1. Write a Python program that takes an integer input and determines whether the number

Logic:
is even or odd. Explain why the modulo operator works for this check and discuss
behavior for zero and negative numbers.

```
def is_even_or_odd(n: int) -> str:
    return "Even" if n % 2 == 0 else "Odd"

n = int(input("Enter an integer: "))
print(is_even_or_odd(n))
```

2. Demonstrate how to swap two variables in Python without using a temporary variable.

Logic:
Explain why this works in Python.

```
a, b = 5, 10
a, b = b, a
print(a, b)
```

3. Write a function to check whether a given integer is a prime number.

Logic:
Handle edge cases and justify the time complexity.

```
def is_prime(n: int) -> bool:
    if n <= 1:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True

print(is_prime(29))
```

4. Implement a recursive function to compute factorial of a non-negative integer.

Logic:
Explain base case and recursive step.

```
def factorial(n: int) -> int:
    if n < 0:
        raise ValueError("Negative input not allowed")
    if n == 0:
        return 1
    return n * factorial(n - 1)
print(factorial(5))
```

5. Generate the first N terms of the Fibonacci sequence using an iterative approach.

Logic:
Explain why iteration is preferred over recursion here.

```
def fibonacci(n: int):
    a, b = 0, 1
    seq = []
    for _ in range(n):
        seq.append(a)
        a, b = b, a + b
    return seq

print(fibonacci(10))
```

6. Write a program to check whether a given year is a leap year.

Logic:
Explain the Gregorian calendar rules involved.

```
def is_leap_year(year: int) -> bool:
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)

print(is_leap_year(2024))
```

7. Print all Armstrong numbers between 100 and 999.

Logic:
Explain what defines an Armstrong number.

```
for num in range(100, 1000):
    if num == sum(int(d) ** 3 for d in str(num)):
        print(num)
```

8. Reverse an integer without losing the sign of the number.

Logic:

```
def reverse_int(n: int) -> int:
    sign = -1 if n < 0 else 1
    return sign * int(str(abs(n))[::-1])

print(reverse_int(-12345))
```

9. Compute the sum of digits of a given integer.

Logic:

```
def sum_of_digits(n: int) -> int:
    return sum(int(d) for d in str(abs(n)))

print(sum_of_digits(123))
```

10. Compute the greatest common divisor (GCD) of two integers using Euclid’s algorithm.

Logic:

```
def gcd(a: int, b: int) -> int:
    while b:
        a, b = b, a % b
    return abs(a)

print(gcd(48, 18))
```

11. Compute the sum of numbers from 1 to N using a single Python expression.

Logic:

```
N = 100
print(sum(range(1, N + 1)))
```

12. Demonstrate how to create an infinite loop and explain safe termination.

Logic:

```
while True:
    pass  # Stop using KeyboardInterrupt (Ctrl+C)
```

13. Explain the difference between 'break' and 'continue' statements using examples.

Logic:

```
# break exits the loop entirely
# continue skips to the next iteration
```

14. Convert temperature from Celsius to Fahrenheit using a formula.

Logic:

```
def c_to_f(c):
    return (c * 9 / 5) + 32

print(c_to_f(25))
```

15. Read multiple integers from a single line of input.

Logic:

```
x, y = map(int, input("Enter two numbers: ").split())
print(x, y)
```

16. Explain the purpose of the 'pass' statement in Python.

Logic:

```
# pass is a null statement used as a placeholder
```

17. Determine the data type of a variable dynamically.

Logic:

```
x = 3.14
print(type(x))
```

18. Find the largest among three numbers without using conditional statements.

Logic:

```
print(max(10, 25, 15))
```

19. Generate a random integer between 1 and 100.

Logic:

```
import random
print(random.randint(1, 100))
```

20. Demonstrate the use of a ternary conditional expression in Python.

Logic:

```
result = "Pass" if 75 >= 40 else "Fail"
print(result)
```

21. Reverse a string using Python slicing.

Logic:

```
s = "Python"
print(s[::-1])
```

22. Check whether a given string is a palindrome.

Logic:

```
def is_palindrome(s: str) -> bool:
    return s == s[::-1]

print(is_palindrome("madam"))
```

23. Count the frequency of each character in a string.

Logic:

```
from collections import Counter
print(Counter("hello"))
```

24. Check whether two strings are anagrams of each other.

Logic:

```
def are_anagrams(s1, s2):
    return sorted(s1) == sorted(s2)

print(are_anagrams("listen", "silent"))
```

25. Remove duplicate characters from a string while preserving order.

Logic:

```
def remove_duplicates(s: str) -> str:
    seen = set()
    result = []
    for c in s:
```

```
        if c not in seen:
            seen.add(c)
            result.append(c)
    return ''.join(result)

print(remove_duplicates("hello"))
```

26. Find the first non-repeating character in a given string.

Logic:
If no such character exists, return None.
Discuss the time and space complexity of your approach.

```
def first_non_repeating(s: str):
    from collections import Counter
    freq = Counter(s)
    for c in s:
        if freq[c] == 1:
            return c
    return None

print(first_non_repeating("swiss"))
```

27. Split a sentence into words and return them as a list.

Logic:
Explain how Python handles whitespace by default.

```
sentence = "Python   is  great"
print(sentence.split())
```

28. Join a list of words into a single string separated by spaces.

Logic:
Explain why join is preferred over string concatenation in loops.

```
words = ["Python", "is", "great"]
print(" ".join(words))
```

29. Check whether a given string represents a valid non-negative integer.

Logic:
Explain the limitations of str.isdigit().

```
def is_numeric(s: str) -> bool:
    return s.isdigit()

print(is_numeric("123"))
print(is_numeric("-123"))
```

30. Replace all occurrences of a character in a string with another character.

Logic:
Explain why strings are immutable in Python.

```
s = "hello"
print(s.replace("l", "z"))
```

31. Count the number of vowels and consonants in a string.

Logic:
Ignore digits, spaces, and special characters.

```
def count_vowels_consonants(s: str):
    vowels = set("aeiouAEIOU")
    v = c = 0
    for ch in s:
        if ch.isalpha():
            if ch in vowels:
                v += 1
            else:
                c += 1
    return v, c

print(count_vowels_consonants("Hello World"))
```

32. Find the longest word in a given sentence.

Logic:
If multiple words have the same maximum length, return the first one.

```
def longest_word(sentence: str) -> str:
    words = sentence.split()
    return max(words, key=len)

print(longest_word("Python programming language"))
```

33. Toggle the case of each character in a string.

Logic:
Explain how swapcase works internally.

```
s = "PyThOn"
print(s.swapcase())
```

34. Check whether a string starts with a given prefix and ends with a given suffix.

Logic:

```
s = "HelloWorld"
print(s.startswith("Hello"), s.endswith("World"))
```

35. Generate all possible substrings of a given string.

Logic:
Discuss the time complexity.

```
def all_substrings(s: str):
    return [s[i:j] for i in range(len(s)) for j in range(i + 1, len(s) + 1)]

print(all_substrings("abc"))
```

36. Remove leading and trailing whitespace from a string.

Logic:
Explain the difference between strip, lstrip, and rstrip.

```
s = "  hello  "
print(s.strip())
```

37. Convert a sentence to title case.

Logic:
Explain a real-world limitation of str.title().

```
s = "hello world from python"
print(s.title())
```

38. Check whether a string is a valid Python identifier.

Logic:
Explain naming rules briefly.

```
print("var_1".isidentifier())
print("1var".isidentifier())
```

39. Find the maximum and minimum elements in a list.

Logic:
Explain the time complexity.

```
nums = [4, 1, 9, 2]
print(max(nums), min(nums))
```

40. Find the second largest unique number in a list.

Logic:
Handle edge cases where it may not exist.

```
def second_largest(nums):
    unique = sorted(set(nums))
    if len(unique) < 2:
        raise ValueError("Second largest element does not exist")
    return unique[-2]

print(second_largest([10, 20, 4, 45, 99]))
```

41. Remove duplicate elements from a list while preserving order.

Logic:
Explain why set alone is insufficient.

```
def remove_duplicates(lst):
    seen = set()
    result = []
    for x in lst:
        if x not in seen:
            seen.add(x)
            result.append(x)
    return result

print(remove_duplicates([1, 2, 2, 3, 1]))
```

42. Sort a list without using the built-in sort() or sorted() functions.

Logic:
Explain the algorithm used.

```
def bubble_sort(lst):
    n = len(lst)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lst[j] > lst[j + 1]:
                lst[j], lst[j + 1] = lst[j + 1], lst[j]
    return lst

print(bubble_sort([3, 1, 2]))
```

43. Reverse a list in-place.

Logic:
Explain how in-place operations save memory.

```
lst = [1, 2, 3]
lst.reverse()
print(lst)
```

44. Merge two sorted lists into a single sorted list.

Logic:
Discuss time complexity.

```
def merge_sorted(l1, l2):
    i = j = 0
    res = []
    while i < len(l1) and j < len(l2):
        if l1[i] <= l2[j]:
            res.append(l1[i]); i += 1
        else:
            res.append(l2[j]); j += 1
    return res + l1[i:] + l2[j:]

print(merge_sorted([1, 3, 5], [2, 4, 6]))
```

45. Find common elements between two lists.

Logic:
Explain how sets improve performance.

```
a = [1, 2, 3]
b = [2, 3, 4]
print(list(set(a) & set(b)))
```

46. Generate a list of squares of numbers from 0 to N-1 using list comprehension.

Logic:

```
N = 5
print([x ** 2 for x in range(N)])
```

47. Flatten a nested list of depth 1.

Logic:
Explain limitations of this approach for deeper nesting.

```
nested = [[1, 2], [3, 4]]
flat = [x for sub in nested for x in sub]
print(flat)
```

48. Shuffle elements of a list randomly.

Logic:
Explain why this operation mutates the list.

```
import random
lst = [1, 2, 3, 4]
random.shuffle(lst)
print(lst)
```

49. Compute the sum of elements in a list.

Logic:
Explain why sum is preferred over manual loops.

```
nums = [1, 2, 3, 4]
print(sum(nums))
```

50. Check whether a list is empty.

Logic:
Explain Pythonic ways to do this.

```
lst = []
if not lst:
    print("List is empty")
```

51. Compute the sum of all elements in a list of integers.

Logic:
Explain why using the built-in sum() function is preferred over manual loops.

```
nums = [1, 2, 3, 4]
print(sum(nums))
```

52. Check whether a given list is empty.

Logic:
Explain the Pythonic way of doing this.

```
lst = []
if not lst:
    print("List is empty")
```

53. Find the index of a given element in a list.

Logic:
Explain what happens if the element is not present.

```
nums = [10, 20, 30]
print(nums.index(20))  # Raises ValueError if element not found
```

54. Count the frequency of a given element in a list.

Logic:
Discuss the time complexity.

```
nums = [1, 2, 2, 3, 2]
print(nums.count(2))
```

55. Split a list into chunks of size k.

Logic:
Handle cases where the list size is not a multiple of k.

```
def chunk_list(lst, k):
    return [lst[i:i+k] for i in range(0, len(lst), k)]

print(chunk_list([1, 2, 3, 4, 5], 2))
```

56. Explain the difference between list.append() and list.extend().

Logic:
Demonstrate with examples.

```
lst = [1]
lst.append([2, 3])
print(lst)

lst = [1]
lst.extend([2, 3])
print(lst)
```

57. Remove the N-th element from a list using index-based deletion.

Logic:
Explain what happens if the index is invalid.

```
lst = [10, 20, 30]
lst.pop(1)
print(lst)
```

58. Rotate a list to the right by k positions.

Logic:
Handle cases where k is greater than the list length.

```
def rotate_list(lst, k):
    if not lst:
        return lst
    k %= len(lst)
    return lst[-k:] + lst[:-k]

print(rotate_list([1, 2, 3, 4, 5], 2))
```

59. Check whether two lists are equal.

Logic:
Explain the difference between equality (==) and identity (is).

```
a = [1, 2]
b = [1, 2]
print(a == b)
print(a is b)
```

60. Generate all prime numbers up to N using the Sieve of Eratosthenes.

Logic:
Explain the algorithm briefly.

```
def sieve(n):
    if n < 2:
        return []
    primes = [True] * (n + 1)
    primes[0] = primes[1] = False
    for p in range(2, int(n ** 0.5) + 1):
        if primes[p]:
            for i in range(p * p, n + 1, p):
                primes[i] = False
    return [i for i in range(2, n + 1) if primes[i]]

print(sieve(20))
```

61. Sort a dictionary by its values in ascending order.

Logic:
Explain the output type.

```
d = {"a": 2, "b": 1, "c": 3}
sorted_dict = dict(sorted(d.items(), key=lambda x: x[1]))
print(sorted_dict)
```

62. Merge two dictionaries into a new dictionary.

Logic:
Ensure compatibility with Python versions prior to 3.9.

```
d1 = {"a": 1}
d2 = {"b": 2}
merged = {**d1, **d2}
print(merged)
```

63. Iterate over keys and values of a dictionary.

Logic:
Explain why items() is preferred.

```
d = {"a": 1, "b": 2}
for key, value in d.items():
    print(key, value)
```

64. Check whether a given key exists in a dictionary.

Logic:
Explain the time complexity.

```
d = {"a": 1, "b": 2}
print("a" in d)
```

65. Remove a key-value pair from a dictionary safely.

Logic:
Explain how to avoid KeyError.

```
d = {"a": 1}
d.pop("a", None)
print(d)
```

66. Demonstrate the use of collections.defaultdict.

Logic:
Explain its advantage over normal dictionaries.

```
from collections import defaultdict

d = defaultdict(int)
d["count"] += 1
print(d)
```

67. Create a dictionary from two lists: one of keys and one of values.

Logic:
Explain what happens if lengths differ.

```
keys = ["a", "b"]
values = [1, 2]
print(dict(zip(keys, values)))
```

68. Find the intersection of two sets.

Logic:
Explain how sets improve performance.

```
s1 = {1, 2, 3}
s2 = {2, 3, 4}
print(s1 & s2)
```

69. Remove duplicate elements from a list using a set.

Logic:
Explain why order is not preserved.

```
nums = [1, 1, 2, 3]
print(list(set(nums)))
```

70. Create a word-frequency dictionary from a sentence.

Logic:
Ignore case and punctuation.

```
import string

def word_frequency(sentence):
    freq = {}
    for word in sentence.lower().split():
        word = word.strip(string.punctuation)
        freq[word] = freq.get(word, 0) + 1
    return freq

print(word_frequency("Hello world hello"))
```

71. Explain the difference between a set and a dictionary in Python.

Logic:
Focus on structure, usage, and constraints.

```
# Set: collection of unique, unordered hashable elements
# Dict: collection of key-value pairs with unique, hashable keys
```

72. Retrieve all keys from a dictionary.

Logic:
Explain the returned object type.

```
d = {"a": 1, "b": 2}
print(list(d.keys()))
```

73. Create a dictionary using dictionary comprehension.

Logic:
Provide an example.

```
squares = {x: x**2 for x in range(5)}
print(squares)
```

74. Remove all elements from a set.

Logic:
Explain the difference between clear() and reassigning.

```
s = {1, 2, 3}
s.clear()
print(s)
```

75. Find the difference between two sets.

Logic:
Explain a real-world use case.

```
a = {1, 2, 3}
b = {2}
print(a - b)
```

76. Define a class in Python with one method.

Logic:
Explain how objects are created from a class.

```
class Dog:
    def bark(self):
        return "Woof"

d = Dog()
print(d.bark())
```

77. Demonstrate single inheritance in Python.

Logic:
Explain how method resolution works in this case.

```
class Animal:
    def speak(self):
        return "Sound"

class Dog(Animal):
    pass

d = Dog()
print(d.speak())
```

78. Explain the purpose of the __init__ method.

Logic:
Demonstrate initialization of instance variables.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p = Person("Alice", 30)
print(p.name, p.age)
```

79. Explain the role of 'self' in Python classes.

Logic:
Why is it required?

```
# 'self' refers to the current object instance
# It allows access to instance variables and methods
```

80. Differentiate between class variables and instance variables

Logic:
with an example.

```
class Counter:
    count = 0  # class variable

    def __init__(self):
        Counter.count += 1
        self.id = Counter.count  # instance variable

a = Counter()
b = Counter()
print(a.id, b.id, Counter.count)
```

81. Explain how Python handles private variables.

Logic:
Demonstrate name mangling with an example.

```
class A:
    def __init__(self):
        self.__x = 10  # name-mangled to _A__x

a = A()
print(a._A__x)
```

82. Write and explain a simple decorator.

Logic:
Explain how decorators modify function behavior.

```
def my_decorator(func):
    def wrapper():
        print("Before function")
        func()
        print("After function")
    return wrapper

@my_decorator
def say_hello():
    print("Hello")

say_hello()
```

83. Explain generators in Python.

Logic:
Write a generator function and show how it is consumed.

```
def simple_generator():
    for i in range(3):
        yield i

for value in simple_generator():
    print(value)
```

84. Explain lambda functions.

Logic:
Provide an example and a limitation.

```
add = lambda x, y: x + y
print(add(2, 3))
```

85. Demonstrate exception handling using try, except, and finally.

Logic:
Explain the role of each block.

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
finally:
    print("Execution completed")
```

86. Explain the use of the global keyword.

Logic:
Demonstrate modifying a global variable inside a function.

```
x = 10

def modify():
    global x
    x = 20

modify()
print(x)
```

87. Read a text file line by line safely.

Logic:
Explain why using 'with' is preferred.

```
# with open("file.txt", "r") as f:
#     for line in f:
#         print(line.strip())
```

88. Explain list comprehensions and their advantages.

Logic:
Provide a simple example.

```
squares = [x**2 for x in range(5)]
print(squares)
```

89. Explain *args and **kwargs with examples.

Logic:

```
def demo(*args, **kwargs):
    print(args)
    print(kwargs)
demo(1, 2, a=3, b=4)
```

90. Explain the difference between shallow copy and deep copy.

Logic:
Provide examples.

```
import copy

a = [[1, 2], [3, 4]]
b = copy.copy(a)
c = copy.deepcopy(a)

a[0][0] = 99
print(b)  # affected
print(c)  # not affected
```

91. Implement binary search on a sorted list.

Logic:
Explain time complexity.

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

print(binary_search([1, 3, 5, 7], 5))
```

92. Create a dictionary by mapping two lists together.

Logic:
Explain what happens if lengths differ.

```
keys = ["a", "b"]
values = [1, 2, 3]
print(dict(zip(keys, values)))
```

93. Print a pyramid pattern using stars.

Logic:
Explain the logic behind spacing and star count.

```
n = 5
for i in range(n):
    print(" " * (n - i - 1) + "*" * (2 * i + 1))
```

94. Find the missing number from an array containing numbers from 1 to N.

Logic:
Explain the mathematical approach.

```
def missing_number(arr, n):
    return n * (n + 1) // 2 - sum(arr)

print(missing_number([1, 2, 4, 5], 5))
```

95. Explain the 'with' statement in Python.

Logic:
Provide a real-world example.

```
# 'with' ensures proper acquisition and release of resources
# Example: file handling, locks, database connections
```

96. Convert a string to a datetime object.

Logic:
Explain the format string.

```
from datetime import datetime

dt = datetime.strptime("2024-01-01", "%Y-%m-%d")
print(dt)
```

97. Transpose a matrix using Python.

Logic:
Explain the logic.

```
matrix = [[1, 2], [3, 4]]
transpose = [[row[i] for row in matrix] for i in range(len(matrix[0]))]
print(transpose)
```

98. Retrieve the current working directory.

Logic:
Explain why this is useful.

```
import os
print(os.getcwd())
```

99. Check whether parentheses in a string are balanced.

Logic:
Explain stack usage.

```
def is_balanced(s):
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}
    for ch in s:
        if ch in mapping.values():
            stack.append(ch)
        elif ch in mapping:
            if not stack or stack.pop() != mapping[ch]:
                return False
    return not stack

print(is_balanced("()[]{}"))
```

100. Measure execution time of a code block.

Logic:
Explain why precise benchmarking is hard.

```
import time

start = time.time()
time.sleep(1)
end = time.time()
print(end - start)
```

101. Given an array of integers and a target value, return the indices of the two numbers

Logic:
such that they add up to the target.
Assume exactly one solution exists and the same element cannot be used twice.
Explain the time and space complexity.

```
def two_sum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        diff = target - num
        if diff in seen:
            return [seen[diff], i]
        seen[num] = i

print(two_sum([2, 7, 11, 15], 9))
```

102. Given an array where each element represents the price of a stock on a given day,

Logic:
find the maximum profit you can achieve by buying once and selling once.
Explain why a greedy approach works.

```
def max_profit(prices):
    min_price = float('inf')
    profit = 0
    for price in prices:
        min_price = min(min_price, price)
        profit = max(profit, price - min_price)
    return profit

print(max_profit([7, 1, 5, 3, 6, 4]))
```

103. Given an array of integers, return an array such that each element is the product

Logic:
of all elements except itself.
Do this without using division.

```
def product_except_self(nums):
    n = len(nums)
    res = [1] * n

    prefix = 1
    for i in range(n):
        res[i] = prefix
        prefix *= nums[i]

    postfix = 1
    for i in range(n - 1, -1, -1):
        res[i] *= postfix
        postfix *= nums[i]

    return res

print(product_except_self([1, 2, 3, 4]))
```

104. Given an array representing heights of vertical lines, find two lines that together

Logic:
with the x-axis form a container holding the most water.
Explain the two-pointer strategy.

```
def max_area(height):
    left, right = 0, len(height) - 1
    max_water = 0

    while left < right:
        width = right - left
        max_water = max(max_water, width * min(height[left], height[right]))
        if height[left] < height[right]:
            left += 1
        else:
            right -= 1
    return max_water
```

```
print(max_area([1,8,6,2,5,4,8,3,7]))
```

105. Given an integer array nums, return all unique triplets [nums[i], nums[j], nums[k]]

Logic:
such that i != j != k and nums[i] + nums[j] + nums[k] == 0.
Explain how sorting helps.

```
def three_sum(nums):
    nums.sort()
    res = []

    for i in range(len(nums)):
        if i > 0 and nums[i] == nums[i - 1]:
            continue
        left, right = i + 1, len(nums) - 1
        while left < right:
            s = nums[i] + nums[left] + nums[right]
            if s == 0:
                res.append([nums[i], nums[left], nums[right]])
                left += 1
                while left < right and nums[left] == nums[left - 1]:
                    left += 1
            elif s < 0:
                left += 1
            else:
                right -= 1
    return res

print(three_sum([-1,0,1,2,-1,-4]))
```

106. Find the contiguous subarray with the maximum sum.

Logic:
Explain Kadane’s algorithm.

```
def max_subarray(nums):
    current_sum = nums[0]
    max_sum = nums[0]

    for num in nums[1:]:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)

    return max_sum

print(max_subarray([-2,1,-3,4,-1,2,1,-5,4]))
```

107. Rotate a square matrix by 90 degrees clockwise in-place.

Logic:
Explain the transformation steps.

```
def rotate_matrix(matrix):
    matrix.reverse()
```

```
    for i in range(len(matrix)):
        for j in range(i):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]

m = [[1,2,3],[4,5,6],[7,8,9]]
rotate_matrix(m)
print(m)
```

108. Return all elements of a matrix in spiral order.

Logic:
Explain the boundary shrinking technique.

```
def spiral_order(matrix):
    res = []
    while matrix:
        res += matrix.pop(0)
        matrix = list(zip(*matrix))[::-1]
    return res

print(spiral_order([[1,2,3],[4,5,6],[7,8,9]]))
```

109. Search for a target value in a rotated sorted array.

Logic:
Explain how binary search is adapted.

```
def search_rotated(nums, target):
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid

        if nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1

    return -1

print(search_rotated([4,5,6,7,0,1,2], 0))
```

110. Merge overlapping intervals.

Logic:
Explain why sorting by start time is necessary.

```
def merge_intervals(intervals):
    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]
```

```
    for start, end in intervals[1:]:
        last_end = merged[-1][1]
        if start <= last_end:
            merged[-1][1] = max(last_end, end)
        else:
            merged.append([start, end])

    return merged

print(merge_intervals([[1,3],[2,6],[8,10],[15,18]]))
```

111. Insert a new interval into a list of non-overlapping intervals and merge if necessary.

Logic:

```
def insert_interval(intervals, new_interval):
    res = []
    for interval in intervals:
        if new_interval[1] < interval[0]:
            res.append(new_interval)
            return res + intervals[intervals.index(interval):]
        elif new_interval[0] > interval[1]:
            res.append(interval)
        else:
            new_interval = [
                min(new_interval[0], interval[0]),
                max(new_interval[1], interval[1])
            ]
    res.append(new_interval)
    return res

print(insert_interval([[1,3],[6,9]], [2,5]))
```

112. Find the length of the longest consecutive sequence in an unsorted array.

Logic:
Explain why a set is used.

```
def longest_consecutive(nums):
    num_set = set(nums)
    longest = 0

    for num in num_set:
        if num - 1 not in num_set:
            length = 1
            while num + length in num_set:
                length += 1
            longest = max(longest, length)

    return longest

print(longest_consecutive([100,4,200,1,3,2]))
```

113. Move all zeroes in an array to the end while maintaining the order of non-zero elements.

Logic:

Explain the two-pointer approach.

```
def move_zeroes(nums):
    pos = 0
    for i in range(len(nums)):
        if nums[i] != 0:
            nums[pos], nums[i] = nums[i], nums[pos]
            pos += 1

nums = [0,1,0,3,12]
move_zeroes(nums)
print(nums)
```

114. Find the duplicate number in an array containing n+1 integers

Logic:
where each integer is between 1 and n.
Explain Floyd’s cycle detection.

```
def find_duplicate(nums):
    slow = fast = 0
    while True:
        slow = nums[slow]
        fast = nums[nums[fast]]
        if slow == fast:
            break

    slow2 = 0
    while slow != slow2:
        slow = nums[slow]
        slow2 = nums[slow2]

    return slow

print(find_duplicate([1,3,4,2,2]))
```

115. Find the total number of continuous subarrays whose sum equals k.

Logic:
Explain prefix sum usage.

```
def subarray_sum(nums, k):
    count = 0
    prefix_sum = 0
    sums = {0: 1}

    for num in nums:
        prefix_sum += num
        count += sums.get(prefix_sum - k, 0)
        sums[prefix_sum] = sums.get(prefix_sum, 0) + 1

    return count

print(subarray_sum([1,1,1], 2))
```

116. Find the maximum value in each sliding window of size k.

Logic:
Explain why a deque is used.

```
from collections import deque

def max_sliding_window(nums, k):
    dq = deque()
    res = []

    for i, num in enumerate(nums):
        while dq and dq[0] <= i - k:
            dq.popleft()
        while dq and nums[dq[-1]] < num:
            dq.pop()
        dq.append(i)
        if i >= k - 1:
            res.append(nums[dq[0]])

    return res

print(max_sliding_window([1,3,-1,-3,5,3,6,7], 3))
```

117. Given an elevation map, compute how much water it can trap after raining.

Logic:
Explain the two-pointer approach.

```
def trap_rain_water(height):
    left, right = 0, len(height) - 1
    left_max = right_max = 0
    water = 0

    while left < right:
        if height[left] < height[right]:
            left_max = max(left_max, height[left])
            water += left_max - height[left]
            left += 1
        else:
            right_max = max(right_max, height[right])
            water += right_max - height[right]
            right -= 1

    return water

print(trap_rain_water([0,1,0,2,1,0,1,3,2,1,2,1]))
```

118. Find the smallest missing positive integer in an unsorted array.

Logic:
Explain why in-place hashing works.

```
def first_missing_positive(nums):
    n = len(nums)
    for i in range(n):
        while 1 <= nums[i] <= n and nums[nums[i] - 1] != nums[i]:
            nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]

    for i in range(n):
        if nums[i] != i + 1:
```

```
            return i + 1

    return n + 1

print(first_missing_positive([3,4,-1,1]))
```

119. Sort an array containing only 0s, 1s, and 2s without using sorting.

Logic:
Explain the Dutch National Flag algorithm.

```
def sort_colors(nums):
    left, right, i = 0, len(nums) - 1, 0
    while i <= right:
        if nums[i] == 0:
            nums[left], nums[i] = nums[i], nums[left]
            left += 1
        elif nums[i] == 2:
            nums[right], nums[i] = nums[i], nums[right]
            right -= 1
            i -= 1
        i += 1

nums = [2,0,2,1,1,0]
sort_colors(nums)
print(nums)
```

120. Find the median of two sorted arrays.

Logic:
Explain the binary search partition approach.

```
def find_median_sorted_arrays(nums1, nums2):
    A, B = nums1, nums2
    if len(A) > len(B):
        A, B = B, A

    total = len(A) + len(B)
    half = total // 2
    left, right = 0, len(A)

    while True:
        i = (left + right) // 2
        j = half - i

        Aleft = A[i-1] if i > 0 else float('-inf')
        Aright = A[i] if i < len(A) else float('inf')
        Bleft = B[j-1] if j > 0 else float('-inf')
        Bright = B[j] if j < len(B) else float('inf')

        if Aleft <= Bright and Bleft <= Aright:
            if total % 2:
                return min(Aright, Bright)
            return (max(Aleft, Bleft) + min(Aright, Bright)) / 2
        elif Aleft > Bright:
            right = i - 1
        else:
            left = i + 1

print(find_median_sorted_arrays([1,3], [2]))
```

121. Reverse a singly linked list.

Logic:
Explain pointer manipulation.

```
# Conceptual question – implementation depends on ListNode definition
```

122. Merge two sorted linked lists.

Logic:
Explain iterative vs recursive approaches.

```
# Conceptual question – implementation depends on ListNode definition
```

123. Reorder a linked list in a specific pattern.

Logic:
Explain the steps involved.

```
# Conceptual question – implementation depends on ListNode definition
```

124. Remove the N-th node from the end of a linked list.

Logic:
Explain the two-pointer technique.

```
# Conceptual question – implementation depends on ListNode definition
```

125. Detect a cycle in a linked list.

Logic:
Explain Floyd’s cycle detection algorithm.

```
# Conceptual question – implementation depends on ListNode definition
```

126. Define a binary tree and explain the role of a TreeNode.

Logic:
Write the basic TreeNode class used in most interview problems.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
```

```
        self.val = val
        self.left = left
        self.right = right
```

127. Implement inorder traversal of a binary tree using recursion.

Logic:
Explain why recursion naturally fits tree traversal.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def inorder(root):
    if not root:
        return []
    return inorder(root.left) + [root.val] + inorder(root.right)
```

128. Implement preorder traversal of a binary tree using recursion.

Logic:
Explain the traversal order.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def preorder(root):
    if not root:
        return []
    return [root.val] + preorder(root.left) + preorder(root.right)
```

129. Implement postorder traversal of a binary tree using recursion.

Logic:
Explain a real-world use case.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def postorder(root):
    if not root:
        return []
    return postorder(root.left) + postorder(root.right) + [root.val]
```

130. Implement level-order traversal (BFS) of a binary tree.

Logic:
Explain why a queue is required.

```
from collections import deque

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def level_order(root):
    if not root:
        return []
    q = deque([root])
    res = []
    while q:
        node = q.popleft()
        res.append(node.val)
        if node.left:
            q.append(node.left)
        if node.right:
            q.append(node.right)
    return res
```

131. Compute the maximum depth (height) of a binary tree.

Logic:
Explain the recursive relation.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def max_depth(root):
    if not root:
        return 0
    return 1 + max(max_depth(root.left), max_depth(root.right))
```

132. Check whether two binary trees are identical.

Logic:
Explain the base and recursive cases.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def is_same_tree(p, q):
    if not p and not q:
        return True
```

```
    if not p or not q or p.val != q.val:
        return False
    return is_same_tree(p.left, q.left) and is_same_tree(p.right, q.right)
```

133. Check whether a binary tree is symmetric.

Logic:
Explain how mirroring is used.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def is_symmetric(root):
    def mirror(a, b):
        if not a and not b:
            return True
        if not a or not b:
            return False
        return (
            a.val == b.val and
            mirror(a.left, b.right) and
            mirror(a.right, b.left)
        )
    return mirror(root.left, root.right) if root else True
```

134. Invert a binary tree.

Logic:
Explain how recursion swaps subtrees.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def invert_tree(root):
    if root:
        root.left, root.right = invert_tree(root.right), invert_tree(root.left)
    return root
```

135. Determine whether a binary tree is height-balanced.

Logic:
Explain why height computation is combined with checking.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
def is_balanced(root):
    def check(node):
```

```
        if not node:
            return 0
        left = check(node.left)
        if left == -1:
            return -1
        right = check(node.right)
        if right == -1 or abs(left - right) > 1:
            return -1
        return 1 + max(left, right)
    return check(root) != -1
```

136. Validate whether a binary tree is a binary search tree (BST).

Logic:
Explain the min-max constraint approach.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def is_valid_bst(root, low=float('-inf'), high=float('inf')):
    if not root:
        return True
    if not (low < root.val < high):
        return False
    return (
        is_valid_bst(root.left, low, root.val) and
        is_valid_bst(root.right, root.val, high)
    )
```

137. Find the lowest common ancestor (LCA) of two nodes in a BST.

Logic:
Explain how BST properties simplify the solution.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def lca_bst(root, p, q):
    if root.val > p.val and root.val > q.val:
        return lca_bst(root.left, p, q)
    if root.val < p.val and root.val < q.val:
        return lca_bst(root.right, p, q)
    return root
```

138. Compute the diameter of a binary tree.

Logic:
Explain why depth and diameter are computed together.

```
class TreeNode:
```

```
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def diameter(root):
    ans = 0
    def depth(node):
        nonlocal ans
        if not node:
            return 0
        left = depth(node.left)
        right = depth(node.right)
        ans = max(ans, left + right)
        return 1 + max(left, right)
    depth(root)
    return ans
```

139. Check if a root-to-leaf path sums to a given value.

Logic:
Explain DFS usage.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def has_path_sum(root, target):
    if not root:
        return False
    if not root.left and not root.right:
        return target == root.val
    return (
        has_path_sum(root.left, target - root.val) or
        has_path_sum(root.right, target - root.val)
    )
```

140. Find all root-to-leaf paths in a binary tree.

Logic:
Explain backtracking.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def binary_tree_paths(root):
    res = []
    def dfs(node, path):
        if not node:
            return
        path.append(str(node.val))
        if not node.left and not node.right:
            res.append("->".join(path))
        else:
            dfs(node.left, path)
            dfs(node.right, path)
```

```
        path.pop()
    dfs(root, [])
    return res
```

141. Find the minimum depth of a binary tree.

Logic:
Explain why BFS is preferred.

```
from collections import deque

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def min_depth(root):
    if not root:
        return 0
    q = deque([(root, 1)])
    while q:
        node, depth = q.popleft()
        if not node.left and not node.right:
            return depth
        if node.left:
            q.append((node.left, depth + 1))
        if node.right:
            q.append((node.right, depth + 1))
```

142. Flatten a binary tree to a linked list in-place.

Logic:
Explain preorder traversal usage.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def flatten(root):
    def dfs(node):
        if not node:
            return None
        left_tail = dfs(node.left)
        right_tail = dfs(node.right)
        if left_tail:
            left_tail.right = node.right
            node.right = node.left
            node.left = None
        return right_tail or left_tail or node
    dfs(root)
```

143. Serialize and deserialize a binary tree.

Logic:

Explain why preorder traversal is commonly used.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def serialize(root):
    vals = []
    def dfs(node):
        if not node:
            vals.append('#')
            return
        vals.append(str(node.val))
        dfs(node.left)
        dfs(node.right)
    dfs(root)
    return ','.join(vals)

def deserialize(data):
    vals = iter(data.split(','))
    def dfs():
        v = next(vals)
        if v == '#':
            return None
        node = TreeNode(int(v))
        node.left = dfs()
        node.right = dfs()
        return node
    return dfs()
```

144. Count the number of nodes in a complete binary tree.

Logic:
Explain the height-based optimization.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def count_nodes(root):
    def left_height(node):
        h = 0
        while node:
            h += 1
            node = node.left
        return h

    def right_height(node):
        h = 0
        while node:
            h += 1
            node = node.right
        return h

    if not root:
        return 0
    lh, rh = left_height(root), right_height(root)
    if lh == rh:
        return (1 << lh) - 1
    return 1 + count_nodes(root.left) + count_nodes(root.right)
```

145. Construct a binary tree from preorder and inorder traversal arrays.

Logic:
Explain how recursion splits the tree.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def build_tree(preorder, inorder):
    if not preorder:
        return None
    root_val = preorder.pop(0)
    root = TreeNode(root_val)
    idx = inorder.index(root_val)
    root.left = build_tree(preorder, inorder[:idx])
    root.right = build_tree(preorder, inorder[idx+1:])
    return root
```

146. Explain recursion depth limits in Python.

Logic:
How can deep recursion be handled safely?

```
# Python has a default recursion limit (~1000).
# Solutions: iterative approach, tail recursion elimination, sys.setrecursionlimit()
```

147. Implement a recursive solution to compute power(x, n).

Logic:
Explain divide-and-conquer.

```
def power(x, n):
    if n == 0:
        return 1
    half = power(x, n // 2)
    return half * half if n % 2 == 0 else half * half * x
```

148. Explain memoization using a recursive Fibonacci example.

Logic:

```
from functools import lru_cache

@lru_cache(None)
def fib(n):
    if n < 2:
        return n
```

```
    return fib(n-1) + fib(n-2)
```

149. Convert a sorted array into a height-balanced BST.

Logic:
Explain why the middle element is chosen.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def sorted_array_to_bst(nums):
    if not nums:
        return None
    mid = len(nums) // 2
    root = TreeNode(nums[mid])
    root.left = sorted_array_to_bst(nums[:mid])
    root.right = sorted_array_to_bst(nums[mid+1:])
    return root
```

150. Explain tail recursion and whether Python optimizes it.

Logic:

```
# Tail recursion occurs when the recursive call is the last operation.
# Python does NOT perform tail-call optimization.
```

151. Explain what a graph is and list common ways to represent a graph in Python.

Logic:
Discuss adjacency list vs adjacency matrix.

```
# Graph representations:
# 1. Adjacency List: dict {node: [neighbors]} – space efficient for sparse graphs
# 2. Adjacency Matrix: 2D list – fast edge lookup, high space cost
```

152. Implement Depth-First Search (DFS) for a graph using recursion.

Logic:
Explain visited set usage.

```
def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
    for neigh in graph.get(node, []):
        if neigh not in visited:
```

```
            dfs(graph, neigh, visited)
    return visited

graph = {1: [2, 3], 2: [4], 3: [], 4: []}
print(dfs(graph, 1))
```

153. Implement Breadth-First Search (BFS) for a graph.

Logic:
Explain why a queue is required.

```
from collections import deque

def bfs(graph, start):
    visited = set([start])
    q = deque([start])
    order = []

    while q:
        node = q.popleft()
        order.append(node)
        for neigh in graph.get(node, []):
            if neigh not in visited:
                visited.add(neigh)
                q.append(neigh)
    return order

print(bfs({1: [2, 3], 2: [4], 3: [], 4: []}, 1))
```

154. Detect a cycle in an undirected graph using DFS.

Logic:
Explain parent tracking.

```
def has_cycle_undirected(graph):
    visited = set()

    def dfs(node, parent):
        visited.add(node)
        for neigh in graph.get(node, []):
            if neigh not in visited:
                if dfs(neigh, node):
                    return True
            elif neigh != parent:
                return True
        return False

    for node in graph:
        if node not in visited:
            if dfs(node, None):
                return True
    return False
```

155. Detect a cycle in a directed graph using DFS.

Logic:
Explain recursion stack usage.

```
def has_cycle_directed(graph):
    visited, rec = set(), set()

    def dfs(node):
        visited.add(node)
        rec.add(node)
        for neigh in graph.get(node, []):
            if neigh not in visited:
                if dfs(neigh):
                    return True
            elif neigh in rec:
                return True
        rec.remove(node)
        return False

    return any(dfs(n) for n in graph if n not in visited)
```

156. Count the number of connected components in an undirected graph.

Logic:
Explain why DFS/BFS works.

```
def count_components(graph):
    visited = set()
    count = 0

    def dfs(node):
        visited.add(node)
        for neigh in graph.get(node, []):
            if neigh not in visited:
                dfs(neigh)

    for node in graph:
        if node not in visited:
            dfs(node)
            count += 1
    return count
```

157. Perform topological sorting of a directed acyclic graph using DFS.

Logic:
Explain post-order processing.

```
def topo_sort(graph):
    visited = set()
    stack = []

    def dfs(node):
        visited.add(node)
        for neigh in graph.get(node, []):
            if neigh not in visited:
                dfs(neigh)
        stack.append(node)

    for node in graph:
        if node not in visited:
            dfs(node)

    return stack[::-1]
```

158. Perform topological sorting using Kahn’s algorithm.

Logic:
Explain indegree usage.

```
from collections import deque

def topo_kahn(graph):
    indeg = {u: 0 for u in graph}
    for u in graph:
        for v in graph[u]:
            indeg[v] += 1

    q = deque([u for u in indeg if indeg[u] == 0])
    res = []

    while q:
        u = q.popleft()
        res.append(u)
        for v in graph[u]:
            indeg[v] -= 1
            if indeg[v] == 0:
                q.append(v)
    return res
```

159. Find the shortest path in an unweighted graph using BFS.

Logic:
Explain why BFS guarantees shortest path.

```
from collections import deque

def shortest_path(graph, start, end):
    q = deque([(start, [start])])
    visited = set([start])

    while q:
        node, path = q.popleft()
        if node == end:
            return path
        for neigh in graph.get(node, []):
            if neigh not in visited:
                visited.add(neigh)
                q.append((neigh, path + [neigh]))
    return None
```

160. Check if a graph is bipartite.

Logic:
Explain graph coloring.

```
def is_bipartite(graph):
    color = {}

    for node in graph:
        if node not in color:
            color[node] = 0
            stack = [node]
            while stack:
                u = stack.pop()
                for v in graph[u]:
                    if v not in color:
```

```
                        color[v] = 1 - color[u]
                        stack.append(v)
                    elif color[v] == color[u]:
                        return False
    return True
```

161. Explain the Union-Find (Disjoint Set) data structure.

Logic:
List its main operations.

```
# Operations:
# find(x): find representative
# union(x, y): merge sets
# Used in cycle detection, Kruskal's MST
```

162. Implement Union-Find with path compression.

Logic:
Explain how it improves performance.

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        rx, ry = self.find(x), self.find(y)
        if rx != ry:
            self.parent[ry] = rx
```

163. Detect a cycle in an undirected graph using Union-Find.

Logic:
Explain why this works.

```
def has_cycle_union_find(edges, n):
    uf = UnionFind(n)
    for u, v in edges:
        if uf.find(u) == uf.find(v):
            return True
        uf.union(u, v)
    return False
```

164. Find the minimum spanning tree using Kruskal’s algorithm.

Logic:
Explain edge sorting.

```
def kruskal(edges, n):
    uf = UnionFind(n)
    mst = []
    edges.sort(key=lambda x: x[2])
    for u, v, w in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, w))
    return mst
```

165. Explain Dijkstra’s algorithm for shortest paths.

Logic:
Mention its limitation.

```
# Dijkstra finds shortest paths from a source.
# Limitation: does not work with negative edge weights.
```

166. Implement Dijkstra’s algorithm using a priority queue.

Logic:

```
import heapq

def dijkstra(graph, start):
    dist = {node: float('inf') for node in graph}
    dist[start] = 0
    pq = [(0, start)]

    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v, w in graph[u]:
            if dist[v] > d + w:
                dist[v] = d + w
                heapq.heappush(pq, (dist[v], v))
    return dist
```

167. Explain Bellman-Ford algorithm.

Logic:
Explain when it is preferred over Dijkstra.

```
# Bellman-Ford handles negative weights and detects negative cycles.
# Slower than Dijkstra.
```

168. Detect a negative cycle using Bellman-Ford.

Logic:

```
def has_negative_cycle(edges, n):
    dist = [0] * n
    for _ in range(n - 1):
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            return True
    return False
```

169. Explain Floyd-Warshall algorithm.

Logic:
Discuss time complexity.

```
# All-pairs shortest paths
# Time complexity: O(V^3)
```

170. Implement Floyd-Warshall algorithm.

Logic:

```
def floyd_warshall(dist):
    n = len(dist)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    return dist
```

171. Solve the course schedule problem.

Logic:
Explain cycle detection in directed graph.

```
def can_finish(numCourses, prerequisites):
    graph = {i: [] for i in range(numCourses)}
    for a, b in prerequisites:
        graph[b].append(a)
    return not has_cycle_directed(graph)
```

172. Find all nodes eventually safe in a directed graph.

Logic:
Explain DFS coloring.

```
def eventual_safe_nodes(graph):
    n = len(graph)
    color = [0] * n  # 0=unvisited,1=visiting,2=safe

    def dfs(u):
```

```
        if color[u]:
            return color[u] == 2
        color[u] = 1
        for v in graph[u]:
            if not dfs(v):
                return False
        color[u] = 2
        return True

    return [i for i in range(n) if dfs(i)]
```

173. Clone an undirected graph.

Logic:
Explain hashmap usage.

```
# Conceptual – requires Node definition and DFS/BFS cloning
```

174. Check if all paths lead to a destination.

Logic:
Explain graph pruning.

```
# Conceptual – DFS with memoization and terminal checks
```

175. Explain why graphs are harder than trees in interviews.

Logic:

```
# Graphs allow cycles, multiple parents, disconnected components.
# Requires visited tracking and cycle handling.
```

176. Explain Dynamic Programming (DP).

Logic:
Differentiate between top-down and bottom-up approaches.

```
# Dynamic Programming solves problems by breaking them into overlapping subproblems.
# Top-down: recursion + memoization
# Bottom-up: iterative table filling
```

177. Compute the N-th Fibonacci number using Dynamic Programming.

Logic:
Explain why DP improves over naive recursion.

```
def fib(n):
    if n < 2:
        return n
    dp = [0, 1]
    for i in range(2, n + 1):
        dp.append(dp[i-1] + dp[i-2])
    return dp[n]

print(fib(10))
```

178. Solve the Coin Change problem.

Logic:
Given coin denominations and an amount, return the minimum number of coins needed.

```
def coin_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0
    for coin in coins:
        for x in range(coin, amount + 1):
            dp[x] = min(dp[x], dp[x - coin] + 1)
    return dp[amount] if dp[amount] != float('inf') else -1

print(coin_change([1,2,5], 11))
```

179. Solve the Longest Increasing Subsequence problem.

Logic:
Explain time complexity.

```
def length_of_lis(nums):
    dp = [1] * len(nums)
    for i in range(len(nums)):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)

print(length_of_lis([10,9,2,5,3,7,101,18]))
```

180. Solve the 0/1 Knapsack problem using Dynamic Programming.

Logic:
Explain table construction.

```
def knapsack(weights, values, W):
    n = len(weights)
    dp = [[0]*(W+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for w in range(W+1):
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w],
                               values[i-1] + dp[i-1][w-weights[i-1]])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][W]

print(knapsack([2,3,4], [3,4,5], 5))
```

181. Solve the House Robber problem.

Logic:
Explain the recurrence relation.

```
def rob(nums):
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]
    dp1, dp2 = 0, 0
    for num in nums:
        dp1, dp2 = dp2, max(dp2, dp1 + num)
    return dp2

print(rob([2,7,9,3,1]))
```

182. Solve the Longest Common Subsequence problem.

Logic:
Explain why greedy fails.

```
def lcs(a, b):
    m, n = len(a), len(b)
    dp = [[0]*(n+1) for _ in range(m+1)]
    for i in range(m):
        for j in range(n):
            if a[i] == b[j]:
                dp[i+1][j+1] = dp[i][j] + 1
            else:
                dp[i+1][j+1] = max(dp[i][j+1], dp[i+1][j])
    return dp[m][n]

print(lcs("abcde", "ace"))
```

183. Solve the Edit Distance problem.

Logic:
Explain allowed operations.

```
def edit_distance(a, b):
    m, n = len(a), len(b)
    dp = [[0]*(n+1) for _ in range(m+1)]
    for i in range(m+1):
        dp[i][0] = i
    for j in range(n+1):
        dp[0][j] = j

    for i in range(1, m+1):
        for j in range(1, n+1):
            if a[i-1] == b[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(
                    dp[i-1][j],    # delete
                    dp[i][j-1],    # insert
                    dp[i-1][j-1]   # replace
```

```
                )
    return dp[m][n]

print(edit_distance("horse", "ros"))
```

184. Explain matrix-chain multiplication problem.

Logic:
Explain optimal substructure.

```
# Matrix-chain multiplication finds optimal parenthesization.
# Uses DP to minimize scalar multiplications.
```

185. Solve the Climbing Stairs problem.

Logic:
Explain relation to Fibonacci.

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

print(climb_stairs(5))
```

186. Explain Greedy algorithms.

Logic:
Provide an example where greedy works.

```
# Greedy makes locally optimal choices.
# Example: Activity selection, interval scheduling.
```

187. Solve the Activity Selection problem.

Logic:
Explain greedy choice.

```
def activity_selection(intervals):
    intervals.sort(key=lambda x: x[1])
    res = [intervals[0]]
    for s, e in intervals[1:]:
        if s >= res[-1][1]:
            res.append((s, e))
    return res

print(activity_selection([(1,3),(2,4),(3,5),(0,6)]))
```

188. Solve the N-Queens problem.

Logic:
Explain backtracking.

```
def solve_n_queens(n):
    res = []
    cols, diag1, diag2 = set(), set(), set()

    def backtrack(r, board):
        if r == n:
            res.append(board[:])
            return
        for c in range(n):
            if c in cols or r-c in diag1 or r+c in diag2:
                continue
            cols.add(c); diag1.add(r-c); diag2.add(r+c)
            backtrack(r+1, board + [c])
            cols.remove(c); diag1.remove(r-c); diag2.remove(r+c)

    backtrack(0, [])
    return res

print(len(solve_n_queens(4)))
```

189. Generate all permutations of a list.

Logic:
Explain time complexity.

```
def permutations(nums):
    res = []
    def backtrack(path, remaining):
        if not remaining:
            res.append(path)
            return
        for i in range(len(remaining)):
            backtrack(path + [remaining[i]], remaining[:i] + remaining[i+1:])
    backtrack([], nums)
    return res

print(permutations([1,2,3]))
```

190. Generate all subsets of a given set.

Logic:
Explain bitmask vs backtracking.

```
def subsets(nums):
    res = [[]]
    for num in nums:
        res += [curr + [num] for curr in res]
    return res

print(subsets([1,2,3]))
```

191. Count the number of set bits in an integer.

Logic:
Explain Brian Kernighan’s algorithm.

```
def count_bits(n):
    count = 0
    while n:
        n &= n - 1
        count += 1
    return count

print(count_bits(15))
```

192. Check if a number is a power of two.

Logic:
Explain bitwise logic.

```
def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0

print(is_power_of_two(16))
```

193. Find the single number in an array where every element appears twice except one.

Logic:
Explain XOR usage.

```
def single_number(nums):
    res = 0
    for n in nums:
        res ^= n
    return res

print(single_number([4,1,2,1,2]))
```

194. Reverse bits of a 32-bit integer.

Logic:
Explain shifting.

```
def reverse_bits(n):
    res = 0
    for _ in range(32):
        res = (res << 1) | (n & 1)
        n >>= 1
    return res
```

195. Explain why bit manipulation is efficient.

Logic:

```
# Bit operations are constant-time and memory efficient.
```

196. Explain memoization vs caching.

Logic:

```
# Memoization is function-level caching.
# Caching is broader (DB, HTTP, etc.).
```

197. Explain time vs space trade-offs with an example.

Logic:

```
# Using extra memory (hash maps) to reduce time complexity.
```

198. Explain the difference between process and thread.

Logic:

```
# Process: independent memory
# Thread: shared memory within process
```

199. Explain deadlock and its necessary conditions.

Logic:

```
# Mutual exclusion, hold and wait, no preemption, circular wait.
```

200. Summarize how to approach unseen interview problems.

Logic:

```
# Clarify problem, identify pattern, discuss complexity, code cleanly.
```

201. Given an array of integers and a target value, return the indices of the two numbers

Logic:

such that they add up to the target.
Explain why a hash-map based solution is optimal.

```
def two_sum(nums, target):
    seen = {}
    for i, n in enumerate(nums):
        if target - n in seen:
            return [seen[target - n], i]
        seen[n] = i
    return []
```

202. Check whether two strings are anagrams of each other.

Logic:
Explain why character frequency comparison works.

```
def is_anagram(s, t):
    if len(s) != len(t):
        return False
    freq = {}
    for c in s:
        freq[c] = freq.get(c, 0) + 1
    for c in t:
        if c not in freq or freq[c] == 0:
            return False
        freq[c] -= 1
    return True
```

203. Find the contiguous subarray with the maximum sum.

Logic:
Explain Kadane’s Algorithm and why resetting the sum is valid.

```
def max_subarray(nums):
    best = nums[0]
    current = 0
    for n in nums:
        current = max(n, current + n)
        best = max(best, current)
    return best
```

204. Return an array where each index contains the product of all numbers

Logic:
except itself, without using division.

```
def product_except_self(nums):
    res = [1] * len(nums)
    prefix = 1
    for i in range(len(nums)):
        res[i] = prefix
        prefix *= nums[i]
    postfix = 1
    for i in range(len(nums)-1, -1, -1):
        res[i] *= postfix
```

```
        postfix *= nums[i]
    return res
```

205. Validate whether a string of parentheses is balanced.

Logic:
Explain why a stack is required.

```
def is_valid_parentheses(s):
    stack = []
    pairs = {')': '(', ']': '[', '}': '{'}
    for c in s:
        if c in pairs:
            if not stack or stack.pop() != pairs[c]:
                return False
        else:
            stack.append(c)
    return not stack
```

206. Find the maximum profit achievable from a single stock buy-sell transaction.

Logic:
Explain the greedy choice.

```
def max_profit(prices):
    min_price = float('inf')
    profit = 0
    for p in prices:
        min_price = min(min_price, p)
        profit = max(profit, p - min_price)
    return profit
```

207. Find the length of the longest substring without repeating characters.

Logic:
Explain the sliding window invariant.

```
def longest_unique_substring(s):
    seen = set()
    l = 0
    best = 0
    for r in range(len(s)):
        while s[r] in seen:
            seen.remove(s[l])
            l += 1
        seen.add(s[r])
        best = max(best, r - l + 1)
    return best
```

208. Find two vertical lines that form a container holding the maximum water.

Logic:
Explain why the two-pointer strategy is optimal.

```
def max_area(height):
    l, r = 0, len(height) - 1
    best = 0
    while l < r:
        best = max(best, (r - l) * min(height[l], height[r]))
        if height[l] < height[r]:
            l += 1
        else:
            r -= 1
    return best
```

209. Find all unique triplets in an array that sum to zero.

Logic:
Explain duplicate elimination.

```
def three_sum(nums):
    nums.sort()
    res = []
    for i in range(len(nums)):
        if i > 0 and nums[i] == nums[i-1]:
            continue
        l, r = i+1, len(nums)-1
        while l < r:
            s = nums[i] + nums[l] + nums[r]
            if s == 0:
                res.append([nums[i], nums[l], nums[r]])
                l += 1
                while l < r and nums[l] == nums[l-1]:
                    l += 1
            elif s < 0:
                l += 1
            else:
                r -= 1
    return res
```

210. Merge overlapping intervals.

Logic:
Explain why sorting is mandatory.

```
def merge_intervals(intervals):
    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]
    for s, e in intervals[1:]:
        if s <= merged[-1][1]:
            merged[-1][1] = max(merged[-1][1], e)
        else:
            merged.append([s, e])
    return merged
```

211. Reverse a singly linked list.

Logic:
Explain pointer reassignment.

```
def reverse_list(head):
    prev, curr = None, head
    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt
    return prev
```

212. Detect a cycle in a linked list.

Logic:
Explain Floyd’s Tortoise and Hare algorithm.

```
def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

213. Merge two sorted linked lists.

Logic:
Explain dummy node usage.

```
def merge_two_lists(l1, l2):
    dummy = tail = ListNode()
    while l1 and l2:
        if l1.val < l2.val:
            tail.next, l1 = l1, l1.next
        else:
            tail.next, l2 = l2, l2.next
        tail = tail.next
    tail.next = l1 or l2
    return dummy.next
```

214. Invert a binary tree.

Logic:
Explain subtree swapping.

```
def invert_tree(root):
    if root:
        root.left, root.right = invert_tree(root.right), invert_tree(root.left)
    return root
```

215. Find the maximum depth of a binary tree.

Logic:
Explain recursive height calculation.

```
def max_depth(root):
    if not root:
        return 0
    return 1 + max(max_depth(root.left), max_depth(root.right))
```

216. Count the number of islands in a grid.

Logic:
Explain DFS flooding.

```
def num_islands(grid):
    rows, cols = len(grid), len(grid[0])
    def dfs(r, c):
        if r<0 or c<0 or r>=rows or c>=cols or grid[r][c] != '1':
            return
        grid[r][c] = '0'
        dfs(r+1,c); dfs(r-1,c); dfs(r,c+1); dfs(r,c-1)

    count = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '1':
                dfs(r,c)
                count += 1
    return count
```

217. Find the lowest common ancestor of two nodes in a BST.

Logic:
Explain BST ordering property.

```
def lca_bst(root, p, q):
    while root:
        if p.val < root.val and q.val < root.val:
            root = root.left
        elif p.val > root.val and q.val > root.val:
            root = root.right
        else:
            return root
```

218. Compute number of ways to climb stairs.

Logic:
Explain Fibonacci relation.

```
def climb_stairs(n):
    a, b = 1, 1
    for _ in range(n):
        a, b = b, a+b
```

```
    return a
```

219. Solve the House Robber problem.

Logic:
Explain optimal substructure.

```
def rob(nums):
    prev, curr = 0, 0
    for n in nums:
        prev, curr = curr, max(curr, prev + n)
    return curr
```

220. Find the longest palindromic substring.

Logic:
Explain expand-around-center approach.

```
def longest_palindrome(s):
    res = ""
    for i in range(len(s)):
        for l, r in [(i,i),(i,i+1)]:
            while l>=0 and r<len(s) and s[l]==s[r]:
                if r-l+1 > len(res):
                    res = s[l:r+1]
                l-=1; r+=1
    return res
```

221. Perform binary search on a sorted array.

Logic:
Explain termination condition.

```
def binary_search(nums, target):
    l, r = 0, len(nums)-1
    while l <= r:
        m = (l+r)//2
        if nums[m] == target:
            return m
        elif nums[m] < target:
            l = m+1
        else:
            r = m-1
    return -1
```

222. Find the minimum element in a rotated sorted array.

Logic:
Explain pivot logic.

```
def find_min(nums):
    l, r = 0, len(nums)-1
```

```
    while l < r:
        m = (l+r)//2
        if nums[m] > nums[r]:
            l = m+1
        else:
            r = m
    return nums[l]
```

223. Find the K-th largest element using a heap.

Logic:
Explain why heap size is limited.

```
import heapq
def kth_largest(nums, k):
    h = []
    for n in nums:
        heapq.heappush(h, n)
        if len(h) > k:
            heapq.heappop(h)
    return h[0]
```

224. Flatten a deeply nested list.

Logic:
Explain recursion depth.

```
def flatten(lst):
    res = []
    for x in lst:
        if isinstance(x, list):
            res.extend(flatten(x))
        else:
            res.append(x)
    return res
```

225. Check whether a number is prime efficiently.

Logic:
Explain square-root optimization.

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True
```

226. Implement a timing decorator.

Logic:
Explain functools.wraps usage.

```
import time, functools

def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(time.time() - start)
        return result
    return wrapper
```

227. Implement an LRU Cache.

Logic:
Explain why OrderedDict works.

```
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key):
        if key not in self.cache:
            return -1
        self.cache.move_to_end(key)
        return self.cache[key]

    def put(self, key, value):
        if key in self.cache:
            self.cache.move_to_end(key)
        self.cache[key] = value
        if len(self.cache) > self.capacity:
            self.cache.popitem(last=False)
```

228. Encode and decode a list of strings.

Logic:
Explain delimiter safety.

```
def encode(strs):
    return ''.join(f'{len(s)}#{s}' for s in strs)

def decode(s):
    res, i = [], 0
    while i < len(s):
        j = i
        while s[j] != '#':
            j += 1
        length = int(s[i:j])
        res.append(s[j+1:j+1+length])
        i = j+1+length
    return res
```

229. Implement a Trie data structure.

Logic:

Explain prefix-based search.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for c in word:
            node = node.children.setdefault(c, TrieNode())
        node.end = True

    def search(self, word):
        node = self.root
        for c in word:
            if c not in node.children:
                return False
            node = node.children[c]
        return node.end
```