Explain code with example

Let's go through the code step-by-step.

Code:

```python
class Solution:
    def maximumWealth(self, accounts: list[list[int]]) -> int:
        max_wealth = 0  # Initialize maximum wealth to 0
        for account in accounts:
            # Sum the wealth of the current customer
            wealth = sum(account)
            # Update max_wealth if the current customer's wealth is greater
            max_wealth = max(max_wealth, wealth)
        return max_wealth
```

Explanation:

- The method **`maximumWealth`** takes a list of lists called **`accounts`**.
- Each inner list represents the amount of money a customer has in multiple bank accounts.
- The task is to find the **maximum total wealth** among all customers.
- **`max_wealth`** keeps track of the largest total wealth seen so far.

Step-by-step example:

Suppose:

```python
accounts = [
    [1, 2, 3],
    [3, 2, 1],
    [4, 5, 1]
]
```

Now, let's trace the execution:

1. **Initialization**

   ```python
   max_wealth = 0
   ```

Initially, no customer has been checked, so **`max_wealth = 0`**.

2. **First iteration** (**`account = [1, 2, 3]`**)

   ```
   python

   wealth = sum([1, 2, 3])  # wealth = 6
   max_wealth = max(0, 6)   # max_wealth = 6
   ```

   The first customer's total wealth = 6.
   Since 6 > 0, update **`max_wealth = 6`**.

3. **Second iteration** (**`account = [3, 2, 1]`**)

   ```
   python

   wealth = sum([3, 2, 1])  # wealth = 6
   max_wealth = max(6, 6)   # max_wealth = 6
   ```

   The second customer's total wealth = 6.
   It equals the current maximum, so **`max_wealth`** stays 6.

4. **Third iteration** (**`account = [4, 5, 1]`**)

   ```
   python

   wealth = sum([4, 5, 1])  # wealth = 10
   max_wealth = max(6, 10)  # max_wealth = 10
   ```

   The third customer's total wealth = 10.
   Since 10 > 6, update **`max_wealth = 10`**.

5. **End of loop**

   ```
   python

   return max_wealth
   ```

   The final result returned is **`10`**.

Output:

```
python


solution = Solution()
print(solution.maximumWealth([[1,2,3],[3,2,1],[4,5,1]]))
# Output: 10
```

Summary of flow:

Customer	Accounts	Wealth	max_wealth after iteration
1	[1, 2, 3]	6	6
2	[3, 2, 1]	6	6
3	[4, 5, 1]	10	10

Final returned value = **10**

Let’s break down this code and walk through it step by step with an example.

Code:

```
python

class Solution:
    def kidsWithCandies(self, candies: list[int], extraCandies: int) -> list[bool]:
        max_candies = max(candies)  # Find the current maximum number of candies
        return [candy + extraCandies >= max_candies for candy in candies]
```

Explanation:

- The function **`kidsWithCandies`** takes two parameters:
  - **`candies`**: a list of integers where each number represents how many candies a child currently has.
  - **`extraCandies`**: the number of extra candies you can give to each child.
- The goal is to find out, **for each child**, whether giving them all the extra candies would make them have **the greatest or equal number of candies** compared to others.
- The function returns a **list of boolean values (`True` or `False`)**.

Step-by-step example:

Suppose:

```
python

candies = [2, 3, 5, 1, 3]
extraCandies = 3
```

---

1. **Find the maximum candies among all children**

   ```
   python

   max_candies = max(candies)  # max_candies = 5
   ```

   Currently, the child with the most candies has 5.

---

2. **List comprehension**

```python
[candy + extraCandies >= max_candies for candy in candies]
```

This goes through each child and checks if:

```sql
(current candies + extraCandies) >= max_candies
```

3. **Evaluate step by step**

Child	Candies	After adding extra	Comparison	Result
1	2	2 + 3 = 5	5 >= 5	True
2	3	3 + 3 = 6	6 >= 5	True
3	5	5 + 3 = 8	8 >= 5	True
4	1	1 + 3 = 4	4 >= 5	False
5	3	3 + 3 = 6	6 >= 5	True

4. **Final returned list**

```python
[True, True, True, False, True]
```

Output:

```python
solution = Solution()
print(solution.kidsWithCandies([2, 3, 5, 1, 3], 3))
# Output: [True, True, True, False, True]
```

Summary of logic:

1. Find the maximum number of candies any child currently has.
2. For each child, check if adding all extra candies would make them at least as rich in candies as the richest child.
3. Return a list of True/False values for each child based on that comparison.

Let’s go through this code line by line and explain what happens, including a ste
with an example.

Code:

```
python

class Solution:
    def reverse(self, x: int) -> int:
        # Handle negative numbers
        sign = -1 if x < 0 else 1
        x = abs(x)

        # Reverse the digits of the integer
        reversed_x = 0
        while x != 0:
            # Pop the last digit
            digit = x % 10
            x //= 10

            # Push the digit into the reversed integer
            reversed_x = reversed_x * 10 + digit

        # Apply the sign to the result
        reversed_x *= sign

        # Check for overflow: 32-bit signed integer range is [-2^31, 2^31 - 1]
        if reversed_x < -2**31 or reversed_x > 2**31 - 1:
            return 0

        return reversed_x
```

Explanation

This function reverses the digits of a 32-bit signed integer.
If the reversed integer overflows the 32-bit range, it returns **`0`**.

Step-by-step example

Let’s take:

```
python

x = -123
```

Step 1: Handle negative numbers

```python
sign = -1 if x < 0 else 1
x = abs(x)
```

- `x < 0` → True → `sign = -1`
- `x = abs(-123)` → `x = 123`

We'll reverse `123` first, and later multiply by `-1` to restore the negative sign.

Step 2: Initialize reversed_x

```python
reversed_x = 0
```

This will store the reversed number as we build it.

Step 3: Reverse the digits using a while loop

The loop continues as long as `x != 0`.

We repeatedly:

- Extract the **last digit** using `% 10`
- Remove the **last digit** using integer division `// 10`
- Append that digit to the reversed number

Iteration 1:

- `digit = x % 10 = 123 % 10 = 3`
- `x = x // 10 = 123 // 10 = 12`
- `reversed_x = reversed_x * 10 + digit = 0 * 10 + 3 = 3`

Now:

```ini
x = 12
reversed_x = 3
```

Iteration 2:

- **`digit = 12 % 10 = 2`**
- **`x = 12 // 10 = 1`**
- **`reversed_x = 3 * 10 + 2 = 32`**

Now:

```ini
x = 1
reversed_x = 32
```

Iteration 3:

- **`digit = 1 % 10 = 1`**
- **`x = 1 // 10 = 0`**
- **`reversed_x = 32 * 10 + 1 = 321`**

Now:

```ini
x = 0
reversed_x = 321
```

Loop ends since **`x == 0`**.

Step 4: Apply the sign

```python
reversed_x *= sign
```

`reversed_x = 321 * -1 = -321`

Step 5: Check for overflow

```python
if reversed_x < -2**31 or reversed_x > 2**31 - 1:
    return 0
```

The valid 32-bit signed integer range is:

```css
[-2147483648, 2147483647]
```

-321 is within range, so no overflow.

Step 6: Return result

```python
return reversed_x
```

Output:

```diff
-321
```

Another example

Input:

```python
x = 120
```

Step-by-step:

Step	x before	digit (x % 10)	x after (x // 10)	reversed_x after
1	120	0	12	0
2	12	2	1	2
3	1	1	0	21

Final reversed value = **21**
Returned output = **21**

Summary of logic

1. Record the sign (**+** or **-**).
2. Work with the absolute value of the integer.
3. Extract digits one by one and build the reversed integer.
4. Reapply the original sign.

5. Check for overflow; if it occurs, return 0.
6. Return the reversed integer.

Let's carefully go through this code line by line and then walk through an exam
how it executes internally.

Code:

```python
class Solution:
    def romanToInt(self, s: str) -> int:
        # Mapping of Roman numerals to integers
        roman_map = {
            'I': 1, 'V': 5, 'X': 10, 'L': 50,
            'C': 100, 'D': 500, 'M': 1000
        }

        # Initialize the result integer
        total = 0

        # Iterate over the string, checking each character except the last one
        for i in range(len(s) - 1):
            # If current numeral is smaller than the next one, subtract it
            if roman_map[s[i]] < roman_map[s[i + 1]]:
                total -= roman_map[s[i]]
            else:
                total += roman_map[s[i]]

        # Add the last numeral's value (always add it)
        total += roman_map[s[-1]]

        return total
```

Purpose

This function converts a **Roman numeral string** into its **integer equivalent**.

Roman numeral rules reminder:

Roman numerals use combinations of letters:

```mathematica
I=1, V=5, X=10, L=50, C=100, D=500, M=1000
```

- Normally, numerals are added left to right (**`VI = 5 + 1 = 6`**)
- If a smaller numeral appears before a larger one, you subtract it (**`IV = 5 - 1 = 4`**)

Example walkthrough

Input:

```
python

s = "MCMXCIV"
```

We expect output **`1994`**.

Step 1: Setup

```
python

roman_map = {'I':1, 'V':5, 'X':10, 'L':50, 'C':100, 'D':500, 'M':1000}
total = 0
```

String indices:

Index	Character	Value
0	M	1000
1	C	100
2	M	1000
3	X	10
4	C	100
5	I	1
6	V	5

Step 2: Loop through `len(s) - 1` → from index 0 to 5

i = 0:

Compare **`s[0] = 'M' (1000)`** and **`s[1] = 'C' (100)`**

```
yaml

1000 < 100 ? No
→ total += 1000
total = 1000
```

---

i = 1:

Compare **`s[1] = 'C' (100)`** and **`s[2] = 'M' (1000)`**

```
yaml

100 < 1000 ? Yes
→ total -= 100
```

```
total = 900
```

---

i = 2:

Compare `s[2] = 'M' (1000)` and `s[3] = 'X' (10)`

```yaml
1000 < 10 ? No
→ total += 1000
total = 1900
```

---

i = 3:

Compare `s[3] = 'X' (10)` and `s[4] = 'C' (100)`

```yaml
10 < 100 ? Yes
→ total -= 10
total = 1890
```

---

i = 4:

Compare `s[4] = 'C' (100)` and `s[5] = 'I' (1)`

```yaml
100 < 1 ? No
→ total += 100
total = 1990
```

---

i = 5:

Compare `s[5] = 'I' (1)` and `s[6] = 'V' (5)`

```yaml
1 < 5 ? Yes
→ total -= 1
total = 1989
```

Step 3: Add last numeral

After loop, always add the last one (`s[-1] = 'V' = 5`)

```makefile
total += 5
total = 1994
```

Step 4: Return result

```python
return total
```

Output:

```yaml
1994
```

Another example

`s = "LVIII"`

Symbol	Meaning	Operation	Running total
L (50)	Add	+50	50
V (5)	Add	+5	55
I (1)	Add	+1	56
I (1)	Add	+1	57
I (1)	Add	+1	58

Output: **`58`**

Summary of algorithm

1. Map Roman numerals to integers.
2. Loop through the string except the last character:
   - If current numeral < next numeral → subtract it.
   - Otherwise → add it.
3. Add the last numeral (since it's never subtracted).
4. Return the total.

Let’s go line by line through this code, then walk through an example to see ho
internally.

Code:

```
python

class Solution:
    def isValid(self, s: str) -> bool:
        # Stack to keep track of opening parentheses
        stack = []

        # Mapping of closing parentheses to their corresponding opening parentheses
        mapping = {')': '(', '}': '{', ']': '['}

        # Iterate through each character in the string
        for char in s:
            # If it's a closing parenthesis
            if char in mapping:
                # Pop the top element from the stack, or use a dummy value if the
stack is empty
                top_element = stack.pop() if stack else '#'

                # If the popped element doesn't match the corresponding opening
parenthesis, return False
                if mapping[char] != top_element:
                    return False
            else:
                # If it's an opening parenthesis, push it onto the stack
                stack.append(char)

        # If the stack is empty, all the parentheses were matched correctly
        return not stack
```

Purpose

This function checks whether a string containing parentheses (`()`, `{}`, `[]`) is **valid**.
A valid string means:

- Every opening bracket has a matching closing bracket of the same type.
- Brackets are properly nested.

Data structures used

- **Stack**: to store opening brackets.
- **Mapping**: to check if closing brackets match the correct opening ones.

Step-by-step example

Input:

```python
s = "({[]})"
```

We expect the output to be **`True`**.

Step 1: Initialize

```python
stack = []
mapping = {')': '(', '}': '{', ']': '['}
```

Step 2: Iterate over the string

Step	char	Action	stack before	stack after	Result / Reason
1	**(**	opening	[]	['(']	push opening bracket
2	**{**	opening	['(']	['(', '{']	push opening bracket
3	**[**	opening	['(', '{']	['(', '{', '[']	push opening bracket
4	**]**	closing	['(', '{', '[']	['(', '{']	pop **[**, matches **[**
5	**}**	closing	['(', '{']	['(']	pop **{**, matches **{**
6	**)**	closing	['(']	[]	pop **(**, matches **(**

Step 3: Final check

After the loop:

```python
return not stack
```

`stack` is empty → all brackets were matched → return **`True`**.

Example 2 — Invalid input

Input:

```python
s = "({[)]}"
```

Trace:

Step	char	Action	stack before	stack after	
1	**(**	opening	[]	['(']	push
2	**{**	opening	['(']	['(', '{']	push
3	**[**	opening	['(', '{']	['(', '{', '[']	push
4	**)**	closing	['(', '{', '[']	top = **[**, not **(**	mismatch

At step 4, the mapping for `)` expects `(`, but the top element is `[`.
So:

```python
return False
```

Example 3 — Edge cases

1. `s = ""`
   → empty string → stack remains empty → returns `True`.
2. `s = "("`
   → one unmatched opening bracket → stack not empty → returns `False`.
3. `s = ")"`
   → starts with closing → stack empty → dummy `#` used → mismatch → returns `False`.

Summary of algorithm

1. Use a stack to store opening brackets.
2. When a closing bracket appears:
   - Pop the top of the stack.
   - Check if it matches the correct opening bracket using `mapping`.
   - If not, return `False`.
3. After the loop:
   - If the stack is empty → all brackets matched → return `True`.
   - If not empty → there are unmatched openings → return `False`.

Time and Space Complexity

- **Time Complexity:** O(n) — each character is processed once.

- **Space Complexity:** O(n) — in the worst case (all opening brackets), the sta
  elements.

Let's go through this code carefully, fix its indentation, explain how it works, an
by step with an example.

Corrected and properly formatted code

```
python


class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        # Create a dummy node to simplify edge cases
        dummy = ListNode()
        current = dummy

        # Traverse both lists
        while l1 and l2:
            if l1.val < l2.val:
                current.next = l1
                l1 = l1.next
            else:
                current.next = l2
                l2 = l2.next
            current = current.next

        # If one list is not empty, append it to the result
        if l1:
            current.next = l1
        elif l2:
            current.next = l2

        # Return the merged list starting from the node after the dummy node
        return dummy.next
```

Purpose

This function merges two **sorted linked lists** into a single sorted linked list and returns the **head** of the merged list.

Step-by-step explanation

1. Dummy node

A dummy node is created:

```
python


dummy = ListNode()
current = dummy
```

- It acts as a placeholder for the head of the merged list.
- **`current`** will be used to build the new list.
- At the end, we return **`dummy.next`** (the first real node of the merged list).

---

2. Traverse both lists

The **`while`** loop runs as long as both lists have nodes:

```
python

while l1 and l2:
```

At each step:

- Compare the current node values from both lists.
- Append the smaller node to the merged list.
- Move forward in that list.
- Advance the **`current`** pointer to the newly added node.

---

3. Append the remainder

After one list is fully processed, the other list may still have remaining nodes.
Since both lists are sorted, we can attach the remaining part directly:

```
python

if l1:
    current.next = l1
elif l2:
    current.next = l2
```

---

4. Return merged list

Finally:

```
python

return dummy.next
```

We skip the dummy head and return the start of the merged list.

Example walkthrough

Input lists:

```makefile
l1: 1 → 2 → 4
l2: 1 → 3 → 4
```

Step 1:

Initialize:

```sql
dummy → None
current → dummy
```

Step 2: Compare and build

Step	l1.val	l2.val	Append	Resulting merged list	Move pointer
1	1	1	l2 (equal, either works)	1	l2 → 3
2	1	3	l1	1 → 1	l1 → 2
3	2	3	l1	1 → 1 → 2	l1 → 4
4	4	3	l2	1 → 1 → 2 → 3	l2 → 4
5	4	4	l2 (equal, choose either)	1 → 1 → 2 → 3 → 4	l2 → None

Step 3:

`l2` is now **`None`**, but `l1` still has one node left (**`4`**).

Attach it:

```ini
current.next = l1
```

Merged list:

```
1 → 1 → 2 → 3 → 4 → 4
```

Step 4:

Return:

```python
return dummy.next
```

Output head points to:

```
1 → 1 → 2 → 3 → 4 → 4
```

Summary of logic

1. Use a dummy node to simplify list merging.
2. Use two pointers (`l1`, `l2`) to traverse both lists.
3. At each step, attach the smaller node to the merged list.
4. If one list is exhausted, attach the remaining part of the other.
5. Return `dummy.next` (head of merged list).

Time and Space Complexity

- **Time Complexity:** O(n + m) — traverse both lists once.
- **Space Complexity:** O(1) — no extra data structure used (only pointers).

Problem Name:
27. Remove Element (LeetCode Problem)

Correct and properly formatted version of your code:

```
python

class Solution:
    def removeElement(self, nums: list[int], val: int) -> int:
        # Initialize a pointer for the new array
        k = 0

        # Traverse the original array
        for i in range(len(nums)):
            # If the current element is not the value to be removed, move it to index
'k'
            if nums[i] != val:
                nums[k] = nums[i]
                k += 1

        # Return the new length of the array
        return k
```

Problem Description

Given an integer array **`nums`** and an integer **`val`**,
remove all occurrences of **`val`** **in-place**.
The relative order of the elements may be changed.

After removal, return the **number of elements** left (new length).

You do **not** need to allocate extra space;
modify the array in-place with O(1) extra memory.

Example

Input:

```
python

nums = [3, 2, 2, 3]
val = 3
```

Process:

Step	i	nums[i]	Condition (!= val)	Action	nums after step	k
1	0	3	False	skip	[3, 2, 2, 3]	0
2	1	2	True	nums[0] = 2	[2, 2, 2, 3]	1

Step	i	nums[i]	Condition (`!= val`)	Action	nums aft	
3	2	2	True	nums[1] = 2	[2, 2, 2, 3]	
4	3	3	False	skip	[2, 2, 2, 3]	2

Output:

```
python


k = 2
nums = [2, 2, _, _]  # (the rest of elements are irrelevant)
```

So the function returns **`2`**, and the first **`2`** elements of **`nums`** are the ones kept.

Explanation of logic

- Use index **`k`** to track the next position to write an element that is **not equal to `val`**.
- Iterate over each element:
  - If it's not equal to **`val`**, copy it to **`nums[k]`** and increment **`k`**.
- After the loop, **`k`** is the count of kept elements.
- The array's first **`k`** elements are the filtered ones.

Time and Space Complexity

- **Time Complexity:** O(n) — each element is checked once.
- **Space Complexity:** O(1) — done in-place, no extra space used.

Problem Name:
28. Find the Index of the First Occurrence in a String (LeetCode Problem)

Code:

```python
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        # If needle is an empty string, return 0
        if needle == "":
            return 0

        # Get the lengths of haystack and needle
        n, m = len(haystack), len(needle)

        # If the needle is longer than the haystack, it cannot be found
        if m > n:
            return -1

        # Iterate through the haystack and compare substrings
        for i in range(n - m + 1):
            if haystack[i:i+m] == needle:
                return i

        # If needle is not found, return -1
        return -1
```

Explanation

The function returns the index of the **first occurrence** of substring **`needle`** in string **`haystack`**. If **`needle`** is not found, it returns **`-1`**.

Step-by-step breakdown

1. **Check for empty needle**

```python
if needle == "":
    return 0
```

If **`needle`** is empty, by definition return 0.

2. **Get lengths**

```python
n, m = len(haystack), len(needle)
```

n → length of haystack
m → length of needle

---

3. **Check size condition**

```
python

if m > n:
    return -1
```

If **needle** is longer than **haystack**, substring match is impossible.

---

4. **Search for substring**

```
python

for i in range(n - m + 1):
    if haystack[i:i+m] == needle:
        return i
```

- Iterate through each possible starting position **i** in **haystack**.
- Extract substring of length **m** starting at **i** → **haystack[i:i+m]**.
- Compare it with **needle**.
- If they match, return **i** (the start index of the first match).

---

5. **If not found**

```
python

return -1
```

If loop finishes without finding a match, return -1.

Example walkthrough

Input:

```
python

haystack = "hello"
needle = "ll"
```

Step 1:

`needle` is not empty → continue.

Step 2:

`n = 5`, **`m = 2`**

Step 3:

`m < n` → continue.

Step 4: Loop **`for i in range(n - m + 1)`**

`range(5 - 2 + 1) = range(4)` → i = 0, 1, 2, 3

i	haystack[i:i+m]	Comparison	Match?	Action
0	"he"	"he" == "ll"?	No	continue
1	"el"	"el" == "ll"?	No	continue
2	"ll"	"ll" == "ll"?	Yes	return 2

At i = 2, match found → function returns 2.

Output:

```
2
```

Another example

Input:

```
python

haystack = "aaaaa"
needle = "bba"
```

Trace:

i	haystack[i:i+3]	Comparison	Match
0	"aaa"	!= "bba"	No
1	"aaa"	!= "bba"	No
2	"aaa"	!= "bba"	No

No match found → returns **`-1`**.

Summary of logic

1. If **`needle`** is empty → return 0
2. For each possible start index in **`haystack`**, compare substring with **`needle`**
3. Return first matching index if found
4. If no match → return -1

Problem Name:
58. Length of Last Word (LeetCode Problem)

Code:

```
python

class Solution:
    def lengthOfLastWord(self, s: str) -> int:
        # Strip leading and trailing spaces
        s = s.strip()
        # Find the last word by splitting the string on spaces
        words = s.split()
        # Return the length of the last word
        return len(words[-1])
```

Explanation

The function returns the **length of the last word** in a given string `s`.
A *word* is defined as a sequence of non-space characters.

Step-by-step breakdown

1. **Remove extra spaces**

   ```
   python

   s = s.strip()
   ```

   - Removes any leading and trailing spaces.
   - This ensures there are no empty words at the ends after splitting.

   Example:
   `"   Hello World   "` → `"Hello World"`

---

2. **Split into words**

   ```
   python

   words = s.split()
   ```

   - Splits the string by spaces into a list of words.
   - Multiple consecutive spaces are treated as one separator.

Example:
`"Hello World"` → `["Hello", "World"]`

---

3. **Get the last word**

```
python

words[-1]
```

- Access the last element in the list (last word).

---

4. **Return its length**

```
python

return len(words[-1])
```

- Compute the number of characters in the last word and return it.

Example walkthrough

Input:

```
python

s = "Hello World"
```

Step 1: **`s.strip()` → `"Hello World"`**
Step 2: **`s.split()` → `["Hello", "World"]`**
Step 3: **`words[-1] = "World"`**
Step 4: **`len("World") = 5`**

Output:

```
5
```

Another example

Input:

```python
s = "   fly me   to   the moon  "
```

Step-by-step:

1. `strip()` → `"fly me   to   the moon"`
2. `split()` → `["fly", "me", "to", "the", "moon"]`
3. Last word = `"moon"`
4. Length = `4`

Output:

```
4
```

Summary of logic

1. Remove extra spaces from both ends.
2. Split the string into words.
3. Return the length of the last word in the list.

Time and Space Complexity

- **Time Complexity:** O(n) — traverse string once for strip and split.
- **Space Complexity:** O(n) — temporary list of words is created.

Problem Name:
66. Plus One (LeetCode Problem)

Code:

```python
class Solution:
    def plusOne(self, digits: list[int]) -> list[int]:
        # Start from the last digit and move to the left
        for i in range(len(digits) - 1, -1, -1):
            # If the current digit is less than 9, add one and return
            if digits[i] < 9:
                digits[i] += 1
                return digits
            # If the current digit is 9, set it to 0 and continue the carry
            digits[i] = 0

        # If we have completed the loop, all digits were 9, so we add 1 at the
beginning
        return [1] + digits
```

Explanation

This function takes a list of digits representing a **non-negative integer**,
adds **one** to that number, and returns the resulting list of digits.

Step-by-step breakdown

1. **Start from the end**

   ```python
   for i in range(len(digits) - 1, -1, -1):
   ```

   - Iterates backward through the list (right to left).
   - This is because addition starts from the least significant digit (units place).

---

2. **If the current digit is less than 9**

   ```python
   if digits[i] < 9:
       digits[i] += 1
       return digits
   ```

   - If the digit is not 9, just add 1 and stop — no carry-over needed.

- Return immediately since the operation is complete.

---

3. **If the digit is 9**

   ```
   python

   digits[i] = 0
   ```

   - Set the current digit to 0 because 9 + 1 = 10.
   - Carry over 1 to the next digit on the left (handled in the next loop iteration).

---

4. **If loop finishes**

   ```
   python

   return [1] + digits
   ```

   - If the loop finishes, it means every digit was 9 (like 9, 99, 999).
   - Example: **`999 + 1 = 1000`**
   - Add a new **`1`** at the beginning of the list.

Example walkthrough

Example 1

```
python

digits = [1, 2, 3]
```

Step	i	digits[i]	Action	Resulting digits
1	2	3	3 < 9 → add 1	[1, 2, 4]

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

Example 2

```
python

digits = [4, 3, 9]
```

Step	i	digits[i]	Action	Resulting digits
1	2	9	set to 0	[4, 3, 0]

Step	i	digits[i]	Action	Resulti
2	1	3	3 < 9 → add 1	[4, 4, 0]

Output: **`[4, 4, 0]`**

Example 3

```
python

digits = [9, 9, 9]
```

Step	i	digits[i]	Action	digits
1	2	9	set to 0	[9, 9, 0]
2	1	9	set to 0	[9, 0, 0]
3	0	9	set to 0	[0, 0, 0]

Loop ends → all were 9 → add **`[1]`** at front
Output: **`[1, 0, 0, 0]`**

Summary of logic

1. Traverse digits from end to start.
2. If a digit < 9, increment and return immediately.
3. If it's 9, set to 0 and continue.
4. If all digits are 9, return **`[1] + digits`**.

Time and Space Complexity

- **Time Complexity:** O(n) — one pass through the list.
- **Space Complexity:** O(1) — modifies in place except for an all-9 case, where one new digit is added.

Problem Name:
67. Add Binary (LeetCode Problem)

Code:

```
python

class Solution:
    def addBinary(self, a: str, b: str) -> str:
        # Initialize variables
        result = []
        carry = 0
        i, j = len(a) - 1, len(b) - 1

        # Iterate through both strings from right to left
        while i >= 0 or j >= 0 or carry:
            # Add the bits from a, b and carry
            bit_a = int(a[i]) if i >= 0 else 0
            bit_b = int(b[j]) if j >= 0 else 0

            # Sum the bits and carry
            total = bit_a + bit_b + carry

            # Append the result bit (0 or 1)
            result.append(str(total % 2))

            # Update the carry (1 or 0)
            carry = total // 2

            # Move to the next bits in a and b
            i -= 1
            j -= 1

        # The result array is in reverse order, so reverse it before joining
        return ''.join(result[::-1])
```

Explanation

This function adds two **binary numbers** represented as strings and returns their **sum as a binary string**.

Step-by-step breakdown

1. **Initialize variables**
    - **`result`** → a list to store result bits (in reverse order)
    - **`carry`** → to hold carry-over during addition (0 or 1)
    - **`i`**, **`j`** → pointers starting from the rightmost bit of **`a`** and **`b`**
2. **Loop condition**

```python
while i >= 0 or j >= 0 or carry:
```

Continue until both strings are fully processed and there's no carry left.

3. **Extract bits**

```python
bit_a = int(a[i]) if i >= 0 else 0
bit_b = int(b[j]) if j >= 0 else 0
```

If one string is shorter, treat missing bits as `0`.

4. **Add bits and carry**

```python
total = bit_a + bit_b + carry
```

Binary addition rules:

- 0 + 0 + 0 = 0
- 0 + 1 + 0 = 1
- 1 + 1 + 0 = 2 (binary `10`)
- 1 + 1 + 1 = 3 (binary `11`)

5. **Compute new bit and carry**

```python
result.append(str(total % 2))
carry = total // 2
```

- `% 2` gives the current result bit.
- `// 2` gives the carry (either 0 or 1).

6. **Move left**

```python
i -= 1
j -= 1
```

Go to the next bit positions (toward most significant bit).

7. **Reverse and return**

```python
return ''.join(result[::-1])
```

Since bits were appended from right to left, reverse them to get the correct order.

Example walkthrough

Input:

```python
a = "1010"
b = "1011"
```

Step 1: Initialize

`i = 3`, **`j = 3`**, **`carry = 0`**, **`result = []`**

Iteration 1:

`bit_a = 0`, **`bit_b = 1`**

`total = 0 + 1 + 0 = 1`

`result = ['1']`, **`carry = 0`**

Iteration 2:

`bit_a = 1`, **`bit_b = 1`**

`total = 1 + 1 + 0 = 2`

`result = ['1', '0']`, **`carry = 1`**

Iteration 3:

`bit_a = 0`, **`bit_b = 0`**

`total = 0 + 0 + 1 = 1`

`result = ['1', '0', '1']`, **`carry = 0`**

Iteration 4:

`bit_a = 1`, **`bit_b = 1`**

`total = 1 + 1 + 0 = 2`

`result = ['1', '0', '1', '0']`, **`carry = 1`**

End of loop:

`carry = 1` still remains → append one more bit

`result = ['1', '0', '1', '0', '1']`

Reverse → **`['1', '0', '1', '0', '1']`** → **`"10101"`**

Output:

```arduino
"10101"
```

Summary of logic

1. Start from the end of both strings.
2. Add bits along with the carry.
3. Store result bits and compute new carry.
4. Continue until both numbers and carry are processed.
5. Reverse and join the result to get the final binary string.

Time and Space Complexity

- **Time Complexity:** O(max(len(a), len(b)))
- **Space Complexity:** O(max(len(a), len(b)))

Problem Name:
69. Sqrt(x) (LeetCode Problem)

Code:

```
python

class Solution:
    def mySqrt(self, x: int) -> int:
        # Handle the case where x is 0
        if x == 0:
            return 0

        # Binary search range: 1 to x
        left, right = 1, x

        while left <= right:
            mid = (left + right) // 2

            # If mid squared equals x, return mid
            if mid * mid == x:
                return mid

            # If mid squared is less than x, move right
            elif mid * mid < x:
                left = mid + 1

            # If mid squared is greater than x, move left
            else:
                right = mid - 1

        # When loop ends, right is the integer part of sqrt(x)
        return right
```

Explanation

This function returns the **integer square root** of **x**.
It means the greatest integer **r** such that **r * r ≤ x**.

The algorithm uses **binary search** to efficiently find this integer value.

Step-by-step breakdown

1. **Edge case**

   ```
   python

   if x == 0:
       return 0
   ```

   The square root of 0 is 0.

2. **Set up search boundaries**

   ```
   python

   left, right = 1, x
   ```

   Possible range of the square root is between 1 and x.

---

3. **Binary search loop**

   ```
   python

   while left <= right:
       mid = (left + right) // 2
   ```

   Repeatedly narrow down the search space until the correct integer part is found.

---

4. **Check conditions**

   Case 1:

   ```
   python

   if mid * mid == x:
       return mid
   ```

   If the square of mid equals x, we found the exact square root.

   Case 2:

   ```
   python

   elif mid * mid < x:
       left = mid + 1
   ```

   If `mid^2` is smaller, move the search to the right half.

   Case 3:

   ```
   python

   else:
       right = mid - 1
   ```

   If `mid^2` is larger, move the search to the left half.

---

5. **Return integer part**

```
python

return right
```

When the loop ends, **`right`** will hold the largest integer whose square is ≤ x.

Example walkthrough

Example 1

```
python

x = 8
```

Step 1:

`left = 1`, **`right = 8`**

Iteration 1:

`mid = (1 + 8) // 2 = 4`

`mid * mid = 16 > 8` → move left → **`right = 3`**

Iteration 2:

`mid = (1 + 3) // 2 = 2`

`mid * mid = 4 < 8` → move right → **`left = 3`**

Iteration 3:

`mid = (3 + 3) // 2 = 3`

`mid * mid = 9 > 8` → move left → **`right = 2`**

Loop ends (**`left = 3`**, **`right = 2`**).

Return right = 2

Output:

```

2
```

Example 2

```
python

x = 16
```

Iteration:

`mid = 8 → too high`

`mid = 4 → 4 * 4 == 16 → return 4`

Output:

```
4
```

Summary of logic

1. Use binary search between **`1`** and **`x`**.
2. Compare **`mid * mid`** with **`x`**.
3. Adjust range accordingly.
4. Return **`right`** when loop finishes — the integer part of √x.

Time and Space Complexity

- **Time Complexity:** O(log x) — binary search reduces range by half each step.
- **Space Complexity:** O(1) — only constant extra space used.

Problem Name:
70. Climbing Stairs (LeetCode Problem)

Code:

```
python

class Solution:
    def climbStairs(self, n: int) -> int:
        # Handle the base cases
        if n == 1:
            return 1
        if n == 2:
            return 2

        # Use dynamic programming to find the number of ways
        prev1, prev2 = 2, 1

        for i in range(3, n + 1):
            current = prev1 + prev2  # ways to reach the i-th step
            prev2 = prev1            # Update prev2 to the previous step
            prev1 = current          # Update prev1 to the current step

        return prev1
```

Explanation

This problem models the number of **distinct ways to climb a staircase** with `n` steps.
You can climb either **1 step or 2 steps at a time**.

The solution is based on **Dynamic Programming (DP)** — specifically, the **Fibonacci pattern**.

Step-by-step breakdown

1. **Base cases**

   ```
   python

   if n == 1:
       return 1
   if n == 2:
       return 2
   ```

   - If there's only 1 step → only **1 way** (take one step).
   - If there are 2 steps → **2 ways**:
     (1+1) or (2).

---

2. **DP recurrence relation**

- To reach step **`i`**, you can come from:
  - step **`i-1`** (1-step jump), or
  - step **`i-2`** (2-step jump).

Hence:

```
ways(i) = ways(i-1) + ways(i-2)
```

This is the Fibonacci sequence.

---

3. **Initialize the first two values**

```
python

prev1, prev2 = 2, 1
```

- **`prev1`** = ways to reach step 2
- **`prev2`** = ways to reach step 1

---

4. **Iterate from step 3 to n**

```
python

for i in range(3, n + 1):
    current = prev1 + prev2
    prev2 = prev1
    prev1 = current
```

- Calculate the number of ways to reach the current step.
- Update the previous values for the next iteration.

---

5. **Return the result**

```
python

return prev1
```

- After the loop, **`prev1`** holds the number of ways to reach step **`n`**.

Example walkthrough

Example 1

```
python

n = 3
```

- Step 1: 1 way → `[1]`
- Step 2: 2 ways → `[1+1, 2]`
- Step 3: ways(2) + ways(1) = 2 + 1 = 3 ways → `[1+1+1, 1+2, 2+1]`

Output: `3`

Example 2

```
python

n = 5
```

Step	Formula	Result
1	1	1
2	2	2
3	2 + 1	3
4	3 + 2	5
5	5 + 3	8

Output: `8`

Summary of logic

1. For `n = 1` → 1 way
2. For `n = 2` → 2 ways
3. For `n > 2`, use Fibonacci relation:
   `ways(n) = ways(n-1) + ways(n-2)`

Time and Space Complexity

- **Time Complexity:** O(n) — single loop from 3 to n
- **Space Complexity:** O(1) — only two variables (`prev1`, `prev2`) used

Problem Name:
53. Maximum Subarray (LeetCode Problem)

Code:

```
python

class Solution:
    def maxSubArray(self, nums: list[int]) -> int:
        # Initialize current_sum and max_sum with the first element
        current_sum = max_sum = nums[0]

        # Iterate over the array starting from the second element
        for num in nums[1:]:
            # Either add the current number to the current sum or start a new subarray
            current_sum = max(current_sum + num, num)

            # Update the maximum sum encountered so far
            max_sum = max(max_sum, current_sum)

        return max_sum
```

Explanation

This algorithm finds the **maximum sum of a contiguous subarray** in the given integer list **`nums`**.

It uses **Kadane's Algorithm**, which efficiently solves the problem in **O(n)** time and **O(1)** space.

Step-by-step breakdown

1. **Initialization**

   ```
   python

   current_sum = max_sum = nums[0]
   ```

   - **`current_sum`**: Tracks the sum of the current subarray being considered.
   - **`max_sum`**: Tracks the maximum subarray sum found so far.

   Both start with the first element since a subarray must have at least one number.

---

2. **Iterate through the array**

   ```
   python

   for num in nums[1:]:
   ```

Starts from the second element since the first one is already used for initiali

---

3. **Decision at each step**

   ```
   python

   current_sum = max(current_sum + num, num)
   ```

   - Either:
     - Extend the current subarray by adding **`num`**, or
     - Start a **new subarray** beginning at **`num`**,
       whichever gives a larger sum.

---

4. **Update maximum sum**

   ```
   python

   max_sum = max(max_sum, current_sum)
   ```

   Keep track of the largest sum seen so far.

---

5. **Return the result**

   ```
   python

   return max_sum
   ```

   `max_sum` holds the maximum subarray sum after iterating through all elements.

Example walkthrough

Input:

```
python

nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
```

Step 1:
`current_sum = max_sum = -2`

Step 2: Iterate through the rest of the list

num	current_sum (max of current_sum+num, num)	max_sum
1	max(-2+1, 1) = 1	max(-2, 1) = 1

num	current_sum (max of current_sum+num, num)	
-3	max(1-3, -3) = -2	max
4	max(-2+4, 4) = 4	max(1, 4) = 4
-1	max(4-1, -1) = 3	max(4, 3) = 4
2	max(3+2, 2) = 5	max(4, 5) = 5
1	max(5+1, 1) = 6	max(5, 6) = 6
-5	max(6-5, -5) = 1	max(6, 1) = 6
4	max(1+4, 4) = 5	max(6, 5) = 6

Output: `6`

The subarray **`[4, -1, 2, 1]`** gives the maximum sum of **6**.

Another example

Input:

```python
python

nums = [1]
```

- **`current_sum = max_sum = 1`**
- No more elements to process.

Output: `1`

Summary of logic

1. Start with first element as initial sum.
2. For each element, decide whether to continue or start new subarray.
3. Keep track of the maximum sum found.
4. Return that maximum.

Time and Space Complexity

- **Time Complexity:** O(n) — single pass through array.
- **Space Complexity:** O(1) — only uses two variables (**`current_sum`**, **`max_sum`**).

Problem Name:
121. Best Time to Buy and Sell Stock (LeetCode Problem)

Code:

```python
python


class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        # Initialize min_price to a very large value and max_profit to 0
        min_price = float('inf')
        max_profit = 0

        # Iterate over the prices to find the maximum profit
        for price in prices:
            # Update the min_price if the current price is lower
            min_price = min(min_price, price)

            # Calculate the current profit by selling at the current price
            current_profit = price - min_price

            # Update max_profit if the current profit is greater than the previous max
            max_profit = max(max_profit, current_profit)

        return max_profit
```

Explanation

This function finds the **maximum profit** you can achieve from buying and selling one stock. You may only **buy once** and **sell once**, and you must **buy before you sell**.

Step-by-step breakdown

1. **Initialization**

```python
python


min_price = float('inf')
max_profit = 0
```

- **`min_price`** keeps track of the **lowest stock price** seen so far (best day to buy).
- **`max_profit`** stores the **maximum profit** found so far.

---

2. **Iterate through prices**

```python
python


for price in prices:
```

Go through each day's stock price.

---

3. **Update the minimum price**

   ```
   python

   min_price = min(min_price, price)
   ```

   - If today's price is less than **`min_price`**, update it.
   - This means we found a better day to buy.

---

4. **Calculate current profit**

   ```
   python

   current_profit = price - min_price
   ```

   - Profit if we sell today = today's price − lowest price so far.

---

5. **Update maximum profit**

   ```
   python

   max_profit = max(max_profit, current_profit)
   ```

   - If selling today gives higher profit than before, update **`max_profit`**.

---

6. **Return result**

   ```
   python

   return max_profit
   ```

   After scanning all prices, return the highest profit possible.

Example walkthrough

Input:

```
python

prices = [7, 1, 5, 3, 6, 4]
```

Step-by-step:

Day	Price	min_price	current_profit	
1	7	7	0	0
2	1	1	0	0
3	5	1	4	4
4	3	1	2	4
5	6	1	5	5
6	4	1	3	5

Output:

```
5
```

(Buy at price 1, sell at price 6)

Example 2

```
python

prices = [7, 6, 4, 3, 1]
```

- `min_price` keeps decreasing.
- `current_profit` always ≤ 0.
- `max_profit` remains 0.

Output:

```
0
```

(No profitable transaction possible)

Summary of logic

1. Track the lowest buy price so far.
2. For each day, compute possible profit if sold that day.
3. Keep track of the maximum profit.
4. Return that maximum.

Time and Space Complexity

- **Time Complexity:** O(n) — single pass through prices list.
- **Space Complexity:** O(1) — constant extra space used.

Problem Name:
125. Valid Palindrome (LeetCode Problem)

Code:

```python
class Solution:
    def isPalindrome(self, s: str) -> bool:
        # Remove non-alphanumeric characters and convert to lowercase
        filtered_string = ''.join(c.lower() for c in s if c.isalnum())

        # Check if the filtered string is equal to its reverse
        return filtered_string == filtered_string[::-1]
```

Explanation

This function checks whether a given string `s` is a **palindrome**, considering only **alphanumeric characters** and ignoring cases.

A palindrome is a string that reads the same forward and backward.

Step-by-step breakdown

1. **Filter and normalize the string**

   ```python
   filtered_string = ''.join(c.lower() for c in s if c.isalnum())
   ```

   This line:

   - Iterates through each character `c` in the string `s`
   - Keeps only **alphanumeric** characters (letters and digits)
   - Converts them to **lowercase**
   - Joins them back into a single continuous string

   Let's break it down further:

   - `c.lower()`
     → Converts character `c` to lowercase.
     (Predefined string method: `lower()` returns a lowercase version of a string.)
   - `c.isalnum()`
     → Returns `True` if `c` is a letter or digit; otherwise `False`.

(Predefined string method: **`isalnum()`** checks if all characters in a string alphanumeric.)

- **`''.join(iterable)`**
  → Concatenates all strings in the iterable into a single string with no separator.
  (Predefined method: **`join()`** is used to combine iterable elements into a string.)

Example:
`s = "A man, a plan, a canal: Panama"`

Step-by-step transformation:

- After filtering → **`"amanaplanacanalpanama"`**

---

2. **Check palindrome condition**

```
python

filtered_string == filtered_string[::-1]
```

- **`filtered_string[::-1]`** creates a **reversed copy** of the string.
  (Predefined slicing syntax: **`[::-1]`** means start from end to beginning with a step of -1.)
- The expression compares the original and reversed strings.
  If they are equal → the string is a palindrome.

Example:
`"amanaplanacanalpanama" == "amanaplanacanalpanama"` → **`True`**

Example walkthrough

Input:

```
python

s = "A man, a plan, a canal: Panama"
```

Step 1:
Filter and lowercase → **`"amanaplanacanalpanama"`**

Step 2:
Reverse → **`"amanaplanacanalpanama"`**

Compare:
Equal → return **`True`**

Output:

```graphql
True
```

Another example:

```python
s = "race a car"
```

Filtered → **`"raceacar"`**
Reversed → **`"racacear"`**
Not equal → return **`False`**

Output:

```graphql
False
```

Summary of predefined methods and keywords used

Keyword / Method	Description
`str.lower()`	Converts uppercase characters to lowercase.
`str.isalnum()`	Checks if a character is alphanumeric (A–Z, a–z, 0–9).
`''.join(iterable)`	Joins all strings in an iterable into one string.
`[::-1]`	String slicing technique to reverse a string.
`return`	Exits the function and sends back a value.
`for`	Used for iteration over a sequence.
`if`	Conditional statement.
`==`	Checks equality between two values.

Time and Space Complexity

- **Time Complexity:** O(n) — Each character is checked and processed once.
- **Space Complexity:** O(n) — Filtered string and its reversed copy are stored.

Problem Name:
136. Single Number (LeetCode Problem)

Code:

```
python

class Solution:
    def singleNumber(self, nums: list[int]) -> int:
        result = 0
        for num in nums:
            result ^= num  # XOR the current number with the result
        return result
```

Explanation

This problem asks to find the **element that appears only once** in a list where every other element appears **exactly twice**.

The solution uses the **bitwise XOR operator (^)**.

Step-by-step breakdown

1. **Initialize result**

   ```
   python

   result = 0
   ```

   - This variable will store the ongoing XOR result.

---

2. **Iterate through each number**

   ```
   python

   for num in nums:
       result ^= num
   ```

   - The operator ^ is the **bitwise XOR (exclusive OR)**.
   - XOR has the following key properties:
     - **`a ^ a = 0`** (a number XORed with itself cancels out)
     - **`a ^ 0 = a`** (a number XORed with 0 stays the same)
     - XOR is **commutative and associative**, meaning order doesn't matter.

Using these properties:

- All numbers that appear **twice** will cancel each other out (become 0).
- The **unique number** will remain in **`result`**.

---

3. **Return the final result**

```
python

return result
```

- After processing all numbers, **`result`** holds the single number that doesn't repeat.

Example walkthrough

Input:

```
python

nums = [4, 1, 2, 1, 2]
```

Step-by-step XOR process:

Step	Current num	Previous result	Operation	New result
1	4	0	0 ^ 4	4
2	1	4	4 ^ 1	5
3	2	5	5 ^ 2	7
4	1	7	7 ^ 1	6
5	2	6	6 ^ 2	4

Output:

```

4
```

Explanation:
All pairs (1, 1) and (2, 2) cancel each other out → only **`4`** remains.

Another Example

Input:

```python
nums = [2, 2, 3]
```

Process:

```ini
result = 0 ^ 2 = 2
result = 2 ^ 2 = 0
result = 0 ^ 3 = 3
```

Output:

```
3
```

Predefined operators used

Operator / Keyword	Description
`^`	Bitwise XOR operator (exclusive OR).
`for`	Iteration over a list.
`return`	Returns the computed value from the function.

Time and Space Complexity

- **Time Complexity:** O(n) — Iterates once through the list.
- **Space Complexity:** O(1) — Uses only a single variable `result`.

Problem Name:
350. Intersection of Two Arrays II (LeetCode Problem)

Code:

```
python

from collections import Counter

class Solution:
    def intersect(self, nums1: list[int], nums2: list[int]) -> list[int]:
        # Count the frequency of each number in nums1
        count1 = Counter(nums1)
        result = []

        # Traverse through nums2 and check if the element exists in count1
        for num in nums2:
            if count1[num] > 0:
                result.append(num)      # Add the element to the result
                count1[num] -= 1        # Decrease the count to avoid duplicates

        return result
```

Explanation

This function finds the **intersection** of two integer arrays (**`nums1`** and **`nums2`**),
where each element in the result must appear **as many times as it shows in both arrays**.

Step-by-step breakdown

1. **Import `Counter`**

   ```
   python

   from collections import Counter
   ```

   - **`Counter`** is a predefined class in Python's **collections module**.
   - It counts how many times each element appears in an iterable.
   - Example:

     ```
     python

     Counter([1,2,2,3]) → {1:1, 2:2, 3:1}
     ```

---

2. **Count elements in the first list**

```python
count1 = Counter(nums1)
```

- Stores the frequency of each number from **`nums1`**.
- Example:
  If **`nums1 = [4,9,5,9]`**, then
  `count1 = {4:1, 9:2, 5:1}`

---

3. **Initialize result list**

```python
result = []
```

- Used to collect the elements that are common in both arrays.

---

4. **Iterate through the second list**

```python
for num in nums2:
    if count1[num] > 0:
        result.append(num)
        count1[num] -= 1
```

- For each **`num`** in **`nums2`**, check if it exists in **`count1`** with a positive count.
- If yes:
  - Add it to **`result`**
  - Decrease its count in **`count1`** (to prevent adding it more times than it appears in **`nums1`**)

---

5. **Return the result**

```python
return result
```

- The final list will contain all elements that appear in both lists (including duplicates).

Example walkthrough

Input:

```python
nums1 = [4, 9, 5]
nums2 = [9, 4, 9, 8, 4]
```

Step 1:

`count1 = Counter(nums1)` → `{4:1, 9:1, 5:1}`

Step 2: Iterate through `nums2`:

num	count1 before	Action	count1 after	result
9	1	Append 9	9 → 0	[9]
4	1	Append 4	4 → 0	[9, 4]
9	0	Skip	same	[9, 4]
8	0	Skip	same	[9, 4]
4	0	Skip	same	[9, 4]

Output:

```csharp
[9, 4]
```

Predefined module and methods used

Keyword / Method	Description
`collections.Counter`	A dictionary subclass that counts element frequencies.
`Counter(nums1)`	Creates a frequency map of all elements in `nums1`.
`count1[num]`	Returns the count (frequency) of the element `num` in `nums1`.
`list.append()`	Adds an element to the end of a list.
`for` / `if`	Standard iteration and condition syntax.

Time and Space Complexity

- **Time Complexity:** O(n + m)
  Counting `nums1` takes O(n), iterating `nums2` takes O(m).
- **Space Complexity:** O(n)
  For storing the `Counter` dictionary of `nums1`.

Example 2

Input:

```python
nums1 = [1,2,2,1]
nums2 = [2,2]
```

Process:

```wasm
count1 = {1:2, 2:2}
Iterate nums2:
→ 2 found → result = [2], count1[2] = 1
→ 2 found → result = [2, 2], count1[2] = 0
```

Output:

```csharp
[2, 2]
```