

Python Competitive Programming Notebook

PyCPBook Community

September 27, 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

1	Fundamentals	3
2	Standard Library	8
3	Contest & Setup	12
4	Data Structures	15
5	Graph Algorithms	25
6	String Algorithms	35
7	Mathematics & Number Theory	40
8	Geometry	44
9	Dynamic Programming	47

Chapter 1

Fundamentals

Binary Search

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS) Description: 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. Time: $O(\log N)$, where N is the number of elements in the array. Space: $O(1)$ Status: Stress-tested

```
1 def binary_search(arr, target):
2     """
3     Searches for a target value in a sorted array.
4
5     Args:
6         arr (list): A sorted list of elements.
7         target: The value to search for.
8
9     Returns:
10        int: The index of the target in the array
11        ↪ if found, otherwise -1.
12    """
13    low, high = 0, len(arr) - 1
14
15    while low <= high:
16        mid = low + (high - low) // 2
17        if arr[mid] < target:
18            low = mid + 1
19        elif arr[mid] > target:
20            high = mid - 1
21        else:
22            return mid
```

```
22     return -1
23
```

Greedy Algorithms

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS) Description: 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.

Time: $O(N \log N)$, dominated by sorting the activities by finish time. Space: $O(N)$ to store the activities. Status: Stress-tested

```
1 def activity_selection(activities):
2     """
3     Selects the maximum number of non-overlapping
4     ↪ activities.
5
6     Args:
7         activities (list[tuple[int, int]]): A list
8         ↪ of activities, where each
9         ↪ activity is a tuple (start_time,
10        ↪ finish_time).
11
12     Returns:
13        int: The maximum number of non-overlapping
14        ↪ activities.
15    """
16    if not activities:
17        return 0
18
19    # Sort activities by their finish times in
20    ↪ ascending order
21    activities.sort(key=lambda x: x[1])
```

```

17     count = 1
18     last_finish_time = activities[0][1]
19
20
21     for i in range(1, len(activities)):
22         start_time, finish_time = activities[i]
23         if start_time >= last_finish_time:
24             count += 1
25             last_finish_time = finish_time
26
27     return count

```

Prefix Sums

Author: PyCPBook Community Source: CP-Algorithms, TopCoder tutorials Description: 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: $\text{sum} = P[r2][c2] - P[r1][c2] - P[r2][c1] + P[r1][c1]$.

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

```

1 def build_prefix_sum_1d(arr):
2     """
3     Builds a 1D prefix sum array and returns a
4     ↪ query function.
5
6     Args:
7         arr (list[int]): The input 1D array.
8
9     Returns:
10        function: A function `query(left, right)`
11        ↪ that returns the sum of
12        the elements in the range [left,
13        ↪ right-1] in  $O(1)$ .
14    """
15    n = len(arr)
16    prefix_sum = [0] * (n + 1)
17    for i in range(n):
18        prefix_sum[i + 1] = prefix_sum[i] + arr[i]
19
20    def query(left, right):
21        """

```

```

22        Queries the sum of the range [left,
23        ↪ right-1].
24        `left` is inclusive, `right` is exclusive.
25    """
26    if not (0 <= left <= right <= n):
27        return 0
28    return prefix_sum[right] - prefix_sum[left]
29
30    return query
31
32 def build_prefix_sum_2d(grid):
33     """
34     Builds a 2D prefix sum array and returns a
35     ↪ query function.
36
37     Args:
38         grid (list[list[int]]): The input 2D grid.
39
40     Returns:
41         function: A function `query(r1, c1, r2,
42         ↪ c2)` that returns the sum of
43         the elements in the rectangle
44         ↪ from (r1, c1) to (r2-1, c2-1)
45         ↪ in  $O(1)$ .
46    """
47    if not grid or not grid[0]:
48        return lambda r1, c1, r2, c2: 0
49
50    rows, cols = len(grid), len(grid[0])
51    prefix_sum = [[0] * (cols + 1) for _ in
52        ↪ range(rows + 1)]
53
54    for r in range(rows):
55        for c in range(cols):
56            prefix_sum[r + 1][c + 1] = (
57                grid[r][c]
58                + prefix_sum[r][c + 1]
59                + prefix_sum[r + 1][c]
60                - prefix_sum[r][c]
61            )
62
63    def query(r1, c1, r2, c2):
64        """
65        Queries the sum of the rectangle from (r1,
66        ↪ c1) to (r2-1, c2-1).
67        `r1, c1` are inclusive top-left
68        ↪ coordinates.
69        `r2, c2` are exclusive bottom-right
70        ↪ coordinates.
71    """
72    if not (0 <= r1 <= r2 <= rows and 0 <= c1
73        ↪ <= c2 <= cols):
74        return 0
75    return (
76        prefix_sum[r2][c2]
77        - prefix_sum[r1][c2]
78        - prefix_sum[r2][c1]
79        + prefix_sum[r1][c1]
80    )
81
82    return query

```

Python Idioms

Author: PyCPBook Community Source: Collective Python programming experience Description: 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. Time: N/A Space: N/A Status: Not applicable (Informational)

```

1 def python_idioms_examples():
2     """
3     Demonstrates various Python idioms useful in
4     ↪ competitive programming.
5     This function is primarily for inclusion in the
6     ↪ notebook and is called
7     by the stress test to ensure correctness.
8     """
9     # List Comprehensions
10    squares = [x * x for x in range(5)]
11    even_squares = [x * x for x in range(10) if x %
12    ↪ 2 == 0]
13
14    # Set and Dictionary Comprehensions
15    unique_squares = {x * x for x in [-1, 1, -2,
16    ↪ 2]}
17    square_map = {x: x * x for x in range(5)}
18
19    # Advanced Sorting
20    pairs = [(1, 5), (3, 2), (2, 8)]
21    sorted_by_second = sorted(pairs, key=lambda p:
22    ↪ p[1])
23
24    # 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

Author: PyCPBook Community Source: Standard computer science curriculum (e.g., CLRS) Description: 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.

Time: $O(N \cdot 2^N)$. There are 2^N possible subsets.

For each subset, it takes up to $O(N)$ time to create a copy to add to the results list. Space: $O(N)$ for the recursion depth and the temporary list storing the current subset. The output list itself requires $O(N \cdot 2^N)$ space. Status: To be stress-tested

```

1 def generate_subsets(nums):
2     """
3     Generates all possible subsets (the power set)
4     ↪ of a list of numbers.
5
6     Args:
7         nums (list[int]): A list of numbers.
8
9     Returns:
10        list[list[int]]: A list containing all
11        ↪ subsets of nums.
12    """
13
14    result = []
15    current_subset = []
16
17    def backtrack(start_index):
18        # Add the current subset configuration to
19        ↪ the result list.
20        # A copy is made because current_subset
21        ↪ will be modified.
22        result.append(list(current_subset))
23
24        # Explore further choices.
25        for i in range(start_index, len(nums)):
26            # 1. Choose: Include the number nums[i]
27            ↪ in the current subset.
28            current_subset.append(nums[i])
29
30            # 2. Explore: Recursively call with the
31            ↪ next index.
32            backtrack(i + 1)
33
34            # 3. Unchoose: Remove nums[i] to
35            ↪ backtrack and explore other paths.
36            current_subset.pop()
37
38    backtrack(0)
39    return result

```

Stacks And Queues

Author: PyCPBook Community Source: Python official documentation, standard CS texts Description: 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. Time: All stack and deque operations shown are $O(1)$. Space: N/A Status: Not applicable (Informational)

```

1 from collections import deque
2
3
4 def stack_and_queue_examples():
5     """
6     Demonstrates the usage of stacks (with lists)
7     ↪ and queues (with deque).
8     This function is primarily for inclusion in the
9     ↪ notebook and is called
10    by the stress test to ensure correctness.
11    """
12
13    # --- Stack Example (LIFO) ---
14    stack = []
15    stack.append(10) # Stack: [10]
16    stack.append(20) # Stack: [10, 20]
17    stack.append(30) # Stack: [10, 20, 30]
18
19    popped_from_stack = []
20    popped_from_stack.append(stack.pop()) #
21    ↪ Returns 30, Stack: [10, 20]
22    popped_from_stack.append(stack.pop()) #
23    ↪ Returns 20, Stack: [10]
24
25    # --- Queue Example (FIFO) ---
26    queue = deque()
27    queue.append(10) # Queue: deque([10])
28    queue.append(20) # Queue: deque([10, 20])
29    queue.append(30) # Queue: deque([10, 20, 30])
30
31    popped_from_queue = []
32    popped_from_queue.append(queue.popleft()) #
33    ↪ Returns 10, Queue: deque([20, 30])
34    popped_from_queue.append(queue.popleft()) #
35    ↪ Returns 20, Queue: deque([30])
36
37    return {
38        "final_stack": stack,
39        "popped_from_stack": popped_from_stack,
40        "final_queue": list(queue), # Convert to
41        ↪ list for easy comparison
42        "popped_from_queue": popped_from_queue,
43    }

```

Two Pointers

Author: PyCPBook Community Source: LeetCode, TopCoder Description: 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.

Time: $O(N)$, where N is the length of the input string/array, because each pointer traverses the data structure at most once. **Space:** $O(K)$ or $O(\Sigma)$, where K is the number of distinct elements allowed or Σ is the size of the character set, to store the elements in the window. **Status:** To be stress-tested.

```

1 from collections import defaultdict
2
3
4 def longest_substring_with_k_distinct(s, k):
5     """
6     Finds the length of the longest substring of s
7     ↪ that contains at most k
8     distinct characters.
9
10    Args:
11        s (str): The input string.
12        k (int): The maximum number of distinct
13        ↪ characters allowed.
14
15    Returns:
16        int: The length of the longest valid
17        ↪ substring.
18    """
19    if k == 0:

```

```

17         return 0
18
19    n = len(s)
20    left = 0
21    max_len = 0
22    char_counts = defaultdict(int)
23
24    for right in range(n):
25        char_counts[s[right]] += 1
26
27        while len(char_counts) > k:
28            char_left = s[left]
29            char_counts[char_left] -= 1
30            if char_counts[char_left] == 0:
31                del char_counts[char_left]
32            left += 1
33
34        max_len = max(max_len, right - left + 1)
35
36    return max_len
37

```


Chapter 2

Standard Library

Bisect Library

Author: PyCPBook Community Source: Python official documentation Description: 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."

Time: `bisect_left` and `bisect_right` are $O(\log N)$. `insort_left` is $O(N)$ due to the list insertion. Space: $O(1)$ for searches. Status: To be stress-tested

```
1 import bisect
2
3
4 def bisect_examples():
5     """
6     Demonstrates the usage of the bisect module.
7     This function is primarily for inclusion in the
8     ↪ notebook and is called
9     by the stress test to ensure correctness.
10    """
11    data = [10, 20, 20, 30, 40]
12
13    # --- bisect_left ---
14    # Find insertion point for 20 (before existing
15    ↪ 20s)
16    idx_left_20 = bisect.bisect_left(data, 20)
17    # Find insertion point for 25 (between 20 and
18    ↪ 30)
19    idx_left_25 = bisect.bisect_left(data, 25)
```

```
18 # --- bisect_right ---
19 # Find insertion point for 20 (after existing
20 ↪ 20s)
21 idx_right_20 = bisect.bisect_right(data, 20)
22 # Find insertion point for 25 (same as
23 ↪ bisect_left)
24 idx_right_25 = bisect.bisect_right(data, 25)
25
26 # --- insort ---
27 # insort_left inserts at the position found by
28 ↪ bisect_left
29 data_for_insort = [10, 20, 20, 30, 40]
30 bisect.insort_left(data_for_insort, 25)
31
32 return {
33     "idx_left_20": idx_left_20,
34     "idx_left_25": idx_left_25,
35     "idx_right_20": idx_right_20,
36     "idx_right_25": idx_right_25,
37     "list_after_insort": data_for_insort,
38 }
```

Collections Library

Author: PyCPBook Community Source: Python official documentation Description: 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)).

Time: All key operations for these classes (append, popleft for deque; element access and update for Counter and defaultdict) are amortized $O(1)$. Space: $O(N)$ where N is the number of elements stored. Status: Stress-tested

```

1 from collections import deque, Counter, defaultdict
2
3
4 def collections_examples():
5     """
6     Demonstrates the usage of deque, Counter, and
7     ↪ defaultdict.
8     This function is primarily for inclusion in the
9     ↪ notebook and is called
10    by the stress test to ensure correctness.
11    """
12    # --- deque ---
13    q = deque([1, 2, 3])
14    q.append(4)
15    q.appendleft(0)
16    q_pop_left = q.popleft()
17    q_pop_right = q.pop()
18
19    # --- Counter ---
20    data = ["a", "b", "c", "a", "b", "a"]
21    counts = Counter(data)
22    count_of_a = counts["a"]
23    count_of_d = counts["d"]
24
25    # --- defaultdict ---
26    adj = defaultdict(list)
27    edges = [(0, 1), (0, 2), (1, 2)]
28    for u, v in edges:
29        adj[u].append(v)
30        adj[v].append(u)
31
32    # Access a missing key to trigger the default
33    ↪ factory
34    missing_key_val = adj[5]
35
36    return {
37        "final_deque": list(q),
38        "q_pop_left": q_pop_left,
39        "q_pop_right": q_pop_right,
40        "counter_a": count_of_a,
41        "counter_d": count_of_d,
42        "adj_list": dict(adj),
43        "adj_list_missing": missing_key_val,
44    }

```

Functools Library

Author: PyCPBook Community Source: Python official documentation Description: 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.

Time: The decorated function's complexity becomes proportional to the number of unique states it's called with, rather than the total number of calls. Space: $O(S)$ where S is the number of unique states (sets of arguments) stored in the cache. Status: Stress-tested

```

1 import functools
2
3
4 @functools.cache
5 def fibonacci_with_cache(n):
6     """
7     Computes the n-th Fibonacci number using
8     ↪ recursion with memoization.
9     This function is primarily for demonstrating
10    ↪ @functools.cache.
11    """
12    if n < 2:
13        return n
14    return fibonacci_with_cache(n - 1) +
15        ↪ fibonacci_with_cache(n - 2)

```

Heapq Library

Author: PyCPBook Community Source: Python official documentation Description: 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 re-

turns 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).

Time: `heappush` and `heappop` are $O(\log N)$. `heapify` is $O(N)$. Space: $O(N)$ for storing N elements in the list. Status: To be stress-tested

```

1 import heapq
2
3
4 def heapq_examples():
5     """
6     Demonstrates the usage of the heapq module.
7     This function is primarily for inclusion in the
8     ↪ notebook and is called
9     by the stress test to ensure correctness.
10    """
11    # --- heappush and heappop ---
12    min_heap = []
13    heapq.heappush(min_heap, 4)
14    heapq.heappush(min_heap, 1)
15    heapq.heappush(min_heap, 7)
16
17    # After pushes, the heap (list) is [1, 4, 7]
18    # The smallest element is at index 0
19    smallest_element = min_heap[0]
20
21    popped_elements = []
22    popped_elements.append(heapq.heappop(min_heap))
23    ↪ # Pops 1
24    popped_elements.append(heapq.heappop(min_heap))
25    ↪ # Pops 4
26
27    # --- heapify ---
28    data_list = [5, 8, 2, 9, 1, 4]
29    heapq.heapify(data_list)
30    # After heapify, data_list is now [1, 4, 2, 9,
31    ↪ 8, 5] (or similar,
32    # it only guarantees the heap property, not a
33    ↪ fully sorted list)
34    heapified_list = list(data_list)
35    smallest_after_heapify = data_list[0]
36
37    return {
38        "smallest_element": smallest_element,
39        "final_heap": min_heap,
40        "popped_elements": popped_elements,
41        "heapified_list": heapified_list,
42        "smallest_after_heapify":
43            ↪ smallest_after_heapify,
44    }

```

Itertools Library

Author: PyCPBook Community Source: Python official documentation Description: 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. Time: The number of items returned is the primary factor. For an iterable of length N , `permutations` returns $P(N, r)$ items, `combinations` returns $C(N, r)$ items, and `product` returns N^k items for k iterables. Space: $O(r)$ or $O(N)$ for storing the intermediate tuple. Status: Stress-tested

```

1 import itertools
2
3
4 def itertools_examples():
5     """
6     Demonstrates the usage of common itertools
7     ↪ functions.
8     This function is primarily for inclusion in the
9     ↪ notebook and is called
10    by the stress test to ensure correctness.
11    """
12    elements = ["A", "B", "C"]
13
14    # --- Permutations ---
15    # All full-length permutations of elements
16    perms_full =
17    ↪ list(itertools.permutations(elements))
18    # All 2-element permutations of elements
19    perms_partial =
20    ↪ list(itertools.permutations(elements, 2))
21
22    # --- Combinations ---
23    # All 2-element combinations of elements
24    combs = list(itertools.combinations(elements,
25    ↪ 2))
26
27    # --- Cartesian Product ---
28    pool1 = ["x", "y"]
29    pool2 = [1, 2]

```

```

25     prod = list(itertools.product(pool1, pool2))
26
27     return {
28         "perms_full": perms_full,
29         "perms_partial": perms_partial,
30         "combs": combs,
31         "prod": prod,
32     }
33

```

Math Library

Author: PyCPBook Community Source: Python official documentation Description: 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. Time: `gcd` is $O(\log(\min(a, b)))$. `isqrt` is faster than `sqrt` for integers. The rest are typically $O(1)$. Space: $O(1)$ for all functions. Status: Stress-tested

```

23     # Infinity constant
24     infinity = math.inf
25
26
27     return {
28         "gcd_val": gcd_val,
29         "ceil_val": ceil_val,
30         "floor_val": floor_val,
31         "sqrt_val": sqrt_val,
32         "isqrt_val": isqrt_val,
33         "log2_val": log2_val,
34         "infinity": infinity,
35     }
36

```

```

1  import math
2
3
4  def math_examples():
5      """
6      Demonstrates the usage of common math module
7      ↪ functions.
8      This function is primarily for inclusion in the
9      ↪ notebook and is called
10     by the stress test to ensure correctness.
11     """
12
13     # Greatest Common Divisor
14     gcd_val = math.gcd(54, 24)
15
16
17     # Ceiling and Floor
18     ceil_val = math.ceil(4.2)
19     floor_val = math.floor(4.8)
20
21
22     # Square Roots
23     sqrt_val = math.sqrt(25)
24     isqrt_val = math.isqrt(26)
25
26
27     # Logarithm
28     log2_val = math.log2(16)
29

```

Chapter 3

Contest & Setup

Debugging Tricks

Author: PyCPBook Community Source: Collective experience from competitive programmers. Description: 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)`

2. 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 $N=2$.

3. 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.

4. 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.

5. 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)

```
2
3
4 def example_debug_print():
5     """
6     A simple example demonstrating how to print
7     ↪ debug information
8     to stderr without affecting the program's
9     ↪ actual output.
10    """
11    data = [10, 20, 30]
12
13    # This is the actual output that the judge will
14    ↪ see.
15    print("Processing started.")
16
17    total = 0
18    for i, item in enumerate(data):
19        # This is a debug message. It goes to
20        ↪ stderr and is ignored by the judge.
21        print(f"DEBUG: Processing item {i} with
22        ↪ value {item}", file=sys.stderr)
23        total += item
24
25    # This is the final output.
26    print(f"The final total is: {total}")
```

Fast Io

Author: PyCPBook Community Source: Python official documentation, TopCoder tutorials Description: 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.

2. 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()`

3. 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.

4. 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.

5. 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())`

6. 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)

```

1 import sys
2
3
4 def get_ints_from_line():
5     """
6     Reads a line of space-separated values and
7     ↪ parses them into a list of integers.
8     This is a common helper function.
9     """
10    return list(map(int,
11    ↪ sys.stdin.readline().split()))
12
13
14 def process_all_lines():
15     """
16     Demonstrates reading all lines at once and
17     ↪ processing them.
18     This example calculates the sum of the first
19     ↪ integer on each line.
20     """
21    lines = sys.stdin.readlines()
22    total = 0
23    for line in lines:
24        if line.strip():
25            total += int(line.split()[0])
26    return total
27
28
29 def iterate_over_stdin():
30     """
31     Demonstrates processing input by iterating over
32     ↪ sys.stdin, which reads
33     line by line until EOF.
34     """
35    total = 0
36    for line in sys.stdin:
37        if line.strip():
38            total += sum(map(int, line.split()))
39    return total

```

```

37 def demonstrate_usage_conceptual():
38     """
39     This function is for inclusion in the notebook
40     ↪ as a clear example and
41     is not meant to be executed directly as part of
42     ↪ a test. It shows a common
43     contest pattern: reading a count `N`, followed
44     ↪ by `N` lines of data.
45     """
46    try:
47        # Read N, the number of lines to follow
48        n_str = sys.stdin.readline()
49        if not n_str:
50            return
51        n = int(n_str)
52
53        # Read N lines into a matrix
54        matrix = []
55        for _ in range(n):
56            row = list(map(int,
57            ↪ sys.stdin.readline().split()))
58            matrix.append(row)
59
60        # In a real problem, you would process the
61        ↪ matrix here.
62        # For demonstration, we just show it was
63        ↪ read.
64        # print("Matrix received:", matrix)
65
66    except (IOError, ValueError):
67        # Handle potential empty input or parsing
68        ↪ errors gracefully.
69        pass

```

Template

Author: PyCPBook Community Source: Various competitive programming resources Description: 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)

```

1 import sys
2 import math
3 import os
4

```

```
5 sys.setrecursionlimit(10**6)
6
7 input = sys.stdin.readline
8
9
10 def get_int():
11     """Reads a single integer from a line."""
12     return int(input())
13
14
15 def get_ints():
16     """Reads a list of space-separated integers
17     ↪ from a line."""
18     return list(map(int, input().split()))
19
20 def get_str():
21     """Reads a single string from a line, stripping
22     ↪ trailing whitespace."""
23     return input().strip()
24
25 def get_strs():
26     """Reads a list of space-separated strings from
27     ↪ a line."""
28     return input().strip().split()
29
30 def solve():
31     """
32     This is the main function where the solution
33     ↪ logic for a single
34     test case should be implemented.
35     """
36     try:
37         n, m = get_ints()
38         print(n + m)
39     except (IOError, ValueError):
40         pass
41
42 def main():
43     """
44     Main execution function.
45     Handles multiple test cases if required.
46     """
47     # t = get_int()
48     # for _ in range(t):
49     #     solve()
50     solve()
51
52
53 if __name__ == "__main__":
54     main()
55
```


Chapter 4

Data Structures

Binary Search Tree

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS) Description: 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).

Time: Average case for `search`, `insert`, `delete` is $O(\log N)$. Worst case is $O(N)$. Space: $O(N)$ to store the nodes of the tree. Status: Stress-tested

```
1 class Node:
2     """Represents a single node in the Binary
3     ↳ Search Tree."""
4
5     def __init__(self, key):
6         self.key = key
7         self.left = None
8         self.right = None
9
10 class BinarySearchTree:
11     """A standard (unbalanced) Binary Search Tree
12     ↳ implementation."""
13
14     def __init__(self):
15         self.root = None
16
17     def search(self, key):
18         """Searches for a key in the BST."""
19         return self._search_recursive(self.root,
20         ↳ key) is not None
21
22     def _search_recursive(self, node, key):
23         if node is None or node.key == key:
24             return node
25         if key < node.key:
```

```
24         return
25         ↳ self._search_recursive(node.left,
26         ↳ key)
27     def insert(self, key):
28         """Inserts a key into the BST."""
29         if self.root is None:
30             self.root = Node(key)
31         else:
32             self._insert_recursive(self.root, key)
33
34     def _insert_recursive(self, node, key):
35         if key < node.key:
36             if node.left is None:
37                 node.left = Node(key)
38             else:
39                 self._insert_recursive(node.left,
40                 ↳ key)
41         elif key > node.key:
42             if node.right is None:
43                 node.right = Node(key)
44             else:
45                 self._insert_recursive(node.right,
46                 ↳ key)
47
48     def delete(self, key):
49         """Deletes a key from the BST."""
50         self.root =
51         ↳ self._delete_recursive(self.root, key)
52
53     def _delete_recursive(self, node, key):
54         if node is None:
55             return node
56
57         if key < node.key:
58             node.left =
59             ↳ self._delete_recursive(node.left,
60             ↳ key)
61         elif key > node.key:
62             node.right =
63             ↳ self._delete_recursive(node.right,
64             ↳ key)
65         else:
66             if node.left is None:
67                 return node.right
68             elif node.right is None:
69                 return node.left
70
71             # Node with two children: Get the
72             ↳ in-order successor (smallest in the
73             ↳ right subtree)
74             temp = self._min_value_node(node.right)
75             node.key = temp.key
76             node.right =
77             ↳ self._delete_recursive(node.right,
78             ↳ temp.key)
79             return node
```



```

69 def _min_value_node(self, node):
70     current = node
71     while current.left is not None:
72         current = current.left
73     return current
74
75

```

Fenwick Tree

Author: PyCPBook Community Source: Based on common implementations in competitive programming resources Description: 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 $[0, \text{right}]$.

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`.

Time: $O(\log N)$ for both `add` (point update) and `query` (prefix sum). Space: $O(N)$ to store the tree array. Status: Stress-tested

```

1 class FenwickTree:
2     """
3     A class that implements a 1D Fenwick Tree
4     ↪ (Binary Indexed Tree).
5     This implementation uses 0-based indexing for
6     ↪ its public methods.
7     """
8
9     def __init__(self, size):
10         """
11         Initializes the Fenwick Tree for an array
12         ↪ of a given size.
13         All elements are initially zero.
14
15         Args:
16             size (int): The number of elements the
17             ↪ tree will support.
18         """
19         self.tree = [0] * size
20
21     def add(self, idx, delta):
22         """

```

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`.

"""

```

26 while idx < len(self.tree):
27     self.tree[idx] += delta
28     idx |= idx + 1

```

```

30 def query(self, right):

```

"""
Computes the prefix sum of elements up to
↪ (but not including) `right`.
This is the sum of the range $[0, \text{right}-1]$.

Args:

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

Returns:

`int`: The sum of elements in the prefix
↪ $[0, \text{right}-1]$.

"""

```

41 idx = right - 1
42 total_sum = 0
43 while idx >= 0:
44     total_sum += self.tree[idx]
45     idx = (idx & (idx + 1)) - 1
46 return total_sum

```

```

48 def query_range(self, left, right):

```

"""
Computes the sum of elements in the range
↪ $[\text{left}, \text{right}-1]$.

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.

"""

```

59 if left >= right:
60     return 0
61 return self.query(right) - self.query(left)
62

```

Fenwick Tree 2D

Author: PyCPBook Community Source: KACTL, TopCoder tutorials Description: 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 sub-rectangle. 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)`.

Time: $O(\log R \cdot \log C)$ for `add` and `query` on an $R \times C$ grid. Space: $O(R \cdot C)$ to store the 2D tree. Status: Stress-tested

```

1 class FenwickTree2D:
2     """
3     A class that implements a 2D Fenwick Tree using
4     ↪ 0-based indexing.
5     """
6     def __init__(self, rows, cols):
7         """
8         Initializes the 2D Fenwick Tree for a grid
9         ↪ of a given size.
10        All elements are initially zero.
11
12        Args:
13            rows (int): The number of rows in the
14            ↪ grid.
15            cols (int): The number of columns in
16            ↪ the grid.
17        """
18        self.rows = rows
19        self.cols = cols
20        self.tree = [[0] * cols for _ in
21            ↪ range(rows)]
22
23        def add(self, r, c, delta):
24            """
25            Adds a delta value to the element at grid
26            ↪ cell (r, c).
27
28            Args:
29                r (int): The 0-based row index of the
30                ↪ element to update.
31                c (int): The 0-based column index of
32                ↪ the element to update.
33                delta (int): The value to add.
34            """
35            i = r
36            while i < self.rows:
37                j = c
38                while j < self.cols:
39                    self.tree[i][j] += delta
40                    j |= j + 1
41                i |= i + 1

```

```

36 def query(self, r, c):
37     """
38     Computes the prefix sum of the rectangle
39     ↪ from (0, 0) to (r-1, c-1).
40
41     Args:
42         r (int): The 0-based exclusive row
43         ↪ bound of the query rectangle.
44         c (int): The 0-based exclusive column
45         ↪ bound of the query rectangle.
46
47     Returns:
48         int: The sum of the elements in the
49         ↪ rectangle [0..r-1, 0..c-1].
50     """
51     total_sum = 0
52     i = r - 1
53     while i >= 0:
54         j = c - 1
55         while j >= 0:
56             total_sum += self.tree[i][j]
57             j = (j & (j + 1)) - 1
58         i = (i & (i + 1)) - 1
59     return total_sum
60
61 def query_range(self, r1, c1, r2, c2):
62     """
63     Computes the sum of the rectangle from (r1,
64     ↪ c1) to (r2-1, c2-1).
65
66     Args:
67         r1, c1 (int): The 0-based inclusive
68         ↪ top-left coordinates.
69         r2, c2 (int): The 0-based exclusive
70         ↪ bottom-right coordinates.
71
72     Returns:
73         int: The sum of elements in the
74         ↪ specified rectangular range.
75     """
76     if r1 >= r2 or c1 >= c2:
77         return 0
78
79     total = self.query(r2, c2)
80     total -= self.query(r1, c2)
81     total -= self.query(r2, c1)
82     total += self.query(r1, c1)
83     return total

```

Hash Map Custom

Author: PyCPBook Community Source: KACTL, neal wu's blog Description: 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. Time: The hash computation is $O(1)$. Dictionary operations remain amortized $O(1)$. Space: Adds a small constant overhead per key for the wrapper object. Status: Not applicable (Utility/Informational)

```
1 import time
2
3 # A fixed random seed ensures the same hash
4 # ↪ function for each run,
5 # but it's generated based on time to be
6 # ↪ unpredictable.
7 SPLITMIX64_SEED = int(time.time()) ^
8 # ↪ 0x9E3779B97F4A7C15
9
10 def splitmix64(x):
11     """A fast, high-quality hash function for
12     ↪ 64-bit integers."""
13     x += 0x9E3779B97F4A7C15
14     x = (x ^ (x >> 30)) * 0xBF58476D1CE4E5B9
15     x = (x ^ (x >> 27)) * 0x94D049BB133111EB
16     return x ^ (x >> 31)
17
18 class CustomHash:
19     """
20     A wrapper class for hashable objects to use a
21     ↪ custom hash function.
22     This helps prevent collisions from anti-hash
23     ↪ test cases.
24     """
25
26     def __init__(self, obj):
27         self.obj = obj
28
29     def __hash__(self):
30         # Combine the object's hash with a fixed
31         # ↪ random seed using a robust function.
32         return splitmix64(hash(self.obj) +
33         # ↪ SPLITMIX64_SEED)
34
35     def __eq__(self, other):
36         # The wrapped objects must still be
37         # ↪ comparable.
38         return self.obj == other.obj
39
40     def __repr__(self):
41         return f"CustomHash({self.obj})"
```

```
35
36
37 # Example of how to use it
38 def custom_hash_example():
39     # Standard dictionary, potentially vulnerable
40     standard_dict = {}
41     # Dictionary with custom hash, much more robust
42     custom_dict = {}
43
44     key = (12345, 67890) # A tuple key, common in
45     # ↪ geometry or hashing problems
46
47     # Using the standard hash
48     standard_dict[key] = "some value"
49
50     # Using the custom hash
51     custom_key = CustomHash(key)
52     custom_dict[custom_key] = "some value"
53
54     print(f"Standard hash for {key}: {hash(key)}")
55     print(f"Custom hash for {key}:
56     # ↪ {hash(custom_key)}")
57
58     # Verifying that it works
59     assert custom_key in custom_dict
60     assert CustomHash(key) in custom_dict
61     assert CustomHash((0, 0)) not in custom_dict
```

Line Container

Author: PyCPBook Community Source: KACTL, CP-Algorithms Description: 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.

Time: $O(\log N)$ for query due to binary search.

Amortized $O(1)$ for add because each line is added and removed at most once. Space: $O(N)$ to store the lines on the convex hull. Status: Stress-tested

```

1 class LineContainer:
2     """
3     A data structure for the Convex Hull Trick,
4     ↪ optimized for minimum queries
5     and monotonically decreasing slopes.
6     """
7     def __init__(self):
8         # Each line is stored as a tuple (m, c)
9         ↪ representing  $y = mx + c$ .
10        self.hull = []
11
12    def _is_redundant(self, l1, l2, l3):
13        """
14        Checks if line l2 is redundant given its
15        ↪ neighbors l1 and l3.
16        l2 is redundant if the intersection of l1
17        ↪ and l3 is to the left of
18        the intersection of l1 and l2.
19        Intersection of (m1, c1) and (m2, c2) is x
20        ↪ = (c2 - c1) / (m1 - m2).
21        We check if (c3-c1)/(m1-m3) <=
22        ↪ (c2-c1)/(m1-m2).
23        To avoid floating point, we use
24        ↪ cross-multiplication.
25        Since slopes are decreasing,  $m1 > m2 > m3$ ,
26        ↪ so (m1-m3) and (m1-m2) are positive.
27        The inequality becomes (c3-c1)*(m1-m2) <=
28        ↪ (c2-c1)*(m1-m3).
29        """
30        m1, c1 = l1
31        m2, c2 = l2
32        m3, c3 = l3
33        # Note the direction of inequality might
34        ↪ change based on max/min query
35        # and increasing/decreasing slopes. This is
36        ↪ for min query, decr. slopes.
37        return (c3 - c1) * (m1 - m2) <= (c2 - c1) *
38        ↪ (m1 - m3)
39
40    def add(self, m, c):
41        """
42        Adds a new line  $y = mx + c$  to the
43        ↪ container.
44        Assumes that m is less than the slope of
45        ↪ any previously added line.
46        """
47        new_line = (m, c)
48        while len(self.hull) >= 2 and
49        ↪ self._is_redundant(
50        ↪ self.hull[-2], self.hull[-1], new_line
51        ):
52            self.hull.pop()
53        self.hull.append(new_line)
54
55    def query(self, x):
56        """
57        Finds the minimum value of  $y = mx + c$  for a
58        ↪ given x among all lines.
59        """
60        if not self.hull:
61            return float("inf")

```

```

48    # Binary search for the optimal line.
49    # The function `f(i) = m_i * x + c_i` is
50    ↪ not monotonic, but the
51    # optimal line index is. Specifically, the
52    ↪ function `f(i+1) - f(i)`
53    # is monotonic. We are looking for the
54    ↪ point where the function
55    # transitions from decreasing to
56    ↪ increasing.
57    low, high = 0, len(self.hull) - 1
58    res_idx = 0
59
60    while low <= high:
61        mid = (low + high) // 2
62        # Check if mid is better than mid+1
63        if mid + 1 < len(self.hull):
64            val_mid = self.hull[mid][0] * x +
65            ↪ self.hull[mid][1]
66            val_next = self.hull[mid + 1][0] *
67            ↪ x + self.hull[mid + 1][1]
68            if val_mid > val_next:
69                low = mid + 1
70            else:
71                res_idx = mid
72                high = mid - 1
73        else:
74            res_idx = mid
75            high = mid - 1
76
77    m, c = self.hull[res_idx]
78    return m * x + c

```

Ordered Set

Author: PyCPBook Community Source: KACTL, CP-Algorithms (adapted from Treap) Description: 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.

Time: $O(\log N)$ on average for `insert`, `delete`, `search`, `find_by_order`, and `order_of_key` operations, where N is the number of elements in the set.

Space: $O(N)$ to store the nodes of the set. Status:
Stress-tested

```

1 import random
2
3
4 class Node:
5     """Represents a single node in the Ordered
6     ↪ Set's underlying Treap."""
7
8     def __init__(self, key):
9         self.key = key
10        self.priority = random.random()
11        self.size = 1
12        self.left = None
13        self.right = None
14
15 def _get_size(t):
16     return t.size if t else 0
17
18 def _update_size(t):
19     if t:
20         t.size = 1 + _get_size(t.left) +
21             ↪ _get_size(t.right)
22
23 def _split(t, key):
24     """
25     Splits the tree `t` into two trees: one with
26     ↪ keys < `key` (l)
27     and one with keys >= `key` (r).
28     """
29     if not t:
30         return None, None
31     if t.key < key:
32         l, r = _split(t.right, key)
33         t.right = l
34         _update_size(t)
35         return t, r
36     else:
37         l, r = _split(t.left, key)
38         t.left = r
39         _update_size(t)
40         return l, t
41
42 def _merge(t1, t2):
43     """Merges two trees `t1` and `t2`, assuming
44     ↪ keys in `t1` < keys in `t2`."""
45     if not t1:
46         return t2
47     if not t2:
48         return t1
49     if t1.priority > t2.priority:
50         t1.right = _merge(t1.right, t2)
51         _update_size(t1)
52         return t1
53     else:
54         t2.left = _merge(t1, t2.left)
55         _update_size(t2)
56         return t2
57
58
59 class OrderedSet:
60     """

```

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

```

63
64
65 def __init__(self):
66     self.root = None
67
68 def search(self, key):
69     node = self.root
70     while node:
71         if node.key == key:
72             return True
73         node = node.left if key < node.key else
74             ↪ node.right
75     return False
76
77 def insert(self, key):
78     if self.search(key):
79         return
80     new_node = Node(key)
81     l, r = _split(self.root, key)
82     self.root = _merge(_merge(l, new_node), r)
83
84 def delete(self, key):
85     if not self.search(key):
86         return
87     l, r = _split(self.root, key)
88     _, r_prime = _split(r, key + 1)
89     self.root = _merge(l, r_prime)
90
91 def find_by_order(self, k):
92     """Finds the k-th smallest element
93     ↪ (0-indexed)."""
94     node = self.root
95     while node:
96         left_size = _get_size(node.left)
97         if left_size == k:
98             return node.key
99         elif k < left_size:
100             node = node.left
101         else:
102             k -= left_size + 1
103             node = node.right
104     return None
105
106 def order_of_key(self, key):
107     """Finds the number of elements strictly
108     ↪ smaller than key."""
109     count = 0
110     node = self.root
111     while node:
112         if key == node.key:
113             count += _get_size(node.left)
114             break
115         elif key < node.key:
116             node = node.left
117         else:
118             count += _get_size(node.left) + 1
119             node = node.right
120     return count
121
122 def __len__(self):
123     return _get_size(self.root)

```


Segment Tree Lazy

Author: PyCPBook Community Source: CP-Algorithms, various competitive programming tutorials Description: 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.

Time: $O(\log N)$ for both update (range update) and query (range query). The initial build operation takes $O(N)$ time. Space: $O(N)$ to store the tree and lazy arrays. A size of $4N$ is allocated to be safe for a complete binary tree representation. Status: Stress-tested

```

1 class SegmentTree:
2     def __init__(self, arr):
3         self.n = len(arr)
4         self.tree = [0] * (4 * self.n)
5         self.lazy = [0] * (4 * self.n)
6         self.arr = arr
7         self._build(1, 0, self.n - 1)
8
9     def _build(self, v, tl, tr):
10        if tl == tr:
11            self.tree[v] = self.arr[tl]
12        else:
13            tm = (tl + tr) // 2
14            self._build(2 * v, tl, tm)
15            self._build(2 * v + 1, tm + 1, tr)
16            self.tree[v] = self.tree[2 * v] +
17                ↪ self.tree[2 * v + 1]
18
19    def _push(self, v, tl, tr):
20        if self.lazy[v] == 0:
21            return
22
23        range_len = tr - tl + 1
24        self.tree[v] += self.lazy[v] * range_len
25
26        if tl != tr:
27            self.lazy[2 * v] += self.lazy[v]
28            self.lazy[2 * v + 1] += self.lazy[v]
29
30        self.lazy[v] = 0

```

```

30
31 def _update(self, v, tl, tr, l, r, addval):
32     self._push(v, tl, tr)
33     if l > r:
34         return
35     if l == tl and r == tr:
36         self.lazy[v] += addval
37         self._push(v, tl, tr)
38     else:
39         tm = (tl + tr) // 2
40         self._update(2 * v, tl, tm, l, min(r,
41             ↪ tm), addval)
42         self._update(2 * v + 1, tm + 1, tr,
43             ↪ max(l, tm + 1), r, addval)
44
45         # After children are updated, update
46         ↪ self based on pushed children
47         self._push(2 * v, tl, tm)
48         self._push(2 * v + 1, tm + 1, tr)
49         self.tree[v] = self.tree[2 * v] +
50             ↪ self.tree[2 * v + 1]
51
52 def _query(self, v, tl, tr, l, r):
53     if l > r:
54         return 0
55     self._push(v, tl, tr)
56     if l == tl and r == tr:
57         return self.tree[v]
58
59     tm = (tl + tr) // 2
60     left_sum = self._query(2 * v, tl, tm, l,
61         ↪ min(r, tm))
62     right_sum = self._query(2 * v + 1, tm + 1,
63         ↪ tr, max(l, tm + 1), r)
64     return left_sum + right_sum
65
66 def update(self, l, r, addval):
67     # Updates range [l, r] (inclusive)
68     if l > r:
69         return
70     self._update(1, 0, self.n - 1, l, r,
71         ↪ addval)
72
73 def query(self, l, r):
74     # Queries range [l, r] (inclusive)
75     if l > r:
76         return 0
77     return self._query(1, 0, self.n - 1, l, r)

```

Sparse Table

Author: PyCPBook Community Source: CP-Algorithms, USACO Guide Description: 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 .

Time: Precomputation is $O(N \log N)$. Each query is $O(1)$. Space: $O(N \log N)$ to store the sparse table. Status: Stress-tested

```

1 import math
2
3
4 class SparseTable:
5     """
6     A class that implements a Sparse Table for
7     ↪ efficient Range Minimum Queries.
8     This implementation assumes 0-based indexing
9     ↪ for the input array and queries.
10    """
11
12    def __init__(self, arr):
13        """
14        Initializes the Sparse Table from an input
15        ↪ array.
16
17        Args:
18            arr (list[int]): The static list of
19            ↪ numbers to be queried.
20        """
21        self.n = len(arr)
22        if self.n == 0:
23            return
24
25        self.max_log = self.n.bit_length() - 1
26        self.st = [[0] * self.n for _ in
27            ↪ range(self.max_log + 1)]
28        self.st[0] = list(arr)
29
30        for k in range(1, self.max_log + 1):
31            for i in range(self.n - (1 << k) + 1):
32                self.st[k][i] = min(
33                    self.st[k - 1][i], self.st[k -
34                        ↪ 1][i + (1 << (k - 1))])
35
36        self.log_table = [0] * (self.n + 1)
37        for i in range(2, self.n + 1):
38            self.log_table[i] = self.log_table[i >>
39                ↪ 1] + 1
40
41    def query(self, l, r):
42        """
43        Queries the minimum value in the inclusive
44        ↪ range [l, r].
45
46        Args:
47            l (int): The 0-based inclusive starting
48            ↪ index of the range.
49            r (int): The 0-based inclusive ending
50            ↪ index of the range.

```

```

43 Returns:
44     int: The minimum value in the range [l,
45         ↪ r]. Returns infinity
46         ↪ if the table is empty or the range
47         ↪ is invalid.
48
49     """
50     if self.n == 0 or l > r:
51         return float("inf")
52
53     length = r - l + 1
54     k = self.log_table[length]
55     return min(self.st[k][l], self.st[k][r - (1
56         ↪ << k) + 1])

```

Treap

Author: PyCPBook Community Source: KACTL, CP-Algorithms Description: 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.

Time: $O(\log N)$ on average for `insert`, `delete`, and `search` operations, where N is the number of nodes in the Treap. The performance depends on the randomness of the priorities. Space: $O(N)$ to store the nodes of the Treap. Status: Stress-tested

```

1 import random
2
3
4 class Node:
5     """
6     Represents a single node in the Treap.
7     Each node contains a key, a randomly generated
8     ↪ priority, and left/right children.
9     """
10
11    def __init__(self, key):
12        self.key = key
13        self.priority = random.random()
14        self.left = None
15        self.right = None

```



```

16
17 def _split(t, key):
18     """
19     Splits the tree rooted at `t` into two trees
20     ↳ based on `key`.
21     Returns a tuple (left_tree, right_tree), where
22     ↳ left_tree contains all keys
23     from `t` that are less than `key`, and
24     ↳ right_tree contains all keys that are
25     greater than or equal to `key`.
26     """
27     if not t:
28         return None, None
29     if t.key < key:
30         l, r = _split(t.right, key)
31         t.right = l
32         return t, r
33     else:
34         l, r = _split(t.left, key)
35         t.left = r
36         return l, t
37
38 def _merge(t1, t2):
39     """
40     Merges two trees `t1` and `t2`.
41     Assumes all keys in `t1` are less than all keys
42     ↳ in `t2`.
43     The merge is performed based on node priorities
44     ↳ to maintain the heap property.
45     """
46     if not t1:
47         return t2
48     if not t2:
49         return t1
50     if t1.priority > t2.priority:
51         t1.right = _merge(t1.right, t2)
52         return t1
53     else:
54         t2.left = _merge(t1, t2.left)
55         return t2
56
57 class Treap:
58     """
59     The Treap class providing a public API for
60     balanced BST operations.
61     """
62
63     def __init__(self):
64         """Initializes an empty Treap."""
65         self.root = None
66
67     def search(self, key):
68         """
69         Searches for a key in the Treap.
70         Returns True if the key is found, otherwise
71         ↳ False.
72         """
73         node = self.root
74         while node:
75             if node.key == key:
76                 return True
77             elif key < node.key:
78                 node = node.left
79             else:
80                 node = node.right
81         return False

```

```

77
78 def insert(self, key):
79     """
80     Inserts a key into the Treap. If the key
81     ↳ already exists, the tree is unchanged.
82     """
83     if self.search(key):
84         return # Don't insert duplicates
85
86     new_node = Node(key)
87     l, r = _split(self.root, key)
88     # l has keys < key, r has keys >= key.
89     # Merge new_node with r first, then merge l
90     ↳ with the result.
91     self.root = _merge(l, _merge(new_node, r))
92
93 def delete(self, key):
94     """
95     Deletes a key from the Treap. If the key is
96     ↳ not found, the tree is unchanged.
97     """
98     if not self.search(key):
99         return
100
101     # Split to isolate the node to be deleted.
102     l, r = _split(self.root, key) # l has keys
103     ↳ < key, r has keys >= key
104     _, r_prime = _split(r, key + 1) # r_prime
105     ↳ has keys > key
106
107     # Merge the remaining parts back together.
108     self.root = _merge(l, r_prime)

```

Union Find

Author: PyCPBook Community Source: Based on common implementations in competitive programming resources Description: 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. Time: $O(\alpha(N))$ on average for both `find` and `union` operations, where α is the extremely slow-growing inverse Ackermann function. For all practical purposes, this is considered constant time. Space: $O(N)$ to store the parent and size arrays for N elements. Sta-

tus: Stress-tested

```
1 class UnionFind:
2     """
3     A class that implements the Union-Find data
4     ↳ structure with path compression
5     and union by size optimizations.
6     """
7
8     def __init__(self, n):
9         """
10        Initializes the Union-Find structure for n
11        ↳ elements, where each element
12        is initially in its own set.
13        Args:
14            n (int): The number of elements.
15        """
16        self.parent = list(range(n))
17        self.size = [1] * n
18
19    def find(self, i):
20        """
21        Finds the representative (root) of the set
22        ↳ containing element i.
23        Applies path compression along the way.
24        Args:
25            i (int): The element to find.
26        Returns:
27            int: The representative of the set
28            ↳ containing i.
29        """
30        if self.parent[i] == i:
31            return i
32        self.parent[i] = self.find(self.parent[i])
33        return self.parent[i]
34
35    def union(self, i, j):
36        """
37        Merges the sets containing elements i and
38        ↳ j.
39        Applies union by size.
40        Args:
41            i (int): An element in the first set.
42            j (int): An element in the second set.
43        Returns:
44            bool: True if the sets were merged,
45            ↳ False if they were already in the
46            ↳ same set.
47        """
48        root_i = self.find(i)
49        root_j = self.find(j)
50        if root_i != root_j:
51            if self.size[root_i] <
52                ↳ self.size[root_j]:
53                root_i, root_j = root_j, root_i
54            self.parent[root_j] = root_i
55            self.size[root_i] += self.size[root_j]
56            return True
57        return False
```

Chapter 5

Graph Algorithms

Bellman Ford

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS) Description: 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.

Time: $O(V \cdot E)$, where V is the number of vertices and E is the number of edges. The algorithm iterates through all edges V times. Space: $O(V + E)$ to store the edge list and the distances array. Status: Stress-tested

```
1 def bellman_ford(edges, start_node, n):
2     """
3     Finds shortest paths from a start node,
4     ↪ handling negative weights and
5     detecting negative cycles.
6
7     Args:
8         edges (list[tuple[int, int, int]]): A list
9         ↪ of all edges in the graph,
10        where each tuple is (u, v, weight) for
11        ↪ an edge u -> v.
12        start_node (int): The node from which to
13        ↪ start the search.
14        n (int): The total number of nodes in the
15        ↪ graph.
16
17     Returns:
18         tuple[list[float], bool]: A tuple
19         ↪ containing:
20         - A list of shortest distances.
21         ↪ 'float('inf')' for unreachable
22         ↪ nodes.
23         - A boolean that is True if a negative
24         ↪ cycle is detected, False otherwise.
25     """
```

```
17 if not (0 <= start_node < n):
18     return [float("inf")] * n, False
19
20 dist = [float("inf")] * n
21 dist[start_node] = 0
22
23 for i in range(n - 1):
24     updated = False
25     for u, v, w in edges:
26         if dist[u] != float("inf") and dist[u]
27         ↪ + w < dist[v]:
28             dist[v] = dist[u] + w
29             updated = True
30     if not updated:
31         break
32
33 for u, v, w in edges:
34     if dist[u] != float("inf") and dist[u] + w
35     ↪ < dist[v]:
36         return dist, True
37
38 return dist, False
```

Bipartite Matching

Author: PyCPBook Community Source: CP-Algorithms, USACO Guide Description: 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 re-

cursively 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.

Time: $O(E \cdot V)$, where $V = |U| + |V|$ is the total number of vertices and E is the number of edges. For each vertex in U , we may perform a DFS that traverses the entire graph. Space: $O(V)$ to store the matching and visited arrays for the DFS. Status: Stress-tested

```

1 def bipartite_matching(adj, n1, n2):
2     """
3     Finds the maximum matching in a bipartite
4     ↪ graph.
5
6     Args:
7         adj (list[list[int]]): Adjacency list for
8         ↪ the left partition.
9         `adj[u]` contains a list of neighbors
10        ↪ of node `u` (from the left set)
11        in the right set. Nodes in the left set
12        ↪ are indexed 0 to n1-1.
13        Nodes in the right set are indexed 0 to
14        ↪ n2-1.
15
16        n1 (int): The number of vertices in the
17        ↪ left partition.
18        n2 (int): The number of vertices in the
19        ↪ right partition.
20
21    Returns:
22        int: The size of the maximum matching.
23    """
24    match_right = [-1] * n2
25    matching_size = 0
26
27    def dfs(u, visited):
28        for v in adj[u]:
29            if not visited[v]:
30                visited[v] = True
31                if match_right[v] < 0 or
32                ↪ dfs(match_right[v], visited):
33                    match_right[v] = u
34                    return True
35        return False
36
37    for u in range(n1):
38        visited = [False] * n2
39        if dfs(u, visited):
40            matching_size += 1
41
42    return matching_size

```

Dijkstra

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS) Description: 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).

Time: $O(E \log V)$, where V is the number of vertices and E is the number of edges. The log factor comes from the priority queue operations. Space: $O(V + E)$ to store the adjacency list, distances array, and the priority queue. Status: Stress-tested

```

1 import heapq
2
3
4 def dijkstra(adj, start_node, n):
5     """
6     Finds the shortest paths from a start node to
7     ↪ all other nodes in a graph.
8
9     Args:
10        adj (list[list[tuple[int, int]]]): The
11        ↪ adjacency list representation of
12        the graph. adj[u] contains tuples (v,
13        ↪ weight) for edges u -> v.
14        start_node (int): The node from which to
15        ↪ start the search.
16        n (int): The total number of nodes in the
17        ↪ graph.
18
19    Returns:
20        list[float]: A list of shortest distances
21        ↪ from the start_node to each
22        node. `float('inf')` indicates
23        ↪ an unreachable node.
24
25    """
26    if not (0 <= start_node < n):
27        return [float("inf")] * n
28
29    dist = [float("inf")] * n
30    dist[start_node] = 0
31    pq = [(0, start_node)]
32
33    while pq:
34        d, u = heapq.heappop(pq)
35
36        if d > dist[u]:
37            continue

```

```

30         for v, weight in adj[u]:
31             if dist[u] + weight < dist[v]:
32                 dist[v] = dist[u] + weight
33                 heapq.heappush(pq, (dist[v], v))
34
35     return dist
36
37

```

Dinic

Author: PyCPBook Community Source: CP-Algorithms, KACTL Description: 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 $\text{level}[v] == \text{level}[u] + 1$. 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.

Time: $O(V^2E)$ in general graphs. It is much faster on certain types of graphs, such as $O(E\sqrt{V})$ for bipartite matching and $O(E \min(V^{2/3}, E^{1/2}))$ for unit-capacity networks. Space: $O(V + E)$ to store the graph, capacities, and level information. Status: Stress-tested

```

1 from collections import deque
2
3
4 class Dinic:
5     def __init__(self, n):
6         self.n = n
7         self.graph = [[] for _ in range(n)]
8         self.level = [-1] * n
9         self.ptr = [0] * n
10        self.inf = float("inf")
11
12    def add_edge(self, u, v, cap):
13        # Forward edge
14        self.graph[u].append([v, cap,
15                               ↪ len(self.graph[v])])
16        # Backward edge
17        self.graph[v].append([u, 0,
18                               ↪ len(self.graph[u]) - 1])
19

```

```

18 def _bfs(self, s, t):
19     self.level = [-1] * self.n
20     self.level[s] = 0
21     q = deque([s])
22     while q:
23         u = q.popleft()
24         for i in range(len(self.graph[u])):
25             v, cap, rev = self.graph[u][i]
26             if cap > 0 and self.level[v] < 0:
27                 self.level[v] = self.level[u] +
28                     ↪ 1
29                 q.append(v)
30     return self.level[t] != -1
31
32 def _dfs(self, u, t, pushed):
33     if pushed == 0:
34         return 0
35     if u == t:
36         return pushed
37
38     while self.ptr[u] < len(self.graph[u]):
39         edge_idx = self.ptr[u]
40         v, cap, rev_idx =
41             ↪ self.graph[u][edge_idx]
42
43         if self.level[v] != self.level[u] + 1
44             ↪ or cap == 0:
45             self.ptr[u] += 1
46             continue
47
48         tr = self._dfs(v, t, min(pushed, cap))
49         if tr == 0:
50             self.ptr[u] += 1
51             continue
52
53         self.graph[u][edge_idx][1] -= tr
54         self.graph[v][rev_idx][1] += tr
55         return tr
56     return 0
57
58 def max_flow(self, s, t):
59     if s == t:
60         return 0
61     total_flow = 0
62     while self._bfs(s, t):
63         self.ptr = [0] * self.n
64         pushed = self._dfs(s, t, self.inf)
65         while pushed > 0:
66             total_flow += pushed
67             pushed = self._dfs(s, t, self.inf)
68     return total_flow
69

```

Euler Path

Author: PyCPBook Community Source: CP-Algorithms, Wikipedia (Hierholzer's algorithm) Description: 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 $\text{out-degree} - \text{in-degree} = 1$ (the start), at most one vertex has $\text{in-degree} - \text{out-degree} = 1$ (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.

Time: $O(V + E)$, as each edge and vertex is visited a constant number of times. Space: $O(V + E)$ to store the graph representation, degree counts, and the path. Status: Stress-tested

```

1 from collections import Counter
2
3
4 def find_euler_path(adj, n, directed=False):
5     """
6     Finds an Eulerian path or cycle in a graph.
7
8     Args:
9         adj (list[list[int]]): The adjacency list
10            ↪ representation of the graph.
11            Handles multigraphs if neighbors are
12            ↪ repeated.
13         n (int): The total number of nodes in the
14            ↪ graph.
15         directed (bool): True if the graph is
16            ↪ directed, False otherwise.
17
18     Returns:
19         list[int] / None: A list of nodes
20            ↪ representing the Eulerian path,
21            or None if no such path
22            ↪ exists.
23     """
24     if n == 0:
25         return []
26
27     num_edges = 0
28     if directed:
29         in_degree = [0] * n
30         out_degree = [0] * n
31         for u in range(n):
32             out_degree[u] = len(adj[u])
33             num_edges += len(adj[u])
34             for v in adj[u]:
35                 in_degree[v] += 1
36
37     start_node, end_node_count = -1, 0

```

```

32     for i in range(n):
33         if out_degree[i] - in_degree[i] == 1:
34             if start_node != -1:
35                 return None
36             start_node = i
37         elif in_degree[i] - out_degree[i] == 1:
38             end_node_count += 1
39             if end_node_count > 1:
40                 return None
41         elif in_degree[i] != out_degree[i]:
42             return None
43
44     if start_node == -1:
45         for i in range(n):
46             if out_degree[i] > 0:
47                 start_node = i
48                 break
49     if start_node == -1:
50         return [0] if n > 0 else []
51
52     else:
53         degree = [0] * n
54         for u in range(n):
55             degree[u] = len(adj[u])
56             num_edges += len(adj[u])
57         num_edges //= 2
58
59         odd_degree_nodes = [i for i, d in
60            ↪ enumerate(degree) if d % 2 != 0]
61         if len(odd_degree_nodes) > 2:
62             return None
63
64         start_node = -1
65         if odd_degree_nodes:
66             start_node = odd_degree_nodes[0]
67         else:
68             for i in range(n):
69                 if degree[i] > 0:
70                     start_node = i
71                     break
72             if start_node == -1:
73                 return [0] if n > 0 else []
74
75     adj_counts = [Counter(neighbors) for neighbors
76            ↪ in adj]
77     path = []
78     stack = [start_node]
79
80     while stack:
81         u = stack[-1]
82         if adj_counts[u]:
83             v = next(iter(adj_counts[u]))
84             adj_counts[u][v] -= 1
85             if adj_counts[u][v] == 0:
86                 del adj_counts[u][v]
87
88             if not directed:
89                 adj_counts[v][u] -= 1
90                 if adj_counts[v][u] == 0:
91                     del adj_counts[v][u]
92
93             stack.append(v)
94         else:
95             path.append(stack.pop())
96
97     path.reverse()

```



```

97     if len(path) == num_edges + 1:
98         return path
99     else:
100         return None
101 
```

Floyd Warshall

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS) Description: 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: $\text{dist}(i, j) = \min(\text{dist}(i, j), \text{dist}(i, k) + \text{dist}(k, j))$

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 ($\text{dist}[i][i]$) 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.

Time: $O(V^3)$, where V is the number of vertices. The three nested loops dominate the runtime. Space: $O(V^2)$ to store the distance matrix. Status: Stress-tested

```

1 def floyd_warshall(edges, n):
2     """
3     Finds all-pairs shortest paths in a graph using
4     ↪ the Floyd-Warshall algorithm.
5
6     Args:
7     edges (list[tuple[int, int, int]]): A list
8     ↪ of all edges in the graph,
9     ↪ where each tuple is (u, v, weight) for
10    ↪ an edge u -> v.
11    n (int): The total number of nodes in the
12    ↪ graph.
13
14    Returns:
15    tuple[list[list[float]], bool]: A tuple
16    ↪ containing:
17    - A 2D list of shortest distances.
18    ↪ `dist[i][j]` is the shortest
19    ↪ distance from node `i` to node `j`.
20    ↪ `float('inf')` for unreachable
21    ↪ pairs.
22    - A boolean that is True if a negative
23    ↪ cycle is detected, False otherwise.
24
25    """
26     if n == 0:
27         return [], False

```

```

18     dist = [[float("inf")] * n for _ in range(n)]
19
20     for i in range(n):
21         dist[i][i] = 0
22
23     for u, v, w in edges:
24         dist[u][v] = min(dist[u][v], w)
25
26     for k in range(n):
27         for i in range(n):
28             for j in range(n):
29                 if dist[i][k] != float("inf") and
30                 ↪ dist[k][j] != float("inf"):
31                     dist[i][j] = min(dist[i][j],
32                     ↪ dist[i][k] + dist[k][j])
33
34     has_negative_cycle = False
35     for i in range(n):
36         if dist[i][i] < 0:
37             has_negative_cycle = True
38             break
39
40     return dist, has_negative_cycle

```

Lca Binary Lifting

Author: PyCPBook Community Source: CP-Algorithms, USACO Guide Description: 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 $\text{up}[i][j]$ is built, where $\text{up}[i][j]$ 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. $\text{up}[i][j] = \text{up}[\text{up}[i][j-1]][j-1]$.

2. 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., $\text{up}[u][j] \neq \text{up}[v][j]$). - 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 $\text{up}[u][0]$.

Time: Precomputation is $O(N \log N)$. Each query is $O(\log N)$. Space: $O(N \log N)$ to store the up table. Status: Stress-tested

```

1 class LCA:
2     def __init__(self, n, adj, root=0):
3         self.n = n

```



```

4     self.adj = adj
5     self.max_log = (n).bit_length()
6     self.depth = [-1] * n
7     self.up = [[-1] * self.max_log for _ in
8         ↪ range(n)]
9     self._dfs(root, -1, 0)
10    self._precompute_ancestors()
11
12    def _dfs(self, u, p, d):
13        self.depth[u] = d
14        self.up[u][0] = p
15        for v in self.adj[u]:
16            if v != p:
17                self._dfs(v, u, d + 1)
18
19    def _precompute_ancestors(self):
20        for j in range(1, self.max_log):
21            for i in range(self.n):
22                if self.up[i][j - 1] != -1:
23                    self.up[i][j] =
24                        ↪ self.up[self.up[i][j -
25                            ↪ 1]][j - 1]
26
27    def query(self, u, v):
28        if self.depth[u] < self.depth[v]:
29            u, v = v, u
30
31        for j in range(self.max_log - 1, -1, -1):
32            if self.depth[u] - (1 << j) >=
33                ↪ self.depth[v]:
34                u = self.up[u][j]
35
36        if u == v:
37            return u
38
39        for j in range(self.max_log - 1, -1, -1):
40            if self.up[u][j] != -1 and
41                ↪ self.up[v][j] != self.up[v][j]:
42                u = self.up[u][j]
43                v = self.up[v][j]
44
45        return self.up[u][0]

```

Prim Kruskal

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS) Description: 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.

Time: - 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. Space: - 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. Status: Stress-tested

```

1  import heapq
2  import sys
3  import os
4
5  # Add content directory to path to import the
6  ↪ solution
7  sys.path.append(
8      os.path.join(os.path.dirname(__file__),
9          ↪ "../content/data_structures")
10 )
11 from union_find import UnionFind
12
13 def kruskal(edges, n):
14     """
15     Finds the MST of a graph using Kruskal's
16     ↪ algorithm.
17
18     Args:
19         edges (list[tuple[int, int, int]]): A list
20             ↪ of all edges in the graph,
21             where each tuple is (u, v, weight).
22         n (int): The total number of nodes in the
23             ↪ graph.
24
25     Returns:
26         tuple[int, list[tuple[int, int, int]]]: A
27             ↪ tuple containing:
28             - The total weight of the MST.
29             - A list of edges (u, v, weight) that
30                 ↪ form the MST.
31         Returns (inf, []) if the graph is not
32             ↪ connected and cannot form a single
33             ↪ MST.
34
35     """
36     if n == 0:
37         return 0, []
38
39     sorted_edges = sorted([(w, u, v) for u, v, w in
40         ↪ edges])
41     uf = UnionFind(n)
42     mst_weight = 0
43     mst_edges = []
44
45     for weight, u, v in sorted_edges:
46         if uf.union(u, v):
47             mst_weight += weight

```

```

38         mst_edges.append((u, v, weight))
39         if len(mst_edges) == n - 1:
40             break
41
42     if len(mst_edges) < n - 1:
43         # This indicates the graph is not
44         ↪ connected.
45         # The result is a minimum spanning forest.
46         pass
47
48     return mst_weight, mst_edges
49
50 def prim(adj, n, start_node=0):
51     """
52     Finds the MST of a graph using Prim's
53     ↪ algorithm.
54
55     Args:
56         adj (list[list[tuple[int, int]]]): The
57         ↪ adjacency list representation of
58         the graph. adj[u] contains tuples (v,
59         ↪ weight) for edges u → v.
60         n (int): The total number of nodes in the
61         ↪ graph.
62         start_node (int): The node to start
63         ↪ building the MST from.
64
65     Returns:
66         tuple[int, list[tuple[int, int, int]]]: A
67         ↪ tuple containing:
68         - The total weight of the MST.
69         - A list of edges (u, v, weight) that
70         ↪ form the MST.
71         Returns (inf, []) if the graph is not
72         ↪ connected.
73     """
74     if n == 0:
75         return 0, []
76     if not (0 <= start_node < n):
77         return float("inf"), []
78
79     visited = [False] * n
80     pq = [(0, start_node, -1)] # (weight,
81     ↪ current_node, previous_node)
82     mst_weight = 0
83     mst_edges = []
84     edges_count = 0
85
86     while pq and edges_count < n:
87         weight, u, prev = heapq.heappop(pq)
88
89         if visited[u]:
90             continue
91
92         visited[u] = True
93         mst_weight += weight
94         if prev != -1:
95             mst_edges.append((prev, u, weight))
96         edges_count += 1
97
98         for v, w in adj[u]:
99             if not visited[v]:
100                 heapq.heappush(pq, (w, v, u))
101
102     if edges_count < n:
103         # This indicates the graph is not
104         ↪ connected.

```

```

95         return float("inf"), []
96
97     return mst_weight, mst_edges
98

```

Scc

Author: PyCPBook Community Source: Based on Tarjan's algorithm from Introduction to Algorithms (CLRS) Description: 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.

Time: $O(V + E)$, where V is the number of vertices and E is the number of edges, because the algorithm is based on a single DFS traversal. Space: $O(V)$ to store the discovery times, low-link values, the stack, and the recursion depth of the DFS. Status: Stress-tested

```

1 def find_sccs(adj, n):
2     """
3     Finds all Strongly Connected Components of a
4     ↪ directed graph using Tarjan's algorithm.
5
6     Args:
7         adj (list[list[int]]): The adjacency list
8         ↪ representation of the graph.
9         n (int): The total number of nodes in the
10        ↪ graph.
11
12     Returns:
13         list[list[int]]: A list of lists, where
14         ↪ each inner list contains the
15         nodes of a single Strongly
16         ↪ Connected Component.
17     """
18     if n == 0:
19         return []
20
21     disc = [-1] * n
22     low = [-1] * n
23     on_stack = [False] * n
24     stack = []
25     time = 0

```

```

21     sccs = []
22
23     def tarjan_dfs(u):
24         nonlocal time
25         disc[u] = low[u] = time
26         time += 1
27         stack.append(u)
28         on_stack[u] = True
29
30         for v in adj[u]:
31             if disc[v] == -1:
32                 tarjan_dfs(v)
33                 low[u] = min(low[u], low[v])
34             elif on_stack[v]:
35                 low[u] = min(low[u], disc[v])
36
37         if low[u] == disc[u]:
38             component = []
39             while True:
40                 node = stack.pop()
41                 on_stack[node] = False
42                 component.append(node)
43                 if node == u:
44                     break
45             sccs.append(component)
46
47     for i in range(n):
48         if disc[i] == -1:
49             tarjan_dfs(i)
50
51     return sccs
52

```

Topological Sort

Author: PyCPBook Community Source: Based on Kahn's Algorithm from Introduction to Algorithms (CLRS) Description: 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.

Time: $O(V + E)$, where V is the number of vertices and E is the number of edges. Each vertex

is enqueued and dequeued once, and every edge is processed once. Space: $O(V + E)$ to store the adjacency list, in-degree array, and the queue. Status: Stress-tested

```

1  from collections import deque
2
3
4  def topological_sort(adj, n):
5      """
6      Performs a topological sort on a directed
7      ↪ graph.
8
9      Args:
10         adj (list[list[int]]): The adjacency list
11         ↪ representation of the graph.
12         n (int): The total number of nodes in the
13         ↪ graph.
14
15     Returns:
16         list[int]: A list of nodes in topological
17         ↪ order. Returns an empty list
18         ↪ if the graph contains a cycle.
19
20     """
21     if n == 0:
22         return []
23
24     in_degree = [0] * n
25     for u in range(n):
26         for v in adj[u]:
27             in_degree[v] += 1
28
29     q = deque([i for i in range(n) if in_degree[i]
30         ↪ == 0])
31     topo_order = []
32
33     while q:
34         u = q.popleft()
35         topo_order.append(u)
36
37         for v in adj[u]:
38             in_degree[v] -= 1
39             if in_degree[v] == 0:
40                 q.append(v)
41
42     if len(topo_order) == n:
43         return topo_order
44     else:
45         # Graph has a cycle
46         return []
47

```

Traversal

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS) Description: 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.

Time: $O(V + E)$ for both BFS and DFS, where V is the number of vertices and E is the number of edges. Each vertex and edge is visited exactly once. Space: $O(V)$ in the worst case for storing the queue (BFS) or stack (DFS), and the `visited` array. Status: Stress-tested

```

1 from collections import deque
2
3
4 def bfs(adj, start_node, n):
5     """
6     Performs a Breadth-First Search on a graph.
7
8     Args:
9         adj (list[list[int]]): The adjacency list
10        ↪ representation of the graph.
11        start_node (int): The node from which to
12        ↪ start the traversal.
13        n (int): The total number of nodes in the
14        ↪ graph.
15
16    Returns:
17        list[int]: A list of nodes in the order
18        ↪ they were visited.
19    """
20    if not (0 <= start_node < n):
21        return []
22
23    q = deque([start_node])
24    visited = [False] * n
25    visited[start_node] = True
26    traversal_order = []
27
28    while q:
29        u = q.popleft()
30        traversal_order.append(u)
31        for v in adj[u]:
32            if not visited[v]:
33                visited[v] = True
34                q.append(v)

```

```

31 return traversal_order
32
33
34 def dfs(adj, start_node, n):
35     """
36     Performs a Depth-First Search on a graph.
37
38     Args:
39         adj (list[list[int]]): The adjacency list
40        ↪ representation of the graph.
41        start_node (int): The node from which to
42        ↪ start the traversal.
43        n (int): The total number of nodes in the
44        ↪ graph.
45
46    Returns:
47        list[int]: A list of nodes in the order
48        ↪ they were visited.
49    """
50    if not (0 <= start_node < n):
51        return []
52
53    stack = [start_node]
54    visited = [False] * n
55    # Mark as visited when pushed to stack to avoid
56    ↪ re-adding
57    visited[start_node] = True
58    traversal_order = []
59
60    # This loop produces a traversal order
61    ↪ different from the recursive one.
62    # To get a more standard pre-order traversal
63    ↪ iteratively, we need a slight change.
64
65    # Reset for a more standard iterative DFS
66    ↪ traversal order
67    visited = [False] * n
68    stack = [start_node]
69
70    while stack:
71        u = stack.pop()
72
73        if not visited[u]:
74            visited[u] = True
75            traversal_order.append(u)
76
77            # Add neighbors to the stack in reverse
78            ↪ order to process them in
79            ↪ lexicographical order
80            for v in reversed(adj[u]):
81                if not visited[v]:
82                    stack.append(v)
83
84    return traversal_order

```

Two Sat

Author: PyCPBook Community Source: CP-Algorithms, KACTL Description: 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 $\neg x_i$. 2. Each clause $(a \text{ OR } b)$ is equivalent to two implications: $(\neg a \Rightarrow b)$ and $(\neg b \Rightarrow a)$. 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 $\neg x_i$ 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 $\neg x_i$ and $\neg x_i$ 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 $\neg x_i$ 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.

Time: $O(V + E) = O(N + M)$, where N is the number of variables and M is the number of clauses. The graph has $2N$ vertices and $2M$ edges. Space: $O(N + M)$ to store the implication graph and SCC information. Status: Stress-tested

```

1 import sys
2 import os
3
4 # The stress test runner adds the project root to
5 # ↪ the path.
6 # This allows importing other content modules using
7 # ↪ their full path.
8 from content.graph.scc import find_sccs
9
10 class TwoSAT:
11     def __init__(self, n):
12         self.n = n
13         self.graph = [[] for _ in range(2 * n)]
14
15     def _map_var(self, var):
16         """Maps a 1-indexed variable to a 0-indexed
17         ↪ graph node."""
18         if var > 0:
19             return var - 1
20         return -var - 1 + self.n
21
22     def add_clause(self, i, j):
23         """
24         Adds a clause (i OR j) to the formula.
25         Variables are 1-indexed. A negative value
26         ↪ -k denotes the negation of x_k.
27         This adds two implications: (-i => j) and
28         ↪ (-j => i).
29         """
30         # Add edge for (-i => j)

```

```

31     ↪ self.graph[self._map_var(-i)].append(self._map_var(j))
32     # Add edge for (-j => i)
33     ↪ self.graph[self._map_var(-j)].append(self._map_var(i))
34
35 def solve(self):
36     """
37     Solves the 2-SAT problem.
38
39     Returns:
40         tuple[bool, list[bool] | None]: A tuple
41         ↪ where the first element is
42         True if a solution exists, False
43         ↪ otherwise. If a solution exists,
44         the second element is a list of boolean
45         ↪ values representing a
46         satisfying assignment. Otherwise, it is
47         ↪ None.
48     """
49     sccs = find_sccs(self.graph, 2 * self.n)
50     component_id = [-1] * (2 * self.n)
51     for idx, comp in enumerate(sccs):
52         for node in comp:
53             component_id[node] = idx
54
55     for i in range(self.n):
56         if component_id[i] == component_id[i +
57         ↪ self.n]:
58             return False, None
59
60     assignment = [False] * self.n
61     # sccs are returned in reverse topological
62     ↪ order
63     for i in range(self.n):
64         # If component of x_i comes after
65         ↪ component of not(x_i) in topo order
66         # (i.e., has a smaller index in the
67         ↪ reversed list), then x_i must be
68         ↪ true.
69         if component_id[i] < component_id[i +
70         ↪ self.n]:
71             assignment[i] = True
72
73     return True, assignment

```


Chapter 6

String Algorithms

Aho Corasick

Author: PyCPBook Community Source: CP-Algorithms, KACTL Description: 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. Pre-processing (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.

Time: Preprocessing is $O(L)$, where L is the total length of all patterns. Searching is $O(N+Z)$, where N is the length of the text and Z is the total number of matches found. Space: $O(L)$ to store the trie and associated data. Status: Stress-tested

```
1 from collections import deque
2
3
4 class AhoCorasick:
5     def __init__(self, patterns):
6         self.patterns = patterns
7         self.trie = [{"children": {}, "output": [],
8             ↪ "fail_link": 0}]
9         self._build_trie()
10        self._build_failure_links()
11
12    def _build_trie(self):
13        for i, pattern in enumerate(self.patterns):
14            node_idx = 0
15            for char in pattern:
16                if char not in
17                    ↪ self.trie[node_idx]["children"]:
```

```
16
17
18                ↪ self.trie[node_idx]["children"][char]
19                ↪ = len(self.trie)
20                self.trie.append({"children":
21                    ↪ {}, "output": [],
22                    ↪ "fail_link": 0})
23                node_idx =
24                ↪ self.trie[node_idx]["children"][char]
25                self.trie[node_idx]["output"].append(i)
26
27    def _build_failure_links(self):
28        q = deque()
29        for char, next_node_idx in
30            ↪ self.trie[0]["children"].items():
31            q.append(next_node_idx)
32
33        while q:
34            curr_node_idx = q.popleft()
35            for char, next_node_idx in
36                ↪ self.trie[curr_node_idx]["children"].items():
37                fail_idx =
38                ↪ self.trie[curr_node_idx]["fail_link"]
39                while char not in
40                    ↪ self.trie[fail_idx]["children"]
41                    ↪ and fail_idx != 0:
42                    fail_idx =
43                    ↪ self.trie[fail_idx]["fail_link"]
44
45            if char in
46                ↪ self.trie[fail_idx]["children"]:
47
48                ↪ self.trie[next_node_idx]["fail_link"]
49                ↪ = self.trie[fail_idx][
50                    ↪ "children"
51                ][char]
52            else:
53
54                ↪ self.trie[next_node_idx]["fail_link"]
55                ↪ = 0
56
57        # Append outputs from the failure
58        ↪ link node
59        fail_output_idx =
60        ↪ self.trie[next_node_idx]["fail_link"]
61
62        ↪ self.trie[next_node_idx]["output"].extend
63
64        ↪ self.trie[fail_output_idx]["output"]
65        )
66        q.append(next_node_idx)
67
68    def search(self, text):
69        """
70        Finds all occurrences of the patterns in
71        ↪ the given text.
72
73        Args:
74            text (str): The text to search within.
```

```

54 Returns:
55 list[tuple[int, int]]: A list of
56 ↪ tuples, where each tuple is
57 ↪ (pattern_index, end_index_in_text).
58 ↪ `end_index_in_text` is the
59 ↪ index where the pattern ends.
60 """
61 matches = []
62 curr_node_idx = 0
63 for i, char in enumerate(text):
64     while (
65         char not in
66         ↪ self.trie[curr_node_idx]["children"]
67         ↪ and curr_node_idx != 0
68     ):
69         curr_node_idx =
70         ↪ self.trie[curr_node_idx]["fail_link"]
71
72     if char in
73     ↪ self.trie[curr_node_idx]["children"]:
74         curr_node_idx =
75         ↪ self.trie[curr_node_idx]["children"]
76     else:
77         curr_node_idx = 0
78
79     if self.trie[curr_node_idx]["output"]:
80         for pattern_idx in
81         ↪ self.trie[curr_node_idx]["output"]]:
82             matches.append((pattern_idx,
83                             ↪ i))
84
85 return matches

```

Kmp

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS) Description: 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 $P[0\dots i]$ that is also a suffix of $P[0\dots i]$. A "proper" prefix is one that is not equal to the entire string.

Example: For pattern $P = \text{"ababa"}$, 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 re-checking previously matched characters. Specifically, if a mismatch occurs at `pattern[j]`, we know that the prefix `pattern[0\dots j-1]` matched

the text. The value `lps[j-1]` gives the length of the longest prefix of `pattern[0\dots 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.

Time: $O(N + M)$, where N is the length of the text and M is the length of the pattern. $O(M)$ for building the LPS array and $O(N)$ for the search. Space: $O(M)$ to store the LPS array for the pattern. Status: Stress-tested

```

1 def compute_lps(pattern):
2     """
3     Computes the Longest Proper Prefix Suffix (LPS)
4     ↪ array for the KMP algorithm.
5
6     Args:
7         pattern (str): The pattern string.
8
9     Returns:
10        list[int]: The LPS array for the pattern.
11    """
12    m = len(pattern)
13    lps = [0] * m
14    length = 0
15    i = 1
16    while i < m:
17        if pattern[i] == pattern[length]:
18            length += 1
19            lps[i] = length
20            i += 1
21        else:
22            if length != 0:
23                length = lps[length - 1]
24            else:
25                lps[i] = 0
26                i += 1
27
28    return lps
29
30 def kmp_search(text, pattern):
31     """
32     Finds all occurrences of a pattern in a text
33     ↪ using the KMP algorithm.
34
35     Args:
36         text (str): The text to search within.
37         pattern (str): The pattern to search for.
38
39     Returns:
40        list[int]: A list of 0-based starting
41        ↪ indices of all occurrences
42        ↪ of the pattern in the text.
43    """
44    n = len(text)
45    m = len(pattern)
46    if m == 0:
47        return list(range(n + 1))
48    if n == 0 or m > n:
49        return []
50
51    lps = compute_lps(pattern)
52    occurrences = []
53    i = 0
54    j = 0
55    while i < n:

```



```

53     if pattern[j] == text[i]:
54         i += 1
55         j += 1
56     if j == m:
57         occurrences.append(i - j)
58         j = lps[j - 1]
59     elif i < n and pattern[j] != text[i]:
60         if j != 0:
61             j = lps[j - 1]
62         else:
63             i += 1
64     return occurrences
65

```

Manacher

Author: PyCPBook Community Source: CP-Algorithms, GeeksForGeeks Description: 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.

Time: $O(N)$, where N is the length of the string.
Space: $O(N)$ to store the transformed string and the palindrome radii array. Status: Stress-tested

```

1 def manacher(s):
2     """
3     Finds the longest palindromic substring in a
4     ↪ string using Manacher's algorithm.
5
6     Args:
7         s (str): The input string.
8
9     Returns:
10        str: The longest palindromic substring
11        ↪ found in `s`. If there are
12        ↪ multiple of the same maximum length,
13        ↪ it returns the first one found.

```

```

11 """
12 if not s:
13     return ""
14
15 t = "#" + "#".join(s) + "#"
16 n = len(t)
17 p = [0] * n
18 center, right = 0, 0
19 max_len, max_center = 0, 0
20
21 for i in range(n):
22     mirror = 2 * center - i
23
24     if i < right:
25         p[i] = min(right - i, p[mirror])
26
27     while (
28         i - (p[i] + 1) >= 0
29         and i + (p[i] + 1) < n
30         and t[i - (p[i] + 1)] == t[i + (p[i] +
31         ↪ 1)]
32     ):
33         p[i] += 1
34
35     if i + p[i] > right:
36         center = i
37         right = i + p[i]
38
39     if p[i] > max_len:
40         max_len = p[i]
41         max_center = i
42
43 start = (max_center - max_len) // 2
44 end = start + max_len
45 return s[start:end]

```

Polynomial Hashing

Author: PyCPBook Community Source: CP-Algorithms, KACTL Description: 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 $s = s_0s_1\dots s_{k-1}$ is defined as: $H(s) = (s_0p^0 + s_1p^1 + \dots + s_{k-1}p^{k-1}) \bmod m$ 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 (m_1, m_2). 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 .

Time: Precomputation is $O(N)$. Each query is $O(1)$. Space: $O(N)$ to store precomputed prefix hashes and powers of the base. Status: Stress-tested

```

1 import random
2
3
4 class StringHasher:
5     def __init__(self, s):
6         self.s = s
7         self.n = len(s)
8
9         self.m1 = 10**9 + 7
10        self.m2 = 10**9 + 9
11
12        self.p = random.randint(257, self.m1 - 1)
13
14        self.p_powers1 = [1] * (self.n + 1)
15        self.p_powers2 = [1] * (self.n + 1)
16        for i in range(1, self.n + 1):
17            self.p_powers1[i] = (self.p_powers1[i - 1] * self.p) % self.m1
18            self.p_powers2[i] = (self.p_powers2[i - 1] * self.p) % self.m2
19
20        self.h1 = [0] * (self.n + 1)
21        self.h2 = [0] * (self.n + 1)
22        for i in range(self.n):
23            self.h1[i + 1] = (self.h1[i] * self.p + ord(self.s[i])) % self.m1
24            self.h2[i + 1] = (self.h2[i] * self.p + ord(self.s[i])) % self.m2
25
26        def query(self, l, r):
27            """
28            Computes the hash of the substring
29            ↪ s[l...r-1].
30
31            Args:
32                l (int): The 0-based inclusive starting
33                ↪ index.
34                r (int): The 0-based exclusive ending
35                ↪ index.
36
37            Returns:
38                tuple[int, int]: A tuple containing the
39                ↪ two hash values for the substring.
40            """
41            if l >= r:
42                return 0, 0
43
44            len_sub = r - l
45            hash1 = (
46                self.h1[r] - (self.h1[l] *
47                ↪ self.p_powers1[len_sub]) % self.m1
48                ↪ + self.m1
49            ) % self.m1
50            hash2 = (
51                self.h2[r] - (self.h2[l] *
52                ↪ self.p_powers2[len_sub]) % self.m2
53                ↪ + self.m2
54            ) % self.m2
55            return hash1, hash2

```

Suffix Array

Author: PyCPBook Community Source: CP-Algorithms, GeeksForGeeks Description: 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.

Time: $O(N \log^2 N)$ for building the suffix array and $O(N)$ for the LCP array. Total time complexity is dominated by the suffix array construction. Space: $O(N)$ to store the suffix array, LCP array, and auxiliary arrays for sorting. Status: Stress-tested

```

1 def build_suffix_array(s):
2     """
3     Builds the suffix array for a string using an
4     ↪  $O(N \log^2 N)$  sorting-based approach.
5
6     Args:
7         s (str): The input string.
8
9     Returns:
10        list[int]: The suffix array, containing
11        ↪ starting indices of suffixes in
12        ↪ lexicographically sorted order.
13    """
14    n = len(s)
15    sa = list(range(n))
16    rank = [ord(c) for c in s]
17    k = 1
18    while k < n:
19        sa.sort(key=lambda i: (rank[i], rank[i + k]
20        ↪ if i + k < n else -1))
21        new_rank = [0] * n
22        new_rank[sa[0]] = 0
23        for i in range(1, n):

```

```

21     prev, curr = sa[i - 1], sa[i]
22     r_prev = (rank[prev], rank[prev + k] if
23     ↪ prev + k < n else -1)
24     r_curr = (rank[curr], rank[curr + k] if
25     ↪ curr + k < n else -1)
26     if r_curr == r_prev:
27         new_rank[curr] = new_rank[prev]
28     else:
29         new_rank[curr] = new_rank[prev] + 1
30     rank = new_rank
31     if rank[sa[-1]] == n - 1:
32         break
33     k *= 2
34     return sa
35
36 def build_lcp_array(s, sa):
37     """
38     Builds the LCP array using Kasai's algorithm in
39     ↪ O(N) time.
40
41     Args:
42     s (str): The input string.
43     sa (list[int]): The suffix array for the
44     ↪ string `s`.
45
46     Returns:
47     list[int]: The LCP array. `lcp[i]` is the
48     ↪ LCP of suffixes `sa[i-1]` and `sa[i]`.
49     ↪ `lcp[0]` is conventionally 0.
50
51     """
52     n = len(s)
53     if n == 0:
54         return []
55
56     rank = [0] * n
57     for i in range(n):
58         rank[sa[i]] = i
59
60     lcp = [0] * n
61     h = 0
62     for i in range(n):
63         if rank[i] == 0:
64             continue
65         j = sa[rank[i] - 1]
66         if h > 0:
67             h -= 1
68         while i + h < n and j + h < n and s[i + h]
69             ↪ == s[j + h]:
70             h += 1
71         lcp[rank[i]] = h
72     return lcp

```

Z Algorithm

Author: PyCPBook Community Source: CP-Algorithms, USACO Guide Description: 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 ($i \leq r$), 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 $z[k] \geq r - i + 1$, 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 .

Time: $O(N)$, where N is the length of the string. Space: $O(N)$ to store the Z-array. Status: Stress-tested

```

1 def z_function(s):
2     """
3     Computes the Z-array for a given string.
4
5     Args:
6     s (str): The input string.
7
8     Returns:
9     list[int]: The Z-array for the string `s`.
10
11     """
12     n = len(s)
13     if n == 0:
14         return []
15
16     z = [0] * n
17     l, r = 0, 0
18     for i in range(1, n):
19         if i <= r:
20             z[i] = min(r - i + 1, z[i - l])
21             while i + z[i] < n and s[z[i]] == s[i +
22             ↪ z[i]]:
23                 z[i] += 1
24             if i + z[i] - 1 > r:
25                 l, r = i, i + z[i] - 1
26     return z

```

Chapter 7

Mathematics & Number Theory

Chinese Remainder Theorem

Author: PyCPBook Community Source: CP-
Algorithms Description: Implements a solver for a
system of linear congruences using the Chinese Re-
mainder Theorem (CRT). Given a system of con-
gruences: $x \equiv a_1 \pmod{n_1}$ $x \equiv a_2 \pmod{n_2}$...
 $x \equiv a_k \pmod{n_k}$ the algorithm finds a solution x
that satisfies all of them. This implementation han-
dles the general case where the moduli n_i are not
necessarily pairwise coprime.

The algorithm works by iteratively combin-
ing pairs of congruences. Given a solution for
the first $i-1$ congruences, $x \equiv a_{\text{res}} \pmod{n_{\text{res}}}$, it combines this with the i -th congruence
 $x \equiv a_i \pmod{n_i}$.

This requires solving a linear congruence of the
form $k * n_{\text{res}} \equiv a_i - a_{\text{res}} \pmod{n_i}$. A solution exists if and only if $(a_i - a_{\text{res}})$
is divisible by $g = \gcd(n_{\text{res}}, n_i)$. If a solution exists, the two congruences are
merged into a new one: $x \equiv a_{\text{new}} \pmod{n_{\text{new}}}$, where $n_{\text{new}} = \text{lcm}(n_{\text{res}}, n_i)$.
This process is repeated for all congruences. If at
any step a solution does not exist, the entire system
has no solution.

Time: $O(K \cdot \log(\max(n_i)))$, where K is the
number of congruences. Each merge step involves
`extended_gcd`, which is logarithmic. Space: $O(1)$
Status: Stress-tested

```
1 from content.math.modular_arithmetic import
2     ↪ extended_gcd
3
4 def chinese_remainder_theorem(remainders, moduli):
5     """
6     Solves a system of linear congruences.
7     `x ≡ remainders[i] (mod moduli[i])` for
8     ↪ all i.
9
10    Args:
11        remainders (list[int]): A list of
12        ↪ remainders (a_i).
13        moduli (list[int]): A list of moduli (n_i).
14
15    Returns:
16        tuple[int, int] | None: A tuple `(result,
17        ↪ lcm)` representing the solution
18        `x ≡ result (mod lcm)`, or None if no
19        ↪ solution exists.
20    """
21    if not remainders or not moduli or
22    ↪ len(remainders) != len(moduli):
23        return 0, 1
```

```
19
20 a1 = remainders[0]
21 n1 = moduli[0]
22
23 for i in range(1, len(remainders)):
24     a2 = remainders[i]
25     n2 = moduli[i]
26
27     g, x, _ = extended_gcd(n1, n2)
28
29     if (a1 - a2) % g != 0:
30         return None
31
32     # Solve k * n1 ≡ a2 - a1 (mod n2)
33     # k * (n1/g) ≡ (a2 - a1)/g (mod n2/g)
34     # k ≡ ((a2 - a1)/g) * inv(n1/g) (mod
35     ↪ n2/g)
36     # inv(n1/g) mod (n2/g) is x from
37     ↪ extended_gcd(n1, n2)
38     k0 = (x * ((a2 - a1) // g)) % (n2 // g)
39
40     # New solution: x = a1 + k*n1. With k = k0
41     ↪ + t*(n2/g)
42     # x = a1 + (k0 + t*(n2/g)) * n1 = (a1 +
43     ↪ k0*n1) + t*lcm(n1, n2)
44     a1 = a1 + k0 * n1
45     n1 = n1 * (n2 // g) # lcm(n1, n2)
46     a1 %= n1
47
48 return a1, n1
```

Miller Rabin

Author: PyCPBook Community Source: Introduc-
tion to Algorithms (CLRS), Wikipedia Description:
Implements the Miller-Rabin primality test, a prob-
abilistic algorithm for determining whether a given
number is prime. It is highly efficient and is the
standard method for primality testing in compet-
itive 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 $x = a^d \pmod{n}$. 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 $x =$
 $x^2 \pmod{n}$. 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.

Time: $O(k \cdot (\log n)^2)$, where k is the number of witnesses. Space: $O(1)$ Status: Stress-tested

```

1 from content.math.modular_arithmetic import power
2
3
4 def is_prime(n):
5     """
6     Checks if a number is prime using the
7     ↪ Miller-Rabin primality test.
8     This implementation is deterministic for all
9     ↪ integers up to  $2^{64}$ .
10
11     Args:
12         n (int): The number to test for primality.
13
14     Returns:
15         bool: True if n is prime, False otherwise.
16     """
17     if n < 2:
18         return False
19     if n == 2 or n == 3:
20         return True
21     if n % 2 == 0 or n % 3 == 0:
22         return False
23
24     d = n - 1
25     s = 0
26     while d % 2 == 0:
27         d //= 2
28         s += 1
29
30     # A set of witnesses that is deterministic for
31     ↪ all 64-bit integers.
32     witnesses = [2, 3, 5, 7, 11, 13, 17, 19, 23,
33     ↪ 29, 31, 37]
34
35     for a in witnesses:
36         if a >= n:
37             break
38         x = power(a, d, n)
39         if x == 1 or x == n - 1:
40             continue
41
42         is_composite = True
43         for _ in range(s - 1):
44             x = power(x, 2, n)
45             if x == n - 1:
46                 is_composite = False
47                 break
48         if is_composite:
49             return False
50     return True

```

Modular Arithmetic

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS), CP-Algorithms Description: 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.

Time: - `power`: $O(\log(exp))$ - `extended_gcd`: $O(\log(\min(a, b)))$ - `mod_inverse`: $O(\log m)$ Space: - All functions use $O(1)$ extra space for iterative versions. Status: Stress-tested

```

1 def power(base, exp, mod):
2     """
3     Computes (base^exp) % mod using binary
4     ↪ exponentiation.
5     """
6     res = 1
7     base %= mod
8     while exp > 0:
9         if exp % 2 == 1:
10             res = (res * base) % mod
11             base = (base * base) % mod
12             exp //= 2
13     return res
14
15 def extended_gcd(a, b):
16     """
17     Returns (gcd, x, y) such that a*x + b*y =
18     ↪ gcd(a, b).
19     """
20     if a == 0:
21         return b, 0, 1
22     gcd, x1, y1 = extended_gcd(b % a, a)
23     x = y1 - (b // a) * x1
24     y = x1

```



```

24     return gcd, x, y
25
26
27 def mod_inverse(a, m):
28     """
29     Computes the modular multiplicative inverse of
30     ↪ a modulo m.
31     Returns None if the inverse does not exist.
32     """
33     gcd, x, y = extended_gcd(a, m)
34     if gcd != 1:
35         return None
36     else:
37         return (x % m + m) % m

```

Ntt

Author: PyCPBook Community Source: CP-Algorithms, KACTL Description: 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 $c \cdot 2^k + 1$ 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.

Time: $O(N \log N)$ for multiplying two polynomials of degree up to N . Space: $O(N)$ to store the polynomials and intermediate values. Status: Stress-tested

```

1  from content.math.modular_arithmetic import power
2
3  MOD = 998244353
4  ROOT = 3
5  ROOT_PW = 1 << 23
6  ROOT_INV = power(ROOT, MOD - 2, MOD)
7
8
9  def ntt(a, invert):
10     n = len(a)
11     j = 0
12     for i in range(1, n):
13         bit = n >> 1
14         while j & bit:
15             j ^= bit

```

```

16         bit >>= 1
17         j ^= bit
18         if i < j:
19             a[i], a[j] = a[j], a[i]
20
21     length = 2
22     while length <= n:
23         wlen = power(ROOT_INV if invert else ROOT,
24             ↪ (MOD - 1) // length, MOD)
25         i = 0
26         while i < n:
27             w = 1
28             for j in range(length // 2):
29                 u = a[i + j]
30                 v = (a[i + j + length // 2] * w) %
31                     ↪ MOD
32                 a[i + j] = (u + v) % MOD
33                 a[i + j + length // 2] = (u - v +
34                     ↪ MOD) % MOD
35                 w = (w * wlen) % MOD
36             i += length
37         length <<= 1
38
39     if invert:
40         n_inv = power(n, MOD - 2, MOD)
41         for i in range(n):
42             a[i] = (a[i] * n_inv) % MOD
43
44 def multiply(a, b):
45     if not a or not b:
46         return []
47
48     res_len = len(a) + len(b) - 1
49     n = 1
50     while n < res_len:
51         n <<= 1
52
53     fa = a[:] + [0] * (n - len(a))
54     fb = b[:] + [0] * (n - len(b))
55
56     ntt(fa, False)
57     ntt(fb, False)
58
59     for i in range(n):
60         fa[i] = (fa[i] * fb[i]) % MOD
61
62     ntt(fa, True)
63
64     return fa[:res_len]

```

Pollard Rho

Author: PyCPBook Team Source: CP-Algorithms, Wikipedia Description: 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`. Time: The complexity is heuristic. Finding a factor `p` takes roughly $O(p^{1/2})$ with trial division, but Pollard's Rho takes about $O(p^{1/4})$ or $O(n^{1/4})$ on average. The overall factorization time depends on the size of the prime factors of `n`. Space: $O(\log n)$ for recursion depth in factorization. Status: Stress-tested

```

1 import math
2 import random
3 from content.math.miller_rabin import is_prime
4
5
6 def _pollard_rho_factor(n):
7     """Finds a non-trivial factor of n using
8     ↪ Pollard's Rho. n must be composite."""
9     if n % 2 == 0:
10         return 2
11
12     f = lambda val, c: (pow(val, 2, n) + c) % n
13
14     while True:
15         x = random.randint(1, n - 2)
16         y = x
17         c = random.randint(1, n - 1)
18         d = 1
19
20         while d == 1:
21             x = f(x, c)
22             y = f(f(y, c), c)
23             d = math.gcd(abs(x - y), n)
24
25         if d != n:
26             return d
27
28 def factorize(n):
29     if n <= 1:
30         return []
31
32     factors = []
33
34     def get_factors(num):
35         if num <= 1:
36             return
37         if is_prime(num):
38             factors.append(num)
39             return
40
41         factor = _pollard_rho_factor(num)
42         get_factors(factor)
43         get_factors(num // factor)
44
45     get_factors(n)
46     factors.sort()
47     return factors
48

```

Sieve

Author: PyCPBook Community Source: CP-Algorithms, Wikipedia Description: 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.

Time: $O(N \log \log N)$, which is nearly linear. Space: $O(N)$ to store the boolean sieve array. Status: Stress-tested

```

1 def sieve(n):
2     """
3     Generates a sieve of primes up to n using the
4     ↪ Sieve of Eratosthenes.
5
6     Args:
7         n (int): The upper limit for the sieve
8         ↪ (inclusive).
9
10    Returns:
11        list[bool]: A boolean list of size n+1
12        ↪ where is_prime[i] is True if i
13        ↪ is a prime number, and False
14        ↪ otherwise.
15    """
16    if n < 2:
17        return [False] * (n + 1)
18
19    is_prime = [True] * (n + 1)
20    is_prime[0] = is_prime[1] = False
21
22    for p in range(2, int(n**0.5) + 1):
23        if is_prime[p]:
24            for multiple in range(p * p, n + 1, p):
25                is_prime[multiple] = False
26
27    return is_prime
28

```

Chapter 8

Geometry

Convex Hull

Author: PyCPBook Community Source: CP-Algorithms (Monotone Chain Algorithm) Description: 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. Time: $O(N \log N)$, dominated by the initial sorting of points. Space: $O(N)$ to store the points and the resulting hull. Status: Stress-tested

```
1 from content.geometry.point import Point,
2   ↪ orientation
3
4 def convex_hull(points):
5     """
6     Computes the convex hull of a set of points
7     ↪ using the Monotone Chain algorithm.
8
9     Args:
10         points (list[Point]): A list of Point
11         ↪ objects.
12
13     Returns:
14         list[Point]: A list of Point objects
15         ↪ representing the vertices of the
16         ↪ convex hull in
17         ↪ counter-clockwise order.
18         ↪ Returns an empty
19         ↪ list if fewer than 3 points
20         ↪ are provided.
```

```
15 """
16 n = len(points)
17 if n <= 2:
18     return points
19
20 # Sort points lexicographically
21 points.sort()
22
23 # Build lower hull
24 lower_hull = []
25 for p in points:
26     while (
27         len(lower_hull) >= 2 and
28         ↪ orientation(lower_hull[-2],
29         ↪ lower_hull[-1], p) <= 0
30     ):
31         lower_hull.pop()
32     lower_hull.append(p)
33
34 # Build upper hull
35 upper_hull = []
36 for p in reversed(points):
37     while (
38         len(upper_hull) >= 2 and
39         ↪ orientation(upper_hull[-2],
40         ↪ upper_hull[-1], p) <= 0
41     ):
42         upper_hull.pop()
43     upper_hull.append(p)
44
45 # Combine the hulls, removing duplicate
46 ↪ start/end points
47 return lower_hull[:-1] + upper_hull[:-1]
```

Line Intersection

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS), CP-Algorithms Description: 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`. Time: All functions are $O(1)$. Space: All functions are $O(1)$. Status: Stress-tested

```

1 from content.geometry.point import Point,
  ↳ orientation
2
3
4 def on_segment(p, q, r):
5     """
6     Given three collinear points p, q, r, the
7     ↳ function checks if point q
8     ↳ lies on line segment 'pr'.
9     """
10    return (
11        q.x <= max(p.x, r.x)
12        and q.x >= min(p.x, r.x)
13        and q.y <= max(p.y, r.y)
14        and q.y >= min(p.y, r.y)
15    )
16
17 def segments_intersect(p1, q1, p2, q2):
18     """
19     Checks if line segment 'p1q1' and 'p2q2'
20     ↳ intersect.
21     """
22    o1 = orientation(p1, q1, p2)
23    o2 = orientation(p1, q1, q2)
24    o3 = orientation(p2, q2, p1)
25    o4 = orientation(p2, q2, q1)
26
27    if o1 != 0 and o2 != 0 and o3 != 0 and o4 != 0:
28        if o1 != o2 and o3 != o4:
29            return True
30        return False
31
32    if o1 == 0 and on_segment(p1, p2, q1):
33        return True
34    if o2 == 0 and on_segment(p1, q2, q1):
35        return True
36    if o3 == 0 and on_segment(p2, p1, q2):
37        return True
38    if o4 == 0 and on_segment(p2, q1, q2):
39        return True
40    return False
41
42
43 def line_line_intersection(p1, p2, p3, p4):
44     """
45     Finds the intersection point of two infinite
46     ↳ lines defined by (p1, p2) and (p3, p4).
47     Returns the intersection point as a Point
48     ↳ object with float coordinates,
49     ↳ or None if the lines are parallel or collinear.
50     """
51    v1 = p2 - p1

```

```

50    v2 = p4 - p3
51    denominator = v1.cross(v2)
52
53    if abs(denominator) < 1e-9:
54        return None
55
56    t = (p3 - p1).cross(v2) / denominator
57    return p1 + v1 * t
58

```

Point

Author: PyCPBook Community Source: KACTL, CP-Algorithms, standard geometry texts Description: 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. Time: All `Point` methods and the `orientation` function are $O(1)$. Space: $O(1)$ per `Point` object. Status: Stress-tested

```

1 import math
2
3
4 class Point:
5     def __init__(self, x, y):
6         self.x = x
7         self.y = y
8
9     def __repr__(self):
10        return f"Point({self.x}, {self.y})"
11
12    def __eq__(self, other):
13        return self.x == other.x and self.y ==
14        ↳ other.y
15
16    def __lt__(self, other):
17        if self.x != other.x:
18            return self.x < other.x
19        return self.y < other.y
20
21    def __add__(self, other):
22        return Point(self.x + other.x, self.y +
23        ↳ other.y)
24
25    def __sub__(self, other):
26        return Point(self.x - other.x, self.y -
27        ↳ other.y)

```

```

26 def __mul__(self, scalar):
27     return Point(self.x * scalar, self.y *
    ↪ scalar)
28
29 def __truediv__(self, scalar):
30     return Point(self.x / scalar, self.y /
    ↪ scalar)
31
32 def dot(self, other):
33     return self.x * other.x + self.y * other.y
34
35 def cross(self, other):
36     return self.x * other.y - self.y * other.x
37
38 def dist_sq(self, other):
39     dx = self.x - other.x
40     dy = self.y - other.y
41     return dx * dx + dy * dy
42
43
44 def orientation(p, q, r):
45     """
46     Determines the orientation of the ordered
    ↪ triplet (p, q, r).
47
48     Returns:
49         int: > 0 for counter-clockwise, < 0 for
    ↪ clockwise, 0 for collinear.
50     """
51     val = (q.x - p.x) * (r.y - q.y) - (q.y - p.y) *
    ↪ (r.x - q.x)
52     if val == 0:
53         return 0
54     return 1 if val > 0 else -1
55

```

Polygon Area

Author: PyCPBook Community Source: Wikipedia (Shoelace formula), CP-Algorithms
 Description: 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). Time: $O(N)$ for both area and centroid calculation, where N is the number of vertices. Space: $O(1)$ Status: Stress-tested

```

1 from content.geometry.point import Point
2
3
4 def polygon_area(vertices):
5     """
6     Calculates the area of a simple polygon using
    ↪ the Shoelace formula.
7
8     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

Author: PyCPBook Community Source: Introduction to Algorithms (CLRS), CP-Algorithms
Description: 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 .

Time: - 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. **Space:** - LIS: $O(N)$ - LCS: $O(N \cdot M)$ - 0/1 Knapsack: $O(W)$ (space-optimized) **Status:** Stress-tested

```
3
4 def longest_increasing_subsequence(arr):
5     """
6     Finds the length of the longest increasing
7     ↪ subsequence in  $O(N \log N)$ .
8     """
9     if not arr:
10        return 0
11
12    tails = []
13    for num in arr:
14        idx = bisect.bisect_left(tails, num)
15        if idx == len(tails):
16            tails.append(num)
17        else:
18            tails[idx] = num
19    return len(tails)
20
21 def longest_common_subsequence(s1, s2):
22     """
23     Finds the length of the longest common
24     ↪ subsequence in  $O(N \cdot M)$ .
25     """
26    n, m = len(s1), len(s2)
27    dp = [[0] * (m + 1) for _ in range(n + 1)]
28
29    for i in range(1, n + 1):
30        for j in range(1, m + 1):
31            if s1[i - 1] == s2[j - 1]:
32                dp[i][j] = 1 + dp[i - 1][j - 1]
33            else:
34                dp[i][j] = max(dp[i - 1][j],
35                               ↪ dp[i][j - 1])
36    return dp[n][m]
37
38 def knapsack_01(weights, values, capacity):
39     """
40     Solves the 0/1 Knapsack problem with space
41     ↪ optimization.
42     """
43    n = len(weights)
44    dp = [0] * (capacity + 1)
45
46    for i in range(n):
47        for w in range(capacity, weights[i] - 1,
48                       ↪ -1):
49            dp[w] = max(dp[w], values[i] + dp[w -
50                       ↪ weights[i]])
51
52    return dp[capacity]
```

```
1 import bisect
```

```
2
```

Dp Optimizations

Author: PyCPBook Community Source: CP-Algorithms, USACO Guide Description: 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 $dp[i][j] = C[i][j] + \min_{i \leq k < j} (dp[i][k] + dp[k+1][j])$, such as in the optimal binary search tree problem. It can be used if the cost function C satisfies the quadrangle inequality ($C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$ for $a \leq b \leq c \leq d$). The key insight is that the optimal splitting point k for $dp[i][j]$, denoted $opt[i][j]$, is monotonic: $opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$. 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 $dp[i][j] = \min_{0 \leq k < j} (dp[i-1][k] + C[k][j])$. 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.

Time: Varies by optimization. CHT: $O(N \log N)$ or $O(N)$ amortized. **Space:** Varies. **Status:** Conceptual (Knuth-Yao, D&C), Stress-tested (CHT example).

```
1 import sys
2 import os
3
4 # The stress test runner adds the project root to
  ↪ the path.
```

```
5 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(
  ↪ ".../..")
6 from content.data_structures.line_container import
  ↪ LineContainer
7
8
9 def convex_hull_trick_example(p, C):
10     """
11     Solves an example problem using the Convex Hull
12     ↪ Trick.
13     Problem: Given n points on a line with
14     ↪ increasing coordinates p[0]...p[n-1],
15     find the minimum cost to travel from point 0 to
16     ↪ point n-1. The cost of
17     ↪ jumping from point i to point j is (p[j] -
18     ↪ p[i])^2 + C.
19
20     DP recurrence: dp[i] = min_{j < i} (dp[j] + (p[i]
21     ↪ - p[j])^2 + C)
22     This can be rewritten as:
23     dp[i] = p[i]^2 + C + min_{j < i} (-2*p[j]*p[i] +
24     ↪ dp[j] + p[j]^2)
25     This fits the form y = mx + c, where:
26     - x = p[i]
27     - m_j = -2 * p[j]
28     - c_j = dp[j] + p[j]^2
29     Since p is increasing, the slopes m_j are
30     ↪ decreasing, matching the
31     ↪ 'LineContainer's requirement.
32
33     Args:
34     p (list[int]): A list of increasing integer
35     ↪ coordinates.
36     C (int): A constant cost for each jump.
37
38     Returns:
39     int: The minimum cost to reach the last
40     ↪ point.
41     """
42     n = len(p)
43     if n <= 1:
44         return 0
45
46     dp = [0] * n
47     lc = LineContainer()
48
49     # Base case: dp[0] = 0. Add the first line to
50     ↪ the container.
51     # m_0 = -2*p[0], c_0 = dp[0] + p[0]^2 = p[0]^2
52     lc.add(-2 * p[0], p[0] ** 2)
53
54     for i in range(1, n):
55         # Query for the minimum value at x = p[i]
56         min_val = lc.query(p[i])
57         dp[i] = p[i] ** 2 + C + min_val
58
59         # Add the new line corresponding to state i
60         ↪ to the container
61         # m_i = -2*p[i], c_i = dp[i] + p[i]^2
62         lc.add(-2 * p[i], dp[i] + p[i] ** 2)
63
64     return dp[n - 1]
```