Math

1. Palindrome Number

Pattern: Math / Two Pointers (Digit Extraction)

Problem Statement

Given an integer `x`, return `true` if `x` is a **palindrome**, and `false` otherwise.
An integer is a palindrome when it reads the same backward as forward.
Follow-up: Could you solve it without converting the integer to a string?

Sample Input & Output

```
Input: 121
Output: true
Explanation: 121 reads the same forward and backward.
```

```
Input: -121
Output: false
Explanation: Negative numbers are not palindromes due to the '-' sign.
```

```
Input: 10
Output: false
Explanation: 10 reversed is "01" → 1, which is not equal to 10.
```

LeetCode Editorial Solution + Inline Tests

```python
class Solution:
    def isPalindrome(self, x: int) -> bool:
        # STEP 1: Initialize structures
        #   - Reject negative numbers and numbers ending with 0
        #     (except 0 itself) since they can't be palindromes.
        if x < 0 or (x % 10 == 0 and x != 0):
            return False

        # STEP 2: Main loop / recursion
        #   - Reverse only half the number by building reversed_half
        #   - Stop when original half <= reversed_half
        reversed_half = 0
        while x > reversed_half:
            reversed_half = reversed_half * 10 + x % 10
            x //= 10

        # STEP 3: Update state / bookkeeping
        #   - For even-digit numbers: x == reversed_half
        #   - For odd-digit numbers: x == reversed_half // 10
        #   - This handles the middle digit being irrelevant

        # STEP 4: Return result
        #   - Check both even and odd length cases
        return x == reversed_half or x == reversed_half // 10

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    assert sol.isPalindrome(121) == True

    #   Test 2: Edge case
    assert sol.isPalindrome(0) == True

    #   Test 3: Tricky/negative
    assert sol.isPalindrome(-121) == False
    assert sol.isPalindrome(10) == False
    assert sol.isPalindrome(1221) == True
    assert sol.isPalindrome(12321) == True
```

```python
print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `isPalindrome(1221)` step by step:

1. **Initial check**:
   - `x = 1221` → not negative, and `1221 % 10 = 1   0`, so continue.
2. **Initialize**:
   - `reversed_half = 0`
3. **First loop iteration** (`x = 1221`, `reversed_half = 0`):
   - Condition: `1221 > 0` → enter loop
   - `reversed_half = 0 * 10 + 1221 % 10 = 0 + 1 = 1`
   - `x = 1221 // 10 = 122`
   - State: `x=122`, `reversed_half=1`
4. **Second loop iteration** (`x = 122`, `reversed_half = 1`):
   - Condition: `122 > 1` → enter loop
   - `reversed_half = 1 * 10 + 122 % 10 = 10 + 2 = 12`
   - `x = 122 // 10 = 12`
   - State: `x=12`, `reversed_half=12`
5. **Loop condition check**:
   - `x = 12`, `reversed_half = 12` → `12 > 12` is **false** → exit loop
6. **Final check**:
   - `x == reversed_half` → `12 == 12` → `True`

- Return `True`

Output: `True` — correctly identified as palindrome.

Now try `12321` (odd digits):

- After loop: `x = 12`, `reversed_half = 123`
- Check: `12 == 123 // 10` → `12 == 12` → `True`

This shows how the middle digit (`3`) is safely ignored.

Complexity Analysis

- **Time Complexity**: `O(log (n))`

  We divide `x` by 10 in each iteration. Number of digits = `log (x) + 1`, so we loop ~half that → still `O(log n)`.

- **Space Complexity**: `O(1)`

  Only a few integer variables (`reversed_half`, `x`) are used — constant extra space.

2. Reverse Integer

Pattern: Math & Digit Manipulation

Problem Statement

> Given a signed 32-bit integer `x`, return `x` with its digits reversed. If reversing `x` causes the value to go outside the signed 32-bit integer range $[-2^{31}, 2^{31} - 1]$, then return `0`.
>
> **Assume the environment does not allow you to store 64-bit integers (signed or unsigned).**

Sample Input & Output

```
Input: x = 123
Output: 321
Explanation: Digits reversed normally.
```

```
Input: x = -123
Output: -321
Explanation: Sign preserved; digits reversed.
```

```
Input: x = 1534236469
Output: 0
Explanation: Reversed value (9646324351) exceeds 2³¹-1 → return 0.
```

LeetCode Editorial Solution + Inline Tests

```python
class Solution:
    def reverse(self, x: int) -> int:
        # STEP 1: Initialize structures
        #   - Use 'rev' to build reversed number digit by digit.
        #   - Track sign separately to handle negative cleanly.
        sign = -1 if x < 0 else 1
        x_abs = abs(x)
        rev = 0

        # STEP 2: Main loop / recursion
        #   - Extract last digit via % 10, add to rev.
        #   - Remove last digit via // 10.
        #   - Invariant: rev holds reversed digits of processed part.
        while x_abs != 0:
            digit = x_abs % 10

            # STEP 3: Update state / bookkeeping
            #   - Check overflow BEFORE updating rev.
            #   - Use bounds: 2**31 = 2147483648
            #   - Max allowed rev before *10 + digit:
            #       pos: (2**31 - 1) // 10 = 214748364
```

```python
            #       neg: 2**31 // 10 = 214748364
            if rev > 214748364 or (rev == 214748364 and digit > 7):
                return 0
            if rev < -214748364 or (rev == -214748364 and digit > 8):
                return 0

            rev = rev * 10 + digit
            x_abs //= 10

        # STEP 4: Return result
        #   - Apply original sign.
        #   - Edge: x=0 handled naturally (loop skipped).
        return sign * rev

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.reverse(123) == 321

    #  Test 2: Edge case (negative, ends with zero)
    assert sol.reverse(-123) == -321
    assert sol.reverse(120) == 21

    #  Test 3: Tricky/negative (overflow)
    assert sol.reverse(1534236469) == 0
    assert sol.reverse(-2147483648) == 0  # -2**31 reversed overflows

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `reverse(123)` step by step:

1. **Initial setup**

    - `x = 123`
    - `sign = 1` (since 123 0)
    - `x_abs = 123`
    - `rev = 0`

2. **First loop iteration** (`x_abs = 123`)
    - `digit = 123 % 10 = 3`
    - Check overflow: `rev = 0` → safe
    - `rev = 0 * 10 + 3 = 3`
    - `x_abs = 123 // 10 = 12`
3. **Second loop iteration** (`x_abs = 12`)
    - `digit = 12 % 10 = 2`
    - `rev = 3` → safe
    - `rev = 3 * 10 + 2 = 32`
    - `x_abs = 12 // 10 = 1`
4. **Third loop iteration** (`x_abs = 1`)
    - `digit = 1 % 10 = 1`
    - `rev = 32` → safe
    - `rev = 32 * 10 + 1 = 321`
    - `x_abs = 1 // 10 = 0`
5. **Loop ends** (`x_abs = 0`)
    - Return `sign * rev = 1 * 321 = 321`

Final output: **321**

Key insight: We never store a number larger than 32-bit because we check *before* multiplying by 10 and adding the new digit. This respects the problem's constraint of no 64-bit storage.

Complexity Analysis

- **Time Complexity**: `O(log x)`

  We process each digit once. Number of digits in `x` is `log |x| + 1`, so time is logarithmic in input magnitude.

- **Space Complexity**: `O(1)`

  Only a few integer variables (`sign`, `x_abs`, `rev`, `digit`) are used — constant extra space.

3. Roman to Integer

Pattern: Arrays & Hashing

Problem Statement

Roman numerals are represented by seven different symbols: `I`, `V`, `X`, `L`, `C`, `D`, and `M`.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Roman numerals are usually written largest to smallest from left to right. However, there are six instances where subtraction is used:
- `I` can be placed before `V` (5) and `X` (10) to make 4 and 9.
- `X` can be placed before `L` (50) and `C` (100) to make 40 and 90.
- `C` can be placed before `D` (500) and `M` (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

Sample Input & Output

```
Input: "III"
Output: 3
Explanation: III = 1 + 1 + 1 = 3
```

```
Input: "IV"
Output: 4
Explanation: IV = 5 - 1 = 4 (subtraction case)
```

```
Input: "MCMXCIV"
Output: 1994
Explanation: M=1000, CM=900, XC=90, IV=4 → 1000+900+90+4 = 1994
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Dict

class Solution:
    def romanToInt(self, s: str) -> int:
        # STEP 1: Initialize structures
        #   - Use a hash map for O(1) symbol-to-value lookup
        roman_map: Dict[str, int] = {
            'I': 1, 'V': 5, 'X': 10, 'L': 50,
            'C': 100, 'D': 500, 'M': 1000
        }

        total = 0
        n = len(s)

        # STEP 2: Main loop / recursion
        #   - Traverse left to right
        #   - If current < next → subtract current
        #   - Else → add current
        for i in range(n):
            current_val = roman_map[s[i]]
```

```
            # Check if not last char and current < next
            if i < n - 1 and current_val < roman_map[s[i + 1]]:
                total -= current_val
            else:
                total += current_val

        # STEP 4: Return result
        #   - All cases handled in loop; no special edge return needed
        return total

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    assert sol.romanToInt("III") == 3

    #   Test 2: Edge case (single character)
    assert sol.romanToInt("V") == 5

    #   Test 3: Tricky/negative (multiple subtractive pairs)
    assert sol.romanToInt("MCMXCIV") == 1994

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `romanToInt("IV")` step by step:

1. **Initialize `roman_map`**
   - A dictionary mapping each Roman symbol to its integer value is created.
2. **Set `total = 0`, `n = 2`**
   - Input string `"IV"` has length 2.
3. **Start loop at `i = 0`**

- `current_val = roman_map['I'] = 1`
- Check: `i < 1` (yes) and `1 < roman_map['V'] = 5` → **true**
- So: `total -= 1` → `total = -1`

4. **Next iteration: `i = 1`**
    - `current_val = roman_map['V'] = 5`
    - Now `i = 1` is last index → skip subtraction check
    - So: `total += 5` → `total = -1 + 5 = 4`
5. **Return `total = 4`**

Final output: `4`
Key insight: By processing left-to-right and *subtracting* when a smaller numeral precedes a larger one, we naturally handle all six subtraction cases without special logic.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We iterate through the string exactly once. Each hash map lookup is `O(1)`. So total time is linear in input length.

- **Space Complexity**: `O(1)`

  The hash map uses fixed space (7 entries). No additional space scales with input size.

4. Pow(x, n)

Pattern: Divide and Conquer (Recursion with Exponentiation by Squaring)

Problem Statement

Implement pow(x, n), which calculates `x` raised to the power `n` (i.e., `x^n`).

- `-100.0 < x < 100.0`
- `-2^31 <= n <= 2^31 - 1`
- `n` is an integer.
- Either `x` is not zero or `n > 0`.
- `-10^4 <= x^n <= 10^4`

Sample Input & Output

```
Input: x = 2.00000, n = 10
Output: 1024.00000
Explanation: 2^10 = 1024
```

```
Input: x = 2.10000, n = 3
Output: 9.26100
Explanation: 2.1^3 = 2.1 * 2.1 * 2.1   9.261
```

```
Input: x = 2.00000, n = -2
Output: 0.25000
Explanation: 2^(-2) = 1 / (2^2) = 1/4 = 0.25
```

LeetCode Editorial Solution + Inline Tests

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        # STEP 1: Handle base cases
        #   - Any number to power 0 is 1
```

```python
        if n == 0:
            return 1.0

        # STEP 2: Handle negative exponent
        #   - Convert to positive and invert base later
        if n < 0:
            return 1.0 / self.myPow(x, -n)

        # STEP 3: Divide and conquer via exponentiation by squaring
        #   - If n is even: x^n = (x^(n/2))^2
        #   - If n is odd: x^n = x * (x^(n//2))^2
        half = self.myPow(x, n // 2)
        if n % 2 == 0:
            return half * half
        else:
            return x * half * half

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    result1 = sol.myPow(2.0, 10)
    print(f"Test 1: {result1:.5f}")  # Expected: 1024.00000

    #  Test 2: Edge case - negative exponent
    result2 = sol.myPow(2.0, -2)
    print(f"Test 2: {result2:.5f}")  # Expected: 0.25000

    #  Test 3: Tricky case - fractional base, positive exponent
    result3 = sol.myPow(2.1, 3)
    print(f"Test 3: {result3:.5f}")  # Expected: ~9.26100
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `myPow(2.0, 10)` step by step.

Initial Call: `myPow(2.0, 10)`
- `n = 10` (positive, not zero) → skip base cases
- Compute `half = myPow(2.0, 5)`

Call 1: `myPow(2.0, 5)`
- `n = 5` → odd
- Compute `half = myPow(2.0, 2)`

Call 2: `myPow(2.0, 2)`
- `n = 2` → even
- Compute `half = myPow(2.0, 1)`

Call 3: `myPow(2.0, 1)`
- `n = 1` → odd
- Compute `half = myPow(2.0, 0)`

Call 4: `myPow(2.0, 0)`
- `n == 0` → return `1.0`

Unwind Call 3:
- `half = 1.0`
- `n = 1` is odd → return `2.0 * 1.0 * 1.0 = 2.0`

Unwind Call 2:
- `half = 2.0`
- `n = 2` is even → return `2.0 * 2.0 = 4.0`

Unwind Call 1:
- `half = 4.0`
- `n = 5` is odd → return `2.0 * 4.0 * 4.0 = 32.0`

Unwind Initial Call:
- `half = 32.0`
- `n = 10` is even → return `32.0 * 32.0 = 1024.0`

Final output: `1024.0`

This recursive halving reduces the problem size by half each time—classic **Divide and Conquer**.

Complexity Analysis

- **Time Complexity**: `O(log n)`

  Each recursive call halves `n`, so depth is `log |n|`. Only one recursive call per level.

- **Space Complexity**: `O(log n)`

  Due to recursion stack depth of `O(log n)`. No additional data structures scale with input.

5. Random Pick with Weight

Pattern: Prefix Sum + Binary Search

Problem Statement

You are given a 0-indexed array of positive integers `w` where `w[i]` describes the weight of index `i`.

Implement the `Solution` class:
- `Solution(int[] w)` Initializes the object with the array `w`.
- `int pickIndex()` Picks an index in the range `[0, w.length - 1]` and returns it. The probability of picking index `i` is `w[i] / sum(w)`.

For example, if `w = [1, 3]`, then the probability of picking index `0` is `1 / (1 + 3) = 25%`, and the probability of picking index `1` is `3 / (1 + 3) = 75%`.

Sample Input & Output

```
Input: ["Solution", "pickIndex", "pickIndex", "pickIndex"]
       [[[1]], [], [], []]
Output: [null, 0, 0, 0]
Explanation: Only one index exists, so it's always picked.
```

```
Input: ["Solution", "pickIndex", "pickIndex", "pickIndex", "pickIndex",
        "pickIndex"]
       [[[1, 3]], [], [], [], [], []]
Output: [null, 1, 1, 1, 0, 1]  # (example; actual output varies randomly)
Explanation: Index 1 should appear ~75% of the time.
```

```
Input: ["Solution", "pickIndex"]
       [[[1000]], []]
Output: [null, 0]
Explanation: Edge case - single large weight.
```

LeetCode Editorial Solution + Inline Tests

```python
import random
from typing import List
import bisect

class Solution:
    def __init__(self, w: List[int]):
        # STEP 1: Build prefix sum array
        #   - Why? To map random number to weighted index.
        #   - Example: w = [1,3] → prefix = [1,4]
        self.prefix = []
        total = 0
        for weight in w:
            total += weight
            self.prefix.append(total)
        # Now self.prefix[-1] == sum(w)

    def pickIndex(self) -> int:
        # STEP 2: Generate random target in [1, total_weight]
        #   - Why 1-based? Because prefix sums start at w[0]   1.
        target = random.randint(1, self.prefix[-1])

        # STEP 3: Use binary search to find first prefix   target
        #   - This maps target to correct weighted index.
        #   - bisect_left returns smallest i s.t. prefix[i] >= target
```

```python
        index = bisect.bisect_left(self.prefix, target)

        # STEP 4: Return index (guaranteed valid due to target range)
        return index

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    #   Test 1: Normal case - [1, 3]
    sol1 = Solution([1, 3])
    counts = [0, 0]
    for _ in range(1000):
        idx = sol1.pickIndex()
        counts[idx] += 1
    # Expect ~25% for index 0, ~75% for index 1
    print("Test 1 (w=[1,3]) - Counts:", counts)
    (assert 200 <= counts[0] <= 300,
     f"Index 0 count out of expected range: {counts[0]}")
    (assert 700 <= counts[1] <= 800,
     f"Index 1 count out of expected range: {counts[1]}")

    #   Test 2: Edge case - single element
    sol2 = Solution([5])
    result = sol2.pickIndex()
    print("Test 2 (w=[5]) - Result:", result)
    assert result == 0, f"Expected 0, got {result}"

    #   Test 3: Tricky case - large weights, ensure no off-by-one
    sol3 = Solution([1, 1, 1, 1, 96])  # last index should dominate
    counts = [0] * 5
    for _ in range(1000):
        idx = sol3.pickIndex()
        counts[idx] += 1
    print("Test 3 (w=[1,1,1,1,96]) - Counts:", counts)
    assert counts[4] >= 900, f"Index 4 should dominate; got {counts[4]}"
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's walk through `w = [1, 3]` step by step:

1. **Initialization (`__init__`)**
   - `w = [1, 3]`
   - `total = 0`
   - Loop:
     - Add `1` → `total = 1` → `prefix = [1]`
     - Add `3` → `total = 4` → `prefix = [1, 4]`
   - Final `self.prefix = [1, 4]`
2. **First `pickIndex()` call**
   - `self.prefix[-1] = 4`
   - `random.randint(1, 4)` → suppose it returns `3`
   - `bisect.bisect_left([1, 4], 3)`
     - Compare `3` with `1` → too big
     - Compare `3` with `4` → `4 >= 3` → return index `1`
   - Output: `1`
3. **Second call**
   - Suppose `random.randint(1, 4)` returns `1`
   - `bisect_left([1,4], 1)` → first element `1 >= 1` → index `0`
   - Output: `0`
4. **Why it works**
   - Numbers `1` map to index `0`
   - Numbers `2,3,4` map to index `1`
   - So index `0`: 1/4 chance, index `1`: 3/4 chance → matches weights!

Complexity Analysis

- **Time Complexity**:
  - `__init__`: `O(n)` — one pass to build prefix sum.
  - `pickIndex`: `O(log n)` — binary search over prefix array.

    Building prefix is one-time cost. Each pick is fast via binary search.
- **Space Complexity**: `O(n)`

    We store the prefix sum array of length `n`. No extra space per call.