Python Competitive Programming Notebook

PyCPBook Community

October 8, 2025

Abstract

This document is a reference notebook for competitive programming in Python. It contains a collection of curated algorithms and data structures, complete with explanations and optimized, copy-pasteable code.

Contents

Chapter 1

Fundamentals

Binary Search

Implements the classic binary search algorithm to find the index of a specific target value within a sorted array. Binary search is a highly efficient search algorithm that works by repeatedly dividing the search interval in half. This implementation searches for an exact match of a `target` value within a sorted array `arr`. The algorithm maintains a search space as an inclusive range `[low, high]`. In each step, it examines the middle element `arr[mid]`:

- If `arr[mid]` is equal to the `target`, the index `mid` is returned.
- If `arr[mid]` is less than the `target`, the search continues in the right half of the array, by setting `low = mid + 1`.
- If `arr[mid]` is greater than the `target`, the search continues in the left half of the array, by setting `high = mid - 1`.

The loop continues as long as `low <= high`. If the loop terminates without finding the target, it means the target is not present in the array, and the function returns -1. This version is suitable for problems where you need to check for the presence of a specific value and get its index. For problems requiring finding the first element satisfying a condition (lower/upper bound), a different variant of binary search is needed.

```python
def binary_search(arr, target):
    """
    Searches for a target value in a sorted array.

    Args:
        arr (list): A sorted list of elements.
        target: The value to search for.

    Returns:
        int: The index of the target in the array if
        ↪  found, otherwise -1.
    """
    low, high = 0, len(arr) - 1

    while low <= high:
        mid = low + (high - low) // 2
        if arr[mid] < target:
            low = mid + 1
        elif arr[mid] > target:
            high = mid - 1
        else:
            return mid
    return -1
```

Bitwise Operations

Bitwise operations are foundational in competitive programming. They allow constant-time manipulation of integer masks, which represent sets and states. This section explains the core operators and provides utility functions used frequently in problems involving subsets, DP on masks, and low-level tricks.

Operators and meaning:

- `&` bitwise AND: keeps a bit only if it is set in both operands
- `|` bitwise OR: sets a bit if it is set in either operand
- `^` bitwise XOR: sets a bit if it is set in exactly one operand
- `~` bitwise NOT: flips all bits
- `<<` left shift: shifts bits left by k positions, introducing zeros on the right
- `>>` right shift: shifts bits right by k positions

In Python, integers are arbitrary precision and use sign-magnitude with infinite precision logically, so `~x` is `-(x+1)`. For mask manipulations we usually work with non-negative integers and carefully limit ourselves to the lower N bits.

Common patterns:

- Test i-th bit: `(x >> i) & 1`
- Set i-th bit: `x | (1 << i)`
- Unset i-th bit: `x & ~(1 << i)`
- Toggle i-th bit: `x ^ (1 << i)`
- Lowest set bit (lowbit): `x & -x`

Complexities:

- All single-step operations are $O(1)$.
- Loops over bits are $O(B)$ where B is the number of bits visited.
- Kernighan's popcount loop runs in $O(\text{number of set bits})$.

Tips and pitfalls:

- Use `bit_length()` to derive indices quickly. MSB index is `x.bit_length() - 1`.

- Right shifting negative numbers is generally avoided in CP code; stick to non-negative masks.
- For subset DP, iterate submasks of `mask` using the idiom below. The loop visits each submask once, so total work is proportional to the number of submasks.

This module provides:

- Bit manipulations: set, unset, toggle, test
- Queries: `is_power_of_two`, `count_set_bits`, `lowest_set_bit`, `msb_index`, `lsb_index`
- Iterators: `iterate_submasks(mask)`, `iterate_bits(mask)`
- Helper: `next_power_of_two(x)`

Use cases include subset enumeration, DP on bitmasks, and constructing fast checks for properties like power-of-two and bit positions.

```python
def bit_set(x, i):
    return x | (1 << i)


def bit_unset(x, i):
    return x & ~(1 << i)


def bit_toggle(x, i):
    return x ^ (1 << i)


def bit_test(x, i):
    return ((x >> i) & 1) == 1


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


def count_set_bits(x):
    c = 0
    while x:
        x &= x - 1
        c += 1
    return c


def lowest_set_bit(x):
    return x & -x


def msb_index(x):
    return x.bit_length() - 1 if x else -1


def lsb_index(x):
    return (x & -x).bit_length() - 1 if x else -1


def iterate_submasks(mask):
    s = mask
    while s:
        yield s
        s = (s - 1) & mask
    yield 0


def iterate_bits(mask):
    m = mask
    while m:
        lb = m & -m
        yield lb.bit_length() - 1
        m ^= lb


def next_power_of_two(x):
    if x <= 0:
        return 1
    return 1 << (x - 1).bit_length()
```

Greedy Algorithms

This guide explains the greedy problem-solving paradigm, a technique for solving optimization problems by making the locally optimal choice at each stage with the hope of finding a global optimum. For a greedy algorithm to work, the problem must exhibit two key properties:

1. Greedy Choice Property: A globally optimal solution can be arrived at by making a locally optimal choice. In other words, the choice made at the current step, without regard for future choices, can lead to a global solution.
2. Optimal Substructure: An optimal solution to the problem contains within it optimal solutions to subproblems.

The example below, the Activity Selection Problem, is a classic illustration of the greedy method. Given a set of activities each with a start and finish time, the goal is to select the maximum number of non-overlapping activities that can be performed by a single person. The greedy choice is to always select the next activity that finishes earliest among those that do not conflict with the last-chosen activity. This choice maximizes the remaining time for other activities.

```python
def activity_selection(activities):
    """
    Selects the maximum number of non-overlapping activities.

    Args:
        activities (list[tuple[int, int]]): A list of activities, where each
            activity is a tuple (start_time, finish_time).

    Returns:
        int: The maximum number of non-overlapping activities.
    """
    if not activities:
        return 0

    # Sort activities by their finish times in ascending order
    activities.sort(key=lambda x: x[1])

    count = 1
    last_finish_time = activities[0][1]
```

```python

    for i in range(1, len(activities)):
        start_time, finish_time = activities[i]
        if start_time >= last_finish_time:
            count += 1
            last_finish_time = finish_time

    return count

```

Is Checks

Predicate-style checks commonly used in competitive programming, with efficient helpers and concise examples. Includes primality testing optimized for contest ranges with a fallback to Miller-Rabin for larger values, configurable palindrome checking for strings and integers, and a curated set of Python built-ins that are useful for quick validations and transformations. Demonstrates Unicode nuances between `str.isdigit`, `str.isdecimal`, and `str.isnumeric`. fallback is $O(k \cdot (\log n)^2)$

```python
from content.math.miller_rabin import is_prime as mr_is_prime


def is_prime(n):
    if n < 2:
        return False
    small_primes = [2, 3, 5]
    for p in small_primes:
        if n == p:
            return True
        if n % p == 0:
            return False
    if n < 10 ** 9:
        i = 5
        while i * i <= n:
            if n % i == 0 or n % (i + 2) == 0:
                return False
            i += 6
        return True
    return mr_is_prime(n)


def is_palindrome(s, normalize=False, alnum_only=False):
    if not isinstance(s, str):
        s = str(s)
    t = s
    if normalize:
        t = t.casefold()
    if alnum_only:
        t = "".join(ch for ch in t if ch.isalnum())
    return t == t[::-1]


def is_checks_examples():
    prime_small_true = is_prime(97)
    prime_small_false = is_prime(221)
    prime_large_probable = is_prime(1000000007)

    pal_simple_true = is_palindrome("racecar")
    pal_simple_false = is_palindrome("python")
    pal_casefold_true = is_palindrome("AbBa", normalize=True)
    pal_alnum_true = is_palindrome("A man, a plan, a canal: Panama!", normalize=True, alnum_only=True)
    pal_int_true = is_palindrome(12321)

    ch_digit = "2"
    ch_superscript_two = "²"
    ch_roman_twelve = "Ⅻ"
    ch_arabic_indic_two = "٢"

    digits_info = {
        "digit_isdigit": ch_digit.isdigit(),
        "digit_isdecimal": ch_digit.isdecimal(),
        "digit_isnumeric": ch_digit.isnumeric(),
        "sup2_isdigit": ch_superscript_two.isdigit(),
        "sup2_isdecimal": ch_superscript_two.isdecimal(),
        "sup2_isnumeric": ch_superscript_two.isnumeric(),
        "roman_isdigit": ch_roman_twelve.isdigit(),
        "roman_isdecimal": ch_roman_twelve.isdecimal(),
        "roman_isnumeric": ch_roman_twelve.isnumeric(),
        "arabic_isdigit": ch_arabic_indic_two.isdigit(),
        "arabic_isdecimal": ch_arabic_indic_two.isdecimal(),
        "arabic_isnumeric": ch_arabic_indic_two.isnumeric(),
    }

    str_flags = {
        "alpha": "abcXYZ".isalpha(),
        "alnum": "abc123".isalnum(),
        "lower": "hello".islower(),
        "upper": "WORLD".isupper(),
        "space": " \t\n".isspace(),
        "ascii_true": "ASCII".isascii(),
        "ascii_false": "π".isascii(),
    }

    agg_logic = {
        "any_true": any([0, 0, 3]),
        "all_true": all([1, 2, 3]),
        "sum_": sum([1, 2, 3, 4]),
        "min_": min([5, 2, 9]),
        "max_": max([5, 2, 9]),
    }

    conv_num = {
        "abs_": abs(-42),
        "round_": round(3.6),
        "divmod_": divmod(17, 5),
        "pow_mod": pow(2, 10, 1000),
        "ord_A": ord("A"),
        "chr_65": chr(65),
        "bin_": bin(10),
        "oct_": oct(10),
        "hex_": hex(255),
    }

    sorting_iter = {
        "sorted_key": sorted([("a", 3), ("b", 1), ("c", 2)], key=lambda x: x[1]),
        "reversed_list": list(reversed([1, 2, 3])),
        "enumerate_list": list(enumerate(["x", "y"], start=1)),
        "zip_list": list(zip([1, 2, 3], ["a", "b", "c"])),
    }

    return {
        "prime_small_true": prime_small_true,
        "prime_small_false": prime_small_false,
        "prime_large_probable": prime_large_probable,
        "pal_simple_true": pal_simple_true,
        "pal_simple_false": pal_simple_false,
        "pal_casefold_true": pal_casefold_true,
        "pal_alnum_true": pal_alnum_true,
        "pal_int_true": pal_int_true,
        "digits_info": digits_info,
        "str_flags": str_flags,
        "agg_logic": agg_logic,
        "conv_num": conv_num,
        "sorting_iter": sorting_iter,
    }

```

List Operations

This comprehensive guide covers Python lists, one of the most fundamental data structures in competitive programming. Lists are dynamic arrays that can store elements of different types and are highly optimized for various operations. Basic List Operations: Lists support numerous operations for manipulation and querying. Understanding the time complexity of each operation is crucial for competitive programming efficiency.

- `append(x)`: Adds element x to the end of the list. This is an amortized $O(1)$ operation, making it the preferred way to build lists incrementally.
- `pop()` / `pop(i)`: Removes and returns the last element (default) or element at index i. `pop()` is $O(1)$, but `pop(i)` is $O(N)$ as it requires shifting elements. Avoid `pop(0)` for large lists - use `collections.deque` instead.
- `insert(i, x)`: Inserts element x at index i, shifting subsequent elements. This is $O(N)$ due to element shifting, so use sparingly.
- `remove(x)`: Removes the first occurrence of x. This is $O(N)$ as it must search through the list. For frequent removals, consider using sets.
- `extend(iterable)`: Appends all elements from an iterable. More efficient than multiple `append()` calls for adding multiple elements.
- `clear()`: Removes all elements, equivalent to `del lst[:]`.

Searching and Counting Operations:

- `index(x)`: Returns the index of the first occurrence of x. Raises ValueError if not found. Use `try/except` or check `x in lst` first.
- `count(x)`: Returns the number of occurrences of x in the list.
- `x in lst`: Membership test returning True/False. Both `index()` and `count()` must scan the list, so they're $O(N)$.

Sorting and Reversing:

- `sort()`: Sorts the list in-place in $O(N \log N)$ time. Use `sort(reverse=True)` for descending order. For custom sorting, use the `key` parameter.
- `sorted(lst)`: Returns a new sorted list without modifying the original.
- `reverse()`: Reverses the list in-place in $O(N)$ time.
- `lst[::-1]`: Creates a new reversed list using slicing.

List Initialization Patterns:

- `[]`: Empty list initialization.
- `[0] * n`: Creates a list of n zeros. Use with immutable types only.
- `[[0] * m for _ in range(n)]`: Creates a proper 2D list (n×m matrix).
- `[[0] * m] * n`: WRONG! Creates n references to the same list object.

Common Pitfalls:

1. Shallow vs Deep Copy: `lst[:]` and `lst.copy()` create shallow copies. For nested structures, use `copy.deepcopy()`.
2. Default Mutable Arguments: Never use `def func(lst=[])` as the same list object is reused across function calls. Use `def func(lst=None)` instead.
3. List Multiplication with Mutable Objects: `[[0] * 3] * 2` creates a list where all rows reference the same object.

Performance Tips for Competitive Programming:

- Pre-allocate lists when size is known: `lst = [0] * n`
- Use list comprehensions instead of loops for better performance
- For frequent insertions/deletions at the beginning, use `collections.deque`
- When building large lists, `append()` is faster than `insert(0, x)`
- Use `enumerate()` for index-value pairs instead of manual indexing

Memory Considerations:

- Lists grow dynamically, so memory usage can be higher than expected
- Use `del lst[i]` or `lst.pop(i)` to free memory when elements are no longer needed
- For very large datasets, consider using generators or processing in chunks

```python
import copy


def list_operations_examples():
    """
    Demonstrates comprehensive Python list operations and common patterns
```

```python
    used in competitive programming. This function is
    ↪ primarily for inclusion
    in the notebook and is called by the stress test to
    ↪ ensure correctness.
    """

    # === Basic List Creation and Initialization ===
    empty_list = []
    filled_list = [1, 2, 3, 4, 5]
    zeros_list = [0] * 5
    matrix_2d = [[0] * 3 for _ in range(2)]

    # Common mistake - all rows reference the same object
    wrong_matrix = [[0] * 3] * 2
    wrong_matrix[0][0] = 1

    # === Basic Modifications ===
    # append() - O(1) amortized
    lst = [1, 2, 3]
    lst.append(4)
    lst.append(5)

    # extend() - O(k) where k is length of iterable
    lst.extend([6, 7, 8])

    # insert() - O(N)
    lst.insert(0, 0)
    lst.insert(3, 99)

    # remove() - O(N)
    lst.remove(99)

    # pop() operations
    last_element = lst.pop()  # O(1)
    element_at_2 = lst.pop(2)  # O(N)

    # clear()
    temp_list = [1, 2, 3]
    temp_list.clear()

    # === Searching and Counting ===
    search_list = [1, 2, 3, 2, 4, 2, 5]

    # index() - O(N)
    first_two_index = search_list.index(2)

    # count() - O(N)
    count_of_twos = search_list.count(2)

    # membership test - O(N)
    has_three = 3 in search_list
    has_six = 6 in search_list

    # === Sorting and Reversing ===
    sort_list = [3, 1, 4, 1, 5, 9, 2, 6]

    # sort() - in-place, O(N log N)
    sort_list.sort()
    sorted_asc = sort_list.copy()

    # sort(reverse=True)
    sort_list.sort(reverse=True)
    sorted_desc = sort_list.copy()

    # sorted() - returns new list
    original = [3, 1, 4, 1, 5]
    sorted_new = sorted(original)

    # reverse() - in-place, O(N)
    reverse_list = [1, 2, 3, 4, 5]
    reverse_list.reverse()

    # slicing for reversal - creates new list
    reverse_slice = [1, 2, 3, 4, 5][::-1]

    # === Slicing Operations ===
    slice_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    # Basic slicing
    first_half = slice_list[:5]
    second_half = slice_list[5:]
    middle = slice_list[2:7]

    # Step slicing
    even_indices = slice_list[::2]
    odd_indices = slice_list[1::2]
    reversed_slice = slice_list[::-1]

    # Negative indexing
    last_element = slice_list[-1]
    last_three = slice_list[-3:]

    # === Copying Operations ===
    original_copy = [1, 2, [3, 4]]

    # Shallow copy methods
    shallow_copy1 = original_copy[:]
    shallow_copy2 = original_copy.copy()

    # Deep copy
    deep_copy = copy.deepcopy(original_copy)

    # Demonstrate shallow vs deep copy
    original_copy[2][0] = 99

    # === List Comprehensions ===
    squares = [x * x for x in range(5)]
    even_squares = [x * x for x in range(10) if x % 2 ==
    ↪ 0]
    matrix_flatten = [x for row in [[1, 2], [3, 4]] for x
    ↪ in row]

    # === Advanced Operations ===
    # enumerate() for index-value pairs
    enumerated = list(enumerate(['a', 'b', 'c']))

    # zip() for pairing lists
    zipped = list(zip([1, 2, 3], ['a', 'b', 'c']))

    # max, min, sum
    numbers = [1, 5, 3, 9, 2]
    max_val = max(numbers)
    min_val = min(numbers)
    sum_val = sum(numbers)

    return {
        # Initialization results
        "empty_list": empty_list,
        "filled_list": filled_list,
        "zeros_list": zeros_list,
        "matrix_2d": matrix_2d,
        "wrong_matrix": wrong_matrix,

        # Basic modifications
        "lst_after_operations": lst,
        "last_element": last_element,
        "element_at_2": element_at_2,
        "temp_list_cleared": temp_list,

        # Searching results
        "first_two_index": first_two_index,
        "count_of_twos": count_of_twos,
        "has_three": has_three,
        "has_six": has_six,

        # Sorting results
        "sorted_asc": sorted_asc,
        "sorted_desc": sorted_desc,
        "sorted_new": sorted_new,
        "original_unchanged": original,
        "reverse_list": reverse_list,
        "reverse_slice": reverse_slice,

        # Slicing results
        "first_half": first_half,
        "second_half": second_half,
```

```python
        "middle": middle,
        "even_indices": even_indices,
        "odd_indices": odd_indices,
        "reversed_slice": reversed_slice,
        "last_element_slice": last_element,
        "last_three": last_three,

        # Copying results
        "shallow_copy1": shallow_copy1,
        "shallow_copy2": shallow_copy2,
        "deep_copy": deep_copy,
        "original_copy_modified": original_copy,

        # List comprehensions
        "squares": squares,
        "even_squares": even_squares,
        "matrix_flatten": matrix_flatten,

        # Advanced operations
        "enumerated": enumerated,
        "zipped": zipped,
        "max_val": max_val,
        "min_val": min_val,
        "sum_val": sum_val,
    }

```

Prefix Sums

Implements 1D and 2D prefix sum arrays for fast range sum queries. Prefix sums (also known as summed-area tables in 2D) allow for the sum of any contiguous sub-array or sub-rectangle to be calculated in constant time after an initial linear-time precomputation. 1D Prefix Sums: Given an array `A`, its prefix sum array `P` is defined such that `P[i]` is the sum of all elements from `A[0]` to `A[i-1]`. The sum of a range `[l, r-1]` can then be calculated in $O(1)$ as `P[r] - P[l]`. 2D Prefix Sums: This extends the concept to a 2D grid. The prefix sum `P[i][j]` stores the sum of the rectangle from `(0, 0)` to `(i-1, j-1)`. The sum of an arbitrary rectangle defined by its top-left corner `(r1, c1)` and bottom-right corner `(r2-1, c2-1)` is calculated using the principle of inclusion-exclusion: `sum = P[r2][c2] - P[r1][c2] - P[r2][c1] + P[r1][c1]`.

- 1D: $O(N)$ for precomputation, $O(1)$ for each range query.
- 2D: $O(R \cdot C)$ for precomputation, $O(1)$ for each range query.
- 1D: $O(N)$ to store the prefix sum array.
- 2D: $O(R \cdot C)$ to store the prefix sum grid.

```python
def build_prefix_sum_1d(arr):
    """
    Builds a 1D prefix sum array and returns a query function.

    Args:
        arr (list[int]): The input 1D array.

    Returns:
        function: A function `query(left, right)` that returns the sum of
                  the elements in the range [left, right-1] in O(1).
    """
    n = len(arr)
    prefix_sum = [0] * (n + 1)
    for i in range(n):
        prefix_sum[i + 1] = prefix_sum[i] + arr[i]

    def query(left, right):
        """
        Queries the sum of the range [left, right-1].
        `left` is inclusive, `right` is exclusive.
        """
        if not (0 <= left <= right <= n):
            return 0
        return prefix_sum[right] - prefix_sum[left]

    return query


def build_prefix_sum_2d(grid):
    """
    Builds a 2D prefix sum array and returns a query function.

    Args:
        grid (list[list[int]]): The input 2D grid.

    Returns:
        function: A function `query(r1, c1, r2, c2)` that returns the sum of
                  the elements in the rectangle from (r1, c1) to (r2-1, c2-1) in O(1).
    """
    if not grid or not grid[0]:
        return lambda r1, c1, r2, c2: 0

    rows, cols = len(grid), len(grid[0])
    prefix_sum = [[0] * (cols + 1) for _ in range(rows + 1)]

    for r in range(rows):
        for c in range(cols):
            prefix_sum[r + 1][c + 1] = (
                grid[r][c]
                + prefix_sum[r][c + 1]
                + prefix_sum[r + 1][c]
                - prefix_sum[r][c]
            )

    def query(r1, c1, r2, c2):
        """
        Queries the sum of the rectangle from (r1, c1) to (r2-1, c2-1).
        `r1, c1` are inclusive top-left coordinates.
        `r2, c2` are exclusive bottom-right coordinates.
        """
        if not (0 <= r1 <= r2 <= rows and 0 <= c1 <= c2 <= cols):
            return 0
        return (
            prefix_sum[r2][c2]
            - prefix_sum[r1][c2]
            - prefix_sum[r2][c1]
            + prefix_sum[r1][c1]
        )

    return query
```

Python Idioms

This section provides a reference for common and powerful Python idioms that are particularly useful in competitive programming for writing concise, efficient, and readable code. List, Set, and Dictionary Comprehensions: A concise way to create lists, sets, and dictionaries. The syntax is `[expression for item in iterable if condition]`. This is often faster and more readable than using explicit `for` loops with `.append()`. Advanced Sorting: Python's `sorted()` function and the `.sort()` list method are highly optimized. They can be customized using a `key` argument, which is typically a `lambda` function. This allows for sorting complex objects based on specific attributes or computed values without writing a full comparison function. String Manipulations:

- Slicing: Python's slicing `s[start:stop:step]` is a powerful tool for substrings and reversing. `s[::-1]` reverses a string in $O(N)$ time.
- `split()` and `join()`: These methods are the standard way to parse space-separated input and format list-based output. `line.split()` handles various whitespace, and `' '.join(map(str, my_list))` is a common output pattern.

Character and Number Conversions:

- `ord(c)`: Returns the ASCII/Unicode integer value of a single character `c`. For example, `ord('a')` is 97. This is useful for character arithmetic, like `ord(char) - ord('a')` to get a 0-indexed alphabet position.
- `chr(i)`: The inverse of `ord()`. Returns the character for an integer ASCII value `i`. For example, `chr(97)` is `'a'`.
- `int(s)` and `str(i)`: Standard functions to convert strings to integers and integers to strings, respectively.

```python
def python_idioms_examples():
    """
    Demonstrates various Python idioms useful in competitive programming.
    This function is primarily for inclusion in the notebook and is called
    by the stress test to ensure correctness.
    """
    # List Comprehensions
    squares = [x * x for x in range(5)]
    even_squares = [x * x for x in range(10) if x % 2 == 0]

    # Set and Dictionary Comprehensions
    unique_squares = {x * x for x in [-1, 1, -2, 2]}
    square_map = {x: x * x for x in range(5)}

    # Advanced Sorting
    pairs = [(1, 5), (3, 2), (2, 8)]
    sorted_by_second = sorted(pairs, key=lambda p: p[1])

    # String Manipulations
    sentence = "this is a sentence"
    words = sentence.split()
    rejoined = "-".join(words)
    reversed_sentence = sentence[::-1]

    # Character and Number Conversions
    char_a = "a"
    ord_a = ord(char_a)
    chr_97 = chr(97)
    num_str = "123"
    num_int = int(num_str)
    back_to_str = str(num_int)

    # The function can return the values to be checked by a test script.
    return {
        "squares": squares,
        "even_squares": even_squares,
        "unique_squares": unique_squares,
        "square_map": square_map,
        "sorted_by_second": sorted_by_second,
        "words": words,
        "rejoined": rejoined,
        "reversed_sentence": reversed_sentence,
        "ord_a": ord_a,
        "chr_97": chr_97,
        "num_int": num_int,
        "back_to_str": back_to_str,
    }

```

Recursion Backtracking

This guide provides a template and explanation for recursion and backtracking. Backtracking is a general algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, and removing those solutions ("backtracking") that fail to satisfy the constraints of the problem at any point in time. The core of backtracking is a recursive function that follows a "choose, explore, unchoose" pattern:

1. **Choose**: Make a choice at the current state. This could be including an element in a subset, placing a queen on a chessboard, or moving to a new cell in a maze.
2. **Explore**: Recursively call the function to explore further possibilities that arise from the choice made.
3. **Unchoose**: After the recursive call returns, undo the choice made in step 1. This is the "backtracking" step. It allows the algorithm to explore other paths from the current state.

The example below, "generating all subsets," demonstrates this pattern perfectly. To generate all subsets of a set of numbers, we can iterate through the numbers. For each number, we have two choices: include it in the current subset, or not include it. The backtracking function explores both paths. takes up to $O(N)$ time to create a copy to add to the results list. subset. The output list itself requires $O(N \cdot 2^N)$ space.

```python
def generate_subsets(nums):
    """
    Generates all possible subsets (the power set) of a list of numbers.

    Args:
```

```python
        nums (list[int]): A list of numbers.

    Returns:
        list[list[int]]: A list containing all subsets of nums.
    """
    result = []
    current_subset = []

    def backtrack(start_index):
        # Add the current subset configuration to the result list.
        # A copy is made because current_subset will be modified.
        result.append(list(current_subset))

        # Explore further choices.
        for i in range(start_index, len(nums)):
            # 1. Choose: Include the number nums[i] in the current subset.
            current_subset.append(nums[i])

            # 2. Explore: Recursively call with the next index.
            backtrack(i + 1)

            # 3. Unchoose: Remove nums[i] to backtrack and explore other paths.
            current_subset.pop()

    backtrack(0)
    return result

```

Stacks And Queues

This guide explains how to implement and use stacks and queues, two of the most fundamental linear data structures in computer science, using Python's built-in features. Stack (LIFO - Last-In, First-Out): A stack is a data structure that follows the LIFO principle. The last element added to the stack is the first one to be removed. Think of it like a stack of plates: you add a new plate to the top and also remove a plate from the top. In Python, a standard `list` can be used as a stack.

- `append()`: Pushes a new element onto the top of the stack. This is an amortized $O(1)$ operation.
- `pop()`: Removes and returns the top element of the stack. This is an $O(1)$ operation.

Queue (FIFO - First-In, First-Out): A queue is a data structure that follows the FIFO principle. The first element added to the queue is the first one to be removed, like a checkout line at a store. While a Python `list` can be used as a queue with `append()` and `pop(0)`, this is inefficient because `pop(0)` takes $O(N)$ time, as all subsequent elements must be shifted. The correct and efficient way to implement a queue is using `collections.deque` (double-ended queue).

- `append()`: Adds an element to the right end (back) of the queue in $O(1)$.
- `popleft()`: Removes and returns the element from the left end (front) of the queue in $O(1)$.

`deque` is highly optimized for appends and pops from both ends.

```python
from collections import deque


def stack_and_queue_examples():
    """
    Demonstrates the usage of stacks (with lists) and queues (with deque).
    This function is primarily for inclusion in the notebook and is called
    by the stress test to ensure correctness.
    """
    # --- Stack Example (LIFO) ---
    stack = []
    stack.append(10)  # Stack: [10]
    stack.append(20)  # Stack: [10, 20]
    stack.append(30)  # Stack: [10, 20, 30]

    popped_from_stack = []
    popped_from_stack.append(stack.pop())  # Returns 30, Stack: [10, 20]
    popped_from_stack.append(stack.pop())  # Returns 20, Stack: [10]

    # --- Queue Example (FIFO) ---
    queue = deque()
    queue.append(10)  # Queue: deque([10])
    queue.append(20)  # Queue: deque([10, 20])
    queue.append(30)  # Queue: deque([10, 20, 30])

    popped_from_queue = []
    popped_from_queue.append(queue.popleft())  # Returns 10, Queue: deque([20, 30])
    popped_from_queue.append(queue.popleft())  # Returns 20, Queue: deque([30])

    return {
        "final_stack": stack,
        "popped_from_stack": popped_from_stack,
        "final_queue": list(queue),  # Convert to list for easy comparison
        "popped_from_queue": popped_from_queue,
    }

```

Two Pointers

This guide explains the Two Pointers and Sliding Window techniques, which are powerful for solving array and string problems efficiently. Two Pointers: The two-pointers technique involves using two pointers to traverse a data structure, often an array or string, in a coordinated way. The pointers can move in various patterns:

1. Converging Pointers: One pointer starts at the beginning and the other at the end. They move towards each other until they meet or cross. This is common for problems on sorted arrays, like finding a pair with a specific sum.
2. Same-Direction Pointers (Sliding Window): Both pointers start at or near the beginning and move in the same direction. One pointer

(`right`) expands a "window," and the other (`left`) contracts it.

Sliding Window: This is a specific application of the two-pointers technique. A "window" is a subsegment of the data (e.g., a subarray or substring) represented by the indices `[left, right]`. The `right` pointer expands the window, and the `left` pointer contracts it, typically to maintain a certain property or invariant within the window. This avoids the re-computation that plagues naive O(N^2) solutions by only adding/removing one element at a time. The example below, "Longest Substring with At Most K Distinct Characters," is a classic sliding window problem. The window `s[left:right+1]` is expanded by incrementing `right`. If the number of distinct characters in the window exceeds `k`, the window is contracted from the left by incrementing `left` until the condition is met again. pointer traverses the data structure at most once. or Σ is the size of the character set, to store the elements in the window.

```python
from collections import defaultdict


def longest_substring_with_k_distinct(s, k):
    """
    Finds the length of the longest substring of s that contains at most k
    distinct characters.

    Args:
        s (str): The input string.
        k (int): The maximum number of distinct characters allowed.

    Returns:
        int: The length of the longest valid substring.
    """
    if k == 0:
        return 0

    n = len(s)
    left = 0
    max_len = 0
    char_counts = defaultdict(int)

    for right in range(n):
        char_counts[s[right]] += 1

        while len(char_counts) > k:
            char_left = s[left]
            char_counts[char_left] -= 1
            if char_counts[char_left] == 0:
                del char_counts[char_left]
            left += 1

        max_len = max(max_len, right - left + 1)

    return max_len
```

Chapter 2

Standard Library

Bisect Library

This guide explains how to use Python's `bisect` module to efficiently search for elements and maintain the sorted order of a list. The module provides functions for binary searching, which is significantly faster than a linear scan for large lists. The `bisect` module is particularly useful for finding insertion points for new elements while keeping a list sorted, without having to re-sort the entire list after each insertion. Key functions:

- `bisect.bisect_left(a, x)`: Returns an insertion point which comes before (to the left of) any existing entries of `x` in `a`. This is equivalent to finding the index of the first element greater than or equal to `x`.
- `bisect.bisect_right(a, x)`: Returns an insertion point which comes after (to the right of) any existing entries of `x` in `a`. This is equivalent to finding the index of the first element strictly greater than `x`.
- `bisect.insort_left(a, x)`: Inserts `x` into `a` in sorted order. This is efficient for finding the position, but the insertion itself can be slow ($O(N)$) as it requires shifting elements.

These functions are fundamental for problems that require maintaining a sorted collection or performing searches like "count elements less than x" or "find the first element satisfying a condition." due to the list insertion.

```
import bisect


def bisect_examples():
    """
    Demonstrates the usage of the bisect module.
    This function is primarily for inclusion in the
    ↪  notebook and is called
    by the stress test to ensure correctness.
    """
    data = [10, 20, 20, 30, 40]

    # --- bisect_left ---
    # Find insertion point for 20 (before existing 20s)
    idx_left_20 = bisect.bisect_left(data, 20)
    # Find insertion point for 25 (between 20 and 30)
    idx_left_25 = bisect.bisect_left(data, 25)

    # --- bisect_right ---
    # Find insertion point for 20 (after existing 20s)
    idx_right_20 = bisect.bisect_right(data, 20)
    # Find insertion point for 25 (same as bisect_left)
    idx_right_25 = bisect.bisect_right(data, 25)

    # --- insort ---
    # insort_left inserts at the position found by
    ↪  bisect_left
    data_for_insort = [10, 20, 20, 30, 40]
    bisect.insort_left(data_for_insort, 25)

    return {
        "idx_left_20": idx_left_20,
        "idx_left_25": idx_left_25,
        "idx_right_20": idx_right_20,
        "idx_right_25": idx_right_25,
        "list_after_insort": data_for_insort,
    }
```

Collections Library

This guide covers essential data structures from Python's `collections` module that are extremely useful in competitive programming: `deque`, `Counter`, and `defaultdict`. `collections.deque`: A double-ended queue that supports adding and removing elements from both ends in $O(1)$ time. This makes it a highly efficient implementation for both queues (using `append` and `popleft`) and stacks (using `append` and `pop`). It is generally preferred over a `list` for queue operations because `list.pop(0)` is an $O(N)$ operation. `collections.Counter`: A specialized dictionary subclass for counting hashable objects. It's a convenient way to tally frequencies of elements in a list or characters in a string. It supports common operations like initialization from an iterable, accessing counts (which defaults to 0 for missing items), and arithmetic operations for combining counters. `collections.defaultdict`: A dictionary subclass that calls a factory function to supply missing values. When a key is accessed for the first time, it is not present in the dictionary, so the factory function is called to create a default value for that key. This is useful for avoiding `KeyError` checks when, for example, building an adjacency list (`defaultdict(list)`) or counting items (`defaultdict(int)`). element access and update for `Counter` and `defaultdict`) are amortized $O(1)$.

```
from collections import deque, Counter, defaultdict


def collections_examples():
    """
    Demonstrates the usage of deque, Counter, and
    ↪  defaultdict.
    This function is primarily for inclusion in the
    ↪  notebook and is called
    by the stress test to ensure correctness.
    """
    # --- deque ---
```

```python
    q = deque([1, 2, 3])
    q.append(4)
    q.appendleft(0)
    q_pop_left = q.popleft()
    q_pop_right = q.pop()

    # --- Counter ---
    data = ["a", "b", "c", "a", "b", "a"]
    counts = Counter(data)
    count_of_a = counts["a"]
    count_of_d = counts["d"]

    # --- defaultdict ---
    adj = defaultdict(list)
    edges = [(0, 1), (0, 2), (1, 2)]
    for u, v in edges:
        adj[u].append(v)
        adj[v].append(u)

    # Access a missing key to trigger the default factory
    missing_key_val = adj[5]

    return {
        "final_deque": list(q),
        "q_pop_left": q_pop_left,
        "q_pop_right": q_pop_right,
        "counter_a": count_of_a,
        "counter_d": count_of_d,
        "adj_list": dict(adj),
        "adj_list_missing": missing_key_val,
    }

```

Functools Library

This guide explains how to use `@functools.cache` for transparently adding memoization to a function. Memoization is an optimization technique where the results of expensive function calls are stored and returned for the same inputs, avoiding redundant computation. `@functools.cache`: This decorator wraps a function with a memoizing callable that saves up to the `maxsize` most recent calls. Because it's a hash-based cache, all arguments to the function must be hashable. In competitive programming, this is extremely powerful for simplifying dynamic programming problems that have a natural recursive structure. A recursive solution that would normally be too slow due to recomputing the same subproblems can become efficient by simply adding the `@cache` decorator. The example below demonstrates this with the Fibonacci sequence. The naive recursive solution has an exponential time complexity, $O(2^N)$. With `@cache`, each state `fib(n)` is computed only once, reducing the complexity to linear, $O(N)$, the same as a standard iterative DP solution. unique states it's called with, rather than the total number of calls. in the cache.

```python
import functools


@functools.cache
def fibonacci_with_cache(n):
    """
    Computes the n-th Fibonacci number using recursion
    ↪ with memoization.
    This function is primarily for demonstrating
    ↪ @functools.cache.
    """
    if n < 2:
        return n
    return fibonacci_with_cache(n - 1) +
    ↪ fibonacci_with_cache(n - 2)

```

Heapq Library

This guide explains how to use Python's `heapq` module to implement a min-priority queue. A heap is a specialized tree-based data structure that satisfies the heap property. In a min-heap, for any given node `C`, if `P` is a parent of `C`, then the key of `P` is less than or equal to the key of `C`. This means the smallest element is always at the root of the tree. The `heapq` module provides an efficient implementation of the min-heap algorithm. It operates directly on a standard Python `list`, which is a key aspect of its design. Key functions:

- `heapq.heappush(heap, item)`: Pushes an `item` onto the `heap` (a list), maintaining the heap property. This operation is $O(\log N)$.
- `heapq.heappop(heap)`: Pops and returns the smallest `item` from the `heap`, maintaining the heap property. This is also $O(\log N)$.
- `heapq.heapify(x)`: Transforms a list `x` into a heap, in-place, in $O(N)$ time.

Since `heapq` implements a min-heap, the element at index 0 (`heap[0]`) is always the smallest. To implement a max-heap, a common trick is to store the negative of the values (or use a custom wrapper class).

```python
import heapq


def heapq_examples():
    """
    Demonstrates the usage of the heapq module.
    This function is primarily for inclusion in the
    ↪ notebook and is called
    by the stress test to ensure correctness.
    """
    # --- heappush and heappop ---
    min_heap = []
    heapq.heappush(min_heap, 4)
    heapq.heappush(min_heap, 1)
    heapq.heappush(min_heap, 7)

    # After pushes, the heap (list) is [1, 4, 7]
    # The smallest element is at index 0
    smallest_element = min_heap[0]

    popped_elements = []
    popped_elements.append(heapq.heappop(min_heap))  #
    ↪ Pops 1
    popped_elements.append(heapq.heappop(min_heap))  #
    ↪ Pops 4

    # --- heapify ---
    data_list = [5, 8, 2, 9, 1, 4]
    heapq.heapify(data_list)
    # After heapify, data_list is now [1, 4, 2, 9, 8, 5]
    ↪ (or similar,
```

```python
    # it only guarantees the heap property, not a fully sorted list)
    heapified_list = list(data_list)
    smallest_after_heapify = data_list[0]

    return {
        "smallest_element": smallest_element,
        "final_heap": min_heap,
        "popped_elements": popped_elements,
        "heapified_list": heapified_list,
        "smallest_after_heapify": smallest_after_heapify,
    }

```

Itertools Library

This guide showcases powerful combinatorial iterators from Python's `itertools` module. These functions are highly optimized and provide a clean, efficient way to handle tasks involving permutations, combinations, and Cartesian products, which are common in competitive programming problems. `itertools.permutations(iterable, r=None)`: Returns successive r-length permutations of elements from the iterable. If r is not specified or is None, then r defaults to the length of the iterable, and all possible full-length permutations are generated. The elements are treated as unique based on their position, not their value. `itertools.combinations(iterable, r)`: Returns r-length subsequences of elements from the input iterable. The combination tuples are emitted in lexicographic ordering according to the order of the input iterable. Elements are treated as unique based on their position, not their value. `itertools.product(*iterables, repeat=1)`: Computes the Cartesian product of input iterables. It is equivalent to nested for-loops. For example, `product(A, B)` returns the same as `((x,y) for x in A for y in B)`. These functions are implemented in C, making them significantly faster than equivalent Python-based recursive or iterative solutions. length N, `permutations` returns $P(N, r)$ items, `combinations` returns $C(N, r)$ items, and `product` returns N^k items for k iterables.

```python
import itertools


def itertools_examples():
    """
    Demonstrates the usage of common itertools functions.
    This function is primarily for inclusion in the notebook and is called
    by the stress test to ensure correctness.
    """
    elements = ["A", "B", "C"]

    # --- Permutations ---
    # All full-length permutations of elements
    perms_full = list(itertools.permutations(elements))
    # All 2-element permutations of elements
    perms_partial = list(itertools.permutations(elements, 2))

    # --- Combinations ---
    # All 2-element combinations of elements
    combs = list(itertools.combinations(elements, 2))

    # --- Cartesian Product ---
    pool1 = ["x", "y"]
    pool2 = [1, 2]
    prod = list(itertools.product(pool1, pool2))

    return {
        "perms_full": perms_full,
        "perms_partial": perms_partial,
        "combs": combs,
        "prod": prod,
    }

```

Math Library

This guide highlights essential functions from Python's `math` module that are frequently used in competitive programming. These functions provide standard mathematical operations and constants. Key functions and constants:

- `math.gcd(a, b)`: Computes the greatest common divisor of two integers.
- `math.ceil(x)`: Returns the smallest integer greater than or equal to `x`.
- `math.floor(x)`: Returns the largest integer less than or equal to `x`.
- `math.sqrt(x)`: Returns the floating-point square root of `x`.
- `math.isqrt(x)`: Returns the integer square root of a non-negative integer `x`, which is `floor(sqrt(x))`. This is often faster and more precise for integer-only contexts.
- `math.log2(x)`: Returns the base-2 logarithm of `x`.
- `math.inf`: A floating-point representation of positive infinity. Useful for initializing minimum/maximum values.

These tools are fundamental for a wide range of problems, from number theory to geometry, providing a reliable and efficient standard library implementation. The rest are typically $O(1)$.

```python
import math


def math_examples():
    """
    Demonstrates the usage of common math module functions.
    This function is primarily for inclusion in the notebook and is called
    by the stress test to ensure correctness.
    """
    # Greatest Common Divisor
    gcd_val = math.gcd(54, 24)

    # Ceiling and Floor
    ceil_val = math.ceil(4.2)
```

```python
    floor_val = math.floor(4.8)

    # Square Roots
    sqrt_val = math.sqrt(25)
    isqrt_val = math.isqrt(26)

    # Logarithm
    log2_val = math.log2(16)

    # Infinity constant
    infinity = math.inf

    return {
        "gcd_val": gcd_val,
        "ceil_val": ceil_val,
        "floor_val": floor_val,
        "sqrt_val": sqrt_val,
        "isqrt_val": isqrt_val,
        "log2_val": log2_val,
        "infinity": infinity,
    }

```

Chapter 3

Contest & Setup

Debugging Tricks

This section outlines common debugging techniques and tricks useful in a competitive programming context. Since standard debuggers are often unavailable or too slow on online judges, these methods are invaluable.

1. Debug Printing to stderr:

- The most common technique is to print variable states at different points in the code.
- Always print to standard error (`sys.stderr`) instead of standard output (`sys.stdout`). The online judge ignores `stderr`, so your debug messages won't interfere with the actual output and cause a "Wrong Answer" verdict.
- Example: `print(f"DEBUG: Current value of x is {x}", file=sys.stderr)`

1. Test with Edge Cases:

- Before submitting, always test your code with edge cases.
- Minimum constraints: e.g., N=0, N=1, empty list.
- Maximum constraints: e.g., N=10^5. (Check for TLE - Time Limit Exceeded).
- Special values: e.g., zeros, negative numbers, duplicates.
- A single off-by-one error can often be caught by testing N=1 or N=2.

1. Assertions:

- Use `assert` to check for invariants in your code. An invariant is a condition that should always be true at a certain point.
- For example, if a variable `idx` should always be non-negative, you can add `assert idx >= 0`.
- If the assertion fails, your program will crash with an `AssertionError`, immediately showing you where the logic went wrong.
- Assertions are automatically disabled in Python's optimized mode (`python -O`), so they have no performance penalty on the judge if it runs in that mode.

1. Naive Solution Comparison:

- If you have a complex, optimized algorithm, write a simple, brute-force (naive) solution that is obviously correct but slow.
- Generate a large number of small, random test cases.
- Run both your optimized solution and the naive solution on each test case and assert that their outputs are identical.
- If they differ, print the failing test case. This is the core idea behind the stress tests used in this project.

1. Rubber Duck Debugging:

- Explain your code, line by line, to someone else or even an inanimate object (like a rubber duck).
- The act of verbalizing your logic often helps you spot the flaw yourself.

Time: N/A Space: N/A Status: Not applicable (Informational)

```python
import sys


def example_debug_print():
    """
    A simple example demonstrating how to print debug
    ↪ information
    to stderr without affecting the program's actual
    ↪ output.
    """
    data = [10, 20, 30]

    # This is the actual output that the judge will see.
    print("Processing started.")

    total = 0
    for i, item in enumerate(data):
        # This is a debug message. It goes to stderr and
        ↪ is ignored by the judge.
        print(f"DEBUG: Processing item {i} with value
        ↪ {item}", file=sys.stderr)
        total += item

    # This is the final output.
    print(f"The final total is: {total}")

```

Fast Io

This guide provides a comprehensive overview of fast I/O techniques in Python for competitive programming. Standard `input()` can be too slow for problems with large inputs, leading to Time Limit Exceeded (TLE) verdicts. Using the `sys` module provides a much faster alternative.

1. The `sys.stdin` Object:

- `sys.stdin` is a file object representing standard input. You can read from it like you would from a file. This is more efficient than the built-in `input()` function, which performs extra processing on each line.

1. Reading a Single Line: `sys.stdin.readline()`

- This is the most common replacement for `input()`. It reads one line from standard input, including the trailing newline character (\\n).
- You almost always need to strip this newline using `.strip()`.
- Example: `line = sys.stdin.readline().strip()`

1. Reading All Lines: `sys.stdin.readlines()`

- This function reads all lines from standard input until EOF (End-of-File) and returns them as a list of strings.
- This is useful when the entire input fits into memory and can be processed at once. Each string in the list retains its trailing newline.

1. Reading the Entire Stream: `sys.stdin.read()`

- This function reads the entire input stream until EOF and returns it as a single string. This can be useful for problems with non-line-based input.

1. Iterating over `sys.stdin`:

- Since `sys.stdin` is an iterator, you can loop over it directly. This is an elegant way to process input line by line when the number of lines is not given beforehand.
- Example: `for line in sys.stdin: process(line.strip())`

1. Using `next(sys.stdin)`:

- This allows you to consume lines from the iterator one at a time, which can be cleaner than mixing `readline()` with a `for` loop.
- Example: `n = int(next(sys.stdin))`
- Example: `data = [int(x) for x in next(sys.stdin).split()]`

Common Parsing Patterns: The functions below demonstrate how to wrap these techniques into convenient helpers, similar to those in `template.py`. Time: N/A Space: N/A Status: Not applicable (Informational)

```python
import sys


def get_ints_from_line():
    """
    Reads a line of space-separated values and parses them into a list of integers.
    This is a common helper function.
    """
    return list(map(int, sys.stdin.readline().split()))


def process_all_lines():
    """
    Demonstrates reading all lines at once and processing them.
    This example calculates the sum of the first integer on each line.
    """
    lines = sys.stdin.readlines()
    total = 0
    for line in lines:
        if line.strip():
            total += int(line.split()[0])
    return total


def iterate_over_stdin():
    """
    Demonstrates processing input by iterating over sys.stdin, which reads
    line by line until EOF.
    """
    total = 0
    for line in sys.stdin:
        if line.strip():
            total += sum(map(int, line.split()))
    return total


def demonstrate_usage_conceptual():
    """
    This function is for inclusion in the notebook as a clear example and
    is not meant to be executed directly as part of a test. It shows a common
    contest pattern: reading a count `N`, followed by `N` lines of data.
    """
    try:
        # Read N, the number of lines to follow
        n_str = sys.stdin.readline()
        if not n_str:
            return
        n = int(n_str)

        # Read N lines into a matrix
        matrix = []
        for _ in range(n):
            row = list(map(int, sys.stdin.readline().split()))
            matrix.append(row)

        # In a real problem, you would process the matrix here.
        # For demonstration, we just show it was read.
        # print("Matrix received:", matrix)
```

```python

    except (IOError, ValueError):
        # Handle potential empty input or parsing errors gracefully.
        pass

```

Template

A standard template for Python in programming contests. It provides fast I/O, an increased recursion limit, and common helper functions to accelerate development under time constraints.

Fast I/O: This template redefines `input` to use `sys.stdin.readline()` for performance. For a detailed guide on various fast I/O patterns and their usage, please refer to the "Fast I/O" section in this chapter.

Recursion Limit: Python's default recursion limit (often 1000) is too low for problems that involve deep recursion. `sys.setrecursionlimit(10**6)` increases this limit to avoid `RecursionError` on large test cases.

Usage: Place problem-solving logic inside the `solve()` function. The main execution block is set up to call this function, with a commented-out loop for handling multiple test cases. Time: N/A Space: N/A Status: Not applicable (Utility)

```python
import sys
import math
import os

sys.setrecursionlimit(10**6)

input = sys.stdin.readline


def get_int():
    """Reads a single integer from a line."""
    return int(input())


def get_ints():
    """Reads a list of space-separated integers from a line."""
    return list(map(int, input().split()))


def get_str():
    """Reads a single string from a line, stripping trailing whitespace."""
    return input().strip()


def get_strs():
    """Reads a list of space-separated strings from a line."""
    return input().strip().split()


def solve():
    """
    This is the main function where the solution logic for a single
    test case should be implemented.
    """
    try:
        n, m = get_ints()
        print(n + m)
    except (IOError, ValueError):
        pass


def main():
    """
    Main execution function.
    Handles multiple test cases if required.
    """
    # t = get_int()
    # for _ in range(t):
    #     solve()
    solve()


if __name__ == "__main__":
    main()

```

Chapter 4

Data Structures

Binary Search Tree

Implements a standard, unbalanced Binary Search Tree (BST). A BST is a rooted binary tree data structure whose internal nodes each store a key greater than all keys in the node's left subtree and less than those in its right subtree. This data structure provides efficient average-case time complexity for search, insert, and delete operations. However, its primary drawback is that these operations can degrade to $O(N)$ in the worst case if the tree becomes unbalanced (e.g., when inserting elements in sorted order, the tree becomes a linked list). This implementation serves as a foundational example and a good contrast to the balanced BSTs (like Treaps) also included in this notebook, which guarantee $O(\log N)$ performance. The `delete` operation handles the three standard cases:

1. The node to be deleted is a leaf (no children).
2. The node has one child.
3. The node has two children (in which case it's replaced by its in-order successor).

Average case for `search`, `insert`, `delete` is $O(\log N)$. Worst case is $O(N)$. Space complexity is $O(N)$ to store the nodes of the tree.

```python
class Node:
    """Represents a single node in the Binary Search Tree."""

    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


class BinarySearchTree:
    """A standard (unbalanced) Binary Search Tree implementation."""

    def __init__(self):
        self.root = None

    def search(self, key):
        """Searches for a key in the BST."""
        return self._search_recursive(self.root, key) is not None

    def _search_recursive(self, node, key):
        if node is None or node.key == key:
            return node
        if key < node.key:
            return self._search_recursive(node.left, key)
        return self._search_recursive(node.right, key)

    def insert(self, key):
        """Inserts a key into the BST."""
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = Node(key)
            else:
                self._insert_recursive(node.left, key)
        elif key > node.key:
            if node.right is None:
                node.right = Node(key)
            else:
                self._insert_recursive(node.right, key)

    def delete(self, key):
        """Deletes a key from the BST."""
        self.root = self._delete_recursive(self.root, key)

    def _delete_recursive(self, node, key):
        if node is None:
            return node

        if key < node.key:
            node.left = self._delete_recursive(node.left, key)
        elif key > node.key:
            node.right = self._delete_recursive(node.right, key)
        else:
            if node.left is None:
                return node.right
            elif node.right is None:
                return node.left

            # Node with two children: Get the in-order successor (smallest in the right subtree)
            temp = self._min_value_node(node.right)
            node.key = temp.key
            node.right = self._delete_recursive(node.right, temp.key)
        return node

    def _min_value_node(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current

```

Fenwick Tree

Implements a 1D Fenwick Tree, also known as a Binary Indexed Tree (BIT). This data structure is used to efficiently calculate prefix sums (or any other associative and invertible operation) on an array while supporting point updates. A Fenwick Tree of size N allows for two main operations, both in logarithmic time:

1. add(idx, delta): Adds **delta** to the element at index **idx**.

2. query(right): Computes the sum of the elements in the range [0, right).

The core idea is that any integer can be represented as a sum of powers of two. Similarly, a prefix sum can be represented as a sum of sums over certain sub-ranges, where the size of these sub-ranges are powers of two. The tree stores these precomputed sub-range sums. This implementation is 0-indexed for user-facing operations, which is a common convention in Python. The internal logic is adapted to work with this indexing.

- To find the next index to update in `add`, we use `idx |= idx + 1`.
- To find the next index to sum in `query`, we use `idx = (idx & (idx + 1)) - 1`.

```python
class FenwickTree:
    """
    A class that implements a 1D Fenwick Tree (Binary Indexed Tree).
    This implementation uses 0-based indexing for its public methods.
    """

    def __init__(self, size):
        """
        Initializes the Fenwick Tree for an array of a given size.
        All elements are initially zero.

        Args:
            size (int): The number of elements the tree will support.
        """
        self.tree = [0] * size

    def add(self, idx, delta):
        """
        Adds a delta value to the element at a specific index.
        This operation updates all prefix sums that include this index.

        Args:
            idx (int): The 0-based index of the element to update.
            delta (int): The value to add to the element at `idx`.
        """
        while idx < len(self.tree):
            self.tree[idx] += delta
            idx |= idx + 1

    def query(self, right):
        """
        Computes the prefix sum of elements up to (but not including) `right`.
        This is the sum of the range [0, right-1].

        Args:
            right (int): The 0-based exclusive upper bound of the query range.

        Returns:
            int: The sum of elements in the prefix `[0, right-1]`.
        """
        idx = right - 1
        total_sum = 0
        while idx >= 0:
            total_sum += self.tree[idx]
            idx = (idx & (idx + 1)) - 1
        return total_sum

    def query_range(self, left, right):
        """
        Computes the sum of elements in the range [left, right-1].

        Args:
            left (int): The 0-based inclusive lower bound of the query range.
            right (int): The 0-based exclusive upper bound of the query range.

        Returns:
            int: The sum of elements in the specified range.
        """
        if left >= right:
            return 0
        return self.query(right) - self.query(left)
```

Fenwick Tree 2D

Implements a 2D Fenwick Tree (Binary Indexed Tree). This data structure extends the 1D Fenwick Tree to support point updates and prefix rectangle sum queries on a 2D grid. The primary operations are:

1. add(r, c, delta): Adds **delta** to the element at grid cell (r, c).

2. query(r, c): Computes the sum of the rectangle from (0, 0) to (r-1, c-1).

A 2D Fenwick Tree can be conceptualized as a Fenwick Tree where each element is itself another Fenwick Tree. The **add** and **query** operations therefore involve traversing the tree structure in both dimensions, resulting in a time complexity that is the product of the logarithmic complexities of each dimension. The **query_range** method uses the principle of inclusion-exclusion on the prefix rectangle sums to calculate the sum of any arbitrary subrectangle. Given a rectangle defined by top-left (r1, c1) and bottom-right (r2-1, c2-1), the sum is: Sum(r2, c2) - Sum(r1, c2) - Sum(r2, c1) + Sum(r1, c1), where Sum(r, c) is the prefix sum from (0,0) to (r-1, c-1).

```python
class FenwickTree2D:
    """
    A class that implements a 2D Fenwick Tree using 0-based indexing.
    """

    def __init__(self, rows, cols):
        """
        Initializes the 2D Fenwick Tree for a grid of a given size.
        All elements are initially zero.
```

```python
        Args:
            rows (int): The number of rows in the grid.
            cols (int): The number of columns in the grid.
        """
        self.rows = rows
        self.cols = cols
        self.tree = [[0] * cols for _ in range(rows)]

    def add(self, r, c, delta):
        """
        Adds a delta value to the element at grid cell (r, c).

        Args:
            r (int): The 0-based row index of the element to update.
            c (int): The 0-based column index of the element to update.
            delta (int): The value to add.
        """
        i = r
        while i < self.rows:
            j = c
            while j < self.cols:
                self.tree[i][j] += delta
                j |= j + 1
            i |= i + 1

    def query(self, r, c):
        """
        Computes the prefix sum of the rectangle from (0, 0) to (r-1, c-1).

        Args:
            r (int): The 0-based exclusive row bound of the query rectangle.
            c (int): The 0-based exclusive column bound of the query rectangle.

        Returns:
            int: The sum of the elements in the rectangle [0..r-1, 0..c-1].
        """
        total_sum = 0
        i = r - 1
        while i >= 0:
            j = c - 1
            while j >= 0:
                total_sum += self.tree[i][j]
                j = (j & (j + 1)) - 1
            i = (i & (i + 1)) - 1
        return total_sum

    def query_range(self, r1, c1, r2, c2):
        """
        Computes the sum of the rectangle from (r1, c1) to (r2-1, c2-1).

        Args:
            r1, c1 (int): The 0-based inclusive top-left coordinates.
            r2, c2 (int): The 0-based exclusive bottom-right coordinates.

        Returns:
            int: The sum of elements in the specified rectangular range.
        """
        if r1 >= r2 or c1 >= c2:
            return 0

        total = self.query(r2, c2)
        total -= self.query(r1, c2)
        total -= self.query(r2, c1)
        total += self.query(r1, c1)
        return total

```

Hash Map Custom

Provides an explanation and an example of a custom hash for Python's dictionaries and sets to prevent slowdowns from anti-hash tests. In competitive programming, some problems use test cases specifically designed to cause many collisions in standard hash table implementations (like Python's dict), degrading their performance from average $O(1)$ to worst-case $O(N)$. This can be mitigated by using a hash function with a randomized component, so that the hash values are unpredictable to an adversary. A common technique is to XOR the object's standard hash with a fixed, randomly generated constant. The `splitmix64` function shown below is a high-quality hash function that can be used for this purpose. It's simple, fast, and provides good distribution. To use a custom hash, you can wrap integer or tuple keys in a custom class that overrides the `__hash__` and `__eq__` methods. Example usage with a dictionary: `my_map = {} my_map[CustomHash(123)] = "value"` This forces Python's `dict` to use your `CustomHash` object's `__hash__` method, thus using the randomized hash function. This is particularly useful in problems involving hashing of tuples, such as coordinates or polynomial hash values.

```python
import time

# A fixed random seed ensures the same hash function for each run,
# but it's generated based on time to be unpredictable.
SPLITMIX64_SEED = int(time.time()) ^ 0x9E3779B97F4A7C15


def splitmix64(x):
    """A fast, high-quality hash function for 64-bit integers."""
    x += 0x9E3779B97F4A7C15
    x = (x ^ (x >> 30)) * 0xBF58476D1CE4E5B9
    x = (x ^ (x >> 27)) * 0x94D049BB133111EB
    return x ^ (x >> 31)


class CustomHash:
    """
    A wrapper class for hashable objects to use a custom hash function.
    This helps prevent collisions from anti-hash test cases.
    """

    def __init__(self, obj):
        self.obj = obj

    def __hash__(self):
        # Combine the object's hash with a fixed random seed using a robust function.
        return splitmix64(hash(self.obj) + SPLITMIX64_SEED)

    def __eq__(self, other):
        # The wrapped objects must still be comparable.
        return self.obj == other.obj

    def __repr__(self):
        return f"CustomHash({self.obj})"

```

```python
# Example of how to use it
def custom_hash_example():
    # Standard dictionary, potentially vulnerable
    standard_dict = {}
    # Dictionary with custom hash, much more robust
    custom_dict = {}

    key = (12345, 67890)  # A tuple key, common in geometry or hashing problems

    # Using the standard hash
    standard_dict[key] = "some value"

    # Using the custom hash
    custom_key = CustomHash(key)
    custom_dict[custom_key] = "some value"

    print(f"Standard hash for {key}: {hash(key)}")
    print(f"Custom hash for {key}: {hash(custom_key)}")

    # Verifying that it works
    assert custom_key in custom_dict
    assert CustomHash(key) in custom_dict
    assert CustomHash((0, 0)) not in custom_dict

```

Line Container

Implements a Line Container for the Convex Hull Trick. This data structure maintains a set of lines of the form `y = mx + c` and allows for efficiently querying the minimum `y` value for a given `x`. This is a key component in optimizing certain dynamic programming problems. This implementation is specialized for the following common case:

- Queries ask for the minimum value.
- The slopes `m` of the lines added are monotonically decreasing.

The lines are stored in a deque, which acts as the lower convex hull. When a new line is added, we maintain the convexity of the hull by removing any lines from the back that become redundant. A line becomes redundant if the intersection point of its neighbors moves left, violating the convexity property. This check is done using cross-products to avoid floating-point arithmetic. Queries are performed using a binary search on the hull to find the optimal line for the given `x`. If the `x` values for queries are also monotonic, the query time can be improved to amortized $O(1)$ by using a pointer instead of a binary search. To adapt for maximum value queries, change the inequalities in `add` and `query`. To handle monotonically increasing slopes, add lines to the front of the deque and adjust the `add` method's popping logic accordingly. because each line is added and removed at most once.

```python
class LineContainer:
    """
    A data structure for the Convex Hull Trick, optimized for minimum queries
    and monotonically decreasing slopes.
    """

    def __init__(self):
        # Each line is stored as a tuple (m, c) representing y = mx + c.
        self.hull = []

    def _is_redundant(self, l1, l2, l3):
        """
        Checks if line l2 is redundant given its neighbors l1 and l3.
        l2 is redundant if the intersection of l1 and l3 is to the left of
        the intersection of l1 and l2.
        Intersection of (m1, c1) and (m2, c2) is x = (c2 - c1) / (m1 - m2).
        We check if (c3-c1)/(m1-m3) <= (c2-c1)/(m1-m2).
        To avoid floating point, we use cross-multiplication.
        Since slopes are decreasing, m1 > m2 > m3, so (m1-m3) and (m1-m2) are positive.
        The inequality becomes (c3-c1)*(m1-m2) <= (c2-c1)*(m1-m3).
        """
        m1, c1 = l1
        m2, c2 = l2
        m3, c3 = l3
        # Note the direction of inequality might change based on max/min query
        # and increasing/decreasing slopes. This is for min query, decr. slopes.
        return (c3 - c1) * (m1 - m2) <= (c2 - c1) * (m1 - m3)

    def add(self, m, c):
        """
        Adds a new line y = mx + c to the container.
        Assumes that m is less than the slope of any previously added line.
        """
        new_line = (m, c)
        while len(self.hull) >= 2 and self._is_redundant(
            self.hull[-2], self.hull[-1], new_line
        ):
            self.hull.pop()
        self.hull.append(new_line)

    def query(self, x):
        """
        Finds the minimum value of y = mx + c for a given x among all lines.
        """
        if not self.hull:
            return float("inf")

        # Binary search for the optimal line.
        # The function `f(i) = m_i * x + c_i` is not monotonic, but the
        # optimal line index is. Specifically, the function `f(i+1) - f(i)`
        # is monotonic. We are looking for the point where the function
        # transitions from decreasing to increasing.
        low, high = 0, len(self.hull) - 1
        res_idx = 0

        while low <= high:
            mid = (low + high) // 2
            # Check if mid is better than mid+1
            if mid + 1 < len(self.hull):
                val_mid = self.hull[mid][0] * x + self.hull[mid][1]
                val_next = self.hull[mid + 1][0] * x + self.hull[mid + 1][1]
                if val_mid > val_next:
                    low = mid + 1
                else:
                    res_idx = mid
                    high = mid - 1
            else:
                res_idx = mid
```

```python
                high = mid - 1

        m, c = self.hull[res_idx]
        return m * x + c

```

Ordered Set

Implements an Ordered Set data structure using a randomized balanced binary search tree (Treap). An Ordered Set supports all the standard operations of a balanced BST (insert, delete, search) and two additional powerful operations:

1. `find_by_order(k)`: Finds the k-th smallest element in the set (0-indexed).
2. `order_of_key(key)`: Finds the number of elements in the set that are strictly smaller than the given key (i.e., its rank).

To achieve this, each node in the underlying Treap is augmented to store the size of the subtree rooted at that node. This `size` information is updated during insertions and deletions. The ordered set operations then use these sizes to navigate the tree efficiently. For example, to find the k-th element, we can compare `k` with the size of the left subtree to decide whether to go left, right, or stop at the current node. The implementation is based on the elegant `split` and `merge` operations, which are modified to maintain the subtree size property. `find_by_order`, and `order_of_key` operations, where N is the number of elements in the set.

```python
import random


class Node:
    """Represents a single node in the Ordered Set's underlying Treap."""

    def __init__(self, key):
        self.key = key
        self.priority = random.random()
        self.size = 1
        self.left = None
        self.right = None


def _get_size(t):
    return t.size if t else 0


def _update_size(t):
    if t:
        t.size = 1 + _get_size(t.left) + _get_size(t.right)


def _split(t, key):
    """
    Splits the tree `t` into two trees: one with keys < `key` (l)
    and one with keys >= `key` (r).
    """
    if not t:
        return None, None
    if t.key < key:
        l, r = _split(t.right, key)
        t.right = l
        _update_size(t)
        return t, r
    else:
        l, r = _split(t.left, key)
        t.left = r
        _update_size(t)
        return l, t


def _merge(t1, t2):
    """Merges two trees `t1` and `t2`, assuming keys in `t1` < keys in `t2`."""
    if not t1:
        return t2
    if not t2:
        return t1
    if t1.priority > t2.priority:
        t1.right = _merge(t1.right, t2)
        _update_size(t1)
        return t1
    else:
        t2.left = _merge(t1, t2.left)
        _update_size(t2)
        return t2


class OrderedSet:
    """
    An Ordered Set implementation using a Treap.
    Supports finding the k-th element and the rank of an element.
    """

    def __init__(self):
        self.root = None

    def search(self, key):
        node = self.root
        while node:
            if node.key == key:
                return True
            node = node.left if key < node.key else node.right
        return False

    def insert(self, key):
        if self.search(key):
            return
        new_node = Node(key)
        l, r = _split(self.root, key)
        self.root = _merge(_merge(l, new_node), r)

    def delete(self, key):
        if not self.search(key):
            return
        l, r = _split(self.root, key)
        _, r_prime = _split(r, key + 1)
        self.root = _merge(l, r_prime)

    def find_by_order(self, k):
        """Finds the k-th smallest element (0-indexed)."""
        node = self.root
        while node:
            left_size = _get_size(node.left)
            if left_size == k:
                return node.key
            elif k < left_size:
                node = node.left
            else:
                k -= left_size + 1
                node = node.right
        return None

    def order_of_key(self, key):
```

```python
        """Finds the number of elements strictly smaller than key."""
        count = 0
        node = self.root
        while node:
            if key == node.key:
                count += _get_size(node.left)
                break
            elif key < node.key:
                node = node.left
            else:
                count += _get_size(node.left) + 1
                node = node.right
        return count

    def __len__(self):
        return _get_size(self.root)
```

Segment Tree Lazy

Implements a Segment Tree with lazy propagation. This powerful data structure is designed to handle range updates and range queries efficiently. While a standard Segment Tree can perform range queries in $O(\log N)$ time, updates are limited to single points. Lazy propagation extends this capability to allow range updates (e.g., adding a value to all elements in a range) to also be performed in $O(\log N)$ time. The core idea is to postpone updates to tree nodes and apply them only when necessary. When an update is requested for a range `[l, r]`, we traverse the tree. If a node's range is fully contained within `[l, r]`, instead of updating all its children, we store the pending update value in a `lazy` array for that node and update the node's main value. We then stop traversing down that path. This pending update is "pushed" down to its children only when a future query or update needs to access one of the children. This implementation supports range addition updates and range sum queries. The logic can be adapted for other associative operations like range minimum/maximum and range assignment. The initial `build` operation takes $O(N)$ time. to be safe for a complete binary tree representation.

```python
class SegmentTree:
    def __init__(self, arr):
        self.n = len(arr)
        self.tree = [0] * (4 * self.n)
        self.lazy = [0] * (4 * self.n)
        self.arr = arr
        self._build(1, 0, self.n - 1)

    def _build(self, v, tl, tr):
        if tl == tr:
            self.tree[v] = self.arr[tl]
        else:
            tm = (tl + tr) // 2
            self._build(2 * v, tl, tm)
            self._build(2 * v + 1, tm + 1, tr)
            self.tree[v] = self.tree[2 * v] + self.tree[2 * v + 1]

    def _push(self, v, tl, tr):
        if self.lazy[v] == 0:
            return

        range_len = tr - tl + 1
        self.tree[v] += self.lazy[v] * range_len

        if tl != tr:
            self.lazy[2 * v] += self.lazy[v]
            self.lazy[2 * v + 1] += self.lazy[v]

        self.lazy[v] = 0

    def _update(self, v, tl, tr, l, r, addval):
        self._push(v, tl, tr)
        if l > r:
            return
        if l == tl and r == tr:
            self.lazy[v] += addval
            self._push(v, tl, tr)
        else:
            tm = (tl + tr) // 2
            self._update(2 * v, tl, tm, l, min(r, tm), addval)
            self._update(2 * v + 1, tm + 1, tr, max(l, tm + 1), r, addval)

            # After children are updated, update self based on pushed children
            self._push(2 * v, tl, tm)
            self._push(2 * v + 1, tm + 1, tr)
            self.tree[v] = self.tree[2 * v] + self.tree[2 * v + 1]

    def _query(self, v, tl, tr, l, r):
        if l > r:
            return 0
        self._push(v, tl, tr)
        if l == tl and r == tr:
            return self.tree[v]

        tm = (tl + tr) // 2
        left_sum = self._query(2 * v, tl, tm, l, min(r, tm))
        right_sum = self._query(2 * v + 1, tm + 1, tr, max(l, tm + 1), r)
        return left_sum + right_sum

    def update(self, l, r, addval):
        # Updates range [l, r] (inclusive)
        if l > r:
            return
        self._update(1, 0, self.n - 1, l, r, addval)

    def query(self, l, r):
        # Queries range [l, r] (inclusive)
        if l > r:
            return 0
        return self._query(1, 0, self.n - 1, l, r)
```

Sparse Table

Implements a Sparse Table for fast Range Minimum Queries (RMQ). This data structure is ideal for answering range queries on a static array for idempotent functions like min, max, or gcd. The core idea is to precompute the answers for all ranges that have a length that is a power of two. The table `st[k][i]` stores the minimum value in the range `[i, i + 2^k - 1]`. This precomputation takes $O(N \log N)$ time. Once the table is built, a query for any arbitrary range `[l, r]` can be answered in $O(1)$ time. This is achieved by finding the largest power of two, `2^k`, that is less than or equal to the range length `r - l + 1`. The query then returns the minimum of two overlapping ranges: `[l, l + 2^k - 1]` and `[r - 2^k + 1, r]`. Because `min` is an idempotent function, the overlap does not affect the result. This implementation is for range minimum, but can be easily adapted for range maximum by changing `min` to `max`.

```python
import math


class SparseTable:
    """
    A class that implements a Sparse Table for efficient Range Minimum Queries.
    This implementation assumes 0-based indexing for the input array and queries.
    """

    def __init__(self, arr):
        """
        Initializes the Sparse Table from an input array.

        Args:
            arr (list[int]): The static list of numbers to be queried.
        """
        self.n = len(arr)
        if self.n == 0:
            return

        self.max_log = self.n.bit_length() - 1
        self.st = [[0] * self.n for _ in range(self.max_log + 1)]
        self.st[0] = list(arr)

        for k in range(1, self.max_log + 1):
            for i in range(self.n - (1 << k) + 1):
                self.st[k][i] = min(
                    self.st[k - 1][i], self.st[k - 1][i + (1 << (k - 1))]
                )

        self.log_table = [0] * (self.n + 1)
        for i in range(2, self.n + 1):
            self.log_table[i] = self.log_table[i >> 1] + 1

    def query(self, l, r):
        """
        Queries the minimum value in the inclusive range [l, r].

        Args:
            l (int): The 0-based inclusive starting index of the range.
            r (int): The 0-based inclusive ending index of the range.

        Returns:
            int: The minimum value in the range [l, r]. Returns infinity
             if the table is empty or the range is invalid.
        """
        if self.n == 0 or l > r:
            return float("inf")

        length = r - l + 1
        k = self.log_table[length]
        return min(self.st[k][l], self.st[k][r - (1 << k) + 1])
```

Treap

Implements a Treap, a randomized balanced binary search tree. A Treap is a data structure that combines the properties of a binary search tree and a heap. Each node in the Treap has both a key and a randomly assigned priority. The keys follow the binary search tree property (left child's key < parent's key < right child's key), while the priorities follow the max-heap property (parent's priority > children's priorities). The random assignment of priorities ensures that, with high probability, the tree remains balanced, leading to logarithmic time complexity for standard operations. This implementation uses `split` and `merge` operations, which are a clean and powerful way to handle insertions and deletions.

- `split(key)`: Splits the tree into two separate trees: one containing all keys less than `key`, and another containing all keys greater than or equal to `key`.
- `merge(left, right)`: Merges two trees, `left` and `right`, under the assumption that all keys in `left` are smaller than all keys in `right`.

Using these, `insert` and `delete` can be implemented elegantly. where N is the number of nodes in the Treap. The performance depends on the randomness of the priorities.

```python
import random


class Node:
    """
    Represents a single node in the Treap.
    Each node contains a key, a randomly generated priority, and left/right children.
    """

    def __init__(self, key):
        self.key = key
        self.priority = random.random()
        self.left = None
        self.right = None


def _split(t, key):
    """
    Splits the tree rooted at `t` into two trees based on `key`.
    Returns a tuple (left_tree, right_tree), where left_tree contains all keys
    from `t` that are less than `key`, and right_tree contains all keys that are
    greater than or equal to `key`.
    """
    if not t:
        return None, None
    if t.key < key:
        l, r = _split(t.right, key)
        t.right = l
        return t, r
    else:
        l, r = _split(t.left, key)
        t.left = r
        return l, t
```

```python

def _merge(t1, t2):
    """
    Merges two trees `t1` and `t2`.
    Assumes all keys in `t1` are less than all keys in `t2`.
    The merge is performed based on node priorities to maintain the heap property.
    """
    if not t1:
        return t2
    if not t2:
        return t1
    if t1.priority > t2.priority:
        t1.right = _merge(t1.right, t2)
        return t1
    else:
        t2.left = _merge(t1, t2.left)
        return t2


class Treap:
    """
    The Treap class providing a public API for balanced BST operations.
    """

    def __init__(self):
        """Initializes an empty Treap."""
        self.root = None

    def search(self, key):
        """
        Searches for a key in the Treap.
        Returns True if the key is found, otherwise False.
        """
        node = self.root
        while node:
            if node.key == key:
                return True
            elif key < node.key:
                node = node.left
            else:
                node = node.right
        return False

    def insert(self, key):
        """
        Inserts a key into the Treap. If the key already exists, the tree is unchanged.
        """
        if self.search(key):
            return  # Don't insert duplicates

        new_node = Node(key)
        l, r = _split(self.root, key)
        # l has keys < key, r has keys >= key.
        # Merge new_node with r first, then merge l with the result.
        self.root = _merge(l, _merge(new_node, r))

    def delete(self, key):
        """
        Deletes a key from the Treap. If the key is not found, the tree is unchanged.
        """
        if not self.search(key):
            return

        # Split to isolate the node to be deleted.
        l, r = _split(self.root, key)  # l has keys < key, r has keys >= key
        _, r_prime = _split(r, key + 1)  # r_prime has keys > key

        # Merge the remaining parts back together.
        self.root = _merge(l, r_prime)

```

Union Find

Implements the Union-Find data structure, also known as Disjoint Set Union (DSU). It is used to keep track of a partition of a set of elements into a number of disjoint, non-overlapping subsets. The two primary operations are finding the representative (or root) of a set and merging two sets. This implementation includes two key optimizations:

1. Path Compression: During a `find` operation, it makes every node on the path from the query node to the root point directly to the root. This dramatically flattens the tree structure.

2. Union by Size: During a `union` operation, it always attaches the root of the smaller tree to the root of the larger tree. This helps in keeping the trees shallow, which speeds up future `find` operations.

The combination of these two techniques makes the amortized time complexity of both `find` and `union` operations nearly constant. is the extremely slow-growing inverse Ackermann function. For all practical purposes, this is considered constant time.

```python
class UnionFind:
    """
    A class that implements the Union-Find data structure with path compression
    and union by size optimizations.
    """

    def __init__(self, n):
        """
        Initializes the Union-Find structure for n elements, where each element
        is initially in its own set.
        Args:
            n (int): The number of elements.
        """
        self.parent = list(range(n))
        self.size = [1] * n

    def find(self, i):
        """
        Finds the representative (root) of the set containing element i.
        Applies path compression along the way.
        Args:
            i (int): The element to find.
        Returns:
            int: The representative of the set containing i.
        """
        if self.parent[i] == i:
            return i
        self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    def union(self, i, j):
        """
        Merges the sets containing elements i and j.
        Applies union by size.
        Args:
            i (int): An element in the first set.
            j (int): An element in the second set.
```

```
        Returns:
            bool: True if the sets were merged, False if
            ↪  they were already in the same set.
        """
        root_i = self.find(i)
        root_j = self.find(j)
        if root_i != root_j:
            if self.size[root_i] < self.size[root_j]:
                root_i, root_j = root_j, root_i
            self.parent[root_j] = root_i
            self.size[root_i] += self.size[root_j]
            return True
        return False

```

Chapter 5

Graph Algorithms

Bellman Ford

Implements the Bellman-Ford algorithm for finding the single-source shortest paths in a weighted graph. Unlike Dijkstra's algorithm, Bellman-Ford can handle graphs with negative edge weights. The algorithm works by iteratively relaxing edges. It repeats a relaxation step $V-1$ times, where V is the number of vertices. In each relaxation step, it iterates through all edges `(u, v)` and updates the distance to `v` if a shorter path is found through `u`. After $V-1$ iterations, the shortest paths are guaranteed to be found, provided there are no negative-weight cycles reachable from the source. A final, V-th iteration is performed to detect negative-weight cycles. If any distance can still be improved during this iteration, it means a negative-weight cycle exists, and the shortest paths are not well-defined (they can be infinitely small). This implementation takes an edge list as input, which is a common and convenient representation for this algorithm. of edges. The algorithm iterates through all edges V times.

```python
def bellman_ford(edges, start_node, n):
    """
    Finds shortest paths from a start node, handling negative weights and
    detecting negative cycles.

    Args:
        edges (list[tuple[int, int, int]]): A list of all edges in the graph,
            where each tuple is (u, v, weight) for an edge u -> v.
        start_node (int): The node from which to start the search.
        n (int): The total number of nodes in the graph.

    Returns:
        tuple[list[float], bool]: A tuple containing:
            - A list of shortest distances. `float('inf')` for unreachable nodes.
            - A boolean that is True if a negative cycle is detected, False otherwise.
    """
    if not (0 <= start_node < n):
        return [float("inf")] * n, False

    dist = [float("inf")] * n
    dist[start_node] = 0

    for i in range(n - 1):
        updated = False
        for u, v, w in edges:
            if dist[u] != float("inf") and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                updated = True
        if not updated:
            break

    for u, v, w in edges:
        if dist[u] != float("inf") and dist[u] + w < dist[v]:
            return dist, True

    return dist, False
```

Bipartite Matching

Implements an algorithm to find the maximum matching in a bipartite graph. A bipartite graph is one whose vertices can be divided into two disjoint and independent sets, U and V, such that every edge connects a vertex in U to one in V. A matching is a set of edges without common vertices. The goal is to find a matching with the maximum possible number of edges. This implementation uses the augmenting path algorithm, a common approach based on Ford-Fulkerson. It works by repeatedly finding "augmenting paths" in the graph. An augmenting path is a path that starts from an unmatched vertex in the left partition (U), ends at an unmatched vertex in the right partition (V), and alternates between edges that are not in the current matching and edges that are. The algorithm proceeds as follows:

1. Initialize an empty matching.
2. For each vertex `u` in the left partition U: a. Try to find an augmenting path starting from `u` using a Depth-First Search (DFS). b. The DFS explores neighbors `v` of `u`. If `v` is unmatched, we have found an augmenting path of length 1. We match `u` with `v`. c. If `v` is already matched with some vertex `u'`, the DFS recursively tries to find an alternative match for `u'`. If it succeeds, we can then match `u` with `v`.
3. If an augmenting path is found, the size of the matching increases by one. The edges in the matching are updated by "flipping" the status of edges along the path.
4. The process continues until no more augmenting paths can be found. The size of the resulting matching is the maximum possible.

E is the number of edges. For each vertex in U, we may perform a DFS that traverses the entire graph.

```python
def bipartite_matching(adj, n1, n2):
    """
    Finds the maximum matching in a bipartite graph.

```

```python
    Args:
        adj (list[list[int]]): Adjacency list for the left partition.
            `adj[u]` contains a list of neighbors of node `u` (from the left set)
            in the right set. Nodes in the left set are indexed 0 to n1-1.
            Nodes in the right set are indexed 0 to n2-1.
        n1 (int): The number of vertices in the left partition.
        n2 (int): The number of vertices in the right partition.

    Returns:
        int: The size of the maximum matching.
    """
    match_right = [-1] * n2
    matching_size = 0

    def dfs(u, visited):
        for v in adj[u]:
            if not visited[v]:
                visited[v] = True
                if match_right[v] < 0 or dfs(match_right[v], visited):
                    match_right[v] = u
                    return True
        return False

    for u in range(n1):
        visited = [False] * n2
        if dfs(u, visited):
            matching_size += 1

    return matching_size

```

Dijkstra

Implements Dijkstra's algorithm for finding the single-source shortest paths in a weighted graph with non-negative edge weights. Dijkstra's algorithm maintains a set of visited vertices and finds the shortest path from a source vertex to all other vertices in the graph. It uses a priority queue to greedily select the unvisited vertex with the smallest distance from the source. The algorithm proceeds as follows:

1. Initialize a distances array with infinity for all vertices except the source, which is set to 0.

2. Initialize a priority queue and add the source vertex with a distance of 0.

3. While the priority queue is not empty, extract the vertex `u` with the smallest distance.

4. If `u` has already been processed with a shorter path, skip it.

5. For each neighbor `v` of `u`, calculate the distance through `u`. If this new path is shorter than the known distance to `v`, update the distance and add `v` to the priority queue with its new, shorter distance.

This implementation uses Python's `heapq` module as a min-priority queue. The graph is represented by an adjacency list where each entry is a tuple (neighbor, weight). of edges. The log factor comes from the priority queue operations. priority queue.

```python
import heapq


def dijkstra(adj, start_node, n):
    """
    Finds the shortest paths from a start node to all other nodes in a graph.

    Args:
        adj (list[list[tuple[int, int]]]): The adjacency list representation of
            the graph. adj[u] contains tuples (v, weight) for edges u -> v.
        start_node (int): The node from which to start the search.
        n (int): The total number of nodes in the graph.

    Returns:
        list[float]: A list of shortest distances from the start_node to each
                     node. `float('inf')` indicates an unreachable node.
    """
    if not (0 <= start_node < n):
        return [float("inf")] * n

    dist = [float("inf")] * n
    dist[start_node] = 0
    pq = [(0, start_node)]

    while pq:
        d, u = heapq.heappop(pq)

        if d > dist[u]:
            continue

        for v, weight in adj[u]:
            if dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                heapq.heappush(pq, (dist[v], v))

    return dist

```

Dinic

Implements Dinic's algorithm for computing the maximum flow in a flow network from a source `s` to a sink `t`. Dinic's is one of the most efficient algorithms for this problem. The algorithm operates in phases. In each phase, it does the following:

1. Build a "level graph" using a Breadth-First Search (BFS) from the source `s` on the residual graph. The level of a vertex is its shortest distance from `s`. The level graph only contains edges `(u, v)` where `level[v] == level[u] + 1`. If the sink `t` is not reachable from `s` in the residual graph, the algorithm terminates.

2. Find a "blocking flow" in the level graph using a Depth-First Search (DFS) from `s`. A blocking flow is a flow where every path from `s` to `t` in the level graph has at least one saturated edge. The DFS pushes as much flow as possible along

paths from `s` to `t`. Pointers are used to avoid re-exploring dead-end paths within the same phase.

3. Add the blocking flow found in the phase to the total maximum flow.

The process is repeated until the sink is no longer reachable from the source. such as $O(E\sqrt{V})$ for bipartite matching and $O(E\min(V^{2/3}, E^{1/2}))$ for unit-capacity networks.

```python
from collections import deque


class Dinic:
    def __init__(self, n):
        self.n = n
        self.graph = [[] for _ in range(n)]
        self.level = [-1] * n
        self.ptr = [0] * n
        self.inf = float("inf")

    def add_edge(self, u, v, cap):
        # Forward edge
        self.graph[u].append([v, cap,
        ↪ len(self.graph[v])])
        # Backward edge
        self.graph[v].append([u, 0, len(self.graph[u]) -
        ↪ 1])

    def _bfs(self, s, t):
        self.level = [-1] * self.n
        self.level[s] = 0
        q = deque([s])
        while q:
            u = q.popleft()
            for i in range(len(self.graph[u])):
                v, cap, rev = self.graph[u][i]
                if cap > 0 and self.level[v] < 0:
                    self.level[v] = self.level[u] + 1
                    q.append(v)
        return self.level[t] != -1

    def _dfs(self, u, t, pushed):
        if pushed == 0:
            return 0
        if u == t:
            return pushed

        while self.ptr[u] < len(self.graph[u]):
            edge_idx = self.ptr[u]
            v, cap, rev_idx = self.graph[u][edge_idx]

            if self.level[v] != self.level[u] + 1 or cap
            ↪ == 0:
                self.ptr[u] += 1
                continue

            tr = self._dfs(v, t, min(pushed, cap))
            if tr == 0:
                self.ptr[u] += 1
                continue

            self.graph[u][edge_idx][1] -= tr
            self.graph[v][rev_idx][1] += tr
            return tr
        return 0

    def max_flow(self, s, t):
        if s == t:
            return 0
        total_flow = 0
        while self._bfs(s, t):
            self.ptr = [0] * self.n
            pushed = self._dfs(s, t, self.inf)
            while pushed > 0:
                total_flow += pushed
                pushed = self._dfs(s, t, self.inf)
        return total_flow

```

Euler Path

Implements Hierholzer's algorithm to find an Eulerian path or cycle in a graph. An Eulerian path visits every edge of a graph exactly once. An Eulerian cycle is an Eulerian path that starts and ends at the same vertex. The existence of an Eulerian path/cycle depends on the degrees of the vertices: For an undirected graph:

- An Eulerian cycle exists if and only if every vertex has an even degree, and all vertices with a non-zero degree belong to a single connected component.
- An Eulerian path exists if and only if there are zero or two vertices of odd degree, and all vertices with a non-zero degree belong to a single component. If there are two odd-degree vertices, the path must start at one and end at the other.

For a directed graph:

- An Eulerian cycle exists if and only if for every vertex, the in-degree equals the out-degree, and the graph is strongly connected (ignoring isolated vertices).
- An Eulerian path exists if and only if at most one vertex has `out-degree - in-degree = 1` (the start), at most one vertex has `in-degree - out-degree = 1` (the end), every other vertex has equal in- and out-degrees, and the underlying undirected graph is connected.

Hierholzer's algorithm finds the path by starting a traversal from a valid starting node. It follows edges until it gets stuck, and then backtracks, forming the path in reverse. This implementation uses an iterative approach with a stack.

```python
from collections import Counter


def find_euler_path(adj, n, directed=False):
    """
    Finds an Eulerian path or cycle in a graph.

    Args:
        adj (list[list[int]]): The adjacency list
        ↪ representation of the graph.
          Handles multigraphs if neighbors are
          ↪ repeated.
        n (int): The total number of nodes in the graph.
        directed (bool): True if the graph is directed,
        ↪ False otherwise.

    Returns:
```

```python
        list[int] | None: A list of nodes representing the Eulerian path,
                              or None if no such path exists.
    """
    if n == 0:
        return []

    num_edges = 0
    if directed:
        in_degree = [0] * n
        out_degree = [0] * n
        for u in range(n):
            out_degree[u] = len(adj[u])
            num_edges += len(adj[u])
            for v in adj[u]:
                in_degree[v] += 1

        start_node, end_node_count = -1, 0
        for i in range(n):
            if out_degree[i] - in_degree[i] == 1:
                if start_node != -1:
                    return None
                start_node = i
            elif in_degree[i] - out_degree[i] == 1:
                end_node_count += 1
                if end_node_count > 1:
                    return None
            elif in_degree[i] != out_degree[i]:
                return None

        if start_node == -1:
            for i in range(n):
                if out_degree[i] > 0:
                    start_node = i
                    break
            if start_node == -1:
                return [0] if n > 0 else []

    else:
        degree = [0] * n
        for u in range(n):
            degree[u] = len(adj[u])
            num_edges += len(adj[u])
        num_edges //= 2

        odd_degree_nodes = [i for i, d in enumerate(degree) if d % 2 != 0]
        if len(odd_degree_nodes) > 2:
            return None

        start_node = -1
        if odd_degree_nodes:
            start_node = odd_degree_nodes[0]
        else:
            for i in range(n):
                if degree[i] > 0:
                    start_node = i
                    break
            if start_node == -1:
                return [0] if n > 0 else []

    adj_counts = [Counter(neighbors) for neighbors in adj]
    path = []
    stack = [start_node]

    while stack:
        u = stack[-1]
        if adj_counts[u]:
            v = next(iter(adj_counts[u]))
            adj_counts[u][v] -= 1
            if adj_counts[u][v] == 0:
                del adj_counts[u][v]

            if not directed:
                adj_counts[v][u] -= 1
                if adj_counts[v][u] == 0:
                    del adj_counts[v][u]

            stack.append(v)
        else:
            path.append(stack.pop())

    path.reverse()

    if len(path) == num_edges + 1:
        return path
    else:
        return None
```

Floyd Warshall

Implements the Floyd-Warshall algorithm for finding all-pairs shortest paths in a weighted directed graph. This algorithm can handle graphs with negative edge weights. The algorithm is based on a dynamic programming approach. It iteratively considers each vertex `k` and updates the shortest path between all pairs of vertices `(i, j)` to see if a path through `k` is shorter. The core recurrence is: `dist(i, j) = min(dist(i, j), dist(i, k) + dist(k, j))` After running the algorithm with all vertices `k` from 0 to `V-1`, the resulting distance matrix contains the shortest paths between all pairs of vertices. A key feature of Floyd-Warshall is its ability to detect negative-weight cycles. If, after the algorithm completes, the distance from any vertex `i` to itself (`dist[i][i]`) is negative, it indicates that there is a negative-weight cycle reachable from `i`. This implementation takes an edge list as input, builds an adjacency matrix, runs the algorithm, and then checks for negative cycles. dominate the runtime.

```python
def floyd_warshall(edges, n):
    """
    Finds all-pairs shortest paths in a graph using the Floyd-Warshall algorithm.

    Args:
        edges (list[tuple[int, int, int]]): A list of all edges in the graph,
            where each tuple is (u, v, weight) for an edge u -> v.
        n (int): The total number of nodes in the graph.

    Returns:
        tuple[list[list[float]], bool]: A tuple containing:
        - A 2D list of shortest distances. `dist[i][j]` is the shortest
          distance from node `i` to node `j`. `float('inf')` for unreachable pairs.
        - A boolean that is True if a negative cycle is detected, False otherwise.
    """
    if n == 0:
        return [], False

    dist = [[float("inf")] * n for _ in range(n)]

    for i in range(n):
        dist[i][i] = 0

    for u, v, w in edges:
        dist[u][v] = min(dist[u][v], w)
```

```python

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] != float("inf") and
                ↪  dist[k][j] != float("inf"):
                    dist[i][j] = min(dist[i][j],
                    ↪  dist[i][k] + dist[k][j])

    has_negative_cycle = False
    for i in range(n):
        if dist[i][i] < 0:
            has_negative_cycle = True
            break

    return dist, has_negative_cycle

```

Lca Binary Lifting

Implements Lowest Common Ancestor (LCA) queries on a tree using the binary lifting technique. This method allows for finding the LCA of any two nodes in logarithmic time after a precomputation step. The algorithm consists of two main parts:

1. Precomputation:

- A Depth-First Search (DFS) is performed from the root of the tree to calculate the depth of each node and to determine the immediate parent of each node.
- A table `up[i][j]` is built, where `up[i][j]` stores the `2^j`-th ancestor of node `i`. This table is filled using dynamic programming: the `2^j`-th ancestor of `i` is the `2^(j-1)`-th ancestor of its `2^(j-1)`-th ancestor. `up[i][j] = up[up[i][j-1]][j-1]`.

1. Querying for LCA(u, v):

- First, the depths of `u` and `v` are equalized by moving the deeper node upwards. This is done efficiently by "lifting" it in jumps of powers of two.
- If `u` and `v` become the same node, that node is the LCA.
- Otherwise, `u` and `v` are lifted upwards together, step by step, using the largest possible jumps (`2^j`) that keep them below their LCA (i.e., `up[u][j] != up[v][j]`).
- After this process, `u` and `v` will be direct children of the LCA. The LCA is then the parent of `u` (or `v`), which is `up[u][0]`.

```python
class LCA:
    def __init__(self, n, adj, root=0):
        self.n = n
        self.adj = adj
        self.max_log = (n).bit_length()
        self.depth = [-1] * n
        self.up = [[-1] * self.max_log for _ in range(n)]
        self._dfs(root, -1, 0)
        self._precompute_ancestors()

    def _dfs(self, u, p, d):
        self.depth[u] = d
        self.up[u][0] = p
        for v in self.adj[u]:
            if v != p:
                self._dfs(v, u, d + 1)

    def _precompute_ancestors(self):
        for j in range(1, self.max_log):
            for i in range(self.n):
                if self.up[i][j - 1] != -1:
                    self.up[i][j] = self.up[self.up[i][j
                    ↪  - 1]][j - 1]

    def query(self, u, v):
        if self.depth[u] < self.depth[v]:
            u, v = v, u

        for j in range(self.max_log - 1, -1, -1):
            if self.depth[u] - (1 << j) >= self.depth[v]:
                u = self.up[u][j]

        if u == v:
            return u

        for j in range(self.max_log - 1, -1, -1):
            if self.up[u][j] != -1 and self.up[u][j] !=
            ↪  self.up[v][j]:
                u = self.up[u][j]
                v = self.up[v][j]

        return self.up[u][0]

```

Prim Kruskal

This file implements two classic greedy algorithms for finding the Minimum Spanning Tree (MST) of an undirected, weighted graph: Kruskal's algorithm and Prim's algorithm. An MST is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. Kruskal's Algorithm: This algorithm treats the graph as a forest and each node as an individual tree. It sorts all the edges by weight in non-decreasing order. Then, it iterates through the sorted edges, adding an edge to the MST if and only if it does not form a cycle with the edges already added. A Union-Find data structure is used to efficiently detect cycles. The algorithm terminates when V-1 edges have been added to the MST (for a connected graph). Prim's Algorithm: This algorithm grows the MST from an arbitrary starting vertex. It maintains a set of vertices already in the MST. At each step, it finds the minimum-weight edge that connects a vertex in the MST to a vertex outside the MST and adds this edge and vertex to the tree. A priority queue is used to efficiently select this minimum-weight edge.

- Kruskal's: $O(E \log E)$ or $O(E \log V)$, dominated by sorting the edges.
- Prim's: $O(E \log V)$ using a binary heap as a

priority queue.

- Kruskal's: $O(V + E)$ for the edge list and Union-Find structure.
- Prim's: $O(V + E)$ for the adjacency list, priority queue, and visited array.

```python
import heapq
import sys
import os

# Add content directory to path to import the solution
sys.path.append(
    os.path.join(os.path.dirname(__file__), "../../content/data_structures")
)
from union_find import UnionFind


def kruskal(edges, n):
    """
    Finds the MST of a graph using Kruskal's algorithm.

    Args:
        edges (list[tuple[int, int, int]]): A list of all edges in the graph,
            where each tuple is (u, v, weight).
        n (int): The total number of nodes in the graph.

    Returns:
        tuple[int, list[tuple[int, int, int]]]: A tuple containing:
            - The total weight of the MST.
            - A list of edges (u, v, weight) that form the MST.
            Returns (inf, []) if the graph is not connected and cannot form a single MST.
    """
    if n == 0:
        return 0, []

    sorted_edges = sorted([(w, u, v) for u, v, w in edges])
    uf = UnionFind(n)
    mst_weight = 0
    mst_edges = []

    for weight, u, v in sorted_edges:
        if uf.union(u, v):
            mst_weight += weight
            mst_edges.append((u, v, weight))
            if len(mst_edges) == n - 1:
                break

    if len(mst_edges) < n - 1:
        # This indicates the graph is not connected.
        # The result is a minimum spanning forest.
        pass

    return mst_weight, mst_edges


def prim(adj, n, start_node=0):
    """
    Finds the MST of a graph using Prim's algorithm.

    Args:
        adj (list[list[tuple[int, int]]]): The adjacency list representation of
            the graph. adj[u] contains tuples (v, weight) for edges u -> v.
        n (int): The total number of nodes in the graph.
        start_node (int): The node to start building the MST from.

    Returns:
        tuple[int, list[tuple[int, int, int]]]: A tuple containing:
            - The total weight of the MST.
            - A list of edges (u, v, weight) that form the MST.
            Returns (inf, []) if the graph is not connected.
    """
    if n == 0:
        return 0, []
    if not (0 <= start_node < n):
        return float("inf"), []

    visited = [False] * n
    pq = [(0, start_node, -1)]  # (weight, current_node, previous_node)
    mst_weight = 0
    mst_edges = []
    edges_count = 0

    while pq and edges_count < n:
        weight, u, prev = heapq.heappop(pq)

        if visited[u]:
            continue

        visited[u] = True
        mst_weight += weight
        if prev != -1:
            mst_edges.append((prev, u, weight))
        edges_count += 1

        for v, w in adj[u]:
            if not visited[v]:
                heapq.heappush(pq, (w, v, u))

    if edges_count < n:
        # This indicates the graph is not connected.
        return float("inf"), []

    return mst_weight, mst_edges
```

Scc

Implements Tarjan's algorithm for finding Strongly Connected Components (SCCs) in a directed graph. An SCC is a maximal subgraph where for any two vertices u and v in the subgraph, there is a path from u to v and a path from v to u. Tarjan's algorithm performs a single Depth-First Search (DFS) from an arbitrary start node. It maintains two key values for each vertex `u`:

1. `disc[u]`: The discovery time of `u`, which is the time (a counter) when `u` is first visited.
2. `low[u]`: The "low-link" value of `u`, which is the lowest discovery time reachable from `u` (including itself) through its DFS subtree, possibly including one back-edge.

The algorithm also uses a stack to keep track of the nodes in the current exploration path. A node `u` is the root of an SCC if its discovery time is equal to its low-link value (`disc[u] == low[u]`). When such a node is found, all nodes in its SCC are on the top of the stack and can be popped off until `u` is reached. These popped nodes form one complete

SCC. edges, because the algorithm is based on a single DFS traversal. the recursion depth of the DFS.

```python
def find_sccs(adj, n):
    """
    Finds all Strongly Connected Components of a directed graph using Tarjan's algorithm.

    Args:
        adj (list[list[int]]): The adjacency list representation of the graph.
        n (int): The total number of nodes in the graph.

    Returns:
        list[list[int]]: A list of lists, where each inner list contains the
                         nodes of a single Strongly Connected Component.
    """
    if n == 0:
        return []

    disc = [-1] * n
    low = [-1] * n
    on_stack = [False] * n
    stack = []
    time = 0
    sccs = []

    def tarjan_dfs(u):
        nonlocal time
        disc[u] = low[u] = time
        time += 1
        stack.append(u)
        on_stack[u] = True

        for v in adj[u]:
            if disc[v] == -1:
                tarjan_dfs(v)
                low[u] = min(low[u], low[v])
            elif on_stack[v]:
                low[u] = min(low[u], disc[v])

        if low[u] == disc[u]:
            component = []
            while True:
                node = stack.pop()
                on_stack[node] = False
                component.append(node)
                if node == u:
                    break
            sccs.append(component)

    for i in range(n):
        if disc[i] == -1:
            tarjan_dfs(i)

    return sccs

```

Topological Sort

Implements Topological Sort for a Directed Acyclic Graph (DAG). A topological sort or topological ordering of a DAG is a linear ordering of its vertices such that for every directed edge from vertex `u` to vertex `v`, `u` comes before `v` in the ordering. This implementation uses Kahn's algorithm, which is BFS-based. The algorithm proceeds as follows:

1. Compute the in-degree (number of incoming edges) for each vertex.
2. Initialize a queue with all vertices that have an in-degree of 0. These are the starting points of the graph.
3. While the queue is not empty, dequeue a vertex `u`. Add `u` to the result list.
4. For each neighbor `v` of `u`, decrement its in-degree. If the in-degree of `v` becomes 0, it means all its prerequisites have been met, so enqueue `v`.
5. After the loop, if the number of vertices in the result list is equal to the total number of vertices in the graph, the list represents a valid topological sort. If the count is less, it indicates that the graph contains at least one cycle, and a topological sort is not possible. In such a case, this function returns an empty list.

Each vertex is enqueued and dequeued once, and every edge is processed once.

```python
from collections import deque


def topological_sort(adj, n):
    """
    Performs a topological sort on a directed graph.

    Args:
        adj (list[list[int]]): The adjacency list representation of the graph.
        n (int): The total number of nodes in the graph.

    Returns:
        list[int]: A list of nodes in topological order. Returns an empty list
                   if the graph contains a cycle.
    """
    if n == 0:
        return []

    in_degree = [0] * n
    for u in range(n):
        for v in adj[u]:
            in_degree[v] += 1

    q = deque([i for i in range(n) if in_degree[i] == 0])
    topo_order = []

    while q:
        u = q.popleft()
        topo_order.append(u)

        for v in adj[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                q.append(v)

    if len(topo_order) == n:
        return topo_order
    else:
        # Graph has a cycle
        return []

```

Traversal

This file implements Breadth-First Search (BFS) and Depth-First Search (DFS), the two most fundamental graph traversal algorithms. Breadth-First Search (BFS): BFS explores a graph layer by layer from a starting source node. It finds all nodes at a distance of 1 from the source, then all nodes at a distance of 2, and so on. It's guaranteed to find the shortest path from the source to any other node in an unweighted graph. The algorithm proceeds as follows:

1. Initialize a queue and add the `start_node` to it.
2. Initialize a `visited` array or set to keep track of visited nodes, marking the `start_node` as visited.
3. While the queue is not empty, dequeue a node `u`.
4. For each neighbor `v` of `u`, if `v` has not been visited, mark `v` as visited and enqueue it.
5. Repeat until the queue is empty. The collection of dequeued nodes forms the traversal order.

Depth-First Search (DFS): DFS explores a graph by traversing as far as possible along each branch before backtracking. It's commonly used for tasks like cycle detection, topological sorting, and finding connected components. The iterative algorithm is as follows:

1. Initialize a stack and push the `start_node` onto it.
2. Initialize a `visited` array or set, marking the `start_node` as visited.
3. While the stack is not empty, pop a node `u`.
4. For each neighbor `v` of `u`, if `v` has not been visited, mark `v` as visited and push it onto the stack.
5. Repeat until the stack is empty. The collection of popped nodes forms the traversal order.

E is the number of edges. Each vertex and edge is visited exactly once. and the visited array.

```python
from collections import deque


def bfs(adj, start_node, n):
    """
    Performs a Breadth-First Search on a graph.

    Args:
        adj (list[list[int]]): The adjacency list representation of the graph.
        start_node (int): The node from which to start the traversal.
        n (int): The total number of nodes in the graph.

    Returns:
        list[int]: A list of nodes in the order they were visited.
    """
    if not (0 <= start_node < n):
        return []

    q = deque([start_node])
    visited = [False] * n
    visited[start_node] = True
    traversal_order = []

    while q:
        u = q.popleft()
        traversal_order.append(u)
        for v in adj[u]:
            if not visited[v]:
                visited[v] = True
                q.append(v)
    return traversal_order


def dfs(adj, start_node, n):
    """
    Performs a Depth-First Search on a graph.

    Args:
        adj (list[list[int]]): The adjacency list representation of the graph.
        start_node (int): The node from which to start the traversal.
        n (int): The total number of nodes in the graph.

    Returns:
        list[int]: A list of nodes in the order they were visited.
    """
    if not (0 <= start_node < n):
        return []

    stack = [start_node]
    visited = [False] * n
    # Mark as visited when pushed to stack to avoid re-adding
    visited[start_node] = True
    traversal_order = []

    # This loop produces a traversal order different from the recursive one.
    # To get a more standard pre-order traversal iteratively, we need a slight change.

    # Reset for a more standard iterative DFS traversal order
    visited = [False] * n
    stack = [start_node]

    while stack:
        u = stack.pop()

        if not visited[u]:
            visited[u] = True
            traversal_order.append(u)

            # Add neighbors to the stack in reverse order to process them in lexicographical order
            for v in reversed(adj[u]):
                if not visited[v]:
                    stack.append(v)

    return traversal_order
```

Two Sat

Implements a solver for 2-Satisfiability (2-SAT) problems. A 2-SAT problem consists of a boolean formula in 2-Conjunctive Normal Form, which is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of two literals. The goal is to find a satisfying assignment of true/false values to the variables. This problem can be solved in linear time by reducing it to a graph problem. The reduction works as follows:

1. Create an "implication graph" with `2N` vertices for `N` variables. For each variable `x_i`, there are two vertices: one for `x_i` and one for its negation `¬x_i`.

2. Each clause `(a OR b)` is equivalent to two implications: `(¬a => b)` and `(¬b => a)`. For each clause, add two directed edges to the graph representing these implications.

3. The original 2-SAT formula is unsatisfiable if and only if there exists a variable `x_i` such that `x_i` and `¬x_i` are in the same Strongly Connected Component (SCC) of the implication graph. This is because if they are in the same SCC, it means `x_i` implies `¬x_i` and `¬x_i` implies `x_i`, which is a contradiction.

4. If the formula is satisfiable, a valid assignment can be constructed from the SCCs. The SCCs form a Directed Acyclic Graph (DAG). We can find a reverse topological ordering of this "condensation graph". For each variable `x_i`, if the SCC containing `¬x_i` appears before the SCC containing `x_i` in this ordering, we must assign `x_i` to true. Otherwise, we assign it to false.

This implementation uses the `find_sccs` function (Tarjan's algorithm) to solve the problem. number of clauses. The graph has $2N$ vertices and $2M$ edges.

```python
import sys
import os

# The stress test runner adds the project root to the path.
# This allows importing other content modules using their full path.
from content.graph.scc import find_sccs


class TwoSAT:
    def __init__(self, n):
        self.n = n
        self.graph = [[] for _ in range(2 * n)]

    def _map_var(self, var):
        """Maps a 1-indexed variable to a 0-indexed graph node."""
        if var > 0:
            return var - 1
        return -var - 1 + self.n

    def add_clause(self, i, j):
        """
        Adds a clause (i OR j) to the formula.
        Variables are 1-indexed. A negative value -k denotes the negation of x_k.
        This adds two implications: (-i => j) and (-j => i).
        """
        # Add edge for (-i => j)
        self.graph[self._map_var(-i)].append(self._map_var(j))
        # Add edge for (-j => i)
        self.graph[self._map_var(-j)].append(self._map_var(i))

    def solve(self):
        """
        Solves the 2-SAT problem.

        Returns:
            tuple[bool, list[bool] | None]: A tuple where the first element is
            True if a solution exists, False otherwise. If a solution exists,
            the second element is a list of boolean values representing a
            satisfying assignment. Otherwise, it is None.
        """
        sccs = find_sccs(self.graph, 2 * self.n)
        component_id = [-1] * (2 * self.n)
        for idx, comp in enumerate(sccs):
            for node in comp:
                component_id[node] = idx

        for i in range(self.n):
            if component_id[i] == component_id[i + self.n]:
                return False, None

        assignment = [False] * self.n
        # sccs are returned in reverse topological order
        for i in range(self.n):
            # If component of x_i comes after component of not(x_i) in topo order
            # (i.e., has a smaller index in the reversed list), then x_i must be true.
            if component_id[i] < component_id[i + self.n]:
                assignment[i] = True

        return True, assignment
```

Chapter 6

String Algorithms

Aho Corasick

Implements the Aho-Corasick algorithm for finding all occurrences of multiple patterns in a text simultaneously. This algorithm combines a trie (prefix tree) with failure links to achieve linear time complexity with respect to the sum of the text length and the total length of all patterns. The algorithm works in two main stages:

1. Preprocessing (Building the Automaton): a. A trie is constructed from the set of all patterns. Each node in the trie represents a prefix of one or more patterns. b. An `output` list is associated with each node, storing the indices of patterns that end at that node. c. "Failure links" are computed for each node. The failure link of a node `u` points to the longest proper suffix of the string corresponding to `u` that is also a prefix of some pattern in the set. These links are computed using a Breadth-First Search (BFS) starting from the root.

2. Searching: a. The algorithm processes the text character by character, traversing the automaton. It starts at the root. b. For each character in the text, it transitions to the next state. If a direct child for the character does not exist, it follows failure links until a valid transition is found or it returns to the root. c. At each state, it collects all matches. This is done by checking the `output` of the current node and recursively following failure links to find all patterns that end as a suffix of the current prefix.

Searching is $O(N+Z)$, where N is the length of the text and Z is the total number of matches found.

```python
from collections import deque


class AhoCorasick:
    def __init__(self, patterns):
        self.patterns = patterns
        self.trie = [{"children": {}, "output": [], "fail_link": 0}]
        self._build_trie()
        self._build_failure_links()

    def _build_trie(self):
        for i, pattern in enumerate(self.patterns):
            node_idx = 0
            for char in pattern:
                if char not in self.trie[node_idx]["children"]:
                    self.trie[node_idx]["children"][char] = len(self.trie)
                    self.trie.append({"children": {}, "output": [], "fail_link": 0})
                node_idx = self.trie[node_idx]["children"][char]
            self.trie[node_idx]["output"].append(i)

    def _build_failure_links(self):
        q = deque()
        for char, next_node_idx in self.trie[0]["children"].items():
            q.append(next_node_idx)

        while q:
            curr_node_idx = q.popleft()
            for char, next_node_idx in self.trie[curr_node_idx]["children"].items():
                fail_idx = self.trie[curr_node_idx]["fail_link"]
                while char not in self.trie[fail_idx]["children"] and fail_idx != 0:
                    fail_idx = self.trie[fail_idx]["fail_link"]

                if char in self.trie[fail_idx]["children"]:
                    self.trie[next_node_idx]["fail_link"] = self.trie[fail_idx][
                        "children"
                    ][char]
                else:
                    self.trie[next_node_idx]["fail_link"] = 0

                # Append outputs from the failure link node
                fail_output_idx = self.trie[next_node_idx]["fail_link"]
                self.trie[next_node_idx]["output"].extend(
                    self.trie[fail_output_idx]["output"]
                )
                q.append(next_node_idx)

    def search(self, text):
        """
        Finds all occurrences of the patterns in the given text.

        Args:
            text (str): The text to search within.

        Returns:
            list[tuple[int, int]]: A list of tuples, where each tuple is
            (pattern_index, end_index_in_text). `end_index_in_text` is the
            index where the pattern ends.
        """
        matches = []
        curr_node_idx = 0
        for i, char in enumerate(text):
            while (
                char not in self.trie[curr_node_idx]["children"]
                and curr_node_idx != 0
            ):
                curr_node_idx = self.trie[curr_node_idx]["fail_link"]
```

```python
            if char in self.trie[curr_node_idx]["children"]:
                curr_node_idx = self.trie[curr_node_idx]["children"][char]
            else:
                curr_node_idx = 0

            if self.trie[curr_node_idx]["output"]:
                for pattern_idx in self.trie[curr_node_idx]["output"]:
                    matches.append((pattern_idx, i))
        return matches
```

Kmp

Implements the Knuth-Morris-Pratt (KMP) algorithm for efficient string searching. KMP finds all occurrences of a pattern `P` within a text `T` in linear time. The core of the KMP algorithm is the precomputation of a "prefix function" or Longest Proper Prefix Suffix (LPS) array for the pattern. The LPS array, `lps`, for a pattern of length `M` stores at each index `i` the length of the longest proper prefix of `P[0...i]` that is also a suffix of `P[0...i]`. A "proper" prefix is one that is not equal to the entire string. Example: For pattern `P = "ababa"`, the LPS array is `[0, 0, 1, 2, 3]`.

- `lps[0]` is always 0.
- `lps[1]` ("ab"): No proper prefix is a suffix. Length is 0.
- `lps[2]` ("aba"): "a" is both a prefix and a suffix. Length is 1.
- `lps[3]` ("abab"): "ab" is both a prefix and a suffix. Length is 2.
- `lps[4]` ("ababa"): "aba" is both a prefix and a suffix. Length is 3.

During the search, when a mismatch occurs between the text and the pattern at `text[i]` and `pattern[j]`, the LPS array tells us how many characters of the pattern we can shift without rechecking previously matched characters. Specifically, if a mismatch occurs at `pattern[j]`, we know that the prefix `pattern[0...j-1]` matched the text. The value `lps[j-1]` gives the length of the longest prefix of `pattern[0...j-1]` that is also a suffix. This means we can shift the pattern and continue the comparison from `pattern[lps[j-1]]` without losing any potential matches. the pattern. $O(M)$ for building the LPS array and $O(N)$ for the search.

```python
def compute_lps(pattern):
    """
    Computes the Longest Proper Prefix Suffix (LPS) array for the KMP algorithm.

    Args:
        pattern (str): The pattern string.

    Returns:
        list[int]: The LPS array for the pattern.
    """
    m = len(pattern)
    lps = [0] * m
    length = 0
    i = 1
    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
    return lps


def kmp_search(text, pattern):
    """
    Finds all occurrences of a pattern in a text using the KMP algorithm.

    Args:
        text (str): The text to search within.
        pattern (str): The pattern to search for.

    Returns:
        list[int]: A list of 0-based starting indices of all occurrences
                   of the pattern in the text.
    """
    n = len(text)
    m = len(pattern)
    if m == 0:
        return list(range(n + 1))
    if n == 0 or m > n:
        return []

    lps = compute_lps(pattern)
    occurrences = []
    i = 0
    j = 0
    while i < n:
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == m:
            occurrences.append(i - j)
            j = lps[j - 1]
        elif i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
    return occurrences
```

Manacher

Implements Manacher's algorithm for finding the longest palindromic substring in a given string in linear time. Standard naive approaches take $O(N^2)$ or $O(N^3)$ time. The algorithm cleverly handles both odd and even length palindromes by transforming the input string. A special character (e.g., '#') is inserted between each character and at the ends. For example, "aba" becomes "#a#b#a#" and "abba" becomes "#a#b#b#a#". In this new string, every palindrome, regardless of its original length, is of odd length and has a distinct center. The core of the algorithm is to compute an array `p`, where `p[i]` stores the radius of the palindrome centered at index `i` in the transformed string. It does this efficiently by maintaining the center `c` and right boundary `r` of the palindrome that extends furthest to the right. When computing `p[i]`, it uses the information from the mirror position `i_mirror = 2*c - i` to get an initial guess for `p[i]`. It then expands from this guess, avoiding redundant character comparisons. This optimization is what brings the complexity down to linear time. After computing the `p` array, the maximum value in `p` corresponds to the radius of the longest palindromic substring. From this radius and its center, the original substring can be reconstructed.

```python
def manacher(s):
    """
    Finds the longest palindromic substring in a string using Manacher's algorithm.

    Args:
        s (str): The input string.

    Returns:
        str: The longest palindromic substring found in `s`. If there are
            multiple of the same maximum length, it returns the first one found.
    """
    if not s:
        return ""

    t = "#" + "#".join(s) + "#"
    n = len(t)
    p = [0] * n
    center, right = 0, 0
    max_len, max_center = 0, 0

    for i in range(n):
        mirror = 2 * center - i

        if i < right:
            p[i] = min(right - i, p[mirror])

        while (
            i - (p[i] + 1) >= 0
            and i + (p[i] + 1) < n
            and t[i - (p[i] + 1)] == t[i + (p[i] + 1)]
        ):
            p[i] += 1

        if i + p[i] > right:
            center = i
            right = i + p[i]

        if p[i] > max_len:
            max_len = p[i]
            max_center = i

    start = (max_center - max_len) // 2
    end = start + max_len
    return s[start:end]
```

Polynomial Hashing

Implements a string hashing class using the polynomial rolling hash technique. This allows for efficient comparison of substrings. After an initial $O(N)$ precomputation on a string of length N, the hash of any substring can be calculated in $O(1)$ time. The hash of a string $s = s_0 s_1 ... s_{k-1}$ is defined as: $H(s) = (s_0 p^0 + s_1 p^1 + ... + s_{k-1} p^{k-1}) \mod m$ where `p` is a base and `m` is a large prime modulus. To prevent collisions, especially against adversarial test cases, this implementation uses two key techniques:

1. Randomized Base: The base `p` is chosen randomly at runtime. It should be larger than the size of the character set.
2. Multiple Moduli: Hashing is performed with two different large prime moduli (`m1`, `m2`). Two substrings are considered equal only if their hash values match for both moduli. This drastically reduces the probability of collisions.

The `query(l, r)` method calculates the hash of the substring `s[l...r-1]` by using precomputed prefix hashes and powers of `p`.

```python
import random


class StringHasher:
    def __init__(self, s):
        self.s = s
        self.n = len(s)

        self.m1 = 10**9 + 7
        self.m2 = 10**9 + 9

        self.p = random.randint(257, self.m1 - 1)

        self.p_powers1 = [1] * (self.n + 1)
        self.p_powers2 = [1] * (self.n + 1)
        for i in range(1, self.n + 1):
            self.p_powers1[i] = (self.p_powers1[i - 1] * self.p) % self.m1
            self.p_powers2[i] = (self.p_powers2[i - 1] * self.p) % self.m2

        self.h1 = [0] * (self.n + 1)
        self.h2 = [0] * (self.n + 1)
        for i in range(self.n):
            self.h1[i + 1] = (self.h1[i] * self.p + ord(self.s[i])) % self.m1
            self.h2[i + 1] = (self.h2[i] * self.p + ord(self.s[i])) % self.m2

    def query(self, l, r):
        """
        Computes the hash of the substring s[l...r-1].

        Args:
            l (int): The 0-based inclusive starting index.
            r (int): The 0-based exclusive ending index.

        Returns:
            tuple[int, int]: A tuple containing the two hash values for the substring.
        """
        if l >= r:
            return 0, 0

```

```python
        len_sub = r - l
        hash1 = (
            self.h1[r] - (self.h1[l] * self.p_powers1[len_sub]) % self.m1 + self.m1
        ) % self.m1
        hash2 = (
            self.h2[r] - (self.h2[l] * self.p_powers2[len_sub]) % self.m2 + self.m2
        ) % self.m2
        return hash1, hash2

```

Suffix Array

Implements the construction of a Suffix Array and a Longest Common Prefix (LCP) Array. A suffix array is a sorted array of all suffixes of a given string. The LCP array stores the length of the longest common prefix between adjacent suffixes in the sorted suffix array. Suffix Array Construction ($O(N \log^2 N)$): The algorithm works by repeatedly sorting the suffixes based on prefixes of increasing lengths that are powers of two.

1. Initially, suffixes are sorted based on their first character.
2. In the k-th iteration, suffixes are sorted based on their first 2^k characters. This is done efficiently by using the ranks from the previous iteration. Each suffix `s[i:]` is represented by a pair of ranks: the rank of its first 2^{k-1} characters and the rank of the next 2^{k-1} characters (starting at `s[i + 2^{k-1}:]`).
3. This process continues for $\log N$ iterations, with each sort taking $O(N \log N)$ time, leading to an overall complexity of $O(N \log^2 N)$.

LCP Array Construction (Kasai's Algorithm, $O(N)$): After the suffix array `sa` is built, the LCP array can be constructed in linear time using Kasai's algorithm. The algorithm utilizes the observation that the LCP of two suffixes `s[i:]` and `s[j:]` is related to the LCP of `s[i-1:]` and `s[j-1:]`. It processes the suffixes in their original order in the string, not the sorted order, which allows it to compute the LCP values efficiently. Total time complexity is dominated by the suffix array construction.

```python
def build_suffix_array(s):
    """
    Builds the suffix array for a string using an O(N log^2 N) sorting-based approach.

    Args:
        s (str): The input string.

    Returns:
        list[int]: The suffix array, containing starting indices of suffixes in
                   lexicographically sorted order.
    """
    n = len(s)
    sa = list(range(n))
    rank = [ord(c) for c in s]
    k = 1
    while k < n:
        sa.sort(key=lambda i: (rank[i], rank[i + k] if i + k < n else -1))
        new_rank = [0] * n
        new_rank[sa[0]] = 0
        for i in range(1, n):
            prev, curr = sa[i - 1], sa[i]
            r_prev = (rank[prev], rank[prev + k] if prev + k < n else -1)
            r_curr = (rank[curr], rank[curr + k] if curr + k < n else -1)
            if r_curr == r_prev:
                new_rank[curr] = new_rank[prev]
            else:
                new_rank[curr] = new_rank[prev] + 1
        rank = new_rank
        if rank[sa[-1]] == n - 1:
            break
        k *= 2
    return sa


def build_lcp_array(s, sa):
    """
    Builds the LCP array using Kasai's algorithm in O(N) time.

    Args:
        s (str): The input string.
        sa (list[int]): The suffix array for the string `s`.

    Returns:
        list[int]: The LCP array. `lcp[i]` is the LCP of suffixes `sa[i-1]` and `sa[i]`.
                   `lcp[0]` is conventionally 0.
    """
    n = len(s)
    if n == 0:
        return []

    rank = [0] * n
    for i in range(n):
        rank[sa[i]] = i

    lcp = [0] * n
    h = 0
    for i in range(n):
        if rank[i] == 0:
            continue
        j = sa[rank[i] - 1]
        if h > 0:
            h -= 1
        while i + h < n and j + h < n and s[i + h] == s[j + h]:
            h += 1
        lcp[rank[i]] = h
    return lcp

```

Z Algorithm

Implements the Z-algorithm, which computes the Z-array for a given string `s` of length `N`. The Z-array `z` is an array of length `N` where `z[i]` is the length of the longest common prefix (LCP) between the original string `s` and the suffix of `s` starting at index `i`. By convention, `z[0]` is usually set to 0 or `N`; here it is set to 0. The algorithm computes the Z-array in linear time. It does this by maintaining the bounds of the rightmost substring that is also a prefix of `s`. This is called the "Z-box", denoted by `[l, r]`. The algorithm iterates from `i = 1` to `N-1`:

1. If `i` is outside the current Z-box (`i > r`), it computes `z[i]` naively by comparing characters from the start of the string and from index `i`. It then updates the Z-box `[l, r]` if a new rightmost one is found.
2. If `i` is inside the current Z-box (`i <= r`), it can use previously computed Z-values to initialize `z[i]`. Let `k = i - l`. `z[i]` can be at least `min(z[k], r - i + 1)`.

- If `z[k] < r - i + 1`, then `z[i]` is exactly `z[k]`, and the Z-box does not change.
- If `z[k] >= r - i + 1`, it means `z[i]` might be even longer. The algorithm then continues comparing characters from `r+1` onwards to extend the match and updates the Z-box `[l, r]`.

The Z-algorithm is very powerful for pattern matching. To find a pattern `P` in a text `T`, one can compute the Z-array for the concatenated string ‘P + $'' + T$‘, *where*‘ `is a character not in` P `or` T`. Any` z[i] `equal to the length of` P `indicates an occurrence of` P `in` T‘.

```python
def z_function(s):
    """
    Computes the Z-array for a given string.

    Args:
        s (str): The input string.

    Returns:
        list[int]: The Z-array for the string `s`.
    """
    n = len(s)
    if n == 0:
        return []

    z = [0] * n
    l, r = 0, 0
    for i in range(1, n):
        if i <= r:
            z[i] = min(r - i + 1, z[i - l])
        while i + z[i] < n and s[z[i]] == s[i + z[i]]:
            z[i] += 1
        if i + z[i] - 1 > r:
            l, r = i, i + z[i] - 1
    return z

```

Chapter 7

Mathematics & Number Theory

Chinese Remainder Theorem

Implements a solver for a system of linear congruences using the Chinese Remainder Theorem (CRT). Given a system of congruences: $x \equiv a_1 \pmod{n_1}$ $x \equiv a_2 \pmod{n_2}$ ... $x \equiv a_k \pmod{n_k}$ the algorithm finds a solution `x` that satisfies all of them. This implementation handles the general case where the moduli `n_i` are not necessarily pairwise coprime. The algorithm works by iteratively combining pairs of congruences. Given a solution for the first `i-1` congruences, `x \equiv a_{res} (mod n_{res})`, it combines this with the i-th congruence `x \equiv a_i (mod n_i)`. This requires solving a linear congruence of the form `k * n_{res} \equiv a_i - a_{res} (mod n_i)`. A solution exists if and only if `(a_i - a_{res})` is divisible by `g = gcd(n_{res}, n_i)`. If a solution exists, the two congruences are merged into a new one: `x \equiv a_{new} (mod n_{new})`, where `n_{new} = lcm(n_{res}, n_i)`. This process is repeated for all congruences. If at any step a solution does not exist, the entire system has no solution. Each merge step involves `extended_gcd`, which is logarithmic.

```python
from content.math.modular_arithmetic import extended_gcd


def chinese_remainder_theorem(remainders, moduli):
    """
    Solves a system of linear congruences.
    `x \equiv remainders[i] (mod moduli[i])` for all i.

    Args:
        remainders (list[int]): A list of remainders (a_i).
        moduli (list[int]): A list of moduli (n_i).

    Returns:
        tuple[int, int] | None: A tuple `(result, lcm)` representing the solution
        `x \equiv result (mod lcm)`, or None if no solution exists.
    """
    if not remainders or not moduli or len(remainders) != len(moduli):
        return 0, 1

    a1 = remainders[0]
    n1 = moduli[0]

    for i in range(1, len(remainders)):
        a2 = remainders[i]
        n2 = moduli[i]

        g, x, _ = extended_gcd(n1, n2)

        if (a1 - a2) % g != 0:
            return None

        # Solve k * n1 \equiv a2 - a1 (mod n2)
        # k * (n1/g) \equiv (a2 - a1)/g (mod n2/g)
        # k \equiv ((a2 - a1)/g) * inv(n1/g) (mod n2/g)
        # inv(n1/g) mod (n2/g) is x from extended_gcd(n1, n2)
        k0 = (x * ((a2 - a1) // g)) % (n2 // g)

        # New solution: x = a1 + k*n1. With k = k0 + t*(n2/g)
        # x = a1 + (k0 + t*(n2/g)) * n1 = (a1 + k0*n1) + t*lcm(n1, n2)
        a1 = a1 + k0 * n1
        n1 = n1 * (n2 // g)  # lcm(n1, n2)
        a1 %= n1

    return a1, n1
```

Miller Rabin

Implements the Miller-Rabin primality test, a probabilistic algorithm for determining whether a given number is prime. It is highly efficient and is the standard method for primality testing in competitive programming for numbers that are too large for a sieve. The algorithm is based on properties of square roots of unity modulo a prime number and Fermat's Little Theorem. For a number `n` to be tested, we first write `n - 1` as `2^s * d`, where `d` is odd. The test then proceeds:

1. Pick a base `a` (a "witness").
2. Compute `x = a^d mod n`.
3. If `x == 1` or `x == n - 1`, `n` might be prime, and this test passes for this base.
4. Otherwise, for `s-1` times, compute `x = x^2 mod n`. If `x` becomes `n - 1`, the test passes for this base.
5. If after these steps, `x` is not `n - 1`, then `n` is definitely composite.

If `n` passes this test for multiple well-chosen bases `a`, it is prime with a very high probability. For 64-bit integers, a specific set of deterministic witnesses can be used to make the test 100% accurate. This implementation uses such a set, making it reliable for contest use.

```python
from content.math.modular_arithmetic import power


def is_prime(n):
    """
    Checks if a number is prime using the Miller-Rabin primality test.
```

```python
    This implementation is deterministic for all integers up to 2^64.

    Args:
        n (int): The number to test for primality.

    Returns:
        bool: True if n is prime, False otherwise.
    """
    if n < 2:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False

    d = n - 1
    s = 0
    while d % 2 == 0:
        d //= 2
        s += 1

    # A set of witnesses that is deterministic for all 64-bit integers.
    witnesses = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]

    for a in witnesses:
        if a >= n:
            break
        x = power(a, d, n)
        if x == 1 or x == n - 1:
            continue

        is_composite = True
        for _ in range(s - 1):
            x = power(x, 2, n)
            if x == n - 1:
                is_composite = False
                break
        if is_composite:
            return False

    return True
```

Modular Arithmetic

This module provides essential functions for modular arithmetic, a cornerstone of number theory in competitive programming. It includes modular exponentiation, the Extended Euclidean Algorithm, and modular multiplicative inverse. Modular Exponentiation: The `power` function computes $(base^{exp}) \pmod{mod}$ efficiently using the binary exponentiation (also known as exponentiation by squaring) method. This avoids the massive intermediate numbers that would result from calculating $base^{exp}$ directly. The time complexity is logarithmic in the exponent. Extended Euclidean Algorithm: The `extended_gcd` function computes the greatest common divisor (GCD) of two integers `a` and `b`. In addition, it finds two integer coefficients, `x` and `y`, that satisfy Bezout's identity: $a \cdot x + b \cdot y = \gcd(a, b)$. This is fundamental for many number-theoretic calculations. Modular Multiplicative Inverse: The `mod_inverse` function finds a number `x` such that $(a \cdot x) \equiv 1 \pmod{m}$. This `x` is the modular multiplicative inverse of `a` modulo `m`. An inverse exists if and only if `a` and `m` are coprime (i.e., $\gcd(a, m) = 1$). This implementation uses the Extended Euclidean Algorithm. From $a \cdot x + m \cdot y = 1$, taking the equation modulo `m` gives $a \cdot x \equiv 1 \pmod{m}$. Thus, the coefficient `x` is the desired inverse.

- `power`: $O(log(exp))$
- `extended_gcd`: $O(log(min(a, b)))$
- `mod_inverse`: $O(logm)$
- All functions use $O(1)$ extra space for iterative versions.

```python
def power(base, exp, mod):
    """
    Computes (base^exp) % mod using binary exponentiation.
    """
    res = 1
    base %= mod
    while exp > 0:
        if exp % 2 == 1:
            res = (res * base) % mod
        base = (base * base) % mod
        exp //= 2
    return res


def extended_gcd(a, b):
    """
    Returns (gcd, x, y) such that a*x + b*y = gcd(a, b).
    """
    if a == 0:
        return b, 0, 1
    gcd, x1, y1 = extended_gcd(b % a, a)
    x = y1 - (b // a) * x1
    y = x1
    return gcd, x, y


def mod_inverse(a, m):
    """
    Computes the modular multiplicative inverse of a modulo m.
    Returns None if the inverse does not exist.
    """
    gcd, x, y = extended_gcd(a, m)
    if gcd != 1:
        return None
    else:
        return (x % m + m) % m
```

Ntt

Implements the Number Theoretic Transform (NTT) for fast polynomial multiplication over a finite field. NTT is an adaptation of the Fast Fourier Transform (FFT) for modular arithmetic, avoiding floating-point precision issues. It is commonly used in problems involving polynomial convolution, such as multiplying large numbers or finding the number of ways to form a sum. The algorithm works by:

1. Choosing a prime modulus `MOD` of the form `c * 2^k + 1` and a primitive root `ROOT` of `MOD`.
2. Evaluating the input polynomials at the powers of `ROOT` (the "roots of unity"). This is the forward NTT, which transforms the polynomials from coefficient representation to point-value representation in $O(N \log N)$ time.
3. Multiplying the resulting point-value representations element-wise in $O(N)$ time.
4. Interpolating the resulting polynomial back to coefficient representation using the inverse NTT in $O(N \log N)$ time.

This implementation uses the prime `MOD = 998244353`, which is a standard choice in competitive programming.

```python
from content.math.modular_arithmetic import power

MOD = 998244353
ROOT = 3
ROOT_PW = 1 << 23
ROOT_INV = power(ROOT, MOD - 2, MOD)


def ntt(a, invert):
    n = len(a)
    j = 0
    for i in range(1, n):
        bit = n >> 1
        while j & bit:
            j ^= bit
            bit >>= 1
        j ^= bit
        if i < j:
            a[i], a[j] = a[j], a[i]

    length = 2
    while length <= n:
        wlen = power(ROOT_INV if invert else ROOT, (MOD - 1) // length, MOD)
        i = 0
        while i < n:
            w = 1
            for j in range(length // 2):
                u = a[i + j]
                v = (a[i + j + length // 2] * w) % MOD
                a[i + j] = (u + v) % MOD
                a[i + j + length // 2] = (u - v + MOD) % MOD
                w = (w * wlen) % MOD
            i += length
        length <<= 1

    if invert:
        n_inv = power(n, MOD - 2, MOD)
        for i in range(n):
            a[i] = (a[i] * n_inv) % MOD


def multiply(a, b):
    if not a or not b:
        return []

    res_len = len(a) + len(b) - 1
    n = 1
    while n < res_len:
        n <<= 1

    fa = a[:] + [0] * (n - len(a))
    fb = b[:] + [0] * (n - len(b))

    ntt(fa, False)
    ntt(fb, False)

    for i in range(n):
        fa[i] = (fa[i] * fb[i]) % MOD

    ntt(fa, True)

    return fa[:res_len]
```

Pollard Rho

Implements Pollard's Rho algorithm for integer factorization, combined with Miller-Rabin primality test for a complete factorization routine. Pollard's Rho is a probabilistic algorithm to find a non-trivial factor of a composite number `n`. It's particularly efficient at finding small factors. The algorithm uses Floyd's cycle-detection algorithm on a sequence of pseudorandom numbers modulo `n`, defined by $x_{i+1} = (x_i^2 + c)$ *modn*. A factor is likely found when `\\text{gcd}(|x_j - x_i|, n) > 1`. The `factorize` function returns a sorted list of prime factors of a given number `n`. It first checks for primality using Miller-Rabin. If `n` is composite, it uses Pollard's Rho to find one factor `d`, and then recursively factorizes `d` and `n/d`.

```python
import math
import random
from content.math.miller_rabin import is_prime


def _pollard_rho_factor(n):
    """Finds a non-trivial factor of n using Pollard's Rho. n must be composite."""
    if n % 2 == 0:
        return 2

    f = lambda val, c: (pow(val, 2, n) + c) % n

    while True:
        x = random.randint(1, n - 2)
        y = x
        c = random.randint(1, n - 1)
        d = 1

        while d == 1:
            x = f(x, c)
            y = f(f(y, c), c)
            d = math.gcd(abs(x - y), n)

        if d != n:
            return d


def factorize(n):
    if n <= 1:
        return []

    factors = []

    def get_factors(num):
        if num <= 1:
            return
        if is_prime(num):
            factors.append(num)
```

```python
            return

        factor = _pollard_rho_factor(num)
        get_factors(factor)
        get_factors(num // factor)

    get_factors(n)
    factors.sort()
    return factors
```

Sieve

Implements the Sieve of Eratosthenes, a highly efficient algorithm for finding all prime numbers up to a specified integer `n`. The algorithm works by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2.

1. Create a boolean list `is_prime` of size `n+1`, initializing all entries to `True`. `is_prime[0]` and `is_prime[1]` are set to `False`.
2. Iterate from `p = 2` up to `sqrt(n)`.
3. If `is_prime[p]` is still `True`, then `p` is a prime number.
4. For this prime `p`, iterate through its multiples starting from `p*p` (i.e., `p*p`, `p*p + p`, `p*p + 2p`, ...) and mark them as not prime by setting `is_prime[multiple]` to `False`. We can start from `p*p` because any smaller multiple `k*p` where `k < p` would have already been marked by a smaller prime factor `k`.
5. After the loop, the `is_prime` array contains `True` at indices that are prime numbers and `False` otherwise.

This implementation returns the boolean array itself, which is often more versatile in contests than a list of primes (e.g., for quick primality checks). A list of primes can be easily generated from this array if needed.

```python
def sieve(n):
    """
    Generates a sieve of primes up to n using the Sieve of Eratosthenes.

    Args:
        n (int): The upper limit for the sieve (inclusive).

    Returns:
        list[bool]: A boolean list of size n+1 where is_prime[i] is True if i
                    is a prime number, and False otherwise.
    """
    if n < 2:
        return [False] * (n + 1)

    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False

    for p in range(2, int(n**0.5) + 1):
        if is_prime[p]:
            for multiple in range(p * p, n + 1, p):
                is_prime[multiple] = False

    return is_prime
```

Chapter 8

Geometry

Convex Hull

Implements the Monotone Chain algorithm (also known as Andrew's algorithm) to find the convex hull of a set of 2D points. The convex hull is the smallest convex polygon that contains all the given points. The algorithm works as follows:

1. Sort all points lexicographically (first by x-coordinate, then by y-coordinate). This step takes $O(N \log N)$ time.
2. Build the lower hull of the polygon. Iterate through the sorted points and maintain a list representing the lower hull. For each point, check if adding it to the hull would create a non-left (i.e., clockwise or collinear) turn with the previous two points on the hull. If it does, pop the last point from the hull until the turn becomes counter-clockwise. This ensures the convexity of the lower hull.
3. Build the upper hull in a similar manner, but by iterating through the sorted points in reverse order.
4. Combine the lower and upper hulls to form the complete convex hull. The endpoints (the lexicographically smallest and largest points) will be included in both hulls, so they must be removed from one to avoid duplication.

This implementation relies on the `Point` class and `orientation` primitive from the `content.geometry.point` module.

```python
from content.geometry.point import Point, orientation


def convex_hull(points):
    """
    Computes the convex hull of a set of points using the
    ↪ Monotone Chain algorithm.

    Args:
        points (list[Point]): A list of Point objects.

    Returns:
        list[Point]: A list of Point objects representing
        ↪ the vertices of the
                     convex hull in counter-clockwise
                     ↪ order. Returns an empty
                     list if fewer than 3 points are
                     ↪ provided.
    """
    n = len(points)
    if n <= 2:
        return points

    # Sort points lexicographically
    points.sort()

    # Build lower hull
    lower_hull = []
    for p in points:
        while (
            len(lower_hull) >= 2 and
            ↪ orientation(lower_hull[-2],
            ↪ lower_hull[-1], p) <= 0
        ):
            lower_hull.pop()
        lower_hull.append(p)

    # Build upper hull
    upper_hull = []
    for p in reversed(points):
        while (
            len(upper_hull) >= 2 and
            ↪ orientation(upper_hull[-2],
            ↪ upper_hull[-1], p) <= 0
        ):
            upper_hull.pop()
        upper_hull.append(p)

    # Combine the hulls, removing duplicate start/end
    ↪ points
    return lower_hull[:-1] + upper_hull[:-1]
```

Line Intersection

Provides functions for detecting and calculating intersections between lines and line segments in 2D space. This is a fundamental component for many geometric algorithms. The module includes:

- `segments_intersect(p1, q1, p2, q2)`: Determines if two line segments intersect. It uses orientation tests to handle the general case where segments cross each other. If the orientations of the endpoints of one segment with respect to the other segment are different, they intersect. Special handling is required for collinear cases, where we check if the segments overlap.
- `line_line_intersection(p1, p2, p3, p4)`: Finds the intersection point of two infinite lines defined by pairs of points `(p1, p2)` and `(p3, p4)`. It uses a formula based on cross products to solve the system of linear equations representing the lines. This method returns `None` if the lines are parallel or collinear, as there is no unique intersection point.

All functions rely on the `Point` class and `orientation` primitive from `content.geometry.point`.

```python
from content.geometry.point import Point, orientation


def on_segment(p, q, r):
    """
    Given three collinear points p, q, r, the function
    ↪ checks if point q
    lies on line segment 'pr'.
    """
    return (
        q.x <= max(p.x, r.x)
        and q.x >= min(p.x, r.x)
        and q.y <= max(p.y, r.y)
        and q.y >= min(p.y, r.y)
    )


def segments_intersect(p1, q1, p2, q2):
    """
    Checks if line segment 'p1q1' and 'p2q2' intersect.
    """
    o1 = orientation(p1, q1, p2)
    o2 = orientation(p1, q1, q2)
    o3 = orientation(p2, q2, p1)
    o4 = orientation(p2, q2, q1)

    if o1 != 0 and o2 != 0 and o3 != 0 and o4 != 0:
        if o1 != o2 and o3 != o4:
            return True
        return False

    if o1 == 0 and on_segment(p1, p2, q1):
        return True
    if o2 == 0 and on_segment(p1, q2, q1):
        return True
    if o3 == 0 and on_segment(p2, p1, q2):
        return True
    if o4 == 0 and on_segment(p2, q1, q2):
        return True

    return False


def line_line_intersection(p1, p2, p3, p4):
    """
    Finds the intersection point of two infinite lines
    ↪ defined by (p1, p2) and (p3, p4).
    Returns the intersection point as a Point object with
    ↪ float coordinates,
    or None if the lines are parallel or collinear.
    """
    v1 = p2 - p1
    v2 = p4 - p3
    denominator = v1.cross(v2)

    if abs(denominator) < 1e-9:
        return None

    t = (p3 - p1).cross(v2) / denominator
    return p1 + v1 * t

```

Point

Implements a foundational Point class for 2D geometry problems. The class supports standard vector operations through overloaded operators, making geometric calculations intuitive and clean. It can handle both integer and floating-point coordinates. Operations supported:

- Addition/Subtraction: `p1 + p2`, `p1 - p2`
- Scalar Multiplication/Division: `p * scalar`, `p / scalar`
- Dot Product: `p1.dot(p2)`
- Cross Product: `p1.cross(p2)` (returns the 2D magnitude)
- Squared Euclidean Distance: `p1.dist_sq(p2)`
- Comparison: `p1 == p2`, `p1 < p2` (lexicographical)

A standalone `orientation` function is also provided to determine the orientation of three ordered points (collinear, clockwise, or counter-clockwise), which is a fundamental primitive for many geometric algorithms.

```python
import math


class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __lt__(self, other):
        if self.x != other.x:
            return self.x < other.x
        return self.y < other.y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Point(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        return Point(self.x * scalar, self.y * scalar)

    def __truediv__(self, scalar):
        return Point(self.x / scalar, self.y / scalar)

    def dot(self, other):
        return self.x * other.x + self.y * other.y

    def cross(self, other):
        return self.x * other.y - self.y * other.x

    def dist_sq(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return dx * dx + dy * dy


def orientation(p, q, r):
    """
    Determines the orientation of the ordered triplet (p,
    ↪ q, r).

    Returns:
        int: > 0 for counter-clockwise, < 0 for clockwise,
        ↪ 0 for collinear.
    """
    val = (q.x - p.x) * (r.y - q.y) - (q.y - p.y) * (r.x
    ↪ - q.x)
    if val == 0:
        return 0
```

```python
    return 1 if val > 0 else -1
```

Polygon Area

Implements functions to calculate the area and centroid of a simple (non-self-intersecting) polygon. The area is calculated using the Shoelace formula, which computes the signed area based on the cross products of adjacent vertices. The absolute value of this result gives the geometric area. The centroid calculation uses a related formula derived from the shoelace principle. Both functions assume the polygon vertices are provided in a consistent order (either clockwise or counter-clockwise).

```python
from content.geometry.point import Point


def polygon_area(vertices):
    """
    Calculates the area of a simple polygon using the
    ↪ Shoelace formula.

    Args:
        vertices (list[Point]): A list of Point objects
        ↪ representing the
                                vertices of the polygon in
                                ↪ order.

    Returns:
        float: The area of the polygon.
    """
    n = len(vertices)
    if n < 3:
        return 0.0

    area = 0.0
    for i in range(n):
        p1 = vertices[i]
        p2 = vertices[(i + 1) % n]
        area += p1.cross(p2)

    return abs(area) / 2.0


def polygon_centroid(vertices):
    """
    Calculates the centroid of a simple polygon.

    Args:
        vertices (list[Point]): A list of Point objects
        ↪ representing the
                                vertices of the polygon in
                                ↪ order.

    Returns:
        Point | None: A Point object representing the
        ↪ centroid, or None if the
                      polygon's area is zero.
    """
    n = len(vertices)
    if n < 3:
        return None

    signed_area = 0.0
    centroid_x = 0.0
    centroid_y = 0.0

    for i in range(n):
        p1 = vertices[i]
        p2 = vertices[(i + 1) % n]
        cross_product = p1.cross(p2)

        signed_area += cross_product
        centroid_x += (p1.x + p2.x) * cross_product
        centroid_y += (p1.y + p2.y) * cross_product

    if abs(signed_area) < 1e-9:
        return None

    area = signed_area / 2.0
    centroid_x /= 6.0 * area
    centroid_y /= 6.0 * area

    return Point(centroid_x, centroid_y)
```

Chapter 9

Dynamic Programming

Common Patterns

This file provides implementations for three classic dynamic programming patterns that are foundational in competitive programming: Longest Increasing Subsequence (LIS), Longest Common Subsequence (LCS), and the 0/1 Knapsack problem. Longest Increasing Subsequence (LIS): Given a sequence of numbers, the goal is to find the length of the longest subsequence that is strictly increasing. The standard DP approach takes $O(N^2)$ time. This file implements a more efficient $O(N \log N)$ solution. The algorithm maintains an auxiliary array (e.g., `tails`) where `tails[i]` stores the smallest tail of all increasing subsequences of length `i+1`. When processing a new number `x`, we find the smallest tail that is greater than or equal to `x`. If `x` is larger than all tails, it extends the LIS. Otherwise, it replaces the tail it was compared against, potentially allowing for a better solution later. This search and replacement is done using binary search. Longest Common Subsequence (LCS): Given two sequences, the goal is to find the length of the longest subsequence present in both of them. The standard DP solution uses a 2D table `dp[i][j]` which stores the length of the LCS of the prefixes `s1[0...i-1]` and `s2[0...j-1]`. The recurrence relation is:

- If `s1[i-1] == s2[j-1]`, then `dp[i][j] = 1 + dp[i-1][j-1]`.
- Otherwise, `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`.

0/1 Knapsack Problem: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. In the 0/1 version, you can either take an item or leave it. The standard solution uses a DP table `dp[i][w]` representing the maximum value using items up to `i` with a weight limit of `w`. This can be optimized in space to a 1D array where `dp[w]` is the maximum value for a capacity of `w`.

- LIS: $O(N \log N)$
- LCS: $O(N \cdot M)$ where N and M are the lengths of the sequences.
- 0/1 Knapsack: $O(N \cdot W)$ where N is number of items, W is capacity.
- LIS: $O(N)$
- LCS: $O(N \cdot M)$
- 0/1 Knapsack: $O(W)$ (space-optimized)

```python
import bisect


def longest_increasing_subsequence(arr):
    """
    Finds the length of the longest increasing subsequence
    ↪ in O(N log N).
    """
    if not arr:
        return 0

    tails = []
    for num in arr:
        idx = bisect.bisect_left(tails, num)
        if idx == len(tails):
            tails.append(num)
        else:
            tails[idx] = num
    return len(tails)


def longest_common_subsequence(s1, s2):
    """
    Finds the length of the longest common subsequence in
    ↪ O(N*M).
    """
    n, m = len(s1), len(s2)
    dp = [[0] * (m + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = 1 + dp[i - 1][j - 1]
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j -
                ↪ 1])
    return dp[n][m]


def knapsack_01(weights, values, capacity):
    """
    Solves the 0/1 Knapsack problem with space
    ↪ optimization.
    """
    n = len(weights)
    dp = [0] * (capacity + 1)

    for i in range(n):
        for w in range(capacity, weights[i] - 1, -1):
            dp[w] = max(dp[w], values[i] + dp[w -
            ↪ weights[i]])

    return dp[capacity]
```

Dp Optimizations

This file explains and demonstrates several advanced dynamic programming optimizations. The primary focus is the Convex Hull Trick, with conceptual explanations for Knuth-Yao Speedup and Divide and Conquer Optimization. Convex Hull Trick (CHT): This optimization applies to DP recurrences of the form: `dp[i] = min_{j<i} (dp[j] + b[j] * a[i])` (or similar). For a fixed `i`, each `j` defines a line `y = m*x + c`, where `m = b[j]`, `x = a[i]`, and `c = dp[j]`. The problem then becomes finding the minimum value among a set of lines for a given x-coordinate `a[i]`. A `LineContainer` data structure is used to maintain the lower envelope (convex hull) of these lines, allowing for efficient queries. The example below solves a problem with the recurrence `dp[i] = C + min_{j<i} (dp[j] + (p[i] - p[j])^2)`, which can be rearranged into the required line form. This works efficiently if the slopes of the lines being added are monotonic. Knuth-Yao Speedup: This optimization applies to recurrences of the form `dp[i][j] = C[i][j] + min_{i<=k<j} (dp[i][k] + dp[k+1][j])`, such as in the optimal binary search tree problem. It can be used if the cost function `C` satisfies the quadrangle inequality (`C[a][c] + C[b][d] <= C[a][d] + C[b][c]` for `a <= b <= c <= d`). The key insight is that the optimal splitting point `k` for `dp[i][j]`, denoted `opt[i][j]`, is monotonic: `opt[i][j-1] <= opt[i][j] <= opt[i+1][j]`. This property allows us to reduce the search space for `k` from `O(j-i)` to `opt[i+1][j] - opt[i][j-1]`, improving the total time complexity from $O(N^3)$ to $O(N^2)$. Divide and Conquer Optimization: This technique applies to recurrences of the form `dp[i][j] = min_{0<=k<j} (dp[i-1][k] + C[k][j])`. A naive computation would take $O(N^2)$ for each `i`, leading to $O(K*N^2)$ total time for `K` states. The optimization is based on the observation that if the cost function `C` has certain properties (often related to the quadrangle inequality), the optimal choice of `k` for `dp[i][j]` is monotonic with `j`. We can compute all `dp[i][j]` values for a fixed `i` and `j` in a range `[l, r]` by first finding the optimal `k` for the midpoint `mid = (l+r)/2`. Then, recursively, the optimal `k` for the left half `[l, mid-1]` must be in a smaller range, and similarly for the right half. This divide and conquer approach computes all `dp[i][j]` for a fixed `i` in $O(N \log N)$ time.

```python
import sys
import os

# The stress test runner adds the project root to the path.
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "../../")))
from content.data_structures.line_container import LineContainer


def convex_hull_trick_example(p, C):
    """
    Solves an example problem using the Convex Hull Trick.
    Problem: Given n points on a line with increasing coordinates p[0]...p[n-1],
    find the minimum cost to travel from point 0 to point n-1. The cost of
    jumping from point i to point j is (p[j] - p[i])^2 + C.

    DP recurrence: dp[i] = min_{j<i} (dp[j] + (p[i] - p[j])^2 + C)
    This can be rewritten as:
    dp[i] = p[i]^2 + C + min_{j<i} (-2*p[j]*p[i] + dp[j] + p[j]^2)
    This fits the form y = mx + c, where:
    - x = p[i]
    - m_j = -2 * p[j]
    - c_j = dp[j] + p[j]^2
    Since p is increasing, the slopes m_j are decreasing, matching the
    `LineContainer`'s requirement.

    Args:
        p (list[int]): A list of increasing integer coordinates.
        C (int): A constant cost for each jump.

    Returns:
        int: The minimum cost to reach the last point.
    """
    n = len(p)
    if n <= 1:
        return 0

    dp = [0] * n
    lc = LineContainer()

    # Base case: dp[0] = 0. Add the first line to the container.
    # m_0 = -2*p[0], c_0 = dp[0] + p[0]^2 = p[0]^2
    lc.add(-2 * p[0], p[0] ** 2)

    for i in range(1, n):
        # Query for the minimum value at x = p[i]
        min_val = lc.query(p[i])
        dp[i] = p[i] ** 2 + C + min_val

        # Add the new line corresponding to state i to the container
        # m_i = -2*p[i], c_i = dp[i] + p[i]^2
        lc.add(-2 * p[i], dp[i] + p[i] ** 2)

    return dp[n - 1]
```