

Python Competitive Programming Notebook

PyCPBook Community

October 8, 2025

Abstract

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

Contents

1	Fundamentals	3
2	Standard Library	16
3	Contest & Setup	20
4	Data Structures	23
5	Graph Algorithms	32
6	String Algorithms	41
7	Mathematics & Number Theory	46
8	Geometry	50
9	Dynamic Programming	53

Chapter 1

Fundamentals

Binary Search

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

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

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

```
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 if
11            ↳ 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
23    return -1
```

Bitwise Operations

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

Operators and meaning:

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

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

Common patterns:

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

Complexities:

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

Tips and pitfalls:

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

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

This module provides:

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

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

```

1 def bit_set(x, i):
2     return x | (1 << i)
3
4
5 def bit_unset(x, i):
6     return x & ~(1 << i)
7
8
9 def bit_toggle(x, i):
10    return x ^ (1 << i)
11
12
13 def bit_test(x, i):
14    return ((x >> i) & 1) == 1
15
16
17 def is_power_of_two(x):
18    return x > 0 and (x & (x - 1)) == 0
19
20
21 def count_set_bits(x):
22    c = 0
23    while x:
24        x &= x - 1
25        c += 1
26    return c
27
28
29 def lowest_set_bit(x):
30    return x & -x
31
32
33 def msb_index(x):
34    return x.bit_length() - 1 if x else -1
35
36
37 def lsb_index(x):
38    return (x & -x).bit_length() - 1 if x else -1
39
40
41 def iterate_submasks(mask):
42    s = mask
43    while s:
44        yield s
45        s = (s - 1) & mask
46    yield 0
47
48
49 def iterate_bits(mask):

```

```

50 m = mask
51 while m:
52     lb = m & -m
53     yield lb.bit_length() - 1
54     m ^= lb
55
56
57 def next_power_of_two(x):
58     if x <= 0:
59         return 1
60     return 1 << (x - 1).bit_length()
61
62
63

```

Dictionaries Hash Maps

Dictionaries (hash maps) are essential in competitive programming for constant average-time lookups, counts, grouping, and memoization. This section covers core operations, performance notes, common patterns like frequency maps and "dictionary of lists", and typical CP use-cases.

Basics:

- Creation: `{}`, `dict()`, from pairs: `dict([(k, v), ...])`
- Access: `d[k]`, safe get: `d.get(k, default)`, membership: `k in d`
- Update: `d[k] = v`, `d.update(other)`
- Removal: `del d[k]`, `d.pop(k, default)`, `d.popitem()`
- Iteration: `for k in d, for k, v in d.items(), for v in d.values()`
- Copy: `d.copy()`, `dict(d)`; note nested structures require explicit copies

Performance and behavior:

- Average $O(1)$ insert/lookup/delete with good hashing
- Keys must be hashable and compared by equality; do not use mutable keys
- Python preserves insertion order, but algorithms should not depend on it

Frequency maps and counters:

- Manual counting: increment with guard or using `get`
- `collections.Counter` provides counting, `most_common`, and arithmetic, but manual dicts are often faster to type and sufficient for contests

Dictionary of lists (grouping/bucketing):

- Guard pattern: initialize list when missing

- `setdefault`: concise way to initialize-and-append
- `defaultdict(list)`: best when many keys are created dynamically

Memoization tables:

- Use dicts for top-down DP caching when states are irregular or sparse
- Compare with `functools.lru_cache` when using pure functions

Common CP use-cases:

- Two-sum index map, sliding window frequency map
- Grouping by key (e.g., modulo buckets), adjacency lists
- Coordinate compression maps from values to ranks

Pitfalls:

- Mutating a shared list across keys by accident; create new lists per key
- Using unhashable keys (lists, dicts); prefer tuples for composite keys
- Memory growth from unbounded maps; clear or reuse when appropriate

```

1 from collections import defaultdict, Counter
2
3
4 def freq_map(arr):
5     d = {}
6     for x in arr:
7         d[x] = d.get(x, 0) + 1
8     return d
9
10
11 def freq_map_counter(arr):
12     return dict(Counter(arr))
13
14
15 def group_by_mod(arr, k):
16     d = {}
17     for x in arr:
18         r = x % k
19         if r not in d:
20             d[r] = []
21         d[r].append(x)
22     return d
23
24
25 def add_to_bucket_dict(d, k, v):
26     if k not in d:
27         d[k] = []
28     d[k].append(v)
29     return d
30
31
32 def add_to_bucket_setdefault(d, k, v):
33     d.setdefault(k, []).append(v)
34     return d
35
36
37 def add_to_bucket_defaultdict(d, k, v):

```

```

38     d[k].append(v)
39     return d
40
41
42 def make_defaultdict_list():
43     return defaultdict(list)
44
45
46

```

Greedy Algorithms

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

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

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

```

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 of
8         ↪ 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 ascending
20     ↪ order
21     activities.sort(key=lambda x: x[1])
22
23     count = 1
24     last_finish_time = activities[0][1]
25
26     for i in range(1, len(activities)):
27         start_time, finish_time = activities[i]
28         if start_time >= last_finish_time:
29             count += 1
30             last_finish_time = finish_time
31
32     return count

```

```

25         last_finish_time = finish_time
26
27     return count
28

```

Is Checks

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

```

1 from content.math.miller_rabin import is_prime as
  ↪ mr_is_prime
2
3
4 def is_prime(n):
5     if n < 2:
6         return False
7     small_primes = [2, 3, 5]
8     for p in small_primes:
9         if n == p:
10            return True
11        if n % p == 0:
12            return False
13    if n < 10 ** 9:
14        i = 5
15        while i * i <= n:
16            if n % i == 0 or n % (i + 2) == 0:
17                return False
18            i += 6
19        return True
20    return mr_is_prime(n)
21
22
23 def is_palindrome(s, normalize=False, alnum_only=False):
24     if not isinstance(s, str):
25         s = str(s)
26     t = s
27     if normalize:
28         t = t.casefold()
29     if alnum_only:
30         t = "".join(ch for ch in t if ch.isalnum())
31     return t == t[::-1]
32
33
34 def is_checks_examples():
35     prime_small_true = is_prime(97)
36     prime_small_false = is_prime(221)
37     prime_large_probable = is_prime(1000000007)
38
39     pal_simple_true = is_palindrome("racecar")
40     pal_simple_false = is_palindrome("python")
41     pal_casefold_true = is_palindrome("AbBa",
42     ↪ normalize=True)
43     pal_alnum_true = is_palindrome("A man, a plan, a
44     ↪ canal: Panama!", normalize=True, alnum_only=True)
45     pal_int_true = is_palindrome(12321)
46
47     ch_digit = "2"
48     ch_superscript_two = "²"
49     ch_roman_twelve = "XII"
50     ch_arabic_indic_two = "٢"
51
52     digits_info = {
53         "digit_isdigit": ch_digit.isdigit(),

```

```

52         "digit_isdecimal": ch_digit.isdecimal(),
53         "digit_isnumeric": ch_digit.isnumeric(),
54         "sup2_isdigit": ch_superscript_two.isdigit(),
55         "sup2_isdecimal": ch_superscript_two.isdecimal(),
56         "sup2_isnumeric": ch_superscript_two.isnumeric(),
57         "roman_isdigit": ch_roman_twelve.isdigit(),
58         "roman_isdecimal": ch_roman_twelve.isdecimal(),
59         "roman_isnumeric": ch_roman_twelve.isnumeric(),
60         "arabic_isdigit": ch_arabic_indic_two.isdigit(),
61         "arabic_isdecimal":
62     ↪ ch_arabic_indic_two.isdecimal(),
63         "arabic_isnumeric":
64     ↪ ch_arabic_indic_two.isnumeric(),
65     }
66
67     str_flags = {
68         "alpha": "abcXYZ".isalpha(),
69         "alnum": "abc123".isalnum(),
70         "lower": "hello".islower(),
71         "upper": "WORLD".isupper(),
72         "space": "\t\n".isspace(),
73         "ascii_true": "ASCII".isascii(),
74         "ascii_false": "π".isascii(),
75     }
76
77     agg_logic = {
78         "any_true": any([0, 0, 3]),
79         "all_true": all([1, 2, 3]),
80         "sum_": sum([1, 2, 3, 4]),
81         "min_": min([5, 2, 9]),
82         "max_": max([5, 2, 9]),
83     }
84
85     conv_num = {
86         "abs_": abs(-42),
87         "round_": round(3.6),
88         "divmod_": divmod(17, 5),
89         "pow_mod": pow(2, 10, 1000),
90         "ord_A": ord("A"),
91         "chr_65": chr(65),
92         "bin_": bin(10),
93         "oct_": oct(10),
94         "hex_": hex(255),
95     }
96
97     sorting_iter = {
98         "sorted_key": sorted([("a", 3), ("b", 1), ("c",
99     ↪ 2)], key=lambda x: x[1]),
100        "reversed_list": list(reversed([1, 2, 3])),
101        "enumerate_list": list(enumerate(["x", "y"],
102    ↪ start=1)),
103        "zip_list": list(zip([1, 2, 3], ["a", "b",
104    ↪ "c"])),
105    }
106
107     return {
108         "prime_small_true": prime_small_true,
109         "prime_small_false": prime_small_false,
110         "prime_large_probable": prime_large_probable,
111         "pal_simple_true": pal_simple_true,
112         "pal_simple_false": pal_simple_false,
113         "pal_casefold_true": pal_casefold_true,
114         "pal_alnum_true": pal_alnum_true,
115         "pal_int_true": pal_int_true,
116         "digits_info": digits_info,
117         "str_flags": str_flags,
118         "agg_logic": agg_logic,
119         "conv_num": conv_num,
120         "sorting_iter": sorting_iter,
121     }

```

Linked List

Linked lists are node-based sequences where each node holds a value and references to neighbors. They enable $O(1)$ insertion/removal at ends or at a known node without shifting elements. This is useful when many insertions and deletions are required and random access is not needed. In competitive programming, arrays (lists) are usually faster, but linked lists are important to understand for interview-style problems and for learning pointer-like manipulations in Python using object references.

Singly linked list maintains `head` and optionally `tail` with nodes pointing forward. Doubly linked list maintains both `prev` and `next` pointers for each node, making deletions of arbitrary nodes simpler. Python uses garbage collection, so removing references typically frees nodes without manual memory management.

Complexities (typical):

- Access by index: $O(N)$
- `push_front` / `pop_front`: $O(1)$
- `push_back` / `pop_back`: $O(1)$ if `tail` maintained, else $O(N)$
- find by value: $O(N)$
- insert after known node: $O(1)$, after value: $O(N)$ to find + $O(1)$ insert
- delete by value: $O(N)$ to find + $O(1)$ unlink

Classic patterns:

- Reverse in-place in $O(N)$ using three-pointer technique
- Middle node using fast/slow pointers; returns index $\lfloor N/2 \rfloor$
- Cycle detection using Floyd's algorithm in $O(N)$ and $O(1)$ space

Python specifics:

- Objects are referenced; pointers are simulated by storing object references
- No pointer arithmetic; operations rewire `next/prev` fields
- Avoid negative indexing expectations; traversal is explicit

```

1 class NodeS:
2     def __init__(self, val):
3         self.val = val
4         self.next = None
5
6
7 class SinglyLinkedList:
8     def __init__(self):
9         self.head = None
10        self.tail = None
11        self.size = 0
12

```

```

13
14 def push_front(self, val):
15     node = NodeS(val)
16     node.next = self.head
17     self.head = node
18     if self.size == 0:
19         self.tail = node
20     self.size += 1
21
22 def push_back(self, val):
23     node = NodeS(val)
24     if self.size == 0:
25         self.head = node
26         self.tail = node
27     else:
28         self.tail.next = node
29         self.tail = node
30     self.size += 1
31
32 def pop_front(self):
33     if self.size == 0:
34         return None
35     v = self.head.val
36     self.head = self.head.next
37     self.size -= 1
38     if self.size == 0:
39         self.tail = None
40     return v
41
42 def pop_back(self):
43     if self.size == 0:
44         return None
45     if self.size == 1:
46         return self.pop_front()
47     prev = self.head
48     while prev.next is not self.tail:
49         prev = prev.next
50     v = self.tail.val
51     prev.next = None
52     self.tail = prev
53     self.size -= 1
54     return v
55
56 def find(self, val):
57     cur = self.head
58     while cur:
59         if cur.val == val:
60             return cur
61         cur = cur.next
62     return None
63
64 def insert_after_value(self, target, val):
65     cur = self.head
66     while cur and cur.val != target:
67         cur = cur.next
68     if not cur:
69         return False
70     node = NodeS(val)
71     node.next = cur.next
72     cur.next = node
73     if cur is self.tail:
74         self.tail = node
75     self.size += 1
76     return True
77
78 def delete_value(self, val):
79     if self.size == 0:
80         return False
81     if self.head.val == val:
82         self.pop_front()
83         return True
84     prev = self.head
85     cur = self.head.next
86     while cur and cur.val != val:
87         prev = cur
88         cur = cur.next
89     if not cur:
90         return False
91     prev.next = cur.next

```

```

91         if cur is self.tail:
92             self.tail = prev
93         self.size -= 1
94         return True
95
96     def reverse(self):
97         prev = None
98         cur = self.head
99         self.tail = self.head
100        while cur:
101            nxt = cur.next
102            cur.next = prev
103            prev = cur
104            cur = nxt
105        self.head = prev
106        if self.head is None:
107            self.tail = None
108
109    def middle(self):
110        slow = self.head
111        fast = self.head
112        while fast and fast.next:
113            slow = slow.next
114            fast = fast.next.next
115        return slow
116
117    def has_cycle(self):
118        slow = self.head
119        fast = self.head
120        while fast and fast.next:
121            slow = slow.next
122            fast = fast.next.next
123            if slow is fast:
124                return True
125        return False
126
127    def to_list(self):
128        res = []
129        cur = self.head
130        cnt = 0
131        while cur and cnt < self.size:
132            res.append(cur.val)
133            cur = cur.next
134            cnt += 1
135        return res
136
137    @classmethod
138    def from_list(cls, arr):
139        ll = cls()
140        for v in arr:
141            ll.push_back(v)
142        return ll
143
144
145    class NodeD:
146        def __init__(self, val):
147            self.val = val
148            self.prev = None
149            self.next = None
150
151
152    class DoublyLinkedList:
153        def __init__(self):
154            self.head = None
155            self.tail = None
156            self.size = 0
157
158        def push_front(self, val):
159            node = NodeD(val)
160            node.next = self.head
161            if self.head:
162                self.head.prev = node
163            self.head = node
164            if self.size == 0:
165                self.tail = node
166            self.size += 1
167
168        def push_back(self, val):

```

```

169            node = NodeD(val)
170            if self.size == 0:
171                self.head = node
172                self.tail = node
173            else:
174                node.prev = self.tail
175                self.tail.next = node
176                self.tail = node
177            self.size += 1
178
179        def pop_front(self):
180            if self.size == 0:
181                return None
182            v = self.head.val
183            self.head = self.head.next
184            if self.head:
185                self.head.prev = None
186            self.size -= 1
187            if self.size == 0:
188                self.tail = None
189            return v
190
191        def pop_back(self):
192            if self.size == 0:
193                return None
194            v = self.tail.val
195            self.tail = self.tail.prev
196            if self.tail:
197                self.tail.next = None
198            self.size -= 1
199            if self.size == 0:
200                self.head = None
201            return v
202
203        def find(self, val):
204            cur = self.head
205            while cur:
206                if cur.val == val:
207                    return cur
208                cur = cur.next
209            return None
210
211        def insert_after_value(self, target, val):
212            cur = self.head
213            while cur and cur.val != target:
214                cur = cur.next
215            if not cur:
216                return False
217            node = NodeD(val)
218            node.prev = cur
219            node.next = cur.next
220            if cur.next:
221                cur.next.prev = node
222            cur.next = node
223            if cur is self.tail:
224                self.tail = node
225            self.size += 1
226            return True
227
228        def delete_value(self, val):
229            cur = self.head
230            while cur and cur.val != val:
231                cur = cur.next
232            if not cur:
233                return False
234            if cur.prev:
235                cur.prev.next = cur.next
236            else:
237                self.head = cur.next
238            if cur.next:
239                cur.next.prev = cur.prev
240            else:
241                self.tail = cur.prev
242            self.size -= 1
243            return True
244
245        def to_list(self):
246            res = []

```



```

247     cur = self.head
248     cnt = 0
249     while cur and cnt < self.size:
250         res.append(cur.val)
251         cur = cur.next
252         cnt += 1
253     return res
254
255 @classmethod
256 def from_list(cls, arr):
257     ll = cls()
258     for v in arr:
259         ll.push_back(v)
260     return ll
261
262
263

```

List Operations

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

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

Searching and Counting Operations:

- **index(x)**: Returns the index of the first occurrence of x . Raises **ValueError** if not found. Use **try/except** or check **x in lst** first.
- **count(x)**: Returns the number of occurrences of x in the list.

- **x in lst**: Membership test returning True/False. Both **index()** and **count()** must scan the list, so they're $O(N)$.

Sorting and Reversing:

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

List Initialization Patterns:

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

Common Pitfalls:

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

Performance Tips for Competitive Programming:

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

Memory Considerations:

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

```

1 import copy
2
3
4 def list_operations_examples():
5     """
6     Demonstrates comprehensive Python list operations and
7     common patterns
8     used in competitive programming. This function is
9     primarily for inclusion
10    in the notebook and is called by the stress test to
11    ensure correctness.
12    """
13
14    # === Basic List Creation and Initialization ===
15    empty_list = []
16    filled_list = [1, 2, 3, 4, 5]
17    zeros_list = [0] * 5
18    matrix_2d = [[0] * 3 for _ in range(2)]
19
20    # Common mistake - all rows reference the same object
21    wrong_matrix = [[0] * 3] * 2
22    wrong_matrix[0][0] = 1
23
24    # === Basic Modifications ===
25    # append() - O(1) amortized
26    lst = [1, 2, 3]
27    lst.append(4)
28    lst.append(5)
29
30    # extend() - O(k) where k is length of iterable
31    lst.extend([6, 7, 8])
32
33    # insert() - O(N)
34    lst.insert(0, 0)
35    lst.insert(3, 99)
36
37    # remove() - O(N)
38    lst.remove(99)
39
40    # pop() operations
41    last_element = lst.pop() # O(1)
42    element_at_2 = lst.pop(2) # O(N)
43
44    # clear()
45    temp_list = [1, 2, 3]
46    temp_list.clear()
47
48    # === Searching and Counting ===
49    search_list = [1, 2, 3, 2, 4, 2, 5]
50
51    # index() - O(N)
52    first_two_index = search_list.index(2)
53
54    # count() - O(N)
55    count_of_twos = search_list.count(2)
56
57    # membership test - O(N)
58    has_three = 3 in search_list
59    has_six = 6 in search_list
60
61    # === Sorting and Reversing ===
62    sort_list = [3, 1, 4, 1, 5, 9, 2, 6]
63
64    # sort() - in-place, O(N log N)
65    sort_list.sort()
66    sorted_asc = sort_list.copy()

```

```

65    # sort(reverse=True)
66    sort_list.sort(reverse=True)
67    sorted_desc = sort_list.copy()
68
69    # sorted() - returns new list
70    original = [3, 1, 4, 1, 5]
71    sorted_new = sorted(original)
72
73    # reverse() - in-place, O(N)
74    reverse_list = [1, 2, 3, 4, 5]
75    reverse_list.reverse()
76
77    # slicing for reversal - creates new list
78    reverse_slice = [1, 2, 3, 4, 5][::-1]
79
80    # === Slicing Operations ===
81    slice_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
82
83    # Basic slicing
84    first_half = slice_list[:5]
85    second_half = slice_list[5:]
86    middle = slice_list[2:7]
87
88    # Step slicing
89    even_indices = slice_list[::2]
90    odd_indices = slice_list[1::2]
91    reversed_slice = slice_list[::-1]
92
93    # Negative indexing
94    last_element = slice_list[-1]
95    last_three = slice_list[-3:]
96
97    # === Copying Operations ===
98    original_copy = [1, 2, [3, 4]]
99
100    # Shallow copy methods
101    shallow_copy1 = original_copy[:]
102    shallow_copy2 = original_copy.copy()
103
104    # Deep copy
105    deep_copy = copy.deepcopy(original_copy)
106
107    # Demonstrate shallow vs deep copy
108    original_copy[2][0] = 99
109
110    # === List Comprehensions ===
111    squares = [x * x for x in range(5)]
112    even_squares = [x * x for x in range(10) if x % 2 ==
113    #   0]
114    matrix_flatten = [x for row in [[1, 2], [3, 4]] for x
115    #   in row]
116
117    # === Advanced Operations ===
118    # enumerate() for index-value pairs
119    enumerated = list(enumerate(['a', 'b', 'c']))
120
121    # zip() for pairing lists
122    zipped = list(zip([1, 2, 3], ['a', 'b', 'c']))
123
124    # max, min, sum
125    numbers = [1, 5, 3, 9, 2]
126    max_val = max(numbers)
127    min_val = min(numbers)
128    sum_val = sum(numbers)
129
130    return {
131        # Initialization results
132        "empty_list": empty_list,
133        "filled_list": filled_list,
134        "zeros_list": zeros_list,
135        "matrix_2d": matrix_2d,
136        "wrong_matrix": wrong_matrix,
137
138        # Basic modifications
139        "lst_after_operations": lst,
140        "last_element": last_element,
141        "element_at_2": element_at_2,
142        "temp_list_cleared": temp_list,

```

```

141     # Searching results
142     "first_two_index": first_two_index,
143     "count_of_twos": count_of_twos,
144     "has_three": has_three,
145     "has_six": has_six,
146
147     # Sorting results
148     "sorted_asc": sorted_asc,
149     "sorted_desc": sorted_desc,
150     "sorted_new": sorted_new,
151     "original_unchanged": original,
152     "reverse_list": reverse_list,
153     "reverse_slice": reverse_slice,
154
155     # Slicing results
156     "first_half": first_half,
157     "second_half": second_half,
158     "middle": middle,
159     "even_indices": even_indices,
160     "odd_indices": odd_indices,
161     "reversed_slice": reversed_slice,
162     "last_element_slice": last_element,
163     "last_three": last_three,
164
165     # Copying results
166     "shallow_copy1": shallow_copy1,
167     "shallow_copy2": shallow_copy2,
168     "deep_copy": deep_copy,
169     "original_copy_modified": original_copy,
170
171     # List comprehensions
172     "squares": squares,
173     "even_squares": even_squares,
174     "matrix_flatten": matrix_flatten,
175
176     # Advanced operations
177     "enumerated": enumerated,
178     "zipped": zipped,
179     "max_val": max_val,
180     "min_val": min_val,
181     "sum_val": sum_val,
182
183 }
184

```

Prefix Sums

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

- 1D: $O(N)$ for precomputation, $O(1)$ for each range query.

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

```

1  def build_prefix_sum_1d(arr):
2      """
3      Builds a 1D prefix sum array and returns a query
4      ↪ function.
5
6      Args:
7          arr (list[int]): The input 1D array.
8
9      Returns:
10         function: A function `query(left, right)` that
11             ↪ returns the sum of
12                 the elements in the range [left,
13                 ↪ right-1] in  $O(1)$ .
14
15     """
16     n = len(arr)
17     prefix_sum = [0] * (n + 1)
18     for i in range(n):
19         prefix_sum[i + 1] = prefix_sum[i] + arr[i]
20
21     def query(left, right):
22         """
23         Queries the sum of the range [left, 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
33  def build_prefix_sum_2d(grid):
34      """
35      Builds a 2D prefix sum array and returns a query
36      ↪ function.
37
38      Args:
39          grid (list[list[int]]): The input 2D grid.
40
41      Returns:
42         function: A function `query(r1, c1, r2, c2)` that
43             ↪ returns the sum of
44                 the elements in the rectangle from (r1,
45                 ↪ c1) to (r2-1, c2-1) in  $O(1)$ .
46
47     """
48     if not grid or not grid[0]:
49         return lambda r1, c1, r2, c2: 0
50
51     rows, cols = len(grid), len(grid[0])
52     prefix_sum = [[0] * (cols + 1) for _ in 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, c1) to
66         ↪ (r2-1, c2-1).
67         `r1, c1` are inclusive top-left coordinates.
68         `r2, c2` are exclusive bottom-right coordinates.
69         """
70         if not (0 <= r1 <= r2 <= rows and 0 <= c1 <= c2
71             ↪ <= cols):

```

```

62         return 0
63     return (
64         prefix_sum[r2][c2]
65         - prefix_sum[r1][c2]
66         - prefix_sum[r2][c1]
67         + prefix_sum[r1][c1]
68     )
69
70     return query
71

```

Python Idioms

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

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

Character and Number Conversions:

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

```

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

```

```

8     squares = [x * x for x in range(5)]
9     even_squares = [x * x for x in range(10) if x % 2 ==
10                     ↪ 0]
11
12     # Set and Dictionary Comprehensions
13     unique_squares = {x * x for x in [-1, 1, -2, 2]}
14     square_map = {x: x * x for x in range(5)}
15
16     # Advanced Sorting
17     pairs = [(1, 5), (3, 2), (2, 8)]
18     sorted_by_second = sorted(pairs, key=lambda p: p[1])
19
20     # String Manipulations
21     sentence = "this is a sentence"
22     words = sentence.split()
23     rejoined = "-".join(words)
24     reversed_sentence = sentence[::-1]
25
26     # Character and Number Conversions
27     char_a = "a"
28     ord_a = ord(char_a)
29     chr_97 = chr(97)
30     num_str = "123"
31     num_int = int(num_str)
32     back_to_str = str(num_int)
33
34     # The function can return the values to be checked by
35     ↪ a test script.
36     return {
37         "squares": squares,
38         "even_squares": even_squares,
39         "unique_squares": unique_squares,
40         "square_map": square_map,
41         "sorted_by_second": sorted_by_second,
42         "words": words,
43         "rejoined": rejoined,
44         "reversed_sentence": reversed_sentence,
45         "ord_a": ord_a,
46         "chr_97": chr_97,
47         "num_int": num_int,
48         "back_to_str": back_to_str,
49     }

```

Recursion Backtracking

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

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

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

```

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

```

Set Operations

Sets in Python provide $O(1)$ average-time membership and support expressive algebra via operators. This section covers the core operations used most often in problem solving: union, intersection, difference, and symmetric difference, along with subset/superset/disjointness relations, in-place updates, basic construction patterns, and a note on `frozenset`.

Key operations (A and B are sets):

- Union: $A \mid B$ combines elements from both
- Intersection: $A \& B$ keeps common elements
- Difference: $A - B$ removes elements of B from A

- Symmetric difference: $A \wedge B$ keeps elements in exactly one of A, B

Relations:

- Subset: $A \leq B$ (A is subset of B)
- Superset: $A \geq B$ (A is superset of B)
- Disjointness: `A.isdisjoint(B)`

In-place updates (mutate the left operand):

- $A \mid= B$, $A \&= B$, $A -= B$, $A \wedge= B$ Method forms: `update`, `intersection_update`, `difference_update`, `symmetric_difference_update`.

Construction patterns:

- Literal: `{1, 2, 3}`
- From iterable: `set(iterable)`
- Comprehension: `{f(x) for x in xs if cond(x)}`

`frozenset` is an immutable set, hashable and usable as dict/set keys. It supports the same non-in-place operations as `set`.

Complexity notes:

- Membership and add/remove are average $O(1)$ with good hashing
- Set operations scale with operand sizes; e.g., union is roughly linear in total elements
- Elements must be hashable; unhashable types (like lists) cannot be members

```

1 def set_union(a, b):
2     return a | b
3
4
5 def set_intersection(a, b):
6     return a & b
7
8
9 def set_difference(a, b):
10    return a - b
11
12
13 def set_symmetric_difference(a, b):
14    return a ^ b
15
16
17 def is_subset(a, b):
18    return a <= b
19
20
21 def is_superset(a, b):
22    return a >= b
23
24
25 def is_disjoint(a, b):
26    return a.isdisjoint(b)
27
28
29 def inplace_update_union(a, b):
30    a |= b
31    return a
32

```



```

33
34 def inplace_update_intersection(a, b):
35     a &= b
36     return a
37
38
39 def inplace_update_difference(a, b):
40     a -= b
41     return a
42
43
44 def inplace_update_symmetric_difference(a, b):
45     a ^= b
46     return a
47
48
49 def freeze(iterable):
50     return frozenset(iterable)
51
52
53

```

Stacks And Queues

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

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

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

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

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

```

1 from collections import deque
2
3
4 def stack_and_queue_examples():
5     """

```

```

6 Demonstrates the usage of stacks (with lists) and
7   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  # --- Stack Example (LIFO) ---
13  stack = []
14  stack.append(10) # Stack: [10]
15  stack.append(20) # Stack: [10, 20]
16  stack.append(30) # Stack: [10, 20, 30]
17
18  popped_from_stack = []
19  popped_from_stack.append(stack.pop()) # Returns 30,
20  ↳ Stack: [10, 20]
21  popped_from_stack.append(stack.pop()) # Returns 20,
22  ↳ Stack: [10]
23
24  # --- Queue Example (FIFO) ---
25  queue = deque()
26  queue.append(10) # Queue: deque([10])
27  queue.append(20) # Queue: deque([10, 20])
28  queue.append(30) # Queue: deque([10, 20, 30])
29
30  popped_from_queue = []
31  popped_from_queue.append(queue.popleft()) # Returns
32  ↳ 10, Queue: deque([20, 30])
33  popped_from_queue.append(queue.popleft()) # Returns
34  ↳ 20, Queue: deque([30])
35
36  return {
37      "final_stack": stack,
38      "popped_from_stack": popped_from_stack,
39      "final_queue": list(queue), # Convert to list
40      ↳ for easy comparison
41      "popped_from_queue": popped_from_queue,
42  }

```

Two Pointers

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

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

Sliding Window: This is a specific application of the two-pointers technique. A "window" is a subsegment of the data (e.g., a subarray or substring) represented by the indices `[left, right]`. The `right` pointer expands the window, and the `left` pointer contracts it, typically to maintain a certain property or invariant within the window. This avoids

the re-computation that plagues naive $O(N^2)$ solutions by only adding/removing one element at a time. The example below, "Longest Substring with At Most K Distinct Characters," is a classic sliding window problem. The window `s[left:right+1]` is expanded by incrementing `right`. If the number of distinct characters in the window exceeds `k`, the window is contracted from the left by incrementing `left` until the condition is met again. pointer traverses the data structure at most once. or Σ is the size of the character set, to store the elements in the window.

```
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 that
7     ↪ contains at most k
8     distinct characters.
9
10    Args:
11        s (str): The input string.
12        k (int): The maximum number of distinct characters
13        ↪ allowed.
14
15    Returns:
16        int: The length of the longest valid substring.
17    """
18    if k == 0:
19        return 0
20
21    n = len(s)
22    left = 0
23    max_len = 0
24    char_counts = defaultdict(int)
25
26    for right in range(n):
27        char_counts[s[right]] += 1
28
29        while len(char_counts) > k:
30            char_left = s[left]
31            char_counts[char_left] -= 1
32            if char_counts[char_left] == 0:
33                del char_counts[char_left]
34            left += 1
35
36        max_len = max(max_len, right - left + 1)
37
38    return max_len
```

Chapter 2

Standard Library

Bisect Library

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

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

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

```
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 20s)
15    idx_left_20 = bisect.bisect_left(data, 20)
16    # Find insertion point for 25 (between 20 and 30)
17    idx_left_25 = bisect.bisect_left(data, 25)
18
19    # --- bisect_right ---
20    # Find insertion point for 20 (after existing 20s)
21    idx_right_20 = bisect.bisect_right(data, 20)
22    # Find insertion point for 25 (same as bisect_left)
23    idx_right_25 = bisect.bisect_right(data, 25)
24
25    # --- insort ---
```

```
25 # insort_left inserts at the position found by
26 ↪ bisect_left
27 data_for_insort = [10, 20, 20, 30, 40]
28 bisect.insort_left(data_for_insort, 25)
29
30 return {
31     "idx_left_20": idx_left_20,
32     "idx_left_25": idx_left_25,
33     "idx_right_20": idx_right_20,
34     "idx_right_25": idx_right_25,
35     "list_after_insort": data_for_insort,
36 }
```

Collections Library

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

```
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 ---
```



```

11 q = deque([1, 2, 3])
12 q.append(4)
13 q.appendleft(0)
14 q_pop_left = q.popleft()
15 q_pop_right = q.pop()
16
17 # --- Counter ---
18 data = ["a", "b", "c", "a", "b", "a"]
19 counts = Counter(data)
20 count_of_a = counts["a"]
21 count_of_d = counts["d"]
22
23 # --- defaultdict ---
24 adj = defaultdict(list)
25 edges = [(0, 1), (0, 2), (1, 2)]
26 for u, v in edges:
27     adj[u].append(v)
28     adj[v].append(u)
29
30 # Access a missing key to trigger the default factory
31 missing_key_val = adj[5]
32
33 return {
34     "final_deque": list(q),
35     "q_pop_left": q_pop_left,
36     "q_pop_right": q_pop_right,
37     "counter_a": count_of_a,
38     "counter_d": count_of_d,
39     "adj_list": dict(adj),
40     "adj_list_missing": missing_key_val,
41 }
42

```

Functools Library

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

```

1 import functools
2
3
4 @functools.cache
5 def fibonacci_with_cache(n):
6     """
7     Computes the n-th Fibonacci number using recursion
8     ↳ with memoization.
9     This function is primarily for demonstrating
10    ↳ @functools.cache.

```

```

9 """
10 if n < 2:
11     return n
12 return fibonacci_with_cache(n - 1) +
13 ↳ fibonacci_with_cache(n - 2)

```

Heapq Library

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

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

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

```

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, 8, 5]
31    ↳ (or similar,

```

```

28 # it only guarantees the heap property, not a fully
    ↪ sorted list)
29 heapified_list = list(data_list)
30 smallest_after_heapify = data_list[0]
31
32 return {
33     "smallest_element": smallest_element,
34     "final_heap": min_heap,
35     "popped_elements": popped_elements,
36     "heapified_list": heapified_list,
37     "smallest_after_heapify": smallest_after_heapify,
38 }
39

```

```

19 # All 2-element combinations of elements
20 combs = list(itertools.combinations(elements, 2))
21
22 # --- Cartesian Product ---
23 pool1 = ["x", "y"]
24 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

```

Itertools Library

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

```

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

```

Math Library

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

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

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

```

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    # Greatest Common Divisor
13    gcd_val = math.gcd(54, 24)
14
15    # Ceiling and Floor
16    ceil_val = math.ceil(4.2)

```

```
15     floor_val = math.floor(4.8)
16
17     # Square Roots
18     sqrt_val = math.sqrt(25)
19     isqrt_val = math.isqrt(26)
20
21     # Logarithm
22     log2_val = math.log2(16)
23
24     # Infinity constant
25     infinity = math.inf
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
```

Chapter 3

Contest & Setup

Bash Loops

Bash/Zsh loops are useful for running a solution against many inputs quickly. Below are practical snippets to streamline local testing.

Loop over all input files in a directory:

```
bash for f in tests/*.in; do python3 solve.py < "$f" done
```

Compare outputs to expected .out files:

```
'bash for f in tests/*.in; do base=\f
```

Run a solution N times (useful for randomized generators):

```
bash for i in $(seq 1 100); do python3 solve.py < input.txt done
```

Parallelize runs (careful with I/O ordering):

```
bash seq 1 8 | xargs -P 4 -I{} sh -c 'python3 solve.py < input{}.txt > output{}.txt'
```

Zsh specifics and brace expansion:

```
bash for f in **/*.in; do python3 solve.py < "$f"; done python3 solve.py < input{1..10}.txt
```

Python snippet for multiple test cases in one file:

```
python import sys it = iter(sys.stdin.read().strip().split()) t = int(next(it, 0)) for _ in range(t): # read one test case from iterator pass
```

Notes:

- macOS default shell is zsh; bash is available as well. Quoting paths avoids issues with spaces. Use `time` to measure performance.

messages won't interfere with the actual output and cause a "Wrong Answer" verdict.

- Example: `print(f"DEBUG: Current value of x is {x}", file=sys.stderr)`

1. Test with Edge Cases:

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

1. Assertions:

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

1. Naive Solution Comparison:

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

Debugging Tricks

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

1. Debug Printing to stderr:

- The most common technique is to print variable states at different points in the code.
- Always print to standard error (`sys.stderr`) instead of standard output (`sys.stdout`). The online judge ignores `stderr`, so your debug

1. Rubber Duck Debugging:

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

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

```

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

```

Fast Io

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

1. The `sys.stdin` Object:

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

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

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

- Example: `line = sys.stdin.readline().strip()`

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

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

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

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

1. Iterating over `sys.stdin`:

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

- Example: `for line in sys.stdin: process(line.strip())`

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

- This allows you to consume lines from the iterator one at a time, which can be cleaner than mixing `readline()` with a `for` loop.

- Example: `n = int(next(sys.stdin))`

- Example: `data = [int(x) for x in next(sys.stdin).split()]`

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

```

1 import sys
2
3
4 def get_ints_from_line():
5     """
6     Reads a line of space-separated values and parses them
7     ↪ into a list of integers.
8     This is a common helper function.
9     """
10    return list(map(int, sys.stdin.readline().split()))
11
12 def process_all_lines():
13     """
14     Demonstrates reading all lines at once and processing
15     ↪ them.
16     This example calculates the sum of the first integer
17     ↪ on each line.
18     """
19    lines = sys.stdin.readlines()
20    total = 0

```

```

19     for line in lines:
20         if line.strip():
21             total += int(line.split()[0])
22     return total
23
24
25 def iterate_over_stdin():
26     """
27     Demonstrates processing input by iterating over
28     ↪ sys.stdin, which reads
29     line by line until EOF.
30     """
31     total = 0
32     for line in sys.stdin:
33         if line.strip():
34             total += sum(map(int, line.split()))
35     return total
36
37 def demonstrate_usage_conceptual():
38     """
39     This function is for inclusion in the notebook as a
40     ↪ clear example and
41     is not meant to be executed directly as part of a
42     ↪ test. It shows a common
43     contest pattern: reading a count `N`, followed by `N`
44     ↪ 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 matrix
61         ↪ here.
62         # For demonstration, we just show it was read.
63         # print("Matrix received:", matrix)
64
65     except (IOError, ValueError):
66         # Handle potential empty input or parsing errors
67         ↪ gracefully.
68         pass
69

```

Template

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

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

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

Usage: Place problem-solving logic inside the

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

```

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 from a
17     ↪ 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 a
27     ↪ line."""
28     return input().strip().split()
29
30 def solve():
31     """
32     This is the main function where the solution logic for
33     ↪ 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

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

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

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

```
1 class Node:
2     """Represents a single node in the Binary Search
3     ↳ 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, key) is
20         ↳ 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:
26             return self._search_recursive(node.left, key)
27         return self._search_recursive(node.right, key)
28
29     def insert(self, key):
30         """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, key)
40         elif key > node.key:
41             if node.right is None:
42                 node.right = Node(key)
43             else:
44                 self._insert_recursive(node.right, key)
45
46     def delete(self, key):
47         """Deletes a key from the BST."""
48         self.root = self._delete_recursive(self.root,
49         ↳ key)
50
51     def _delete_recursive(self, node, key):
52         if node is None:
53             return node
54
55         if key < node.key:
56             node.left = self._delete_recursive(node.left,
57             ↳ key)
58         elif key > node.key:
59             node.right =
60             ↳ self._delete_recursive(node.right, key)
61         else:
62             if node.left is None:
63                 return node.right
64             elif node.right is None:
65                 return node.left
66
67             # Node with two children: Get the in-order
68             ↳ successor (smallest in the right subtree)
69             temp = self._min_value_node(node.right)
70             node.key = temp.key
71             node.right =
72             ↳ self._delete_recursive(node.right,
73             ↳ temp.key)
74             return node
75
76     def _min_value_node(self, node):
77         current = node
78         while current.left is not None:
79             current = current.left
80         return current
```

Fenwick Tree

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

in logarithmic time:

1. `add(idx, delta)`: Adds `delta` to the element at index `idx`.
2. `query(right)`: Computes the sum of the elements in the range `[0, right]`.

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

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

```

1 class FenwickTree:
2     """
3     A class that implements a 1D Fenwick Tree (Binary
4     ↳ Indexed Tree).
5     This implementation uses 0-based indexing for its
6     ↳ public methods.
7     """
8
9     def __init__(self, size):
10         """
11         Initializes the Fenwick Tree for an array of a
12         ↳ given size.
13         All elements are initially zero.
14
15         Args:
16             size (int): The number of elements the tree
17             ↳ will support.
18         """
19         self.tree = [0] * size
20
21     def add(self, idx, delta):
22         """
23         Adds a delta value to the element at a specific
24         ↳ index.
25         This operation updates all prefix sums that
26         ↳ include this index.
27
28         Args:
29             idx (int): The 0-based index of the element to
30             ↳ update.
31             delta (int): The value to add to the element
32             ↳ at `idx`.
33         """
34         while idx < len(self.tree):
35             self.tree[idx] += delta
36             idx |= idx + 1
37
38     def query(self, right):
39         """
40         Computes the prefix sum of elements up to (but not
41         ↳ including) `right`.
42         This is the sum of the range [0, right-1].
43
44         Args:
45             right (int): The 0-based exclusive upper bound
46             ↳ of the query range.
47
48         Returns:
49             int: The sum of elements in the prefix `[0,
50             ↳ right-1]`.

```

```

40
41     """
42     idx = right - 1
43     total_sum = 0
44     while idx >= 0:
45         total_sum += self.tree[idx]
46         idx = (idx & (idx + 1)) - 1
47     return total_sum
48
49     def query_range(self, left, right):
50         """
51         Computes the sum of elements in the range [left,
52         ↳ right-1].
53
54         Args:
55             left (int): The 0-based inclusive lower bound
56             ↳ of the query range.
57             right (int): The 0-based exclusive upper bound
58             ↳ of the query range.
59
60         Returns:
61             int: The sum of elements in the specified
62             ↳ range.
63         """
64         if left >= right:
65             return 0
66         return self.query(right) - self.query(left)
67

```

Fenwick Tree 2D

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

1. `add(r, c, delta)`: Adds `delta` to the element at grid cell `(r, c)`.
2. `query(r, c)`: Computes the sum of the rectangle from `(0, 0)` to `(r-1, c-1)`.

A 2D Fenwick Tree can be conceptualized as a Fenwick Tree where each element is itself another Fenwick Tree. The `add` and `query` operations therefore involve traversing the tree structure in both dimensions, resulting in a time complexity that is the product of the logarithmic complexities of each dimension. The `query_range` method uses the principle of inclusion-exclusion on the prefix rectangle sums to calculate the sum of any arbitrary 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)`.

```

1 class FenwickTree2D:
2     """
3     A class that implements a 2D Fenwick Tree using
4     ↳ 0-based indexing.
5     """
6
7     def __init__(self, rows, cols):
8         """
9         Initializes the 2D Fenwick Tree for a grid of a
10        ↳ given size.
11        All elements are initially zero.

```



```

11     Args:
12         rows (int): The number of rows in the grid.
13         cols (int): The number of columns in the
14             ↪ grid.
15     """
16     self.rows = rows
17     self.cols = cols
18     self.tree = [[0] * cols for _ in range(rows)]
19
20 def add(self, r, c, delta):
21     """
22     Adds a delta value to the element at grid cell (r,
23     ↪ c).
24
25     Args:
26         r (int): The 0-based row index of the element
27             ↪ to update.
28         c (int): The 0-based column index of the
29             ↪ element to update.
30         delta (int): The value to add.
31     """
32     i = r
33     while i < self.rows:
34         j = c
35         while j < self.cols:
36             self.tree[i][j] += delta
37             j |= j + 1
38         i |= i + 1
39
40 def query(self, r, c):
41     """
42     Computes the prefix sum of the rectangle from (0,
43     ↪ 0) to (r-1, c-1).
44
45     Args:
46         r (int): The 0-based exclusive row bound of
47             ↪ the query rectangle.
48         c (int): The 0-based exclusive column bound of
49             ↪ the query rectangle.
50
51     Returns:
52         int: The sum of the elements in the rectangle
53             ↪ [0..r-1, 0..c-1].
54     """
55     total_sum = 0
56     i = r - 1
57     while i >= 0:
58         j = c - 1
59         while j >= 0:
60             total_sum += self.tree[i][j]
61             j = (j & (j + 1)) - 1
62         i = (i & (i + 1)) - 1
63     return total_sum
64
65 def query_range(self, r1, c1, r2, c2):
66     """
67     Computes the sum of the rectangle from (r1, c1) to
68     ↪ (r2-1, c2-1).
69
70     Args:
71         r1, c1 (int): The 0-based inclusive top-left
72             ↪ coordinates.
73         r2, c2 (int): The 0-based exclusive
74             ↪ bottom-right coordinates.
75
76     Returns:
77         int: The sum of elements in the specified
78             ↪ rectangular range.
79     """
80     if r1 >= r2 or c1 >= c2:
81         return 0
82
83     total = self.query(r2, c2)
84     total -= self.query(r1, c2)
85     total -= self.query(r2, c1)
86     total += self.query(r1, c1)
87     return total

```

Hash Map Custom

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

```

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

```

```

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}: {hash(custom_key)}")
56
57     # Verifying that it works
58     assert custom_key in custom_dict
59     assert CustomHash(key) in custom_dict
60     assert CustomHash((0, 0)) not in custom_dict

```

Line Container

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

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

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

```

1 class LineContainer:
2     """
3     A data structure for the Convex Hull Trick, optimized
4     ↪ 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 neighbors
15        ↪ l1 and l3.
16        l2 is redundant if the intersection of l1 and l3
17        ↪ is to the left of
18        the intersection of l1 and l2.
19        Intersection of (m1, c1) and (m2, c2) is x = (c2 -
20        ↪ c1) / (m1 - m2).
21        We check if (c3-c1)/(m1-m3) <= (c2-c1)/(m1-m2).
22        To avoid floating point, we use
23        ↪ cross-multiplication.
24        Since slopes are decreasing, m1 > m2 > m3, so
25        ↪ (m1-m3) and (m1-m2) are positive.
26        The inequality becomes (c3-c1)*(m1-m2) <=
27        ↪ (c2-c1)*(m1-m3).
28        """
29        m1, c1 = l1
30        m2, c2 = l2
31        m3, c3 = l3
32        # Note the direction of inequality might change
33        ↪ based on max/min query
34        # and increasing/decreasing slopes. This is for
35        ↪ min query, decr. slopes.
36        return (c3 - c1) * (m1 - m2) <= (c2 - c1) * (m1 -
37        ↪ m3)
38
39    def add(self, m, c):
40        """
41        Adds a new line y = mx + c to the container.
42        Assumes that m is less than the slope of any
43        ↪ previously added line.
44        """
45        new_line = (m, c)
46        while len(self.hull) >= 2 and self._is_redundant(
47            self.hull[-2], self.hull[-1], new_line
48        ):
49            self.hull.pop()
50        self.hull.append(new_line)
51
52    def query(self, x):
53        """
54        Finds the minimum value of y = mx + c for a given
55        ↪ x among all lines.
56        """
57        if not self.hull:
58            return float("inf")
59
60        # Binary search for the optimal line.
61        # The function `f(i) = m_i * x + c_i` is not
62        ↪ monotonic, but the
63        # optimal line index is. Specifically, the
64        ↪ function `f(i+1) - f(i)`
65        # is monotonic. We are looking for the point where
66        ↪ the function
67        # transitions from decreasing to increasing.
68        low, high = 0, len(self.hull) - 1
69        res_idx = 0
70
71        while low <= high:
72            mid = (low + high) // 2
73            # Check if mid is better than mid+1
74            if mid + 1 < len(self.hull):
75                val_mid = self.hull[mid][0] * x +
76                ↪ self.hull[mid][1]
77                val_next = self.hull[mid + 1][0] * x +
78                ↪ self.hull[mid + 1][1]
79                if val_mid > val_next:
80                    low = mid + 1
81            else:
82                res_idx = mid
83                high = mid - 1
84        else:
85            res_idx = mid

```

```

69         high = mid - 1
70
71     m, c = self.hull[res_idx]
72     return m * x + c
73

```

Ordered Set

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

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

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

```

1 import random
2
3
4 class Node:
5     """Represents a single node in the Ordered Set's
6     ↳ 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
19 def _update_size(t):
20     if t:
21         t.size = 1 + _get_size(t.left) +
22         ↳ _get_size(t.right)
23
24 def _split(t, key):
25     """
26     Splits the tree `t` into two trees: one with keys <
27     ↳ `key` (l)
28     and one with keys >= `key` (r).
29     """
30     if not t:
31         return None, None
32     if t.key < key:

```

```

33         l, r = _split(t.right, key)
34         t.right = l
35         _update_size(t)
36         return t, r
37     else:
38         l, r = _split(t.left, key)
39         t.left = r
40         _update_size(t)
41         return l, t
42
43 def _merge(t1, t2):
44     """Merges two trees `t1` and `t2`, assuming keys in
45     ↳ `t1` < keys in `t2`."""
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         _update_size(t1)
53         return t1
54     else:
55         t2.left = _merge(t1, t2.left)
56         _update_size(t2)
57         return t2
58
59 class OrderedSet:
60     """
61     An Ordered Set implementation using a Treap.
62     Supports finding the k-th element and the rank of an
63     ↳ element.
64     """
65
66     def __init__(self):
67         self.root = None
68
69     def search(self, key):
70         node = self.root
71         while node:
72             if node.key == key:
73                 return True
74             node = node.left if key < node.key else
75             ↳ node.right
76         return False
77
78     def insert(self, key):
79         if self.search(key):
80             return
81         new_node = Node(key)
82         l, r = _split(self.root, key)
83         self.root = _merge(_merge(l, new_node), r)
84
85     def delete(self, key):
86         if not self.search(key):
87             return
88         l, r = _split(self.root, key)
89         _, r_prime = _split(r, key + 1)
90         self.root = _merge(l, r_prime)
91
92     def find_by_order(self, k):
93         """Finds the k-th smallest element
94         ↳ (0-indexed)."""
95         node = self.root
96         while node:
97             left_size = _get_size(node.left)
98             if left_size == k:
99                 return node.key
100            elif k < left_size:
101                node = node.left
102            else:
103                k -= left_size + 1
104                node = node.right
105        return None
106
107     def order_of_key(self, key):

```

```

105     """Finds the number of elements strictly smaller
106     ↪ than key."""
107     count = 0
108     node = self.root
109     while node:
110         if key == node.key:
111             count += _get_size(node.left)
112             break
113         elif key < node.key:
114             node = node.left
115         else:
116             count += _get_size(node.left) + 1
117             node = node.right
118     return count
119
120 def __len__(self):
121     return _get_size(self.root)

```

Segment Tree Lazy

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

```

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] + self.tree[2 * v + 1]
17
18     def _push(self, v, tl, tr):
19         if self.lazy[v] != 0:
20             return

```

```

22     range_len = tr - tl + 1
23     self.tree[v] += self.lazy[v] * range_len
24
25     if tl != tr:
26         self.lazy[2 * v] += self.lazy[v]
27         self.lazy[2 * v + 1] += self.lazy[v]
28
29     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, tm),
41             ↪ addval)
42         self._update(2 * v + 1, tm + 1, tr, max(l, tm
43             ↪ + 1), r, addval)
44
45     # After children are updated, update self
46     ↪ 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] + self.tree[2 * v + 1]
50
51 def _query(self, v, tl, tr, l, r):
52     if l > r:
53         return 0
54     self._push(v, tl, tr)
55     if l == tl and r == tr:
56         return self.tree[v]
57
58     tm = (tl + tr) // 2
59     left_sum = self._query(2 * v, tl, tm, l, min(r,
60         ↪ tm))
61     right_sum = self._query(2 * v + 1, tm + 1, tr,
62         ↪ max(l, tm + 1), r)
63     return left_sum + right_sum
64
65 def update(self, l, r, addval):
66     # Updates range [l, r] (inclusive)
67     if l > r:
68         return
69     self._update(1, 0, self.n - 1, l, r, addval)
70
71 def query(self, l, r):
72     # Queries range [l, r] (inclusive)
73     if l > r:
74         return 0
75     return self._query(1, 0, self.n - 1, l, r)

```

Sparse Table

Implements a Sparse Table for fast Range Minimum Queries (RMQ). This data structure is ideal for answering range queries on a static array for idempotent functions like min, max, or gcd. The core idea is to precompute the answers for all ranges that have a length that is a power of two. The table `st[k][i]` stores the minimum value in the range $[i, i + 2^k - 1]$. This precomputation takes $O(N \log N)$ time. Once the table is built, a query for any arbitrary range $[l, r]$ can be answered in $O(1)$ time. This is achieved by finding the largest power of two, 2^k , that is less than or equal

to the range length $r - l + 1$. The query then returns the minimum of two overlapping ranges: $[l, l + 2^k - 1]$ and $[r - 2^k + 1, r]$. Because \min is an idempotent function, the overlap does not affect the result. This implementation is for range minimum, but can be easily adapted for range maximum by changing \min to \max .

```

1 import math
2
3
4 class SparseTable:
5     """
6     A class that implements a Sparse Table for efficient
7     ↪ Range Minimum Queries.
8     This implementation assumes 0-based indexing for the
9     ↪ input array and queries.
10    """
11
12    def __init__(self, arr):
13        """
14        Initializes the Sparse Table from an input array.
15
16        Args:
17            arr (list[int]): The static list of numbers to
18            ↪ be queried.
19        """
20        self.n = len(arr)
21        if self.n == 0:
22            return
23
24        self.max_log = self.n.bit_length() - 1
25        self.st = [[0] * self.n for _ in
26            ↪ range(self.max_log + 1)]
27        self.st[0] = list(arr)
28
29        for k in range(1, self.max_log + 1):
30            for i in range(self.n - (1 << k) + 1):
31                self.st[k][i] = min(
32                    self.st[k - 1][i], self.st[k - 1][i +
33                    ↪ (1 << (k - 1))]
34                )
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 >> 1] +
39            ↪ 1
40
41    def query(self, l, r):
42        """
43        Queries the minimum value in the inclusive range
44        ↪ [l, r].
45
46        Args:
47            l (int): The 0-based inclusive starting index
48            ↪ of the range.
49            r (int): The 0-based inclusive ending index of
50            ↪ the range.
51
52        Returns:
53            int: The minimum value in the range [l, r].
54            ↪ Returns infinity
55            if the table is empty or the range is
56            ↪ invalid.
57        """
58        if self.n == 0 or l > r:
59            return float("inf")
60
61        length = r - l + 1
62        k = self.log_table[length]
63        return min(self.st[k][l], self.st[k][r - (1 << k)
64            ↪ + 1])

```

Treap

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

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

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

```

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 based on
20        ↪ `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 right_tree
24        ↪ 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

```



```

35
36 def _merge(t1, t2):
37     """
38     Merges two trees `t1` and `t2`.
39     Assumes all keys in `t1` are less than all keys in
40     ↪ `t2`.
41     The merge is performed based on node priorities to
42     ↪ maintain the heap property.
43     """
44     if not t1:
45         return t2
46     if not t2:
47         return t1
48     if t1.priority > t2.priority:
49         t1.right = _merge(t1.right, t2)
50         return t1
51     else:
52         t2.left = _merge(t1, t2.left)
53         return t2
54
55 class Treap:
56     """
57     The Treap class providing a public API for balanced
58     ↪ BST operations.
59     """
60
61     def __init__(self):
62         """Initializes an empty Treap."""
63         self.root = None
64
65     def search(self, key):
66         """
67         Searches for a key in the Treap.
68         Returns True if the key is found, otherwise
69         ↪ False.
70         """
71         node = self.root
72         while node:
73             if node.key == key:
74                 return True
75             elif key < node.key:
76                 node = node.left
77             else:
78                 node = node.right
79         return False
80
81     def insert(self, key):
82         """
83         Inserts a key into the Treap. If the key already
84         ↪ exists, the tree is unchanged.
85         """
86         if self.search(key):
87             return # Don't insert duplicates
88
89         new_node = Node(key)
90         l, r = _split(self.root, key)
91         # l has keys < key, r has keys >= key.
92         # Merge new_node with r first, then merge l with
93         ↪ the result.
94         self.root = _merge(l, _merge(new_node, r))
95
96     def delete(self, key):
97         """
98         Deletes a key from the Treap. If the key is not
99         ↪ found, the tree is unchanged.
100        """
101         if not self.search(key):
102             return
103
104         # Split to isolate the node to be deleted.
105         l, r = _split(self.root, key) # l has keys <
106         ↪ key, r has keys >= key
107         _, r_prime = _split(r, key + 1) # r_prime has
108         ↪ keys > key
109
110         # Merge the remaining parts back together.
111         self.root = _merge(l, r_prime)

```

104

Union Find

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

1. Path Compression: During a find operation, it makes every node on the path from the query node to the root point directly to the root. This dramatically flattens the tree structure.
2. Union by Size: During a union operation, it always attaches the root of the smaller tree to the root of the larger tree. This helps in keeping the trees shallow, which speeds up future find operations.

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

```

1 class UnionFind:
2     """
3     A class that implements the Union-Find data structure
4     ↪ 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 containing
28            ↪ 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 j.
38        Applies union by size.
39        Args:
40            i (int): An element in the first set.
41            j (int): An element in the second set.

```

```
38     Returns:
39         bool: True if the sets were merged, False if
40         ↔ they were already in the same set.
41         """
42     root_i = self.find(i)
43     root_j = self.find(j)
44     if root_i != root_j:
45         if self.size[root_i] < self.size[root_j]:
46             root_i, root_j = root_j, root_i
47         self.parent[root_j] = root_i
48         self.size[root_i] += self.size[root_j]
49         return True
50     return False
```

Chapter 5

Graph Algorithms

Bellman Ford

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

```
1 def bellman_ford(edges, start_node, n):
2     """
3     Finds shortest paths from a start node, handling
4     ↪ negative weights and
5     detecting negative cycles.
6
7     Args:
8         edges (list[tuple[int, int, int]]): A list of all
9         ↪ edges in the graph,
10        where each tuple is (u, v, weight) for an edge
11        ↪ u -> v.
12        start_node (int): The node from which to start the
13        ↪ search.
14        n (int): The total number of nodes in the graph.
15
16     Returns:
17         tuple[list[float], bool]: A tuple containing:
18         - A list of shortest distances. `float('inf')`
19         ↪ for unreachable nodes.
20         - A boolean that is True if a negative cycle
21         ↪ is detected, False otherwise.
22
23     """
24     if not (0 <= start_node < n):
25         return [float("inf")] * n, False
26
27     dist = [float("inf")] * n
28     dist[start_node] = 0
29
30     for i in range(n - 1):
31         updated = False
32         for u, v, w in edges:
33             if dist[u] != float("inf") and dist[u] + w <
34                 dist[v]:
35                 dist[v] = dist[u] + w
36                 updated = True
37         if not updated:
38             break
```

```
32 for u, v, w in edges:
33     if dist[u] != float("inf") and dist[u] + w <
34         ↪ dist[v]:
35         return dist, True
36
37 return dist, False
```

Bipartite Matching

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

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

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

```
1 def bipartite_matching(adj, n1, n2):
2     """
3     Finds the maximum matching in a bipartite graph.
4
```



```

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

```

Dijkstra

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

1. Initialize a distances array with infinity for all vertices except the source, which is set to 0.
2. Initialize a priority queue and add the source vertex with a distance of 0.
3. While the priority queue is not empty, extract the vertex u with the smallest distance.
4. If u has already been processed with a shorter path, skip it.
5. For each neighbor v of u , calculate the distance through u . If this new path is shorter than the known distance to v , update the distance and add v to the priority queue with its new, shorter distance.

This implementation uses Python's `heapq` module as a min-priority queue. The graph is represented

by an adjacency list where each entry is a tuple (neighbor, weight). of edges. The log factor comes from the priority queue operations. priority queue.

```

1  import heapq
2
3
4  def dijkstra(adj, start_node, n):
5      """
6      Finds the shortest paths from a start node to all
7      ↳ other nodes in a graph.
8
9      Args:
10         adj (list[list[tuple[int, int]]]): The adjacency
11         ↳ list representation of
12         ↳ the graph. adj[u] contains tuples (v, weight)
13         ↳ for edges u -> v.
14         start_node (int): The node from which to start the
15         ↳ search.
16         n (int): The total number of nodes in the graph.
17
18     Returns:
19         list[float]: A list of shortest distances from the
20         ↳ start_node to each
21         ↳ node. `float('inf')` indicates an
22         ↳ unreachable node.
23     """
24     if not (0 <= start_node < n):
25         return [float("inf")] * n
26
27     dist = [float("inf")] * n
28     dist[start_node] = 0
29     pq = [(0, start_node)]
30
31     while pq:
32         d, u = heapq.heappop(pq)
33
34         if d > dist[u]:
35             continue
36
37         for v, weight in adj[u]:
38             if dist[u] + weight < dist[v]:
39                 dist[v] = dist[u] + weight
40                 heapq.heappush(pq, (dist[v], v))
41
42     return dist

```

Dinic

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

1. Build a "level graph" using a Breadth-First Search (BFS) from the source s on the residual graph. The level of a vertex is its shortest distance from s . The level graph only contains edges (u, v) where $\text{level}[v] == \text{level}[u] + 1$. 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. such as $O(E\sqrt{V})$ for bipartite matching and $O(E \min(V^{2/3}, E^{1/2}))$ for unit-capacity networks.

```

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, len(self.graph[u]) -
18                               ↪ 1])
19
20    def _bfs(self, s, t):
21        self.level = [-1] * self.n
22        self.level[s] = 0
23        q = deque([s])
24        while q:
25            u = q.popleft()
26            for i in range(len(self.graph[u])):
27                v, cap, rev = self.graph[u][i]
28                if cap > 0 and self.level[v] < 0:
29                    self.level[v] = self.level[u] + 1
30                    q.append(v)
31        return self.level[t] != -1
32
33    def _dfs(self, u, t, pushed):
34        if pushed == 0:
35            return 0
36        if u == t:
37            return pushed
38
39        while self.ptr[u] < len(self.graph[u]):
40            edge_idx = self.ptr[u]
41            v, cap, rev_idx = self.graph[u][edge_idx]
42
43            if self.level[v] != self.level[u] + 1 or cap
44            ↪ == 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

```

```

61        pushed = self._dfs(s, t, self.inf)
62        while pushed > 0:
63            total_flow += pushed
64            pushed = self._dfs(s, t, self.inf)
65        return total_flow
66

```

Euler Path

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

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

For a directed graph:

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

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

```

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 graph.
14         directed (bool): True if the graph is directed,
15            ↪ False otherwise.
16
17     Returns:

```

```

15     list[int] | None: A list of nodes representing the
16     ↪ Eulerian path,
17     or None if no such path exists.
18 """
19 if n == 0:
20     return []
21
22 num_edges = 0
23 if directed:
24     in_degree = [0] * n
25     out_degree = [0] * n
26     for u in range(n):
27         out_degree[u] = len(adj[u])
28         num_edges += len(adj[u])
29         for v in adj[u]:
30             in_degree[v] += 1
31
32 start_node, end_node_count = -1, 0
33 for i in range(n):
34     if out_degree[i] - in_degree[i] == 1:
35         if start_node != -1:
36             return None
37         start_node = i
38     elif in_degree[i] - out_degree[i] == 1:
39         end_node_count += 1
40         if end_node_count > 1:
41             return None
42     elif in_degree[i] != out_degree[i]:
43         return None
44
45 if start_node == -1:
46     for i in range(n):
47         if out_degree[i] > 0:
48             start_node = i
49             break
50 if start_node == -1:
51     return [0] if n > 0 else []
52
53 else:
54     degree = [0] * n
55     for u in range(n):
56         degree[u] = len(adj[u])
57         num_edges += len(adj[u])
58     num_edges //= 2
59
60 odd_degree_nodes = [i for i, d in
61 ↪ enumerate(degree) if d % 2 != 0]
62 if len(odd_degree_nodes) > 2:
63     return None
64
65 start_node = -1
66 if odd_degree_nodes:
67     start_node = odd_degree_nodes[0]
68 else:
69     for i in range(n):
70         if degree[i] > 0:
71             start_node = i
72             break
73 if start_node == -1:
74     return [0] if n > 0 else []
75
76 adj_counts = [Counter(neighbors) for neighbors in
77 ↪ adj]
78 path = []
79 stack = [start_node]
80
81 while stack:
82     u = stack[-1]
83     if adj_counts[u]:
84         v = next(iter(adj_counts[u]))
85         adj_counts[u][v] -= 1
86         if adj_counts[u][v] == 0:
87             del adj_counts[u][v]
88
89     if not directed:
90         adj_counts[v][u] -= 1
91         if adj_counts[v][u] == 0:
92             del adj_counts[v][u]

```

```

90
91     stack.append(v)
92     else:
93         path.append(stack.pop())
94
95     path.reverse()
96
97     if len(path) == num_edges + 1:
98         return path
99     else:
100         return None
101

```

Floyd Warshall

Implements the Floyd-Warshall algorithm for finding all-pairs shortest paths in a weighted directed graph. This algorithm can handle graphs with negative edge weights. The algorithm is based on a dynamic programming approach. It iteratively considers each vertex k and updates the shortest path between all pairs of vertices (i, j) to see if a path through k is shorter. The core recurrence is: $\text{dist}(i, j) = \min(\text{dist}(i, j), \text{dist}(i, k) + \text{dist}(k, j))$. 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. dominate the run-time.

```

1 def floyd_warshall(edges, n):
2     """
3     Finds all-pairs shortest paths in a graph using the
4     ↪ Floyd-Warshall algorithm.
5
6     Args:
7         edges (list[tuple[int, int, int]]): A list of all
8         ↪ edges in the graph,
9         where each tuple is (u, v, weight) for an edge
10        ↪ u -> v.
11        n (int): The total number of nodes in the graph.
12
13     Returns:
14        tuple[list[list[float]], bool]: A tuple
15        ↪ containing:
16        - A 2D list of shortest distances.
17        ↪ `dist[i][j]` is the shortest
18        ↪ distance from node `i` to node `j`.
19        ↪ `float('inf')` for unreachable pairs.
20        - A boolean that is True if a negative cycle
21        ↪ is detected, False otherwise.
22
23     """
24     if n == 0:
25         return [], False
26
27     dist = [[float("inf")] * n for _ in range(n)]
28
29     for i in range(n):
30         dist[i][i] = 0
31
32     for u, v, w in edges:
33         dist[u][v] = min(dist[u][v], w)

```

```

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

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

1. Precomputation:

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

1. Querying for LCA(`u`, `v`):

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

```

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 range(n)]
8         self._dfs(root, -1, 0)

```

```

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

```

Prim Kruskal

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

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

priority queue.

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

```

1 import heapq
2 import sys
3 import os
4
5 # Add content directory to path to import the solution
6 sys.path.append(
7     os.path.join(os.path.dirname(__file__),
8         ↪ ".../content/data_structures")
9 )
10 from union_find import UnionFind
11
12 def kruskal(edges, n):
13     """
14     Finds the MST of a graph using Kruskal's algorithm.
15
16     Args:
17         edges (list[tuple[int, int, int]]): A list of all
18         ↪ edges in the graph,
19         where each tuple is (u, v, weight).
20         n (int): The total number of nodes in the graph.
21
22     Returns:
23         tuple[int, list[tuple[int, int, int]]]: A tuple
24         ↪ containing:
25         - The total weight of the MST.
26         - A list of edges (u, v, weight) that form the
27         ↪ MST.
28         Returns (inf, []) if the graph is not
29         ↪ connected and cannot form a single MST.
30     """
31     if n == 0:
32         return 0, []
33
34     sorted_edges = sorted([(w, u, v) for u, v, w in
35         ↪ edges])
36     uf = UnionFind(n)
37     mst_weight = 0
38     mst_edges = []
39
40     for weight, u, v in sorted_edges:
41         if uf.union(u, v):
42             mst_weight += weight
43             mst_edges.append((u, v, weight))
44             if len(mst_edges) == n - 1:
45                 break
46
47     if len(mst_edges) < n - 1:
48         # This indicates the graph is not connected.
49         # The result is a minimum spanning forest.
50         pass
51
52     return mst_weight, mst_edges
53
54 def prim(adj, n, start_node=0):
55     """
56     Finds the MST of a graph using Prim's algorithm.
57
58     Args:
59         adj (list[list[tuple[int, int]]]): The adjacency
60         ↪ list representation of
61         the graph. adj[u] contains tuples (v, weight)
62         ↪ for edges u -> v.
63         n (int): The total number of nodes in the graph.
64         start_node (int): The node to start building the
65         ↪ MST from.
66
67     Returns:
68         tuple[int, list[tuple[int, int, int]]]: A tuple
69         ↪ containing:
70         - The total weight of the MST.
71         - A list of edges (u, v, weight) that form the
72         ↪ MST.
73         Returns (inf, []) if the graph is not
74         ↪ connected.
75     """
76     if n == 0:
77         return 0, []
78
79     visited = [False] * n
80     pq = [(0, start_node, -1)] # (weight, current_node,
81         ↪ 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 connected.
104         return float("inf"), []
105
106     return mst_weight, mst_edges

```

```

61     tuple[int, list[tuple[int, int, int]]]: A tuple
62     ↪ containing:
63     - The total weight of the MST.
64     - A list of edges (u, v, weight) that form the
65     ↪ MST.
66     Returns (inf, []) if the graph is not
67     ↪ connected.
68
69     """
70     if n == 0:
71         return 0, []
72     if not (0 <= start_node < n):
73         return float("inf"), []
74
75     visited = [False] * n
76     pq = [(0, start_node, -1)] # (weight, current_node,
77         ↪ previous_node)
78     mst_weight = 0
79     mst_edges = []
80     edges_count = 0
81
82     while pq and edges_count < n:
83         weight, u, prev = heapq.heappop(pq)
84
85         if visited[u]:
86             continue
87
88         visited[u] = True
89         mst_weight += weight
90         if prev != -1:
91             mst_edges.append((prev, u, weight))
92         edges_count += 1
93
94         for v, w in adj[u]:
95             if not visited[v]:
96                 heapq.heappush(pq, (w, v, u))
97
98     if edges_count < n:
99         # This indicates the graph is not connected.
100         return float("inf"), []
101
102     return mst_weight, mst_edges

```

Sc

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. $\text{disc}[u]$: The discovery time of u , which is the time (a counter) when u is first visited.
2. $\text{low}[u]$: 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 ($\text{disc}[u] == \text{low}[u]$). When such a node is found, all nodes in its SCC are on the top of the stack and can be popped off until u is reached. These popped nodes form one complete

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

```

1 def find_sccs(adj, n):
2     """
3     Finds all Strongly Connected Components of a directed
4     ↪ 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 graph.
10
11     Returns:
12         list[list[int]]: A list of lists, where each inner
13         ↪ list contains the
14                             nodes of a single Strongly
15                             ↪ Connected Component.
16
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
26     sccs = []
27
28     def tarjan_dfs(u):
29         nonlocal time
30         disc[u] = low[u] = time
31         time += 1
32         stack.append(u)
33         on_stack[u] = True
34
35         for v in adj[u]:
36             if disc[v] == -1:
37                 tarjan_dfs(v)
38             low[u] = min(low[u], low[v])
39             elif on_stack[v]:
40                 low[u] = min(low[u], disc[v])
41
42         if low[u] == disc[u]:
43             component = []
44             while True:
45                 node = stack.pop()
46                 on_stack[node] = False
47                 component.append(node)
48                 if node == u:
49                     break
50             sccs.append(component)
51
52     for i in range(n):
53         if disc[i] == -1:
54             tarjan_dfs(i)
55
56     return sccs

```

Topological Sort

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

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

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

```

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

```

Traversal

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

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

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

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

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

```

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 start the
12        ↪ traversal.
13        n (int): The total number of nodes in the graph.

```

```

13 Returns:
14     list[int]: A list of nodes in the order they were
15     ↪ visited.
16 """
17 if not (0 <= start_node < n):
18     return []
19
20 q = deque([start_node])
21 visited = [False] * n
22 visited[start_node] = True
23 traversal_order = []
24
25 while q:
26     u = q.popleft()
27     traversal_order.append(u)
28     for v in adj[u]:
29         if not visited[v]:
30             visited[v] = True
31             q.append(v)
32     return traversal_order
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 start the
42        ↪ traversal.
43        n (int): The total number of nodes in the graph.
44
45     Returns:
46         list[int]: A list of nodes in the order they were
47         ↪ visited.
48 """
49 if not (0 <= start_node < n):
50     return []
51
52 stack = [start_node]
53 visited = [False] * n
54 # Mark as visited when pushed to stack to avoid
55 ↪ re-adding
56 visited[start_node] = True
57 traversal_order = []
58
59 # This loop produces a traversal order different from
60 ↪ the recursive one.
61 # To get a more standard pre-order traversal
62 ↪ iteratively, we need a slight change.
63
64 # Reset for a more standard iterative DFS traversal
65 ↪ order
66 visited = [False] * n
67 stack = [start_node]
68
69 while stack:
70     u = stack.pop()
71
72     if not visited[u]:
73         visited[u] = True
74         traversal_order.append(u)
75
76     # Add neighbors to the stack in reverse order
77     ↪ to process them in lexicographical order
78     for v in reversed(adj[u]):
79         if not visited[v]:
80             stack.append(v)
81
82 return traversal_order

```

Two Sat

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

1. Create an "implication graph" with $2N$ vertices for N variables. For each variable x_i , there are two vertices: one for x_i and one for its negation $\neg x_i$.
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. number of clauses. The graph has $2N$ vertices and $2M$ edges.

```

23 Variables are 1-indexed. A negative value -k
24 ↪ denotes the negation of x_k.
25 This adds two implications: (-i => j) and (-j =>
26 ↪ i).
27 """
28 # Add edge for (-i => j)
29
30 ↪ self.graph[self._map_var(-i)].append(self._map_var(j))
31 # Add edge for (-j => i)
32
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 where
41         ↪ the first element is
42         True if a solution exists, False otherwise. If
43         ↪ a solution exists,
44         the second element is a list of boolean values
45         ↪ representing a
46         satisfying assignment. Otherwise, it is None.
47     """
48     sccs = find_sccs(self.graph, 2 * self.n)
49     component_id = [-1] * (2 * self.n)
50     for idx, comp in enumerate(sccs):
51         for node in comp:
52             component_id[node] = idx
53
54     for i in range(self.n):
55         if component_id[i] == component_id[i +
56 ↪ self.n]:
57             return False, None
58
59     assignment = [False] * self.n
60     # sccs are returned in reverse topological order
61     for i in range(self.n):
62         # If component of x_i comes after component of
63         ↪ not(x_i) in topo order
64         # (i.e., has a smaller index in the reversed
65         ↪ list), then x_i must be true.
66         if component_id[i] < component_id[i +
67 ↪ self.n]:
68             assignment[i] = True
69
70     return True, assignment

```

```

1 import sys
2 import os
3
4 # The stress test runner adds the project root to the
5 ↪ path.
6 # This allows importing other content modules using their
7 ↪ 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 graph
17         ↪ 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.

```


Chapter 6

String Algorithms

Aho Corasick

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

1. Preprocessing (Building the Automaton): a. A trie is constructed from the set of all patterns. Each node in the trie represents a prefix of one or more patterns. b. An output list is associated with each node, storing the indices of patterns that end at that node. c. "Failure links" are computed for each node. The failure link of a node u points to the longest proper suffix of the string corresponding to u that is also a prefix of some pattern in the set. These links are computed using a Breadth-First Search (BFS) starting from the root.
2. Searching: a. The algorithm processes the text character by character, traversing the automaton. It starts at the root. b. For each character in the text, it transitions to the next state. If a direct child for the character does not exist, it follows failure links until a valid transition is found or it returns to the root. c. At each state, it collects all matches. This is done by checking the output of the current node and recursively following failure links to find all patterns that end as a suffix of the current prefix.

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

```
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"]:
18                    self.trie[node_idx]["children"][char]
19                    ↪ = len(self.trie)
20                    self.trie.append({"children": {},
21                    ↪ "output": [], "fail_link": 0})
```

```
18        node_idx =
19            ↪ self.trie[node_idx]["children"][char]
20        self.trie[node_idx]["output"].append(i)
21
22    def _build_failure_links(self):
23        q = deque()
24        for char, next_node_idx in
25            ↪ self.trie[0]["children"].items():
26            q.append(next_node_idx)
27
28        while q:
29            curr_node_idx = q.popleft()
30            for char, next_node_idx in
31                ↪ self.trie[curr_node_idx]["children"].items():
32                fail_idx =
33                    ↪ self.trie[curr_node_idx]["fail_link"]
34                while char not in
35                    ↪ self.trie[fail_idx]["children"] and
36                    ↪ fail_idx != 0:
37                    fail_idx =
38                        ↪ self.trie[fail_idx]["fail_link"]
39
40                if char in
41                    ↪ self.trie[fail_idx]["children"]:
42                    self.trie[next_node_idx]["fail_link"]
43                    ↪ = self.trie[fail_idx][
44                        ↪ "children"
45                        ↪ ][char]
46            else:
47                self.trie[next_node_idx]["fail_link"]
48                ↪ = 0
49
50        # Append outputs from the failure link
51        ↪ node
52        fail_output_idx =
53            ↪ self.trie[next_node_idx]["fail_link"]
54
55        ↪ self.trie[next_node_idx]["output"].extend(
56            ↪ self.trie[fail_output_idx]["output"]
57        )
58        q.append(next_node_idx)
59
60    def search(self, text):
61        """
62        Finds all occurrences of the patterns in the given
63        ↪ text.
64
65        Args:
66            text (str): The text to search within.
67
68        Returns:
69            list[tuple[int, int]]: A list of tuples, where
70            ↪ each tuple is
71            ↪ (pattern_index, end_index_in_text).
72            ↪ `end_index_in_text` is the
73            ↪ index where the pattern ends.
74        """
75        matches = []
76        curr_node_idx = 0
77        for i, char in enumerate(text):
78            while (
79                char not in
80                ↪ self.trie[curr_node_idx]["children"]
81                ↪ and curr_node_idx != 0
82            ):
83                curr_node_idx =
84                    ↪ self.trie[curr_node_idx]["fail_link"]
```

```

67         if char in
        ↪ self.trie[curr_node_idx]["children"]:
68             curr_node_idx =
        ↪ self.trie[curr_node_idx]["children"][char]
69         else:
70             curr_node_idx = 0
71
72         if self.trie[curr_node_idx]["output"]:
73             for pattern_idx in
        ↪ self.trie[curr_node_idx]["output"]:
74                 matches.append((pattern_idx, i))
75     return matches
76

```

Kmp

Implements the Knuth-Morris-Pratt (KMP) algorithm for efficient string searching. KMP finds all occurrences of a pattern P within a text T in linear time. The core of the KMP algorithm is the precomputation of a "prefix function" or Longest Proper Prefix Suffix (LPS) array for the pattern. The LPS array, `lps`, for a pattern of length M stores at each index i the length of the longest proper prefix of $P[0 \dots i]$ 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. the pattern. $O(M)$ for building the LPS array and $O(N)$ for the search.

```

1 def compute_lps(pattern):
2     """
3     Computes the Longest Proper Prefix Suffix (LPS) array
        ↪ for the KMP algorithm.
4
5     Args:
6         pattern (str): The pattern string.
7
8     Returns:

```

```

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

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.

```

1 def manacher(s):
2     """
3     Finds the longest palindromic substring in a string
4     ↪ using Manacher's algorithm.
5
6     Args:
7         s (str): The input string.
8
9     Returns:
10        str: The longest palindromic substring found in
11        ↪ `s`. If there are
12        ↪ multiple of the same maximum length, it
13        ↪ returns the first one found.
14    """
15    if not s:
16        return ""
17
18    t = "#" + s + "#"
19    n = len(t)
20    p = [0] * n
21    center, right = 0, 0
22    max_len, max_center = 0, 0
23
24    for i in range(n):
25        mirror = 2 * center - i
26
27        if i < right:
28            p[i] = min(right - i, p[mirror])
29
30        while (
31            i - (p[i] + 1) >= 0
32            and i + (p[i] + 1) < n
33            and t[i - (p[i] + 1)] == t[i + (p[i] + 1)]
34        ):
35            p[i] += 1
36
37        if i + p[i] > right:
38            center = i
39            right = i + p[i]
40
41        if p[i] > max_len:
42            max_len = p[i]
43            max_center = i
44
45    start = (max_center - max_len) // 2
46    end = start + max_len
47    return s[start:end]

```

Polynomial Hashing

Implements a string hashing class using the polynomial rolling hash technique. This allows for efficient comparison of substrings. After an initial $O(N)$ precomputation on a string of length N , the hash of any substring can be calculated in $O(1)$ time. The hash of a string $s = s_0s_1\dots s_{k-1}$ 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 .

```

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] *
18                                ↪ self.p) % self.m1
19            self.p_powers2[i] = (self.p_powers2[i - 1] *
20                                ↪ self.p) % self.m2
21
22        self.h1 = [0] * (self.n + 1)
23        self.h2 = [0] * (self.n + 1)
24        for i in range(self.n):
25            self.h1[i + 1] = (self.h1[i] * self.p +
26                              ↪ ord(self.s[i])) % self.m1
27            self.h2[i + 1] = (self.h2[i] * self.p +
28                              ↪ ord(self.s[i])) % self.m2
29
30    def query(self, l, r):
31        """
32        Computes the hash of the substring s[l...r-1].
33
34        Args:
35            l (int): The 0-based inclusive starting
36            ↪ index.
37            r (int): The 0-based exclusive ending index.
38
39        Returns:
40            tuple[int, int]: A tuple containing the two
41            ↪ hash values for the substring.
42        """
43        if l >= r:
44            return 0, 0

```

```

40     len_sub = r - 1
41     hash1 = (
42         self.h1[r] - (self.h1[1] *
43             ↪ self.p_powers1[len_sub]) % self.m1 +
44             ↪ self.m1
45     ) % self.m1
46     hash2 = (
47         self.h2[r] - (self.h2[1] *
48             ↪ self.p_powers2[len_sub]) % self.m2 +
49             ↪ self.m2
50     ) % self.m2
51     return hash1, hash2

```

Suffix Array

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

- Initially, suffixes are sorted based on their first character.
- 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 $s[i + 2^{k-1}]$).
- This process continues for $\log N$ iterations, with each sort taking $O(N \log N)$ time, leading to an overall complexity of $O(N \log^2 N)$.

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

```

1 def build_suffix_array(s):
2     """
3     Builds the suffix array for a string using an  $O(N \log^2 N)$ 
4     ↪ sorting-based approach.
5
6     Args:
7         s (str): The input string.
8
9     Returns:
10        list[int]: The suffix array, containing starting
11        ↪ indices of suffixes in
12        ↪ lexicographically sorted order.
13    """
14    n = len(s)

```

```

13    sa = list(range(n))
14    rank = [ord(c) for c in s]
15    k = 1
16    while k < n:
17        sa.sort(key=lambda i: (rank[i], rank[i + k] if i
18            ↪ + k < n else -1))
19        new_rank = [0] * n
20        new_rank[sa[0]] = 0
21        for i in range(1, n):
22            prev, curr = sa[i - 1], sa[i]
23            r_prev = (rank[prev], rank[prev + k] if prev
24                ↪ + k < n else -1)
25            r_curr = (rank[curr], rank[curr + k] if curr
26                ↪ + k < n else -1)
27            if r_curr == r_prev:
28                new_rank[curr] = new_rank[prev]
29            else:
30                new_rank[curr] = new_rank[prev] + 1
31        rank = new_rank
32        if rank[sa[-1]] == n - 1:
33            break
34        k *= 2
35    return sa
36
37 def build_lcp_array(s, sa):
38     """
39     Builds the LCP array using Kasai's algorithm in  $O(N)$ 
40     ↪ time.
41
42     Args:
43         s (str): The input string.
44         sa (list[int]): The suffix array for the string
45         ↪ `s`.
46
47     Returns:
48         list[int]: The LCP array. `lcp[i]` is the LCP of
49         ↪ suffixes `sa[i-1]` and `sa[i]`.
50         ↪ `lcp[0]` is conventionally 0.
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] == s[j
69            ↪ + h]:
70            h += 1
71        lcp[rank[i]] = h
72    return lcp

```

Z Algorithm

Implements the Z-algorithm, which computes the Z-array for a given string `s` of length `N`. The Z-array `z` is an array of length `N` where `z[i]` is the length of the longest common prefix (LCP) between the original string `s` and the suffix of `s` starting at index `i`. By convention, `z[0]` is usually set to 0 or `N`; here it is set to 0. The algorithm computes the Z-array in

linear time. It does this by maintaining the bounds of the rightmost substring that is also a prefix of s . This is called the "Z-box", denoted by $[l, r]$. The algorithm iterates from $i = 1$ to $N-1$:

1. If i is outside the current Z-box ($i > r$), it computes $z[i]$ naively by comparing characters from the start of the string and from index i . It then updates the Z-box $[l, r]$ if a new rightmost one is found.
2. If i is inside the current Z-box ($i \leq r$), 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 ' $P + \text{'\textasciitilde'} + T$ ', where '\textasciitilde' is a character not in P or T . Any $z[i]$ equal to the length of P indicates an occurrence of P in T .

```
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    n = len(s)
12    if n == 0:
13        return []
14
15    z = [0] * n
16    l, r = 0, 0
17    for i in range(1, n):
18        if i <= r:
19            z[i] = min(r - i + 1, z[i - l])
20            while i + z[i] < n and s[z[i]] == s[i + z[i]]:
21                z[i] += 1
22            if i + z[i] - 1 > r:
23                l, r = i, i + z[i] - 1
24    return z
25
```

Chapter 7

Mathematics & Number Theory

Chinese Remainder Theorem

Implements a solver for a system of linear congruences using the Chinese Remainder Theorem (CRT). Given a system of congruences: $x \equiv a_1 \pmod{n_1}$, $x \equiv a_2 \pmod{n_2}$, ..., $x \equiv a_k \pmod{n_k}$, the algorithm finds a solution x that satisfies all of them. This implementation handles the general case where the moduli n_i are not necessarily pairwise coprime. The algorithm works by iteratively combining pairs of congruences. Given a solution for the first $i-1$ congruences, $x \equiv a_{\text{res}} \pmod{n_{\text{res}}}$, 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. Each merge step involves `extended_gcd`, which is logarithmic.

```
1 from content.math.modular_arithmetic import extended_gcd
2
3
4 def chinese_remainder_theorem(remainders, moduli):
5     """
6     Solves a system of linear congruences.
7     `x \equiv remainders[i] (mod moduli[i])` for all i.
8
9     Args:
10         remainders (list[int]): A list of remainders
11         ↪ (a_i).
12         moduli (list[int]): A list of moduli (n_i).
13
14     Returns:
15         tuple[int, int] | None: A tuple `(result, lcm)`
16         ↪ representing the solution
17         `x \equiv result (mod lcm)`, or None if no
18         ↪ solution exists.
19     """
20     if not remainders or not moduli or len(remainders) !=
21     ↪ len(moduli):
22         return 0, 1
23
24     a1 = remainders[0]
25     n1 = moduli[0]
26
27     for i in range(1, len(remainders)):
28         a2 = remainders[i]
29         n2 = moduli[i]
30
31         g, x, _ = extended_gcd(n1, n2)
32
33         if (a1 - a2) % g != 0:
34             return None
```

```
31
32
33     # Solve k * n1 \equiv a2 - a1 (mod n2)
34     # k * (n1/g) \equiv (a2 - a1)/g (mod n2/g)
35     # k \equiv ((a2 - a1)/g) * inv(n1/g) (mod n2/g)
36     # inv(n1/g) mod (n2/g) is x from extended_gcd(n1,
37     ↪ 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 + k0*n1) +
43     ↪ 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

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

1. Pick a base a (a "witness").
2. Compute $x = a^d \pmod{n}$.
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.

```
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 Miller-Rabin
7     ↪ primality test.
```


*This implementation is deterministic for all integers
 \hookrightarrow up to 2^{64} .*

Args:

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

Returns:

`bool`: True if `n` is prime, False otherwise.

"""

if `n` < 2:

 return False

if `n` == 2 or `n` == 3:

 return True

if `n` % 2 == 0 or `n` % 3 == 0:

 return False

`d` = `n` - 1

`s` = 0

while `d` % 2 == 0:

`d` //= 2

`s` += 1

*# A set of witnesses that is deterministic for all
 \hookrightarrow 64-bit integers.*

`witnesses` = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
 \hookrightarrow 37]

for `a` in `witnesses`:

 if `a` >= `n`:

 break

`x` = power(`a`, `d`, `n`)

 if `x` == 1 or `x` == `n` - 1:

 continue

`is_composite` = True

 for `_` in range(`s` - 1):

`x` = power(`x`, 2, `n`)

 if `x` == `n` - 1:

`is_composite` = False

 break

 if `is_composite`:

 return False

return True

and only if `a` and `m` are coprime (i.e., $\gcd(a, m) = 1$). This implementation uses the Extended Euclidean Algorithm. From $a \cdot x + m \cdot y = 1$, taking the equation modulo `m` gives $a \cdot x \equiv 1 \pmod{m}$. Thus, the coefficient `x` is the desired inverse.

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

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

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

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

Modular Arithmetic

This module provides essential functions for modular arithmetic, a cornerstone of number theory in competitive programming. It includes modular exponentiation, the Extended Euclidean Algorithm, and modular multiplicative inverse. Modular Exponentiation: The `power` function computes $(\text{base}^{\text{exp}}) \pmod{\text{mod}}$ 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

Ntt

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

1. Choosing a prime modulus MOD of the form $c \cdot 2^k + 1$ 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.

```

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, (MOD -
24             ↪ 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) % MOD
31                 a[i + j] = (u + v) % MOD
32                 a[i + j + length // 2] = (u - v + MOD) %
33                 ↪ MOD
34                 w = (w * wlen) % MOD
35             i += length
36             length <<= 1
37
38     if invert:
39         n_inv = power(n, MOD - 2, MOD)
40         for i in range(n):
41             a[i] = (a[i] * n_inv) % MOD
42
43 def multiply(a, b):
44     if not a or not b:
45         return []
46
47     res_len = len(a) + len(b) - 1
48     n = 1
49     while n < res_len:
50         n <<= 1
51
52     fa = a[:] + [0] * (n - len(a))
53     fb = b[:] + [0] * (n - len(b))

```

```

53 ntt(fa, False)
54 ntt(fb, False)
55
56 for i in range(n):
57     fa[i] = (fa[i] * fb[i]) % MOD
58
59 ntt(fa, True)
60
61 return fa[:res_len]
62
63

```

Pollard Rho

Implements Pollard's Rho algorithm for integer factorization, combined with Miller-Rabin primality test for a complete factorization routine. Pollard's Rho is a probabilistic algorithm to find a non-trivial factor of a composite number n . It's particularly efficient at finding small factors. The algorithm uses Floyd's cycle-detection algorithm on a sequence of pseudorandom numbers modulo n , defined by $x_{i+1} = (x_i^2 + c) \bmod n$. A factor is likely found when $\gcd(|x_j - x_i|, n) > 1$. 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 .

```

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 Pollard's
8     ↪ 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

```

```

19     if is_prime[p]:
20         for multiple in range(p * p, n + 1, p):
21             is_prime[multiple] = False
22
23     return is_prime
24

```

Sieve

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

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

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

```

1 def sieve(n):
2     """
3     Generates a sieve of primes up to n using the Sieve of
4     ↳ 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 where
12        ↳ 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):

```

Chapter 8

Geometry

Convex Hull

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

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

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

```
1 from content.geometry.point import Point, orientation
2
3
4 def convex_hull(points):
5     """
6     Computes the convex hull of a set of points using the
7     ↪ Monotone Chain algorithm.
8
9     Args:
10         points (list[Point]): A list of Point objects.
11
12     Returns:
13         list[Point]: A list of Point objects representing
14         ↪ the vertices of the
15         ↪ convex hull in counter-clockwise
16         ↪ order. Returns an empty
17         ↪ list if fewer than 3 points are
18         ↪ provided.
19
20     """
21     n = len(points)
22     if n <= 2:
23         return points
24
25     # 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 start/end
46 ↪ points
47 return lower_hull[:-1] + upper_hull[:-1]
```

Line Intersection

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

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

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

```

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 function
7     ↪ 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' intersect.
20     """
21    o1 = orientation(p1, q1, p2)
22    o2 = orientation(p1, q1, q2)
23    o3 = orientation(p2, q2, p1)
24    o4 = orientation(p2, q2, q1)
25
26    if o1 != 0 and o2 != 0 and o3 != 0 and o4 != 0:
27        if o1 != o2 and o3 != o4:
28            return True
29        return False
30
31    if o1 == 0 and on_segment(p1, p2, q1):
32        return True
33    if o2 == 0 and on_segment(p1, q2, q1):
34        return True
35    if o3 == 0 and on_segment(p2, p1, q2):
36        return True
37    if o4 == 0 and on_segment(p2, q1, q2):
38        return True
39
40    return False
41
42
43 def line_line_intersection(p1, p2, p3, p4):
44     """
45     Finds the intersection point of two infinite lines
46     ↪ defined by (p1, p2) and (p3, p4).
47     Returns the intersection point as a Point object with
48     ↪ float coordinates,
49     or None if the lines are parallel or collinear.
50     """
51    v1 = p2 - p1
52    v2 = p4 - p3
53    denominator = v1.cross(v2)
54
55    if abs(denominator) < 1e-9:
56        return None
57
58    t = (p3 - p1).cross(v2) / denominator
59    return p1 + v1 * t

```

Point

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

- Addition/Subtraction: $p1 + p2$, $p1 - p2$

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

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

```

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 == other.y
14
15    def __lt__(self, other):
16        if self.x != other.x:
17            return self.x < other.x
18        return self.y < other.y
19
20    def __add__(self, other):
21        return Point(self.x + other.x, self.y + other.y)
22
23    def __sub__(self, other):
24        return Point(self.x - other.x, self.y - other.y)
25
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,
47     ↪ q, r).
48
49     Returns:
50         int: > 0 for counter-clockwise, < 0 for clockwise,
51         ↪ 0 for collinear.
52     """
53    val = (q.x - p.x) * (r.y - q.y) - (q.y - p.y) * (r.x - q.x)
54    if val == 0:
55        return 0

```

```

54     return 1 if val > 0 else -1
55

```

Polygon Area

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

```

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
7      ↪ Shoelace formula.
8
9      Args:
10         vertices (list[Point]): A list of Point objects
11         ↪ representing the vertices of the polygon in
12         ↪ order.
13
14     Returns:
15         float: The area of the polygon.
16     """
17     n = len(vertices)
18     if n < 3:
19         return 0.0
20
21     area = 0.0
22     for i in range(n):
23         p1 = vertices[i]
24         p2 = vertices[(i + 1) % n]
25         area += p1.cross(p2)
26
27     return abs(area) / 2.0
28
29 def polygon_centroid(vertices):
30     """
31     Calculates the centroid of a simple polygon.
32
33     Args:
34         vertices (list[Point]): A list of Point objects
35         ↪ representing the vertices of the polygon in
36         ↪ order.
37
38     Returns:
39         Point / None: A Point object representing the
40         ↪ centroid, or None if the polygon's area is zero.
41     """
42     n = len(vertices)
43     if n < 3:
44         return None
45
46     signed_area = 0.0
47     centroid_x = 0.0
48     centroid_y = 0.0
49
50     for i in range(n):
51         p1 = vertices[i]
52         p2 = vertices[(i + 1) % n]

```

```

51     cross_product = p1.cross(p2)
52
53     signed_area += cross_product
54     centroid_x += (p1.x + p2.x) * cross_product
55     centroid_y += (p1.y + p2.y) * cross_product
56
57     if abs(signed_area) < 1e-9:
58         return None
59
60     area = signed_area / 2.0
61     centroid_x /= 6.0 * area
62     centroid_y /= 6.0 * area
63
64     return Point(centroid_x, centroid_y)
65

```


Chapter 9

Dynamic Programming

Common Patterns

This file provides implementations for three classic dynamic programming patterns that are foundational in competitive programming: Longest Increasing Subsequence (LIS), Longest Common Subsequence (LCS), and the 0/1 Knapsack problem.

Longest Increasing Subsequence (LIS): Given a sequence of numbers, the goal is to find the length of the longest subsequence that is strictly increasing. The standard DP approach takes $O(N^2)$ time. This file implements a more efficient $O(N \log N)$ solution. The algorithm maintains an auxiliary array (e.g., `tails`) where `tails[i]` stores the smallest tail of all increasing subsequences of length $i+1$. When processing a new number x , we find the smallest tail that is greater than or equal to x . If x is larger than all tails, it extends the LIS. Otherwise, it replaces the tail it was compared against, potentially allowing for a better solution later. This search and replacement is done using binary search.

Longest Common Subsequence (LCS): Given two sequences, the goal is to find the length of the longest subsequence present in both of them. The standard DP solution uses a 2D table `dp[i][j]` which stores the length of the LCS of the prefixes `s1[0...i-1]` and `s2[0...j-1]`. The recurrence relation is:

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

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

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

- LIS: $O(N)$
- LCS: $O(N \cdot M)$
- 0/1 Knapsack: $O(W)$ (space-optimized)

```
1 import bisect
2
3
4 def longest_increasing_subsequence(arr):
5     """
6     Finds the length of the longest increasing subsequence
7     ↪ 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 subsequence in
24     ↪  $O(N \cdot M)$ .
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], dp[i][j - 1])
35                 ↪ 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, -1):
48             dp[w] = max(dp[w], values[i] + dp[w -
49                 ↪ weights[i]])
50
51     return dp[capacity]
```

Dp Optimizations

This file explains and demonstrates several advanced dynamic programming optimizations. The primary focus is the Convex Hull Trick, with conceptual explanations for Knuth-Yao Speedup and Divide and Conquer Optimization. Convex Hull Trick (CHT): This optimization applies to DP recurrences of the form: $dp[i] = \min_{j < i} (dp[j] + b[j] * a[i])$ (or similar). For a fixed i , each j defines a line $y = m * x + c$, where $m = b[j]$, $x = a[i]$, and $c = dp[j]$. The problem then becomes finding the minimum value among a set of lines for a given x -coordinate $a[i]$. A `LineContainer` data structure is used to maintain the lower envelope (convex hull) of these lines, allowing for efficient queries. The example below solves a problem with the recurrence $dp[i] = C + \min_{j < i} (dp[j] + (p[i] - p[j])^2)$, which can be rearranged into the required line form. This works efficiently if the slopes of the lines being added are monotonic. Knuth-Yao Speedup: This optimization applies to recurrences of the form $dp[i][j] = C[i][j] + \min_{i \leq k < j} (dp[i][k] + dp[k+1][j])$, 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 $[1, r]$ by first finding the optimal k for the midpoint $mid = (1+r)/2$. Then, recursively, the optimal k for the left half $[1, mid-1]$ 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.

```

1 import sys
2 import os
3
4 # The stress test runner adds the project root to the
5 # path.
6 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
7 # ..../)))
8 from content.data_structures.line_container import
9 LineContainer
10
11 def convex_hull_trick_example(p, C):
12     """
13     Solves an example problem using the Convex Hull
14     Trick.

```

```

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

```

Knapsack

Knapsack problems appear in many variations. This section provides contest-ready implementations and explains when to use each:

- 0/1 Knapsack: each item can be taken once. Weight-optimized 1D DP in $O(NC)$.
- Unbounded Knapsack: unlimited copies of each item. Forward 1D DP in $O(NC)$.
- Bounded Knapsack: limited copies per item. Use binary splitting to reduce to 0/1.
- Subset Sum via bitset: fast using Python integer shifts.
- 0/1 Value-Optimized: minimize weight to

achieve value; useful when C is large but total value is moderate.

- Bounded Knapsack (Monotone Queue): optimize per residue class to handle large capacities with counts.

Implementation notes:

- For 0/1, iterate capacity descending to avoid reusing an item.
- For unbounded, iterate ascending to allow multiple uses.
- Binary splitting transforms count into $O(\log m)$ items per original item.
- Bitset subset sum uses integer bit operations; shift left by weight and OR.
- Value-optimized uses value dimension; check minimal weight within capacity.
- Monotone queue optimization maintains a deque per residue class to enforce counts.

```

1 def knapsack_01(weights, values, capacity):
2     dp = [0] * (capacity + 1)
3     for w, v in zip(weights, values):
4         for c in range(capacity, w - 1, -1):
5             nv = dp[c - w] + v
6             if nv > dp[c]:
7                 dp[c] = nv
8     return dp[capacity]
9
10
11 def knapsack_unbounded(weights, values, capacity):
12     dp = [0] * (capacity + 1)
13     for w, v in zip(weights, values):
14         for c in range(w, capacity + 1):
15             nv = dp[c - w] + v
16             if nv > dp[c]:
17                 dp[c] = nv
18     return dp[capacity]
19
20
21 def knapsack_bounded(weights, values, counts, capacity):
22     items_w = []
23     items_v = []
24     for w, v, m in zip(weights, values, counts):
25         k = 1
26         while m > 0:
27             take = k if k <= m else m
28             items_w.append(w * take)
29             items_v.append(v * take)
30             m -= take
31             k <<= 1
32     return knapsack_01(items_w, items_v, capacity)
33
34
35 def subset_sum_possible(nums, S):
36     if S < 0:
37         return False
38     bits = 1
39     for x in nums:
40         if 0 <= x <= S:
41             bits |= bits << x
42     return ((bits >> S) & 1) == 1
43
44
45 def knapsack_01_value_optimized(weights, values,
46     ↪ capacity):
47     V = sum(values)

```

```

47     INF = capacity + 1
48     dp = [INF] * (V + 1)
49     dp[0] = 0
50     for w, v in zip(weights, values):
51         for val in range(V, v - 1, -1):
52             nw = dp[val - v] + w
53             if nw < dp[val]:
54                 dp[val] = nw
55
56     best = 0
57     for val in range(V + 1):
58         if dp[val] <= capacity and val > best:
59             best = val
60     return best
61
62 def knapsack_bounded_mq(weights, values, counts,
63     ↪ capacity):
64     dp = [0] * (capacity + 1)
65     for w, v, m in zip(weights, values, counts):
66         if w == 0:
67             if v > 0 and m > 0:
68                 return v * m + dp[capacity]
69             continue
70         for r in range(w):
71             deq = []
72             idx = 0
73             for c in range(r, capacity + 1, w):
74                 base = dp[c] - idx * v
75                 while deq and deq[-1][0] <= base:
76                     deq.pop()
77                 deq.append((base, idx))
78                 while deq and deq[0][1] < idx - m:
79                     deq.pop(0)
80                 dp[c] = deq[0][0] + idx * v
81                 idx += 1
82     return dp[capacity]
83
84

```