Team Notebook

November 18, 2025

Contents

1 data structures

1.1 Disjoint Set Union

```
class DSU:
    def __init__(self, n):
        self.p = list(range(n))
        self.sz = [1] * n

    def find(self, x):
        while x != self.p[x]:
            self.p[x] = self.p[self.p[x]]
            x = self.p[x]
        return x

    def unite(self, a, b):
        a, b = self.find(a), self.find(b)
        if a == b: return False
        if self.sz[a] < self.sz[b]:
            a, b = b, a
        self.p[b] = a
        self.sz[a] += self.sz[b]
        return True

    def same(self, a, b):
        return self.find(a) == self.find(b)
```

1.2 Fenwick Tree

```
class Fenwick:
    # 0-indexed, supports prefix sums on [0, i)
    def __init__(self, n):
        self.s = [0] * n

    def add(self, pos, delta):
        # a[pos] += delta
        n = len(self.s)
        while pos < n:
            self.s[pos] += delta
            pos |= pos + 1

    def sum(self, pos):
        # sum of [0, pos)
        res = 0
        while pos > 0:
            res += self.s[pos - 1]
            pos &= pos - 1
        return res

    def range_sum(self, l, r): # [l, r)
        return self.sum(r) - self.sum(l)

    def lower_bound(self, target):
        # min pos s.t. sum[0..pos] >= target, returns n if
            none
        if target <= 0: return -1
        n = len(self.s)
        pos = 0
        pw = 1 << (n.bit_length())
        while pw:
            nxt = pos + pw
            if nxt <= n and self.s[nxt - 1] < target:
                target -= self.s[nxt - 1]
                pos = nxt
            pw >>= 1
        return pos # in [0..n]
```

1.3 Segment Tree

```
class SegTree:
    # supports any associative op, default: min
    def __init__(self, n, func=min, unit=INF):
        self.N = 1
        while self.N < n:
            self.N <<= 1
        self.f = func
        self.unit = unit
        self.st = [unit] * (2 * self.N)

    def build(self, arr):
        for i, v in enumerate(arr):
            self.st[self.N + i] = v
        for i in range(self.N - 1, 0, -1):
            self.st[i] = self.f(self.st[2 * i], self.st[2 * i
                + 1])

    def update(self, pos, val):
        i = self.N + pos
        self.st[i] = val
        i >>= 1
        while i:
            self.st[i] = self.f(self.st[2 * i], self.st[2 * i
                + 1])
            i >>= 1

    def query(self, l, r):
        # [l, r)
        resl = self.unit
        resr = self.unit
        l += self.N
        r += self.N
        while l < r:
            if l & 1:
                resl = self.f(resl, self.st[l])
                l += 1
            if r & 1:
                r -= 1
                resr = self.f(self.st[r], resr)
            l >>= 1
            r >>= 1
        return self.f(resl, resr)
```

1.4 Sparse Table

```
class SparseTable:
    # for static array, idempotent op like min/max/gcd
    def __init__(self, arr, func=min):
        self.f = func
        n = len(arr)
        self.log = [0]*(n+1)
        for i in range(2, n+1):
            self.log[i] = self.log[i//2] + 1
        K = self.log[n] + 1
        st = [arr[:]]
        j = 1
        while (1 << j) <= n:
            prev = st[j-1]
            cur = []
            step = 1 << j
            half = step >> 1
            for i in range(0, n - step + 1):
                cur.append(self.f(prev[i], prev[i + half]))
            st.append(cur)
            j += 1
        self.st = st

    def query(self, l, r):
        # [l, r) (like KACTL RMQ)
        length = r - l
        k = self.log[length]
        return self.f(self.st[k][l], self.st[k][r - (1 << k)
            ])
```

2 graphs

2.1 BFS and DFS

```
# Assume g is adjacency list: g[u] = [(v, w), ...] for
    weighted.
def bfs(start, g):
    n = len(g)
    dist = [INF]*n
    dist[start] = 0
    dq = deque([start])
    while dq:
        u = dq.popleft()
        for v in g[u]:
            if dist[v] == INF:
                dist[v] = dist[u] + 1
                dq.append(v)
    return dist

def dfs(u, g, vis):
    vis[u] = True
    for v in g[u]:
        if not vis[v]:
            dfs(v, g, vis)
```

2.2 Dijkstra

```
def dijkstra(start, g):
    n = len(g)
    dist = [INF]*n
    dist[start] = 0
    pq = [(0, start)]
    while pq:
        d, u = heapq.heappop(pq)
        if d != dist[u]:
            continue
        for v, w in g[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                heapq.heappush(pq, (nd, v))
    return dist
```

2.3 Kruskal MST

```
def kruskal(n, edges):
    # edges: (w, u, v)
    dsu = DSU(n)
    edges.sort()
    total = 0
    used = []
    for w, u, v in edges:
        if dsu.unite(u, v):
            total += w
            used.append((u, v, w))
    return total, used
```

2.4 Strongly Connected Components

```
def scc(graph):
    n = len(graph)
    rg = [[] for _ in range(n)]
    for u in range(n):
        for v in graph[u]:
            rg[v].append(u)

    vis = [False]*n
    order = []

    def dfs1(u):
        vis[u] = True
        for v in graph[u]:
            if not vis[v]:
                dfs1(v)
        order.append(u)

    for i in range(n):
        if not vis[i]:
            dfs1(i)

    comp = [-1]*n
    cid = 0

    def dfs2(u, cid):
        comp[u] = cid
        for v in rg[u]:
            if comp[v] == -1:
                dfs2(v, cid)

    for u in reversed(order):
        if comp[u] == -1:
            dfs2(u, cid)
            cid += 1

    return comp, cid # comp[i] in [0..cid-1]
```

2.5 Topo Sort

```
def topo_sort(g):
    n = len(g)
    indeg = [0]*n
    for u in range(n):
        for v in g[u]:
            indeg[v] += 1
    q = deque([i for i in range(n) if indeg[i] == 0])
    order = []
    while q:
        u = q.popleft()
        order.append(u)
        for v in g[u]:
            indeg[v] -= 1
            if indeg[v] == 0:
                q.append(v)
    return order # len < n if cycle
```

3 math

3.1 Number Theory

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

def lcm(a, b):
    return a // gcd(a, b) * b

def extgcd(a, b):
    if b == 0:
        return a, 1, 0
    g, x1, y1 = extgcd(b, a % b)
    return g, y1, x1 - (a // b) * y1

def modinv(a, m=MOD):
    g, x, _ = extgcd(a, m)
    if g != 1:
        return None
    return x % m

def modpow(a, e, m=MOD):
```

```
    r = 1
    a %= m
    while e:
        if e & 1:
            r = r * a % m
        a = a * a % m
        e >>= 1
    return r
```

3.2 nCr

```
# Precompute up to N
NMAX = 2 * 10**5
fact = [1] * (NMAX + 1)
invfact = [1] * (NMAX + 1)
for i in range(1, NMAX + 1):
    fact[i] = fact[i - 1] * i % MOD
invfact[NMAX] = modpow(fact[NMAX], MOD - 2)
for i in range(NMAX, 0, -1):
    invfact[i - 1] = invfact[i] * i % MOD

def nCr(n, r):
    if r < 0 or r > n: return 0
    return fact[n] * invfact[r] % MOD * invfact[n - r] % MOD
```

4 prefix sums

4.1 2D Prefix Sums

```
class SubMatrix:
    # prefix sums on matrix, query sum over [u:d) x [l:r)
    def __init__(self, v):
        R, C = len(v), len(v[0])
        p = [[0]*(C+1) for _ in range(R+1)]
        for r in range(R):
            pr = p[r+1]
            for c in range(C):
                pr[c+1] = v[r][c] + p[r][c+1] + p[r+1][c] - p[
                    r][c]
        self.p = p

    def sum(self, u, l, d, r):
        p = self.p
        return p[d][r] - p[d][l] - p[u][r] + p[u][l]
```

5 search

5.1 Binary Search

```
import bisect

def bin_search_low(lo, hi, ok):
    # find min x in [lo, hi] with ok(x) True
    while lo < hi:
        mid = (lo + hi) // 2
        if ok(mid): hi = mid
        else: lo = mid + 1
    return lo

def bin_search_high(lo, hi, ok):
    # find max x in [lo, hi] with ok(x) True
    while lo < hi:
        mid = (lo + hi + 1) // 2
        if ok(mid): lo = mid
        else: hi = mid - 1
    return lo


def lower_bound(a, x):
    return bisect.bisect_left(a, x)

def upper_bound(a, x):
    return bisect.bisect_right(a, x)
```

6 strings

6.1 KMP

```
def prefix_function(s):
    n = len(s)
    p = [0]*n
    for i in range(1, n):
        j = p[i - 1]
        while j and s[i] != s[j]:
            j = p[j - 1]
        if s[i] == s[j]:
            j += 1
        p[i] = j
    return p

def kmp_match(text, pat):
    if not pat:
        return list(range(len(text) + 1))
    s = pat + "#" + text
    p = prefix_function(s)
    res = []
    m = len(pat)
    for i in range(m + 1, len(s)):
        if p[i] == m:
            res.append(i - 2*m)
    return res # starting indices
```

6.2 Manarcher

```
def manacher(s):
    n = len(s)
    # p[0][i]: even, p[1][i]: odd radii
    p0 = [0]*n # even
    p1 = [0]*n # odd

    # odd
    l = r = 0
    for i in range(n):
        k = 1 if i > r else min(p1[l + r - i], r - i + 1)
        while i - k >= 0 and i + k < n and s[i-k] == s[i+k]:
            k += 1
        p1[i] = k
        if i + k - 1 > r:
            l, r = i - k + 1, i + k - 1

    # even
    l = r = 0
    for i in range(n):
        k = 0 if i > r else min(p0[l + r - i + 1], r - i + 1)
        while i - k - 1 >= 0 and i + k < n and s[i-k-1] == s[
            i+k]:
            k += 1
        p0[i] = k
        if i + k - 1 > r:
            l, r = i - k, i + k - 1

    return p0, p1
```

6.3 Z Function

```
def z_function(s):
    n = len(s)
    z = [0]*n
    l = r = 0
```

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

7 utils

7.1 Header and IO

```
import sys, math, random, bisect, heapq
from collections import deque, defaultdict, Counter
sys.setrecursionlimit(10**7)
input = sys.stdin.readline
```

```
INF = 10**18
MOD = 10**9 + 7 # or 998244353

# Read helpers
def ints(): return list(map(int, input().split()))
def int1(): return int(input())
def strs(): return input().strip().split()
```