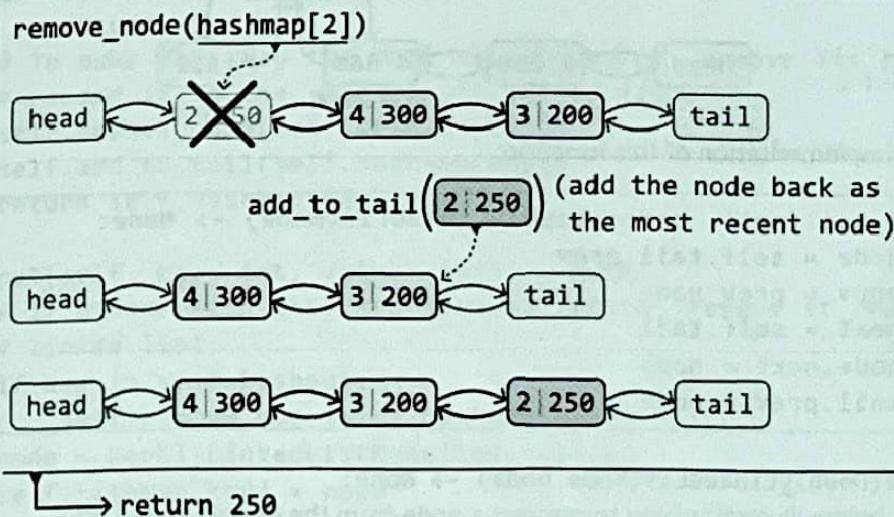
`get(key: int) -> int:`

Below is the process for retrieving a key's value from the cache:



Now let's take a look at an example of how the doubly linked list is updated during a get function call:

get(2):



Now that we understand how the doubly linked list and hash map are used to design the LRU cache, let's dive into its implementation details, including the details of the helper methods `remove_node` and `add_to_tail`.

Implementation

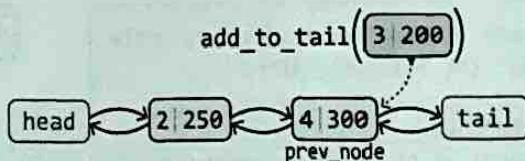
We can use the custom class below to represent a node in a doubly linked list:

```
class DoublyLinkedListNode:  
    def __init__(self, key: int, val: int):
```

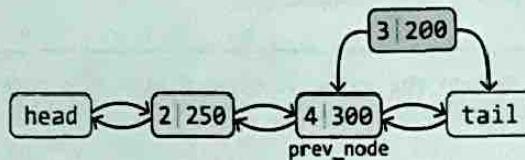
```
    self.key = key  
    self.val = val  
    self.next = self.prev = None
```

add_to_tail(node: DoublyLinkedListNode) -> None:

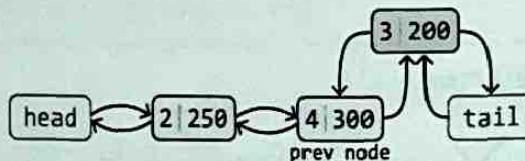
The example below illustrates how to add a node to the tail of the linked list. Let's refer to the node before the tail as `prev_node`.



The new node should appear after `prev_node` and before the tail node. Let's set the new node's `prev` and `next` pointers to reflect this:



Now connect `prev_node` and `tail` to the new node:

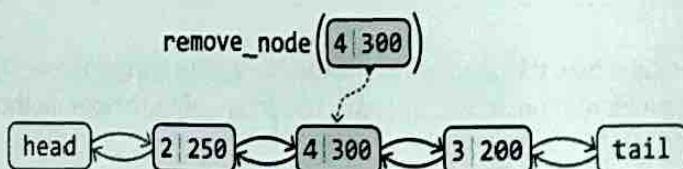


Below is the implementation of this function:

```
def add_to_tail(self, node: DoublyLinkedListNode) -> None:  
    prev_node = self.tail.prev  
    node.prev = prev_node  
    node.next = self.tail  
    prev_node.next = node  
    self.tail.prev = node
```

remove_node(DoublyLinkedListNode node) -> None:

The example below illustrates how to remove a node from the doubly linked list:



To remove a node, we make its two adjacent nodes point at each other, effectively excluding the node to be removed from the linked list:



Below is the implementation of this function:

```
def remove_node(self, node: DoublyLinkedListNode) -> None:  
    node.prev.next = node.next  
    node.next.prev = node.prev
```

With the help of the above two functions, we can complete the full implementation of the LRU cache.

LRU Cache

```
class LRUCache:  
    def __init__(self, capacity: int):  
        self.capacity = capacity  
        # A hash map that maps keys to nodes.  
        self.hashmap = {}  
        # Initialize the head and tail dummy nodes and connect them to  
        # each other to establish a basic two-node doubly linked list.  
        self.head = DoublyLinkedListNode(-1, -1)  
        self.tail = DoublyLinkedListNode(-1, -1)  
        self.head.next = self.tail  
        self.tail.prev = self.head  
  
    def get(self, key: int) -> int:  
        if key not in self.hashmap:  
            return -1  
        # To make this key the most recently used, remove its node and  
        # re-add it to the tail of the linked list.  
        self.remove_node(self.hashmap[key])  
        self.add_to_tail(self.hashmap[key])  
        return self.hashmap[key].val  
  
    def put(self, key: int, value: int) -> None:  
        # If a node with this key already exists, remove it from the  
        # linked list.  
        if key in self.hashmap:  
            self.remove_node(self.hashmap[key])  
        node = DoublyLinkedListNode(key, value)  
        self.hashmap[key] = node  
        # Remove the least recently used node from the cache if adding  
        # this new node will result in an overflow.  
        if len(self.hashmap) > self.capacity:  
            del self.hashmap[self.head.next.key]  
            self.remove_node(self.head.next)  
        self.add_to_tail(node)
```

Complexity Analysis

Time complexity: The time complexity for the helper functions `remove_node` and `add_tail_node` is $O(1)$ because they perform constant-time operations on a doubly linked list. The `put` and `get` functions utilize these helper functions, while also performing constant-time hash map operations. Consequently, they also have an $O(1)$ time complexity.

Space complexity: The overall space complexity of this solution is $O(n)$, where n is the capacity of the cache. This is because both the doubly linked list and hash map can each occupy $O(n)$ space.

Interview Tip

Tip: Explore how combining data structures can help achieve certain functionality.



It's possible to encounter situations where no single data structure provides the functionality required for your solution. In such cases, try to work out if this functionality can be achieved using a combination of data structures. For instance, in this problem we combined a doubly-linked list and a hash map to achieve the functionality required for the LRU cache.

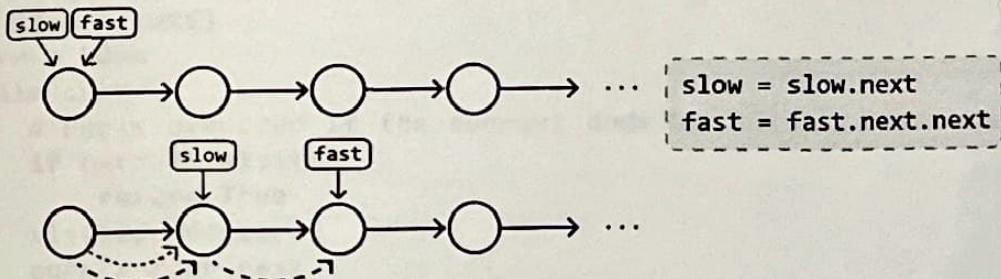
Fast and Slow Pointers

Introduction to Fast and Slow Pointers

The fast and slow pointer technique is a specialized variant of the two-pointer pattern, characterized by the differing speeds at which two pointers traverse a data structure. In this technique, we designate a fast pointer and a slow pointer:

- Usually, the slow pointer moves one step in each iteration.
- Usually, the fast pointer moves two steps in each iteration.

This creates a dynamic in which the fast pointer moves at twice the speed of the slow pointer:



Keep in mind these pointers aren't limited to just one and two steps. As long as the fast pointer advances more steps than the slow pointer does, the logic of the fast and slow pointer technique still applies.

Real-world Example

Detecting cycles in symlinks: Symlinks are shortcuts that point to files or directories in a file system. A real-world example of using the fast and slow pointer technique is detecting cycles in symlinks.

In this process, the slow pointer follows each symlink one step at a time, while the fast pointer moves two steps at a time. If the fast pointer catches up to the slow pointer, it indicates a loop in the symlinks, which can cause infinite loops or errors when accessing files. To understand how fast and slow pointers detect cycles in this way, study the [Linked List Loop](#) problem.

Chapter Outline

Fast and slow pointers are particularly beneficial in data structures like linked lists, where index-based access isn't available. They allow us to gather crucial information about the data structure's

elements through the relative positions of the two pointers, rather than with indexing. This will become clearer as we explore some common use cases in this chapter.

Fast and Slow Pointers

Cycle Detection
- Linked List Loop

Sequence Analysis
- Happy Number

Fractional Point Identification
- Linked List Midpoint



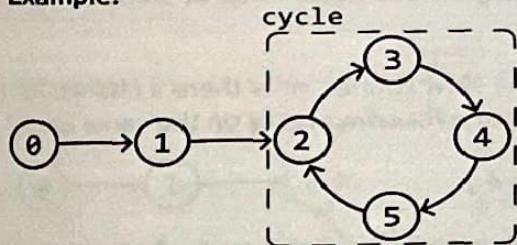
fast pointer
slow pointer

en(100% required)

Linked List Loop

Given a singly linked list, determine if it contains a cycle. A cycle occurs if a node's next pointer references an earlier node in the list, causing a loop.

Example:



| Output: True

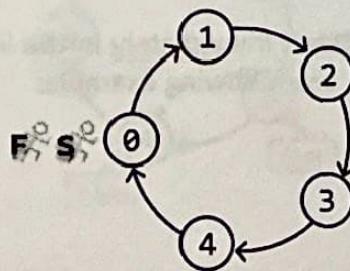
Intuition

A straightforward approach is to iterate through the linked list while keeping track of the nodes that were already visited in a hash set. Encountering a previously-visited node during the traversal indicates the presence of a cycle. Below is the code snippet for this approach:

```
def linked_list_loop_naive(head: ListNode) -> bool:
    visited = set()
    curr = head
    while curr:
        # Cycle detected if the current node has already been visited.
        if curr in visited:
            return True
        visited.add(curr)
        curr = curr.next
    return False
```

This solution takes $O(n)$ time, where n denotes the number of nodes in the linked list, since each node is visited once. However, this comes at the cost of $O(n)$ extra space due to the hash set. Is there a way to achieve a linear time complexity while using constant space?

Imagine a race track represented as a circular linked list (i.e., a linked list with a perfect cycle) where two runners start at the same node.



If both runners move at the same speed, they will always be together at each node. However, consider what happens when one runner (the slow runner) moves one step at a time, while the other

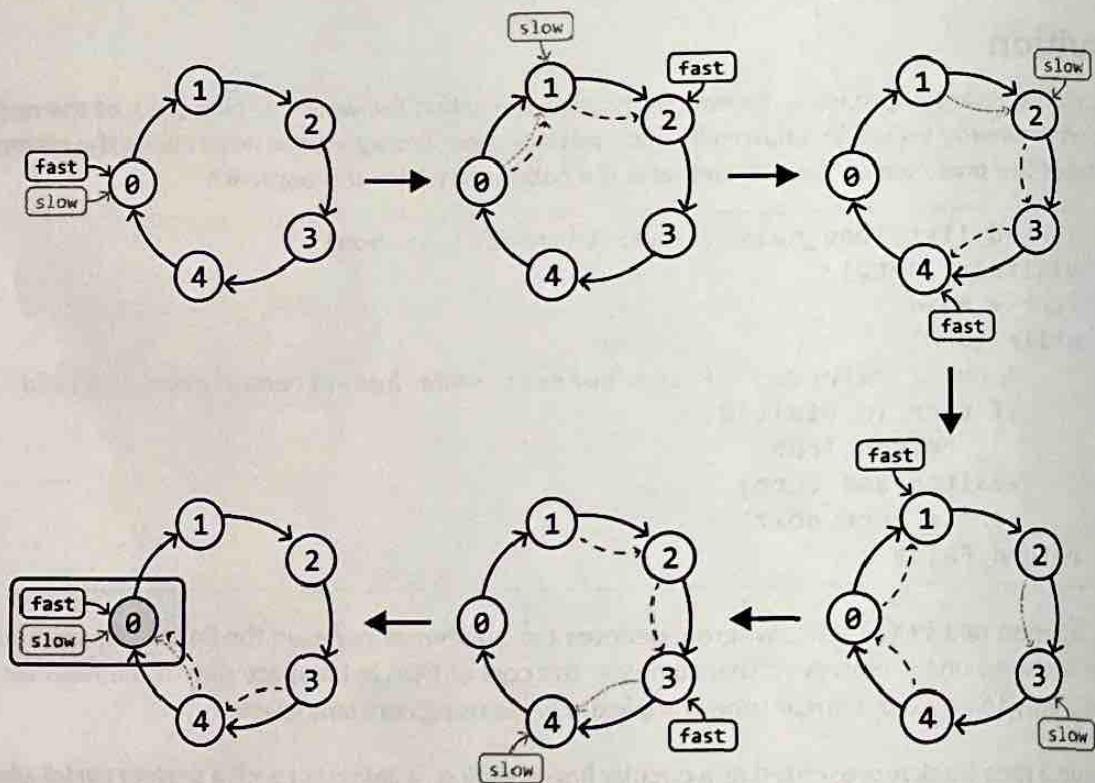
runner (the fast runner) moves two steps at a time. In this scenario, the fast runner will overtake the slow runner at some point since the track is cyclic. But how can we use this information to detect a cycle?

In a linked list, detecting whether the fast runner has overtaken the slow runner is difficult due to the lack of positional indicators (like indexes in an array). A better way to find a cycle is to see if the fast runner reunites with the slow runner by both landing on the same node at some point. This would be a clear sign the linked list has a cycle.

The question now is, will the fast runner reunite with the slow runner, or is there a chance for the fast runner to consistently bypass the slow runner without ever converging on the same node? To answer this, let's start by looking at some examples.

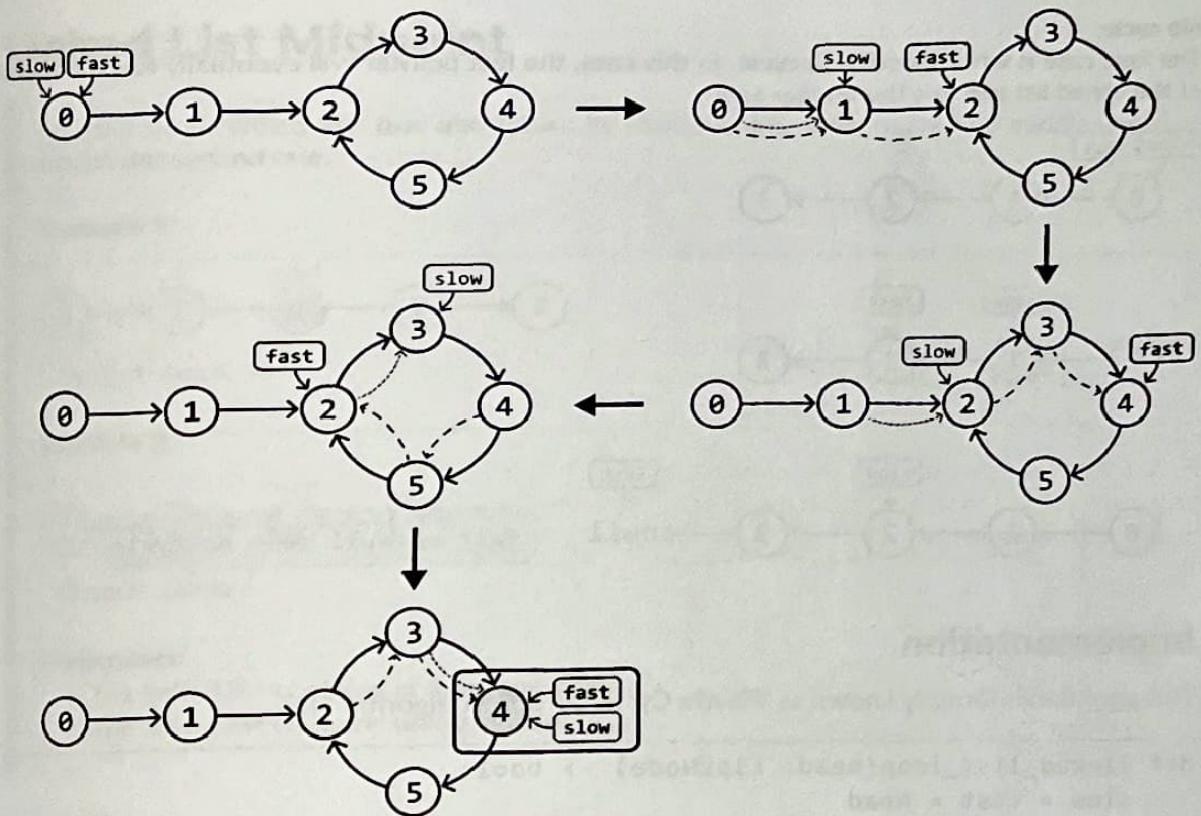
Perfect cycle

First, let's check whether the two runners, represented as a slow pointer and a fast pointer, will reunite in a linked list that forms a perfect cycle. As we can see from the figure below, the pointers will eventually meet.



Delayed cycle

What about when the cycle doesn't start immediately in the linked list? Consider simulating the fast and slow pointer technique over the following example:

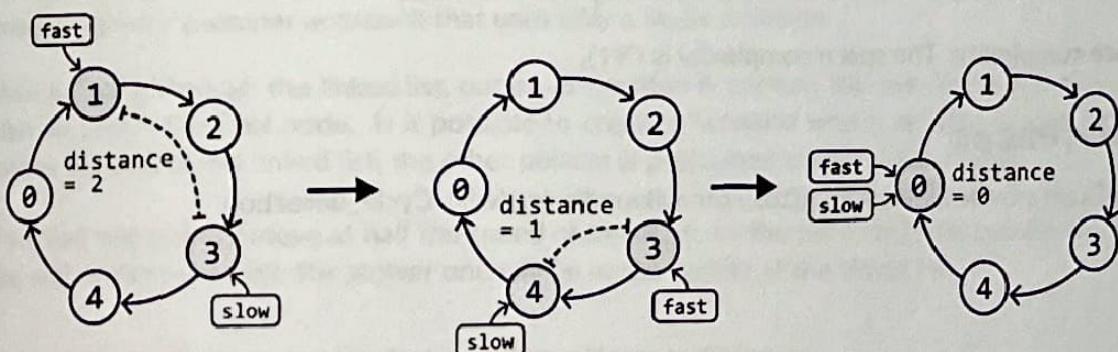


Again, the pointers eventually met in the cycle despite the fact that fast and slow entered the cycle at different times.

Will fast always catch up with slow?

In both cases, it might seem like the fast pointer could keep overtaking the slow pointer without ever meeting it, but this isn't true. Here's an easier way to understand why they will meet.

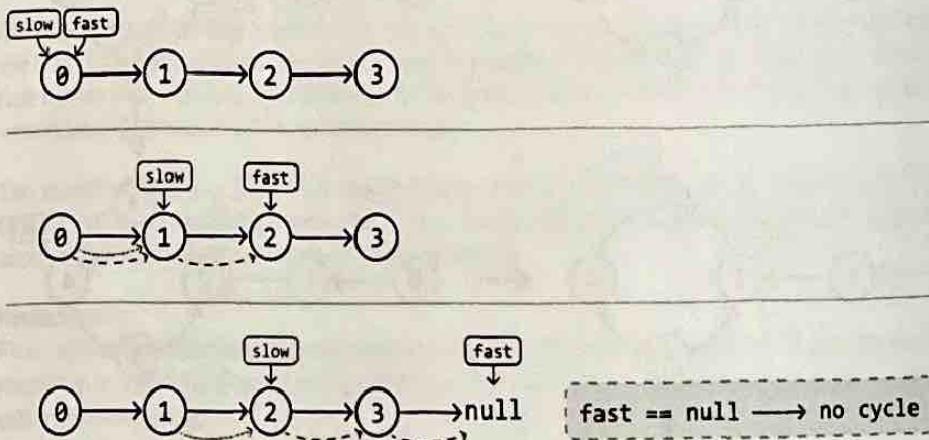
The fast pointer moves 2 steps at a time, and the slow pointer moves 1 step at a time, so the fast pointer will gain a distance of 1 node over the slow pointer at each iteration. This can be observed below, where the distance between the fast and slow pointers reduces by one in each iteration until they inevitably meet.



Therefore, the maximal number of steps required for the fast pointer to catch up with the slower pointer is k steps (once both are in the cycle), where k is the length of the cycle. In the worst case, the cycle will contain all the linked list's nodes, and the pointers will eventually meet in n steps.

No cycle

The final case is when there is no cycle. In this case, the fast pointer will eventually reach the end of the linked list and exit the while-loop:



Implementation

This algorithm is formally known as 'Floyd's Cycle Detection' algorithm [1].

```
def linked_list_loop(head: ListNode) -> bool:
    slow = fast = head
    # Check both 'fast' and 'fast.next' to avoid null pointer
    # exceptions when we perform 'fast.next' and 'fast.next.next'.
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if fast == slow:
            return True
    return False
```

Complexity Analysis

Time complexity: The time complexity of `linked_list_loop` is $O(n)$ because the fast pointer will meet the slow pointer in a linear number of steps, as described.

Space complexity: The space complexity is $O(1)$.

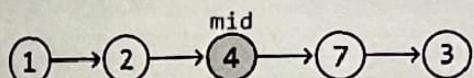
References

- [1] Floyd's Cycle Detection: https://en.wikipedia.org/wiki/Cycle_detection

Linked List Midpoint

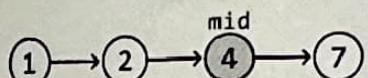
Given a singly linked list, find and return its middle node. If there are two middle nodes, return the second one.

Example 1:



Output: Node 4

Example 2:



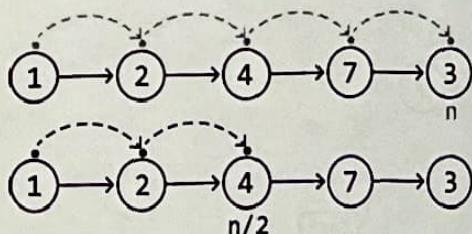
Output: Node 4

Constraints:

- The linked list contains at least one node.
- The linked list contains unique values.

Intuition

The most intuitive approach to solve this problem is to traverse the linked list to find its length (n), and then traverse the linked list a second time to find the middle node ($n/2$):



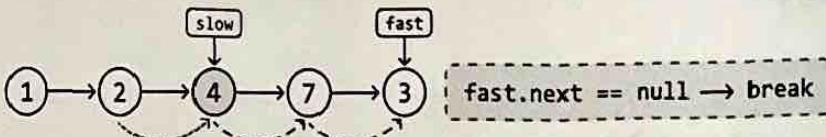
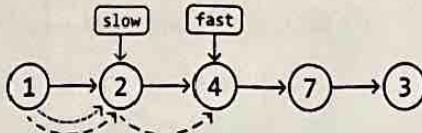
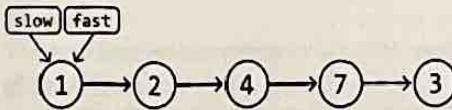
This approach solves the problem in $O(n)$ time, but requires two iterations to find the midpoint. However, there's a cleaner approach that uses only a single iteration.

When iterating through the linked list, our exact position is unclear. We only know where we are when we reach the final node. Is it possible to create a scenario where, as soon as one pointer reaches the end of the linked list, the other pointer is positioned at the middle node?

If we had one pointer move at half the speed of the other, by the time the faster pointer reaches the end of the linked list, the slower one will be at the middle of the linked list.

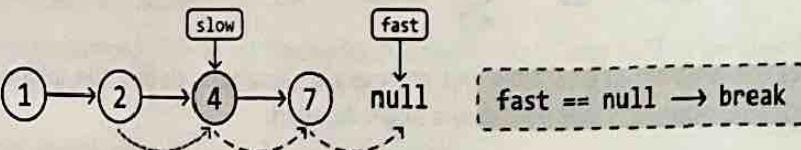
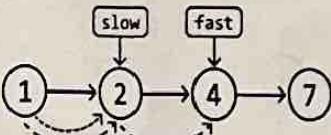
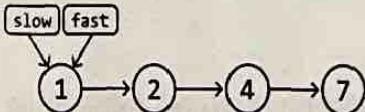
We can achieve this by using the **fast and slow pointer technique**:

- Move a slow pointer one node at a time.
- Move a fast pointer two nodes at a time.



One thing we must be careful about is when we should stop advancing the fast pointer. We need to stop when the slow pointer reaches the middle node. When the linked list length is odd, this happens when `fast.next` equals `null`, as we can see above.

What about when the length of the linked list is even? Consider the below example:



As you can see, to have slow point at the second middle node, we'd need to stop fast when it reaches a null node.

Implementation

```
def linked_list_midpoint(head: ListNode) -> ListNode:
    slow = fast = head
    # When the fast pointer reaches the end of the list, the slow
    # pointer will be at the midpoint of the linked list.
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
```

```
return slow
```

Complexity Analysis

Time complexity: The time complexity of `linked_list_midpoint` is $O(n)$ because we traverse the linked list linearly using two pointers.

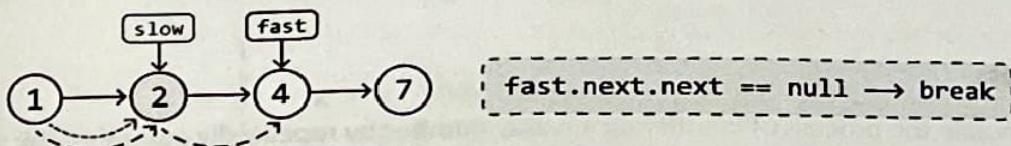
Space complexity: The space complexity is $O(1)$.

Stop and Think

What if you were asked to modify your algorithm to return the first middle node when the linked list is of even length?

Answer:

Following the same method, we should stop advancing the fast pointer when `fast.next.next` is null. This way, `slow` will end up pointing to the first middle node:



Interview Tip

Tip: Be prepared to address potential gaps in the information provided. 

During an interview, it's possible the interviewer won't specify which middle node should be returned for linked lists of even length, leaving it up to you to recognize and address this special scenario. You might be expected to identify ambiguities like this and actively engage with the interviewer to discuss a suitable resolution.

Happy Number

In number theory, a happy number is defined as a number that, when repeatedly subjected to the process of squaring its digits and summing those squares, eventually leads to 1 [1]. An unhappy number will never reach 1 during this process, and will get stuck in an infinite loop.

Given an integer, determine if it's a happy number.

Example:

Input: $n = 23$

Output: True

Explanation: $2^2 + 3^2 = 13 \Rightarrow 1^2 + 3^2 = 10 \Rightarrow 1^2 + 0^2 = 1$

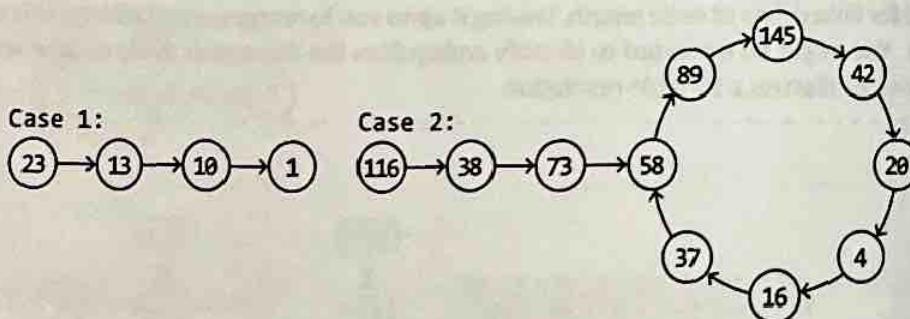
Intuition

We can simulate the process of identifying a happy number by repeatedly summing the squares of each digit of a number, and then applying the same process to the resulting sum.

According to the problem statement, this process could conclude in one of two ways:

- Case 1: the process continues until the final number is 1.
- Case 2: the process gets stuck in an infinite loop.

If we diagram both scenarios, we observe something interesting:



This looks quite similar to a linked list problem. In particular, the problem of determining if a linked list has a cycle (case 2) or doesn't (case 1).

We can reduce this problem to the same cycle detection challenge as the *Linked List Loop* problem. By applying the fast and slow pointer technique (i.e., Floyd's Cycle Detection algorithm), we can efficiently determine if a cycle exists.

However, in this problem, we don't have an actual linked list to perform the fast and slow pointer algorithm on. Therefore, we need to find a way to traverse the sequence of numbers generated in the happy number process.

Conveniently, we already know what the 'next' number in the sequence is for any number x . As described in the problem statement, the next number can be calculated by summing the square

of each digit of x . So, if each number were a node in a linked list, we could get the 'next' node by calculating the next number in the sequence.

Getting the next number in the sequence

To calculate the next number of x , we need a way to access each digit of x . This can be done in two steps:

1. The modulo operation ($x \% 10$) is used to extract the last digit of a number x .
2. Divide x by 10 ($x = x / 10$) to truncate the last digit, positioning the next digit as the new last digit.

We can see this unfold in full below for $x = 123$:

```
x = 123
get_next_num(x):
    x = 1 2 ③ → digit = x % 10
                           = 123 % 10
                           = 3

    x = x / 10
      = 1 2 3 / 10
      = 1 ② → digit = x % 10
                           = 12 % 10
                           = 2

    x = x / 10
      = 1 2 / 10
      = ① → digit = x % 10
                           = 1 % 10
                           = 1

    x = x / 10
      = 1 / 10
      = 0 → stop

next_num = 32 + 22 + 12
          = 14
```

Now that we have a way to traverse the sequence, we can implement Floyd's Cycle Detection algorithm. To start, set the fast and slow pointers at the start of this sequence. Then move the pointers as follows:

- Advance the slow pointer one number at a time:
 $\text{slow} = \text{get_next_num}(\text{slow})$.
- Advance the fast pointer two numbers at a time:
 $\text{fast} = \text{get_next_num}(\text{get_next_num}(\text{fast}))$.

If the fast and slow pointers meet during the process, it indicates the presence of a cycle, meaning the number is not a happy number. Otherwise, the algorithm will end when we reach 1, in which case the number is a happy number.

Implementation

```
def happy_number(n: int) -> bool:
```

```

slow = fast = n
while True:
    slow = get_next_num(slow)
    fast = get_next_num(get_next_num(fast))
    if fast == 1:
        return True
    # If the fast and slow pointers meet, a cycle is detected.
    # Hence, 'n' is not a happy number.
    elif fast == slow:
        return False

def get_next_num(x: int) -> int:
    next_num = 0
    while x > 0:
        # Extract the last digit of 'x'.
        digit = x % 10
        # Truncate (remove) the last digit from 'x' using floor
        # division.
        x //= 10
        # Add the square of the extracted digit to the sum.
        next_num += digit ** 2
    return next_num

```

Complexity Analysis

Time complexity: The time complexity of `happy_number` is $O(\log(n))$. The full analysis of this time complexity is quite complicated and beyond the scope of interviews. For interested readers, please see the reference below for a detailed analysis [2].

Space complexity: The space complexity is $O(1)$.

Interview Tip

Tip: Visualize the problem.



At first glance, this problem seems like it requires mathematical reasoning to solve. However, when we visualized the problem, we were able to formulate a solution using an algorithm we already know (Floyd's Cycle Detection). Visualizing a problem can help uncover hidden patterns or data structures that can lead to the solution.

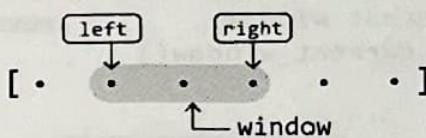
References

- [1] Happy Number: https://en.wikipedia.org/wiki/Happy_number
- [2] The time complexity analysis of Happy Numbers can be found in the bonus PDF.

Sliding Windows

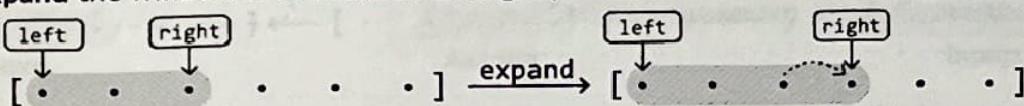
Introduction to Sliding Windows

The sliding window technique is a subset of the two-pointer pattern, as it uses two pointers (generally **left** and **right**) to define the bounds of a 'window' in iterable data structures like arrays. The window defines a subcomponent of the data structure (e.g., a subarray or substring), and it *slides* across the data structure unidirectionally, typically searching for a subcomponent that meets a certain requirement.

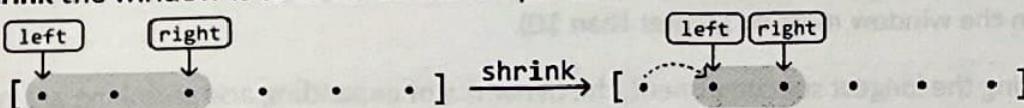


Sliding windows are particularly valuable in scenarios where algorithms might otherwise rely on using two nested loops to search through all possible subcomponents to find an answer, resulting in an $O(n^2)$ time complexity, or worse. When using a sliding window, the subcomponent(s) we're looking for can usually be found in $O(n)$ time, in comparison. But before discussing how they work, let's establish some terminology:

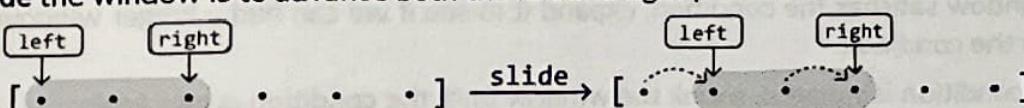
- To **expand** the window is to advance the right pointer:



- To **shrink** the window is to advance the left pointer:



- To **slide** the window is to advance both the left and right pointers:



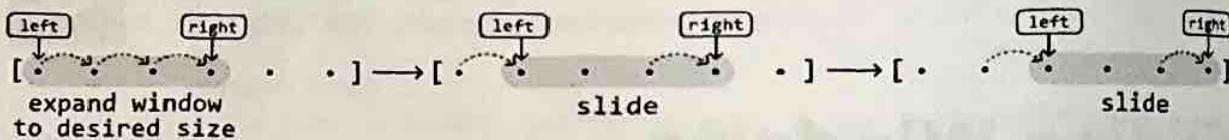
Note that sliding is equivalent to expanding and shrinking the window at the same time.

Now, let's break down the two main types of sliding window algorithms:

1. Fixed sliding window.
2. Dynamic sliding window.

Fixed Sliding Window

A fixed sliding window maintains a specific length as it slides across a data structure.



We use the fixed sliding window technique when the problem asks us to find a subcomponent of a certain length. If we know what this length is, we can fix our window to that length and slide it through the data structure.

If a fixed window of length k traverses a data structure from start to finish, it's guaranteed to see every subcomponent of length k in that data structure.

Below is a generic template of how a fixed sliding window traverses a data structure.

```
left = right = 0
while right < n:
    # If the window has reached the expected fixed length, we slide
    # the window (move both left and right).
    if right - left + 1 == fixed_window_size:
        # Process the current window.
        result = process_current_window()
    left += 1
    right += 1
```

Dynamic Sliding Window

Unlike fixed sliding windows, dynamic windows can expand or shrink in length as they traverse a data structure.



Generally, dynamic sliding windows can be applied to problems that ask us to find the longest or shortest subcomponent that satisfies a condition (for example, a condition could be that the numbers in the window must be greater than 10).

When finding the longest subcomponent, the dynamics of expanding and shrinking are typically as follows:

- If a window satisfies the condition, expand it to see if we can find a longer window that also meets the condition.
- If the condition is violated, shrink the window until the condition is met again.

Below is a generic template demonstrating how this works:

```
left = right = 0
while right < n:
    # While the condition is violated, the window is invalid, so
```

```
# shrink the window by advancing the left pointer.  
while condition is violated:  
    left += 1  
# Once the window is valid, process it and then expand the window  
# by advancing the right pointer.  
result = process_current_window()  
right += 1
```

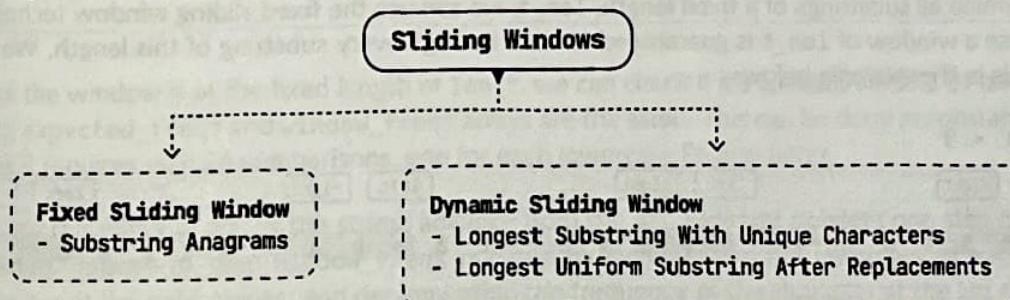
Note that the provided templates for fixed and dynamic windows primarily emphasize the movement of the left and right pointers rather than the specifics of updating the window itself. If the problem requires updating the window, the logic around it will be highly context-dependent. This is explored in detail through specific problems in this chapter.

Real-world Example

Buffering in video streaming: In video streaming, a dynamic sliding window can be used to manage buffering and ensure smooth playback.

For instance, when streaming a video, the player downloads chunks of the video data and stores them in a buffer. A sliding window controls which part of the video is buffered, with the window 'sliding' forward as the video plays. The sliding window ensures the video player can adapt to varying network conditions by dynamically adjusting the buffer size and position, leading to a smoother streaming experience for the viewer.

Chapter Outline



Substring Anagrams

Given two strings, s and t , both consisting of lowercase English letters, return the number of substrings in s that are anagrams of t .

An **anagram** is a word or phrase formed by rearranging the letters of another word or phrase, using all the original letters exactly once.

Example:

Input: $s = \text{"caabab"}$, $t = \text{"aba"}$

Output: 2

Explanation: There is an anagram of t starting at index 1 ("caabab") and another starting at index 2 ("caabab")

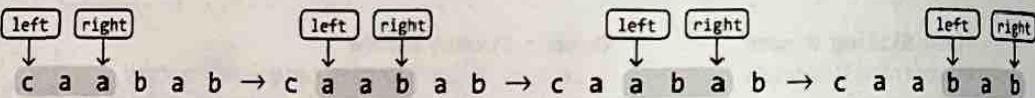
Intuition

We can reframe how we think about an anagram by altering the provided definition. A substring of s qualifies as an anagram of t if it contains exactly the same characters as t in any order.

For a substring in s to be an anagram of t , it must have the same length as t (denoted as len_t). This means we only need to consider substrings of s that match the length len_t , which saves us from examining every possible substring.

To examine all substrings of a fixed length, len_t , we can use the **fixed sliding window** technique because a window of len_t is guaranteed to slide through every substring of this length. We can see this in the example below:

$\text{len_t} = 3$



Now, we just need a way to check if a window is an anagram of t . Remember that in an anagram, the order of the letters doesn't matter; only the frequency of each letter does. By comparing the frequency of each character in a window against the frequencies of characters in string t , we can determine if that window is an anagram of t .

Let's explore this reasoning with an example. Before starting the sliding window algorithm, we need a way to store the frequencies of the characters in string t . We could use a hash map for this, or an array, expected_freqs , to store the frequencies of each character in string t .

expected_freqs is an integer array of size 26, with each index representing one of the lowercase English letters (0 for 'a', 1 for 'b', and so on, up to 25 for 'z').

$s = \text{c a a b a b}$, $t = \text{a b a}$

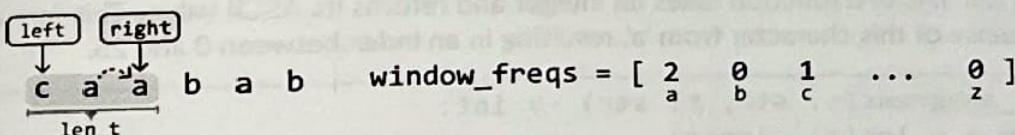
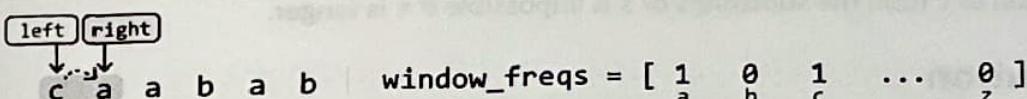
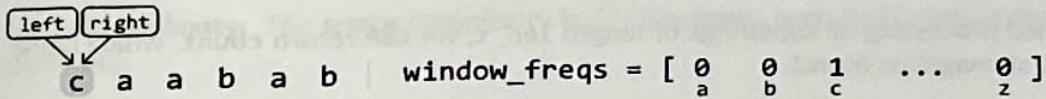
$\text{expected_freqs} = [\begin{matrix} 2 & 1 & 0 & \dots & 0 \end{matrix}]$

Diagram showing the expected frequency array for string t. The array has 26 slots, indexed from 0 to 25. The values are: a=2, b=1, c=0, ..., z=0.

To set up the sliding window algorithm, let's define the following components:

- Left and right pointers: Initialize both at the start of the string to define the window's boundaries.
- `window_freqs`: Use an array of size 26 to keep track of the frequencies of characters within the window.
- `count`: Maintain a variable to count the number of anagrams detected.

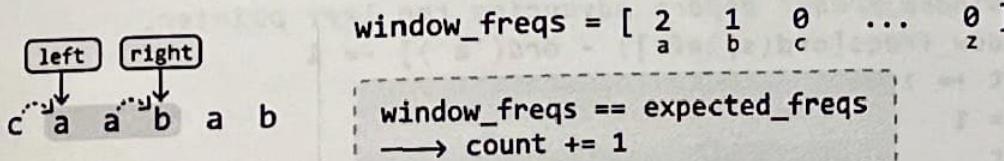
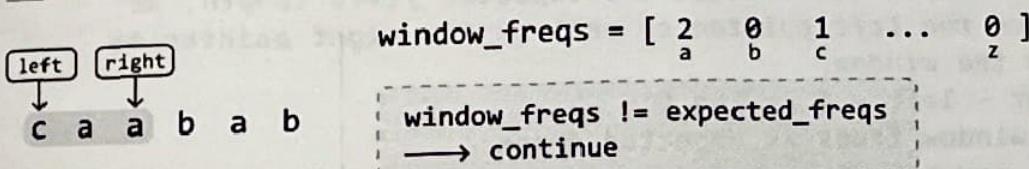
Before we slide the window, we first need to expand it to a fixed length of `len_t`. This can be done by advancing the right pointer until the window length is equal to `len_t`. As we expand, ensure to keep `window_freqs` updated to reflect the frequencies of the characters in the window:

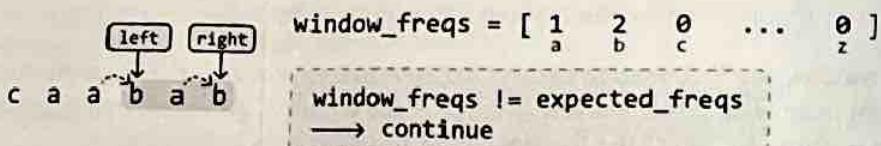
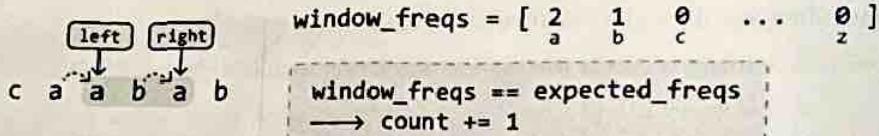


Once the window is at the fixed length of `len_t`, we can check if it's an anagram of `t` by checking if the `expected_freqs` and `window_freqs` arrays are the same. This can be done in constant time since it requires only 26 comparisons, one for each lowercase English letter.

To slide this window across the string, advance both the left and right pointers one step in each iteration. Ensure to keep `window_freqs` updated by incrementing the frequency of each new character at the right pointer and decrementing the frequency of the character at the left pointer as we move passed this left character:

$$\text{expected_freqs} = [2 \underset{a}{\cancel{1}} \underset{b}{\cancel{0}} \underset{c}{\cancel{...}} \underset{z}{\cancel{0}}]$$





Once we've finished processing all substrings of length `len_t`, we can return `count`, which represents the number of anagrams found.

A small optimization we can make is returning 0 if `t`'s length exceeds the length of `s` because forming an anagram of `t` from the substrings of `s` is impossible if `t` is longer.

Implementation

In Python, we can use `ord(character) - ord('a')` to find the index of a lowercase English letter in an array of size 26. The `ord` function takes an integer and returns its ASCII value. This formula calculates the distance of this character from 'a', resulting in an index between 0 and 25.

```
def substring_anagrams(s: str, t: str) -> int:
    len_s, len_t = len(s), len(t)
    if len_t > len_s:
        return 0
    count = 0
    expected_freqs, window_freqs = [0] * 26, [0] * 26
    # Populate 'expected_freqs' with the characters in string 't'.
    for c in t:
        expected_freqs[ord(c) - ord('a')] += 1
    left = right = 0
    while right < len_s:
        # Add the character at the right pointer to 'window_freqs'
        # before sliding the window.
        window_freqs[ord(s[right]) - ord('a')] += 1
        # If the window has reached the expected fixed length, we
        # advance the left pointer as well as the right pointer to
        # slide the window.
        if right - left + 1 == len_t:
            if window_freqs == expected_freqs:
                count += 1
            # Remove the character at the left pointer from
            # 'window_freqs' before advancing the left pointer.
            window_freqs[ord(s[left]) - ord('a')] -= 1
        left += 1
        right += 1
    return count
```

Complexity Analysis

Time complexity: The time complexity of `substring_anagrams` is $O(n)$, where n denotes the length of s . Here's why:

- Populating the `expected_freqs` array takes $O(m)$ time, where m denotes the length of t . Since m is guaranteed to be less than or equal to n at this point, it's not a dominant term in the time complexity.
- Then, we traverse string s linearly with two pointers, which takes $O(n)$ time.
- Note that at each iteration, the comparison performed between the two frequency arrays (`expected_freqs` and `window_freqs`) takes $O(1)$ time because each array contains only 26 elements.

Space complexity: The space complexity is $O(1)$ because each frequency array contains only 26 elements.

Longest Substring With Unique Characters

Given a string, determine the length of its longest substring that consists only of unique characters.

Example 1:

Input: $s = \text{"abcba"}$

Output: 3

Explanation: Substring "abc" is the longest substring of length 3 that contains unique characters ("cba" also fits this description).

Intuition

The brute force approach involves examining all possible substrings and checking if any consist of exclusively unique characters. Let's break down this approach:

- Checking a substring for uniqueness can be done in $O(n)$ time by scanning the substring and using a hash set to keep track of each character, where n denotes the length of s . If we encounter a character already in the hash set, we know it's a duplicate character.
- Iterating through all possible substrings takes $O(n^2)$ time.

This means the brute force approach would take $O(n^3)$ time overall. This is quite slow, largely because we look through every substring. Is there a way to reduce the number of substrings we examine?

Sliding window

Sliding window approaches can be quite useful for problems that involve substrings. In particular, because we're looking for the longest substring that satisfies a specific condition (i.e., contains unique characters), a dynamic sliding window algorithm might be the way to go, as discussed in the introduction.

We can categorize any window in two ways. A window either:

- Consists only of unique characters (a window with no duplicate characters).

no duplicate characters

- Contains at least one character of a frequency greater than 1.

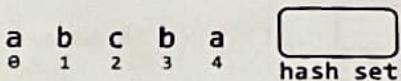
'b' is a duplicate

If A is true, we should expand the window by advancing the right pointer to find a longer window that also contains no duplicates.

If B is true because we encounter a duplicate character in the window, we should shrink the window by advancing the left pointer until it no longer contains a duplicate.

Let's try this strategy over the following example. We initialize a hash set to keep track of the

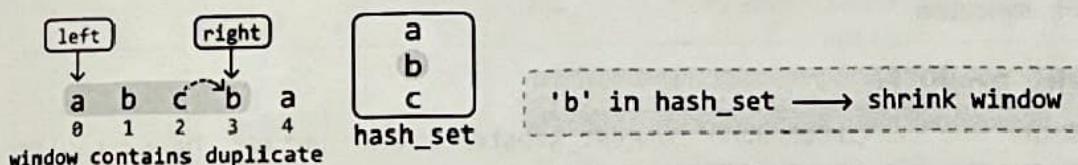
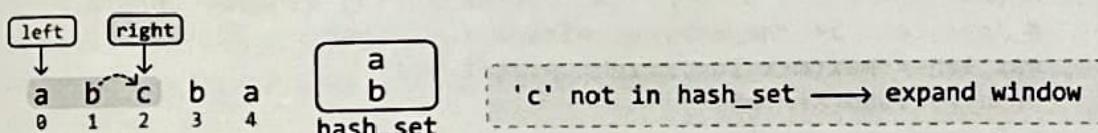
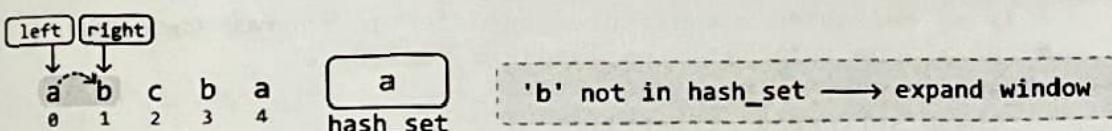
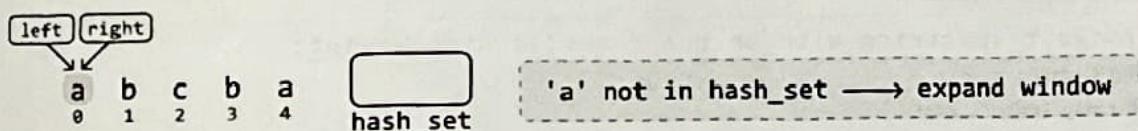
characters in a window.



To implement the sliding window technique, we should establish the following:

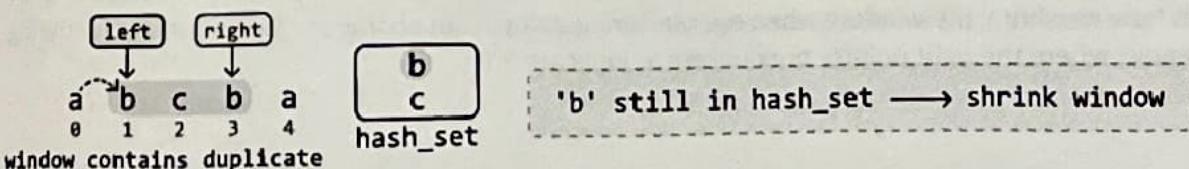
- **Left and right pointers:** Initialize both at the start of the string to define the window's boundaries.
- **hash_set:** Maintain a hash set to record the unique characters within the window, updating it as the window expands. Note, the hash set shown in the diagram displays its state before the character at the right pointer is added to it.

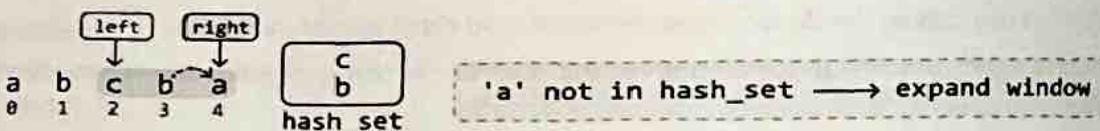
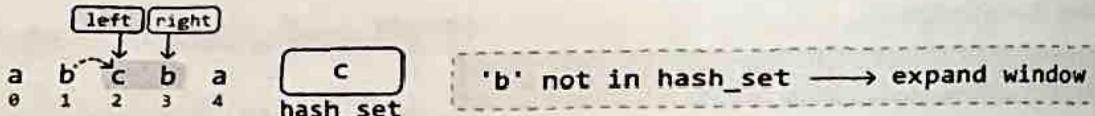
Now, let's start looking for the longest window. Expand the window from the beginning of the string by advancing the right pointer. Keep expanding until a duplicate character is found:



We see above that the 'b' at index 3 is a duplicate character in the window because 'b' is already in the hash set.

Now that we found a duplicate, we should shrink the window by advancing the left pointer until the window no longer contains a duplicate 'b'. Once the window is valid again, continue expanding:





Expanding the window any further will cause the right pointer to exceed the string's boundary, at which point we end our search. The longest substring we've found with no duplicates is of length 3. We can use the variable `max_len` to keep track of this length during our search.

Implementation

```
def longest_substring_with_unique_chars(s: str) -> int:
    max_len = 0
    hash_set = set()
    left = right = 0
    while right < len(s):
        # If we encounter a duplicate character in the window, shrink
        # the window until it's no longer a duplicate.
        while s[right] in hash_set:
            hash_set.remove(s[left])
            left += 1
        # Once there are no more duplicates in the window, update
        # 'max_len' if the current window is larger.
        max_len = max(max_len, right - left + 1)
        hash_set.add(s[right])
        # Expand the window.
        right += 1
    return max_len
```

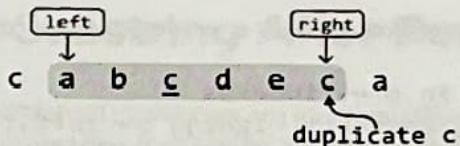
Complexity Analysis

Time complexity: The time complexity of `longest_substring_with_unique_chars` is $O(n)$ because we traverse the string linearly with two pointers.

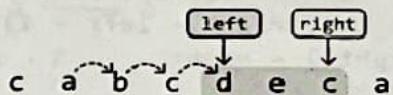
Space complexity: The space complexity is $O(m)$ because we use a hash set to store unique characters, where m represents the total number of unique characters within the string.

Optimization

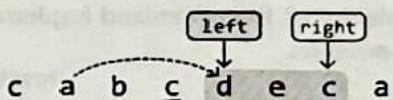
The above approach solves the problem, but we can still optimize it. The optimization has to do with how we shrink the window when encountering a duplicate character. Consider the following example, where the right pointer encounters a duplicate 'c':



In the previous approach, we respond to encountering a duplicate by continuously advancing the left pointer to shrink the window until the window no longer contains a duplicate:



The crucial insight here is that we advanced the left pointer until it passed the previous occurrence of 'c' in the window. This indicates that if we know the index of the previous occurrence of 'c', we can move our left pointer immediately past that index to remove it from the window:



This gives us a new strategy for advancing the left pointer: if the right pointer encounters a character whose previous index (i.e., previous occurrence) is in the window, move the left pointer one index past that previous index.

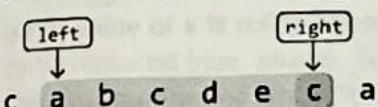
We can use a hash map (`prev_indexes`) to store the previous index of each character in the string.

Now we just need to ensure the previous index of a character is in the window. To do this, we compare its index to the left pointer:

- If this index is after the left pointer, it's inside the window.
- If it is before the left pointer, it's outside the window.

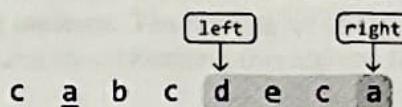
Below is a visual of how to check whether a character is inside the window:

Previous index inside the window:



`prev_indexes['c'] = 3 >= left`
 \rightarrow inside window
 \rightarrow duplicate in the window

Previous index outside the window:



`prev_indexes['a'] = 1 < left`
 \rightarrow outside window
 \rightarrow not a duplicate in the window

Implementation – Optimized Approach

```
def longest_substring_with_unique_chars_optimized(s: str) -> int:
    max_len = 0
    prev_indexes = {}
    left = right = 0
    while right < len(s):
        # If a previous index of the current character is present in
```

```
# the current window, it's a duplicate character in the
# window.
if (s[right] in prev_indexes
    and prev_indexes[s[right]] >= left):
    # Shrink the window to exclude the previous occurrence of
    # this character.
    left = prev_indexes[s[right]] + 1
# Update 'max_len' if the current window is larger.
max_len = max(max_len, right - left + 1)
prev_indexes[s[right]] = right
# Expand the window.
right += 1
return max_len
```

Complexity Analysis

Time complexity: The time complexity of the optimized implementation is $O(n)$ because we traverse the string linearly with two pointers.

Space complexity: The space complexity is $O(m)$ because we use a hash map to store unique characters, where m represents the total number of unique characters within the string.

Longest Uniform Substring After Replacements

A uniform substring is one in which all characters are identical. Given a string, determine the length of the longest uniform substring that can be formed by replacing up to k characters.

Example:

a a b^c c d^c c c a → longest uniform substring
a a c c c c c a

Input: $s = "aabdcda"$, $k = 2$

Output: 5

Explanation: if we can only replace 2 characters, the longest uniform substring we can achieve is "cccc", obtained by replacing b and d with c.

Intuition

Determining if a substring is uniform

Before we try finding the longest uniform substring, let's first determine the most efficient way to make a string uniform with the fewest character replacements. Consider the example below:

a b a a b a c

Which characters should we replace to ensure the minimum number of replacements are performed to make the string uniform? There are three main choices: make the string all 'a's, or all 'b's, or all 'c's. The most efficient choice, requiring the fewest replacements, is to make all characters 'a', which involves just three replacements:

a b^a a a b^a a c → a a a a a a a

The key observation is that the minimum number of replacements needed to achieve uniformity is obtained by replacing all characters except the most frequent one.

This suggests that if we know the highest frequency of a character in a substring, we can determine if our value of k is sufficient to make that substring uniform. The number of characters that need to be replaced (`num_chars_to_replace`) can be found by subtracting this highest frequency from the total number of characters in the substring:

a b a a b a c

```
num_chars_to_replace = len(substring) - highest_freq
                      = 7 - 4
                      = 3
```

Once we've calculated `num_chars_to_replace` for a given substring, we can assess if the substring can be made uniform:

- If `num_chars_to_replace` $\leq k$, the substring can be made uniform.
- If `num_chars_to_replace` $> k$, the substring cannot be made uniform.

To calculate `num_chars_to_replace`, we need to know the value of `highest_freq`. This requires tracking the frequency of each character, which can be efficiently managed using a hash map

(`freqs`). This hash map allows us to update `highest_freq` whenever we encounter a character with a higher frequency. Below is an illustration of how `freqs` is updated:

↓ a b b c	<table border="1"><tr><td colspan="2">freqs</td></tr><tr><td>a</td><td>1</td></tr><tr><td>char</td><td>freq</td></tr></table>	freqs		a	1	char	freq	<code>highest_freq = max(highest_freq, freqs['a'])</code> = 1				
freqs												
a	1											
char	freq											
↓ a b b c	<table border="1"><tr><td colspan="2">freqs</td></tr><tr><td>a</td><td>1</td></tr><tr><td>b</td><td>1</td></tr><tr><td>char</td><td>freq</td></tr></table>	freqs		a	1	b	1	char	freq	<code>highest_freq = max(highest_freq, freqs['b'])</code> = 1		
freqs												
a	1											
b	1											
char	freq											
↓ a b b c	<table border="1"><tr><td colspan="2">freqs</td></tr><tr><td>a</td><td>1</td></tr><tr><td>b</td><td>2</td></tr><tr><td>char</td><td>freq</td></tr></table>	freqs		a	1	b	2	char	freq	<code>highest_freq = max(highest_freq, freqs['b'])</code> = 2		
freqs												
a	1											
b	2											
char	freq											
↓ a b b c	<table border="1"><tr><td colspan="2">freqs</td></tr><tr><td>a</td><td>1</td></tr><tr><td>b</td><td>2</td></tr><tr><td>c</td><td>1</td></tr><tr><td>char</td><td>freq</td></tr></table>	freqs		a	1	b	2	c	1	char	freq	<code>highest_freq = max(highest_freq, freqs['c'])</code> = 2
freqs												
a	1											
b	2											
c	1											
char	freq											

Now that we have the tools to determine if a substring can be made uniform, the next step is to figure out how to identify the longest uniform substring. Let's explore a technique that lets us do this.

Dynamic sliding window

We know sliding windows can be useful for solving problems involving substrings. This problem requires that we find the longest substring that satisfies a specific condition:

| `num_chars_to_replace <= k`

So, a dynamic sliding window might be appropriate, as discussed in the chapter introduction.

We can use the above condition to determine how to expand or shrink the window:

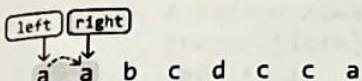
- If the condition is met (i.e., the window is valid), we **expand** the window to find a longer window that still meets this condition.
- If the condition is violated (i.e., the window is invalid), we **shrink** the window until it meets the condition again.

Let's see how this works over the example below:

a a b c d c c a, $k = 2$

Start by defining the left and right boundaries of the window at index 0. Continue expanding the window for as long as it satisfies our condition (`num_chars_to_replace <= k`):

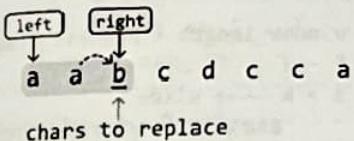
$k = 2$	<table border="1"><tr><td>left</td><td>right</td></tr></table>	left	right	<code>highest_freq = 1</code> $num_chars_to_replace = window_length - highest_freq$ = 1 - 1 = 0 \leq k \longrightarrow expand
left	right			



```

highest_freq = 2
num_chars_to_replace = window_length - highest_freq
= 2 - 2
= 0 ≤ k → expand

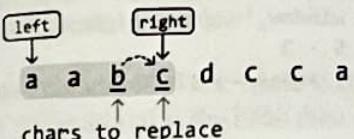
```



```

highest_freq = 2
num_chars_to_replace = window_length - highest_freq
= 3 - 2
= 1 ≤ k → expand

```

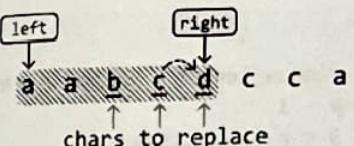


```

highest_freq = 2
num_chars_to_replace = window_length - highest_freq
= 4 - 2
= 2 ≤ k → expand

```

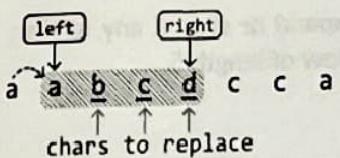
Once the window expands to the fifth character ('d'), it will contain 3 characters that must be replaced to make the window uniform. Since we can only replace up to $k = 2$ characters, the window is invalid. So, we shrink the window:



```

highest_freq = 2
num_chars_to_replace = window_length - highest_freq
= 5 - 2
= 3 > k → shrink

```



highest_freq = 2 X

Notice that after shrinking the window, the value of `highest_freq` is still 2, which is no longer correct. Recall that our current method for updating `highest_freq` only increases it when encountering a character with a higher frequency, meaning it can only remain the same or increase, but it can never decrease.

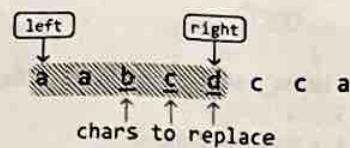
One way to work around this is to develop a new method for updating `highest_freq` that accurately decreases it when the highest frequency in a window decreases. However, our goal is to find the longest substring that meets the condition, so shrinking the window might not even be necessary. The crucial point here is that when we find a valid window of a certain length, no shorter window will provide a longer uniform substring.

This means we can just slide the window instead of shrinking it whenever we encounter an invalid window, effectively maintaining the length of the current window.

With this observation, we should correct our previous logic:

- If the window satisfies the condition: expand.
- If the window doesn't satisfy the condition: slide.

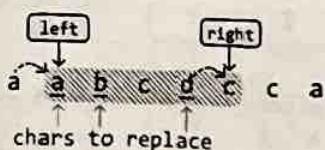
Let's correct the action taken in the first invalid window above by sliding instead of shrinking. Then, we can continue processing the rest of the string.



```

highest_freq = 2
num_chars_to_replace = window_length - highest_freq
= 5 - 2
= 3 > k → slide

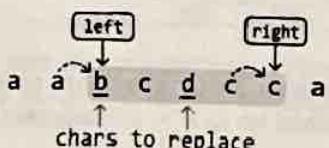
```



```

highest_freq = 2
num_chars_to_replace = window_length - highest_freq
= 5 - 2
= 3 > k → slide

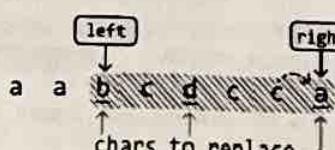
```



```

highest_freq = 3
num_chars_to_replace = window_length - highest_freq
= 5 - 3
= 2 ≤ k → expand

```



```

highest_freq = 3
num_chars_to_replace = window_length - highest_freq
= 6 - 3
= 3 < k → slide

```

The above window is the final window because we cannot expand or slide it any further. The longest valid window encountered during this process is a window of length 5.

Implementation

```

def longest_uniform_substring_after_replacements(s: str, k: int) -> int:
    freqs = {}
    highest_freq = max_len = 0
    left = right = 0
    while right < len(s):
        # Update the frequency of the character at the right pointer
        # and the highest frequency for the current window.
        freqs[s[right]] = freqs.get(s[right], 0) + 1
        highest_freq = max(highest_freq, freqs[s[right]])
        # Calculate replacements needed for the current window.
        num_chars_to_replace = (right - left + 1) - highest_freq
        # Slide the window if the number of replacements needed exceeds
        # 'k'. The right pointer always gets advanced, so we just need
        # to advance 'left'.
        if num_chars_to_replace > k:
            # Remove the character at the left pointer from the hash map

```

```
# before advancing the left pointer.  
freqs[s[left]] -= 1  
left += 1  
# Since the length of the current window increases or stays the  
# same, assign the length of the current window to 'max_len'.  
max_len = right - left + 1  
# Expand the window.  
right += 1  
return max_len
```

Complexity Analysis

Time complexity: The time complexity of `longest_uniform_substring_after_replacements` is $O(n)$, where n denotes the length of the input string. This is because we traverse the string linearly with two pointers.

Space complexity: The space complexity is $O(m)$, where m is the number of unique characters in the string stored in the hash map `freqs`.

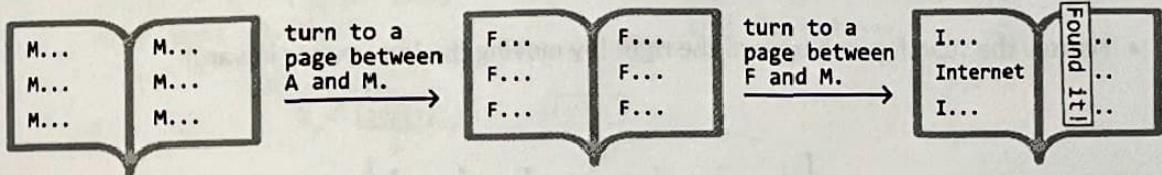
Binary Search

Introduction to Binary Search

Let's say you have a standard, hard-copy English dictionary, and you want to find the definition of the word 'internet.' How could we do this?

One approach is to flick through it page by page, until reaching the page with 'internet.' But this is inefficient, and realistically, you're more likely to immediately open a page somewhere in the middle.

Let's say you do this and land on a page of words beginning with 'M.' Realizing that 'internet' is somewhere to the left of 'M' in the alphabet, you open a page somewhere between the letters 'A' and 'M.' You continue this process of checking if 'internet' is to the right or left of the current page, and narrowing the search down until you find the correct page. With this method, you've significantly reduced the number of pages to be searched, in contrast to linearly turning every page in order. This is the fundamental concept of binary search.



Implementing Binary Search

Binary search is an algorithm that searches for a value in a sorted data set.

Even experienced developers can find it quite tricky to implement binary search correctly because "the devil's in the detail."

- How should the boundary variables `left` and `right` be initialized?
- Should we use `left < right` or `left ≤ right` as the exit condition in our while-loop?
- How should the boundary variables be updated? Should we choose `left = mid`, `left = mid + 1`, `right = mid`, or `right = mid - 1`?

This chapter offers clear, intuitive guidance on how to master these challenges, and confidently handle even the trickiest edge case.

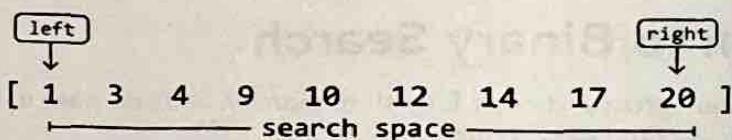
To begin any binary search implementation, do the following:

1. Define the search space.
2. Define the behavior inside the loop for narrowing the search space.
3. Choose an exit condition for the while-loop.
4. Return the correct value.

Let's break down these steps.

1. Defining the search space

The search space encompasses all possible values that may include the value we're searching for. For instance, when searching for a target in a sorted array, the search space should cover the entire array, as the target could be anywhere within it. This is illustrated in the array below, where the left and right pointers define the search space:

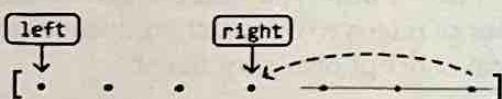


2. Narrowing the search space

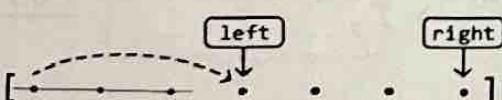
Narrowing the search space involves progressively moving the left or right pointer inward until the search space is reduced to one element or none.

At each point in the binary search, we need to decide how to narrow the search space. We can either:

- Narrow the search space toward the left (by moving the right pointer inward):



- Narrow the search space toward the right (by moving the left pointer inward):



Using the midpoint

We decide whether to move the left or right pointer based on the value in the middle of the search space, indicated by the midpoint variable (`mid`).

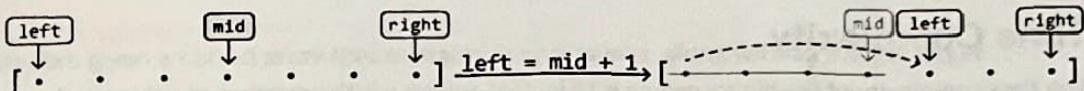


The main question to ask at each iteration of binary search is: **is the value being searched for to the left or the right of the midpoint?**

Here's the general idea: if the value we're looking for is to the right of the midpoint, narrow the search space toward the right. Otherwise, narrow the search space toward the left.

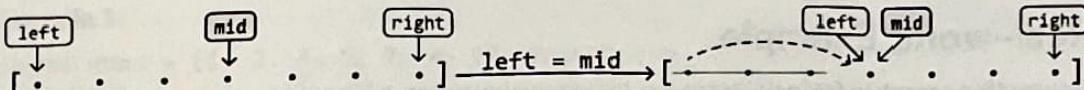
To narrow the search space towards the right, there are two options:

- `left = mid + 1`



We do this if the midpoint value is definitely not the value we're looking for and should be excluded from the search space.

- `left = mid`



We do this if the midpoint value itself could potentially be the value we're looking for and should still be included in the search space.

The exclude/include logic applies when narrowing the search space towards the left, as well (i.e., between `right = mid - 1` and `right = mid`).

Calculating the midpoint

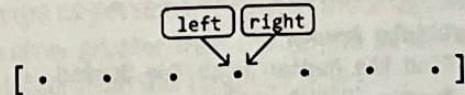
In most cases, the midpoint is calculated using `mid = (left + right) // 2` in Python.¹ To lower the risk of integer overflow, `mid = left + (right - left) // 2` is preferred in many other languages.

3. Choosing an exit condition

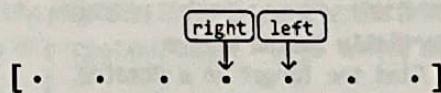
Choosing an exit condition for when the while-loop should terminate can be tricky. Our choices are primarily between `left < right` and `left ≤ right`. Both conditions are applicable in different situations.

- When the exit condition is `left < right`, the while-loop will break when left and right meet.²
- On the other hand, `left ≤ right` ends once left has surpassed right.

'while `left < right`' ends
once `left == right`



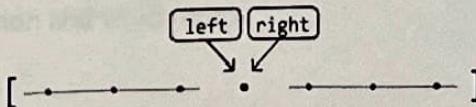
'while `left ≤ right`' ends
once `left > right`



When the left and right pointers meet after exiting the `left < right` condition, both converge to a single value. This is the final value in the search space after the binary search process is complete. This will be the exit condition we use throughout this chapter.

4. Returning the correct value

As previously mentioned, the while-loop terminates once we've narrowed the search space down to a final value (assuming no value was returned during earlier iterations), pointed at by both left and right:



¹ In rare cases, we calculate the midpoint differently, such as when doing an upper-bound binary search. The details are explained in the *First and Last Occurrences of a Number* problem.

² There are some rare situations where left and right will cross, which is also explored in the *First and Last Occurrences of a Number* problem.

This final value is the answer we're looking for, assuming a valid answer exists.

Time Complexity

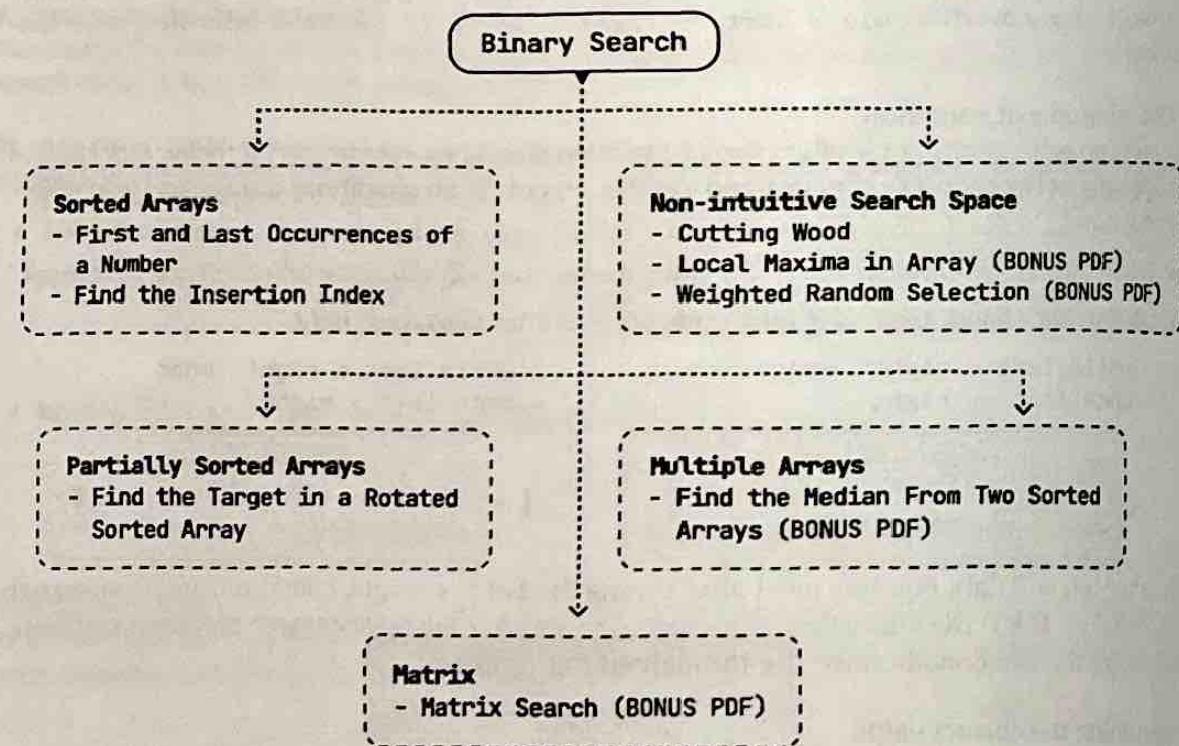
The time complexity of the binary search is $O(\log(n))$, where n is the number of values in the search space. The algorithm is logarithmic because, in each iteration of the algorithm, it divides the search space in half until a single value is located, or no value is found. This reduction by half in each step is characteristic of logarithmic behavior.

Real-world Example

Transaction search in financial systems: In financial systems, a binary search can be used to quickly find a transaction or record by narrowing down the search range, as the data is typically stored in order. This makes it efficient to retrieve specific entries without searching through the entire database.

Chapter Outline

Binary search has many applications, and we explore a range of problems covering various uses.



Find the Insertion Index

You are given a sorted array that contains unique values, along with an integer target.

- If the array contains the target value, return its index.
- Otherwise, return the insertion index. This is the index where the target would be if it were inserted in order, maintaining the sorted sequence of the array.

Example 1:

Input: `nums = [1, 2, 4, 5, 7, 8, 9], target = 4`

Output: 2

Example 2:

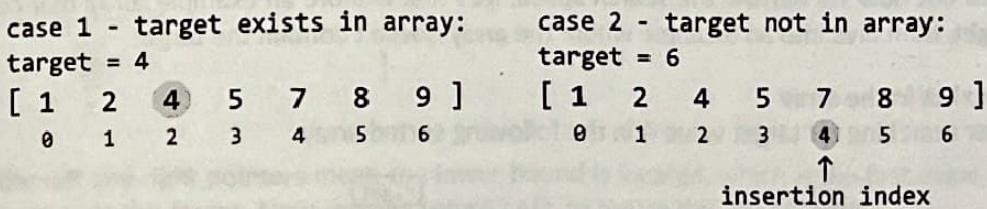
Input: `nums = [1, 2, 4, 5, 7, 8, 9], target = 6`

Output: 4

Explanation: 6 would be inserted at index 4 to be positioned between 5 and 7: [1, 2, 4, 5, 6, 7, 8, 9].

Intuition

The goal of this problem varies depending on whether the sorted input array contains the target value or not. If it does, we should return its index. If it doesn't, we return its insertion index.



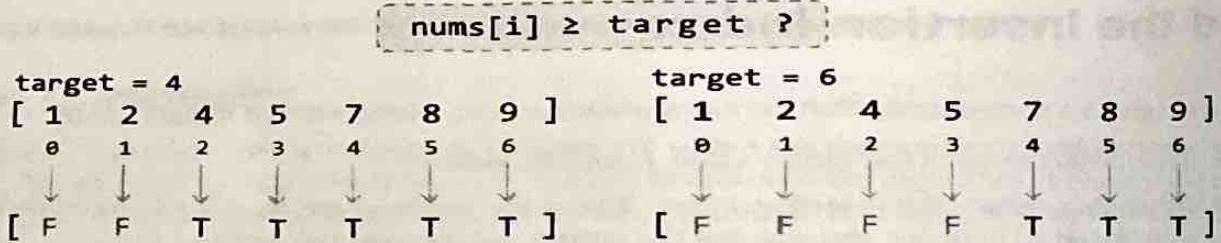
When the target doesn't exist in the array, we observe the insertion index is found at the first value in the array greater than the target, as can be seen in the second diagram above, where the first value larger than the target of 6, is 7.

Since we don't know if the target exists in the array before we start looking for it, we can combine both cases by finding the first value greater than or equal to the target. This gives us a universal objective regardless of whether the target is in the array or not.

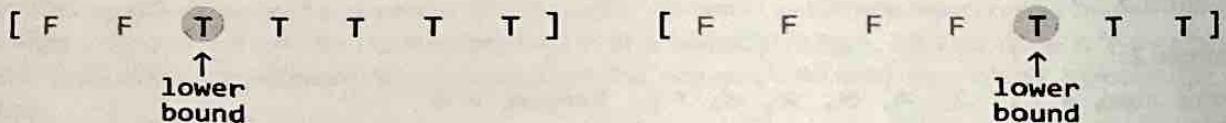
As the array is sorted, we can use **binary search** to find the desired index.

Binary search

In this binary search, we're effectively looking for the first value that matches a **condition**, where the condition is that the number is greater than or equal to the target. Let's visualize which values of the array meet this condition and which don't:



From this diagram, we can see the value we want to find is effectively the **lower bound** of values that satisfy this condition:



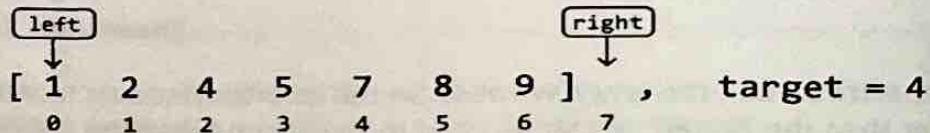
Note that finding the lower bound is equivalent to finding the leftmost value. With this in mind, let's come up with an algorithm.

First, define the search space. If the target exists in the array, it could be located at any index within the range from 0 to $n - 1$. However, if the target is not in the array and is larger than all the elements, its insertion index is n . Therefore, our search space should cover all indexes in the range $[0, n]$.

To figure out how we narrow the search space, let's first explore an example array that contains the target, then dive into an example where the array doesn't contain the target.

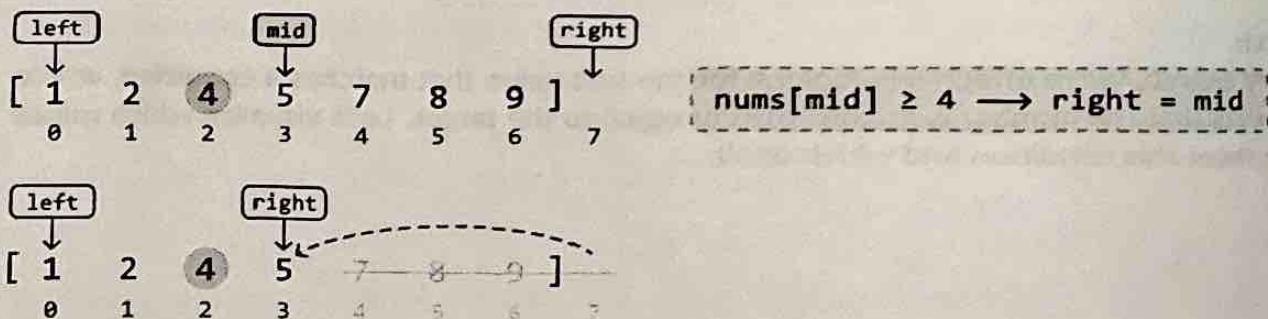
Target exists in the array

Consider searching for target value 4 in the following sorted array.



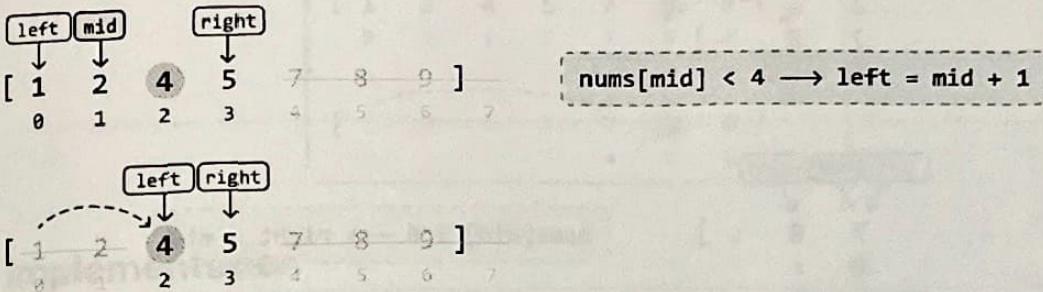
Initially, the midpoint is positioned at element 5, which is a number that satisfies our condition of being greater than or equal to the target. This means the lower bound is either at this midpoint, or to its left since the lower bound is the leftmost value that satisfies the condition.

So, narrow the search space toward the left, while including the midpoint (i.e., $\text{right} = \text{mid}$):

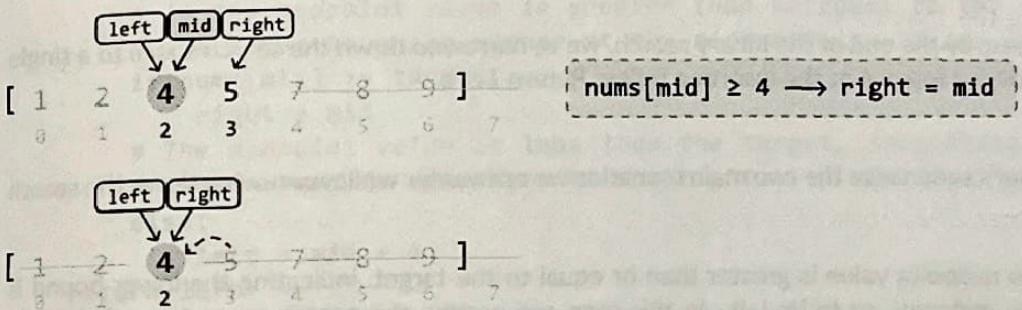


Now, the midpoint is positioned at element 2. The lower bound should be a value greater than or equal to the target (4), which means the current midpoint is too small. To look for a larger value, we need to search to the right of the midpoint.

So, narrow the search space toward the right while excluding the midpoint:



Now, the midpoint is positioned at element 4, which satisfies the condition of being greater than or equal to the target. So, narrow the search space toward the left while including the midpoint:

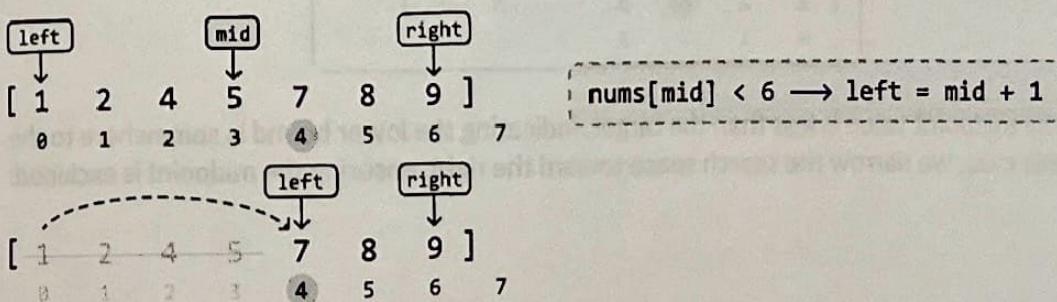


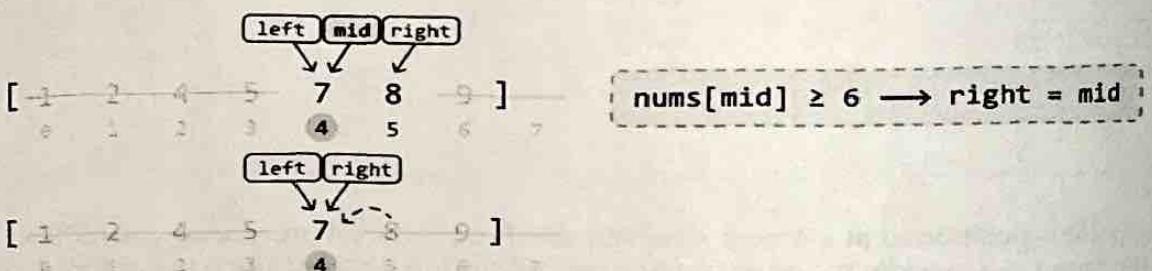
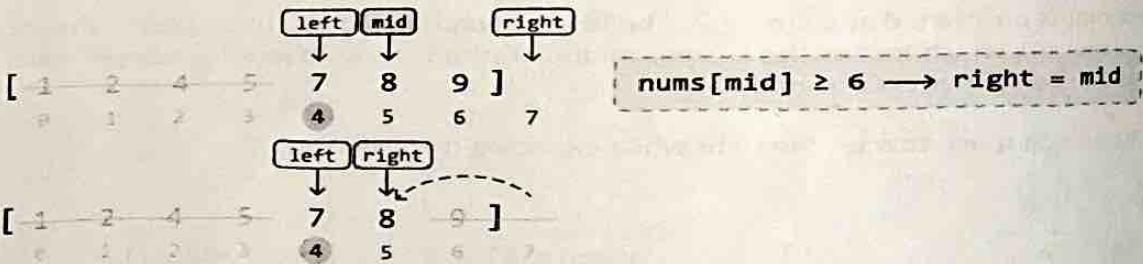
Once the left and right pointers meet, the lower bound is located, which is the first value greater than or equal to the target. Now, we just return `left` to return this value's index.

Target doesn't exist in the array

Let's test the above logic in the following example where the target is not in the array:

[1 2 4 5 7 8 9] , target = 6



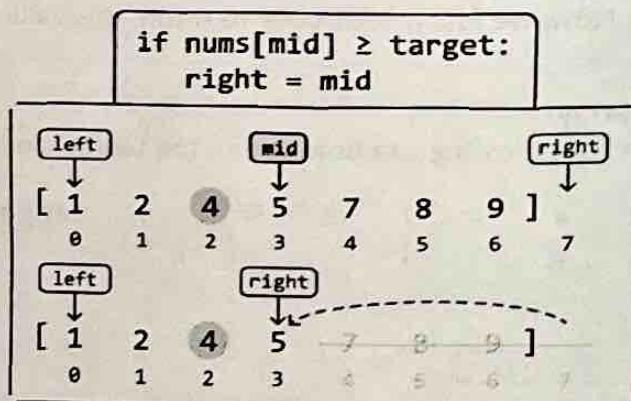


As we can see, by the end of the binary search, we've narrowed down the search space to a single value, identifying index 4 as the insertion index. Return `left`.

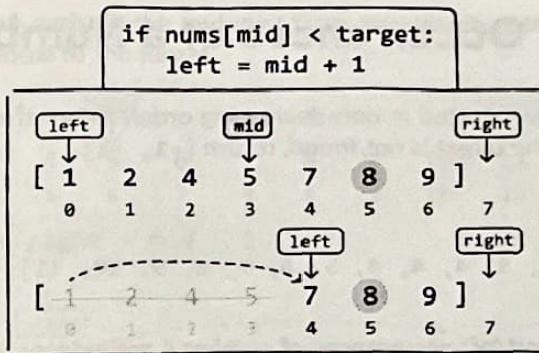
Summary

For clarity, let's summarize the two main scenarios we encounter while narrowing down the search space.

Case 1: The midpoint value is greater than or equal to the target, indicating the lower bound is either at the midpoint, or to its left. In this case, we narrow the search space toward the left, ensuring the midpoint is included:



Case 2: The midpoint value is less than the target, indicating the lower bound is somewhere to the right. In this case, we narrow the search space toward the right, ensuring the midpoint is excluded:



Implementation

```
def find_the_insertion_index(nums: List[int], target: int) -> int:
    left, right = 0, len(nums)
    while left < right:
        mid = (left + right) // 2
        # If the midpoint value is greater than or equal to the target,
        # the lower bound is either at the midpoint, or to its left.
        if nums[mid] >= target:
            right = mid
        # The midpoint value is less than the target, indicating the
        # lower bound is somewhere to the right.
        else:
            left = mid + 1
    return left
```

Complexity Analysis

Time complexity: The time complexity of `find_the_insertion_index` is $O(\log(n))$ because it performs a binary search over a search space of size $n + 1$.

Space complexity: The space complexity is $O(1)$.

First and Last Occurrences of a Number

Given an array of integers sorted in non-decreasing order, return the first and last indexes of a target number. If the target is not found, return `[-1, -1]`.

Example 1:

Input: `nums = [1, 2, 3, 4, 4, 4, 5, 6, 7, 8, 9, 10, 11]`, target = 4
Output: `[3, 5]`

Explanation: The first and last occurrences of number 4 are indexes 3 and 5, respectively.

Intuition

A brute-force solution to this problem involves a linear search to find the first and last occurrences of the target number. However, since the array is sorted, we can try searching for these two occurrences using binary search.

The challenge of using binary search here is that we need to find two separate occurrences of the same number. This means that a standard binary search alone isn't sufficient.

To help us find a solution, it's important to understand that the problem is effectively asking us to find the lower and upper bound of a number in the array:

[1	2	3	4	4	4	5	6	7	8	9	10	11]
	0	1	2	3	4	5	6	7	8	9	10	11	12	
				↓ lower bound		↓ upper bound								

This indicates we can solve this problem in two main steps:

1. Perform a binary search to find the lower bound of the target.
2. Perform a binary search to find the upper bound of the target.

Lower-bound binary search

To find the start position of the target, let's first define the search space. The target could be anywhere in the array, so the search space should encompass all the array's indexes.

Next, let's figure out how to narrow the search space. At each point in the binary search, there are three conditions to consider based on the midpoint value:

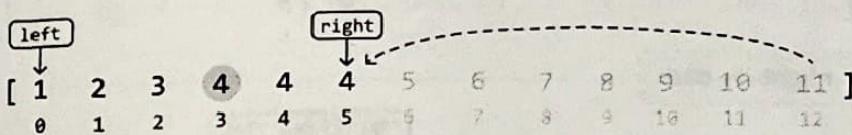
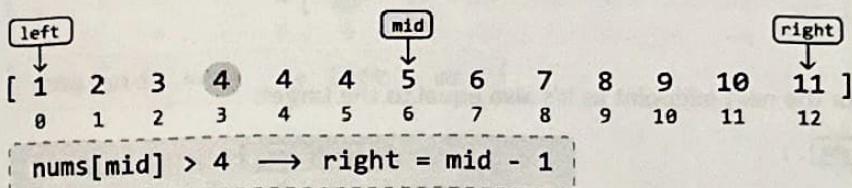
- When it's greater than the target.
- When it's less than the target.
- When it's equal to the target.

In each of these cases, think about where the target is in relation to the midpoint. Note that we're effectively looking for the leftmost occurrence of the target value.

Midpoint value is greater than the target

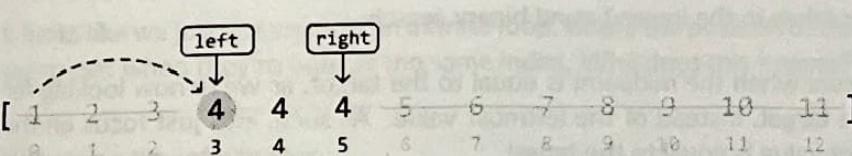
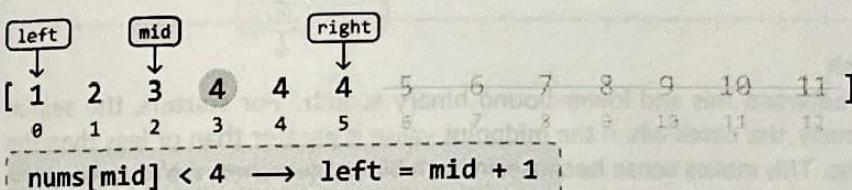
If the midpoint value is greater, it means the target is to the left of this number. So, narrow the search space toward the left.

When we do this, we can exclude the midpoint from the search space (i.e., $\text{right} = \text{mid} - 1$) because its value is not equal to the target:



Midpoint value is less than the target

If the midpoint value is smaller, it means the target is to the right of this number. So, narrow the search space toward the right. Again, we can exclude the midpoint from the search space (i.e., $\text{left} = \text{mid} + 1$) because its value is not equal to the target:

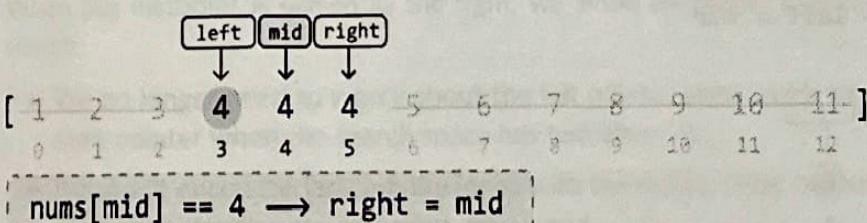


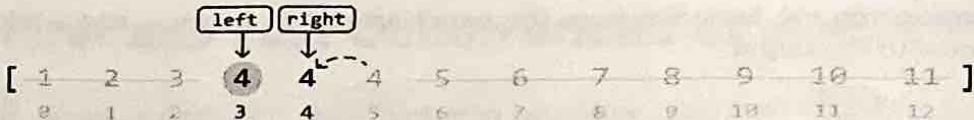
Midpoint value is equal to the target

Now is when things get interesting. When the midpoint value is equal to the target, there are two possibilities:

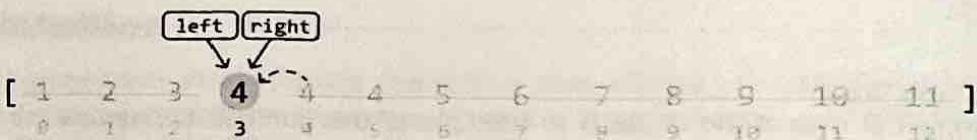
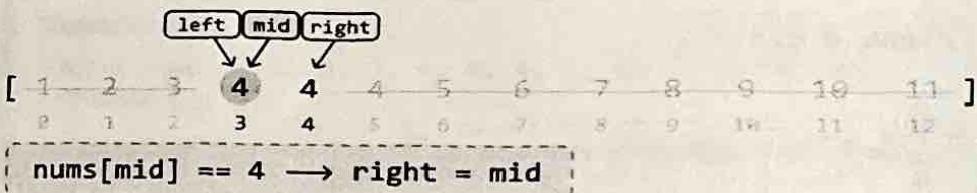
1. This is the lower bound of the target value.
2. This is not the lower bound, so the lower bound is somewhere further to the left.

We don't know which possibility is true at the moment, so we should narrow the search space toward the left to continue looking for the lower bound, while also including the midpoint itself in the search space (i.e., $\text{right} = \text{mid}$).





Continue this reasoning for the next midpoint as it's also equal to the target:



The final value, once the left and right pointers meet, is the lower bound of the target.

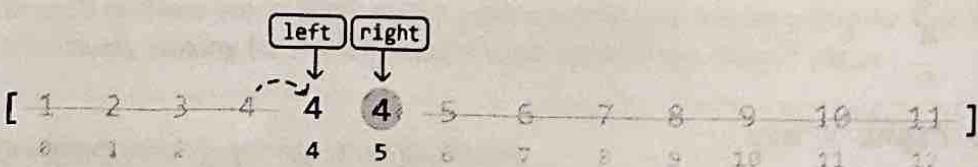
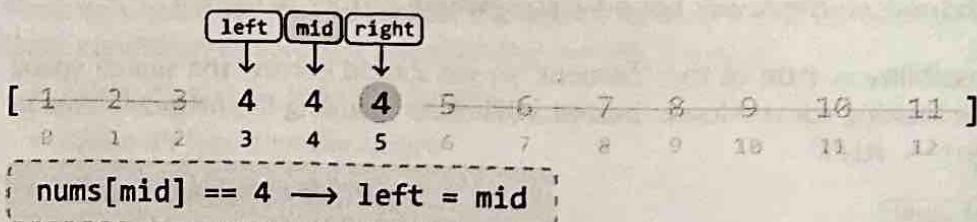
Upper-bound binary search

There's a lot of similarity between this and lower-bound binary search. For starters, the search space is identical. Additionally, the cases when the midpoint value is greater than or less than the target are handled the same. This makes sense because, in both binary searches, we're seeking the same target value. So, when the midpoint value is not equal to the target, the actions we take will be the same as the actions taken in the lower-bound binary search.

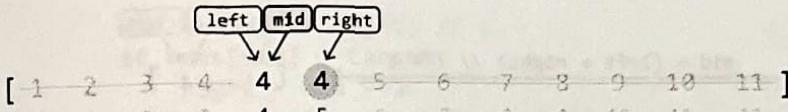
The difference in logic occurs when the midpoint is equal to the target, as we're now looking for the **rightmost value of the target**, instead of the **leftmost value**. As such, let's just focus on the logic for when the midpoint value is equal to the target.

Midpoint value is equal to the target

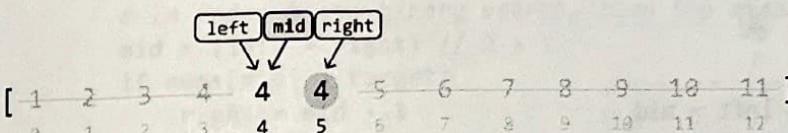
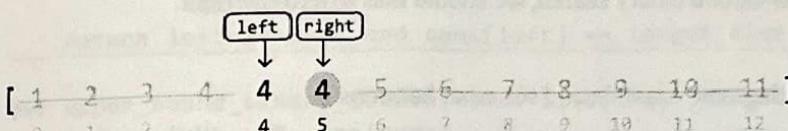
Similar to lower-bound binary search, there are two possibilities: either this is the upper bound, or it's not. Again, we're not sure which is true. So, let's narrow the search space toward the right to continue looking for an upper bound while including the midpoint in the search space (i.e., `left = mid`):



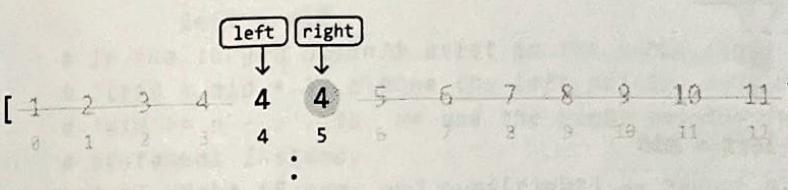
When we continue this logic for the next midpoint values, we notice something peculiar happen:



$\text{nums[mid]} == 4 \rightarrow \text{left} = \text{mid}$



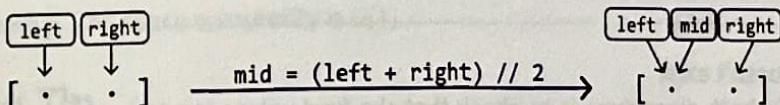
$\text{nums[mid]} == 4 \rightarrow \text{left} = \text{mid}$



It looks like we just got stuck in an infinite loop, where the position of the left pointer keeps getting set to mid when they're both at the same index. Why does this happen?

Debugging the infinite loop

When the left and right pointers are right next to each other, mid ends up being positioned where the left pointer is:

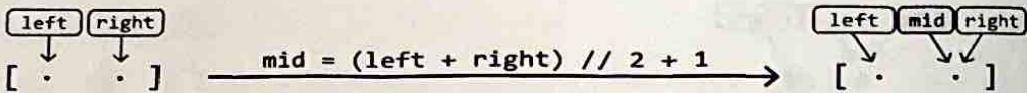


Since one of our operations is $\text{left} = \text{mid}$, we get stuck in an infinite loop where left and mid are continuously set to each other's positions, which means progress cannot be made. The reason this wasn't a problem during lower-bound binary search was that we never had $\text{left} = \text{mid}$ as an operation in the logic. One way to avoid this issue is by **biasing the midpoint to the right**.

When the midpoint is biased to the right, we avoid an infinite loop during upper-bound binary search.

- We no longer need to worry about the left pointer since mid is now being positioned at the right pointer when the search space has two elements.
- We won't encounter any infinite loops with the right pointer because it never gets set to the midpoint's position, as we use $\text{right} = \text{mid} - 1$.

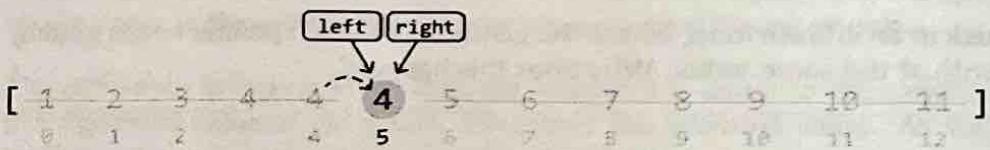
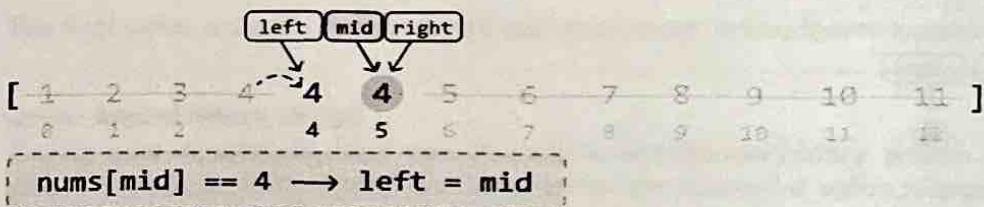
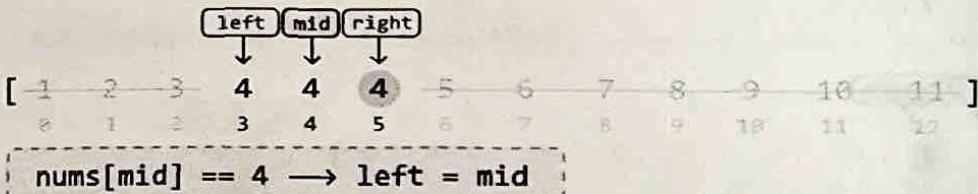
A right bias of the midpoint can be achieved using $\text{mid} = (\text{left} + \text{right}) // 2 + 1$:



Now, performing `left = mid` allows us to properly narrow the search space.

| As a general rule, in upper-bound binary search, we should bias `mid` to the right.

Let's apply this change to the same example and see what happens:



As we can see, we've avoided an infinite loop, with the left and right pointers meeting at the upper bound of the target.

If the target doesn't exist

The last step in both algorithms is to check that the final values located are equal to the target. If the target does not exist in the array, the final values in both binary search algorithms won't be equal to the target. In this case, we should return -1.

Implementation

```
def first_and_last_occurrences_of_a_number(nums: List[int],
                                         target: int) -> List[int]:
    lower_bound = lower_bound_binary_search(nums, target)
    upper_bound = upper_bound_binary_search(nums, target)
    return [lower_bound, upper_bound]

def lower_bound_binary_search(nums: List[int], target: int) -> int:
    left, right = 0, len(nums) - 1
    while left < right:
```

```

        mid = (left + right) // 2
        if nums[mid] > target:
            right = mid - 1
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid
    return left if nums and nums[left] == target else -1

def upper_bound_binary_search(nums: List[int], target: int) -> int:
    left, right = 0, len(nums) - 1
    while left < right:
        # In upper-bound binary search, bias the midpoint to the right.
        mid = (left + right) // 2 + 1
        if nums[mid] > target:
            right = mid - 1
        elif nums[mid] < target:
            left = mid + 1
        else:
            left = mid
    # If the target doesn't exist in the array, then it's possible that
    # 'left = mid + 1' places the left pointer outside the array when
    # 'mid == n - 1'. So, we use the right pointer in the return
    # statement instead.
    return right if nums and nums[right] == target else -1

```

Complexity Analysis

Time complexity: The time complexity of both the `lower_bound_binary_search` and `upper_bound_binary_search` helper functions is $O(\log(n))$, where n denotes the length of the input array. This is because each function performs a binary search over the entire array. Therefore, `first_and_last_occurrences_of_a_number` is also $O(\log(n))$ because it calls each helper function once.

Space complexity: The space complexity is $O(1)$.

Interview Tip



Tip: Always test your algorithm.

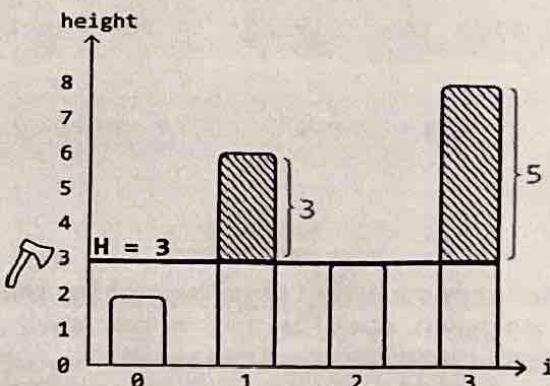
Binary search can be a tricky algorithm to implement. The best way to uncover unexpected errors is to test your code. The infinite loop encountered during the upper-bound binary search problem is quite easy to miss while designing the algorithm. If you're unable to recognize this issue during implementation, manual testing can help reveal it. In binary search, an infinite loop can be uncovered when testing a search space that contains just two elements, similar to what we did in the explanation.

Cutting Wood

You are given an array representing the heights of trees, and an integer k representing the total length of wood that needs to be cut.

For this task, a woodcutting machine is set to a certain height, H . The machine cuts off the top part of all trees taller than H , while trees shorter than H remain untouched. Determine the highest possible setting of the woodcutter (H) so that it cuts at least k meters of wood. Assume the woodcutter cannot be set higher than the height of the tallest tree in the array.

Example:



Input: heights = [2, 6, 3, 8], $k = 7$

Output: 3

Explanation: The highest possible height setting that yields at least $k = 7$ meters of wood is 3, which yields 8 meters of wood. Any height setting higher than this will yield less than 7 meters of wood.

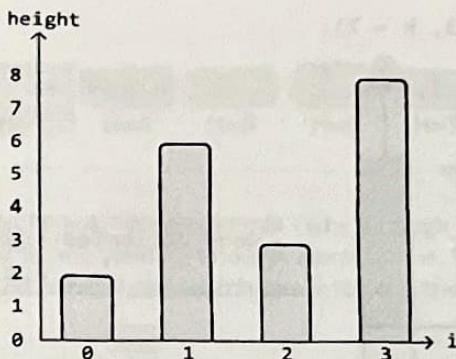
Constraints:

- It's always possible to attain at least k meters of wood.
- There's at least one tree.

Intuition

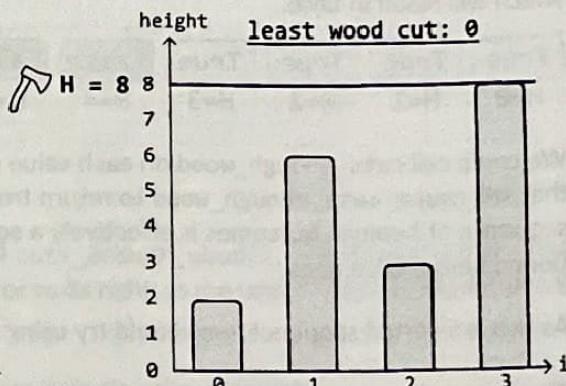
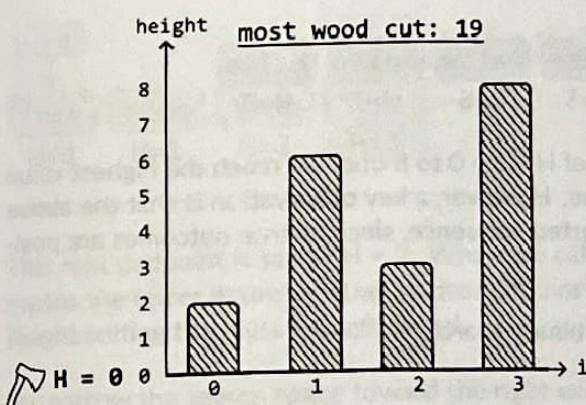
At first, it might strike you as strange that this problem is in the Binary Search chapter. The given input array isn't necessarily sorted, so how is binary search applicable here? Well, this is an example of a common application of binary search, where the search space does not encompass the input array.

Let's consider a visualization of four trees of heights [2, 6, 3, 8] and assume $k = 7$:



The tallest tree above has a height of 8. So the height setting, H , can be set to any height between 0 and 8.

- The most amount of wood is cut at $H = 0$, where all trees are cut from the bottom.
- The least amount of wood is cut at $H = 8$, where no wood is cut at all.



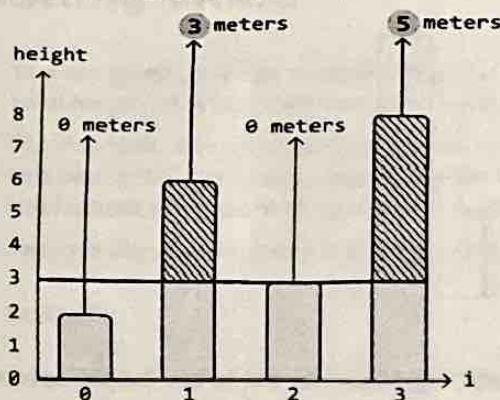
Gradually increasing the height setting of the woodcutter from $H = 0$ to $H = 8$ yields less and less wood, and our goal is to find the highest value of H that gives us at least k meters of wood.

Determining if a height setting yields enough wood

We need a function that determines if any given height setting H yields at least k meters of wood.

Let's name this function `cuts_enough_wood(H, k)`, which will calculate the total wood obtained by cutting the trees taller than H , and return true if this total meets or exceeds k . Below is a visual representation of how to determine if a height setting of 3 yields enough wood in the example:

`cuts_enough_wood(H = 3, k = 7):`



$$\begin{aligned} \text{wood_collected} &= 3 + 5 \\ &= 8 \geq k \longrightarrow \text{return True} \end{aligned}$$

Applying this function to all possible values of H (from 0 to 8) gives us the outcome below, where heights 0 to 3 yield at least k meters of wood and heights 4 to 8 are too high and don't yield enough. Note that here, we visualize which H values make the function `cuts_enough_wood` return true, and which will result in false.

True	True	True	True	False	False	False	False	False
$H=0$	$H=1$	$H=2$	$H=3$	$H=4$	$H=5$	$H=6$	$H=7$	$H=8$

We could call `cuts_enough_wood` on each value of H from 0 to 8 until we reach the highest value that still causes `cuts_enough_wood` to return true. However, a key observation is that the above sequence of boolean outcomes is effectively a **sorted sequence**, since all true outcomes are positioned before false ones.

As this is a sorted sequence, we should try using binary search.

Binary search

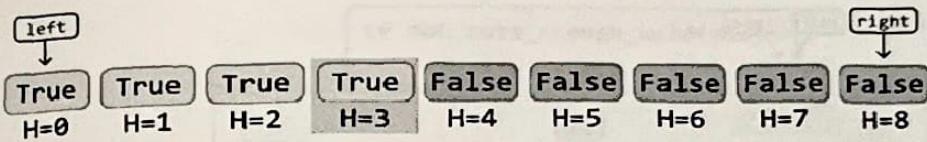
The goal is to find the last value of H that `cuts_enough_wood` returns true. In other words, we're looking for the upper bound value of H that satisfies this condition.

True	True	True	True	False	False	False	False	False
$H=0$	$H=1$	$H=2$	$H=3$	$H=4$	$H=5$	$H=6$	$H=7$	$H=8$
↑ upper bound								

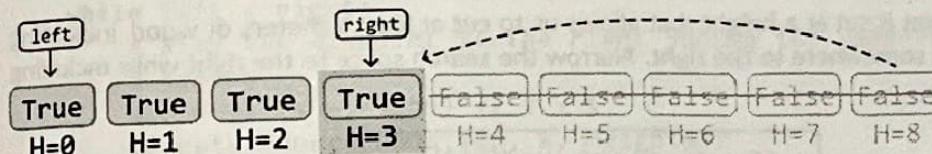
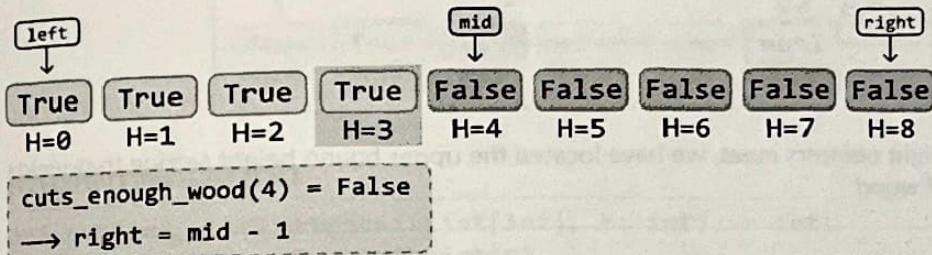
As such, we should use **upper-bound binary search**. This means we'll need to calculate the mid-point using `mid = (left + right) // 2 + 1`, as mentioned in the *First and Last Occurrences of a Number* problem.

Let's first define the search space. Our search space should encompass all values of H between 0 and the height of the tallest tree in the array, as these are all possible answers.

To figure out how to narrow the search space, let's use the example below, setting left and right pointers at the ends of the search space:

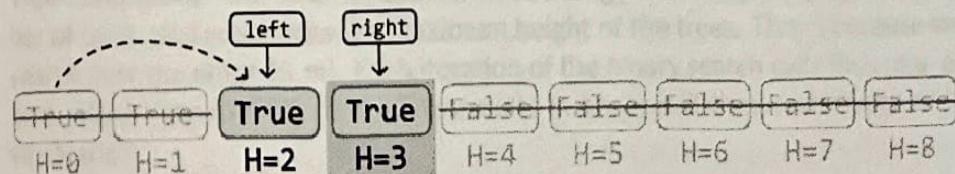
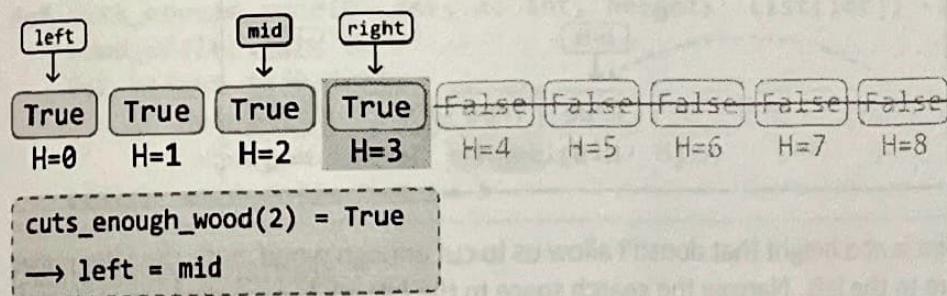


Initially, the midpoint is set to $H = 4$. When we call `cuts_enough_wood(4, k)`, it returns false. This means the height setting is not yielding enough wood and is, hence, set too high. To find a lower height setting, we should narrow our search space toward the left:

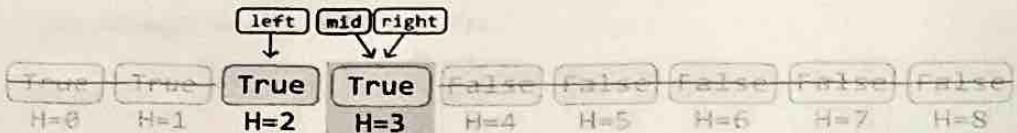


The next midpoint is set to $H = 2$. When we call `cuts_enough_wood(2, k)`, it returns true. This means the upper bound is either at the midpoint or to its right, as the upper bound is the rightmost height setting that cuts enough wood.

So, narrow the search space toward the right while including the midpoint:

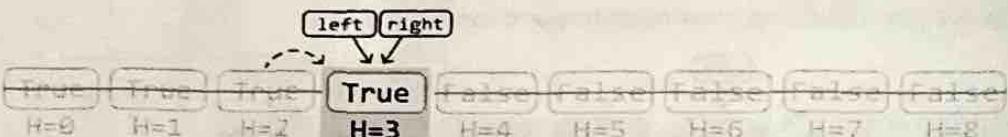


The next midpoint is set to $H = 3$. When we call `cuts_enough_wood(3, k)`, it returns true. So, narrow the search space toward the right while including the midpoint:



`cuts_enough_wood(3) = True`

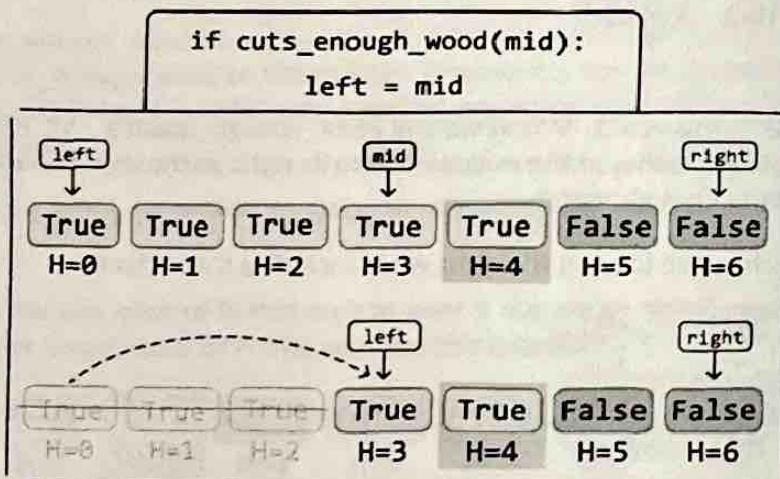
→ `left = mid`



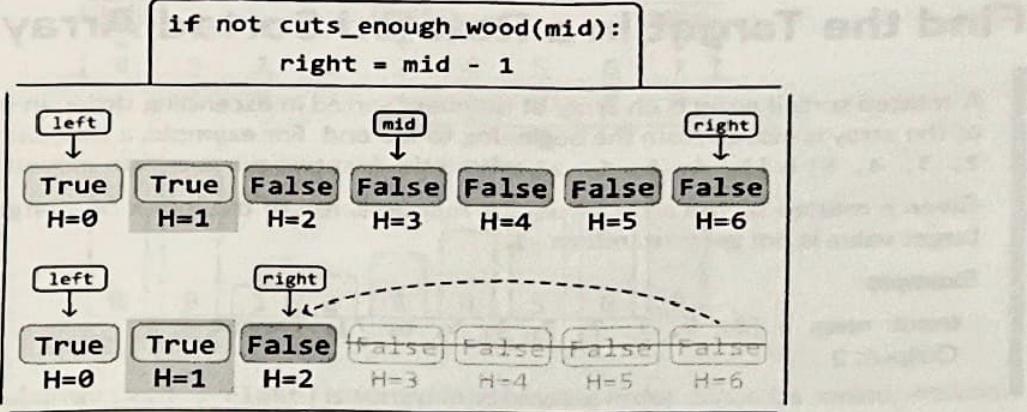
Once the left and right pointers meet, we have located the upper bound height setting that yields at least k meters of wood.

Summary

Case 1: The midpoint is set at a height that allows us to cut at least k meters of wood, indicating the upper bound is somewhere to the right. Narrow the search space to the right while including the midpoint:



Case 2: The midpoint is at a height that doesn't allow us to cut enough wood, indicating the upper bound is somewhere to the left. Narrow the search space to the left while excluding the midpoint:



Implementation

```

def cutting_wood(heights: List[int], k: int) -> int:
    left, right = 0, max(heights)
    while left < right:
        # Bias the midpoint to the right during the upper-bound binary
        # search.
        mid = (left + right) // 2 + 1
        if cuts_enough_wood(mid, k, heights):
            left = mid
        else:
            right = mid - 1
    return right

# Determine if the current value of 'H' cuts at least 'k' meters of
# wood.
def cuts_enough_wood(H: int, k: int, heights: List[int]) -> bool:
    wood_collected = 0
    for height in heights:
        if height > H:
            wood_collected += (height - H)
    return wood_collected >= k

```

Complexity Analysis

Time complexity: The time complexity of `cutting_wood` is $O(n \log(m))$, where n denotes the number of trees, and m denotes the maximum height of the trees. This is because we perform a binary search over the range $[0, m]$. Each iteration of the binary search calls the `cuts_enough_wood` function, which runs in $O(n)$ time. This results in an overall time complexity of $O(\log(m)) \cdot O(n) = O(n \log(m))$.

Space complexity: The space complexity is $O(1)$.

Find the Target in a Rotated Sorted Array

A rotated sorted array is an array of numbers sorted in ascending order, in which a portion of the array is moved from the beginning to the end. For example, a possible rotation of [1, 2, 3, 4, 5] is [3, 4, 5, 1, 2], where the first two numbers are moved to the end.

Given a rotated sorted array of unique numbers, return the index of a target value. If the target value is not present, return -1.

Example:

Input: `nums = [8, 9, 1, 2, 3, 4, 5, 6, 7]`, `target = 1`

Output: 2

Intuition

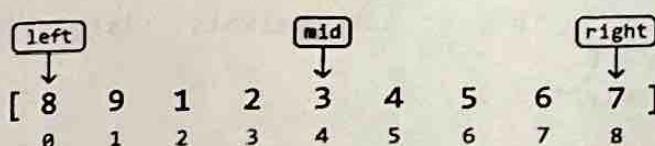
A naive solution is to iterate through the input array until we find the target number. This strategy takes linear time, and doesn't take into account that the input is a rotated sorted array. Given the array was sorted before it was rotated, we should figure out how binary search might be useful in finding the target.

First, let's define the search space for the binary search. Since the target value could exist anywhere in the array, the search space should encompass the entire array.

Now, let's figure out how to narrow the search space, which is an interesting challenge considering the array isn't perfectly sorted. Let's work through this by exploring an example:

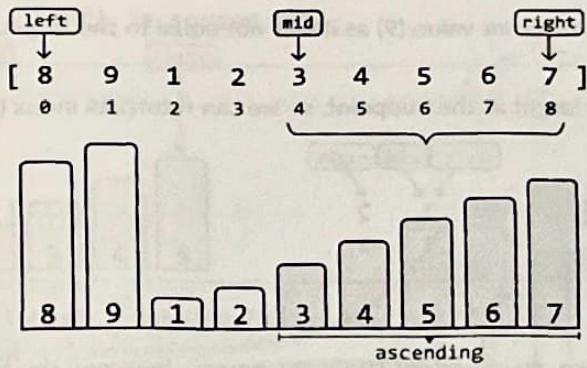
[8 9 1 2 3 4 5 6 7], target = 1
 0 1 2 3 4 5 6 7 8

Let's set left and right pointers at the boundaries of the array and consider the first midpoint value:



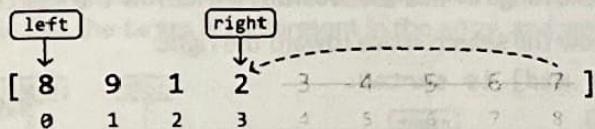
In a normal sorted array, we'd be able to logically assess whether to search to the left or the right of the midpoint, based solely on the midpoint value and the target. In a rotated sorted array, it's much less straightforward to determine which side the target value is on.

To decide whether to narrow the search space toward the left or right of the midpoint, let's visualize the height of each number in the array and pay attention to subarrays [left : mid] and [mid : right] separately. Note, we include the midpoint in both subarrays since the midpoint is used to decide which subarray to narrow the search space towards:



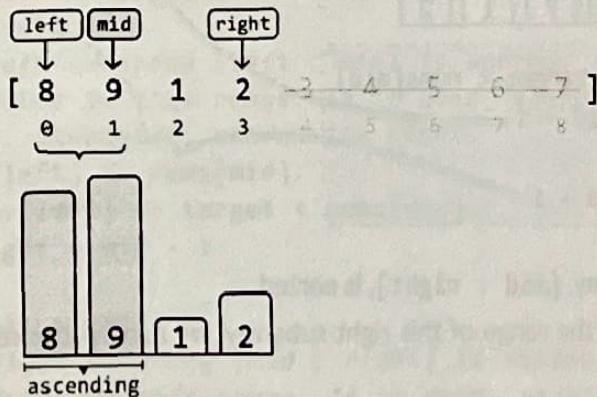
Here, we see the subarray `[mid : right]` is sorted in ascending order. Since it's sorted, we know what the smallest and largest numbers in that range are: 3 and 7, respectively. This means we can check if the target is in this subarray by checking if it's in between 3 and 7. The target (1) is not in this range, so therefore, it must be in the left subarray.

So, we should narrow the search space toward the left, excluding the midpoint (`right = mid - 1`):

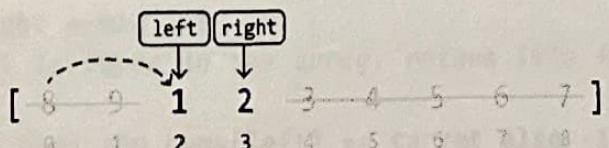


The reason we excluded the midpoint value was because it wasn't equal to the target, and so should no longer be considered in the search space.

Let's continue with the example.

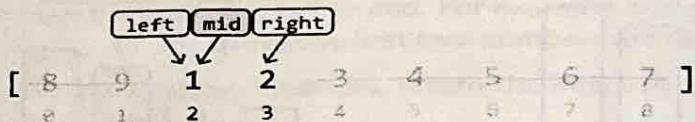


This time, the sorted subarray is the left subarray, `[left : mid]`. So, we can use this subarray to check where the target value is. Since the target (1) doesn't fall within the range between 8 and 9, it indicates the target resides in the right subarray. Therefore, we narrow our search space toward the right, excluding the midpoint (`left = mid + 1`):



Again, we excluded the midpoint value (9) as it was not equal to the target.

Finally, we've found the target at the midpoint, so we can return its index (mid):



Summary

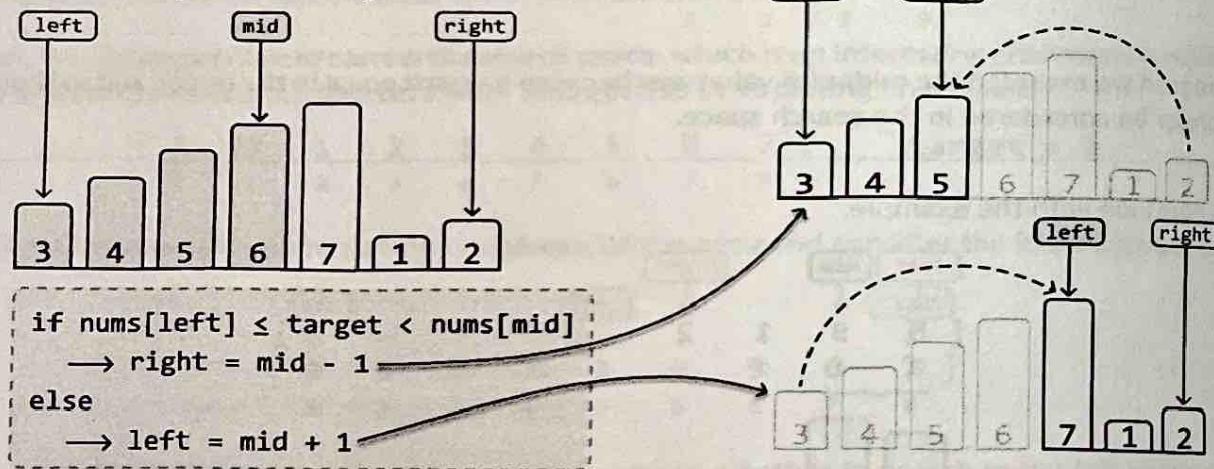
From our discussion above, an important strategy emerges: between the two subarrays, [left : mid] and [mid : right], we can use the sorted one to determine where the target is.

Before examining the subarrays, let's first compare the target with the value at the midpoint. If they match, we've found our target at mid. If not, then we decide where to adjust our search depending on which subarray is sorted, as detailed in the following two test cases:

Case 1: the left subarray, [left : mid], is sorted

- If the target falls in the range of this left subarray, we narrow the search space toward the left.
- Otherwise, we narrow the search space toward the right.

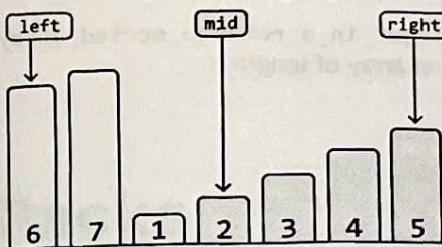
Case 1 - [left : mid] is sorted:



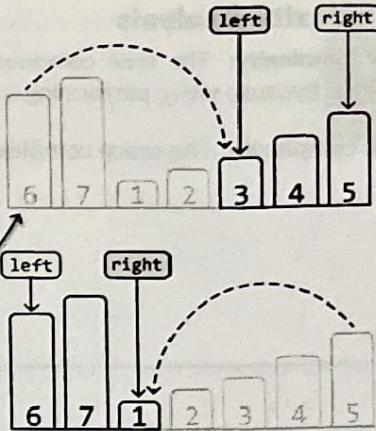
Case 2: the right subarray, [mid : right], is sorted

- If the target falls in the range of this right subarray, we narrow the search space toward the right.
- Otherwise, narrow the search space toward the left.

Case 2 - [mid : right] is sorted:



```
if nums[mid] < target ≤ nums[right]
    → left = mid + 1
else
    → right = mid - 1
```



It's possible to encounter a situation where both subarrays are sorted. In this case, it doesn't matter which subarray we use to check where the target is.

One final thing to note is that the array might not contain the target value at all. In this case, once the binary search terminates and narrows down to a single value, we need to check if this value is the target. If not, it indicates the target is not present in the array, and we return -1.

Implementation

```
def find_the_target_in_a_rotated_sorted_array(nums: List[int],
                                              target: int) -> int:
    left, right = 0, len(nums) - 1
    while left < right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        # If the left subarray [left : mid] is sorted, check if the
        # target falls in this range. If it does, search the left
        # subarray. Otherwise, search the right.
        elif nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        # If the right subarray [mid : right] is sorted, check if the
        # target falls in this range. If it does, search the right
        # subarray. Otherwise, search the left.
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1
    # If the target is found in the array, return its index. Otherwise,
    # return -1.
    return left if nums[left] == target else -1
```

Complexity Analysis

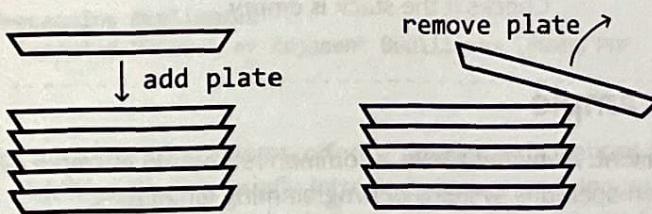
Time complexity: The time complexity of `find_the_target_in_a_rotated_sorted_array` is $O(\log(n))$ because we're performing a binary search over an array of length n .

Space complexity: The space complexity is $O(1)$.

Stacks

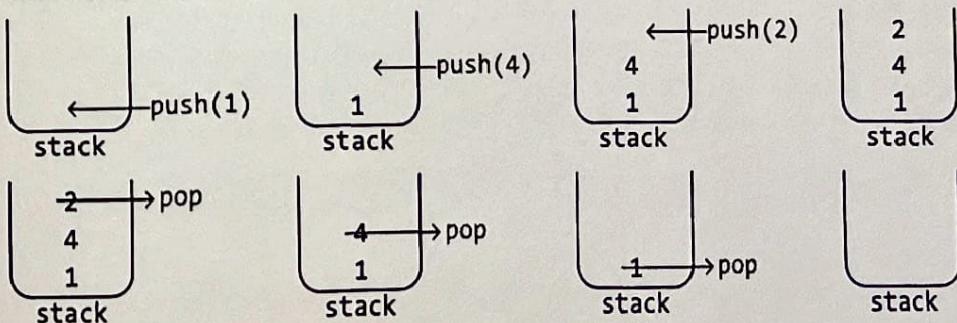
Introduction to Stacks

Imagine a stack of plates. You can only add a new plate to the top of the stack, and when you need a plate, you take the one from the top. It's not possible to take a plate from the bottom or middle without first removing all the plates above it.



This analogy encapsulates the essence of the stack data structure. Adding a plate to and taking a plate from the top of the stack, physically demonstrates the two main stack operations:

- Push (adds an element to the top of the stack).
- Pop (removes and returns the element at the top of the stack).



LIFO (Last-In-First-Out)

Stacks follow the LIFO principle, meaning the most recently added item is the first to be removed. This unique characteristic makes stacks particularly useful in various scenarios where the order of processing or removal is critical. Here are a few key applications:

- Handling nested structures: Stacks are a good option for parsing or validating nested structures such as nested parentheses in a string (e.g., "((())())"). They allow us to process the innermost nested structures first due to the LIFO principle.
- Reverse order: When elements are added (pushed) onto a stack and then removed (popped),

they come out in the reverse order of how they were added. This property is useful for reversing sequences.

- Substitute for recursion: Recursive algorithms use the recursive call stack to manage recursive calls. Ultimately, this recursive call stack is itself a stack. As such, we can often implement recursive functions iteratively using the stack data structure.
- Monotonic stacks: These special-purpose stacks maintain elements in a consistent, increasing or decreasing sorted order. Before adding a new element to the stack, any elements that break this order are removed from the top of the stack, ensuring the stack remains sorted.

Some examples of the above applications are explored in this chapter.

Below is a time complexity breakdown of common stack operations:

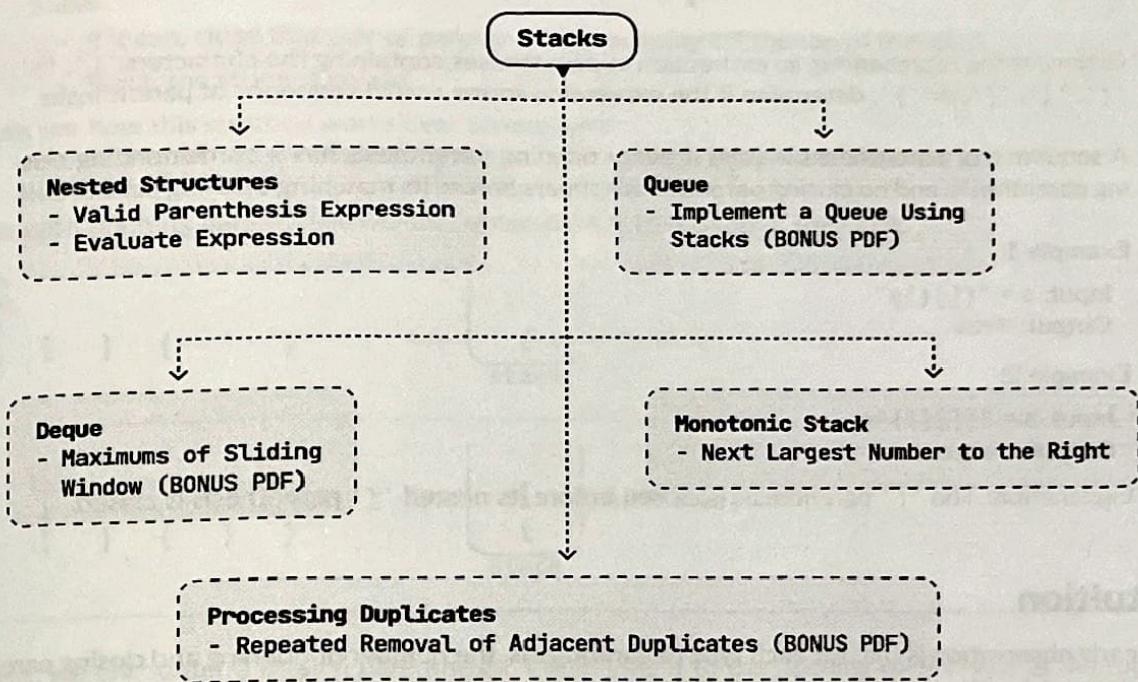
Operation	Worst case	Description
Push	$O(1)$	Adds an element to the top of the stack.
Pop	$O(1)$	Removes and returns the element at the top of the stack.
Peek	$O(1)$	Returns the element at the top of the stack without removing it.
IsEmpty	$O(1)$	Checks if the stack is empty.

Real-world Example

Function call management: As hinted above, a common real-world example of stacks is in function call management within operating systems or programming languages.

When a function is called, the program pushes the function's state (including its parameters, local variables, and the return address) onto the call stack. As functions call other functions, their states are also pushed onto the stack. When a function completes, its state is popped off the stack, and the program returns to the calling function. This stack-based approach ensures that functions return control in the correct order, managing nested or recursive function calls efficiently.

Chapter Outline



This chapter explores a variety of problems, offering detailed explanations for how to use stacks in problem solving. Additionally, we briefly introduce queues and deques, which are two data structures that share similarities with stacks, but operate on different principles.

The first problem is validating parentheses, which we put back to the top of the stack when we encounter an opening parenthesis and remove it when we see a closing parenthesis. If we end up with zero elements left in the stack, then the parentheses are valid. If there are still elements left in the stack, then the parentheses are invalid.

The second problem is finding the maximum element in a sliding window. We can use a stack to keep track of the maximum element in each window. As we move the window, we can either remove the previous maximum if it's not in the current window or add the new maximum if it is. This way, we can efficiently find the maximum element in each window.

The third problem is implementing a queue using stacks. We can use two stacks, one for enqueueing and one for dequeuing. To enqueue, we simply push the element onto the top of the enqueue stack. To dequeue, we pop all elements from the enqueue stack and push them onto the top of the dequeue stack. Then, we can simply pop the top element from the dequeue stack.

Valid Parenthesis Expression

Given a string representing an expression of parentheses containing the characters '(', ')', '[', ']', '{', or '}', determine if the expression forms a valid sequence of parentheses.

A sequence of parentheses is valid if every opening parenthesis has a corresponding closing parenthesis, and no closing parenthesis appears before its matching opening parenthesis.

Example 1:

Input: s = "([{}])"
Output: True

Example 2:

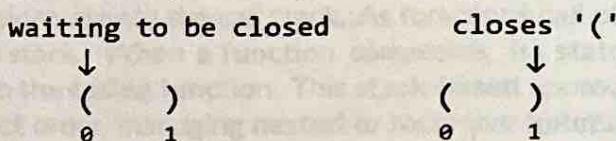
Input: s = "([{}])"
Output: False

Explanation: The '(' parenthesis is closed before its nested '{' parenthesis is closed.

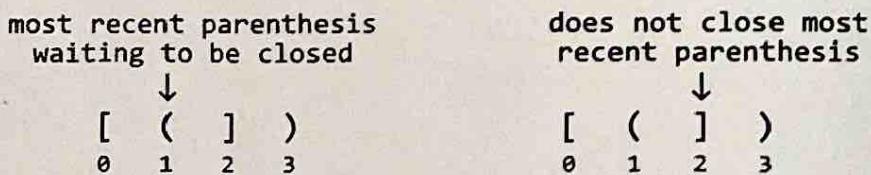
Intuition

An early observation is that for each type of parenthesis, the number of opening and closing parenthesis must be identical. However, to check if an expression is valid, this observation alone isn't enough. For example, the string "()()" has the same number of opening and closing parentheses, but is still invalid. This means we need a way to account for the order of parentheses.

Consider the string "()". The first parenthesis is opening, and we're waiting for it to be closing. Upon reaching the second parenthesis, the first parenthesis gets closed.



Now, consider the string "[())". When we reach index 1, we have two opening parentheses waiting to be closed. In particular, we expect ')' to be closed before '['. The first closing parenthesis we encounter is ')', which does not close ')'. Therefore, this string is invalid.



The key observation here is that the **most recent opening parenthesis we encounter** should be the **first parenthesis that gets closed**. So, opening parentheses are processed from most recent to least recent, which is indicative of a last-in-first-out (LIFO) dynamic. This leads to the idea that a stack can be used to solve this problem.

Stack

Here's a high-level strategy:

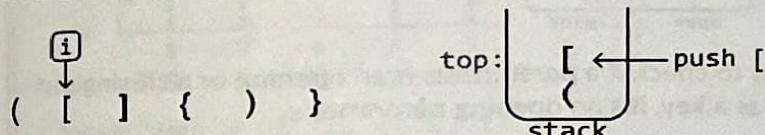
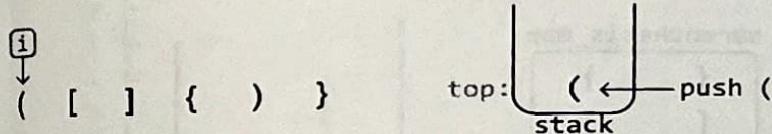
- Add each opening parenthesis we encounter to the stack. This way, the **most recent parenthesis** is always at the top of the stack.

- When encountering a closing parenthesis, check if it can close the most recent opening parenthesis.
 - If it can, close that pair of parenthesis by popping off the top of the stack.
 - If not, the string is invalid.

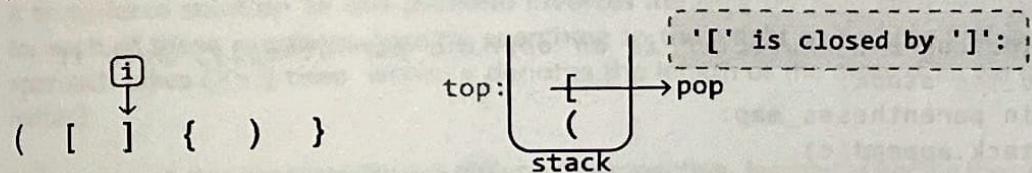
Let's see how this strategy works over an example:

([] {) }

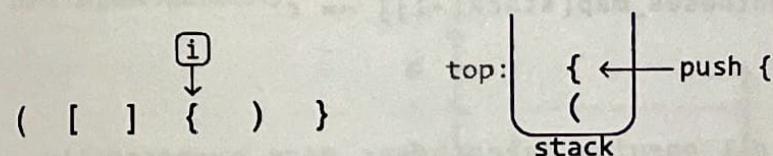
For each opening parenthesis we encounter, push it to the top of the stack:



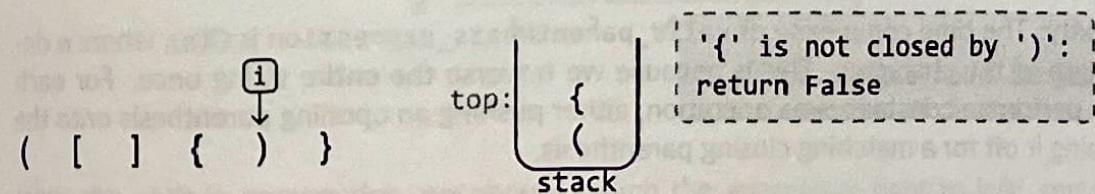
Next, we encounter a closing parenthesis. Comparing it to the opening parenthesis at the top of the stack, we see that it correctly closes that opening parenthesis. So, we can pop off the opening parenthesis at the top of the stack:



The next character is an opening parenthesis, which we just push to the top of the stack:



The next character is a closing parenthesis, ')', which does not close the opening parenthesis at the top of the stack, '{'. This means this parenthesis expression is invalid. As such, we return false.



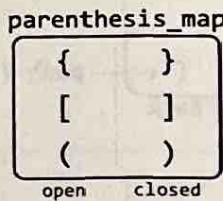
If we've iterated over the entire string without returning false, that means we've accounted for all closing parentheses in the string.

Edge case: extra opening parentheses

We only check for invalidity at closing parenthesis, so we need to perform a final check to ensure there aren't any opening parentheses in the string left unclosed. This can be done by checking if the stack is empty after processing the whole input string, as a non-empty stack indicates opening parentheses remain in the stack.

Managing three types of parentheses

In our algorithm, we need a way to ensure we compare the correct types of opening and closing parentheses. We can use a hash map for this, which maps each type of opening parenthesis to its corresponding closing parenthesis:



This hash map can also be used as a way to check if a parenthesis is an opening or a closing one: if the parenthesis exists in this hash map as a key, it's an opening parenthesis.

Implementation

```
def valid_parenthesis_expression(s: str) -> bool:
    parentheses_map = {')': ')', '}': '}', ']': ']'}
    stack = []
    for c in s:
        # If the current character is an opening parenthesis, push it
        # onto the stack.
        if c in parentheses_map:
            stack.append(c)
        # If the current character is a closing parenthesis, check if
        # it closes the opening parenthesis at the top of the stack.
        else:
            if stack and parentheses_map[stack[-1]] == c:
                stack.pop()
            else:
                return False
    # If the stack is empty, all opening parentheses were successfully
    # closed.
    return not stack
```

Complexity Analysis

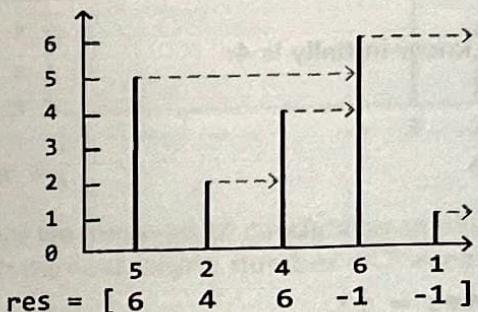
Time complexity: The time complexity of `valid_parenthesis_expression` is $O(n)$, where n denotes the length of the character. This is because we traverse the entire string once. For each character, we perform a constant-time operation, either pushing an opening parenthesis onto the stack or popping it off for a matching closing parenthesis.

Space complexity: The space complexity is $O(n)$ because the stack stores at most n characters, and the hash map takes up $O(1)$ space.

Next Largest Number to the Right

Given an integer array `nums`, return an output array `res` where, for each value `nums[i]`, `res[i]` is the first number to the right that's larger than `nums[i]`. If no larger number exists to the right of `nums[i]`, set `res[i]` to -1.

Example:

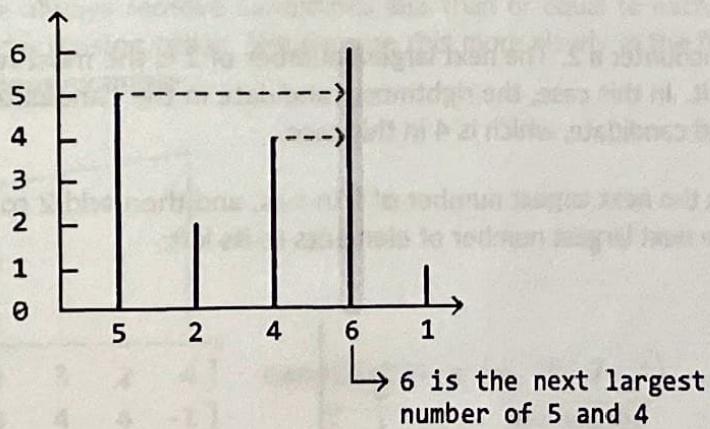


Input: `nums` = [5, 2, 4, 6, 1]
Output: [6, 4, 6, -1, -1]

Intuition

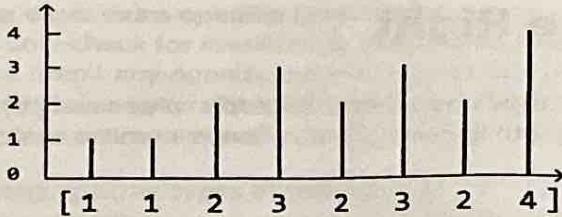
A brute-force solution to this problem involves iterating through each number in the array and, for each of these numbers, linearly searching to their right to find the first larger number. This approach takes $O(n^2)$ time, where n denotes the length of the array. Can we think of something better?

Let's approach this problem from a different perspective. Instead of finding the next largest number for each value, what if we check whether the value itself is the next largest number for any value(s) to its left? For example, can we figure out which values in the following example have 6 as their next largest number?:

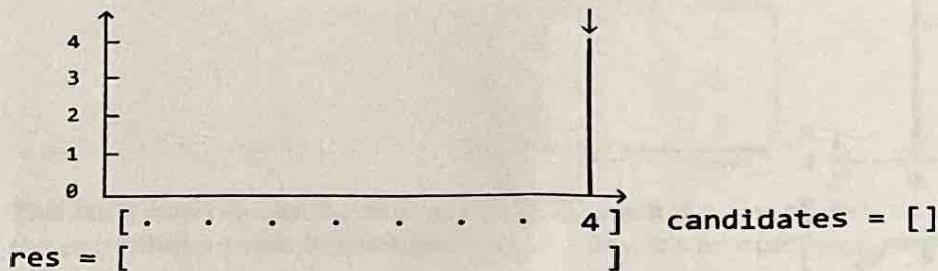


With this shift in perspective, we should search the array from right to left: certain values we encounter from the right could potentially be the next largest number of values to their left. Let's call these values "candidates." But how do we determine which numbers qualify as candidates?

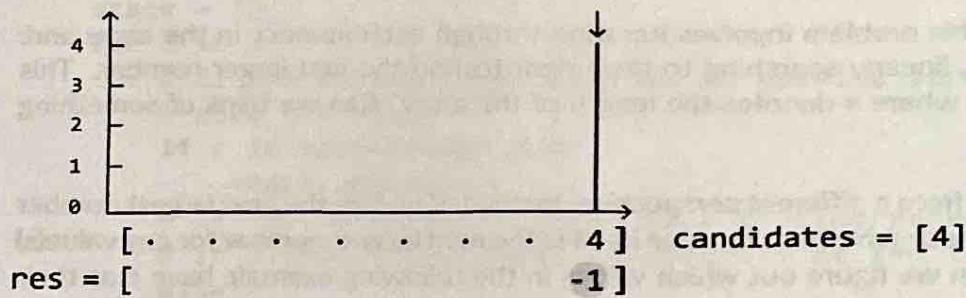
Consider the example below:



Let's start at the rightmost index, where the only value we know initially is 4:

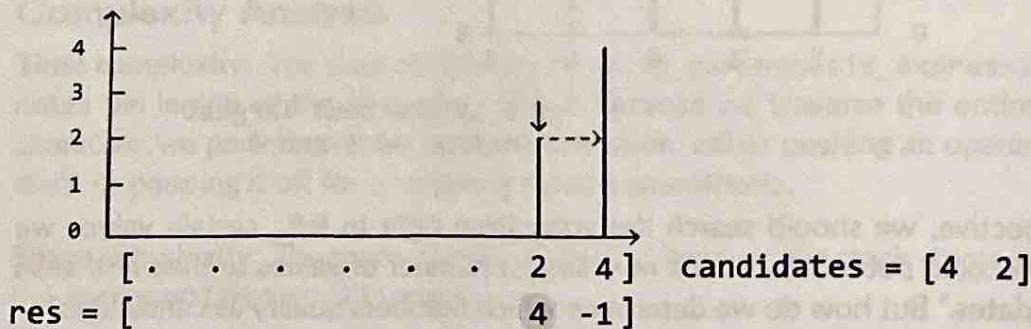


Right now, we can say 4 is a candidate as it might be the next largest number of values to its left. No candidates have been encountered before 4 because it's the rightmost element, so we should mark the result for 4 in `res` as -1:

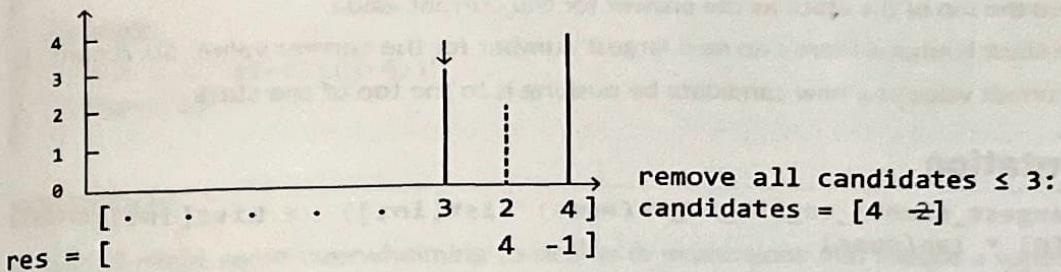


Next, we encounter a 2. The next largest number of 2 is the most recently added candidate that's larger than it. In this case, the rightmost candidate in the candidates list represents the most recently added candidate, which is 4 in this case.

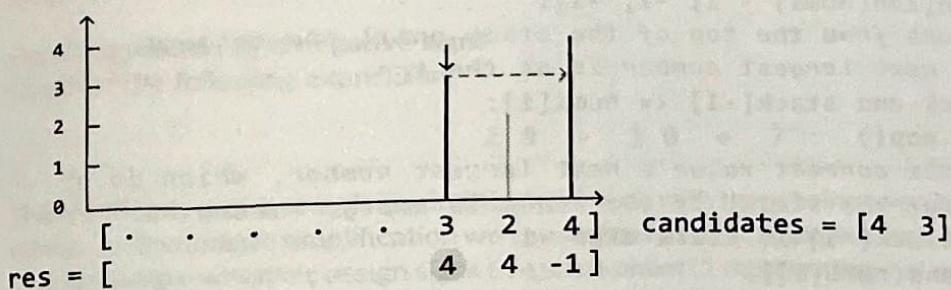
Record 4 as the next largest number of 2 in `res`, and then add 2 to the candidates list because it could be the next largest number of elements to its left:



The next number is 3. Notice that with the introduction of 3, number 2 should no longer be considered a candidate. This is because it's now impossible for 2 to be the next largest number of any value to its left. Since 3 is both larger and further to the left in the array, it will always be prioritized over 2 as the next largest number. So, let's remove 2 from the candidates list, as well as any other candidate that's less than or equal to 3:



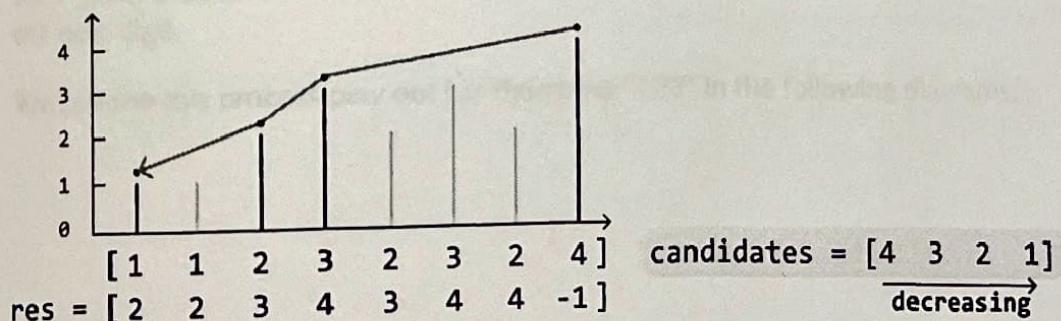
Since we removed all candidates smaller than 3, the rightmost candidate is now 4. So, let's record 4 as the next largest number of 3 in res and add 3 to the candidates list:



This provides a crucial insight:

Whenever we move to a new number, all candidates less than or equal to this number should be removed from the candidates list.

Another key observation is that the list of candidates always maintains a strictly decreasing order of values. This is because we always remove candidates less than or equal to each new value, ensuring values are added in decreasing order. We can see this more clearly in the final state of the candidates list of the previous example:



This indicates a stack is the ideal data structure for storing the candidates list, since stacks can be used to efficiently maintain a monotonic decreasing order of values, as mentioned in the introduction.

The top of the stack represents the most recent candidate to the right of each new number encountered. Given this, here's how to use the stack to add and remove candidates at each value:

1. Pop off all candidates from the top of the stack less than or equal to the current value.
2. The top of the stack will then represent the next largest number of the current value.
 - Record the top of the stack as the answer for the current value.
 - If the stack is empty, there's no next largest number for the current value. So, record -1.
3. Add the current value as a new candidate by pushing it to the top of the stack.

Implementation

```
def next_largest_number_to_the_right(nums: List[int]) -> List[int]:  
    res = [0] * len(nums)  
    stack = []  
    # Find the next largest number of each element, starting with the  
    # rightmost element.  
    for i in range(len(nums) - 1, -1, -1):  
        # Pop values from the top of the stack until the current  
        # value's next largest number is at the top.  
        while stack and stack[-1] <= nums[i]:  
            stack.pop()  
        # Record the current value's next largest number, which is at  
        # the top of the stack. If the stack is empty, record -1.  
        res[i] = stack[-1] if stack else -1  
        stack.append(nums[i])  
    return res
```

Complexity Analysis

Time complexity: The time complexity of `next_largest_number_to_the_right` is $O(n)$. This is because each value of `nums` is pushed and popped from the stack at most once.

Space complexity: The space complexity is $O(n)$ because the stack can potentially store all n values.

Evaluate Expression

Given a string representing a mathematical expression containing integers, parentheses, addition, and subtraction operators, evaluate and return the result of the expression.

Example:

Input: $s = "18-(7+(2-4))"$

Output: 13

Intuition

At first, it might seem overwhelming to deal with expressions that include a variety of elements like negative numbers, nested expressions inside parentheses, and numbers with multiple digits. The key to managing this complexity is to break down the problem into smaller, more manageable parts. Let's first focus on evaluating simple expressions that contain no parentheses.

Handling positive and negative signs

Consider the following expression:

2 8 - 1 0 + 7

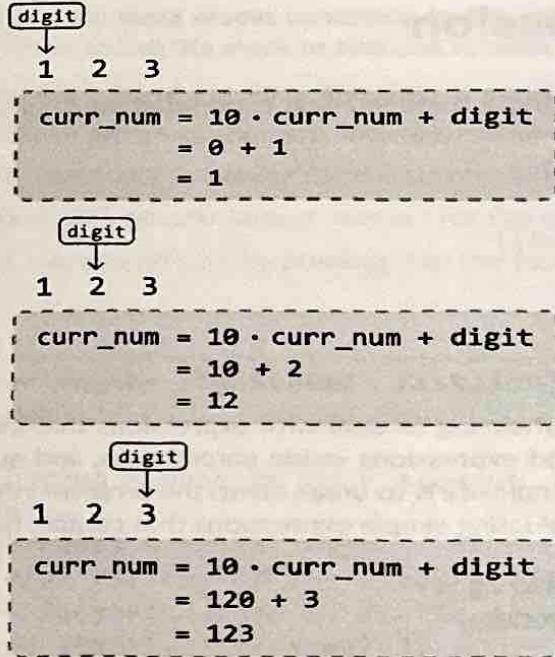
There's already some complexity in this expression with there being two signs to consider: plus and minus. An immediate simplification we can make is to treat all expressions as ones of pure addition. This is possible when we assign signs to each number (1 representing '+' and -1 representing '-'). This sign can be multiplied by the number to attain its correct value. This allows us to just focus on performing additions:

sign = 1 sign = -1 sign = 1
↑ ↑ ↑
(+) 2 8 (-) 1 0 (+) 7

Processing numbers with multiple digits

Another complexity in this expression is that some numbers have multiple digits. We'll need a way to build numbers digit by digit until we reach the end of the number. We can build a number using the variable `curr_num`, which is initially set to 0. Every time we encounter a new digit, we multiply `curr_num` by 10 and add the new digit to it, effectively shifting all digits to the left and appending the new digit.

We can see this process play out for the string "123" in the following diagrams:



We can stop building this number once we encounter a non-digit character, indicating the end of the number.

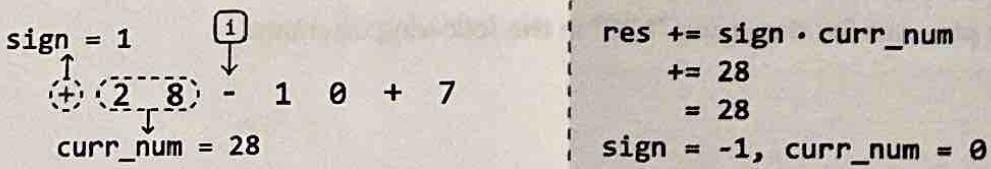
Evaluating an expression without parentheses

With the information from the section above, let's evaluate the following expression, which contains no parentheses. We'll start off with a sign of 1:

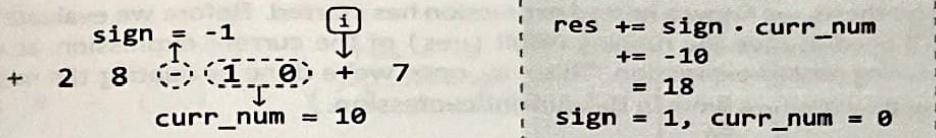
2 8 - 1 0 + 7, sign = 1, curr_num = 0

Upon reaching the '-' operator, we've reached the end of the first number (28). So, let's:

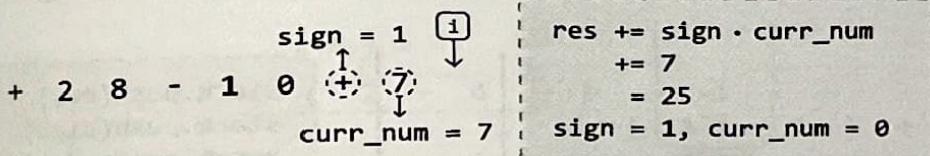
1. Multiply the current number (28) by its sign (1).
2. Add the resulting product (28) to the result.
3. Update the sign to -1 since the current operator is a minus sign.
4. Reset curr_num to 0 before building the next number.



Once we reach the second operator, we multiply the current number (10) by its sign of -1 before adding the resulting product (-10) to the result. This effectively subtracts 10 from the result:



Finally, once we've reached the end of the string, we just add the final number (7) to the result after multiplying it by its sign of 1:



Evaluating expressions containing parentheses

Now that we can solve simple expressions, it's time to bring parentheses into the discussion. Moving forward, we define a nested expression as one that's inside a pair of parentheses.

One challenge is that we need to evaluate the results of nested expressions before we can calculate the original expression. Once all nested expressions are evaluated, we can evaluate the original expression.

$$\begin{array}{r}
 1 \ 8 \ - \ (\ 7 \ + \ \underline{(\ 2 \ - \ 4 \)} \) \\
 \text{solve} \\
 1 \ 8 \ - \ \underline{(\ 7 \ - \ 2 \)} \\
 \text{solve} \\
 \hline
 1 \ 8 \ - \ 5 \\
 \text{solve}
 \end{array}$$

Consider another problem in this chapter that also contains parentheses: *Valid Parenthesis Expression*. In that problem, we used a **stack** to process nested parentheses in the right order. This suggests a stack might also help us evaluate nested expressions in the right order. Let's explore this idea further.

Similar to *Valid Parenthesis Expression*, an opening parenthesis '(' indicates the start of a new nested expression, whereas a closing parenthesis ')' indicates the end of one. Understanding this, let's try to use a stack to solve the following expression.

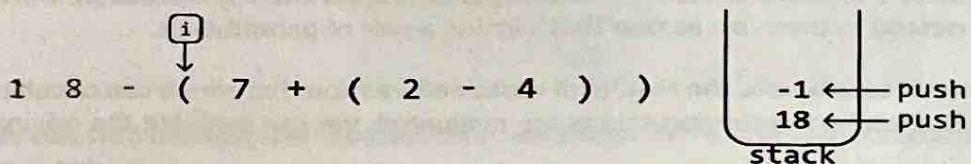
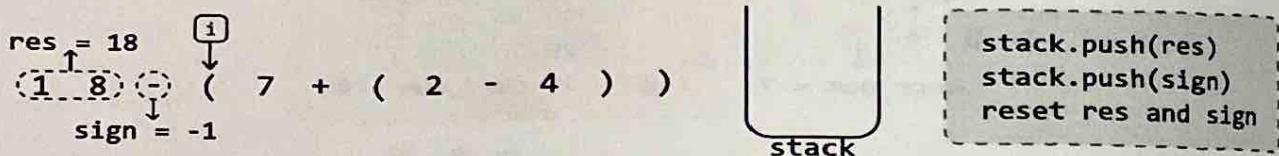
$$1 \ 8 \ - \ (\ 7 \ + \ (\ 2 \ - \ 4 \) \) \quad \boxed{} \text{ stack}$$

We already know how to evaluate expressions without parentheses, so let's just focus on what to do when we encounter a parenthesis.

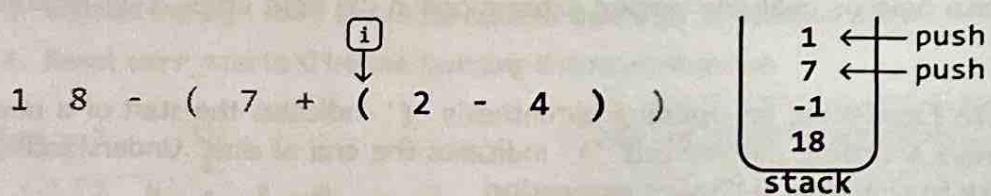
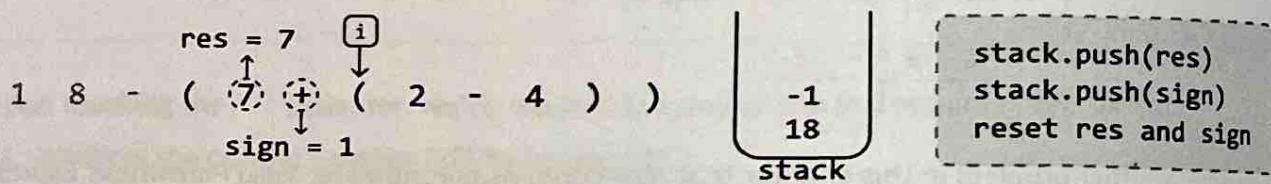
At the first opening parenthesis, we know a nested expression has started. Before we evaluate the nested expression, we'll need to save the running result (`res`) of the current expression, as well as the sign of this upcoming nested expression. This way, once we're done evaluating the nested expression, we can resume where we were in the current expression.

Here are the steps for when we encounter an opening parenthesis:

1. `stack.push(res)`: Save the running result on the stack.
2. `stack.push(sign)`: Save the sign of the upcoming nested expression on the stack.
3. `res = 0, sign = 1`: Reset these variables because we're about to begin calculating a new expression.

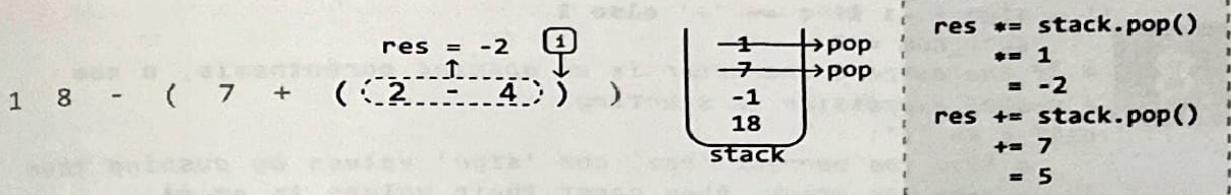


The next parenthesis we encounter is an opening parenthesis. Again, this indicates the start of a new nested expression. Let's save the current result and sign on the stack, before resetting them to evaluate the upcoming nested expression:

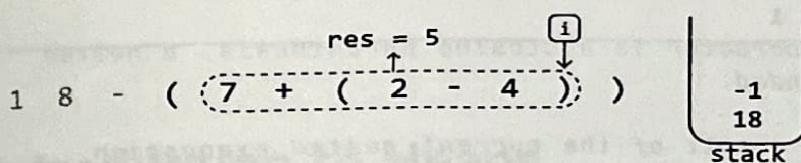


The next parenthesis we encounter is a closing parenthesis. This means the current nested expression just ended, and we need to merge its result with the outer expression. Here's how we do this:

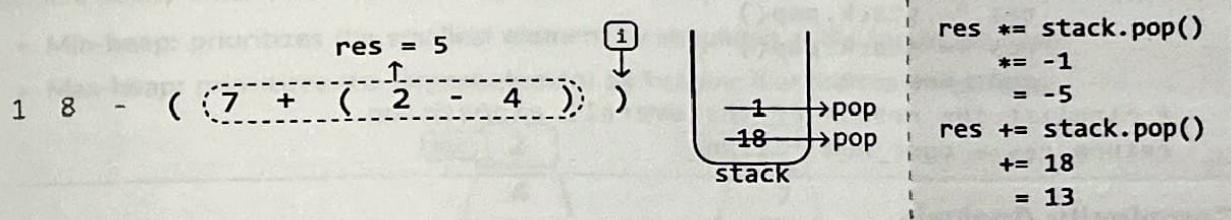
1. `res *= stack.pop()`: Apply the sign of the current nested expression to its result.
2. `res += stack.pop()`: Add the result of the outer expression to the result of the current nested expression.



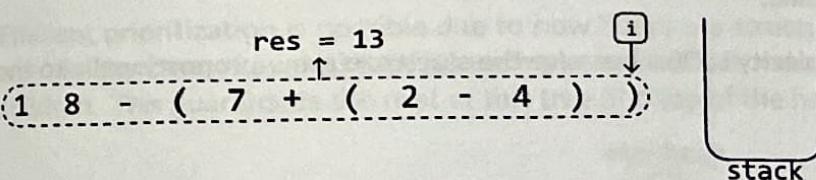
After applying those operations, the value of res will be 5, representing the result of the highlighted part of the expression below:



At the final closing parenthesis, we can apply the same steps:



Finally, the value of res will be 13, representing the result of the entire expression:



Now that we've reached the end of the string, we can return res.

Implementation

```

def evaluate_expression(s: str) -> int:
    stack = []
    curr_num, sign, res = 0, 1, 0
    for c in s:
        if c.isdigit():
            curr_num = curr_num * 10 + int(c)
        # If the current character is an operator, add 'curr_num' to
        # the result after multiplying it by its sign.
        elif c == '+' or c == '-':
            res += curr_num * sign
            # Update the sign and reset 'curr_num'.
            curr_num = 0
            sign = 1 if c == '+' else -1
    return res
  
```

```

        sign = -1 if c == '-' else 1
        curr_num = 0
    # If the current character is an opening parenthesis, a new
    # nested expression is starting.
    elif c == '(':
        # Save the current 'res' and 'sign' values by pushing them
        # onto the stack, then reset their values to start
        # calculating the new nested expression.
        stack.append(res)
        stack.append(sign)
        res, sign = 0, 1
    # If the current character is a closing parenthesis, a nested
    # expression has ended.
    elif c == ')':
        # Finalize the result of the current nested expression.
        res += sign * curr_num
        # Apply the sign of the current nested expression's result
        # before adding this result to the result of the outer
        # expression.
        res *= stack.pop()
        res += stack.pop()
        curr_num = 0
    # Finalize the result of the overall expression.
return res + curr_num * sign

```

Complexity Analysis

Time complexity: The time complexity of `evaluate_expression` is $O(n)$ because we traverse each character of the expression once, processing nested expressions using the stack, where each stack push or pop operation takes $O(1)$ time.

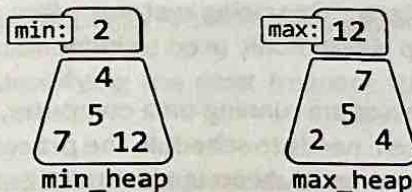
Space complexity: The space complexity is $O(n)$ because the stack can grow proportionally to the length of the expression.

Heaps

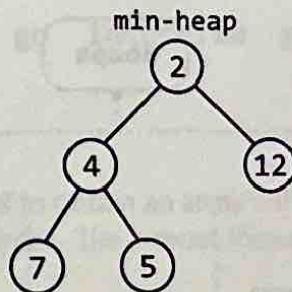
Introduction to Heaps

A heap is a data structure that organizes elements based on priority, ensuring the highest-priority element is always at the top of the heap. This allows for efficient access to the highest-priority element at any time. There are two main types of heaps:

- **Min-heap:** prioritizes the smallest element by keeping it at the top of the heap.
- **Max-heap:** prioritizes the largest element by keeping it at the top of the heap.



Efficient prioritization is possible due to how heaps are structured. A heap is essentially a binary tree. In the case of a min-heap, for example, each node's value is less than or equal to that of its children. This guarantees the root of this tree (the top of the heap) is always the smallest element:



Here's a time complexity breakdown of common heap operations:

Operation	Time complexity	Description
Insert	$O(\log(n))$	Adds an element to the heap, ensuring the binary tree remains correctly ordered.
Deletion	$O(\log(n))$	Removes the element at the top of the heap, then restructures the heap to replace the top element.
Peek	$O(1)$	Retrieves the top element of the heap without removing it.
Heapify	$O(n)$	Transforms an unsorted list of values into a heap [1].

This chapter discusses the practical uses of heaps. For a deeper understanding of how a heap works, we recommend diving into the details behind its internal implementation, and how its binary tree structure is consistently maintained during various operations [2].

Priority queue

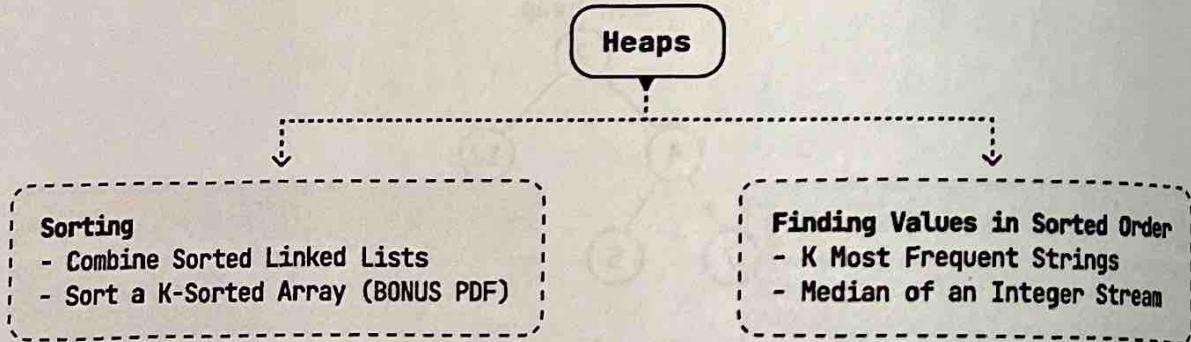
A priority queue is a special type of heap that follows the structure of min-heaps or max-heaps but allows customization in how elements are prioritized (e.g., prioritizing strings with a higher number of vowels).

Real-world Example

Managing tasks in operating systems: Operating systems often use a priority queue to manage the execution of tasks, and a heap is commonly used to implement this priority queue efficiently.

For example, when multiple processes are running on a computer, each process might be assigned a priority level. The operating system needs to schedule the processes so that higher-priority tasks are executed before lower-priority ones. A heap is ideal for this because it allows the system to quickly access the highest-priority task and efficiently re-arrange the priorities as new tasks are added or existing tasks are completed.

Chapter Outline



References

- [1] Time complexity analysis of the heapify operation: <https://www.prepbytes.com/blog/heap-time-complexity-of-building-a-heap/>
- [2] Breakdown of heap operations: https://en.wikipedia.org/wiki/Binary_heap

K Most Frequent Strings

Find the k most frequently occurring strings in an array, and return them sorted by frequency in descending order. If two strings have the same frequency, sort them in lexicographical order.

Example:

Input: `strs = ["go", "coding", "byte", "byte", "go", "interview", "go"]`,
 $k = 2$
Output: `["go", "byte"]`

Explanation: The strings "go" and "byte" appear the most frequently, with frequencies of 3 and 2, respectively.

Constraints:

- $k \leq n$, where n denotes the length of the array.

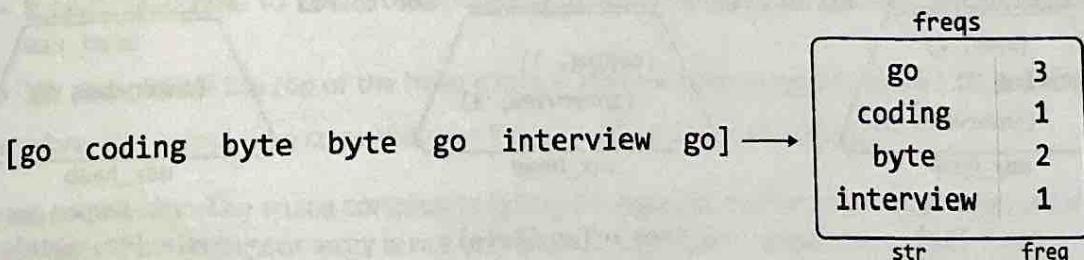
Intuition – Max-Heap

The two main challenges to this problem are:

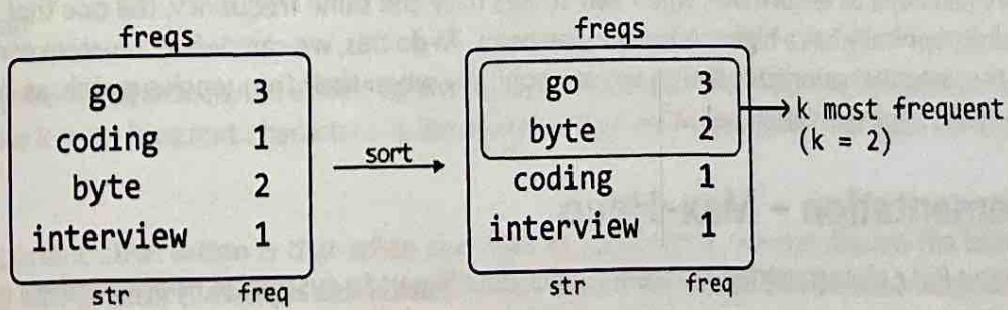
1. Identifying the k most frequent strings.
2. Sorting those strings first by frequency and then lexicographically.

For now, let's concentrate on identifying the most frequent strings and address lexicographical ordering afterward.

First, we need a way to keep track of the frequencies of each string. We can use a hash map for this, where the keys represent the strings and the values represent frequencies:



The most straightforward approach is to obtain an array containing the strings from the hash map, sorted by frequency in descending order. The k most frequent strings would be the first k strings in this array.



The main inefficiency of this solution is that it involves sorting all n strings, even though we only

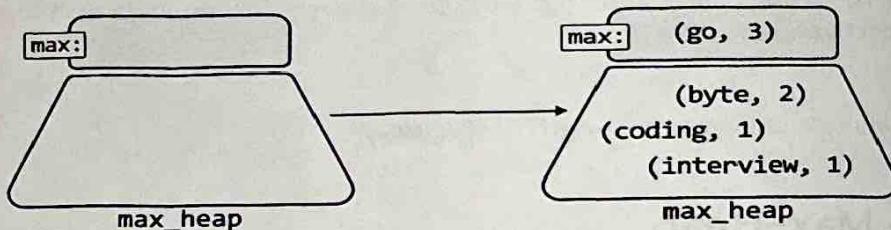
need the top k frequent ones to be sorted.

Something useful to consider: if we remove the most frequent string, the new most frequent string after this removal represents the second-most frequent overall. By repeatedly identifying and removing the most frequent string k times, we efficiently obtain our answer.

To implement this idea, we need a data structure that allows efficient access to the most frequent string at any time. A **max-heap** is perfect for this.

Max-heap

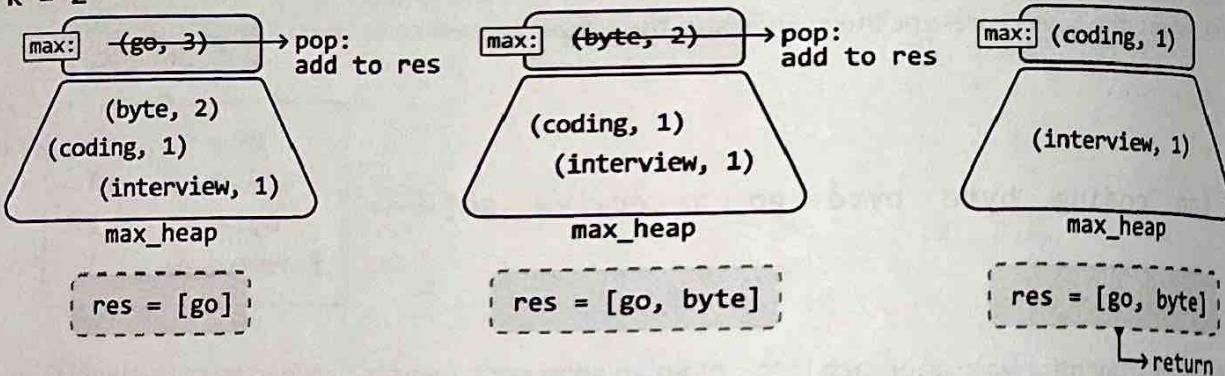
Let's find the k most frequent strings from the previous input, this time using a max-heap. First, populate the heap with each string along with their frequencies.



One way to populate the heap is to push all n strings into it one by one, which will take $O(n \log(n))$ time. Instead, we can perform the **heapify** operation on an array containing all the string-frequency pairs to create the max-heap in $O(n)$ time.

To collect the k most frequent strings, pop off the top element from the heap k times and store the corresponding strings in the output array `res`:

$k = 2$



Now, we just need to ensure that when two strings have the same frequency, the one that comes first lexicographically has a higher priority in the heap. To do this, we can define a custom comparator for the heap that prioritizes strings lexicographically when their frequencies match, as demonstrated in the implementation below.

Implementation – Max-Heap

We create a `Pair` class for string-frequency pairs, enabling us to customize priority using a custom comparator.

```

class Pair:
    def __init__(self, str, freq):
        self.str = str
        self.freq = freq

    # Define a custom comparator.
    def __lt__(self, other):
        # Prioritize lexicographical order for strings with equal
        # frequencies.
        if self.freq == other.freq:
            return self.str < other.str
        # Otherwise, prioritize strings with higher frequencies.
        return self.freq > other.freq

def k_most_frequent_strings_max_heap(strs: List[str],
                                      k: int) -> List[str]:
    # We use 'Counter' to create a hash map that counts the frequency
    # of each string.
    freqs = Counter(strs)
    # Create the max heap by performing heapify on all string-frequency
    # pairs.
    max_heap = [Pair(str, freq) for str, freq in freqs.items()]
    heapq.heapify(max_heap)
    # Pop the most frequent string off the heap 'k' times and return
    # these 'k' most frequent strings.
    return [heapq.heappop(max_heap).str for _ in range(k)]

```

Complexity Analysis

Time complexity: The time complexity of `k_most_frequent_strings_max_heap` is $O(n+k \log(n))$.

- It takes $O(n)$ time to count the frequency of each string using Counter, and to build the `max_heap`.
- We also pop off the top of the heap k times, with each pop operation taking $O(\log(n))$ time.

Therefore, the overall time complexity is $O(n) + k \cdot O(\log(n)) = O(n + k \log(n))$.

Space complexity: The space complexity is $O(n)$ because the hash map and heap store at most n pairs. Note that the output array is not considered in the space complexity.

Intuition – Min-Heap

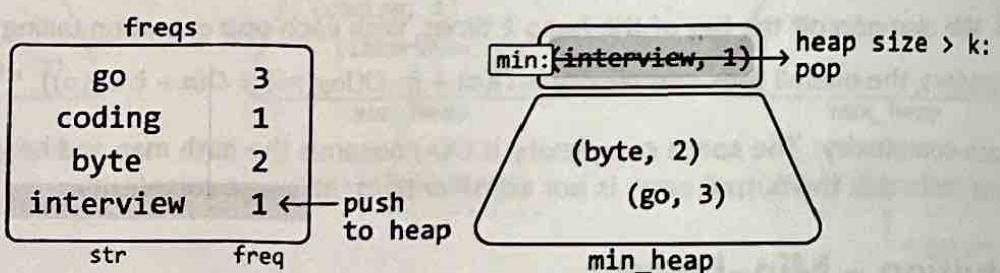
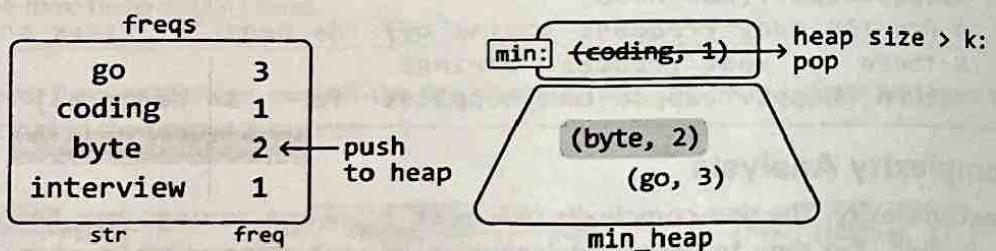
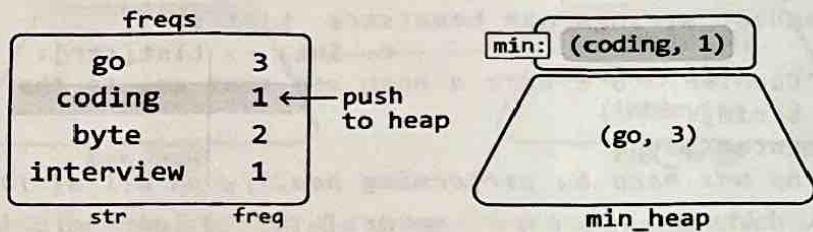
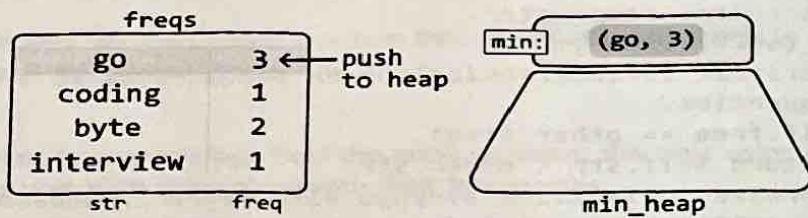
As a follow up, your interviewer may ask you to modify your solution to reduce the space used by the heap.

In the previous approach, we ended up storing up to n items in the heap. However, since we only need the k most frequent characters, is there a way to maintain a heap with a space complexity of $O(k)$?

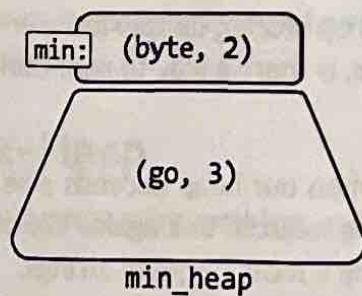
An important observation is that when our heap exceeds size k , we can **discard the lowest frequency strings until the heap's size is reduced to k again**. We can do this because those discarded strings definitely won't be among the k most frequent strings.

However, we can't implement this strategy with a max-heap because we won't have access to the lowest frequency string. Instead, we need to use a min-heap.

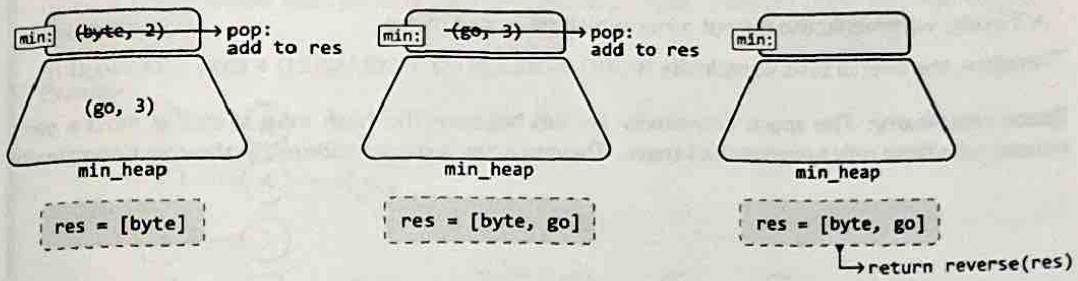
Let's observe how this works over an example:



In the end, the strings remaining in the heap are the top k frequent strings:



To retrieve these strings, pop them from the heap until it's empty. Because we're using a min-heap, we're popping off the less frequent strings first. So, we need to reverse the order of the retrieved strings before returning the result:



```
class Pair:  
    def __init__(self, str, freq):  
        self.str = str  
        self.freq = freq  
    # Since this is a min-heap comparator, we can use the same  
    # comparator as the one used in the max-heap, but reversing the  
    # inequality signs to invert the priority.  
    def __lt__(self, other):  
        if self.freq == other.freq:  
            return self.str > other.str  
        return self.freq < other.freq  
  
def k_most_frequent_strings_min_heap(strs: List[str],  
                                     k: int) -> List[str]:  
    freqs = Counter(strs)  
    min_heap = []  
    for str, freq in freqs.items():  
        heapq.heappush(min_heap, Pair(str, freq))  
    # If heap size exceeds 'k', pop the lowest frequency string to  
    # ensure the heap only contains the 'k' most frequent words so  
    # far.  
    if len(min_heap) > k:  
        heapq.heappop(min_heap)  
    # Return the 'k' most frequent strings by popping the remaining 'k'  
    # strings from the heap. Since we're using a min-heap, we need to  
    # reverse the result after popping the elements to ensure the most  
    # frequent strings are listed first.  
    res = [heapq.heappop(min_heap).str for _ in range(k)]  
    res.reverse()  
    return res
```

Complexity Analysis

Time complexity: The time complexity of `k_most_frequent_strings_min_heap` is $O(n \log(k))$.

- It takes $O(n)$ time to count the frequency of each string using `Counter`.
- To populate the heap, we push n words onto it, with each push and pop operation taking

$O(\log(k))$ time. This takes $O(n \log(k))$ time.

- Then, we extract k strings from the heap by performing the pop operation k times. This takes $O(k \log(k))$ time.
- Finally, we reverse the output array, which takes $O(k)$ time.

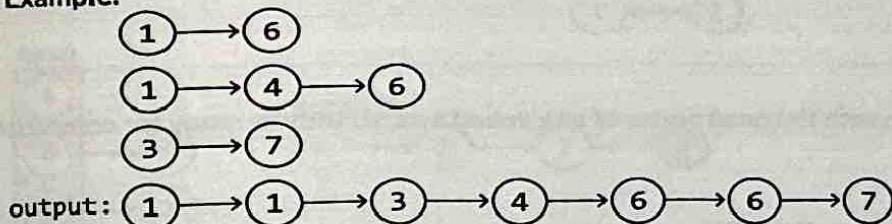
Therefore, the overall time complexity is $O(n) + O(n \log(k)) + O(k \log(k)) + O(k) = O(n \log(k))$.

Space complexity: The space complexity is $O(n)$ because the hash map stores at most n pairs, whereas the heap only takes up $O(k)$ space. The `res` array is not considered in the space complexity.

Combine Sorted Linked Lists

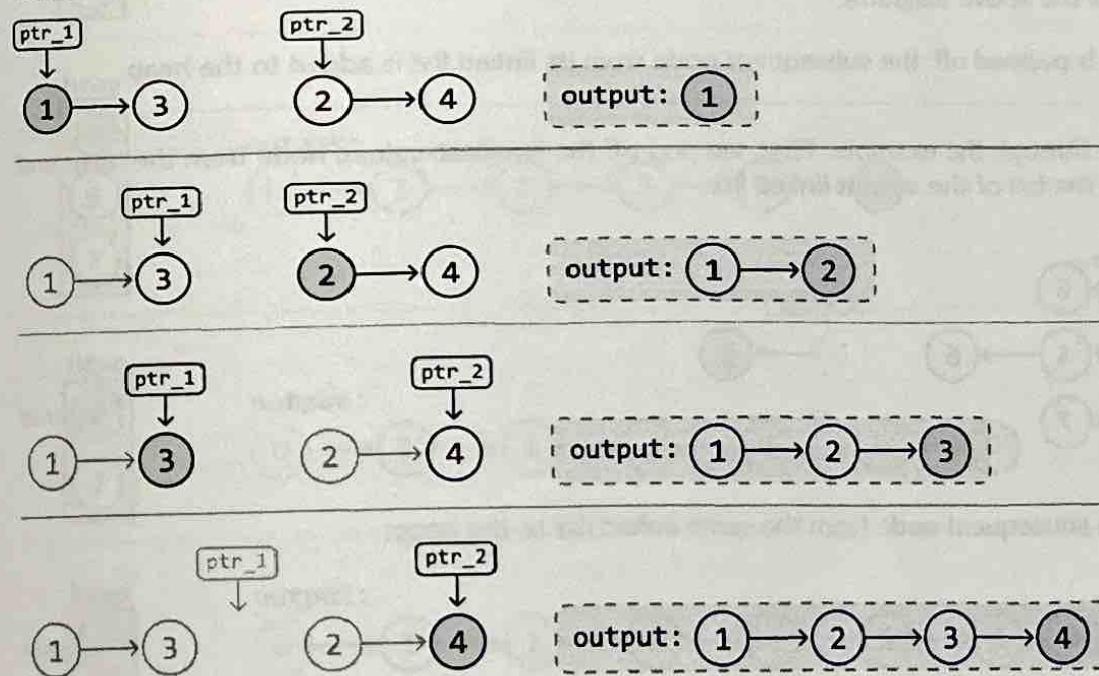
Given k singly linked lists, each sorted in ascending order, combine them into one sorted linked list.

Example:



Intuition

A good place to start with this problem is by figuring out how to merge just two sorted linked lists. We can do this by initiating a pointer at the start of both linked lists. Comparing the nodes at these pointers, add the smaller one to the output linked list and advance the corresponding pointer. This results in a combined sorted linked list:

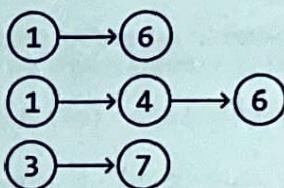


But what if we have more than two linked lists? Combining two linked lists involves comparing two nodes at each iteration, but combining k linked lists would require k comparisons per iteration.

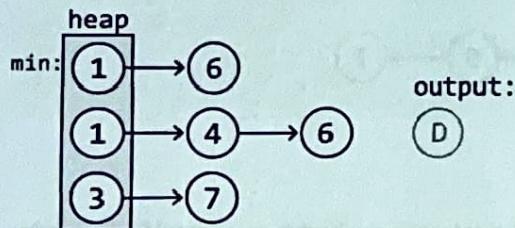
The reason we need to make so many comparisons is that we don't know which node has the smallest value at any point in the iteration, requiring us to search for it. Wouldn't it be nice to have an efficient way to access the smallest-valued node at any given point? A **min-heap** is perfect for this.

We can essentially do the same thing as in our initial approach, but instead of using pointers to

determine the smallest node, we use a min-heap. Let's see how this works over the three sorted linked lists below:



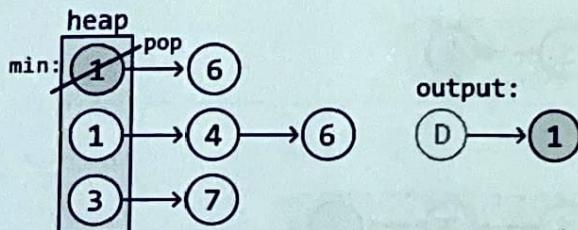
To start, populate the heap with the head nodes of all k linked lists, so they're ready for comparison:



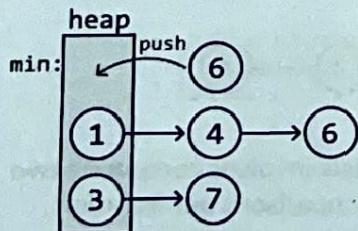
Then, let's implement our strategy of adding the smallest-valued node to the output linked list, using the heap to identify it. We'll use a dummy node to help build the output linked list (denoted as node 'D' in the above diagram).

After a node is popped off, the subsequent node from its linked list is added to the heap.

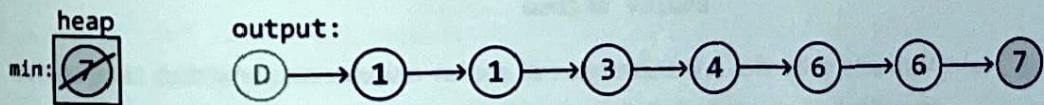
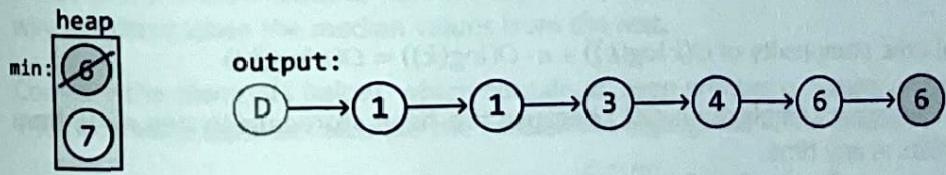
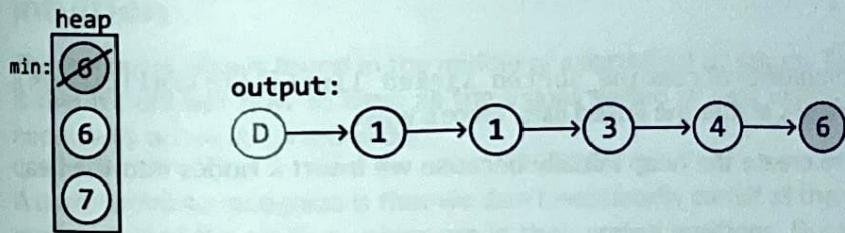
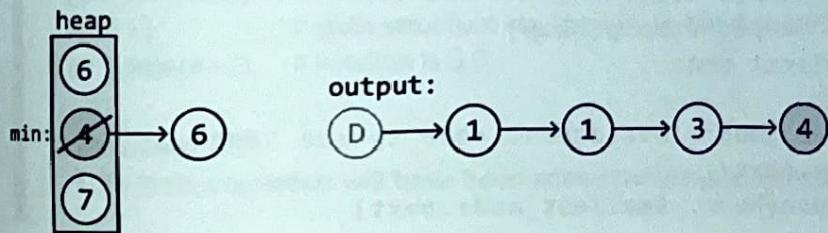
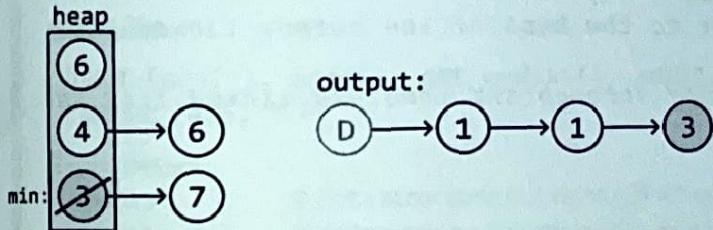
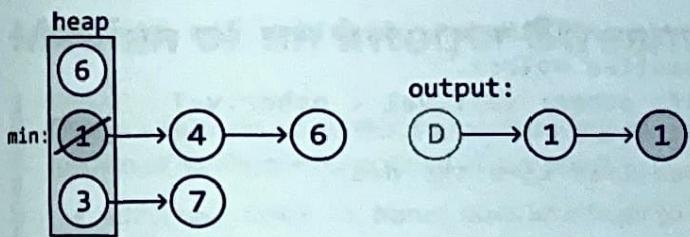
Now, let's go through the example. First, we pop off the smallest-valued node from the heap and connect it to the tail of the output linked list:



Then, add the subsequent node from the same linked list to the heap:



Continue this until we've added each node from all k linked lists to the output linked list:



Once the heap is empty, we can return `dummy.next`, which is the head of the combined linked list.

Implementation

Note that in the implementation below, we modify the `ListNode` class globally to simplify the solution. It's important to confirm with your interviewer that global variables are acceptable.

```
def combine_sorted_linked_lists(lists: List[ListNode]) -> ListNode:
```

```

# Define a custom comparator for 'ListNode', enabling the min-heap
# to prioritize nodes with smaller values.
ListNode.__lt__ = lambda self, other: self.val < other.val
heap = []
# Push the head of each linked list into the heap.
for head in lists:
    if head:
        heapq.heappush(heap, head)
# Set a dummy node to point to the head of the output linked list.
dummy = ListNode(-1)
# Create a pointer to iterate through the combined linked list as
# we add nodes to it.
curr = dummy
while heap:
    # Pop the node with the smallest value from the heap and add it
    # to the output linked list.
    smallest_node = heapq.heappop(heap)
    curr.next = smallest_node
    curr = curr.next
    # Push the popped node's subsequent node to the heap.
    if smallest_node.next:
        heapq.heappush(heap, smallest_node.next)
return dummy.next

```

Complexity Analysis

Time complexity: The time complexity of `combine_sorted_linked_lists` is $O(n \log(k))$, where n denotes the total number of nodes across the linked lists. Here's why:

- It takes $O(k \log(k))$ time to create the heap initially because we insert k nodes into the heap one by one.
- Then, for all n nodes, we perform a push and pop operation on the heap, each taking $O(\log(k))$ time.

This results in a total time complexity of $O(k \log(k)) + n \cdot O(\log(k)) = O(n \log(k))$.

Space complexity: The space complexity is $O(k)$ because the heap stores up to one node from each of the k linked lists at any time.

Median of an Integer Stream

Design a data structure that supports adding integers from a data stream and retrieving the median of all elements received at any point.

- `add(num: int) -> None`: adds an integer to the data structure.
- `get_median() -> float`: returns the median of all integers so far.

Example:

Input: [add(3), add(6), get_median(), add(1), get_median()]
Output: [4.5, 3.0]

Explanation:

```
add(3)      # data structure contains [3] when sorted  
add(6)      # data structure contains [3, 6] when sorted  
get_median() # median is (3 + 6) / 2 = 4.5  
add(1)      # data structure contains [1, 3, 6] when sorted  
get_median() # median is 3.0
```

Constraints:

- At least one value will have been added before `get_median` is called.

Intuition

The median is always found in the middle of a sorted list of values. The challenge with this problem is that it's unclear how to keep all the values sorted as new values arrive, since the values don't necessarily arrive in sorted order.

A useful point to recognize is that we don't necessarily care if all the values are sorted. What really matters is that the median values are in their sorted positions. But is it possible to position these values in the middle without maintaining a fully sorted list of values? If it is, we'd need to find a way to differentiate the median values from the rest.

Consider the elements below, which contain an even number of values arranged in sorted order. The two values used to calculate the median are highlighted in the middle:

0 1 2 2 [3 4] 5 7 8 9
median values

When a list contains two median values, we can make the following observations:

- The first median is the largest value in the left half of the sorted integers.
- The second median is the smallest value in the right half of the sorted integers.

0 1 2 2 [3 4] 5 7 8 9
largest of smallest of
left half right half

If we had a way to split the data into two halves, with one half containing the smaller values and the other half containing the larger values, we would just need an efficient method to identify the

largest value in the smaller half, and the smallest value in the larger half. This is where heaps come in.

We can use a combination of a min-heap and a max-heap:

- A max-heap manages the left half, where the top value represents the first median value.
- A min-heap manages the right half, where the top value represents the second median value.

If the total number of elements is odd, there's only one median. In this case, we just need to use one of the heaps to store it. Let's designate the max-heap to store this median:



Populating the heaps

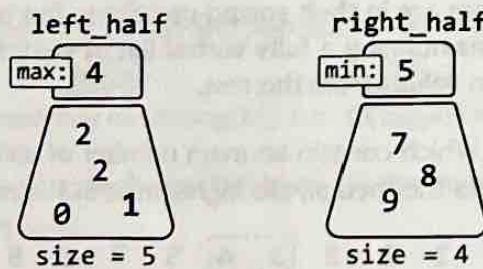
Before identifying how we populate each heap, it's useful to understand the behavior they must follow. Here are a couple of observations we can make:

- All values in the left half must be less than or equal to any value in the right half.
- The two halves should contain an equal number of values, except when the total number of values is odd, in which case the left half has one more value, as specified earlier.

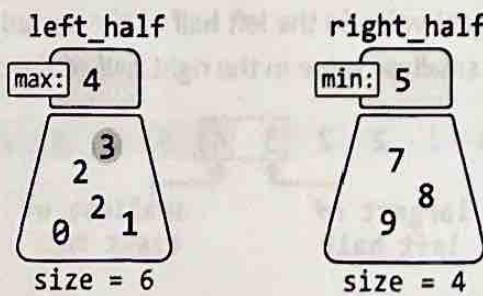
These observations help us define two rules for managing the heaps:

1. The maximum value of the max-heap (left half) must be less than or equal to the minimum value of the min-heap (right half), ensuring all values in the left half are less than or equal to those in the right half.
2. The heaps should be of equal size, but the max-heap can have one more element than the min-heap.

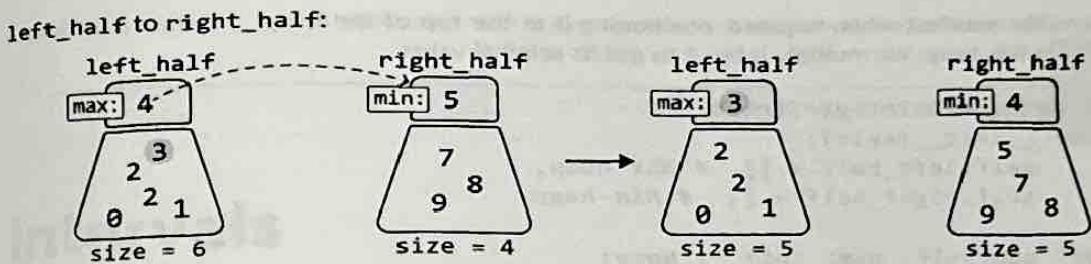
Let's figure out how to maintain these rules in an example in which we try to add number 3 to the heaps. Note, the heaps before adding this number meet the above rules.



Since 3 is less than the maximum value of `left_half` (4), it belongs in the `left_half` heap. So, let's add 3 to this heap:

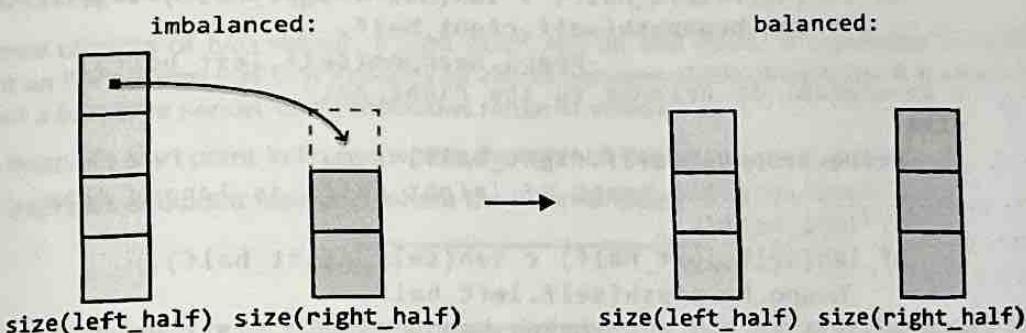


After adding 3, we notice rule 2 has been violated, since the size of the `left_half` heap is more than one element larger than the `right_half` heap. We can fix this by moving the max value of

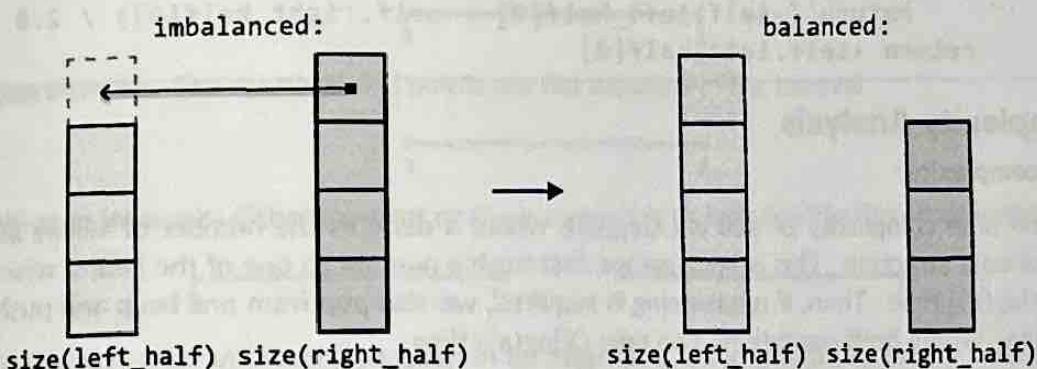


So, to ensure the sizes of the heaps don't violate rule 2, we need to rebalance the heaps after adding a value:

- If the `left_half` heap's size exceeds the `right_half` heap's size by more than one, rebalance the heaps by transferring `left_half`'s top value to `right_half`:



- If the `right_half` heap's size exceeds the `left_half` heap's size, rebalance the heaps by transferring its top value to the `left_half`:



Returning the median

With the median values at the top of the heaps, returning the median boils down to two cases:

- If the total number of elements is even, both median values can be found at the top of each heap. So, we return their sum divided by 2.
- If the total number of elements is odd, the median will be at the top of the `left_half` heap.

Implementation

Note that in Python, heaps are min-heaps by default. To mimic the functionality of a max-heap, we can insert numbers as negatives in the `left_half` heap. This way, the largest original value

becomes the smallest when negated, positioning it at the top of the heap. When we retrieve a value from this heap, we multiply it by -1 to get its original value.

```
class MedianOfAnIntegerStream:
    def __init__(self):
        self.left_half = [] # Max-heap.
        self.right_half = [] # Min-heap.

    def add(self, num: int) -> None:
        # If 'num' is less than or equal to the max of 'left_half', it
        # belongs to the left half.
        if not self.left_half or num <= -self.left_half[0]:
            heapq.heappush(self.left_half, -num)
            # Rebalance the heaps if the size of the 'left_half'
            # exceeds the size of the 'right_half' by more than one.
            if len(self.left_half) > len(self.right_half) + 1:
                heapq.heappush(self.right_half,
                                -heapq.heappop(self.left_half))
        # Otherwise, it belongs to the right half.
        else:
            heapq.heappush(self.right_half, num)
            # Rebalance the heaps if 'right_half' is larger than
            # 'left_half'.
            if len(self.left_half) < len(self.right_half):
                heapq.heappush(self.left_half,
                                -heapq.heappop(self.right_half))

    def get_median(self) -> float:
        if len(self.left_half) == len(self.right_half):
            return (-self.left_half[0] + self.right_half[0]) / 2.0
        return -self.left_half[0]
```

Complexity Analysis

Time complexity:

- The time complexity of `add` is $O(\log(n))$, where n denotes the number of values added to the data structure. This is because we first push a number to one of the heaps, which takes $O(\log(n))$ time. Then, if rebalancing is required, we also pop from one heap and push to the other, where both operations also take $O(\log(n))$ time.
- The time complexity of `get_median` is $O(1)$ because accessing the top element of a heap takes $O(1)$ time.

Space complexity: The space complexity is $O(n)$ because the two heaps together store n elements.

Note: this explanation refers to the two middle values as "median values" to keep things simple. However, it's important to understand that these two values aren't technically "medians," as there's only ever one median. These are just the two values used to calculate the median.

Intervals

Introduction to Intervals

An interval consists of two values: a start point and an end point. It represents a continuous segment on the number line that includes all values between these two points. It is often used to represent a line, time period, or a continuous range of values.

- An interval's start point indicates where the interval begins.
- An interval's end point indicates where the interval ends.



Intervals can be closed, open, or half-open, based on whether their start or end points are included in the interval.

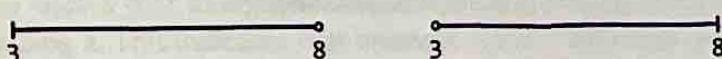
- **Closed intervals:** Both the start and end points are included in the interval.



- **Open intervals:** The start and end points are not included in the interval.



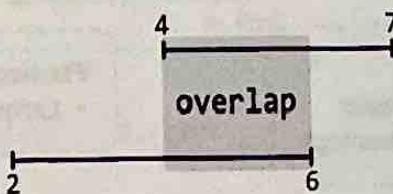
- **Half-open intervals:** Either the start or the end point is included, while the other is not.



When presented with an interval problem in an interview, it's important to clarify whether the intervals are open, closed, or half-open, as this can change the nature of how intervals overlap.

Overlapping intervals

Two intervals overlap if they share at least one common value.



The central challenge in most interval problems involves managing overlapping intervals effectively. Whether identifying or merging overlapping intervals, it's important to determine how the overlap

between intervals influences the desired outcome of the problem. The problems in this chapter involve handling overlapping intervals in varying situations.

Sorting intervals

In most interval problems, sorting the intervals before solving the problem is quite helpful since it allows them to be processed in a certain order.

We usually sort intervals by their start point so they can be traversed in chronological order. When two or more intervals have the same start point, we might also need to consider each interval's end points during sorting.

Separating start and end points

In certain scenarios, it might be beneficial to process the start and end points of intervals separately. This usually involves creating two sorted arrays: one containing all start points and another containing all end points. For example, this is needed in the sweeping line algorithm, which is explored in the *Largest Overlap of Intervals* problem.

```
intervals = [[1, 6] [2, 4] [5, 9] [8, 10] [10, 12]]  
start points = [ 1   2   5   8   10 ]  
end points = [ 4   6   9   10  12 ]
```

Interval class definition

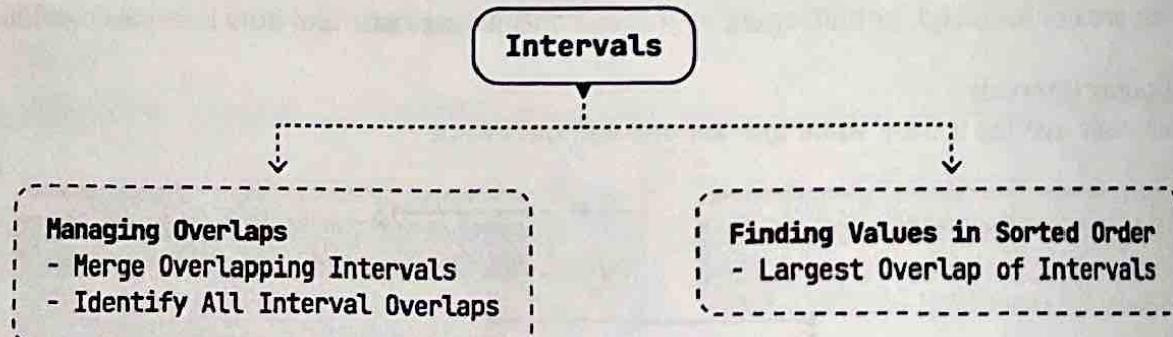
For the problems in this chapter, intervals are represented using the class below.

```
class Interval:  
    def __init__(self, start, end):  
        self.start = start  
        self.end = end
```

Real-world Example

Scheduling systems: Intervals are widely used in scheduling systems. For instance, in a conference room booking system, each booking is represented as an interval. The interval representation is used if the system requires functionality, such as determining the maximum number of overlapping bookings to ensure sufficient room availability. By analyzing these intervals, the system can efficiently allocate resources and prevent double bookings.

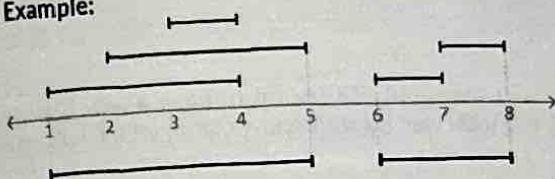
Chapter Outline



Merge Overlapping Intervals

Merge an array of intervals so there are no overlapping intervals, and return the resultant merged intervals.

Example:



Input: intervals = [[3, 4], [7, 8], [2, 5], [6, 7], [1, 4]]

Output: [[1, 5], [6, 8]]

Constraints:

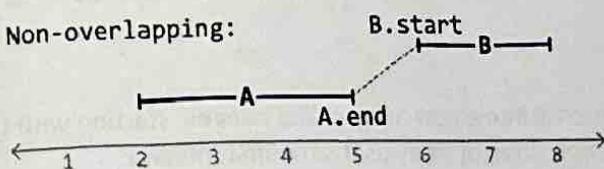
- The input contains at least one interval.
- For every index i in the array, $\text{intervals}[i].\text{start} \leq \text{intervals}[i].\text{end}$.

Intuition

There are two main challenges to this problem:

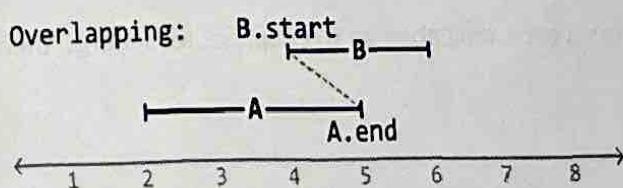
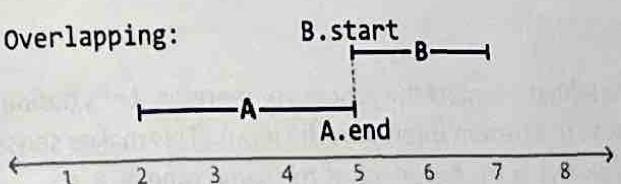
- Identifying which intervals overlap each other.
- Merging those intervals.

Let's start by tackling the first challenge. Consider two intervals, A and B, where interval A starts before B. Below, we visualize the case when these two intervals don't overlap:



The dashed line above shows that interval A ends before interval B starts, which eliminates the possibility of B overlapping A. This indicates that intervals A and B will never overlap when $A.\text{end} < B.\text{start}$.

Now, consider a couple of cases where these two intervals do overlap:



In these cases, we see that B starts before (or when) A ends ($A.end \geq B.start$). In other words, some portion of B overlaps A since interval A hasn't ended before interval B starts. Therefore, intervals A and B overlap when $A.end \geq B.start$.

We have now established the two cases that cover all overlapping and non-overlapping scenarios for two intervals, given interval A starts before interval B:

- If $A.end < B.start$, the intervals don't overlap.
- If $A.end \geq B.start$, the intervals overlap.

To apply these conditions to any two intervals in the input, it's useful to have a way to identify which interval starts first. One idea is to sort the intervals by their start value, which will make it clear which one of each two adjacent intervals starts first.

Merging intervals

With the above logic in mind, let's tackle an example. Consider the following intervals:

[3, 4] [7, 8] [2, 5] [6, 7] [1, 4]

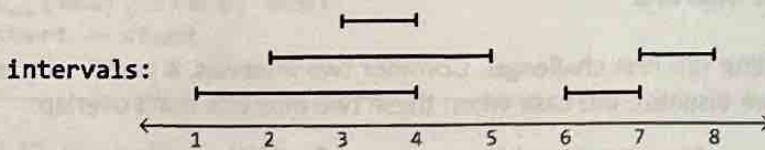
The first step is to sort these intervals by start value:

[3, 4] [7, 8] [2, 5] [6, 7] [1, 4]

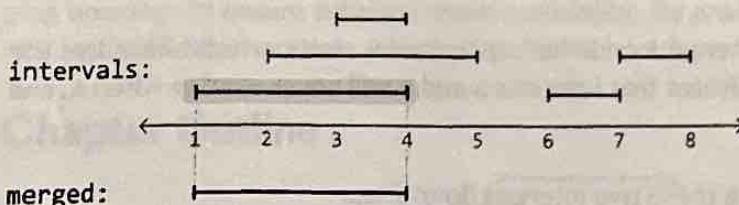
↓
sort

[1, 4] [2, 5] [3, 4] [6, 7] [7, 8]

To aid the explanation, let's represent the intervals visually:

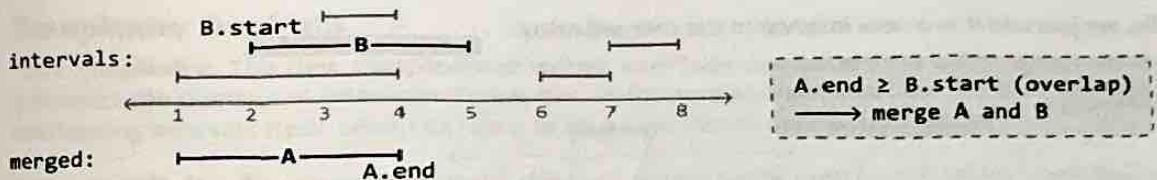


Let's add/merge each interval into a new array called merged, starting with the first one, which we can add to the merged array straight away as it's the first interval:

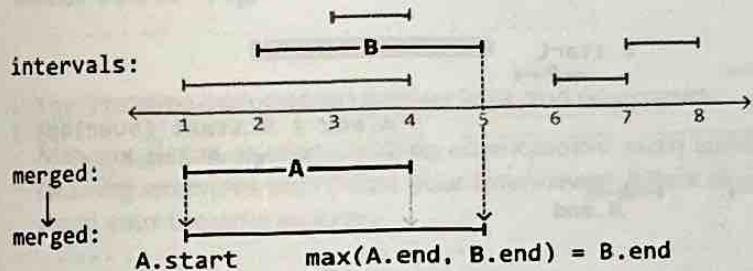


After the first interval is added, we start the process of merging. Let's define A as the last interval in the merged array and B as the current interval in the input. This makes sense since the last interval in the merged array (A) always starts before or at the same time as B.

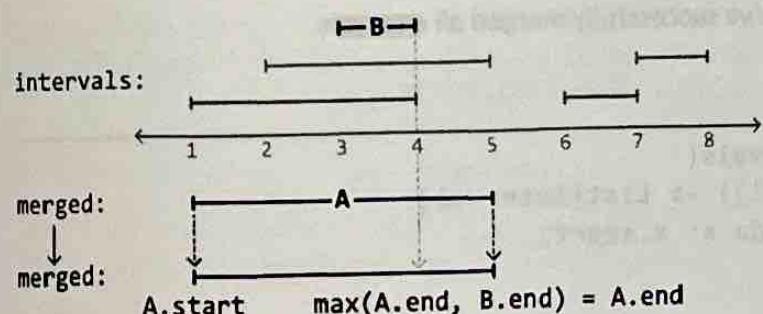
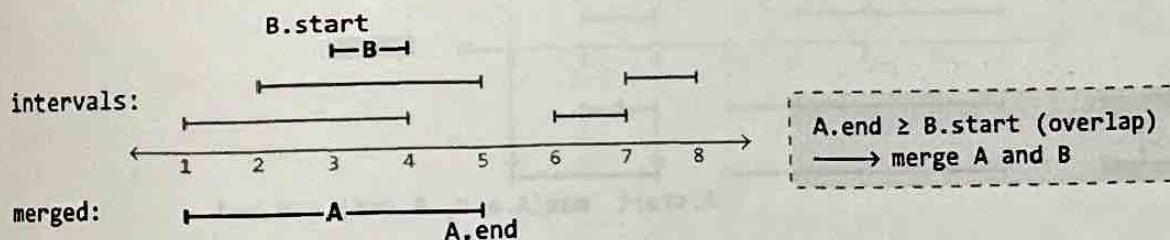
We notice B starts before A ends, indicating an overlap. So, let's merge them:



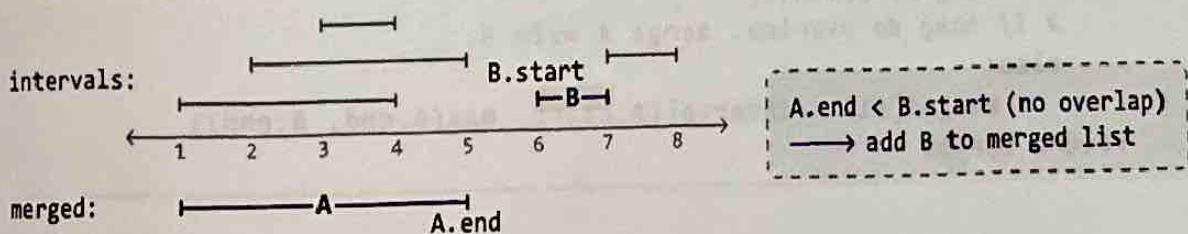
When merging A and B, we use the leftmost start value and the rightmost end value between them. Since A will always start before or at the same time as B, we always use A.start as the start point. This means we just need to identify the end point, which is the largest value between the end points of A and B:



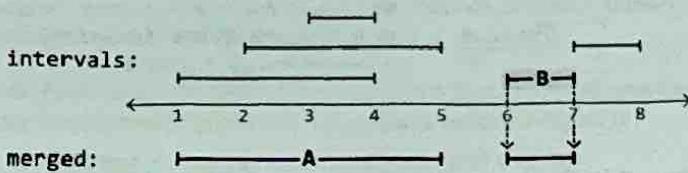
We can apply the same logic to the next interval:



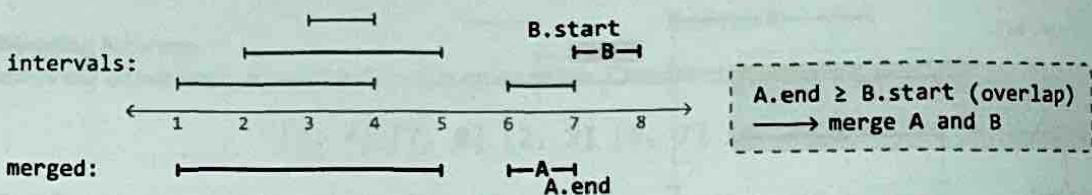
When we reach the fourth interval, we notice B starts after A ends, indicating there is no overlap.



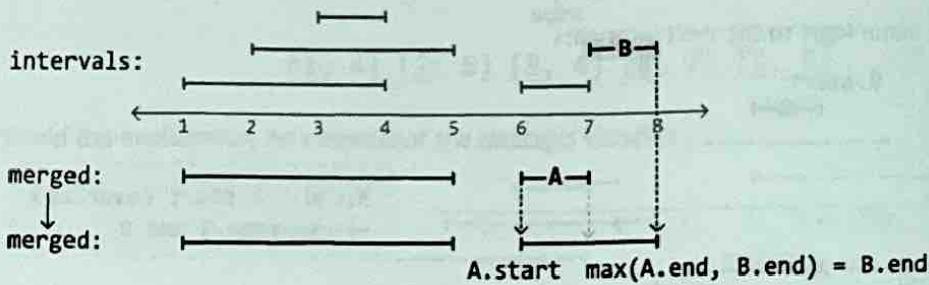
So, we just add it as a new interval to the merged array:



The next interval, B, overlaps the last interval in the merged array, A, since B starts when A ends ($A.end == B.start$).



So, let's merge A with B:



After processing the last interval, we've successfully merged all intervals.

Implementation

```
def merge_overlapping_intervals(  
    intervals: List[Interval]) -> List[Interval]:  
    intervals.sort(key=lambda x: x.start)  
    merged = [intervals[0]]  
    for B in intervals[1:]:  
        A = merged[-1]  
        # If A and B don't overlap, add B to the merged list.  
        if A.end < B.start:  
            merged.append(B)  
        # If they do overlap, merge A with B.  
        else:  
            merged[-1] = Interval(A.start, max(A.end, B.end))  
    return merged
```

Complexity Analysis

Time complexity: The time complexity of `merge_overlapping_intervals` is $O(n \log(n))$, where n denotes the number of intervals. This is due to the sorting algorithm. The process of merging overlapping intervals itself takes $O(n)$ time because we iterate over every interval.

Space complexity: The space complexity depends on the space used by the sorting algorithm. In Python, the built-in sorting algorithm, Tim sort, uses $O(n)$ space. Note that the merged array is not considered in the space complexity calculation because we're only concerned with extra space used, not space taken up by the output.

Interview Tip



Tip: Visualize intervals to uncover logic and edge cases.

Managing intervals and handling edge cases is much easier when visualizing example inputs. Drawing examples also helps your interviewer follow along with your reasoning and understand your thought process.

Identify All Interval Overlaps

Return an array of all overlaps between two arrays of intervals; `intervals1` and `intervals2`. Each individual interval array is sorted by start value, and contains no overlapping intervals within itself.

Example:

`intervals1:`



`intervals2:`



`intersections:`



Input: `intervals1 = [[1, 4], [5, 6], [9, 10]]`,

`intervals2 = [[2, 7], [8, 9]]`

Output: `[[2, 4], [5, 6], [9, 9]]`

Constraints:

- For every index `i` in `intervals1`, `intervals1[i].start < intervals1[i].end`.
- For every index `j` in `intervals2`, `intervals2[j].start < intervals2[j].end`.

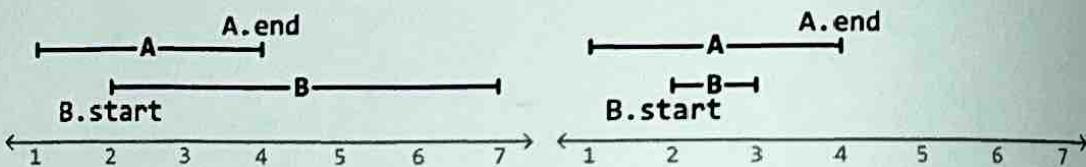
Intuition

We're given two arrays of intervals, each containing non-overlapping intervals. This implies an overlap can only occur between an interval from the first array and an interval from the second array.

Let's start by learning how to identify an overlap between two overlapping intervals.

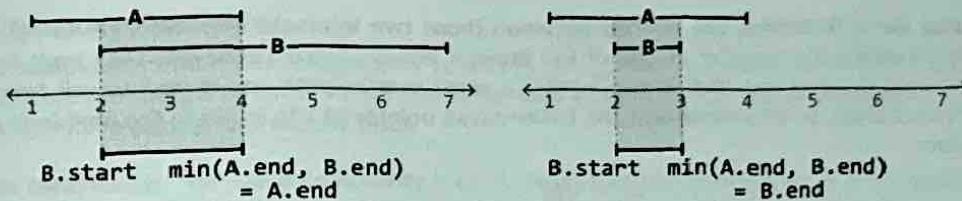
Identifying the overlap between two overlapping intervals

We know from the *Merge Overlapping Intervals* problem that two intervals, A and B, overlap when $A.end \geq B.start$, assuming we know A starts before B. Let's have a look at a couple of examples which each contain two overlapping intervals that match this condition:



To extract the overlap between these two overlapping intervals, we'll need to identify when it starts and ends.

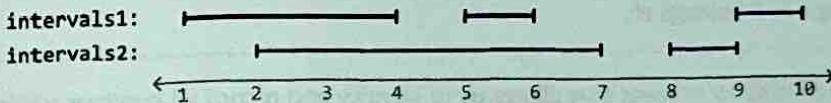
- The overlap starts at the furthest start point, which is always `B.start`.
- The overlap ends at the earliest end point (`min(A.end, B.end)`).



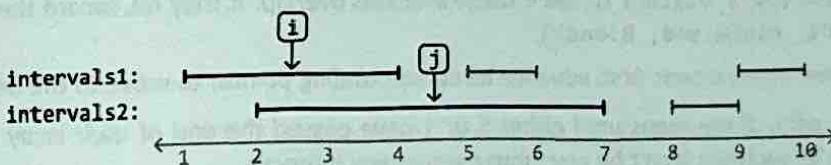
Therefore, when two intervals overlap, their overlap is defined by the range $[B.start, \min(A.end, B.end)]$. Remember that in all these cases, interval A always starts first.

Identifying all overlaps

Now, let's return to the two arrays of intervals. Consider this example:



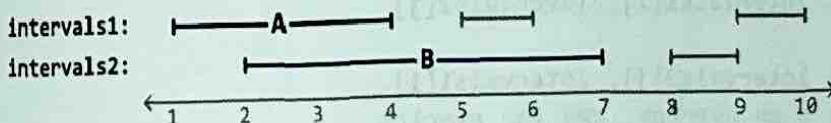
Let's start by considering the first interval from each array:



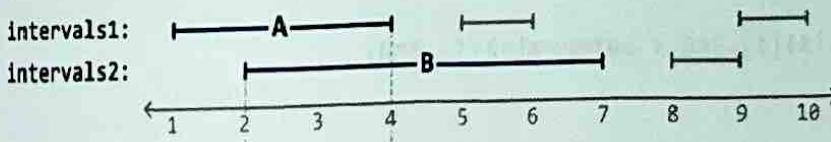
To check if these intervals overlap, we'll need to identify which interval between `intervals1[i]` and `intervals2[j]` starts first, so we can assign that interval as interval A and the other as interval B. The code snippet for this is provided below:

```
# Set A to the interval that starts first and B to the other interval.
if intervals1[i].start <= intervals2[j].start:
    A, B = intervals1[i], intervals2[j]
else:
    A, B = intervals2[j], intervals1[i]
```

In this example, `intervals1[i]` starts first:

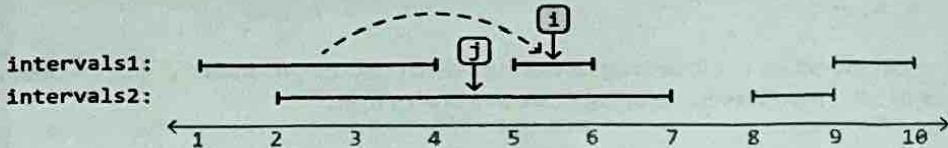


Intervals A and B overlap when $A.end \geq B.start$, which is true here. Since they overlap, let's record their overlap: $[B.start, \min(A.end, B.end)]$:



intersections: B.start $\min(A.end, B.end) = A.end$

Now that we've identified the overlap between those two intervals, let's move on to the next pair by advancing the pointer at one of the interval arrays. Since `intervals1[i]` ends before `intervals2[j]`, we know that `intervals1[i]` won't overlap with any more intervals from the `intervals2` array, so let's increment the `intervals1` pointer (`i`) to move to the next interval in this array:



Note, we use `intervals1[i]` and `intervals2[j]` instead of `A` and `B` since we don't know which interval array `A` or `B` belongs to.

We've now identified a process that allows us to identify and record all overlaps while traversing the arrays of intervals. For the pair of intervals being considered at `i` and `j`:

1. Set `A` as the interval that starts first, and `B` as the other interval.
2. Check if `A.end ≥ B.start` to see if these intervals overlap. If they do, record the overlap as `[B.start, min(A.end, B.end)]`.
3. Whichever interval ends first, advance its corresponding pointer to move to the next interval.

Continue to apply these steps until either `i` or `j` have passed the end of their array. Once this happens, we know there won't be any more overlapping intervals.

Implementation

```
def identify_all_interval_overlaps(  
    intervals1: List[Interval], intervals2: List[Interval]  
) -> List[Interval]:  
    overlaps = []  
    i = j = 0  
    while i < len(intervals1) and j < len(intervals2):  
        # Set A to the interval that starts first and B to the other  
        # interval.  
        if intervals1[i].start <= intervals2[j].start:  
            A, B = intervals1[i], intervals2[j]  
        else:  
            A, B = intervals2[j], intervals1[i]  
        # If there's an overlap, add the overlap.  
        if A.end >= B.start:  
            overlaps.append(Interval(B.start, min(A.end, B.end)))  
        # Advance the pointer associated with the interval that ends  
        # first.  
        if intervals1[i].end < intervals2[j].end:  
            i += 1  
        else:  
            j += 1  
    return overlaps
```

Complexity Analysis

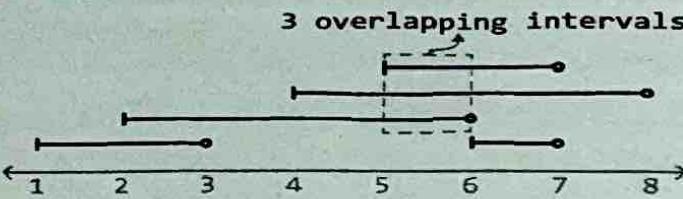
Time complexity: The time complexity of `identify_all_interval_overlaps` is $O(n + m)$ where n and m are the lengths of `intervals1` and `intervals2`, respectively. This is because we traverse each interval in both arrays exactly once.

Space complexity: The space complexity is $O(1)$. Note that the `overlaps` array is not considered because space complexity is only concerned with extra space used and not space taken up by the output.

Largest Overlap of Intervals

Given an array of intervals, determine the maximum number of intervals that overlap at any point. Each interval is half-open, meaning it includes the start point but excludes the end point.

Example:



Input: intervals = [[1, 3], [2, 6], [4, 8], [6, 7], [5, 7]]
Output: 3

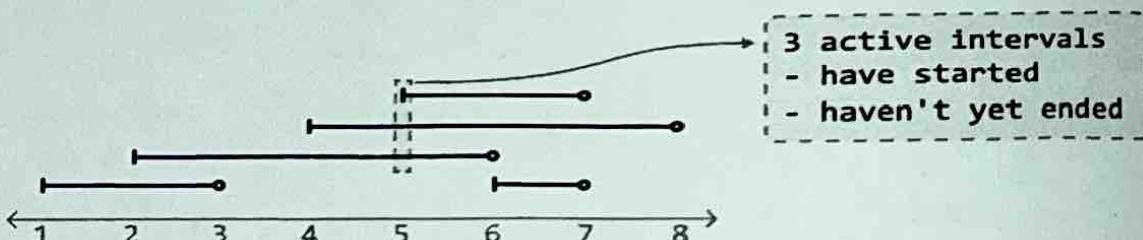
Constraints:

- The input will contain at least one interval.
- For every index i in the array, $\text{intervals}[i].\text{start} < \text{intervals}[i].\text{end}$.

Intuition

Think about what it means when x intervals overlap at a certain point in time. This means at this point, there are x 'active' intervals, where an interval is active if it has started but not ended.

In the example below, we see three active intervals at time 5 (intervals that started at or before this time, and haven't yet ended):

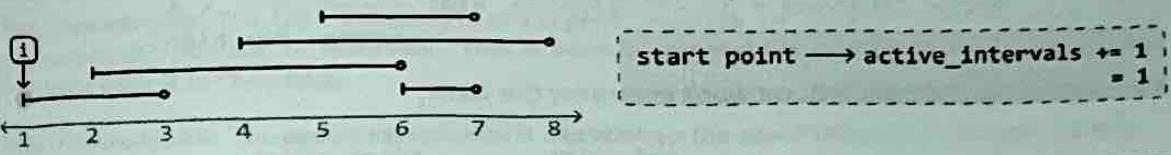


To find the number of active intervals at any point in time, we need to identify when an interval has started and when an interval has ended. The start point of an interval indicates the start of a new active interval, whereas an end point represents an active interval finishing. This suggests an approach which looks at start and end points individually could be useful. Let's explore this idea further.

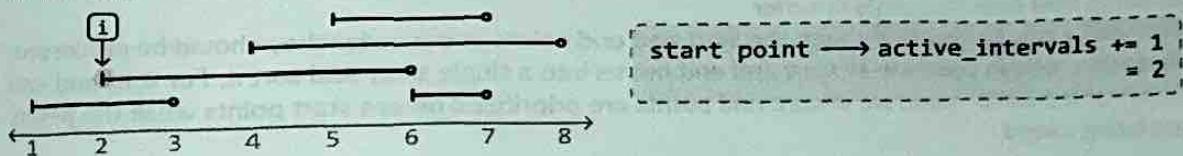
Processing start points and end points

Let's step through each point in the array of intervals in chronological order, and see if we can determine the number of active intervals at each point.

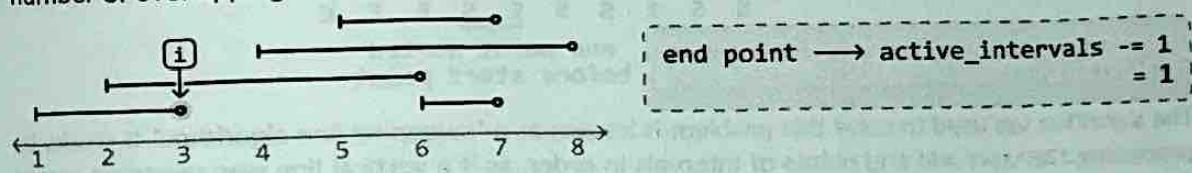
The first point is a start point, indicating the start of the first active interval. So, let's increment our counter:



The next point is another start point. This is the start of the second active interval, so let's increment our counter again. Now, the number of active intervals is 2, which correctly corresponds with the number of overlapping intervals at this point:



The next point is an end point, which means an active interval has just finished. So, let's decrement our counter. Now, the number of active intervals is 1, which also corresponds with the current number of overlapping intervals:



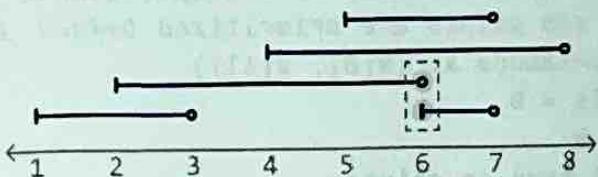
We've now rationalized what we need to do whenever we encounter a start point or end point:

- If we encounter a **start point**: **increment active_intervals**.
- If we encounter an **end point**: **decrement active_intervals**.

Processing the remaining points allows us to attain the number of active intervals at each point. The final answer is obtained by recording the largest value of **active_intervals**.

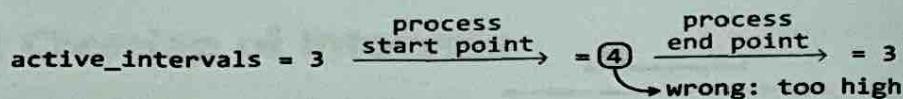
Edge case: processing concurrent start and end points

An edge case to consider is when a start and end point occur simultaneously:

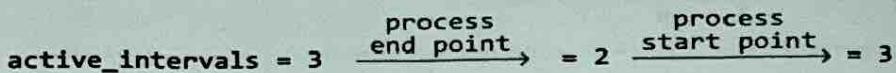


Which point should we process first? Keep in mind the value of **active_intervals** is 3 right before we reach time 6 in the above example.

At time 6, if we process the start point first and increment **active_intervals**, we would update it to 4 first, which is incorrect as there are never 4 active intervals at this moment. This is an issue because our final answer is the largest value of **active_intervals** encountered, which means we'll incorrectly record 4 as the answer.



If we process the end point first, we won't encounter this issue:



Therefore, for start and end points that occur simultaneously, we should process end points before start points.

Iterating over interval points in order

We need a way to iterate through the start and end points in the order they should be processed. To do this, we can combine all start and end points into a single array and sort it. For start and end points of the same value, we ensure end points are prioritized before start points while the points are being sorted.

Let's use 'S' and 'E' to differentiate between start and end points, respectively:

```
intervals = [1, 3] [2, 6], [4, 8] [6, 7] [5, 7]
→ points = [ 1 2 3 4 5 6 6 7 7 8 ]
      S S E S S E S E E
      end point sorted
      before start point
```

The algorithm we used to solve this problem is known as a "sweeping line algorithm." It works by processing the start and end points of intervals in order, as if a vertical line was sweeping across them. This method efficiently handles the dynamic nature of interval overlaps by specifically focusing on start and end points, rather than individual intervals.

Implementation

```
def largest_overlap_of_intervals(intervals: List[Interval]) -> int:
    points = []
    for interval in intervals:
        points.append((interval.start, 'S'))
        points.append((interval.end, 'E'))
    # Sort in chronological order. If multiple points occur at the same
    # time, ensure end points are prioritized before start points.
    points.sort(key=lambda x: (x[0], x[1]))
    active_intervals = 0
    max_overlaps = 0
    for time, point_type in points:
        if point_type == 'S':
            active_intervals += 1
        else:
            active_intervals -= 1
        max_overlaps = max(max_overlaps, active_intervals)
    return max_overlaps
```

Complexity Analysis

Time complexity: The time complexity of `largest_overlap_of_intervals` is $O(n \log(n))$, where n denotes the number of intervals. This is because we sort the `points` array of size $2n$ before iterating over it in $O(n)$ time.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the `points` array.

Prefix Sums

Introduction to Prefix Sums

Imagine keeping track of how much money you spend on takeout meals each day over a period of days.

```
spendings = [ 10 15 20 10 5 ]
             Mon Tue Wed Thu Fri
```

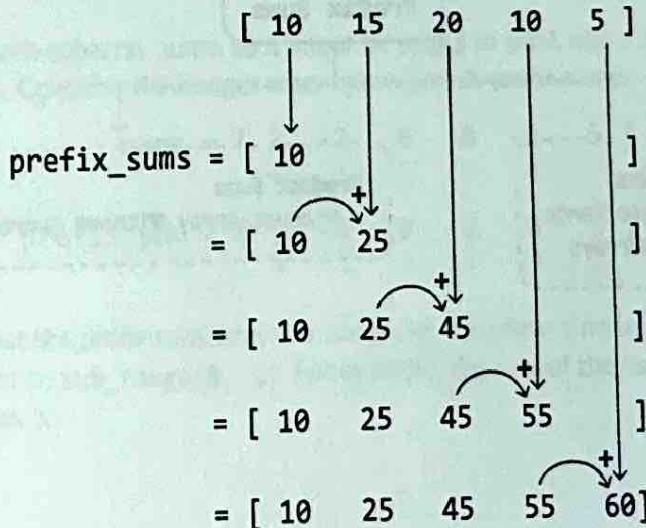
Let's say you want to know the total spent on takeout food up until a particular day. For example, you might like to know that the total you've spent up until Wednesday is \$45 ($\$10 + \$15 + \20). This is information which a prefix sum array can store. For an array of integers, a prefix sum array maintains the running sum of values up to each index in the array.

```
spendings = [ 10 15 20 10 5 ]
             Mon Tue Wed Thu Fri
```



```
prefix_sums = [ 10 25 45 55 60 ]
```

To obtain the prefix sum at each index, we just add the current number from the input array to the prefix sum from the previous index.



In code, the above process looks like this:

```
def compute_prefix_sums(nums):
    # Start by adding the first number to the prefix sums array.
    prefix_sum = [nums[0]]
    # For all remaining indexes, add 'nums[i]' to the cumulative sum
    # from the previous index.
    for i in range(1, len(nums)):
        prefix_sum.append(prefix_sum[-1] + nums[i])
```

As you can see, building a prefix sum array takes $O(n)$ time and $O(n)$ space, where n denotes the length of the array.

Applications of prefix sums

Aside from allowing us to have constant-time access to running sums at any index within an array, prefix sums are commonly used to efficiently determine the sum of subarrays. This application is examined in depth in the problems in this chapter.

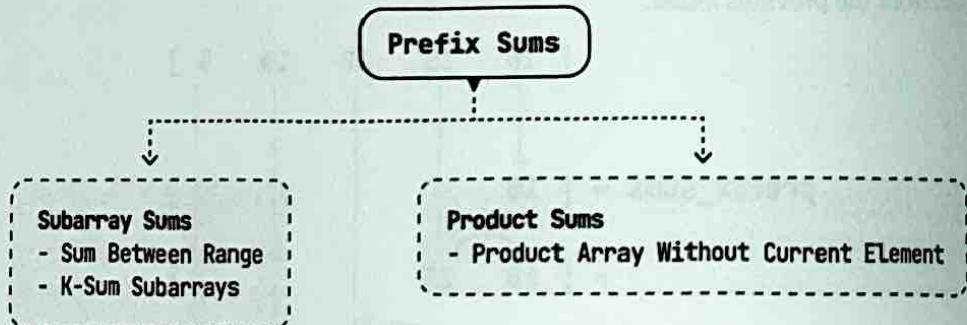
Another interesting variant of prefix sums is prefix products, which populates an array with a running product instead of a running sum. Similar to prefix sums, prefix products provide an efficient way to determine the product of subarrays.

Real-world Example

Financial analysis: As hinted at earlier, a real-world use of prefix sums is for financial analysis, particularly in calculating cumulative earnings or expenses over time.

For instance, consider a company's daily revenue over a month. A prefix sum array can be used to quickly calculate the total revenue for any given period within that month. By precomputing the prefix sums, the company can instantly determine the revenue from day 5 to day 20 without having to sum each day's revenue individually. This is especially useful for generating financial reports, where quick calculations over various periods are necessary to analyze trends.

Chapter Outline



Sum Between Range

Given an integer array, write a function which returns the sum of values between two indexes.

Example:

$\text{sum_range}(0, 3) = [\begin{matrix} 3 & -7 & 6 & \text{sum } = 2 \\ 0 & 1 & 2 & 3 \end{matrix}]$

$\text{sum_range}(2, 4) = [\begin{matrix} 3 & -7 & 6 & 0 & \text{sum } = 4 \\ 0 & 1 & 2 & 3 & 4 \end{matrix}]$

$\text{sum_range}(2, 2) = [\begin{matrix} 3 & -7 & 6 & 0 & -2 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}]$

Input: $\text{nums} = [3, -7, 6, 0, -2, 5]$, $[\text{sum_range}(0, 3), \text{sum_range}(2, 4), \text{sum_range}(2, 2)]$
Output: $[2, 4, 6]$

Constraints:

- nums contains at least one element.
- Each sum_range operation will query a valid range of the input array.

Intuition

We need to code a function $\text{sum_range}(i, j)$, where i and j are the indexes defining the boundaries of the range to be summed up.

A naive solution is to iteratively sum the array values from index i to j , which takes linear time for each call to sum_range . Since we have access to the input array before any calls to sum_range are made, we should consider if any preprocessing can be done to improve the efficiency of sum_range .

This problem deals with subarray sums, so it might be useful to think about how prefix sums can be applied to solve it. Consider the integer array below and its prefix sums:

$\text{nums} = [\begin{matrix} 3 & -7 & 6 & 0 & -2 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}]$

$\text{prefix_sum} = [\begin{matrix} 3 & -4 & 2 & 2 & 0 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}]$

We already notice that the prefix sum array has some use: the prefix sum up to any index j essentially gives the answer to $\text{sum_range}(0, j)$. For example, the sum of the range $[0, 3]$ is just the prefix sum up to index 3:

```

sum_range(0, 3) = prefix_sum[3]:
sum = 2
nums = [ 3 -7 6 0 -2 5 ]
          0 1 2 3 4 5
prefix_sum = [ 3 -4 2 2 0 5 ]
          0 1 2 3 4 5

```

Therefore, when $i == 0$:

```
| sum_range(0, j) = prefix_sum[j]
```

What about when the requested range doesn't start at 0? Let's say we want to find the sum in the range [2, 4]:

```
[ 3 -7 6 0 -2 5 ]
  0 1 2 3 4 5
```

Is there a way to get this using only prefix sums? All prefix sum values are sums for ranges that start at index 0. So, let's see how we could make use of these ranges. Consider the sum of the range [0, 4], which corresponds to `prefix_sum[4]`:

```
sum[0 : 4]
[ 3 -7 6 0 -2 5 ]
  0 1 2 3 4 5
```

The key observation here is that the sum of the range [2, 4] can be obtained by subtracting the sum of the range [0, 1] from the sum above. This can be visualized:

```
sum[0 : 4]
[ 3 -7 6 0 -2 5 ]
  0 1 2 3 4 5
sum[0 : 1] sum[2 : 4]
```

Since the sums of ranges [0, 4] and [0, 1] are both values in our prefix sum array, we can obtain the sum of the range [2, 4] from the following expression: `prefix_sum[4] - prefix_sum[1]`.

Therefore, when $i > 0$:

```
| sum_range(i, j) = prefix_sum[j] - prefix_sum[i - 1]
```

Implementation

```

class SumBetweenRange:
    def __init__(self, nums: List[int]):
        self.prefix_sum = [nums[0]]
        for i in range(1, len(nums)):
            self.prefix_sum.append(self.prefix_sum[-1] + nums[i])

    def sum_range(self, i: int, j: int) -> int:
        if i == 0:
            return self.prefix_sum[j]
        return self.prefix_sum[j] - self.prefix_sum[i - 1]

```

Complexity Analysis

Time complexity: The time complexity of the constructor is $O(n)$, where n denotes the length of the array. This is because we populate a `prefix_sum` array of length n . The time complexity of `sum_range` is $O(1)$.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the `prefix_sum` array.

K-Sum Subarrays

Find the number of subarrays in an integer array that sum to k.

Example:

```
sum = 3 [ 1 2 -1 1 2 ] sum = 3 [ 1 2 -1 1 2 ] sum = 3 [ 1 2 -1 1 2 ]
```

| Input: nums = [1, 2, -1, 1, 2], k = 3

Output: 3

Intuition

The brute force solution to this problem involves iterating through every possible subarray and checking if their sum equals k . It takes $O(n^2)$ time to iterate over all subarrays, and finding the sum of each subarray takes $O(n)$ time, resulting in an overall time complexity of $O(n^3)$, where n denotes the length of the array. This solution is quite inefficient, so let's think of something better.

Since we're working with subarray sums, it's worth considering how prefix sums can be used to solve this problem.

Prefix sums

As described in the *Sum Between Range* problem in this chapter, the sum of a subarray between two indexes, i and j , can be calculated with the following formula:

$$sum[i : j] = prefix_sum[j] - prefix_sum[i - 1]$$

$[1 \ 2 \ -1 \ 1 \ 2] = [1 \ 2 \ -1 \ 1 \ 2] - [1 \ 2 \ -1 \ 1 \ 2]$
i j j i-1

For subarrays which start at the beginning of the array (i.e., when $i == 0$), the formula is just:

In this problem, we already know the sum we're looking for (k), meaning our goal is to find:

- All pairs of i and j such that $\text{prefix_sum}[j] - \text{prefix_sum}[i - 1] == k$ when $i > 0$.
 - All values of j such that $\text{prefix_sum}[j] == k$ when $i == 0$.

We can unify both cases by recognizing that the formula $\text{prefix_sum}[j] == k$ is the same as the formula $\text{prefix_sum}[j] - \text{prefix_sum}[i - 1] == k$ when $\text{prefix_sum}[i - 1]$ equals 0 (i.e., $\text{prefix_sum}[j] - 0 == k$).

One issue with this is when $i == 0$, index $i - 1$ is invalid. To make this unification possible while avoiding the out-of-bounds issue, we can prepend '[0]' to the prefix sums array, making it possible for $\text{prefix_sum}[i - 1]$ to equal 0 when $i - 1 == 0$.

nums = [1 2 -1 1 2]	start sums with 0	prefix sums [0 1 3 2 3 5]
prefix_sums = [1 3 2 3 5] 0 1 2 3 4	↓	[0 1 3 2 3 4 5] 0 1 2 3 4 5

Keep in mind that we should iterate over the array from index 1 because we added this 0 to the start of the prefix sum array.

Here's the code snippet for this approach:

```
def k_sum_subarrays(nums: List[int], k: int) -> int:
    n = len(nums)
    count = 0
    # Populate the prefix sum array, setting its first element to 0.
    prefix_sum = [0]
    for i in range(0, n):
        prefix_sum.append(prefix_sum[-1] + nums[i])
    # Loop through all valid pairs of prefix sum values to find all
    # subarrays that sum to 'k'.
    for j in range(1, n + 1):
        for i in range(1, j + 1):
            if prefix_sum[j] - prefix_sum[i - 1] == k:
                count += 1
    return count
```

This is an improvement on the brute force solution, which reduces the time complexity to $O(n^2)$. Can we optimize this solution further?

Optimization - hash map

An important point is that we don't need to treat both `prefix_sum[j]` and `prefix_sum[i - 1]` as unknowns in the formula. If we know the value of `prefix_sum[j]`, we can find `prefix_sum[i - 1]` using `prefix_sum[i - 1] = prefix_sum[j] - k`.

Therefore, for each prefix sum (`curr_prefix_sum`), we need to find the number of times `curr_prefix_sum - k` previously appeared as a prefix sum before.

This is similar to the problem presented in *Pair Sum - Unsorted* in the Hash Maps and Sets chapter, where we learn a **hash map** is useful for implementing the above idea efficiently. In this context, if we store encountered prefix sum values in a hash map, we can check if `curr_prefix_sum - k` was encountered before in constant time.

Note, it's also important to track the frequency of each prefix sum we encounter using the hash map, as the same prefix sum may appear multiple times.

Let's try using a hash map (`prefix_sum_map`) on the example below with $k = 3$. Initialize `prefix_sum_map` with one zero for the same reason we prepended 0 to the prefix sum array in the $O(n^2)$ solution discussed earlier:

$k = 3$	\downarrow	prefix_sum_map				
[1 2 -1 1 2]		<table border="1"> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>sum</td> <td>freq</td> </tr> </table>	0	1	sum	freq
0	1					
sum	freq					

We're using a hash map to keep track of prefix sums, we no longer need a separate array to store each individual prefix sum.

Initially, the prefix sum (`curr_prefix_sum`) is equal to 1. Its complement, -2, is not in the hash map as illustrated below. So, we continue:

$k = 3$	\downarrow	prefix_sum_map				
[1 2 -1 1 2]		<table border="1"> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>sum</td> <td>freq</td> </tr> </table>	0	1	sum	freq
0	1					
sum	freq					

$curr_prefix_sum - k = -2$ (not in hash map)
→ continue

Store the (`curr_prefix_sum`, `freq`) pair (1, 1) in the hash map before moving to the next prefix sum.

The next `curr_prefix_sum` value is 3 (1 + 2). Its complement, 0, exists in the hash map with a frequency of 1. This means we found 1 subarray of sum k . So, we add 1 to our count:

$k = 3$	\downarrow	prefix_sum_map						
[1 2 -1 1 2]		<table border="1"> <tr> <td>1</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>sum</td> <td>freq</td> </tr> </table>	1	1	0	1	sum	freq
1	1							
0	1							
sum	freq							

$curr_prefix_sum - k = 0$ (in hash map)
→ count += prefix_sum_map[0]
+= 1

Store the (`curr_prefix_sum`, `freq`) pair (3, 1) in the hash map before moving on to the next value.

We now have a strategy for processing each value in the array:

1. Update `curr_prefix_sum` by adding the current value of the array to it.
2. If `curr_prefix_sum - k` exists in the hash map, add its frequency (`prefix_sum_map[curr_prefix_sum - k]`) to count.
3. Add (`curr_prefix_sum`, `freq`) to the hash map. If the key is already present, increase its frequency; if not, set it to 1.

Repeat this process for the rest of the array:

$k = 3$	\downarrow	prefix_sum_map								
[1 2 -1 1 2]		<table border="1"> <tr> <td>3</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>sum</td> <td>freq</td> </tr> </table>	3	1	1	1	0	1	sum	freq
3	1									
1	1									
0	1									
sum	freq									

$curr_prefix_sum - k = -1$ (not in hash map)
→ continue

prefix_sum_map	
sum	freq
2	1
3	1
1	1
0	1

$\text{curr_prefix_sum} - k = 0$ (in hash map)
 $\rightarrow \text{count} += \text{prefix_sum_map}[0]$
 $+ = 1$

prefix_sum_map	
sum	freq
2	1
3	2
1	1
0	1

$\text{curr_prefix_sum} - k = 2$ (in hash map)
 $\rightarrow \text{count} += \text{prefix_sum_map}[2]$
 $+ = 1$

Once we've processed all the prefix sum values, we return count, which stores the number of subarrays that sum to k.

Implementation

```
def k_sum_subarrays_optimized(nums: List[int], k: int) -> int:
    count = 0
    # Initialize the map with 0 to handle subarrays that sum to 'k'
    # from the start of the array.
    prefix_sum_map = {0: 1}
    curr_prefix_sum = 0
    for num in nums:
        # Update the running prefix sum by adding the current number.
        curr_prefix_sum += num
        # If a subarray with sum 'k' exists, increment 'count' by the
        # number of times it has been found.
        if curr_prefix_sum - k in prefix_sum_map:
            count += prefix_sum_map[curr_prefix_sum - k]
        # Update the frequency of 'curr_prefix_sum' in the hash map.
        freq = prefix_sum_map.get(curr_prefix_sum, 0)
        prefix_sum_map[curr_prefix_sum] = freq + 1
    return count
```

Complexity Analysis

Time complexity: The time complexity of k_sum_subarrays_optimized is $O(n)$ because we iterate through each value in the nums array.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the hash map.

Product Array Without Current Element

Given an array of integers, return an array `res` so that `res[i]` is equal to the product of all the elements of the input array except `nums[i]` itself.

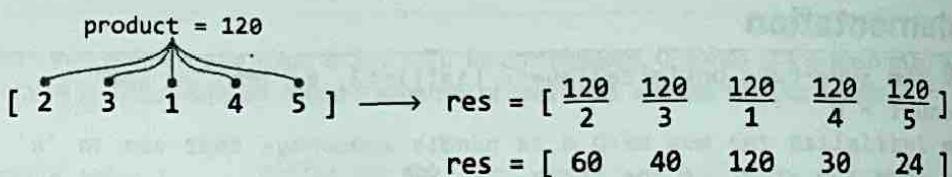
Example:

Input: `nums` = [2, 3, 1, 4, 5]
Output: [60, 40, 120, 30, 24]

Explanation: The output value at index 0 is the product of all numbers except `nums[0]` ($3 \cdot 1 \cdot 4 \cdot 5 = 60$). The same logic applies to the rest of the output.

Intuition

The straightforward solution to this problem is to find the total product of the array and divide it by each of the values in `nums` individually to get the output array:



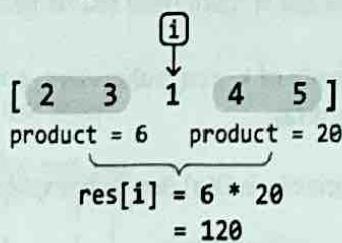
This approach allows us to solve the problem in linear time and constant space. However, a potential follow-up question by an interviewer is: what if we can't use division? Let's explore a solution to this.

Avoiding division

A brute force approach involves calculating the output value for each index one by one. This would take $O(n)$ time per index, leading to an overall time complexity of $O(n^2)$, where n denotes the length of the array. This is inefficient, so let's look at other approaches.

An important insight is that the output for any given index can be determined by multiplying two things:

1. The product of all numbers to the left of the index.
2. The product of all numbers to the right of the index.



Why is this helpful? If we have precomputed the products of all values to the left and right of each index, we can quickly calculate the output for each index. More specifically, we would need two arrays that contain the left and right products of each index, respectively:

- `left_products`: an array where `left_products[i]` is the product of all values to the left of `i`.

- `right_products`: an array where `right_products[i]` is the product of all values to the right of `i`.

To obtain the `left_products` array, we need to keep track of a cumulative product of all elements we encounter as we move from left to right. The value of this product at a specific index should represent the product of all values to its left. The same is true of the `right_products` array, but the cumulative products start from the right. Once we have these arrays, multiplying the left and right product values at each index gives us the output value of that index.

```

nums = [ 2   3   1   4   5 ]
left_products = [ 1   2   6   6   24 ]
                     *   *   *   *
right_products = [ 60  20  20  5   1 ]
                     "   "   "   "
res = [ 60  40  120  30  24 ]

```

Since the left and right product arrays are formed through cumulative multiplication, this leads us to the concept of prefix products.

Prefix products

Prefix products are created in the same way as a prefix sum array, with two key differences:

1. Instead of cumulative addition, we use cumulative multiplication.
2. We initialize the prefix product array with 1 instead of 0, to avoid multiplying the cumulative products by 0.

Let's try creating the `left_products` array, initializing it with 1 at index 0:

```

nums = [ 2   3   1   4   5 ]
left_products = [ 1           ]

```

For each subsequent index in the `left_products` array, we calculate its value by multiplying the running product by the previous value in the `nums` array:

```

nums = [ 2   3   1   4   5 ]
left_products = [ 1   2           ]

```

```

nums = [ 2   3   1   4   5 ]
left_products = [ 1   2   6           ]

```

```

nums = [ 2   3   1   4   5 ]
left_products = [ 1   2   6   6           ]

```

```
nums = [ 2      3      1      4      5 ]  
left_products = [ 1      2      6      6      24 ]
```

The same can be done for the `right_products` array, but starting on the right and moving leftward:

```
nums = [ 2      3      1      4      5 ]  
right_products = [           1 ]
```

```
nums = [ 2      3      1      4      5 ]  
right_products = [           5      1 ]
```

```
nums = [ 2      3      1      4      5 ]  
right_products = [           20     5      1 ]
```

```
nums = [ 2      3      1      4      5 ]  
right_products = [           20     20     5      1 ]
```

```
nums = [ 2      3      1      4      5 ]  
right_products = [ 60     20     20     5      1 ]
```

Once both arrays are populated, we can compute each value of the output array, where `res[i]` is equal to the product of `left_products[i]` and `right_products[i]`, as previously demonstrated.

Reducing space

We have successfully found a solution that doesn't involve division and runs in linear time. However, this solution takes up linear space due to the left and right product arrays. Can we compute the output array in place without taking up extra space?

An important thing to realize is that we don't necessarily need to create the left and right product arrays to populate the output array. Instead, we can directly compute and store the left and right products in the output array as we calculate them.

This can be done in two steps:

1. First, populate the output array (`res`) the same way we populated `left_products`. This prepares the output array to be multiplied by the right products:

```
nums = [ 2      3      1      4      5 ]  
left_products = [ 1      2      6      6      24 ]  
res = [ 1      2      6      6      24 ]
```

2. Then, instead of populating a `right_products` array, we directly multiply the running product from the right (`right_product`) into the output array:

<pre>nums = [2 3 1 4 5] res = [1 2 6 6 24]</pre>	$i \downarrow$ $\text{right_product} = 1$ $\text{res}[1] = \text{res}[1] * \text{right_product}$ $= 24 * 1$ $= 24$ $\text{right_product} = \text{right_product} * \text{nums}[1]$ $= 1 * 5$ $= 5$
<pre>nums = [2 3 1 4 5] res = [1 2 6 30 24]</pre>	$i \downarrow$ $\text{res}[1] = \text{res}[1] * \text{right_product}$ $= 6 * 5$ $= 30$ $\text{right_product} = \text{right_product} * \text{nums}[1]$ $= 5 * 4$ $= 20$
<pre>nums = [2 3 1 4 5] res = [1 2 120 30 24]</pre>	$i \downarrow$ $\text{res}[1] = \text{res}[1] * \text{right_product}$ $= 6 * 20$ $= 120$ $\text{right_product} = \text{right_product} * \text{nums}[1]$ $= 20 * 1$ $= 20$
<pre>nums = [2 3 1 4 5] res = [1 40 120 30 24]</pre>	$i \downarrow$ $\text{res}[1] = \text{res}[1] * \text{right_product}$ $= 2 * 20$ $= 40$ $\text{right_product} = \text{right_product} * \text{nums}[1]$ $= 20 * 3$ $= 60$
<pre>nums = [2 3 1 4 5] res = [60 40 120 30 24]</pre>	$i \downarrow$ $\text{res}[1] = \text{res}[1] * \text{right_product}$ $= 1 * 60$ $= 60$ $\text{right_product} = \text{right_product} * \text{nums}[1]$ $= 60 * 2$ $= 120$

Implementation

```
def product_array_without_current_element(nums: List[int]) -> List[int]:
    n = len(nums)
    res = [1] * n
    # Populate the output with the running left product.
```

```
for i in range(1, n):
    res[i] = res[i - 1] * nums[i - 1]
    # Multiply the output with the running right product, from right to
    # left.
    right_product = 1
    for i in range(n - 1, -1, -1):
        res[i] *= right_product
        right_product *= nums[i]
return res
```

Complexity Analysis

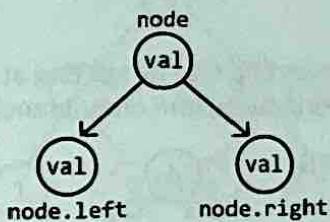
Time complexity: The time complexity of `product_array_without_current_element` is $O(n)$ because we iterate over the `nums` array twice.

Space complexity: The space complexity is $O(1)$. The `res` array is not included in the space complexity analysis.

Trees

Introduction to Trees

A tree is a hierarchical data structure composed of nodes, where each node connects to one or more child nodes. Each node in a tree contains the data it stores (`val`) and references to its child nodes. The most common type of tree is a **binary tree**, in which each node connects to up to two children: a left child and a right child.



Below is the implementation of the `TreeNode` class:

```

class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
  
```

Terminology:

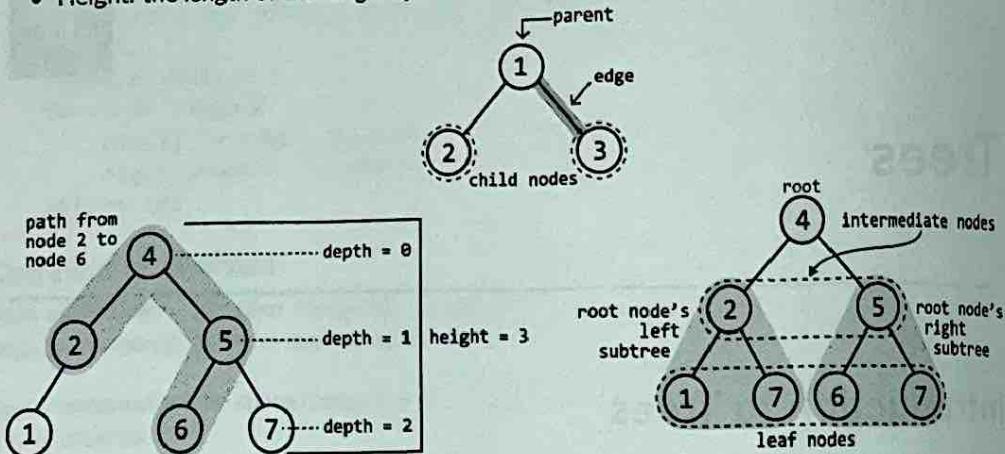
- Parent: a node with one or more children.
- Child: a node that has a parent.
- Subtree: a tree formed by a node and its descendants.
- Path: a single, continuous sequence of nodes connected by edges.
- Depth: the number of edges from the root to a given node.

Attributes of a tree:

- Root: the topmost node of the tree and the only node without a parent.
- Intermediate node: a node with a parent node and at least one child.
- Leaf: a node with no children.
- Edge: the connection between two nodes. Trees usually have directed edges, meaning the

edges only point from parent to child.

- Height: the length of the longest path from the root to a leaf.¹

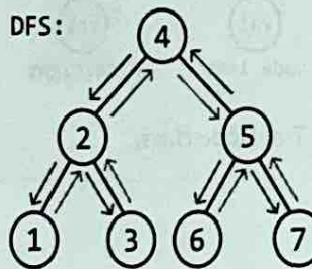


In the following discussions, we primarily focus on binary trees.

Tree Traversals

Depth-first search (DFS)

DFS is a method for exploring all nodes of a tree by starting at the root and moving as far down a branch as possible, before backtracking to explore other branches.



DFS is typically implemented recursively, following a structure similar to the code snippet below:

```
def dfs(node: TreeNode):  
    if node is None:  
        return  
    process(node) # Process the current node.  
    dfs(node.left) # Traverse the left subtree.  
    dfs(node.right) # Traverse the right subtree.
```

The above recursive implementation follows the order of preorder traversal. Two other common DFS traversal techniques include inorder traversal and postorder traversal. Here's how they differ:

¹ Some sources may define the height of a tree differently. In this book, we use a definition that makes designing recursive algorithms more intuitive. Here, the height of a tree with just one node is considered 1.

Preorder traversal	Inorder traversal	Postorder traversal
- process(node)	- dfs(node.left)	- dfs(node.left)
- dfs(node.left)	- process(node)	- dfs(node.right)
- dfs(node.right)	- dfs(node.right)	- process(node)

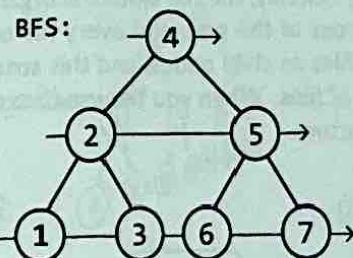
DFS has many use cases and is the most common choice for tree traversal.

- Preorder traversal is the most common type of DFS traversal. It's also used when we need to process the root node of each subtree before its children.
- Inorder traversal is used when we want to process the nodes of a tree from left to right.
- Postorder traversal is the least frequently used traversal method, but is important when each node's subtrees must be processed before their root.

The problems in this chapter explore the use cases of DFS in more detail, providing practical examples of when and how to apply these traversal methods.

Breadth-first search (BFS)

BFS traverses the nodes of a tree level by level. It processes the nodes at the present level before moving on to nodes at the next depth level.



BFS is typically implemented iteratively using a queue, and the reason for this will become clear as we explore the problems in this chapter. The basic structure of BFS is reflected in the following code snippet:

```

def bfs(root: TreeNode):
    if root is None:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        process(node) # Process the current node.
        if node.left:
            # Add the left child to the queue.
            queue.append(node.left)
        if node.right:
            # Add the right child to the queue.
            queue.append(node.right)
  
```

BFS is commonly used to find the shortest path to a specific destination in a tree, or to process the tree level by level. When it's important to know the specific level of each node during traversal, we use a variant of BFS called level-order traversal, which is discussed in detail in this chapter.

Complexity breakdown

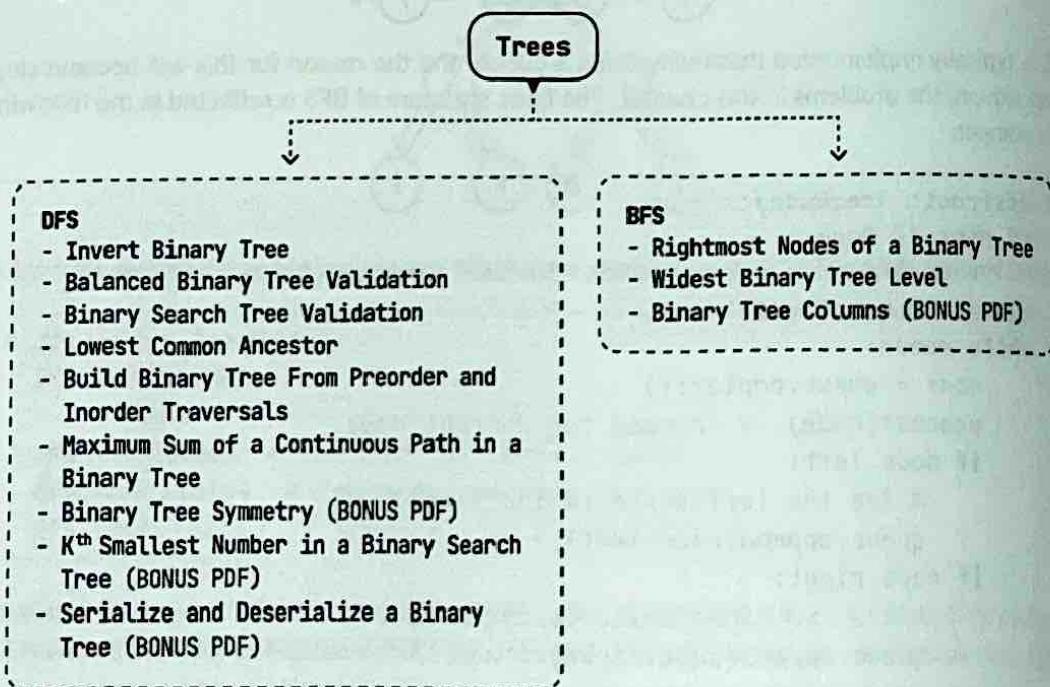
Below, n denotes the number of nodes in the tree, and h denotes the height of the tree.

Operation	Time	Space	Description
DFS	$O(n)$	$O(h)$	DFS visits each node, resulting in an $O(n)$ time complexity. The space complexity is determined by the maximum depth of the recursive call stack, which can be as deep as the height of the tree, h . In the worst case, the height of the tree is n . In a balanced tree whose height is minimized, the height of the tree is approximately $\log(n)$.
BFS	$O(n)$	$O(n)$	BFS visits each node, resulting in an $O(n)$ time complexity. The space complexity is determined by the maximum number of nodes stored in the queue at any time. In the worst case, the queue can store the entire bottom level of the tree, which could contain around $n/2$ nodes.

Real-world Example

File systems: In many operating systems, the file system is organized as a hierarchical tree structure. The root directory is the root of the tree, and every file or folder in the system is a node. Folders can have subfolders or files as child nodes, and this structure allows for efficient organization, navigation, and retrieval of files. When you browse through folders on a computer, you're essentially navigating a tree structure.

Chapter Outline

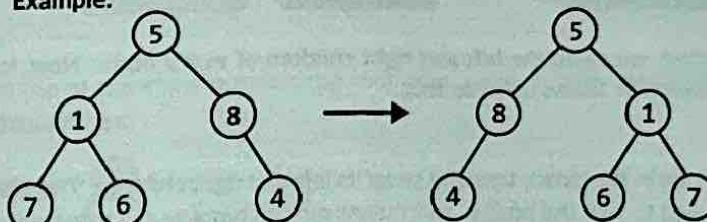


Note that some of the problems listed under DFS can be solved using BFS or other traversal algorithms, too. The same applies to the problems under BFS.

Invert Binary Tree

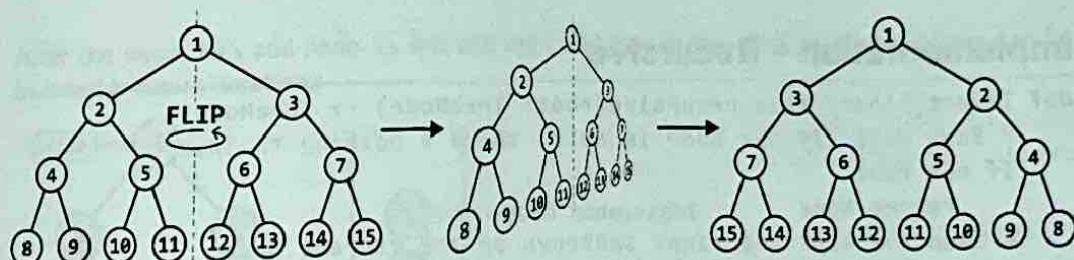
Invert a binary tree and return its root. When a binary tree is inverted, it becomes the mirror image of itself.

Example:

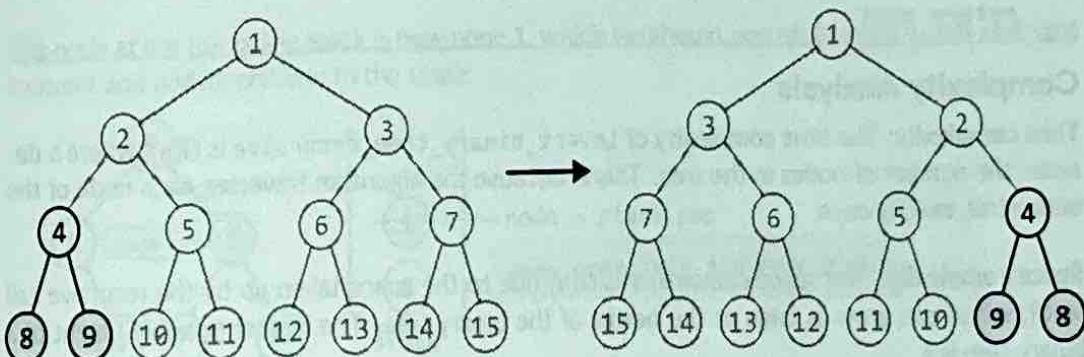


Intuition – Recursive

To invert a binary tree is to essentially 'flip' the tree along a vertical axis, as visualized below:

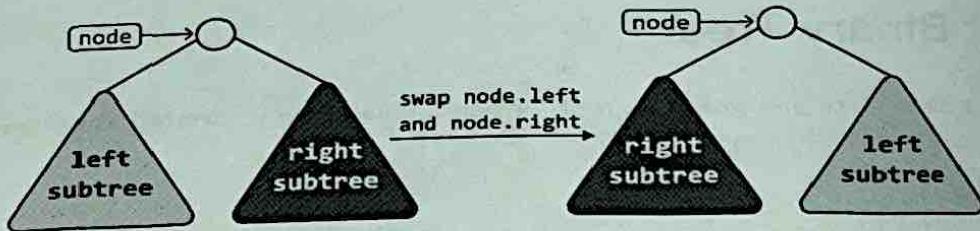


To understand how we invert a binary tree algorithmically, let's focus on how the positions of nodes change after the inversion. For example, pay attention to nodes 8 and 9 in the above tree. After the tree is inverted, we see that they've swapped places relative to their parent node:



The key observation is that this swap of the left and right child happens to each node in the binary tree during inversion.

It's also important to note that when a left child node moves to the right, all nodes under it move as well, and the same happens when a right child moves to the left. In other words, we swap the left subtree and the right subtree of each node. This indicates we're not just swapping the left and right node values, but the nodes themselves.



Therefore, to invert a binary tree, we swap the left and right children of every node. Now, let's explore a tree traversal algorithm that allows us to do this.

Depth-first search

Our strategy is to visit every node in the binary tree and swap its left and right children. There's no particular order in which we need to visit the nodes. This means we can employ any tree traversal algorithm, as long as every node is visited.

Let's tackle this problem recursively using DFS. After swapping the left and right children of the root node, we recursively call our invert function on the left and right children to invert their subtrees as well.

Implementation – Recursive

```
def invert_binary_tree_recursive(root: TreeNode) -> TreeNode:
    # Base case: If the node is null, there's nothing to invert.
    if not root:
        return None
    # Swap the left and right subtrees of the current node.
    root.left, root.right = root.right, root.left
    # Recursively invert the left and right subtrees.
    invert_binary_tree_recursive(root.left)
    invert_binary_tree_recursive(root.right)
    return root
```

Complexity Analysis

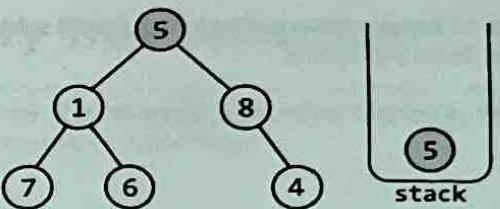
Time complexity: The time complexity of `invert_binary_tree_recursive` is $O(n)$, where n denotes the number of nodes in the tree. This is because the algorithm traverses each node of the binary tree exactly once.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the recursive call stack, which can grow as large as the height of the binary tree. The largest possible height of a binary tree is n .

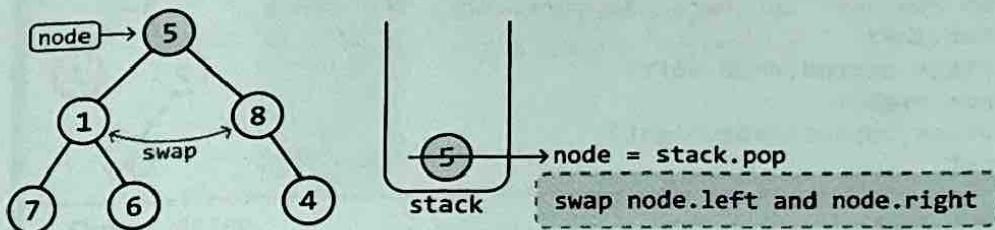
Intuition – Iterative

As a starting point, we can try developing an iterative DFS solution by using the recursive DFS solution as a reference. In the recursive solution, each time a call is made to the left or right child, it's added to a recursive call stack.

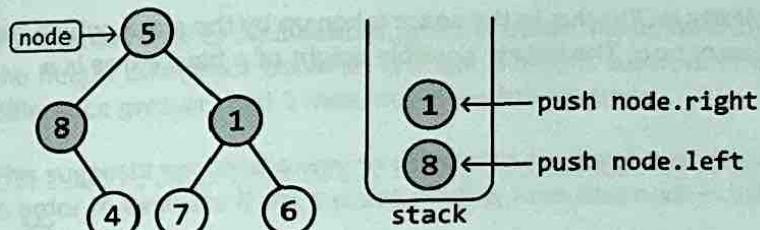
Ultimately, the recursive call stack is a stack, which means we can use a stack to mimic the recursive approach. Let's try using a stack on the following tree. Start by adding the root node to the stack;



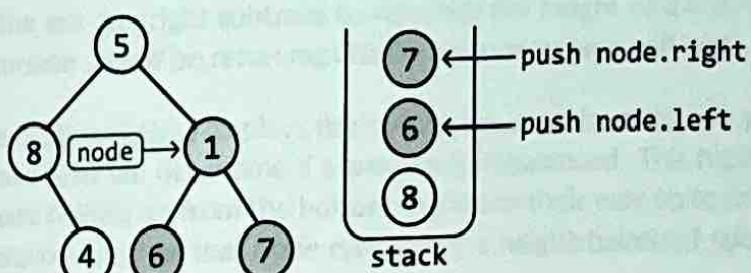
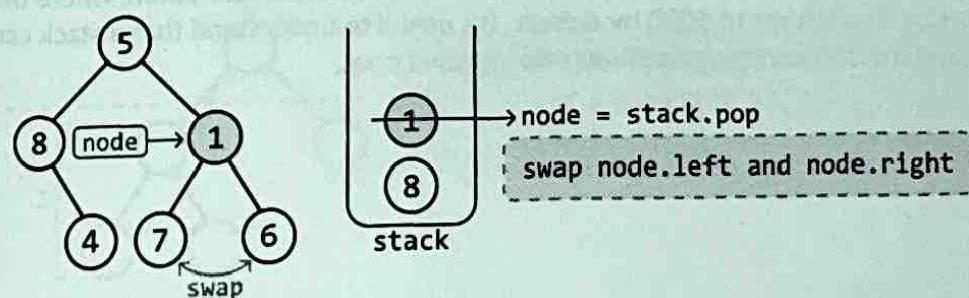
The top of the stack contains the root node, node 5. Let's pop it off so we can swap its left and right subtrees:



After the swap, let's add node 5's left and right children to the stack so their subtrees can be inverted in future iterations:



The node at the top of the stack is now node 1, which we should pop off to swap its left and right subtrees and add its children to the stack:



We repeat the above process of iteratively swapping left and right subtrees until the stack is empty, indicating that all nodes have been processed.

```
def invert_binary_tree_iterative(root: TreeNode) -> TreeNode:
    if not root:
        return None
    stack = [root]
    while stack:
        node = stack.pop()
        # Swap the left and right subtrees of the current node.
        node.left, node.right = node.right, node.left
        # Push the left and right subtrees onto the stack.
        if node.left:
            stack.append(node.left)
        if node.right:
            stack.append(node.right)
    return root
```

Complexity Analysis

Time complexity: The time complexity of `invert_binary_tree_iterative` is $O(n)$ because it processes each node in the binary tree exactly once.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the stack, which can grow as large as the height of the binary tree. The largest possible height of a binary tree is n .

Interview Tip

Tip: Use a stack to convert a recursive solution to an iterative one.

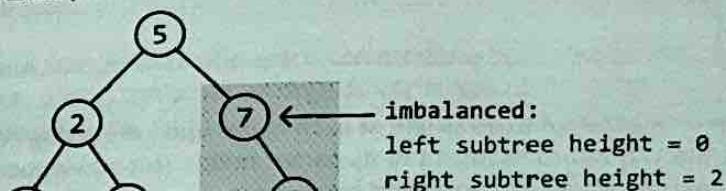


A common follow-up an interviewer may ask is to solve a problem iteratively rather than recursively. This approach is particularly useful when the tree's height can be large, and we want to avoid potential stack overflow errors. For example, this could happen in Python, where the maximum recursion depth is set to 1000 by default. It's useful to understand that a stack can often be used to transform recursive solutions into iterative ones.

Balanced Binary Tree Validation

Determine if a binary tree is height-balanced, meaning no node's left subtree and right subtree have a height difference greater than 1.

Example:



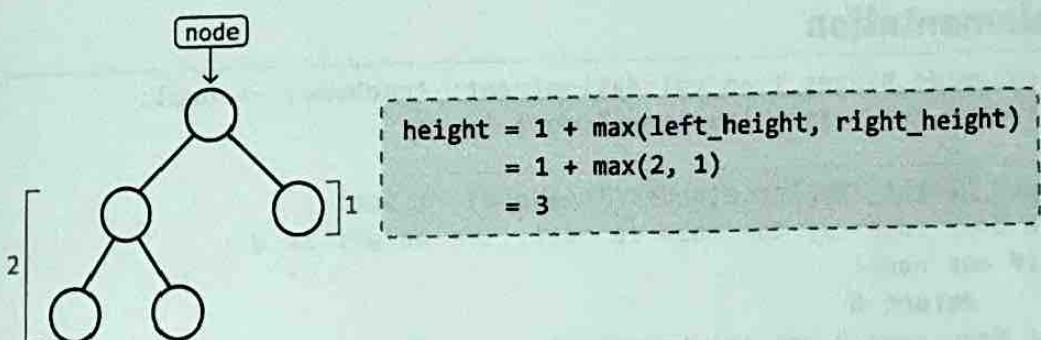
| Output: False

Intuition

For a binary tree to be balanced, all its subtrees would need to be balanced too. This implies that the height difference between the left and right subtrees of each node should be at most 1. A difference greater than 1 indicates a height imbalance.

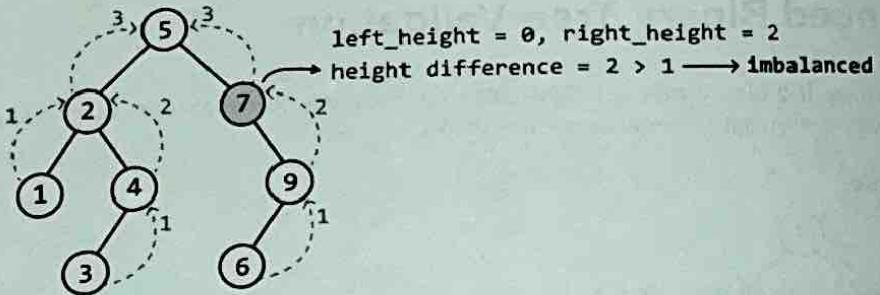
This suggests we need a way to determine the heights of the left and right subtrees at each node in order to evaluate if the subtree rooting from that node is balanced or not.

A key insight is that the height of a tree is equal to the depth of its deepest subtree, plus 1, to include the tree's root node.



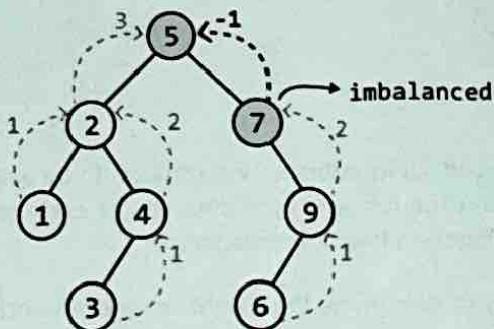
The above formula reveals a recursive relationship, where we can recursively determine the heights of the left and right subtrees to calculate the height of the current subtree. The base case of the recursion would be returning 0 upon encountering a null node, since they have a height of 0.

The diagram below displays the heights returned from the left and right children of each node, and shows how we determine if a subtree is imbalanced. This highlights the recursive process, where values bubble up from the bottom and make their way up to the root node. At each node, we also evaluate whether that node represents a height-balanced subtree. This reveals an imbalance at node 7:



However, there's a flaw in only returning the subtree's height at each node: upon detecting node 7 is imbalanced, all we did about this was return its height to its parent node. This consequently means its parent node (node 5) could be mistakenly considered balanced.

An important thing to remember is that if one subtree is imbalanced, the entire tree is considered imbalanced. This means node 5 should also be marked as imbalanced. We can fix this by returning -1 upon encountering an imbalanced node, essentially informing parent nodes of this imbalance:



To finalize our answer, we return false if the root node of the binary tree returns -1, and true otherwise.

Implementation

```
def balanced_binary_tree_validation(root: TreeNode) -> bool:
    return get_height_imbalance(root) != -1

def get_height_imbalance(node: TreeNode) -> int:
    # Base case: if the node is null, its height is 0.
    if not node:
        return 0
    # Recursively get the height of the left and right subtrees. If
    # either subtree is imbalanced, propagate -1 up the tree.
    left_height = get_height_imbalance(node.left)
    right_height = get_height_imbalance(node.right)
    if left_height == -1 or right_height == -1:
        return -1
    # If the current node's subtree is imbalanced
    # (height difference > 1), return -1.
    if abs(left_height - right_height) > 1:
        return -1
```

```
# Return the height of the current subtree.  
return 1 + max(left_height, right_height)
```

Complexity Analysis

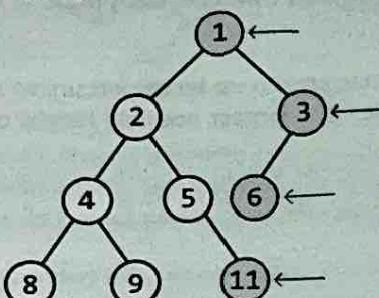
Time complexity: The time complexity of `balanced_binary_tree_validation` is $O(n)$, where n denotes the number of nodes in the tree. This is because it recursively traverses each node of the tree once.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the recursive call stack, which can grow as large as the height of the binary tree. The largest possible height of a binary tree is n .

Rightmost Nodes of a Binary Tree

Return an array containing the values of the rightmost nodes at each level of a binary tree.

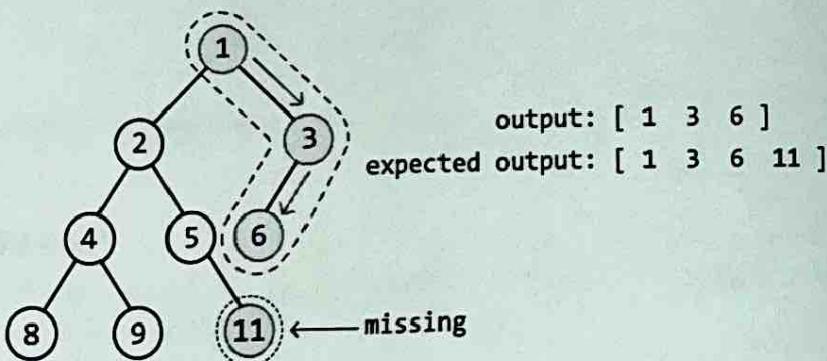
Example:



Output: [1, 3, 6, 11]

Intuition

At first glance, the solution to this problem might seem as simple as traversing the rightmost branch of the tree until we reach a leaf node. But this doesn't work. Why not? Consider the tree below. We see that traversing just the rightmost branch results in missing the rightmost node at the fourth level of the tree:

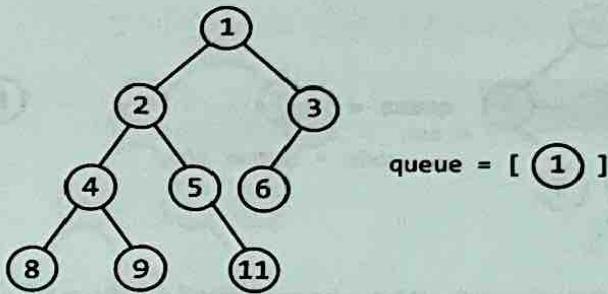


This means we need to consider the entire tree to attain the correct output, and not just a single branch. What would be useful is a way to traverse the tree level by level, allowing us to identify and retrieve the last (i.e., rightmost) node at each level.

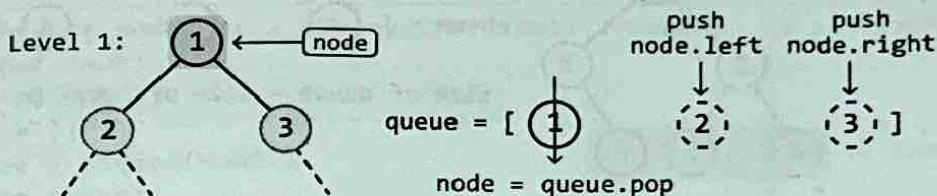
We know BFS traverses nodes level by level. However, standard BFS doesn't provide explicit markers for when one level ends and another begins. In contrast, there is a type of BFS traversal that allows us to process one level at a time. This algorithm is called level-order traversal.

Level-order traversal

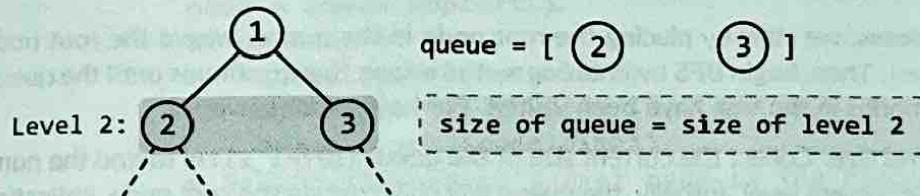
The core idea of level-order traversal is that at any level of the tree, the children of the nodes at that level comprise the next level. This means the children of level 1's nodes make up level 2, and likewise, the children of level 2's nodes make up level 3, and so on. To see how this works, consider the binary tree below, with the BFS queue initialized with the tree's root node:



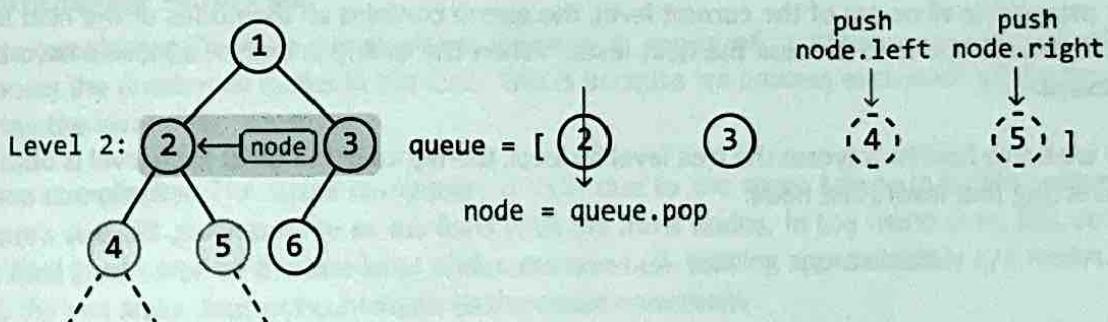
We know level 1 consists of only the root node. So, the children of the root make up the nodes of the second level. Let's pop the root node off and then add its children to the queue:

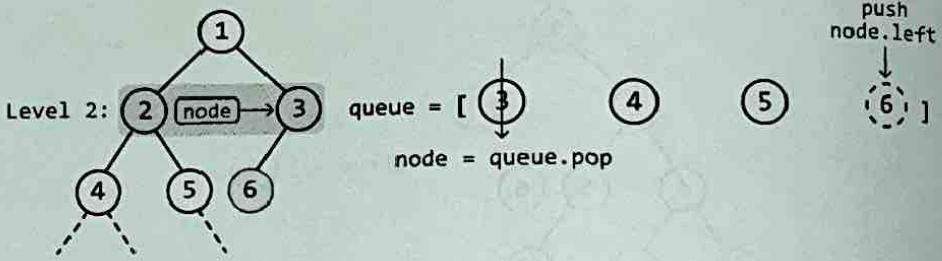


Since we've removed the only level 1 node from the queue and added its children, the queue now only contains the nodes of level 2. Therefore, the size of the queue currently corresponds to the number of nodes in level 2.

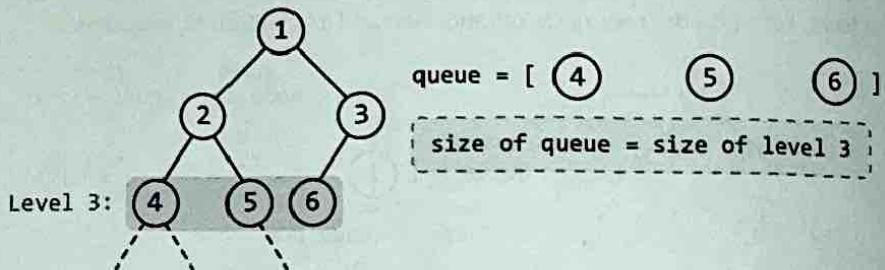


The current queue size is 2, indicating the second level has 2 nodes. So, let's pop off the next 2 nodes in the queue and add their children to the queue:





After the two nodes on the second level have been popped off the queue, the remaining nodes in the queue represent the third level.



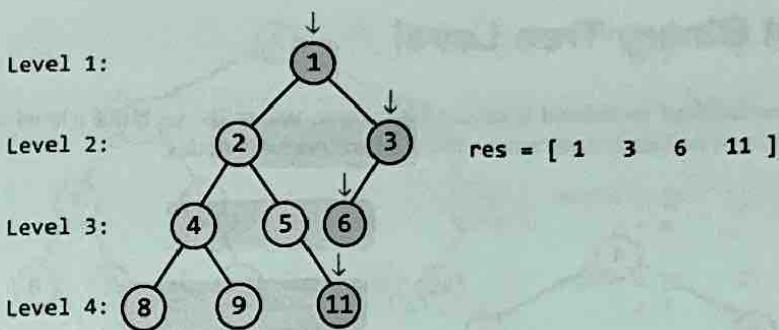
The size of the queue is 3, meaning that to process the third level, we must process the next 3 nodes.

To summarize this process, we start by placing the root node in the queue, where the root node represents the first level. Then, begin BFS by entering a while-loop that continues until the queue is empty, meaning all nodes in the tree have been visited. For level-order traversal:

- 1. Determine the level size:** Collect the current size of the queue (`level_size`) to find the number of nodes in the current level. Initially, the queue will only contain the root node, indicating the first level is of size 1.
- 2. Process the current level:** For each node at this level, pop it from the queue and add its children to the queue.

After processing all nodes of the current level, the queue contains all the nodes of the next level. Repeat steps 1 and 2 to process the next level. When the queue is empty, all levels have been processed.

Once we know how to traverse the tree level by level, the rightmost node at each level is obtained by collecting that level's last node:



Implementation

```

def rightmost_nodes_of_a_binary_tree(root: TreeNode) -> List[int]:
    if not root:
        return []
    res = []
    queue = deque([root])
    while queue:
        level_size = len(queue)
        # Add all the non-null child nodes of the current level to the
        # queue.
        for i in range(level_size):
            node = queue.popleft()
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
            # Record this level's last node to the result array.
            if i == level_size - 1:
                res.append(node.val)
    return res

```

Complexity Analysis

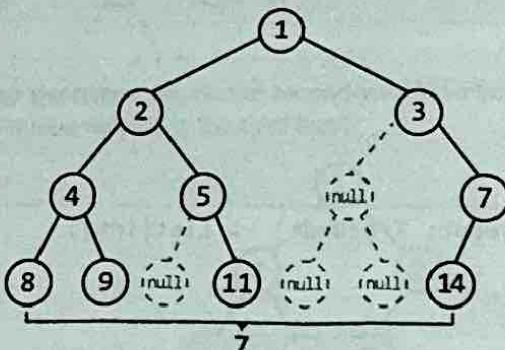
Time complexity: The time complexity of `rightmost_nodes_of_a_binary_tree` is $O(n)$, where n denotes the number of nodes in the tree. This is because we process each node of the tree once during the level-order traversal.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the queue. The queue's size will grow as large as the level with the most nodes. In the worst case, this occurs at the final level when all the last-level nodes are non-null, totaling approximately $n/2$ nodes. Note that the `res` array does not contribute to the space complexity.

Widest Binary Tree Level

Return the width of the widest level in a binary tree, where the width of a level is defined as the distance between its leftmost and rightmost non-null nodes.

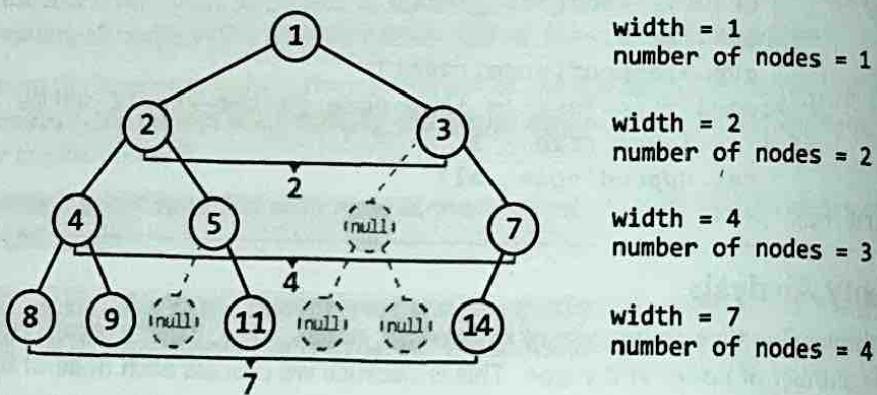
Example:



| Output: 7

Intuition

Let's first understand how width is defined in a binary tree. An important distinction to make is that the width of a level is not necessarily equivalent to the number of nodes in that level.

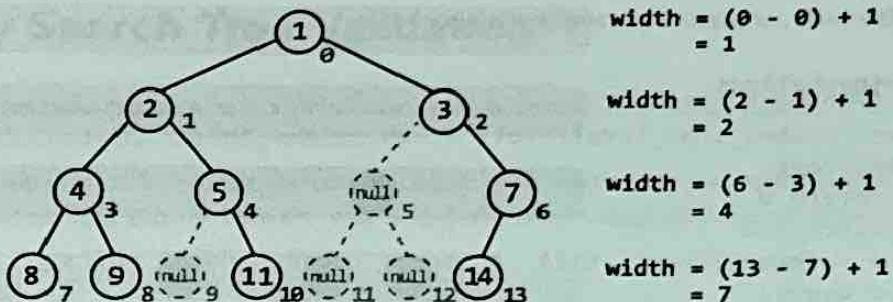


As we can see, the null nodes between the leftmost and rightmost nodes are considered in the width, as well.

Think about a data structure in which determining the width or distance between two elements is simple, such as an array, where the distance of two elements can be obtained by the difference between their indexes. If our binary tree also has indexes, it would similarly be possible to obtain the width between two nodes at a level. Let's explore a method to assign an index to each node.

Indexing a binary tree

Below, we see how indexes can be assigned at each node, starting with index 0 for the root node.



These indexes enable us to calculate the width of a level using `rightmost_index - leftmost_index + 1`, as shown in the diagram above. Note, the indexes at the null nodes are added only for visualization.

But how can we set up something like this? The key observation is that each node's index can be determined from its parent's index. We can derive the following relationship for any node at index `i`:

- Its **left child** will be at index $2*i + 1$.
- Its **right child** will be at index $2*i + 2$.

Now, we just need a way to traverse each level of the tree to determine the width at individual levels, allowing us to obtain the width of the widest level in the tree. We can use level-order traversal for this.

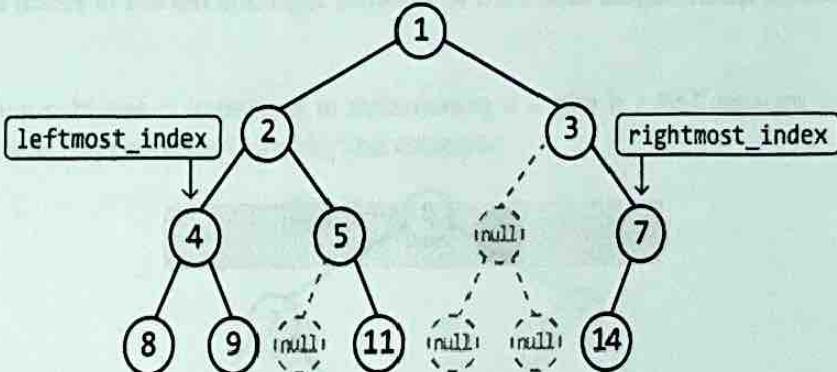
Level-order traversal

If you're unfamiliar with the level-order traversal algorithm, study the [Rightmost Nodes of a Binary Tree](#) problem before continuing.

Remember, level-order traversal utilizes a queue to process the binary tree nodes. In this problem, whenever we push a node into this queue, we also push its respective index along with it, to know which index it's associated with.

The width of a level can be calculated using `rightmost_index - leftmost_index + 1`. To perform this calculation, we need the index of the first (leftmost) and last (rightmost) nodes of that level. Here's how we obtain these indexes:

- Set `leftmost_index` to the index at the first node of the level.
- Start `rightmost_index` at the same point as `leftmost_index` and update it as we traverse the level. This way, it will eventually be set to the last index after traversing the level.



As we calculate the width at each level, we keep track of the largest width using `max_width`, rep-

resenting the width of the widest level in the binary tree.

Implementation

```
def widest_binary_tree_level(root: TreeNode) -> int:
    if not root:
        return 0
    max_width = 0
    queue = deque([(root, 0)]) # Stores (node, index) pairs.
    while queue:
        level_size = len(queue)
        # Set the 'leftmost_index' to the index of the first node in
        # this level. Start 'rightmost_index' at the same point as
        # 'leftmost_index' and update it as we traverse the level,
        # eventually positioning it at the last node.
        leftmost_index = queue[0][1]
        rightmost_index = leftmost_index
        # Process all nodes at the current level.
        for _ in range(level_size):
            node, i = queue.popleft()
            if node.left:
                queue.append((node.left, 2*i + 1))
            if node.right:
                queue.append((node.right, 2*i + 2))
            rightmost_index = i
        max_width = max(max_width, rightmost_index - leftmost_index + 1)
    return max_width
```

Complexity Analysis

Time complexity: The time complexity of `widest_binary_tree_level` is $O(n)$, where n denotes the number of nodes in the tree. This is because we process each node once during level-order traversal.

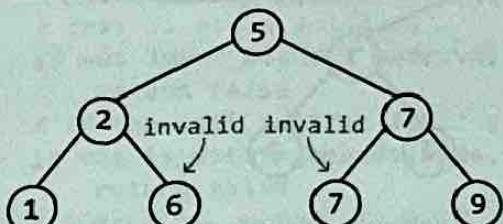
Space complexity: The space complexity is $O(n)$ due to the space taken up by the queue. The queue's size will grow as large as the level with the most nodes. In the worst case, this occurs at the last level when all the last-level nodes are non-null, totaling approximately $n/2$ nodes.

Binary Search Tree Validation

Verify whether a binary tree is a valid binary search tree (BST). A BST is a binary tree where each node meets the following criteria:

- A node's left subtree contains only nodes of lower values than the node's value.
- A node's right subtree contains only nodes of greater values than the node's value.

Example:



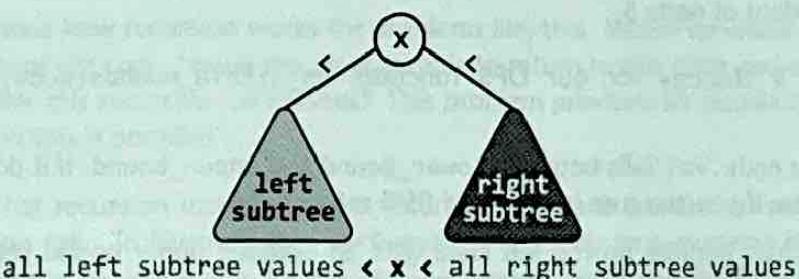
Output: False

Explanation: This tree has two violations of the BST criteria:

- Node 5's left subtree contains node 6, and node 6's value is greater than 5.
- Node 7 has a left child with the same value of 7.

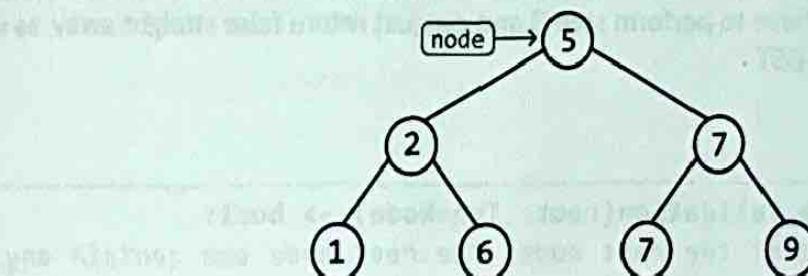
Intuition

A BST maintains a logically sorted order of values by complying with the criteria specified in the problem description. That is, the left subtree of a node with value x must consist of values less than x , and its right subtree must consist of values strictly greater than x .



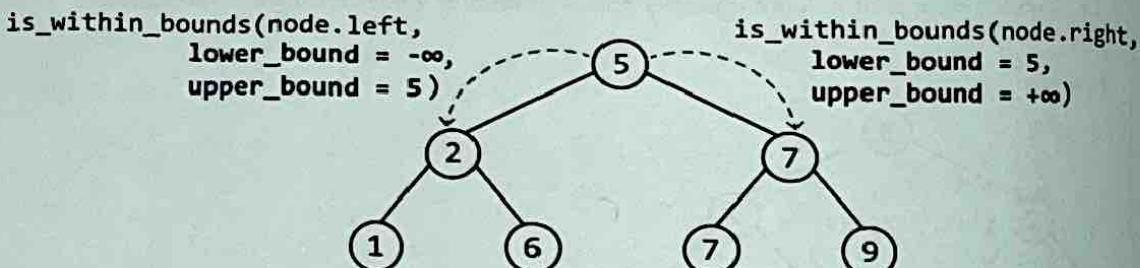
In addition, all nodes in the left and right subtrees of the above diagram must follow these criteria, too.

Since evaluating subtrees is important in determining if a tree is a BST, let's try a recursive DFS approach to solve this problem. Consider this example:

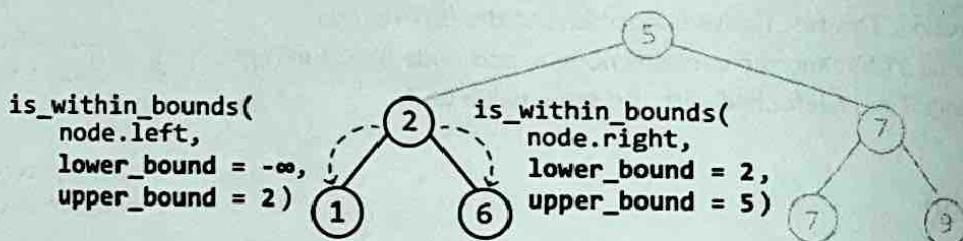


There's nothing to assess at the root node because it can contain any value. So, let's have a look at its children. Here's how we assess these nodes based on the BST rules:

- All nodes to the left of node 5 should be less than 5. So, when we make a recursive call to its left child, we pass in an upper bound of 5.
- All nodes to the right of node 5 should be greater than 5. So, when we make a recursive call to its right child, we pass in a lower bound of 5.



Let's now consider node 2. It satisfies its specified lower and upper bounds of $-\infty$ and 5, respectively. So, let's verify its subtrees. Applying the same logic as before, node 2's left child should have an upper bound of 2, while its right child should have a lower bound of 2.



Above, we see that node 6 violates its expected upper bound of 5 (since $6 > 5$). This violation indicates we're dealing with an invalid binary tree. Note, node 6 has an upper bound of 5 because it's a left descendant of node 5.

We now have a strategy for our DFS function, `is_within_bounds(node, lower_bound, upper_bound)`:

1. Check if the `node.val` falls between `lower_bound` and `upper_bound`. If it does, continue to the next step. If not, the tree is an invalid BST: return false.
2. Call `is_within_bounds` on the left child with an updated upper bound set to the current node's value: `is_within_bounds(node.left, lower_bound, node.val)`.
3. Call `is_within_bounds` on the right child with an updated lower bound set to the current node's value: `is_within_bounds(node.right, node.val, upper_bound)`.
4. If both recursive calls return true, then the current node's subtree is a valid BST: return true.

A minor optimization to make here is to check if the value we get from the recursive call at step 2 is false. If it's false, we don't have to perform step 3 and can just return false straight away, as we already know this tree isn't a BST.

Implementation

```
def binary_search_tree_validation(root: TreeNode) -> bool:  
    # Start validation at the root node. The root node can contain any
```

```
# value, so set the initial lower and upper bounds to -infinity and
# +infinity, respectively.
return is_within_bounds(root, float('-inf'), float('inf'))

def is_within_bounds(node: TreeNode,
                     lower_bound: int, upper_bound: int) -> bool:
    # Base case: if the node is null, it satisfies the BST condition.
    if not node:
        return True
    # If the current node's value is not within the valid bounds, this
    # tree is not a valid BST.
    if not lower_bound < node.val < upper_bound:
        return False
    # If the left subtree isn't a BST, this tree isn't a BST.
    if not is_within_bounds(node.left, lower_bound, node.val):
        return False
    # Otherwise, return true if the right subtree is also a BST.
    return is_within_bounds(node.right, node.val, upper_bound)
```

Complexity Analysis

Time complexity: The time complexity of `binary_search_tree_validation` is $O(n)$, where n denotes the number of nodes in the tree. This is because we process each node recursively at most once.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the recursive call stack, which can grow as large as the height of the binary tree. The largest possible height of a binary tree is n .

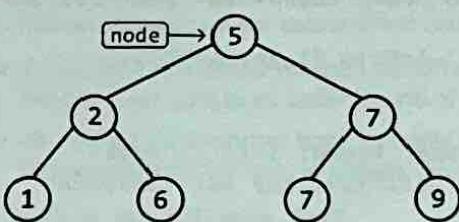
Detailed Recursive Demonstration

You may be curious how recursion works for problems like this. When we make a recursive DFS call in the middle of the code, how is the program able to return to this code and continue running the rest of it, after this recursive call finishes? This problem provides an excellent opportunity to demonstrate how this is possible.

The answer is that recursion makes use of a recursive call stack to manage each instance of a recursive function call. To illustrate this, we'll revisit the binary tree example discussed earlier, stepping through the recursive calls for the first few nodes. We'll also display the recursive call stack to help track the state of each call more clearly.

The first DFS call is made to the root node, which is given a lower and upper bound of $-\infty$ and $+\infty$, respectively:





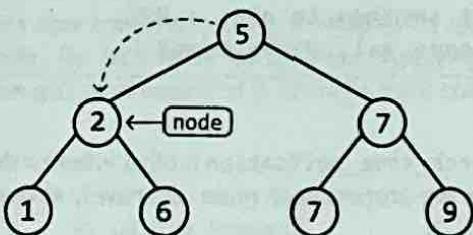
→ `is_within_bounds(5, -∞, +∞)`

recursive call stack

`lower_bound < 5 < upper_bound`
 $(-\infty)$ $(+\infty)$

We see that node 5 satisfies its specified lower and upper bounds.

At each node, after we confirm that its value falls within its expected lower and upper bounds, we make a recursive call to its left child before any right children are processed:

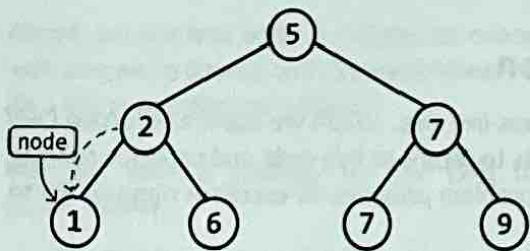


→ `is_within_bounds(2, -∞, 5)`

`is_within_bounds(5, -∞, +∞)`

recursive call stack

`lower_bound < 2 < upper_bound`
 $(-\infty)$ (5)



→ `is_within_bounds(1, -∞, 2)`

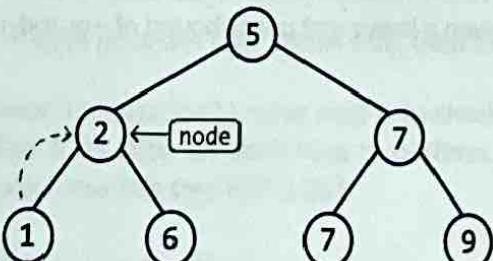
`is_within_bounds(2, -∞, 5)`

`is_within_bounds(5, -∞, +∞)`

recursive call stack

`lower_bound < 1 < upper_bound`
 $(-\infty)$ (2)

After processing node 1, which has no children, recursion naturally progresses back to the recursive call to node 2, which is now at the top of the stack:



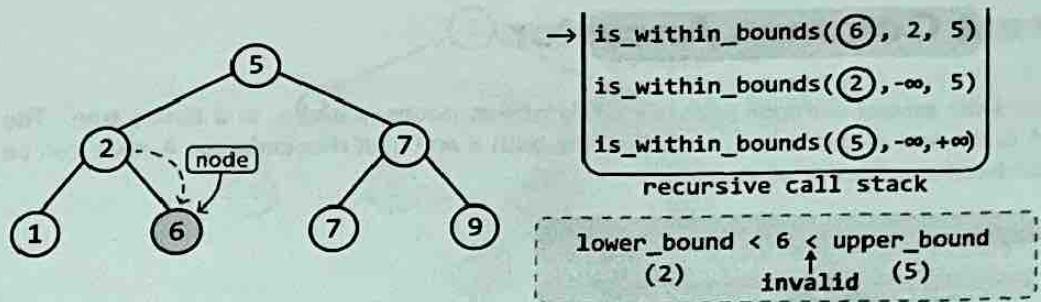
→ `is_within_bounds(1, -∞, 2)`

`is_within_bounds(2, -∞, 5)`

`is_within_bounds(5, -∞, +∞)`

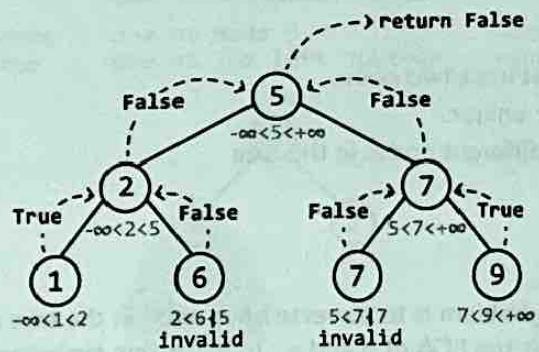
recursive call stack

Continuing from where we left off at this instance, it now processes its right child by making a recursive call to it:



Once we reach node 6, the algorithm notices that its value violates its expected upper bound. As such, it returns false.

In this DFS solution, the recursion starts at the root node, but values start being returned from the leaf nodes and bubble upwards to their parent nodes. Below is a diagram showing how the boolean result at each node moves up from the leaf nodes to the root:

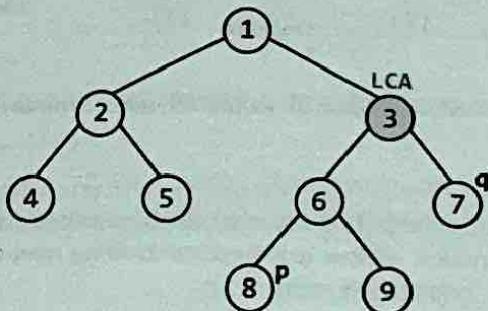


Note that nodes 5, 2, and 7 return false because, while their values fall within their bounds, at least one node in their subtrees does not.

Lowest Common Ancestor

Return the lowest common ancestor (LCA) of two nodes, p and q, in a binary tree. The LCA is defined as the lowest node that has both p and q as descendants. A node can be considered an ancestor of itself.

Example:



Constraints:

- The tree contains at least two nodes.
- All node values are unique.
- p and q represent different nodes in the tree.

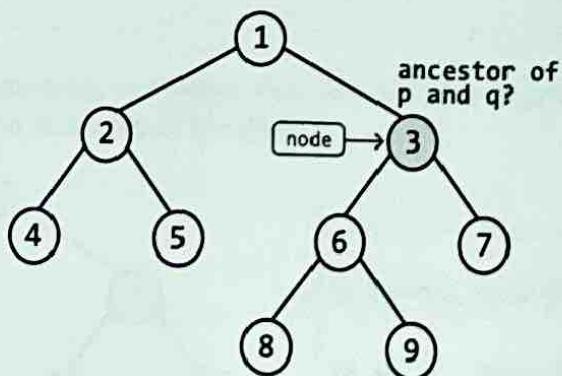
Intuition

One strategy to solve this problem is to traverse each node in the tree and evaluate whether the current node at any point is the LCA of p and q. To make this evaluation, it's crucial to know the conditions a node needs to meet to be the LCA.

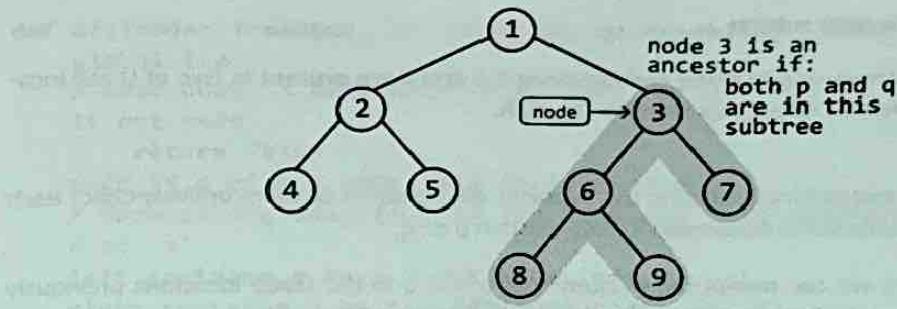
Let's start by discussing where p or q would need to be for the current node to be at least an ancestor of both.

Identifying when the current node is an ancestor of p and q

Consider the following binary tree. Let's explore where p and q would need to be for node 3 to be an ancestor.



Node 3 is only an ancestor of p and q when both nodes are in node 3's subtree. If they were anywhere else, they would not be descendants of node 3.



Now, let's examine the conditions in which a node is the lowest common ancestor of p and q.

Identifying when the current node is the LCA of p and q

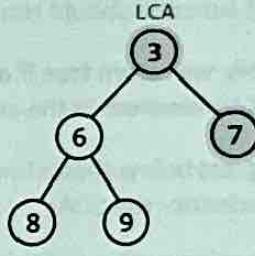
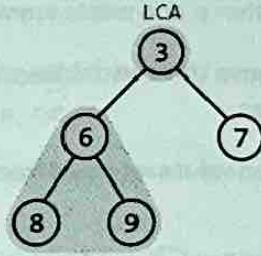
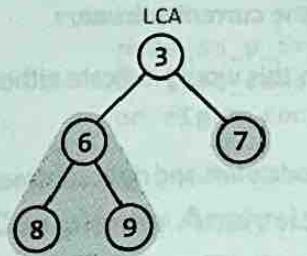
Let's identify where in the subtree p and q should be to qualify node 3 as the lowest ancestor of them both.

If node 3 is the LCA,
where are p and q?

-one at the right subtree
-one at the left subtree

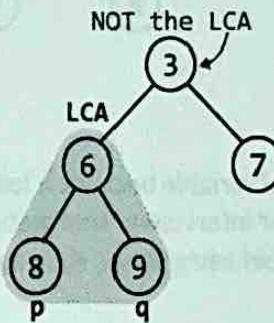
-one at node 3
-one at the left subtree

-one at node 3
-one at the right subtree



As we can see above, there are three possible cases where node 3 is the LCA. This is because in each case, there isn't a node lower than node 3 that contains both p and q as descendants, making node 3 the LCA.

Note that if both p and q were in just one of node 3's subtrees (e.g., the left subtree), node 3 would not be the LCA because there would be a node lower in the left subtree that's an ancestor of both p and q:



What can we derive from these observations? In the three cases where the current node is the LCA, p and q were found in exactly two of the following three locations:

- The current node itself.
- The current node's left subtree.

- The current node's right subtree.

This indicates that by traversing the tree and checking if p and q are present in two of these locations for each node, we can effectively identify the LCA.

Depth-first search

Recursive DFS is well-suited for traversal in this problem, as it enables us to recursively check each node's left and right subtrees to determine if they contain p or q.

In the implementation, we can assess the existence of p and q in the three locations previously mentioned, by attaining the following three boolean variables at each node:

- `node_is_p_or_q = (node == p or node == q)`
 - `left_contains_p_or_q = dfs(node.left)`
 - `right_contains_p_or_q = dfs(node.right)`

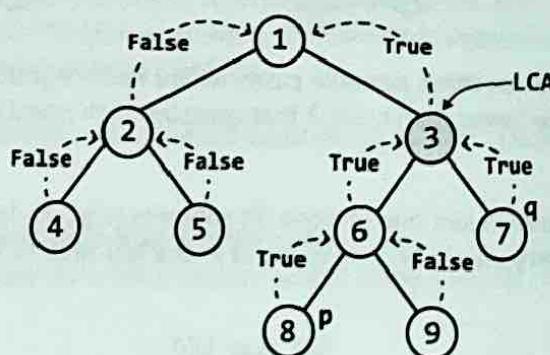
Again, two of these variables need to be true for the current node to be the LCA

Return statement

Whenever we make a recursive DFS call, such as `dfs(node.left)`, we expect that call to return true if the subtree rooting from `node.left` contains p or q. This means the return statement of our DFS function should return true if either p or q exists anywhere in the current subtree.

To do this, we return true if any of the above three variables are true, as this would indicate either p or q is somewhere in the current subtree.

The diagram below shows how the boolean values returned from each node's left and right subtrees help to identify the LCA.



Implementation

Note, this implementation uses a global variable because it leads to a more readable solution. However, it's important to confirm with your interviewer that global variables are acceptable. If not, you may need to adjust the solution to avoid using them, such as passing the variable as an argument, or finding an alternative approach.

```
def lowest_common_ancestor(root: TreeNode,
                           p: TreeNode, q: TreeNode) -> TreeNode:
    dfs(root, p, q)
    return lca
```

```

def dfs(node: TreeNode, p: TreeNode, q: TreeNode) -> bool:
    global lca
    # Base case: a null node is neither 'p' nor 'q'.
    if not node:
        return False
    node_is_p_or_q = node == p or node == q
    # Recursively determine if the left and right subtrees contain 'p'
    # or 'q'.
    left_contains_p_or_q = dfs(node.left, p, q)
    right_contains_p_or_q = dfs(node.right, p, q)
    # If two of the above three variables are true, the current node is
    # the LCA.
    if (
        node_is_p_or_q
        + left_contains_p_or_q
        + right_contains_p_or_q
        == 2
    ):
        lca = node
    # Return true if the current subtree contains 'p' or 'q'.
    return (
        node_is_p_or_q
        or left_contains_p_or_q
        or right_contains_p_or_q
    )

```

Complexity Analysis

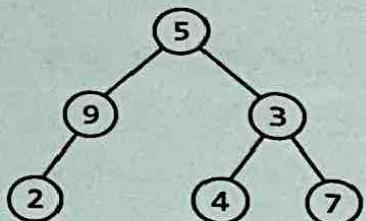
Time complexity: The time complexity of `lowest_common_ancestor` is $O(n)$, where n denotes the number of nodes in the tree. This is because the algorithm traverses each node of the tree once.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the recursive call stack, which can grow as large as the height of the binary tree. The largest possible height of a binary tree is n .

Build a Binary Tree From Preorder and Inorder Traversals

Construct a binary tree using arrays of values obtained after a preorder traversal and an inorder traversal of the tree.

Example:



Input: preorder = [5, 9, 2, 3, 4, 7], inorder = [2, 9, 5, 4, 3, 7]

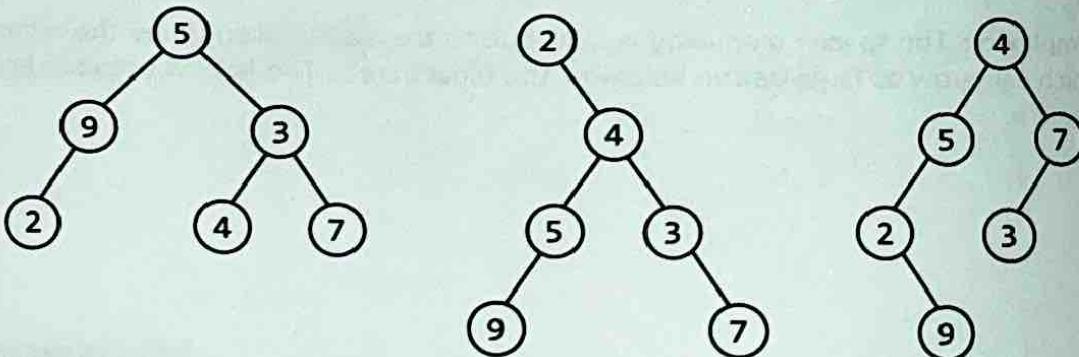
Constraints:

- The tree consists of unique values.

Intuition

The first question that might come to mind is why both the preorder and inorder arrays are needed to solve this problem. Can't we just use one of them? The reason is that each individual traversal order could represent multiple possible trees. For example, below are some trees that correspond to the following inorder traversal:

inorder = [2 9 5 4 3 7]



If one traversal array isn't enough, it might be possible to build the tree using an additional traversal array as a reference point, to help identify how each node should be placed.

The root node

Whatever our approach, we initially need a way to access the root node since the root node is necessary to construct the rest of the tree.

To identify where the root node is, let's first remind ourselves how nodes are processed during each traversal algorithm.

Preorder traversal:

1. Process the current node.
2. Process the left subtree.
3. Process the right subtree.

Inorder traversal:

1. Process the left subtree.
2. Process the current node.
3. Process the right subtree.

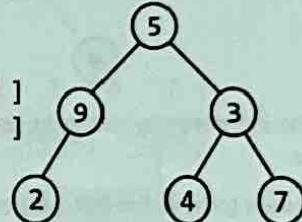
Notice that during preorder traversal, the current node is processed before its subtrees. From this, we can infer that preorder traversal processes the root node first, meaning the **first value in the preorder array is the value of the root node**.

With this established, let's find a method to build the rest of the tree.

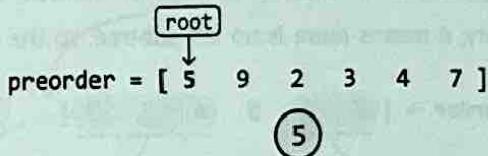
Building the tree

Consider the preorder and inorder traversal arrays below and the corresponding binary tree:

preorder = [5 9 2 3 4 7]
inorder = [2 9 5 4 3 7]

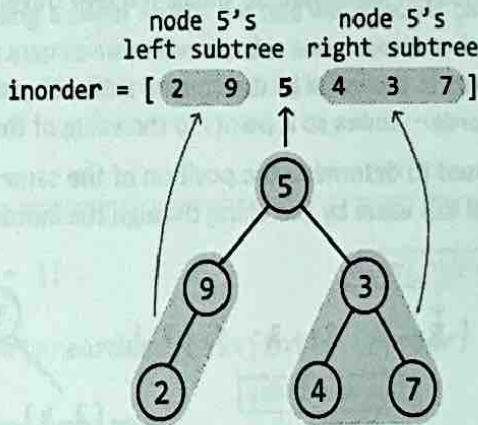


As mentioned above, we know the first value of the preorder array, 5, is the value of the root node.



Now the question is, what node should be placed next? The preorder array alone isn't enough to determine this. So, we should consult the inorder array.

Inorder traversal first visits a node's left subtree before processing the node itself, so we can deduce that all values in the inorder array to the left of 5 are part of its left subtree. Similarly, all values to the right of 5 in the inorder array belong to its right subtree:



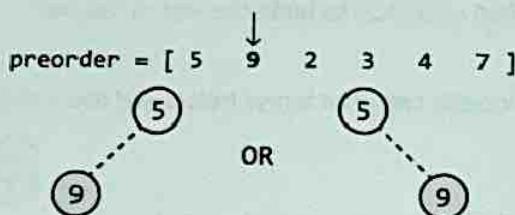
Now what? With the root node placed, we have two subtrees to build: node 5's left and right subtrees.

If we build the tree starting with the already-created root node, followed by the left subtree and then the right subtree, we're effectively **building the tree using preorder traversal**. This is useful

because we happen to have an array of values from a preorder traversal, which means we know the exact order in which the nodes should be created.

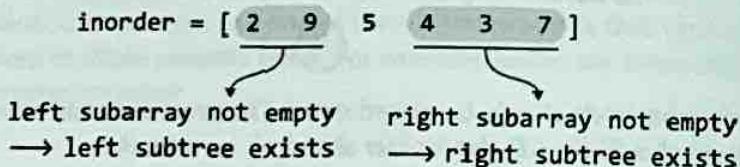
As we're building the tree using preorder traversal, the next node to be created at any point will be the next value in the preorder array. This means we can iterate through the preorder array to place each new node.

Based on this, we know the next node to be placed will have a value of 9. But how do we know if node 9 is a left child or a right child of 5?



This is where the inorder array comes in, as it helps us determine the structure of the tree. Consider the left subtree:

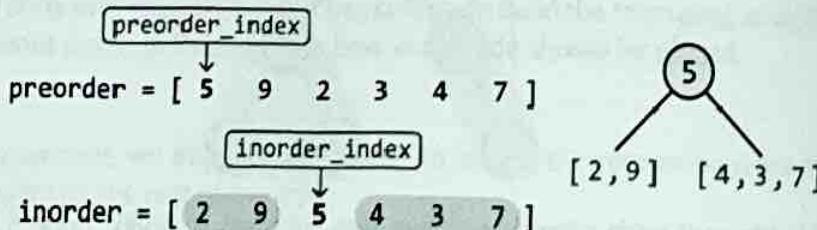
- When we want to build the left subtree of a node, we look at the part of the inorder array that corresponds to the left subtree. In our example, this is the subarray [2, 9].
- If this subarray is not empty, we proceed to build the left subtree, starting with node 9.
- If this subarray is empty, it means there is no left subtree, so the current node's left child is null.



The same logic applies to the right subtree: we only build it if its corresponding inorder subarray isn't empty.

Now, let's devise a strategy for our algorithm. To utilize the two traversal arrays, we can assign a pointer to each:

- A `preorder_index` points to the value of the current subtree's root node. Once this node is created, increment `preorder_index` so it points to the value of the next node to be created.
- An `inorder_index` is used to determine the position of the same value in the inorder array. We can find the index of this value by searching through the inorder array.



At each recursive call, once these indexes are obtained, we:

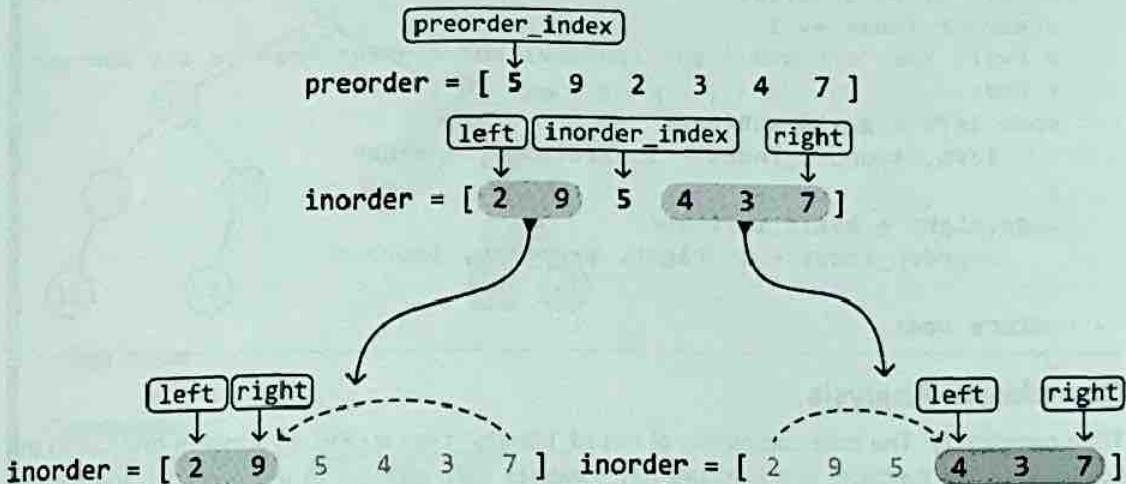
1. Create the current node using the value pointed at by `preorder_index`.

- Increment `preorder_index` so it points to the value of the next node to be created.
- Make a recursive call to build the current node's left and right subtrees:
 - Pass in the subarray `inorder[0, inorder_index - 1]` to build the left subtree.
 - Pass in the subarray `inorder[inorder_index + 1, n - 1]` to build the right subtree.

Optimization - left and right pointers

One inefficiency of the above approach is that we extract entire subarrays out of the inorder array whenever we make a recursive call, which takes $O(n)$ time each time we do this, where n denotes the length of each input array. A more efficient approach is to define the range of each inorder subarray using **left** and **right** pointers. Specifically:

- The left subtree contains the values in the range `[left, inorder_index - 1]`.
- The right subtree contains the values in the range `[inorder_index + 1, right]`.



This allows us to define subarrays by just moving pointers, as opposed to creating completely new subarrays.

Optimization - hash map

At each node, it's necessary to set `inorder_index` to the position of the same value pointed at by `preorder_index`. Performing a linear search for this value would take $O(n)$ time for each node. Instead, we can use a hash map to store the inorder array values and their indexes, allowing us to retrieve any value's index from the inorder array in $O(1)$ time.

Implementation

```

preorder_index = 0
inorder_indexes_map = {}

def build_binary_tree(preorder: List[int], inorder: List[int]) ->
    TreeNode:
    global inorder_indexes_map
    # Populate the hash map with the inorder values and their indexes.
    for i, val in enumerate(inorder):
        inorder_indexes_map[val] = i
    # Build the tree and return its root node.
    return build_subtree(0, len(inorder) - 1, preorder, inorder)
  
```

```
def build_subtree(left: int, right: int, preorder: List[int], inorder: List[int]) -> TreeNode:
    global preorder_index, inorder_indexes_map
    # Base case: if no elements are in this range, return None.
    if left > right:
        return None
    val = preorder[preorder_index]
    # Set 'inorder_index' to the index of the same value pointed at by
    # 'preorder_index'.
    inorder_index = inorder_indexes_map[val]
    node = TreeNode(val)
    # Advance 'preorder_index' so it points to the value of the next
    # node to be created.
    preorder_index += 1
    # Build the left and right subtrees and connect them to the current
    # node.
    node.left = build_subtree(
        left, inorder_index - 1, preorder, inorder
    )
    node.right = build_subtree(
        inorder_index + 1, right, preorder, inorder
    )
return node
```

Complexity Analysis

Time complexity: The time complexity of `build_binary_tree` is $O(n)$, as it makes one call to the `build_subtree` function, which recursively traverses each element in the `preorder` and `inorder` arrays once, resulting in an $O(n)$ runtime.

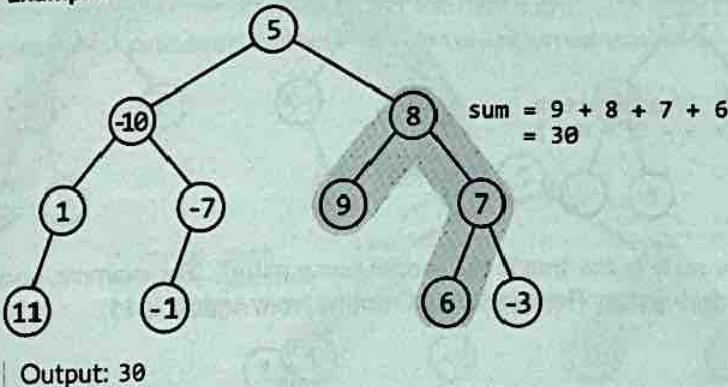
Space complexity: The space complexity is $O(n)$ due to the space taken up by the recursive call stack, which can grow as large as the height of the binary tree. The largest possible height of a binary tree is n . The hash map `inorder_indexes_map` also takes up $O(n)$ space.

Maximum Sum of a Continuous Path in a Binary Tree

Return the maximum sum of a continuous path in a binary tree. A path is defined by the following characteristics:

- Consists of a sequence of nodes that can begin and end at any node in the tree
- Each consecutive pair of nodes in the sequence is connected by an edge
- The path must be a single continuous sequence of nodes that doesn't split into multiple paths

Example:

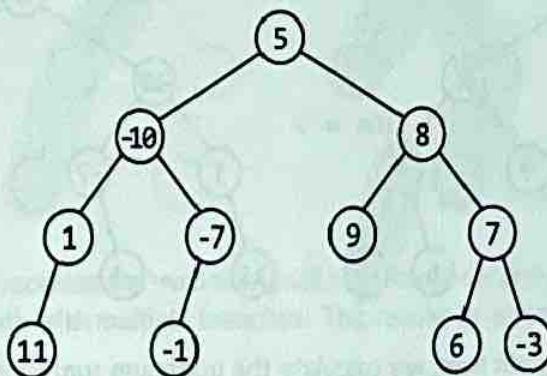


Constraints:

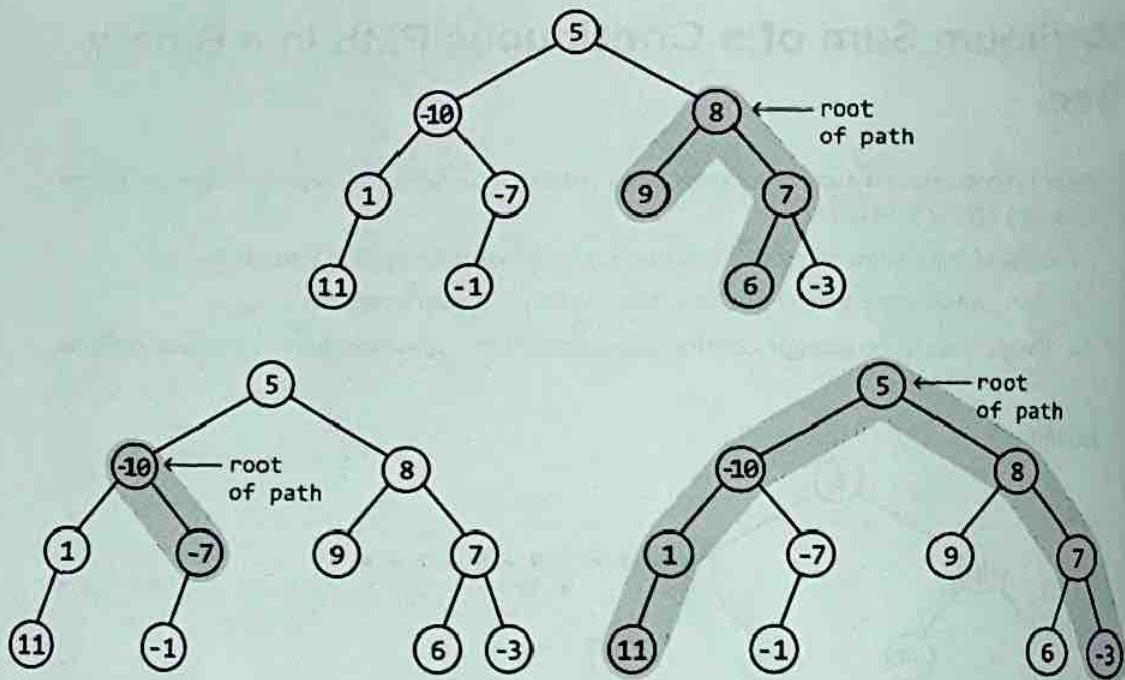
- The tree contains at least one node.

Intuition

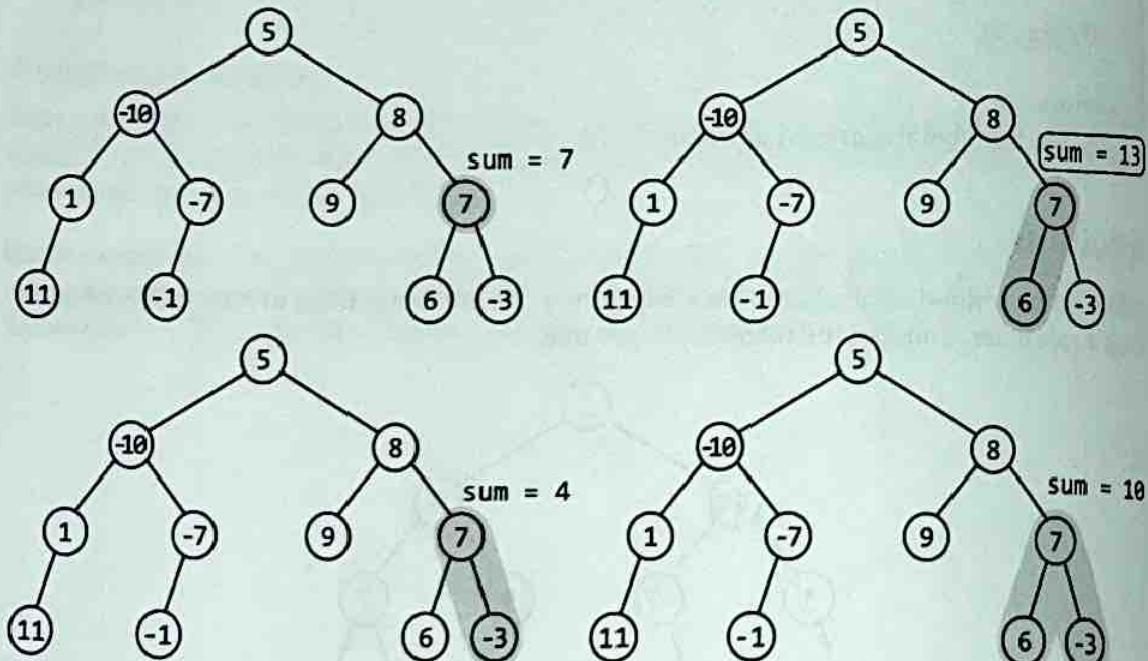
Let's first understand what a path is in a binary tree. An important thing to note is that all paths have a root node. Consider the following binary tree:



Every path that exists in this tree has a corresponding root node, as shown in the three examples below:



Inversely, this means that every node in the tree is the root of some path(s). For example, node 7 in the tree below is the root of four paths. The largest sum rooting from node 7 is 13:

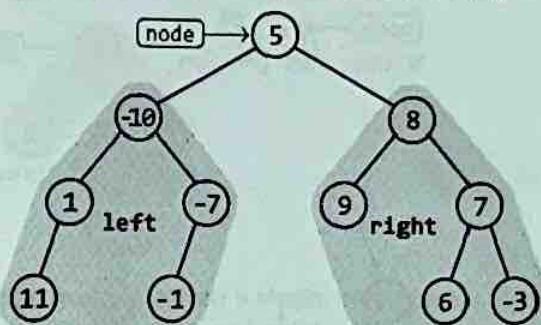


So, to find the maximum path sum, we calculate the maximum sum rooting from each node and return the largest of these sums.

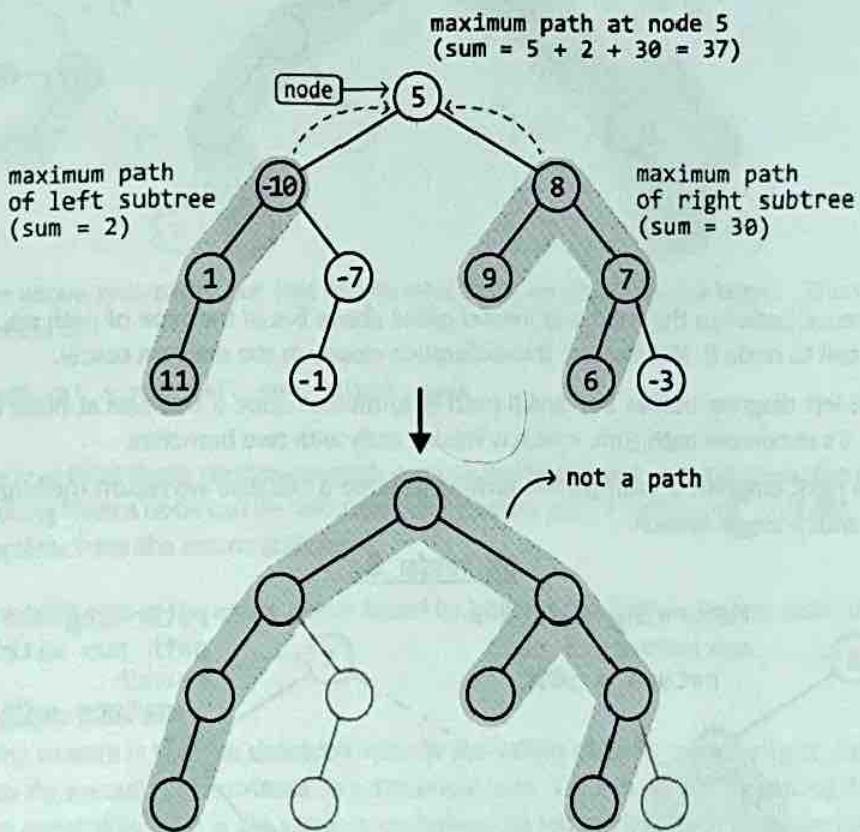
Calculating the maximum path sum at any node

Let's try adopting a recursive strategy for this. Consider the root node of our example. The maximum sum of a path rooting from node 5 involves the maximum gain we can attain from its left subtree and its right subtree.

$$\begin{aligned} \text{max path sum rooting from node 5} \\ = 5 + (\text{max from left}) + (\text{max from right}) \end{aligned}$$

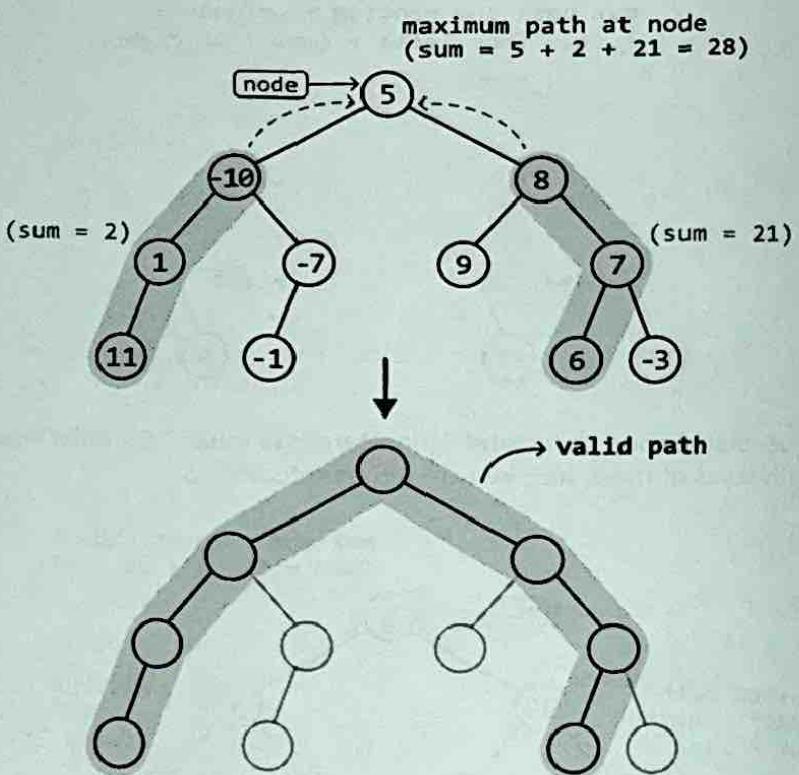


Which sums do we expect node 5's left and right subtrees to return? Consider what happens when the maximum path sums of these subtrees are returned to node 5:



As we can see, when we received the maximum path sum from a recursive call made to node 8, we received the sum of a path with multiple branches. This results in an invalid path formed at node 5.

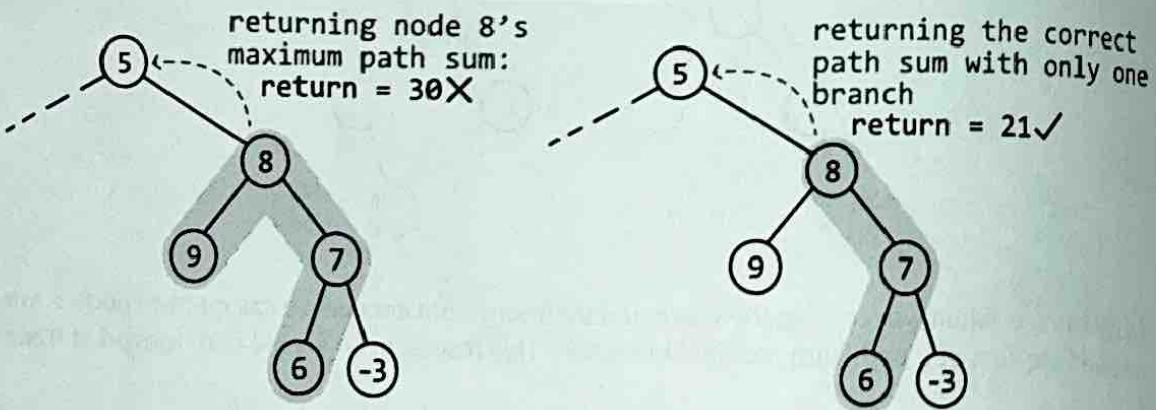
Below, we can see what we actually want. The sums of the two paths returned here correctly give us the maximum path sum rooting from node 5:



The difference between the valid and invalid paths above lies in the type of path returned by the recursive call to node 8. We can see this difference clearly in the diagram below.

- In the left diagram below, an invalid path is formed at node 5 because at node 8, we return node 8's maximum path sum, which is from a path with two branches.
 - In the right diagram, a valid path is formed at node 5 because we return the largest sum of a path with a single branch.

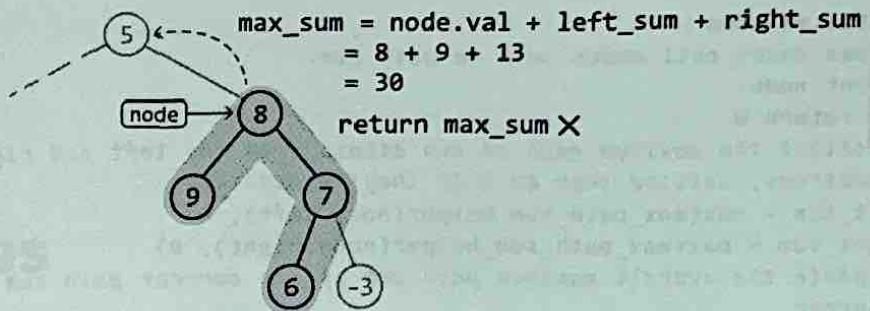
at node 8:



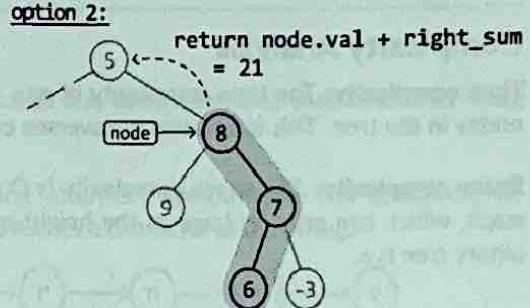
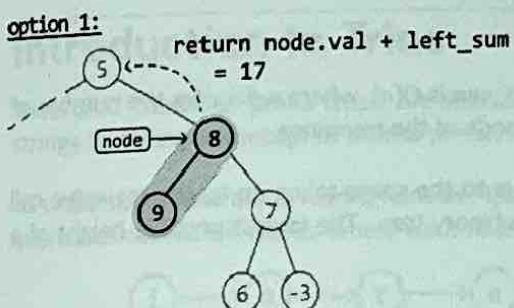
This observation highlights that we can't just return the maximum path sum of a node. So, let's have a closer look at which path sum we should return, instead.

Identifying the value returned during recursion

Consider node 8 from the above example. The maximum path sum rooting from node 8 is 30. We know from the discussion above that we can't just return this maximum path sum value:



We know we need to make sure we **return a single, continuous path** from node 8. This would mean returning a path with node 8 as an end point, which leaves us with two main choices for values we could return:



Between the above two paths, we just return whichever of their sums is larger. Therefore, our return statement is:

```
| return node.val + max(left_sum, right_sum)
```

Now that we're getting single continuous path sums from the left and right subtrees, the **maximum path sum rooting from a node** can be calculated using `node.val + left_sum + right_sum`. This is done separately from the return statement.

The maximum path sum of the entire tree is found by keeping track of the largest path sum formed at every node.

Handling negative path sums

One final thing to note is that we shouldn't include the values of `left_sum` or `right_sum` if either is negative, as they wouldn't contribute to a maximum sum. We can do this by setting their values to 0 if they're negative, which is the same as excluding the left or right path from the sum.

Implementation

```
max_sum = float('-inf')

def max_path_sum(root: TreeNode) -> int:
    global max_sum
    max_path_sum_helper(root)
    return max_sum

def max_path_sum_helper(node: TreeNode) -> int:
```

```
global max_sum
# Base case: null nodes have no path sum.
if not node:
    return 0
# Collect the maximum gain we can attain from the left and right
# subtrees, setting them to 0 if they're negative.
left_sum = max(max_path_sum_helper(node.left), 0)
right_sum = max(max_path_sum_helper(node.right), 0)
# Update the overall maximum path sum if the current path sum is
# larger.
max_sum = max(max_sum, node.val + left_sum + right_sum)
# Return the maximum sum of a single, continuous path with the
# current node as an endpoint.
return node.val + max(left_sum, right_sum)
```

Complexity Analysis

Time complexity: The time complexity of `max_path_sum` is $O(n)$, where n denotes the number of nodes in the tree. This is because it traverses each node of the tree once.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the recursive call stack, which can grow as large as the height of the binary tree. The largest possible height of a binary tree is n .

Tries

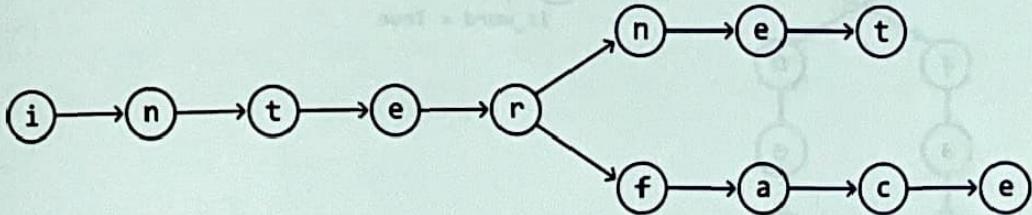
Introduction to Tries

Tries, also known as prefix trees, are specialized tree-like data structures that efficiently store strings by taking advantage of shared prefixes.

To understand how a trie works, consider the string "internet". We can store this string as a sequence of connected nodes, where each node represents a character in the string:



Let's say we also want to store the word "interface". Instead of creating a separate sequence of nodes, we take advantage of their common prefix "inter". Both words can share the nodes for "inter", and the remainder of "interface" can branch out from node 'r':

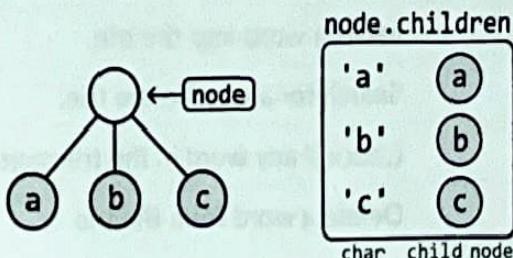


This is, in essence, how a trie works: by allowing words to reuse existing nodes based on shared prefixes, it efficiently stores strings in a way that minimizes redundancy.

TrieNode

There are two attributes each TrieNode should have.

1. Children: Each TrieNode has a data structure to store references to its child nodes. A **hash map** is typically used for this, with a character as the key and its corresponding child node as the value.



Sometimes, an array is used instead of a hash map if the possible characters of words in the trie

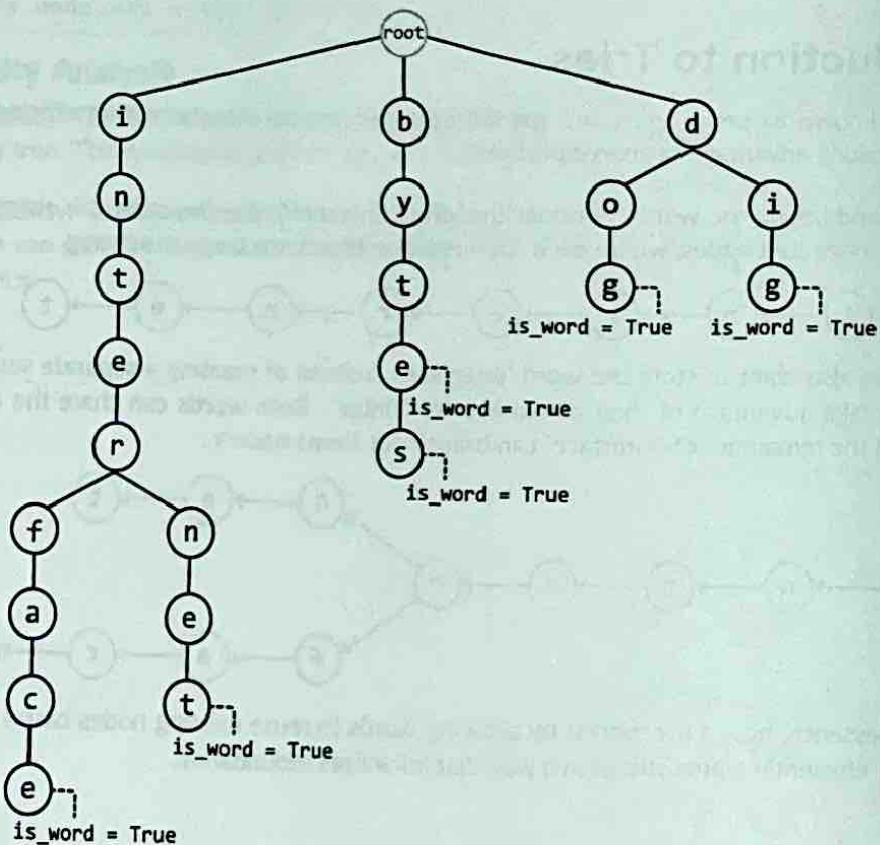
are limited and known in advance. For example, if they only contain lowercase English letters, an array of size 26 can be used, instead.

2. End of word indicator: Each `TrieNode` needs an attribute to indicate whether it marks the end of a word. This can be done in two ways:

- A boolean attribute (`is_word`) to confirm if the node is the end of a word.
- A string variable (`word`) that stores the word itself at the node. This is usually used if we also want to know the specific word that ends at this node.

Trie structure

This is an example of a trie that's been populated by inserting the words "internet", "interface", "byte", "bytes", "dog", and "dig". Every word represented by the trie branches out of a root `TrieNode`, which does not represent any character.



Below is the time complexity breakdown for trie operations involving a word of length k :

Operation	Complexity	Description
Insert	$O(k)$	Insert a word into the trie.
Search	$O(k)$	Search for a word in the trie.
Search prefix	$O(k)$	Check if any word in the trie starts with a given prefix.
Delete	$O(k)$	Delete a word from the trie

The implementation details of most of these functions are discussed in detail in the *Design a Trie*

problem.

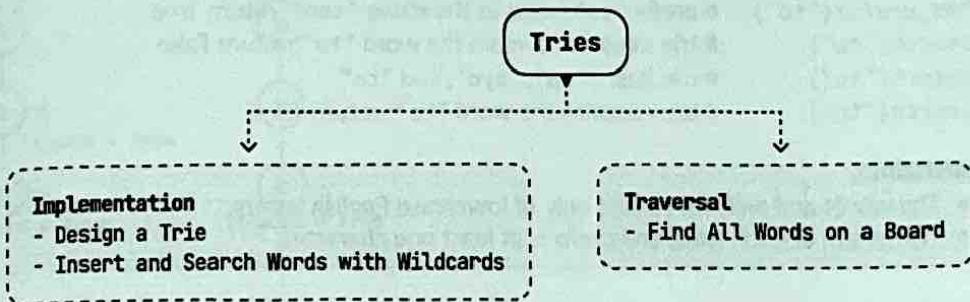
When to use tries

The primary use of a trie is to handle efficient prefix searches. If an interview problem asks you to find all strings that share a common prefix, a trie is likely the ideal data structure. Tries are also useful for word validation, allowing us to quickly verify the existence of words within a set of strings. They also help optimize data storage by reducing redundancy through shared prefixes among strings.

Real-world Example

Autocomplete: When you start typing a word, some systems use a trie to quickly suggest possible completions. Each node in the trie represents a character, and as you type, the system traverses the trie based on the input characters. This allows the system to efficiently retrieve all possible words or phrases that start with the entered prefix, enabling fast autocomplete suggestions.

Chapter Outline



Design a Trie

Design and implement a trie data structure that supports the following operations:

- `insert(word: str) -> None`: Inserts a word into the trie.
- `search(word: str) -> bool`: Returns true if a word exists in the trie, and false if not.
- `has_prefix(prefix: str) -> bool`: Returns true if the trie contains a word with the given prefix, and false if not.

Example:

```
Input: [insert("top"), insert("bye"), has_prefix("to"), search("to"),  
insert("to"), search("to")]
```

```
Output: [True, False, True]
```

Explanation:

```
insert("top")      # trie has: "top"  
insert("bye")      # trie has: "top" and "bye"  
has_prefix("to")   # prefix "to" exists in the string "top": return True  
search("to")       # trie does not contain the word "to": return False  
insert("to")       # trie has: "top", "bye", and "to"  
search("to")       # trie contains the word "to": return True
```

Constraints:

- The words and prefixes consist only of lowercase English letters.
- The length of each word and prefix is at least one character.

Intuition

Let's define a `TrieNode` using the same definition introduced in the introduction. In this implementation, the `is_word` attribute will be used to indicate whether a `TrieNode` marks the end of a word.

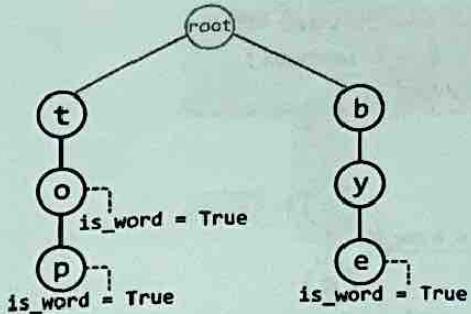
Initializing the trie

To initialize the trie, we define the `root TrieNode` in the constructor. All words inserted into the trie will branch out from this root node.

Inserting a word into the trie

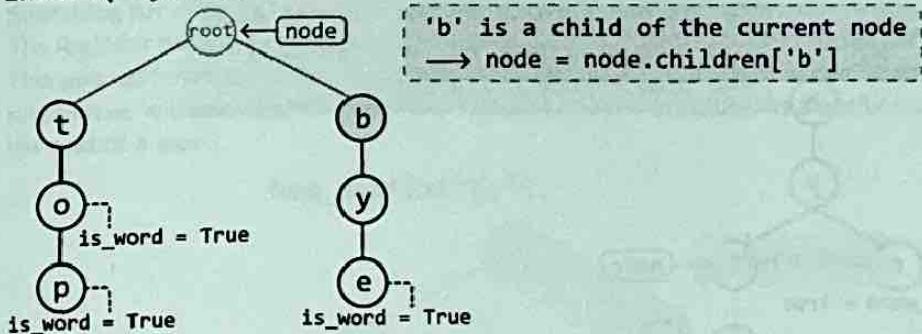
The `insert` function builds the trie word by word. What makes a trie useful is that it reduces redundancy by reusing existing nodes when possible. For example, if we insert "byte" when "bye" already exists in the trie, these two words should share the nodes that make up the prefix "by" to save space. This is an important point that will shape our implementation of this function.

To understand the insertion logic, let's walk through an example. Consider inserting the word "byte" into the trie below.

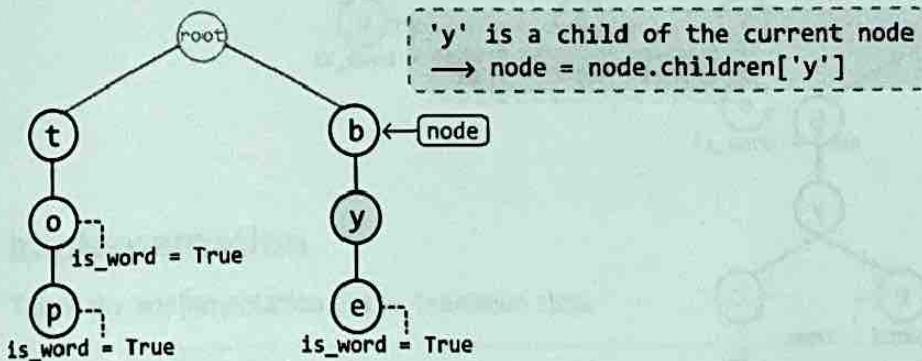


We first check if 'b', the first letter of the string, exists as a child of the root node by querying the hash map containing its children. In this case, it does. So, move to node 'b':

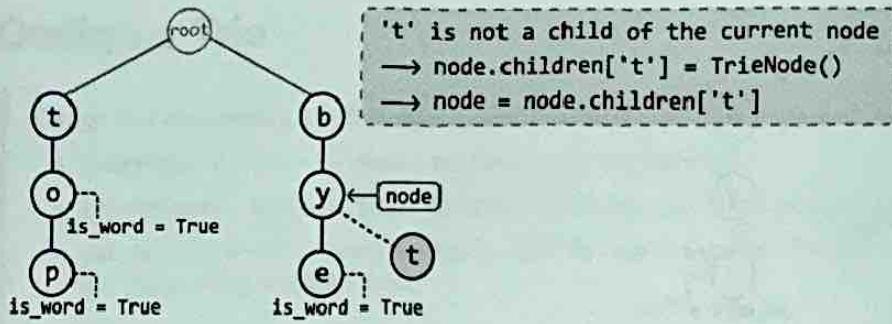
`insert("byte");`



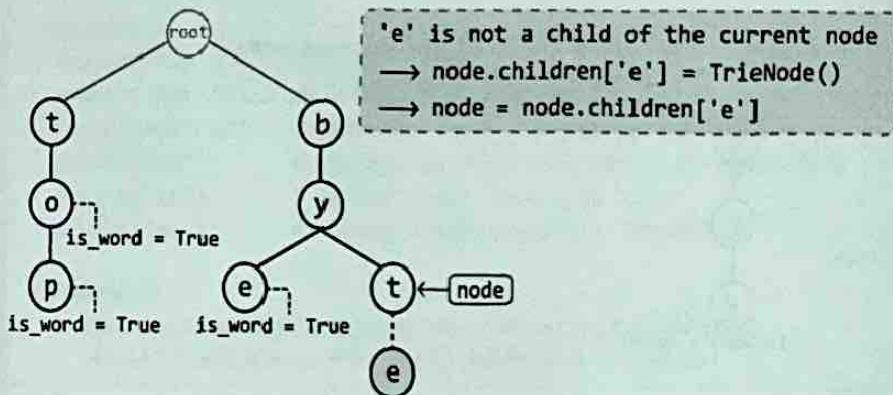
Now consider the second letter, 'y'. Similarly, node 'y' exists as a child of node 'b', so let's move to node 'y':



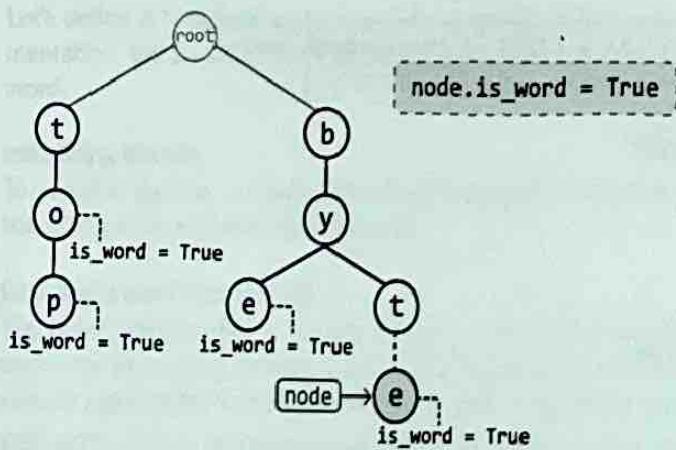
Now consider the next letter, 't'. Since node 't' doesn't exist as a child of node 'y', we need to create it and add it to node 'y's children hash map. Then, we can move to this newly created node 't':



The last letter is 'e', which doesn't exist as a child of node 't'. So, let's create node 'e' and add it to node 't's children:



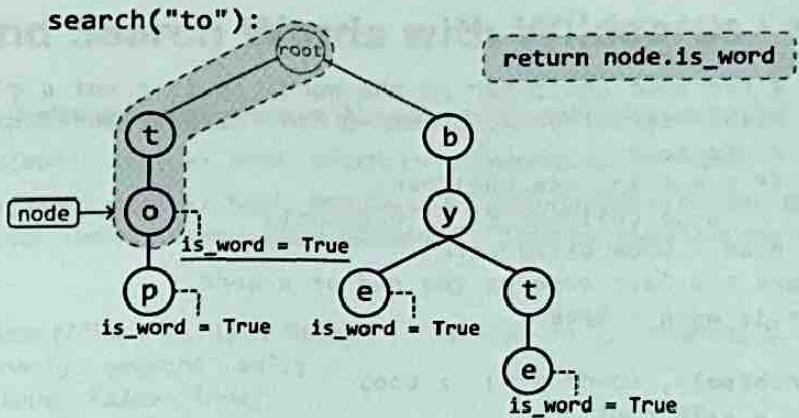
Now that we've reached the end of the word, we should set the `is_word` attribute of node 'e' to true, indicating that it marks the end of a word.



Searching for a word

Searching for a word involves the same strategy as insertion, where we move node by node down the trie. The two main differences are:

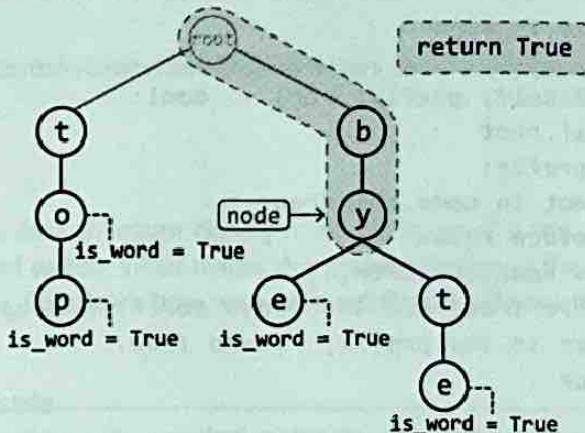
1. If a node corresponding to the current character in the word isn't found at any point, we return false because this would indicate the word doesn't exist in the trie.
2. After traversing all characters of the search term, we return true only if the final node's `is_word` attribute is true.



Searching for a prefix

The logic for finding a prefix is nearly identical to the logic discussed above for the search function. The only difference is after successfully traversing all characters in our search term, we can just return true without checking the final node's `is_word` attribute, as a prefix doesn't need to end at the end of a word.

`has_prefix("by"):`



Implementation

This is the implementation of the `TrieNode` class.

```

class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_word = False
  
```

This is the implementation of the `Trie` class.

```

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        pass
  
```

```

node = self.root
for c in word:
    # For each character in the word, if it's not a child of
    # the current node, create a new TrieNode for that
    # character.
    if c not in node.children:
        node.children[c] = TrieNode()
    node = node.children[c]
# Mark the last node as the end of a word.
node.is_word = True

def search(self, word: str) -> bool:
    node = self.root
    for c in word:
        # For each character in the word, if it's not a child of
        # the current node, the word doesn't exist in the Trie.
        if c not in node.children:
            return False
        node = node.children[c]
    # Return whether the current node is marked as the end of the
    # word.
    return node.is_word

def has_prefix(self, prefix: str) -> bool:
    node = self.root
    for c in prefix:
        if c not in node.children:
            return False
        node = node.children[c]
    # Once we've traversed the nodes corresponding to each
    # character in the prefix, return True.
    return True

```

Complexity Analysis

Time complexity:

- The time complexity of `insert` is $O(k)$, where k is the length of the word being inserted. This is because we traverse through or insert up to k nodes into the trie in each iteration.
- The time complexity of `search` and `has_prefix` is $O(k)$ because we search through at most k characters in the trie.

Space complexity:

- The space complexity of `insert` is $O(k)$ because in the worst case, the inserted word doesn't share any prefix with words already in the trie. In this case, k new nodes are created.
- The space complexity of `search` and `has_prefix` is $O(1)$ because no additional space is used to traverse the search term in the trie.

Insert and Search Words with Wildcards

Design and implement a data structure that supports the following operations:

- `insert(word: str) -> None`: Inserts a word into the data structure.
- `search(word: str) -> bool`: Returns true if a word exists in the data structure and false if not. The word may contain wildcards ('.') that can represent any letter.

Example:

```
Input: [insert("band"), insert("rat"), search("ra."), search("b.."),
       insert("ran"), search(".an")]
Output: [True, False, True]
```

Explanation:

```
insert("band")    # data structure has: "band"
insert("rat")     # data structure has: "band" and "rat"
search("ra.")     # "ra." matches "rat": return True
search("b..")      # no three-letter word starting with 'b' in the data structure:
                   return False
insert("ran")      # data structure has: "band", "rat", and "ran"
search(".an")      # ".an" matches "ran": return True
```

Constraints:

- Words will only contain lowercase English letters and ('.') characters.

Intuition

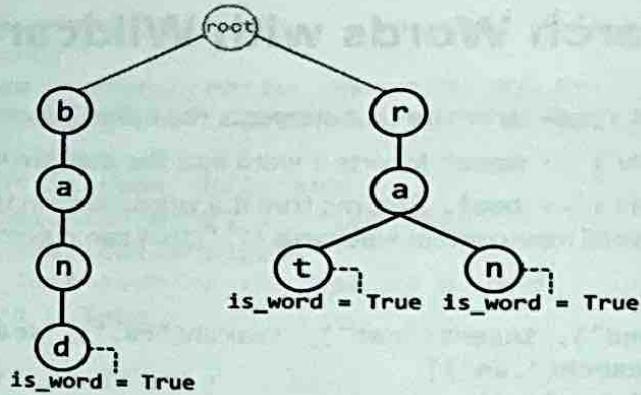
The requirements of this data structure closely resemble those of a trie, as it needs to facilitate the insertion and search of words. What makes this problem unique is the requirement to support wildcards ('.') in searches. Let's learn how we would need to modify our trie functions to support wildcards.

Inserting a word into the trie

The requirements for insertion in this problem match the requirements in a traditional trie. So, let's use the same implementation of `insert` as in the *Design a Trie* problem.

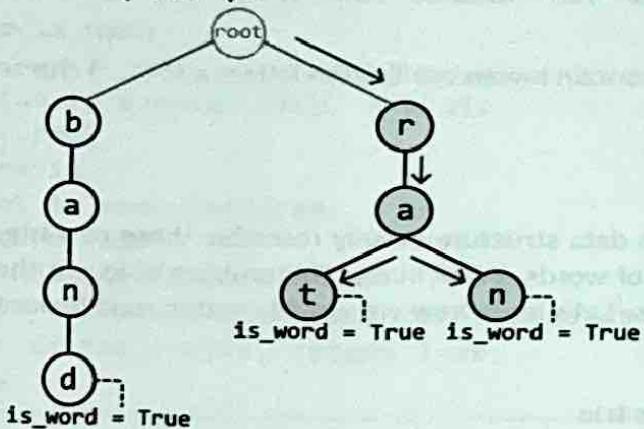
Searching with wildcards

What does it mean to encounter a wildcard? Consider the following trie:

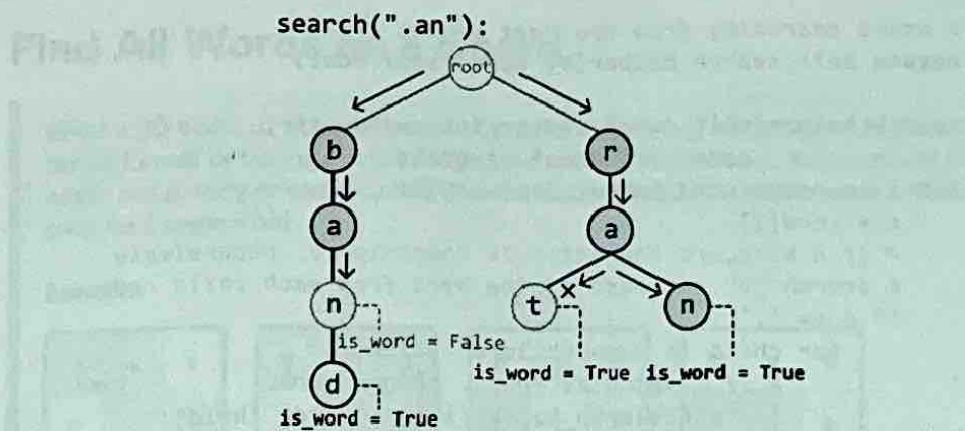


If we perform a search for "ra.", we know we can traverse down nodes 'r' and 'a' until we reach the wildcard character. Since the last letter is a wildcard, it could represent any letter. So, as long as there exists a node branching out from 'a' that represents the end of a word (i.e., has `is_word == True`), the word "ra." exists in the trie. In this case, both nodes 't' and 'n' meet the requirements of this wildcard:

```
search("ra."): 
```



Now, let's say we perform a search for ".an". Since the first character is a wildcard, we need to cover all branches starting from every node representing the first character in the trie. This means starting a search from each child node of the root. We can do this by recursively calling the search function on these nodes, passing in the substring "an" because this substring contains the remaining characters to be searched for.



Notice the nodes forming the string "ban" will not satisfy the search term because node 'n' does not mark the end of a word.

So, at any point in the search, we need to handle two scenarios:

1. When we encounter a letter, we proceed to the child of the current node that corresponds with this letter in the trie.
2. When we encounter a wildcard, we explore all child nodes, as the '.' may match any character. We can perform a recursive call for each child node to search for the remainder of the word.

This strategy for handling wildcards allows us to search every possible branch for a word that matches the search term. As soon as we find one branch that represents a word matching the search term, we return true.

Implementation

In this implementation, we use a helper function (`search_helper`) for searching because we need to pass in two extra parameters at each recursive call:

1. An index that defines the start of the remaining substring that needs to be searched. We pass in an index instead because passing in the substring itself would necessitate creating that substring, which would take linear time for each recursive call.
2. The `TrieNode` we're starting the search from. This ensures we don't restart each recursive call from the root node.

```

class InsertAndSearchWordsWithWildcards:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        node = self.root
        for c in word:
            if c not in node.children:
                node.children[c] = TrieNode()
            node = node.children[c]
        node.is_word = True

    def search(self, word: str) -> bool:

```

```

# Start searching from the root of the trie.
return self.search_helper(0, word, self.root)

def search_helper(self, word_index: int, word: str,
                 node: TrieNode) -> bool:
    for i in range(word_index, len(word)):
        c = word[i]
        # If a wildcard character is encountered, recursively
        # search for the rest of the word from each child node.
        if c == '.':
            for child in node.children.values():
                # If a match is found, return true.
                if self.search_helper(i + 1, word, child):
                    return True
            return False
        elif c in node.children:
            node = node.children[c]
        else:
            return False
    # After processing the last character, return true if we've
    # reached the end of a word.
    return node.is_word

```

Complexity Analysis

Time complexity:

- The time complexity of insert is $O(k)$, where k denotes the length of the word being inserted. This is because we traverse through or insert up to k nodes into the trie in each iteration.
- The time complexity of search is:
 - $O(k)$ when word contains no wildcards because we search through at most k characters in the trie.
 - $O(26^k)$ in the worst case, when word contains only wildcards. For each wildcard, we potentially need to explore up to 26 different characters (one for each lowercase English letter). With k wildcards, approximately 26^k recursive calls are made.

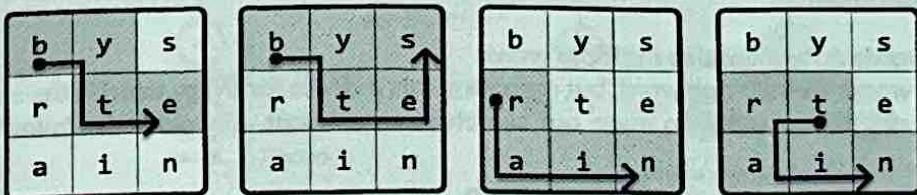
Space complexity:

- The space complexity of insert is $O(k)$ because in the worst case, the inserted word doesn't share a prefix with words already in the trie. In this case, k new nodes are created.
- The space complexity of search is:
 - $O(1)$ when word contains no wildcards.
 - $O(k)$ in the worst case when word contains only wildcards due to the space taken up by the recursive call stack, which can grow up to k in size.

Find All Words on a Board

Given a 2D board of characters and an array of words, find all the words in the array that can be formed by tracing a path through adjacent cells in the board. Adjacent cells are those which horizontally or vertically neighbor each other. We can't use the same cell more than once for a single word.

Example:



Input: board = [['b', 'y', 's'], ['r', 't', 'e'], ['a', 'i', 'n']],
words = ["byte", "bytes", "rat", "rain", "trait", "train"]
Output: ["byte", "bytes", "rain", "train"]

Intuition

There are many layers to this problem, so let's start by considering a simpler version where we're only required to search for one word on the board.

Simplified problem: words array contains one word

With only one word to find, a straightforward approach is to iterate through each cell of the board. If any cell contains the first letter of the word, perform a DFS from that cell in all four directions (left, right, up, down) to find the rest of the word. This process involves backtracking from a cell when we cannot find the next letter of the word in any of its adjacent cells. The process continues until the word is found, or we can no longer find any more letters on the board.

word = "stare"

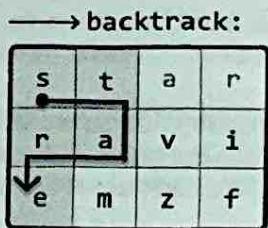
s	t	a	r
r	a	v	i
e	m	z	f

no 'e' in adjacent cells

→ backtrack:

s	t	a	r
r	a	v	i
e	m	z	f

no 'r' in adjacent cells



found it!

If you're not familiar with backtracking, it might be useful to review the Backtracking chapter before continuing with this problem.

Original problem - words array contains multiple words

The above approach works well for one word, but repeating this process for every word in the array is quite expensive. Let's devise a way to make our search more efficient. Consider the following board:

b	y	s
r	t	e
a	i	n

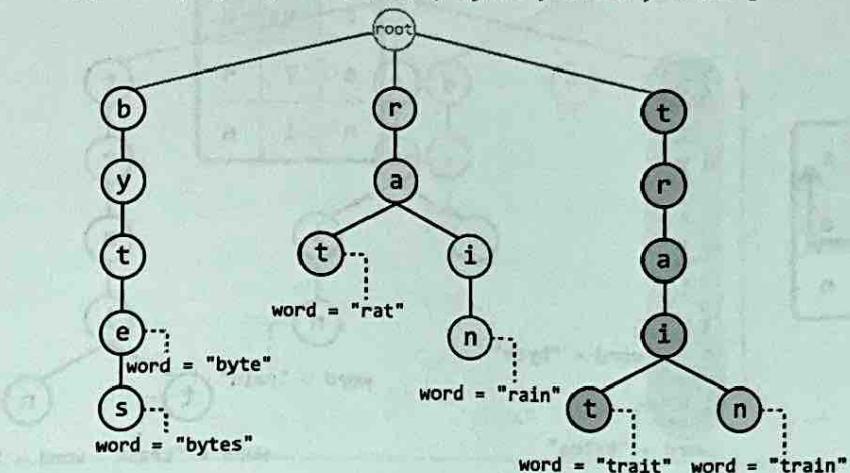
Let's say the words array contains the words "byte" and "bytes". Once we've found "byte", we'd ideally like to extend the search by just one more cell to also find the word "bytes":

b	y	s
r	t	e
a	i	n

However, with our initial algorithm, we'd need to restart the search entirely to find "bytes." This is quite inefficient. What we want is a data structure that allows us to efficiently search words with shared prefixes, allowing us to find multiple words without restarting the search for each of them. This is where the trie data structure comes into play, as it is excellent for managing prefixes.

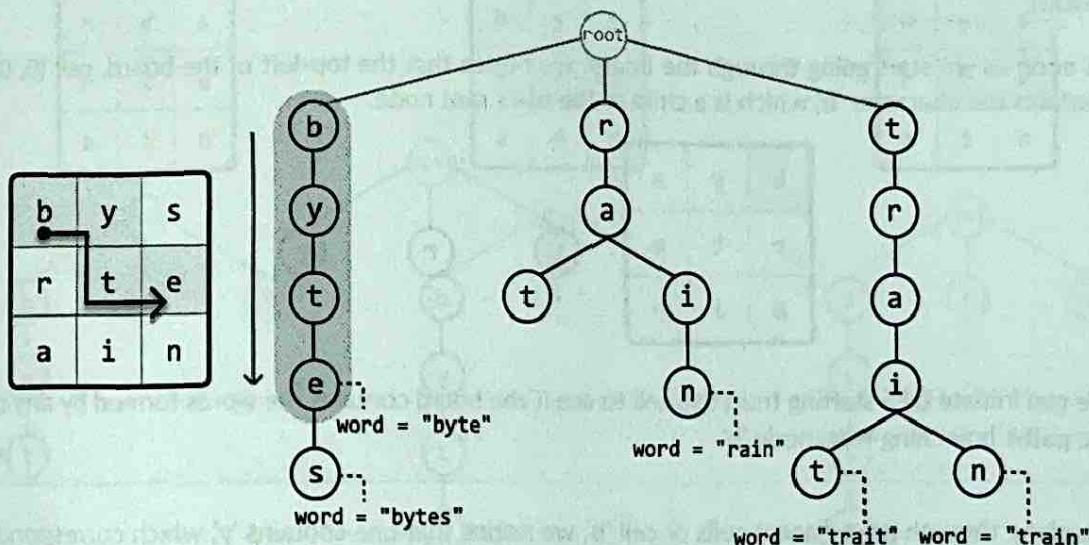
Let's begin creating the trie by inserting each word from the provided words array:

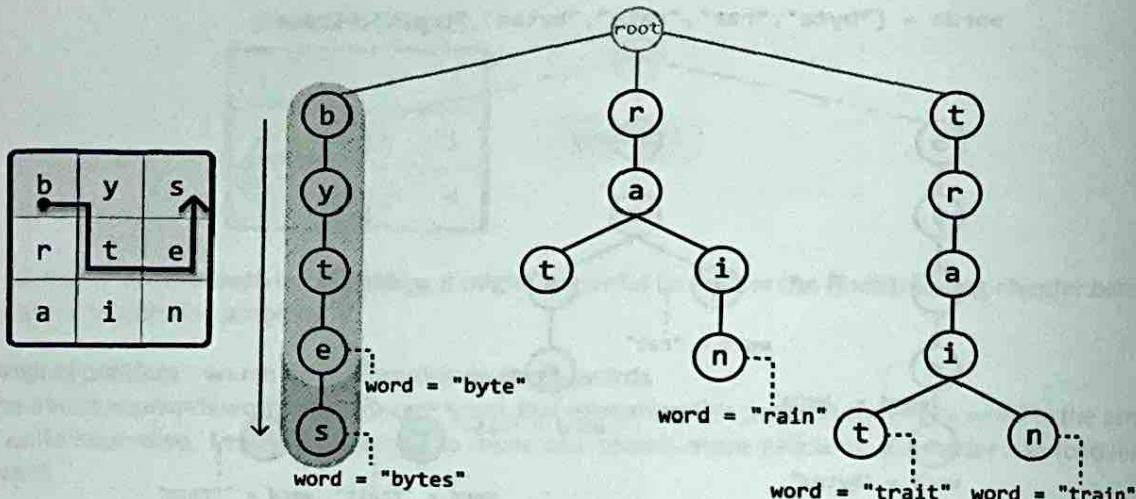
```
words = ["byte", "rat", "rain", "bytes", "trait", "train"]
```



In this chapter's introduction, we discussed two options to mark the end of a word in a `TrieNode`. Here, we use the `word` attribute instead of `is_word` to determine if a `TrieNode` represents the end of a word, and to know which specific word has ended. This will be important later.

Let's now use this trie to search for words over the board. We do this by seeing if any paths in the trie correspond with any paths on the board:



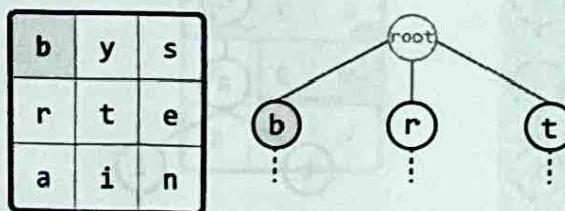


Similarly to how we used backtracking to search through the board when looking for a single word, we can also use backtracking here. Let's have a closer look at how this works.

Backtracking using a trie

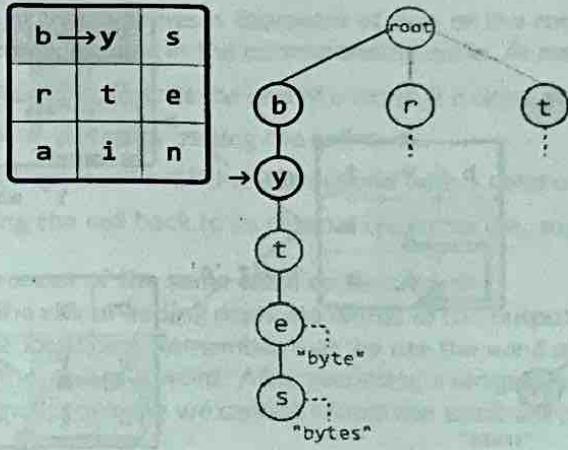
The first step is similar to what was discussed earlier: we go through the board until we find a cell whose character matches any of the root node's children in the trie, representing the first letter of a word.

As soon as we start going through the board, we notice that the top-left of the board, cell (0, 0), contains the character 'b', which is a child of the trie's root node.



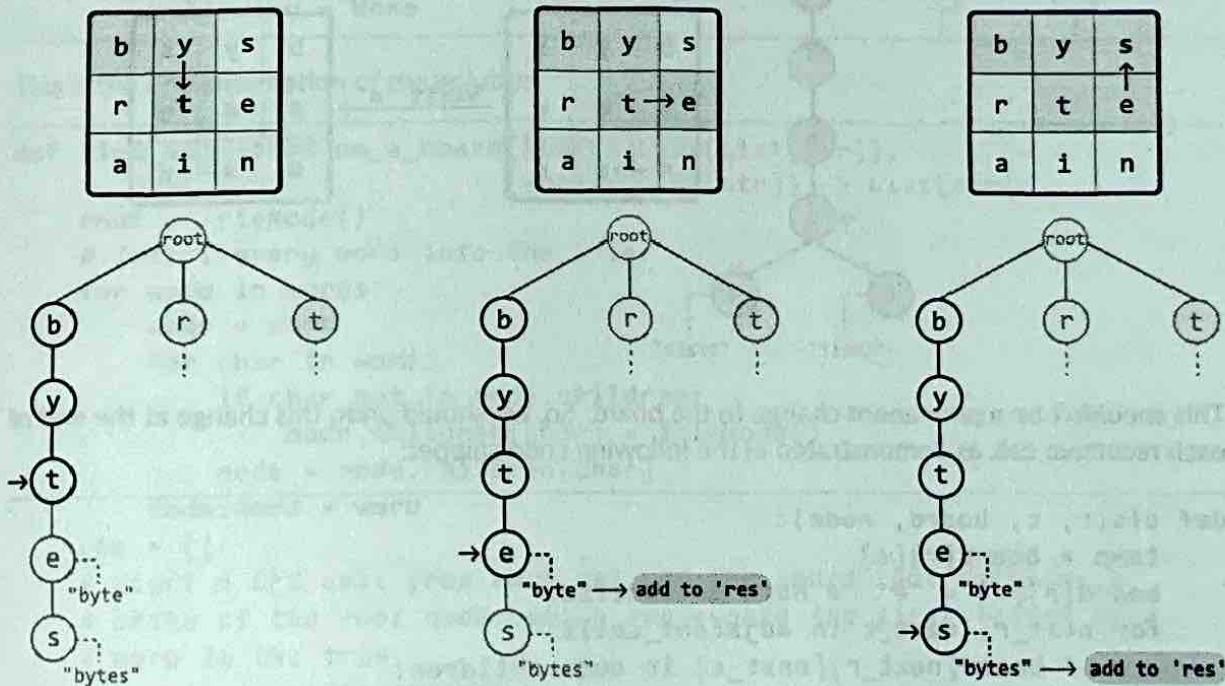
We can initiate DFS starting from this cell to see if the board contains the words formed by any of the paths branching from node 'b'.

Checking through the adjacent cells of cell 'b', we notice that one contains 'y', which corresponds to a child of node 'b'. So, let's make a DFS call to this cell to continue looking for the rest of this trie path on the board.



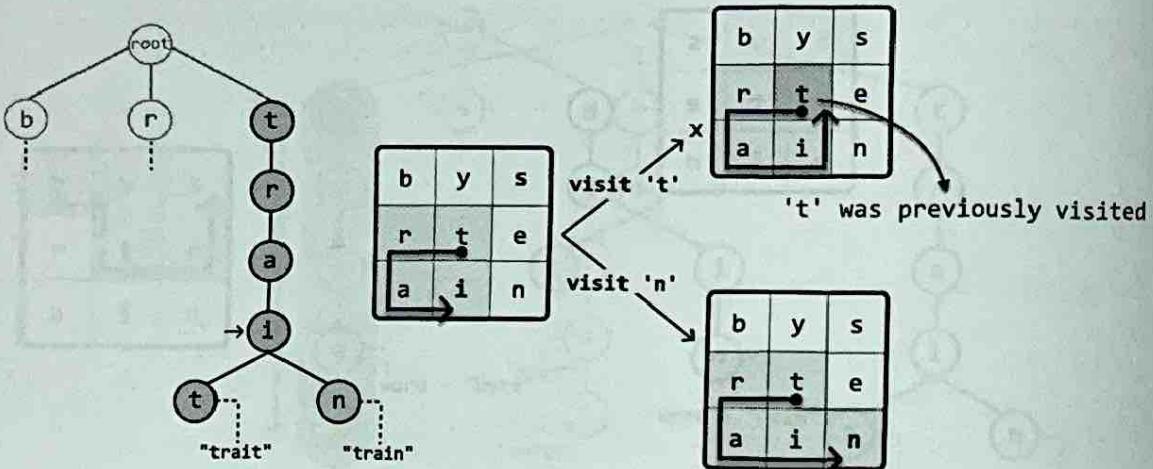
We continue this process until no more trie nodes can be found at an adjacent cell on the board. When this happens, we backtrack to the previous cell on the board to explore a different path.

If we ever reach a node that represents the end of the word (i.e., contains a non-null word attribute), we can record that word in our output:

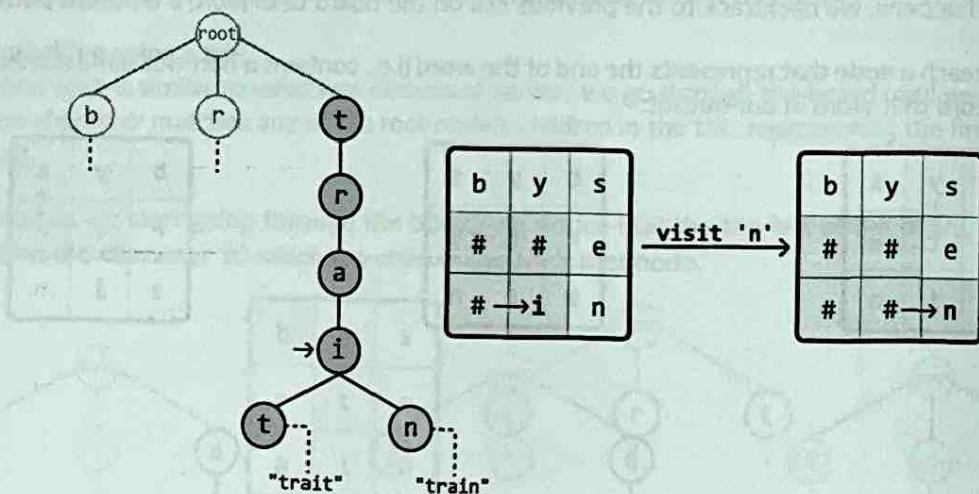


As shown, we found two words on the board from the DFS call that started at cell (0, 0). Now, we restart this process for any other cells on the board that match the character of one of the root node's children.

One important aspect of this backtracking approach is keeping track of visited cells as we explore the board. Without this, we might revisit a cell unintentionally. For example, when exploring both children of node 'i', we could end up revisiting cell 't':



The remedy for this is to either keep track of visited cells using a hash set, or keep track of them in place by changing the visited cell to a special character (like '#') as we traverse:



This shouldn't be a permanent change to the board. So, we should undo this change at the end of each recursive call, as demonstrated in the following code snippet:

```
def dfs(r, c, board, node):
    temp = board[r][c]
    board[r][c] = '#' # Mark as visited.
    for next_r, next_c in adjacent_cells:
        if board[next_r][next_c] in node.children:
            dfs(
                next_r,
                next_c,
                board,
                node.children[board[next_r][next_c]])
    board[r][c] = temp # Mark as unvisited.
```

Now, let's walk through this process in detail.

For each cell on the board that matches a character of one of the root node's children, make a recursive DFS call to that cell, passing in the corresponding node. At each of these DFS calls:

1. Check if the current node represents the end of a word. If it does, add that word to the output.
2. Mark the current cell as visited by setting the cell to '#'.
3. Recursively explore all adjacent cells that correspond with a child of the current TrieNode.
4. Backtrack by reverting the cell back to its original character (i.e., marking it as unvisited).

Handling multiple occurrences of the same word on the board

We need to be aware of the risk of adding duplicate words to the output, as the board may contain the same word in multiple locations. Remember that we use the `word` attribute on each `TrieNode` to check if it represents the end of a word. After recording a word in our output, we can set that node's `word` attribute to `None`, ensuring we cannot record the same word again.

Implementation

This is the implementation of the `TrieNode` class.

```
class TrieNode:  
    def __init__(self):  
        self.children = {}  
        self.word = None
```

This is the implementation of the solution.

```
def find_all_words_on_a_board(board: List[List[str]],  
                               words: List[str]) -> List[str]:  
    root = TrieNode()  
    # Insert every word into the trie.  
    for word in words:  
        node = root  
        for char in word:  
            if char not in node.children:  
                node.children[char] = TrieNode()  
            node = node.children[char]  
        node.word = word  
    res = []  
    # Start a DFS call from each cell of the board that contains a  
    # child of the root node, which represents the first letter of a  
    # word in the trie.  
    for r in range(len(board)):  
        for c in range(len(board[0])):  
            if board[r][c] in root.children:  
                dfs(board, r, c, root.children[board[r][c]], res)  
    return res  
  
def dfs(board: List[List[str]], r: int, c: int,  
       node: TrieNode, res: List[str]) -> None:  
    # If the current node represents the end of a word, add the word to  
    # the result.
```

```

if node.word:
    res.append(node.word)
    # Ensure the current word is only added once.
    node.word = None
temp = board[r][c]
# Mark the current cell as visited.
board[r][c] = '#'
# Explore all adjacent cells that correspond with a child of the
# current TrieNode.
dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]
for d in dirs:
    next_r, next_c = r + d[0], c + d[1]
    if (is_within_bounds(next_r, next_c, board)
        and board[next_r][next_c] in node.children):
        dfs(
            board, next_r, next_c,
            node.children[board[next_r][next_c]],
            res
        )
# Backtrack by reverting the cell back to its original character.
board[r][c] = temp

def is_within_bounds(r: int, c: int, board: List[str]) -> bool:
    return 0 <= r < len(board) and 0 <= c < len(board[0])

```

Complexity Analysis

Time complexity: The time complexity of `find_all_words_on_a_board` is $O(N \cdot L + m \cdot n \cdot 3^L)$ where N denotes the number of words in the words array, L denotes the length of the longest word, and $m \cdot n$ denotes the size of the board. Here's why:

- To build the trie, we insert each word from the input array into it, with each word containing a maximum of L characters. This takes $O(N \cdot L)$ time.
- Then, in the main search process, we perform a DFS for each of the $m \cdot n$ cells on the board. Each DFS call takes $O(3^L)$ time because, at each point in the DFS, we make up to 3 recursive calls: one for each of the 3 adjacent cells (this excludes the cell we came from). This is repeated for, at most, the length of the longest word, L .

Therefore, the overall time complexity is $O(N \cdot L) + m \cdot n \cdot O(3^L) = O(N \cdot L + m \cdot n \cdot 3^L)$.

Space complexity: The space complexity is $O(N \cdot L)$. Here's why:

- The trie has a space complexity of $O(N \cdot L)$. In the worst case, if all words have unique prefixes, we store every character of every word in the trie. Each word attribute stored at the end of a path in the trie takes $O(L)$ space, and with N words. This contributes an additional $O(N \cdot L)$ space.
- The maximum depth of the recursive call stack is L .

Therefore, the overall space complexity is $O(N \cdot L) + O(L) = O(N \cdot L)$.

Graphs

Introduction to Graphs

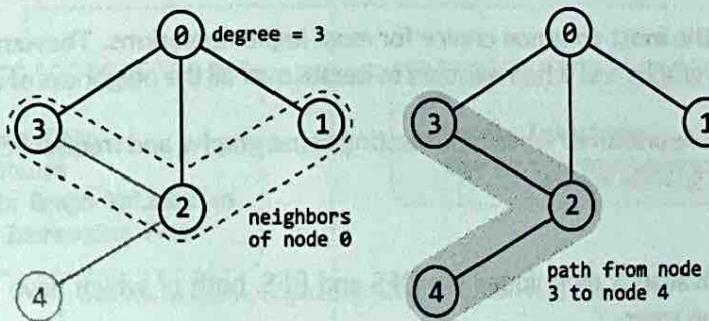
A graph is a data structure composed of nodes (vertices) connected by edges. Graphs are used to model relationships, where the edges define the relationships.

Below is the implementation of the `GraphNode` class:

```
class GraphNode:
    def __init__(self, val):
        self.val = val
        self.neighbors = []
```

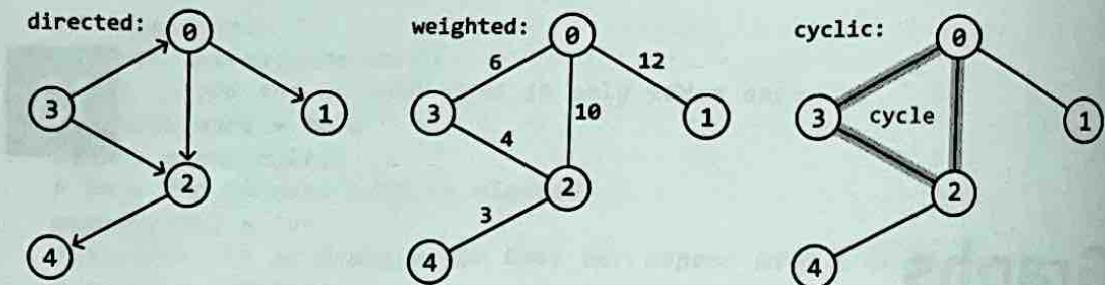
Terminology:

- Adjacent node/neighbor: two nodes are adjacent if there's an edge connecting them.
- Degree: the number of edges connected to a node.
- Path: a sequence of nodes connected by edges.



Attributes:

- Directed vs. undirected: in a directed graph, edges have a direction associated with them.
- Weighted vs. unweighted: in a weighted graph, edges have a weight associated with them, such as distance or cost.
- Cyclic vs. acyclic: A cyclic graph contains at least one cycle, which is a path that starts and ends at the same node.

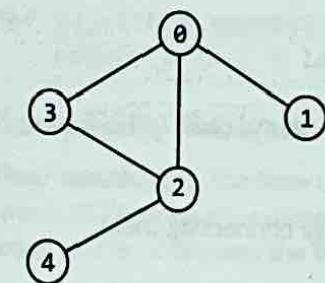


Representations

In some problems, you might not be given the graph directly. In these situations, it's usually necessary to create your own representation of the graph. The two most common representations to choose from are an adjacency list and an adjacency matrix.

In an **adjacency list**, the neighbors of each node are stored as a list. Adjacency lists can be implemented using a hash map, where the key represents the node, and its corresponding value represents the list of that node's neighbors.

In an **adjacency matrix**, the graph is represented as a 2D matrix where `matrix[i][j]` indicates an edge between nodes *i* and *j*.



adjacency list:

node	neighbors
0	[1 2 3]
1	[0]
2	[0 3 4]
3	[0 2]
4	[2]

adjacency matrix:

0	1	2	3	4	
0	0	1	1	1	0
1	1	0	0	0	0
2	1	0	0	1	1
3	1	0	1	0	0
4	0	0	1	0	0

Adjacency lists are the most common choice for most implementations. They are preferred when representing sparse graphs and when we need to iterate over all the neighbors of a node efficiently.

Adjacency matrices are preferred when representing dense graphs, and frequent checks are needed for the existence of specific edges.

Traversals

The primary graph traversal techniques are DFS and BFS, both of which have similar use cases when they're used on trees.

When traversing a graph, it might also be necessary to keep track of visited nodes using a data structure such as a hash set, to ensure each node is only visited once.

DFS is typically implemented recursively, as demonstrated in the code snippet below:

```
def dfs(node: GraphNode, visited: Set[GraphNode]):
    visited.add(node)
    process(node)
    for neighbor in node.neighbors:
```

```
if neighbor not in visited:  
    dfs(neighbor, visited)
```

BFS is typically implemented iteratively, as demonstrated in the code snippet below:

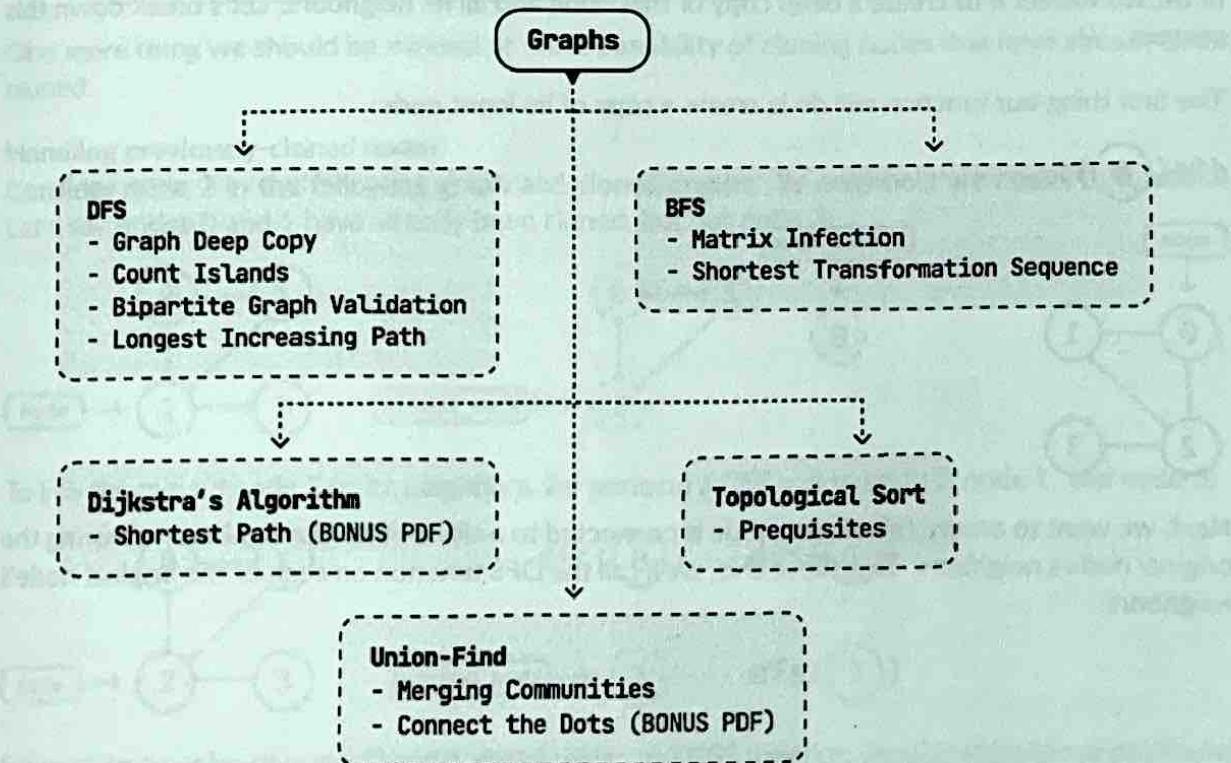
```
def bfs(node: GraphNode):  
    visited = set()  
    queue = deque([node])  
    while queue:  
        node = queue.popleft()  
        if node not in visited:  
            visited.add(node)  
            process(node)  
            for neighbor in node.neighbors:  
                queue.append(neighbor)
```

Both DFS and BFS have a time complexity of $O(n + e)$, where n denotes the number of nodes and e denotes the number of edges. This is because during traversal, each node is visited once, and each edge is explored once. They both also share a space complexity of $O(n)$. For DFS, this is due to the space taken up by the recursive call stack, and for BFS, it's due to the space taken up by the queue.

Real-world Example

Social networks: Users of social media sites like LinkedIn are typically represented as nodes, and connections or friendships between users are represented as edges. The graph structure allows platforms to analyze relationships, suggest new connections, and identify groups or communities within their networks.

Chapter Outline

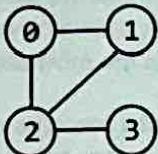


Graph Deep Copy

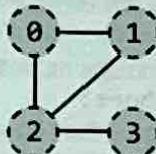
Given a reference to a node within an undirected graph, create a deep copy (clone) of the graph. The copied graph must be completely independent of the original one. This means you need to make new nodes for the copied graph instead of reusing any nodes from the original graph.

Example:

Original Graph:



Cloned Graph:



All nodes are deep copies:
no references to the
original graph.

Constraints:

- The value of each node is unique.
- Every node in the graph is reachable from the given node.

Intuition

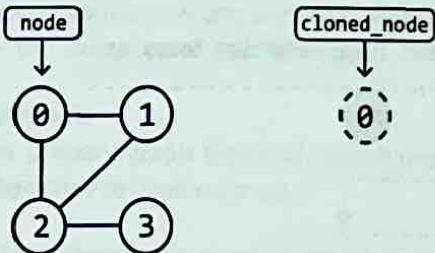
Our strategy for this problem is to traverse the original graph and create the deep copy during the traversal, effectively cloning each node while we traverse. Any traversal method will suffice for this strategy. In this explanation, we'll use **DFS**.

Traversing the graph

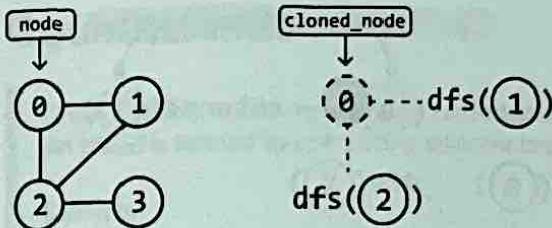
Start by defining exactly what we want our DFS function to do. When we call DFS on the input node, we expect it to create a deep copy of that node and all its neighbors. Let's break down this process.

The first thing our function will do is create a copy of its input node:

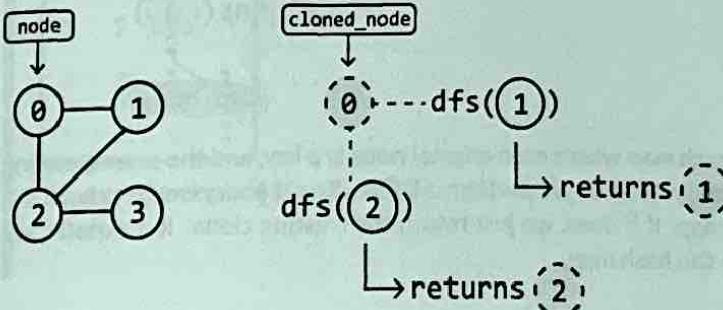
`dfs(0):`



Next, we want to ensure this cloned node is connected to a clone of all its neighbors, mirroring the original node's neighbors. To achieve this, we'll call the DFS function on each of the original node's neighbors:



Each of these DFS instances will also do the same thing by creating a clone of their input node and returning it when it has been connected to its neighbors:



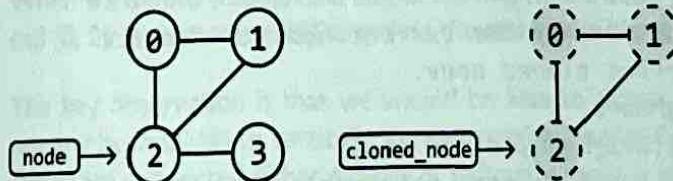
In pseudocode, this is what the process looks like:

```
dfs(node):
    cloned_node = new GraphNode(node)
    for neighbor in node.neighbors:
        cloned_neighbor = dfs(neighbor)
        cloned_node.neighbors.add(cloned_neighbor)
    return cloned_node
```

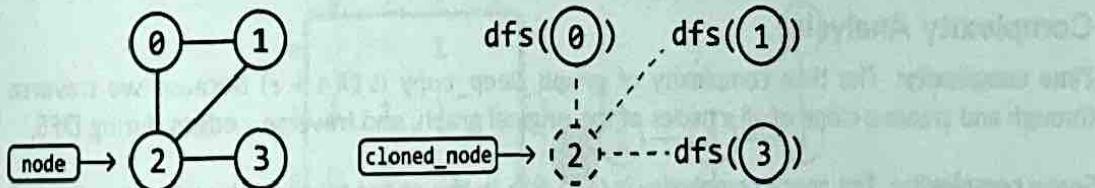
One more thing we should be mindful of is the possibility of cloning nodes that have already been cloned.

Handling previously-cloned nodes

Consider node 2 in the following graph and cloned graphs. Its neighbors are nodes 0, 1, and 3. Let's say nodes 0 and 1 have already been cloned, but not node 3:

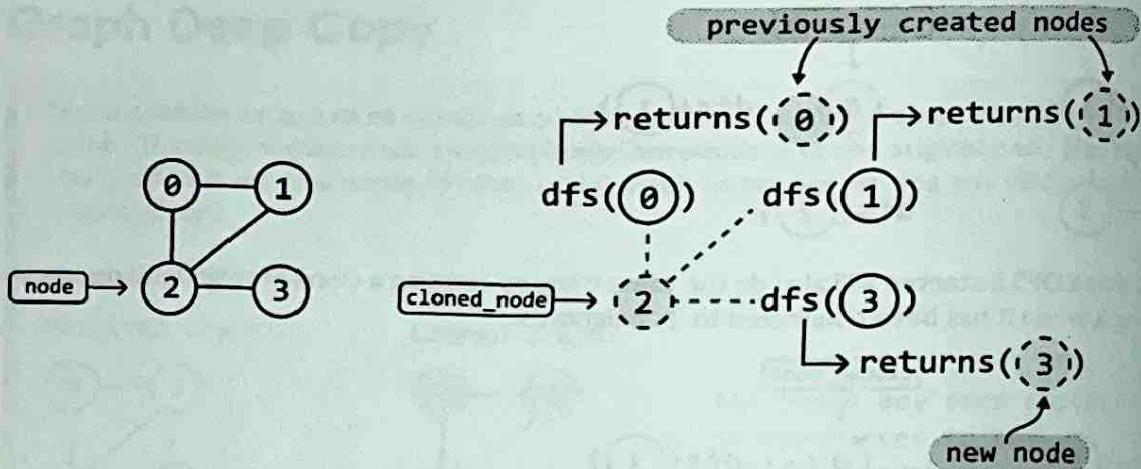


To link the cloned node 2 to its neighbors, we perform a DFS call to node 0, node 1, and node 3:



Since cloned copies of nodes 0 and 1 already exist, our DFS function should return these previously created nodes, instead of creating new ones:

Graph Deep Copy



We can manage this by using a **hash map** where each original node is a key, and the corresponding cloned node is the value. This way, whenever we perform a DFS call on a node, we first check if it already has a clone in our hash map. If it does, we just return the existing clone. If it doesn't, we create a new clone and add it to the hash map.

Implementation

```
def graph_deep_copy(node: GraphNode) -> GraphNode:
    if not node:
        return None
    return dfs(node)

def dfs(node: GraphNode, clone_map = {}) -> GraphNode:
    # If this node was already cloned, then return this previously
    # cloned node.
    if node in clone_map:
        return clone_map[node]
    # Clone the current node.
    cloned_node = GraphNode(node.val)
    # Store the current clone to ensure it doesn't need to be created
    # again in future DFS calls.
    clone_map[node] = cloned_node
    # Iterate through the neighbors of the current node to connect
    # their clones to the current cloned node.
    for neighbor in node.neighbors:
        cloned_neighbor = dfs(neighbor, clone_map)
        cloned_node.neighbors.append(cloned_neighbor)
    return cloned_node
```

Complexity Analysis

Time complexity: The time complexity of `graph_deep_copy` is $O(n + e)$ because we traverse through and create a clone of all n nodes of the original graph, and traverse e edges during DFS.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the recursive call stack, which can grow as large as n . In addition, the `clone_map` hash map stores a key-value pair for each of the n nodes.

Count Islands

Given a binary matrix representing 1s as land and 0s as water, return the number of islands. An island is formed by connecting adjacent lands 4-directionally (up, down, left, and right).

Example:

island			
1	1	0	0
1	1	0	0
0	0	1	1
0	0	0	1

Input: `matrix = [[1, 1, 0, 0], [1, 1, 0, 0], [0, 0, 1, 1], [0, 0, 0, 1]]`
Output: 2

Intuition

Before determining how to find all the islands in a matrix, let's consider an input that contains only one island.

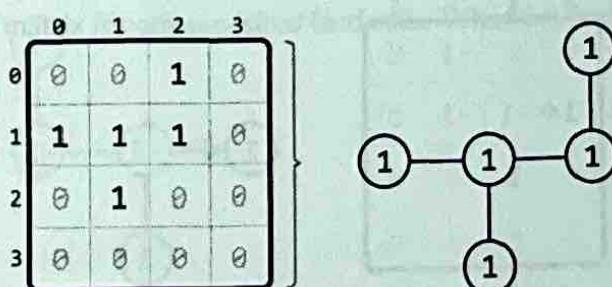
Matrix with just one island

Consider the following matrix containing one island:

0	1	2	3	
0	0	0	1	0
1	1	1	1	0
2	0	1	0	0
3	0	0	0	0

When we iterate through the matrix starting from the top left, the first land cell we encounter is cell (0, 2). From here, we'd like to find the rest of the island.

The key observation is that we should be able to access every land cell on the same island by moving horizontally or vertically through neighboring land cells. This means that all 1s forming an island are connected either directly or indirectly through adjacent 1s. Conceptually, this is similar to a graph, where each cell is a node, and each connection to an adjacent cell is an edge:



This demonstrates that we can identify the rest of the island by performing a graph traversal algorithm. Most traversal algorithms will suit this purpose. In this explanation, we'll use DFS.

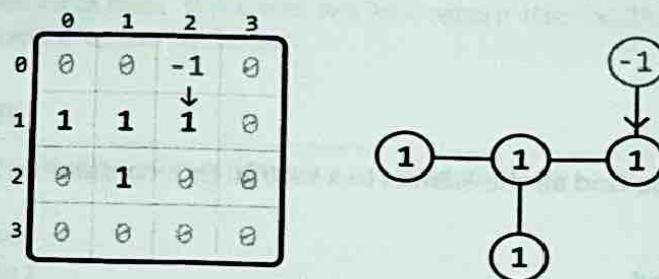
Depth-first search

For each cell we visit during the traversal, we need to mark that cell as visited to ensure it's not visited again. There are two ways to do this:

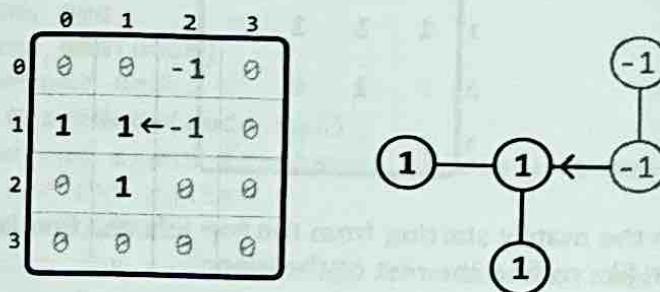
1. Use a separate data structure, such as a hash set, to keep track of the coordinates of visited cells.
2. Modify the matrix by changing the value of a visited cell from 1 to -1, ensuring it doesn't get revisited.

We'll proceed with the second option because it doesn't require the use of extra space.

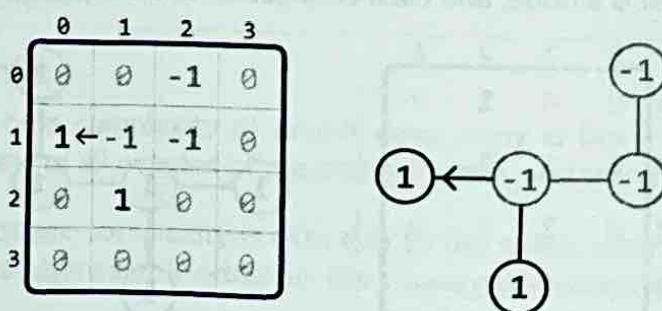
Now, let's begin DFS traversal. Mark the first cell, (0, 2), as visited by modifying its value to -1. Then, continue exploring by calling DFS on any neighboring land cells. Here, the only neighboring land cell is cell (1, 2):



From cell (1, 2), we similarly mark it as visited and explore its neighboring land cells. Again, there's only one neighboring land cell: (1, 1). So, let's make a recursive DFS call to it:



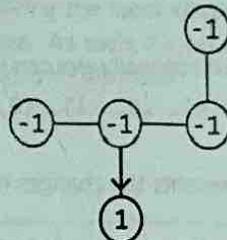
From cell (1, 1), mark it as visited and explore both of its neighboring land cells. Let's continue exploring from cell (1, 0) first:



At (1, 0), there's no neighboring land. So, the recursive process naturally goes back to cell (1, 1) to continue exploring any other neighboring land cells:

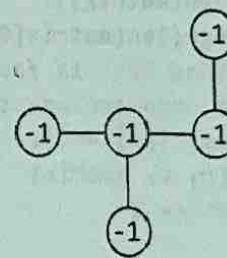
0	1	2	3	
0	0	0	-1	0
1	-1	-1	-1	0
2	0	1	0	0
3	0	0	0	0

0	1	2	3	
0	0	0	-1	0
1	-1	-1	-1	0
2	0	1	0	0
3	0	0	0	0



We've now finished exploring this island:

0	1	2	3	
0	0	0	-1	0
1	-1	-1	-1	0
2	0	-1	0	0
3	0	0	0	0

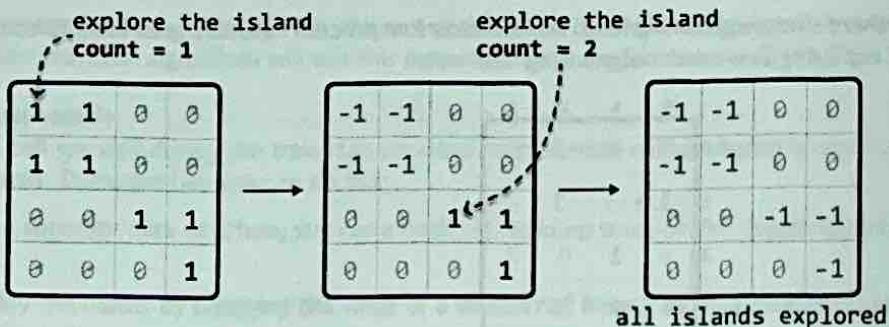


With this island completely explored, let's increment a variable count to indicate that one new island has been found. Now, let's consider the main problem where there could be multiple islands in the matrix.

Matrix with multiple islands

We can identify all islands using the following steps:

1. Search through the matrix, starting from cell (0, 0), until we find a land cell.
2. Upon encountering a land cell, explore its island using DFS, marking each land cell we encounter as visited (-1) to avoid visiting them again.
3. Increment count by 1, indicating the discovery of the island we just explored.
4. Keep searching the matrix for any unvisited land cells. When we find one, repeat steps 2 to 4.



Implementation

To traverse the matrix 4-directionally, we can use an array of direction vectors:

```
dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]
```

Each pair in the array represents the changes needed to move one step in a specific direction:

```
def count_islands(matrix: List[List[int]]) -> int:
    if not matrix:
        return 0
    count = 0
    for r in range(len(matrix)):
        for c in range(len(matrix[0])):
            # If a land cell is found, perform DFS to explore the full
            # island, and include this island in our count.
            if matrix[r][c] == 1:
                dfs(r, c, matrix)
                count += 1
    return count

def dfs(r: int, c: int, matrix: List[List[int]]) -> None:
    # Mark the current land cell as visited.
    matrix[r][c] = -1
    # Define direction vectors for up, down, left, and right.
    dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    # Recursively call DFS on each neighboring land cell to continue
    # exploring this island.
    for d in dirs:
        next_r, next_c = r + d[0], c + d[1]
        if (is_within_bounds(next_r, next_c, matrix)
            and matrix[next_r][next_c] == 1):
            dfs(next_r, next_c, matrix)

def is_within_bounds(r: int, c: int, matrix: List[List[int]]) -> bool:
    return 0 <= r < len(matrix) and 0 <= c < len(matrix[0])
```

Complexity Analysis

Time complexity: The time complexity of `count_islands` is $O(m \cdot n)$, where m denotes the number of rows and n denotes the number of columns. This is because each cell of the matrix is visited

at most twice: once when searching for land cells in the `count_islands` function, and up to one more time during DFS.

Space complexity: The space complexity is $O(m \cdot n)$ mostly due to the recursive call stack during DFS, which can grow up to $m \cdot n$ in size.

Interview Tip

Tip: Check with an interviewer if modifications to the input are acceptable.



During DFS, we marked cells as visited by modifying the input directly. However, in some situations, input modification may not be desirable. As such, it's worth confirming this with the interviewer before making in-place modifications to the input.

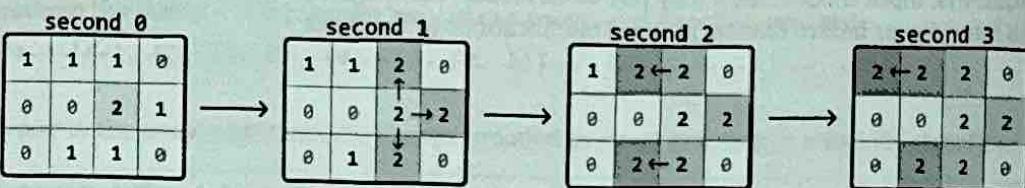
Matrix Infection

You are given a matrix where each cell is either:

- 0: Empty
- 1: Uninfected
- 2: Infected

With each passing second, every infected cell (2) infects its uninfected neighboring cells (1) that are 4-directionally adjacent. Determine the number of seconds required for all uninfected cells to become infected. If this is impossible, return -1.

Example:



Input: matrix = [[1, 1, 1, 0], [0, 0, 2, 1], [0, 1, 1, 0]]
Output: 3

Intuition

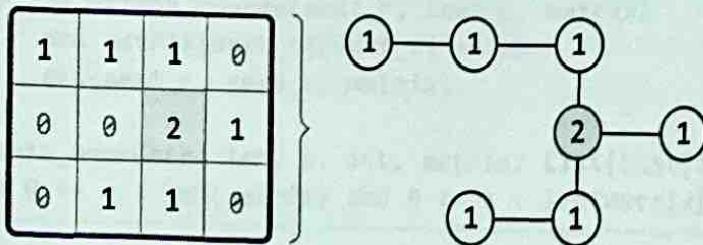
Let's begin tackling this problem by considering a simple case where the initial matrix contains only one infected cell.

Matrix with one infected cell

Consider the following matrix, containing just one 2:

1	1	1	0
0	0	2	1
0	1	1	0

An observation is that each infected (2) and uninfected (1) cell can be considered as nodes in a graph, where edges exist between cells that are 4-directionally adjacent. Therefore, we can visualize these cells as a connected graph:

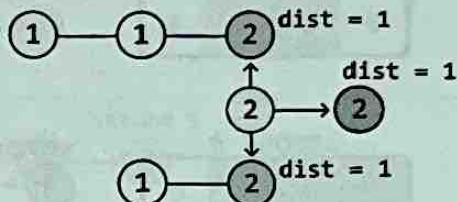


This helps us think about this problem as a graph traversal problem. Which traversal algorithm will allow us to simulate the infection process? To find out, let's observe how cells get infected each

second.

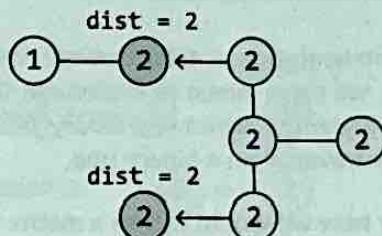
After the first second, the adjacent uninfected neighbors of the first infected cell become infected. These cells are a distance of 1 away from the initially infected cell:

1	1	2	0
0	0	2 → 2	
0	1	2	0



One second later, the neighbors of the most recently infected cells get infected. These cells are a distance of 2 from the initial infected cell:

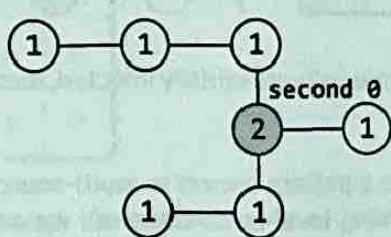
1	2 ← 2	2	0
0	0	2	2
0	2 ← 2	2	0



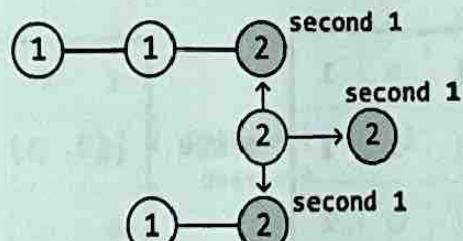
As we can see, the outward expansion from the initially infected cell is similar to how level-order traversal works in a tree, where each level represents nodes that are at a specific distance from the initially infected node.

So, let's perform a level-order traversal to infect cells, starting at the infected cell. Each level that gets traversed corresponds to 1-second passing in the infection process.

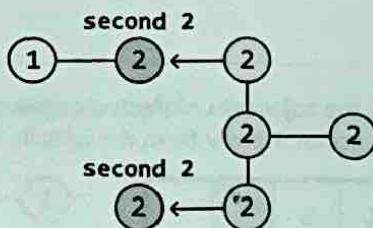
second 0			
1	1	1	0
0	0	2	1
0	1	1	0



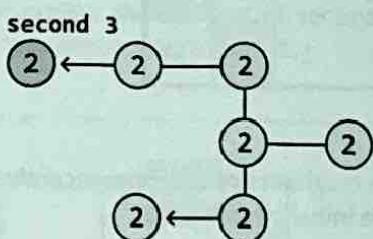
second 1			
1	1	2	0
0	0	2 → 2	
0	1	2	0



second 2				
1	2 ← 2	2	2	0
0	0	2	2	
0	2 ← 2	2	0	



second 3				
2 ← 2	2	2	0	
0	0	2	2	
0	2	2	0	



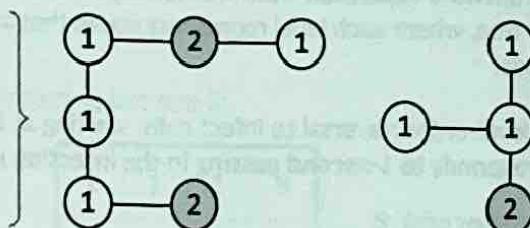
Regarding the implementation of this traversal, we know that level-order traversal is a modified version of BFS. So, we use a queue to implement this traversal. If you're unfamiliar with how this works, review the *Rightmost Nodes of a Binary Tree* problem from the *Tree* chapter, which implements a level-order traversal on a binary tree.

Now, let's consider how we would handle a matrix which initially contains multiple infected cells.

Matrix with multiple infected cells

Consider the following example and its corresponding graph visualization:

1	2	1	0	1
1	0	0	1	1
1	2	0	0	2



In this example, multiple cells are initially infected, meaning there are multiple cells at level 0 of the traversal.

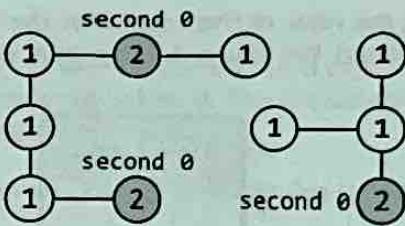
To handle this, we use a pattern known as **multi-source BFS**. Instead of adding just one cell to the queue before performing level-order traversal, we add every initially infected cell to the queue. This way, the traversal starts with all initially infected cells as level 0, allowing the infection process to begin simultaneously from multiple starting points:

0	1	2	3	4	
0	1	2	1	0	1
1	1	0	0	1	1
2	1	2	0	0	2

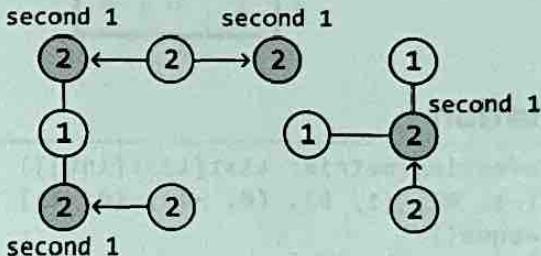
queue = [(0, 1) (2, 1) (2, 4)]

When we start with multiple cells in the queue, we can see what the level order process looks like:

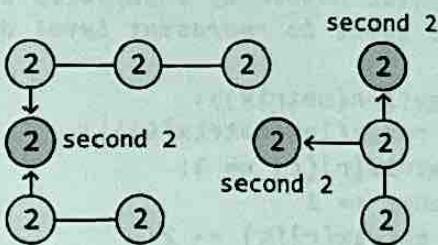
second 0				
1	2	1	0	1
1	0	0	1	1
1	2	0	0	2



second 1				
2 ← 2 → 2	0	1	1	1
1	0	0	1	2
2 ← 2	0	0	0	2



second 2				
2	2	2	0	2
2	0	0	2 ← 2	1
2	2	0	0	2



Unreachable uninfected cells

It's important to keep in mind that it's not always possible to infect all uninfected cells. We could encounter situations where it's impossible to reach an uninfected cell, such as in the following example:

this will never
be infected

1	2	1
0	0	0
1	0	0

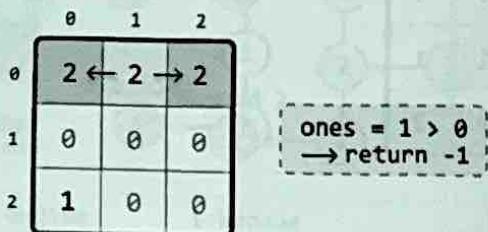
One way to account for this is to search through the matrix after level-order traversal and check if any 1s remain. However, there's a cleaner way to accomplish this. First, in the loop where we search for all the level 0 infected cells, we can also count how many 1s there are:

0	1	2	
0	1	2	1
1	0	0	0
2	1	0	0

queue = [(0, 1)]
ones = 3

Then, as we perform the level-order traversal, we can decrement this count for each uninfected

cell we infect. This way, the value of this count after the traversal will represent the number of cells that remained uninfected. We return -1 if this count is greater than 0:



Implementation

```
def matrix_infection(matrix: List[List[int]]) -> int:
    dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    queue = deque()
    ones = seconds = 0
    # Count the total number of uninfected cells and add each infected
    # cell to the queue to represent level 0 of the level-order
    # traversal.
    for r in range(len(matrix)):
        for c in range(len(matrix[0])):
            if matrix[r][c] == 1:
                ones += 1
            elif matrix[r][c] == 2:
                queue.append((r, c))
    # Use level-order traversal to determine how long it takes to
    # infect the uninfected cells.
    while queue and ones > 0:
        # 1 second passes with each level of the matrix that's explored.
        seconds += 1
        for _ in range(len(queue)):
            r, c = queue.popleft()
            # Infect any neighboring 1s and add them to the queue to be
            # processed in the next level.
            for d in dirs:
                next_r, next_c = r + d[0], c + d[1]
                if (is_within_bounds(next_r, next_c, matrix)
                    and matrix[next_r][next_c] == 1):
                    matrix[next_r][next_c] = 2
                    ones -= 1
                    queue.append((next_r, next_c))
    # If there are still uninfected cells left, return -1. Otherwise,
    # return the time passed.
    return seconds if ones == 0 else -1

def is_within_bounds(r: int, c: int, matrix: List[List[int]]) -> bool:
    return 0 <= r < len(matrix) and 0 <= c < len(matrix[0])
```

Complexity Analysis

Time complexity: The time complexity of `matrix_infection` is $O(m \cdot n)$, where m denotes the number of rows, and n denotes the number of columns. This is because in the worst case, every cell in the matrix is explored during level-order traversal.

Space complexity: The space complexity is $O(m \cdot n)$, primarily due to the queue, which can store up to $m \cdot n$ cells.

Given a grid of size $m \times n$, we require space proportional to $m \cdot n$ to store the queue. We also need to store the current state of each cell in the grid, which requires $m \cdot n$ additional cells. Thus, the total space complexity is $O(m \cdot n)$.

Implementation: Although writing a generic `Matrix` class and `infect` function is a good idea, it will add unnecessary code and complexity. Instead, we can directly use the `grid` variable to store the matrix and the `queue` variable to store the cells to be processed. This makes the implementation cleaner and easier to understand.

Let's implement this algorithm using the concepts learned so far in this chapter.



Implementation: Implementing this algorithm is relatively straightforward. We start by defining a `Matrix` class to store the grid and a `queue` to store the cells to be processed.



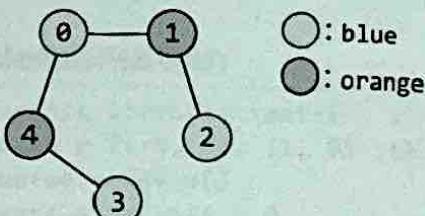
Implementation: Let's implement this algorithm. Note that node 0, having value 0, needs to be processed first. To do this, we will use a `queue` to store the cells to be processed.

Bipartite Graph Validation

Given an undirected graph, determine if it's bipartite. A graph is bipartite if the nodes can be colored in one of two colors, so that no two adjacent nodes are the same color.

The input is presented as an adjacency list, where `graph[i]` is a list of all nodes adjacent to node `i`.

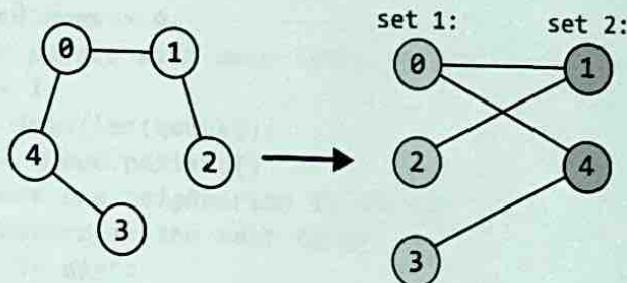
Example:



Input: `graph = [[1, 4], [0, 2], [1], [4], [0, 3]]`
Output: True

Intuition

Before diving into a solution, let's first understand what makes a graph bipartite. If the nodes of a graph can be divided into two distinct sets, with the edges only running between nodes from different sets, then the graph is bipartite. We can visually rearrange the nodes of the following graph to demonstrate this:



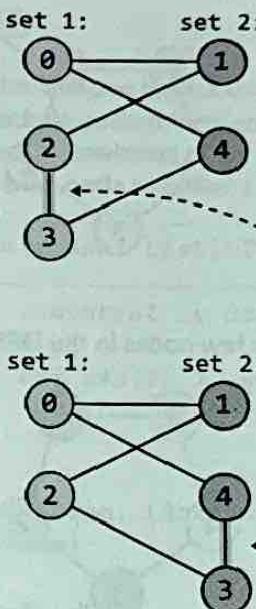
With a graph that isn't bipartite, it's impossible to arrange the nodes this way without an edge existing between two nodes of the same set:

Implementation

In the implementation, you will need to keep track of which nodes have been visited, which nodes are currently being explored, and which nodes are still unexplored.

You will also need to keep track of which nodes belong to which set, and whether there are any edges between nodes from different sets.

Finally, you will need to implement the logic for validating the bipartite property of the graph.



To determine if a graph is bipartite, we can use graph coloring, where we attempt to color one set of nodes with one color, and the other set of nodes with another color, while ensuring no adjacent nodes share the same color. Let's explore this idea.

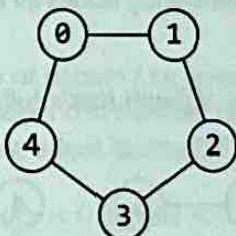
Graph coloring

Let's use blue and orange in our coloring process. One potential strategy is: **for each node we color blue, color all of its neighbors orange, and vice versa.** Most traversal algorithms allow us to color neighboring nodes in this way. In this explanation, we use **DFS**.

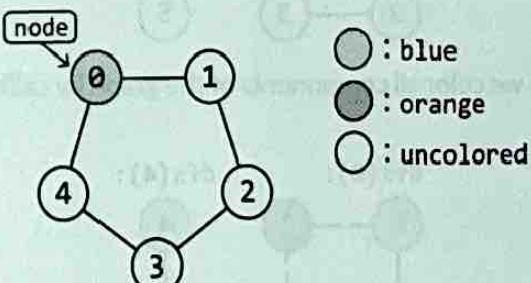
Let's try applying this coloring process to the example below and see how it works:

Complexity Analysis

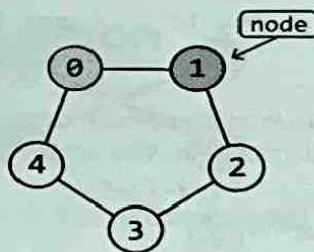
This example illustrates the time complexity of the algorithm. We start at node 0 and explore all of its neighbors. Then we move to node 1 and explore all of its neighbors. Finally, we move to node 2 and explore all of its neighbors.



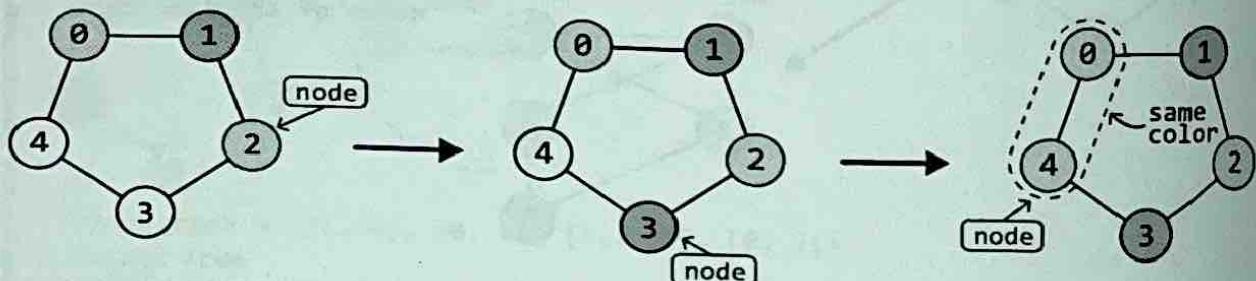
This example illustrates the time complexity of the algorithm. We start at node 0 and explore all of its neighbors. Then we move to node 1 and explore all of its neighbors. Finally, we move to node 2 and explore all of its neighbors.



Using DFS, we explore the neighbors of node 0, starting with node 1. All of its neighbors need to be colored orange, so let's make a DFS call to node 1 to color it orange:



Let's continue doing this for the next few nodes in the DFS process:

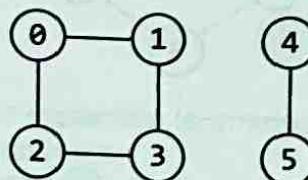


Above, we encountered an issue. We needed to color node 4 blue, but one of node 4's neighbors, node 0, is also colored blue. This means the graph cannot be colored using our graph coloring strategy. In other words, the graph is not bipartite.

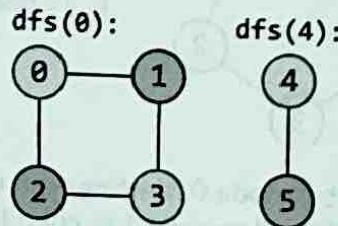
We now have a strategy to confirm if a graph is bipartite using the two-coloring technique. But how can we be sure that it always works? We've been coloring adjacent nodes in alternating colors from the beginning of the DFS. This means if we encounter a situation where two adjacent nodes are the same color, it means there's no way to color the graph differently, since we've been following the rule of using different colors for neighboring nodes all along.

Handling multiple components

Keep in mind the input isn't necessarily a graph that's fully connected. It could be a graph with multiple components, such as this:



As such, we need to ensure we color all components of the graph by calling DFS on every uncolored node:



If we can confirm that all components can be colored using two colors, the graph is bipartite. However, if any of these components cannot be colored this way, the graph is not bipartite.

Implementation

In this implementation, we color the nodes blue and orange using the numbers 1 and -1 to represent these colors, respectively. To keep track of each node's color, we use an array called `colors`, initialized with all 0s, where 0 represents an unvisited node. As we explore the graph using DFS, we update the `colors` array by setting each node to either 1 (blue) or -1 (orange).

```
def bipartite_graph_validation(graph: List[List[int]]) -> bool:
    colors = [0] * len(graph)
    # Determine if each graph component is bipartite.
    for i in range(len(graph)):
        if colors[i] == 0 and not dfs(i, 1, graph, colors):
            return False
    return True

def dfs(node: int, color: int, graph: List[List[int]],
       colors: List[int]) -> bool:
    colors[node] = color
    for neighbor in graph[node]:
        # If the current neighbor has the same color as the current
        # node, the graph is not bipartite.
        if colors[neighbor] == color:
            return False
        # If the current neighbor is not colored, color it with the
        # other color and continue the DFS.
        if (colors[neighbor] == 0
            and not dfs(neighbor, -color, graph, colors)):
            return False
    return True
```

Complexity Analysis

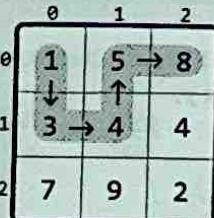
Time complexity: The time complexity of `bipartite_graph_validation` is $O(n + e)$ where n denotes the number of nodes and e denotes the number of edges. This is because we explore all nodes in the graph and traverse across e edges during DFS.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the recursive call stack, which can grow as large as n . In addition, the `colors` array also contributes $O(n)$ space.

Longest Increasing Path

Find the longest strictly increasing path in a matrix of positive integers. A path is a sequence of cells where each one is 4-directionally adjacent (up, down, left, or right) to the previous one.

Example:

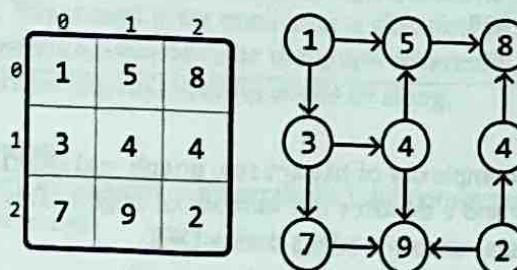


Output: 5

Intuition

In this problem, we're tasked with finding the longest strictly increasing path. Note, "strictly" means the path cannot include any two cells of equal value.

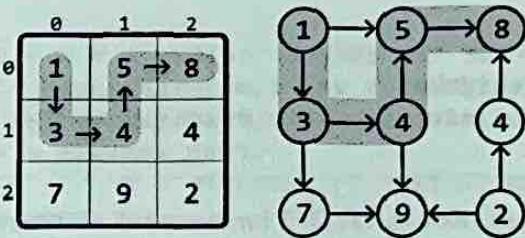
From any cell, we can move to a neighboring cell if that neighbor has a larger value than the current cell. We can map this relationship between cells as nodes in a graph, with edges representing moves from one cell to a higher-value neighboring cell:



Upon closer inspection, we notice this is a **directed acyclic graph (DAG)**. Let's break down what this means:

1. The graph is directed because we can go from a smaller cell to a larger one, but not the other way around.
2. The graph is acyclic because it's not possible to return to a smaller, previous value in the path because the path only extends to larger cells.

This highlights that finding the longest increasing path in a matrix is effectively the same as **finding the longest path in a DAG**.



With this in mind, let's figure out how to traverse the matrix.

Traversing the matrix

One strategy is to find the length of the longest path that starts at each cell and return the maximum of these lengths. To do this, we need a way to explore all paths which extend from each cell, and return the longest one. Many traversal algorithms allow us to do this. In this explanation, we use DFS since it has a slightly simpler implementation for finding the longest path.

Let's see how this works over an example. Start by performing DFS at the first cell, (0, 0):

dfs(0, 0):

	0	1	2
0	1	5	8
1	3	4	4
2	7	9	2

To determine the longest path starting from this cell, we need to explore its higher-value neighboring cells. By making a DFS call to each of these larger neighbors, we can find the lengths of the paths that start from them:

dfs(0, 1)

	0	1	2
0	1 → 5	8	
1	3	4	4
2	7	9	2

The largest path starting at cell (0, 0) will be equal to whichever of these DFS calls returns a larger path, plus 1 to include cell (0, 0) itself.

Note that since this matrix resembles a DAG, we don't need to mark cells as visited, since it's not possible to cycle back to previously visited cells.

The pseudocode for finding the length of the longest path starting at a given cell is provided below:

```
def dfs(cell):
```

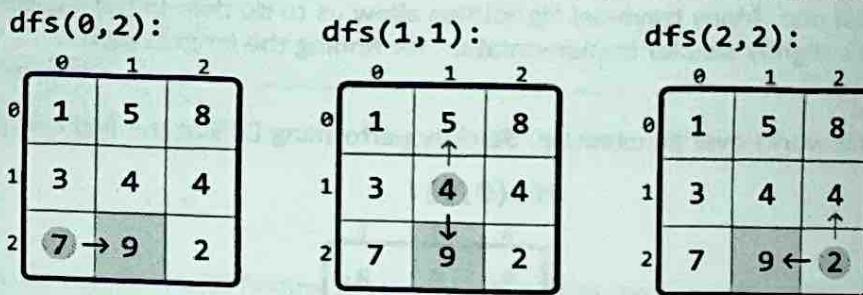
```

max_path = 1
for each neighbor of the current cell:
    if value of neighbor > value of cell:
        max_path = max(max_path, dfs(neighbor) + 1)
return max_path

```

To get the final result, we just need to call DFS for every cell in the matrix, ensuring we find the lengths of the longest paths starting from each cell. The maximum of these lengths is equal to the length of the longest increasing path.

An important thing to notice is that it's possible for DFS to be called on the same cell multiple times. We can see this below, where we call DFS on cell (2, 1) on three different occasions:



After we first calculate the longest path starting at a certain cell, we don't need to calculate it again for that cell. This suggests we should use **memoization**: by storing the DFS result of each cell, we can just return the saved result whenever a DFS call is made to that cell again.

Implementation

```

def longest_increasing_path(matrix: List[List[int]]) -> int:
    if not matrix:
        return 0
    res = 0
    m, n = len(matrix), len(matrix[0])
    memo = [[0] * n for _ in range(m)]
    # Find the longest increasing path starting at each cell. The
    # maximum of these is equal to the overall longest increasing
    # path.
    for r in range(m):
        for c in range(n):
            res = max(res, dfs(r, c, matrix, memo))
    return res

def dfs(r: int, c: int, matrix: List[List[int]],
        memo: List[List[int]]) -> int:
    if memo[r][c] != 0:
        return memo[r][c]
    max_path = 1
    dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    # The longest path starting at the current cell is equal to the
    # longest path of its larger neighboring cells, plus 1.

```

```

for d in dirs:
    next_r, next_c = r + d[0], c + d[1]
    if (is_within_bounds(next_r, next_c, matrix)
        and matrix[next_r][next_c] > matrix[r][c]):
        max_path = max(max_path,
                        1 + dfs(next_r, next_c, matrix, memo))
memo[r][c] = max_path
return max_path

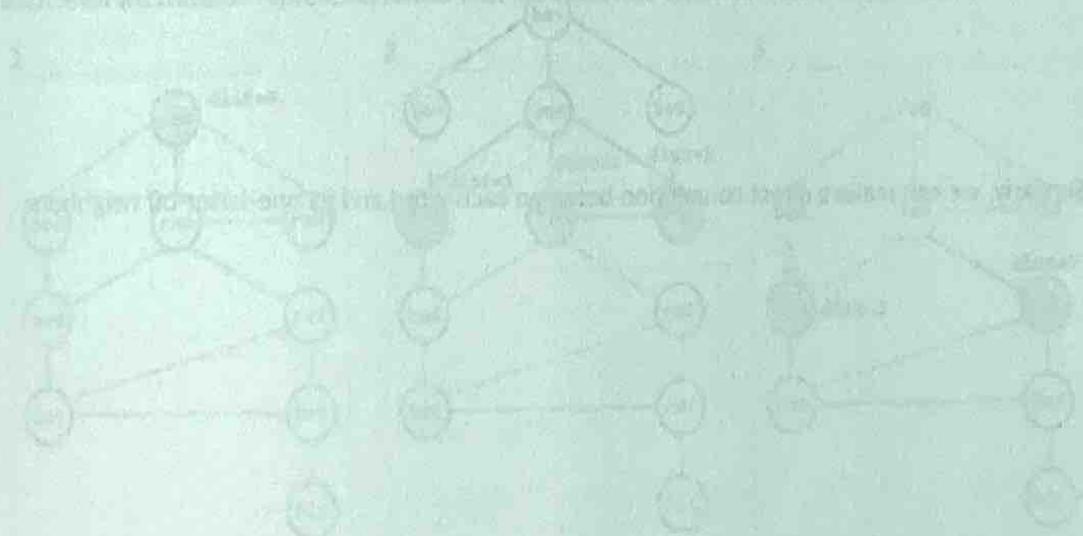
def is_within_bounds(r: int, c: int, matrix: List[List[int]]) -> bool:
    return 0 <= r < len(matrix) and 0 <= c < len(matrix[0])

```

Complexity Analysis

Time complexity: The time complexity of `longest_increasing_path` is $O(m \cdot n)$ where m denotes the number of rows, and n denotes the number of columns. This is because each cell of the matrix is visited at most twice: once in the `longest_increasing_path` function, and during DFS where each cell is visited at most once due to memoization.

Space complexity: The space complexity is $O(m \cdot n)$ primarily due to the recursive call stack during DFS, and the memoization table, both of which can grow to $m \cdot n$ in size.



Shortest Transformation Sequence

Given two words, `start` and `end`, and a dictionary containing an array of words, return the length of the shortest transformation sequence to transform `start` to `end`. A transformation sequence is a series of words in which:

- Each word differs from the preceding word by exactly one letter.
- Each word in the sequence exists in the dictionary.

If no such transformation sequence exists, return 0.

Example:

`r e d` → `r a d` → `r a t` → `h a t` → `h i t`

Input: `start = "red"`, `end = "hit"`, `dictionary = ["red", "bed", "hat", "rod", "rad", "rat", "hit", "bad", "bat"]`

Output: 5

Constraints:

- All words are the same length.
- All words contain only lowercase English letters.
- The dictionary contains no duplicate words.

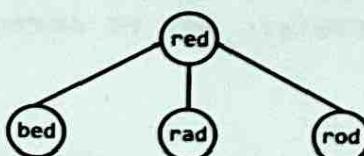
Intuition

In a transformation sequence, each transformation involves changing one letter of a word to get another word. Consider the following example:

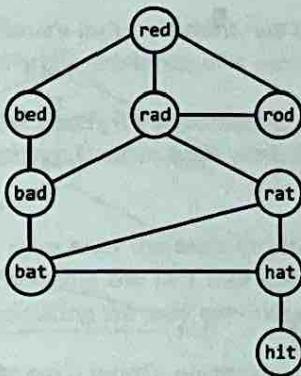
`start = "red", end = "hit"`

`dictionary = ["red" "bed" "hat" "rod" "rad" "rat" "hit" "bad" "bat"]`

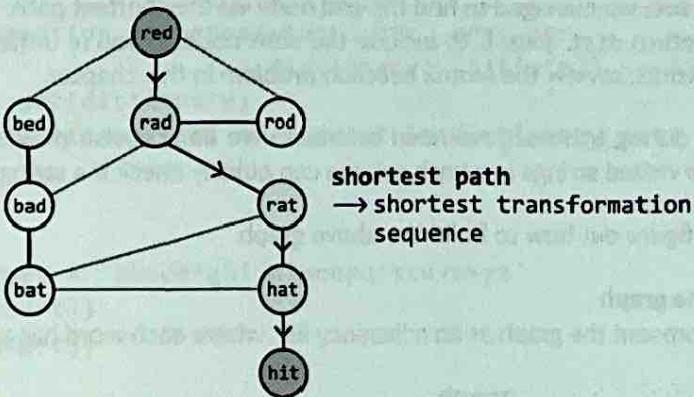
We know our transformation sequence should start with the word "red", but how can we identify subsequent words in the sequence? The words that could immediately follow "red" are "bed", "rad", or "rod", since each differs from "red" by one letter:



Similarly, we can make a direct connection between each word and its one-letter-off neighbors:



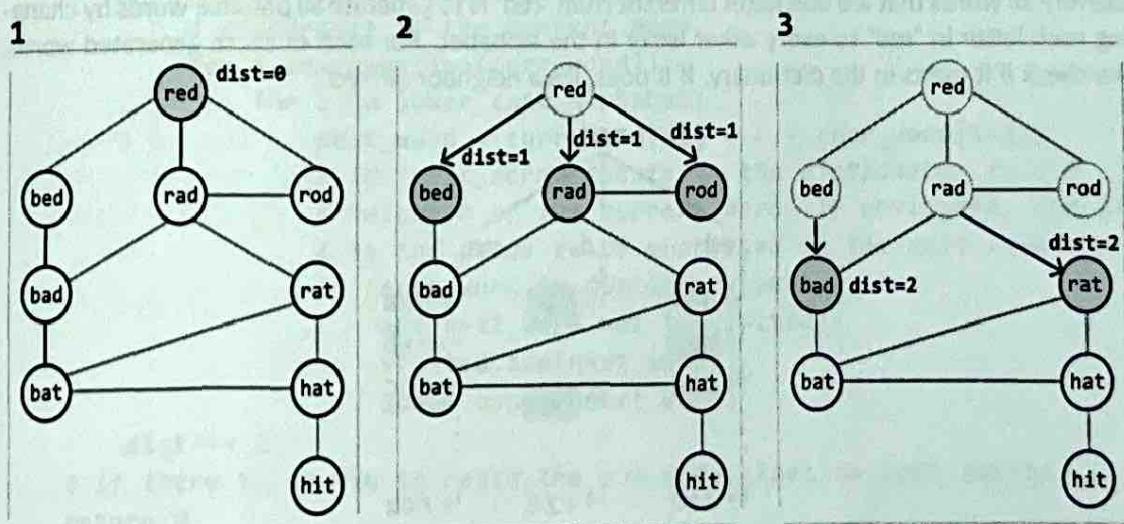
This structure resembles a graph, indicating that finding the shortest transformation sequence involves finding the shortest path in this graph from the start node ("red") to the end node ("hit"):



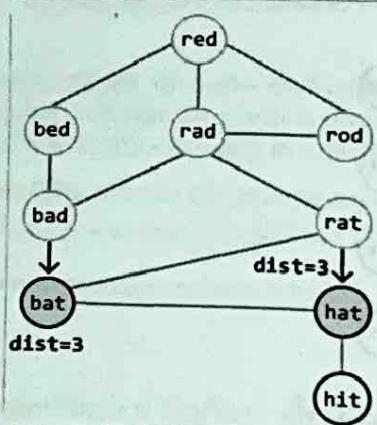
Finding the shortest path

When tasked with a graph problem that involves finding the shortest path between two nodes, BFS should come to mind. From the start node, BFS works by exploring all neighbors at a distance of 1 from the start node, followed by exploring nodes at a distance of 2, and so on. This means that once we find the end node, we've reached it via the shortest possible distance.

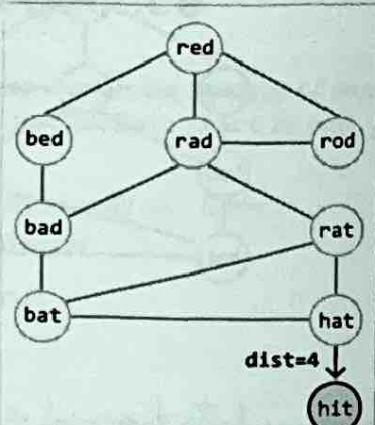
So, let's use **level-order traversal**, a variant of BFS, to find the shortest path in this graph, where each level we traverse represents nodes that are a specific distance from the start node:



4



5



As we can see, we managed to find the end node via the shortest path. To return the length of this path, we return `dist`, plus 1, to include the start node. If you're unfamiliar with how level-order traversal works, review the *Matrix Infection* problem in this chapter.

Note that during traversal, we need to ensure we don't revisit previously traversed strings. By storing the visited strings in a hash set, we can quickly check if a string was already visited.

Now, let's figure out how to build the above graph.

Building the graph

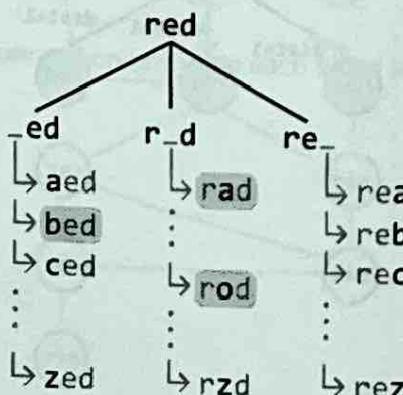
We can represent the graph as an adjacency list, where each word has a list of all of its neighboring words:

`graph:`

```

"red": [ "bed", "rad", "rod" ]
"bed": [ "red", "bad" ]
"hat": [ "rat", "bat", "hit" ]
:
"hit": [ "hat" ]
  
```

The challenge here is finding each word's neighbors. Consider the word "red". One way we can identify all words that are one letter different from "red" is to generate all possible words by changing each letter in "red" to every other letter in the alphabet. For each of these generated words, we check if it exists in the dictionary. If it does, it's a neighbor of "red":



To make checking the dictionary more efficient, we can store the words from the dictionary in a hash set, enabling us to verify the existence of a word in constant time.

Note that before we build the graph, it's important to check if start or end exists in the dictionary. If either is missing, we can return 0, since each word in the sequence needs to be in the dictionary.

Space optimization

The purpose of the above adjacency list is to facilitate graph traversal. However, we can avoid building it altogether by leveraging the fact that each word in the dictionary is only visited once during the BFS traversal, indicating we only ever need access to the neighbors of each word once.

If we had a way to generate each word's neighbors when needed during BFS traversal, we could avoid creating the adjacency list, and save some space that would otherwise be taken up by it.

Implementation

```
def shortest_transformation_sequence(start: str, end: str,
                                      dictionary: List[str]) -> int:
    dictionary_set = set(dictionary)
    if start not in dictionary_set or end not in dictionary_set:
        return 0
    if start == end:
        return 1
    lower_case_alphabet = 'abcdefghijklmnopqrstuvwxyz'
    queue = deque([start])
    visited = set([start])
    dist = 0
    # Use level-order traversal to find the shortest path from the
    # start word to the end word.
    while queue:
        for _ in range(len(queue)):
            curr_word = queue.popleft()
            # If we found the end word, we've reached it via the
            # shortest path.
            if curr_word == end:
                return dist + 1
            # Generate all possible words that have a one-letter
            # difference to the current word.
            for i in range(len(curr_word)):
                for c in lower_case_alphabet:
                    next_word = curr_word[:i] + c + curr_word[i+1:]
                    # If 'next_word' exists in the dictionary, it's a
                    # neighbor of the current word. If unvisited, add it
                    # to the queue to be processed in the next level.
                    if (next_word in dictionary_set
                        and next_word not in visited):
                        visited.add(next_word)
                        queue.append(next_word)
            dist += 1
    # If there is no way to reach the end node, then no path exists.
    return 0
```

Complexity Analysis

Time complexity: The time complexity of `shortest_transformation_sequence` is $O(n \cdot L^2)$, where n denotes the number of words in the dictionary, and L denotes the length of a word. Here's why:

- Creating a hash set containing all the words in the dictionary takes $O(n \cdot L)$ time, because hashing each of the n words takes $O(L)$ time.
- Level-order traversal processes at most n words from the dictionary. At each of these words, we generate up to $26 \cdot L$ transformations, and it takes $O(L)$ time to check if a transformation exists in the `visited` and `dictionary_set` hash sets, and to enqueue it. This means level-order traversal takes approximately $O(n \cdot 26 \cdot L \cdot L) = O(n \cdot L^2)$ time.

Therefore, the overall time complexity is $O(n \cdot L) + O(n \cdot L^2) = O(n \cdot L^2)$.

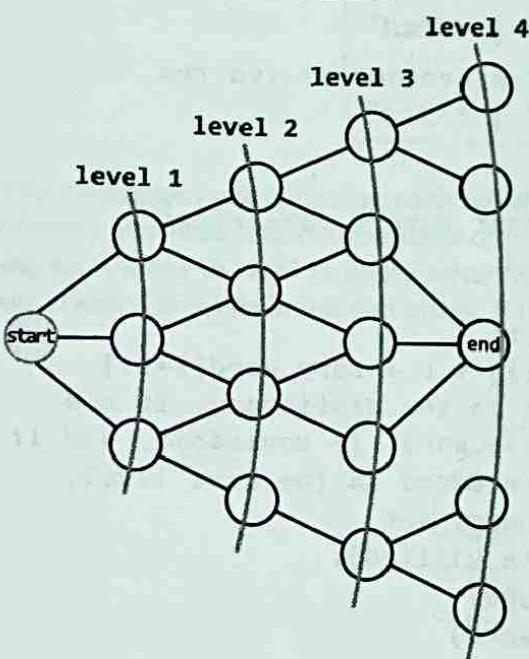
Space complexity: The space complexity is $O(n \cdot L)$, taken up by the `dictionary_set` hash set, the `visited` hash set, and the queue.

Optimization – Bidirectional Traversal

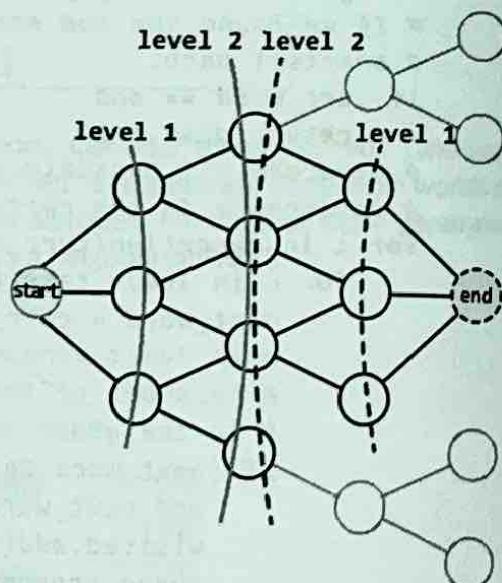
An important observation is that we don't necessarily need to begin level-order traversal at the start word: we can also start a search at the end word. In fact, we can combine these by performing them simultaneously to find the shortest path. This is known as bidirectional BFS, or in this case, bidirectional level-order traversal.

When we perform two searches, one from start and one from end, the idea is that they will meet in the middle if a path between these two words exists. If a path doesn't exist, the searches will never meet, indicating that a transformation sequence does not exist. This optimization allows us to identify the shortest distance more quickly. We can see in the example below that regular level-order traversal ends up traversing through more nodes than in the bidirectional traversal, while also requiring 4 iterations instead of 2.

Level-order traversal:



Bidirectional level-order traversal:



To simulate this process, we alternate between the two level-order traversals and progress through each search one level at a time:

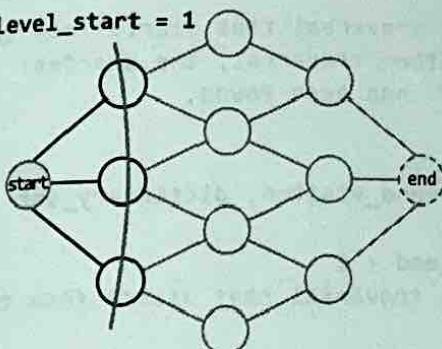
- Start traversal: progress one level in the traversal that started from the start node.
- End traversal: progress one level in the traversal that started from the end node.

We alternate between these two steps until a node visited in one search has already been visited in the other search, indicating that the traversals have met. Here's what this alternating process looks like:

start traversal:

`[level_start += 1]`

`level_start = 1`

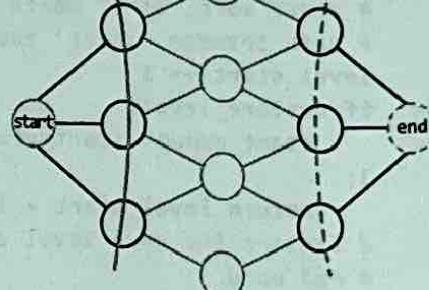


end traversal:

`[level_end += 1]`

`level_start = 1`

`level_end = 1`

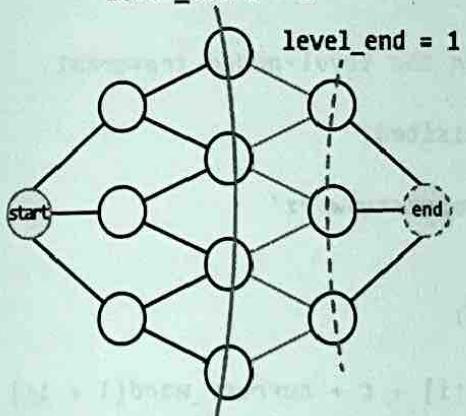


start traversal:

`[level_start += 1]`

`level_start = 2`

`level_end = 1`



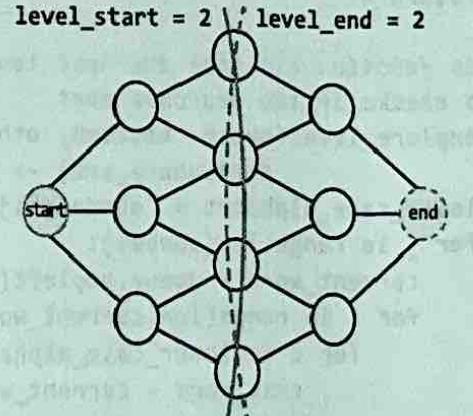
end traversal:

`[level_end += 1]`

`traversals have met`

`→ return level_start + level_end + 1`

`level_start = 2` / `level_end = 2`



To check if a node has already been visited by the other traversal, we can query the visited hash set used by the other traversal. If a word exists in the visited hash set of the other traversal, we know the searches have met.

Implementation – Bidirectional Traversal

```
def shortest_transformation_sequence_optimized(
    start: str, end: str, dictionary: List[str]) -> int:
    dictionary_set = set(dictionary)
    if start not in dictionary_set or end not in dictionary_set:
        return 0
```

```

if start == end:
    return 1
start_queue = deque([start])
start_visited = {start}
end_queue = deque([end])
end_visited = {end}
level_start = level_end = 0
# Perform a level-order traversal from the start word and another
# from the end word.
while start_queue and end_queue:
    # Explore the next level of the traversal that starts from the
    # start word. If it meets the other traversal, the shortest
    # path between 'start' and 'end' has been found.
    level_start += 1
    if explore_level(
        start_queue, start_visited, end_visited, dictionary_set
    ):
        return level_start + level_end + 1
    # Explore the next level of the traversal that starts from the
    # end word.
    level_end += 1
    if explore_level(
        end_queue, end_visited, start_visited, dictionary_set
    ):
        return level_start + level_end + 1
    # If the traversals never met, then no path exists.
return 0

# This function explores the next level in the level-order traversal
# and checks if two searches meet.
def explore_level(queue, visited, other_visited,
                  dictionary_set) -> bool:
    lower_case_alphabet = 'abcdefghijklmnopqrstuvwxyz'
    for _ in range(len(queue)):
        current_word = queue.popleft()
        for i in range(len(current_word)):
            for c in lower_case_alphabet:
                next_word = current_word[:i] + c + current_word[i + 1:]
                # If 'next_word' has been visited during the other
                # traversal, it means both searches have met.
                if next_word in other_visited:
                    return True
                if (next_word in dictionary_set
                    and next_word not in visited):
                    visited.add(next_word)
                    queue.append(next_word)
    # If no word has been visited by the other traversal, the searches
    # have not met yet.
    return False

```

Complexity Analysis

Time complexity: The time complexity of `shortest_transformation_sequence_optimized` is $O(n \cdot L^2)$, since we're performing two level-order traversals. Note that this is more efficient in practice since there are potentially fewer nodes to traverse when using bidirectional traversal.

Space complexity: The space complexity is $O(n \cdot L)$, taken up by the `dictionary_set` hash set, both `visited` hash sets, and both queues.

Merging Communities

There are n people numbered from 0 to $n - 1$, with each person initially belonging to a separate community. When two people from different communities connect, their communities merge into a single community.

Your goal is to write two functions:

- `connect(x: int, y: int) -> None`: Connects person x with person y and merges their communities.
- `get_community_size(x: int) -> int`: Returns the size of the community which person x belongs to.

Example:

`connect(0, 1):`



`connect(1, 2):`



`get_community_size(3):`

1

`get_community_size(0):`

3

`connect(3, 4):`



`get_community_size(4):`

2

Input: $n = 5$, [`connect(0, 1)`, `connect(1, 2)`, `get_community_size(3)`,
`get_community_size(0)`, `connect(3, 4)`, `get_community_size(4)`]
Output: [1, 3, 2]

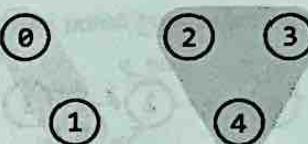
Intuition

In this problem, we start with n individuals, each in their own separate community. As we connect pairs of individuals, their respective communities merge. The challenge is to efficiently manage these connections and quickly determine the size of the community for any given individual.

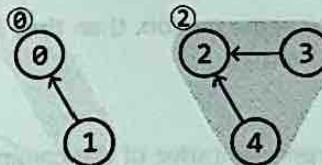
This is where the **Union-Find** data structure, also known as the Disjoint Set Union (DSU) data structure, comes in. Union-Find consists of two operations:

- **Union**: takes two elements from different sets and makes them part of the same set.
- **Find**: determines what set an element belongs to.

In the context of this problem, the union operation can be used to merge the communities of two people, while the find operation can be used to determine which community a person belongs to. To make this work, we need a way to represent each community. For example, let's say there are 5 people, with persons 0 and 1 in one community, and persons 2, 3, and 4 in another:



One way to distinguish between them is to designate a representative for each community. Let's assign person 0 as the representative of the left community, and person 2 as the representative of the right community. We can represent these communities as a graph, where each community is a connected component, and each person in a community points to their representative:

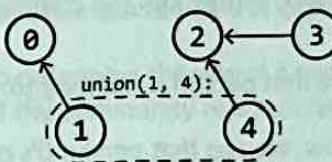


This can be reflected using a parent array, where `parent[i]` stores the parent of person `i`. This parent array is discussed later.

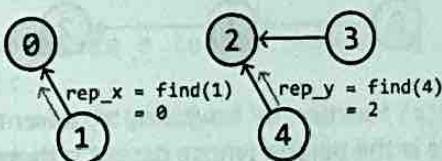
Now that we have a way to represent communities, let's discuss how the `union` and `find` functions would work in more detail.

Union

Consider the previous example of 5 people. Let's say we want to connect persons 1 and 4 (i.e., `union(1, 4)`):

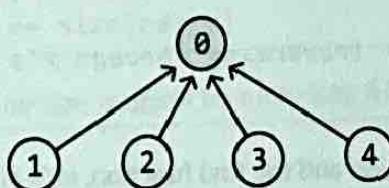


We first need to identify their representatives so we know which communities these two people belong to. We can use the `find` function for this, which will be discussed later:



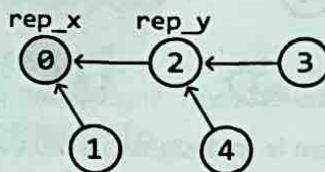
Before merging these communities, we need to pick one of the two current representatives as the representative of the merged community. For now, let's just pick person 0.

From here, one strategy is to connect everyone in the second community directly to person 0:



This would require individually setting the parent of all these people to person 0, which can be quite expensive if the community has many people. A more efficient strategy is to just connect the representative of this community, person 2, to person 0.

In code, this is done by setting the parent of `rep_y` to be `rep_x` (`parent[rep_y] = rep_x`):

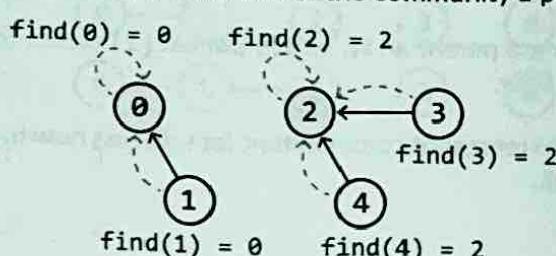


This way, everyone originally in the second community is indirectly connected to person 0, making person 0 the representative of all nodes.

Note that if `rep_x` and `rep_y` are the same person, then they belong to the same community, in which case we don't need to merge anything.

Find

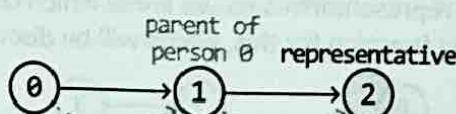
The `find` function should return the representative of the community a person is in:



At this point, it's important to note the difference between a parent and a representative:

- A representative is the person who represents the community. This is effectively the "root" node of the community.
- A parent of a person is the node that person is pointing to.

For example, in the community below, we see that person 0's parent is person 1, whereas person 0's representative is person 2:



We can implement the `find(x)` function by traversing `x`'s parent chain until we reach the representative. The representative is the person whose parent is themselves, so we can identify them by checking if the current person is their own parent. The code snippet for this is provided below.

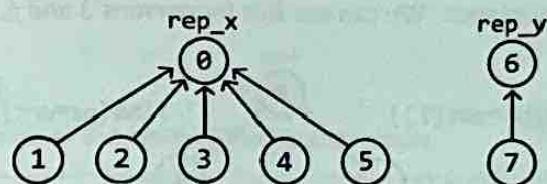
```
def find(x: int) -> int:  
    # If x is equal to its parent, we found the representative.  
    if x == parent[x]:  
        return x  
    # Otherwise, continue traversing through x's parent chain.  
    return find(parent[x])
```

Now we've discussed both the union and the `find` function, let's discuss what optimizations can be made to them.

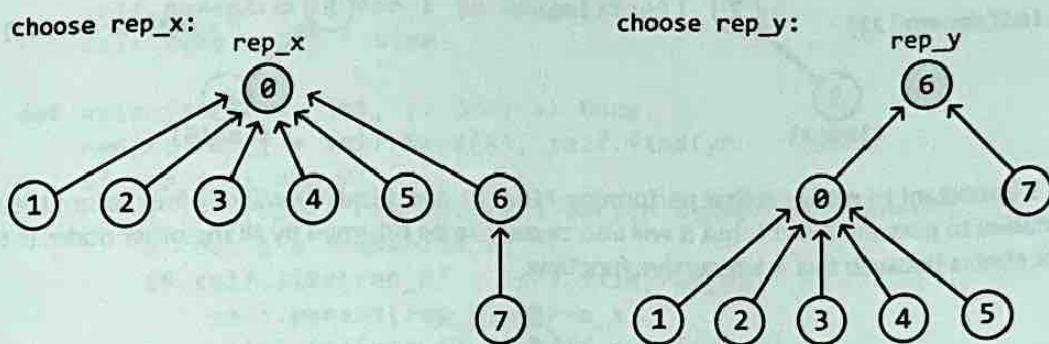
Union by size

Recall that before merging two communities, we need to pick someone to represent the merged

community. Our choice is between `rep_x` and `rep_y`. Is one a better choice than the other? Consider the following two communities:



Let's see what the difference is when we pick `rep_x` to be the representative versus picking `rep_y`:



As we can see, when we choose a representative from one of two communities, the people in the other community have a longer distance to their new representative, with the distance increased by 1. Therefore, it's better to pick the representative from the larger community, so that only the people from the smaller community are impacted by this adjusted distance.

This optimization requires a way to determine the size of a community. We can do this using a size array, where `size[i]` is the size of the community represented by person `i`. The code for this can be seen below:

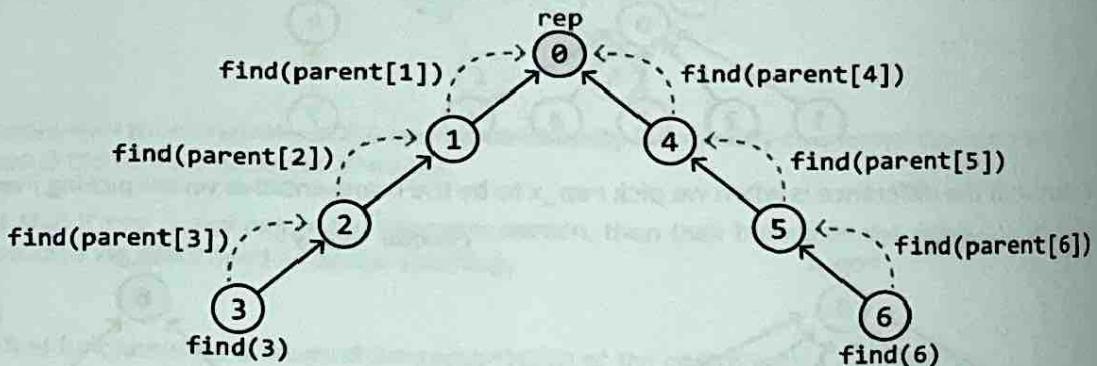
```
def union(x: int, y: int) -> None:
    rep_x, rep_y = find(x), find(y)
    if rep_x != rep_y:
        # If 'rep_x' represents a larger community, connect 'rep_y's
        # community to it.
        if size[rep_x] > size[rep_y]:
            parent[rep_y] = rep_x
            size[rep_x] += size[rep_y]
        # If 'rep_y' represents a larger community, or both communities
        # are of the same size, connect 'rep_x's community to it.
        else:
            parent[rep_x] = rep_y
            size[rep_y] += size[rep_x]
```

A similar optimization to union by size is union by rank, which optimizes the union function in a very similar way [1].

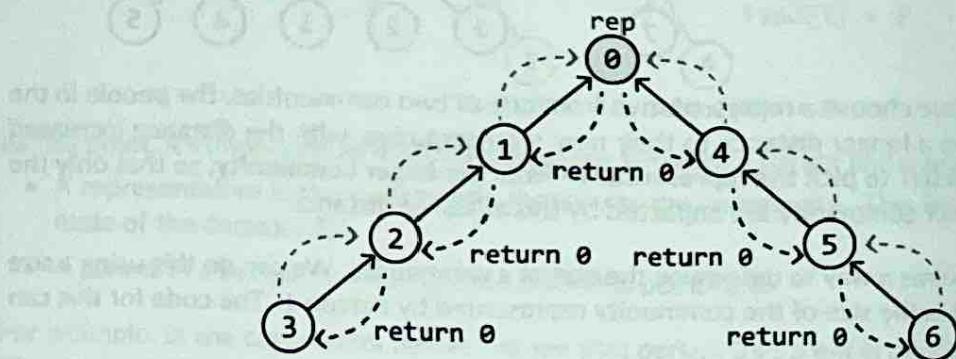
Note that the `size` array makes the implementation of `get_community_size(x)` quite straightforward: we just return the size of `x`'s representative (i.e., `return size[find(x)]`).

Path compression

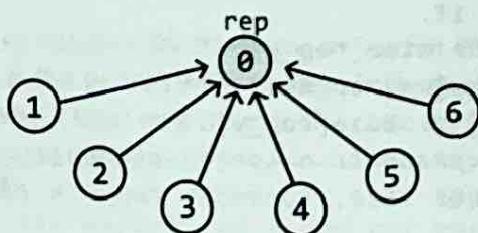
There's another major optimization that can be made. Consider the following community, with person 0 as the representative. The further down we go in this diagram, the further away the people are from the representative. We can see this for persons 3 and 6, for example:



What's important to realize is that performing `find(3)` and `find(6)` will not only return the representative to persons 3 and 6, but it will also cause it to be returned by all the other nodes in their parent chains because this is a recursive function:



This means every person in their parent chains also knows their representative is person 0. So, if we update the parent of each of these people to person 0 in the `find` function, all the nodes in this chain will connect to the representative. This technique is known as path compression, and it allows us to flatten the graph:



The code for this optimization is provided below:

```
def find(x: int) -> int:
    if x == parent[x]:
        return x
    # Path compression: updates the representative of x, flattening
    # the structure of the community as we go.
    parent[x] = find(parent[x])
    return parent[x]
```

This path compression optimization allows us to restructure the graph at every `find` call, effectively reducing the distance between people and their representative, and making future `find` calls more efficient.

Implementation

Below is the implementation of the Union-Find data structure.

```
class UnionFind:
    def __init__(self, size: int):
        self.parent = [i for i in range(size)]
        self.size = [1] * size

    def union(self, x: int, y: int) -> None:
        rep_x, rep_y = self.find(x), self.find(y)
        if rep_x != rep_y:
            # If 'rep_x' represents a larger community, connect
            # 'rep_y's community to it.
            if self.size[rep_x] > self.size[rep_y]:
                self.parent[rep_y] = rep_x
                self.size[rep_x] += self.size[rep_y]
            # Otherwise, connect 'rep_x's community to 'rep_y'.
            else:
                self.parent[rep_x] = rep_y
                self.size[rep_y] += self.size[rep_x]

    def find(self, x: int) -> int:
        if x == self.parent[x]:
            return x
        # Path compression.
        self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def get_size(self, x: int) -> int:
        return self.size[self.find(x)]
```

Below is the implementation of the main problem using the above Union-Find data structure:

```
class MergingCommunities:
    def __init__(self, n: int):
        self.uf = UnionFind(n)

    def connect(self, x: int, y: int) -> None:
        self.uf.union(x, y)

    def get_community_size(self, x: int) -> int:
        return self.uf.get_size(x)
```

Complexity Analysis

Time complexity: Let's break down the time complexity of the Union-Find functions:

- With the path compression and union by size optimizations, `find` has a time complexity of amortized $O(1)$ ¹ because the branches of the graph become very short over time, making the function effectively constant-time in most cases.
- Since `union` just uses the `find` function twice, it also has a time complexity of amortized $O(1)$.
- Since `get_size` just uses the `find` function once, it also has a time complexity of amortized $O(1)$.

Therefore, the time complexities of `connect` and `get_community_size` are both amortized $O(1)$. The time complexity of the constructor is $O(n)$ because we initialize two arrays of size n when creating the `UnionFind` object.

Space complexity: The space complexity is $O(n)$ because the Union-Find data structure has two arrays of size n : `parent` and `size`. The space taken up by the recursive call stack is amortized $O(1)$ since the branches of the graph become very short over time, resulting in fewer recursive calls made to the `find` function.

References

- [1] Disjoint-set data structure: https://en.wikipedia.org/wiki/Disjoint-set_data_structure

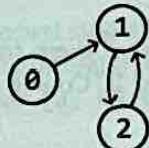
¹ We can also write the time complexity here as $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, which grows extremely slowly but is not perfectly constant.

Prerequisites

Given an integer n representing the number of courses labeled from 0 to $n - 1$, and an array of prerequisite pairs, determine if it's possible to enroll in all courses.

Each prerequisite is represented as a pair $[a, b]$, indicating that course a must be taken before course b .

Example:



Input: $n = 3$, prerequisites = $[[0, 1], [1, 2], [2, 1]]$

Output: False

Explanation: Course 1 cannot be taken without first completing course 2, and vice versa.

Constraints:

- For any prerequisite $[a, b]$, a will not equal b .

Intuition

Our goal is to check if enrollment into all courses is possible. Let's start by identifying which situations make enrollment impossible.

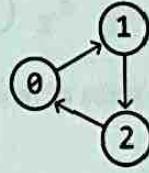
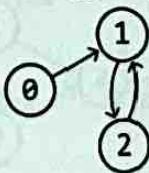
Consider a simple case where there are just two courses. A scenario where enrollment into all courses is impossible occurs when each course is a prerequisite to the other, as graphically represented below:

prerequisites = $[[0,1], [1,0]]$



Here are a couple of impossible enrollment scenarios that could occur with prerequisites for three courses:

prerequisites = $[[0,1], [1,2], [2,1]]$ prerequisites = $[[2,0], [0,1], [1,2]]$



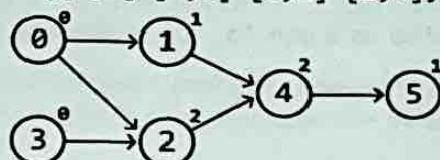
What do we notice about these cases and their graphical representations? They both have a circular relationship: a cycle. This highlights that it's impossible to enroll in all courses if there exists a circular dependency between courses. In other words, there must not be a cycle in the graphical representation of the courses for complete enrollment to be possible.

Another thing to note is that the first courses which can be completed are those without prerequisites. In the graphical representation, such a course would have no directed arrows pointing at

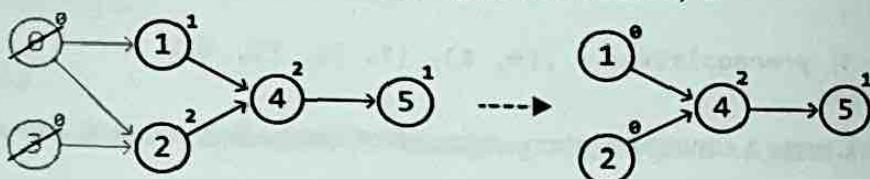
them. Let's refer to the number of directed edges incoming to a node as the **in-degree** of that node.

In the example below, each node's in-degree is displayed at the top right:

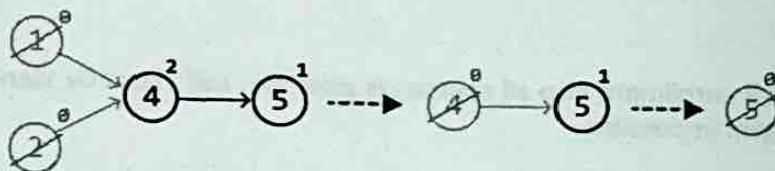
```
prerequisites = [[0,1] [0,2] [3,2] [1,4], [2,4], [4,5]]
```



Courses 0 and 3 have an in-degree of 0. So, let's complete them first and remove them from the graph. By doing this, we reduce the number of prerequisites for courses 1 and 2. Course 1's indegree then decreases by 1, and course 2's indegree decreases by 2:



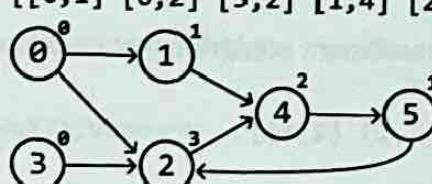
Now, courses 1 and 2 have an in-degree of 0, which means we can enroll in them and remove them from the graph. Observe what happens when we continue the process of removing courses with an in-degree of 0 from the graph:



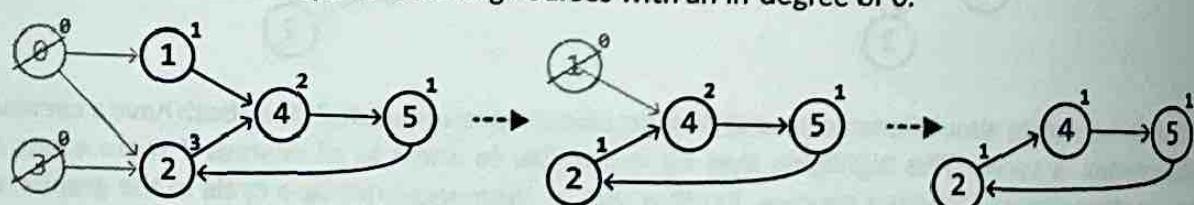
By the end of this process, no courses remain, indicating it's possible to enroll in all courses.

Now, consider an example with a cyclic dependency:

```
prerequisites = [[0,1] [0,2] [3,2] [1,4] [2,4] [4,5] [5,2]]
```



Let's follow the same steps of removing courses with an in-degree of 0:



Here, there is no way to progress because there aren't any courses with an in-degree of 0. When this happens, and there are still unvisited courses, a cyclic dependency exists, meaning enrolment is impossible.

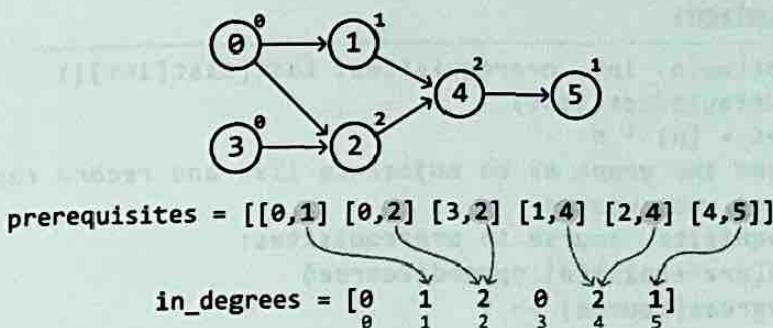
Now, let's identify a way to simulate the above process algorithmically.

Topological sort

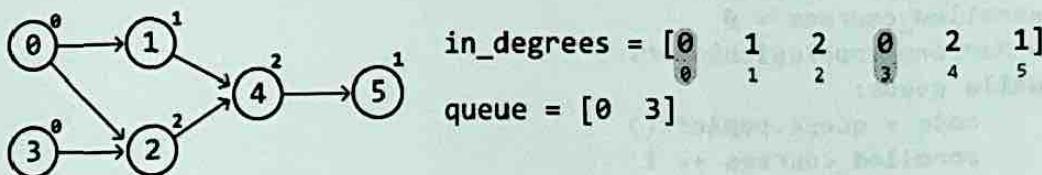
The process we described above is essentially topological sorting, where vertices of a graph are sorted in such a way that for every directed edge $u \rightarrow v$, node u comes before node v in the ordering of the topological sort.

An algorithm designed to perform topological sort is Kahn's algorithm. Let's see how we can use it to solve this problem.

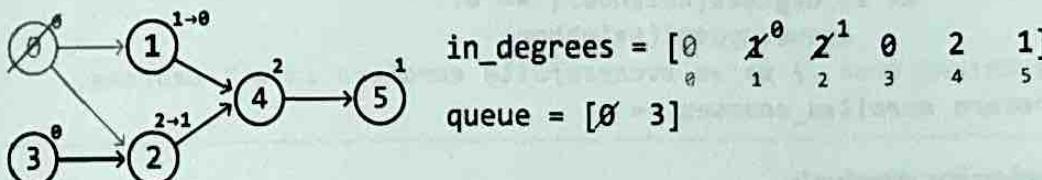
The first step of Kahn's algorithm is to determine the in-degree of each course. This can be achieved by counting the number of times each course appears as a dependent in the prerequisite pairs: for a pair $[a, b]$, course b depends on course a , so course b 's in-degree is incremented by one:



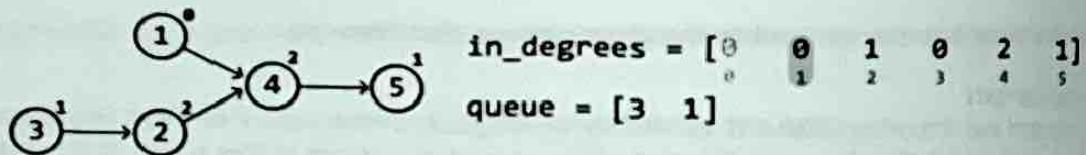
Now, we want to process all the courses with an in-degree of 0 first. We can add these courses to a queue to be processed:



To begin processing, pop the first course from the queue: course 0. Then, for each course that has course 0 as a prerequisite (i.e., courses pointed to by course 0), reduce their in-degree by one:



If any of the courses have an in-degree of 0 after being decremented, add them to the queue. Course 1 now has an in-degree of 0, so we add it to the queue:



We can continue the above process until the queue is empty, indicating that there aren't any more courses with an in-degree of 0.

- If we've processed all n courses, enrollment to all courses is possible.
- If we couldn't process all n courses, a cycle was found, indicating enrollment to all courses is impossible.

Implementation

```
def prerequisites(n: int, prerequisites: List[List[int]]) -> bool:
    graph = defaultdict(list)
    in_degrees = [0] * n
    # Represent the graph as an adjacency list and record the in-
    # degree of each course.
    for prerequisite, course in prerequisites:
        graph[prerequisite].append(course)
        in_degrees[course] += 1
    queue = deque()
    # Add all courses with an in-degree of 0 to the queue.
    for i in range(n):
        if in_degrees[i] == 0:
            queue.append(i)
    enrolled_courses = 0
    # Perform topological sort.
    while queue:
        node = queue.popleft()
        enrolled_courses += 1
        for neighbor in graph[node]:
            in_degrees[neighbor] -= 1
            # If the in-degree of a neighboring course becomes 0, add
            # it to the queue.
            if in_degrees[neighbor] == 0:
                queue.append(neighbor)
    # Return true if we've successfully enrolled in all courses.
    return enrolled_courses == n
```

Complexity Analysis

Time complexity: The time complexity of prerequisites is $O(n + e)$, where e denotes the number of edges derived from the prerequisites array. Here's why:

- Creating the adjacency list and recording the in-degrees takes $O(e)$ time because we iterate through each prerequisite once.
- Adding all courses with in-degree 0 to the queue takes $O(n)$ time because we check the in-degree of each course once.

- Performing Kahn's algorithm takes $O(n + e)$ time because each course and prerequisite is processed at most once during the traversal.

Space complexity: The space complexity is $O(n + e)$, since the adjacency list takes up $O(n + e)$ space, while the `in_degrees` array and queue each take up $O(n)$ space.

Backtracking

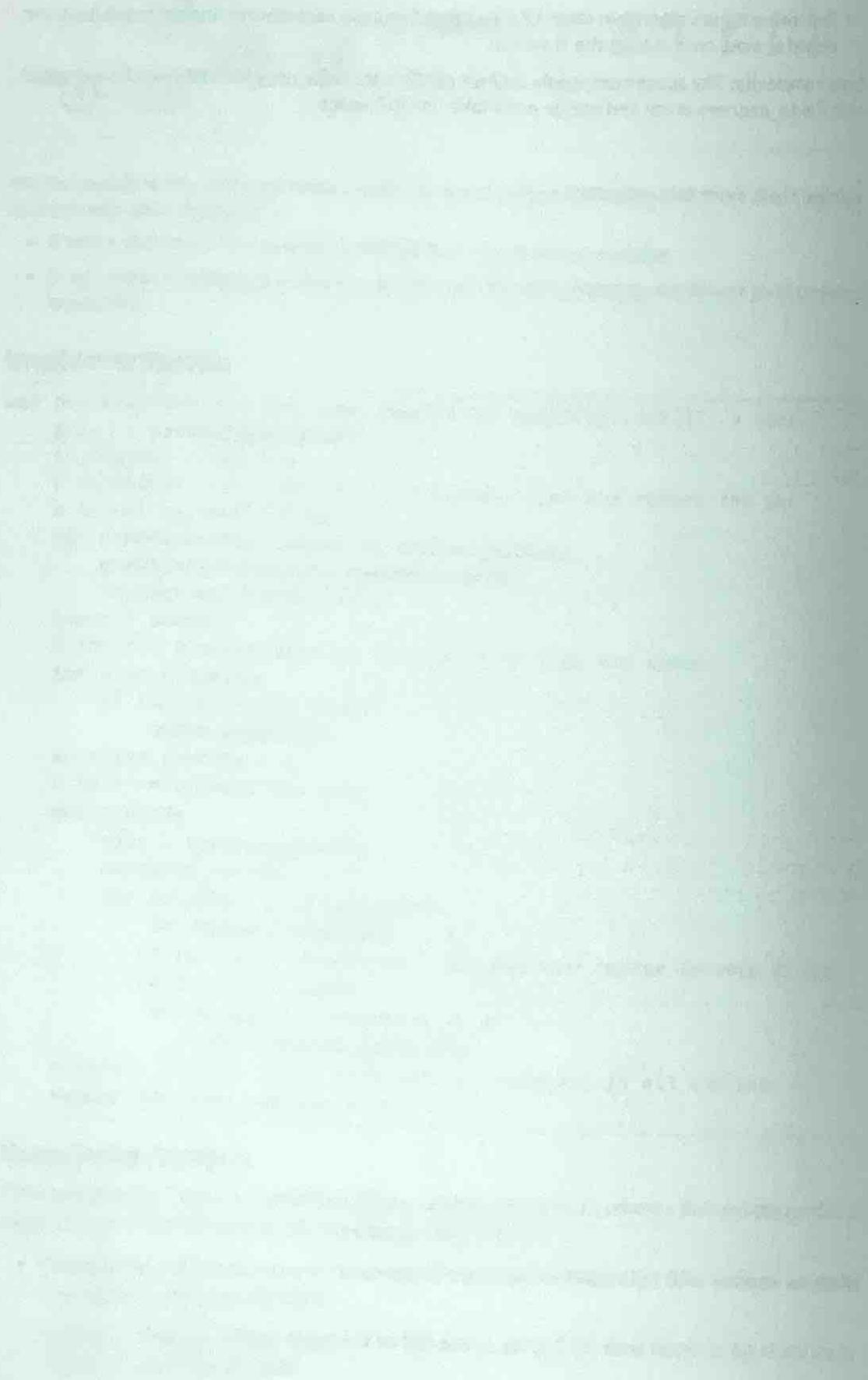
Introduction to Backtracking

Backtracking is a general algorithmic technique used for solving problems recursively by exploring all possible solutions in a search tree. If some solutions do not lead to the goal, they are abandoned, or pruned, from the search tree.



Backtracking is often used for solving combinatorial problems. By backtracking, you explore every node of the search tree, trying every possible solution until you find the one that matches the problem statement. Many famous problems can be solved using backtracking, such as N-Queens, Eight Queen's Problem, etc.

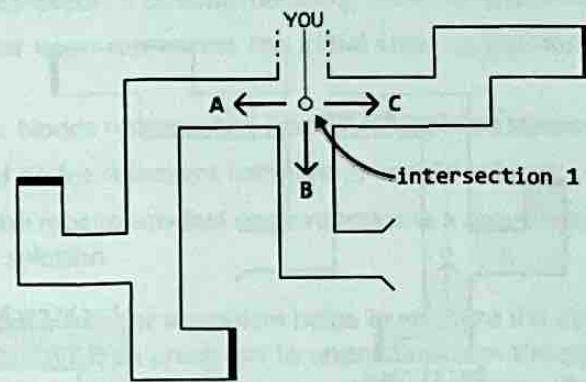




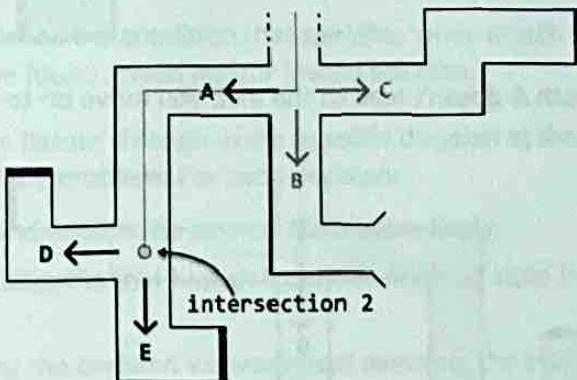
Backtracking

Introduction to Backtracking

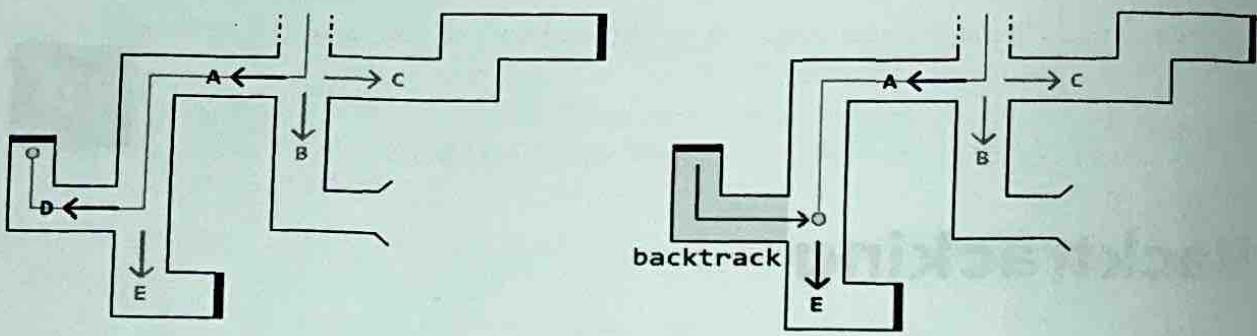
Imagine you're stuck at an intersection point in a maze, and you know one of the three routes ahead leads to the exit:



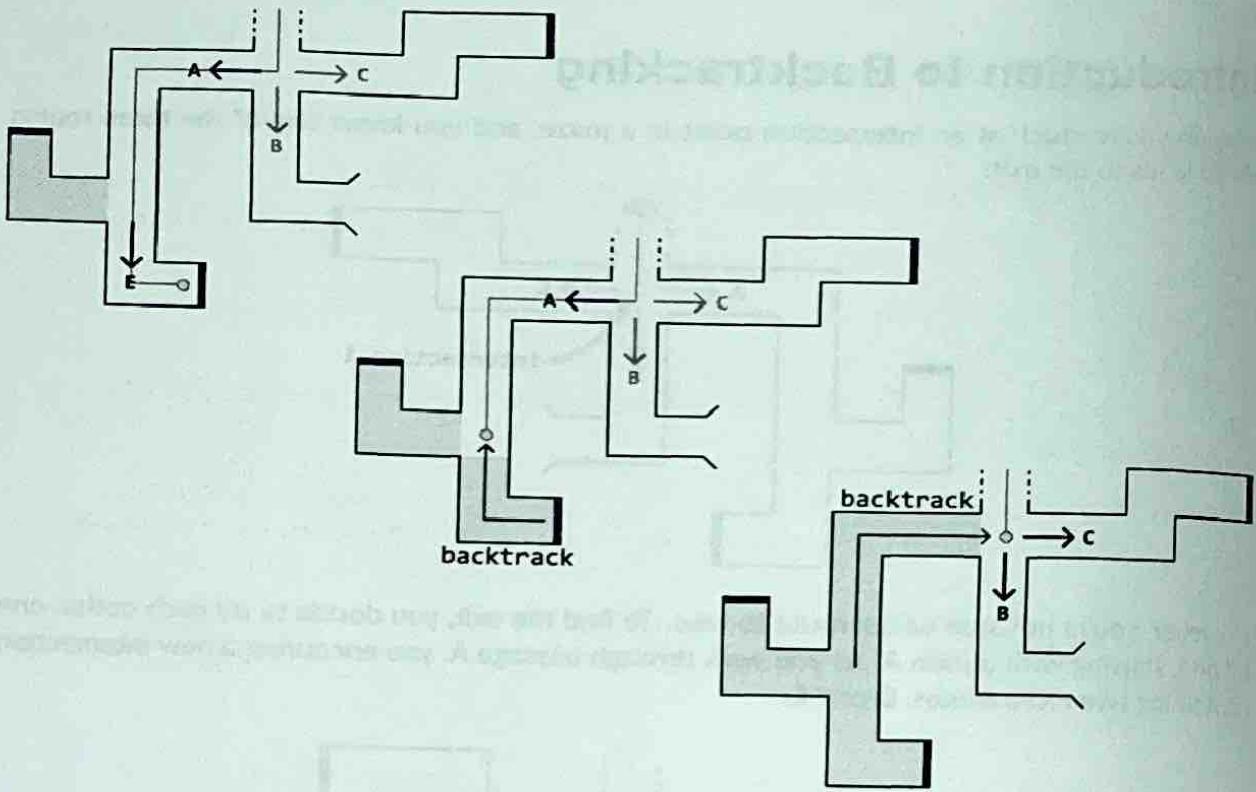
However, you're not sure which route to take. To find the exit, you decide to try each option one by one, starting with option A. As you walk through passage A, you encounter a new intersection containing two more routes, D and E:



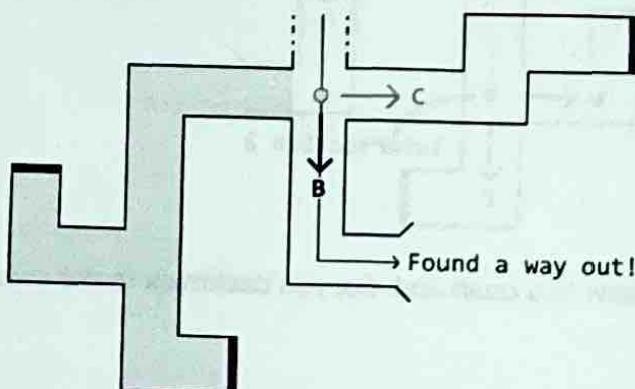
You try option D, but it leads to a dead end. So, you backtrack to the second intersection point:



Next, you try option E, but it also leads to a dead end, so you backtrack to the second junction point. After concluding that neither path D or E works, you backtrack again to the first intersection point:



Having determined that path A doesn't lead to the exit, you move on to path B and discover that it leads to the exit:

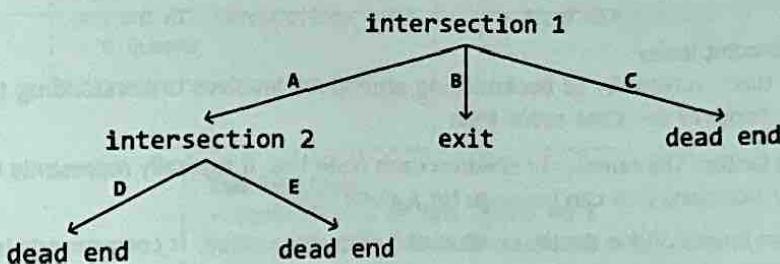


This brute force process of testing all possible paths and backtracking upon failure is called 'backtracking.'

State space tree

In backtracking, the state space tree, also known as the decision tree, is a conceptual tree constructed by considering every possible decision that can be made at each point in a process.

For example, here's how we would represent the state space tree for the maze scenario:



Here's a simplified explanation of a state space tree:

- **Edges:** Each edge represents a possible decision, move, or action.
- **Root node:** The root node represents the initial state or position before any decisions are made.
- **Intermediate nodes:** Nodes representing partially completed states or intermediate positions.
- **Leaf nodes:** The leaf nodes represent complete or invalid solutions.
- **Path:** A path from the root to any leaf node represents a sequence of decisions that lead to a complete or invalid solution.

Drawing out the state space tree for a problem helps to visualize the entire solution space, and all possible decisions. In addition, it's a great way to understand how the algorithm works. By figuring out how to traverse this tree, we essentially create the backtracking algorithm.

Backtracking algorithm

Traversing the state space tree is typically done using recursive DFS. Let's discuss how it's implemented at a high level.

Termination condition: Define the condition that specifies when a path should end. This condition should define when we've found a valid and/or invalid solution.

Iterate through decisions: Iterate through every possible decision at the current node, which contains the current state of the problem. For each decision:

1. Make that decision and update the current state accordingly.
2. Recursively explore all paths that branch from this updated state by calling the DFS function on this state.
3. Backtrack by undoing the decision we made and reverting the state.

Below is a crude template for backtracking:

```
def dfs(state):
    # Termination condition.
    if meets_termination_condition(state):
```

```
process_solution(state)
return
# Explore each possible decision that can be made at the current
# state.
for decision in possible_decisions(state):
    make_decision(state, decision)
    dfs(state)
    undo_decision(state, decision) # Backtrack.
```

Analyzing time complexity

Analyzing the time complexity of backtracking algorithms involves understanding the branching factor and the depth of the state space tree:

- **Branching factor:** The number of children each node has. It typically represents the maximum number of decisions that can be made for a given state.
- **Depth:** The length of the deepest path in the state space tree. It corresponds to the number of decisions or steps required to reach a complete solution.

The time complexity is often estimated as $O(b^d)$, where b denotes the branching factor and d denotes the depth. This is because in the worst case, every node at each level of the tree needs to be explored during a typical backtracking algorithm.

When to use backtracking

Backtracking is useful when we need to explore all possible solutions to a problem. For example, if we need to find all possible ways to arrange items, or generate all possible subsets, permutations, or combinations, backtracking can help to identify every possible solution.

Real-world Example

AI algorithms for games: backtracking is used in AI algorithms for games like chess and Go to explore possible moves and strategies. The programs examine each potential move, simulate the game's progression, and evaluate the outcome. If a move leads to an unfavorable position, the program will backtrack to the previous move and try alternative options, systematically exploring the game tree until it finds the optimal strategy.

Chapter Outline

Backtracking

- Permutations**
 - Find All Permutations
 - N Queens

- Subsets**
 - Find All Subsets

Combinations

- Combinations of Sum (BONUS PDF)
- Phone Keypad Combination (BONUS PDF)

Find All Permutations

Return all possible permutations of a given array of unique integers. They can be returned in any order.

Example:

Input: `nums = [4, 5, 6]`

Output: `[[4, 5, 6], [4, 6, 5], [5, 4, 6], [5, 6, 4], [6, 4, 5], [6, 5, 4]]`

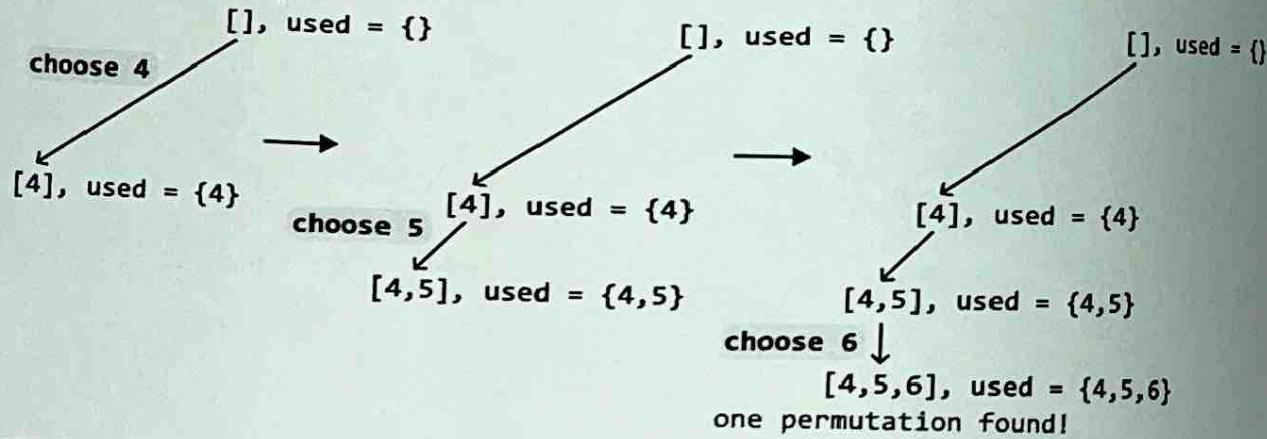
Intuition

Our task in this problem is quite straightforward: find all permutations of a given array. The key word here is "all". To achieve this, we need an algorithm that generates each possible permutation one at a time. The technique that naturally fits this requirement is **backtracking**. As with any backtracking solution, it's useful to first visualize the state space tree.

State space tree

Let's figure out how to build just one permutation. Consider the array `[4, 5, 6]`. We can start by picking one number from this array for the first position of this permutation. For the second position, let's pick a different number. We can keep adding numbers like this until all the numbers from the array are used. To avoid reusing numbers, let's also keep track of the used numbers using a hash set.

`nums = [4 5 6]:`

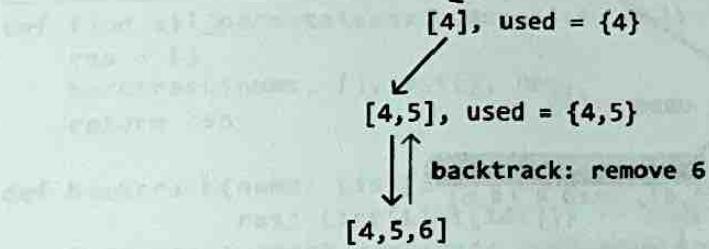


Now that we've found one permutation, let's *backtrack* to find others. Start by removing the most recently added number, 6, bringing us back to `[4, 5]`:

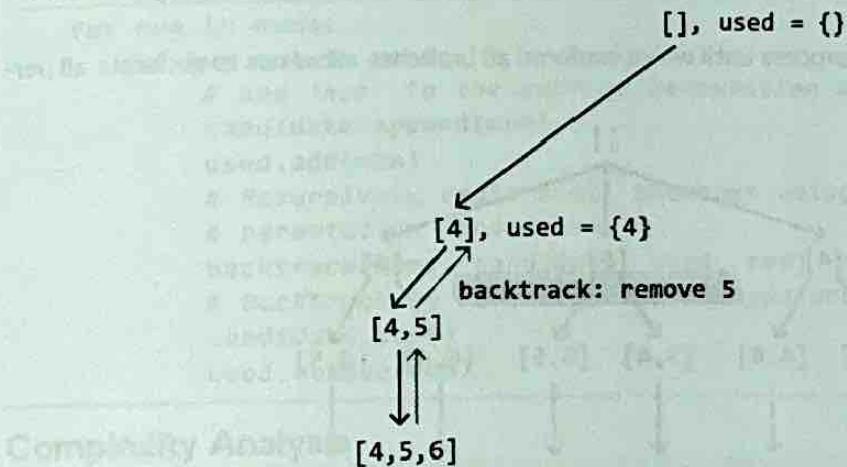
```
[], used = {}
```

Whenever a permutation is found, it is added to the results.

Implementation

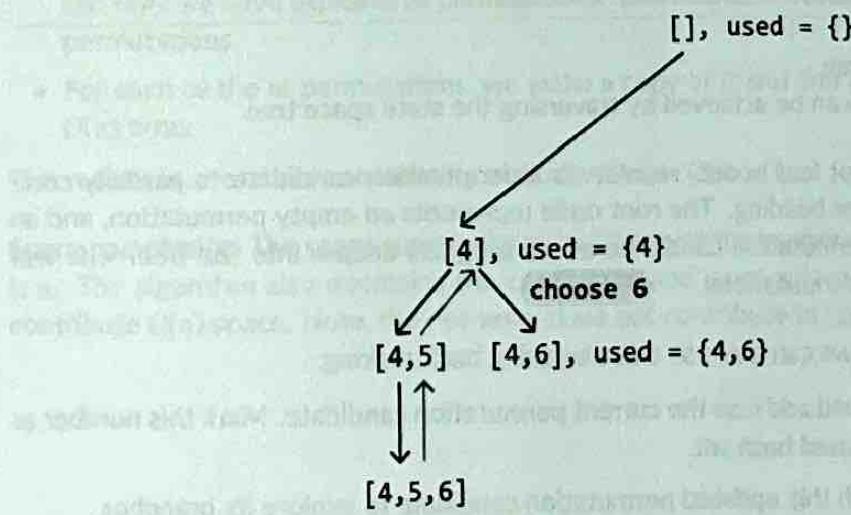


Are there any other numbers we can append to [4, 5]? Well, 6 is the only option at this point, which we already explored. So, let's backtrack again by removing 5, bringing us back to [4]:

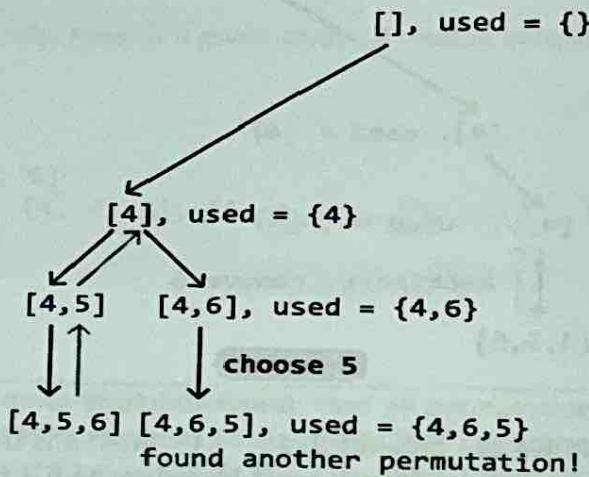


Complexity Analysis

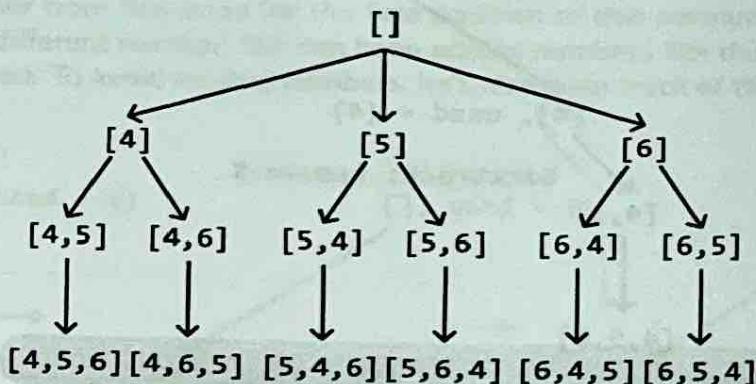
Are there any numbers other than 5 we can add to [4] at this point? Yes, we can use 6, so let's add it and continue searching:



The only number we can use at this point is 5, so let's add it to [4, 6], giving us another permutation:



Following this backtracking process until we've explored all branches allows us to generate all permutations:



Every time we reach a permutation (i.e., when the permutation we're building reaches a size of n , where n denotes the length of the input array), add it to our output.

Traversing the state space tree

Generating all permutations can be achieved by traversing the state space tree.

Each node in this tree, except leaf nodes, represents a permutation candidate: a partially completed permutation that we're building. The root node represents an empty permutation, and an element is added to each permutation candidate as we progress deeper into the tree. The leaf nodes represent completed permutations.

Starting from the root node, we can traverse this tree using backtracking:

1. Pick an unused number and add it to the current permutation candidate. Mark this number as used by adding it to the used hash set.
2. Make a recursive call with this updated permutation candidate to explore its branches.
3. Backtrack: remove the last number we added to the current candidate array, and the used

hash set.

Whenever a permutation candidate reaches the length of n , add it to our output.

Implementation

```
def find_all_permutations(nums: List[int]) -> List[List[int]]:
    res = []
    backtrack(nums, [], set(), res)
    return res

def backtrack(nums: List[int], candidate: List[int], used: Set[int],
             res: List[List[int]]) -> None:
    # If the current candidate is a complete permutation, add it to the
    # result.
    if len(candidate) == len(nums):
        res.append(candidate[:])
        return
    for num in nums:
        if num not in used:
            # Add 'num' to the current permutation and mark it as used.
            candidate.append(num)
            used.add(num)
            # Recursively explore all branches using the updated
            # permutation candidate.
            backtrack(nums, candidate, used, res)
            # Backtrack by reversing the changes made.
            candidate.pop()
            used.remove(num)
```

Complexity Analysis

Time complexity: The time complexity of `find_all_permutations` is $O(n \cdot n!)$. Here's why:

- Starting from the root, we recursively explore n candidates.
- For each of these n candidates, we explore $n - 1$ more candidates, then $n - 2$ more candidates, etc, until we have explored all permutations. This results in a total of $n \cdot (n-1) \cdot (n-2) \cdots 1 = n!$ permutations.
- For each of the $n!$ permutations, we make a copy of it and add it to the output, which takes $O(n)$ time.

This results in a total time complexity of $O(n!) \cdot O(n) = O(n \cdot n!)$.

Space complexity: The space complexity is $O(n)$ because the maximum depth of the recursion tree is n . The algorithm also maintains the candidate and used data structures, both of which also contribute $O(n)$ space. Note, the `res` array does not contribute to space complexity.

Find All Subsets

Return all possible subsets of a given set of unique integers. Each subset can be ordered in any way, and the subsets can be returned in any order.

Example:

Input: `nums = [4, 5, 6]`

Output: `[[], [4], [4, 5], [4, 5, 6], [4, 6], [5], [5, 6], [6]]`

Intuition

The key intuition for solving this problem lies in understanding that each subset is formed by making a specific decision for every number in the input array: to include the number, or exclude it. For example, from the array `[4, 5, 6]`, the subset `[4, 6]` is created by including 4, excluding 5, and including 6.

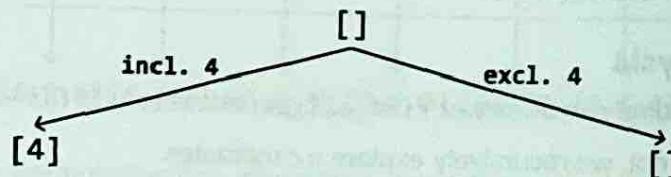
Let's have a look at what the state space tree looks like when making this decision for every element.

State space tree

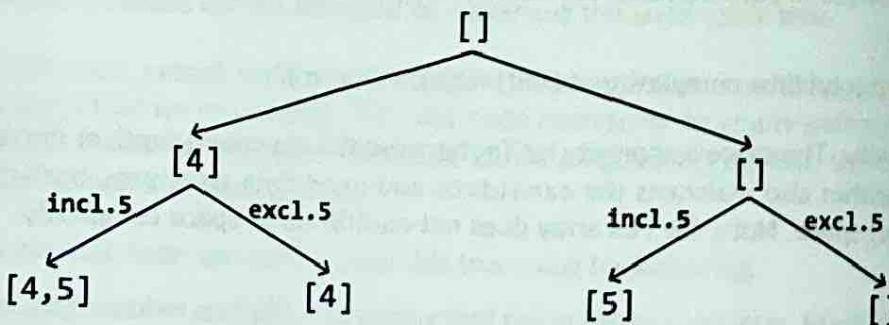
Consider the input array `[4, 5, 6]`. Let's start with the root node of the tree, which is an empty subset:

`[]`

To figure out how we branch out from here, let's consider our decision of whether to include or exclude an element. Let's make this decision with the first element of the input array, 4:

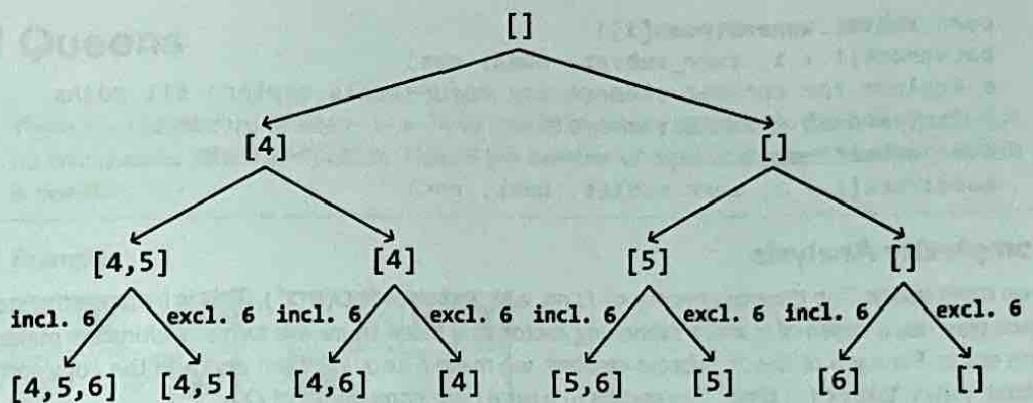


For each of these subsets, we repeat the process, branching out again based on the same choice for the second element: include or exclude it:

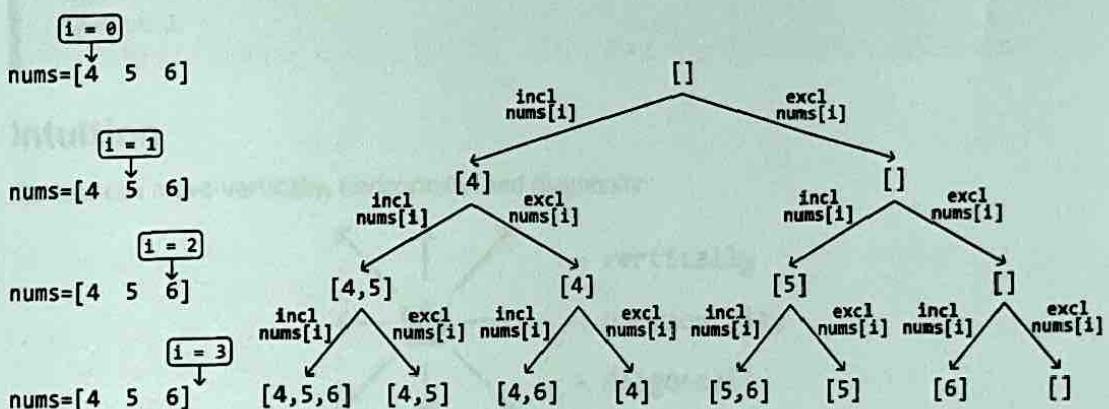


Finally, for the third element, we continue branching out for each existing subset based on whether we include or exclude this element:

N Queens



One important thing missing from this state space tree is a way to tell which element of the input array we're making a decision on at each node of the tree. We can use an index, i , for this:



As shown, the final level of the tree (i.e., when $i == n$, where n denotes the length of the input array) contains all the subsets of the input array. We can add each of these subsets to our output. To get to these subsets, we need to traverse the tree, and backtracking is great for this.

Implementation

```
def find_all_subsets(nums: List[int]) -> List[List[int]]:
    res = []
    backtrack(0, [], nums, res)
    return res

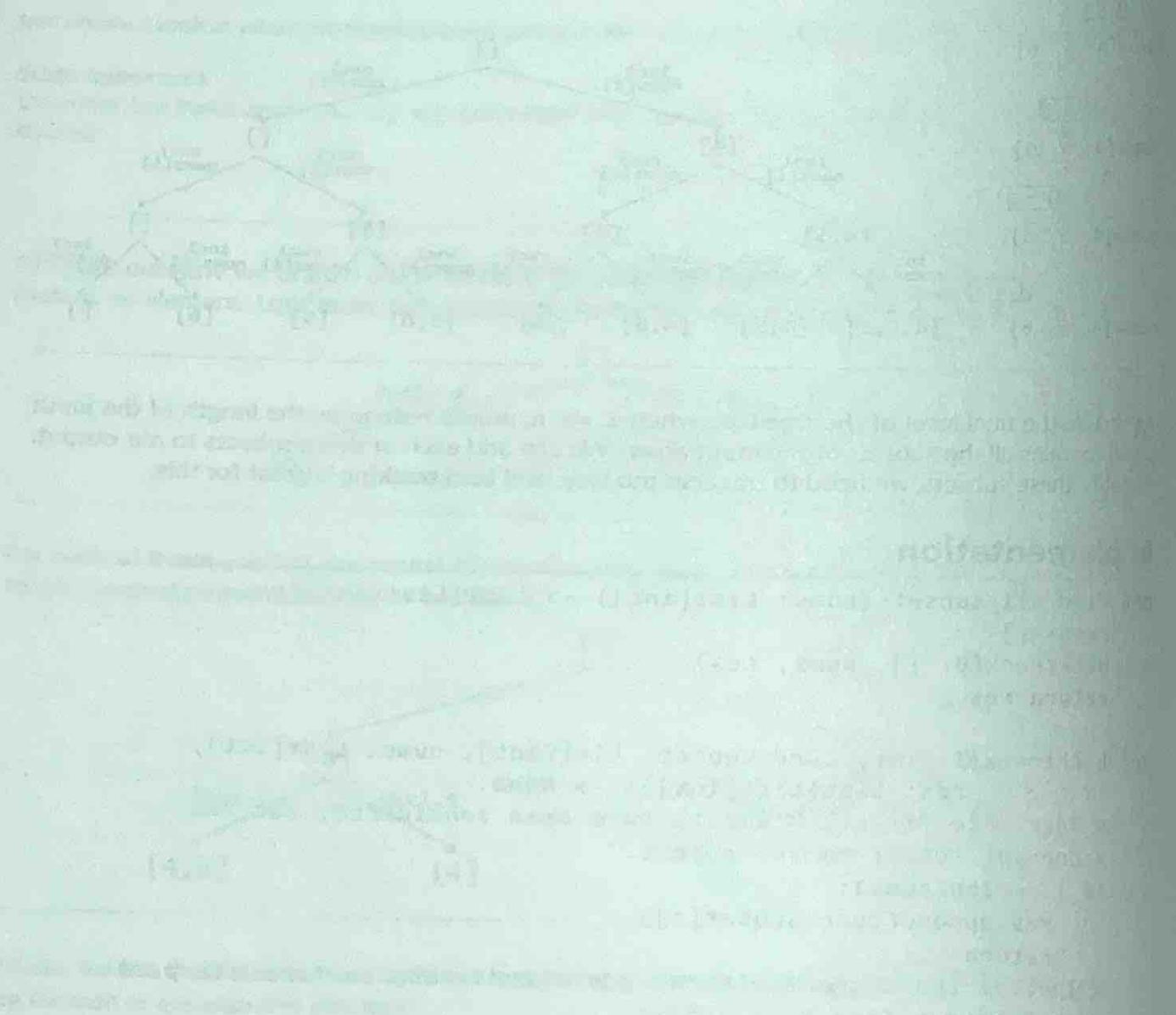
def backtrack(i: int, curr_subset: List[int], nums: List[int],
             res: List[List[int]]) -> None:
    # Base case: if all elements have been considered, add the
    # current subset to the output.
    if i == len(nums):
        res.append(curr_subset[:])
        return
    # Include the current element and recursively explore all paths
    # that branch from this subset.
    for j in range(i, len(nums)):
        curr_subset.append(nums[j])
        backtrack(i + 1, curr_subset, nums, res)
        curr_subset.pop()
```

```
curr_subset.append(nums[i])
backtrack(i + 1, curr_subset, nums, res)
# Exclude the current element and recursively explore all paths
# that branch from this subset.
curr_subset.pop()
backtrack(i + 1, curr_subset, nums, res)
```

Complexity Analysis

Time complexity: The time complexity of `find_all_subsets` is $O(n \cdot 2^n)$. This is because the state space tree has a depth of n and a branching factor of 2 since there are two decisions we make at each state. For each of the 2^n subsets created, we make a copy of them and add the copy to the output, which takes $O(n)$ time. This results in a total time complexity of $O(n \cdot 2^n)$.

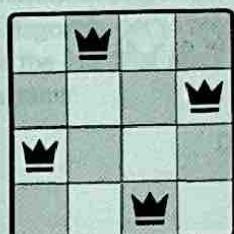
Space complexity: The space complexity is $O(n)$ because the maximum depth of the recursion tree is n . The algorithm also maintains the `curr_subset` data structure, which also contributes $O(n)$ space. Note, the `res` array does not contribute to space complexity.



N Queens

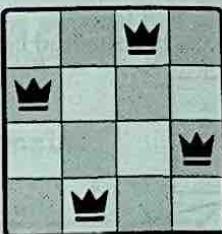
There is a chessboard of size $n \times n$. Your goal is to place n queens on the board such that no two queens attack each other. Return the number of distinct configurations where this is possible.

Example:



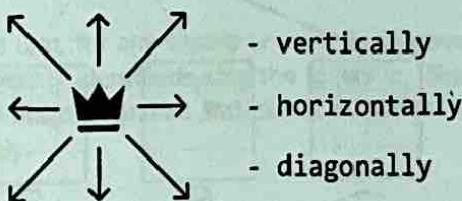
Input: $n = 4$

Output: 2



Intuition

Queens can move vertically, horizontally, and diagonally:



So, it's only possible to place a queen on a square of the board when:

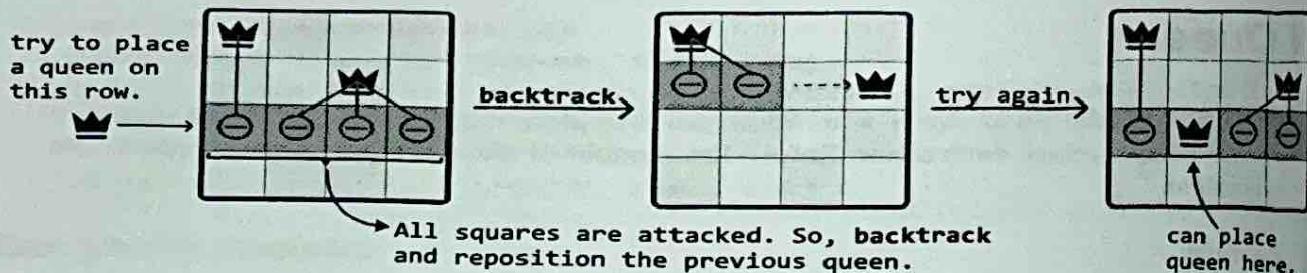
- No other queen occupies the same row of that square.
- No other queen occupies the same column of that square.
- No other queen occupies either diagonal of that square.

Based on this, let's identify a method for placing the queens.

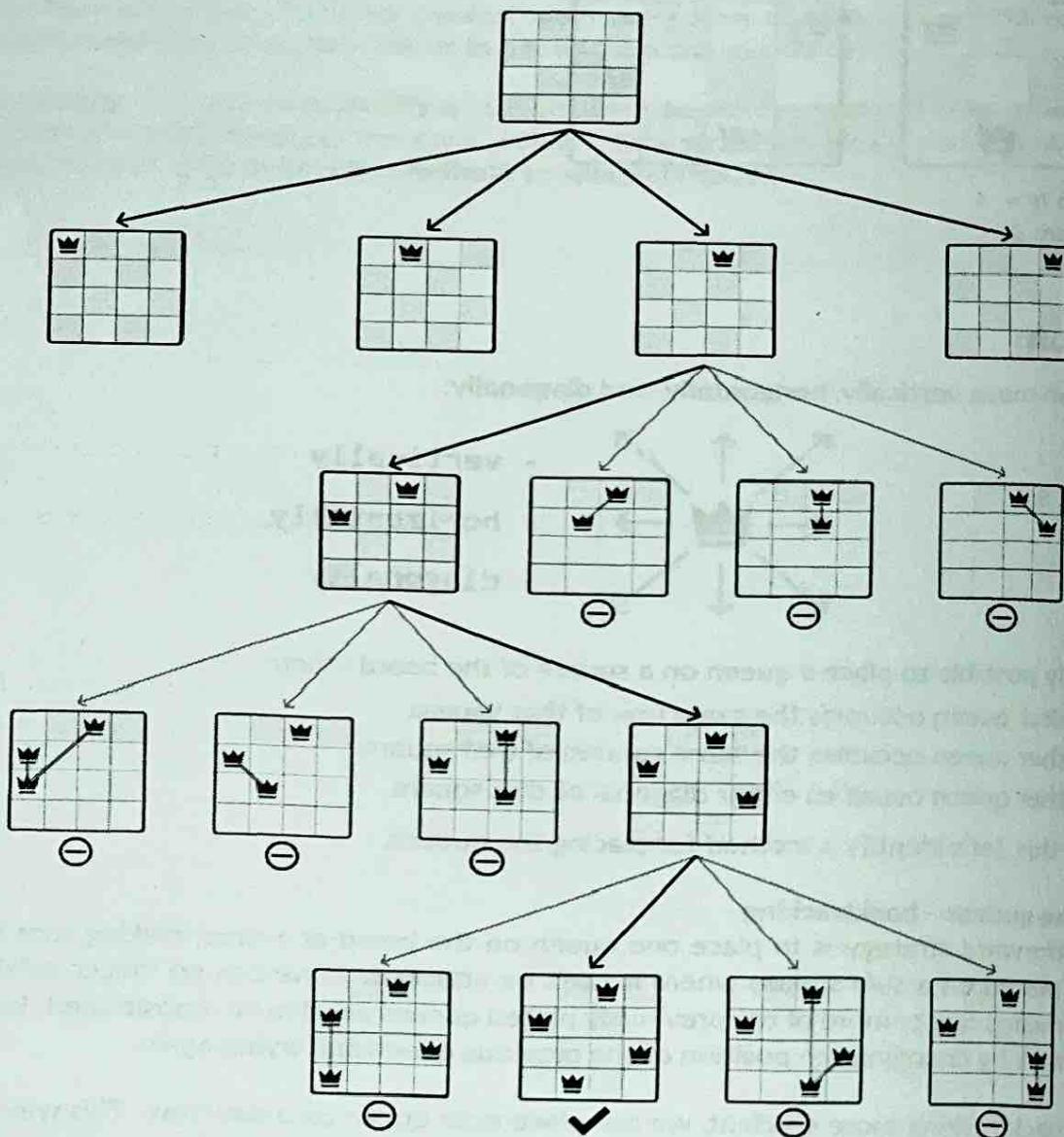
Placing the queens - backtracking

A straightforward strategy is to place one queen on the board at a time, making sure each new queen is placed on a safe square where it can't be attacked. If we can no longer safely place a queen, it means one or more of the previously placed queens need to be repositioned. In this case, we backtrack by changing the position of the previous queen and trying again.

To make backtracking more efficient, we can place each queen on a new row. This way, we don't have to worry about conflicts between queens on the same row, and only need to check for an opposing queen on the same column and along the diagonals of the square where the new queen is placed. If a queen cannot be placed anywhere on this new row, we backtrack, reposition the previous row's queen, and then try again:



A partial state space tree for this backtracking process is visualized below for $n = 4$:



We're still left with some questions. In particular, how can we tell if a square is being attacked, and how exactly do we 'place' or 'remove' a queen?

Detecting opposing queens

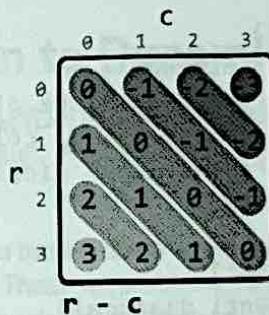
One challenge in this problem is determining if a square is attacked by another queen. We could do a linear search across the row, column, and diagonals every time we want to place a new queen,

but this is quite inefficient. A key observation is that we don't necessarily need to know the exact positions of all the queens. We only need to know if there exists a queen in any given square's row, column, or diagonals. We can use hash sets to efficiently check for this.

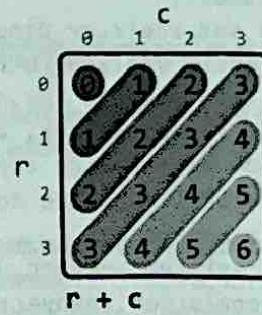
Note that we don't need a hash set for rows because we always place each queen on a different row. For columns, whenever we place a new queen on a square (r, c) , we can add that square's column id (c) to a column hash set.

What about diagonals? How can we determine which diagonal we're on? Since there are two types of diagonals, let's refer to the diagonal that goes from top-left to bottom-right as the 'diagonal', and the one that goes from top-right to bottom-left as the 'anti-diagonal'. Consider the following diagrams:

Diagonals:



Anti-diagonals:



The key observation here is that, for any square (r, c) , its diagonal can be identified using the id $r - c$, and its anti-diagonal is identified using the id $r + c$. Similarly to how we keep track of column ids, we can use a diagonal and an anti-diagonal hash set to keep track of diagonal and anti-diagonal ids, respectively.

Placing and removing a queen

Now that we have a way to identify opposing queens, we know the action of 'placing' a queen means adding its column, diagonal, and anti-diagonal ids to their respective hash sets. Inversely, to remove a queen, we just remove those exact ids from the hash sets.

Implementation

Note, this implementation uses a global variable as it leads to a more readable solution. However, it's important to confirm with your interviewer whether global variables are acceptable.

```
res = 0

def n_queens(n: int) -> int:
    dfs(0, set(), set(), set(), n)
    return res

def dfs(r: int, diagonals_set: Set[int], anti_diagonals_set: Set[int],
       cols_set: Set[int], n: int) -> None:
    global res
    # Termination condition: If we have reached the end of the rows,
    # we've placed all 'n' queens.
```

```

if r == n:
    res += 1
    return
for c in range(n):
    curr_diagonal = r - c
    curr_anti_diagonal = r + c
    # If there are queens on the current column, diagonal or
    # anti-diagonal, skip this square.
    if (c in cols_set or curr_diagonal in diagonals_set or
        curr_anti_diagonal in anti_diagonals_set):
        continue
    # Place the queen by marking the current column, diagonal, and
    # anti-diagonal as occupied.
    cols_set.add(c)
    diagonals_set.add(curr_diagonal)
    anti_diagonals_set.add(curr_anti_diagonal)
    # Recursively move to the next row to continue placing queens.
    dfs(r + 1, diagonals_set, anti_diagonals_set, cols_set, n)
    # Backtrack by removing the current column, diagonal, and
    # anti-diagonal from the hash sets.
    cols_set.remove(c)
    diagonals_set.remove(curr_diagonal)
    anti_diagonals_set.remove(curr_anti_diagonal)

```

Complexity Analysis

Time complexity: The time complexity of `n_queens` is $O(n!)$. Here's why:

- For the first queen, there are n choices for its position.
- For the second queen, there are $n - a$ choices for its position, where a denotes the number of squares on the second row attacked by the first queen.
- The third queen has $n - b$ choices, where b denotes the number of squares on the third row attacked by the previous two queens, and $b < a$.
- This process continues for subsequent queens, resulting in a total of $n \cdot (n - a) \cdot (n - b) \cdot \dots \cdot 1$ choices. Even though this doesn't exactly equate to $n! (n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1)$, this trend approximately results in a factorial growth of the search space.

Space complexity: The space complexity is $O(n)$ because the maximum depth of the recursion tree is n . The hash sets also contribute to this space complexity because they each store up to n values.

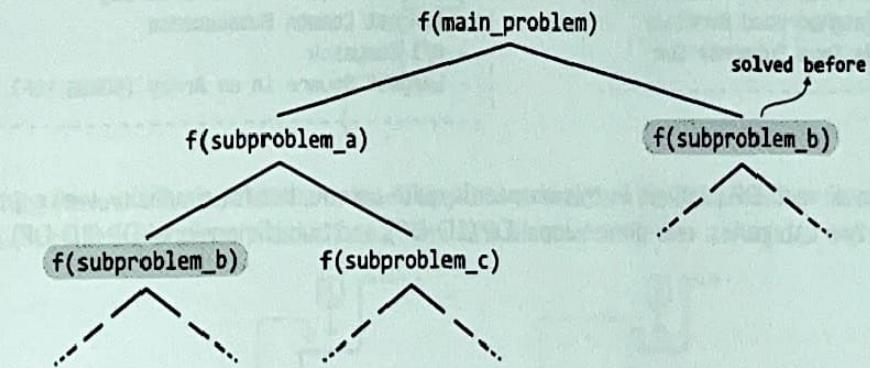
Dynamic Programming

Introduction to Dynamic Programming

Dynamic programming (DP) may seem daunting at first, but we'll break it down into manageable concepts and techniques. First, let's get an idea of what DP aims to do by considering the bigger picture.

Some problems can be broken down into subproblems, where each subproblem is a smaller version of the main problem. These subproblems may themselves be broken down into more subproblems as well. This isn't a foreign concept to us. Recursion is often used to solve problems like these, where we make recursive calls to solve each subproblem.

However, in the recursive process, it's possible to generate and solve the same subproblem multiple times, which can be unnecessarily expensive.



DP is the antidote to this. It's a technique that stores solutions to each subproblem, so they can be reused when they're needed again. In other words, it's an efficient tool that ensures each subproblem is solved at most one time. This can greatly increase the performance of an algorithm.

The DP process

DP is often perceived as challenging, but it follows a pretty consistent problem-solving process, best demonstrated with an example. First, we'll briefly explain some key DP terms, and then dive into the *Climbing Stairs* problem to learn how to identify a DP problem and develop a DP solution.

- **Optimal substructure:** the optimal solution to a problem can be constructed from the optimal solutions to its subproblems.
- **Overlapping subproblems:** if the same subproblems are solved repeatedly during the problem-

solving process.

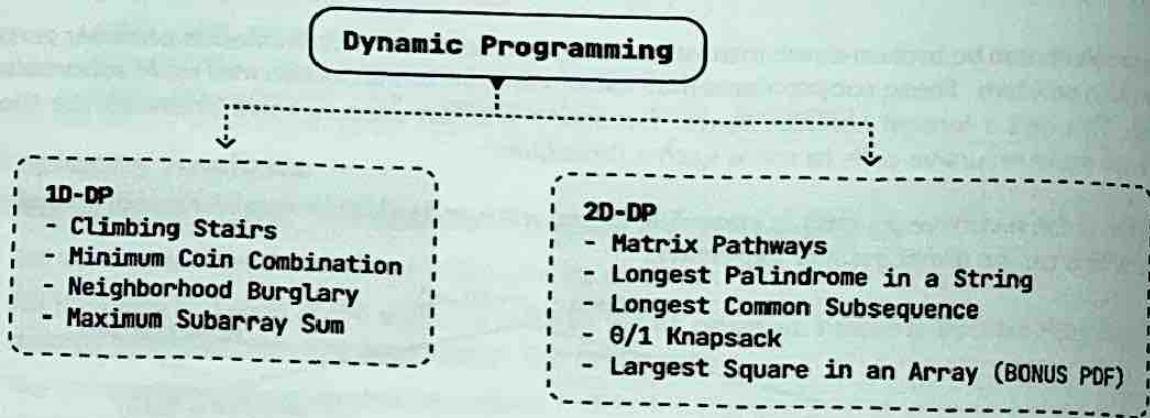
- **Recurrence relation:** a formula that expresses the solution to the problem in terms of the solutions to its subproblems.
- **Base cases:** the simplest instances of the problem where the solution is already known, without needing to be decomposed into more subproblems.

The first two terms are essential attributes that a problem must have to be solvable using DP. The last two terms are essential components in every DP solution. These definitions may seem abstract now, but keep them in mind as they will become clearer in the context of a problem.

Real-world Example

Word segmentation: Search engines use DP in a process called “word segmentation.” When users enter a search query without spaces, DP is employed to determine if white spaces can be added to form valid words. For example, given a query without spaces (like “bestrestaurants”), DP checks all possible ways to insert spaces (“best restaurants,” “best rest aunts”) by solving each segment (subproblem) separately, and storing their solutions to avoid recalculation.

Chapter Outline

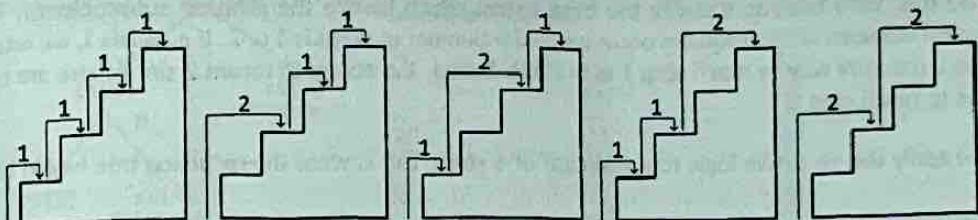


The nature of each DP problem in this chapter is quite unique, but for simplicity, we've grouped them into two categories: one-dimensional DP (1D-DP), and two-dimensional DP (2D-DP).

Climbing Stairs

Determine the number of distinct ways to climb a staircase of n steps by taking either 1 or 2 steps at a time.

Example:

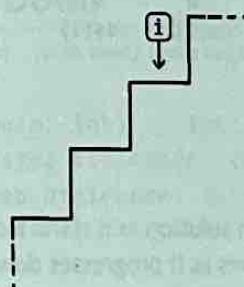


Input: $n = 4$

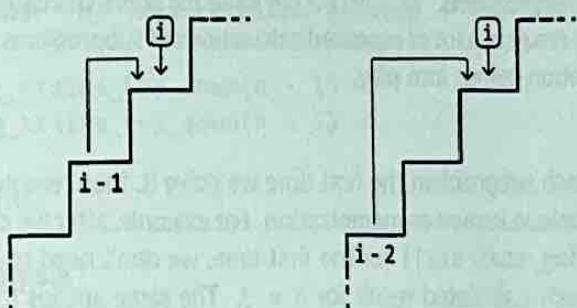
Output: 5

Intuition - Top-Down

A brute force solution to this problem is to go through all possible combinations of moving 1 or 2 steps up the stairs until reaching the top. How would we do this? Think about how to get to stair i :



One thing we know for sure is that to reach step i , we need to reach it from either step $i - 1$, or step $i - 2$ since we can only climb 1 or 2 steps at a time:



This is all the information we need. If we want to know all the different ways we can get to step i , we just need to know:

1. The number of ways to get to step $i - 1$ (`climbing_stairs(i - 1)`).
2. The number of ways to get to step $i - 2$ (`climbing_stairs(i - 2)`).

This highlights that this problem has an optimal substructure where, in order to solve

`climbing_stairs(n)`, we need the answers to two of its subproblems. We can translate this to a recurrence relation:

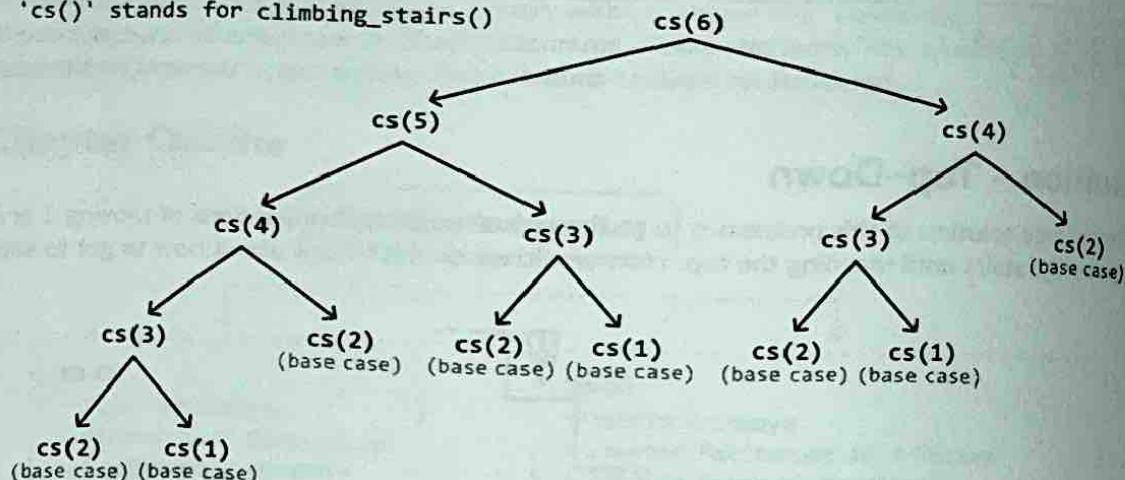
$$\text{climbing_stairs}(n) = \text{climbing_stairs}(n - 1) + \text{climbing_stairs}(n - 2)$$

Let's first implement this using recursion.

To do this, we'll need to identify the **base cases**, which handle the simplest subproblems. The simplest versions of this problem occur when the number of steps is 1 or 2. If n equals 1, we return 1 since the only way to reach step 1 is to climb 1 step. If n equals 2, return 2 since there are two ways to reach step 2.

If we apply this recursive logic to a staircase of 6 steps, this is what the recursion tree would look like:

'`cs()`' stands for `climbing_stairs()`

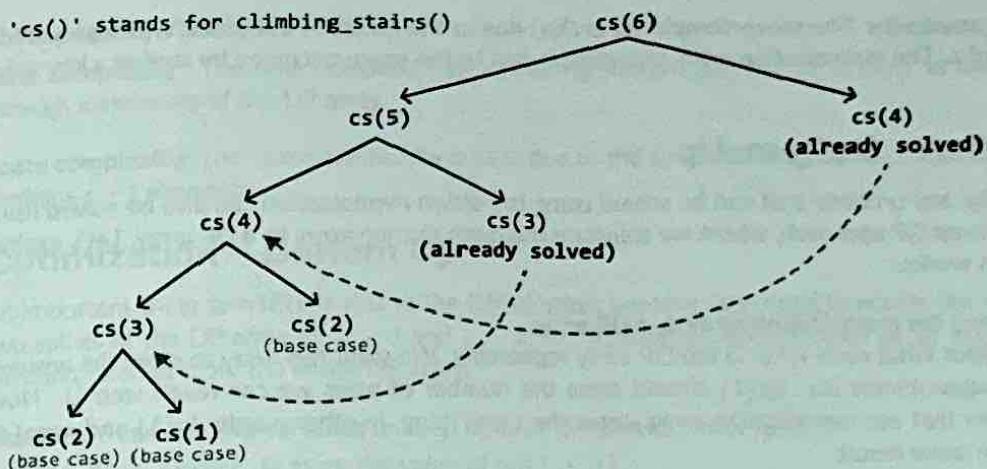


This solution is considered a **top-down solution** as it starts from the main problem, and recursively breaks it down into smaller subproblems as it progresses down the recursive tree.

You may have noticed in the recursion tree that we do some repeated work by calling the same subproblem multiple times (e.g., `climbing_stairs(4)` is called twice). This highlights the existence of **overlapping subproblems**. This isn't a big issue for short staircases, but for a taller one with more steps, it can result in a lot of repeated calculations of subproblems we've already solved. This is where memoization comes into play.

Memoization

Storing the result of each subproblem the first time we solve it, then reusing these stored results when needed, is a technique known as memoization. For example, after we calculate the subproblem of $n = 3$ (`climbing_stairs(3)`) for the first time, we don't need to calculate it again; we can just fetch the already-calculated result for $n = 3$. The same applies to $n = 4$. This greatly reduces the size of the recursion tree:



We use a hash map for memoization to store the results of subproblems for constant-time access. For example, after calculating the result for the subproblem $n = 3$, we store the result in the hash map as a value, where the key is 3.

As we can see, we've successfully implemented a DP solution using top-down memoization. We identified the subproblems, used them to create the recurrence relation, specified our base cases, and applied memoization to ensure each subproblem is solved only once.

Implementation – Top-Down

```

memo = {}

def climbing_stairs_top_down(n: int) -> int:
    # Base cases: With a 1-step staircase, there's only one way to
    # climb it. With a 2-step staircase, there are two ways to climb it.
    if n <= 2:
        return n
    if n in memo:
        return memo[n]
    # The number of ways to climb to the n-th step is equal to the sum
    # of the number of ways to climb to step n - 1 and to n - 2.
    memo[n] = (
        climbing_stairs_top_down(n - 1) +
        climbing_stairs_top_down(n - 2)
    )
    return memo[n]

```

Complexity Analysis

Time complexity:

- Without memoization, the time complexity of `climbing_stairs_top_down` is $O(2^n)$ because the depth of the recursion tree is n , and its branching factor is 2 since we make 2 recursive calls at each point in the tree.
- With memoization, we ensure each subproblem is solved only once. Since there are n possible subproblems (one for each step from step 1 to step n), the time complexity is $O(n)$.

Space complexity: The space complexity is $O(n)$ due to the recursive call stack, which grows to a height of n . The memoization array also contributes to the space occupied by storing n key-value pairs.

Intuition – Bottom-Up

Generally, any problem that can be solved using top-down memoization can also be solved using a bottom-up DP approach, where we translate the memoization array to a DP array. Let's explore how this works.

Translating the memoization array to a DP array

Think about what each value in the DP array represents. We want this array to store the answers to our subproblems (i.e., $dp[i]$ should store the number of ways we can reach step i). Now, remember that our memoization array stores the same thing. In other words, $dp[i]$ and $\text{memo}[i]$ store the same result.

In our top-down implementation, the memoization stores results like so:

```
| memo[n] = climbing_stairs(n - 1) + climbing_stairs(n - 2)
```

However, if the results of `climbing_stairs(n - 1)` and `climbing_stairs(n - 2)` were already calculated and memoized, this is what would actually be going on:

```
| memo[n] = memo[n - 1] + memo[n - 2]
```

Now that we have this tabular relationship for the memoization array, simply change 'memo' to 'dp':

```
| dp[n] = dp[n - 1] + dp[n - 2]
```

We call this a **bottom-up solution** because we need to calculate the solutions to smaller subproblems before we can solve the larger ones. In other words, we "build up" to the main solution as opposed to the top-down solution, where we start with the main problem, n , and work our way down.

Base cases

Our base cases stay the same: the answers to $dp[1]$ and $dp[2]$ are 1 and 2, respectively.

Return statement

In our top-down solution, we return `memo[n]`. Since there's a one-to-one relationship between the memoization array and the DP array, we can just return `dp[n]` in our bottom-up solution.

Implementation – Bottom-Up

```
def climbing_stairs_bottom_up(n: int) -> int:
    if n <= 2:
        return n
    dp = [0] * (n + 1)
    # Base cases.
    dp[1], dp[2] = 1, 2
    # Starting from step 3, calculate the number of ways to reach each
    # step until the n-th step.
    for i in range(3, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
```

Complexity Analysis

Time complexity: The time complexity of `climbing_stairs_bottom_up` is $O(n)$ as we iterate through n elements of the DP array.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the DP array, which contains $n + 1$ elements.

Optimization – Bottom Up

An important thing to notice is that in the DP solution, we only ever need to access the previous two values of the DP array (at $i - 1$ and $i - 2$) to calculate the current value (at i). This means we don't need to store the entire DP array.

Instead, we can use two variables to keep track of the previous two values:

- `one_step_before`: to store the value of $dp[i - 1]$.
- `two_steps_before`: to store the value of $dp[i - 2]$.

As we iterate through the steps, we update these two variables to always hold the values for the previous two steps. This approach retains the time complexity of $O(n)$, while reducing space complexity to $O(1)$. The adjusted implementation is below:

```
def climbing_stairs_bottom_up_optimized(n: int) -> int:  
    if n <= 2:  
        return n  
    # Set 'one_step_before' and 'two_steps_before' as the base cases.  
    one_step_before, two_steps_before = 2, 1  
    for i in range(3, n + 1):  
        # Calculate the number of ways to reach the current step.  
        current = one_step_before + two_steps_before  
        # Update the values for the next iteration.  
        two_steps_before = one_step_before  
        one_step_before = current  
    return one_step_before
```

Interview Tip

Tip: If you're having trouble coming up with the bottom-up solution, try starting with the top-down solution.



Designing a top-down solution first is often easier because we can first identify the recurrence relation, and then apply memoization to optimize it. The bottom-up solution, on the other hand, requires considering both steps at the same time. In addition, a bottom-up solution starts by solving subproblems first, which can be less intuitive, whereas a top-down solution starts with the main problem before working downward.

Once you have a working top-down solution, you can translate it into a bottom-up solution as described in the intuition above. Over time, you'll get better at mapping a recurrence relation directly to a bottom-up tabular relation, allowing you to skip the top-down approach.

Minimum Coin Combination

You are given an array of coin values and a target amount of money. Return the minimum number of coins needed to total the target amount. If this isn't possible, return -1. You may assume there's an unlimited supply of each coin.

Example 1:

Input: coins = [1, 2, 3], target = 5
Output: 2

Explanation: Use one 2-dollar coin and one 3-dollar coin to make 5 dollars.

Example 2:

Input: coins = [2, 4], target = 5
Output: -1

Intuition – Top-Down

In this problem, there's no restriction on the number of coins we can use, which makes a brute force approach that tries every possible coin combination impossible, due to the infinite number of possible combinations. This indicates the need for a more efficient method.

Consider the example below:

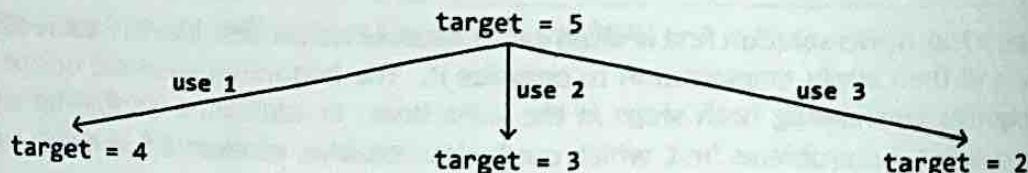
[1 2 3], target = 5

If we use a 3-dollar coin from the array, then we'll only need 2 dollars more to make 5. This gives us a new target: find the fewest number of coins needed to make 2 dollars:

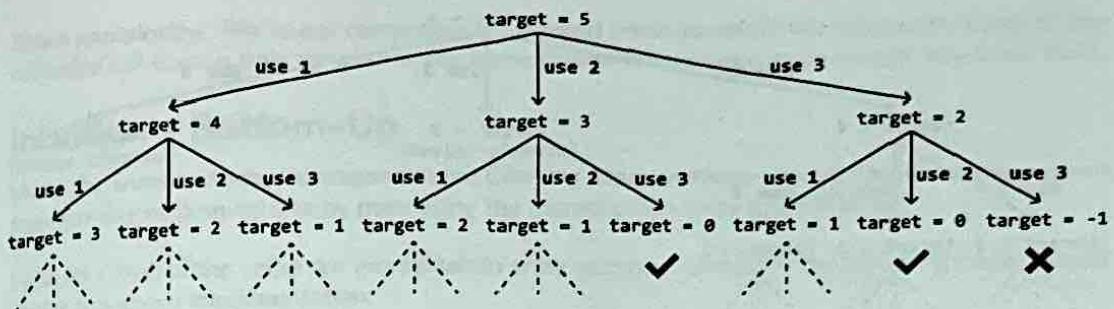
use
↓
[1 2 3], target = 5 → [1 2 3], target = 2

This indicates we've identified subproblems within the main problem, where each subproblem requires finding the fewest number of coins needed to make a smaller target.

Each coin we use creates a new subproblem. For example, using a 1-dollar coin changes our target from 5 to 4 dollars. Let's visualize how these smaller targets, representing new subproblems, are created after using each coin:



In extension, each of these subproblems can be solved by breaking them down into further subproblems:



A path that ends with a target of 0 means the coins used in that path add up to 5. If the target becomes negative, it means the path is invalid, so we should stop extending the path.

We've observed how new subproblems are created, but haven't yet addressed how to attain the solutions to them. Remember, each subproblem needs to return the minimum number of coins needed to reach its target.

Consider the main problem with a target of 5. To solve this, we first need to find the minimum number of coins needed to reach each of its three subproblems. The solution to the main problem is the smallest result among these subproblems, plus 1, to account for the coin used to create the subproblem. This highlights an **optimal substructure** in the problem, allowing us to define the following recurrence relation:

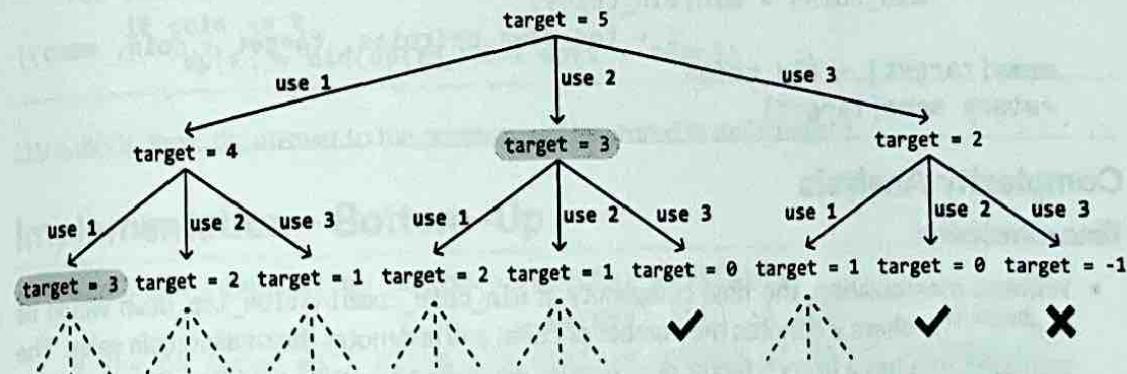
```
min_coin_combination(target) = 1 + min(min_coin_combination(target - coini) | coini ∈ coins)
```

Base case

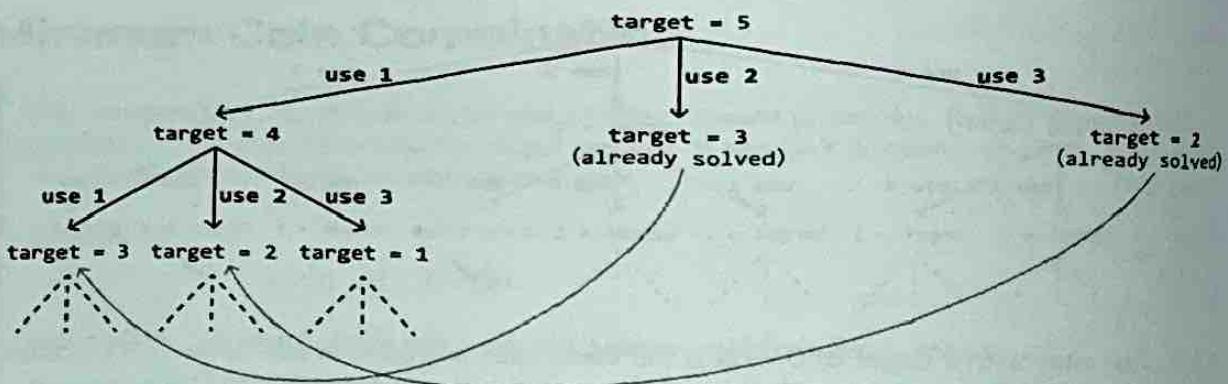
Naturally, we need a base case for this formula. The base case occurs when the target equals 0, which is the simplest version of this problem, as no coins are needed to meet the target. In this case, we return 0.

Memoization

An important thing to notice is that we might end up solving the same subproblem multiple times. For instance, we calculate the subproblem target = 3 two times in the previous example:



This highlights the existence of **overlapping subproblems**. Memoization improves our solution by storing the solutions to subproblems as they are computed, ensuring each subproblem is solved only once. This eliminates redundant calculations, and can significantly reduce the size of the recursion tree:



Implementation – Top-Down

```

def min_coin_combination_top_down(coins: List[int], target: int) -> int:
    res = top_down_dp(coins, target, {})
    return -1 if res == float('inf') else res

def top_down_dp(coins: List[int], target: int,
               memo: Dict[int, int]) -> int:
    # Base case: if the target is 0, then 0 coins are needed to reach
    # it.
    if target == 0:
        return 0
    if target in memo:
        return memo[target]
    # Initialize 'min_coins' to a large number.
    min_coins = float('inf')
    for coin in coins:
        # Avoid negative targets.
        if coin <= target:
            # Calculate the minimum number of coins needed if we use
            # the current coin.
            min_coins = min(min_coins,
                            1 + top_down_dp(coins, target - coin, memo))
    memo[target] = min_coins
    return memo[target]

```

Complexity Analysis

Time complexity:

- Without memoization, the time complexity of `min_coin_combination_top_down` would be $O(n^{\text{target}/m})$, where n denotes the number of coins, and m denotes the smallest coin value. The recursion tree has a branch factor of n because we make a recursive call for up to n coins. The depth of the tree is target/m because in the worst case, we continually reduce the target value by the smallest coin.
- With memoization, each subproblem is solved only once. Since there are at most target subproblems, and we iterate through all n coins for each subproblem, the time complexity is $O(\text{target} \cdot n)$.

Space complexity: The space complexity is $O(\text{target})$ because, while the maximum depth of the recursive call stack is only target/m , the memoization array stores up to target key-value pairs.

Intuition – Bottom-Up

Using the same technique discussed in the *Climbing Stairs* problem, we can convert our top-down solution to a bottom-up one by translating the memoization array to a DP array.

First, let's look at the value our memoization array stores, as shown in the following code snippet of the top-down implementation:

```
for coin in coins:  
    if coin <= target:  
        min_coins = min(min_coins,  
                         1 + top_down_dp(coins, target - coin, memo))  
memo[target] = min_coins
```

Translating this to a DP array provides the following code:

```
for coin in coins:  
    if coin <= target:  
        dp[target] = min(dp[target], 1 + dp[target - coin])
```

This code snippet only includes the calculation for one target value. In our top-down solution, this calculation is repeated for every target value from the initial target, down to the base case ($\text{target} == 0$).

In the bottom-up solution, we need to reverse this order by starting with the base case and working our way up to the initial target value (hence the name “bottom-up”). This is necessary because our DP array calculation depends on the DP values of smaller targets. So, we need to calculate the answers for smaller targets first. This can be done using a for-loop from 1 to the target (starting at 1 since the base case of 0 is already set):

```
for t in range(1, target + 1):  
    for coin in coins:  
        if coin <= t:  
            dp[t] = min(dp[t], 1 + dp[t - coin])
```

Once this is done, the answer to the problem will be stored in $\text{dp}[\text{target}]$.

Implementation – Bottom-Up

```
def min_coin_combination_bottom_up(coins: List[int],  
                                    target: int) -> int:  
    # The DP array will store the minimum number of coins needed for  
    # each amount. Set each element to a large number initially.  
    dp = [float('inf')] * (target + 1)  
    # Base case: if the target is 0, then 0 coins are needed.  
    dp[0] = 0  
    # Update the DP array for all target amounts greater than 0.
```

```
for t in range(1, target + 1):
    for coin in coins:
        if coin <= t:
            dp[t] = min(dp[t], 1 + dp[t - coin])
return dp[target] if dp[target] != float('inf') else -1
```

Complexity Analysis

Time complexity: The time complexity of `min_coin_combination_bottom_up` is $O(\text{target} \cdot n)$ because we loop through all n coins for each value between 1 and `target`.

Space complexity: The space complexity is $O(\text{target})$ due to the space occupied by the DP array, which is of size `target + 1`.

Interview Tip

Tip: When a problem asks for the minimum or maximum of something, it might be a DP problem.



If you spot keywords like 'minimum', 'maximum', 'longest', or 'shortest', in the problem description, consider whether a DP approach might be appropriate, as many DP problems involve optimizing a certain value, such as finding the *minimum cost*, or *longest sequence*.

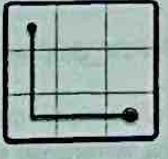
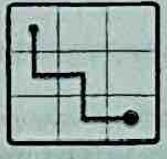
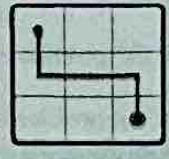
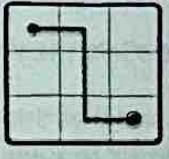
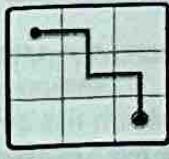
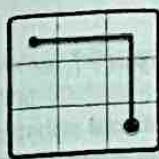
Complexity Analysis

`min_coin_combination_top_down` - `min_coin_combination_bottom_up`

Matrix Pathways

You are positioned at the top-left corner of a $m \times n$ matrix, and can only move downward or rightward through the matrix. Determine the number of unique pathways you can take to reach the bottom-right corner of the matrix.

Example:



Input: $m = 3$, $n = 3$

Output: 6

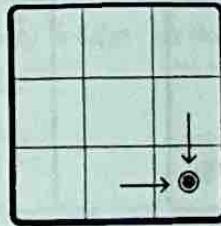
Constraints:

- $m, n \geq 1$

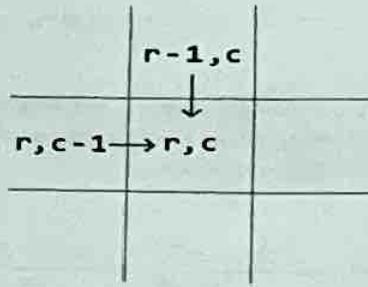
Intuition

At each cell, we can either move right or move down. No matter which direction we choose at any point, it will always move us closer to the destination. This means we just need to keep moving either right or down until we can no longer do so, at which point we've reached the bottom-right corner.

Let's think about this problem backward. Assume we have already reached the bottom-right corner. How did we get here? We know for certain we came from either the cell directly above, or the cell directly to the left of the current position.



This is equally true for any cell on the matrix, which means a generalization can be made: the number of paths to any cell is equal to the sum of the number of paths to the cell above it and the cell to its left.



`matrix_pathways(r, c) = matrix_pathways(r - 1, c) + matrix_pathways(r, c - 1)`

This demonstrates the existence of subproblems, and that this problem has an optimal substructure, where we need to solve two subproblems in order to solve the main problem. This makes this problem well-suited for DP. So, let's translate the above recurrence relation to a DP formula:

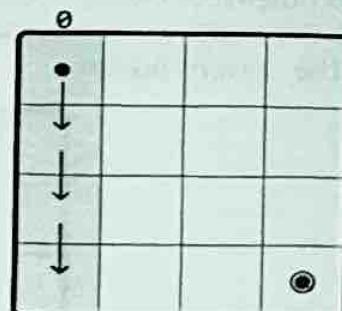
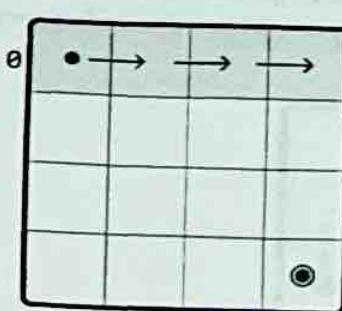
$dp[r][c] = dp[r - 1][c] + dp[r][c - 1]$, where $dp[r][c]$ represents the total number of paths that lead to cell (r, c) .

Before populating the DP table, we need to know our base cases.

Base cases

We know $dp[0][0]$ should be 1 because there's only one path leading to cell $(0, 0)$.

What else do we know for certain? Since we can only move right or down, once we leave a row or column, we can never return to it because we can't move left or up. This means, for any cell in row 0 or column 0, there's only one path to those cells:



Therefore, we can set all cells in row 0 and column 0 to 1 as the base cases.

Problem-solving tip: another way to identify row 0 and column 0 as base cases is by examining the DP formula. Since we need the values from row $r - 1$ and column $c - 1$ to populate $dp[r][c]$, all values at $r = 0$ and $c = 0$ must be pre-populated before using the formula to avoid index out-of-bound errors.

Populating the DP table

Once the base cases are set, we can populate the remaining DP table, starting from cell $(1, 1)$, using our DP formula ($dp[r][c] = dp[r - 1][c] + dp[r][c - 1]$):

	0	1	2	3
0	1	1	1	1
1	1			
2	1			
3	1			

	0	1	2	3
0	1	1	1	1
1	1	2	3	4
2	1	3	6	10
3	1	4	10	20

After we fill in the DP table, we can return $dp[m - 1][n - 1]$, which contains the number of paths to the bottom-right corner.

Implementation

```
def matrix_pathways(m: int, n: int) -> int:
    # Base cases: Set all cells in row 0 and column 0 to 1. We can
    # do this by initializing all cells in the DP table to 1.
    dp = [[1] * n for _ in range(m)]
    # Fill in the rest of the DP table.
    for r in range(1, m):
        for c in range(1, n):
            # Paths to current cell = paths from above + paths from
            # left.
            dp[r][c] = dp[r - 1][c] + dp[r][c - 1]
    return dp[m - 1][n - 1]
```

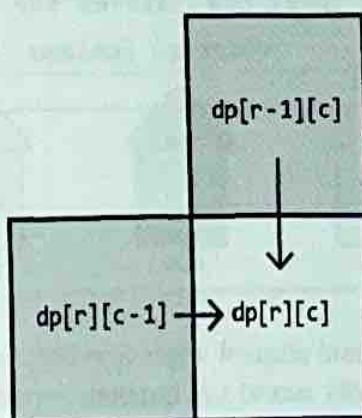
Complexity Analysis

Time complexity: The time complexity of `matrix_pathways` is $O(m \cdot n)$ because each cell in the DP table is populated once.

Space complexity: The space complexity is $O(m \cdot n)$ due to the DP table, which contains $m \cdot n$ elements.

Optimization

We can optimize our solution by understanding that, for each cell in the DP table, we only need to access the cells directly above it and to its left.



- To get the cell above it ($dp[r - 1][c]$), we only need access to the previous row.

- To get the cell to its left ($dp[r][c - 1]$), we just need to look at the cell to the left of the current cell, which is in the same row we're currently populating.

Therefore, we only need to maintain two rows:

- `prev_row`: the previous row.
- `curr_row`: the current row being populated.

	0	1	2	3
0	1	1	1	1
1	1	2	3	4
prev_row: 2	1	3	6	10
curr_row: 3	1	→	→	→

↓ ↓ ↓ ↓

← to calculate row 3, the
only other row we need
is row 2.

This effectively reduces the space complexity to $O(n)$ because we now only need to maintain two arrays of size n . After populating the DP values for the current row, we'll need to make sure to update `prev_row` with the values from `curr_row` to prepare for the next iteration since the next row's previous row is the current row. Below is the optimized code:

```

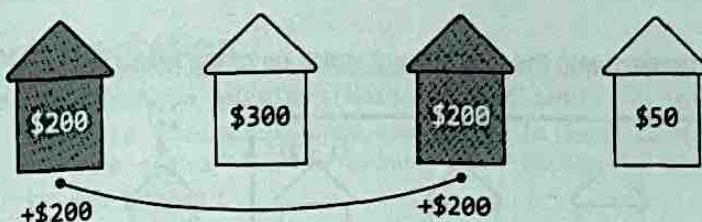
def matrix_pathways_optimized(m: int, n: int) -> int:
    # Initialize 'prev_row' as the DP values of row 0, which are all 1s.
    prev_row = [1] * n
    # Iterate through the matrix starting from row 1.
    for r in range(1, m):
        # Set the first cell of 'curr_row' to 1. This is done by
        # setting the entire row to 1.
        curr_row = [1] * n
        for c in range(1, n):
            # The number of unique paths to the current cell is the sum
            # of the paths from the cell above it ('prev_row[c]')
            # and
            # the cell to the left ('curr_row[c - 1]').
            curr_row[c] = prev_row[c] + curr_row[c - 1]
        # Update 'prev_row' with 'curr_row' values for the next
        # iteration.
        prev_row = curr_row
    # The last element in 'prev_row' stores the result for the
    # bottom-right cell.
    return prev_row[n - 1]

```

Neighborhood Burglary

You plan to rob houses in a street where each house stores a certain amount of money. The neighborhood has a security system that sets off an alarm when two adjacent houses are robbed. Return the maximum amount of cash that can be stolen without triggering the alarms.

Example:



Input: houses = [200, 300, 200, 50]

Output: 400

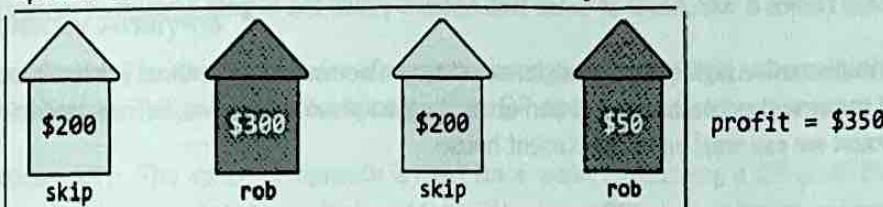
Explanation: Stealing from the houses at indexes 0 and 2 yields $200 + 200 = 400$ dollars.

Intuition

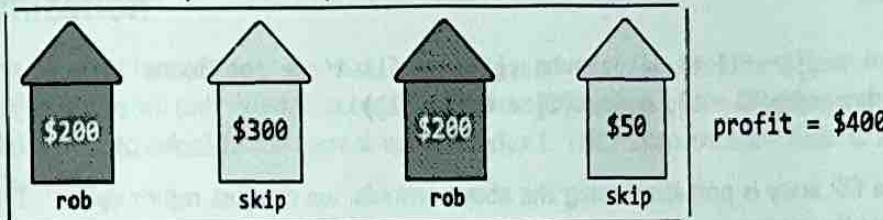
Ideally, we would want to rob every house, and collect the total sum of all cash contained therein. However, with the alarm system in place, we need to be more strategic about which houses to rob and which to skip.

A simple, greedy approach of always robbing the house with the most money fails because it overlooks the long-term consequences of its choices, and doesn't always yield the highest total profit. This is visualized in the example below.

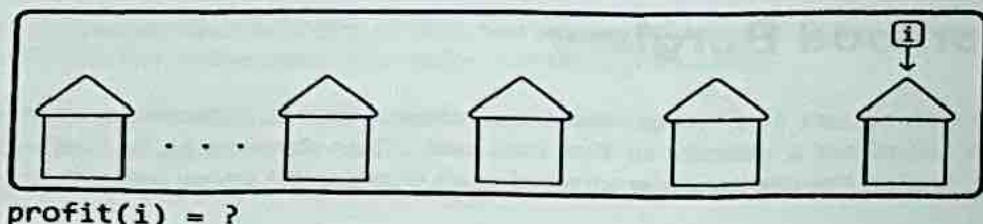
option 1: rob the house with the most money first



option 2: optimal solution

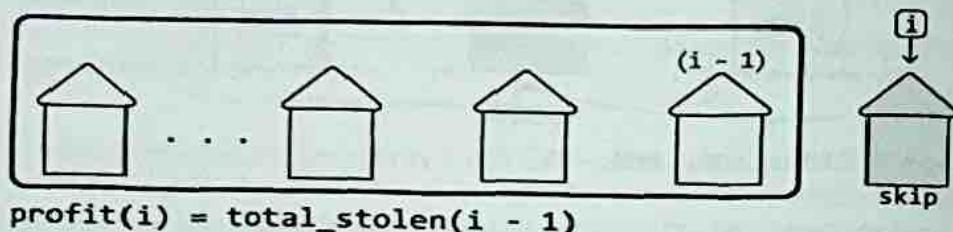


Let's approach this problem from a different angle. Imagine breaking into houses all along a street, and eventually reaching the last house, denoted as i below. How much money has been stolen up to this point?



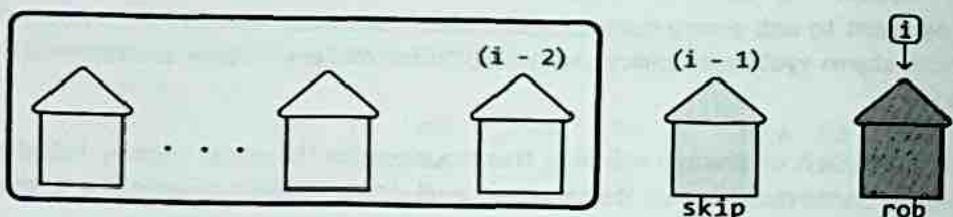
To answer this question, we need to consider the two choices that can be made at this last house: do we skip it or rob it?

- If we skip it, we end our burglary with the total amount stolen up to the house at $i - 1$:



$$\text{profit}(i) = \text{total_stolen}(i - 1)$$

- If we rob it, we couldn't have robbed the previous house at $i - 1$, so we end the burglary with the money stolen from this final house, plus the total amount stolen up to the house at $i - 2$, which is two houses back:



$$\text{profit}(i) = \text{houses}[i] + \text{total_stolen}(i - 2)$$

The optimal choice is whichever of these two options yields the largest amount of money.

Notice this discussion highlights the existence of subproblems, and an optimal substructure where we need to know the total amount stolen up to the two previous houses, before determining the total amount we can steal up to the current house.

This means we should try using DP to solve this problem. Let's say $\text{dp}[i]$ represents the maximum amount we're able to steal by the time we reach house i . Based on our previous discussion, we know that:

$$\begin{aligned} \text{dp}[i] &= \max(\text{profit if we skip house } i, \text{ profit if we rob house } i) \\ &= \max(\text{dp}[i - 1], \text{houses}[i] + \text{dp}[i - 2]) \end{aligned}$$

Once the DP array is populated using the above formula, we can just return $\text{dp}[n - 1]$, which represents the maximum amount that can be stolen once we reach the end of the street. Now, let's think about what our base cases should be.

Base cases

For starters, let's consider what to do if there's just one house. This is the simplest possible sub-

problem, where the total amount stolen is just the money in that house. So, one of our base cases is $dp[0] = houses[0]$.

Keep in mind that to use our DP formula ($dp[i] = \max(dp[i - 1], houses[i] + dp[i - 2])$), we need to access values at indexes $i - 1$ and $i - 2$. This means we must set initial values for index 0 and index 1. With these base cases, we can safely start using the formula from index 2 onward without causing any index out-of-bound errors. So, what's the most money that can be stolen at $i = 1$ (i.e., when there are just two adjacent houses)? We can only steal from one of these houses, so $dp[1] = \max(houses[0], houses[1])$.

Implementation

```
def neighborhood_burglary(houses: List[int]) -> int:
    # Handle the cases when the array is less than the size of 2 to
    # avoid out-of-bounds errors when assigning the base case values.
    if not houses:
        return 0
    if len(houses) == 1:
        return houses[0]
    dp = [0] * len(houses)
    # Base case: when there's only one house, rob that house.
    dp[0] = houses[0]
    # Base case: when there are two houses, rob the one with the most
    # money.
    dp[1] = max(houses[0], houses[1])
    # Fill in the rest of the DP array.
    for i in range(2, len(houses)):
        # 'dp[i]' = max(profit if we skip house 'i', profit if we rob
        # house 'i').
        dp[i] = max(dp[i - 1], houses[i] + dp[i - 2])
    return dp[len(houses) - 1]
```

Complexity Analysis

Time complexity: The time complexity of `neighborhood_burglary` is $O(n)$, where n denotes the number of houses. This is because each index of the DP array is populated at most once.

Space complexity: The space complexity is $O(n)$ since we're maintaining a DP array that has n elements.

Optimization

From the DP array formula $dp[i] = \max(dp[i - 1], houses[i] + dp[i - 2])$, an important observation is that we only need to access the previous two values of the DP array, index $i - 1$ and index $i - 2$, to calculate the current value at index i . This means we don't need to store the entire DP array.

Instead, we can use two variables to keep track of the previous two values:

- `prev_max_profit`: stores the value of $dp[i - 1]$.
- `prev_prev_max_profit`: stores the value of $dp[i - 2]$.

This optimization reduces the space complexity to $O(1)$ since we're no longer maintaining any aux-

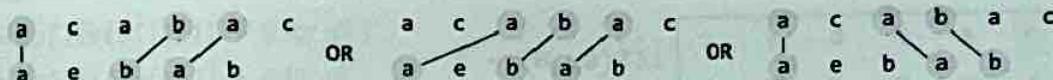
iliary data structures. The adjusted implementation can be seen below:

```
def neighborhood_burglary_optimized(houses: List[int]) -> int:
    if not houses:
        return 0
    if len(houses) == 1:
        return houses[0]
    # Initialize the variables with the base cases.
    prev_max_profit = max(houses[0], houses[1])
    prev_prev_max_profit = houses[0]
    for i in range(2, len(houses)):
        curr_max_profit = max(prev_max_profit,
                              houses[i] + prev_prev_max_profit)
        # Update the values for the next iteration.
        prev_prev_max_profit = prev_max_profit
        prev_max_profit = curr_max_profit
    return prev_max_profit
```

Longest Common Subsequence

Given two strings, find the length of their longest common subsequence (LCS). A subsequence is a sequence of characters that can be derived from a string by deleting zero or more elements, without changing the order of the remaining elements.

Example:



Input: $s_1 = \text{"acabac"}$, $s_2 = \text{"aebab"}$

Output: 3

Intuition

A naive approach to this problem is to generate every possible subsequence for both strings and identify the LCS among them. This is extremely inefficient, so we need to think of something better.

One way to think about this problem is to realize that for any character from either string, we have a choice to either include it in the LCS, or exclude it. This will help us figure out the next steps in finding the length of the LCS.

Let's start by considering the first character of each string and whether we should include or exclude them. There are two primary cases to discuss:

- Case 1: the characters are the same.
- Case 2: the characters are different.

Case 1: equal characters

Consider the following two strings, where we're trying to find the length of their LCS, starting from index 0 of each string ($\text{LCS}(0, 0)$):

a	c	a	b	a	c
a	e	b	a	b	

$\text{LCS}(0, 0) = ?$

The first characters of these two strings are equal. What should we do about them? We should include these characters in the LCS as they form the beginning of a common subsequence. Including them also means our LCS will have a length of at least 1. But how do we find the length of the rest of the LCS? We can do this by computing the LCS of the remainder of both strings. That is, the LCS of their substrings starting at index 1 ($\text{LCS}(1, 1)$):

a	c	a	b	a	c
a	e	b	a	b	

$\text{LCS}(0, 0) = 1 + \text{LCS}(1, 1)$

equal

We've just identified that this case can be solved by solving a subproblem that also computes the LCS of two strings, indicating this problem has an optimal substructure.

Therefore, we can generalize a recurrence relation for this case. Below, index i and j represent the start of the substring of s_1 and s_2 , respectively.

```
if  $s_1[i] == s_2[j]$ :  
     $LCS(i, j) = 1 + LCS(i + 1, j + 1)$ 
```

Case 2: different characters

Now, let's say the first characters of the two strings are different:

a	c	a	b	a	c
k	e	b	a	b	

↓
different

$$LCS(0, 0) = ?$$

This means the LCS cannot include both of these characters. It could include one of them, but certainly not both. Therefore, we have two choices to find the LCS:

1. Exclude 'a' from the first string to find the LCS between the two strings after this exclusion:

a	c	a	b	a	c
k	e	b	a	b	

$$LCS(0, 0) = LCS(1, 0)$$

2. Exclude 'k' from the second string to find the LCS between the two strings after this exclusion:

a	c	a	b	a	c
k	e	b	a	b	

$$LCS(0, 0) = LCS(0, 1)$$

The length of the LCS will be the larger length between these two options. Again, here we see that we're dealing with a problem of optimal substructure. The recurrence relation for this case is:

```
if  $s_1[i] != s_2[j]$ :  
     $LCS(i, j) = \max(LCS(i + 1, j), LCS(i, j + 1))$   
    (i.e.  $\max(LCS \text{ excluding } s_1[i], LCS \text{ excluding } s_2[j])$ )
```

Dynamic programming

Since we're dealing with overlapping subproblems, where the solutions to a subproblem can be used multiple times, we can convert our recurrence relation to a DP formula. Let's say $dp[i][j]$ represents $LCS(i, j)$. Based on our previous discussion, we know that:

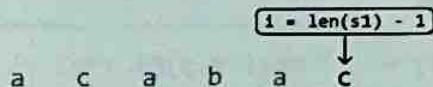
```
if  $s_1[i] == s_2[j]$ :  
     $dp[i][j] = 1 + dp[i + 1][j + 1]$   
else:  
     $dp[i][j] = \max(dp[i + 1][j], dp[i][j + 1])$ 
```

Now, we need to think about what the base cases should be.

Base cases

The simplest version of our problem is when one or both strings are empty. In this case, their LCS has a length of 0. But which values of the DP table should we populate for these base cases?

We know that when $i = \text{len}(s1) - 1$, only one character of $s1$ is being considered:



This implies that when $i = \text{len}(s1)$, the substring of $s1$ contains no characters. The equivalent is true for $s2$ when $j = \text{len}(s2)$. Therefore, we can populate the DP table with the base case values like so:

- $\text{dp}[\text{len}(s1)][j] = 0$ for all j
- $\text{dp}[i][\text{len}(s2)] = 0$ for all i

Let's draw the DP table with just these base cases to get a better idea of what this looks like:

		s2					
		a	e	b	a	b	..
s1	a	0	end				0
	c	1					0
	a	2					0
	b	3					0
	a	4					0
	c	5				start	0
..		6	0	0	0	0	0

As we can see, the last row and last column are set to 0 for our base cases.

Populating the DP table

We populate the DP table starting from the smallest subproblems (excluding the base cases). Specifically, we begin by populating $\text{dp}[\text{len}(s1) - 1][\text{len}(s2) - 1]$, which considers the LCS of only the last character of each string. From there, we iteratively populate the DP table in reverse order, moving backward through the table until we reach cell $(0, 0)$.

Once the DP table is populated, we return $\text{dp}[0][0]$, which stores the length of the LCS between the entire first string and the entire second string.

Implementation

```
def longest_common_subsequence(s1: str, s2: str) -> int:  
    # Base case: Set the last row and last column to 0 by  
    # initializing the entire DP table with 0s.  
    dp = [[0] * (len(s2) + 1) for _ in range(len(s1) + 1)]  
    # Populate the DP table.  
    for i in range(len(s1) - 1, -1, -1):  
        for j in range(len(s2) - 1, -1, -1):  
            # If the characters match, the length of the LCS at
```

```

# 'dp[i][j]' is 1 + the LCS length of the remaining
# substrings.
if s1[i] == s2[j]:
    dp[i][j] = 1 + dp[i + 1][j + 1]
# If the characters don't match, the LCS length at
# 'dp[i][j]' can be found by either:
# 1. Excluding the current character of s1.
# 2. Excluding the current character of s2.
else:
    dp[i][j] = max(dp[i + 1][j], dp[i][j + 1])
return dp[0][0]

```

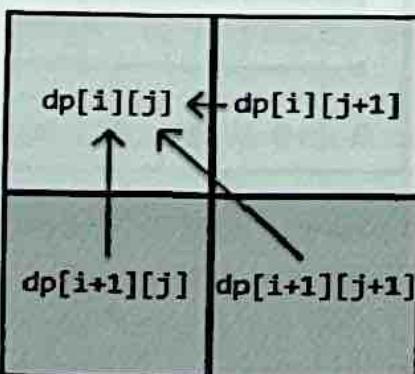
Complexity Analysis

Time complexity: The time complexity of `longest_common_subsequence` is $O(m \cdot n)$, where m and n denote the lengths of $s1$ and $s2$, respectively. This is because each cell in the DP table is populated once.

Space complexity: The space complexity is $O(m \cdot n)$ since we're maintaining a 2D DP table that has $(m + 1) \cdot (n + 1)$ elements.

Optimization

We can optimize our solution by noticing that for each cell in the DP table, we only need to access the cell below it, the cell to its right, and the bottom-right diagonal cell.



- To get the cell below it, we only need access to the row below.
- To get the cell to its right, we just need to look at the cell to the right of the current cell.
- To get the bottom-right diagonal cell, we also only need access to the row below.

Therefore, we only need to maintain two rows:

- `curr_row`: the current row being populated.
- `prev_row`: the row below the current row.

	s2					
	a	e	b	a	b	..
curr_row:	a ₀					0
prev_row:	c ₁	3	2	2	2	1 0
	a ₂	3	2	2	2	1 0
s1	b ₃	2	2	2	1	1 0
	a ₄	1	1	1	1	0 0
	c ₅	0	0	0	0	0 0
..	6	0	0	0	0	0 0

to calculate row 0,
the only other row
we need is row 1.

This effectively reduces the space complexity to $O(n)$. Below is the optimized code:

```
def longest_common_subsequence_optimized(s1: str, s2: str) -> int:
    # Initialize 'prev_row' as the DP values of the last row.
    prev_row = [0] * (len(s2) + 1)
    for i in range(len(s1) - 1, -1, -1):
        # Set the last cell of 'curr_row' to 0 to set the base case for
        # this row. This is done by initializing the entire row to 0.
        curr_row = [0] * (len(s2) + 1)
        for j in range(len(s2) - 1, -1, -1):
            # If the characters match, the length of the LCS at
            # 'curr_row[j]' is 1 + the LCS length of the remaining
            # substrings ('prev_row[j + 1]').
            if s1[i] == s2[j]:
                curr_row[j] = 1 + prev_row[j + 1]
            # If the characters don't match, the LCS length at
            # 'curr_row[j]' can be found by either:
            # 1. Excluding the current character of s1 ('prev_row[j]').
            # 2. Excluding the current character of s2
            # ('curr_row[j + 1]').
            else:
                curr_row[j] = max(prev_row[j], curr_row[j + 1])
        # Update 'prev_row' with 'curr_row' values for the next
        # iteration.
        prev_row = curr_row
    return prev_row[0]
```

Longest Palindrome in a String

Return the longest palindromic substring within a given string.

Example:

Input: $s = \text{"abccbab"}$

Output: "abccba"

Intuition

A naive solution to this problem is to check every possible substring and save the longest palindrome found. It takes approximately $O(n^2)$ time to generate all substrings for a string of length n , and for each of these substrings, it takes $O(n)$ time to check if it's a palindrome. This results in an overall time complexity of $O(n^3)$, which is expensive. So, we should consider a more efficient approach.

Determining if a substring is a palindrome

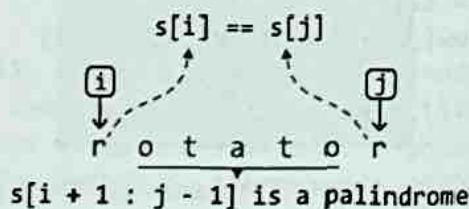
An important observation is that palindromes contain shorter palindromes within them. We can observe this in the string "rotator", for example:

$r \underline{o \ t \ a \ t \ o} r$
palindrome

This highlights a subproblem: to identify if a string is a palindrome, we can check if its inner substring is also a palindrome.

More specifically, a substring from index i to j is a palindrome given two conditions:

1. Its first and last characters are the same ($s[i] == s[j]$).
2. The substring from index $i + 1$ to $j - 1$ ($s[i + 1 : j - 1]$) is also a palindrome.



The only situation where this isn't true is when the substring is of length 0, 1, or 2, in which case there is no inner substring. We'll discuss these cases later.

This problem has an **optimal substructure** since we solve a subproblem to obtain the solution to the main problem. This indicates that DP is well-suited for solving this problem. Let's say that $dp[i][j]$ tells us if the substring $s[i : j]$ is a palindrome. Based on our earlier observations, we can say that:

$| dp[i][j] = \text{True if } s[i] == s[j] \text{ and } dp[i + 1][j - 1]$

Naturally, we need to specify the base cases for this formula.

Base cases

We mentioned earlier that substrings of length 1 and 2 have no inner substrings. In other words, there are no further subproblems to solve when determining if they are palindromes. As such, these substrings define our base cases:

- Base case: All substrings of length 1 are palindromes. So, set $dp[i][i]$ to true for all values of i .

a b c c b a b a → mark all length 1 substrings as palindromes

- Base case: Substrings of length 2 are palindromes if both its characters are the same. So, set $dp[i][i + 1]$ to true if $s[i] == s[i + 1]$.

a b c c b a b a → mark all length 2 substrings where both characters are the same as palindromes

With the base cases set up, let's discuss how to populate the rest of the DP table.

Populating the DP table

Determining if longer substrings are palindromes depends on the DP values of shorter substrings. Therefore, we should populate the DP table for the shortest substrings first, starting with checking all substrings of length 3 and working our way up to length n , where n denotes the length of the input string.

Keeping track of the longest palindromic substring

As we populate the DP table, we also need to keep track of the longest palindromic substring encountered. We use two variables for this: `start_index` and `max_len`.

- `start_index`: stores the starting index of the longest palindromic substring found so far.
- `max_len`: stores the length of the longest palindromic substring found so far.

When we find a new, longer palindromic substring, we update these two variables. By the end, `start_index` and `max_len` will indicate the position and length of the longest palindrome.

Finally, return the substring using `s[start_index : start_index + max_len]`.

This solution is a classic example of "interval DP," which is used to solve optimization problems involving subproblems over intervals of data. In this case, the "intervals" are effectively the substrings between indexes i and j .

Implementation

```
def longest_palindrome_in_a_string(s: str) -> str:  
    n = len(s)  
    if n == 0:  
        return ""  
    dp = [[False] * n for _ in range(n)]  
    max_len = 1  
    start_index = 0  
    # Base case: a single character is always a palindrome.  
    for i in range(n):  
        dp[i][i] = True  
    # Base case: a substring of length two is a palindrome if both  
    # characters are the same.
```

```

for i in range(n - 1):
    if s[i] == s[i + 1]:
        dp[i][i + 1] = True
        max_len = 2
        start_index = i
# Find palindromic substrings of length 3 or greater.
for substring_len in range(3, n + 1):
    # Iterate through each substring of length 'substring_len'.
    for i in range(n - substring_len + 1):
        j = i + substring_len - 1
        # If the first and last characters are the same, and the
        # inner substring is a palindrome, then the current
        # substring is a palindrome.
        if s[i] == s[j] and dp[i + 1][j - 1]:
            dp[i][j] = True
            max_len = substring_len
            start_index = i
return s[start_index : start_index + max_len]

```

Complexity Analysis

Time complexity: The time complexity of `longest_palindrome_in_a_string` is $O(n^2)$ because each cell of the $n \times n$ DP table is populated once.

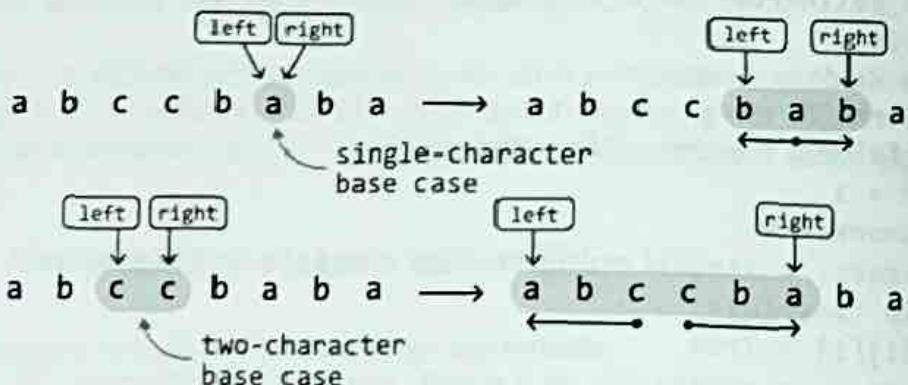
Space complexity: The space complexity is $O(n^2)$ because we're maintaining a DP table that has n^2 elements. Note, the output string is not considered in the space complexity.

Optimized Approach

An important observation of the previous approach is that the base cases represent the centers of palindromes. Understanding this, another possible strategy is to expand outward from each base case to find the longest palindrome.

There are two types of base cases: single-character substrings and two-character substrings. We can treat each as the center of potential palindromes, and expand outward from each of them to find these palindromes.

We can do this by setting left and right pointers at the center, expanding them outward – as long as the characters at both pointers match – and stopping once they can no longer form a larger palindrome. Here are two examples of this:



All we need to do is keep track of the start index and length of the longest palindromic we find, just as in the previous approach.

This approach makes use of some of the information from the previous DP approach, while solving the problem using constant space.

Implementation – Optimized Approach

```
def longest_palindrome_in_a_string_expanding(s: str) -> str:
    n = len(s)
    start, max_len = 0, 0
    for center in range(n):
        # Check for odd-length palindromes.
        odd_start, odd_length = expand_palindrome(center, center, s)
        if odd_length > max_len:
            start = odd_start
            max_len = odd_length
        # Check for even-length palindromes.
        if center < n - 1 and s[center] == s[center + 1]:
            even_start, even_length = expand_palindrome(
                center, center + 1, s
            )
            if even_length > max_len:
                start = even_start
                max_len = even_length
    return s[start : start + max_len]

# Expands outward from the center of a base case to identify the start
# index and length of the longest palindrome that extends from this
# base case.
def expand_palindrome(left: int, right: int, s: str) -> Tuple[int, int]:
    while (left > 0 and right < len(s) - 1 and
           s[left - 1] == s[right + 1]):
        left -= 1
        right += 1
    return left, right - left + 1
```

Complexity Analysis

Time complexity: The time complexity of `longest_palindrome_in_a_string_expanding` is $O(n^2)$ because expanding from the center of a base case takes up to $O(n)$ time. Doing this for each base case takes $O(n^2)$ time.

Space complexity: The space complexity is $O(1)$ since we aren't maintaining any auxiliary data structures. The output string is not considered in the space complexity.

Manacher's Algorithm

Aside from the above quadratic-time solutions, there's a more efficient algorithm for finding the longest palindromic substring: Manacher's Algorithm. This algorithm runs in $O(n)$ time.

However, Manacher's Algorithm's specialized nature makes it less common in coding interviews.

Most interviewers want to see solutions that show you deeply understand basic concepts and have strong problem-solving skills. They are less interested in tricky solutions that would be unlikely for candidates to come up with during interviews.

For those interested in learning more about Manacher's Algorithm, check out the following resources [1, 2].

References

- [1] GeeksforGeeks: Manacher's Algorithm – Linear Time Longest Palindromic Substring: <https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/>
- [2] Wikipedia: Manacher's Algorithm: https://en.wikipedia.org/wiki/Longest_palindromic_substring#Manacher's_algorithm

Maximum Subarray Sum

Given an array of integers, return the sum of the subarray with the largest sum.

Example:

Input: `nums = [3, 1, -6, 2, -1, 4, -9]`

Output: 5

Explanation: subarray [2, -1, 4] has the largest sum of 5.

Constraints:

- The input array contains at least one element.

Intuition

Brute force approaches to this problem involve calculating the sum of every possible subarray. This would take at least $O(n^2)$ time, where n denotes the length of the array. So, let's consider alternative methods.

The challenge with this problem lies in the presence of negative numbers. If the array consisted entirely of non-negative numbers, the answer would simply be the sum of the entire array.

To find the maximum sum given the presence of negative numbers, let's try keeping track of the sum of a contiguous subarray, starting at index 0.

As this subarray expands and we add each number to the running sum, we'll need to decide whether to continue with the current subarray's sum, or start a new subarray beginning with the current element. To understand how we might make such a decision, let's dive into an example.

Consider the following input array:

[3	1	-6	2	-1	4	-9]
	0	1	2	3	4	5	6	

The first two values of the array are positive, so we can continue expanding the current subarray by adding these to our sum (`curr_sum`), initialized at 0:

num
↓
[3 1 -6 2 -1 4 -9] curr_sum += 3
0 1 2 3 4 5 6 = 3

num
↓
[3 1 -6 2 -1 4 -9] curr_sum += 1
0 1 2 3 4 5 6 = 4

When we reach index 2, we land on the first negative number (-6). Adding it to the current sum gives us a negative sum of -2.

What should we do now? If we restart the subarray at this point, the new subarray will start with a sum of -6, which is less than the current sum of -2.

num							
[3	1	-6	2	-1	4	-9]
0	1	2	3	4	5	6	

$\text{curr_sum} += -6$
 $= -2 \leftarrow \text{larger}$

num							
[3	1	-6	2	-1	4	-9]
0	1	2	3	4	5	6	

$\text{curr_sum} = -6 \leftarrow \text{smaller}$

Therefore, it is better to continue with the current subarray for now.

At index 3, we reach another important decision point:

- If we include 2 in the current subarray, its sum increases to 0:

num							
[3	1	-6	2	-1	4	-9]
0	1	2	3	4	5	6	

$\text{curr_sum} += 2$
 $= 0 \leftarrow \text{smaller}$

- If we restart the subarray at this index, we begin a new subarray of sum 2:

num							
[3	1	-6	2	-1	4	-9]
0	1	2	3	4	5	6	

$\text{curr_sum} = 2 \leftarrow \text{larger}$

It's evident here that the better choice is to start tracking the sum of a new subarray, beginning at index 3.

As observed, for each number in the array during this process, there are two choices:

- Continue: Add the current number to the ongoing subarray sum ($\text{curr_sum} + \text{num}$).
- Restart: Begin keeping track of a new subarray starting with the current number, with an initial sum of num .

The best choice is the larger of the two: $\max(\text{curr_sum} + \text{num}, \text{num})$.

Let's apply this logic to the rest of the array:

num							(previous curr_sum value)
[3	1	-6	2	-1	4	-9]
0	1	2	3	4	5	6	

$\text{curr_sum} = \max(\text{curr_sum} + \text{num}, \text{num})$
 $= \max(2 + (-1), -1)$
 $= 1$

num							
[3	1	-6	2	-1	4	-9]
0	1	2	3	4	5	6	

$\text{curr_sum} = \max(\text{curr_sum} + \text{num}, \text{num})$
 $= \max(1 + 4, 4)$
 $= 5$

\downarrow num $[\begin{matrix} 3 & 1 & -6 & 2 & -1 & 4 & -9 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{matrix}]$	$\begin{aligned} curr_sum &= \max(curr_sum + num, num) \\ &= \max(5 + (-9), -9) \\ &= -4 \end{aligned}$
---	---

Now that we have a strategy to linearly track subarray sums, the only other thing to do is keep track of the largest value of `curr_sum` encountered. To do this, we can update a variable `max_sum` whenever we encounter a larger `curr_sum` value. Then, `max_sum` will be the answer to the problem.

Kadane's algorithm

The algorithm described above is formally known as "Kadane's algorithm". Although it may not seem like it, Kadane's algorithm is actually a DP algorithm. What's interesting is that we didn't explicitly detect and solve subproblems to come up with this algorithm, like we typically do in DP. Instead, we solved it by linearly keeping track of a subarray sum and making decisions along the way.

So, to fully understand why this is a DP problem, let's explore how we would solve it using the traditional DP approach of breaking the problem into smaller subproblems, and solving them step-by-step. This is demonstrated in the next section of this explanation.

Implementation

```
def maximum_subarray_sum(nums: List[int]) -> int:
    if not nums:
        return 0
    # Set the sum variables to negative infinity to ensure negative
    # sums can be considered.
    max_sum = current_sum = float('-inf')
    # Iterate through the array to find the maximum subarray sum.
    for num in nums:
        # Either add the current number to the existing running sum, or
        # start a new subarray at the current number.
        current_sum = max(current_sum + num, num)
        max_sum = max(max_sum, current_sum)
    return max_sum
```

Complexity Analysis

Time complexity: The time complexity of `maximum_subarray_sum` is $O(n)$ because we iterate through each element of the input array once.

Space complexity: The space complexity is $O(1)$.

Intuition – DP

Let's discuss how we would approach this problem as we did with other DP problems in this chapter.

An important observation is that every possible subarray ends at a certain index. This inversely means each index signifies the end of several subarrays, one of which will have the maximum subarray sum (shortened to "max subarray" moving forward) ending at that index.

For example, we can see the max subarray that ends at index 3 below by considering all subarrays ending at index 3:

[3 1 -6 2 -1 4 -9]	sum = 0
[3 1 -6 2 -1 4 -9]	sum = -3
[3 1 -6 2 -1 4 -9]	sum = -4
[3 1 -6 2 -1 4 -9]	max sum = 2

So, how can we find the max subarray that ends at each index? Consider the last index of the array:

[3 1 -6 2 -1 4 -9]	max_subarray(i) = ?
--	---------------------

One thing we know for sure is the max subarray ending at this index will definitely include the value at this index. We just need to determine if there are any elements to the left that also contribute to this max subarray. In other words, we need to find the max subarray that ends right before the last index:

[3 1 -6 2 -1 4 -9]	max_subarray(i - 1) + (-9)
--	----------------------------

Another thing to consider is the possibility that $\text{max_subarray}(i - 1)$ is negative. This would mean the max subarray should only consist of -9, as a further negative contribution will only decrease the sum. Therefore, the formula becomes:

$$\boxed{\text{max_subarray}(i) = \max(\text{max_subarray}(i - 1) + \text{nums}[i], \text{nums}[i])}$$

As we see, this is a recurrence relation that takes advantage of an optimal substructure, where the max subarray at the current index depends on the max subarray at the previous index.

This indicates we can solve this problem using DP. Translating the above recurrence relation to a DP formula gives us:

$$\boxed{dp[i] = \max(dp[i - 1] + \text{nums}[i], \text{nums}[i])}$$

Now, let's consider what the base case for this problem is.

Base case

The simplest subproblem occurs when we consider only the first element of the array (i.e., when i

$= 0$). When there's only one element, there's only one subarray. Therefore, we can set $dp[0]$ to $nums[0]$ as our base case.

Populating the DP array

With the base case established, we can populate the rest of the DP array. Starting from index 1 and going up to index $n - 1$, we calculate the maximum subarray sum ending at each index using the aforementioned recurrence relation.

As we populate the DP array, we need to keep track of the maximum value in the DP array, max_sum , representing the largest sum of any subarray within the entire array. By the time we finish populating the DP array, we can just return max_sum .

Implementation – DP

```
def maximum_subarray_sum_dp(nums: List[int]) -> int:
    n = len(nums)
    if n == 0:
        return 0
    dp = [0] * n
    # Base case: the maximum subarray sum of an array with just one
    # element is that element.
    dp[0] = nums[0]
    max_sum = dp[0]
    # Populate the rest of the DP array.
    for i in range(1, n):
        # Determine the maximum subarray sum ending at the current
        # index.
        dp[i] = max(dp[i - 1] + nums[i], nums[i])
        max_sum = max(max_sum, dp[i])
    return max_sum
```

Complexity Analysis

Time complexity: The time complexity of `maximum_subarray_sum_dp` is $O(n)$ since we iterate through n elements of the DP array.

Space complexity: The space complexity is $O(n)$ because we're maintaining a DP array that contains n elements.

Optimization

An important thing to note is that in the DP solution, we only ever need to access the previous value of the DP array (at $i - 1$) to calculate the current value (at i). This means we don't need to store the entire DP array.

Instead, we can use a single variable to keep track of the current subarray sum and update this value to calculate the next subarray sum.

This approach reduces the space complexity to $O(1)$. The adjusted implementation of this can be seen below:

```
def maximum_subarray_sum_dp_optimized(nums: List[int]) -> int:
```

```
n = len(nums)
if n == 0:
    return 0
current_sum = nums[0]
max_sum = nums[0]
for i in range(1, n):
    current_sum = max(nums[i], current_sum + nums[i])
    max_sum = max(max_sum, current_sum)
return max_sum
```

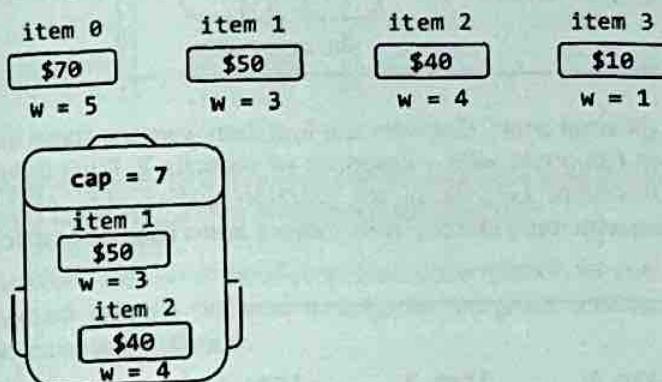
As we can see, we've ended up with an optimized DP solution that's nearly identical to the solution we came up with in the first approach: Kadane's algorithm.

0/1 Knapsack

You are a thief planning to rob a store. However, you can only carry a knapsack with a maximum capacity of cap units. Each item (i) in the store has a weight ($\text{weights}[i]$) and a value ($\text{values}[i]$).

Find the maximum total value of items you can carry in your knapsack.

Example:



Input: $\text{cap} = 7$, $\text{weights} = [5, 3, 4, 1]$, $\text{values} = [70, 50, 40, 10]$

Output: 90

Explanation: The most valuable combination of items that can fit in the knapsack together are items 1 and 2. These items have a combined value of $50 + 40 = 90$ and a total weight of $3 + 4 = 7$, which fits within the knapsack's capacity.

Intuition

For each item, we have two choices: include it in the knapsack, or exclude it. This binary decision is why this classic problem is called "0/1 Knapsack."

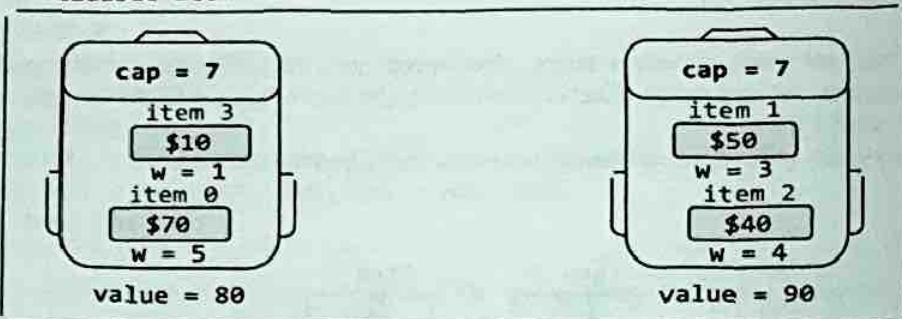
The brute force approach involves making this decision for every item. Since two choices can be made for each item, this results in $2 \cdot 2 \cdot 2 \cdots \cdot 2 = 2^n$ possible combinations of choices. As we can see, generating all possible combinations is inefficient.

A greedy solution that involves picking the most valuable items first isn't a good choice either, as it doesn't always lead to the optimal outcome, which we can see in the example below:

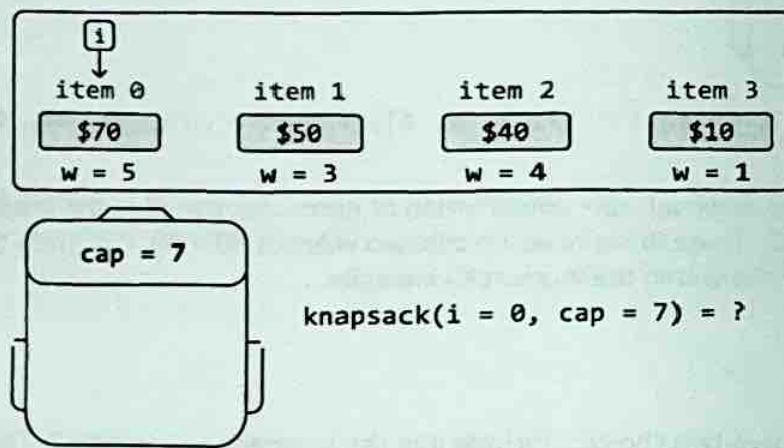
item 0	item 1	item 2	item 3
\$70	\$50	\$40	\$10
w = 5	w = 3	w = 4	w = 1

option 1: take the most valuable item first

option 2: optimal solution



So, let's approach this problem from a different angle. Consider the first item from the above item list ($i = 0$). What's the most value we can attain with a knapsack of capacity 7, if we include this item? What about if we exclude this item? Let's define the function $\text{knapsack}(i, cap)$ to represent the maximum value achievable with items starting from index i and a knapsack capacity of cap :

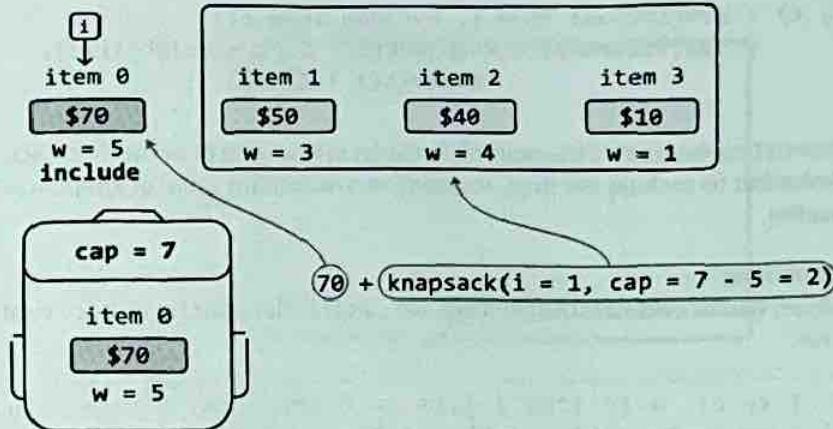


We explore the implications of including or excluding this item, separately.

Including item i

Picking the first item gives a value of 70. This item weighs 5, so our knapsack now has a remaining capacity of $7 - 5 = 2$. With an updated capacity of 2, what's the optimal combination possible with the remaining items in our selection?

This question leads us to realize that if we determine the maximum value that can be obtained from the remaining items (starting from index 1) with a knapsack capacity of 2, we can find the solution:



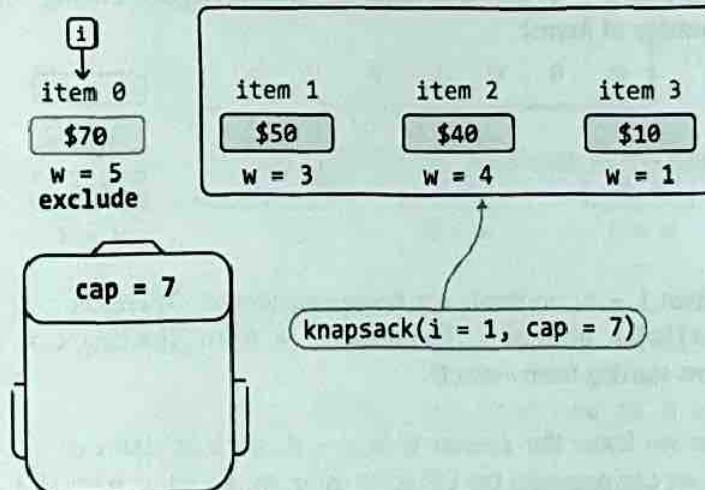
We've identified that this case can be solved by solving a subproblem, which means we're dealing with a problem that has an optimal substructure.

Therefore, we can generalize a recurrence relation for this case. Below, c denotes the remaining knapsack capacity we want to solve for (we give it a different name because it can be different to the initial value of cap):

If we include item i, the most value we can get is
 $\text{values}[i] + \text{knapsack}(i + 1, c - \text{weights}[i])$

Excluding item i

Now, let's say we exclude item 0. This means our knapsack will maintain a capacity of 7. Here, the most value we can get is just from the maximum value from the rest of the items, with a knapsack of capacity 7:



Again, this case can be solved by solving a subproblem:

If we exclude item i, the most value we can get is $\text{knapsack}(i + 1, c)$.

Now that we've established the recurrence relations for both cases (including and excluding item i), we can combine them: the maximum value we can get from any selection of items is the larger value obtained from these two cases:

```

knapsack(i, c) = max(include item i, exclude item i)
                = max(values[i] + knapsack(i + 1, c - weights[i]),
                      knapsack(i + 1, c))

```

One case we haven't covered yet is the possibility the item does not fit in the knapsack. In this case, we have no choice but to exclude the item, resulting in a maximum value of $\text{knapsack}(i + 1, c)$, as discussed earlier.

Dynamic programming

Given this problem has an **optimal substructure**, we can translate our recurrence relations directly into DP formulas:

```

if weights[i] <= c: # If item i fits in a knapsack of capacity c.
    dp[i][c] = max(values[i] + dp[i + 1][c - weights[i]],
                    dp[i + 1][c])
else: # If item i doesn't fit.
    dp[i][c] = dp[i + 1][c]

```

For clarity, there are two dimensions in our DP table:

- One for the current item, i , represented by the rows of the DP table.
- One for the current knapsack capacity, c , represented by the columns on the DP table.

With this in mind, let's think about what the base cases should be.

Base cases

The simplest version of this problem is when there are no items in our selection, meaning the maximum value we can attain is 0. But which cells of the DP table represent this base case?

We know that when $i = n - 1$, only one item from the selection is being considered (where n denotes the total number of items):

$i = n - 1$	item 0	item 1	item 2	item 3
	\$70	\$50	\$40	\$10
	$w = 5$	$w = 3$	$w = 4$	$w = 1$

This implies that when $i = n$, no items are being considered. Therefore, we can populate the DP table using $dp[n][c] = 0$ for all c . The reason $i = 0$ isn't the base case is because $i = 0$ encapsulates all items starting from index 0.

Another subproblem we know the answer to is $c = 0$, since no items can fit in a knapsack of capacity 0. For this, we can populate the DP table using $dp[i][0] = 0$ for all i .

Let's draw the DP table with just the base case values, to get a better idea of what this looks like:

	0	1	2	3	4	5	6	7
0	0							
1	0							
2	0							
3	0							
4	0	0	0	0	0	0	0	0

As shown, the first column and last row are set to 0 for the base cases.

Populating the DP table

We populate the DP table starting from the smallest subproblems (excluding base cases). Specifically, this means starting from row $i = n - 1$, where only the last item is considered, and ending at $i = 0$, where we consider all items. For each of these rows, we iterate through each possible knapsack capacity from $c = 1$ to $c = \text{cap}$.

	0	1	2	3	4	5	6	7
0	0							end
1	0							→
2	0							→
3	0	start						→
4	0	0	0	0	0	0	0	0

Once the DP table is populated, we return $\text{dp}[0][\text{cap}]$, which stores the maximum value after all items and knapsack capacities are considered.

Implementation

```
def knapsack(cap: int, weights: List[int], values: List[int]) -> int:
    n = len(values)
    # Base case: Set the first column and last row to 0 by
    # initializing the entire DP table to 0.
    dp = [[0 for x in range(cap + 1)] for x in range(n + 1)]
    # Populate the DP table.
    for i in range(n - 1, -1, -1):
        for c in range(1, cap + 1):
            # If the item 'i' fits in the current knapsack capacity,
            # the maximum value at 'dp[i][c]' is the largest of either:
            # 1. The maximum value if we include item 'i'.
            # 2. The maximum value if we exclude item 'i'.
            if weights[i] <= c:

```

```
        dp[i][c] = max(values[i] + dp[i + 1][c - weights[i]],
                         dp[i + 1][c])
    # If it doesn't fit, we have to exclude it.
    else:
        dp[i][c] = dp[i + 1][c]
return dp[0][cap]
```

Complexity Analysis

Time complexity: The time complexity of knapsack is $O(n \cdot \text{cap})$ because each cell of the DP table is populated once.

Space complexity: The space complexity is $O(n \cdot \text{cap})$ because we maintain a DP table that stores $(n+1) \cdot (\text{cap}+1)$ elements.

Optimization

We can optimize the solution by recognizing that, for each cell in the DP table, we only need access cells from the row below it.

Therefore, we only need to maintain two rows:

- `curr_row`: the current row being populated.
- `prev_row`: the row below the current row.

This effectively reduces the space complexity to $O(\text{cap})$. Below is the optimized code:

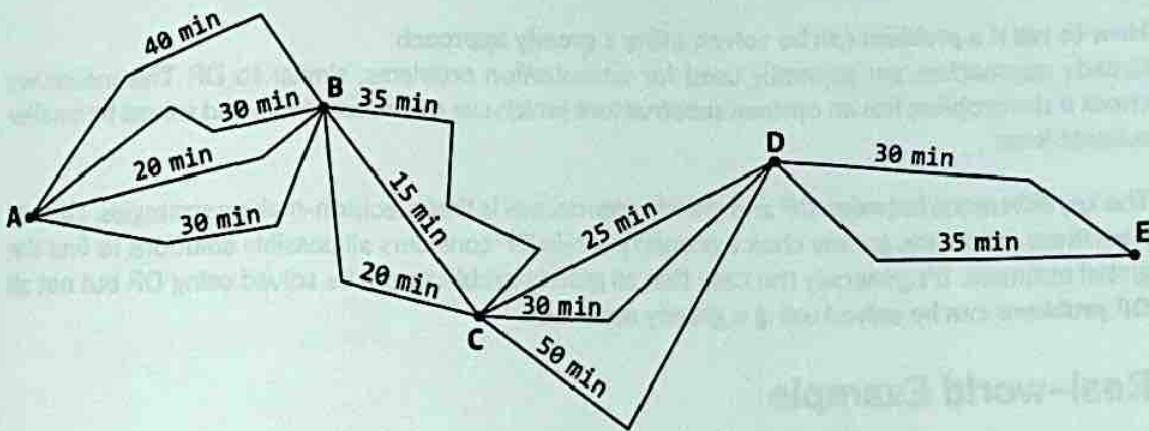
```
def knapsack_optimized(cap: int, weights: List[int], values: List[int]) -> int:
    n = len(values)
    # Initialize 'prev_row' as the DP values of the row below the
    # current row.
    prev_row = [0] * (cap + 1)
    for i in range(n - 1, -1, -1):
        # Set the first cell of the 'curr_row' to 0 to set the base
        # case for this row. This is done by initializing the entire
        # row to 0.
        curr_row = [0] * (cap + 1)
        for c in range(1, cap + 1):
            # If item 'i' fits in the current knapsack capacity, the
            # maximum value at 'curr_row[c]' is the largest of either:
            # 1. The maximum value if we include item 'i'.
            # 2. The maximum value if we exclude item 'i'.
            if weights[i] <= c:
                curr_row[c] = max(values[i] + prev_row[c - weights[i]],
                                   prev_row[c])
            # If item 'i' doesn't fit, we exclude it.
            else:
                curr_row[c] = prev_row[c]
        # Set 'prev_row' to 'curr_row' values for the next iteration.
        prev_row = curr_row
    return prev_row[cap]
```

Greedy

Introduction to Greedy Algorithms

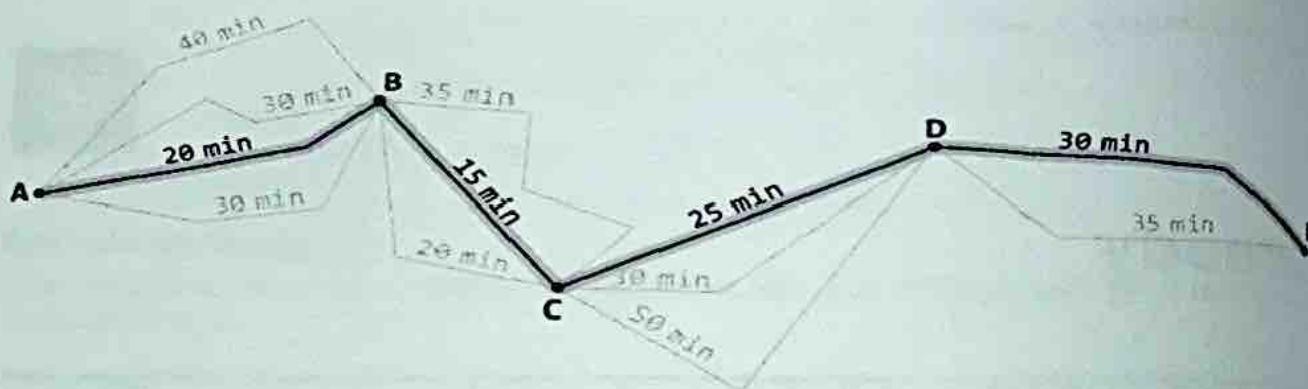
Greedy algorithms are a class of algorithms that make a series of decisions, where each decision is the best immediate choice given the options available. To understand this, let's dive into an analogy.

Imagine you're planning a road trip from city A to city E, and you want to visit cities B, C, and D along the way:



You want the fastest route for this journey, so you aim to optimize the route. One option is to check every possible route to see which one takes the least amount of time to drive through, but this approach is quite time consuming.

Instead, you decide to take the route with the shortest duration at each leg of the trip, understanding that it will result in the quickest journey:



This is effectively a greedy approach to the problem, where you choose the best option at each step, aiming to find the best solution overall.

How does a greedy algorithm work?

More formally, a greedy algorithm follows the **greedy choice property**, which states that the best overall solution to a problem (global optimum) can be arrived at by making the best possible decision at each step (local optimum).

Each decision is made based only on the current context, and ignores its impact on future steps. This process continues until the algorithm reaches a final solution.

How to tell if a problem can be solved using a greedy approach

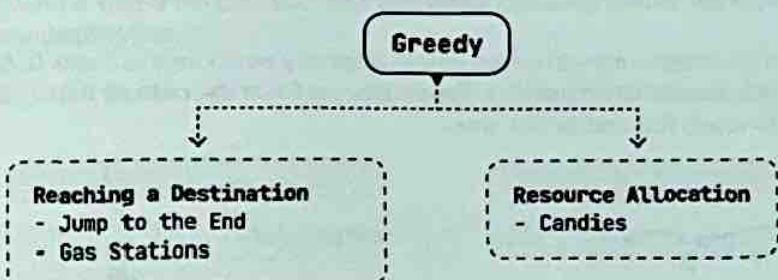
Greedy approaches are generally used for optimization problems, similar to DP. This means we check if the problem has an **optimal substructure** which can be broken down and solved by smaller subproblems.

The key difference between DP and greedy approaches is their decision-making strategies. Greedy algorithms follow the **greedy choice property**, while DP considers all possible solutions to find the global optimum. It's generally the case that all greedy problems can be solved using DP, but not all DP problems can be solved using a greedy approach.

Real-world Example

Huffman coding in data compression: Huffman coding is an algorithm that assigns variable-length codes to input characters based on their frequencies, with the most frequent characters getting the shortest codes. The goal is to minimize the overall size of the encoded data. The greedy approach works by always combining the two least frequent characters first (local optimum), ensuring that each step reduces the size of the overall encoding (global optimum). This method is widely used in file compression formats like ZIP, and media compression standards like JPEG.

Chapter Outline



It's impractical to provide a one-size-fits-all framework for solving greedy problems because each one is unique. Instead, this chapter explores a variety of unique situations in which a problem can be solved using the greedy choice property to provide a general understanding of how this property works.

Jump to the End

You are given an integer array in which you're originally positioned at index 0. Each number in the array represents the maximum jump distance from the current index. Determine if it's possible to reach the end of the array.

Example 1:



Input: `nums = [3, 2, 0, 2, 5]`

Output: True

Example 2:



Input: `nums = [2, 1, 0, 3]`

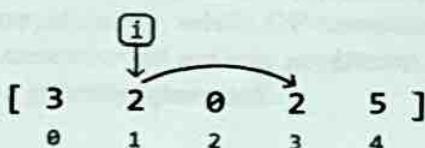
Output: False

Constraints:

- There is at least one element in `nums`.
- All integers in `nums` are non-negative integers.

Intuition

From any index i in the array, we can jump up to $\text{nums}[i]$ positions to the right. This means the furthest index we can reach from any given index i , is $i + \text{nums}[i]$:



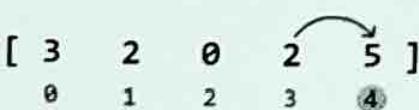
Furthest index we can jump to from $i = i + \text{nums}[i]$

If the array consisted entirely of positive numbers, jumping from index 0 to the last index would be straightforward, as there would always be a way to progress forward toward the last index. The challenge arises when we encounter a 0 in the array, as a 0 is effectively a dead end, since landing on it disallows any further movement.

Consider the example below:

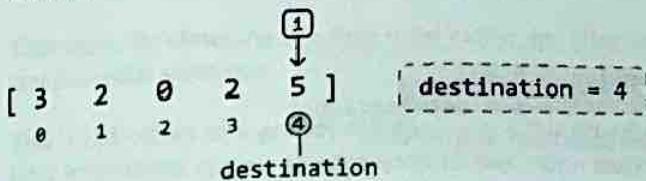


Let's think about this problem backward. Our destination is the last index, index 4, but let's say we've already made it there. How did we reach this index? In this example, it's possible to make it to index 4 from index 3:



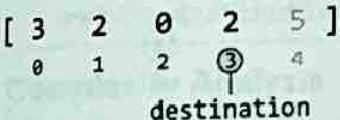
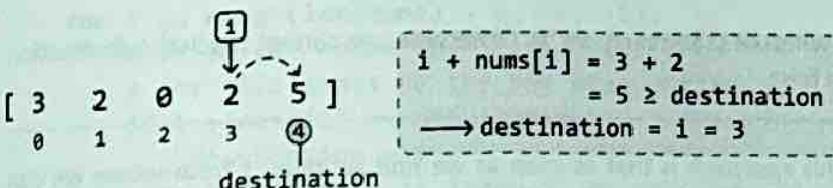
This means that if we find a way to reach index 3, we know for sure we can make it to index 4. The key observation here is that if we can reach the last index from any earlier index, this earlier index becomes our new destination.

With this in mind, let's go through this example in full, starting with the last index as our initial destination:



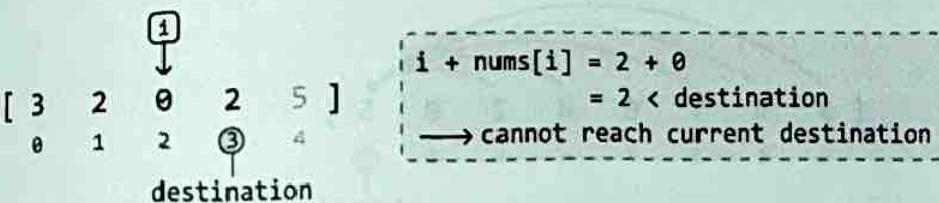
To find earlier indexes that can reach the destination, let's move backward through the array, starting at index 3. As we do this for each index, we check if we can reach the current destination from this index. If we can, this index becomes the new destination. We do this by checking if it's possible to jump to the destination from this index:

| if $i + \text{nums}[i] \geq \text{destination}$, we can jump to destination from index i .

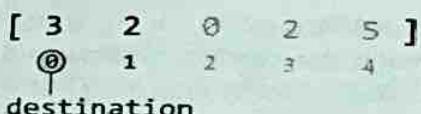
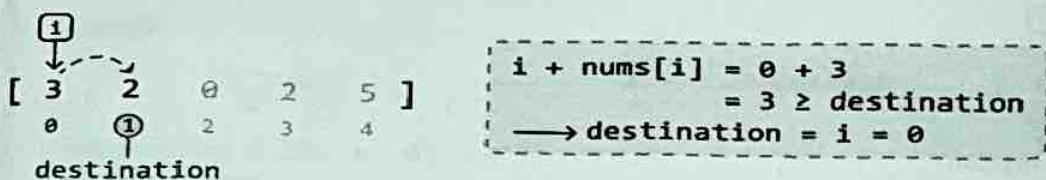
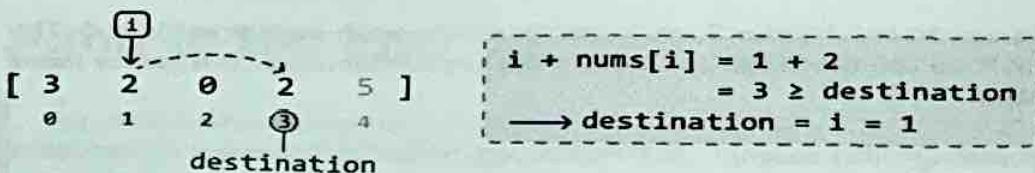


With the destination at index 3, let's continue moving backward through the array.

Now, we're at index 2. Below, we see we cannot reach the destination from index 2, so the destination is not updated.



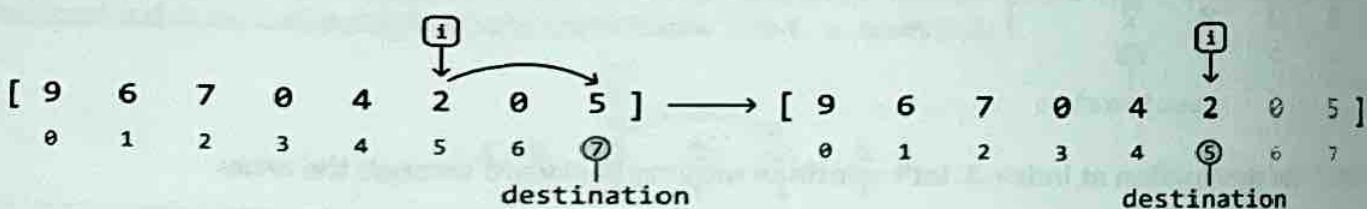
Continue with this logic for the remaining numbers in the array:



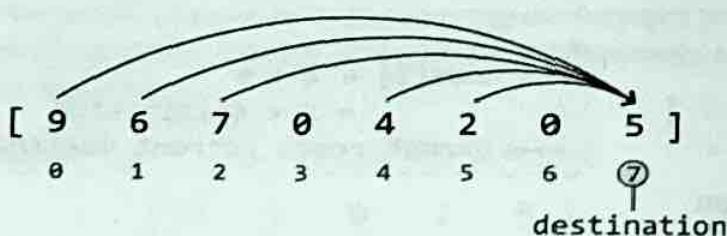
Finally, we see that once we've finished iterating through each index, the destination is set to index 0. This means we've successfully found a way to jump to the end from index 0.

Therefore, we return true when `destination == 0`. Otherwise, we cannot reach the destination from index 0, so we return false.

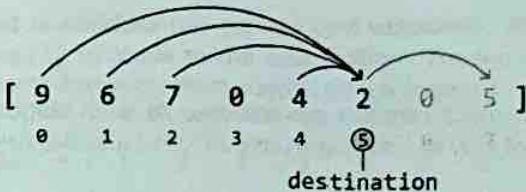
An interesting aspect of this approach is that as soon as we find an index *i* from where we can reach the destination, we update the destination to that index and assume that this is the correct decision:



The thing is, there can sometimes be multiple indexes which can reach the destination. So, how do we know that choosing the first valid index we encounter from the right is the best choice?



The key to understanding why is realizing that all the other indexes which can reach the destination can also reach this first valid index:



Therefore, by choosing the first valid index, we effectively simplify our problem without missing any potential solutions.

This is indicative of a **greedy solution**; since the greedy choice property is satisfied, we make the best immediate choice at each step as we move backward through the array (local optimum), hoping it leads to the overall solution (global optimum).

Implementation

```
def jump_to_the_end(nums: List[int]) -> bool:
    # Set the initial destination to the last index in the array.
    destination = len(nums) - 1
    # Traverse the array in reverse to see if the destination can be
    # reached by earlier indexes.
    for i in range(len(nums) - 1, -1, -1):
        # If we can reach the destination from the current index,
        # set this index as the new destination.
        if i + nums[i] >= destination:
            destination = i
    # If the destination is index 0, we can jump to the end from index
    # 0.
    return destination == 0
```

Complexity Analysis

Time complexity: The time complexity of `jump_to_the_end` is $O(n)$, where n denotes the length of the array. This is because we iterate through each element of `nums` in reverse order.

Space complexity: The space complexity is $O(1)$.

Gas Stations

There's a circular route which contains gas stations. At each station, you can fill your car with a certain amount of gas, and moving from that station to the next one consumes some fuel.

Find the index of the gas station you would need to start at, in order to complete the circuit without running out of gas. Assume your car starts with an empty tank. If it's not possible to complete the circuit, return -1. If it's possible, assume only one solution exists.

Example:

Input: `gas = [2, 5, 1, 3], cost = [3, 2, 1, 4]`
Output: 1

Explanation:

Start at station 1: gain 5 gas (tank = 5), costs 2 gas to go to station 2 (tank = 3).

At station 2: gain 1 gas (tank = 4), costs 1 gas to go to station 3 (tank = 3).

At station 3: gain 3 gas (tank = 6), costs 4 gas to go to station 3 (tank = 2).

At station 0: gain 2 gas (tank = 4), costs 3 gas to go to station 1 (tank = 1).

We started and finished the circuit at station 1 without running out of gas.

Intuition

Before deciding which gas station to start with, let's first determine if it's even possible to complete the circuit with the total amount of gas available.

Total gas vs total cost

Case 1: $\text{sum(gas)} < \text{sum(cost)}$:

The first thing to realize is if the total gas is less than the total cost, it's impossible to complete the circuit. No matter where we start, we'll run out of gas before completing the circuit. So, in this situation, we should return -1.

Case 2: $\text{sum(gas)} \geq \text{sum(cost)}$:

Now, let's consider the more interesting case where the total travel cost is less than or equal to the total amount of gas available.

Here's a potential hypothesis:

Since there's enough total gas to cover the total cost of travel, there must be a start point in the circuit that allows us to complete it without ever running out of gas.

It's tough to confirm this hypothesis without examining an example, so let's dive into one.

Finding a start point

Consider the following example where $\text{sum(gas)} > \text{sum(cost)}$:

<code>gas = [</code>	3	2	1	3	3	2	3	4	<code>]</code>
<code>cost = [</code>	2	1	4	1	2	6	0	3	<code>]</code>

We don't necessarily need to consider the gas and cost separately. At any station i , we collect $\text{gas}[i]$ and consume $\text{cost}[i]$ to move to the next station. We can consider both at the same time by getting the difference between these values, which provides the net gas gained or lost at each station:

$$\begin{array}{cccccccccc} \text{gas} & = & [& 3 & 2 & 1 & 3 & 3 & 2 & 3 & 4] \\ \text{cost} & = & [& 2 & 1 & 4 & 1 & 2 & 6 & 0 & 3] \\ & & \downarrow \\ \text{net gas:} & & 1 & 1 & -3 & 2 & 1 & -4 & 3 & 1 \end{array}$$

Let's start at station 0 with an empty gas tank and see how far we can go. The net gas at this station is positive (1), which means we have enough gas to reach the next station. Let's add 1 to our tank:

i	↓	1	1	-3	2	1	-4	3	1	tank += 1
		0	1	2	3	4	5	6	7	= 1
start										

Note that index i refers to the current gas station, whereas start refers to the gas station we started from.

At station 1, we encounter the same situation:

i	↓	1	1	-3	2	1	-4	3	1	tank += 1
		0	1	2	3	4	5	6	7	= 2
start										

At station 2, our tank falls below 0, indicating we don't have enough gas to make it to the next station:

i	↓	1	1	-3	2	1	-4	3	1	tank += (-3)
		0	1	2	3	4	5	6	7	= -1 < 0
start										→ cannot make it to the next station

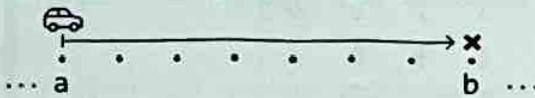
This means we cannot start our journey at station 0. Should we go back and try station 1? The key observation here is that if we didn't have enough gas to get from station 0 to station 3, we also wouldn't have enough if we started at any other station before station 3:

1	1	-3	2	1	-4	3	1	tank = 0
0	1	2	3	4	5	6	7	

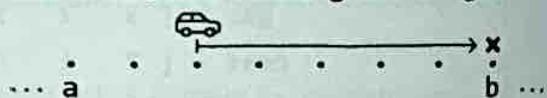
This is a general rule:

If we cannot make it to station b from station a, we cannot make it to station b from any of the stations in between, either:

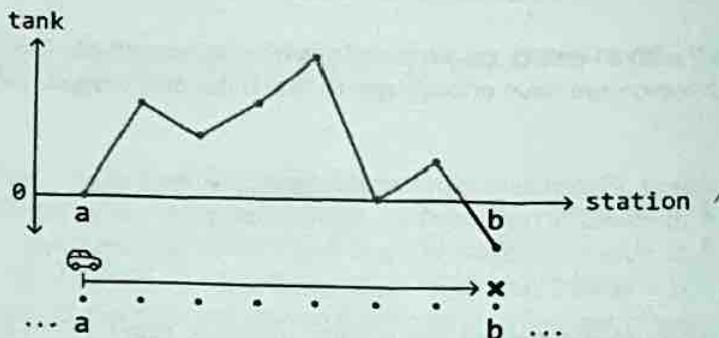
if we cannot drive from a to b without running out of gas,



we cannot start anywhere between a and b without running out of gas.



Let's try to understand why. If we only just ran out of gas right before reaching station b, this means our tank maintained a non-negative amount of gas until station b:



Consequently, starting anywhere else before station b will result in us missing a non-negative amount of gas from the previous stations. Therefore, starting at any of these in-between stations doesn't allow us to progress to station b.

Back to our example. Let's now try resetting our tank to 0 and restarting at station 3 (at $i + 1$), since we just discussed how starting at stations 0 to i doesn't work:

		i	
1	1	-3	2 1 -4 3 1
0	1	2	3 4 5 6 7

start

tank = 0

We continue until we reach a point where we cannot proceed to the next station:

		i	
1	1	-3	2 1 -4 3 1
0	1	2	3 4 5 6 7

start

tank += 2
= 2

		i	
1	1	-3	2 1 -4 3 1
0	1	2	3 4 5 6 7

start

tank += 1
= 3

				1				
1	1	-3	2	1	-4	3	1	tank += (-4)
0	1	2	3	4	5	6	7	= -1 < 0 → cannot make it to the next station

As we can see, we ran out of gas at station 5, which means we can't start from stations 3 to 5 either. So, let's try restarting at station 6.

After resetting the tank to 0, let's continue traveling through the stations:

				1				
1	1	-3	2	1	-4	3	1	tank += 3
0	1	2	3	4	5	6	7	= 3

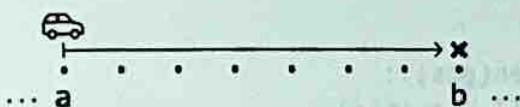
				1				
1	1	-3	2	1	-4	3	1	tank += 1
0	1	2	3	4	5	6	7	= 4

We've reached the end of the array. Should we go back to the start of the array to check if starting from station 6 allows us to complete the circuit? Or is reaching the end from station 6 enough to finish the circuit? Let's look into this.

Proving we have enough gas to complete the circuit after reaching the end of the array

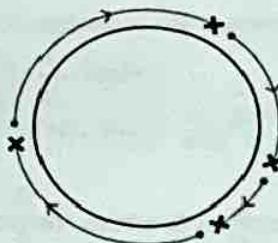
We can determine this via a proof by contradiction. If the gas we have by the end of the array is not enough, that means no solution exists: no station at which we could start in order to complete the circuit. This implies that no matter where we start, we will hit a deficit (i.e., a point where our tank falls below 0) before completing the circuit.

Consider a segment of the circuit where we run into a deficit:



We learned earlier that we cannot start at any station between a and b (inclusive) without running into a deficit. So, we can characterize this entire segment as having less total gas than the total cost required to travel through it.

After concluding that we cannot start anywhere from stations a to b, we decide to restart at the next station after b, which represents the start of the next segment. Keep in mind that since there is no solution in this proof, every segment in the circuit will end with a deficit:



This means each of these segments has less total gas than total cost. Therefore, for there to be no starting point, sum(gas) would have to be less than sum(cost) .

However, we know that $\text{sum(gas)} \geq \text{sum(cost)}$, confirming there must be a valid start point that allows us to complete the circuit.

Therefore, in our example, we can confirm that station 6 is the answer for the following reasons.

- $\text{sum(gas)} \geq \text{sum(cost)}$ implies that a solution must exist.
- We confirmed that starting anywhere before station 6 will result in us running into a deficit.
- We didn't encounter any deficit from station 6 to the last station in the array.

So, we just need to return `start`, which is station 6 in our example.

This is considered a **greedy solution** because we assume the first station we encounter that doesn't run into a deficit by the end of our array, is the start point that allows us to complete the circuit without testing every possible station in the array as the start point. The locally optimal choices (moving forward when possible, resetting when encountering a deficit) lead to the globally optimal solution (finding the correct starting point).

Implementation

```
def gas_stations(gas: List[int], cost: List[int]) -> int:
    # If the total gas is less than the total cost, completing the
    # circuit is impossible.
    if sum(gas) < sum(cost):
        return -1
    start = tank = 0
    for i in range(len(gas)):
        tank += gas[i] - cost[i]
        # If our tank has negative gas, we cannot continue through the
        # circuit from the current start point, nor from any station
        # before or including the current station 'i'.
        if tank < 0:
            # Set the next station as the new start point and reset the
            # tank.
            start, tank = i + 1, 0
    return start
```

Complexity Analysis

Time complexity: The time complexity of `gas_stations` is $O(n)$, where n denotes the length of the input arrays. This is because we iterate through each element in the `gas` and `cost` arrays.

Space complexity: The space complexity is $O(1)$.

Interview Tip

Tip: Demonstrate your greedy solution with examples if proving it formally is too difficult.



In some problems, such as this one, proving that a greedy solution works might be complicated, especially in an interview setting. If you and the interviewer are on the same page about this, a good compromise is to demonstrate the solution's correctness with a few diverse examples. This approach allows both you and the interviewer to have confidence in your solution in the absence of a thorough proof.

Candies

You teach a class of children sitting in a row, each of whom has a rating based on their performance. You want to distribute candies to the children while abiding by the following rules:

1. Each child must receive at least one candy.
2. If two children sit next to each other, the child with the higher rating must receive more candies.

Determine the minimum number of candies you need to distribute to satisfy these conditions.

Example 1:

Input: ratings = [4, 3, 2, 4, 5, 1]
Output: 12

Explanation: You can distribute candies to each child as follows: [3, 2, 1, 2, 3, 1].

Example 2:

Input: ratings = [1, 3, 3]
Output: 4

Explanation: You can distribute candies to each child as follows: [1, 2, 1].

Intuition

For starters, we know we need to give at least 1 candy to each of the n children to satisfy the first requirement of this problem. As for the other requirement, let's start off by considering a few specific cases.

Uniform ratings

Consider the situation where all children have the same rating. Since no child has a higher rating than another, we can give each child one candy to meet both requirements.

```
ratings = [ 2 == 2 == 2 == 2 == 2 == 2 ]  
candies = [ 1     1     1     1     1 ]
```

Increasing ratings

Now, consider the case where the children's ratings are in strictly increasing order. Here, each child should receive one more candy than their left-side neighbor.

```
ratings = [ 1 < 3 < 4 < 6 < 8 ]  
candies = [ 1     2     3     4     5 ]
```

The first child gets one candy because they have no left-side neighbor, and a smaller rating than their right-side neighbor. Each subsequent child has a higher rating than their left-side neighbor, so gets one more candy than that neighbor.

```

ratings = [ 1      3      4      6      8 ]
candies = [ 1      2      3      4      5 ]
                                         ↑
                                         rating[i] > ratings[i - 1]
                                         → candies[i] = candies[i - 1] + 1

```

Decreasing ratings

Here, each child needs to receive one more candy than their right-side neighbor.

```

ratings = [ 7      > 6      > 4      > 3      > 2      ]
candies = [ 5      4      3      2      1      ]

```

This is the same as the previous case, but in reverse. To populate the candies array in reverse, we start with the child furthest to the right, giving them one candy. Moving leftward, we give each child one more candy than their right-side neighbor:

```

ratings = [ 7      6      4      3      2      ]
candies = [ 5      4      3      2      1      ]
                                         ←
                                         rating[i] > ratings[i + 1]
                                         → candies[i] = candies[i + 1] + 1

```

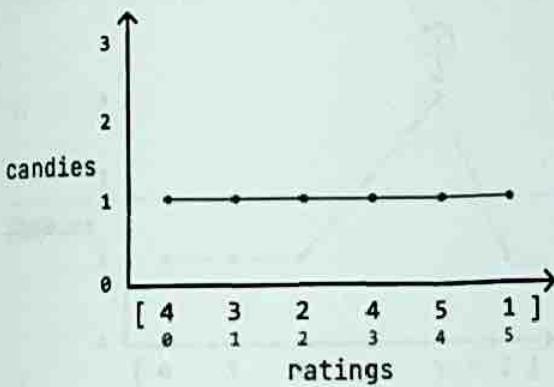
Non-uniform ratings

Unlike previous cases, in a non-uniform distribution of ratings, children can have higher ratings than their left-side neighbor, right-side neighbor, both neighbors, or neither. This makes handling non-uniform ratings more complex.

What if we can handle these cases separately? Specifically, we could use:

- One pass to ensure children with a higher rating than their left-side neighbor get more candy (handle increasing ratings).
- A second pass to ensure children with a higher rating than their right-side neighbor get more candy (handle decreasing ratings).

This allows us to apply the logic we used for increasing and decreasing ratings in two separate passes. Let's see if this strategy works over the following example, starting with an initial distribution of 1 candy for each child to ensure we meet the first requirement of this problem:



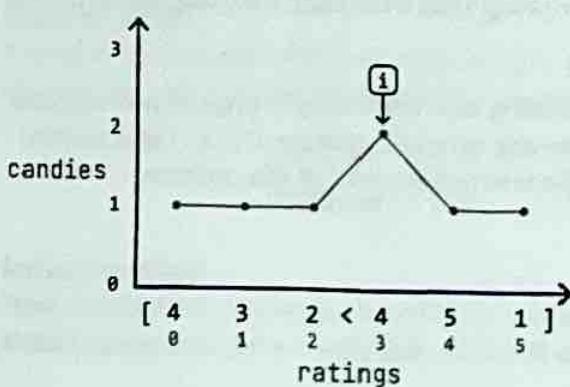
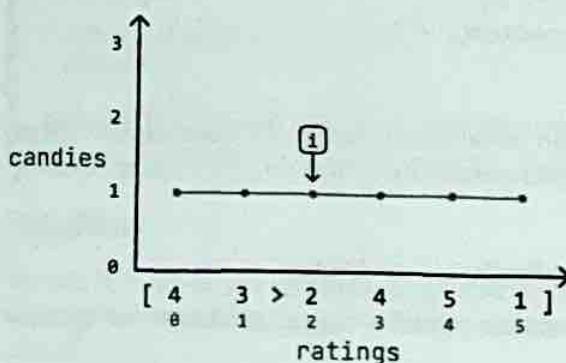
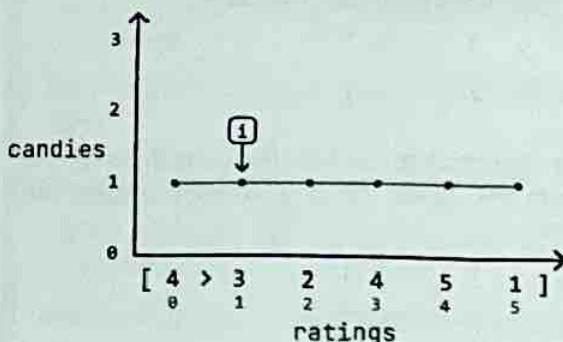
First pass: handle increasing ratings

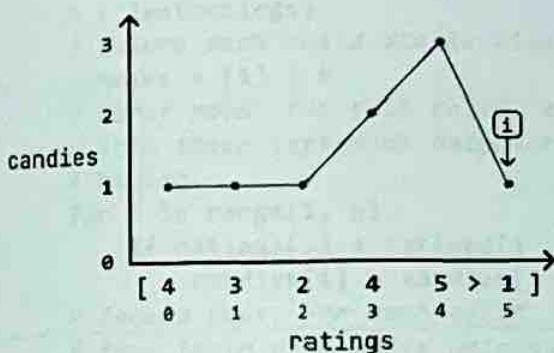
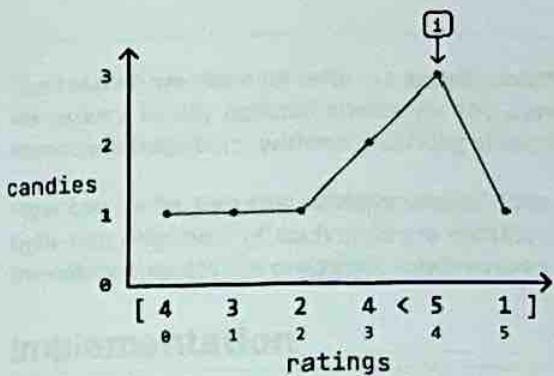
Iterate through the ratings array and, for each child, check if they have a higher rating than their

left-side neighbor (if `ratings[i] > ratings[i - 1]`). Start from index 1 since the child at index 0 doesn't have a left-side neighbor.

- If a child's rating is higher than their left-side neighbor's rating, make sure they have at least one more candy than their left-side neighbor (`candies[i] = candies[i - 1] + 1`).
- Otherwise, just continue to the next child's rating.

We can see how this process updates the candy distribution below:



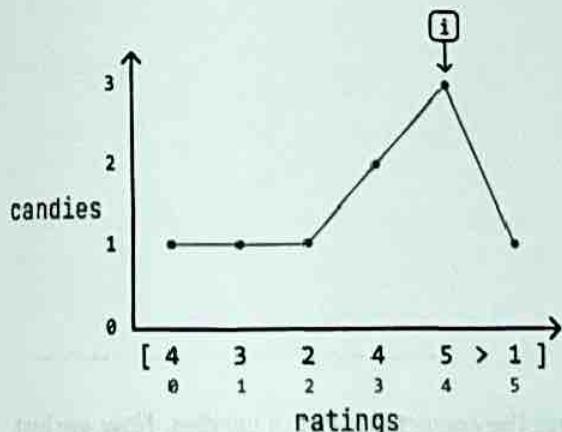


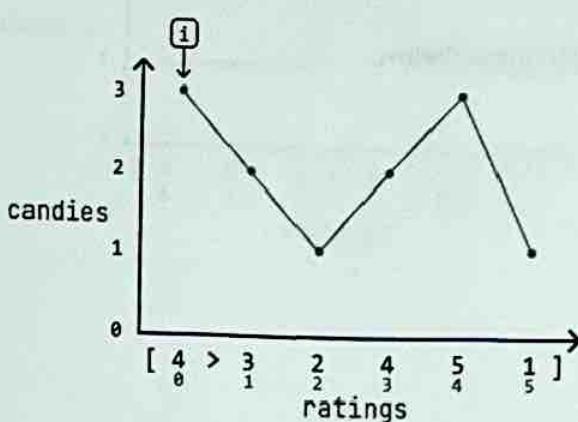
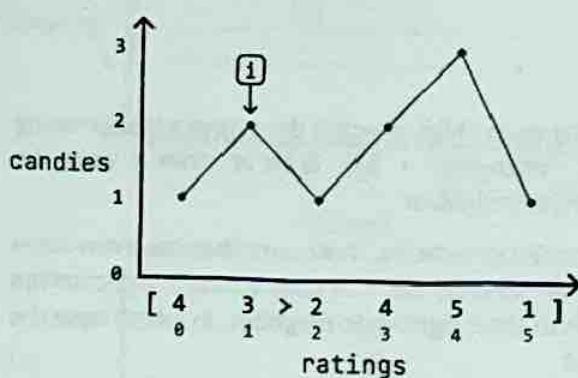
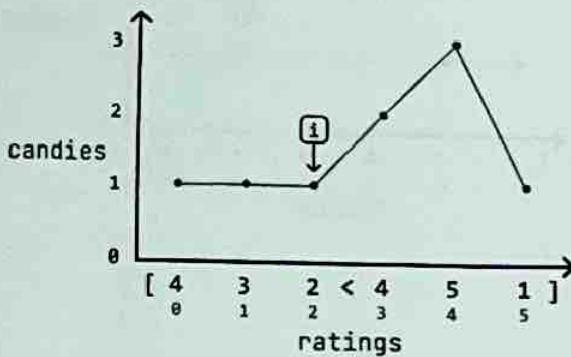
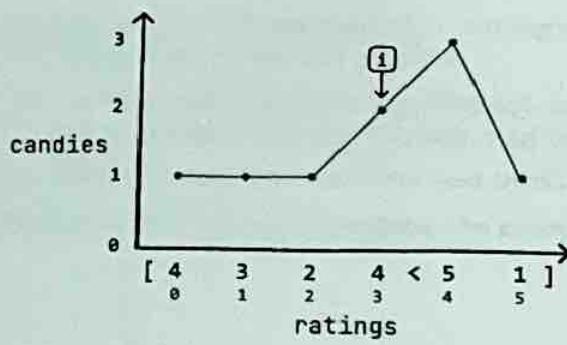
Second pass: handle decreasing ratings

Iterate through the `ratings` array in reverse and, for each child, check if they have a higher rating than their right-side neighbor (`if ratings[i] > ratings[i + 1]`). Start at index 4 since the child at index 5 (the last child) doesn't have a right-side neighbor.

- If a child's rating is higher than their right-side neighbor's rating, make sure they have one more candy than that neighbor. Note that because we already did one pass through the `candies` array, they might already have more candies than their right-side neighbor, in which case the current amount of candy they have is sufficient.
- Otherwise, continue to the next child's rating.

We can see how this process updates the candy distribution below:





At the end of the second pass, each child should have the correct number of candies. Now, we just need to return the sum of all the candies in the candies array.

The solution we came up with is a greedy solution because it satisfies the greedy choice property: we make a locally optimal choice for the current child by only considering the ratings of their immediate neighbors, without worrying about the ratings of other children.

How can we be sure this strategy works? Over our two passes, we ensure both the left-side and right-side neighbors of each child are considered when distributing candies. This guarantees that the solution meets the problem's requirements.

Implementation

```
def candies(ratings: List[int]) -> int:
    n = len(ratings)
    # Ensure each child starts with 1 candy.
    candies = [1] * n
    # First pass: for each child, ensure the child has more candies
    # than their left-side neighbor if the current child's rating is
    # higher.
    for i in range(1, n):
        if ratings[i] > ratings[i - 1]:
            candies[i] = candies[i - 1] + 1
    # Second pass: for each child, ensure the child has more candies
    # than their right-side neighbor if the current child's rating is
    # higher.
    for i in range(n - 2, -1, -1):
        if ratings[i] > ratings[i + 1]:
            # If the current child already has more candies than their
            # right-side neighbor, keep the higher amount.
            candies[i] = max(candies[i], candies[i + 1] + 1)
    return sum(candies)
```

Complexity Analysis

Time complexity: The time complexity of candies is $O(n)$ because we perform two passes over the `nums` array.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the `candies` array.

Additional resources for learning algorithms

• [GeeksforGeeks](#): A comprehensive resource for learning algorithms and data structures.

• [LeetCode](#): A platform for practicing algorithmic problems with various difficulty levels.

• [HackerRank](#): Another platform for solving algorithmic challenges and improving your skills.

Sort and Search

Introduction to Sort and Search

When sorting and searching items in data structures, efficiency is key. This chapter covers common sorting methods and their roles in efficient searching.

First, we present a comparison of the time and space complexities for various sorting algorithms. Below, n denotes the number of elements in the data structure:

Algorithm	Time complexity			Space complexity
	Best case	Average case	Worst case	
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Quicksort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(\log(n))$ (average) $O(n)$ (worst)
Heapsort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
Counting sort ¹	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$
Bucket sort ²	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$
Radix sort ³	$O(d \cdot (n + k))$	$O(d \cdot (n + k))$	$O(d \cdot (n + k))$	$O(n + k)$

This chapter focuses on merge sort, quicksort, and counting sort. There is additional information on the other algorithms in the references below, as well as a tool for visualizing how these algorithms work.

Fundamental concepts for sorting algorithms

¹ In counting sort, k represents the range of the values in the input array.

² In bucket sort, k represents the number of buckets used.

³ In radix sort, k represents the range of the inputs and d represents the number of digits in the maximum element.

Here's a list of fundamental attributes of sorting algorithms you should be familiar with:

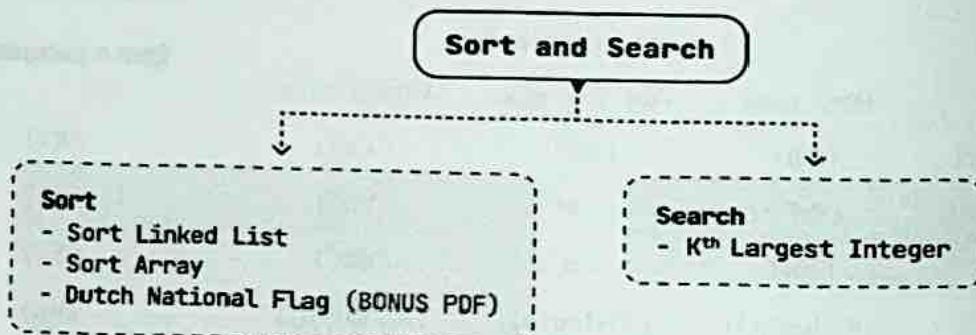
- **Stability:** A sorting algorithm is considered stable if it preserves the relative order of equal elements in the sorted output. If two elements have equal values, their order in the sorted output is the same as in the input.
- **In-place sorting:** An in-place sorting algorithm transforms the input using a constant amount of extra storage space. It involves sorting the elements within the original data structure.
- **Comparison-based sorting:** Comparison-based sorting algorithms sort elements by comparing them pairwise. These algorithms typically have a lower bound of $O(n \log(n))$, whereas non-comparison-based sorting algorithms can achieve linear time complexity, but require specific assumptions about the input data.

These concepts are referenced throughout the problems in this chapter.

Real-world Example

Sorting products by category: When users search for products, the platform often sorts the results based on various criteria such as lowest to highest price, highest to lowest rating, or even relevance to the search query. Efficient sorting algorithms ensure large datasets of products can be quickly arranged according to user preferences.

Chapter Outline



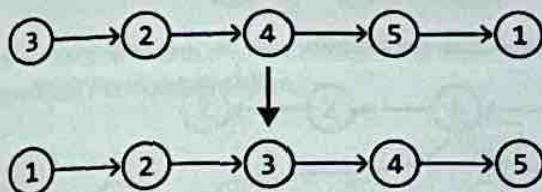
References

- [1] Insertion sort: https://en.wikipedia.org/wiki/Insertion_sort
- [2] Selection sort: https://en.wikipedia.org/wiki/Selection_sort
- [3] Bubble sort: https://en.wikipedia.org/wiki/Bubble_sort
- [4] Heapsort: <https://en.wikipedia.org/wiki/Heapsort>
- [5] Bucket sort: https://en.wikipedia.org/wiki/Bucket_sort
- [6] Radix sort: https://en.wikipedia.org/wiki/Radix_sort
- [7] Visualize sorting algorithms: <https://www.toptal.com/developers/sorting-algorithms>

Sort Linked List

Given the head of a singly linked list, sort the linked list in ascending order.

Example:



Intuition

Let's start by finding a sorting algorithm that allows us to sort a linked list.

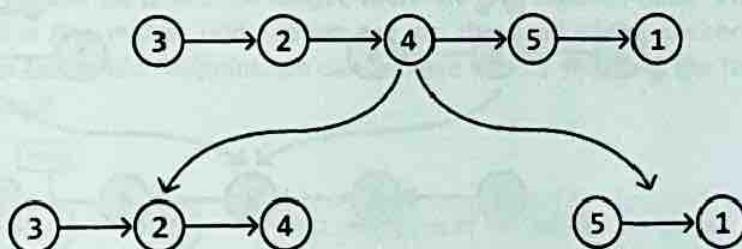
Choosing a sorting algorithm

We're tasked with sorting a linked list, not an array. This distinction is crucial because algorithms like quicksort rely on random access through indexing, which linked lists don't support. Merge sort is a great $O(n \log(n))$ time option, where n denotes the length of the linked list, because it does not require random access and works well with linked lists, as we'll see in this explanation.

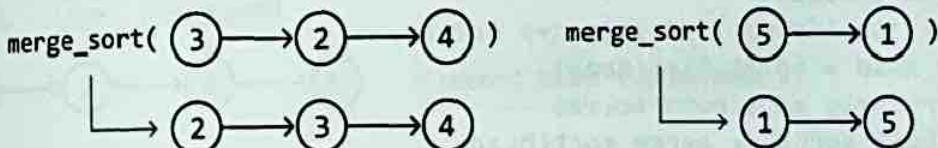
Merge sort

The merge sort algorithm uses a divide and conquer strategy. At a high level, it can be broken down into three steps:

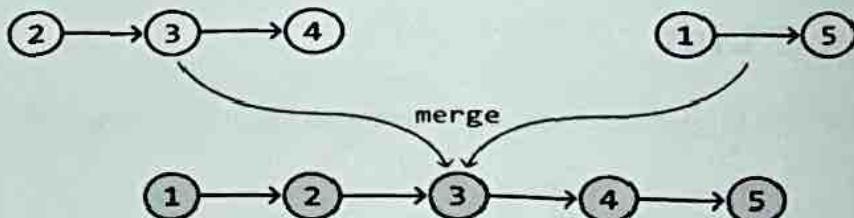
1. Split the linked list into two halves:



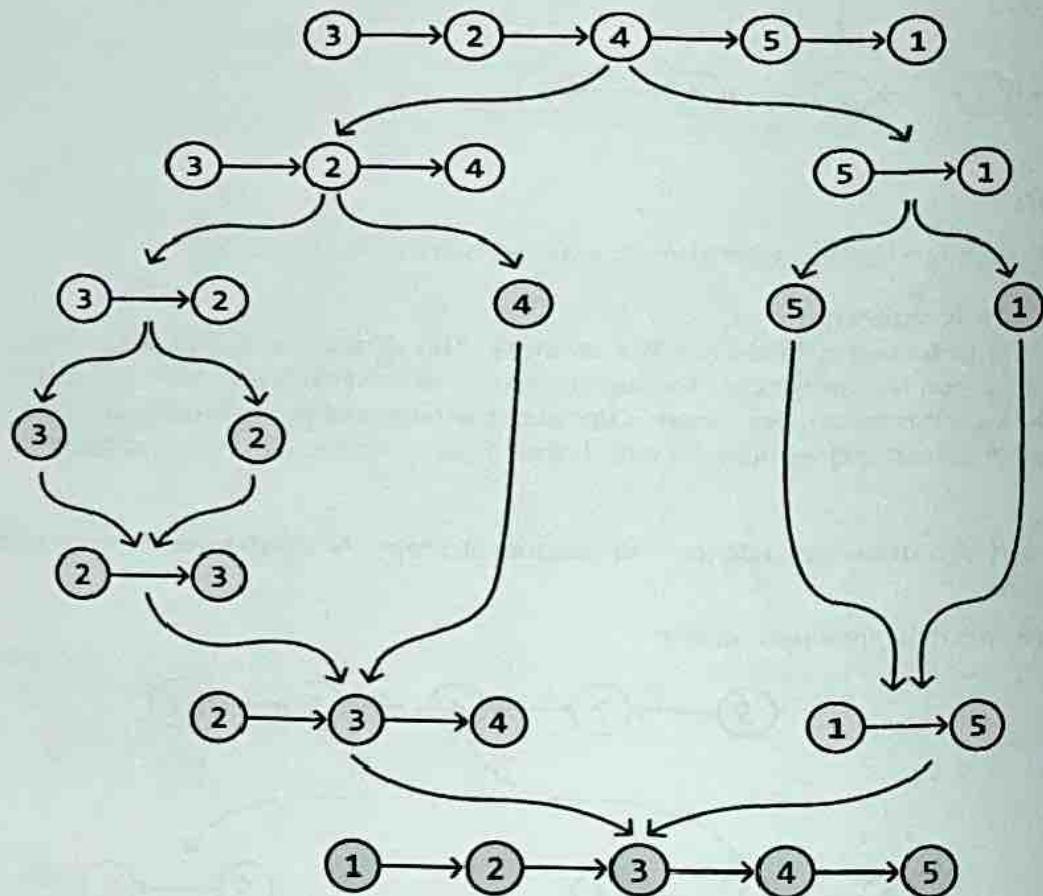
2. Recursively sort both halves:



3. Merge the halves back together in a sorted manner to form a single sorted list:



We can see what the entire process looks like in the diagram below:



This is what this process looks like as pseudocode:

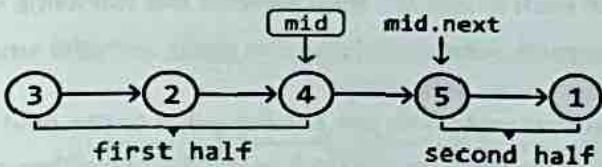
```
def merge_sort(head):
    # Split the linked list into two halves.
    second_head = split_list(head)
    # Recursively sort both halves.
    first_half_sorted = merge_sort(head)
    second_half_sorted = merge_sort(second_head)
    # Merge the sorted sublists.
    return merge(first_half_sorted, second_half_sorted)
```

Let's discuss in more detail how to split a linked list, and how to merge two sorted linked lists.

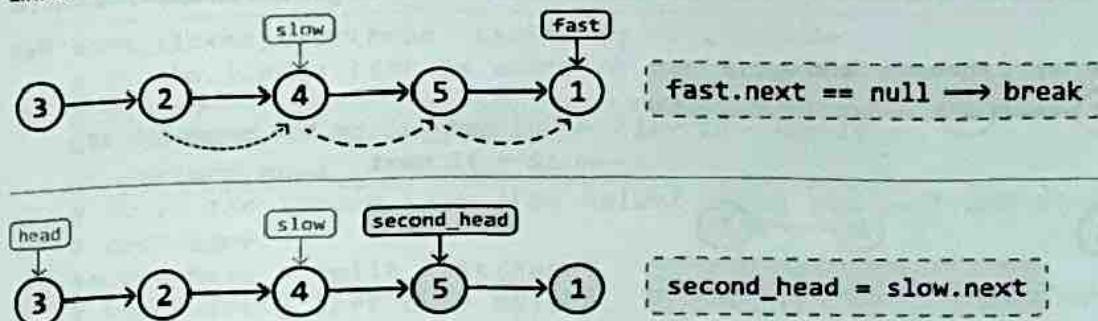
Splitting the linked list in half

To split a linked list in half, we need access to its middle node because the node next to the middle

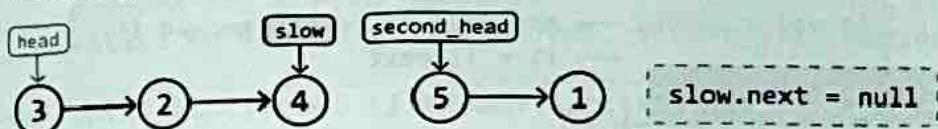
node can represent the head of the second linked list:



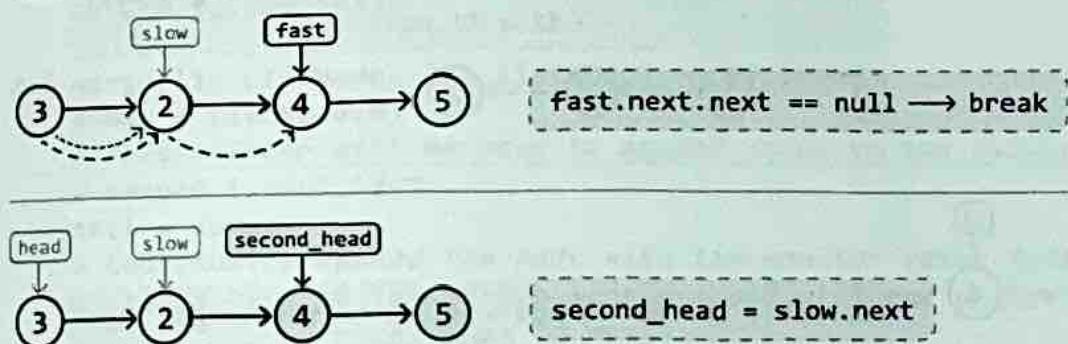
We can retrieve the middle node using the fast and slow pointer technique, as described in the [Linked List Midpoint problem](#):



Then, we just need to disconnect the two halves by setting `slow.next` to null:

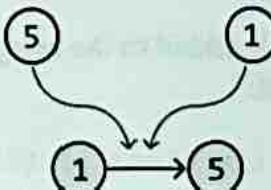


Note that when the linked list is of even length, there are two middle nodes. We want the slow pointer to stop at the first middle node so we can get the head of the second half more easily. As mentioned in [Linked List Midpoint](#), we can achieve this by stopping the fast pointer when `fast.next.next` is null:



Merging two sorted linked lists

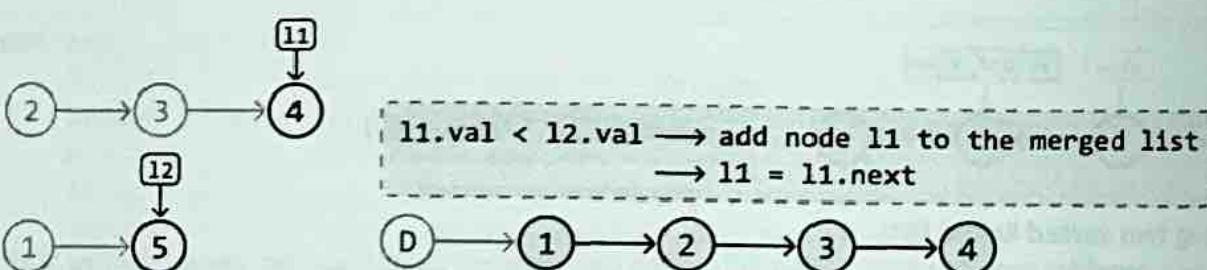
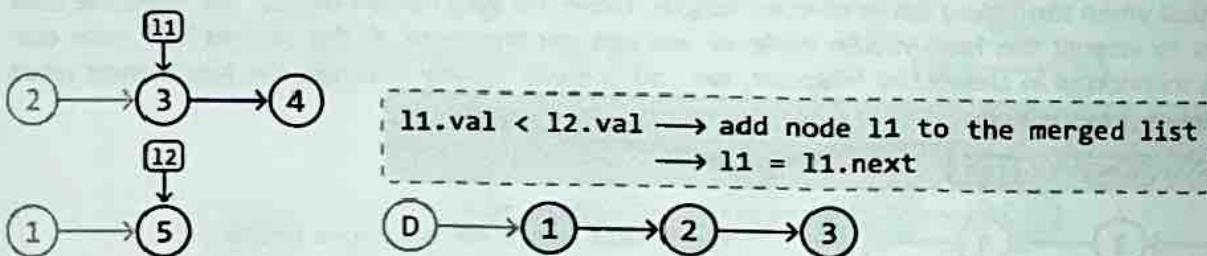
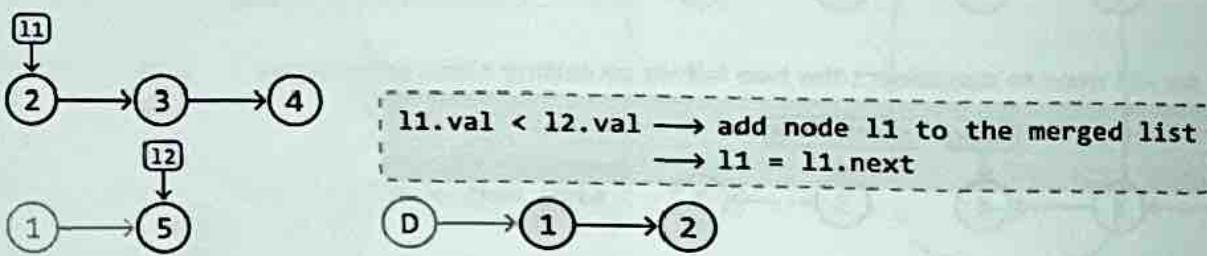
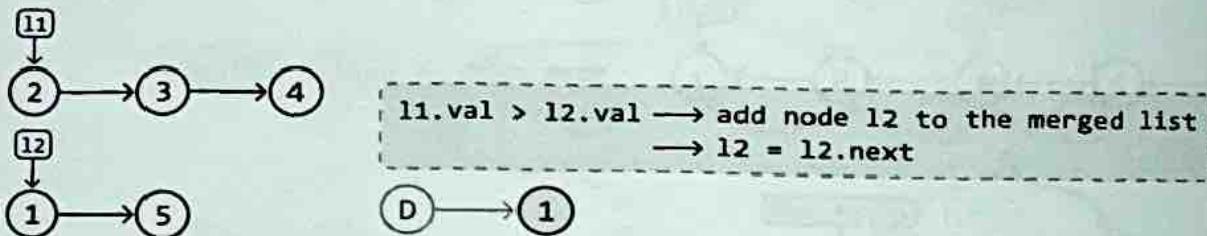
First, let's consider merging two linked lists, each containing a single node, meaning they're both inherently sorted. We merge them by placing the smaller node first, followed by the other node:



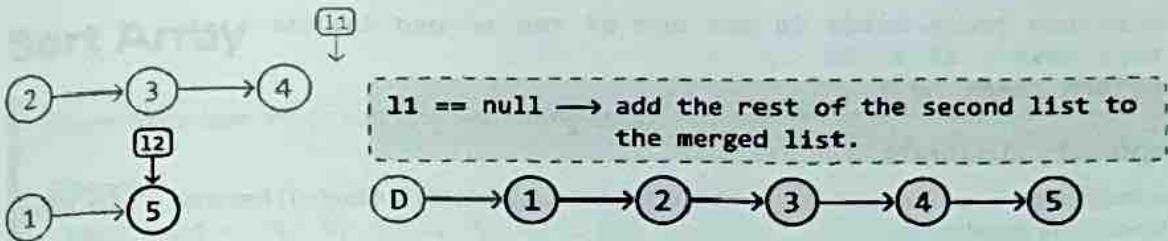
With that established, how would we merge two longer linked lists? To do this, we can set two pointers, one at the start of each linked list, then perform the following steps:

1. Compare the nodes at each pointer and add the node with the smaller value to the merged linked list.
2. Advance the pointer at that node with the smaller value to the next node in its linked list.
3. Repeat the above two steps until we can no longer advance either pointer.

Before discussing the final step, observe how these steps are applied to the following two sorted linked lists. We can use a dummy node to point to the head of the merged linked list:



If one of the linked lists has been entirely added to the merged list, we can just add the rest of the other linked list to the merged linked list:



Implementation

```

def sort_linked_list(head: ListNode) -> ListNode:
    # If the linked list is empty or has only one element, it's already
    # sorted.
    if not head or not head.next:
        return head
    # Split the linked list into halves using the fast and slow pointer
    # technique.
    second_head = split_list(head)
    # Recursively sort both halves.
    first_half_sorted = sort_linked_list(head)
    second_half_sorted = sort_linked_list(second_head)
    # Merge the sorted sublists.
    return merge(first_half_sorted, second_half_sorted)

def split_list(head: ListNode) -> ListNode:
    slow = fast = head
    while fast.next and fast.next.next:
        slow = slow.next
        fast = fast.next.next
    second_head = slow.next
    slow.next = None
    return second_head

def merge(l1: ListNode, l2: ListNode) -> ListNode:
    dummy = ListNode(0)
    # This pointer will be used to append nodes to the tail of the
    # merged linked list.
    tail = dummy
    # Continually append the node with the smaller value from each
    # linked list to the merged linked list until one of the linked
    # lists has no more nodes to merge.
    while l1 and l2:
        if l1.val < l2.val:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
    # One of the two linked lists could still have nodes remaining.

```

```
# Attach those nodes to the end of the merged linked list.  
tail.next = 11 or 12  
return dummy.next
```

Complexity Analysis

Time complexity: The time complexity of `sort_linked_list` is $O(n \log(n))$ because it uses merge sort. Here's the breakdown:

- The linked list is recursively split until each sublist contains only one node. This splitting process happens about $\log_2(n)$ times because each split reduces the size of the linked list by half.
- At each level, we merge the split linked lists. Merging all elements at one level takes about n operations.
- Since there are $\log_2(n)$ levels of splitting and merging, and there are n operations at each level, the time complexity is $O(n \log(n))$.

Space complexity: The space complexity is $O(\log(n))$ due to the recursive call stack, which can grow up to $\log_2(n)$ in height.

Stable Sorting

Merge sort is a stable sorting algorithm. This is important in scenarios where the original order of equal elements must be preserved. For example, if we are sorting a list of nodes that share the same value, but have an additional attribute storing the time they were created, then it's important for the relative order of these nodes to stay the same. Stability can be important for database systems; for instance, where stable sorting is required to maintain data integrity and consistency across multiple operations.

Sort Array

Given an integer array, sort it in ascending order.

Example:

Input: `nums = [6, 8, 4, 2, 7, 3, 1, 5]`

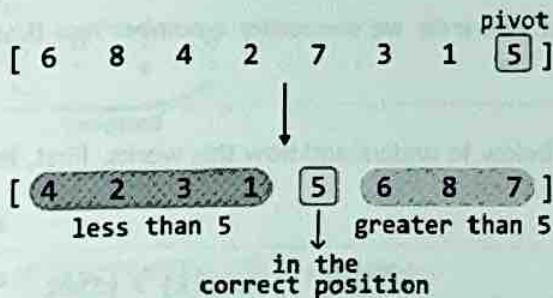
Output: `[1, 2, 3, 4, 5, 6, 7, 8]`

Intuition

This problem is quite open-ended because there are many sorting algorithms we could use to sort an array, each with its own pros and cons. In this explanation, we focus on the quicksort algorithm, which runs in $O(n \log(n))$ time on average, where n denotes the length of the array.

Quicksort

Conceptually, the goal of quicksort is to sort the array by placing each number in its sorted position one at a time. To correctly position a number, we move all numbers smaller than it to its left, and all numbers larger than it to its right. At each step, we call this number the pivot. We discuss how a pivot is selected later in the explanation.



This process is called **partitioning** because we're dividing the array into two parts around the pivot:

1. The left part with elements smaller than the pivot.
2. The right part with elements larger than the pivot.

Note that neither the left nor the right part of the pivot value need to be sorted. All that matters is that the pivot is in the correct position.

After partitioning, we just need to **sort the left and right parts**. We can do this by recursively calling quicksort on both of them. This makes quicksort a divide and conquer strategy. The pseudocode for this is provided below:

```
def quicksort(nums, left, right):
    # Partition the array and obtain the index of the pivot.
    pivot_index = partition(nums, left, right)
    # Sort the left and right parts.
    quicksort(nums, left, pivot_index - 1)
    quicksort(nums, pivot_index + 1, right)
```

Let's discuss the partitioning process in more detail.

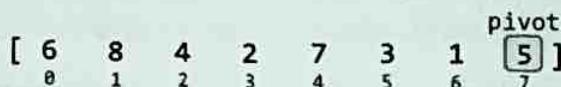
Partitioning

There are two primary steps to partitioning:

1. Selecting the pivot.
2. Rearranging the elements so elements smaller than the pivot are on its left, and elements larger than the pivot are on its right.

Selecting the pivot

We can actually choose any number as the pivot. The method of selecting a pivot can be optimized, which is discussed later, but for simplicity, we can choose the rightmost number as the pivot:



Rearranging elements around the pivot

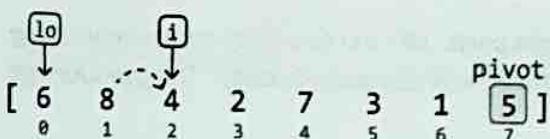
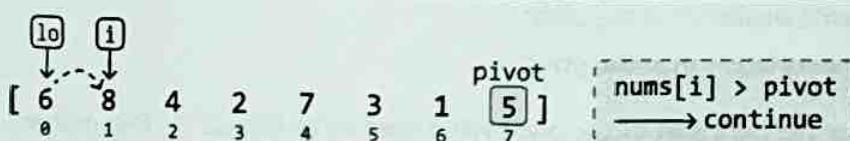
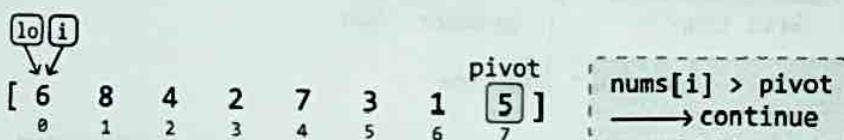
Let's see if we can do this in linear time without using extra space.

If we can ensure all numbers less than the pivot are placed to the left, then numbers greater than or equal to the pivot will consequently be placed to the right. This can be done using two pointers:

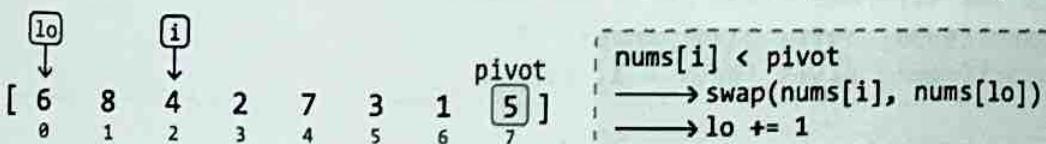
- One pointer at the left of the array (`lo`) to position the numbers less than the pivot.
- One pointer to iterate through the array (`i`), looking for numbers less than the pivot.

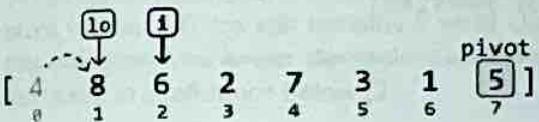
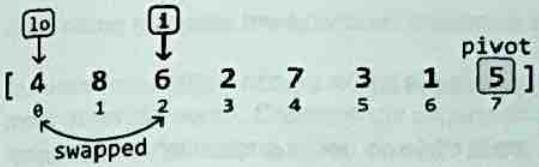
The main idea here is that whenever we encounter a number less than the pivot, swap it with `nums[lo]`.

Let's look at the example below to understand how this works. First, keep advancing `i` until we find a number less than the pivot:

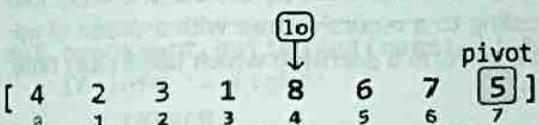


Once a number less than the pivot is found, move it to the left by swapping it with `nums[lo]`. Then, increment `lo` so it points to where the next number less than the pivot should be placed:

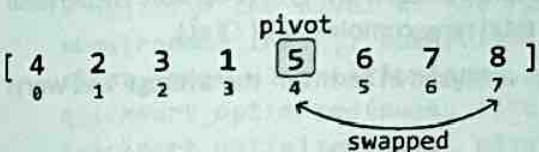




Continue this process until all numbers less than the pivot are on the left of the array:



The last step is to move the pivot to the correct position by swapping the pivot with nums[lo]:



Implementation

```

def sort_array(nums: List[int]) -> List[int]:
    quicksort(nums, 0, len(nums) - 1)
    return nums

def quicksort(nums: List[int], left: int, right: int) -> None:
    # Base case: if the subarray has 0 or 1 element, it's already
    # sorted.
    if left >= right:
        return
    # Partition the array and retrieve the pivot index.
    pivot_index = partition(nums, left, right)
    # Call quicksort on the left and right parts to recursively sort
    # them.
    quicksort(nums, left, pivot_index - 1)
    quicksort(nums, pivot_index + 1, right)

def partition(nums: List[int], left: int, right: int) -> int:
    pivot = nums[right]
    lo = left
    # Move all numbers less than the pivot to the left, which
    # consequently positions all numbers greater than or equal to the
    # pivot to the right.

```

```

for i in range(left, right):
    if nums[i] < pivot:
        nums[lo], nums[i] = nums[i], nums[lo]
        lo += 1
# After partitioning, 'lo' will be positioned where the pivot should
# be. So, swap the pivot number with the number at the 'lo' pointer.
nums[lo], nums[right] = nums[right], nums[lo]
return lo

```

Complexity Analysis

Time complexity: The time complexity of `sort_array` can be analyzed in terms of the average and worst cases:

- **Average case:** $O(n \log(n))$. In the average case, quicksort effectively divides the array into two roughly equal parts after each partition, leading to a recursive tree with a depth of approximately $\log_2(n)$. For each of these levels, we perform a partition which takes $O(n)$ time, resulting in a total time complexity of $O(n \log(n))$.
- **Worst case:** $O(n^2)$. The worst-case scenario occurs when the pivot selection consistently results in extremely unbalanced partitions, such as when the smallest or largest element is always chosen as a pivot, which is explained in more detail in the optimization. Uneven partitioning can result in a recursive depth as deep as n . For each of these n levels of recursion, we perform a partition which takes $O(n)$ time, resulting in a total time complexity of $O(n^2)$.

Space complexity: The space complexity can also be analyzed in terms of the average and worst cases:

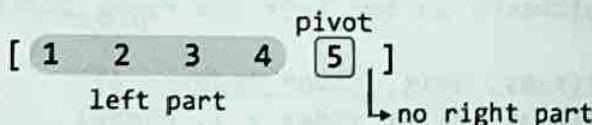
- **Average case:** $O(\log(n))$. In the average case, the depth of the recursive call stack is approximately $\log_2(n)$.
- **Worst case:** $O(n)$. In the worst case, the depth of the recursive call stack can be as deep as n .

Optimization

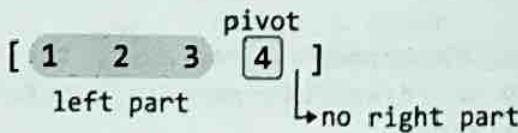
As mentioned in the above complexity analysis, it's possible for a worst-case time complexity of $O(n^2)$ to occur. Let's dive into when this can happen. Consider the following array, which is already sorted:

[1 2 3 4 5]

If we perform quicksort on this array, we choose the rightmost element as the pivot and partition the other elements around this pivot. However, since this pivot is the largest element, there will be $n - 1$ elements less than the pivot and 0 elements greater than or equal to it:



This creates quite an uneven partition. When we call quicksort on the left part, the same imbalance will occur, but with a left part consisting of $n - 2$ elements:



Continuing this until the quicksort process is complete results in a recursion depth of n .

An uneven partition occurs when we choose an **extreme pivot**: one that's larger or smaller than most other elements. Consistently picking an extreme pivot can occur when the array is sorted in increasing or decreasing order, or when there are many duplicates in the array.

To reduce the likelihood of choosing an extreme pivot, we can modify quicksort to choose a **random pivot** instead. There still remains a small chance of consistently picking an extreme pivot, but this outcome is no longer dependent on the order of the input array. A more detailed answer is discussed in a reference below [2].

We can integrate this change into our current solution by randomly selecting an index and swapping its element with the rightmost element, before performing the partition. This way, we don't have to modify the partition function, as it uses the rightmost element:

```
def quicksort_optimized(nums: List[int], left: int, right: int) -> None:
    if left >= right:
        return
    # Choose a pivot at a random index.
    random_index = random.randint(left, right)
    # Swap the randomly chosen pivot with the rightmost element to
    # position the pivot at the rightmost index.
    nums[random_index], nums[right] = nums[right], nums[random_index]
    pivot_index = partition(nums, left, right)
    quicksort_optimized(nums, left, pivot_index - 1)
    quicksort_optimized(nums, pivot_index + 1, right)
```

Interview Follow-Up

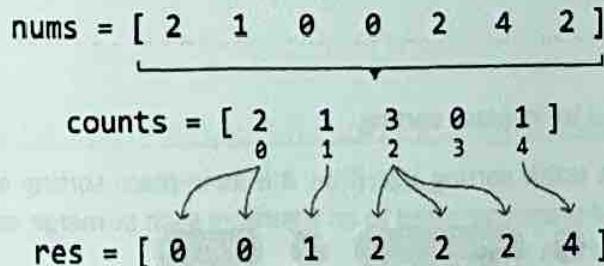
Let's say the interviewer introduces the following constraints to the initial sorting problem:

- The input array does not contain negative values.
- All values in the input array are less than or equal to 10^3 .

Does our approach to the problem change, and should we still use quicksort? Considering that all values in our array now fall within the limited range of $[0, 10^3]$, a counting sort approach becomes appropriate.

Counting sort

Counting sort is a non-comparison-based sorting algorithm that works by counting the number of occurrences of each element in the array, then using these counts to place each element in its correct sorted position:



We can do this in two steps:

- Count occurrences: create a counts array, where each of its indexes represents an element from the original array. Increment the value at each index based on how many times the corresponding element appears in the original array.
- Build sorted array (res): iterate through each index of the counts array and add that index (i) to the sorted array as many times as its value ($\text{counts}[i]$) indicates.

Counting sort is efficient here because we know the largest possible number in the array is at most 10^3 , which means our counts array will have a maximum size of $10^3 + 1$. However, if this problem constraint is not specified and the maximum value in the array may be very large, then a counting sort solution might not be appropriate, due to the potentially large size of the counts array.

Implementation

Note that there's another common method for implementing counting sort, which is detailed in the reference provided [1].

```
def sort_array_counting_sort(nums: List[int]) -> List[int]:
    if not nums:
        return []
    res = []
    # Count occurrences of each element in 'nums'.
    counts = [0] * (max(nums) + 1)
    for num in nums:
        counts[num] += 1
    # Build the sorted array by appending each index 'i' to it a total
    # of 'counts[i]' times.
    for i, count in enumerate(counts):
        res.extend([i] * count)
    return res
```

Complexity Analysis

Time complexity: The time complexity of `sort_array_counting_sort` is $O(n+k)$, where k denotes the maximum value of `nums`. This is because it takes $O(n)$ time to count the occurrences of each element and $O(k)$ time to build the sorted array.

Space complexity: The space complexity is $O(n+k)$, since the `res` array occupies $O(n)$ space, and the `counts` array takes up $O(k)$ space. Note that `res` is considered in the space complexity, as counting sort is not an in-place sorting algorithm requiring an additional array to store the sorted result.

Interview Tip

Tip: Quicksort is useful for in-place sorting.



While quicksort isn't a stable sorting algorithm, it is an in-place sorting algorithm, meaning it requires less additional space compared to an algorithm such as merge sort, which takes $O(n)$ space when used to sort an array.

References

- [1] Counting sort: https://en.wikipedia.org/wiki/Counting_sort
- [2] What is the advantage of Randomized Quicksort?: <https://cs.stackexchange.com/questions/7582/what-is-the-advantage-of-randomized-quicksort>

Kth Largest Integer

Return the kth largest integer in an array.

Example:

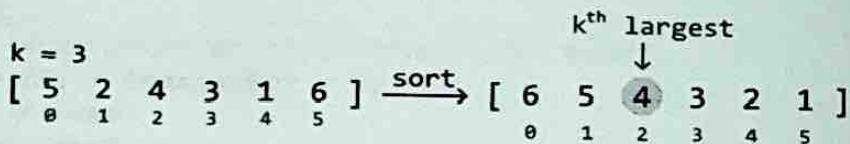
Input: nums = [5, 2, 4, 3, 1, 6], k = 3
Output: 4

Constraints:

- The array contains no duplicates.
- The array contains at least one element.
- $1 \leq k \leq n$, where n denotes the length of the array.

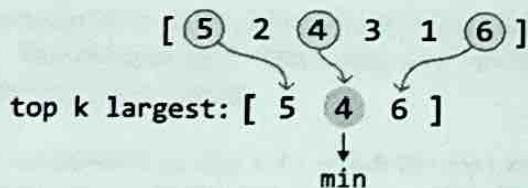
Intuition – Min-Heap

A straightforward solution to this problem is to sort the array in reverse order and return the number at the $(k - 1)^{\text{th}}$ index:



This solution takes $O(n \log(n))$ time, but as we only need the kth largest integer, sorting all the elements in the input array might not be necessary. Let's explore some solutions which take this into account.

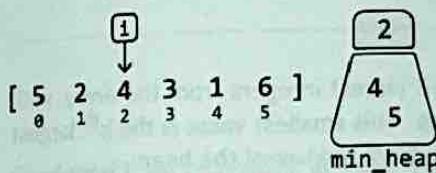
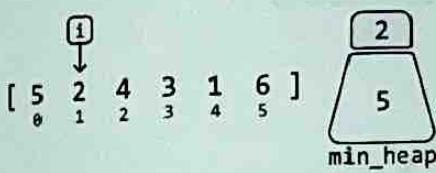
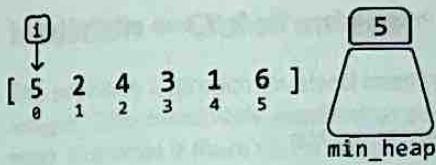
An interesting thing to realize is that if we know what the top k largest integers are, we'd know that the smallest of these would be the kth largest integer:



This leads to the idea that instead of sorting the entire array, we can keep track of the top k largest integers in the array. Is there a way to maintain the top k largest integers, while having access to the smallest of these integers?

A min-heap seems like it should work well for this purpose because it provides efficient access to the smallest element. Let's explore how to use it to maintain the top k integers in the array.

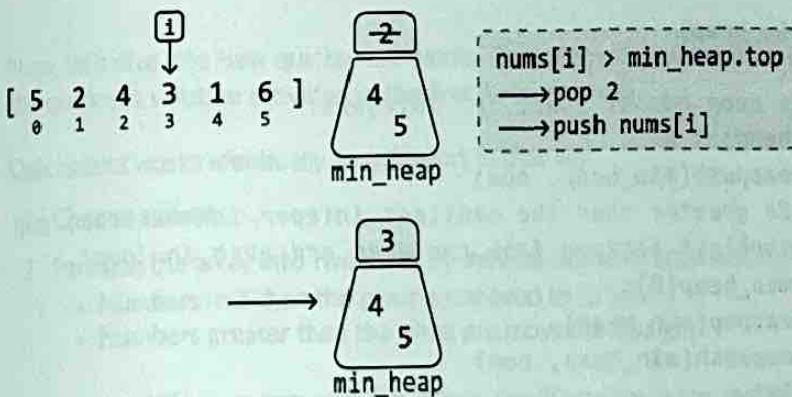
Consider the below example with k = 3. Let's begin by pushing the first k integers to the heap:



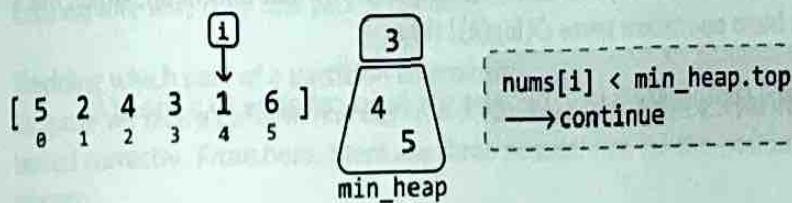
We shouldn't immediately push the next element, 3, in the heap because the heap already contains k integers. This gives us a choice:

- Skip 3, or:
- Remove an integer from the heap and push 3 in.

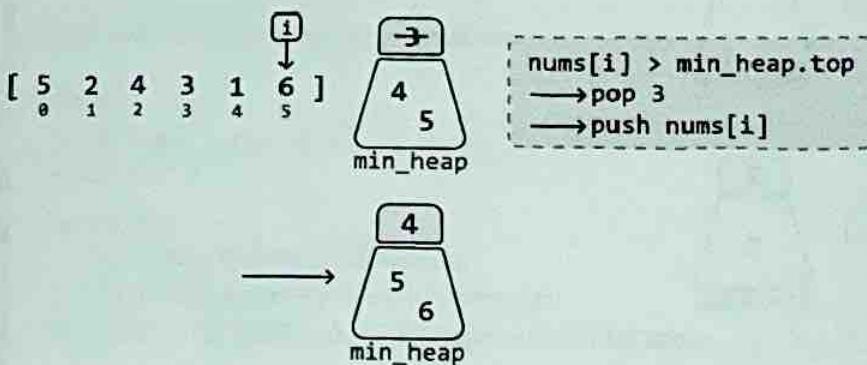
The smallest integer in the heap is 2. Integer 3 is more likely to belong in the top k integers than 2 is, so let's pop 2 from the heap and push 3 in:



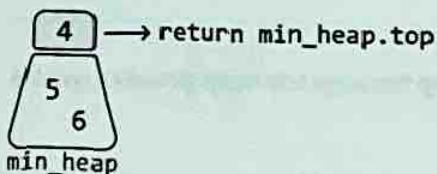
The next integer is 1, which is smaller than the smallest integer in the heap. So, we know it's not among the top k integers:



The final integer is 6. It's larger than the smallest integer in the heap. So, let's pop off the top of the heap and push in 6:



At this point, the k integers in the heap represent the top k largest integers from the array, with the smallest of these integers located at the top of the heap. This smallest value is the k^{th} largest integer in the entire array, so we can just return this integer from the top of the heap:



Implementation – Min-Heap

```
def kth_largest_integer_min_heap(nums: List[int], k: int) -> int:
    min_heap = []
    heapq.heapify(min_heap)
    for num in nums:
        # Ensure the heap has at least 'k' integers.
        if len(min_heap) < k:
            heapq.heappush(min_heap, num)
        # If 'num' is greater than the smallest integer in the heap, pop
        # off this smallest integer from the heap and push in 'num'.
        elif num > min_heap[0]:
            heapq.heappop(min_heap)
            heapq.heappush(min_heap, num)
    return min_heap[0]
```

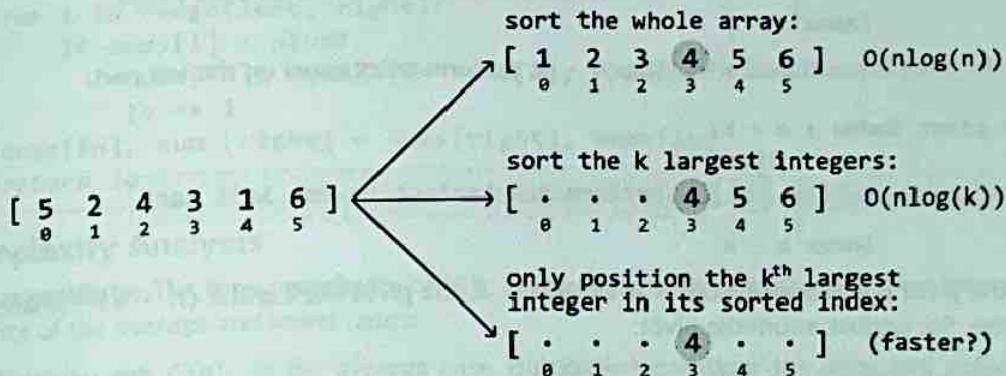
Complexity Analysis

Time complexity: The time complexity of `kth_largest_integer_min_heap` is $O(n \log(k))$ because for each integer, we perform at most one push and pop operation on the min-heap, which has a size no larger than k . Each heap operation takes $O(\log(k))$ time.

Space complexity: The space complexity is $O(k)$ because the heap can grow to a size of k .

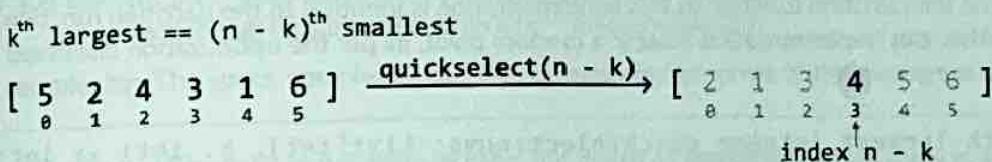
Intuition – Quickselect

The previous approach involved keeping track of the k largest integers to identify the k^{th} largest integer. This effectively required us to keep these k values partially sorted through the use of a heap. But what if there's a way to position only the k^{th} largest value in its expected sorted position, without sorting the other numbers?



Quickselect can be used to achieve this.

Quickselect is an algorithm that leverages the partition step of quicksort, which positions a value in its sorted position. Quickselect is generally used to find the k^{th} smallest element, whereas, in this problem, we're asked to find the k^{th} largest element. To utilize quickselect for this purpose, we can instead find the $(n - k)^{\text{th}}$ smallest integer, which is equivalent to finding the k^{th} largest element:



Now, let's dive into how quickselect works. To understand quickselect, we recommend you study the quicksort solution provided in the Sort Array problem.

Quickselect works identically to quicksort in that we:

1. Choose a pivot.
2. Partition the array into two parts by moving elements around the pivot so that:
 - Numbers less than the pivot are moved to its left.
 - Numbers greater than the pivot are moved to its right.

The primary difference between these two algorithms lies in the recursion step:

- In quicksort, we recursively process both the left and right parts.
- In quickselect, we only need to recursively process one of these parts.

Let's explore why only one part is chosen.

Deciding which part of a partition to process

Suppose we pick a random number as a pivot. After performing a partition, this pivot will be positioned correctly. From here, there are three possibilities for the position of the $(n - k)^{\text{th}}$ smallest integer:

- If the pivot is positioned before index $n - k$, it means the $(n - k)^{\text{th}}$ smallest integer must be somewhere to the right of the pivot. So, perform quickselect on the right part:

```
if pivot_index < n - k:
    pivot
    [ . . . . . ] [ perform quickselect on the right part ]
    ↑
    index n - k
```

- If the pivot is positioned after index $n - k$, perform quickselect on the left part:

```
if pivot_index > n - k:
    pivot
    [ . . . . . ] [ perform quickselect on the left part ]
    ↑
    index n - k
```

- If the pivot is positioned exactly at index $n - k$, the pivot itself is the $(n - k)^{\text{th}}$ smallest integer. So, we just return the pivot:

```
if pivot_index == n - k:
    pivot
    [ . . . . . ] [ return nums[pivot_index] ]
    ↑
    index n - k
```

Implementation – Quickselect

Note that the partition function in this implementation is identical to the partition function in Sort Array. Also, this implementation selects a random pivot, as per the optimization discussed in Sort Array.

```
def kth_largest_integer_quickselect(nums: List[int], k: int) -> int:
    return quickselect(nums, 0, len(nums) - 1, k)

def quickselect(nums: List[int], left: int, right: int, k: int) -> None:
    n = len(nums)
    if left >= right:
        return nums[left]
    random_index = random.randint(left, right)
    nums[random_index], nums[right] = nums[right], nums[random_index]
    pivot_index = partition(nums, left, right)
    # If the pivot comes before 'n - k', the ('n - k')th smallest
    # integer is somewhere to its right. Perform quickselect on the
    # right part.
    if pivot_index < n - k:
        return quickselect(nums, pivot_index + 1, right, k)
    # If the pivot comes after 'n - k', the ('n - k')th smallest integer
    # is somewhere to its left. Perform quickselect on the left part.
    elif pivot_index > n - k:
        return quickselect(nums, left, pivot_index - 1, k)
    # If the pivot is at index 'n - k', it's the ('n - k')th smallest
    # integer.
```

```

else:
    return nums[pivot_index]

def partition(nums: List[int], left: int, right: int) -> int:
    pivot = nums[right]
    lo = left
    for i in range(left, right):
        if nums[i] < pivot:
            nums[lo], nums[i] = nums[i], nums[lo]
            lo += 1
    nums[lo], nums[right] = nums[right], nums[lo]
    return lo

```

Complexity Analysis

Time complexity: The time complexity of `kth_largest_integer_quickselect` can be analyzed in terms of the average and worst cases:

- Average case: $O(n)$. In the average case, quickselect partitions the array and reduces the problem size by approximately half each time by performing a recursive call on only one part of each partition. A linear partition is performed during each of these recursive calls. This results in a total time complexity of $O(n) + O(\frac{n}{2}) + O(\frac{n}{4}) + \dots$ until the base case of quickselect is reached. This can be simplified to an $O(n)$ time complexity.
- Worst case: $O(n^2)$. The worst-case scenario occurs when the pivot selection consistently results in extremely unbalanced partitions. This can result in the problem size only being reduced by one element after each partition, leading to a total time complexity of $O(n) + O(n - 1) + O(n - 2) + \dots$, which can be simplified to an $O(n^2)$ time complexity.

Space complexity: The space complexity can also be analyzed in terms of the average and worst cases:

- Average Case: $O(\log(n))$. In the average case, the depth of the recursive call stack is approximately $\log_2(n)$.
- Worst Case: $O(n)$. In the worst case, the depth of the recursive call stack can be as deep as n .

Bitwise Operators

There are several fundamental bitwise operators used for manipulating binary integers. These are described along with their operators' truth tables.

- *Binary search* is a search algorithm that repeatedly narrows the search range by half until it finds the target value or determines that the target value does not exist.
- *Radix sort* is a sorting algorithm that sorts elements based on their digits. It uses multiple passes to sort the elements by each digit position, starting from the least significant digit and moving towards the most significant digit.
- *Quicksort* is a recursive sorting algorithm that works by partitioning the array into two halves based on a pivot element, then recursively applying the same process to each half until all elements are sorted.
- *Mergesort* is a recursive sorting algorithm that works by dividing the array into smaller and smaller subarrays until they are simple enough to be sorted directly, then merging them back together in sorted order.
- *Heapsort* is a sorting algorithm that uses a heap data structure to efficiently find the maximum or minimum element in each pass of the sort.
- *Counting sort* is a sorting algorithm that counts the occurrences of each value in the array and then uses that information to place the values in the correct order.
- *Bucket sort* is a sorting algorithm that divides the array into several buckets, sorts the elements within each bucket, and then concatenates the sorted buckets to produce the final sorted array.
- *Radix sort* is a sorting algorithm that sorts elements based on their digits. It uses multiple passes to sort the elements by each digit position, starting from the least significant digit and moving towards the most significant digit.
- *Quicksort* is a recursive sorting algorithm that works by partitioning the array into two halves based on a pivot element, then recursively applying the same process to each half until all elements are sorted.
- *Mergesort* is a recursive sorting algorithm that works by dividing the array into smaller and smaller subarrays until they are simple enough to be sorted directly, then merging them back together in sorted order.
- *Heapsort* is a sorting algorithm that uses a heap data structure to efficiently find the maximum or minimum element in each pass of the sort.
- *Counting sort* is a sorting algorithm that counts the occurrences of each value in the array and then uses that information to place the values in the correct order.
- *Bucket sort* is a sorting algorithm that divides the array into several buckets, sorts the elements within each bucket, and then concatenates the sorted buckets to produce the final sorted array.

Bit Manipulation

Introduction to Bit Manipulation

Bit manipulation is a technique used in programming to perform operations at the bit level, which can often lead to more efficient and faster algorithms.

When is bit manipulation useful?

Bit manipulation allows us to work directly with the binary representation of numbers, making certain operations more efficient. Common tasks such as setting, clearing, toggling, and checking bits can be performed quickly using bitwise operators.

For example, one of the most common space optimization techniques involves using an unsigned 32-bit integer to represent a set of boolean values, where each bit in the integer corresponds to a different boolean value. This allows us to store and manipulate up to 32 states without using a boolean array or hash set.

F ↓ 0 31	T ↓ 1 30	F ↓ 0 29	...	F ↓ 0 7	T ↓ 1 6	F ↓ 0 5	F ↓ 0 4	T ↓ 1 3	F ↓ 0 2	T ↓ 1 1	T ↓ 1 0
-------------------	-------------------	-------------------	-----	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------

Bitwise Operators

There are several fundamental bitwise operations, each serving a specific purpose. These are shown below, along with each operation's truth table:

NOT:

a	$\sim a$
0	1
1	0

Example:

$$\begin{array}{r} \text{NOT } 1 \ 0 \ 0 \ 1 \\ \hline = \ 0 \ 1 \ 1 \ 0 \end{array}$$

AND:

a	b	$a \& b$
0	0	0
0	1	0
1	0	0
1	1	1

Example:

$$\begin{array}{r} 1 \ 0 \ 0 \ 1 \\ \text{AND } 0 \ 1 \ 0 \ 1 \\ \hline = \ 0 \ 0 \ 0 \ 1 \end{array}$$

OR:		
a	b	a b
0	0	0
0	1	1
1	0	1
1	1	1

Example:

1	0	0	1
OR	0	1	0
<hr/>			
=	1	1	0

XOR:		
a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Example:

1	0	0	1
NOR	0	1	0
<hr/>			
=	1	1	0

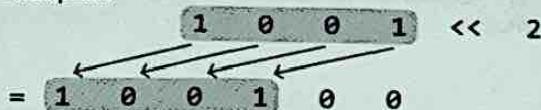
Some useful characteristics of the XOR operator are:

- $a \wedge 0 = a$
- $a \wedge a = 0$

In addition, it's also important to understand the fundamental shift operators:

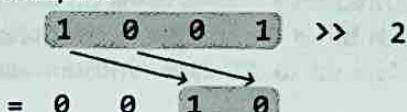
- **Left shift ($<< n$):** Shifts the bits of a number to the left by n positions, adding 0s on the right. This is equivalent to multiplying a number by 2^n .

Example:



- **Right shift ($>> n$):** Shifts the bits of a number to the right by n positions, discarding bits on the right. This is equivalent to dividing a number by 2^n (integer division).

Example:



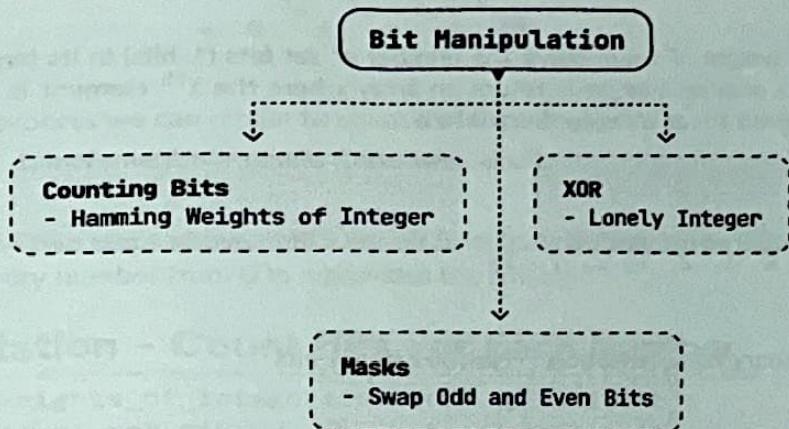
Using these operators, here are some useful bit manipulation techniques to be aware of:

- Setting the i^{th} bit of x to 1: $x |= (1 << i)$
- Clearing the i^{th} bit of x : $x &= ~(1 << i)$
- Toggling the i^{th} bit of x (from 0 to 1 or 1 to 0): $x ^= (1 << i)$
- Checking if the i^{th} bit is set: if $x & (1 << i) != 0$, the i^{th} bit is set
- Checking if a number x is even or odd: if $x & 1 == 0$, x is even
- Checking if a number is a power of 2: if $x > 0$ and $x & (x - 1) == 0$, x is a power of 2

Real-world Example

Data transmission in networks: In many network protocols, bit manipulation is used to efficiently encode, compress, and transmit data for fast communication. For example, IP addresses and subnet masks use bitwise AND operations to determine whether two devices are on the same network. Similarly, in error detection and correction algorithms like checksums or parity bits, bit manipulation promotes data integrity during transmission by identifying and correcting errors in the binary data.

Chapter Outline



To best grasp the fundamentals of bit manipulation, this chapter explores a variety of problems that utilize a range of complex bit manipulation techniques, as well as how to identify the appropriate bitwise operator based on specific requirements.

Complexity Analysis

The complexity: The time complexity for many of the problems in this chapter is linear, integer n from 0 to n . Some problems, such as counting bits or finding the most significant bit, have a complexity of $O(n)$, since we must iterate over all bits. Other problems, such as finding the next power of two, have a complexity of $O(1)$.

The space complexity: The space complexity for many of the problems in this chapter is constant, as they do not require any additional memory to store intermediate results or auxiliary data structures.

Introducing Dynamic Programming A common theme throughout this chapter is the product approach. It's important to note that this approach has a time complexity of $O(n^2)$ for all of the problems. For example, if you wanted to calculate the result for all of the powers of two from 0 to 15, you would need to calculate $2^0 \times 2^1 \times 2^2 \times 2^3 \times 2^4 \times 2^5 \times 2^6 \times 2^7 \times 2^8 \times 2^9 \times 2^{10} \times 2^{11} \times 2^{12} \times 2^{13} \times 2^{14} \times 2^{15}$. If you wanted to calculate the result for all of the powers of three from 0 to 15, you would need to calculate $3^0 \times 3^1 \times 3^2 \times 3^3 \times 3^4 \times 3^5 \times 3^6 \times 3^7 \times 3^8 \times 3^9 \times 3^{10} \times 3^{11} \times 3^{12} \times 3^{13} \times 3^{14} \times 3^{15}$. This is clearly not feasible, so we need to find a more efficient way to calculate these values.

A good way to reduce the time complexity of these calculations is to use dynamic programming. This is the technique of calculating values in a sequence based on previous values. For example, if you wanted to calculate the result for all of the powers of two from 0 to 15, you could use dynamic programming to calculate the result for each power of two sequentially. You would start with $2^0 = 1$, then calculate $2^1 = 2$, then $2^2 = 4$, then $2^3 = 8$, and so on. This way, you can reuse the previous value to calculate the next value, which significantly reduces the time complexity of the calculation.

Hamming Weights of Integers

The Hamming weight of a number is the number of set bits (1-bits) in its binary representation. Given a positive integer n , return an array where the i^{th} element is the Hamming weight of integer i for all integers from 0 to n .

Example:

Input: $n = 7$

Output: [0, 1, 1, 2, 1, 2, 2, 3]

Explanation:

Number	Binary representation	Number of set bits
0	0	0
1	1	1
2	10	1
3	11	2
4	100	1
5	101	2
6	110	2
7	111	3

Intuition – Count Bits For Each Number

The most straightforward strategy is to individually count the number of bits for each number from 0 to n .

Consider a number $x = 25$ and its binary representation:

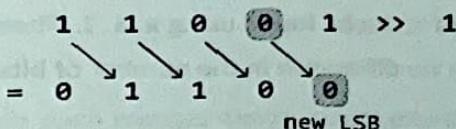
$$\begin{aligned}x &= 25 \quad (\text{base 10}) \\&= 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad (\text{base 2})\end{aligned}$$

To count the number of set bits (1s) in a number, we can check each bit and increase a count whenever we find a set bit. Let's see how this works.

For starters, we can determine the least significant bit (LSB) of x by performing $x \& 1$, which masks all bits of x except the LSB: if $x \& 1 == 1$, the LSB is 1. Otherwise, it's 0. We can see this below for $x = 25$:

$$\begin{array}{r} 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad (x) \\ \& 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad (1) \\ \hline 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad (1) \end{array}$$

Now, how do we check the next bit? If we perform a bitwise right-shift operation on x , we shift all bits of x one position to the right. This effectively makes this next bit the new LSB:



We now have a process we can repeat to count the number of set bits in a number:

1. If $x \& 1 == 1$, increment our count. Otherwise, don't.
2. Right shift x.

Continue with the two steps above until x equals 0, indicating there are no more set bits to count. Doing this for every number from 0 to n provides the answer.

Implementation – Count Bits For Each Number

```
def hamming_weights_of_integers(n: int) -> List[int]:
    return [count_set_bits(x) for x in range(n + 1)]

def count_set_bits(x: int) -> int:
    count = 0
    # Count each set bit of 'x' until 'x' equals 0.
    while x > 0:
        # Increment the count if the LSB is 1.
        count += x & 1
        # Right shift 'x' to shift the next bit to the LSB position.
        x >>= 1
    return count
```

Complexity Analysis

Time complexity: The time complexity of `hamming_weights_of_integers` is $O(n \log(n))$ because for each integer x from 0 to n , counting the number of set bits takes logarithmic time, as there are approximately $\log_2(x)$ bits in that number. If we assume all integers have 32 bits, the time complexity simplifies to just $O(n)$, since counting the set bits for a number will take at most 32 steps, which we do for $n + 1$ numbers.

Space complexity: The space complexity is $O(1)$ because no extra space is used except the space occupied by the output.

Intuition – Dynamic Programming

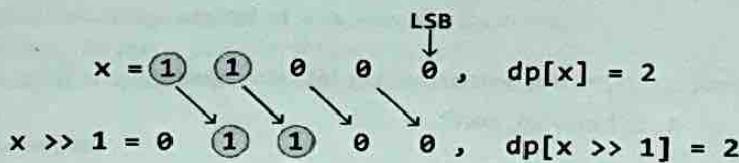
In the previous approach, it's important to note that by the time we reach integer x , we have already computed the result for all integers from 0 to $x - 1$. If we find a way to leverage these previous results, we can improve the efficiency of constructing the output array.

It would be wise to find a way to take advantage of some optimal substructure by treating the results from integers 0 to $x - 1$ as potential subproblems of x . This is the beginning of a DP solution. Let $dp[x]$ represent the number of set bits in integer x .

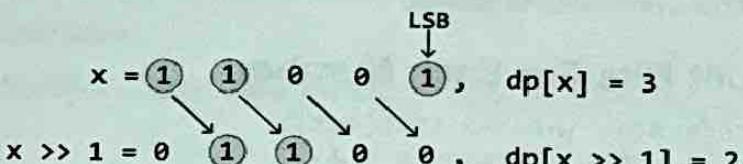
A predictable way to access a subproblem of $dp[x]$ is to right-shift x by 1, effectively removing its LSB. This is the subproblem $dp[x >> 1]$, and the only difference between this and $dp[x]$ is the LSB which was just removed.

As mentioned earlier, the LSB of x can be found using $x \& 1$. Therefore:

- If the LSB of x is 0, there is no difference in the number of bits between x and $x \gg 1$:



- If the LSB of x is 1, the difference in the number of bits between x and $x \gg 1$ is 1:



Therefore, we can obtain $dp[x]$ by using the result of $dp[x \gg 1]$ and adding the LSB to it:

$$dp[x] = dp[x \gg 1] + (x \& 1)$$

Now, we just need to know what our base case is.

Base case

The simplest version of this problem is when n is 0. In this case, there are no set bits, so the number of set bits is 0. We can apply this base case by setting $dp[0]$ to 0.

After the base case is set, we populate the rest of the DP array by applying our formula from $dp[1]$ to $dp[n]$. The answer to the problem is then just the values in the DP array, containing the number of set bits for each number from 0 to n .

Implementation – Dynamic Programming

```
def hamming_weights_of_integers_dp(n: int) -> List[int]:
    # Base case: the number of set bits in 0 is just 0. We set dp[0] to
    # 0 by initializing the entire DP array to 0.
    dp = [0] * (n + 1)
    for x in range(1, n + 1):
        # 'dp[x]' is obtained using the result of 'dp[x >> 1]', plus
        # the LSB of 'x'.
        dp[x] = dp[x >> 1] + (x & 1)
    return dp
```

Complexity Analysis

Time complexity: The time complexity of `hamming_weights_of_integers_dp` is $O(n)$ since we populate each element of the DP array once.

Space complexity: The space complexity is $O(1)$ because no extra space is used, aside from the space taken up by the output, which is the DP array in this case.

Lonely Integer

Given an integer array where each number occurs twice except for one of them, find the unique number.

Example:

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

Output: 2

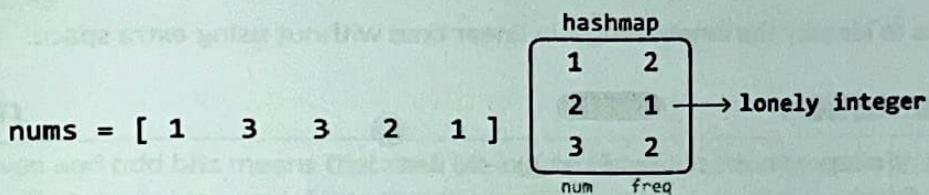
Constraints:

- `nums` contains at least one element.

Intuition

Hash map solution

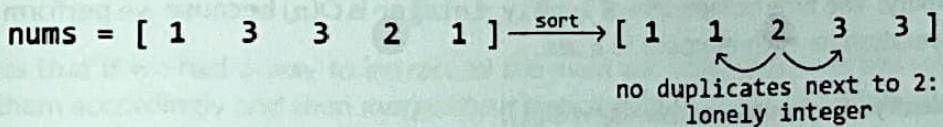
A straightforward way to solve this problem is by using a hash map. The idea is to count the occurrences of each element in the array. We can do this by iterating through the array and increasing the frequency stored in the hash map of each element encountered in the array.



Once populated, we can iterate through the hash map to find the element with a frequency of 1, which is our lonely integer. This approach takes $O(n)$ time, but comes at the cost of $O(n)$ space, where n denotes the length of the input array. Let's see if there's a way to solve this without additional data structures like a hash map.

Sorting solution

Another way to solve this problem is to sort the array first, then look for the lonely integer by iterating through the array, and comparing each element with its neighbors. The lonely integer will be the one that doesn't have a duplicate next to it.



This method takes $O(n \log(n))$ time due to sorting, but has the benefit of not requiring any additional data structures (aside from any used during sorting). Is there a way we can achieve a linear time complexity while also maintaining constant space?

Bit manipulation

A way to avoid using additional space is with bit manipulation. The XOR operation in particular can be useful when handling duplicate integers. Recall the following two characteristics of the XOR operator:

- $a \wedge a == 0$
- $a \wedge 0 == a$

As we can see, when we XOR two identical numbers, the result is 0. As each number except the lonely integer appears twice in the array, if we XOR all the numbers together, all pairs of identical numbers will cancel out to 0. This isolates the lonely integer: once all duplicate elements cancel to 0, XORing 0 with the lonely integer gives us the lonely integer.

This works independently of where the numbers are located in the array, as XOR follows the commutative and associative properties:

- Commutative property: $a \wedge b == b \wedge a$
- Associative property: $(a \wedge b) \wedge c == a \wedge (b \wedge c)$

So, as long as two of the same numbers exist in the array, they will get canceled out when we XOR all the elements. An example of this is shown below:

```
nums = [ 1   3   3   2   1 ]
XOR all elements:  1   ^  3   ^  3   ^  2   ^  1
                    = (1   ^  1) ^ (3   ^  3) ^  2
                    = 0   ^  0   ^  2
                    = 2
```

This allows us to identify the lonely integer in linear time without using extra space.

Implementation

```
def lonely_integer(nums: List[int]) -> int:
    res = 0
    # XOR each element of the array so that duplicate values will
    # cancel each other out ( $x \wedge x == 0$ ).
    for num in nums:
        res ^= num
    # 'res' will store the lonely integer because it would not have
    # been canceled out by any duplicate.
    return res
```

Complexity Analysis

Time complexity: The time complexity of `lonely_integer` is $O(n)$ because we perform a constant-time XOR operation on each element in `nums`.

Space complexity: The space complexity is $O(1)$.

Swap Odd and Even Bits

Given an unsigned 32-bit integer n , return an integer where all of n 's even bits are swapped with their adjacent odd bits.

Example 1:

$\begin{array}{cccccc} 1 & 0 & 1 & 0 & 0 & 1 \\ \curvearrowleft & \curvearrowleft & \curvearrowleft & \curvearrowleft & \curvearrowleft & \curvearrowleft \\ \rightarrow & 0 & 1 & 0 & 1 & 1 & 0 \end{array} \quad (41) \quad (22)$

Input: $n = 41$

Output: 22

Example 2:

$\begin{array}{cccccc} 0 & 1 & 0 & 1 & 1 & 1 \\ \curvearrowleft & \curvearrowleft & \curvearrowleft & \curvearrowleft & \curvearrowleft & \curvearrowleft \\ \rightarrow & 1 & 0 & 1 & 0 & 1 & 1 \end{array} \quad (23) \quad (43)$

Input: $n = 23$

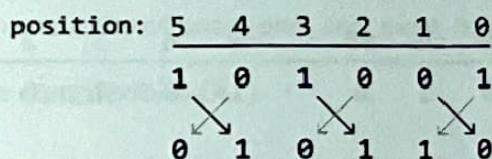
Output: 43

Intuition

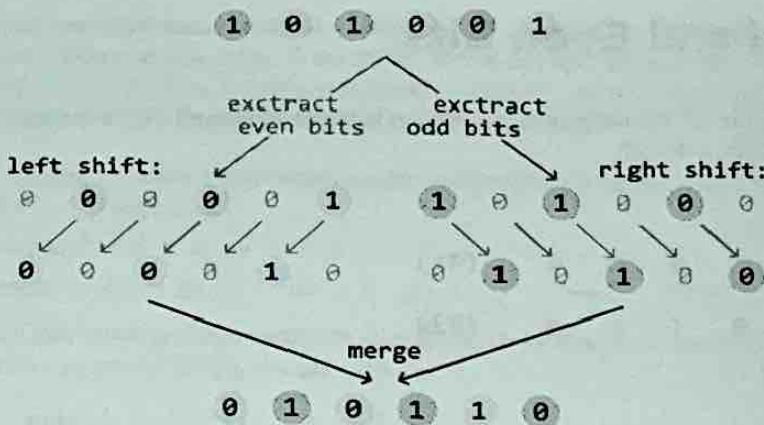
Swapping even and odd bits means that each bit in an even position is swapped with the bit in the next odd position, and vice versa. Note that the positions start at position 0, which is the position of the least significant bit.

The key thing to notice is that, in order to perform the swap:

- All bits in the even positions need to be shifted one position to the left.
- All bits in the odd positions need to be shifted one position to the right.



This suggests that if we had a way to extract all the even and odd-positioned bits separately, we could shift them accordingly and then merge them back together, so the odd-positioned bits are in the even positions, and vice versa.



Let's start by figuring out how to obtain the even and odd bits of n .

Obtaining all even bits

To obtain all even bits of n , we can use a mask which has all even bit positions set to 1:

31	30	29	28	5	4	3	2	1	0
even_mask = 0 1 0 1 ... 0 1 0 1 0 1									

Performing a bitwise-AND with this mask and n gives us an integer where all the bits at odd positions are set to 0, ensuring only the bits in even positions of n are preserved:

0 0 0 0 ... 1 0 1 0 0 1 (n)
AND 0 1 0 1 ... 0 1 0 1 0 1 (even_mask)
0 0 0 0 ... 0 0 0 0 0 1 (even_bits)

Obtaining all odd bits

Similarly, to obtain all the odd bits of n , we can use a mask with all odd bit positions set to 1:

31	30	29	28	5	4	3	2	1	0
odd_mask = 1 0 1 0 ... 1 0 1 0 1 0									

Performing a bitwise-AND with this mask and n gives us an integer where all the bits at even positions are set to 0, ensuring only the bits at odd positions of n are preserved:

0 0 0 0 ... 1 0 1 0 0 1 (n)
AND 1 0 1 0 ... 1 0 1 0 1 0 (odd_mask)
0 0 0 0 ... 1 0 1 0 0 0 (odd_bits)

Now that we've extracted all the even bits and odd bits separately, let's use them to obtain the result, where the bits at odd and even positions are swapped.

Shifting and merging the bits at odd and even positions

We can use the shift operator to shift the bits at even positions to the left once, and the bits at odd positions to the right once:

$$= \begin{array}{ccccccccccccc} 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 1 & \ll 1 \\ \downarrow & \downarrow & \downarrow & \downarrow & \cdots & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \end{array}$$

$$= \begin{array}{ccccccccccccc} 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 1 & 0 & 0 & 0 & \gg 1 \\ \downarrow & \downarrow & \downarrow & \downarrow & \cdots & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \end{array}$$

Then, to merge these together, we can use the bitwise-OR operator because it combines the two sets of bits into the final result.

$$\begin{array}{r} 0 \quad 0 \quad 0 \quad 0 \quad \cdots \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad (\text{shifted even_bits}) \\ \text{OR} \quad 0 \quad 0 \quad 0 \quad 0 \quad \cdots \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad (\text{shifted odd_bits}) \\ \hline 0 \quad 0 \quad 0 \quad 0 \quad \cdots \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \end{array}$$

Now, the odd-positioned bits are in the even positions and vice versa.

Implementation

```
def swap_odd_and_even_bits(n: int) -> int:
    even_mask = 0x55555555 # 010101010101010101010101010101
    odd_mask = 0xAAAAAAAA # 1010101010101010101010101010101
    even_bits = n & even_mask
    odd_bits = n & odd_mask
    # Shift the even bits to the left, the odd bits to the right, and
    # merge these shifted values together.
    return (even_bits << 1) | (odd_bits >> 1)
```

Complexity Analysis

Time complexity: The time complexity of `swap_odd_and_even_bits` is $O(1)$.

Space complexity: The space complexity is $O(1)$.

Chapter Outline

the `~` operator. It's a common mistake to use the `~` operator on a variable instead of its value. For example:

```
int a = 10;
~a // This is wrong!
```

The code above will result in a `Segmentation Fault`. The reason is that the `~` operator is a unary operator that takes a variable as its operand. In this case, the variable `a` is passed by value to the `~` operator, so the operator changes the value of the local variable `a`, which is not what we want. To fix this, we need to use the `&` operator to pass the address of the variable `a` to the `~` operator:

```
int a = 10;
&a // This is correct!
```

Now the `~` operator will change the value of the variable `a` in memory, resulting in the output `-11`.

Math and Geometry

Introduction to Math and Geometry

This chapter tackles several problems relating to important math and geometry concepts in programming, which regularly appear in technical interviews.

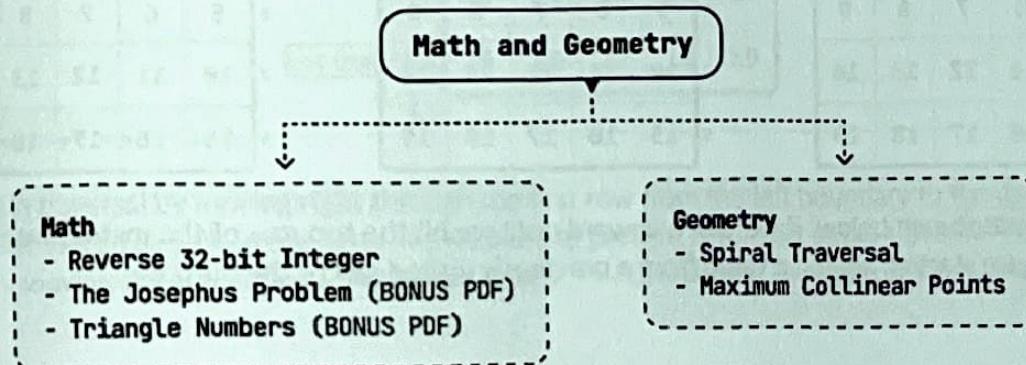
We explore subjects such as:

- Greatest common divisor (GCD).
- Modular arithmetic.
- Handling floating-point precision.
- Handling integer overflow and underflow.
- Recognizing patterns.

Real-world Example

Computer graphics: In 3D rendering, geometry is used to represent objects as shapes like polygons, and mathematical transformations such as rotation, scaling, and translation, are applied to these objects to animate them, or change their perspective. Algorithms that use trigonometry, vector math, and matrix operations, are critical for determining how objects move, interact with light, or cast shadows in virtual environments.

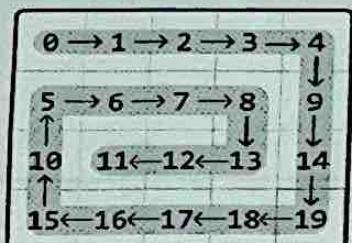
Chapter Outline



Spiral Traversal

Return the elements of a matrix in clockwise spiral order.

Example:



Output: [0, 1, 2, 3, 4, 9, 14, 19, 18, 17, 16, 15, 10, 5, 6, 7, 8, 13, 12, 11]

Intuition

To create the expected output for this problem, let's try simulating exactly what the problem describes and traverse the matrix in spiral order, adding each value to the output as we go. How can we do this?

Spiral traversal involves moving through the matrix in one direction until we can't go any further, then changing direction and continuing. Specifically, the sequence of directions is right, down, left, and up, repeated until all elements are traversed. To achieve this, we need to determine the exact conditions for switching directions.

Initially, our approach may seem simple: we start by moving right until reaching the right-most column of the matrix, at which point we switch directions. We can move and switch directions like this three times without running into any problems:

move right until we reach the rightmost column:

	0	1	2	3	4
0	0 → 1 → 2 → 3 → 4				
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

move down until we reach the bottom row:

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

move left until we reach the leftmost column:

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15 ← 16 ← 17 ← 18 ← 19				

However, as shown below, if we move upward until we hit the top row of the matrix, we'll return to where we started, adding a value from a previously visited cell to the output:

move up until we reach
the top row:

revisited cell → 0

0	1	2	3	4
1	5	6	7	8
2	10	11	12	13
3	15	16	17	18
	19			

A potential solution to this is to keep track of all cells visited by using a hash set. This allows us to stop moving in a direction when we encounter a visited cell. While this approach is effective, it requires $O(m \cdot n)$ space, where m and n are the dimensions of the matrix. This is because we need to store every cell of the matrix in the hash set. Is there a way to avoid revisiting cells without using an additional data structure?

Notice in the above diagrams that when we move in a certain direction, we continue until we reach one of the boundary rows or columns (i.e., the top or bottom row, or the leftmost or rightmost column).

What if we adjust these boundaries as we traverse the matrix, to avoid revisiting previous cells?

Adjusting boundaries

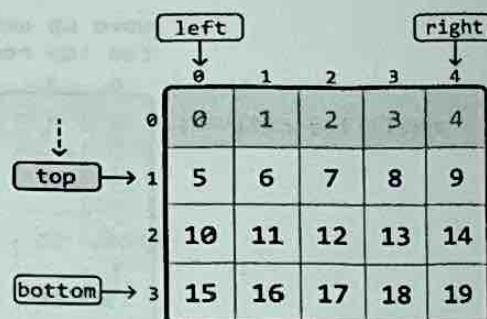
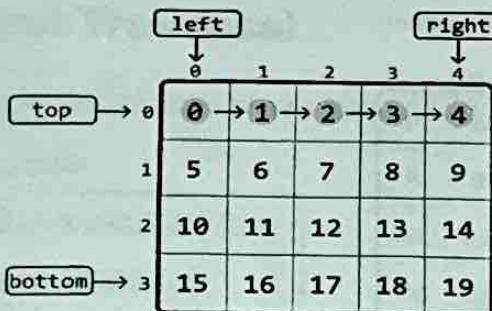
Let's initialize the four boundaries (top, bottom, left, right) with their initial positions:

- $\text{top} = 0$
- $\text{bottom} = m - 1$
- $\text{left} = 0$
- $\text{right} = n - 1$

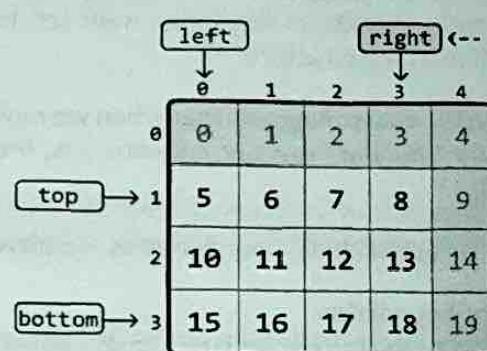
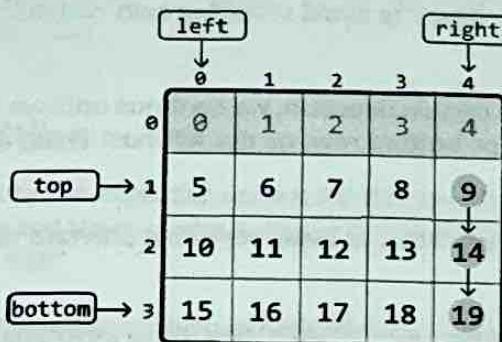
	left		right	
top	→ 0			
		↓		
1				
2				
3				
	0	1	2	3
1	5	6	7	8
2	10	11	12	13
3	15	16	17	18
	19			

We begin traversal by moving **right** through the first row from the left boundary to the right. Since we've just visited all cells in the first row, we need to prevent future access to this row. This can be done by moving the **top** boundary down by 1 ($\text{top} += 1$), ensuring the top row can't be accessed:

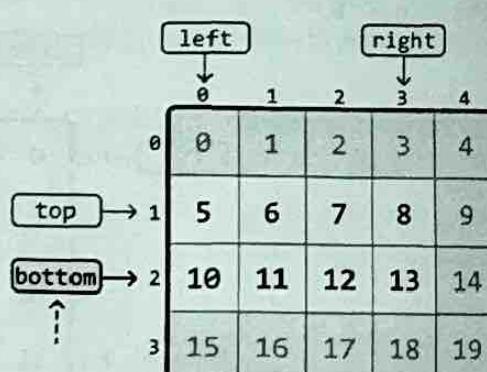
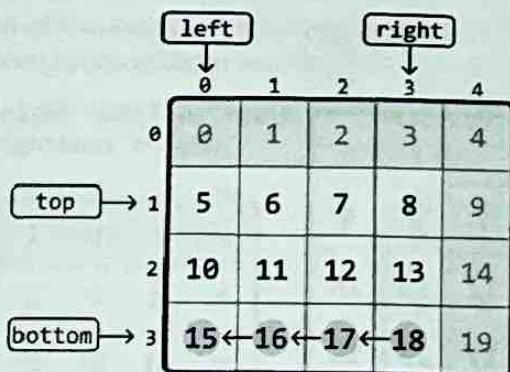
```
int, right = 0, left = 0, top = 0, bottom = 3
for i in range(left, right + 1):
    for j in range(top, bottom + 1):
        print(matrix[i][j], end=" ")
    print()
    top += 1
```



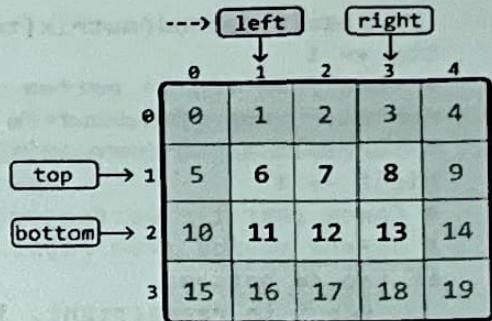
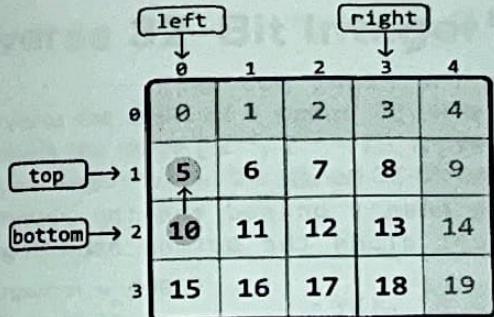
Next, we move down from the top boundary to the bottom boundary. To ensure this column is not revisited, update the right boundary (right -= 1):



Next, we move left from the right boundary to the left. To ensure this row doesn't get revisited, update the bottom boundary (bottom -= 1):



Next, we move up from the bottom boundary to the top boundary. To ensure this column isn't revisited, update the left boundary (left += 1):



We've just discussed how to traverse in each of the four directions and update the corresponding boundaries. These traversals are repeated until either the top boundary surpasses the bottom boundary, or the left boundary surpasses the right boundary. Either of these indicate there are no more cells left to traverse.

In summary, we traverse the matrix in spiral order by repeating the following sequences of traversals:

1. Move from left to right along the top boundary, then update the top boundary (`top += 1`).
2. Move from top to bottom along the right boundary, then update the right boundary (`right -= 1`).
3. Move from right to left along the bottom boundary, then update the bottom boundary (`bottom -= 1`).
4. Move from bottom to top along the left boundary, then update the left boundary (`left += 1`).

This continues while `top <= bottom` and `left <= right`.

A crucial thing to keep in mind is that after updating the top boundary, the top boundary might pass the bottom boundary (`top > bottom`). So, we need to check that `top <= bottom` before traversing the bottom boundary. Similarly, we need to check that `left <= right` before traversing the left boundary to ensure the boundaries haven't crossed.

As we move through the matrix, we add each value we encounter to the output array. This way, the matrix values are recorded in a spiral order.

Implementation

```
def spiral_matrix(matrix: List[List[int]]) -> List[int]:
    if not matrix:
        return []
    result = []
    # Initialize the matrix boundaries.
    top, bottom = 0, len(matrix) - 1
    left, right = 0, len(matrix[0]) - 1
    # Traverse the matrix in spiral order.
    while top <= bottom and left <= right:
        # Move from left to right along the top boundary.
        for i in range(left, right + 1):
```

```

        result.append(matrix[top][i])
    top += 1
    # Move from top to bottom along the right boundary.
    for i in range(top, bottom + 1):
        result.append(matrix[i][right])
    right -= 1
    # Check that the bottom boundary hasn't passed the top boundary
    # before moving from right to left along the bottom boundary.
    if top <= bottom:
        for i in range(right, left - 1, -1):
            result.append(matrix[bottom][i])
        bottom -= 1
    # Check that the left boundary hasn't passed the right boundary
    # before moving from bottom to top along the left boundary.
    if left <= right:
        for i in range(bottom, top - 1, -1):
            result.append(matrix[i][left])
        left += 1
    return result

```

Complexity Analysis

Time complexity: The time complexity of `spiral_matrix` is $O(m \cdot n)$ because we traverse each cell of the matrix once.

Space complexity: The space complexity is $O(1)$. The `res` array is not included in the space complexity.

Reverse 32-Bit Integer

Reverse the digits of a signed 32-bit integer. If the reversed integer overflows (i.e., is outside the range $[-2^{31}, 2^{31} - 1]$), return 0. Assume the environment only allows you to store integers within the signed 32-bit integer range.

Example 1:

Input: n = 420
Output: 24

Example 2:

Input: n = -15
Output: -51

Intuition

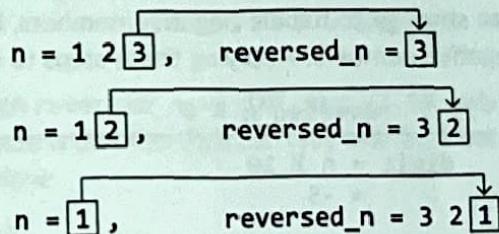
The primary challenge with this problem is in handling its edge cases. Before tackling these edge cases, let's first try handling the more basic cases and later see how we would need to modify our strategy.

Reversing positive numbers

Consider $n = 123$. Let's try building the reversed integer one digit at a time. The first thing to figure out is how to iterate through the digits of n to build our reversed number (initially set to 0):

$n = 1 \ 2 \ 3$, $\text{reversed_n} = 0$

One way to do this is by starting at the last digit of n and appending each digit to reversed_n :



Let's explore how we can do this. To extract the last digit, we can use the modulus operator: $n \% 10$. This operation effectively finds what the remainder of n would be if divided by 10:

```
n = 123
digit = n % 10
      = 3
```

After extracting the last digit, we can remove it by dividing n by 10, which shifts the second-to-last digit to the last position, preparing it for the next iteration:

```
n = n // 10
    = 12
```

Once that's done, let's add the last digit extracted to our reversed number:

```
reversed_n += digit  
= 3
```

Below, we see the current states of n and reversed_n:

```
n = 1 2 , reversed_n = 3
```

To process the next digit, let's extract it from n using the modulus operation, then remove it by dividing n by 10:

```
digit = n % 10  
= 2  
n = n // 10  
= 1
```

To append this digit to reversed_n, we can multiply reversed_n by 10 to shift its digits to the left, making space for the new digit. Then, we just add the new digit as before:

```
reversed_n = 10 * reversed_n + digit  
= 30 + 2  
= 32
```

We can repeat the above process until all digits of n are appended to reversed_n (i.e., until n equals 0). Here's a breakdown of this process:

1. Extract the last digit: $\text{digit} = \text{n} \% 10$.
2. Remove the last digit: $\text{n} = \text{n} // 10$.
3. Append the digit: $\text{reversed_n} = \text{reversed_n} * 10 + \text{digit}$.

Reversing negative numbers

Before considering a separate strategy to handle negative numbers, let's first check if the set of steps above also work for negative numbers. Applying these steps to $\text{n} = -15$ gives:

```
n = -15 , reversed_n = 0  
digit = n % 10  
= -5  
n = n // 10  
= -1  
reversed_n = 10 * reversed_n + digit  
= -5
```

```
n = -1 , reversed_n = -5  
digit = n % 10  
= -1  
n = n // 10  
= 0  
reversed_n = 10 * reversed_n + digit  
= -51
```

As we can see, it works for negative numbers. Now, let's tackle situations in which reversing a number could result in integer overflow or underflow.

Detecting integer overflow

If the reverse of a positive number is larger than $2^{31} - 1$, it will overflow, and we should return 0. Let's call this maximum value INT_MAX.

$$\text{INT_MAX} = 2^{31} - 1 = 2147483647$$

Initially, it might seem sufficient to reverse the number completely, check if it exceeds $2^{31} - 1$, and return 0 if it does. However, in an environment where integers larger than $2^{31} - 1$ cannot be stored, attempting to reverse such an integer would cause an overflow:

$$1999999999 \xrightarrow{\text{reverse}} 9999999991 > \text{INT_MAX} \longrightarrow \text{overflow}$$

So, let's think of another way to detect overflow.

We're constructing the number reversed_n one digit at a time, which means we need to ensure not to cause the number to overflow with each new digit we add. Let's think about when adding a new digit might cause reversed_n to become too large. Here's how we can analyze this:

If reversed_n is equal to 214748364 (i.e., INT_MAX // 10), then the final digit we can add to it must be less than or equal to 7 to avoid an overflow (since 2147483647 == INT_MAX):

$$\begin{aligned} \text{reversed_n} &= 21478364, \quad \text{digit} = 7 \\ \text{append digit: } \text{reversed_n} &= \text{reversed_n} * 10 + \text{digit} \\ &= 214783647 == \text{INT_MAX} \longrightarrow \text{no overflow} \end{aligned}$$

$$\begin{aligned} \text{reversed_n} &= 21478364, \quad \text{digit} = 8 \\ \text{append digit: } \text{reversed_n} &= \text{reversed_n} * 10 + \text{digit} \\ &= 214783648 > \text{INT_MAX} \longrightarrow \text{overflow} \end{aligned}$$

Now, keep in mind that when reversed_n == INT_MAX // 10, only one more digit can be added to it. The key observation here is that this digit can only ever be 1 because a larger final digit would be impossible, as shown below:

$$\begin{aligned} \text{INT_MAX} // 10 &\quad \uparrow \\ \text{reversed_n} &= \boxed{214748364} \xrightarrow{\text{implies}} n = \underline{1}463847412 \\ \text{INT_MAX} // 10 &\quad \uparrow \\ \text{reversed_n} &= \boxed{214748364} \xrightarrow{\text{implies}} n = \underline{2}463847412 > \text{INT_MAX} \rightarrow \text{impossible input} \end{aligned}$$

This means that when reversed_n == INT_MAX // 10, the last digit added to it won't cause an overflow, meaning we don't need to check the last digit in this case.

If reversed_n is already larger than INT_MAX // 10, adding any digit will cause it to overflow. We can handle this case with the following condition:

```
if reversed_n > INT_MAX // 10:  
    return 0
```

Detecting integer underflow

We can apply similar logic to the above for handling integer underflow. Here, we just need to check that `reversed_n` never falls below `INT_MIN // 10`:

```
| if reversed_n < INT_MIN // 10:  
|     return 0
```

Implementation

In Python, using the modulus operator (%) with a negative number gives a positive result. To avoid this, we can instead use `math.fmod` and cast its result to an integer to attain a negative modulus value.

For division, Python's `//` operator performs floor division, which can result in an undesired value when dealing with negative numbers (e.g., `-15 // 10` results in `-2`, instead of the desired `-1`). To achieve the desired behavior, use `/` for division and cast its result to an integer.

```
def reverse_32_bit_integer(n: int) -> int:  
    INT_MAX = 2**31 - 1  
    INT_MIN = -2**31  
    reversed_n = 0  
    # Keep looping until we've added all digits of 'n' to 'reversed_n'  
    # in reverse order.  
    while n != 0:  
        # digit = n % 10  
        digit = int(math.fmod(n, 10))  
        # n = n // 10  
        n = int(n / 10)  
        # Check for integer overflow or underflow.  
        if (reversed_n > int(INT_MAX / 10)  
            or reversed_n < int(INT_MIN / 10)):  
            return 0  
        # Add the current digit to 'reversed_n'.  
        reversed_n = reversed_n * 10 + digit  
    return reversed_n
```

Complexity Analysis

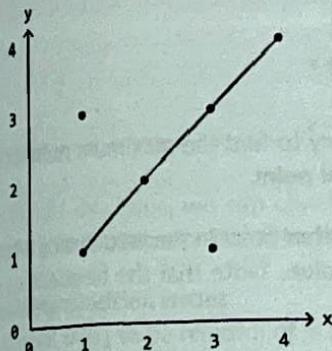
Time complexity: The time complexity of `reverse_32_bit_integer` is $O(\log(n))$ because we loop through each digit of n , of which there are roughly $\log(n)$ digits. As this environment only supports 32-bit integers, the time complexity can also be considered $O(1)$.

Space complexity: The space complexity is $O(1)$.

Maximum Collinear Points

Given a set of points in a two-dimensional plane, determine the maximum number of points that lie along the same straight line.

Example:



Input: points = [[1, 1], [1, 3], [2, 2], [3, 1], [3, 3], [4, 4]]

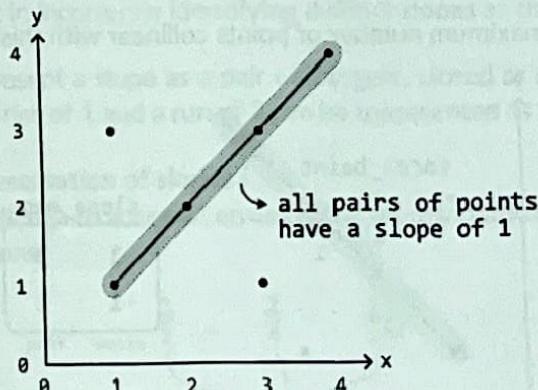
Output: 4

Constraints:

- The input won't contain duplicate points.

Intuition

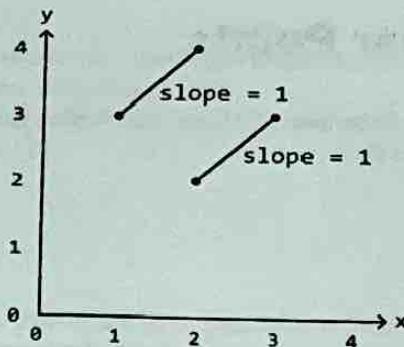
Two or more points are collinear if they lie on the same straight line. In other words, the slope between any pair of points among them will be equal:



Let's remind ourselves how the slope of a line is calculated given two points (x_a, y_a) and (x_b, y_b) :

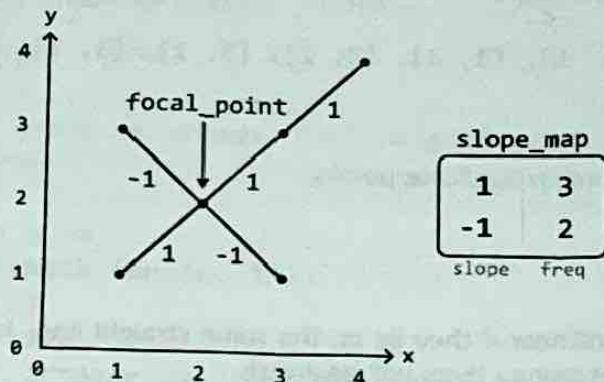
$$\text{slope} = \frac{\text{rise}}{\text{run}} = \frac{y_b - y_a}{x_b - x_a}$$

Using this formula, we can calculate the slope between all pairs of points from the input, and determine the largest number of pairs that share the same slope. However, this approach is flawed because two pairs of points with the same slope value may not be collinear:

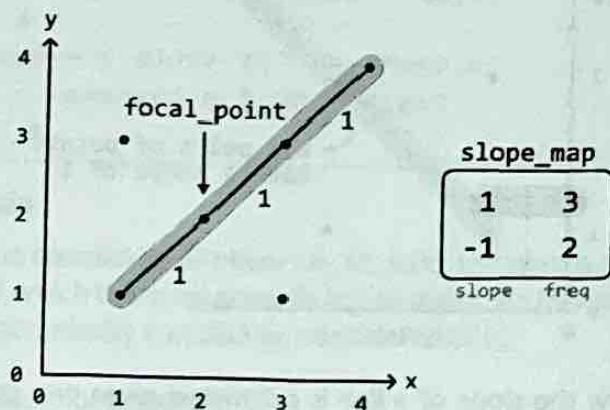


Let's think of a different way to find the answer. What if we try to find the maximum number of points collinear with a specific point? Let's call this point a **focal point**.

We can calculate the slope between the focal point and every other point in the input, using a hash map to count how many points correspond with each slope value. Note that the frequencies of points stored in the hash map do not include the focal point.



This allows us to find the maximum number of points collinear with this focal point:



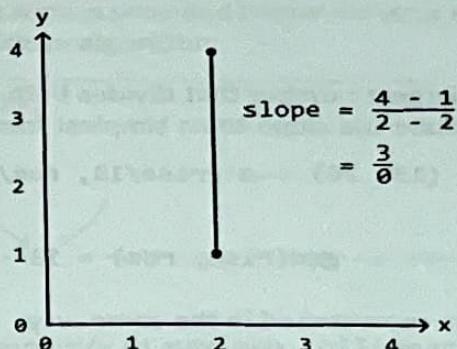
Here, the slope with the highest frequency is 3, which means the number of points on the line defined by that slope is $3 + 1 = 4$, where +1 is used to account for the focal point itself.

By repeating the process for every point, we can determine the maximum number of points that are collinear with each focal point. Our final answer is equal to the largest of these maximums.

Edge case: two points on the same x-axis

When two collinear points share the same x-axis, the denominator of the slope equation will equal

0 (i.e., $\text{run} = x_b - x_a = 0$). This is problematic because dividing by 0 is undefined:



To handle this issue, we can check if our run value is equal to 0 . If so, we can just use infinity (`float('inf')`) to represent the value of this slope.

Avoiding precision issues

A crucial thing to be mindful of is the precision of slopes when storing them as floats or doubles. Consider the following slopes:

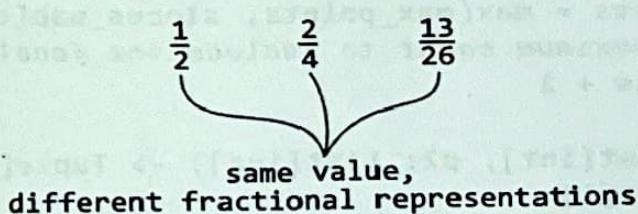
$$\begin{aligned}\text{slope1} &= \frac{\text{rise}_1}{\text{run}_1} = \frac{499999999}{500000000} \xrightarrow{\text{float precision}} 0.999999998 \\ &\quad \begin{array}{c} \uparrow \\ \text{not equal} \\ \downarrow \end{array} \qquad \qquad \qquad \begin{array}{c} \uparrow \\ \text{equal} \\ \downarrow \end{array} \\ \text{slope2} &= \frac{\text{rise}_2}{\text{run}_2} = \frac{499999998}{499999999} \xrightarrow{\text{float precision}} 0.999999998\end{aligned}$$

As we can see, despite the fractions themselves representing different slopes, presenting them as a float or double doesn't provide enough decimal-point precision to distinguish between them accurately. This can result in incorrectly identifying distinct slopes as the same.

To avoid this, we can represent a slope as a pair of integers, stored as a tuple: `(rise, run)`. For example, the slope with a rise of 1 and a run of 2 can be represented as `(1, 2)` instead of $1 / 2 = 0.5$.

Ensuring consistent representation of slopes

There's just one more challenge to address: ensuring the representation of a slope is consistent for all equivalent slope fractions:



If we reduce fractions to their simplest forms, we can consistently represent equal fractions that have different initial representations.

$$\begin{array}{ccc}\frac{1}{2} & \frac{2}{4} & \frac{13}{26} \\ \downarrow & \downarrow & \downarrow \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2}\end{array}$$

But how can we do this? We can reduce fractions by dividing both the rise and run by their greatest common divisor (GCD).

Greatest common divisor

The GCD of two numbers is the largest number that divides both of them exactly. By dividing the rise and run by their GCD, we reduce the slope to its simplest form:

$$\begin{aligned} (\text{rise}, \text{run}) &= (13, 26) \longrightarrow (\text{rise}/13, \text{run}/13) = (1, 2) \\ \text{gcd}(\text{rise}, \text{run}) &= 13 \end{aligned}$$

This ensures all equal fractions are represented in the same way.

Implementation

```
def maximum_collinear_points(points: List[List[int]]) -> int:
    res = 0
    # Treat each point as a focal point, and determine the maximum
    # number of points that are collinear with each focal point. The
    # largest of these maximums is the answer.
    for i in range(len(points)):
        res = max(res, max_points_from_focal_point(i, points))
    return res

def max_points_from_focal_point(focal_point_index: int,
                                 points: List[List[int]]) -> int:
    slopes_map = defaultdict(int)
    max_points = 0
    # For the current focal point, calculate the slope between it and
    # every other point. This allows us to group points that share the
    # same slope.
    for j in range(len(points)):
        if j != focal_point_index:
            curr_slope = get_slope(points[focal_point_index], points[j])
            slopes_map[curr_slope] += 1
        # Update the maximum count of collinear points for the
        # current focal point.
        max_points = max(max_points, slopes_map[curr_slope])
    # Add 1 to the maximum count to include the focal point itself.
    return max_points + 1

def get_slope(p1: List[int], p2: List[int]) -> Tuple[int, int]:
    rise = p2[1] - p1[1]
    run = p2[0] - p1[0]
    # Handle vertical lines separately to avoid dividing by 0.
    if run == 0:
        return (1, 0)
    # Simplify the slope to its reduced form.
    gcd_val = gcd(rise, run)
    return (rise // gcd_val, run // gcd_val)
```

While some programming languages, Python included, have their own internal implementation of the GCD function, its implementation is provided below for your information. This implementation is commonly known as the Euclidean algorithm:

```
# The Euclidean algorithm.  
def gcd(a, b):  
    while b != 0:  
        a, b = b, a % b  
    return a
```

Complexity Analysis

Time complexity: The time complexity of `maximum_collinear_points` is $O(n^2 \log(m))$, where n denotes the number of points, and m denotes the largest value among the coordinates. Here's why:

- The time complexity of the `gcd(rise, run)` function is $O(\log(\min(rise, run)))$, which is approximately equal to $O(\log(m))$ in the worst case.
- The helper function `max_points_from_focal_point`, calls the `gcd` function a total of $n - 1$ times: one for each point excluding the focal point, giving a time complexity of $O(n \log(m))$.
- The `max_points_from_focal_point` function is called a total of n times, resulting in an overall time complexity of $O(n^2 \log(m))$.

Space complexity: The space complexity is $O(n)$ due to the hash map, which in the worst case, stores $n - 1$ key-value pairs: one for each point excluding the focal point.

Afterword

Congratulations on completing this interview guide! You've built a solid foundation of skills and knowledge to tackle coding interviews with confidence. Not everyone has the discipline to put in the time and effort you've invested, so take a moment to recognize your accomplishment. Your hard work will pay off.

Landing your dream job is a journey that takes time, persistence, and consistent practice. Keep honing your skills, and best of luck on your path ahead!

Thank you for choosing and reading this book. It's readers like you who make our work meaningful, and we hope you enjoyed the journey!

If you have comments or questions about this book, feel free to contact us at hi@bytebytego.com. If you notice any errors, please let us know so we can make the necessary corrections for the next edition. Thank you!