**~\AppData\Roaming\Code\User\snippets\python.json**

```
{
    "TrieNode": {
        "prefix": "TrieNode",
        "body": [
            "class TrieNode:",
            "    \"\"\"TrieNode class can quickly process string prefixes, a common feature used in applications like autocomplete and spell checking.\"\"\"",
            "    sid_cnt = 0  # sid counter, representing string index starting from 0",
            "",
            "    def __init__(self):",
            "        \"\"\"Initialize children dictionary and cost. The trie tree is a 26-ary tree.\"\"\"",
            "        self.children_ = {}",
            "        self.cost = INF",
            "        self.is_end_of_word = False  # Flag to indicate end of word",
            "        self.sid = -1  # Unique ID for the node, -1 if not assigned",
            "",
            "    def add(self, word: str, cost: int) -> int:",
            "        \"\"\"Add a word to the trie with the associated cost and return a unique ID.\"\"\"",
            "        node = self",
            "        for c in word:",
            "            if c not in node.children_:",
            "                node.children_[c] = Std.TrieNode()",
            "            node = node.children_[c]",
            "        node.cost = Math.min(node.cost, cost)",
            "        node.is_end_of_word = True  # Mark the end of the word",
            "        if node.sid < 0:",
            "            node.sid = self.sid_cnt",
            "            self.sid_cnt += 1",
            "        return node.sid",
            "",
            "    def search(self, word: str) -> int:",
            "        \"\"\"Search for the exact word in the trie and return its cost or unique ID.\"\"\"",
            "        node = self",
            "        for c in word:",
            "            if c not in node.children_:",
            "                return INF",
            "            node = node.children_[c]",
            "        return node.cost if node.is_end_of_word else INF"
        ],
        "description": "TrieNode"
    },
    "TrieNodeGraph": {
        "prefix": "TrieNodeGraph",
        "body": [
            "class TrieNodeGraph:",
            "    \"\"\"TrieNode class can convert each string into an integer identifier, useful in graph theory.\"\"\"",
            "    sid_cnt = 0",
            "    sid_to_word_ = {}",
            "",
            "    def __init__(self):",
            "        \"\"\"Initialize children dictionary and cost. The trie tree is a 26-ary tree.\"\"\"",
            "        self.children = {}",
            "        self.is_end_of_word = False  # Flag to indicate end of word",
            "        self.sid = -1  # Unique ID for the node, -1 if not assigned",
            "",
            "    def add(self, word: str) -> int:",
            "        \"\"\"Add a word to the trie and return a unique ID.\"\"\"",
            "        node = self",
            "        for c in word:",
            "            if c not in node.children:",
            "                node.children[c] = Std.TrieNodeGraph()",
            "            node = node.children[c]",
            "        node.is_end_of_word = True  # Mark the end of the word",
            "        if node.sid < 0:",
            "            node.sid = self.sid_cnt",
            "            self.sid_cnt += 1",
            "        self.sid_to_word_[node.sid] = word",
            "        return node.sid",
            "",
            "    def _search(self, word: str) -> int:",
            "        \"\"\"Search for the exact word in the trie and return its unique ID, else -1.\"\"\"",
            "        node = self",
            "        for c in word:",
            "            if c not in node.children:",
            "                return -1",
```

```
            "            node = node.children[c]",
            "        return node.sid if node.is_end_of_word else -1",
            "",
            "    def get_id(self, word: str) -> int:",
            "        \"\"\"Retrieve the unique ID for a given word.\"\"\"",
            "        return self._search(word)",
            "",
            "    def get_str(self, sid: int) -> str:",
            "        \"\"\"Retrieve the original string associated with a given unique ID.\"\"\",
            "        return word if (word := self.sid_to_word_.get(sid)) else \"-1\"",
            ""
        ],
        "description": "TrieNodeGraph"
    },
    "TrieNode01": {
        "prefix": "TrieNode01",
        "body": [
            "class TrieNode01:",
            "    \"\"\"",
            "    TrieNode01 class is a binary trie optimized for operations on binary strings.",
            "    Useful for problems like finding maximum XOR of two numbers in an array.",
            "    \"\"\",
            "",
            "    def __init__(self):",
            "        \"\"\"Initialize TrieNode01 with two children and a value.\"\"\",
            "        self.children: List = [None, None]  # Only two children: 0 and 1",
            "        self.value: Optional[int] = None  # Store the actual value if this is an end node",
            "        self.count = 0  # Number of numbers passing through this node",
            "",
            "    def insert(self, num: int):",
            "        \"\"\"Insert a number into the binary trie.\"\"\",
            "        node = self",
            "        for i in range(31, -1, -1):  # Assuming 32-bit integers",
            "            bit = (num >> i) & 1",
            "            if not node.children[bit]:",
            "                node.children[bit] = Std.TrieNode01()",
            "            node = node.children[bit]",
            "            node.count += 1",
            "        node.value = num  # Store the number at the leaf node",
            "",
            "    def find_max_xor(self, num: int) -> int:",
            "        \"\"\"Find the maximum XOR of 'num' with any number in the trie.\"\"\",
            "        node = self",
            "        max_xor = 0",
            "        for i in range(31, -1, -1):  # Assuming 32-bit integers",
            "            bit = (num >> i) & 1",
            "            toggled_bit = 1 - bit",
            "            if node.children[toggled_bit]:  # Prefer the toggled bit if it exists",
            "                max_xor = (max_xor << 1) | 1  # Current bit is 1 (since XOR with toggled bit)",
            "                node = node.children[toggled_bit]",
            "            else:",
            "                max_xor = (max_xor << 1)  # Current bit is 0 (since XOR with the same bit)",
            "                node = node.children[bit]",
            "        return max_xor",
            "",
            "    def count_equal(self, num: int) -> int:",
            "        \"\"\"Count the number of values in the Trie equal to the given num.\"\"\",
            "        node = self",
            "        for i in range(31, -1, -1):",
            "            bit = (num >> i) & 1",
            "            if bit in node.children:",
            "                node = node.children[bit]",
            "            else:",
            "                return 0  # If the bit path does not exist, return 0",
            "        return node.count  # Return the count of values equal to the given num"
        ],
        "description": "TrieNode01"
    },
    "SparseTable": {
        "prefix": "SparseTable",
        "body": [
            "class SparseTable:",
            "    def __init__(self, data: List, func=lambda x, y: x | y):",
            "        self.func = func",
            "        self.st = [list(data)]",
            "        i, n = 1, len(self.st[0])",
            "        while 2 * i <= n:",
            "            pre = self.st[-1]",
```

```
            "            self.st.append([func(pre[j], pre[j + i]) for j in range(n - 2 * i + 1)])",
            "            i <<= 1",
            "",
            "    def query(self, begin: int, end: int) -> int:",
            "        lg = (end - begin + 1).bit_length() - 1",
            "        return self.func(self.st[lg][begin], self.st[lg][end - (1 << lg) + 1])"
        ],
        "description": "SparseTable"
    },
    "SparseTablePro": {
        "prefix": "SparseTablePro",
        "body": [
            "class SparseTable:",
            "    def __init__(self, data: List, func=lambda x, y: x | y):",
            "        n = len(data)",
            "        self.func = func",
            "        self.st = [list(data)]",
            "        self.lg = Arr.array(0, n + 1)",
            "        for i in range(2, n + 1):",
            "            self.lg[i] = self.lg[i >> 1] + 1",
            "        k = self.lg[n]",
            "        step = 1",
            "        for _ in range(k):",
            "            prev = self.st[-1]",
            "            self.st.append([func(prev[i], prev[i + step]) for i in range(n - (step << 1) + 1)])",
            "            step <<= 1",
            "",
            "    def query(self, l: int, r: int) -> int:",
            "        k = self.lg[r - l + 1]",
            "        return self.func(self.st[k][l], self.st[k][r - (1 << k) + 1])"
        ],
        "description": "SparseTablePro"
    },
    "StringHash": {
        "prefix": "StringHash",
        "body": [
            "class StringHash:",
            "    def __init__(self, s: str, mod: int = 1_070_777_777):",
            "        self.mod = mod",
            "        self.base = randint(8 * 10 ** 8, 9 * 10 ** 8)",
            "        self.s = s",
            "        self.n = len(s)",
            "        self.pow_base = [1] + Arr.array(0, self.n)  # pow_base[i] = BASE ^ i",
            "        self.pre_hash = Arr.array(0, self.n + 1)  # pre_hash[i] = hash(s[:i])",
            "        self._compute_hash()",
            "",
            "    def _compute_hash(self):",
            "        for i, b in enumerate(self.s):",
            "            self.pow_base[i + 1] = self.pow_base[i] * self.base % self.mod",
            "            self.pre_hash[i + 1] = (self.pre_hash[i] * self.base + ord(b)) % self.mod",
            "",
            "    def get_hash(self, l: int, r: int) -> int:",
            "        return (self.pre_hash[r + 1] - self.pre_hash[l] * self.pow_base[r - l + 1] % self.mod + self.mod)
% self.mod",
            "",
            "    def compute_hash(self, word: str) -> int:",
            "        h = 0",
            "        for b in word:",
            "            h = (h * self.base + ord(b)) % self.mod",
            "        return h"
        ],
        "description": "StringHash"
    },
    "PrefixSumTwoDim": {
        "prefix": "PrefixSumTwoDim",
        "body": [
            "class PrefixSumTwoDim:",
            "    def __init__(self, matrix: List[List[int]]):",
            "        self.rows = len(matrix)",
            "        self.cols = len(matrix[0])",
            "        self.pre_sum = Arr.array2d(0, self.cols + 1, self.rows + 1)  # 1-based index",
            "",
            "        for i in range(1, self.rows + 1):",
            "            for j in range(1, self.cols + 1):",
            "                self.pre_sum[i][j] = matrix[i - 1][j - 1] + self.pre_sum[i - 1][j] + self.pre_sum[i][j -
1] - self.pre_sum[i - 1][j - 1]",
            "",
            "    def query(self, x1: int, y1: int, x2: int, y2: int) -> int:",
```

```
            "        return self.pre_sum[x2 + 1][y2 + 1] - self.pre_sum[x1][y2 + 1] - self.pre_sum[x2 + 1][y1] + self.pre_sum[x1][y1]"
        ],
        "description": "PrefixSumTwoDim"
    },
    "Bisect": {
        "prefix": "Bisect",
        "body": [
            "class Bisect:",
            "    @staticmethod",
            "    def bisect_left(a, x, key=lambda y: y, lo=0, hi=None) -> int:",
            "        if hi is None:",
            "            hi = len(a)",
            "        left, right = lo, hi",
            "        while left < right:",
            "            mid = (left + right) >> 1",
            "            if key(a[mid]) < x:",
            "                left = mid + 1",
            "            else:",
            "                right = mid",
            "        return left  # If not found, returns hi (insertion point)",
            "",
            "    @staticmethod",
            "    def bisect_right(a, x, key=lambda y: y, lo=0, hi=None) -> int:",
            "        if hi is None:",
            "            hi = len(a)",
            "        left, right = lo, hi",
            "        while left < right:",
            "            mid = (left + right) >> 1",
            "            if key(a[mid]) <= x:",
            "                left = mid + 1",
            "            else:",
            "                right = mid",
            "        return left  # If not found, returns hi (insertion point)"
        ],
        "description": "Bisect"
    },
    "Func (3.8.6)": {
        "prefix": "Func",
        "body": [
            "class Func:",
            "    @staticmethod",
            "    def find(container, value) -> int:",
            "        \"\"\"Returns the index of value in container or -1 if value is not found.\"\"\"",
            "        if isinstance(container, list):",
            "            try:",
            "                return container.index(value)",
            "            except ValueError:",
            "                return -1",
            "        elif isinstance(container, str):",
            "            return container.find(value) # type: ignore",
            "        ",
            "    @staticmethod",
            "    def pairwise(iterable):",
            "        \"\"\"Return successive overlapping pairs taken from the input iterable.\"\"\"",
            "        a, b = tee(iterable)",
            "        next(b, None)",
            "        return zip(a, b)"
        ],
        "description": "Func (3.8.6) Supplement"
    },
    "TreeAncestor": {
        "prefix": "TreeAncestor",
        "body": [
            "class TreeAncestor:",
            "    def __init__(self, n: int, m: int, parent: List[int]):",
            "        \"\"\"",
            "        Initializes the TreeAncestor with the given number of nodes and parent list.",
            "        Args:",
            "            n (int): Number of nodes.",
            "            m (int): Maximum power of 2 to consider.",
            "            parent (List[int]): List where parent[i] is the parent of node i.",
            "        \"\"\"",
            "        # Default: m = n.bit_length()",
            "        self.n = n",
            "        self.m = m",
            "        pa = [[p] + Arr.array(-1, m - 1) for p in parent]  # pa[i][0] = p",
            "        for i in range(m - 1):",
```

```
            "            for x in range(n):",
            "                p = pa[x][i]  # 2^i-th ancestor of node x",
            "                if p != -1:",
            "                    pp = pa[p][i]  # 2^i-th ancestor of p, which will be the 2^(i+1)-th ancestor of x",
            "                    pa[x][i + 1] = pp  # Set the 2^(i+1)-th ancestor of x",
            "        self.pa = pa",
            "",
            "    def get_kth_ancestor(self, node: int, k: int) -> int:",
            "        \"\"\"Returns the k-th ancestor of the given node (The starting node). If not exists, return -1\"\"\"",
            "        for j in range(k.bit_length()):",
            "            if (k >> j) & 1:",
            "                node = self.pa[node][j]",
            "                if node < 0:",
            "                    break",
            "        return node"
        ],
        "description": "TreeAncestor"
    },
    "LCA": {
        "prefix": "LCA",
        "body": [
            "class LCA:",
            "    def __init__(self, edges: List[List[int]]):",
            "        self.n = len(edges) + 1",
            "        self.m = self.n.bit_length()",
            "        self.g = Arr.graph(self.n)",
            "        for x, y in edges:",
            "            self.g[x].append(y)",
            "            self.g[y].append(x)",
            "",
            "        self.depth = Arr.array(0, self.n)",
            "        self.pa = Arr.array2d(-1, self.n, self.m)  # pa[v][k]",
            "",
            "        def _dfs(x: int, fa: int) -> None:",
            "            \"\"\"Depth-first search to initialize the ancestor table and depth array.\"\"\"",
            "            self.pa[x][0] = fa  # init itself",
            "            for y in self.g[x]:",
            "                if y != fa:",
            "                    self.depth[y] = self.depth[x] + 1",
            "                    _dfs(y, x)",
            "        _dfs(0, -1)",
            "",
            "        for k in range(self.m - 1):",
            "            for v in range(self.n):",
            "                if self.pa[v][k] == -1:",
            "                    self.pa[v][k + 1] = -1",
            "                else:",
            "                    self.pa[v][k + 1] = self.pa[self.pa[v][k]][k]",
            "",
            "    def get_lca(self, u: int, v: int) -> int:",
            "        \"\"\"Returns the Lowest Common Ancestor (LCA) of nodes u and v.\"\"\"",
            "        if self.depth[u] > self.depth[v]:",
            "            u, v = v, u",
            "        # Bring u and v to the same depth",
            "        for k in range(self.m):",
            "            if ((self.depth[v] - self.depth[u]) >> k) & 1:",
            "                v = self.pa[v][k]",
            "        if v == u:",
            "            return u",
            "        for k in reversed(range(self.m)):",
            "            if self.pa[u][k] != self.pa[v][k]:",
            "                u = self.pa[u][k]",
            "                v = self.pa[v][k]",
            "        return self.pa[u][0]  # Return the parent of u (or v) as LCA",
            "",
            "    def search(self, v: int, x: int):",
            "        \"\"\"Find the ancestor of node v that is x steps up using binary lifting.\"\"\"",
            "        for k in reversed(range(self.m)):",
            "            if x >> k & 1:",
            "                v = self.pa[v][k]",
            "        return v"
        ],
        "description": "LCA"
    },
    "Floyd": {
        "prefix": "Floyd",
        "body": [
```

```
            "class Floyd:",
            "    def __init__(self, n: int):",
            "        self.n = n",
            "        self.g = Arr.graph(n)",
            "        self.dist = Arr.array2d(INF, self.n, self.n)",
            "        self.dp = Arr.array(0, n)",
            "",
            "    def add_edge(self, u: int, v: int, w: int):",
            "        self.g[u].append((v, w))",
            "",
            "    def floyd(self):",
            "        for u in range(self.n):",
            "            for v, w in self.g[u]:",
            "                self.dist[u][v] = Math.min(self.dist[u][v], w)",
            "",
            "        for i in range(self.n):",
            "            self.dist[i][i] = 0",
            "",
            "        for k in range(self.n):",
            "            for i in range(self.n):",
            "                if self.dist[i][k] > INF // 2:  # If there is no path from i to k, skip",
            "                    continue",
            "                for j in range(self.n):",
            "                    if self.dist[i][j] > self.dist[i][k] + self.dist[k][j]:",
            "                        self.dist[i][j] = self.dist[i][k] + self.dist[k][j]",
            "",
            "    def floyd_01(self):",
            "        n = self.n",
            "        for u in range(n):",
            "            self.dp[u] |= 1 << u",
            "            for v, _ in self.g[u]:",
            "                self.dp[u] |= 1 << v",
            "",
            "        for k in range(n):",
            "            for i in range(n):",
            "                if self.dp[i] >> k & 1:",
            "                    self.dp[i] |= self.dp[k]",
            "",
            "    def get_dist(self, x: int, y: int) -> int:",
            "        return self.dist[x][y] if self.dist[x][y] < INF // 2 else INF",
            "",
            "    def get_dist_01(self, x: int, y: int) -> bool:",
            "        return self.dp[x] >> y & 1"
        ],
        "description": "Floyd"
    },
    "Spfa": {
        "prefix": "Spfa",
        "body": [
            "class Spfa:",
            "    def __init__(self, n: int):",
            "        self.n = n",
            "        self.g = Arr.graph(n)",
            "",
            "    def add_edge(self, u: int, v: int, w: int):",
            "        self.g[u].append((v, w))",
            "",
            "    def spfa(self, s: int) -> List[int]:",
            "        dist = Arr.array(INF, self.n)",
            "        st = Arr.array(0, self.n)",
            "        q = deque()",
            "",
            "        dist[s] = 0",
            "        q.appendleft(s)",
            "        st[s] = 1",
            "",
            "        while q:",
            "            u = q.pop()",
            "            st[u] = 0",
            "            for v, w in self.g[u]:",
            "                if dist[v] > dist[u] + w:",
            "                    dist[v] = dist[u] + w",
            "                    if st[v] == 0:",
            "                        q.appendleft(v)",
            "                        st[v] = 1",
            "",
            "        return dist"
        ],
```

```
        "description": "Spfa"
    },
    "Dijkstra": {
        "prefix": "Dijkstra",
        "body": [
            "class Dijkstra:",
            "    def __init__(self, n: int, val: List[int]):",
            "        self.n = n                                      # 节点数量",
            "        self.val = val                                  # 每个节点的权值",
            "        self.g = Arr.graph(n)                         # 邻接表表示的有向图",
            "        self.dist = Arr.array(INF, n)                # 源点到各节点的最短距离",
            "        self.sum = Arr.array(0, n)                   # 对应最短路的节点权值总和（取最大）",
            "        self.cnt = Arr.array(0, n)                   # 最短路条数",
            "        self.pre = Arr.array(-1, n)                  # 最优前驱节点，用于还原路径",
            "        self.num = Arr.array(0, n)                   # 最短路经过的节点数",
            "",
            "    def add_edge(self, u: int, v: int, w: int):",
            "        \"\"\"Add an edge to the graph.\"\"\"",
            "        self.g[u].append((v, w))",
            "",
            "    def dijkstra(self, s: int):",
            "        \"\"\"",
            "        Dijkstra 算法：计算从源点 s 到所有点的最短路径。",
            "",
            "        同时满足以下四个目标：",
            "        1. 求出每个点的最短距离 self.dist(边权和最小)",
            "        2. 统计到每个点的最短路径条数 self.cnt",
            "        3. 在所有最短路径中，优先选择节点权值总和最大的路径(self.sum)",
            "        4. 若节点权值和相等，则优先选择经过节点数最少的路径(self.num)",
            "",
            "        最终可以用于还原最优路径(self.pre)、比较不同方案等功能。",
            "        \"\"\"",
            "        st_ = Arr.array(0, self.n)",
            "        q = []",
            "",
            "        self.dist[s] = 0",
            "        self.sum[s] = self.val[s]",
            "        self.cnt[s] = self.num[s] = 1",
            "        heappush(q, (0, s))",
            "",
            "        while q:",
            "            _, u = heappop(q)",
            "            if st_[u]:",
            "                continue",
            "            st_[u] = 1",
            "            for v, w in self.g[u]:",
            "                if self.dist[v] > self.dist[u] + w:",
            "                    self.dist[v] = self.dist[u] + w",
            "                    self.pre[v] = u",
            "                    self.sum[v] = self.sum[u] + self.val[v]",
            "                    self.num[v] = self.num[u] + 1",
            "                    self.cnt[v] = self.cnt[u]",
            "                    heappush(q, (self.dist[v], v))",
            "                elif self.dist[v] == self.dist[u] + w:",
            "                    self.cnt[v] += self.cnt[u]",
            "                    if self.sum[u] > self.sum[self.pre[v]]:",
            "                        self.pre[v] = u",
            "                        self.sum[v] = self.sum[u] + self.val[v]",
            "                        self.num[v] = self.num[u] + 1",
            "                        heappush(q, (self.dist[v], v))",
            "                    elif self.sum[u] == self.sum[self.pre[v]] and self.num[v] > self.num[u] + 1:",
            "                        self.pre[v] = u",
            "                        self.num[v] = self.num[u] + 1",
            "                        heappush(q, (self.dist[v], v))"
        ],
        "description": "Dijkstra"
    },
    "DijkstraSimple": {
        "prefix": "DijkstraSimple",
        "body": [
            "class Dijkstra:",
            "    def __init__(self):",
            "        self.g = defaultdict(list)",
            "        self.dist = defaultdict(lambda: INF)",
            "",
            "    def add_edge(self, u: int, v: int, w: int):",
            "        self.g[u].append((v, w))",
            "",
```

```
            "    def dijkstra(self, s: int):",
            "        st = defaultdict(lambda: False)",
            "        q = []",
            "        self.dist[s] = 0",
            "        heappush(q, (0, s))",
            "",
            "        while q:",
            "            _, u = heappop(q)",
            "            if st[u]:",
            "                continue",
            "            st[u] = True",
            "            for v, w in self.g[u]:",
            "                if self.dist[v] > self.dist[u] + w:",
            "                    self.dist[v] = self.dist[u] + w",
            "                    heappush(q, (self.dist[v], v))"
        ],
        "description": "DijkstraSimple"
    },
    "Str": {
        "prefix": "Str",
        "body": [
            "class Str:",
            "    atoi = staticmethod(lambda x: ord(x.upper()) - 65)  # A → 0",
            "    itoa = staticmethod(lambda x: ascii_uppercase[x])   # 0 → A",
            "    removeprefix = staticmethod(lambda s, prefix: s[len(prefix):] if s.startswith(prefix) else s)",
            "    removesuffix = staticmethod(lambda s, suffix: s[:-len(suffix)] if s.endswith(suffix) else s)"
        ],
        "description": "Str"
    },
    "IO": {
        "prefix": "IO",
        "body": [
            "class IO:",
            "    input = staticmethod(lambda: stdin.readline().strip())",
            "    read = staticmethod(lambda: map(int, IO.input().split()))",
            "    read_list = staticmethod(lambda: list(IO.read()))",
            "    read_mixed = staticmethod(lambda *types: [t(v) for t, v in zip(types, IO.input().split())])"
        ],
        "description": "IO"
    },
    "Mod": {
        "prefix": "Mod",
        "body": [
            "class Mod:",
            "    add = staticmethod(lambda *args: (lambda result=0: [(result := (result + num) % MOD) for num in args] and result)())",
            "    sub = staticmethod(lambda a, b: (a - b + MOD) % MOD)",
            "    mul = staticmethod(lambda *args: (lambda result=1: [(result := (result * num) % MOD) for num in args] and result)())",
            "    div = staticmethod(lambda a, b: (a * pow(b, MOD - 2, MOD)) % MOD)",
            "    mod = staticmethod(lambda a: (a % MOD + MOD) % MOD)"
        ],
        "description": "Mod"
    },
    "MonoDequeIdx": {
        "prefix": "MonoDequeIdx",
        "body": [
            "class MonoDequeIdx:",
            "    def __init__(self, is_min: bool = True, size: int = None) -> None:",
            "        # (idx, val)",
            "        self.dq: Deque[tuple[int, int]] = deque()",
            "        self.size = size",
            "        if is_min:",
            "            self._cmp = lambda x, y: x <= y",
            "        else:",
            "            self._cmp = lambda x, y: x >= y",
            "",
            "    def push(self, idx: int, val: int) -> None:",
            "        # 1. 淘汰过期",
            "        if self.size is not None:",
            "            expire = idx - self.size",
            "            while self.dq and self.dq[0][0] <= expire:",
            "                self.dq.popleft()",
            "",
            "        # 2. 保持单调",
            "        while self.dq and self._cmp(val, self.dq[-1][1]):",
            "            self.dq.pop()",
            "",
```

```
            "        # 3. 入队",
            "        self.dq.append((idx, val))",
            "",
            "    def query(self) -> int:",
            "        return self.dq[0][1]"
        ],
        "description": "MonoDequeIdx"
    },
    "MonoStackIdx": {
        "prefix": "MonoStackIdx",
        "body": [
            "class MonoStackIdx:",
            "    def __init__(self, is_min: bool = True) -> None:",
            "        # (idx, val)",
            "        self.stack: List[Tuple[int, Any]] = []",
            "        if is_min:",
            "            self._cmp = lambda x, y: x ≤ y",
            "        else:",
            "            self._cmp = lambda x, y: x ≥ y",
            "",
            "    def push(self, idx: int, val: int) -> None:",
            "        while self.stack and self._cmp(val, self.stack[-1][1]):",
            "            self.stack.pop()",
            "        self.stack.append((idx, val))",
            "",
            "    def pop(self) -> Optional[Tuple[int, Any]]:",
            "        return self.stack.pop() if self.stack else None",
            "",
            "    def top(self) -> Optional[Tuple[int, Any]]:",
            "        return self.stack[-1] if self.stack else None"
        ],
        "description": "MonoStackIdx"
    },
    "BipartiteMatcher": {
        "prefix": "BipartiteMatcher",
        "body": [
            "class BipartiteMatcher:",
            "    def __init__(self, n: int):",
            "        \"\"\"Initialize the BipartiteMatcher with n nodes.\"\"\"",
            "        self.n = n",
            "        self.match_ = Arr.array(0, self.n)",
            "        self.st_ = Arr.array(0, self.n)",
            "        self.g_ = Arr.graph(self.n)",
            "",
            "    def add_edge(self, u: int, v: int):",
            "        \"\"\"Add an edge between node u and node v.\"\"\"",
            "        self.g_[u].append(v)",
            "",
            "    def find(self, u: int) -> bool:",
            "        \"\"\"Find an augmenting path starting from node u.\"\"\"",
            "        for v in self.g_[u]:",
            "            if self.st_[v] == 0:",
            "                self.st_[v] = 1",
            "                if self.match_[v] == 0 or self.find(self.match_[v]):",
            "                    self.match_[v] = u",
            "                    return True",
            "        return False",
            "",
            "    def max_matching(self) -> int:",
            "        \"\"\"Compute the maximum matching in the bipartite graph.\"\"\"",
            "        res = 0",
            "        for i in range(self.n):",
            "            self.st_ = Arr.array(0, self.n)",
            "            if self.find(i):",
            "                res += 1",
            "        return res"
        ],
        "description": "BipartiteMatcher"
    },
    "SegTreeDynamic": {
        "prefix": "SegTreeDynamic",
        "body": [
            "class SegTree:",
            "    \"\"\"",
            "    A segment tree based on dynamic binary tree algorithm. ",
            "    Supports passing callback functions `f1` and `f2` to handle range queries (RMQ) such as range sum, range maximum, and range minimum.",
            "    \"\"\",
```

```
            "",
            "    def __init__(self, f1: Callable, f2: Callable, l: int, r: int, v: int = 0):",
            "        \"\"\"",
            "        Initializes the segment tree [left, right).",
            "",
            "        Example functions:",
            "            Segment Sum:",
            "                f1 = lambda a, b: a + b",
            "                f2 = lambda a, n: a * n",
            "            Segment Maximum:",
            "                f1 = lambda a, b: Math.max(a, b)",
            "                f2 = lambda a, n: a",
            "            Segment Minimum:",
            "                f1 = lambda a, b: Math.min(a, b)",
            "                f2 = lambda a, n: a",
            "        Args:",
            "            f1: Function for combining segment values. (merge values from different intervals)",
            "            f2: Function for applying values to segments. (Spread a value to an interval)",
            "            l (int): Left boundary of the segment.",
            "            r (int): Right boundary of the segment.",
            "            v (int): Initial value for the segment.",
            "        \"\"\",
            "        self._default = v  # Default value for the segments",
            "        self._ans = f2(v, r-l)  # Current result of the segment",
            "        self._f1 = f1",
            "        self._f2 = f2",
            "        self._l = l  # left",
            "        self._r = r  # right",
            "        self._v = v  # init value",
            "        self._lazy_tag = 0  # Lazy tag",
            "        self._left = None  # SubTree(left, bottom)",
            "        self._right = None  # SubTree(right, bottom)",
            "",
            "    def __repr__(self) -> str:",
            "        \"\"\"Returns values of the segment.\"\"\"",
            "        anss = []",
            "        for i in range(self._l, self._r):",
            "            anss.append(str(self.query(i, i + 1)))",
            "        return \"seg: \" + \" \".join(anss)",
            "",
            "    @property",
            "    def _mid_h(self) -> int:",
            "        \"\"\"Returns the midpoint of the segment.\"\"\"",
            "        return self._l + self._r >> 1",
            "",
            "    def _create_subtrees(self):",
            "        \"\"\"Creates left and right subtrees if they do not exist.\"\"\"",
            "        midh = self._mid_h",
            "        if not self._left and midh > self._l:",
            "            self._left = Std.SegTree(self._f1, self._f2, self._l, midh, self._default)",
            "        if not self._right:",
            "            self._right = Std.SegTree(self._f1, self._f2, midh, self._r, self._default)",
            "",
            "    def build(self, arr: List[int]) -> int:",
            "        \"\"\"",
            "        Initializes the segment tree with values from arr.",
            "",
            "        Args:",
            "            arr: List of values to initialize the segment tree.",
            "",
            "        Returns:",
            "            The combined value of the segment tree.",
            "        \"\"\"",
            "        m0 = arr[0]",
            "        self._lazy_tag = 0",
            "        if self._r == self._l + 1:",
            "            self._v = m0",
            "            self._ans = self._f2(m0, len(arr))",
            "            return self._ans",
            "        self._v = '#'",
            "        midh = self._mid_h",
            "        self._create_subtrees()",
            "        self._ans = self._f1(self._left.build(arr[:midh - self._l]), self._right.build(arr[midh -
self._l:]))",
            "        return self._ans",
            "",
            "    def cover_seg(self, l: int, r: int, v: int) -> int:",
            "        \"\"\",
```

```
            "        Covers the segment [left, right) with value v.",
            "",
            "        Args:",
            "            l (int): Left boundary of the cover range.",
            "            r (int): Right boundary of the cover range.",
            "            v: Value to cover the segment with.",
            "",
            "        Returns:",
            "            The combined value of the segment tree.",
            "        \"\"\"",
            "        if self._v == v or l >= self._r or r <= self._l:",
            "            return self._ans",
            "        if l <= self._l and r >= self._r:",
            "            self._v = v",
            "            self._lazy_tag = 0",
            "            self._ans = self._f2(v, self._r - self._l)",
            "            return self._ans",
            "        self._create_subtrees()",
            "        if self._v != '#':",
            "            self._left._v = self._v",
            "            self._left._ans = self._f2(self._v, self._left._r - self._left._l)",
            "            self._right._v = self._v",
            "            self._right._ans = self._f2(self._v, self._right._r - self._right._l)",
            "            self._v = '#'",
            "        # push up",
            "        self._ans = self._f1(self._left.cover_seg(l, r, v), self._right.cover_seg(l, r, v))",
            "        return self._ans",
            "",
            "    def inc_seg(self, l: int, r: int, v: int) -> int:",
            "        \"\"\"",
            "        Increases the segment [left, right) by value v.",
            "",
            "        Args:",
            "            l (int): Left boundary of the increase range.",
            "            r (int): Right boundary of the increase range.",
            "            v: Value to increase the segment by.",
            "",
            "        Returns:",
            "            The combined value of the segment tree.",
            "        \"\"\"",
            "        if v == 0 or l >= self._r or r <= self._l:",
            "            return self._ans",
            "        if l <= self._l and r >= self._r:",
            "            if self._v == '#':",
            "                self._lazy_tag += v",
            "            else:",
            "                self._v += v",
            "            self._ans += self._f2(v, self._r - self._l)",
            "            return self._ans",
            "        self._create_subtrees()",
            "        if self._v != '#':",
            "            self._left._v = self._v",
            "            self._left._ans = self._f2(self._v, self._left._r - self._left._l)",
            "            self._right._v = self._v",
            "            self._right._ans = self._f2(self._v, self._right._r - self._right._l)",
            "            self._v = '#'",
            "        self._pushdown()",
            "        # push up",
            "        self._ans = self._f1(self._left.inc_seg(l, r, v), self._right.inc_seg(l, r, v))",
            "        return self._ans",
            "",
            "    def inc_idx(self, idx: int, v: int) -> int:",
            "        \"\"\"",
            "        Increases the value at index idx by value v.",
            "",
            "        Args:",
            "            idx (int): Index to increase.",
            "            v: Value to increase by.",
            "",
            "        Returns:",
            "            The combined value of the segment tree.",
            "        \"\"\"",
            "        if v == 0 or idx >= self._r or idx < self._l:",
            "            return self._ans",
            "        if idx == self._l == self._r - 1:",
            "            self._v += v",
            "            self._ans += self._f2(v, 1)",
            "            return self._ans",
```

```
            "        self._create_subtrees()",
            "        if self._v != '#':",
            "            self._left._v = self._v",
            "            self._left._ans = self._f2(self._v, self._left._r - self._left._l)",
            "            self._right._v = self._v",
            "            self._right._ans = self._f2(self._v, self._right._r - self._right._l)",
            "            self._v = '#'",
            "        self._pushdown()",
            "        # push up",
            "        self._ans = self._f1(self._left.inc_idx(idx, v), self._right.inc_idx(idx, v))",
            "        return self._ans",
            "",
            "    def _pushdown(self):",
            "        \"\"\"Propagates the lazy tag to the child nodes.\"\"\"",
            "        if self._lazy_tag != 0:",
            "            if self._left._v != '#':",
            "                self._left._v += self._lazy_tag",
            "            else:",
            "                self._left._lazy_tag += self._lazy_tag",
            "            self._left._ans += self._f2(self._lazy_tag, self._left._r - self._left._l)",
            "            if self._right._v != '#':",
            "                self._right._v += self._lazy_tag",
            "            else:",
            "                self._right._lazy_tag += self._lazy_tag",
            "            self._right._ans += self._f2(self._lazy_tag, self._right._r - self._right._l)",
            "            self._lazy_tag = 0",
            "",
            "    def query(self, l: int, r: int) -> int:",
            "        \"\"\"",
            "        Queries the range [left, right) for the combined value.",
            "",
            "        Args:",
            "            l (int): Left boundary of the query range.",
            "            r (int): Right boundary of the query range.",
            "",
            "        Returns:",
            "            The combined value of the range.",
            "        \"\"\"",
            "        if l >= r:",
            "            return 0",
            "        if l <= self._l and r >= self._r:",
            "            return self._ans",
            "        if self._v != '#':",
            "            return self._f2(self._v, Math.min(self._r, r) - Math.max(self._l, l))  # the overlapping
length",
            "        self._create_subtrees()",
            "        midh = self._mid_h",
            "        self._pushdown()",
            "        ans_ = []",
            "        if l < midh:",
            "            ans_.append(self._left.query(l, r))",
            "        if r > midh:",
            "            ans_.append(self._right.query(l, r))",
            "        return reduce(self._f1, ans_)",
            "",
            "    @staticmethod",
            "    def discretize(array):",
            "        \"\"\"Discretize the array and return the mapping dictionary. Index starts from 1\"\"\"",
            "        sorted_unique = sorted(set(array))",
            "        mapping = {val: idx + 1 for idx, val in enumerate(sorted_unique)}",
            "        return [mapping[val] for val in array], mapping"
        ],
        "description": "SegTreeDynamic"
    },
    "SegTreePoint": {
        "prefix": "SegTreePoint",
        "body": [
            "class SegTreePoint:",
            "    def __init__(self, n: int, arr=None):",
            "        self.n = n",
            "        self.val = Arr.array(0, n * 4)",
            "        self._build(1, 1, n, arr or Arr.array(0, n + 1))",
            "",
            "    def _push_up(self, idx: int):",
            "        self.val[idx] = self.val[idx * 2] + self.val[idx * 2 + 1]",
            "",
            "    def _build(self, idx: int, l: int, r: int, a):",
            "        if l == r:",
```

```
            "            self.val[idx] = a[l]",
            "            return",
            "        mid = (l + r) >> 1",
            "        self._build(idx * 2, l, mid, a)",
            "        self._build(idx * 2 + 1, mid + 1, r, a)",
            "        self._push_up(idx)",
            "",
            "    def _update(self, idx: int, l: int, r: int, pos: int, delta: int):",
            "        if l == r:",
            "            self.val[idx] += delta",
            "            return",
            "        mid = (l + r) >> 1",
            "        if pos <= mid:",
            "            self._update(idx * 2, l, mid, pos, delta)",
            "        else:",
            "            self._update(idx * 2 + 1, mid + 1, r, pos, delta)",
            "        self._push_up(idx)",
            "",
            "    def _query(self, idx: int, l: int, r: int, L: int, R: int) -> int:",
            "        if L <= l and r <= R:",
            "            return self.val[idx]",
            "        if R < l or r < L:",
            "            return 0",
            "        mid = (l + r) >> 1",
            "        return (",
            "            self._query(idx * 2, l, mid, L, R)",
            "            + self._query(idx * 2 + 1, mid + 1, r, L, R)",
            "        )",
            "",
            "    def add(self, pos: int, delta: int):",
            "        self._update(1, 1, self.n, pos, delta)",
            "",
            "    def query(self, L: int, R: int) -> int:",
            "        return self._query(1, 1, self.n, L, R)",
            "",
            "    def find_kth(self, k: int) -> int:",
            "        if self.val[1] < k:",
            "            return -1",
            "",
            "        idx, l, r = 1, 1, self.n",
            "        while l < r:",
            "            mid = (l + r) >> 1",
            "            left_sum = self.val[idx * 2]",
            "            if left_sum >= k:",
            "                idx, r = idx * 2, mid",
            "            else:",
            "                k -= left_sum",
            "                idx, l = idx * 2 + 1, mid + 1",
            "        return l",
            "",
            "    @staticmethod",
            "    def discretize(arr):",
            "        sorted_unique = sorted(set(arr))",
            "        mapping = {val: idx + 1 for idx, val in enumerate(sorted_unique)}",
            "        buckets = [mapping[val] for val in arr]",
            "        return buckets, mapping"
        ],
        "description": "SegTreePoint"
    },
    "SegTreeRange": {
        "prefix": "SegTreeRange",
        "body": [
            "class SegTreeRange:",
            "    def __init__(self, n: int, arr=None):",
            "        self.n = n",
            "        self.val = Arr.array(0, n * 4)",
            "        self.tag = Arr.array(0, n * 4)",
            "        self._build(1, 1, n, arr or Arr.array(0, n + 1))",
            "",
            "    def _apply(self, idx: int, l: int, r: int, delta: int):",
            "        self.val[idx] += delta * (r - l + 1)",
            "        self.tag[idx] += delta",
            "",
            "    def _push_up(self, idx: int):",
            "        self.val[idx] = self.val[idx * 2] + self.val[idx * 2 + 1]",
            "",
            "    def _push_down(self, idx: int, l: int, r: int):",
            "        if self.tag[idx] == 0:",
```

```
            "            return",
            "        mid = (l + r) >> 1",
            "        self._apply(idx * 2, l, mid, self.tag[idx])",
            "        self._apply(idx * 2 + 1, mid + 1, r, self.tag[idx])",
            "        self.tag[idx] = 0",
            "",
            "    def _build(self, idx: int, l: int, r: int, a):",
            "        if l == r:",
            "            self.val[idx] = a[l]",
            "            return",
            "        mid = (l + r) >> 1",
            "        self._build(idx * 2, l, mid, a)",
            "        self._build(idx * 2 + 1, mid + 1, r, a)",
            "        self._push_up(idx)",
            "",
            "    def _range_add(self, idx: int, l: int, r: int, L: int, R: int, delta: int):",
            "        if R < l or L > r:",
            "            return",
            "        if L <= l and r <= R:",
            "            self._apply(idx, l, r, delta)",
            "            return",
            "        self._push_down(idx, l, r)",
            "        mid = (l + r) >> 1",
            "        self._range_add(idx * 2, l, mid, L, R, delta)",
            "        self._range_add(idx * 2 + 1, mid + 1, r, L, R, delta)",
            "        self._push_up(idx)",
            "",
            "    def _range_sum(self, idx: int, l: int, r: int, L: int, R: int) -> int:",
            "        if R < l or L > r:",
            "            return 0",
            "        if L <= l and r <= R:",
            "            return self.val[idx]",
            "        self._push_down(idx, l, r)",
            "        mid = (l + r) >> 1",
            "        return (self._range_sum(idx * 2, l, mid, L, R)",
            "                + self._range_sum(idx * 2 + 1, mid + 1, r, L, R))",
            "",
            "    def add(self, L: int, R: int, delta: int):",
            "        self._range_add(1, 1, self.n, L, R, delta)",
            "",
            "    def query(self, L: int, R: int) -> int:",
            "        return self._range_sum(1, 1, self.n, L, R)",
            "",
            "    @staticmethod",
            "    def discretize(arr):",
            "        sorted_unique = sorted(set(arr))",
            "        mapping = {val: idx + 1 for idx, val in enumerate(sorted_unique)}",
            "        buckets = [mapping[val] for val in arr]",
            "        return buckets, mapping"
        ],
        "description": "SegTreeRange"
    },
    "KMP": {
        "prefix": "KMP",
        "body": [
            "class KMP:",
            "    def __init__(self, p: str):",
            "        self.p = p",
            "        self.m = len(p)",
            "        self.next = self._build_next(p)",
            "",
            "    def _build_next(self, p: str) -> List[int]:",
            "        nxt = Arr.array(0, self.m)",
            "        j = 0",
            "        for i in range(1, self.m):",
            "            while j > 0 and p[j] != p[i]:",
            "                j = nxt[j - 1]",
            "            if p[j] == p[i]:",
            "                j += 1",
            "            nxt[i] = j",
            "        return nxt",
            "",
            "    def search(self, t: str) -> List[int]:",
            "        res = []",
            "        j = 0",
            "        for i, ch in enumerate(t):",
            "            while j > 0 and self.p[j] != ch:",
            "                j = self.next[j - 1]",
```

```
            "            if self.p[j] == ch:",
            "                j += 1",
            "            if j == self.m:",
            "                res.append(i - self.m + 1)",
            "                j = self.next[j - 1]",
            "        return res"
        ],
        "description": "KMP"
    },
    "ZAlgorithm": {
        "prefix": "ZAlgorithm",
        "body": [
            "class ZAlgorithm:",
            "    def __init__(self, raw: str):",
            "        self.pattern = raw",
            "        self.n = len(raw)",
            "        self.z = Arr.array(0, self.n)",
            "        self._build_z()",
            "",
            "    def _build_z(self) -> None:",
            "        \"\"\"计算 pattern Z数组\"\"\"",
            "        if self.n == 0:",
            "            return",
            "        self.z[0] = self.n",
            "        c = r = 1",
            "        for i in range(1, self.n):",
            "            if i < r:",
            "                length = Math.min(self.z[i - c], r - i)",
            "            else:",
            "                length = 0",
            "            while (",
            "                i + length < self.n and",
            "                self.pattern[length] == self.pattern[i + length]",
            "            ):",
            "                length += 1",
            "            if i + length > r:",
            "                c, r = i, i + length",
            "            self.z[i] = length",
            "",
            "    def extend_array(self, text: str) -> List[int]:",
            "        \"\"\"E数组, 计算 pattern 在 text 中的匹配长度\"\"\"",
            "        n = len(text)  # text 长度",
            "        m = self.n  # pattern 长度",
            "        e = Arr.array(0, n)  # e[i] 表示 text[i:] 与 pattern 匹配长度",
            "        if m == 0 or n == 0:",
            "            return e",
            "",
            "        c = r = 0",
            "        for i in range(n):",
            "            if i < r:",
            "                length = Math.min(r - i, self.z[i - c])",
            "            else:",
            "                length = 0",
            "            while (",
            "                i + length < n and",
            "                length < m and",
            "                text[i + length] == self.pattern[length]",
            "            ):",
            "                length += 1",
            "            if i + length > r:",
            "                c, r = i, i + length",
            "            e[i] = length",
            "",
            "        return e"
        ],
        "description": "ZAlgorithm(EXKMP)"
    },
    "Factorial": {
        "prefix": "Factorial",
        "body": [
            "class Factorial:",
            "    def __init__(self, n, mod) -> None:",
            "        n += 1",
            "        self.mod = mod",
            "        self.f = Arr.array(1, n)",
            "        self.g = Arr.array(1, n)",
            "        for i in range(1, n):",
            "            self.f[i] = self.f[i - 1] * i % self.mod",
```

```
            "        self.g[-1] = pow(self.f[-1], mod - 2, mod)",
            "        for i in range(n - 2, -1, -1):",
            "            self.g[i] = self.g[i + 1] * (i + 1) % self.mod",
            "",
            "    def inv(self, x):",
            "        return pow(x, -1, self.mod)",
            "",
            "    def fac(self, n):",
            "        return self.f[n]",
            "",
            "    def fac_inv(self, n):",
            "        return self.g[n]",
            "",
            "    def combi(self, n, m):",
            "        if n < m or m < 0 or n < 0:",
            "            return 0",
            "        return self.f[n] * self.g[m] % self.mod * self.g[n - m] % self.mod",
            "",
            "    def permu(self, n, m):",
            "        if n < m or m < 0 or n < 0:",
            "            return 0",
            "        return self.f[n] * self.g[n - m] % self.mod",
            "",
            "    def catalan(self, n):",
            "        return (self.combi(2 * n, n) - self.combi(2 * n, n - 1)) % self.mod",
            "",
            "    def inv(self, n):",
            "        return self.f[n - 1] * self.g[n] % self.mod",
            "",
            "    def lucas(self, n, m):",
            "        if m == 0:",
            "            return 1",
            "        ni, mi = n % self.mod, m % self.mod",
            "        if mi > ni:",
            "            return 0",
            "        return self.combi(ni, mi) * self.lucas(n // self.mod, m // self.mod) % self.mod"
        ],
        "description": "Factorial"
    },
    "UnionFind": {
        "prefix": "UnionFind",
        "body": [
            "class UnionFind:",
            "    def __init__(self, n: int):",
            "        self.n = n",
            "        self.pa = list(range(n))",
            "        self.size = Arr.array(1, n)",
            "        self.comp_cnt = n",
            "",
            "    def _find(self, p: int) -> int:",
            "        if self.pa[p] != p:",
            "            self.pa[p] = self.find(self.pa[p])",
            "        return self.pa[p]",
            "",
            "    def find(self, p: int) -> int:",
            "        root = p",
            "        while self.pa[root] != root:",
            "            root = self.pa[root]",
            "        while self.pa[p] != root:",
            "            self.pa[p], p = root, self.pa[p]",
            "        return root",
            "",
            "    def union(self, p: int, q: int) -> int:",
            "        root_p = self.find(p)",
            "        root_q = self.find(q)",
            "        if root_p != root_q:",
            "            self.pa[root_p] = root_q",
            "            self.size[root_q] += self.size[root_p]",
            "            self.comp_cnt -= 1",
            "        return root_q"
        ],
        "description": "UnionFind"
    },
    "UnionFindWeight": {
        "prefix": "UnionFindWeight",
        "body": [
            "class UnionFindWeight:",
            "    def __init__(self, n: int):",
```

```
            "        self.n = n",
            "        self.pa = list(range(n))",
            "        self.weight = Arr.array(0, n)",
            "        self.size = Arr.array(1, n)",
            "",
            "    def find(self, p: int) → int:",
            "        if self.pa[p] ≠ p:",
            "            f = self.pa[p]",
            "            self.pa[p] = self.find(f)",
            "            self.weight[p] += self.weight[f]",
            "        return self.pa[p]",
            "",
            "    def union(self, head: int, tail: int, d: int) → bool:",
            "        \"\"\"w[tail] - w[head] = d\"\"\"",
            "        rhead, rtail = self.find(head), self.find(tail)",
            "        if rhead == rtail:",
            "            return (self.weight[tail] - self.weight[head]) == d",
            "        self.pa[rtail] = rhead",
            "        self.weight[rtail] = self.weight[head] + d - self.weight[tail]",
            "        self.size[rhead] += self.size[rtail]",
            "        return True",
            "",
            "    def diff(self, head: int, tail: int) → int:",
            "        rhead, rtail = self.find(head), self.find(tail)",
            "        if rhead ≠ rtail:",
            "            return INF",
            "        return self.weight[tail] - self.weight[head]"
        ],
        "description": "UnionFindWeight"
    },
    "UnionFindParity": {
        "prefix": "UnionFindParity",
        "body": [
            "class UnionFindParity:",
            "    def __init__(self, n: int):",
            "        self.n = n",
            "        self.pa = list(range(n))",
            "        self.pw = Arr.array(0, n)",
            "",
            "    def find(self, p: int) → int:",
            "        if self.pa[p] ≠ p:",
            "            f = self.pa[p]",
            "            self.pa[p] = self.find(f)",
            "            self.pw[p] ^= self.pw[f]",
            "        return self.pa[p]",
            "",
            "    def union(self, u: int, v: int, parity: int) → bool:",
            "        ru, rv = self.find(u), self.find(v)",
            "        if ru == rv:",
            "            return (self.pw[u] ^ self.pw[v]) == parity",
            "        self.pa[rv] = ru",
            "        self.pw[rv] = self.pw[u] ^ self.pw[v] ^ parity",
            "        return True",
            "",
            "    def diff(self, u: int, v: int) → int:",
            "        ru, rv = self.find(u), self.find(v)",
            "        if ru ≠ rv:",
            "            return INF",
            "        return self.pw[u] ^ self.pw[v]"
        ],
        "description": "UnionFindParity"
    },
    "KruskalMST": {
        "prefix": "KruskalMST",
        "body": [
            "class KruskalMST:",
            "    def __init__(self, size: int):",
            "        self.size = size",
            "        self.edges = []",
            "",
            "    def add_edge(self, u: int, v: int, w: int):",
            "        self.edges.append((w, u, v))",
            "",
            "    def kruskal(self) → Tuple:",
            "        self.edges.sort()",
            "        uf = Std.UnionFind(self.size)",
            "        mst = []",
            "        tot_w = 0",
```

```
            "",
            "        for w, u, v in self.edges:",
            "            if uf.find(u) ≠ uf.find(v):",
            "                uf.union(u, v)",
            "                mst.append((u, v, w))",
            "                tot_w += w",
            "",
            "        return mst, tot_w"
        ],
        "description": "KruskalMST"
    },
    "PrimMST": {
        "prefix": "PrimMST",
        "body": [
            "class PrimMST:",
            "    def __init__(self, n: int):",
            "        self.n = n",
            "        self.w = Arr.array2d(INF, n, n)",
            "",
            "    def add_edge(self, u: int, v: int, weight: int):",
            "        if weight < self.w[u][v]:",
            "            self.w[u][v] = weight",
            "            self.w[v][u] = weight",
            "",
            "    def prim(self, start: int = 0) → int:",
            "        visited = Arr.array(False, self.n)",
            "        dist = Arr.array(INF, self.n) # dist[v] 如果下一步要把 v 加入树，最便宜的代价",
            "        dist[start] = 0",
            "",
            "        total = 0",
            "        for _ in range(self.n):",
            "            u = -1",
            "            for v in range(self.n):",
            "                if not visited[v] and (u == -1 or dist[v] < dist[u]):",
            "                    u = v",
            "            if dist[u] == INF:",
            "                return INF",
            "",
            "            visited[u] = True",
            "            total += dist[u]",
            "",
            "            for v in range(self.n):",
            "                if (not visited[v]) and self.w[u][v] < dist[v]:",
            "                    dist[v] = self.w[u][v]",
            "",
            "        return total"
        ],
        "description": "PrimMST"
    },
    "BinaryTree": {
        "prefix": "BinaryTree",
        "body": [
            "class BinaryTree:",
            "    def __init__(self, n: int, op=lambda x, y: x + y, default=0, array=None):",
            "        self.n = n",
            "        self.op = op",
            "        self.default = default",
            "        self.tree = Arr.array(default, self.n + 1)",
            "",
            "        if array:",
            "            for i, value in enumerate(array, 1):",
            "                self.update(i, value)",
            "",
            "    def update(self, i: int, value: int):",
            "        while i ⩽ self.n:",
            "            self.tree[i] = self.op(self.tree[i], value)",
            "            i += i & -i",
            "",
            "    def query(self, i: int) → int:",
            "        res = self.default",
            "        while i > 0:",
            "            res = self.op(res, self.tree[i])",
            "            i -= i & -i",
            "        return res",
            "",
            "    def range_query(self, l: int, r: int) → int:",
            "        return self.query(r) - self.query(l - 1)",
            "",
```

```
            "    @staticmethod",
            "    def discretize(array):",
            "        sorted_unique = sorted(set(array))",
            "        mapping = {val: idx + 1 for idx, val in enumerate(sorted_unique)}",
            "        return [mapping[val] for val in array], mapping",
            "",
            "class DifferenceBinaryTree:",
            "    def __init__(self, n: int, array=None):",
            "        if array is None:",
            "            array = []",
            "        self._n = n",
            "        self._diff_tree = Std.BinaryTree(self._n, lambda x, y: x + y)",
            "        if array:",
            "            for i in range(1, self._n + 1):",
            "                delta = array[i - 1] - (array[i - 2] if i > 1 else 0)",
            "                self._diff_tree.update(i, delta)",
            "",
            "    def update_add(self, l: int, r: int, delta: int):",
            "        self._diff_tree.update(l, delta)",
            "        self._diff_tree.update(r + 1, -delta)",
            "",
            "    def query_value(self, i: int) -> int:",
            "        return self._diff_tree.query(i)"
        ],
        "description": "BinaryTree"
    },
    "Graph": {
        "prefix": "Graph",
        "body": [
            "class Graph:",
            "    def __init__(self, n: int):",
            "        self.n = n",
            "        self.g = Arr.graph(n)",
            "",
            "    def add_edge(self, u: int, v: int, w: int):",
            "        self.g[u].append((v, w))"
        ],
        "description": "Graph"
    },
    "Tree": {
        "prefix": "Tree",
        "body": [
            "class Tree:",
            "    def __init__(self, n: int):",
            "        self._n = n",
            "        self._g_ = Arr.graph(n)  # Create adjacency list",
            "        self.depth_ = Arr.array(0, n)  # Depth array for each node",
            "        self.size_ = Arr.array(1, n)  # Size of the subtree rooted at each node",
            "        self.dist_ = Arr.array(0, n)  # Distance sum of all nodes from the current node",
            "        self.dp_ = Arr.array(0, n)  # Longest path in subtree rooted at each node",
            "        self.diameter = 0  # Store the diameter of the tree",
            "",
            "    def add_edge(self, u: int, v: int, w: int):",
            "        \"\"\"Add an edge to the graph.\"\"\"",
            "        self._g_[u].append((v, w))",
            "",
            "    def dfs(self, u: int, fa: int):",
            "        \"\"\"",
            "        First DFS to calculate depth, size of subtrees, and initial distance sum (for the root).",
            "",
            "        Args:",
            "            u (int): Current node.",
            "            fa (int): Parent node of the current node.",
            "        \"\"\"",
            "        for v, w in self._g_[u]:",
            "            if fa == v:",
            "                continue",
            "            self.depth_[v] = self.depth_[u] + w  # Calculate depth",
            "            self.dfs(v, u)",
            "            self.size_[u] += self.size_[v]  # Calculate subtree size",
            "            self.dist_[u] += self.dist_[v] + self.size_[v] * w  # Update the root's initial distance sum",
            "            self.dp_[u] = Math.max(self.dp_[u], self.dp_[v] + w)  # Longest path in the subtree",
            "",
            "    def dfs_cal_dist(self, u: int, fa: int):",
            "        \"\"\"",
            "        DFS to calculate the distance sum for each node based on its parent's distance sum.",
            "",
            "        Args:",
```

```
            "            u (int): Current node.",
            "            fa (int): Parent node of the current node.",
            "        \"\"\"",
            "        for v, w in self._g_[u]:",
            "            if fa == v:",
            "                continue",
            "            # Calculate the distance sum for child node v using the parent's distance sum",
            "            self.dist_[v] = self.dist_[u] + self._n - 2 * self.size_[v]",
            "            self.dfs_cal_dist(v, u)",
            "",
            "    def dfs_cal_diameter(self, u: int, fa: int) -> int:",
            "        \"\"\"",
            "        DFS to calculate the diameter of the tree.",
            "",
            "        Args:",
            "            u (int): Current node.",
            "            fa (int): Parent node of the current node.",
            "        \"\"\"",
            "        max1, max2 = 0, 0",
            "        for v, w in self._g_[u]:",
            "            if fa == v:",
            "                continue",
            "            cand = self.dfs_cal_diameter(v, u) + w",
            "            if cand > max1:",
            "                max2 = max1",
            "                max1 = cand",
            "            elif cand > max2:",
            "                max2 = cand",
            "",
            "        self.diameter = Math.max(self.diameter, max1 + max2)  # Update the diameter",
            "        return max1"
        ],
        "description": "Tree"
    },
    "MoAlgorithm": {
        "prefix": "MoAlgorithm",
        "body": [
            "class MoAlgorithm:",
            "    Query = namedtuple('Query', ['l', 'r', 'id'])",
            "",
            "    def __init__(self, array, queries):",
            "        self.array: List[int] = array",
            "        self.queries = queries",
            "        self.n = len(array)  # Length of the input array.",
            "        self.m = len(queries)  # Number of queries.",
            "        self.size = int(sqrt(self.n))  # Size of each block.",
            "        self.ans = defaultdict(int)  # Stores the result for each query.",
            "        self.cnt = defaultdict(int)  # Frequency array for dynamic updates.",
            "        self.cur = 0  # Current result for the active range.",
            "",
            "    def add(self, num):",
            "        self.cur += self.cnt[self.x ^ num]",
            "        self.cnt[num] += 1",
            "",
            "    def rev(self, num):",
            "        self.cnt[num] -= 1",
            "        self.cur -= self.cnt[self.x ^ num]",
            "",
            "    def cmp(self, query):",
            "        block_id = query.l // self.size",
            "        return (block_id, query.r if block_id % 2 == 0 else -query.r)",
            "",
            "    def process(self, x):",
            "        self.x = x  # Store the XOR condition.",
            "",
            "        # Sort queries in Mo's order.",
            "        self.queries.sort(key=self.cmp)",
            "        l, r = 0, -1  # Active range [l, r].",
            "        for q in self.queries:",
            "            while l > q.l:",
            "                l -= 1",
            "                self.add(self.array[l])",
            "            while r < q.r:",
            "                r += 1",
            "                self.add(self.array[r])",
            "            while l < q.l:",
            "                self.rev(self.array[l])",
            "                l += 1",
```

```
            "            while r > q.r:",
            "                self.rev(self.array[r])",
            "                r -= 1",
            "            # Store the result for the query.",
            "            self.ans[q.id] = self.cur > 0",
            "",
            "        return self.ans"
        ],
        "description": "MoAlgorithm"
    },
    "init": {
        "prefix": "init",
        "body": [
            "# 3.8.6 import",
            "import bisect",
            "from collections import Counter, defaultdict, deque, namedtuple",
            "from datetime import datetime, timedelta",
            "from functools import lru_cache, reduce",
            "from heapq import heapify, heappop, heappush, heappushpop, heapreplace, nlargest, nsmallest",
            "from itertools import combinations, compress, permutations, groupby, accumulate",
            "from math import ceil, floor, fabs, gcd, log, exp, sqrt, hypot, inf, isqrt",
            "from string import ascii_lowercase, ascii_uppercase",
            "from bisect import bisect_left, bisect_right, insort",
            "from sys import exit, setrecursionlimit, stdin",
            "from typing import Any, Callable, Dict, List, Optional, Tuple, Deque",
            "from random import randint",
            "",
            "# Constants",
            "N = int(2e5 + 10)",
            "M = int(20)",
            "INF = int(1e12)",
            "OFFSET = int(100)",
            "MOD = int(1e9 + 7)",
            "",
            "# Set recursion limit",
            "setrecursionlimit(int(1e7))",
            "",
            "",
            "class Arr:",
            "    array = staticmethod(lambda x=0, size=N: [x() if callable(x) else x for _ in range(size)])",
            "    array2d = staticmethod(lambda x=0, rows=N, cols=M: [Arr.array(x, cols) for _ in range(rows)])",
            "    graph = staticmethod(lambda size=N: [[] for _ in range(size)])",
            "",
            "",
            "class Math:",
            "    max = staticmethod(lambda a, b: a if a > b else b)",
            "    min = staticmethod(lambda a, b: a if a < b else b)",
            "",
            "",
            "class IO:",
            "    input = staticmethod(lambda: stdin.readline().strip())",
            "    read = staticmethod(lambda: map(int, IO.input().split()))",
            "    read_list = staticmethod(lambda: list(IO.read()))",
            "    read_mixed = staticmethod(lambda *types: [t(v) for t, v in zip(types, IO.input().split())])",
            "",
            "",
            "class Std:",
            "    pass",
            "",
            "# ---------------------- Division line -----------------------",
            "",
            "",
            "def solve():",
            "    return",
            "",
            "",
            "if __name__ == \"__main__\":",
            "    solve()",
            ""
        ],
        "description": "init"
    },
    "Bitset": {
        "prefix": "Bitset",
        "body": [
            "class Bit(int):",
            "    def __new__(cls, value):",
            "        return super(Bit, cls).__new__(cls, value)",
```

```
            "",
            "    def __init__(self, value):",
            "        self.bin_rep = bin(value)[2:]",
            "",
            "    def bit_length(self): return len(self.bin_rep)",
            "    def bit_count(self): return self.bin_rep.count('1')",
            "    def lowest1(self): return Bit(self & -self)",
            "    def lowest0(self): return Bit(~self & (self + 1))",
            "    def clear_lowest1(self): return Bit(self & (self - 1))",
            "    def clear_lowest0(self): return Bit(self | (self + 1))",
            "    def get_bits(self, start, end): return self & Bit.range_mask(start, end)",
            "    def __or__(self, other): return Bit(super(Bit, self).__or__(other))",
            "    def __and__(self, other): return Bit(super(Bit, self).__and__(other))",
            "    def __sub__(self, other): return Bit(super(Bit, self).__and__(~other))",
            "    def __contains__(self, other): return self & other == other",
            "",
            "    all_ones_mask = staticmethod(lambda length: (1 << length) - 1)",
            "    all_zeros_mask = staticmethod(lambda length: 0)",
            "    single_bit_mask = staticmethod(lambda position: 1 << position)",
            "    range_mask = staticmethod(lambda start, end: ((1 << (end - start + 1)) - 1) << start)"
        ],
        "description": "Bitset"
    },
    "Manacher": {
        "prefix": "Manacher",
        "body": [
            "class Manacher:",
            "    \"\"\"Manacher\"\"\"",
            "",
            "    def __init__(self, raw: str):",
            "        self.raw = raw",
            "        self.ss = self._build_extended(raw)",
            "        self.n = len(self.ss)",
            "        self.p = Arr.array(0, self.n)  # 回文半径数组",
            "",
            "    def _build_extended(self, s: str) -> List[str]:",
            "        m = len(s)",
            "        ext_len = m * 2 + 1",
            "        ext = Arr.array('#', ext_len)",
            "        for j in range(m):",
            "            ext[2 * j + 1] = s[j]",
            "        return ext",
            "",
            "    def longest_pal_length(self) -> int:",
            "        \"\"\"",
            "        返回原串中的最长回文子串长度（原串下标意义）",
            "        \"\"\"",
            "        if not self.raw:",
            "            return 0",
            "",
            "        c = 0   # 当前最右回文的中心位置",
            "        r = 0   # 当前回文覆盖的右边界（开区间）",
            "        ans = 0",
            "        for i in range(self.n):",
            "            if i < r:",
            "                mirror = 2 * c - i",
            "                length = Math.min(self.p[mirror], r - i)",
            "            else:",
            "                length = 1",
            "            while (",
            "                i + length < self.n and",
            "                i - length >= 0 and",
            "                self.ss[i + length] == self.ss[i - length]",
            "            ):",
            "                length += 1",
            "            if i + length > r:",
            "                r = i + length",
            "                c = i",
            "            self.p[i] = length",
            "            ans = Math.max(ans, length)",
            "",
            "        return ans - 1  # 回文长度 = 半径 - 1"
        ],
        "description": "Manacher"
    },
    "DataGenerator": {
        "prefix": "DataGenerator",
        "body": [
```

```
            "import random",
            "from itertools import combinations",
            "",
            "",
            "class DataGenerator:",
            "    def __init__(self, seed=None):",
            "        if seed is not None:",
            "            random.seed(seed)",
            "        self.lines = []          # 存储最终要写入文件的所有行(字符串)",
            "        self._cache = []         # 当前 case 正在构造的行",
            "",
            "    def gen_num(self, lo, hi):",
            "        \"\"\"生成一个范围内的随机整数, 不计入当前 case!!\"\"\"",
            "        return random.randint(lo, hi)",
            "",
            "    def gen_values(self, ranges):",
            "        vals = [self.gen_num(l, r) for l, r in ranges]",
            "        self.add_line(*vals)",
            "        return vals",
            "",
            "    def gen_arr(self, length, lo, hi):",
            "        arr = [self.gen_num(lo, hi) for _ in range(length)]",
            "        self.add_line(*arr)",
            "        return arr",
            "",
            "    def add_line(self, *items):",
            "        \"\"\"把任意数量的数字/列表/字符串拼成一行, 加入当前 case\"\"\"",
            "        flat = []",
            "        for x in items:",
            "            if isinstance(x, (list, tuple)):",
            "                flat.extend(map(str, x))",
            "            else:",
            "                flat.append(str(x))",
            "        self._cache.append(\" \".join(flat))",
            "",
            "    def new_case(self):",
            "        \"\"\"结束当前 case, 把缓存写入总列表, 并插入一个空行分隔\"\"\"",
            "        self.lines.extend(self._cache)",
            "        self.lines.append(\"\")",
            "        self._cache = []",
            "",
            "    def gen_tree(self, n, weight=False, w_lo=1, w_hi=1):",
            "        \"\"\"生成一棵随机树 (无向), 自动输出 n-1 行边。\"\"\"",
            "        parents = list(range(2, n + 1))",
            "        random.shuffle(parents)",
            "        for child in range(2, n + 1):",
            "            parent = random.randint(1, child - 1)",
            "            if weight:",
            "                w = self.gen_num(w_lo, w_hi)",
            "                self.add_line(parent, child, w)",
            "            else:",
            "                self.add_line(parent, child)",
            "",
            "    def gen_graph(self, n, m, connected=True, weight=False, w_lo=1, w_hi=1):",
            "        \"\"\"",
            "        生成一个无向简单图。",
            "        connected=True 保证连通; 否则纯随机 m 条无重复边/自环。",
            "        \"\"\"",
            "        if connected and m < n - 1:",
            "            raise ValueError(\"连通图最少需要 n-1 条边\")",
            "        edges = set()",
            "        if connected:",
            "            for child in range(2, n + 1):",
            "                parent = random.randint(1, child - 1)",
            "                edges.add((parent, child))",
            "        all_pairs = list(combinations(range(1, n + 1), 2))",
            "        random.shuffle(all_pairs)",
            "        for u, v in all_pairs:",
            "            if len(edges) >= m:",
            "                break",
            "            if (u, v) not in edges:",
            "                edges.add((u, v))",
            "        for u, v in edges:",
            "            if weight:",
            "                w = self.gen_num(w_lo, w_hi)",
            "                self.add_line(u, v, w)",
            "            else:",
            "                self.add_line(u, v)",
```

```
            "",
            "    def save(self, filename):",
            "        self.new_case()",
            "        with open(filename, \"w\") as f:",
            "            f.write(\"\\n\".join(self.lines).strip())",
            "",
            "",
            "if __name__ == \"__main__\":",
            "    dg = DataGenerator()",
            "",
            "    n, = dg.gen_values([(1, 10)])",
            "    arr = dg.gen_arr(n, 1, 10)",
            "    dg.save(\"input.txt\")",
            ""
        ],
        "description": "DataGenerator"
    },
    "Duipai": {
        "prefix": "Duipai",
        "body": [
            "import filecmp",
            "import subprocess",
            "",
            "",
            "def run_program(program, input_file, output_file):",
            "    with open(input_file, 'r') as infile, open(output_file, 'w') as outfile:",
            "        subprocess.run(['python', program], stdin=infile, stdout=outfile)",
            "",
            "",
            "def compare_outputs(output_a, output_b):",
            "    return filecmp.cmp(output_a, output_b, shallow=False)",
            "",
            "",
            "def main():",
            "    input_file = \"input.txt\"",
            "    output_force = \"output_force.txt\"",
            "    output_std = \"output_std.txt\"",
            "",
            "    for i in range(100):",
            "        print(f\"Running test {i + 1}...\")",
            "        subprocess.run(['python', 'input_gen.py'])",
            "",
            "        run_program('force.py', input_file, output_force)",
            "        run_program('std.py', input_file, output_std)",
            "",
            "        if not compare_outputs(output_force, output_std):",
            "            print(\"Discrepancy found!\")",
            "            with open(\"discrepancy_input.txt\", 'w') as f:",
            "                with open(input_file, 'r') as infile:",
            "                    f.write(infile.read())",
            "",
            "            with open(\"discrepancy_output_force.txt\", 'w') as f:",
            "                with open(output_force, 'r') as outfile_force:",
            "                    f.write(outfile_force.read())",
            "",
            "            with open(\"discrepancy_output_std.txt\", 'w') as f:",
            "                with open(output_std, 'r') as outfile_std:",
            "                    f.write(outfile_std.read())",
            "",
            "            break",
            "    else:",
            "        print(\"All tests passed!\")",
            "",
            "",
            "if __name__ == \"__main__\":",
            "    main()",
            ""
        ],
        "description": "Duipai"
    },
    "PrimeEraSieve": {
        "prefix": "PrimeEraSieve",
        "body": [
            "class EratosthenesSieve:",
            "    def __init__(self, n: int):",
            "        self.n = n",
            "        self.is_prime = Arr.array(True, n + 1)",
            "        self.primes = []",
```

```
            "        self._sieve()",
            "",
            "    def _sieve(self):",
            "        self.is_prime[0] = self.is_prime[1] = False",
            "        for i in range(2, self.n + 1):",
            "            if self.is_prime[i]:",
            "                self.primes.append(i)",
            "                if i * i > self.n:",
            "                    continue",
            "                for j in range(i * i, self.n + 1, i): # 优化: 不从 2 * i 开始",
            "                    self.is_prime[j] = False"
        ],
        "description": "PrimeEraSieve"
    },
    "PrimeSieve": {
        "prefix": "PrimeSieve",
        "body": [
            "class PrimeSieve:",
            "    def __init__(self, n: int):",
            "        self.n = n",
            "        self.is_prime = Arr.array(True, n + 1)",
            "        self.d_cnt = Arr.array(1, n + 1)  # divisior count",
            "        self.d_sum = Arr.array(1, n + 1)  # divisior sum",
            "        self.num1 = Arr.array(0, n + 1)  # Array to store the count of the smallest prime factor (min_prime_factor_count)",
            "        self.num2 = Arr.array(0, n + 1)  # Array to store the sum of the smallest prime factor's divisior sum",
            "        self.primes = []",
            "        self._generate_sieve()",
            "",
            "    def _generate_sieve(self):",
            "        \"\"\"",
            "        Generates the sieve using the linear sieve algorithm.",
            "        Computes primes, divisor counts, and divisor sums.",
            "        \"\"\"",
            "        self.is_prime[0] = self.is_prime[1] = False",
            "        self.d_cnt[1] = 1",
            "        self.d_sum[1] = self.num2[1] = 1",
            "",
            "        for i in range(2, self.n + 1):",
            "            if self.is_prime[i]:",
            "                self.primes.append(i)",
            "                self.num1[i], self.d_cnt[i] = 1, 2",
            "                self.num2[i] = self.d_sum[i] = i + 1",
            "            for p in self.primes:",
            "                m = i * p",
            "                if m > self.n:",
            "                    break",
            "                self.is_prime[m] = False",
            "                if i % p == 0:  # This ensures that each composite number is only marked by its smallest prime factor.",
            "                    self.num1[m] = self.num1[i] + 1",
            "                    self.d_cnt[m] = self.d_cnt[i] // self.num1[m] * (self.num1[m] + 1)",
            "                    self.num2[m] = self.num2[i] * p + 1",
            "                    self.d_sum[m] = self.d_sum[i] // self.num2[i] * self.num2[m]",
            "                    break",
            "                else:",
            "                    self.num1[m] = 1",
            "                    self.d_cnt[m] = self.d_cnt[i] * 2",
            "                    self.num2[m] = p + 1",
            "                    self.d_sum[m] = self.d_sum[i] * self.num2[m]"
        ],
        "description": "PrimeSieve"
    },
    "MIN_25": {
        "prefix": "MIN_25",
        "body": [
            "class PrimePhiPrefix:",
            "    \"\"\"",
            "    求: ",
            "    1) sum_primes() = Σ_{p≤n} p",
            "    2) sum_phi()    = Σ_{i=1}^{n} φ(i)",
            "    用  ➜  Min 25 (质数和)  +  杜教筛 (φ 前缀和)",
            "    \"\"\"",
            "",
            "    def __init__(self, n: int):",
            "        if n < 1:",
            "            raise ValueError(\"n 必须 ≥ 1\")",
```

```
        "        self.n = n",
        "        self._build_small_tables()        # 线筛 & 前缀",
        "        self._build_min25_prime_sum()     # Min 25 求质数前缀和",
        "",
        "    def sum_primes(self) -> int:",
        "        \"\"\"返回 Σ_{p≤n} p\"\"\"",
        "        return self._prime_sum_ans",
        "",
        "    def sum_phi(self) -> int:",
        "        \"\"\"返回 Σ_{i=1}^{n} φ(i)\"\"\"",
        "        return self._S_phi(self.n)",
        "",
        "    # ---------- 预处理: 线性筛 φ & 素数 ----------",
        "    def _build_small_tables(self) -> None:",
        "        n = self.n",
        "        # √n 用于 Min25; n^{2/3} 用于杜教筛缓存下界",
        "        self.lim = max(isqrt(n) + 2, int(n ** (2 / 3)) + 10)",
        "        lim = self.lim",
        "",
        "        is_comp = [False] * lim",
        "        phi = [0] * lim",
        "        phi[1] = 1",
        "        primes = []",
        "",
        "        for i in range(2, lim):",
        "            if not is_comp[i]:",
        "                primes.append(i)",
        "                phi[i] = i - 1",
        "            for p in primes:",
        "                if i * p >= lim:",
        "                    break",
        "                is_comp[i * p] = True",
        "                if i % p == 0:",
        "                    phi[i * p] = phi[i] * p",
        "                    break",
        "                phi[i * p] = phi[i] * (p - 1)",
        "",
        "        # 前缀和: prime-sum / φ-sum",
        "        pre_psum, pre_phi = [0] * lim, [0] * lim",
        "        ps, ph = 0, 0",
        "        for i in range(1, lim):",
        "            if i >= 2 and not is_comp[i]:",
        "                ps += i",
        "            ph += phi[i]",
        "            pre_psum[i] = ps",
        "            pre_phi[i] = ph",
        "",
        "        self.primes = primes",
        "        self.pre_psum = pre_psum",
        "        self.pre_phi = pre_phi",
        "        self._phi_cache = {}              # 杜教筛记忆化",
        "",
        "    # ---------- Min 25: 质数前缀和 ----------",
        "    def _build_min25_prime_sum(self) -> None:",
        "        n, primes, pre_psum = self.n, self.primes, self.pre_psum",
        "",
        "        # 分块: w = n // i 去重",
        "        w, idx, i = [], {}, 1",
        "        while i <= n:",
        "            v = n // i",
        "            w.append(v)",
        "            idx[v] = len(w) - 1",
        "            i = n // v + 1",
        "        self._w, self._idx = w, idx",
        "",
        "        g = [v * (v + 1) // 2 - 1 for v in w]   # g[v] 初值",
        "",
        "        for p in primes:",
        "            p2 = p * p",
        "            if p2 > n:",
        "                break",
        "            gp1 = g[idx[p - 1]]                 # Σ_prime≤p-1 prime",
        "            for j, v in enumerate(w):",
        "                if v < p2:",
        "                    break",
        "                tv = v // p",
        "                val = g[idx[tv]] if tv in idx else pre_psum[tv]",
        "                g[j] -= p * (val - gp1)",
```

```
            "",
            "        self._prime_sum_ans = g[idx[n]]",
            "",
            "    # ---------- 杜教筛: φ 前缀和 ----------",
            "    @lru_cache(maxsize=None)",
            "    def _S_phi(self, x: int) -> int:",
            "        if x < self.lim:",
            "            return self.pre_phi[x]",
            "        if x in self._phi_cache:",
            "            return self._phi_cache[x]",
            "",
            "        res, i = x * (x + 1) // 2, 2",
            "        while i <= x:",
            "            v = x // i",
            "            j = x // v",
            "            res -= (j - i + 1) * self._S_phi(v)",
            "            i = j + 1",
            "",
            "        self._phi_cache[x] = res",
            "        return res"
        ],
        "description": "MIN_25"
    },
    "EulerReduct": {
        "prefix": "EulerReduct",
        "body": [
            "class EulerReduct:",
            "    \"\"\"Euler power reduction\"\"\"",
            "",
            "    def __init__(self):",
            "        pass",
            "",
            "    @lru_cache(maxsize=None)",
            "    def phi(self, x: int) -> int:",
            "        res, p = x, 2",
            "        while p * p <= x:",
            "            if x % p == 0:",
            "                while x % p == 0:",
            "                    x //= p",
            "                res -= res // p",
            "            p += 1",
            "        if x > 1:",
            "            res -= res // x",
            "        return res",
            "",
            "    def safe_pow(self, a: int, b: int, n: int) -> int:",
            "        exp = b % self.phi(n) + self.phi(n)",
            "        return pow(a % n, exp, n)",
            "",
            "    def tower(self, bases: List[int], n: int) -> int:",
            "        \"\"\"a0^(a1^(a2^( ... ))) mod n\"\"\"",
            "        if len(bases) == 1:",
            "            return bases[0] % n",
            "        lower = self.tower(bases[1:], self.phi(n))",
            "        return self.safe_pow(bases[0], lower, n)"
        ],
        "description": "EulerReduct"
    },
    "Backpack": {
        "prefix": "Backpack",
        "body": [
            "class Backpack:",
            "    def __init__(self, capacity: int):",
            "        self.C = capacity",
            "        self.items: List[Tuple[int, int]] = []   # [(w, v), ...]  after split",
            "",
            "    def add_item(self, weight: int, value: int, count: int = -1):",
            "        \"\"\"",
            "        count =  1  → 01",
            "        count = -1  → completely knapsack",
            "        count = k>1 → multiple knapsack (upper limit k)",
            "        \"\"\"",
            "        if count == 1:",
            "            self.items.append((weight, value))",
            "        elif count == -1:",
            "            k = 1",
            "            while k * weight <= self.C:",
            "                self.items.append((k * weight, k * value))",
```

```
            "                k ⟸ 1",
            "        else:",
            "            k = 1",
            "            while count ⩾ k:",
            "                self.items.append((k * weight, k * value))",
            "                count -= k",
            "                k ⟸ 1",
            "            if count:",
            "                self.items.append((count * weight, count * value))",
            "",
            "    def solve(self) → int:",
            "        dp = [0] * (self.C + 1)",
            "        for w, v in self.items:",
            "            for c in range(self.C, -1, -1):",
            "                if c ⩾ w:",
            "                    dp[c] = Math.max(dp[c], dp[c - w] + v)",
            "        return dp[self.C]",
            "",
            "    def solve_exist(self) → bool:",
            "        dp = [False] * (self.C + 1)  # 只用于 01 背包存在性",
            "        dp[0] = True",
            "        for w, _ in self.items:",
            "            for c in range(self.C, -1, -1):",
            "                if c ⩾ w:",
            "                    dp[c] = dp[c] | dp[c - w]",
            "        return dp[self.C]"
        ],
        "description": "Backpack"
    },
    "DebugDFS": {
        "prefix": "DebugDFS",
        "body": [
            "@lru_cache(None)",
            "def dfs(s1, s2, s3, depth=0):",
            "    if depth > MAX_DEPTH:",
            "        print(\"depth too deep:\", trace[-5:])",
            "        return 0",
            "",
            "    if max(s1, s2, s3) > MAX_STATE:",
            "        return 0",
            "",
            "    trace.append((s1, s2, s3))",
            "",
            "    # ➤ TODO: base case",
            "    # if s1 == 0 and s2 == 0 and s3 == 0:",
            "    #     print(\"合法路径:\", trace[-10:])",
            "    #     trace.pop()",
            "    #     return 1",
            "",
            "    res = 0",
            "    # ➤ TODO: 状态转移",
            "    # if condition:",
            "    #     res += dfs( ... )",
            "",
            "    trace.pop()",
            "    return res",
            "",
            "# print(dfs( ... ))",
            "# print(dfs.cache_info())",
            "# dfs.cache_clear()"
        ],
        "description": "DebugDFS"
    },
    "TimeUtils": {
        "prefix": "TimeUtils",
        "body": [
            "class TimeUtils:",
            "    @staticmethod",
            "    def to_seconds(h: int, m: int, s: int) → int:",
            "        \"\"\"h:m:s → total seconds\"\"\"",
            "        return h * 3600 + m * 60 + s",
            "",
            "    @staticmethod",
            "    def from_seconds(sec: int) → Tuple[int, int, int]:",
            "        \"\"\"total seconds → (h, m, s)\"\"\"",
            "        sec %= 24 * 3600",
            "        h, sec = divmod(sec, 3600)",
            "        m, s = divmod(sec, 60)",
```

```
            "        return h, m, s",
            "",
            "    @staticmethod",
            "    def is_leap(year: int) -> bool:",
            "        \"\"\"Gregorian leap-year rule\"\"\"",
            "        return year % 400 == 0 or (year % 4 == 0 and year % 100 != 0)",
            "",
            "    _DAYS_IN_MONTH = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]",
            "",
            "    @classmethod",
            "    def valid_date(cls, y: int, m: int, d: int) -> bool:",
            "        \"\"\"YYYY-MM-DD Is it legal\"\"\"",
            "        if not 1 <= m <= 12:",
            "            return False",
            "        days = cls._DAYS_IN_MONTH[m - 1]",
            "        if m == 2 and cls.is_leap(y):",
            "            days += 1",
            "        return 1 <= d <= days"
        ],
        "description": "TimeUtils"
    },
    "DigitDP": {
        "prefix": "DigitDP",
        "body": [
            "def countDigitOne(n: int) -> int:",
            "    \"\"\"",
            "    数位DP 不考虑前缀0 版本",
            "    统计所有 `0 ≤ x ≤ n` 的整数中，数字 `1` 出现的总次数。",
            "    \"\"\"",
            "    m = len(str(n))",
            "    high_s = list(map(int, str(n)))",
            "    low_s = list(map(int, str(0).zfill(m)))",
            "",
            "    @lru_cache(None)",
            "    def dfs(i: int, cnt1: int, limit_low: bool, limit_high: bool) -> int:",
            "        if i == m:",
            "            return cnt1",
            "",
            "        res = 0",
            "        lo = low_s[i] if limit_low else 0",
            "        hi = high_s[i] if limit_high else 9",
            "",
            "        for d in range(lo, hi + 1):",
            "            res += dfs(",
            "                i + 1,",
            "                cnt1 + (1 if d == 1 else 0),",
            "                limit_low and d == lo,",
            "                limit_high and d == hi",
            "            )",
            "        return res",
            "",
            "    return dfs(0, 0, True, True)"
        ],
        "description": "DigitDP"
    },
    "DigitDPPre0": {
        "prefix": "DigitDPPre0",
        "body": [
            "def countSpecialNumbers(n: int) -> int:",
            "    \"\"\"",
            "    数位DP 前缀0 版本",
            "    统计区间 **[1, n]** 内的各数位互不相同的正整数",
            "    \"\"\"",
            "    m = len(str(n))",
            "    high_s = list(map(int, str(n)))",
            "    low_s = list(map(int, str(1).zfill(m)))",
            "",
            "    @lru_cache(None)",
            "    def dfs(i: int, mask: int, limit_low: bool, limit_high: bool, is_num: bool) -> int:",
            "        if i == m:",
            "            return int(is_num)",
            "",
            "        res = 0",
            "        if not is_num and low_s[i] == 0:",
            "            res += dfs(i + 1, mask, True, False, False)",
            "",
            "        lo = low_s[i] if limit_low else 0",
            "        hi = high_s[i] if limit_high else 9",
```

```
            "",
            "        d0 = 1 - int(is_num)",
            "        for d in range(Math.max(lo, d0), hi + 1):",
            "            if not (mask & 1 << d):",
            "                res += dfs(",
            "                    i + 1,",
            "                    mask | 1 << d,",
            "                    limit_low and d == lo,",
            "                    limit_high and d == hi,",
            "                    True",
            "                )",
            "",
            "        return res",
            "",
            "    return dfs(0, 0, True, True, False)"
        ],
        "description": "DigitDPPre0"
    },
    "Heap": {
        "prefix": "Heap",
        "body": [
            "class Heap:",
            "    def __init__(self, iterable, min_heap: bool = True):",
            "        self.is_min = min_heap",
            "        self.data = [self._encode(x) for x in iterable]",
            "        heapify(self.data)",
            "",
            "    def _neg(self, x: Any) -> Any:",
            "        if isinstance(x, tuple):",
            "            return tuple(-y for y in x)",
            "        return -x",
            "",
            "    def _encode(self, x: Any) -> Any:",
            "        return x if self.is_min else self._neg(x)",
            "",
            "    def _decode(self, x: Any) -> Any:",
            "        return x if self.is_min else self._neg(x)",
            "",
            "    def push(self, val: Any):",
            "        heappush(self.data, self._encode(val))",
            "",
            "    def pop(self) -> Any:",
            "        return self._decode(heappop(self.data))",
            "",
            "    def top(self) -> Any:",
            "        return self._decode(self.data[0])",
            "",
            "    def __len__(self) -> int:",
            "        return len(self.data)",
            "",
            "    def is_empty(self) -> bool:",
            "        return not self.data",
            "",
            "    @staticmethod",
            "    def nlargest(n: int, iterable: Iterable[Any]) -> List[Any]:",
            "        return nlargest(n, iterable)",
            "",
            "    @staticmethod",
            "    def nsmallest(n: int, iterable: Iterable[Any]) -> List[Any]:",
            "        return nsmallest(n, iterable)"
        ],
        "description": "Heap"
    },
    "SqrtSum": {
        "prefix": "SqrtSum",
        "body": [
            "class SqrtSum:",
            "    def __init__(self, data: List[int]):",
            "        self.n = len(data)",
            "        self.B = int(sqrt(self.n)) or 1",
            "        self.arr = data[:]",
            "        self.blk_cnt = (self.n + self.B - 1) // self.B",
            "        self.blk_sum = Arr.array(0, self.blk_cnt)",
            "        for i, v in enumerate(self.arr):",
            "            self.blk_sum[i // self.B] += v",
            "",
            "    def point_add(self, x: int, delta: int) -> None:",
            "        self.arr[x] += delta",
```

```
            "        self.blk_sum[x // self.B] += delta",
            "",
            "    def query(self, l: int, r: int) -> int:",
            "        b_l, b_r = l // self.B, r // self.B",
            "        if b_l == b_r:",
            "            return sum(self.arr[l: r + 1])",
            "",
            "        res = 0",
            "        end_l = (b_l + 1) * self.B - 1",
            "        for i in range(l, Math.min(end_l, self.n - 1) + 1):",
            "            res += self.arr[i]",
            "",
            "        for b in range(b_l + 1, b_r):",
            "            res += self.blk_sum[b]",
            "",
            "        start_r = b_r * self.B",
            "        for i in range(start_r, r + 1):",
            "            res += self.arr[i]",
            "",
            "        return res"
        ],
        "description": "SqrtSum"
    },
    "SqrtCountLE": {
        "prefix": "SqrtCountLE",
        "body": [
            "class SqrtCountLE:",
            "    def __init__(self, data: List[int]):",
            "        self.n = len(data)",
            "        self.B = int(sqrt(self.n)) or 1",
            "        self.data = data[:]",
            "        self.blk_cnt = (self.n + self.B - 1) // self.B",
            "        self.sorted_blk: List[List[int]] = []",
            "        for b in range(self.blk_cnt):",
            "            st = b * self.B",
            "            ed = Math.min(st + self.B, self.n)",
            "            self.sorted_blk.append(sorted(self.data[st:ed]))",
            "",
            "    def update(self, idx: int, new_val: int) -> None:",
            "        b = idx // self.B",
            "        old = self.data[idx]",
            "        # 1) 在 sorted_blk[b] 找到 old 的下标并 pop",
            "        pos = bisect_left(self.sorted_blk[b], old)",
            "        self.sorted_blk[b].pop(pos)",
            "        # 2) 用 insort 插入 new_val 保持有序",
            "        insort(self.sorted_blk[b], new_val)",
            "        # 3) 更新原数组",
            "        self.data[idx] = new_val",
            "",
            "    def count_leq(self, l: int, r: int, k: int) -> int:",
            "        \"\"\"统计 A[l..r] 中 ≤ k 的元素个数。\"\"\"",
            "        b_l, b_r = l // self.B, r // self.B",
            "        cnt = 0",
            "",
            "        if b_l == b_r:",
            "            for i in range(l, r + 1):",
            "                if self.data[i] ⩽ k:",
            "                    cnt += 1",
            "            return cnt",
            "",
            "        end_l = (b_l + 1) * self.B - 1",
            "        for i in range(l, Math.min(end_l, self.n - 1) + 1):",
            "            if self.data[i] ⩽ k:",
            "                cnt += 1",
            "",
            "        for b in range(b_l + 1, b_r):",
            "            cnt += bisect_right(self.sorted_blk[b], k)",
            "",
            "        start_r = b_r * self.B",
            "        for i in range(start_r, r + 1):",
            "            if self.data[i] ⩽ k:",
            "                cnt += 1",
            "",
            "        return cnt",
            "",
            "    def count_lt(self, l: int, r: int, k: int) -> int:",
            "        \"\"\"统计 A[l..r] 中 < k 的元素个数。\"\"\"",
            "        b_l, b_r = l // self.B, r // self.B",
```

```
            "        cnt = 0",
            "",
            "        if b_l == b_r:",
            "            for i in range(l, r + 1):",
            "                if self.data[i] < k:",
            "                    cnt += 1",
            "            return cnt",
            "",
            "        end_l = (b_l + 1) * self.B - 1",
            "        for i in range(l, Math.min(end_l, self.n - 1) + 1):",
            "            if self.data[i] < k:",
            "                cnt += 1",
            "",
            "        for b in range(b_l + 1, b_r):",
            "            cnt += bisect_left(self.sorted_blk[b], k)",
            "",
            "        start_r = b_r * self.B",
            "        for i in range(start_r, r + 1):",
            "            if self.data[i] < k:",
            "                cnt += 1",
            "",
            "        return cnt"
        ],
        "description": "SqrtCountLE"
    },
    "TopoSort": {
        "prefix": "TopoSort",
        "body": [
            "class TopoSort:",
            "    def __init__(self, n: int, directed: int = 1):",
            "        self.n = n",
            "        self.g = Arr.graph(n)",
            "        self.indegree = Arr.array(0, n)",
            "        self.directed = directed # 是否为有向图，如果是，则 add_edge 加两遍",
            "        self.order = []",
            "",
            "    def add_edge(self, u: int, v: int):",
            "        self.g[u].append(v)",
            "        self.indegree[v] += 1",
            "",
            "    def sort(self) -> bool:",
            "        q = deque(i for i in range(self.n) if self.indegree[i] == self.directed)",
            "        while q:",
            "            u = q.popleft()",
            "            self.order.append(u)",
            "            for v in self.g[u]:",
            "                self.indegree[v] -= 1",
            "                if self.indegree[v] == self.directed:",
            "                    q.append(v)",
            "        return len(self.order) == self.n"
        ],
        "description": "TopoSort"
    },
    "Erfen": {
        "prefix": "Erfen",
        "body": [
            "l, r = L, R",
            "best = -1",
            "while l <= r:",
            "    mid = (l + r + 1) >> 1",
            "    if check(mid):",
            "        best = mid",
            "        l = mid + 1",
            "    else:",
            "        r = mid - 1",
            "",
            "l, r = L, R",
            "best = -1",
            "while l <= r:",
            "    mid = (l + r) >> 1",
            "    if check(mid):",
            "        best = mid",
            "        r = mid - 1",
            "    else:",
            "        l = mid + 1"
        ],
        "description": "Erfen"
    }
```

```
2481 }
```