Solutions to DSA Questions 171-200 (Matrix, Math, Geometry) For 1-2 Years Experience Roles at EPAM

Compiled on September 27, 2025

Introduction

This document provides detailed solutions for 30 Data Structures and Algorithms (DSA) problems (questions 171 to 200) from the Matrix, Math, and Geometry categories, tailored for candidates with 1-2 years of experience preparing for roles at EPAM Systems. Each problem includes a problem statement, dry run with test cases, algorithm, and a Python solution, formatted for clarity. Solutions are optimized for readability and efficiency, suitable for interview preparation.

Contents

1 Rotate Image

1.1 Problem Statement

Given an n x n matrix, rotate it 90 degrees clockwise in-place.

1.2 Dry Run on Test Cases

- **Test Case 1**: matrix = [[1,2,3],[4,5,6],[7,8,9]] → Output: [[7,4,1],[8,5,2],[9,6,3]]
- **Test Case 2**: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]] → Output: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]
- **Test Case 3**: matrix = [[1]] → Output: [[1]]
- **Test Case 4**: matrix = [] → Output: []

1.3 Algorithm

1. Transpose matrix (swap elements across diagonal).
2. Reverse each row.

Time Complexity: $O(n^2)$ **Space Complexity**: $O(1)$

1.4 Python Solution

```
def rotate(matrix):
    n = len(matrix)
    # Transpose
    for i in range(n):
        for j in range(i, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][
                j]
    # Reverse each row
    for i in range(n):
        matrix[i].reverse()
```

2 Set Matrix Zeroes

2.1 Problem Statement

Given an m x n matrix, if an element is 0, set its entire row and column to 0 in-place.

2.2 Dry Run on Test Cases

- **Test Case 1**: matrix = [[1,1,1],[1,0,1],[1,1,1]] → Output: [[1,0,1],[0,0,0],[1,0,1]]
- **Test Case 2**: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]] → Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

- **Test Case 3**: matrix = [[1]] → Output: [[1]]
- **Test Case 4**: matrix = [] → Output: []

2.3 Algorithm

1. Use first row and column as markers.
2. Mark rows/columns to be zeroed.
3. Set zeros using markers, handle first row/column separately.

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(1)$

2.4 Python Solution

```python
def set_zeroes(matrix):
    if not matrix or not matrix[0]:
        return
    rows, cols = len(matrix), len(matrix[0])
    first_row_zero = any(matrix[0][j] == 0 for j in range(cols))
    first_col_zero = any(matrix[i][0] == 0 for i in range(rows))

    # Mark zeros in first row/column
    for i in range(1, rows):
        for j in range(1, cols):
            if matrix[i][j] == 0:
                matrix[i][0] = matrix[0][j] = 0

    # Set zeros based on markers
    for i in range(1, rows):
        if matrix[i][0] == 0:
            for j in range(1, cols):
                matrix[i][j] = 0
    for j in range(1, cols):
        if matrix[0][j] == 0:
            for i in range(1, rows):
                matrix[i][j] = 0

    # Handle first row
    if first_row_zero:
        for j in range(cols):
            matrix[0][j] = 0

    # Handle first column
    if first_col_zero:
        for i in range(rows):
            matrix[i][0] = 0
```

3 Search a 2D Matrix

3.1 Problem Statement

Given a sorted m x n matrix (rows and first column sorted), search for a target.

3.2 Dry Run on Test Cases

- **Test Case 1**: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3 → Output: True
- **Test Case 2**: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13 → Output: False
- **Test Case 3**: matrix = [[1]], target = 1 → Output: True
- **Test Case 4**: matrix = [], target = 1 → Output: False

3.3 Algorithm

1. Start from top-right corner.
2. If target matches, return True.
3. If target smaller, move left; if larger, move down.

Time Complexity: $O(m+n)$ **Space Complexity**: $O(1)$

3.4 Python Solution

```python
def search_matrix(matrix, target):
    if not matrix or not matrix[0]:
        return False
    rows, cols = len(matrix), len(matrix[0])
    i, j = 0, cols - 1
    while i < rows and j >= 0:
        if matrix[i][j] == target:
            return True
        elif matrix[i][j] > target:
            j -= 1
        else:
            i += 1
    return False
```

4 Search a 2D Matrix II

4.1 Problem Statement

Given a sorted m x n matrix (rows and columns sorted), search for a target.

4.2 Dry Run on Test Cases

- **Test Case 1**: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]], target = 5 → Output: True
- **Test Case 2**: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]], target = 20 → Output: False
- **Test Case 3**: matrix = [[1]], target = 1 → Output: True
- **Test Case 4**: matrix = [], target = 1 → Output: False

4.3 Algorithm

1. Start from top-right corner.
2. If target matches, return True.
3. If target smaller, move left; if larger, move down.

Time Complexity: $O(m+n)$ **Space Complexity**: $O(1)$

4.4 Python Solution

```python
def search_matrix_ii(matrix, target):
    if not matrix or not matrix[0]:
        return False
    rows, cols = len(matrix), len(matrix[0])
    i, j = 0, cols - 1
    while i < rows and j >= 0:
        if matrix[i][j] == target:
            return True
        elif matrix[i][j] > target:
            j -= 1
        else:
            i += 1
    return False
```

5 Valid Sudoku

5.1 Problem Statement

Given a 9x9 Sudoku board, determine if it is valid (rows, columns, 3x3 sub-boxes).

5.2 Dry Run on Test Cases

– **Test Case 1**: board = [["5","3",".",".","7",".",".",".",".",".],["6",".",".","1","9","5",".",".",".",".],[".","9","8
→ Output: True

– **Test Case 2**: board = [["8","3",".",".","7",".",".",".",".","."],["6",".",".","1","9","5",".",".",".","."],[".","9","8
→ Output: False

– **Test Case 3**: board = [[".",".","."],[".",".","."],[".",".","."]] → Output: True

– **Test Case 4**: board = [] → Output: True

5.3 Algorithm

1. Use sets to track numbers in each row, column, and 3x3 box.
2. Check each cell; if number exists in respective set, return False.
3. Add valid numbers to sets.

Time Complexity: $O(1)$ (fixed 9x9) **Space Complexity**: $O(1)$

5.4 Python Solution

```python
def is_valid_sudoku(board):
    if not board or not board[0]:
        return True
    rows = [set() for _ in range(9)]
    cols = [set() for _ in range(9)]
    boxes = [set() for _ in range(9)]

    for i in range(9):
        for j in range(9):
            if board[i][j] == '.':
                continue
            num = board[i][j]
            if num in rows[i] or num in cols[j] or num in
                boxes[(i // 3) * 3 + j // 3]:
                return False
            rows[i].add(num)
            cols[j].add(num)
            boxes[(i // 3) * 3 + j // 3].add(num)
    return True
```

6 Factorial Trailing Zeroes

6.1 Problem Statement

Given an integer n, return the number of trailing zeros in n!.

6.2 Dry Run on Test Cases

* **Test Case 1**: n = 3 → Output: 0
* **Test Case 2**: n = 5 → Output: 1

* **Test Case 3**: n = 0 → Output: 0
* **Test Case 4**: n = 10 → Output: 2

6.3 Algorithm

1. Count factors of 5 in n! (since 2s are abundant).
2. Sum n//5, n//25, n//125, etc., until $n//5^k is 0$. **Time Complexity**: $O(\log n)$ **Space Complexity**: $O(1)$

6.4 Python Solution

```
def trailing_zeroes(n):
    count = 0
    while n > 0:
        n //= 5
        count += n
    return count
```

7 Power of N

7.1 Problem Statement

Given x and n, compute x^n.

7.2 Dry Run on Test Cases

- **Test Case 1**: x = 2.00000, n = 10 → Output: 1024.00000
- **Test Case 2**: x = 2.10000, n = 3 → Output: 9.26100
- **Test Case 3**: x = 2.00000, n = -2 → Output: 0.25000
- **Test Case 4**: x = 1.00000, n = 0 → Output: 1.00000

7.3 Algorithm

1. Use binary exponentiation.
2. If n is negative, compute $1/x^{|n|}.Squarebase, halveexponent, multiplyresultifexpone$ **Time Complexity**: $O(\log n)$ **Space Complexity**: $O(1)$

7.4 Python Solution

```
def my_pow(x, n):
    if n == 0:
        return 1.0
    if n < 0:
        x = 1 / x
```

```
        n = -n
    result = 1.0
    while n > 0:
        if n % 2 == 1:
            result *= x
        x *= x
        n //= 2
    return result
```

8 Sqrt(x)

8.1 Problem Statement

Given a non-negative integer x, return the square root of x (integer part).

8.2 Dry Run on Test Cases

3. **Test Case 1**: x = 4 → Output: 2

- **Test Case 2**: x = 8 → Output: 2
- **Test Case 3**: x = 0 → Output: 0
- **Test Case 4**: x = 1 → Output: 1

8.3 Algorithm

1. Use binary search to find largest integer whose square <= x.
2. Search range [0, x].
3. Adjust range based on mid * mid vs x.

Time Complexity: $O(\log x)$ **Space Complexity**: $O(1)$

8.4 Python Solution

```
def my_sqrt(x):
    if x == 0:
        return 0
    left, right = 1, x
    while left <= right:
        mid = (left + right) // 2
        if mid * mid == x:
            return mid
        elif mid * mid < x:
            left = mid + 1
        else:
            right = mid - 1
```

```
    return right
```

9 Divide Two Integers

9.1 Problem Statement

Given two integers dividend and divisor, return quotient without using * or /.

9.2 Dry Run on Test Cases

- **Test Case 1**: dividend = 10, divisor = 3 → Output: 3
- **Test Case 2**: dividend = 7, divisor = -3 → Output: -2
- **Test Case 3**: dividend = 0, divisor = 1 → Output: 0
- **Test Case 4**: dividend = 1, divisor = 1 → Output: 1

9.3 Algorithm

1. Handle signs and convert to positive numbers.
2. Use bit manipulation to subtract doubled divisors.
3. Handle 32-bit integer overflow.

Time Complexity: $O(\log n)$ **Space Complexity**: $O(1)$

9.4 Python Solution

```python
def divide(dividend, divisor):
    MAX_INT = 2**31 - 1
    MIN_INT = -2**31
    sign = -1 if (dividend < 0) ^ (divisor < 0)
       else 1
    dividend, divisor = abs(dividend), abs(divisor)

    quotient = 0
    while dividend >= divisor:
        temp, count = divisor, 1
        while dividend >= (temp << 1):
            temp <<= 1
            count <<= 1
        dividend -= temp
        quotient += count

    result = sign * quotient
    return min(max(result, MIN_INT), MAX_INT)
```

10 Fraction to Recurring Decimal

10.1 Problem Statement

Given numerator and denominator, return fraction as a string with recurring decimal.

10.2 Dry Run on Test Cases

- **Test Case 1**: numerator = 1, denominator = 2 → Output: "0.5"
- **Test Case 2**: numerator = 2, denominator = 1 → Output: "2"
- **Test Case 3**: numerator = 4, denominator = 333 → Output: "0.(012)"
- **Test Case 4**: numerator = -1, denominator = -2147483648 → Output: "0.0000000004656612873077392578125"

10.3 Algorithm

1. Handle sign and compute integer part.
2. For decimal part, use hashmap to detect repeating digits.
3. Add parentheses for repeating sequence.

Time Complexity: $O(\log n)$ **Space Complexity**: $O(\log n)$

10.4 Python Solution

```python
def fraction_to_decimal(numerator, denominator):
    if numerator == 0:
        return "0"
    result = []
    sign = -1 if (numerator < 0) ^ (denominator < 0) else 1
    numerator, denominator = abs(numerator), abs(denominator)

    # Integer part
    integer = numerator // denominator
    result.append(str(integer))

    # Decimal part
    remainder = numerator % denominator
    if remainder == 0:
        return ("-" if sign == -1 else "") + result[0]

    result.append(".")
    seen = {}
```

```
    while remainder:
        if remainder in seen:
            result.insert(seen[remainder], "(")
            result.append(")")
            break
        seen[remainder] = len(result)
        remainder *= 10
        result.append(str(remainder // denominator)
            )
        remainder %= denominator

    return ("-" if sign == -1 else "") + "".join(
        result)
```

11 Happy Number

11.1 Problem Statement

Given a number, determine if it is happy (sum of squares of digits leads to 1).

11.2 Dry Run on Test Cases

- **Test Case 1**: n = 19 → Output: True ($1^2 + 9^2 = 82, 8^2 + 2^2 = 68, 6^2 + 8^2 = 100, 1^2 + 0^2 + 0^2 = 1$)**Test Case 2** : $n = 2$→ Output: False (cycles)
- **Test Case 3**: n = 1 → Output: True
- **Test Case 4**: n = 7 → Output: True

11.3 Algorithm

1. Use set to detect cycles.
2. Compute sum of squares of digits.
3. Return True if sum is 1, False if cycle detected.

Time Complexity: $O(\log n)$ **Space Complexity**: $O(\log n)$

11.4 Python Solution

```
def is_happy(n):
    seen = set()
    while n != 1:
        if n in seen:
            return False
        seen.add(n)
        n = sum(int(d) ** 2 for d in str(n))
```

```
    return True
```

12 Plus One

12.1 Problem Statement

Given a non-negative integer as an array of digits, add one to it.

12.2 Dry Run on Test Cases

- **Test Case 1**: digits = [1,2,3] → Output: [1,2,4]
- **Test Case 2**: digits = [4,3,2,1] → Output: [4,3,2,2]
- **Test Case 3**: digits = [9] → Output: [1,0]
- **Test Case 4**: digits = [9,9] → Output: [1,0,0]

12.3 Algorithm

1. Iterate digits from right to left.
2. Add 1, handle carry.
3. If carry remains, prepend 1.

Time Complexity: $O(n)$ **Space Complexity**: $O(1)$

12.4 Python Solution

```
def plus_one(digits):
    n = len(digits)
    for i in range(n - 1, -1, -1):
        if digits[i] < 9:
            digits[i] += 1
            return digits
        digits[i] = 0
    return [1] + [0] * n
```

13 Climbing Stairs (Math Approach)

13.1 Problem Statement

Given n stairs, find number of ways to climb (1 or 2 steps) using a mathematical approach.

13.2 Dry Run on Test Cases

- **Test Case 1**: n = 2 → Output: 2 ([1,1], [2])
- **Test Case 2**: n = 3 → Output: 3 ([1,1,1], [1,2], [2,1])
- **Test Case 3**: n = 1 → Output: 1
- **Test Case 4**: n = 0 → Output: 1

13.3 Algorithm

1. Recognize as Fibonacci sequence: ways(n) = ways(n-1) + ways(n-2).
2. Use iterative Fibonacci to avoid recursion.

Time Complexity: $O(n)$ **Space Complexity**: $O(1)$

13.4 Python Solution

```python
def climb_stairs(n):
    if n <= 1:
        return 1
    a, b = 1, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

14 Pow(x, n) (Alternative)

14.1 Problem Statement

Given x and n, compute $x^n using an alternative approach.$

14.2 Dry Run on Test Cases

- **Test Case 1**: x = 2.00000, n = 10 → Output: 1024.00000
- **Test Case 2**: x = 2.10000, n = 3 → Output: 9.26100
- **Test Case 3**: x = 2.00000, n = -2 → Output: 0.25000
- **Test Case 4**: x = 1.00000, n = 0 → Output: 1.00000

14.3 Algorithm

1. Use recursive binary exponentiation.
2. If n is odd, multiply by x; if even, square base.

3. Handle negative n by computing $1/x^{|n|}$. **Time Complexity**: $O(\log n)$ **Space Complexity**: $O(\log n)$

14.4 Python Solution

```python
def my_pow(x, n):
    def pow_positive(x, n):
        if n == 0:
            return 1.0
        half = pow_positive(x, n // 2)
        if n % 2 == 0:
            return half * half
        return half * half * x

    if n < 0:
        x = 1 / x
        n = -n
    return pow_positive(x, n)
```

15 Max Points on a Line

15.1 Problem Statement

Given a list of points, find the maximum number of points on a single line.

15.2 Dry Run on Test Cases

- **Test Case 1**: points = [[1,1],[2,2],[3,3]] → Output: 3
- **Test Case 2**: points = [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]] → Output: 4
- **Test Case 3**: points = [[1,1]] → Output: 1
- **Test Case 4**: points = [] → Output: 0

15.3 Algorithm

1. For each point, compute slopes with other points.
2. Use hashmap to count points with same slope.
3. Handle duplicates and vertical lines.

Time Complexity: $O(n^2)$ **Space Complexity**: $O(n)$

15.4 Python Solution

```python
from collections import defaultdict
from math import gcd

```

```python
def max_points(points):
    if not points:
        return 0
    max_points = 1
    for i in range(len(points)):
        slopes = defaultdict(int)
        duplicates = 0
        for j in range(i + 1, len(points)):
            if points[i] == points[j]:
                duplicates += 1
                continue
            dx = points[j][0] - points[i][0]
            dy = points[j][1] - points[i][1]
            if dx == 0:
                slope = 'inf'
            else:
                g = gcd(dx, dy)
                slope = (dy // g, dx // g)
            slopes[slope] += 1
        max_points = max(max_points, max(slopes.
           values(), default=0) + duplicates + 1)
    return max_points
```

16 Valid Number

16.1 Problem Statement

Given a string, determine if it is a valid number (integer, decimal, scientific).

16.2 Dry Run on Test Cases

- **Test Case 1**: s = "0" → Output: True
- **Test Case 2**: s = "e" → Output: False
- **Test Case 3**: s = "2e10" → Output: True
- **Test Case 4**: s = "abc" → Output: False

16.3 Algorithm

1. Split string on 'e' or 'E' for scientific notation.
2. Check if base and exponent (if present) are valid.
3. Base: optional sign, digits, optional decimal with digits.

Time Complexity: $O(n)$ **Space Complexity**: $O(1)$

16.4 Python Solution

```python
def is_number(s):
    def is_decimal(s):
        if not s:
            return False
        if s[0] in '+-':
            s = s[1:]
        if not s or s == '.':
            return False
        dot_seen = False
        for i, c in enumerate(s):
            if c == '.':
                if dot_seen:
                    return False
                dot_seen = True
            elif not c.isdigit():
                return False
        return True

    s = s.strip()
    if not s:
        return False
    parts = s.split('e') if 'e' in s else s.split('E')
    if len(parts) > 2:
        return False
    if len(parts) == 2:
        return is_decimal(parts[0]) and parts[1] and is_decimal(parts[1].replace('+', '-', 1))
    return is_decimal(parts[0])
```

17 Reverse Integer

17.1 Problem Statement

Given a 32-bit signed integer, reverse its digits.

17.2 Dry Run on Test Cases

- **Test Case 1**: x = 123 → Output: 321
- **Test Case 2**: x = -123 → Output: -321
- **Test Case 3**: x = 120 → Output: 21
- **Test Case 4**: x = 0 → Output: 0

17.3 Algorithm

1. Handle sign and convert to positive.
2. Reverse digits using modulo and division.
3. Check for 32-bit integer overflow.

Time Complexity: $O(\log x)$ **Space Complexity**: $O(1)$

17.4 Python Solution

```python
def reverse(x):
    MAX_INT = 2**31 - 1
    MIN_INT = -2**31
    sign = -1 if x < 0 else 1
    x = abs(x)
    result = 0
    while x:
        digit = x % 10
        if result > MAX_INT // 10 or (result ==
           MAX_INT // 10 and digit > MAX_INT % 10):
            return 0
        result = result * 10 + digit
        x //= 10
    return sign * result if MIN_INT <= sign *
       result <= MAX_INT else 0
```

18 Palindrome Number

18.1 Problem Statement

Given an integer, determine if it is a palindrome.

18.2 Dry Run on Test Cases

- **Test Case 1**: x = 121 → Output: True
- **Test Case 2**: x = -121 → Output: False
- **Test Case 3**: x = 10 → Output: False
- **Test Case 4**: x = 0 → Output: True

18.3 Algorithm

1. If negative, return False.
2. Reverse number and compare with original.

Time Complexity: $O(\log x)$ **Space Complexity**: $O(1)$

18.4 Python Solution

```python
def is_palindrome(x):
    if x < 0:
        return False
    original = x
    reversed_num = 0
    while x:
        reversed_num = reversed_num * 10 + x % 10
        x //= 10
    return original == reversed_num
```

19 Roman to Integer

19.1 Problem Statement

Given a Roman numeral string, convert it to an integer.

19.2 Dry Run on Test Cases

- **Test Case 1**: s = "III" → Output: 3
- **Test Case 2**: s = "IV" → Output: 4
- **Test Case 3**: s = "MCMXCIV" → Output: 1994
- **Test Case 4**: s = "LVIII" → Output: 58

19.3 Algorithm

1. Map Roman symbols to values.
2. Iterate string; if current value $<$ next, subtract, else add.

Time Complexity: $O(n)$ **Space Complexity**: $O(1)$

19.4 Python Solution

```python
def roman_to_int(s):
    roman = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C':
        100, 'D': 500, 'M': 1000}
    result = 0
    for i in range(len(s)):
        if i < len(s) - 1 and roman[s[i]] < roman[s
            [i + 1]]:
            result -= roman[s[i]]
        else:
            result += roman[s[i]]
    return result
```

20 Integer to Roman

20.1 Problem Statement

Given an integer, convert it to a Roman numeral string.

20.2 Dry Run on Test Cases

- **Test Case 1**: num = 3 → Output: "III"
- **Test Case 2**: num = 4 → Output: "IV"
- **Test Case 3**: num = 1994 → Output: "MCMXCIV"
- **Test Case 4**: num = 58 → Output: "LVIII"

20.3 Algorithm

1. Define Roman numeral values and symbols in descending order.
2. Greedily select largest possible value, append symbol, subtract.

Time Complexity: $O(1)$ (fixed range 1-3999) **Space Complexity**: $O(1)$

20.4 Python Solution

```python
def int_to_roman(num):
    values = [1000, 900, 500, 400, 100, 90, 50, 40,
        10, 9, 5, 4, 1]
    symbols = ["M", "CM", "D", "CD", "C", "XC", "L"
       , "XL", "X", "IX", "V", "IV", "I"]
    result = ""
    for v, s in zip(values, symbols):
        while num >= v:
            result += s
            num -= v
    return result
```

21 Count Primes

21.1 Problem Statement

Given an integer n, return the number of prime numbers less than n.

21.2 Dry Run on Test Cases

- **Test Case 1**: n = 10 → Output: 4 (2,3,5,7)
- **Test Case 2**: n = 0 → Output: 0

- **Test Case 3**: n = 1 → Output: 0
- **Test Case 4**: n = 2 → Output: 0

21.3 Algorithm

1. Use Sieve of Eratosthenes.
2. Mark multiples of each prime as non-prime.
3. Count remaining primes.

Time Complexity: $O(n \log \log n)$ **Space Complexity**: $O(n)$

21.4 Python Solution

```python
def count_primes(n):
    if n < 2:
        return 0
    is_prime = [True] * n
    is_prime[0] = is_prime[1] = False
    for i in range(2, int(n ** 0.5) + 1):
        if is_prime[i]:
            for j in range(i * i, n, i):
                is_prime[j] = False
    return sum(is_prime)
```

22 Ugly Number

22.1 Problem Statement

Given an integer, determine if it is an ugly number (prime factors only 2, 3, 5).

22.2 Dry Run on Test Cases

- **Test Case 1**: n = 6 → Output: True (2*3)
- **Test Case 2**: n = 8 → Output: True (2^3)**Test Case 3** : $n = 14$→ Output: False (2*7)
- **Test Case 4**: n = 1 → Output: True

22.3 Algorithm

1. Repeatedly divide n by 2, 3, 5 as long as possible.
2. If final n == 1, it is an ugly number.

Time Complexity: $O(\log n)$ **Space Complexity**: $O(1)$

22.4 Python Solution

```python
def is_ugly(n):
    if n <= 0:
        return False
    for prime in [2, 3, 5]:
        while n % prime == 0:
            n //= prime
    return n == 1
```

23 Ugly Number II

23.1 Problem Statement

Given an integer n, return the nth ugly number.

23.2 Dry Run on Test Cases

- **Test Case 1**: n = 10 → Output: 12 (1,2,3,4,5,6,8,9,10,12)
- **Test Case 2**: n = 1 → Output: 1
- **Test Case 3**: n = 7 → Output: 8
- **Test Case 4**: n = 4 → Output: 4

23.3 Algorithm

1. Use min-heap to generate ugly numbers.
2. Track seen numbers to avoid duplicates.
3. Pop n times to get nth ugly number.

Time Complexity: $O(n \log n)$ **Space Complexity**: $O(n)$

23.4 Python Solution

```python
import heapq

def nth_ugly_number(n):
    if n == 1:
        return 1
    heap = [1]
    seen = {1}
    primes = [2, 3, 5]
    for _ in range(n - 1):
        curr = heapq.heappop(heap)
        for prime in primes:
            next_num = curr * prime
            if next_num not in seen:
```

```
                    seen.add(next_num)
                    heapq.heappush(heap, next_num)
    return heap[0]
```

24 Perfect Squares

24.1 Problem Statement

Given an integer n, return the least number of perfect squares that sum to n.

24.2 Dry Run on Test Cases

- **Test Case 1**: n = 12 → Output: 3 (4+4+4)
- **Test Case 2**: n = 13 → Output: 2 (9+4)
- **Test Case 3**: n = 1 → Output: 1
- **Test Case 4**: n = 4 → Output: 1

24.3 Algorithm

1. Use dynamic programming: dp[i] = min squares to sum to i.
2. For each i, try all perfect squares <= i, take minimum.

Time Complexity: $O(n \cdot \sqrt{n})$ **Space Complexity**: $O(n)$

24.4 Python Solution

```
def num_squares(n):
    dp = [float('inf')] * (n + 1)
    dp[0] = 0
    for i in range(1, n + 1):
        j = 1
        while j * j <= i:
            dp[i] = min(dp[i], dp[i - j * j] + 1)
            j += 1
    return dp[n]
```

25 Nth Digit

25.1 Problem Statement

Find the nth digit in the infinite sequence 1, 2, 3, ..., 10, 11, ...

25.2 Dry Run on Test Cases

- **Test Case 1**: n = 3 → Output: 3
- **Test Case 2**: n = 11 → Output: 0 (10)
- **Test Case 3**: n = 1 → Output: 1
- **Test Case 4**: n = 15 → Output: 1 (12)

25.3 Algorithm

1. Find range of numbers (1-digit, 2-digit, etc.) containing nth digit.
2. Compute which number and which digit within that number.

Time Complexity: $O(\log n)$ **Space Complexity**: $O(1)$

25.4 Python Solution

```python
def find_nth_digit(n):
    length = 0
    count = 9
    start = 1
    while n > length * count:
        n -= length * count
        length += 1
        count *= 10
        start *= 10
    start += (n - 1) // length
    return int(str(start)[(n - 1) % length])
```

26 Valid Perfect Square

26.1 Problem Statement

Given a positive integer, determine if it is a perfect square.

26.2 Dry Run on Test Cases

- **Test Case 1**: num = 16 → Output: True
- **Test Case 2**: num = 14 → Output: False
- **Test Case 3**: num = 1 → Output: True
- **Test Case 4**: num = 25 → Output: True

26.3 Algorithm

1. Use binary search to find if num = i * i.

2. Search range [1, num].

Time Complexity: $O(\log n)$ **Space Complexity**: $O(1)$

26.4 Python Solution

```python
def is_perfect_square(num):
    if num < 0:
        return False
    left, right = 0, num
    while left <= right:
        mid = (left + right) // 2
        square = mid * mid
        if square == num:
            return True
        elif square < num:
            left = mid + 1
        else:
            right = mid - 1
    return False
```

27 Arranging Coins

27.1 Problem Statement

Given n coins, find number of complete rows in a staircase (1, 2, 3, ... coins per row).

27.2 Dry Run on Test Cases

- **Test Case 1**: n = 5 → Output: 2 (1+2=3, 5-3=2)
- **Test Case 2**: n = 8 → Output: 3 (1+2+3=6, 8-6=2)
- **Test Case 3**: n = 0 → Output: 0
- **Test Case 4**: n = 1 → Output: 1

27.3 Algorithm

1. Use quadratic formula to solve k*(k+1)/2 <= n.
2. Return floor of solution.

Time Complexity: $O(1)$ **Space Complexity**: $O(1)$

27.4 Python Solution

```python
def arrange_coins(n):
    return int((-1 + (1 + 8 * n) ** 0.5) // 2)
```

28 Sum of Square Numbers

28.1 Problem Statement

Given a non-negative integer c, determine if it can be expressed as sum of two squares.

28.2 Dry Run on Test Cases

- **Test Case 1**: c = 5 → Output: True $(1^2 + 2^2)$**Test Case 2** : $c = 3$→ Output: False
- **Test Case 3**: c = 4 → Output: True $(2^2 + 0^2)$**Test Case 4** : $c = 0$→ Output: True

28.3 Algorithm

- 1. Use two pointers (i, j) where $i^2 + j^2 = c.Adjustpointersbasedonsumvsc.$ **Time Complexity**: $O(\sqrt{c})$ **Space Complexity**: $O(1)$

28.4 Python Solution

```python
def judge_square_sum(c):
    left, right = 0, int(c ** 0.5)
    while left <= right:
        curr_sum = left * left + right * right
        if curr_sum == c:
            return True
        elif curr_sum < c:
            left += 1
        else:
            right -= 1
    return False
```

29 Max Area of Island

29.1 Problem Statement

Given a binary grid, find the maximum area of an island (connected 1s).

29.2 Dry Run on Test Cases

2. **Test Case 1**: grid = [[0,0,1,0,0],[0,0,0,0,0],[0,1,1,0,1],[0,0,0,0,0]] → Output: 3
- **Test Case 2**: grid = [[0,0,0],[0,0,0]] → Output: 0
- **Test Case 3**: grid = [[1]] → Output: 1
- **Test Case 4**: grid = [] → Output: 0

29.3 Algorithm

1. Use DFS to explore each island.
2. Mark visited cells, count area.
3. Track maximum area across all islands.

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

29.4 Python Solution

```python
def max_area_of_island(grid):
    if not grid or not grid[0]:
        return 0
    rows, cols = len(grid), len(grid[0])

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols
            or grid[i][j] != 1:
            return 0
        grid[i][j] = 0
        return 1 + dfs(i+1, j) + dfs(i-1, j) + dfs(
           i, j+1) + dfs(i, j-1)

    max_area = 0
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1:
                max_area = max(max_area, dfs(i, j))
    return max_area
```

30 Number of Islands

30.1 Problem Statement

Given a binary grid, count the number of islands (connected 1s).

30.2 Dry Run on Test Cases

- **Test Case 1**: grid = [["1","1","0","0","0"],["1","1","0","0","0"],["0","0","1","0","0"],["0","0","0","1","1"]] → Output: 3
- **Test Case 2**: grid = [["1","1","1"],["1","1","1"],["1","1","1"]] → Output: 1
- **Test Case 3**: grid = [["0"]] → Output: 0
- **Test Case 4**: grid = [] → Output: 0

30.3 Algorithm

1. Use DFS to mark all cells in an island as visited.
2. Count each new island encountered.

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

30.4 Python Solution

```python
def num_islands(grid):
    if not grid or not grid[0]:
        return 0
    rows, cols = len(grid), len(grid[0])

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols
            or grid[i][j] != "1":
            return
        grid[i][j] = "0"
        dfs(i+1, j)
        dfs(i-1, j)
        dfs(i, j+1)
        dfs(i, j-1)

    count = 0
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == "1":
                dfs(i, j)
                count += 1
    return count
```