

Coding Interview Patterns

BONUS PDF

Alex Xu | Shaun Gunawardane



Coding Interview Patterns

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

All rights reserved. This PDF 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.

Join the community

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

- Interview and coding pattern discussions.
- 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



Contents

Two Pointers.....	4
Shift Zeros to the End.....	4
Next Lexicographical Sequence.....	9
Hash Maps and Sets.....	16
Longest Chain of Consecutive Numbers.....	16
Geometric Sequence Triplets.....	20
Linked Lists.....	27
Palindromic Linked List.....	27
Flatten a Multi-Level Linked List.....	31
Fast and Slow Pointers.....	36
Happy Number Time Complexity Analysis.....	36
Binary Search.....	39
Find the Median From Two Sorted Arrays.....	39
Matrix Search.....	46
Local Maxima in Array.....	52
Weighted Random Selection.....	58
Stacks.....	65
Repeated Removal of Adjacent Duplicates.....	65
Implement a Queue Using Stacks.....	68
Maximums of Sliding Window.....	74
Heaps.....	81
Sort a K-Sorted Array.....	81
Trees.....	88
Binary Tree Symmetry.....	88
Binary Tree Columns.....	92
Kth Smallest Number in a Binary Search Tree.....	97
Serialize and Deserialize a Binary Tree.....	102

Graphs.....	109
Shortest Path.....	109
Connect the Dots.....	117
Backtracking.....	124
Combinations of a Sum.....	124
Phone Keypad Combinations.....	129
Dynamic Programming.....	133
Largest Square in a Matrix.....	133
Sort and Search.....	141
Dutch National Flag.....	141
Math and Geometry.....	146
The Josephus Problem.....	146
Triangle Numbers.....	150

Two Pointers

Shift Zeros to the End

Given an array of integers, modify the array in place to move all zeros to the end while maintaining the relative order of non-zero elements.

Example:

Input: `nums = [0, 1, 0, 3, 2]`

Output: `[1, 3, 2, 0, 0]`

Intuition

This problem has three main requirements:

1. Move all zeros to the end of the array.
2. Maintain the relative order of the non-zero elements.
3. Perform the modification in place.

A naive approach to this problem is to build the output using a separate array (`temp`). We can add all non-zero elements from the left of `nums` to this temporary array and leave the rest of it as zeros. Then, we just set the input array equal to `temp`.

By identifying and moving the non-zero elements from the left side of the array first, we ensure their order is preserved when we add them to the output:

```
def shift_zeros_to_the_end_naive(nums: List[int]) -> None:
    temp = [0] * len(nums)
    i = 0
    # Add all non-zero elements to the left of 'temp'.
    for num in nums:
```

```

if num != 0:
    temp[i] = num
    i += 1
# Set 'nums' to 'temp'.
for j in range(len(nums)):
    nums[j] = temp[j]

```

Unfortunately, this solution violates the third requirement of modifying the input array in place.

However, there's still valuable insight to be gained from this approach. In particular, notice that this solution focuses on the non-zero elements instead of zeros. This means if we change our goal to **move all non-zero elements to the left of the array**, the zeros will consequently end up on the right. Therefore, we only need to focus on non-zero elements:

$[\underbrace{0 \quad 1 \quad 0}_{\text{non-zero numbers go here}} \quad 2 \quad 3] \longrightarrow [\underbrace{1 \quad 2 \quad 3} \quad 0 \quad 0]$

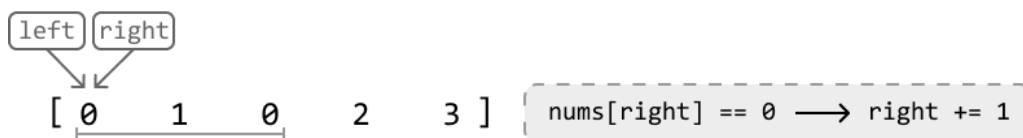
If there was a way to iterate over the above range of the array where the non-zero elements go, we could iteratively place each non-zero element in that range.

Two pointers

We can use two pointers for this:

- A left pointer to iterate over the left of the array where the non-zero elements should be placed.
- A right pointer to find non-zero elements.

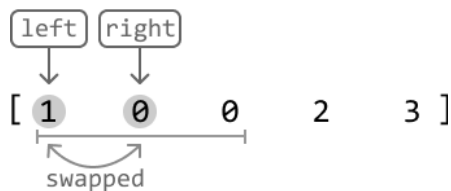
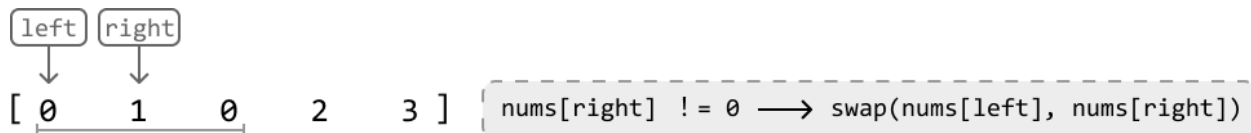
Consider the example below. Start by placing the left and right pointers at the start of the array. Before we move non-zero elements to the left, we need the right pointer to be pointing at a non-zero element. So, we ignore the zero at the first element and increment right:



Remember, we only use the left pointer to keep track of where non-zero elements should be placed. So, until we find a non-zero element, we shouldn't move this pointer.

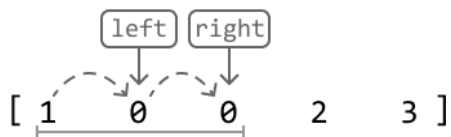
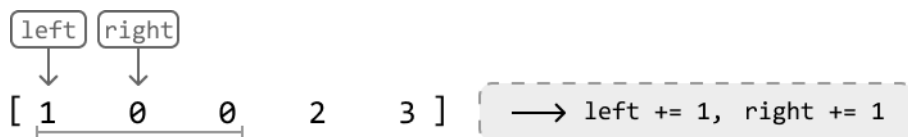
Now, the value at the right pointer is non-zero. Let's discuss how to handle this case.

1. Swap the elements at left and right: First, we'd like to move the element at the right pointer to the left of the array. So, we swap it with the element at the left pointer.

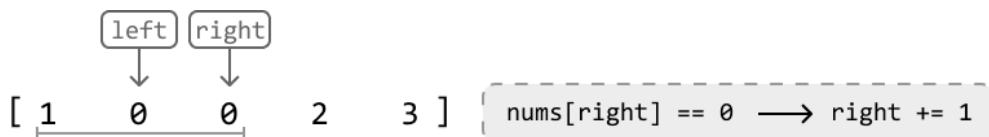


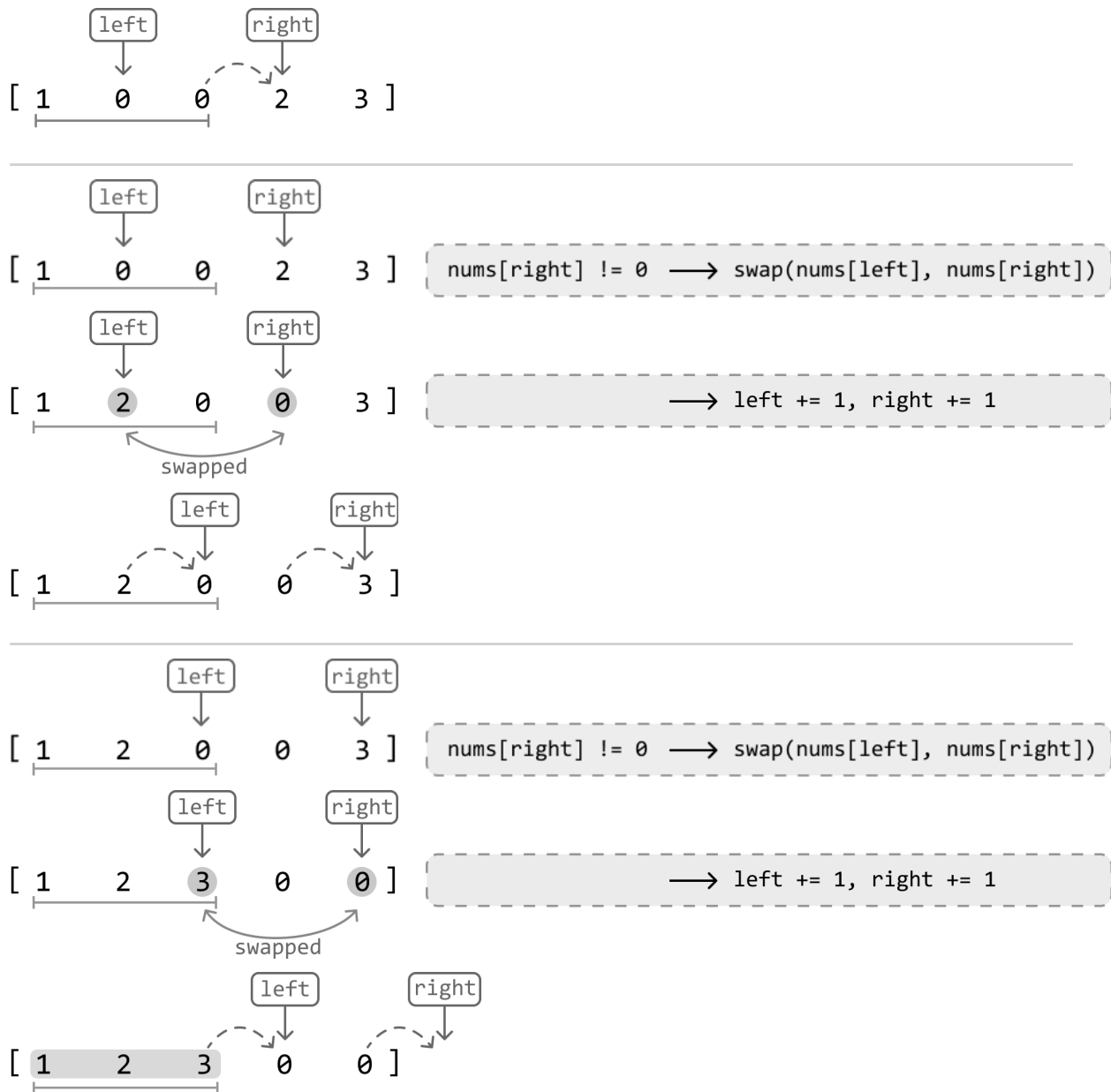
2. Increment the pointers:

- After the swap is complete, we should increment the left pointer to position it where the next non-zero element should go.
- Let's also increment the right pointer in search of the next non-zero element.



We can apply this logic to the rest of the array, incrementing the right pointer at each step to find the next non-zero element:





Once all swapping is done, all zeros will end up at the right end of the array as intended, without disturbing the order of the non-zero elements. The two-pointer strategy used in this problem is unidirectional traversal.

Implementation

You might have noticed that we always move the right pointer forward, regardless of whether it points to a zero or a non-zero. This allows us to use a for-loop to iterate the right pointer.

```
def shift_zeros_to_the_end(nums: List[int]) -> None:
```



```

# The 'left' pointer is used to position non-zero elements.
left = 0
# Iterate through the array using a 'right' pointer to locate non-zero
# elements.
for right in range(len(nums)):
    if nums[right] != 0:
        nums[left], nums[right] = nums[right], nums[left]
        # Increment 'left' since it now points to a position already occupied
        # by a non-zero element.
        left += 1

```

Complexity Analysis

Time complexity: The time complexity of `shift_zeros_to_the_end` is $O(n)$, where n denotes the length of the array. This is because we iterate through the input array once.

Space complexity: The space complexity is $O(1)$ because shifting is done in place.

Test Cases

In addition to the examples discussed, below are more examples 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 an array with one 0.
<code>nums = [1]</code>	<code>[1]</code>	Tests an array with one 1.
<code>nums = [0, 0, 0]</code>	<code>[0, 0, 0]</code>	Tests an array with all 0s.
<code>nums = [1, 3, 2]</code>	<code>[1, 3, 2]</code>	Tests an array with all non-zeros.
<code>nums = [1, 1, 1, 0, 0]</code>	<code>[1, 1, 1, 0, 0]</code>	Tests an array with all zeros already at the end.
<code>nums = [0, 0, 1, 1, 1]</code>	<code>[1, 1, 1, 0, 0]</code>	Tests an array with all zeros at the start.

Next Lexicographical Sequence

Given a string of lowercase English letters, rearrange the characters to form a new string representing the next immediate sequence in lexicographical (alphabetical) order. If the given string is already last in lexicographical order among all possible arrangements, return the arrangement that's first in lexicographical order.

Example 1:

Input: `s = "abcd"`

Output: `"abdc"`

Explanation: `"abdc"` is the next sequence in lexicographical order after rearranging `"abcd"`.

Example 2:

Input: `s = "dcba"`

Output: `"abcd"`

Explanation: Since `"dcba"` is the last sequence in lexicographical order, we return the first sequence: `"abcd"`.

Constraints:

- The string contains at least one character.

Intuition

Before devising a solution, let's first make sure we understand what the next lexicographical sequence of a string is.

It's useful to think about the next lexicographical sequence as the first string that's lexicographically *larger* than the original string. Consider the string `"abc"` and all its permutations in a lexicographically ordered sequence:

a	b	c
a	c	b
b	a	c
b	c	a
c	a	b
c	b	a

lexicographical
order

↓

We can see the increasing order of strings in the sequence when we translate each letter to its position in the alphabet:

a	b	c	→	1	2	3	
a	c	b	→	1	3	2	
b	a	c	→	2	1	3	
b	c	a	→	2	3	1	
c	a	b	→	3	1	2	
c	b	a	→	3	2	1	↓

increasing

From this, we also notice the next string in the sequence after “abc” is “acb”, which is the first string larger than the original string:

a	b	c	→	1	2	3	
a	c	b	→	1	3	2	← next largest after "abc"
b	a	c	→	2	1	3	
b	c	a	→	2	3	1	
c	a	b	→	3	1	2	
c	b	a	→	3	2	1	

This gives us some indication of what we need to find. The next lexicographical string:

1. Incurs the **smallest possible lexicographical increase** from the original string.
2. Uses the same letters as the original string.

Identifying which characters to rearrange

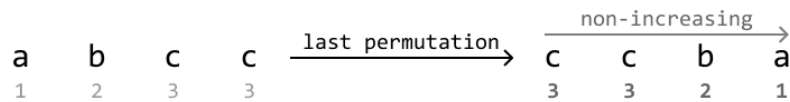
Since our goal is to make the smallest possible increase, we need to somehow rearrange the characters on the right side of the string.

To understand why, imagine trying to “increase” a string’s value. Increasing the rightmost letter results in a smaller increase than increasing the leftmost letter.

a	b	c	e	d	d	a ^b	(small increase)
1	2	3	5	4	4	2	
a ^b	b	c	e	d	d	a	(big increase)
2	2	3	5	4	4	1	

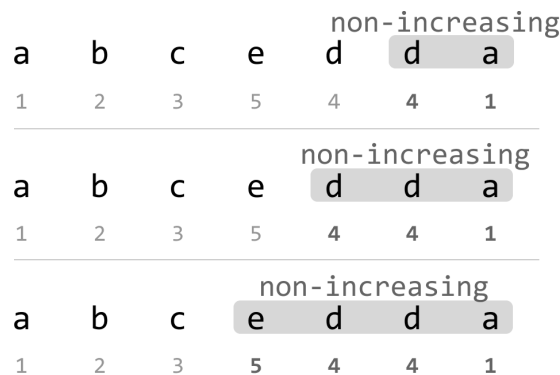
Therefore, we should **focus on rearranging characters on the right-hand side of the string first**, if possible.

A key insight is that the last string in a lexicographical sequence (i.e., the largest permutation) will always follow a **non-increasing** order. We can see this with the string "abcc", for example, with its largest possible permutation being "ccba":

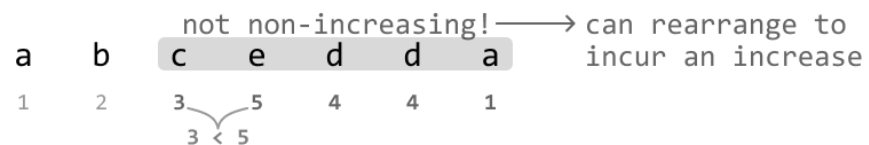


How does this help us? We know we need to rearrange the characters on the right of the string, but we don't know how many to rearrange. From now on, let's refer to the rightmost characters that should be rearranged as the **suffix**.

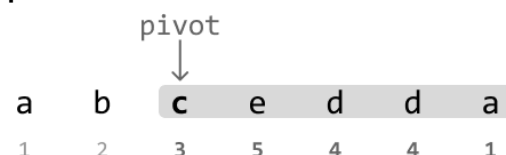
Take the string "abcedda" as an example. We traverse it from right to left with the **goal of finding the shortest suffix that can be rearranged to form a larger permutation**. The last 4 characters form a non-increasing suffix, and cannot be rearranged to make the string larger:



However, the next character, 'c', breaks the non-increasing sequence:



Let's call this character the **pivot**:



If no pivot is found, it means the string is already the last lexicographical sequence. In this case, we need to obtain its first lexicographical permutation as the problem states. This can be done by reversing the string:

d c b a reverse → a b c d

Rearranging characters

Having identified the shortest suffix to rearrange, the next objective is to rearrange this suffix to make the **smallest increase possible**. We have to start the rearrangement at the pivot since the rest of the suffix is already arranged in its largest permutation.

To make the character at the pivot position larger, we'd need to **swap the pivot with a character larger than it on the right**.

In our example, which character should our pivot ('c') swap with? We want to swap it with a character larger than 'c', but not *too* much larger because the increase should be as small as possible. Let's figure out how we find such a character.

Since the substring after the pivot is lexicographically non-increasing, we can find the closest character larger than 'c' by **traversing this suffix from right to left and stopping at the first character larger than it**. In other words, we're finding the rightmost successor to the pivot, which is 'd':

pivot
rightmost
successor
↓
↓
a b c e d d a
1 2 3 5 4 4 1

Now, we swap the pivot and the rightmost successor:

a b c e d d a → a b d e d c a
1 2 3 5 4 4 1 1 2 4 5 4 3 1

The character at the pivot has increased, so to get the next permutation, **we should make the substring after the pivot as small as possible**.

An important observation is that after the previous swap, the substring after the pivot is still lexicographically non-increasing.

pivot
still non-increasing
→ make this as
↓
↓
small as possible.
a b d e d c a
1 2 4 5 4 4 1

This means we can minimize this substring's permutation by reversing it:

a	b	d	e	d	c	a	<u>reverse</u> →	a	b	d	a	c	d	e
1	2	3	5	4	3	1		1	2	4	1	3	4	5

And just like that, we found the next lexicographical sequence! The two-pointer strategy used in this problem is staged traversal, where we first identify the pivot, and then identify the rightmost successor relative to it.

Many steps are involved in identifying the next lexicographical sequence. So, here's a summary:

1. Locate the pivot.
 - The pivot is the first character that breaks the non-increasing sequence from the right of the string.
 - If no pivot is found, the string is already at its last lexicographical sequence, and the result is just the reverse of the string.
2. Find the rightmost successor to the pivot.
3. Swap the rightmost successor with the pivot to increase the lexicographical order of the suffix.
4. Reverse the suffix after the pivot to minimize its permutation.

Implementation

```
def next_lexicographical_sequence(s: str) -> str:
    letters = list(s)
    # Locate the pivot, which is the first character from the right that breaks
    # non-increasing order. Start searching from the second-to-last position,
    # since the last character is neither increasing nor decreasing.
    pivot = len(letters) - 2
    while pivot >= 0 and letters[pivot] >= letters[pivot + 1]:
        pivot -= 1
    # If pivot is not found, the string is already in its largest permutation. In
    # this case, reverse the string to obtain the smallest permutation.
    if pivot == -1:
        return ''.join(reversed(letters))
    # Find the rightmost successor to the pivot.
    rightmost_successor = len(letters) - 1
    while letters[rightmost_successor] <= letters[pivot]:
```

```

    rightmost_successor -= 1
    # Swap the rightmost successor with the pivot to increase the lexicographical
    # order of the suffix.
    letters[pivot], letters[rightmost_successor] = (letters[rightmost_successor],
                                                    letters[pivot])

    # Reverse the suffix after the pivot to minimize its permutation.
    letters[pivot + 1:] = reversed(letters[pivot + 1:])
    return ''.join(letters)

```

Complexity Analysis

Time complexity: The time complexity of `next_lexicographical_sequence` is $O(n)$, where n denotes the length of the input string. This is because we perform a maximum of two iterations across the string: one to find the pivot and another to find the rightmost character in the suffix that's greater in value than the pivot. We also perform one reversal (either at the end or if no pivot is found), which takes $O(n)$ time.

Space complexity: The space complexity is $O(n)$ due to the space taken up by the `letters` list. In Python, this list is created because strings are immutable, which necessitates storing the input string as a list. Note, the final output string is not considered in the space complexity.

Test Cases

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

Input	Expected output	Description
<code>s = 'a'</code>	<code>'a'</code>	Tests a string with a single character.
<code>s = 'aaaa'</code>	<code>'aaaa'</code>	Tests a string with a repeated character.
<code>s = 'ynitsed'</code>	<code>'ynsdeit'</code>	Tests a string with a random pivot character.

Interview Tip

Tip: Be precise with your language.



It's crucial to be precise with your choice of words during an interview, especially for technical descriptions. For instance, in this problem, we use “non-increasing” instead of “decreasing,” as “decreasing” implies each term is strictly smaller than the previous one, which isn't true in this case since adjacent characters can be equal.

Hash Maps and Sets

Longest Chain of Consecutive Numbers

Find the longest chain of consecutive numbers in an array. Two numbers are consecutive if they have a difference of 1.

Example:

Input: `nums = [1, 6, 2, 5, 8, 7, 10, 3]`

Output: 4

Explanation: The longest chain of consecutive numbers is 5, 6, 7, 8.

Intuition

A naive approach to this problem is to sort the array. When all numbers are arranged in ascending order, consecutive numbers will be placed next to each other. This allows us to traverse the array to identify the longest sequence of consecutive numbers.

`[1 6 2 5 8 7 10 3]` $\xrightarrow{\text{sort}}$ `[1 2 3 5 6 7 8 10]`

longest

This approach requires sorting, which takes $O(n \log(n))$ time, where n denotes the length of the array. Let's see how we could do better.

It's important to understand that every number in the array can represent the start of some consecutive chain. One approach is to treat each number as the start of a chain and search through the array to identify the rest of its chain.

To do this, we can leverage the fact that for any number `num`, its next consecutive number will be `num + 1`. This means we'll always know which number to look for when trying to find the next number in a sequence. The code snippet for this approach is provided below:

```
def longest_chain_of_consecutive_numbers_brute_force(nums: List[int]) -> int:
    if not nums:
        return 0
    longest_chain = 0
    # Look for chains of consecutive numbers that start from each number.
    for num in nums:
        current_num = num
        current_chain = 1
        # Continue to find the next consecutive numbers in the current chain.
        while (current_num + 1) in nums:
            current_num += 1
            current_chain += 1
        longest_chain = max(longest_chain, current_chain)
    return longest_chain
```

This brute force approach takes $O(n^3)$ time because of the nested operations involved:

- The outer for-loop iterates through each element, which takes $O(n)$ time.
- For each element, the inner while-loop can potentially run up to n iterations if there's a long consecutive sequence starting from the current number.
- For each, while-loop iteration, an $O(n)$ check is performed to see if the next consecutive number exists in the array.

This is slower than the sorting approach, but we can make a couple of optimizations to improve the time complexity. Let's discuss these.

Optimization - hash set

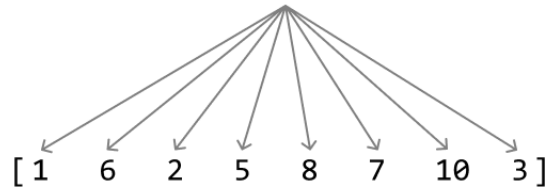
To find the next number in a sequence, we perform a linear search through the array. However, by storing all the numbers in a hash set, we can instead query this hash set in constant time to check if a number exists.

This reduces the time complexity from $O(n^3)$ to $O(n^2)$.

Optimization - identifying the start of each chain

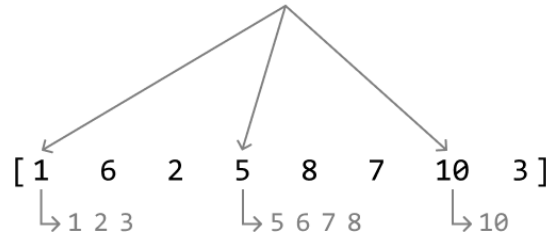
In the brute force approach, we treat each number as the start of a chain. This becomes quite expensive because we perform a linear search at every number to find the rest of its chain:

find the rest of a consecutive chain starting at:



The key observation here is that we don't need to perform this search for every number in a chain. Instead, we only need to perform it for the **smallest number in each chain**, as this number identifies the start of its chain:

find the rest of a consecutive chain starting at:



We can determine if a number is the smallest number in its chain by **checking the array doesn't contain the number that precedes it ($\text{curr_num} - 1$)**. We can also use the hash set for this check.

not the smallest in its chain: $6 - 1 = 5$ exists
↓
[1 6 2 5 8 7 10 3]
smallest in its chain: $5 - 1 = 4$ doesn't exist
↓
[1 6 2 5 8 7 10 3]

This reduces the time complexity from $O(n^2)$ to $O(n)$, as now every chain is searched through only once. This is explained in more detail in the complexity analysis.

Implementation

```
def longest_chain_of_consecutive_numbers(nums: List[int]) -> int:
    if not nums:
        return 0
    num_set = set(nums)
    longest_chain = 0
    for num in num_set:
        # If the current number is the smallest number in its chain, search for
        # the length of its chain.
```

```
if num - 1 not in num_set:
    current_num = num
    current_chain = 1
    # Continue to find the next consecutive numbers in the chain.
    while (current_num + 1) in num_set:
        current_num += 1
        current_chain += 1
    longest_chain = max(longest_chain, current_chain)
return longest_chain
```

Complexity Analysis

Time complexity: The time complexity of `longest_chain_of_consecutive_numbers` is $O(n)$ because, although there are two loops, the inner loop is only executed when the current number is the start of a chain. This ensures each chain is iterated through only once in the inner while-loop. Thus, the total number of iterations for both loops combined is $O(n)$: the outer for-loop runs n times, and the inner while-loop runs a total of n times across all iterations, resulting in a combined time complexity of $O(n + n) = O(n)$.

Space complexity: The space complexity is $O(n)$ since the hash set stores each unique number from the array.

Geometric Sequence Triplets

A geometric sequence triplet is a sequence of three numbers where each successive number is obtained by multiplying the preceding number by a constant called the common ratio.

Let's examine three triplets to understand how this works:

- (1, 2, 4): Geometric sequence with a ratio of 2 (i.e., $[1, 1 \cdot 2 = 2, 2 \cdot 2 = 4]$).
- (5, 15, 45): Geometric sequence with a ratio of 3 (i.e., $[5, 5 \cdot 3 = 15, 15 \cdot 3 = 45]$).
- (2, 3, 4): Not a geometric sequence.

Given an array of integers and a common ratio r , find all triplets of indexes (i, j, k) that follow a geometric sequence for $i < j < k$. It's possible to encounter duplicate triplets in the array.

Example:

$r = 2$, $[\underset{0}{2} \quad 1 \quad 2 \quad \underset{3}{4} \quad \underset{4}{8} \quad 8]$: $(2, 2 \cdot 2 = 4, 4 \cdot 2 = 8)$

$[\underset{0}{2} \quad 1 \quad 2 \quad \underset{3}{4} \quad 8 \quad \underset{5}{8}]$: $(2, 2 \cdot 2 = 4, 4 \cdot 2 = 8)$

$[2 \quad 1 \quad \underset{2}{2} \quad \underset{3}{4} \quad \underset{4}{8} \quad 8]$: $(2, 2 \cdot 2 = 4, 4 \cdot 2 = 8)$

$[2 \quad 1 \quad \underset{2}{2} \quad \underset{3}{4} \quad 8 \quad \underset{5}{8}]$: $(2, 2 \cdot 2 = 4, 4 \cdot 2 = 8)$

$[2 \quad \underset{1}{1} \quad \underset{2}{2} \quad \underset{3}{4} \quad 8 \quad 8]$: $(1, 1 \cdot 2 = 2, 2 \cdot 2 = 4)$

Input: $\text{nums} = [2, 1, 2, 4, 8, 8]$, $r = 2$

Output: 5

Explanation: Triplet $[2, 4, 8]$ occurs at indexes $(0, 3, 4), (0, 3, 5), (2, 3, 4), (2, 3, 5)$. Triplet $[1, 2, 4]$ occurs at indexes $(1, 2, 3)$.

Intuition

For a triplet to form a geometric sequence, it has to adhere to two main rules:

1. It consists of three values that follow a geometric sequence with a common ratio r .
2. The three values forming the triplet must appear in the same order within the array as they do in the geometric sequence. This means for a geometric triplet $(\text{nums}[i], \text{nums}[j], \text{nums}[k])$, the indexes must follow the order $i < j < k$.

How can we represent a geometric sequence so that it follows rule 1? Let's say the first number is x . The second number is the first number multiplied by r (i.e., $x \cdot r$), and the third is the second number multiplied by r (i.e., $x \cdot r \cdot r = x \cdot r^2$). So, a triplet in a geometric sequence can be represented as $(x, x \cdot r, x \cdot r^2)$.

A brute force approach is to iterate over all possible triplet in the array to check if any of them follow a geometric progression. However, it would take three nested for-loops to search through all the triplets, resulting in a time complexity of $O(n^3)$, where n denotes the length of the input array. Can we do better?

An important observation here is that **if we know one value of a triplet, we can calculate what the other two values should be.**

This is because all three values are related by the common ratio r . So, for any number x in the array, we just need to find the values $x \cdot r$ and $x \cdot r^2$ to form a geometric triplet $(x, x \cdot r, x \cdot r^2)$. However, we could run into issues when using this triplet representation. While it's clear the values $x \cdot r$ and $x \cdot r^2$ must be positioned to the right of x , we have to be careful since the order of these values matters: we don't want to accidentally identify a triplet such as $(x, x \cdot r^2, x \cdot r)$, which is invalid:

$$[\cdot \quad \cdot \quad \overset{x}{\downarrow} x \quad \cdot \quad \cdot \quad x \cdot r^2 \quad \cdot \quad x \cdot r]$$
 triplet values $(x, x \cdot r^2, x \cdot r)$ appear in the wrong order

We can work around this issue by using the $(x/r, x, x \cdot r)$ triplet representation, which allows us to always maintain order by looking for x/r to the left of x and $x \cdot r$ to the right:

$$[\cdot \quad \cdot \quad \cdot \quad \cdot] \quad x \quad [\cdot \quad \cdot \quad \cdot]$$

$$\underbrace{\hspace{1.5cm}}_{\text{find } x/r} \qquad \underbrace{\hspace{1.5cm}}_{\text{find } x \cdot r}$$

One way we can find the x/r and $x \cdot r$ values is by linearly searching through the left and right subarrays. This linear search would need to be done for each number in the array, resulting in an $O(n^2)$ time complexity. While this is an improvement from the brute force solution, it would be great if we had a way to find those values faster.

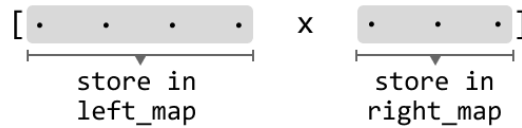
Hash maps

A hash map would be a great way to solve this problem, as it allows us to query specific values in constant time.

What we would need are two hash maps:

- A hash map that contains numbers to the left of each x (`left_map`).

- A hash map that contains numbers to the right of each x (`right_map`).



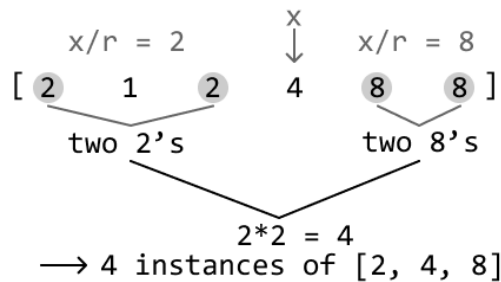
Hash maps allow us to query for both x/r in the left hash map and query for $x \cdot r$ in the right hash map in constant time on average. Note that a hash map would be preferred over a hash set because hash maps can also store the frequency of each value it stores. This is crucial since the array might contain duplicates, and we need to know the frequency of each value to accurately identify all possible triplets.

Finding all $(x/r, x, x \cdot r)$ triplets

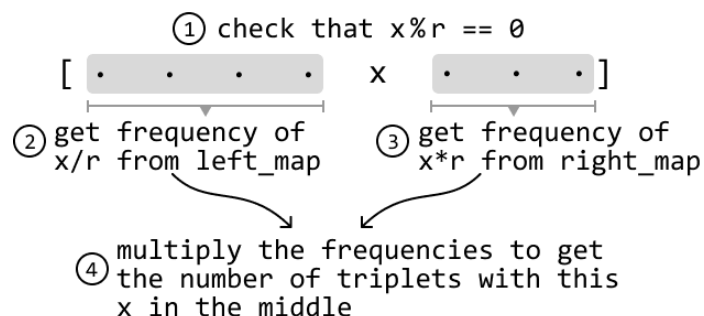
Our goal is to find all triplets that follow a geometric sequence, representing each number in the array as the middle (x) number of a triplet.

Before we find a triplet's x/r value, we need to **check if x is divisible by r** . If it's not, it's impossible to form a triplet from the current value of x . Otherwise, we can proceed to look for the triplet.

For any element x , there could be multiple instances of x/r in `left_map` and multiple instances of $x \cdot r$ in `right_map`, implying that multiple triplets can be formed using x as the middle value. So, to get the total number of triplets that can be formed with x in the middle, **multiply the frequencies of $x \cdot r$ and x/r** :



This overall methodology can be summarized in the following steps:



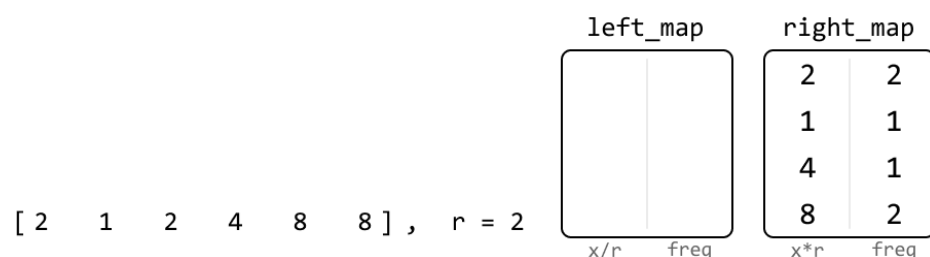
Note that if either x/r or $x \cdot r$ are not found in their hash maps, their frequency is 0 by default.

Let's implement this strategy using the example below:

$[2 \quad 1 \quad 2 \quad 4 \quad 8 \quad 8] , \quad r = 2$

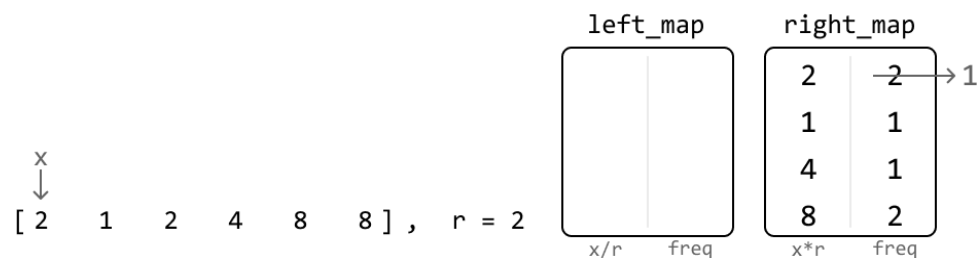
To ensure the hash maps always contain the correct values, we'd need to incorporate a dynamic strategy that involves updating the hash maps as we go because the values in both hash maps will be different depending on the position of x in the array.

Since we're traversing the array from left to right, we should **initially fill the right hash map with all values in the array**. This is because, before the start of the iteration, every element is a potential candidate for $x \cdot r$. Meanwhile, the left hash map is initially empty because there are no preceding elements to consider as potential x/r values:



Now let's look for triplets. Start by representing the first value as the middle value (x) of a triplet.

First, let's **update right_map**. We should remove the current value (2) from right_map since this 2 is not to the right of itself. There are two 2's in right_map, so let's reduce its frequency to 1:



Next, **check if x/r is an integer**. In this case, it is, so let's **find the number of triplets with x as the middle number** by multiplying the frequencies of x/r and $x \cdot r$, which we can get from the respective hash maps. Since left_map doesn't contain x/r at this point, its frequency is 0:

x
 \downarrow
 $[2 \quad 1 \quad 2 \quad 4 \quad 8 \quad 8]$
 $r = 2$

```

x % r == 0 → count += left_map[x/r] * right_map[x*r]
                += left_map[2/2] * right_map[2*2]
                += 0 * 1
                += 0
  
```

Before moving on to the next value, let's **add the current number to the left_map** because it now becomes a potential x/r value for future triplets in the array:

left_map		right_map	
2	1	2	1
		1	1
		4	1
		8	2
x/r	freq	$x*r$	freq

Repeating this process for the rest of the array allows us to find all geometric triplets with a ratio of r . To clarify, the hash maps in the upcoming diagrams represent their state at the current position of x in the array. This means that **left_map** includes values to the left of the current x , and the **right_map** includes values to the right of it.

x
 \downarrow
 $[2 \quad 1 \quad 2 \quad 4 \quad 8 \quad 8]$, $r = 2$

left_map		right_map	
2	1	2	1
		1	0
		4	1
		8	2
x/r	freq	$x*r$	freq

```

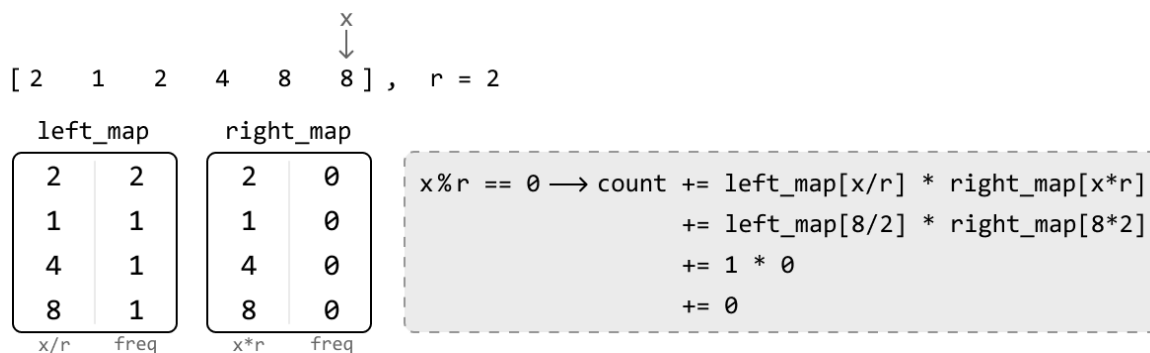
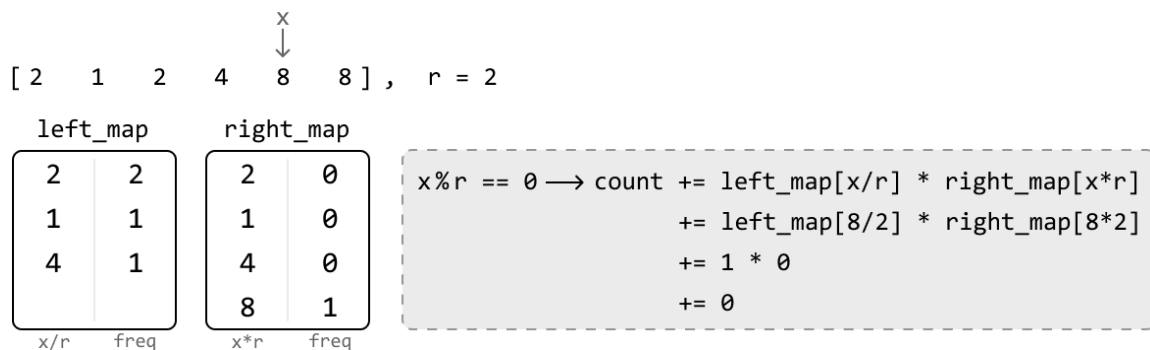
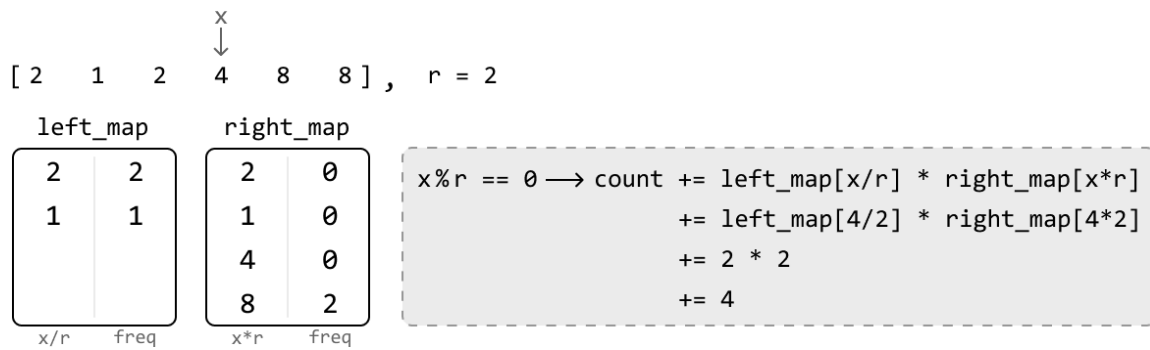
x % r != 0 → can't form a triplet → continue
  
```

x
 \downarrow
 $[2 \quad 1 \quad 2 \quad 4 \quad 8 \quad 8]$, $r = 2$

left_map		right_map	
2	1	2	0
1	1	1	0
		4	1
		8	2
x/r	freq	$x*r$	freq

```

x % r == 0 → count += left_map[x/r] * right_map[x*r]
                += left_map[2/2] * right_map[2*2]
                += 1 * 1
                += 1
  
```

Implementation

```

def geometric_sequence_triplets(nums: List[int], r: int) -> int:
    # Use 'defaultdict' to ensure the default value of 0 is returned when
    # accessing a key that doesn't exist in the hash map. This effectively sets
    # the default frequency of all elements to 0.
    left_map = defaultdict(int)
    right_map = defaultdict(int)
    count = 0
    # Populate 'right_map' with the frequency of each element in the array.

```

```

for x in nums:
    right_map[x] += 1
# Search for geometric triplets that have x as the center.
for x in nums:
    # Decrement the frequency of x in 'right_map' since x is now being
    # processed and is no longer to the right.
    right_map[x] -= 1
    if x % r == 0:
        count += left_map[x // r] * right_map[x * r]
    # Increment the frequency of x in 'left_map' since it'll be a part of the
    # left side of the array once we iterate to the next value of x.
    left_map[x] += 1
return count

```

Complexity Analysis

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

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

Linked Lists

Palindromic Linked List

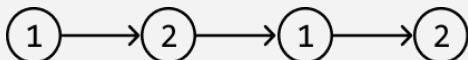
Given the head of a singly linked list, determine if it is a palindrome.

Example 1:



Output: True

Example 2:

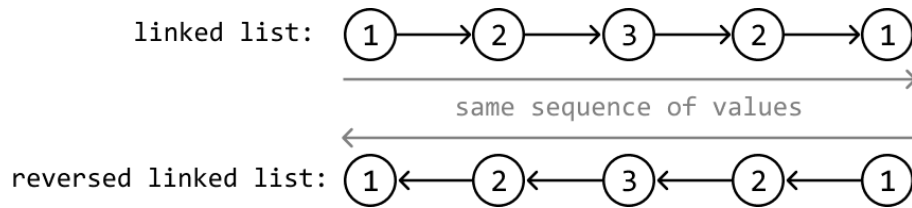


Output: False

Intuition

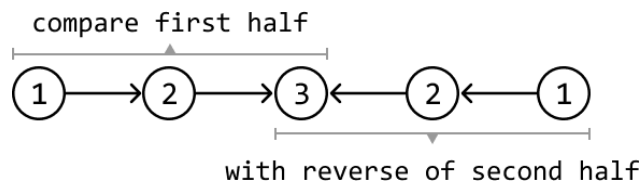
A linked list would be palindromic if its values read the same forward and backward. A naive way to check this would be to store all the values of the linked list in an array, allowing us to freely traverse these values forward and backward to confirm if it's palindromic. However, this uses linear space. Instead, it would be better if we had a way to traverse the linked list in reverse order to confirm if it's a palindrome. Is there a way to go about this?

Going off the above definition, we know that if a linked list is a palindrome, reversing it would result in the same sequence of values.



This means we could create a copy of the linked list, reverse it, and compare its values with the original linked list. However, this would still take up linear space. Can we adjust this idea to avoid creating a new linked list?

An important observation is that we only need to **compare the first half of the original linked list with the reverse of the second half** (if there are an odd number of elements, we can just include the middle node in both halves) to check if the linked list is a palindrome:

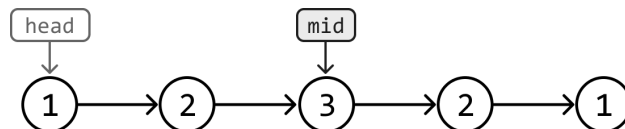


Before we can perform this comparison, we need to:

1. Find the middle of the linked list to get the head of the second half.
2. Reverse the second half of the linked list from this middle node.

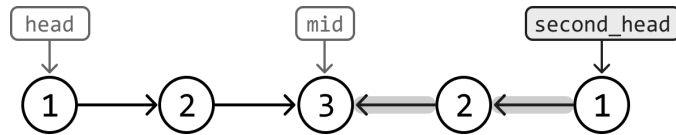
Notice that step 2 involves modifying the input. In this problem, let's assume this is acceptable. However, **it's always good to check with the interviewer if changing the input is allowed before moving forward with the solution.**

Now, let's see how these two steps can be applied. Start by obtaining the middle node (*mid*) of the linked list.



To learn how to get to the middle of a linked list, read the explanation in the *Linked List Midpoint* problem in the *Fast and Slow Pointers* chapter.

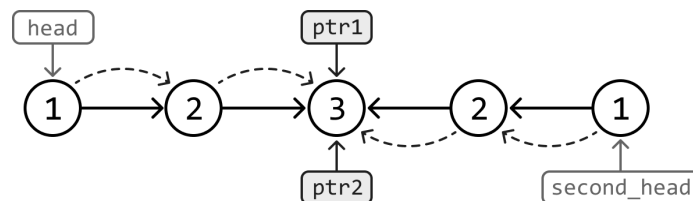
Then, reverse the second half of the linked list starting at *mid*. The last node of the original linked list becomes the head of the second half. This second head is used to traverse the newly reversed second half.



To learn how to reverse a linked list in $O(1)$ time, read the explanation in the *Reverse Linked List* problem in this chapter.

The last thing we need to do is check if the first half matches the now-reversed second half. We can do this by simultaneously traversing both halves node by node, and comparing each node from the first half to the corresponding node from the second half. If at any point the node values don't match, it indicates the linked list is not a palindrome.

We can use two pointers (ptr1 and ptr2) to iterate through the first and the reversed second half of the linked list, respectively:



Implementation

```
def palindromic_linked_list(head: ListNode) -> bool:
    # Find the middle of the linked list and then reverse the second half of the
    # linked list starting at this midpoint.
    mid = find_middle(head)
    second_head = reverse_list(mid)
    # Compare the first half and the reversed second half of the linked list.
    ptr1, ptr2 = head, second_head
    res = True
    while ptr2:
        if ptr1.val != ptr2.val:
            res = False
        ptr1, ptr2 = ptr1.next, ptr2.next
    return res
```

From the 'Reverse Linked List' problem.

```
def reverse_list(head: ListNode) -> ListNode:
    prevNode, currNode = None, head
    while currNode:
        nextNode = currNode.next
        currNode.next = prevNode
        prevNode = currNode
        currNode = nextNode
    return prevNode
```

From the 'Linked List Midpoint' problem.

```
def find_middle(head: ListNode) -> ListNode:
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

Complexity Analysis

Time complexity: The time complexity of `palindromic_linked_list` is $O(n)$, where n denotes the length of the linked list. This is because it involves iterating through the linked list three times: once to find the middle node, once to reverse the second half, and once more to compare the two halves.

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

Interview Tip

Tip: Confirm if it's acceptable to modify the linked list.



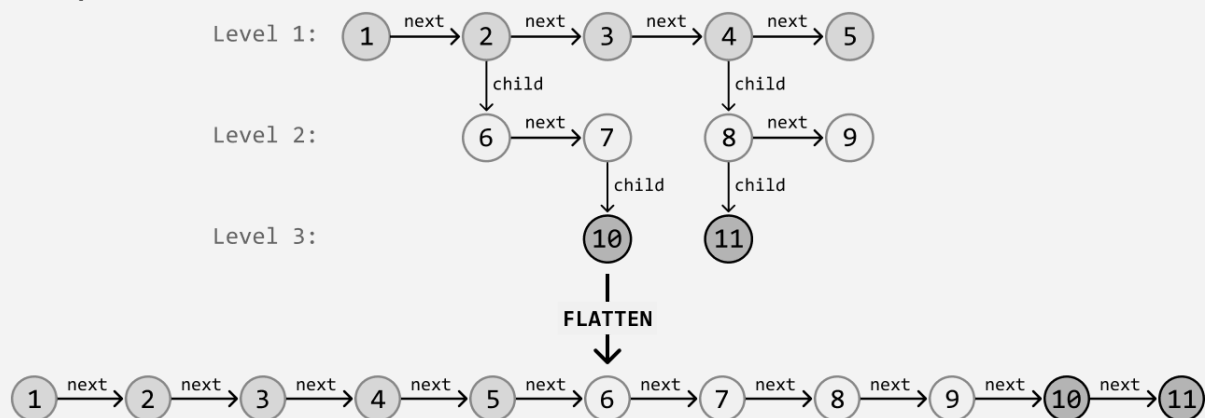
In our solution, we reversed the second half of the linked list which dismantled the input's initial structure. Why does this matter? Oftentimes, the input data structure should not be modified, particularly if it's shared or accessed concurrently. As such, it's important to confirm with your interviewer whether input modification is acceptable and to briefly address the implications of this.

Flatten a Multi-Level Linked List

In a multi-level linked list, each node has a next pointer and child pointer. The next pointer connects to the subsequent node in the same linked list, while the child pointer points to the head of a new linked list under it. This creates multiple levels of linked lists. If a node does not have a child list, its child attribute is set to null.

Flatten the multi-level linked list into a single-level linked list by linking the end of each level to the start of the next one.

Example:



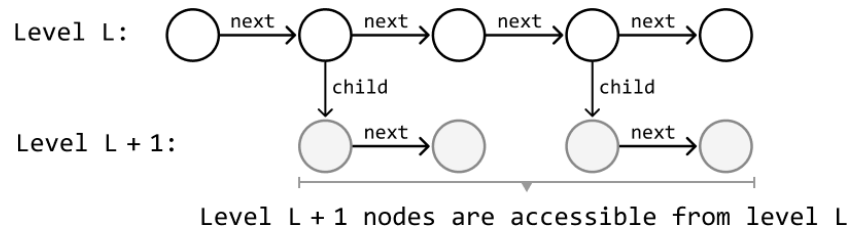
Intuition

Consider the two main conditions required to form the flattened linked list:

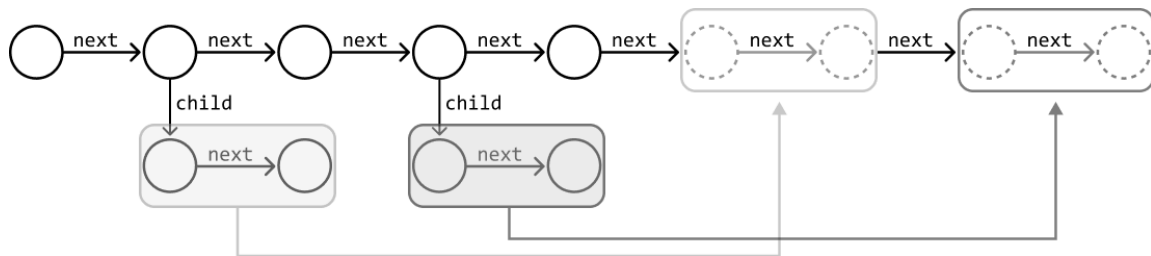
1. The order of the nodes on each level needs to be preserved.
2. All the nodes in one level must connect before appending nodes from the next level.

The challenge with this problem is figuring out how we process linked lists in lower levels. One strategy that might come to mind is level-order traversal using breadth-first search. However, breadth-first search usually involves the use of a queue, which would result in at least a linear space complexity. Is there a way we could merge the levels of the linked lists in place?

A key observation is that **for any level of the multi-level linked list, we have direct access to all the nodes on the next level**. This is because each node's child node at any given level 'L' has direct access to nodes on the next level 'L + 1':

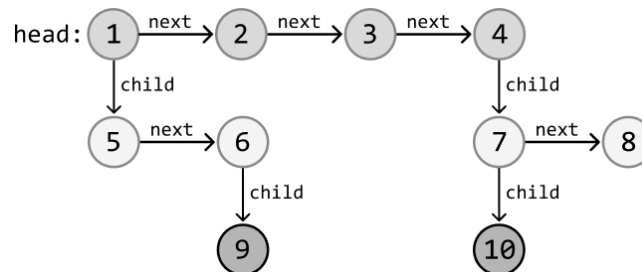


How can we connect the nodes on level 'L + 1' to the end of level 'L'? Since we have access to the nodes at the next level from the current level's child pointers, we can append each child linked list to the end of the current level, which effectively merges these two levels into one.

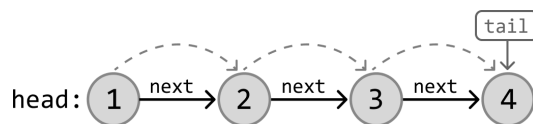


So, with all the nodes on level 'L + 1' appended to level 'L', we can continue this process by appending nodes from level 'L + 2' to level 'L + 1', and so on.

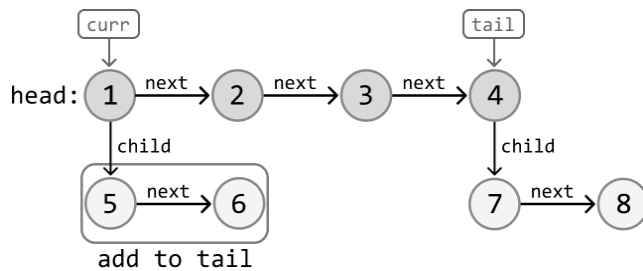
Now that we have a high-level idea about what we should do, let's try this strategy on the following example:



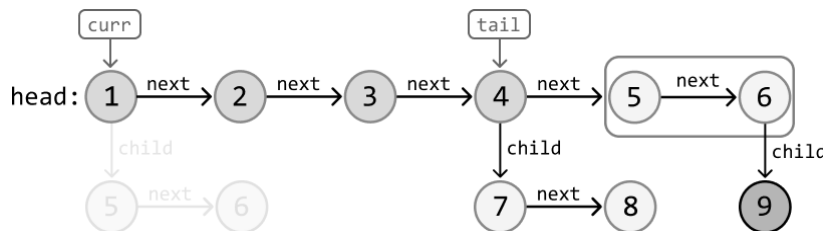
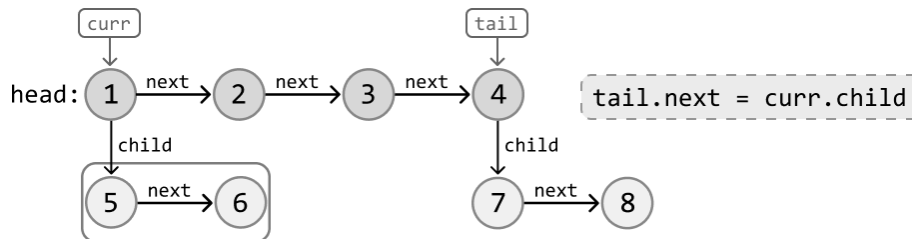
We'll start by appending level 2's nodes to the end of level 1. Before we can do this, we would need a reference to level 1's tail node so we can easily add nodes to the end of the linked list. To set this reference, advance through level 1's linked list using a `tail` pointer until it reaches the last node, which happens when `tail.next` is equal to null:



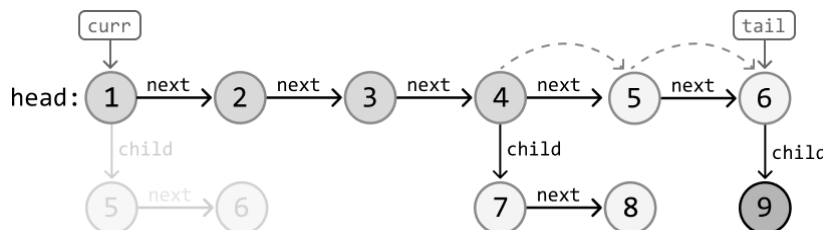
Now, add the child linked lists (5 → 6 and 7 → 8) to the tail node. We must keep the tail pointer fixed at the end of the linked list, so let's introduce a separate pointer, curr, to traverse the linked list. Whenever curr encounters a node with a child node that isn't null, we know we've found a child linked list. In the example, the first node (node 1) has a child linked list, which we want to add to the tail node:



To add this child linked list to the end of the tail node, set `tail.next` to the head of the child list:

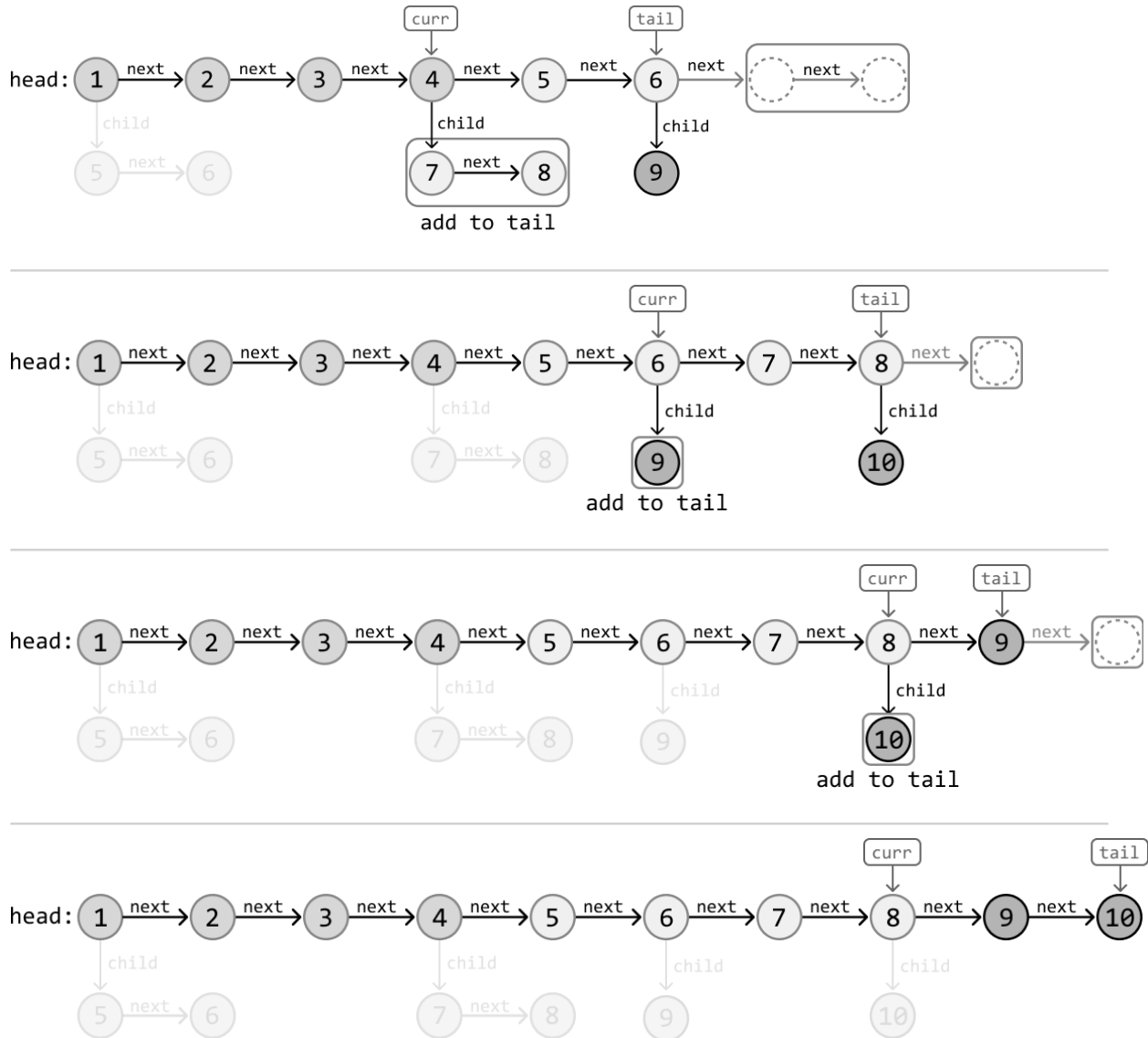


Before incrementing `curr` to find the next node with a child linked list, we need to readjust the position of the tail pointer so it's pointing at the last node of the newly extended linked list (node 6 in this case). Again, we can do this by advancing the tail pointer until its next node is null:



With the tail pointer now repositioned, we can continue this process of:

- Finding the next node with a child linked list using the curr pointer.
- Adding the child linked list to the tail node.
- Advancing the tail pointer to the last node of the flattened linked list.



After the process is complete, we can return head, which is the head of the flattened linked list.

One last important detail to mention is that after appending any child linked list to the tail, we should nullify the child attribute to ensure the linked list is fully flattened.

Implementation

The definition of the `MultiLevelListNode` class is provided below:

```

class MultiLevelListNode:
    def __init__(self, val, next, child):
        self.val = val
        self.next = next
        self.child = child

```

```

def flatten_multi_level_list(head: MultiLevelListNode) -> MultiLevelListNode:
    if not head:
        return None
    tail = head
    # Find the tail of the linked list at the first level.
    while tail.next:
        tail = tail.next
    curr = head
    # Process each node at the current level. If a node has a child linked list,
    # append it to the tail and then update the tail to the end of the extended
    # linked list. Continue until all nodes at the current level are processed.
    while curr:
        if curr.child:
            tail.next = curr.child
            # Disconnect the child linked list from the current node.
            curr.child = None
            while tail.next:
                tail = tail.next
        curr = curr.next
    return head

```

Complexity Analysis

Time complexity: The time complexity of `flatten_multi_level_list` is $O(n)$, where n denotes the number of nodes in the multi-level linked list. This is because we iterate through each node in the multi-level linked list at most twice: once to iterate `tail` and once to iterate `curr`.

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

Fast and Slow Pointers

Happy Number Time Complexity Analysis

The following time complexity analysis establishes an upper bound on the steps required to determine a happy number. In this analysis, we define the "next number" of a number n as the result obtained by summing the squares of the digits of n .

1. Upper limit for the next number

For any number n with a fixed number of digits, the maximum value for its successor is achieved when all its digits are 9. For instance, the maximum next number from a 3-digit number happens when this 3-digit number is 999.

2. Size of the next number relative to the number of digits

- If a number n has 1 or 2 digits, it's possible for the next number to be larger than n (e.g., the next number of 99 is 162, which is one digit longer).
- If a number n has 3 or more digits, the next number is always smaller than the original value of n . The table below highlights how the largest number with 3 or more digits has a smaller next number.

Digits	Largest Number	Next Number
1	9	81
2	99	162
3	999	243
4	9999	324
5	99999	405
6	999999	486

...

...

...

3. Implications for cycles

Since the next number is always smaller for numbers with three or more digits, it means a cycle can only commence in the happy number process once the number falls below 243 (the next number of 999). This is because we've observed that any number n larger than 243 will have the next number smaller than n . However, once we fall below 243, the next number can potentially be larger, potentially cycling back to a previous number.

4. Time complexity for numbers less than 243

Once a number falls below 243, the algorithm will take less than 243 steps to either converge to 1 or to cycle back to a previous number in the sequence. Therefore, since the length of the cycle or the number of steps to reach 1 is bounded by 243, the time complexity of Floyd's cycle detection algorithm for numbers less than 243 is $O(1)$.

5. Time complexity for numbers greater than 243

The number of digits in a number is approximately equal to $\log(n)$ (base 10). So, the calculation of the next number of n will take approximately $\log(n)$ steps (i.e., `get_next_num` will take $\log(n)$ steps to execute).

Let's call n 's next number n_2 . The next number after n_2 (n_3) will take approximately $\log(n_2)$ steps to calculate. The next number after n_3 (n_4) will take approximately $\log(n_3)$ steps to calculate, and so on. From this, we can summarize the time complexity of this process as $O(\log(n) + \log(n_2) + \log(n_3) + \dots)$. Since we've established that $n > n_2 > \dots > n_k$ where n_k is the last number greater than 243, the dominant component of this time complexity is $O(\log(n))$. So, the time complexity for numbers greater than 243 is $O(\log(n))$.

Conclusion

When n is less than 243, the time complexity is $O(1)$, and when n is greater than 243, the time complexity is $O(\log(n))$. Therefore, the overall time complexity of the algorithm is $O(\log(n))$.

Interview Tip

Tip: Don't waste time on complex proofs if it isn't an important part of the interview.



During an interview, correctly deciphering the exact time of an algorithm like the one used to solve this problem isn't usually expected. In situations like this, you can instead make an educated guess about how the algorithm's runtime would grow with larger inputs based on the behavior of the algorithm. Mention any assumptions you make when discussing your estimates.

It might also be helpful to mention what parts of the problem or the solution make it difficult to analyze the time complexity. In this problem, it's initially unclear how many steps the happy number process would take before reaching 1 or revealing a cycle.

Binary Search

Find the Median From Two Sorted Arrays

Given two sorted integer arrays, find their median value as if they were merged into a single sorted sequence.

Example 1:

Input: `nums1 = [0, 2, 5, 6, 8]`, `nums2 = [1, 3, 7]`

Output: `4.0`

Explanation: Merging both arrays results in `[0, 1, 2, 3, 5, 6, 7, 8]`, which has a median of $(3 + 5) / 2 = 4.0$.

Example 2:

Input: `nums1 = [0, 2, 5, 6, 8]`, `nums2 = [1, 3, 7, 9]`

Output: `5.0`

Explanation: Merging both arrays results in `[0, 1, 2, 3, 5, 6, 7, 8, 9]`, which has a median of `5.0`.

Constraints:

- At least one of the input arrays will contain an element.

Intuition

The brute force approach to this problem involves merging both arrays and finding the median in this merged array. This approach takes $O((m + n)\log(m + n))$ time, where m and n denote the lengths of each array, respectively. This is primarily due to the cost of sorting the merged array of length $m + n$. This approach can be improved to $O(m + n)$ time by merging both arrays in order, which is possible because both arrays are already individually sorted. However, is there a way to find the median without merging the two arrays?

In this explanation, we use “total length” to refer to the combined length of both input arrays. Let’s discuss odd and even total lengths separately, as these result in two different types of medians.

Consider the following two arrays that have an even total length:

```
nums1 = [ 0    2    5    6    8 ]
nums2 = [ 1    3    7 ]
```

Below is what these two arrays would look like when merged. Let’s see if we can draw any insights from this.

```
merged array:
[ 0    1    2    3    5    6    7    8 ]
```

Observe that the merged array can be divided into two halves, which reveals the median values on the inner edge of each half.

```

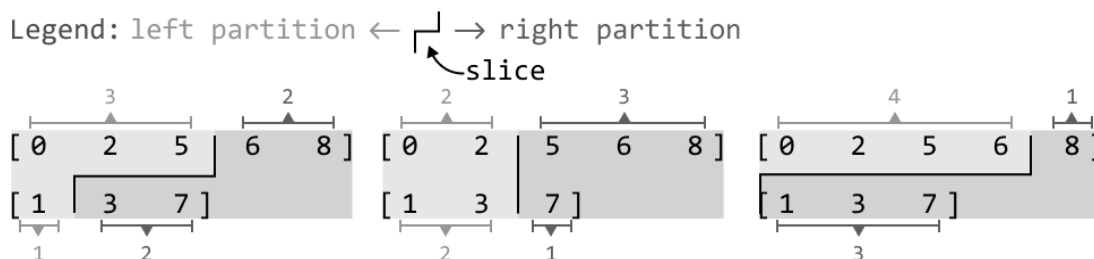
      end of   start of
      left half right half
      ↑       ↑
[ 0    1    2    3    5    6    7    8 ]
  left half  [ 3    5 ]  right half

```

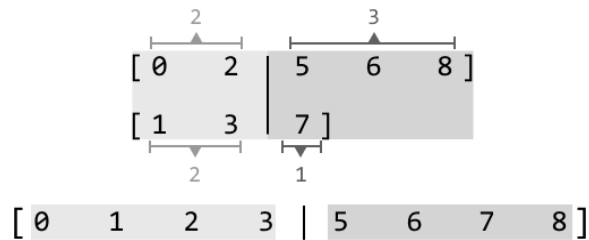
A challenge here is identifying which values in either input array belong to the left half of the merged array, and which belong to the right half. One thing we do know is the size of each half of the merged array: 4, **half of the total length**.

Slicing both arrays

To figure out which values belong to each half, we can try “slicing” both arrays into two segments, where the left segments of both arrays and the right segments of both arrays each have 4 total values. Let’s refer to the values on the left and right of the slice as the “left partition” and “right partition,” respectively. Below are three examples of what this slice could look like:



As we can see, there are several ways to slice the arrays to produce two partitions of equal size (4). However, only one of these slices corresponds to the halves of the merged array. In our example, it’s this slice:

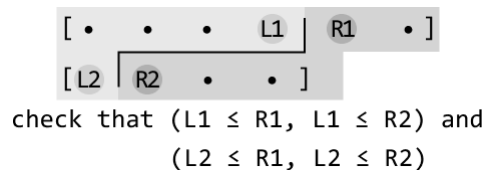


Let's refer to this as the "correct slice." We'll explain how to identify the correct slice shortly, but first, let's consider how to identify which slice correctly corresponds to the halves of the merged array.

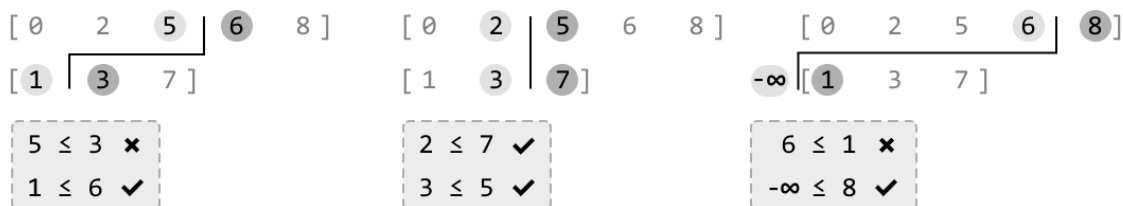
Determining the correct slice

An important observation is that **all values in the left partition must be less than or equal to the values in the right partition.**

We can assess this by comparing the two end values of the left partition with the start values of the right partition (illustrated below). Let's refer to the end values of the left partition as $L1$ and $L2$, respectively. Similarly, let's call the start values of the right partition $R1$ and $R2$.



Since the values in each array are sorted, we know that conditions $L1 \leq R1$ and $L2 \leq R2$ are always true. Then, all we have to do is check that $L1 \leq R2$ and $L2 \leq R1$. We can observe how this comparison reveals the correct slice among the previous three example slices:



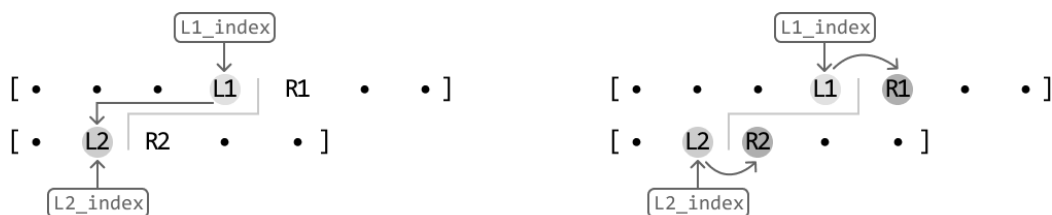
Notice that in the third example above, the second array does not contribute any values to the left partition. So, to work around this, we set the second array's left value to $-\infty$ so that $L2 \leq R1$ is true by default.

Searching for the correct slice

Now, our goal is to search through all possible slices until we find the correct one. We do this by

searching through all possible placements of L1, R1, L2, and R2. Note that we only need to **search for L1** since the other three values can be inferred based on L1's index.

Let's take a closer look at how this works. Once we identify L1's index, we can calculate L2's index based on L1's index, which is demonstrated in the diagram below. R1 and R2 are just the values immediately to the right of L1 and L2, respectively.

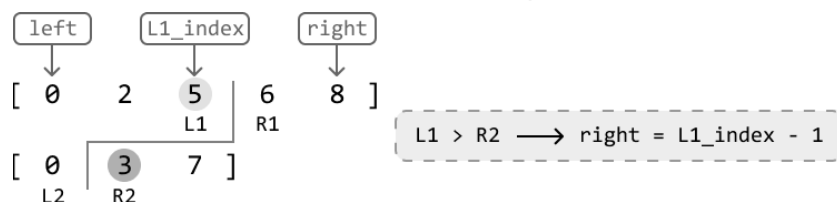


- number of left partition values in $nums1 = (L1_index + 1)$
- hence, number of left partition values in $nums2 = half_total_len - (L1_index + 1)$
- $L2_index = half_total_len - (L1_index + 1) - 1$

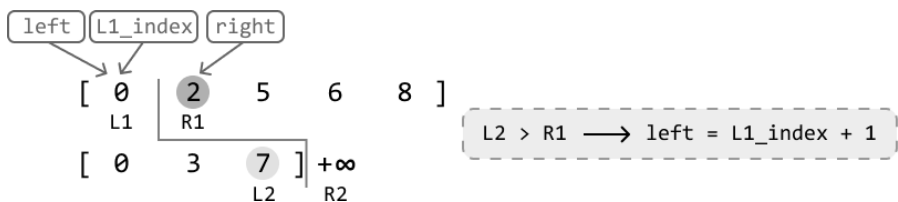
Since we search for L1 over $nums1$, which is a sorted array, we can use **binary search** instead of searching for it linearly. The **search space** will encompass all values of the $nums1$.

Let's figure out how to **narrow the search space**. Here, we'll define the midpoint as $L1_index$, since it's also the index of L1. Let's discuss how the search space is narrowed based on these conditions:

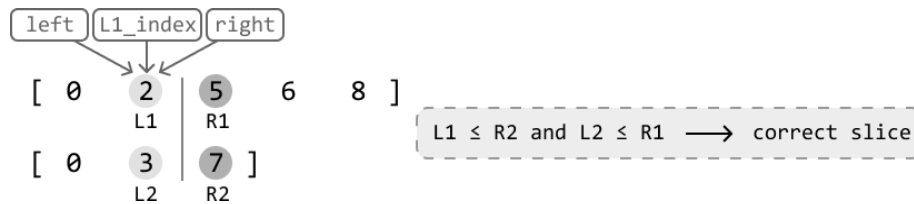
- If $L1 > R2$, then L1 is larger than it should be as we expect L1 to be less than or equal to R2. To search for a smaller L1, narrow the search space toward the left:



- If $L2 > R1$, then R1 is smaller than it should be as we expect R1 to be greater than or equal to L2. To search for a larger R1, narrow the search space toward the right:



- If $L1 \leq R2$, and $L2 \leq R1$, the correct slice has been located:



Search space optimization

A small optimization here is to ensure that `nums1` is the smallest array between the two input arrays. This ensures our search space is as small as possible. If `nums2` is smaller than `nums1`, we can just swap the two arrays, allowing `nums1` to always be the smaller array.

Returning the median

Once binary search has identified the correct slice, we need to return the median. With an even total length, the median is calculated using the array's two middle values. From our set of partition slice values (`L1`, `R1`, `L2`, and `R2`), which of them are the middle two? We know one of the median values is from the left partition and the other is from the right partition. From the left partition, the largest value between `L1` and `L2` will be closest to the middle. From the right, the smallest value between `R1` and `R2` is closer to the middle:



So, to return the median, we just return the sum of these two values, divided by 2 using floating-point division.

What if the total length of both arrays is odd?

The main difference when the total length of both arrays is odd compared to an even length is that we can no longer slice the arrays into two equal halves. One half must have an additional value.

```
nums1 = [ 0   2   5   6   8 ]
nums2 = [ 1   3   7   9 ]
merged array:
[ 0   1   2   3   5   6   7   8   9 ]
```

The diagram above shows that the right half ends up with one extra value. This is because when we calculate the slice position, we ensure the left half has a size of half the total length. In this example, this calculation using integer division gives us a left half size of $(5 + 4) // 2 = 4$. Consequently, this means the right half ends up with 5 values. So, **when the total length is odd, the median can be found in the right half:**

merged array:
 [0 1 2 3 **[5]** 6 7 8 9]
 ↖ start of right half

So, after the binary search narrows down the correct slice, we can just return the smallest value between R1 and R2.

smallest on the right
 [0 2 | **5** 6 8]
 [0 3 | **7**]

Implementation

```
def find_the_median_from_two_sorted_arrays(nums1: List[int],
                                           nums2: List[int]) -> float:
    # Optimization: ensure 'nums1' is the smaller array.
    if len(nums2) < len(nums1):
        nums1, nums2 = nums2, nums1
    m, n = len(nums1), len(nums2)
    half_total_len = (m + n) // 2
    left, right = 0, m - 1
    # A median always exists in a non-empty array, so continue binary search until
    # it's found.
    while True:
        L1_index = (left + right) // 2
        L2_index = half_total_len - (L1_index + 1) - 1
        # Set to -infinity or +infinity if out of bounds.
        L1 = float('-inf') if L1_index < 0 else nums1[L1_index]
        R1 = float('inf') if L1_index >= m - 1 else nums1[L1_index + 1]
        L2 = float('-inf') if L2_index < 0 else nums2[L2_index]
```

```

R2 = float('inf') if L2_index >= n - 1 else nums2[L2_index + 1]
# If 'L1 > R2', then 'L1' is too far to the right. Narrow the search space
# toward the left.
if L1 > R2:
    right = L1_index - 1
# If 'L2 > R1', then 'L1' is too far to the left. Narrow the search space
# toward the right.
elif L2 > R1:
    left = L1_index + 1
# If both 'L1' and 'L2' are less than or equal to both 'R1' and 'R2', we
# found the correct slice.
else:
    if (m + n) % 2 == 0:
        return (max(L1, L2) + min(R1, R2)) / 2.0
    else:
        return min(R1, R2)

```

Complexity Analysis

Time complexity: The time complexity of the `find_the_median_from_two_sorted_arrays` function is $O(\log(\min(m, n)))$ because we perform binary search over the smaller of the two input arrays.

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

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.

Matrix Search

Determine if a target value exists in a matrix. Each row of the matrix is sorted in non-decreasing order, and the first value of each row is greater than or equal to the last value of the previous row.

Example:

target = 21

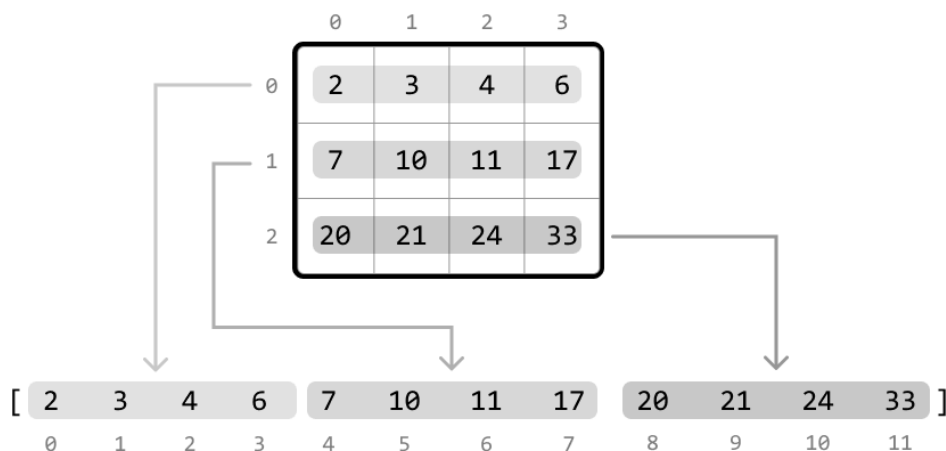
	0	1	2	3
0	2	3	4	6
1	7	10	11	17
2	20	21	24	33

Output: True

Intuition

A naive solution to this problem is to linearly scan the matrix until we encounter the target value. However, this isn't taking advantage of the sorted properties of the matrix.

A key observation is that all values in a given row are greater than or equal to all values in the previous row. This indicates the entire matrix can be considered as a single, continuous, sorted sequence of values:



If we were able to flatten this matrix into a single, sorted array, we could perform a **binary search** on the array. Creating a separate array and populating it with the matrix's values still takes

$O(m \cdot n)$ time, and also takes $O(m \cdot n)$ space, where m and n are the dimensions of the matrix. Is there a way to perform a binary search on the matrix without flattening it?

Let's map the indexes of the flattened array to their corresponding cells in the matrix:

	0	1	2	3	
0	2 0	3 1	4 2	6 3	
1	7 4	10 5	11 6	17 7	
2	20 8	21 9	24 10	33 11	

[2	3	4	6	7	10	11	17	20	21	24	33]
	0	1	2	3	4	5	6	7	8	9	10	11	

This index mapping would give us a way to access the elements of the matrix in a similar way to how we would access them in the flattened array. To figure out how to do this, let's find a way to map any cell (r, c) to its corresponding index in the flattened array.

Let's start by examining the mapped indexes of each row of the matrix:

- Row 0 starts at index 0.
- Row 1 starts at index n .
- Row 2 starts at index $2n$.

From the above observations, we see a pattern: for any row r , the first cell of the row corresponds to the index $r \cdot n$.

	$n = 4$			
	0	1	2	3
$r = 0$	2 0	3	4	6
$r = 1$	7 n	10	11	17
$r = 2$	20 2n	21	24	33

↓
 $r \cdot n$

When we also consider the column value c , we can conclude that for any cell (r, c) , the corresponding index in the flattened array is $r \cdot n + c$.

Now that we understand how the 2D matrix maps to the 1D flattened array, let's work backward to obtain the row and column indexes from an index in the flattened array. Let $i = r \cdot n + c$. The row and column values are:

- $r = i // n$
- $c = i \% n$

We can see how these are obtained below:

$i = rn + c$ $i - c = rn$ $(i - c) // n = r$ $i // n - c // n = r$ $i // n = r \quad (c < n \rightarrow c // n = 0)$	$i = rn + c$ $i \% n = (rn + c) \% n$ $i \% n = rn \% n + c \% n$ $i \% n = c \quad (c < n)$
--	--

Now that we have these formulas, let's use binary search to find the target.

Binary search

To define the **search space**, we need to identify the first and last indexes of the flattened array. The first index is 0, and the last index is $m \cdot n - 1$. So, we set the left and right pointers to 0 and $m \cdot n - 1$, respectively.

To figure out how to **narrow the search space**, let's explore an example matrix that contains the target of 21.

We can calculate mid using the formula: $mid = (left + right) // 2$. Then, determine the corresponding row and column values. Here, the value at the midpoint (10) is less than the target, which means the target is to the right of the midpoint. So, let's narrow the search space toward the right:

target = 21

	0	1	2	3
0	2 0	3 1	4 2	6 3
1	7 4	10 5	11 6	17 7
2	20 8	21 9	24 10	33 11

```

r = mid // n    c = mid % n
  = 5 // 4      = 5 % 4
  = 1           = 1
matrix[r][c] < target → left = mid + 1

```


	0	1	2	3
0	2 0	3 1	4 2	6 3
1	7 4	10 5	11 6	17 7
2	20 8	21 9	24 10	33 11

The new midpoint value is still less than the target, so let's narrow the search space toward the right:

target = 21

	0	1	2	3
0	2 0	3 1	4 2	6 3
1	7 4	10 5	11 6	17 7
2	20 8	21 9	24 10	33 11

```

r = mid // n      c = mid % n
  = 8 // 4        = 8 % 4
  = 2             = 0
matrix[r][c] < target → left = mid + 1

```

	0	1	2	3
0	2 0	3 1	4 2	6 3
1	7 4	10 5	11 6	17 7
2	20 8	21 9	24 10	33 11

The midpoint value is now larger than the target, which means the target is to the left of the midpoint. So, let's narrow the search space toward the left:

target = 21

	0	1	2	3
0	2 0	3 1	4 2	6 3
1	7 4	10 5	11 6	17 7
2	20 8	21 9	24 10	33 11

```

r = mid // n      c = mid % n
  = 10 // 4       = 10 % 4
  = 2             = 2
matrix[r][c] > target → right = mid - 1

```

	0	1	2	3
0	2 0	3 1	4 2	6 3
1	7 4	10 5	11 6	17 7
2	20 8	21 9	24 10	33 11

Now, the midpoint is equal to the target, so we return true to conclude the search.

target = 21

	0	1	2	3
0	2 0	3 1	4 2	6 3
1	7 4	10 5	11 6	17 7
2	20 8	21 9	24 10	33 11

```
r = mid // n    c = mid % n
  = 9 // 4      = 9 % 4
  = 2           = 1
matrix[r][c] == target → return True
```

Note that our **exit condition** should be `while left ≤ right` in order to also examine the above search space where there's just value in it (i.e., when `left == right`).

Implementation

```
def matrix_search(matrix: List[List[int]], target: int) -> bool:
    m, n = len(matrix), len(matrix[0])
    left, right = 0, m * n - 1
    # Perform binary search to find the target.
    while left <= right:
        mid = (left + right) // 2
        r, c = mid // n, mid % n
        if matrix[r][c] == target:
            return True
        elif matrix[r][c] > target:
            right = mid - 1
        else:
            left = mid + 1
    return False
```

Complexity Analysis

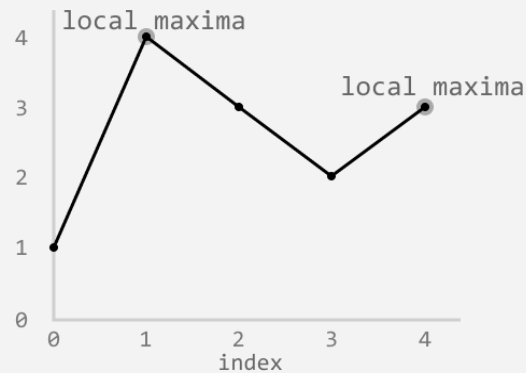
Time complexity: The time complexity of `matrix_search` is $O(\log(m \cdot n))$ because it performs a binary search over a search space of size $m \cdot n$.

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

Local Maxima in Array

A local maxima is a value greater than both its immediate neighbors. Return any local maxima in an array. You may assume that an element is always considered to be strictly greater than a neighbor that is outside the array.

Example:



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

Output: 1 (index 4 is also acceptable)

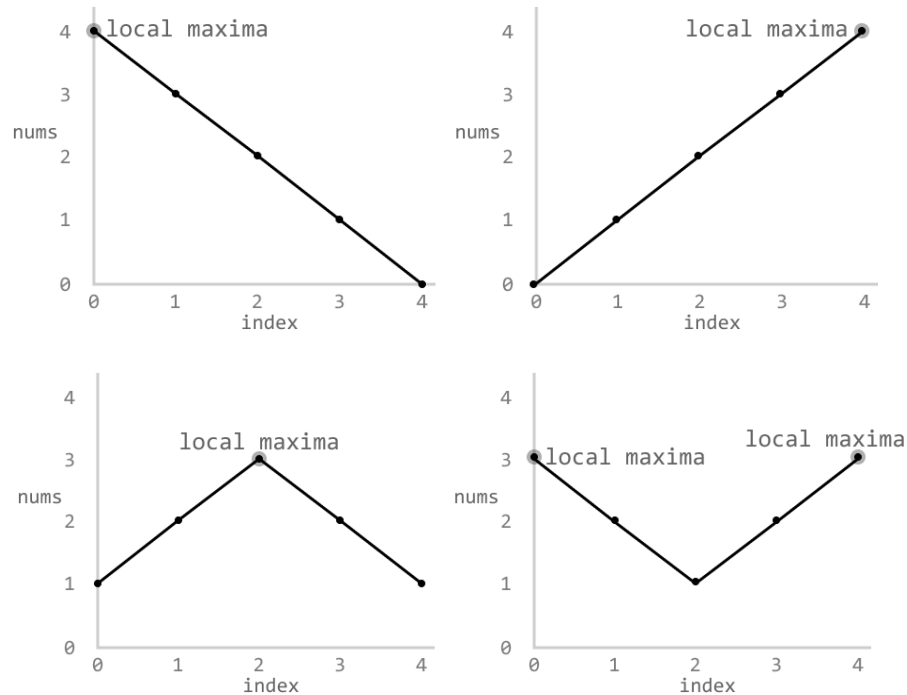
Constraints:

- No two adjacent elements in the array are equal.

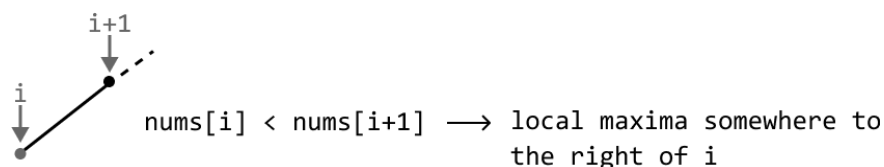
Intuition

A naive way to solve this problem is to linearly search for a local maxima by iteratively comparing each value to its neighbors and returning the first local maxima we find. A linear solution isn't terrible, but since we can return *any* maxima, there's likely a more efficient approach.

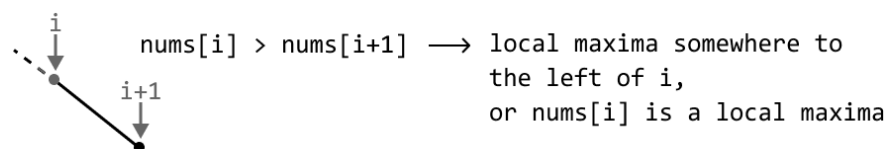
The first important thing to notice is that since this is an array with no adjacent duplicates, **it will always contain at least one local maxima**. If it's not at one of the edges of the array, there'll be at least one somewhere in the middle:



Now, let's say we're at some random index in the array, index i . An interesting observation is that if the next number (at index $i + 1$) is greater than the current, there's definitely a local maxima somewhere to the right of i . This is because the two points at index i and $i + 1$ form an **ascending slope**, and this slope would be heading upward toward some maxima:



The opposite applies if points i and $i + 1$ form a **descending slope**. This would imply a maxima exists somewhere to the left or at i . Notice here that the point at index i itself could be a maxima too:

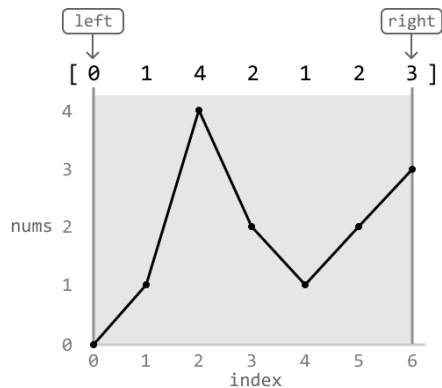


Once we know whether a local maxima exists to the left or to the right, we can continue searching in that direction until we find it. In other words, we narrow our search toward the direction of the maxima. Doesn't this type of reasoning sound similar to how we narrow search space in a binary search? This indicates that it might be possible to find a local maxima using binary search.

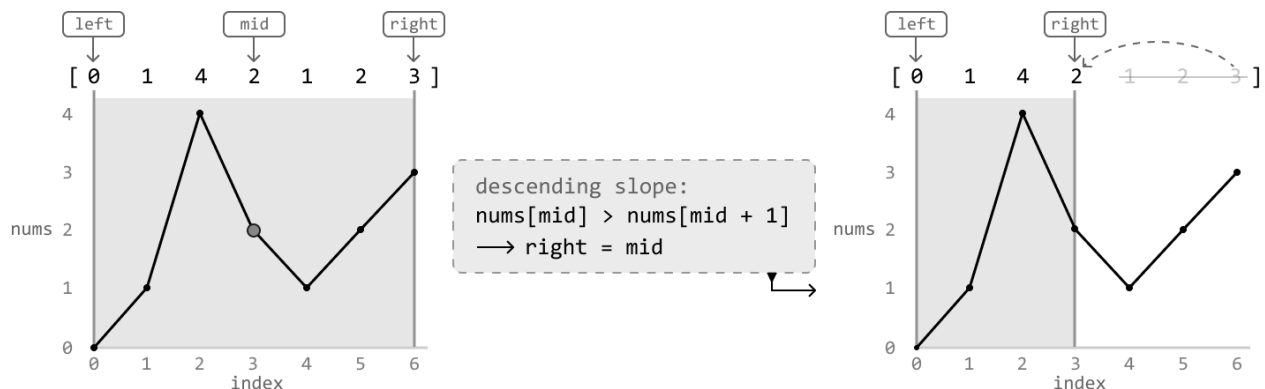
Binary search

First, let's define the **search space**. A local maxima could exist at any index of the array. So, the search space should encompass the entire array.

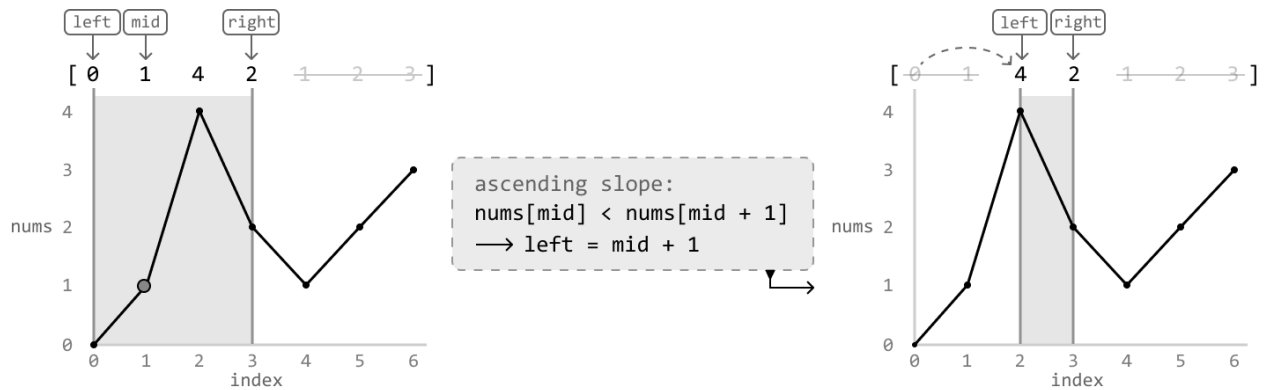
To figure out how we **narrow the search space**, let's use the below example, setting left and right pointers at the boundaries of the array:



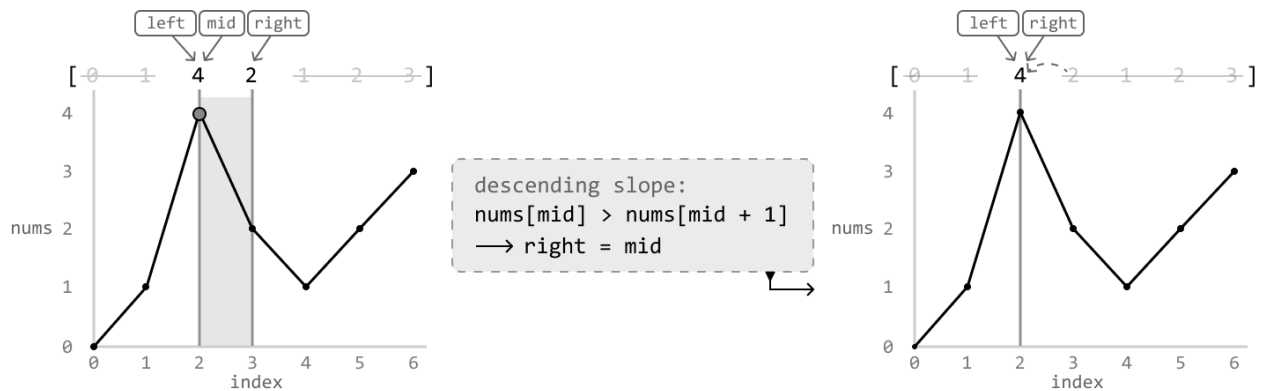
The midpoint is initially set at index 3, which forms a descending slope with its right neighbor since $\text{nums}[\text{mid}] > \text{nums}[\text{mid} + 1]$. This suggests that either a maxima exists to the left of index 3 or that index 3 itself is a maxima). So, we should continue our search to the left, while including the midpoint in the search space:



The next midpoint is set at index 1, which forms an ascending slope with its right neighbor since $\text{nums}[\text{mid}] < \text{nums}[\text{mid} + 1]$. This suggests that a maxima exists somewhere to the right of the midpoint. So, let's continue the search to the right, while excluding the midpoint:



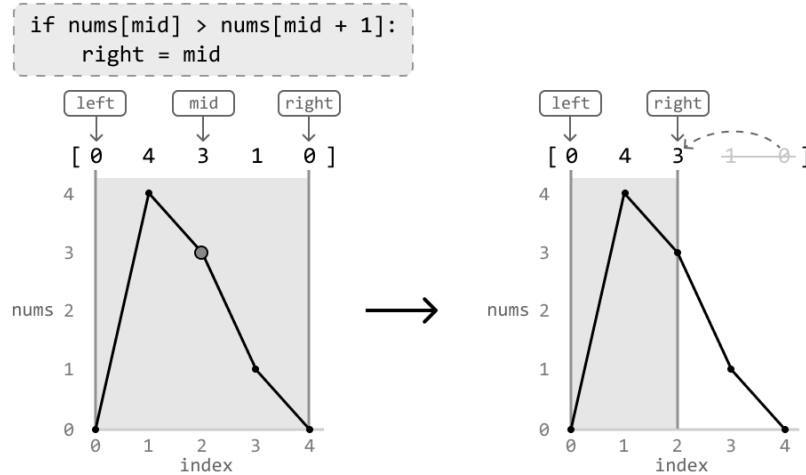
The next midpoint is set at index 2, which forms a descending slope with its right neighbor. So, we continue by searching to the left, while including the midpoint:



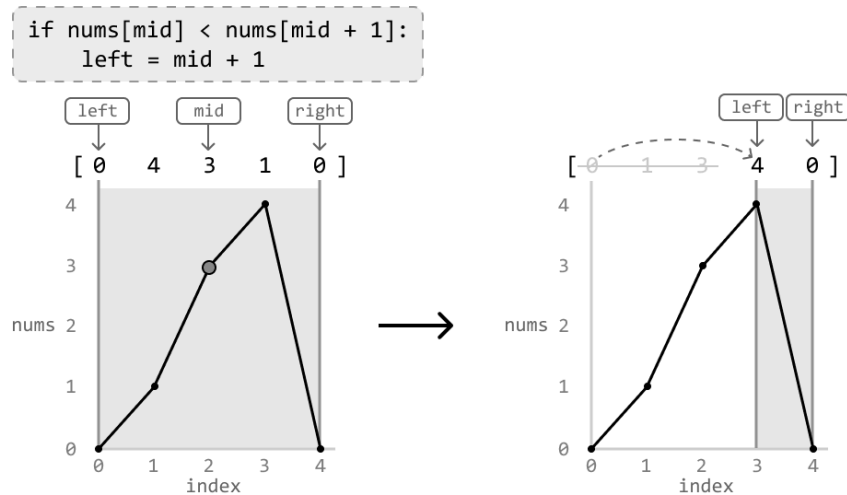
Now that the left and right pointers have met, locating index 2 as a local maxima, we return this maxima's index (left).

Summary

Case 1: The midpoint forms a descending slope with its right neighbor, indicating the midpoint is a local maxima, or that a local maxima exists to the left. Narrow the search space toward the left while including the midpoint:



Case 2: The midpoint forms an ascending slope with its right neighbor, indicating a local maxima exists to the right. Narrow the search space toward the right while excluding the midpoint:



Implementation

```
def local_maxima_in_array(nums: List[int]) -> int:
    left, right = 0, len(nums) - 1
    while left < right:
        mid = (left + right) // 2
        if nums[mid] > nums[mid + 1]:
            right = mid
        else:
            left = mid + 1
```



```
return left
```

Complexity Analysis

Time complexity: The time complexity of `local_maxima_in_array` is $O(\log(n))$, where n denotes the length of the array. This is because we use binary search to find a local maxima.

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

Weighted Random Selection

Given an array of items, each with a corresponding weight, implement a function that randomly selects an item from the array, where the probability of selecting any item is proportional to its weight.

In other words, the probability of picking the item at index i is:
 $\text{weights}[i] / \text{sum}(\text{weights})$.

Return the index of the selected item.

Example:

Input: `weights = [3, 1, 2, 4]`

Explanation:

$\text{sum}(\text{weights}) = 10$

3 has a $3/10$ probability of being selected.

1 has a $1/10$ probability of being selected.

2 has a $2/10$ probability of being selected.

4 has a $4/10$ probability of being selected.

For example, we expect index 0 to be returned 30% of the time.

Constraints:

- The `weights` array contains at least one element.

Intuition

A completely uniform random selection implies every index has an equal chance of being selected. A *weighted* random selection means some items are more likely to be picked than others. If we repeatedly perform a random selection many times, the frequency of each index being picked will match their expected probabilities.

The challenge with this problem is determining a method to randomly select an index based on its probability.

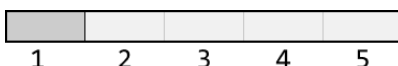
Let's say we had weights 1 and 4 for indexes 0 and 1, respectively:

```
weights = [ 1   4 ]
           [ 0   1 ]
```

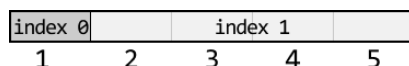
Here, index 1 should be selected with a probability of 4/5, significantly higher than index 0's probability of 1/5:

`weights = [1 4]`
 \downarrow \downarrow
 $\frac{1}{5}$ $\frac{4}{5}$

A useful observation is that all probabilities have the same denominator (which is 5 in this case). Now, imagine we had a line with the same length as this denominator, and we divided this line into two segments of size 1 and 4, respectively:



If we were to randomly pick a number on this line, we'd pick the first segment with a probability of 1/5 and the second segment 4/5 times. Now, imagine index 0 represents the first segment, and index 1 represents the second segment:



If we **randomly select a number on this line**, we'll select index 0 with a probability of 1/5, and index 1 with a probability of 4/5. This reflects their expected probabilities.

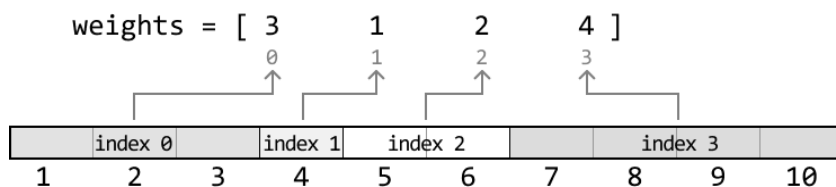
What we need now is a way to identify which numbers on the number line correspond to which index so that when we pick a random number on this line, we know which index to return.

Before we continue, let's establish the definitions of terms used in this explanation:

- "Weights" refers to the values of the elements in the `weights` array.
- "Indexes" refers to the indexes of the `weights` array.
- "Numbers" or "numbers on the number line" refers to the numbers from 1 to `sum(weights)`.

Determining which numbers on the number line correspond to which indexes

As mentioned before, to know which index to return, we need a way to tell which index our random number line number corresponds to. Consider a larger distribution of weights:

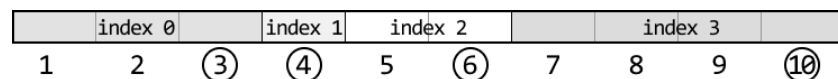


One strategy is to use a hash map. In this hash map, each number on the line is a key, and its corresponding index is the value:

hash map	
1	0
2	0
3	0
4	1
5	2
6	2
7	3
8	3
9	3
10	3
number	index

This method uses a lot of space because we need to store a key-value pair for each number on the number line. Let's consider some other more space-efficient methods.

A more efficient strategy is to store only the **endpoints of each segment** instead.



Naturally, the endpoint of a segment marks where that segment ends. It also helps us know where the next segment begins, as each new segment starts right after the previous one ends. This way, we can determine the start and end of each index's segment.

By storing only the endpoints, we just need to store just n values – one for each endpoint. When storing these endpoints in an array, the array index itself is the same as the endpoint's associated index value on the number line:

[3	4	6	10]
	0	1	2	3	

The question now is, how do we find these endpoints?

Obtaining the endpoints of each index's segment on the line

A key observation is that the endpoint of a segment is equal to the length of all previous segments, plus the length of the current segment. We can see how this works below:

	index 0	index 1	index 2		index 3	
1	2	3	4	5	6	7
		↓	↓		↓	
		3	3+1		3+1+2	
						3+1+2+4
						10

This demonstrates that each endpoint is a cumulative sum, suggesting we can obtain the endpoint of each segment by obtaining the **prefix sums** of the array of weights:

```
weights = [ 3    1    2    4 ]
prefix_sums = [ 3    4    6   10 ]
```

As we can see, the prefix sum array stores the endpoint of each segment.

Now, let's see how the prefix sum array helps us. When we pick a random number from 1 to 10, we need to determine which index it corresponds to using the prefix sum array. Let's see how we can do this.

Using the prefix sums to determine which numbers correspond to which indexes

Let's say we pick a random number from 1 to 10 and get 5. How can we use the prefix sum array to determine which index that 5 corresponds to? To determine the segment, we'll need to find its corresponding endpoint. We know that:

- Either 5 itself is the endpoint, since 5 could be the endpoint of its own segment, or:
- The endpoint is somewhere to the right of 5 since its endpoint cannot be to the left.

Among all endpoints to the right of 5, the closest one to 5 will be the endpoint of its segment. Endpoints farther away belong to different segments:

target = 5

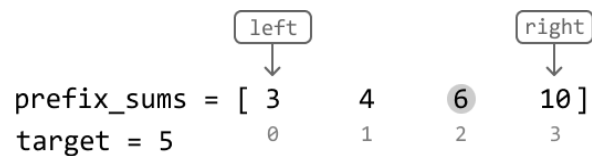
	index 0	index 1	index 2		index 3	
1	2	③	④	⑤	⑥	7
					↑	
					closest endpoint to the right of 5	
						⑩

This means for any target, we're looking for the **first prefix sum (endpoint) greater than or equal to the target**. Below, we can see which prefix sum first meets this condition for a target of 5:

```
prefix_sums = [ 3    4    6   10 ]
               F    F    T    T
```

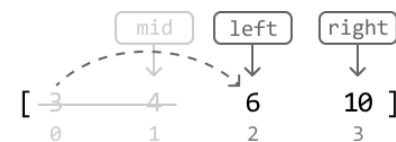
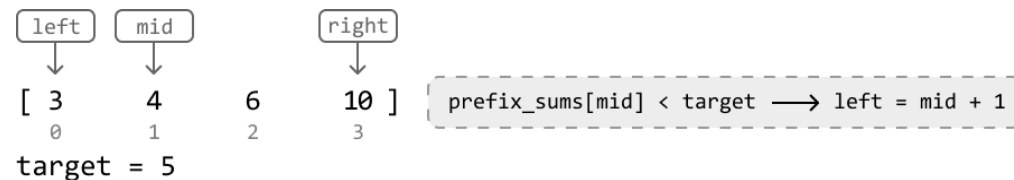
Therefore, the first prefix sum that satisfies this condition is the same as the lower-bound prefix sum that satisfies this condition. Therefore, we can perform a **lower-bound binary search** to find it, since the prefix sum array will always be a sorted array in this problem.

Let's see how this works over our example with a random target of 5. The **search space** should encompass all prefix sum values:

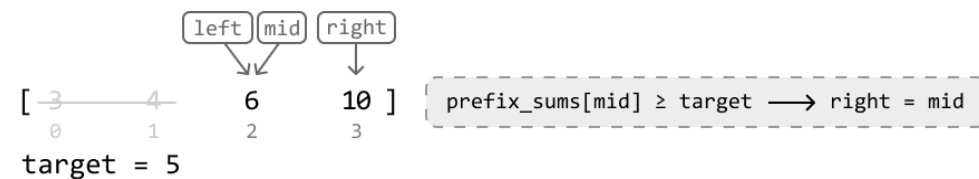


Let's begin narrowing the search space. Remember, we're looking for the lower-bound prefix sum which satisfies the condition `prefix_sums[mid] ≥ target`.

The initial midpoint value is 4, which is less than the target of 5. This means the lower bound is somewhere to the right of the midpoint, so let's narrow the search space toward the right:



The midpoint value is now 6, which is greater than the target. This midpoint satisfies our condition, so it could be the lower bound. If it isn't, then the lower bound is somewhere to the left. So, let's narrow the search space toward the left while including the midpoint:



Now, the left and right pointers have met. So, we can exit the binary search and return the index that corresponds to the final prefix sum in the search space: left:



Implementation

```
class WeightedRandomSelection:
    def __init__(self, weights: List[int]):
        self.prefix_sums = [weights[0]]
        for i in range(1, len(weights)):
            self.prefix_sums.append(self.prefix_sums[-1] + weights[i])

    def select(self) -> int:
        # Pick a random target between 1 and the largest possible endpoint.
        target = random.randint(1, self.prefix_sums[-1])
        left, right = 0, len(self.prefix_sums) - 1
        # Perform lower-bound binary search to find which endpoint (i.e., prefix
        # sum value) corresponds to the target.
        while left < right:
            mid = (left + right) // 2
            if self.prefix_sums[mid] < target:
                left = mid + 1
            else:
                right = mid
        return left
```

Complexity Analysis

Time complexity: The time complexity of the constructor is $O(n)$ because we iterate through each weight in the weights array once. The time complexity of select is $O(\log(n))$ since we perform binary search over the prefix_sums array.

Space complexity: The space complexity of the constructor is $O(n)$ due to the prefix_sums array. The space complexity of select is $O(1)$.

Stacks

Repeated Removal of Adjacent Duplicates

Given a string, continually perform the following operation: remove a pair of adjacent duplicates from the string. Continue performing this operation until the string no longer contains pairs of adjacent duplicates. Return the final string.

Example 1:

~~a~~~~a~~ c a ~~b~~~~b~~ a → c ~~a~~~~a~~ → c

Input: s = "aacabba"

Output: "c"

Example 2:

~~a~~~~a~~ a → a

Input: s = "aaa"

Output: "a"

Intuition

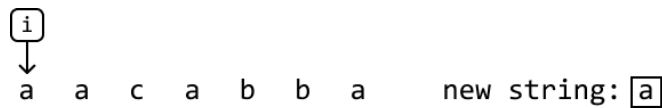
One challenge in solving this problem is how we handle characters which aren't currently adjacent duplicates but will be in the future.

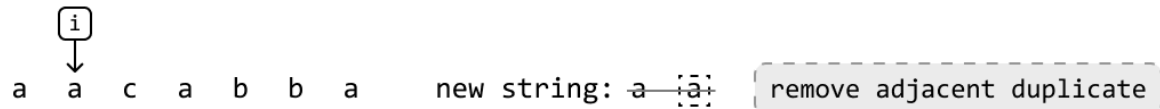
A solution we can try is to iteratively **build the string character by character** and immediately remove each pair of adjacent duplicates that get formed as we're building the string.

It's also possible an adjacent duplicate may be formed after another adjacent duplicate gets removed. For example, with the string "abba", removing "bb" will result in "aa". Building the string character by character ensures the formation of "aa" gets noticed and removed. To better understand how this works, let's dive into an example.

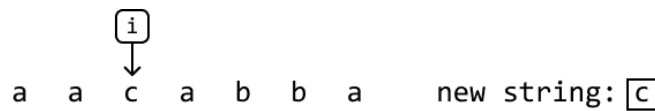
Consider the following string:

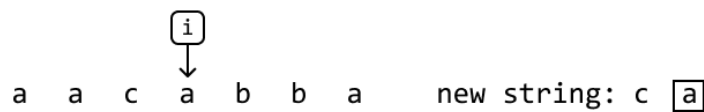
a a c a b b a

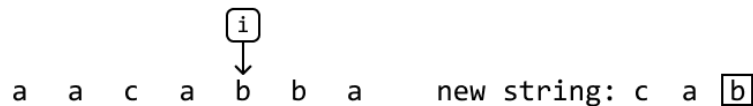

a a c a b b a new string: a

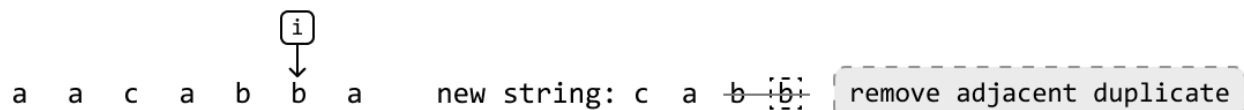

a a c a b b a new string: a ~~a~~ remove adjacent duplicate

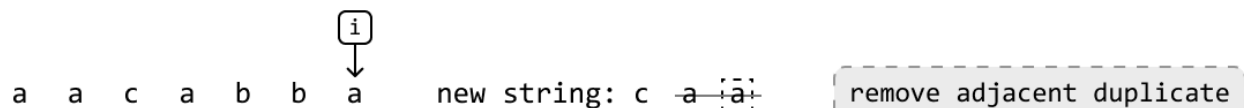
At the second 'a', we notice that adding it would result in an adjacent duplicate forming (i.e., "aa"). So, let's remove this duplicate before adding any new characters. We'll do this for all adjacent duplicates we come across as we build the string:


a a c a b b a new string: c


a a c a b b a new string: c a


a a c a b b a new string: c a b


a a c a b b a new string: c a ~~b~~ remove adjacent duplicate


a a c a b b a new string: c a ~~a~~ remove adjacent duplicate

Once the smoke clears, the resulting string we were building ends up being "c", which is the expected output.

Now that we know how this strategy works, we just need a data structure that'll allow us to:

1. Add letters to one end of it.
2. Remove letters from the same end.

The **stack** data structure is a strong option because it allows for both operations.

As we push characters onto the stack, the top of the stack will represent the previous/most recently added character. Given this, to mimic the process of building the “new string” as shown in the example, we:

- Push the current character onto the stack if it's different from the character at the top (i.e., not a duplicate character.)
- Pop off the character at the top of the stack if it's the same as the current character (i.e., a duplicate.)

Once all characters have been processed, the last thing to do is return the content of the stack as a string, since the final state of the stack will contain all characters that weren't removed.

Implementation

```
def repeated_removal_of_adjacent_duplicates(s: str) -> str:
    stack = []
    for c in s:
        # If the current character is the same as the top character on the stack,
        # a pair of adjacent duplicates has been formed. So, pop the top character
        # from the stack.
        if stack and c == stack[-1]:
            stack.pop()
        # Otherwise, push the current character onto the stack.
        else:
            stack.append(c)
    # Return the remaining characters as a string.
    return ''.join(stack)
```

Complexity Analysis

Time complexity: The time complexity of `repeated_removal_of_adjacent_duplicates` is $O(n)$, where n denotes the length of the string. This is because we traverse the entire string, and we perform a join operation of up to n characters in the stack. The stack push and pop operations contribute $O(1)$ time.

Space complexity: The space complexity is $O(n)$ because the stack can store at most n characters. Note that the space taken up by the final output string is not considered in the space complexity.

Implement a Queue Using Stacks

Implement a queue using the stack data structure. Include the following functions:

- **enqueue**(x: int) -> None: adds x to the end of the queue.
- **dequeue**() -> int: removes and returns the element from the front of the queue.
- **peek**() -> int: returns the front element of the queue.

You may not use any other data structures to implement the queue.

Example:

Input: [enqueue(1), enqueue(2), dequeue(), enqueue(3), peek()]

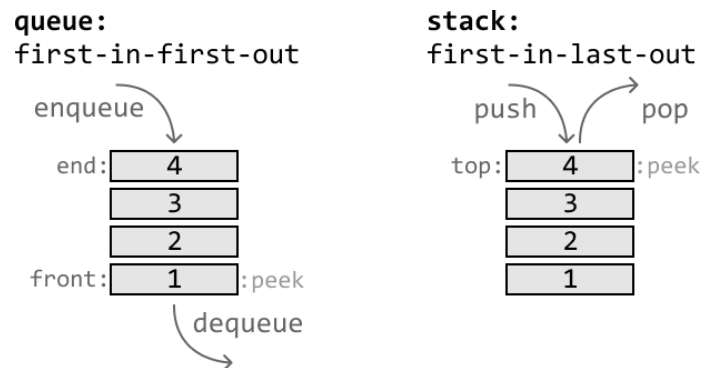
Output: [1, 2]

Constraints:

- The dequeue and peek operations will only be called on a non-empty queue.

Intuition

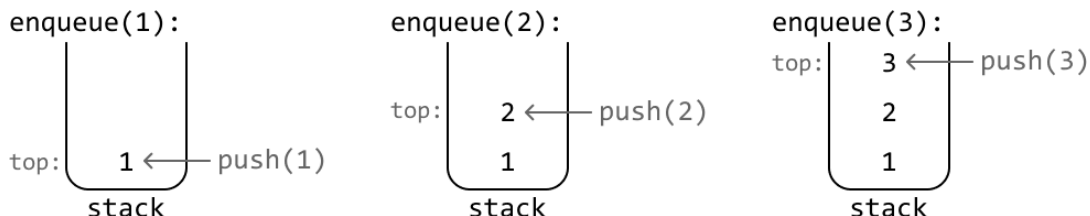
A queue is a first-in-first-out (FIFO) data structure, whereas stacks are a first-in-last-out (FILO) data structure:



The main difference between these data structures is how items are evicted from them. In a queue, the first value to enter is the first to leave, whereas it would be the last to leave in a stack.

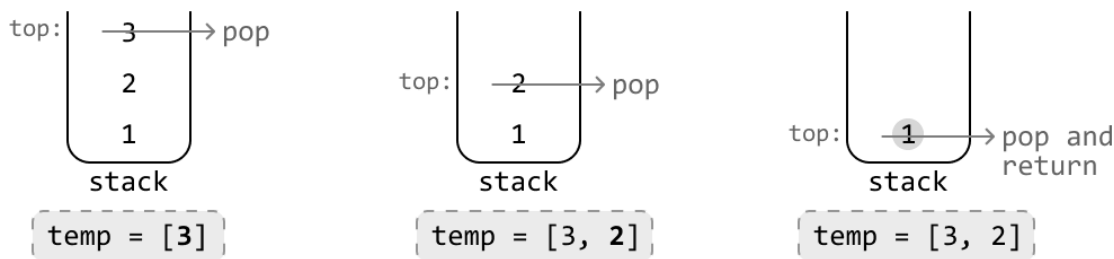
Now that we understand how they work, let's dive into the problem. Let's start by seeing if it's possible to replicate the functionality of a queue with just one stack.

Consider the following stack where we push values 1, 2, and 3 to it after receiving enqueue(1), enqueue(2), and enqueue(3), respectively:

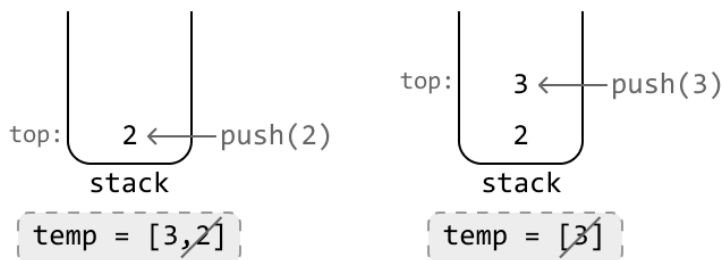


We now encounter a problem with attempting a dequeue operation since popping off the top of the stack would return 3. The value we actually want popped off is 1, since it was the first value that entered the data structure. However, 1 is all the way at the bottom of the stack.

To get to the bottom, we need to pop off all the values from the top of the stack and temporarily store these values in a separate data structure (temp) so we can add them back to the stack later:



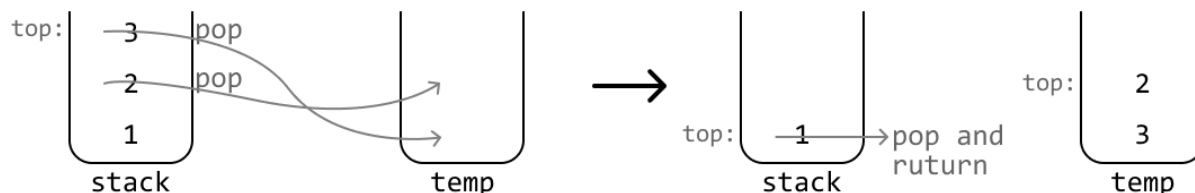
Once we've popped and returned the bottom value (1), push the values stored in temp back onto the stack in reverse order to ensure they're added back correctly:



We know that if we were to use a data structure such as temp, **it'd have to be a stack**, as the problem specifies only stacks can be used. In this temporary data structure, we remove values in the opposite order in which we added them. In other words, it follows the LIFO principle, which is conveniently how a stack works. This means we can use a stack for our temporary storage.

Now, even though we have a solution that works, having to pop off every single value from the top of the stack whenever we want to access the bottom value is quite time-consuming. To find a way

around this, let's have a closer look at the state of our two stacks right after we've moved the stack values to temp:



In our initial solution, we would now move the values from temp back to the main stack. However, notice the top of the temp stack now contains the value we expect to return in the next dequeue call. This is because it's the second value to have entered the data structure, and according to the FIFO eviction policy, it should be the next one to be removed.

So, instead of adding these values back to the main stack, we could just leave them in temp and return the stack's top value at the next dequeue call.

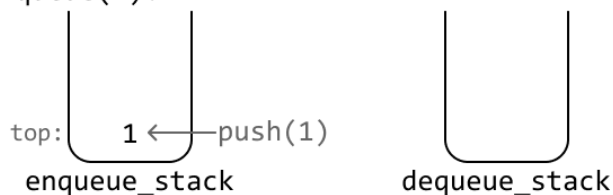
In the above logic, we ended up using two stacks which each serve a unique purpose. In particular, we used:

1. A stack to push values onto during each enqueue call (enqueue_stack).
2. A stack to pop values from during each dequeue call (dequeue_stack).

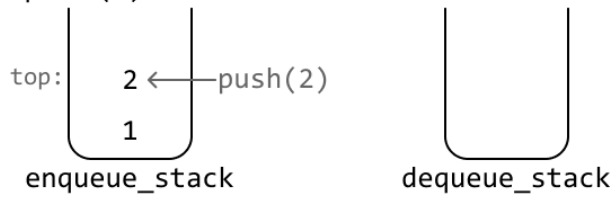
An important thing to realize here is that the dequeue stack won't always be populated with values. So, what should we do when it's empty? We can just populate it by transferring all the values from the enqueue stack to the dequeue stack, just like we did in the example. To understand this more clearly, let's dive into a full example.

Let's start with two enqueue calls and push each number onto the enqueue stack.

enqueue(1):

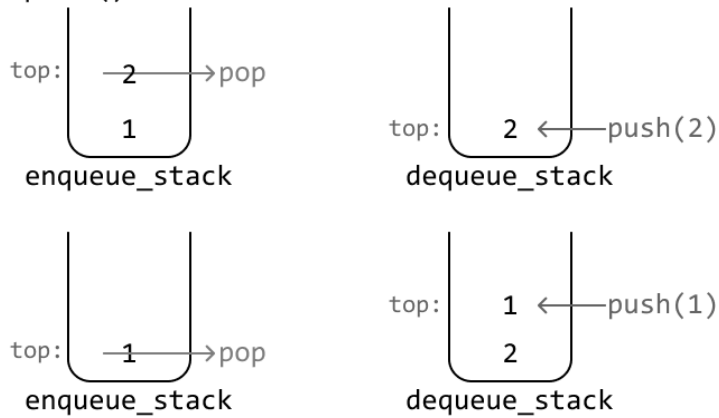


enqueue(2):

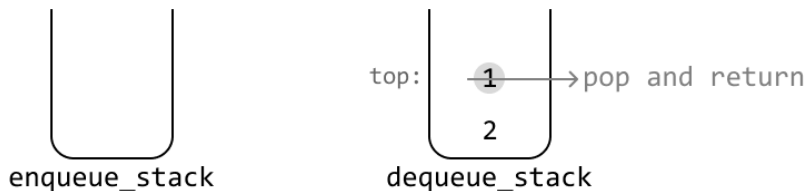


Now, let's try processing a dequeue call. The first step is to pop off each element from the enqueue stack and push them onto the dequeue stack:

dequeue():

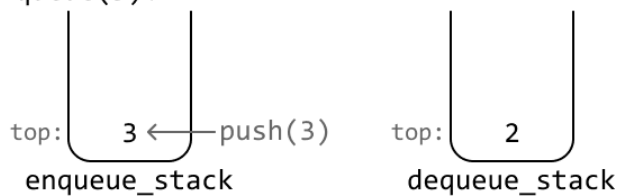


Then, we just return the top value from the dequeue stack:

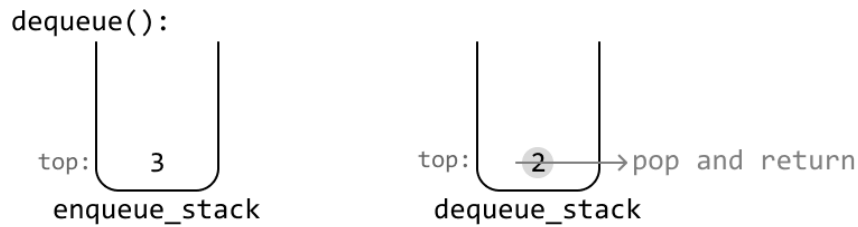


Let's enqueue one more value:

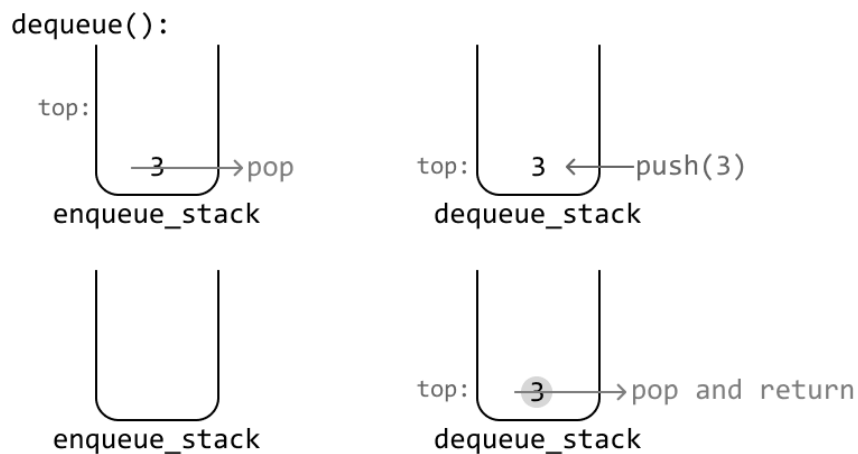
enqueue(3):



If we call dequeue again, we return the value from the top of the dequeue stack:



Now, what happens when we call dequeue and the dequeue stack is empty? We need to repopulate it by popping all the values from the enqueue stack and pushing them into the dequeue stack. Once this is done, we return the top of the dequeue stack as usual:



Regarding the peek function, we follow the same logic as the dequeue function, but instead, we return the top element of the dequeue stack without popping it.

Implementation

As mentioned before, the dequeue and peek functions have mostly the same behavior, with the only difference being that dequeue pops the top value while peek does not. To avoid duplicate code, the shared logic between these functions of transferring values from the enqueue stack to the dequeue stack has been extracted into a separate function, `transfer_enqueue_to_dequeue`.

```
class Queue:
    def __init__(self):
        self.enqueue_stack = []
        self.dequeue_stack = []

    def enqueue(self, x: int) -> None:
```

```

self.enqueue_stack.append(x)

def transfer_enqueue_to_dequeue(self) -> None:
    # If the dequeue stack is empty, push all elements from the enqueue stack
    # onto the dequeue stack. This ensures the top of the dequeue stack
    # contains the least recently added value.
    if not self.dequeue_stack:
        while self.enqueue_stack:
            self.dequeue_stack.append(self.enqueue_stack.pop())

def dequeue(self) -> int:
    self.transfer_enqueue_to_dequeue()
    # Pop and return the value at the top of the dequeue stack.
    return self.dequeue_stack.pop() if self.dequeue_stack else None

def peek(self) -> int:
    self.transfer_enqueue_to_dequeue()
    return self.dequeue_stack[-1] if self.dequeue_stack else None

```

Complexity Analysis

Time complexity: The time complexity of:

- enqueue is $O(1)$ because we add one element to the enqueue stack in constant time.
- dequeue is amortized $O(1)$.
 - In the worst case, all elements from the enqueue stack are moved to the dequeue stack. This takes $O(n)$ time, where n denotes the number of elements currently in the queue.
 - However, each element is only ever moved once during its lifetime. So, over n dequeue calls, at most n elements are moved between stacks, averaging the cost to $O(1)$ time per dequeue operation.
- peek is amortized $O(1)$ for the same reasons as dequeue.

Space complexity: The space complexity is $O(n)$ since we maintain two stacks that collectively store all elements of the queue at any given time.

Maximums of Sliding Window

There's a sliding window of size k that slides through an integer array from left to right. Create a new array that records the largest number found in each window as it slides through.

Example:

[3 2 4 1 2 1 1] max = 4

[3 2 4 1 2 1 1] max = 4

[3 2 4 1 2 1 1] max = 4

[3 2 4 1 2 1 1] max = 2

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

Output: [4, 4, 4, 2]

Intuition

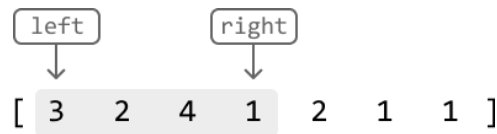
A brute-force approach to solving this problem involves iterating through each element within a window to find the maximum value of that window. Repeating this for each window will take $O(n \cdot k)$ time because we traverse k elements for up to n windows.

The main issue is that as we slide the window, we keep re-examining the same elements we've already looked at in previous windows. This is because two adjacent windows share mostly the same values.

[3		2	4	1		2		1		1]
			same values									
[3		2	4	1		2		1		1]

A more efficient solution likely involves keeping track of values we see in any given window so that at the next window, we don't have to iterate over previously seen values again. Specifically, at each window, we should maintain a record of only values that have the potential to become the maximum of a future window. Let's call these values **candidates**, where all values that aren't candidates can no longer contribute to a maximum. How can we determine which numbers are candidates?

Consider the window of size 4 in the following array. We'll use left and right pointers to define the window:



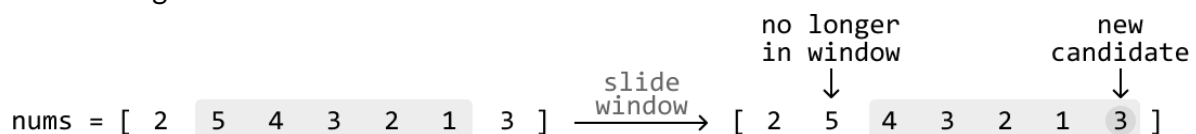
To identify the candidates for the next window, let's look at each number individually.

- 3: Number 3 is a candidate for the current window, but once we move to the next window, we can ignore it since it will no longer be included in the window.
- 2: Could number 2 be a maximum of a future window? The answer is no. This is because of the 4 to its right: all future windows which contain this 2 will also contain 4, and since 4 is larger, it means 2 could never be a maximum of any future windows.
- 4: This is the maximum value in the current window, and it'll be included in some future windows. Therefore, 4 could potentially be a maximum for a future window.
- 1: Could 1 become the maximum of a future window? The answer is yes. While 4 is larger in the current window, it's positioned to the left of 1. As the window shifts to the right, there will eventually be a point where 1 remains in the window while 4 is excluded, making 1 a potential maximum in the future.

Based on the above analysis, we can derive the following strategy whenever the window encounters a new candidate:

1. **Remove smaller or equal candidates:** Any existing candidates less than or equal to the new candidate should be discarded because they can no longer be maximums of future windows.
2. **Adding the new candidate:** Once smaller candidates are discarded, the new value can be added as a new candidate.
3. **Removing outdated candidates:** When the window moves past a value, that value should be discarded to ensure we don't consider values outside the window.

Observe how this strategy is applied to the list of candidates below as the window advances one index to the right:



```
list of candidates: [ 5  4  3  2  1 ]
1) remove values ≤ new candidate (3):  [ 5  4  3 2 1 ]
2) add the new candidate:                [ 5  4  3 ]
3) remove outdated values:              [5  4  3 ]
```

An important observation is that **candidate values always maintain a decreasing order**. This is because each new candidate we encounter removes smaller and equal candidates to its left, ensuring the list of candidates is kept in a decreasing order.

This consequently means **the maximum value for a window is always the first value in the candidate list**.

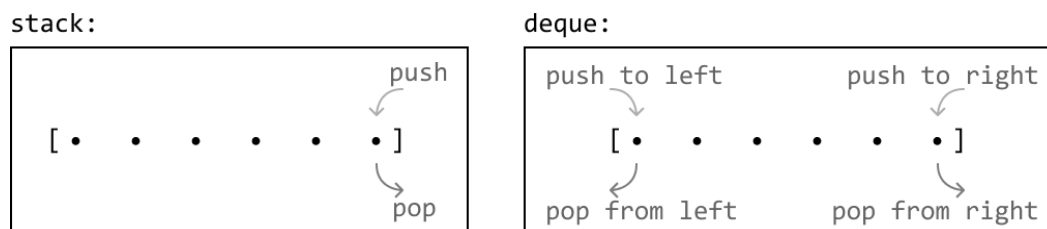
Therefore, to store the candidates, we need a data structure that can maintain a **monotonic decreasing order** of values.

Deque

We know that typically, a stack allows us to maintain a monotonic decreasing order of values, but in this case, it has a critical limitation: it doesn't provide a way to remove outdated candidates. A stack is a last-in-first-out (LIFO) data structure, which means we only have access to the last (i.e., most recent) end of the data structure. From the diagram above, we know we need access to both ends of the list of candidates, so a stack won't be sufficient.

Is there a data structure that allows us to add and remove from both ends? A **double-ended queue**, or deque for short, is a great candidate for this. A deque is essentially a doubly linked list under the hood. It allows us to push and pop values from both ends of the data structure in $O(1)$ time.

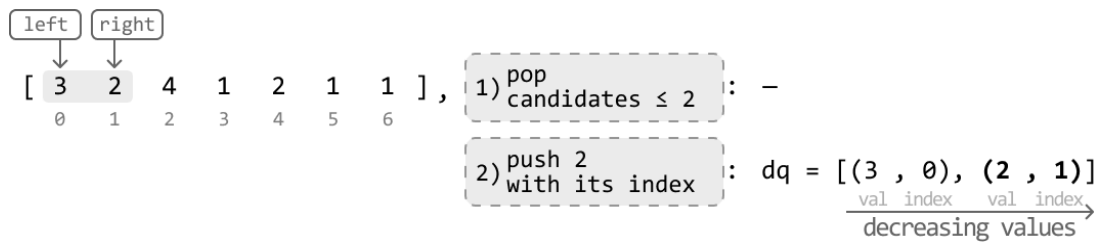
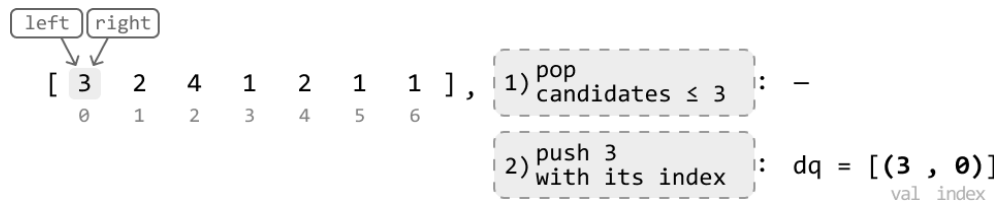
Despite its name, it's easier to think of a deque as a double-ended stack:



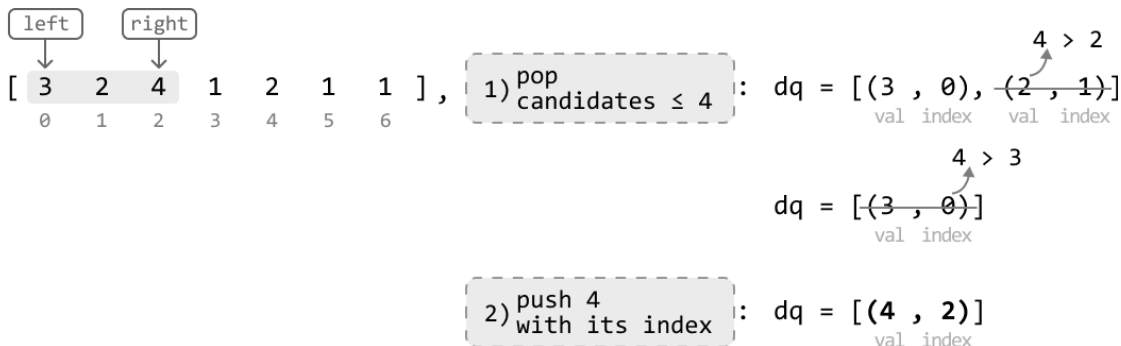
Now that we have our data structure, let's see how we can use it over the following example. Note that **our deque will store tuples containing both a value and its corresponding index**. We keep track of indexes in the deque because they allow us to determine whether a value has moved outside the window. We'll see how this works later in the example.

```
[ 3  2  4  1  2  1  1 ] ,  dq = []
  0  1  2  3  4  5  6
```

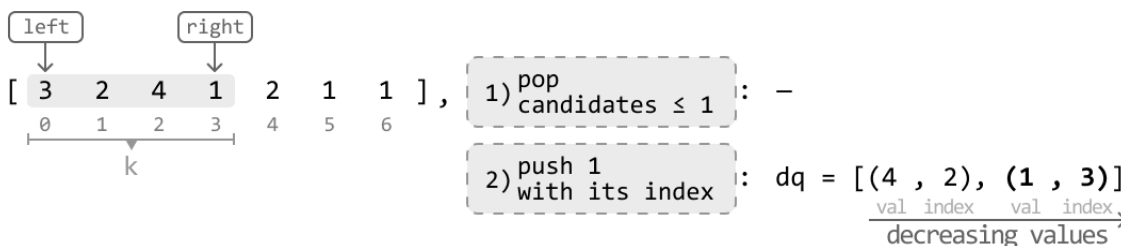
Start by expanding the window until it's of length k. For each candidate we encounter, we need to ensure the values in the deque maintain a monotonic decreasing order before pushing it in:



When we reach 4, we can't add it to the deque straight away because adding it would violate the decreasing order of the deque. So, let's pop any candidates from the right of the deque that are less than 4 before pushing 4 in.

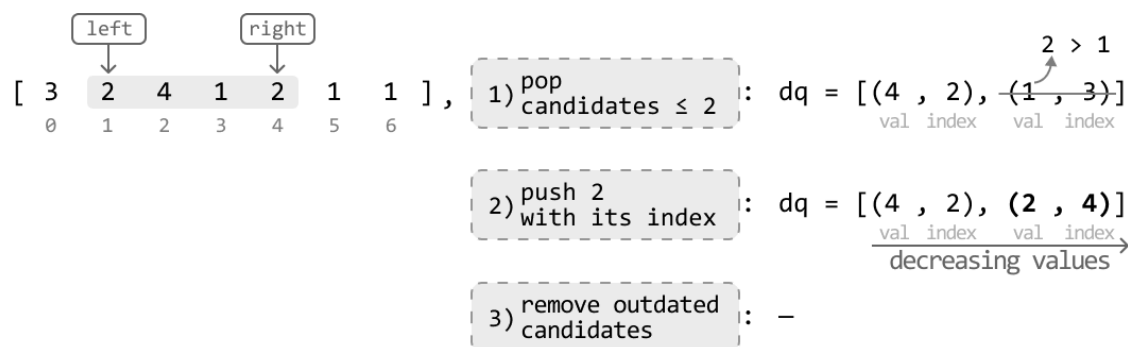


The next expansion of the window will set it at the expected fixed size of k:

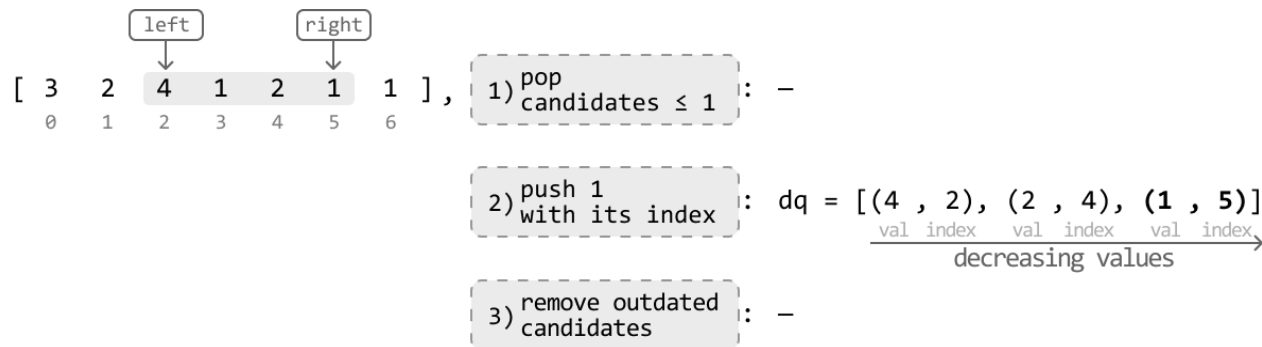


Now that the window is of length k , it's time to begin recording the maximum value of each window. As mentioned earlier, the maximum value is the first value in the candidate list (i.e., the leftmost value of the deque.) The maximum value of this window is 4 since it's the leftmost candidate value.

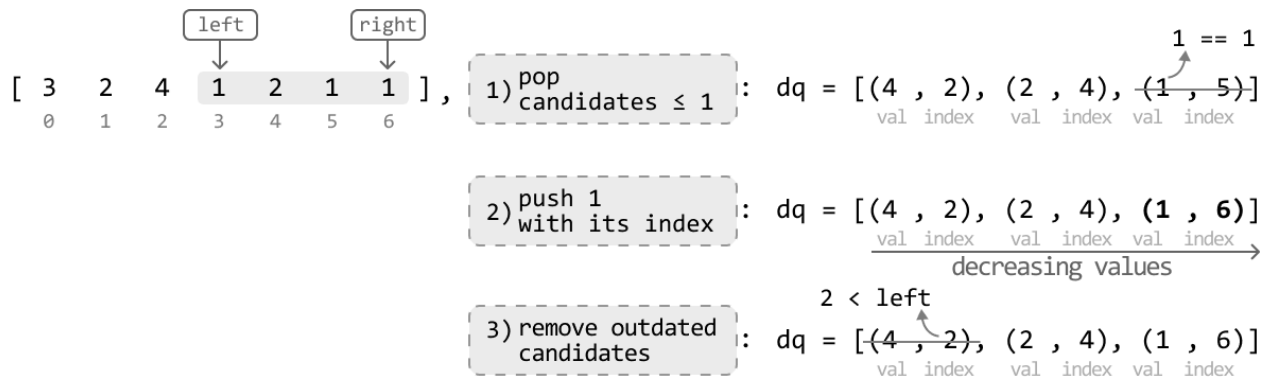
With the window now at a fixed size of k , our approach shifts from expanding the window to sliding it. As we slide, we should remove any values from the deque whose index is before the left pointer, since those values will be outside the window:



The maximum value of the above window after the three operations are performed is revealed to be 4, as it's the leftmost candidate value.



The maximum value of the above window after the three operations are performed is also 4, as it's the leftmost candidate value.



Before recording the maximum value of this window, we ensured 4 was removed from the deque because its index (index 2) occurs before the left pointer (index 3), indicating it's no longer within the current window. The maximum value for this window after this removal is revealed to be 2.

Implementation

```
def maximums_of_sliding_window(nums: List[int], k: int) -> List[int]:
    res = []
    dq = deque()
    left = right = 0
    while right < len(nums):
        # 1) Ensure the values of the deque maintain a monotonic decreasing order
        # by removing candidates  $\geq$  the current candidate.
        while dq and dq[-1][0] <= nums[right]:
            dq.pop()
        # 2) Add the current candidate.
        dq.append((nums[right], right))
        # If the window is of length 'k', record the maximum of the window.
        if right - left + 1 == k:
            # 3) Remove values whose indexes occur outside the window.
            if dq and dq[0][1] < left:
                dq.popleft()
            # The maximum value of this window is the leftmost value in the
            # deque.
            res.append(dq[0][0])
        # Slide the window by advancing both 'left' and 'right'. The right
```

```
        # pointer always gets advanced so we just need to advance 'left'.
        left += 1
        right += 1
    return res
```

Complexity Analysis

Time complexity: The time complexity of `maximums_of_sliding_window` is $O(n)$ because we slide over the array in linear time, and we push and pop values of `nums` into the deque at most once for each number, with each stack operation taking $O(1)$ time.

Space complexity: The space complexity is $O(k)$ because the deque can store up to k elements. Note, we don't consider `res` in the space complexity.

Interview Tip

Tip: If you're unsure about what data structure to use for a problem, first identify what attributes or operations you want from the data structure.



Use these attributes and operations to pinpoint a data structure that satisfies them and can be used to solve the problem. In this problem, we wanted a data structure that could add and remove elements from both ends of it efficiently, and the best data structure that matched these requirements was a deque.

Heaps

Sort a K-Sorted Array

Given an integer array where each element is at most k positions away from its sorted position, sort the array in a non-decreasing order.

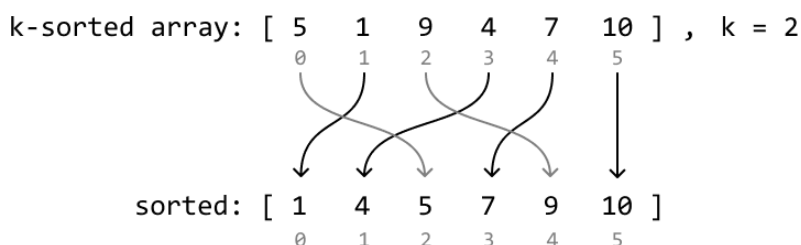
Example:

Input: `nums = [5, 1, 9, 4, 7, 10]`, $k = 2$

Output: `[1, 4, 5, 7, 9, 10]`

Intuition

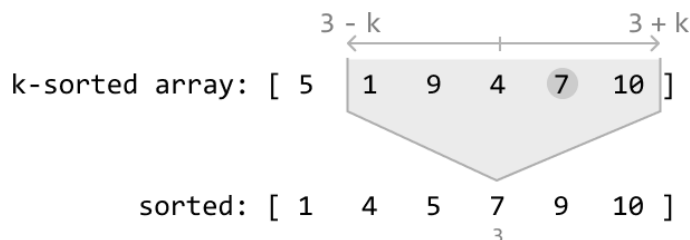
In a k -sorted array, each element is at most k indexes away from where it would be in a fully sorted array. We can visualize this with the following example where $k = 2$, and no number is more than k indexes away from its sorted position:



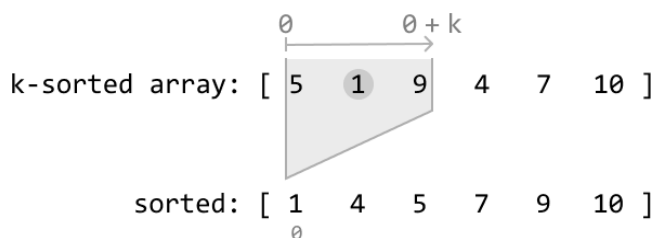
A trivial solution to this problem is to sort the array using a standard sorting algorithm. However, since the input is partially sorted (k -sorted), we should assume there's a faster way to sort the array.

We can think about this problem backward. For any index i , **the element that belongs at index i in the sorted array is located within the range $[i - k, i + k]$** . Below, we visualize how number 7,

which is meant to be at index 3 when sorted, correctly falls within the range $[3 - k, 3 + k]$ in the k-sorted array:

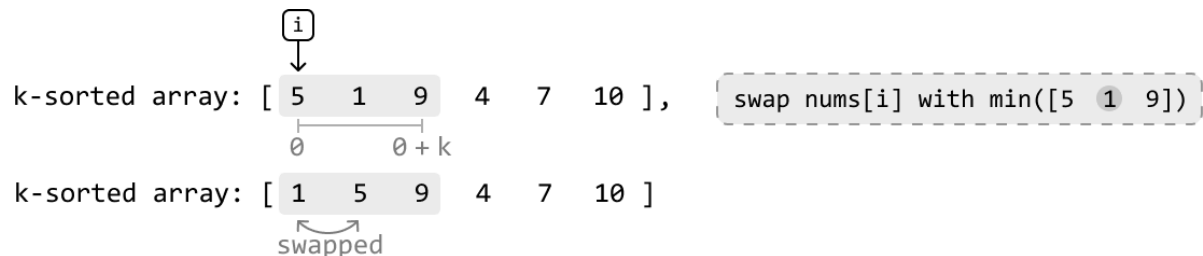


This is a good start, but we can reduce this range even further. Consider index 0 from the above array. We know the number which belongs at index 0 when sorted is somewhere in the range $[0, 0 + k]$:



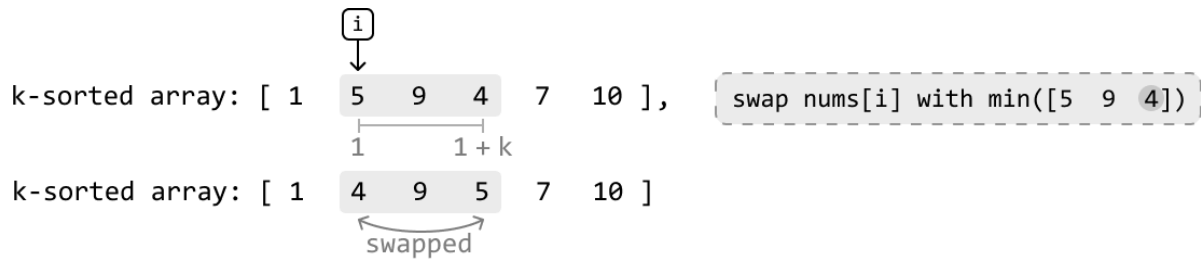
Note that the sorted array in the diagrams is purely provided as a reference point. We don't yet know which number in the range $[0, 0 + k]$ belongs at index 0. However, one fact remains consistent: in a sorted array, index 0 always holds the smallest number. This means the value needed at index 0 is also the smallest number within the range $[0, 0 + k]$ of the k-sorted array, which is 1 in this example.

So, let's swap 1 with the number at index 0 to position 1 as the first value in the sorted array:



Now let's find the number that belongs at index 1: the second smallest number. Since index 0 currently contains the smallest value in the array, we won't need to consider index 0 in our search. Therefore, we can find the value that belongs at index 1 in the range $[1, 1 + k]$. The smallest value in this range will be the second smallest value overall, which is 4 in this case.

So, let's swap 4 with the number at index 1 to position 4 as the second value in the sorted array:



If we continue this process for the rest of the array, we'll successfully sort the k-sorted array.

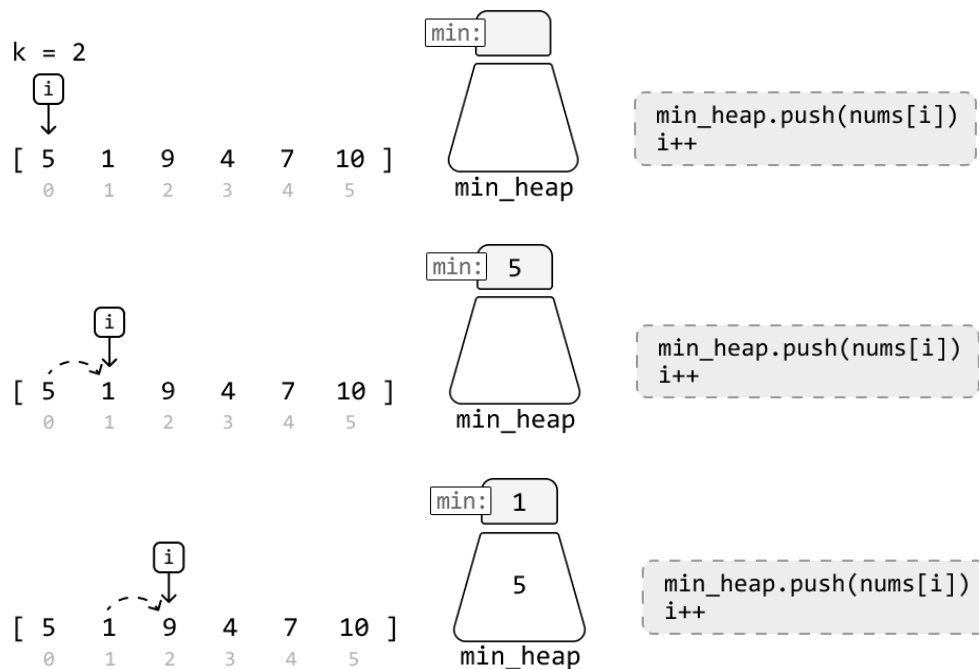
The main inefficiency with this approach is finding the minimum number in the range $[i, i + k]$ at each index i . Linearly searching for it will take $O(k)$ time at each index.

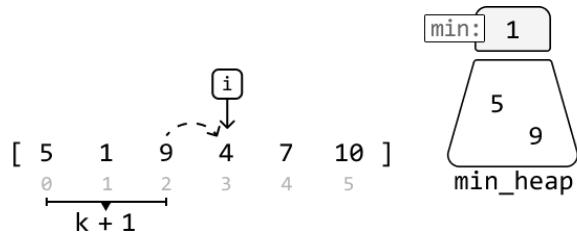
To improve this approach, we'd need a way to efficiently access the minimum value at each of these ranges. A **min-heap** would be perfect for this.

Min-heap

For a min-heap to determine the minimum value within each range $[i, i + k]$, it will always need to be populated with the values in these ranges as we iterate through the array. Let's see how this works over the same example.

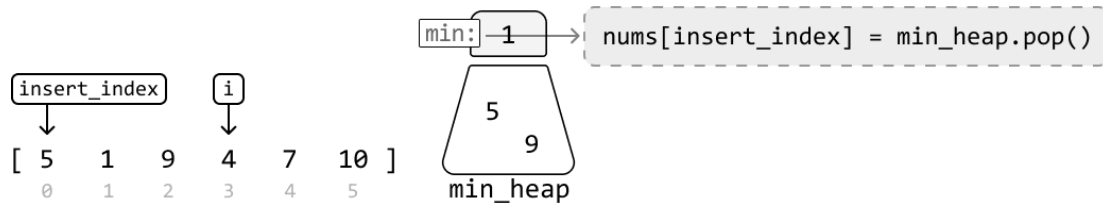
Before we can determine which value belongs at index 0, we'll need to populate the heap with all the values in the range $[0, k]$, which are the first $k + 1$ values (where $k = 2$ in this example):



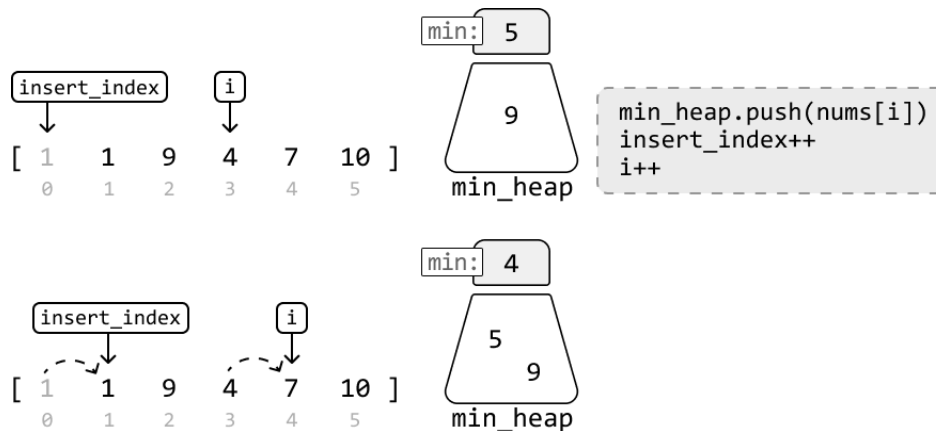


An alternative way to create a heap of the first $k + 1$ elements is to **heapify** a list of the first $k + 1$ elements.

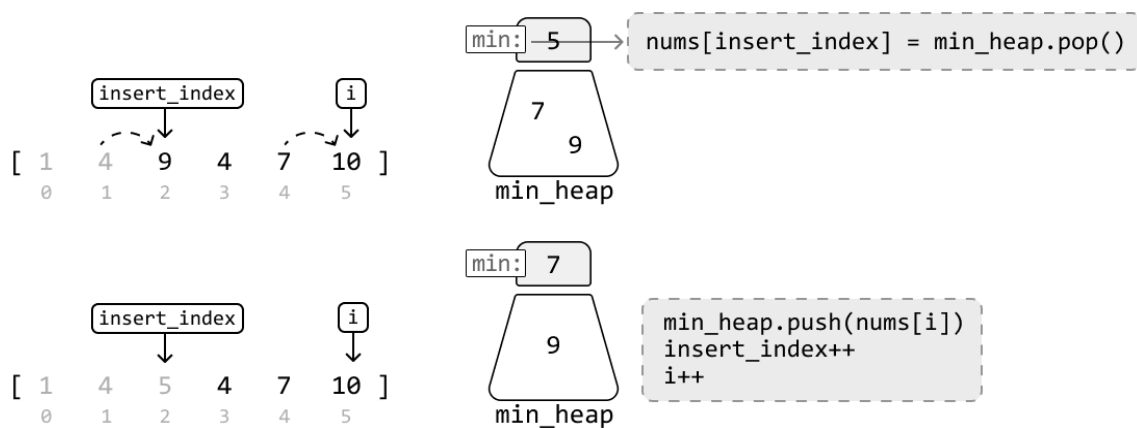
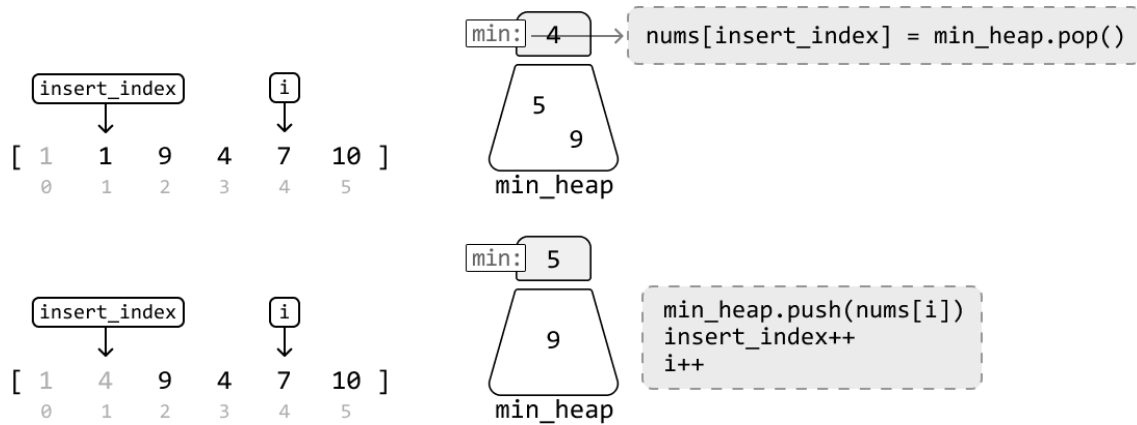
Now, let's begin inserting the smallest elements from the heap into the array, using the `insert_index` pointer. The value that belongs at index 0 in sorted order is the value currently at the top of the heap, which is 1:



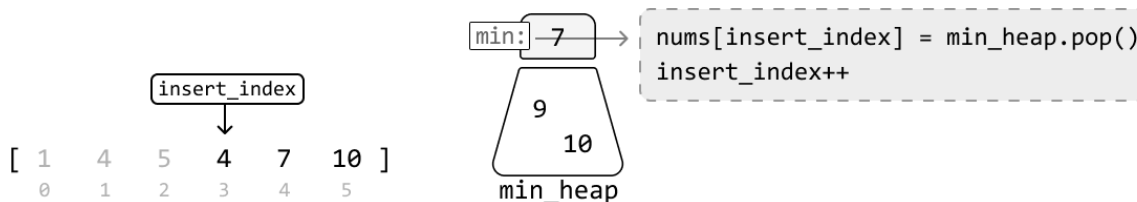
Once we insert 1 at index 0, push the value at index i to the heap before incrementing both pointers:

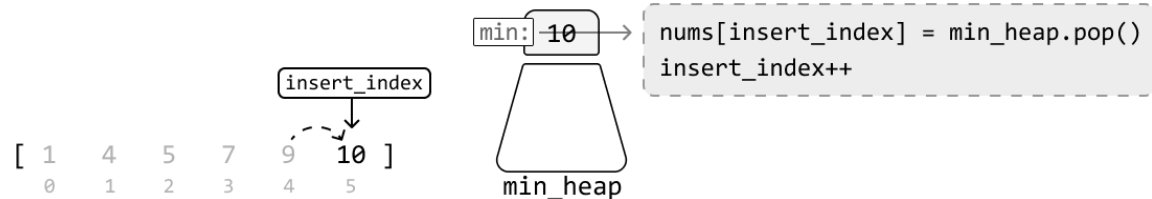
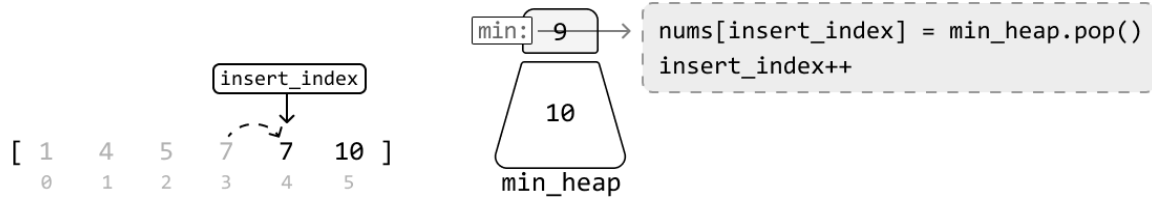


Let's continue this process for the remaining numbers:



Once there are no more elements to push into the heap, the rest of the array can be sorted by inserting the remaining values from the heap:





Once the heap is empty, the array is sorted.

Implementation

```
def sort_a_k_sorted_array(nums: List[int], k: int) -> List[int]:
    # Populate a min-heap with the first k + 1 values in 'nums'.
    min_heap = nums[:k+1]
    heapq.heapify(min_heap)
    # Replace elements in the array with the minimum from the heap at each
    # iteration.
    insert_index = 0
    for i in range(k + 1, len(nums)):
        nums[insert_index] = heapq.heappop(min_heap)
        insert_index += 1
        heapq.heappush(min_heap, nums[i])
    # Pop the remaining elements from the heap to finish sorting the array.
    while min_heap:
        nums[insert_index] = heapq.heappop(min_heap)
```

```
    insert_index += 1
return nums
```

Complexity Analysis

Time complexity: The time complexity of `sort_a_k_sorted_array` is $O(n \log(k))$, where n denotes the length of the array. Here's why:

- We perform heapify on a min_heap of size $k + 1$ which takes $O(k)$ time.
- Then, we perform push and pop operations on approximately $n - k$ values using the heap. Since the heap can grow up to a size of $k + 1$, each push and pop operation takes $O(\log(k))$ time. Therefore, this loop takes $O(n \log(k))$ time in the worst case.
- The final while-loop runs in $O(k \log(k))$ time since we pop $k + 1$ values from the heap.

Therefore, the overall time complexity is $O(k) + O(n \log(k)) + O(k \log(k)) = O(n \log(k))$, since k is upper-bounded by n in each operation above. This is because the heap can only ever contain at most n values.

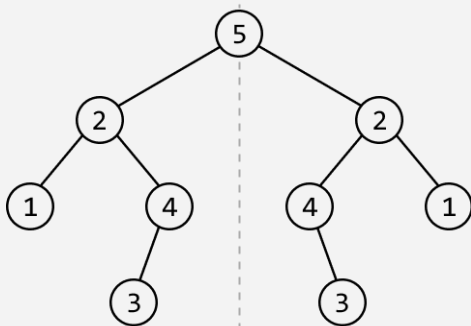
Space complexity: The space complexity is $O(k)$ because the heap can grow up to $k + 1$ in size.

Trees

Binary Tree Symmetry

Determine if a binary tree is vertically symmetric. That is, the left subtree of the root node is a mirror of the right subtree.

Example:

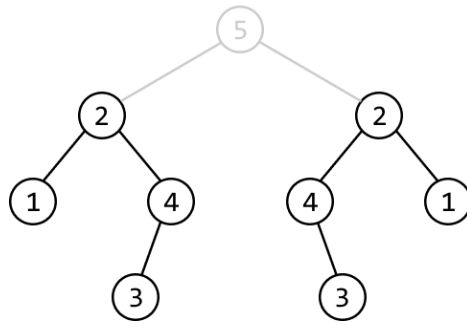


Output: True

Intuition

To check a binary tree's symmetry, we need to assess its left and right subtrees. The first thing to note is that the root node itself doesn't affect the symmetry of the tree. Therefore, we don't need to consider the root node. We now have the task of comparing two subtrees to check if one vertically mirrors the other.

Consider the root node's left and right subtrees in the following example:



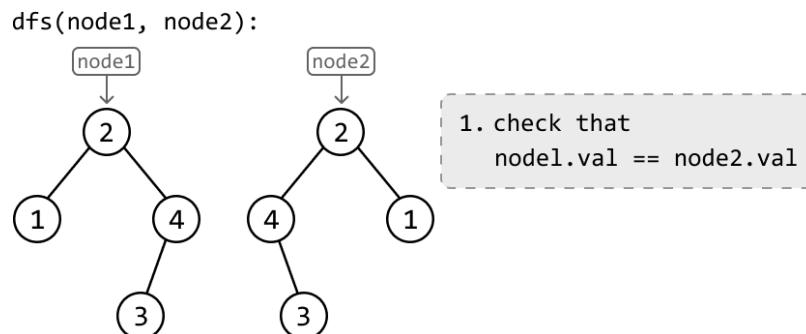
The key observation here is that **the right subtree is an inverted version of the left subtree**.

We've learned from the problem *Invert Binary Tree* that an inversion is performed by swapping the left and right child of every node. This suggests the value of each node's left child in the left subtree should match the value of the right child of the corresponding node in the right subtree, and vice versa.

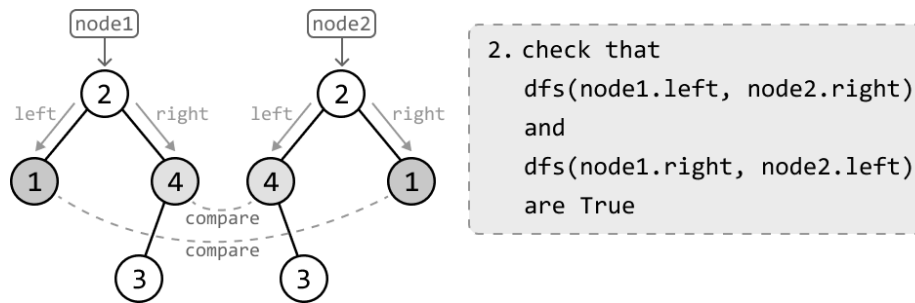
We can start by using DFS to traverse both subtrees. During this traversal, we compare the left and right children of each node in the left subtree with the right and left children of its corresponding node in the right subtree, respectively.

- If the values of any two nodes being compared are not the same, the tree is not symmetric.
- If, at any point, one of the child nodes being compared is null while the other isn't, the tree is also not symmetric.

Initially, we see the values of the root nodes of the left and right subtrees are equal, as we know:



We proceed by comparing their children through recursive DFS calls. Specifically, make two recursive DFS calls to compare the left child of one node with the right child of the other. This checks that node2's children contain the same values as node1's children, but inverted.



If either DFS call returns false, the subtrees are not symmetric. If both DFS calls return true, the subtrees are symmetric. This process is repeated for the entire tree.

Implementation

```
def binary_tree_symmetry(root: TreeNode) -> bool:
    if not root:
        return True
    return dfs(root.left, root.right)

def dfs(node1: TreeNode, node2: TreeNode) -> bool:
    # Base case: if both nodes are null, they're symmetric.
    if not node1 and not node2:
        return True
    # If one node is null and the other isn't, they aren't symmetric.
    if not node1 or not node2:
        return False
    # If the values of the current nodes don't match, trees aren't symmetric.
    if node1.val != node2.val:
        return False
    # Compare the 'node1's left subtree with 'node2's right subtree. If these
    # aren't symmetric, the whole tree is not symmetric.
    if not dfs(node1.left, node2.right):
        return False
    # Compare the 'node1's right subtree with 'node2's left subtree.
    return dfs(node1.right, node2.left)
```

Complexity Analysis

Time complexity: The time complexity of `binary_tree_symmetry` 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 .

Interview Tip

Tip: Cover null cases.

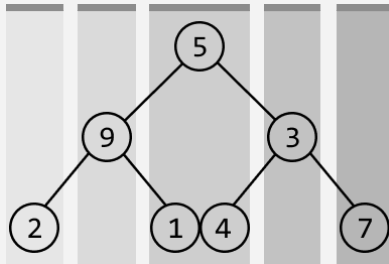


Always check for null or empty inputs before using their attributes in a function. In this problem, the `dfs` function accesses the left and right attributes of the input node, necessitating initial null checks.

Binary Tree Columns

Given the root of a binary tree, return a list of arrays where each array represents a vertical column of the tree. Nodes in the same column should be ordered from top to bottom. Nodes in the same row and column should be ordered from left to right.

Example:



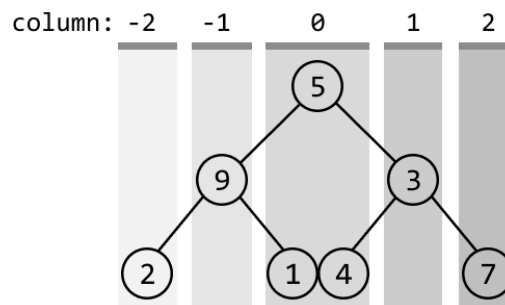
Output: `[[2], [9], [5, 1, 4], [3], [7]]`

Intuition

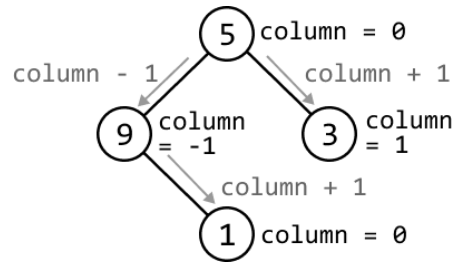
First and foremost, to get the columns of a binary tree, we'll need a way to identify what column each node is in.

Column ids

One way to distinguish between different columns is to represent each column by a distinct numerical value: an id. Initially, we don't know how many columns there are to the left or to the right of the root node, but we at least know the column which contains the root node itself. Let's give this column an id of 0. From here, we can set positive column ids for nodes to the right of the root and negative ids to the left:

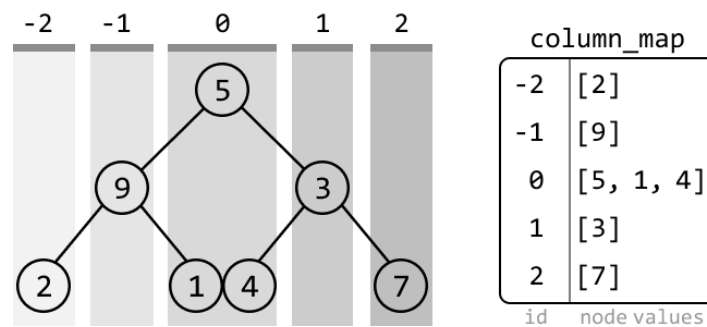


How can we identify what column id a node is associated with? A handy observation is that every time we move to the right, the column id increases by 1, and every time we move to the left, it decreases by 1. This allows us to assign ids as we traverse the tree: for any node, the column ids of `node.left` and `node.right` are `column - 1` and `column + 1`, respectively:



Tracking node values by their column id

Now that we have a way to determine a node's column, we'll need a way to keep track of which node values belong to which columns. This can be done using a **hash map** where keys are column ids and the value of each column id is a list of the node values at that column.



Now, let's consider what traversal algorithm we should use to populate this hash map.

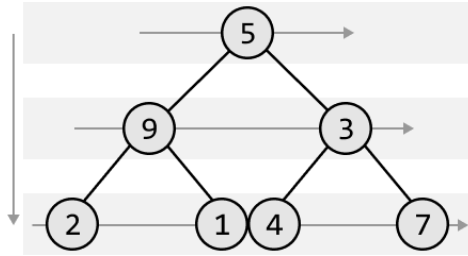
Breadth-first search

In general, we can employ any traversal method to assign column ids to each node, as long as we increment the column id whenever we move right and decrement it whenever we move left. However, we need to be cautious about the following two requirements:

1. Nodes in the same column should be ordered from top to bottom.
2. Nodes in the same row and column should be ordered from left to right.

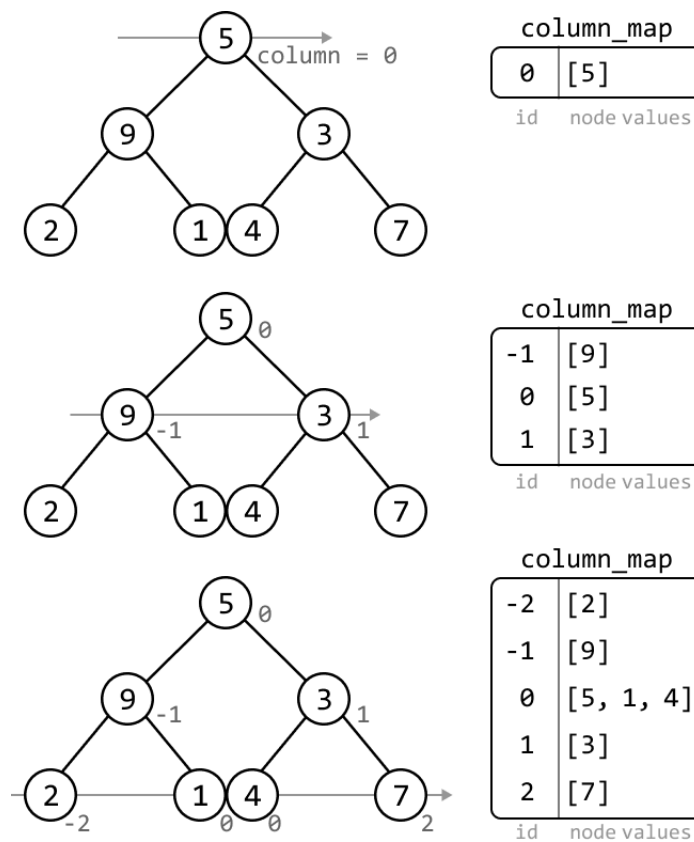
So, we'll need an algorithm that traverses the tree from top to bottom and then from left to right. This calls for **BFS**.

BFS processes nodes level by level, starting from the root and moving horizontally across the tree at each level. This method ensures nodes are visited from top to bottom, and for nodes in the same row (level), they are visited from left to right.



Traversing the tree this way ensures node values are added to the hash map in the desired order, complying with the two requirements above.

We can see how the hash map is populated level by level below:

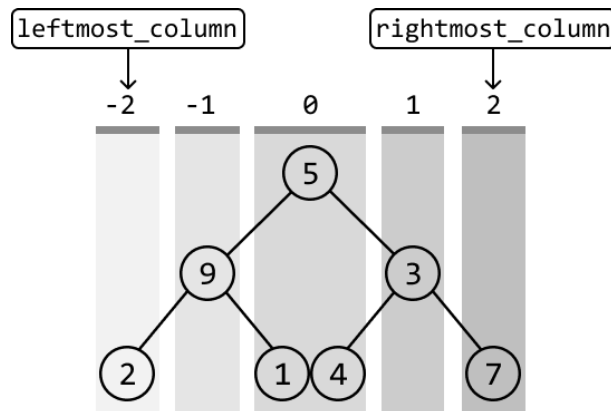


Once BFS concludes, the hash map is populated with lists of values for each column id. However, the hash map itself is not the expected output. So, let's discuss what to do next.

Creating the output

This problem expects us to return the column ids from left to right. As we know, a hash map does not inherently maintain the order of its keys (column ids), which means we'll need to find a way to attain the output in the desired order.

To retrieve the column ids in the desired order, it would be useful to know the **leftmost column id** and the **rightmost column id**. This allows us to increment through all ids in the range `[leftmost_column, rightmost_column]`, ensuring we can build the output in the desired order.



Implementation

```

def binary_tree_columns(root: TreeNode) -> List[List[int]]:
    if not root:
        return []
    column_map = defaultdict(list)
    leftmost_column = rightmost_column = 0
    queue = deque([(root, 0)])
    while queue:
        node, column = queue.popleft()
        if node:
            # Add the current node's value to its corresponding list in the hash
            # map.
            column_map[column].append(node.val)
            leftmost_column = min(leftmost_column, column)
            rightmost_column = max(rightmost_column, column)
            # Add the current node's children to the queue with their respective
            # column ids.
            queue.append((node.left, column - 1))
            queue.append((node.right, column + 1))
    # Construct the output list by collecting values from each column in the hash
    # map in the correct order.

```

```
return [column_map[i] for i in range(leftmost_column, rightmost_column + 1)]
```

Complexity Analysis

Time complexity: The time complexity of `binary_tree_columns` 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.

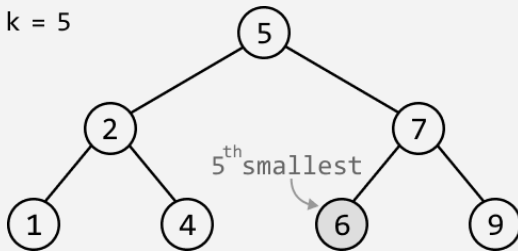
Space complexity: The space complexity is $O(n)$ due to the space taken up by the queue. The queue's size will grow as large as the level with the most nodes. In the worst case, this occurs at the final level when all the last-level nodes are non-null, totaling approximately $n/2$ nodes. Note that the output array created at the return statement does not contribute to the space complexity.

Kth Smallest Number in a Binary Search Tree

Given the root of a binary search tree (BST) and an integer k , find the k^{th} smallest node value.

Example:

$k = 5$



Output: 6

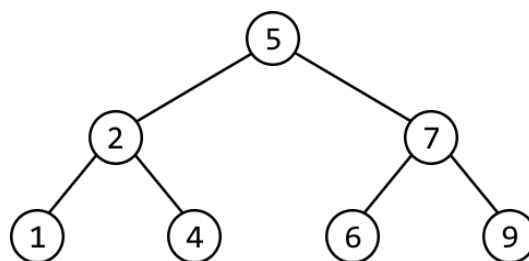
Constraints:

- $n \geq 1$, where n denotes the number of nodes in the tree.
- $1 \leq k \leq n$

Intuition – Recursive

A naive approach to this problem is to traverse the tree and store all the node values in an array, sort the array, and return the k^{th} element. This approach, however, does not take advantage of the fact that we're dealing with a BST.

Consider the BST below:

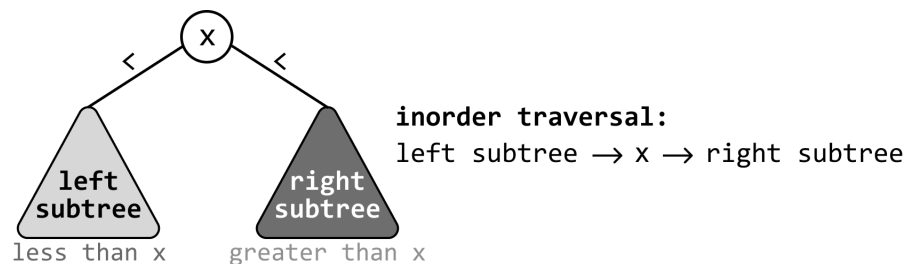


We know that in a BST, each node's value is larger than all the nodes to its left and smaller than all the nodes to its right. This structure means that BSTs inherently possess a sorted order. Given this, it should be possible to construct a sorted array of the tree's values by traversing the tree, without the need for additional sorting.

We now need a method to traverse the binary tree that allows us to encounter the nodes in their sorted order.

What are our options? We can immediately rule out any traversal algorithms that process the root node first, such as breadth-first search and preorder traversal, since the root node is not guaranteed to have the smallest value in a BST. This also indicates that we need an algorithm that starts with the leftmost node since this is always the smallest node in a BST. Additionally, the algorithm should end at the rightmost node since this would be the largest node.

This leads us to an ideal traversal algorithm: **inorder traversal**, where for each node, the left subtree is processed first, followed by the current node, and then the right subtree.



To build the sorted list of values using inorder traversal, we can design a recursive function. When called on the root node, it returns a sorted list of all values in the BST.

When the function is called for any node during the recursive process, it constructs a sorted list of the values in the subtree rooting from that node. This is achieved by first obtaining the sorted values from its left subtree, then adding the current node's value, and finally appending the sorted values from its right subtree.

Once we have the full list of sorted values, we can simply return the value at the $(k - 1)^{\text{th}}$ index to get the k^{th} smallest value.

Implementation – Recursive

```
def kth_smallest_number_in_BST_recursive(root: TreeNode, k: int) -> int:
    sorted_list = inorder(root)
    return sorted_list[k - 1]

# Inorder traversal function to attain a sorted list of nodes from the BST.
def inorder(node: TreeNode) -> List[int]:
    if not node:
        return []
    return inorder(node.left) + [node.val] + inorder(node.right)
```

Complexity Analysis

Time complexity: The time complexity of `kth_smallest_number_in_BST_recursive` is $O(n)$, where n denotes the number of nodes in the tree. This is because we need to traverse through all n nodes of the tree to attain the sorted list.

Space complexity: The space complexity is $O(n)$ due to the space taken up by `sorted_list`, as well as 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

Since we only need the k^{th} smallest value, storing all n values in a list might not be necessary. Ideally, we'd like to find a way to traverse through k nodes instead of n . How can we modify our approach to achieve this?

If we had a way to stop inorder traversal once we've reached the k^{th} node in the traversal, we would land on our answer. An iterative approach would allow for this since we'd be able to exit traversal once we've reached the k^{th} node – something that's quite difficult to achieve using recursion.

We know inorder traversal is a DFS algorithm and that DFS algorithms can be implemented iteratively using a stack. Let's explore this idea further.

Consider what happened during recursive inorder traversal in the previous approach:

- Make a recursive call to the left subtree.
- Process the current node.
- Make a recursive call to the right subtree.

Let's replicate the above steps iteratively using a stack.

1. Move as far **left** as possible, adding each node to the stack as we move left.
 - We do this because, at the start of each recursive call in the recursive approach, a new call is made to the current node's left subtree, continuing until the base case (a null node) is reached. This implies that to mimic this process iteratively, we'll need to move as far left as possible.
 - The reason we push nodes onto the stack as we go is so they can be processed later.

The code snippet for this can be seen below:

```
while node:
    stack.append(node)
    node = node.left
```

2. Once we can no longer move left, we pop the node off the top of the stack. Let's call it the **current node**. Initially, this node represents the smallest node. After this, the next node on the stack subsequently represents the next smallest node, and so on until we reach the k^{th} smallest node.
 - Decrement k , indicating that we now have one less node to visit until we reach the k^{th} smallest node.
 - Once $k == 0$, we found the k^{th} smallest node. Return the value of this node.
3. Move to the current node's **right** child.

Implementation – Iterative

```
def kth_smallest_number_in_BST_iterative(root: TreeNode, k: int) -> int:
    stack = []
    node = root
    while stack or node:
        # Move to the leftmost node and add nodes to the stack as we go so they
        # can be processed in future iterations.
        while node:
            stack.append(node)
            node = node.left
        # Pop the top node from the stack to process it, and decrement 'k'.
        node = stack.pop()
        k -= 1
        # If we have processed 'k' nodes, return the value of the 'k'th smallest
        # node.
        if k == 0:
            return node.val
        # Move to the right subtree.
        node = node.right
```

Complexity Analysis

Time complexity: The time complexity of `kth_smallest_number_in_BST_iterative` is $O(k + h)$, where h denotes the height of the tree. Here's why:

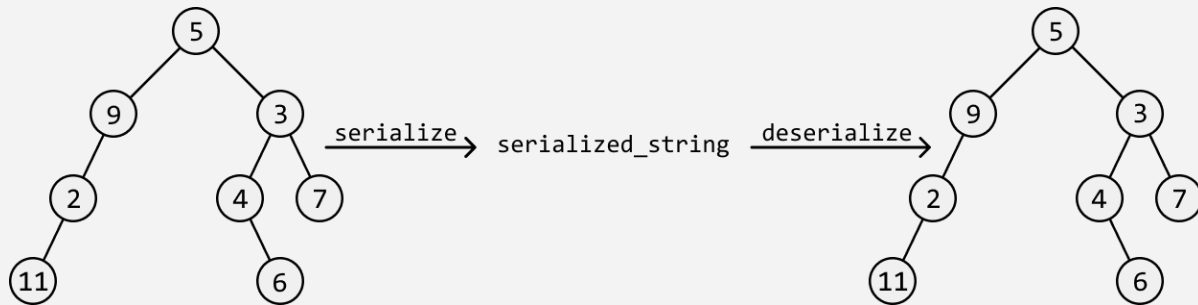
- Iterative inorder traversal ensures we traverse k nodes, which takes at least $O(k)$ time.
- Additionally, traversing to the leftmost node takes up to $O(h)$ time, which should be considered separately since it's possible that $h > k$. In the worst case, the height of the tree is n , resulting in a time complexity of $O(n)$, where n denotes the number of nodes in the tree.

Space complexity: The space complexity is $O(h)$ since the stack can store up to $O(h)$ space during the traversal to the leftmost node. In the worst case, the height of the tree is n , resulting in a space complexity of $O(n)$.

Serialize and Deserialize a Binary Tree

Write a function to serialize a binary tree into a string, and another function to deserialize that string back into the original binary tree structure.

Example:



Intuition

The primary challenge of this problem lies in how we serialize the tree into a string, as this will determine if it's possible to accurately reconstruct the tree using this string alone.

Let's first decide on a traversal strategy because the method we use to serialize the tree will impact how we deserialize the string. Two options:

- Use BFS to serialize the tree level by level.
- Use DFS. In this case, we'd need to choose between inorder, preorder, and postorder traversal.

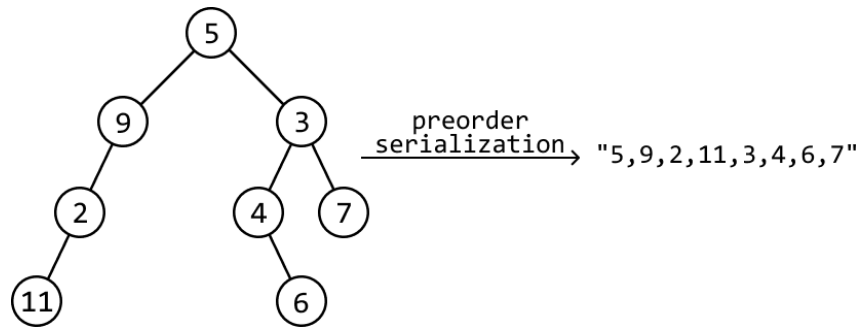
There's flexibility in choosing the traversal algorithm because serializing with a specific traversal method allows us to rebuild (deserialize) the tree using the same traversal algorithm.

Serialization

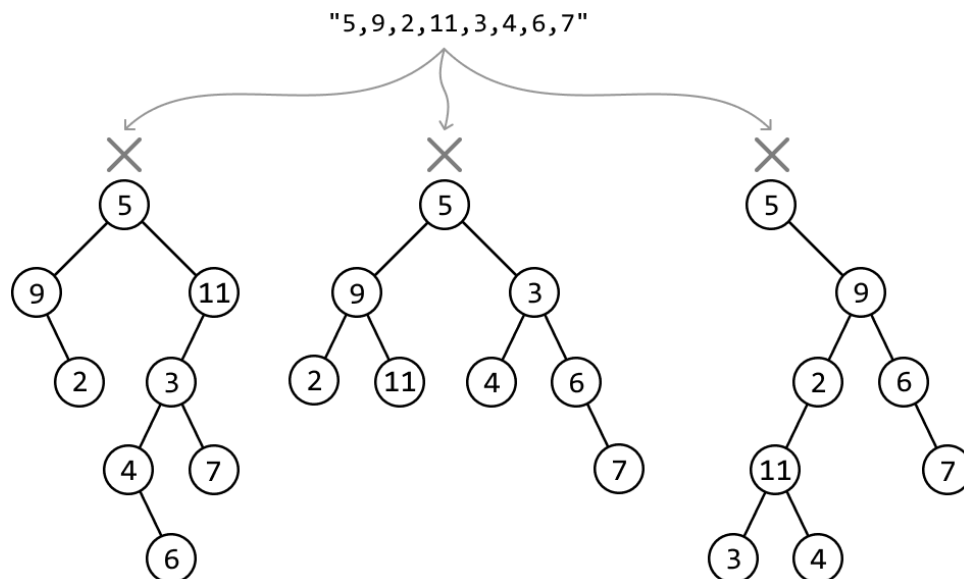
An important piece of information needed in our serialized string is the node values. In addition, we'll need to **ensure we can identify the root node's value**, since this is the first node to create when we deserialize the string.

As such, a traversal algorithm like preorder traversal is a good choice because it processes the root node first, then the left subtree, and finally the right subtree. This ensures the first value in our serialized string is the root node's value.

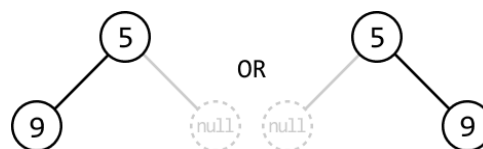
So, let's try serializing the following binary tree using preorder traversal, separating each node with a comma:



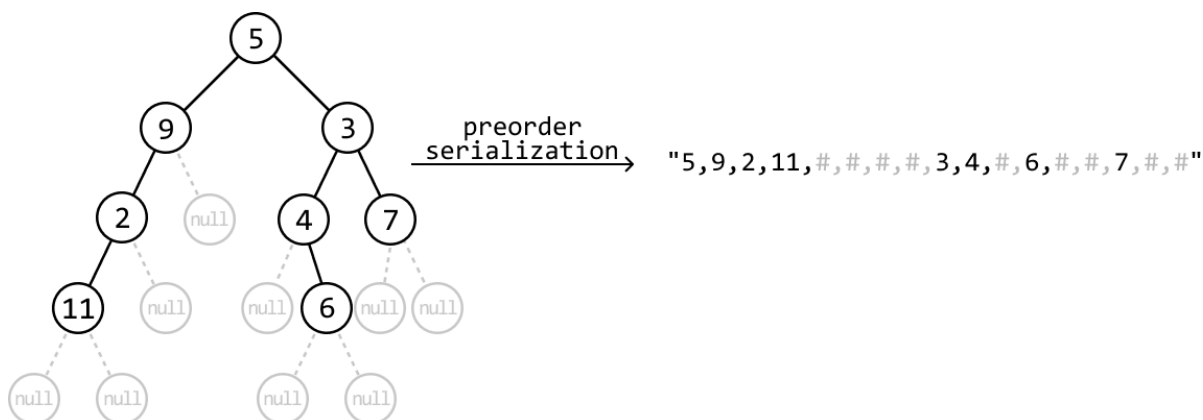
The issue with this serialization is that it doesn't guarantee we can reconstruct the original tree accurately from this serialized string representation. This is because the string could represent multiple different trees, which means preorder deserialization could result in the creation of an invalid tree:



This is a consequence of excluding some critical information from the string: **null child nodes**. For instance, after placing node 5 as the root in the above example, we don't yet know where to place the node of value 9, which is the next value in the string. That is, we can't determine whether node 9 should be the left or right child of node 5:



If the string indicates where the null child nodes are, we can correctly deserialize the tree because we'd have a complete representation of the tree's structure. Let's use the character '#' to represent a null node:



Deserialization

To deserialize a string created with preorder traversal, we also need to use preorder traversal to reconstruct the tree.

The first step is to split the string using the comma delimiter, so each node value and '#' is stored as separate elements in a list:

```

"5,9,2,11,#,#,#,#,3,4,#,6,#,#,7,#,#"
      |
      split
      ↓
[ 5  9  2  11  #  #  #  #  3  4  #  6  #  #  7  #  # ]
  
```

The first value in the list is the root node of the tree:

```

root
↓
[ 5  9  2  11  #  #  #  #  3  4  #  6  #  #  7  #  # ]
  
```

Starting from this root value, recursively construct the tree node by node using preorder traversal. Each new node will be created with the next value in the list of preorder values. Whenever we encounter a '#', we return null. The code snippet for this:

Helper function to construct the tree using preorder traversal.

```
def build_tree(values: List[str]) -> TreeNode:
```

```
    val = next(values)
```

```
    # Base case: '#' indicates a null node.
```

```

if val == '#':
    return None

# Use preorder traversal to create the current node first, then the left and
# right subtrees.
node = TreeNode(int(val))
node.left = build_tree(values)
node.right = build_tree(values)
return node

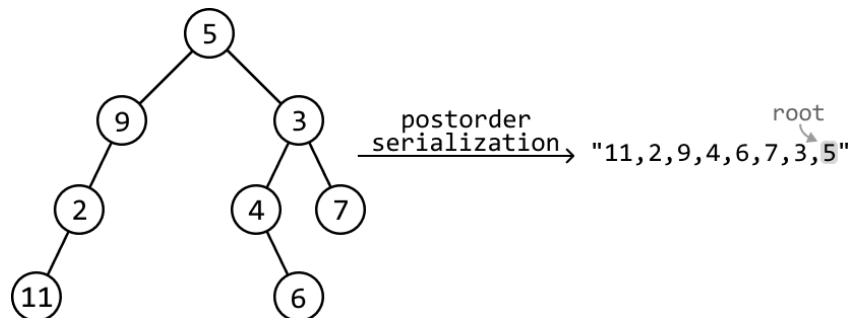
```

Follow-up: what if you must use a different traversal algorithm?

It's possible to serialize and deserialize a binary tree using other traversal algorithms than preorder traversal. Let's explore some alternatives.

Postorder traversal:

When we serialize the tree using postorder traversal, we get the following string (ignoring the null nodes in this discussion to focus on the node values and their order):

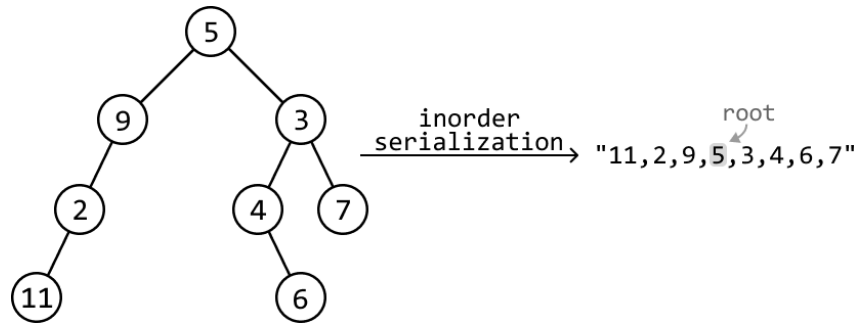


As we see, one big difference is that the root node will be the final value in the string, since postorder traversal processes the left subtree, then the right subtree, and finally the root node.

During deserialization, we'd build the tree by iterating through the node values from right to left instead of left to right, since the root value is at the right. In addition, we'd need to create each node's right subtree before we create its left subtree, as we go through the string in reverse order.

Inorder traversal:

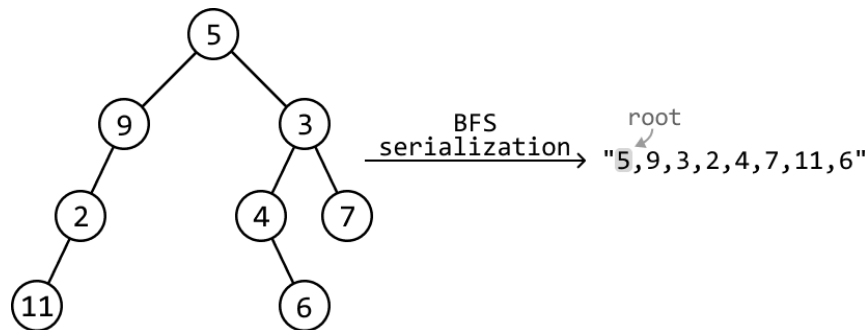
A lot more care needs to be taken when serializing a tree using inorder traversal. The main reason is that it's unclear where the root node of the tree is in the string, and where the root node of each subtree is, as we can see below:



This doesn't make it impossible to use inorder traversal. It just means significantly more information needs to be provided in the serialized string to make deserialization possible. In particular, we need to include details about which value serves as the root node for each subtree as we iterate through them.

Breadth-first search (BFS):

BFS starts processing from the root of the tree, and traverses through it level by level, and from left to right. Serializing the tree using BFS gives the following order of values:



Similarly to preorder traversal, we just need to follow the exact traversal order for reconstructing the tree when deserializing the string.

Implementation

In the following implementation, we opt for preorder traversal for serialization and deserialization.

```
def serialize(root: TreeNode) -> str:
    # Perform a preorder traversal to add node values to a list, then convert the
    # list to a string.
    serialized_list = []
    preorder_serialize(root, serialized_list)
    # Convert the list to a string and separate each value using a comma
```

```

    # delimiter.
    return ','.join(serialized_list)

# Helper function to perform serialization through preorder traversal.
def preorder_serialize(node, serialized_list) -> None:
    # Base case: mark null nodes as '#'.
    if node is None:
        serialized_list.append('#')
        return

    # Preorder traversal processes the current node first, then the left and right
    # children.
    serialized_list.append(str(node.val))
    preorder_serialize(node.left, serialized_list)
    preorder_serialize(node.right, serialized_list)

def deserialize(data: str) -> TreeNode:
    # Obtain the node values by splitting the string using the comma delimiter.
    node_values = iter(data.split(','))
    return build_tree(node_values)

# Helper function to construct the tree using preorder traversal.
def build_tree(values: List[str]) -> TreeNode:
    val = next(values)
    # Base case: '#' indicates a null node.
    if val == '#':
        return None

    # Use preorder traversal processes the current node first, then the left and
    # right children.
    node = TreeNode(int(val))
    node.left = build_tree(values)
    node.right = build_tree(values)
    return node

```

Complexity Analysis

Time complexity: The time complexity of both `serialize` and `deserialize` is $O(n)$, where n denotes the number of nodes in the tree. This is because we visit each of the n nodes in the binary tree exactly once during preorder traversal. The `serialize` function additionally converts the serialized list to a string, which also takes $O(n)$ time.

Space complexity: The space complexity of both `serialize` and `deserialize` 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 . In addition, the `serialize` function uses a `serialized_list` to store the values of the string.

Note: some interviewers might include the serialized string in the space complexity analysis, while others may not, as it is a required data structure specified by the problem and is not always considered as additional space. To ensure a thorough discussion, it's best to address both perspectives when analyzing space complexity during your interview.

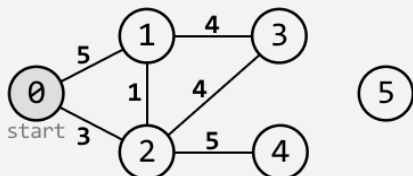
Graphs

Shortest Path

Given an integer n representing nodes labeled from 0 to $n - 1$ in an undirected graph, and an array of non-negative weighted edges, return an array where each index i contains the shortest path length from a specified start node to node i . If a node is unreachable, set its distance to -1 .

Each edge is represented by a triplet of positive integers: the start node, the end node, and the weight of the edge.

Example:



Input: $n = 6$, edges = $[[0, 1, 5], [0, 2, 3], [1, 2, 1], [1, 3, 4], [2, 3, 4], [2, 4, 5]]$, start = 0

Output: $[0, 4, 3, 7, 8, -1]$

Intuition

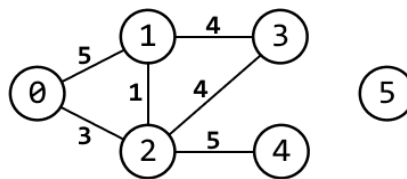
There are a few algorithms that can be employed to find the shortest path in a graph. Let's consider some of our options:

- BFS works well for finding the shortest path when the graph has edges with no weight, or uniform weight across the edges, as BFS doesn't take weight into account in its traversal strategy.

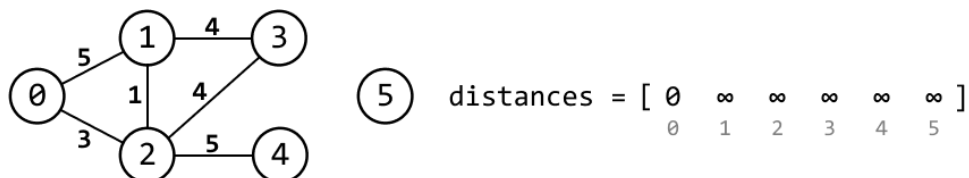
- Dijkstra's algorithm works well for graphs with non-negative weights, as it efficiently finds the shortest path from a single source to all other nodes.
- The Bellman-Ford algorithm works well for graphs with edges that may have negative weights [1].
- The Floyd-Warshall algorithm works well when we need to find the shortest paths between all pairs of nodes in a graph [2].

Among these options, **Dijkstra's algorithm** suits this problem the most since we're dealing with a graph with non-negative weighted edges, and we need to find the shortest path from a start node to all other nodes. This algorithm uses a greedy strategy, which we'll explore during this explanation.

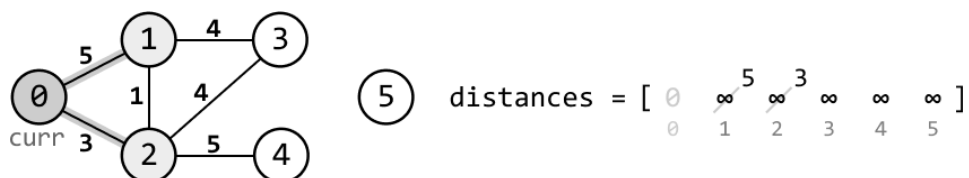
Consider the undirected weighted graph below, with node 0 as the starting node.



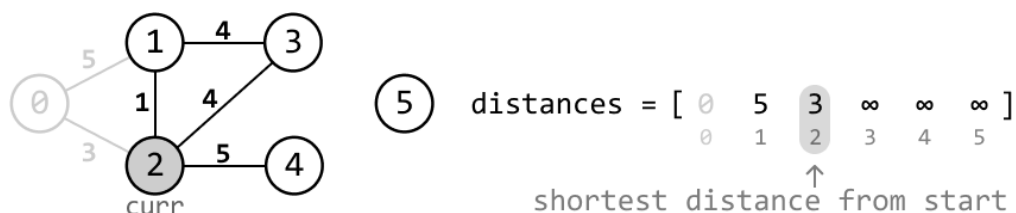
Initially, since we don't know any of the distances between node 0 and the other nodes, we'll set them to infinity. The only distance we do know is from the start node to itself, which is just 0:



Let's begin with the start node. Consider its immediate neighbors, nodes 1 and 2. The distances from node 0 to these nodes are 5 and 3, respectively. We don't know if these are the shortest distances from 0 to them, but they're definitely shorter than infinity. So, let's update the distances to those nodes from node 0:

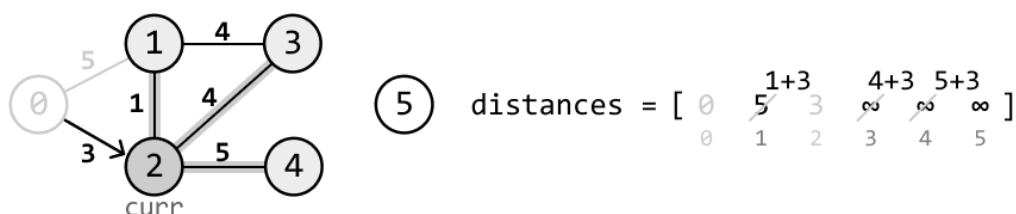


Right now, the closest node to the start node is node 2, with a distance of 3. This confirms that the shortest distance to node 2 from node 0 is 3, as the alternative route through node 1 has a longer distance. Thus, we can be certain that node 2 is reached through the shortest possible path, so let's move to it:

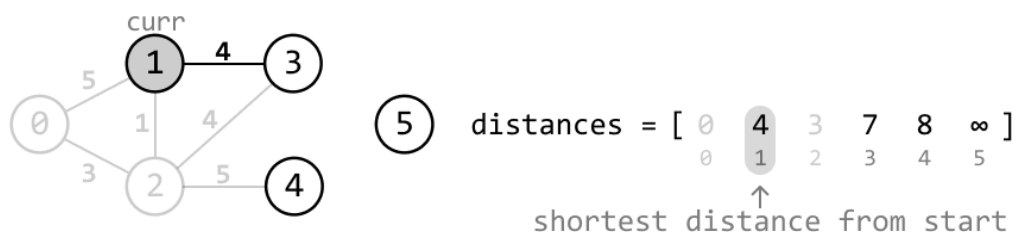


The current node is now node 2. Keep in mind that, so far, we've traversed a distance of $\text{distance}[2] = 3$ from node 0 to reach node 2.

The immediate unvisited neighbors of the current node are nodes 1, 3, and 4. The distances to them from the start node are $1 + 3$, $4 + 3$, and $5 + 3$ (where the $+3$ accounts for the distance traveled so far). Let's update the distance array with these distances since they are smaller than the distances currently set for them:



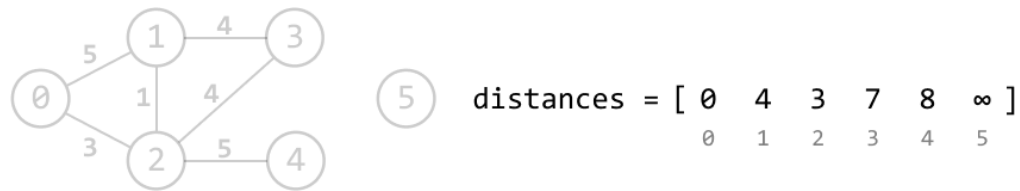
Right now, among the unvisited nodes, the node with the shortest distance from the start node is node 1, with a distance of 4. This means the shortest distance to node 1 from the start node is 4, since all other paths to node 1 involve traversing through distances larger than 4. So, let's move to node 4:



The greedy choice made is now becoming clearer:

At each step, we **move to the unvisited node with the shortest known distance from the start node**.

Applying this greedy choice for the rest of the graph completes Dijkstra's algorithm, giving us an array populated with the shortest path lengths from the start node to each node:



The final step is to convert all infinity values in this array to -1, indicating we weren't able to reach that node:

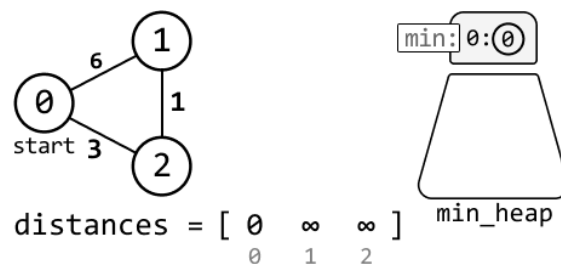
$$\text{distances} = [0 \quad 4 \quad 3 \quad 7 \quad 8 \quad \cancel{\infty}^{-1}]$$

0 1 2 3 4 5

To implement the above strategy, we'd like an efficient way to access the unvisited node with the shortest known distance at any point in the process. We can use a **min-heap** for this, allowing us logarithmic access to the node with the minimum distance.

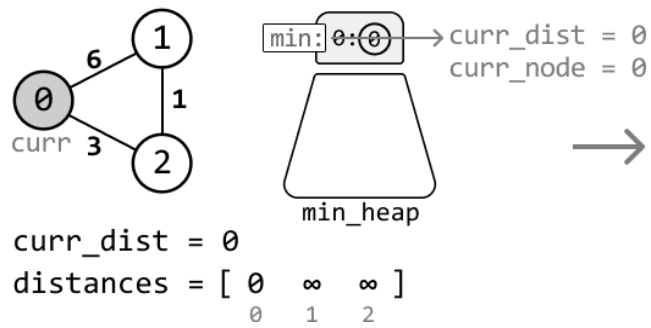
Using the min-heap

To understand how we use the min-heap in Dijkstra's algorithm, consider the following example with the start node, node 0, initially in the min-heap with its corresponding distance of 0:

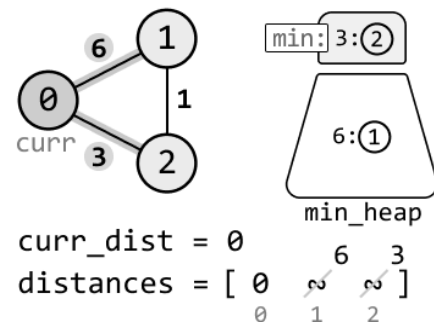


Begin by popping the start node from the top of the heap and setting it to the current node. Then, add the current node's neighbors to the min-heap with their corresponding distances from the start node, and update the distances of these neighbors:

get the current node and its distance:

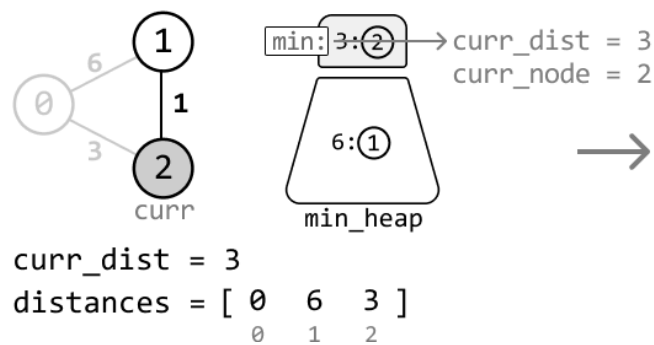


update distances of neighbors:

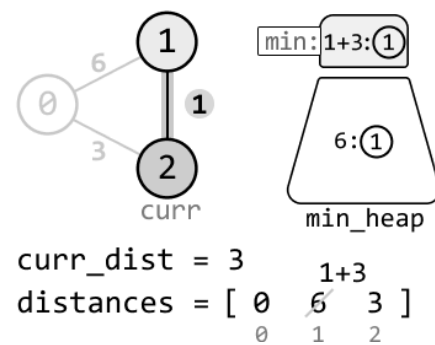


Repeat these two steps for each node at the top of the heap until the heap is empty:

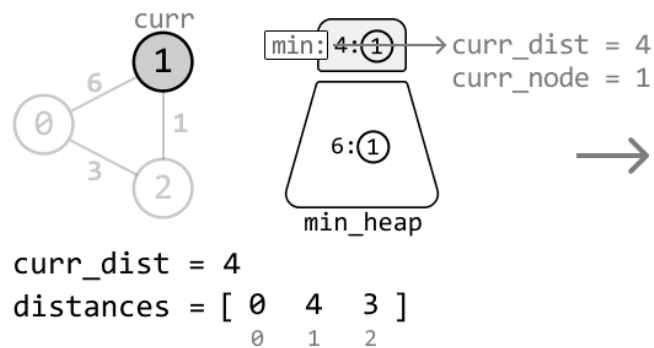
get the current node and its distance:



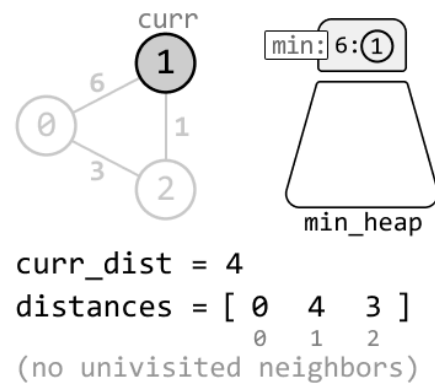
update distances of neighbors:



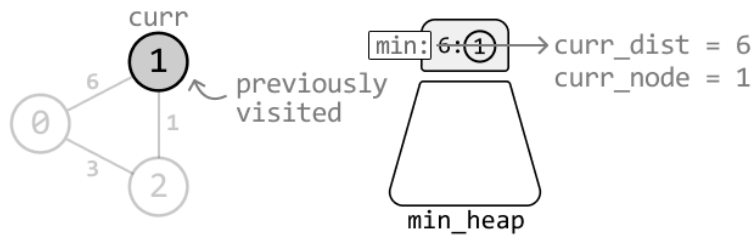
get the current node and its distance:



update distances of neighbors:



get the current node and its distance:



`curr_dist = 6`

`distances = [0 4 3]`
 0 1 2

As we can see, we encounter an issue: **we're revisiting node 1**. The only difference is that this time, we're visiting it at a larger distance (`curr_dist = 6`) from the start than the recorded distance (4). We can avoid this situation with the following if-statement:

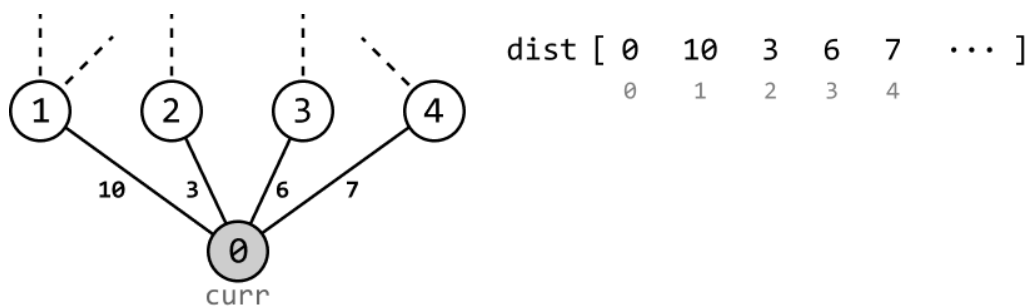
```
if curr_dist > distances[curr_node]:  
    continue
```

This check allows us to avoid using an additional data structure to keep track of visited nodes.

Why does the greedy approach work?

Before reading this section, it's worth reviewing the *Greedy* chapter if you aren't familiar with greedy algorithms.

Dijkstra's algorithm is considered greedy because, at each step, it selects the unvisited node with the shortest known distance from the start node, based on the assumption that this distance is the shortest possible path to that node. This choice is made as a local optimum, with the belief that it will lead to the global optima: the shortest path to all nodes. Can we always guarantee that this assumption is true? Consider the following graph, where the current node is node 0:



The node with the shortest known distance from the start node is node 2, with a distance of 3. We assume this is the shortest distance to node 2, and choose node 2 as the local optimum. Here's a question we could ask regarding the validity of this choice: is it possible to find a path with a

distance less than 3 to another neighboring node, making node 2 no longer the neighbor with the shortest distance from the start node?

The answer is no. This is because we have to pass through one of these neighboring nodes to find any other paths, which would require us to add a distance of at least 3 to our total distance traversed from the start node. To reduce this traversed distance, we would need to encounter negatively weighted edges in the graph, which we know isn't possible in this graph.

This analysis also demonstrates that **Dijkstra's algorithm is only applicable when the graph has no edges with negative weights.**

Implementation

```
def shortest_path(n: int, edges: List[int], start: int) -> List[int]:
    graph = defaultdict(list)
    distances = [float('inf')] * n
    distances[start] = 0
    # Represent the graph as an adjacency list.
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))
    min_heap = [(0, start)] # (distance, node)
    # Use Dijkstra's algorithm to find the shortest path between the start node
    # and all other nodes.
    while min_heap:
        curr_dist, curr_node = heapq.heappop(min_heap)
        # If the current distance to this node is greater than the recorded
        # distance, we've already found the shortest distance to this node.
        if curr_dist > distances[curr_node]:
            continue
        # Update the distances of the neighboring nodes.
        for neighbor, weight in graph[curr_node]:
            neighbor_dist = curr_dist + weight
            # Only update the distance if we find a shorter path to this
            # neighbor.
            if neighbor_dist < distances[neighbor]:
```

```
distances[neighbor] = neighbor_dist
heapq.heappush(min_heap, (neighbor_dist, neighbor))
# Convert all infinity values to -1, representing unreachable nodes.
return [-1 if dist == float('inf') else dist for dist in distances]
```

Complexity Analysis

Time complexity: The time complexity of the `shortest_path` is $O((n + e)\log(n))$, where e represents the number of edges. Here's why:

- Creating the adjacency list takes $O(e)$ time.
- Dijkstra's algorithm traverses up to all n nodes and explores each edge of the graph. To access each node, we pop it from the heap, and for each edge, up to one node is pushed to the heap (when we process each node's neighbors). Since each push and pop operation takes $O(\log(n))$ time, the time complexity of Dijkstra's algorithm is $O((n + e)\log(n))$.

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

Space complexity: The space complexity is $O(n + e)$, since the adjacency list takes up $O(n + e)$ space, whereas the distances array and min_heap take up $O(n)$ space.

References

[1] Bellman-Ford algorithm: https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

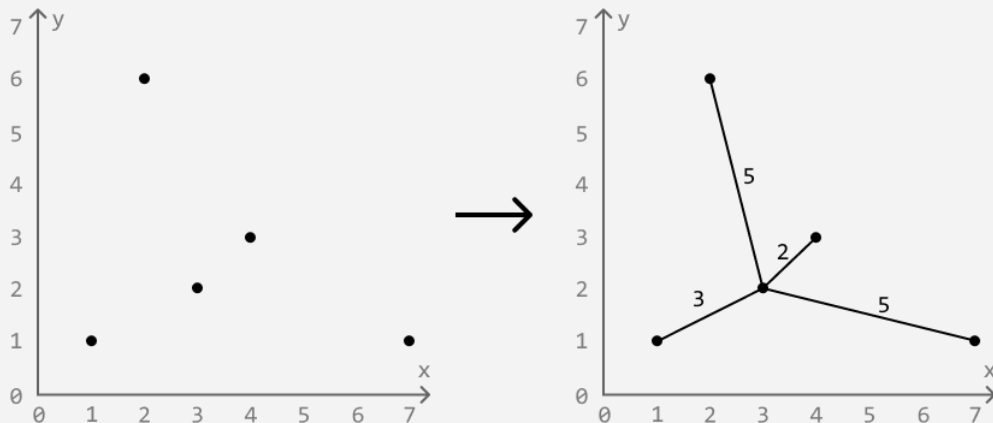
[2] Floyd–Warshall algorithm: https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

Connect the Dots

Given a set of points on a plane, determine the minimum cost to connect all these points.

The cost of connecting two points is equal to the Manhattan distance between them, which is calculated as $|x_1 - x_2| + |y_1 - y_2|$ for two points (x_1, y_1) and (x_2, y_2) .

Example:



Input: points = `[[1, 1], [2, 6], [3, 2], [4, 3], [7, 1]]`

Output: 15

Constraints:

- There will be at least 2 points on the plane.

Intuition

Let's treat this problem as a graph problem, imagining each point as a node, and the cost of connecting any two points as the weight of an edge between those nodes.

The goal is to connect all nodes (points) in such a way that the total cost is minimized. This is essentially the **minimum spanning tree** (MST) problem:

The MST of a weighted graph is a way to connect all points in the graph, ensuring each point is reachable from any other point while minimizing the total weight of the connections.

There are two main algorithms that are used to find the MST of a graph:

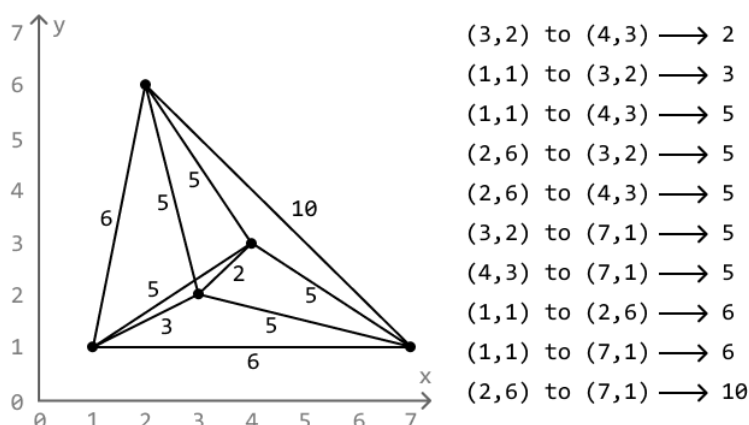
- Kruskal's algorithm
- Prim's algorithm

In this explanation, we'll break down Kruskal's algorithm since it uses a data structure we've already discussed in this chapter: **Union-Find**.

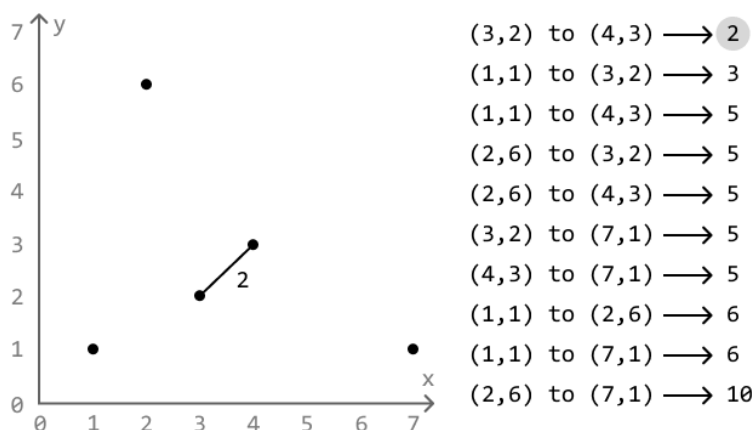
Kruskal's algorithm

Kruskal's algorithm is a greedy method for finding the MST. It essentially builds the MST by connecting nodes with the lowest-weighted edges first while skipping any edges that could cause a cycle.

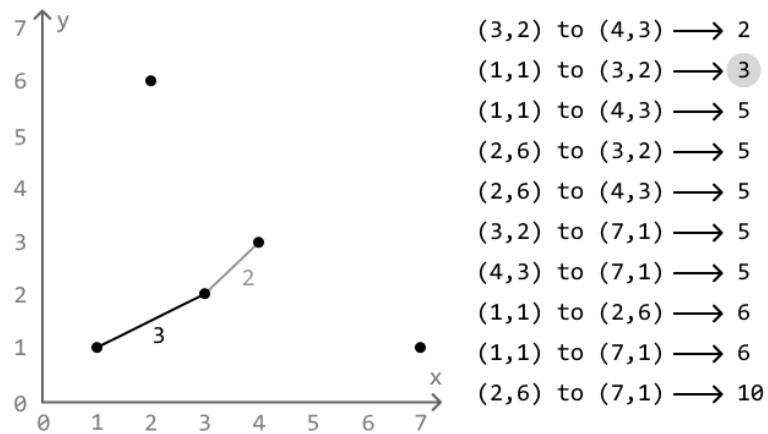
To do this, we first need to identify all the possible edges and sort them by their Manhattan distance:



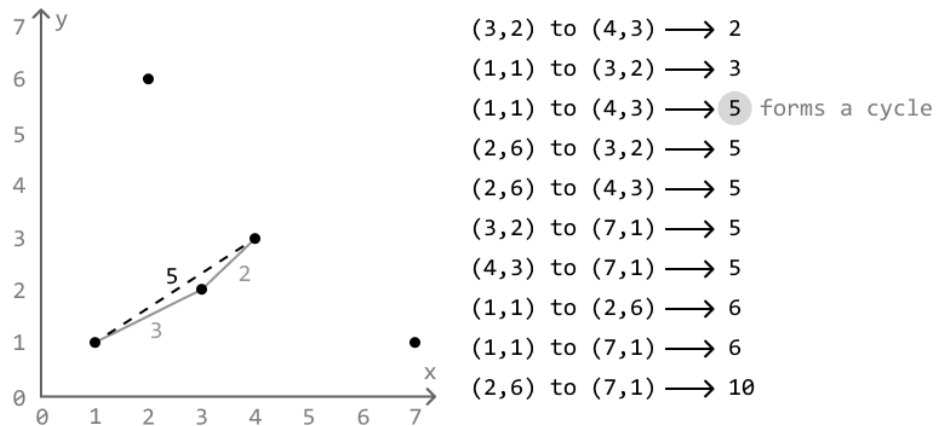
Now, let's implement Kruskal's algorithm. Start by including the lowest weighted edge in the MST, which is the edge from (3, 2) to (4, 3):



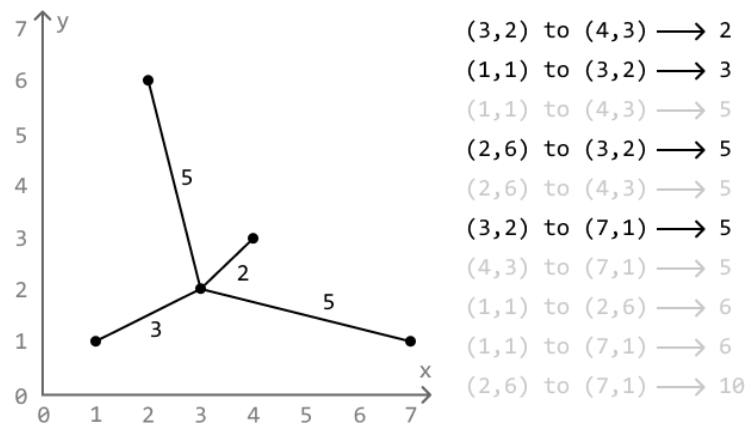
Next, add the next lowest weighted edge, which is the edge from (1, 1) to (3, 2):



Notice that adding the next edge from $(1, 1)$ to $(4, 3)$ will cause a cycle. Edges that cause cycles are avoided in an MST because doing so implies we're connecting two points that are already connected. So, let's skip this edge:



Continuing this process until all points are connected gives us the MST:



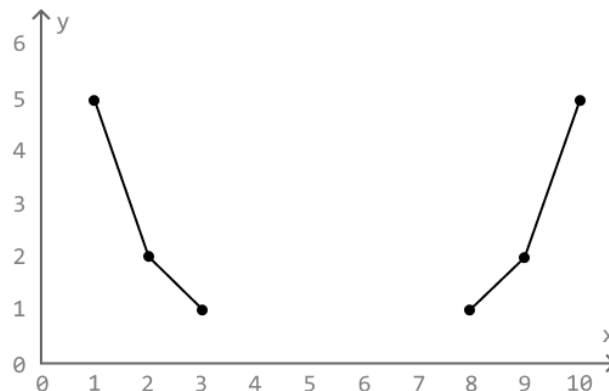
We'll know that all points are connected once we've added a total of $n - 1$ edges to the MST, because this is the least number of edges needed to connect n points without any cycles. To attain

the cost of this MST, we just need to keep track of the Manhattan cost of each edge we add to the MST.

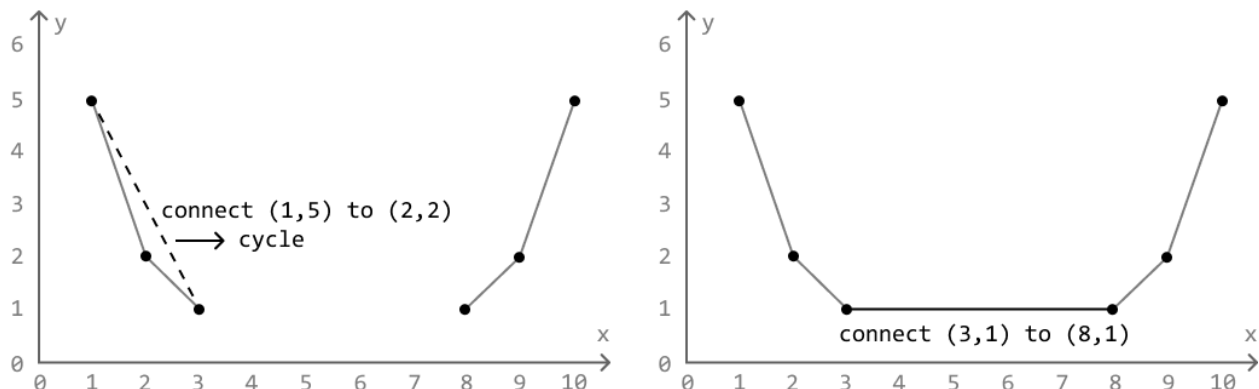
Now that we have a strategy to find the MST of a set of points, we just need a way to determine when connecting two points leads to a cycle.

Avoiding cycles

A cycle is formed when we add an edge to nodes that are already connected in some way. Consider the following set of points, where the group of points to the left are connected, and the group of points to the right are connected:



Connecting any two points in the same group causes a cycle, and connecting two points from two separate groups will result in both groups merging into one:



What would be useful here is a way to determine if two points belong to the same group, and a way to merge two groups together. The **Union-Find** data structure is perfect for this.

We can use the union function to connect two points together.

- If the two points belong to two separate groups, union should merge those groups and return true.

- Otherwise, if they belong to the same group, union should return false.

This way, we can use this boolean return value as a way to determine if attempting to connect two points causes a cycle.

If you're not familiar with Union-Find, study the solution to the *Merging Communities* problem.

Implementation

In this implementation, we'll identify each point using their index in the points array. This means that for n points, each point is represented by an index from 0 to $n - 1$. By doing this, we can set up a Union-Find data structure with a capacity of n so that there are n groups initially, with each group containing one of the n points.

```
def connect_the_dots(points: List[List[int]]) -> int:
    n = len(points)
    # Create and populate an array containing all possible edges.
    edges = []
    for i in range(n):
        for j in range(i + 1, n):
            # Manhattan distance.
            cost = (abs(points[i][0] - points[j][0]) +
                    abs(points[i][1] - points[j][1]))
            edges.append((cost, i, j))
    # Sort the edges by their cost in ascending order.
    edges.sort()
    uf = UnionFind(n)
    total_cost = edges_added = 0
    # Use Kruskal's algorithm to create the MST and identify its minimum cost.
    for cost, p1, p2 in edges:
        # If the points are not already connected (i.e., their representatives are
        # not the same), connect them, and add the cost to the total cost.
        if uf.union(p1, p2):
            total_cost += cost
            edges_added += 1
        # If n - 1 edges have been added to the MST, the MST is complete.
        if edges_added == n - 1:
```



```
    return total_cost
```

The Union-Find data structure remains the same as the implementation provided in *Merging Communities*, with a slight modification made to the union function, adding a boolean return value to it.

```
class UnionFind:
    def __init__(self, size):
        self.parent = [i for i in range(size)]
        self.size = [1] * size

    def union(self, x, y) -> bool:
        rep_x, rep_y = self.find(x), self.find(y)
        if rep_x != rep_y:
            if self.size[rep_x] > self.size[rep_y]:
                self.parent[rep_y] = rep_x
                self.size[rep_x] += self.size[rep_y]
            else:
                self.parent[rep_x] = rep_y
                self.size[rep_y] += self.size[rep_x]
            # Return True if both groups were merged.
            return True
        # Return False if the points belong to the same group.
        return False

    def find(self, x) -> int:
        if x == self.parent[x]:
            return x
        self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
```

Complexity Analysis

Time complexity: The time complexity of the `connect_the_dots` is $O(n^2 \log(n))$, where n denotes the length of the `points` array, and n^2 denotes the number of edges in the `edges` array, as we consider all possible pairs of points when forming edges. Here's why:

- Sorting n^2 edges takes $O(n^2 \log(n^2))$ time, which can be simplified to $O(n^2 \log(n))$.
- We perform up to one union operation for each edge, with each union taking amortized $O(1)$ time, resulting in the union operations contributing amortized $O(n^2)$ time.

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

Space complexity: The space complexity is $O(n^2)$ due to the space taken up by the `edges` array. In addition, the `UnionFind` data structure takes up $O(n)$ space.

Backtracking

Combinations of a Sum

Given an integer array and a target value, find all unique combinations in the array where the numbers in each combination sum to the target. Each number in the array may be used an unlimited number of times in the combination.

Example:

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

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

Constraints:

- All integers in `nums` are positive and unique.
- The target value is positive.
- The output must not contain duplicate combinations. For example, `[1, 1, 2]` and `[1, 2, 1]` are considered the same combination.

Intuition

Since we can use each integer in the input array as many times as we like, we can create an infinite number of combinations. We certainly cannot explore combinations infinitely. So, to manage this, we need to narrow our search.

An important point that will help us with this is that all values in the integer array are positive integers. This means that as we add more values to a combination, its sum will increase. Therefore, we should **stop building a combination once its sum is equal to or exceeds the target value**.

Another thing we should be mindful of is duplicate combinations. Consider the input array `[1, 2, 3]`. The combinations `[1, 3, 2, 1]` and `[2, 1, 3, 1]` represent the same combination. To ensure a **universal representation**, we can represent this combination as `[1, 1, 2, 3]`, where

the integers appear in the same order as in the original array. To ensure every combination has only one version, we need to build the combinations so that they're all ordered this way.

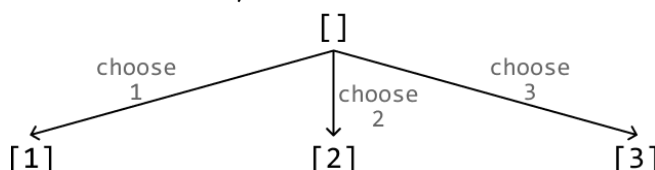
With those two things in mind, let's think about how we find all combinations that sum to the target value. **Backtracking** is ideal for exploring all possible combinations, so let's start by considering the state space tree for this problem.

State space tree

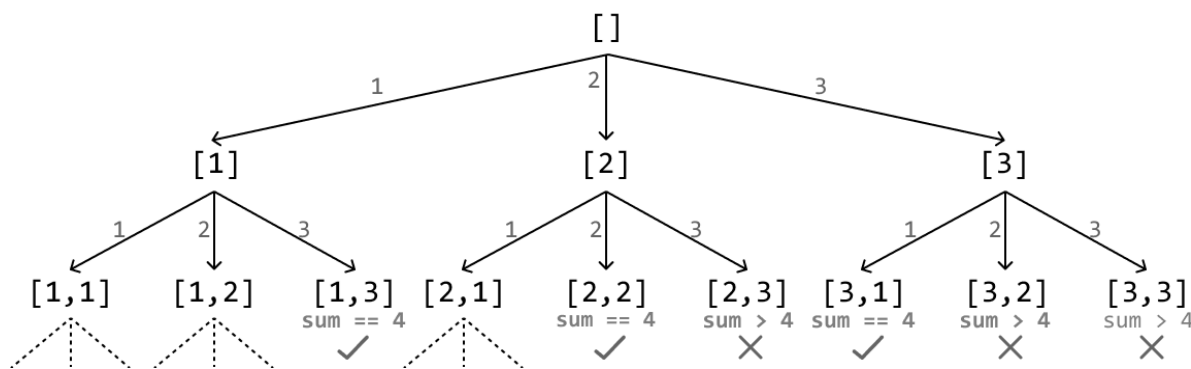
The purpose of a state space tree is to show combinations getting built one number at a time. Consider the input array [1, 2, 3] and a target of 4. Let's start with the root node of the tree, which is an empty combination:

[]

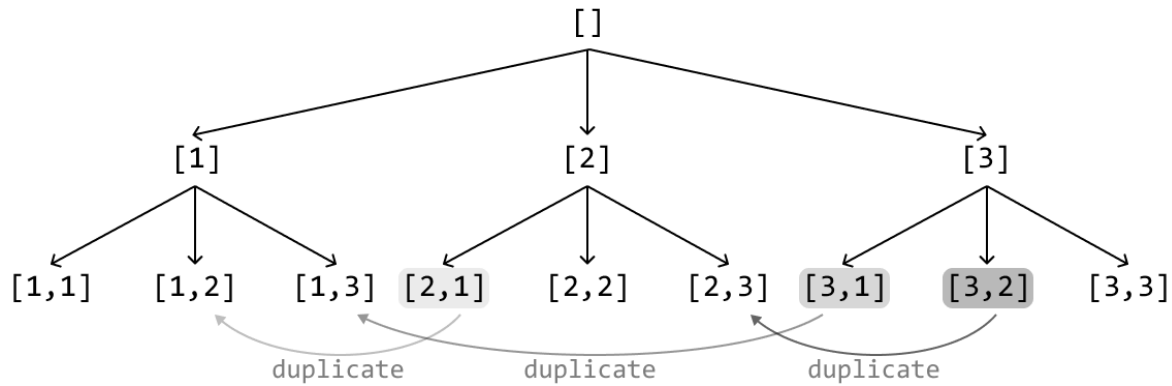
To figure out how to branch out from here, let's identify what decisions we can make. Each element can be included in a combination an unlimited number of times. So, this means we can make three decisions for an array of length 3: include each element from the array (remember that a branch in the state space tree represents a decision):



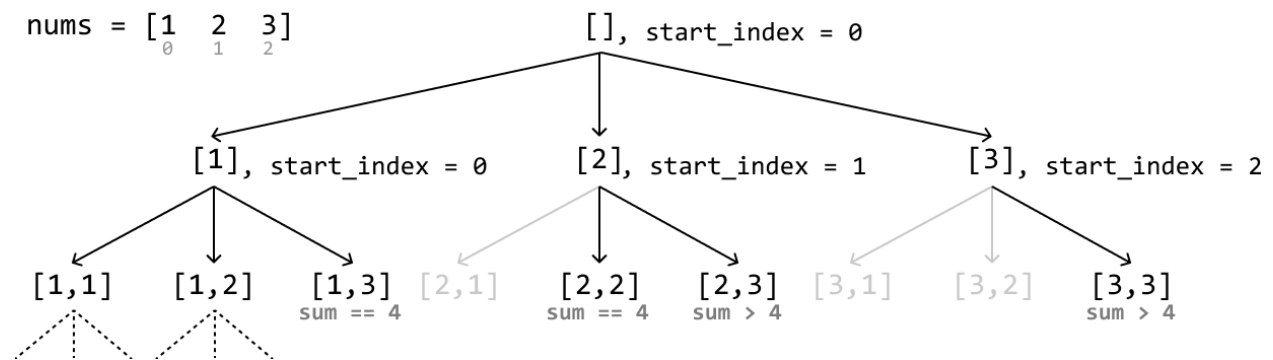
Let's make the same decisions for each of these combinations as well to continue extending the state space tree. Remember that if any combination has a sum equal to 4 or exceeding 4, we stop extending those combinations. These two conditions are effectively our **termination conditions**:



One issue with this approach is that it resulted in duplicate combinations in our tree:



To avoid these duplicates, we should keep in mind our universal representation: each combination should be constructed such that its elements are listed in the same order as in the input array. We can enforce this by specifying an index '**start_index**' for each combination we create. This **start_index** points to a value in the input array and ensures we can only add elements from this value onward. This way, we maintain the correct order and avoid duplicates in our combinations:



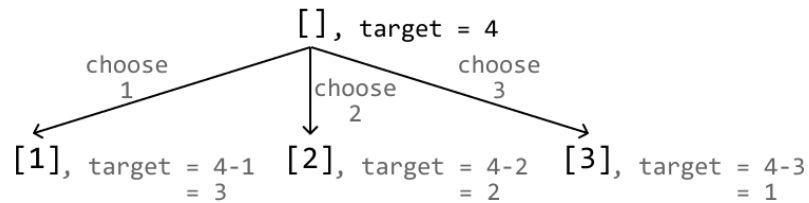
Initially, for the root combination, **start_index** is set to 0. As we recursively build each combination, **start_index** is updated to the index of the current element being added. By doing this, we ensure that in the next recursive call, we only consider elements from the updated index onward in the input array.

This maintains the required order and prevents duplicates because we never revisit previous elements. Since each combination is built by only adding elements that come after the current element in the input array, we avoid generating the same combination in a different order.

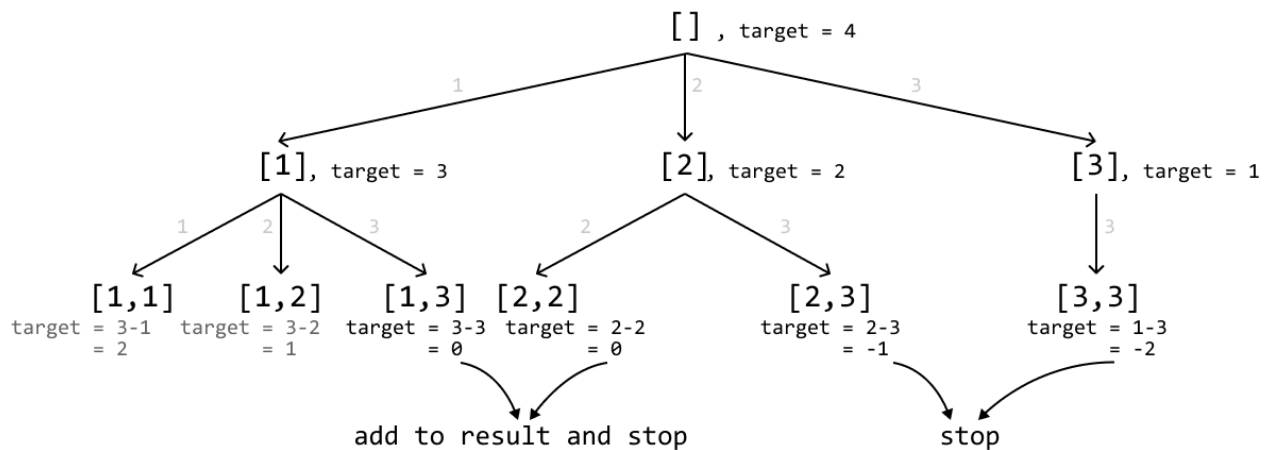
Implementation

In our algorithm, the termination condition requires us to know the sum of the current combination. While we could use a separate variable to track the sum of each combination, this isn't necessary. Instead, we can repurpose our target value. **When we choose a number to add to a combination, we reduce the target value by that number.** This way, the target value dynamically

tracks the remaining sum needed to reach the original target. We see how this works below, where the target gets reduced by the value we add to the combination:



This means that when we reach a target of 0, we've found a valid combination. If the target becomes negative, we can terminate the current branch of the search:



```

def combinations_of_sum_k(nums: List[int], target: int) -> List[List[int]]:
    res = []
    dfs([], 0, nums, target, res)
    return res

def dfs(combination: List[int], start_index: int, nums: List[int], target: int,
        res: List[List[int]]) -> None:
    # Termination condition: If the target is equal to 0, we found a combination
    # that sums to 'k'.
    if target == 0:
        res.append(combination[:])
        return
    # Termination condition: If the target is less than 0, no more valid
    # combinations can be created by adding to the current combination.
    if target < 0:

```

```

    return
    # Starting from 'start_index', explore all combinations after adding
    # 'nums[i]'.
    for i in range(start_index, len(nums)):
        # Add the current number to create a new combination.
        combination.append(nums[i])
        # Recursively explore all paths that branch from this new combination.
        dfs(combination, i, nums, target - nums[i], res)
        # Backtrack by removing the number we just added.
        combination.pop()

```

Complexity Analysis

Time complexity: The time complexity of `combinations_of_sum_k` is $O(n^{target/m})$, where n denotes the length of the array, and m denotes the smallest candidate. This is because, in the worst case, we always add the smallest candidate m to our combination. The recursion tree will branch down until the sum of the smallest candidates reaches or exceeds the target. This results in a tree depth of $target/m$. Since the function makes a recursive call for up to n candidates at each level of the recursion, the branching factor is n , giving us the time complexity of $O(n^{target/m})$.

Space complexity: The space complexity is $O(target/m)$, which includes:

- The recursive call stack depth, which is at most $target/m$ in depth.
- The combination list can also require at most $O(target/m)$ space since the longest combination would consist of the smallest element m repeated $target/m$ times.

Phone Keypad Combinations

You are given a string containing digits from 2 to 9 inclusive. Each digit maps to a set of letters as on a traditional phone keypad:

1	2 abc	3 def
4 ghi	5 jkl	6 mno
7 pqr	8 stu	9 wxyz

Return all possible letter combinations the input digits could represent.

Example:

Input: `digits = "69"`

Output: `["mw", "mx", "my", "mz", "nw", "nx", "ny", "nz", "ow", "ox", "oy", "oz"]`

Intuition

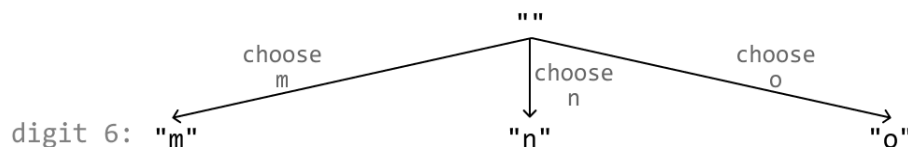
At each digit in the string, we have a decision to make: which letter will this digit represent? Based on this decision, let's illustrate the state space tree which represents the choices at each digit of the input string.

State space tree

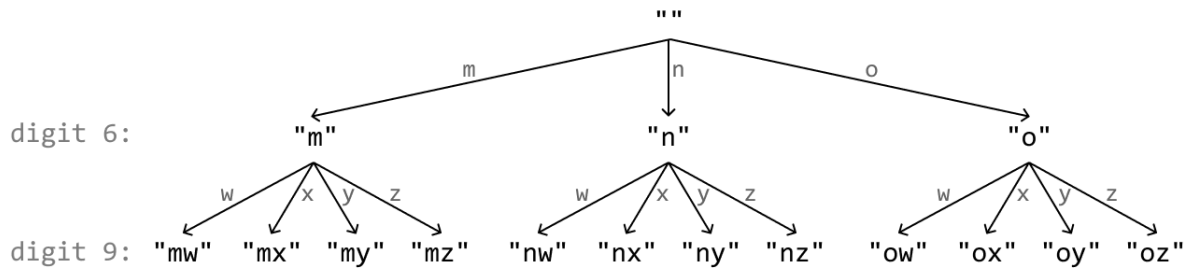
Consider the input string "69". Let's start with the root node of the tree, which is an empty string:

""

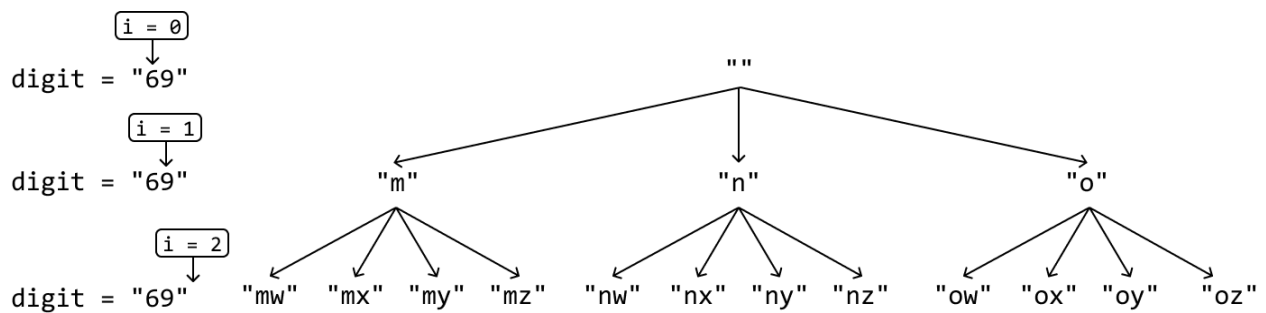
At the first digit, 6, we have the choice of starting our combination with 'm', 'n', or 'o':



For each of these combinations, we now have a new decision to make: which letter of digit 9 ('w', 'x', 'y', 'z') should we choose? These choices are illustrated below:



One important thing missing from this state space tree is information on which digit we're making a decision on at each node. We can use an index *i* to determine which digit we're considering at each node:



The final level of this decision tree (i.e., when *i* == *n*, where *n* denotes the length of the input string) represents all possible combinations that can be created from the provided string. Similar to our approach in *Find All Subsets*, let's use **backtracking** to obtain these keypad combinations.

Mapping digits to letters

The final thing we need to figure out is a way to determine which letters correspond to which digits. A hash map is great for this purpose. In the hash map, digits are the keys, and the associated sets of letters are their values. This allows us to access the letters in constant time:

keypad_map	
'2'	"abc"
'3'	"def"
'4'	"ghi"
'5'	"jkl"
'6'	"mno"
'7'	"pqrs"
'8'	"tuv"
'9'	"wxyz"
digit	letters

Implementation

```
def phone_keypad_combinations(digits: str) -> List[str]:
    keypad_map = {
        '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
        '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
    }
    res = []
    backtrack(0, [], digits, keypad_map, res)
    return res

def backtrack(i: int, curr_combination: List[str], digits: str,
              keypad_map: Dict[str, str], res: List[str]) -> None:
    # Termination condition: if all digits have been considered, add the
    # current combination to the output list.
    if len(curr_combination) == len(digits):
        res.append("".join(curr_combination))
        return
    for letter in keypad_map[digits[i]]:
        # Add the current letter.
        curr_combination.append(letter)
        # Recursively explore all paths that branch from this combination.
        backtrack(i + 1, curr_combination, digits, keypad_map, res)
        # Backtrack by removing the letter we just added.
        curr_combination.pop()
```

Complexity Analysis

Time complexity: The time complexity of `phone_keypad_combinations` is $O(n \cdot 4^n)$. This is because the state space tree will branch down until a decision is made for all n elements. This results in a tree of height n with a branching factor of 4 since there are up to 4 decisions we can make at each digit. For each of the 4^n combinations created, we convert it into a string and add it to the output, which takes $O(n)$ time per combination. This results in a total time complexity of $O(n \cdot 4^n)$.

Space complexity: The space complexity is $O(n)$ due to the recursive call stack, which can grow up to a maximum depth of n . The `keypad_map` only takes constant space since there are only 8 key-value pairs.

Interview Tip

Tip: Check if you can skip trivial implementations.



During an interview, it's crucial to manage your time effectively. If you encounter a trivial and time-consuming task, such as creating the `keypad_map` in this problem, it's possible the interviewer may allow you to skip it or implement it later if there's time left in the interview. Ensure you at least briefly mention how the implementation you're skipping would work before requesting to move on to the core logic of the problem.

Dynamic Programming

Largest Square in a Matrix

Determine the area of the largest square of 1's in a binary matrix.

Example:

1	0	1	1	0
0	0	1	1	1
1	1	1	1	0
1	1	1	0	1
1	1	1	0	1

Output: 9

Intuition

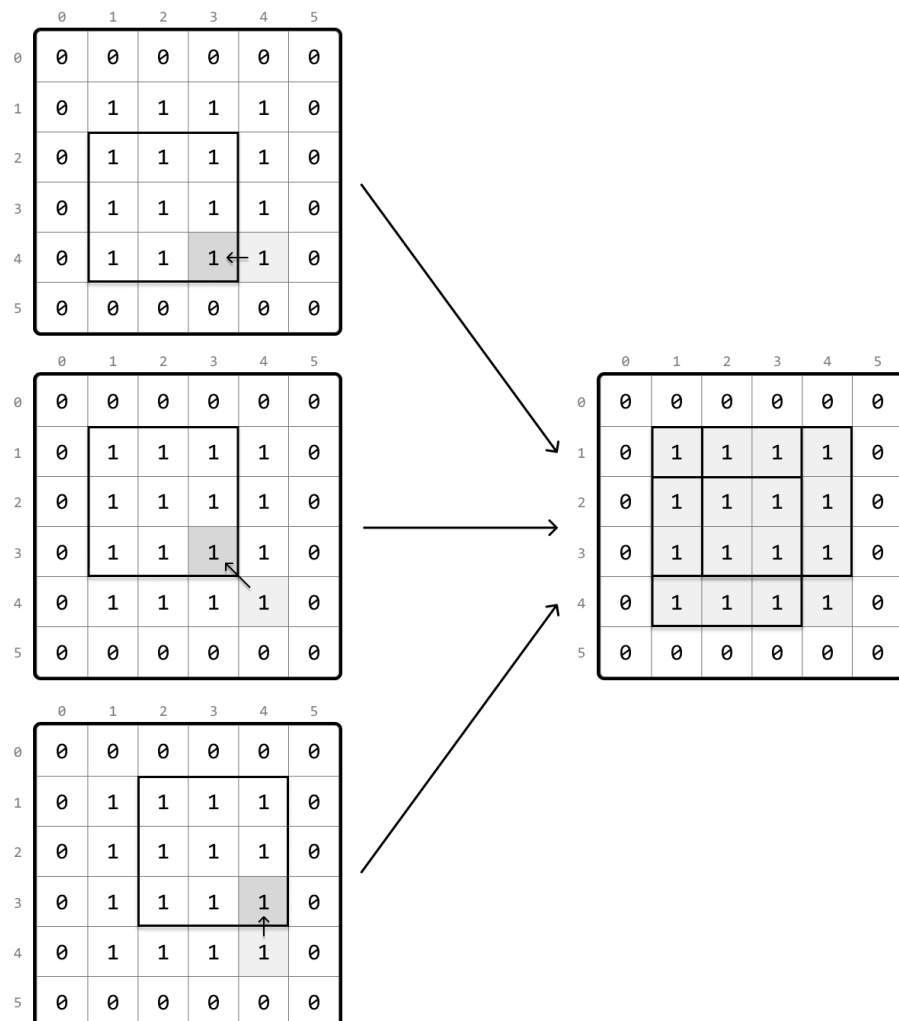
The brute force solution to this problem involves examining every possible submatrix to determine if it forms a square of 1s. This can be done by treating each cell as a potential top-left corner of a square, and checking all possible squares that extend from that cell. For each of these squares, we'll need to verify that all cells within the square are 1's. Repeating this process for each cell allows us to find the largest square. This process is quite inefficient, so let's explore alternatives.

One important thing to understand is that **squares contain smaller squares inside them**. This indicates that subproblems might exist in this problem. Let's see if we can find what the subproblems are and how we can use them. Consider the following 6x6 matrix containing a 4x4 square of 1s:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	1	1	1	0
2	0	1	1	1	1	0
3	0	1	1	1	1	0
4	0	1	1	1	1	0
5	0	0	0	0	0	0

Let's say we're at cell (4, 4). We know just by looking at the matrix that a square of length 4 with all 1s ends at this cell, but what information would we need algorithmically to determine that this 4x4 exists? A key observation here is that there are three 3x3 squares around this cell:

- one that ends directly to the left of the current cell (4, 3).
- one that ends at the top-left diagonal of the current cell (3, 3).
- one that ends directly above the current cell (3, 4).

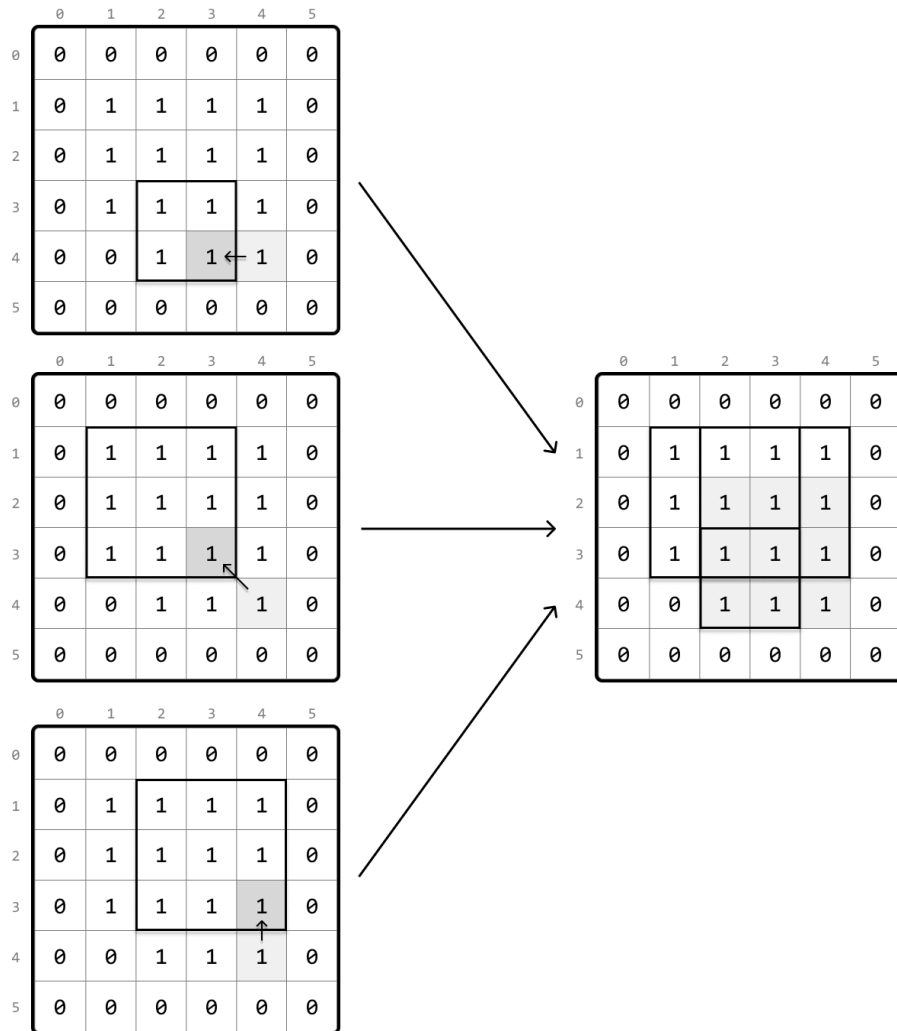


This more clearly highlights the existence of subproblems: the length of a square that ends at the current cell depends on the lengths of the squares that end at its **left**, **top**, and **top-left** neighboring cells.

Let's consider a slightly different scenario, where this time, the input matrix contains one less 1, meaning there's no longer a 4×4 square of 1s:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	1	1	1	0
2	0	1	1	1	1	0
3	0	1	1	1	1	0
4	0	0	1	1	1	0
5	0	0	0	0	0	0

Let's see if our strategy of checking the three neighboring squares around the current cell (4, 4) changes at all here. Keep in mind that this time, the square that ends directly to the left of the current cell only has a length of 2. This means the square that ends at the current cell can, at most, have a length of 3, with 1 unit from the current cell and 2 units from the smallest neighboring square:



This indicates that the length of the current square is restricted by the **smallest of its three neighboring squares**. We can express this as a recursive formula where `matrix[i][j]` represents the value of the current cell, (i, j) :

```
if matrix[i][j] == 1:
    max_square(i, j) = 1 + min(max_square(i - 1, j),
                                max_square(i - 1, j - 1),
                                max_square(i, j - 1))
```

We now have all the information we need. Given this problem has an **optimal substructure**, we can translate the above recurrence relation directly into a DP formula.

```
if matrix[i][j] == 1:
    dp[i][j] = 1 + min(dp[i - 1][j], dp[i - 1][j - 1], dp[i][j - 1])
```

Now, let's think about what the base cases should be.

Base cases

We know $dp[0][0]$ should be 1 if $matrix[0][0]$ is 1 since the top-left cell can only have a square of length 1.

What other base cases are there? Consider row 0 and column 0 of the matrix. These are special because the length of a square ending at any of these cells is at most 1.

So, for the base cases, we can set all cells in row 0 and column 0 to 1 in our DP table, provided those cells in the original matrix are also 1:

DP table:
base cases

	0	1	2	3	4
0	1	0	1	1	0
1	0				
2	1				
3	1				
4	1				

Populating the DP table

We populate the DP table starting from the smallest subproblems (excluding the base cases). Specifically, with row 0 and column 0 populated, we begin by populating cell (1, 1) and work our way down to the last cell ($m - 1, n - 1$) row by row, where m and n are the dimensions of the matrix:

	0	1	2	3	4
0	1	0	1	1	0
1	0	start			
2	1				
3	1				
4	1				end

The largest value in the DP table represents the length of the largest square in our matrix. Therefore, we just need to track the maximum DP value (max_len) as we populate the table. Once done, we just return max_len^2 , which represents the area of the largest square.

Implementation

In this implementation, it's possible to merge the base case handling with the code which populates the DP table. However, in an interview setting, code is often easier to understand when the base cases are defined separately, which is why they're implemented separately here.

```
def largest_square_in_a_matrix(matrix: List[List[int]]) -> int:
    if not matrix:
        return 0
    m, n = len(matrix), len(matrix[0])
    dp = [[0] * n for _ in range(m)]
    max_len = 0

    # Base case: If a cell in row 0 is 1, the largest square ending there has a
    # length of 1.
    for j in range(n):
        if matrix[0][j] == 1:
            dp[0][j] = 1
            max_len = 1

    # Base case: If a cell in column 0 is 1, the largest square ending there has
    # a length of 1.
    for i in range(m):
        if matrix[i][0] == 1:
            dp[i][0] = 1
            max_len = 1

    # Populate the rest of the DP table.
    for i in range(1, m):
        for j in range(1, n):
            if matrix[i][j] == 1:
                # The length of the largest square ending here is determined by
                # the smallest square ending at the neighboring cells (left,
                # top-left, top), plus 1 to include this cell.
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i - 1][j - 1], dp[i][j - 1])
            max_len = max(max_len, dp[i][j])

    return max_len ** 2
```

Complexity Analysis

Time complexity: The time complexity of `largest_square_in_a_matrix` is $O(m \cdot n)$ because each cell of the DP table is populated at most once.

Space complexity: The space complexity is $O(m \cdot n)$ since we're maintaining a 2D DP table that has $m \cdot n$ elements.

Optimization

We can optimize our solution by realizing that, for each cell in the DP table, we only need to access the cell directly above it, the cell to its left, and the top-left diagonal cell.

- To get the cell above it or the top-left diagonal cell, we only need access to the previous row.
- To get the cell to its left, we just need to look at the cell to the left of the current cell in the same row we're 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	4
0	1	0	1	1	0
1	0	0	1	2	1
2	1	1	1	2	0
prev_row: 3	1	2	2	0	1
curr_row: 4	1	→	→	→	→

to calculate row 4,
the only other row we
need is row 3.

This effectively reduces the space complexity to $O(m)$. Below is the optimized code:

```
def largest_square_in_a_matrix_optimized(matrix: List[List[int]]) -> int:
    if not matrix:
        return 0
    m, n = len(matrix), len(matrix[0])
    prev_row = [0] * n
    max_len = 0
    # Iterate through the matrix.
    for i in range(m):
        curr_row = [0] * n
```

```

for j in range(n):
    # Base cases: if we're in row 0 or column 0, the largest square ending
    # here has a length of 1. This can be set by using the value in the
    # input matrix.
    if i == 0 or j == 0:
        curr_row[j] = matrix[i][j]
    else:
        if matrix[i][j] == 1:
            # curr_row[j] = 1 + min(left, top-left, top)
            curr_row[j] = 1 + min(curr_row[j - 1],
                                  prev_row[j - 1],
                                  prev_row[j])

        max_len = max(max_len, curr_row[j])
    # Update 'prev_row' with 'curr_row' values for the next iteration.
    prev_row, curr_row = curr_row, [0] * n
return max_len ** 2

```

Sort and Search

Dutch National Flag

Given an array of 0s, 1s, and 2s representing red, white, and blue, respectively, sort the array in place so it resembles the Dutch national flag, with all reds (0s) coming first, followed by whites (1s), and finally blues (2s).

Example:

Input: `nums = [0, 1, 2, 0, 1, 2, 0]`

Output: `[0, 0, 0, 1, 1, 2, 2]`

Intuition

This problem is just asking us to sort three numbers in ascending order. A straightforward solution would be to use an in-built sorting function. However, this is an $O(n \log(n))$ approach, where n denotes the length of the array, and it isn't taking advantage of an important problem constraint: there are only three types of elements in the array.

To sort these numbers, we essentially want to position all 0s to the left, all 2s to the right, and any 1s in between. A key observation is that if we place the 0s and the 2s in their correct positions, the 1s will automatically be positioned correctly:

1s will naturally get sorted

`[2 0 1 2 0 0 1]` \longrightarrow `[0 0 0 1 1 2 2]`
sort 0s sort 2s

This allows us to focus on only positioning two numbers.

One strategy we could use is to iterate through the array and move any 0s we encounter to the left, and any 2s we encounter to the right.

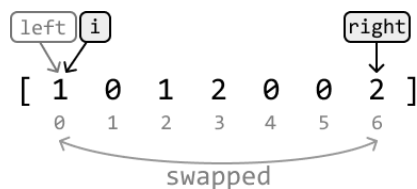
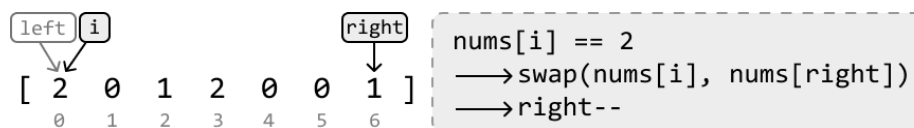
We can set a left pointer to move any 0s we encounter to the left, and a right pointer to move any 2s to the right. To iterate through the array, we can use a separate pointer, *i*:

- When we encounter a 0 at index *i*, swap it with `nums[left]`.
- When we encounter a 2 at index *i*, swap it with `nums[right]`.

To understand how we should adjust these pointers after each swap, let's use the following example:

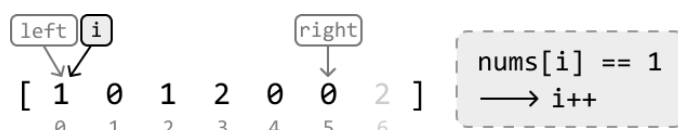
[2 0 1 2 0 0 1]
 0 1 2 3 4 5 6

The first element is 2, so let's swap it with `nums[right]`. Then, let's move the right pointer inward so it points to where the next 2 should be placed:



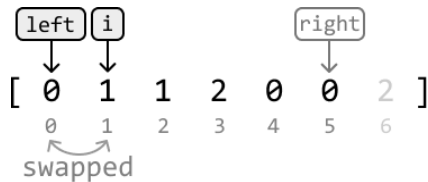
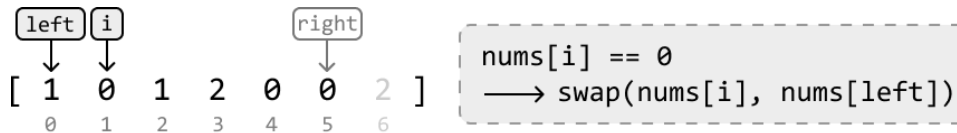
Notice that after this swap, there's now a new element at index *i*. So, we should not yet advance *i*, as we still need to decide whether this new element needs to be positioned elsewhere.

The pointer *i* is now pointing at a 1. We don't need to handle any 1s we encounter, so let's just advance the *i* pointer:





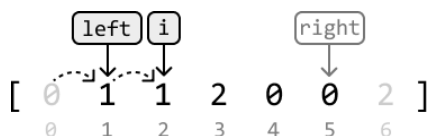
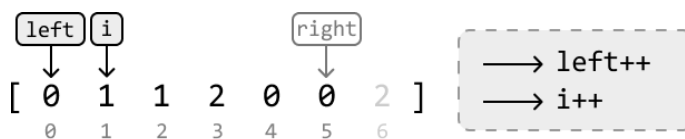
Now, pointer *i* is pointing at a 0, so let's swap it with `nums[left]`:



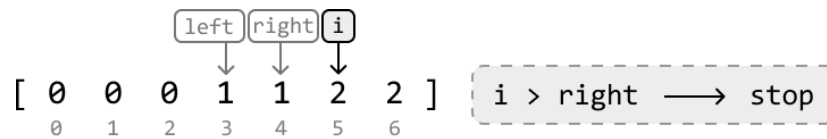
After this swap, there's a new element at index *i*. Since *i* is positioned after the left pointer, this element can only be a 1 for the following reasons:

- Before the swap, all 0s originally to the left of *i* would have already been positioned to the left of the left index.
- Before the swap, all 2s originally to the left of *i* would have already been positioned to the right of the right index.

Therefore, we can also advance the *i* pointer while advancing the left pointer:



We now know what to do whenever we encounter a 0, 1, or 2. We can continue applying this logic until the pointer *i* surpasses the right pointer, indicating all elements have been positioned correctly:

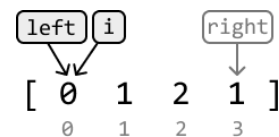


Note that we don't stop the process when `i == right` because the `i` pointer could still be pointing at a 0, which would need to be swapped.

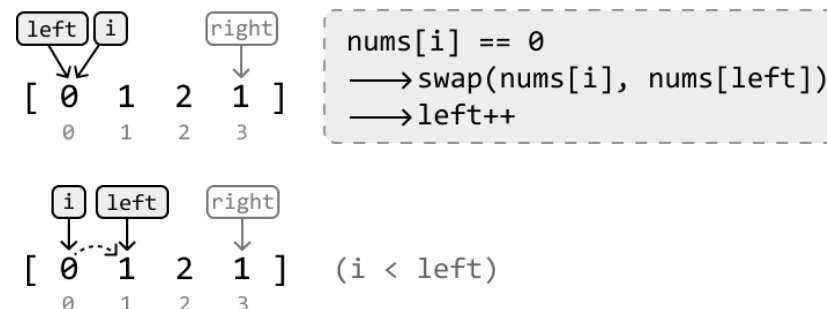
Why do we advance both *i* and *left* pointers when we encounter a 0?

A question we might have regarding the above process is why we advance the `i` pointer along with the `left` pointer when `nums[i] == 0`, since this doesn't happen when we move the `right` pointer.

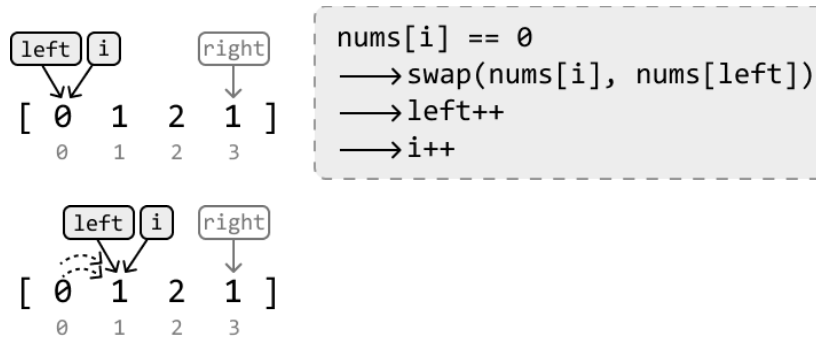
The reason becomes clear when we consider the following example:



Here, `nums[i] == 0`, so the first thing we do is swap `nums[i]` and `nums[left]`, which doesn't change anything in this case since `left` and `i` point to the same element. Now, observe what happens if we only advance the `left` pointer:



As we can see, the `left` pointer will surpass the `i` pointer, which shouldn't happen since `i` needs to stay between `left` and `right` throughout the algorithm. To avoid this, we advance both the `i` and `left` pointers:



Implementation

```
def dutch_national_flag(nums: List[int]) -> None:
    i, left, right = 0, 0, len(nums) - 1
    while i <= right:
        # Swap 0s with the element at the left pointer.
        if nums[i] == 0:
            nums[i], nums[left] = nums[left], nums[i]
            left += 1
            i += 1
        # Swap 2s with the element at the right pointer.
        elif nums[i] == 2:
            nums[i], nums[right] = nums[right], nums[i]
            right -= 1
        else:
            i += 1
```

Complexity Analysis

Time complexity: The time complexity of `dutch_national_flag` is $O(n)$ because we iterate through each element of `nums` once.

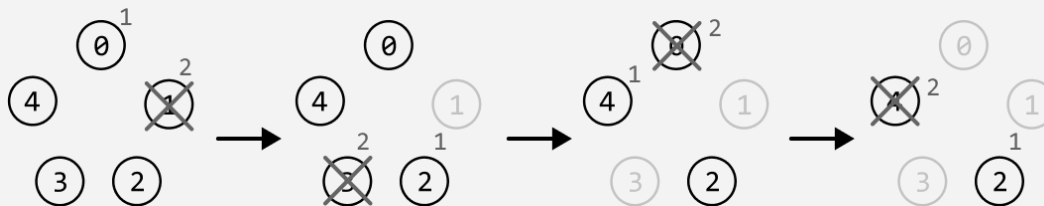
Space complexity: The space complexity is $O(1)$.

Math and Geometry

The Josephus Problem

There are n people standing in a circle, numbered from 0 to $n - 1$. Starting from person 0, count k people clockwise and remove the k^{th} person. After the removal, begin counting again from the next person still in the circle. Repeat this process until only one person remains, and return that person's position.

Example:



Input: $n = 5$, $k = 2$

Output: 2

Constraints:

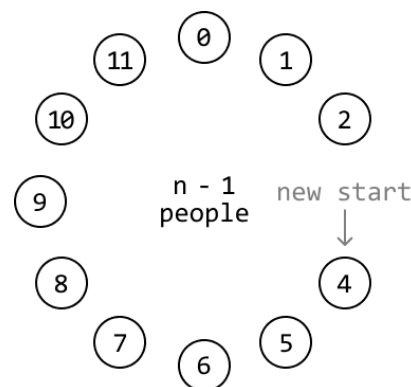
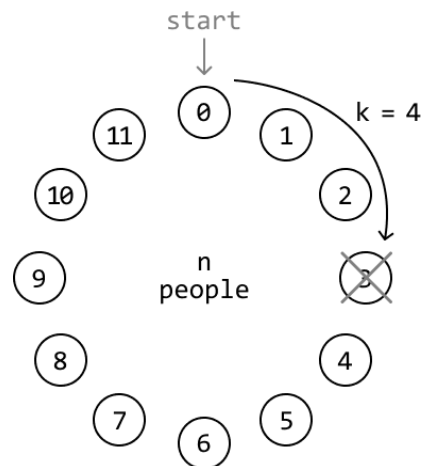
- There will be at least one person in the circle.
- k will at least be equal to 1.

Intuition

The naive approach to solving this problem is to simulate the removal of people step by step. We can create a circular linked list with n nodes. Starting from node 0, we iterate through the linked list, removing every k^{th} person. The last node remaining after all removals will represent the last remaining person.

This approach takes $O(n \cdot k)$ time because, to remove a node, we must iterate through k nodes in the linked list. Therefore, for each of the n nodes, we perform k iterations. Let's see if we can find a faster solution.

Consider an example where $n = 12$ and $k = 4$. For the first removal, we start counting k nodes from node 0 and remove the person we end up on after counting.



As we can see, after the first removal, there is one less person in the circle. Additionally, after the removal, our new start position is at the k^{th} position (person 4).

Now, we effectively need to find the last person remaining in a circle of $n - 1$ people, where we start counting at person k . This indicates that solving the **subproblem** $\text{josephus}(n - 1, k)$ will help us get the answer to the problem $\text{josephus}(n, k)$. Note that the answer to subproblem $\text{josephus}(i, k)$ represents the last person standing in a circle of i people, where we start counting at person 0.

To account for the adjusted start position, we need to add k to the answer returned by $\text{josephus}(n - 1, k)$. This is because, in this subproblem, it won't know to start counting from position 4: it will, by default, count from position 0. So, adding k to this subproblem's result accounts for this difference in the starting position.

This can be expressed with the following recurrence relation:

$$\text{josephus}(n, k) = \text{josephus}(n - 1, k) + k$$

The final consideration is ensuring the value of $\text{josephus}(n - 1, k) + k$ doesn't exceed $n - 1$, as $n - 1$ represents the position of the last person. We can achieve this by applying the modulus operator ($\% n$) to $\text{josephus}(n - 1, k) + k$. This results in the following updated recurrence relation:

$$\text{josephus}(n, k) = (\text{josephus}(n - 1, k) + k) \% n$$

Now, all we need is a base case.

Base case

The simplest version of this problem is when the circle contains only one person: $n = 1$. In this case, the last person remaining is person 0, so we just return person 0 for this base case.

Implementation

```
def josephus(n: int, k: int) -> int:
    # Base case: If there's only one person, the last person is person 0.
    if n == 1:
        return 0

    # Calculate the position of the last person remaining in the reduced problem
    # with 'n - 1' people. We use modulo 'n' to ensure the answer doesn't exceed
    # 'n - 1'.
    return (josephus(n - 1, k) + k) % n
```

Complexity Analysis

Time complexity: The time complexity of josephus is $O(n)$ because we make a total of n recursive calls to this function until we reach the base case.

Space complexity: The space complexity is $O(n)$ due to the recursive call stack, which grows up to a depth of n .

Optimization

We can implement the top-down recursive solution above using a bottom-up iterative approach. Let `res` represent an array that stores the solution to each subproblem, where `res[i]` contains the solution to the subproblem of an i -person circle. Using this array, our formula becomes:

```
res[i] = (res[i - 1] + k) % i
```

The key observation here is that we only ever need access to the previous element of the `res` array (at $i - 1$) to calculate the result of the current subproblem (at i). This means we don't need to store the entire array.

Instead, we can use a single variable to keep track of the solution to the previous subproblem. We can then update this variable to store the solution for the current subproblem:

```
res = (res + k) % i
```

Note that the 'res' value used on the right-hand side of the equation represents the previous subproblem's result.

Implementation

```
def josephus_optimized(n: int, k: int) -> int:
    res = 0
    for i in range(2, n + 1):
        # res[i] = (res[i - 1] + k) % i.
        res = (res + k) % i
    return res
```

Complexity Analysis

Time complexity: The time complexity of `josephus_optimized` is $O(n)$ because we iterate through n subproblems.

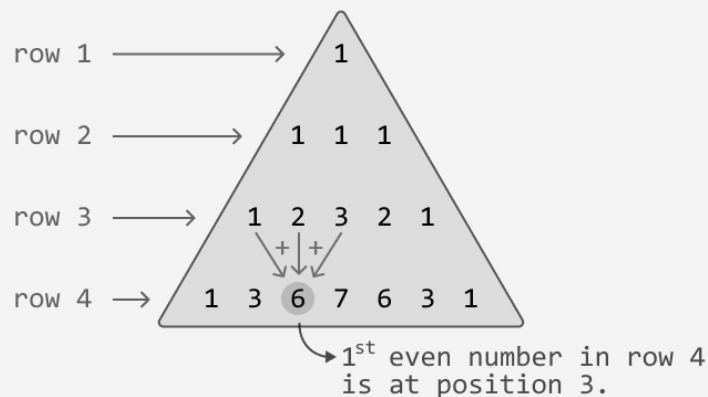
Space complexity: The space complexity is $O(1)$.

Triangle Numbers

Consider a triangle composed of numbers where the top of the triangle is 1. Each subsequent number in the triangle is equal to the sum of three numbers above it: its top-left number, its top number, and its top-right number. If any of these three numbers don't exist, assume they are equal to 0.

Given a value representing a row of this triangle, return the position of the first even number in this row. Assume the first number in each row is at position 1.

Example:



Input: $n = 4$

Output: 3

Constraints:

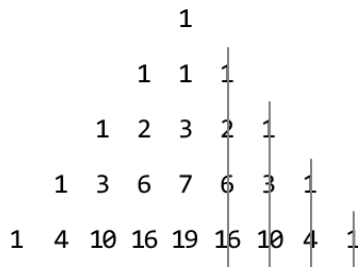
- n will be at least 3.

Intuition

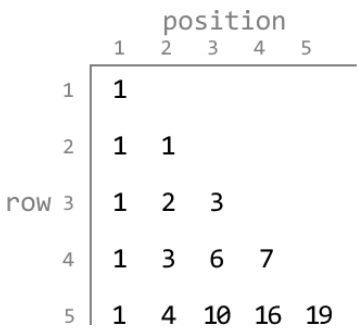
A naive solution to this problem is to generate the entire triangle and all of its values up to the n^{th} row. Then, we can iterate through the n^{th} row until we encounter the first even number. However, this approach is inefficient because it results in an excessive use of time and memory to build the entire triangle. To find a more optimal solution, let's consider how we can simplify the representation of our triangle.

Simplifying the triangle

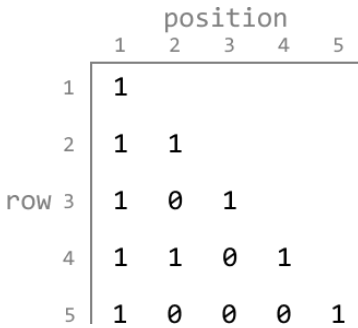
The first key observation is that the triangle is symmetric. This means we can exclude the right half of the triangle because if an even number exists in the right half, it definitely exists in the left half:



To more easily visualize the positions of the numbers in each row, let's draw it such that numbers belonging to the same position are aligned:



The next key observation is that we don't necessarily care about the values themselves: we only care about the parity of each number (i.e., if they're even or odd). Given this, we can simplify the triangle further by representing it as a binary triangle where 0 represents an even number and 1 represents an odd number:



Now that we've simplified the triangle, it'll be easier to identify patterns in the positions of the first even number in each row. Let's explore this further.

Identifying patterns

Let's ignore rows 1 and 2 since even numbers only begin appearing from row 3 onward. A good place to start looking for a pattern is to highlight the first even number at each row and observe their positions:

		position				
		1	2	3	4	5
1		1				
2		1	1			
row (odd) 3		1	0	1		
(even) 4		1	1	0	1	
(odd) 5		1	0	0	0	1

From rows 3 to 5, one possible pattern to observe is that odd-numbered rows have the first even number at position 2. We could also hypothesize that even-numbered rows have the first even at position 3.

To confirm if this observation is consistent, let's look at some more rows:

		position						
		1	2	3	4	5	6	7
1		1						
2		1	1					
(odd) 3		1	0	1				
row 4		1	1	0	1			
(odd) 5		1	0	0	0	1		
6		1	1	1	0	1	1	
(odd) 7		1	0	1	0	0	0	1

So far, our hypothesis for odd-numbered rows is still true, but even-numbered rows seem to be following a different pattern. It's still hard to pinpoint what it could be.

Let's continue by displaying a few more rows to figure out what this pattern is:

	position									
	1	2	3	4	5	6	7	8	9	10
1	1									
2	1	1								
(odd) 3	1	0	1							
4	1	1	0	1						
(odd) 5	1	0	0	0	1					
6	1	1	1	0	1	1				
(odd) 7	1	0	1	0	0	0	1			
8	1	1	0	1	1	0	1	1		
(odd) 9	1	0	0	0	0	0	0	0	0	
10	1	1	1	0	0	0	0	0	0	0

Now, we notice that the first four binary values from rows 3 to 6 repeat for rows 7 to 10. If we were to continue for future rows, we would notice that this pattern continues.

Essentially, the following pattern is consistently repeated, starting from row 3:

3:	1	0	1	0
4:	1	1	0	1
5:	1	0	0	0
6:	1	1	1	0

To understand why this pattern repeats, it's important to realize that the first four values of a row are calculated solely from the four values of the previous row. We can see this visualized below, using the initial representation of the triangle to make it clearer:

Diagram illustrating the iterative step in the construction of a 1D cellular automaton. The sequence of numbers (1, 2, 3, 6, 7) is shown, with arrows indicating transitions and additions. The sequence evolves from 1 to 1 1 1, then to 1 2 1 2 1, and finally to 1 3 6 7 6 3 1. The diagram shows the addition of new elements to the sequence, resulting in the final sequence 1 3 6 7 6 3 1.

So, whenever a specific sequence of four numbers occurs at the beginning of a row, it will generate a predictable sequence of four numbers in the following row. Extending this observation to the

entire pattern, we can conclude that since the pattern repeated once (from rows 3 to 6 to rows 7 to 10), it will continue to repeat indefinitely.

This gives us the below cyclic rationale:

- If n is odd ($n \% 2 \neq 0$), return 2.
- If n is a multiple of 4 ($n \% 4 == 0$), return 3.
- Else, return 4.

all odd rows: position 2	all rows that are a multiple of 4: position 3	all other rows: position 4
↓	↓	↓
3: 1 0 1 0	3: 1 0 1 0	3: 1 0 1 0
4: 1 1 0 1	4: 1 1 0 1	4: 1 1 0 1
5: 1 0 0 0	5: 1 0 0 0	5: 1 0 0 0
6: 1 0 1 0	6: 1 0 1 0	6: 1 0 1 0
7: 1 0 1 0	7: 1 0 1 0	7: 1 0 1 0
8: 1 1 0 1	8: 1 1 0 1	8: 1 1 0 1
9: 1 0 0 0	9: 1 0 0 0	9: 1 0 0 0
10: 1 1 1 0	10: 1 1 1 0	10: 1 1 1 0

This problem demonstrates how recognizing patterns and simplifying the problem can turn a time-consuming solution into a quick, constant-time one.

Implementation

```
def triangle_numbers(n: int) -> int:
    # If n is an odd-numbered row, the first even number always starts at position
    # 2.
    if n % 2 != 0:
        return 2

    # If n is a multiple of 4, the first even number always starts at position 3.
    elif n % 4 == 0:
        return 3

    # For all other rows, the first even number always starts at position 4.
    return 4
```

Complexity Analysis

Time complexity: The time complexity of `triangle_numbers` is $O(1)$.

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