CSP-S易考代码总结

1. 扫描线算法（矩形面积并）

代码块

```
/**
 * 扫描线算法 - 计算多个矩形的面积并
 * 思路：将矩形拆分为左右两条边，按x坐标排序后扫描
 */

struct ScanLine {
    ll x;         // 边的x坐标
    ll y1, y2;    // 边的y坐标范围
    int flag;     // 1: 矩形的入边; -1: 矩形的出边
};

vector<ScanLine> lines;  // 存储所有扫描线
vector<ll> y_vals;       // 存储所有y坐标用于离散化

// 线段树节点（用于维护y轴上的覆盖情况）
struct SegmentTreeNode {
    ll len;      // 当前区间被覆盖的总长度
    int cnt;     // 当前区间被覆盖的次数
} tree[MAXN * 8];

// 构建线段树（离散化y坐标后）
void buildTree(int node, int left, int right) {
    tree[node] = {0, 0};
    if (left == right) return;
    int mid = (left + right) / 2;
    buildTree(node * 2, left, mid);
    buildTree(node * 2 + 1, mid + 1, right);
}

// 更新线段树
void updateTree(int node, int left, int right, int l, int r, int val) {
    if (l > right || r < left) return;

    if (l <= left && right <= r) {
        tree[node].cnt += val;
    } else {
```

```
        int mid = (left + right) / 2;
        updateTree(node * 2, left, mid, l, r, val);
        updateTree(node * 2 + 1, mid + 1, right, l, r, val);
    }

    // 更新当前区间的覆盖长度
    if (tree[node].cnt > 0) {
        tree[node].len = y_vals[right + 1] - y_vals[left];
    } else {
        if (left == right) {
            tree[node].len = 0;
        } else {
            tree[node].len = tree[node * 2].len + tree[node * 2 + 1].len;
        }
    }
}

// 主函数：计算矩形面积并
ll calculateUnionArea(vector<vector<ll>>& rectangles) {
    lines.clear();
    y_vals.clear();

    // 处理每个矩形，生成扫描线
    for (auto& rect : rectangles) {
        ll x1 = rect[0], y1 = rect[1], x2 = rect[2], y2 = rect[3];
        lines.push_back({x1, y1, y2, 1});
        lines.push_back({x2, y1, y2, -1});
        y_vals.push_back(y1);
        y_vals.push_back(y2);
    }

    // 离散化y坐标
    sort(y_vals.begin(), y_vals.end());
    y_vals.erase(unique(y_vals.begin(), y_vals.end()), y_vals.end());

    // 按x坐标排序扫描线
    sort(lines.begin(), lines.end(), [](const ScanLine& a, const ScanLine& b) {
        return a.x < b.x;
    });

    int m = y_vals.size() - 1;
    buildTree(1, 0, m - 1);

    ll total_area = 0;
    for (int i = 0; i < lines.size(); i++) {
        if (i > 0) {
            // 计算当前扫描线与上一条扫描线之间的面积
```

```
            total_area += tree[1].len * (lines[i].x - lines[i - 1].x);
        }

        // 找到当前边在离散化y坐标中的位置
        int l = lower_bound(y_vals.begin(), y_vals.end(), lines[i].y1) -
y_vals.begin();
        int r = lower_bound(y_vals.begin(), y_vals.end(), lines[i].y2) -
y_vals.begin() - 1;

        updateTree(1, 0, m - 1, l, r, lines[i].flag);
    }

    return total_area;
}
```

2. Manacher算法

代码块

```
/**
 * Manacher算法 - 查找最长回文子串
 * 优点：线性时间复杂度 O(n)
 */

// 预处理字符串：插入特殊字符统一奇偶情况
string preprocessString(const string& s) {
    string result = "#";
    for (char c : s) {
        result += c;
        result += '#';
    }
    return result;
}

// Manacher主算法
string longestPalindrome(const string& s) {
    if (s.empty()) return "";

    string t = preprocessString(s);
    int n = t.size();
    vector<int> p(n, 0);  // p[i] 表示以i为中心的最长回文半径

    int center = 0;  // 当前扩展的中心
```

```
    int right = 0;    // 当前扩展到的最右边界
    int max_center = 0;  // 最长回文的中心
    int max_len = 0;     // 最长回文的长度

    for (int i = 0; i < n; i++) {
        // 利用对称性快速获取初始回文半径
        int mirror = 2 * center - i;
        if (i < right) {
            p[i] = min(right - i, p[mirror]);
        }

        // 尝试扩展回文半径
        while (i + p[i] + 1 < n && i - p[i] - 1 >= 0 &&
               t[i + p[i] + 1] == t[i - p[i] - 1]) {
            p[i]++;
        }

        // 更新最右边界和中心
        if (i + p[i] > right) {
            center = i;
            right = i + p[i];
        }

        // 更新最长回文信息
        if (p[i] > max_len) {
            max_len = p[i];
            max_center = i;
        }
    }

    // 从预处理字符串还原原始字符串
    int start = (max_center - max_len) / 2;
    return s.substr(start, max_len);
}
```

3. 单调栈

代码块

```
/**
 * 单调栈 - 解决Next Greater Element问题
 * 用途：查找每个元素右边第一个比它大的元素
 */
```

```cpp

vector<int> nextGreaterElement(const vector<int>& nums) {
    int n = nums.size();
    vector<int> result(n, -1);  // 初始化结果数组，-1表示没有更大的元素
    stack<int> st;              // 单调递减栈（存储索引）

    for (int i = 0; i < n; i++) {
        // 当前元素比栈顶元素大，说明找到了栈顶元素的下一个更大元素
        while (!st.empty() && nums[st.top()] < nums[i]) {
            result[st.top()] = nums[i];
            st.pop();
        }
        st.push(i);
    }

    return result;
}

/**
 * 单调栈 - 柱状图中最大矩形面积
 * 思路：对于每个柱子，找到左右第一个比它矮的柱子
 */
int largestRectangleArea(const vector<int>& heights) {
    int n = heights.size();
    vector<int> left(n), right(n, n);  // 左右边界数组
    stack<int> st;  // 单调递增栈（存储索引）

    // 计算左边界
    for (int i = 0; i < n; i++) {
        while (!st.empty() && heights[st.top()] >= heights[i]) {
            st.pop();
        }
        left[i] = st.empty() ? -1 : st.top();
        st.push(i);
    }

    // 清空栈，计算右边界
    while (!st.empty()) st.pop();
    for (int i = n - 1; i >= 0; i--) {
        while (!st.empty() && heights[st.top()] >= heights[i]) {
            st.pop();
        }
        right[i] = st.empty() ? n : st.top();
        st.push(i);
    }

    // 计算最大面积
```

```
    int max_area = 0;
    for (int i = 0; i < n; i++) {
        int width = right[i] - left[i] - 1;
        max_area = max(max_area, heights[i] * width);
    }

    return max_area;
}
```

4. 单调队列（滑动窗口最大值）

代码块

```
/**
 * 单调队列 - 滑动窗口最大值
 * 维护一个单调递减的双端队列，队首总是当前窗口的最大值
 */

vector<int> maxSlidingWindow(const vector<int>& nums, int k) {
    if (nums.empty() || k == 0) return {};

    deque<int> dq;  // 双端队列，存储索引（维护单调递减）
    vector<int> result;

    for (int i = 0; i < nums.size(); i++) {
        // 移除超出窗口范围的元素（从队首）
        if (!dq.empty() && dq.front() == i - k) {
            dq.pop_front();
        }

        // 维护单调递减性：从队尾移除比当前元素小的元素
        while (!dq.empty() && nums[dq.back()] < nums[i]) {
            dq.pop_back();
        }

        dq.push_back(i);

        // 当窗口形成时，记录当前窗口的最大值
        if (i >= k - 1) {
            result.push_back(nums[dq.front()]);
        }
    }

```

```
    return result;
}
```

5. ST表（Range Maximum Query）

代码块

```
/**
 * ST表 - 解决RMQ问题（区间最大值查询）
 * 预处理：O(n log n)，查询：O(1)
 */

int st_table[MAXN][20];  // ST表，第二维大小取log2(MAXN)
int log_table[MAXN];     // 预处理log2表

// 预处理log2表
void precomputeLogTable(int n) {
    log_table[1] = 0;
    for (int i = 2; i <= n; i++) {
        log_table[i] = log_table[i / 2] + 1;
    }
}

// 构建ST表
void buildST(const vector<int>& arr) {
    int n = arr.size();
    precomputeLogTable(n);

    // 初始化第一层（长度为1的区间）
    for (int i = 0; i < n; i++) {
        st_table[i][0] = arr[i];
    }

    // 动态规划构建ST表
    for (int j = 1; j <= log_table[n]; j++) {
        for (int i = 0; i + (1 << j) <= n; i++) {
            st_table[i][j] = max(st_table[i][j - 1],
                                 st_table[i + (1 << (j - 1))][j - 1]);
        }
    }
}

// 查询区间 [l, r] 的最大值
```

```
int queryST(int l, int r) {
    int j = log_table[r - l + 1];
    return max(st_table[l][j], st_table[r - (1 << j) + 1][j]);
}
```

6. 字符串哈希

代码块

```
/**
 * 字符串哈希 - 快速比较子串是否相等
 * 使用双哈希避免冲突，Base1和Base2选择不同质数
 */

typedef unsigned long long ull;
const ull BASE1 = 131;      // 第一个哈希base
const ull BASE2 = 13131;    // 第二个哈希base（用于双哈希）
const ull MOD1 = 1e9 + 7;   // 第一个模数
const ull MOD2 = 1e9 + 9;   // 第二个模数

// 单哈希结构
struct StringHash {
    vector<ull> prefix;    // 前缀哈希值
    vector<ull> power;     // base的幂次
    ull base, mod;

    void init(const string& s, ull b, ull m) {
        base = b;
        mod = m;
        int n = s.size();
        prefix.resize(n + 1);
        power.resize(n + 1);

        power[0] = 1;
        prefix[0] = 0;

        for (int i = 0; i < n; i++) {
            prefix[i + 1] = (prefix[i] * base + s[i]) % mod;
            power[i + 1] = power[i] * base % mod;
        }
    }

    // 获取子串 [l, r] 的哈希值
```

```
    ull getHash(int l, int r) {
        return (prefix[r + 1] - prefix[l] * power[r - l + 1] % mod + mod) %
mod;
    }
};

// 双哈希结构
struct DoubleHash {
    StringHash hash1, hash2;

    void init(const string& s) {
        hash1.init(s, BASE1, MOD1);
        hash2.init(s, BASE2, MOD2);
    }

    // 返回双哈希的pair
    pair<ull, ull> getHash(int l, int r) {
        return {hash1.getHash(l, r), hash2.getHash(l, r)};
    }

    // 比较两个子串是否相等
    bool equal(int l1, int r1, int l2, int r2) {
        return getHash(l1, r1) == getHash(l2, r2);
    }
};
```

7. 并查集 (Union-Find)

代码块

```
/**
 * 并查集 - 管理不相交集合，支持合并和查询
 * 路径压缩 + 按秩合并优化
 */

int parent[MAXN];  // 父节点数组
int rank_[MAXN];   // 秩（树的高度）

// 初始化并查集
void initUnionFind(int n) {
    for (int i = 0; i <= n; i++) {
        parent[i] = i;      // 初始时每个元素的父节点是自己
        rank_[i] = 0;       // 初始秩为0
```

```
    }
}

// 查找根节点（带路径压缩）
int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]);  // 路径压缩
    }
    return parent[x];
}

// 合并两个集合（按秩合并）
void unionSets(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);

    if (rootX == rootY) return;  // 已经在同一集合

    // 按秩合并：将矮树合并到高树下
    if (rank_[rootX] < rank_[rootY]) {
        parent[rootX] = rootY;
    } else if (rank_[rootX] > rank_[rootY]) {
        parent[rootY] = rootX;
    } else {
        parent[rootY] = rootX;
        rank_[rootX]++;  // 秩相等时，合并后秩加1
    }
}

// 检查两个元素是否在同一集合
bool isConnected(int x, int y) {
    return find(x) == find(y);
}
```

8. 快速幂和逆元

代码块

```
/**
 * 快速幂算法 - 计算 a^b mod m
 * 逆元：a在模m下的逆元满足 a * inv(a) ≡ 1 (mod m)
 */

```

```cpp
// 快速幂计算
ll fastPow(ll a, ll b, ll mod) {
    ll result = 1;
    a %= mod;  // 先取模防止溢出

    while (b > 0) {
        if (b & 1) {  // 如果b是奇数
            result = result * a % mod;
        }
        a = a * a % mod;  // a平方
        b >>= 1;          // b除以2
    }

    return result;
}

// 费马小定理求逆元（要求mod是质数）
ll modInverse(ll a, ll mod) {
    return fastPow(a, mod - 2, mod);
}

// 扩展欧几里得算法求逆元（不要求mod是质数）
ll extendedGcd(ll a, ll b, ll& x, ll& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }

    ll x1, y1;
    ll gcd = extendedGcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return gcd;
}

ll modInverseExt(ll a, ll mod) {
    ll x, y;
    ll gcd = extendedGcd(a, mod, x, y);
    if (gcd != 1) {
        return -1;  // 逆元不存在
    }
    return (x % mod + mod) % mod;  // 确保结果为正数
}
```

9. 线性求逆元

代码块

```
/**
 * 线性求逆元 - O(n)时间内求出1到n所有数的逆元
 * 公式: inv[i] = (mod - mod / i) * inv[mod % i] % mod
 */

ll inv[MAXN];  // 逆元数组

// 线性预处理1到n的逆元
void precomputeInverses(int n, ll mod) {
    inv[1] = 1;  // 1的逆元是1

    for (int i = 2; i <= n; i++) {
        inv[i] = (mod - mod / i) * inv[mod % i] % mod;
    }
}

// 线性预处理阶乘和阶乘逆元（用于组合数计算）
ll fact[MAXN], invFact[MAXN];

void precomputeFactorials(int n, ll mod) {
    fact[0] = invFact[0] = 1;

    // 计算阶乘
    for (int i = 1; i <= n; i++) {
        fact[i] = fact[i - 1] * i % mod;
    }

    // 计算阶乘逆元
    invFact[n] = fastPow(fact[n], mod - 2, mod);
    for (int i = n - 1; i >= 1; i--) {
        invFact[i] = invFact[i + 1] * (i + 1) % mod;
    }
}
```

10. 排列组合数计算

代码块

```
/**
 * 组合数计算 - 使用预处理的阶乘和阶乘逆元
 * C(n, k) = n! / (k! * (n - k)!)
 * A(n, k) = n! / (n - k)!
 */

// 初始化组合数计算所需的数据
void initCombinatorics(int n, ll mod) {
    precomputeFactorials(n, mod);
}

// 计算组合数 C(n, k)
ll comb(int n, int k, ll mod) {
    if (k < 0 || k > n) return 0;
    if (k == 0 || k == n) return 1;

    return fact[n] * invFact[k] % mod * invFact[n - k] % mod;
}

// 计算排列数 A(n, k)
ll perm(int n, int k, ll mod) {
    if (k < 0 || k > n) return 0;
    return fact[n] * invFact[n - k] % mod;
}

// 卢卡斯定理（用于大组合数取模，当mod较小但n很大时）
ll lucas(int n, int k, ll mod) {
    if (k == 0) return 1;
    return comb(n % mod, k % mod, mod) * lucas(n / mod, k / mod, mod) % mod;
}
```

11. 树的直径

代码块

```
/**
 * 树的直径 - 树中最长路径的长度
 * 两次BFS/DFS：第一次从任意点找到最远点，第二次从该点找到最远点
 */

int dist[MAXN];  // 距离数组
int farthest_node;  // 记录最远节点
int max_distance;   // 记录最大距离
```

```
9
10  // DFS查找最远节点
11  void dfsForDiameter(int u, int parent, int current_dist) {
12      dist[u] = current_dist;
13
14      if (current_dist > max_distance) {
15          max_distance = current_dist;
16          farthest_node = u;
17      }
18
19      for (int v : graph[u]) {
20          if (v != parent) {
21              dfsForDiameter(v, u, current_dist + 1);
22          }
23      }
24  }
25
26  // 计算树的直径（返回直径长度）
27  int treeDiameter(int n) {
28      // 第一次DFS：从节点0找到最远点
29      max_distance = -1;
30      dfsForDiameter(0, -1, 0);
31
32      // 第二次DFS：从最远点找到另一个最远点
33      int start_node = farthest_node;
34      max_distance = -1;
35      dfsForDiameter(start_node, -1, 0);
36
37      return max_distance;
38  }
39
40  // 另一种方法：树形DP求直径
41  int diameterDP(int u, int parent, int& diameter) {
42      int max_depth1 = 0, max_depth2 = 0;
43
44      for (int v : graph[u]) {
45          if (v == parent) continue;
46
47          int depth = diameterDP(v, u, diameter);
48          if (depth > max_depth1) {
49              max_depth2 = max_depth1;
50              max_depth1 = depth;
51          } else if (depth > max_depth2) {
52              max_depth2 = depth;
53          }
54      }
55
```

```
    diameter = max(diameter, max_depth1 + max_depth2);
    return max_depth1 + 1;
}
```

12. 树的重心

代码块

```
/**
 * 树的重心 - 树中删除该点后，最大连通分量最小的点
 * 性质：重心最多有两个，且大小不超过 n/2
 */

int size_[MAXN];    // 记录每个节点的子树大小
int max_subtree[MAXN]; // 记录删除该点后最大连通分量大小
int centroid;       // 记录重心
int n;              // 节点总数

// DFS计算子树大小并寻找重心
void findCentroid(int u, int parent) {
    size_[u] = 1;
    max_subtree[u] = 0;

    for (int v : graph[u]) {
        if (v == parent) continue;

        findCentroid(v, u);
        size_[u] += size_[v];
        max_subtree[u] = max(max_subtree[u], size_[v]);
    }

    // 考虑父节点方向的连通分量
    max_subtree[u] = max(max_subtree[u], n - size_[u]);

    // 更新重心（最大连通分量最小的点）
    if (max_subtree[u] < max_subtree[centroid]) {
        centroid = u;
    }
}

// 主函数：寻找树的重心
int getTreeCentroid(int node_count) {
    n = node_count;
```

```
    centroid = 0;
    max_subtree[0] = n; // 初始化

    findCentroid(0, -1);
    return centroid;
}

// 查找所有重心（可能有1或2个）
vector<int> getAllCentroids(int node_count) {
    n = node_count;
    centroid = 0;
    max_subtree[0] = n;

    findCentroid(0, -1);

    vector<int> centroids;
    int min_size = max_subtree[centroid];
    centroids.push_back(centroid);

    // 检查是否可能有第二个重心
    for (int i = 0; i < n; i++) {
        if (i != centroid && max_subtree[i] == min_size) {
            centroids.push_back(i);
        }
    }

    return centroids;
}
```

13. LCA（最近公共祖先）

代码块

```
/**
 * LCA - 最近公共祖先，使用倍增算法
 * 预处理: O(n log n)，查询: O(log n)
 */

int depth[MAXN];        // 节点深度
int parent[MAXN][20];   // 倍增父节点表
int log_value;          // 最大层数

// DFS预处理深度和直接父节点
```

```
11  void dfsForLCA(int u, int p, int d) {
12      depth[u] = d;
13      parent[u][0] = p;
14
15      for (int v : graph[u]) {
16          if (v != p) {
17              dfsForLCA(v, u, d + 1);
18          }
19      }
20  }
21
22  // 构建倍增表
23  void buildLCA(int n, int root) {
24      // 先DFS获取深度和直接父节点
25      dfsForLCA(root, -1, 0);
26
27      // 计算最大层数
28      log_value = log2(n) + 1;
29
30      // 动态规划构建倍增表
31      for (int j = 1; j <= log_value; j++) {
32          for (int i = 0; i < n; i++) {
33              if (parent[i][j - 1] != -1) {
34                  parent[i][j] = parent[parent[i][j - 1]][j - 1];
35              } else {
36                  parent[i][j] = -1;
37              }
38          }
39      }
40  }
41
42  // 查询节点u和v的LCA
43  int queryLCA(int u, int v) {
44      // 确保u是较深的节点
45      if (depth[u] < depth[v]) {
46          swap(u, v);
47      }
48
49      // 将u提升到与v同一深度
50      int diff = depth[u] - depth[v];
51      for (int i = 0; i <= log_value; i++) {
52          if (diff & (1 << i)) {
53              u = parent[u][i];
54          }
55      }
56
57      // 如果此时u == v，说明v就是LCA
```

```
    if (u == v) {
        return u;
    }

    // 同时向上跳跃
    for (int i = log_value; i >= 0; i--) {
        if (parent[u][i] != parent[v][i]) {
            u = parent[u][i];
            v = parent[v][i];
        }
    }

    return parent[u][0];
}

// 查询两点间距离
int getDistance(int u, int v) {
    int lca = queryLCA(u, v);
    return depth[u] + depth[v] - 2 * depth[lca];
}
```

14. Ford（Bellman-Ford最短路径）

代码块

```
/**
 * Bellman-Ford算法 - 单源最短路径，可处理负权边
 * 可检测负权环，时间复杂度O(VE)
 */

struct WeightedEdge {
    int from;
    int to;
    int weight;
};

vector<WeightedEdge> edges;  // 存储所有边
int dist[MAXN];              // 最短距离数组
int n, m;                    // 节点数和边数

// Bellman-Ford算法
bool bellmanFord(int start) {
    // 初始化距离数组
```

```
    fill(dist, dist + n, INF);
    dist[start] = 0;

    // 松弛操作执行n-1次
    for (int i = 0; i < n - 1; i++) {
        bool updated = false;
        for (const auto& edge : edges) {
            if (dist[edge.from] != INF &&
                dist[edge.to] > dist[edge.from] + edge.weight) {
                dist[edge.to] = dist[edge.from] + edge.weight;
                updated = true;
            }
        }
        if (!updated) break; // 提前终止
    }

    // 检查负权环：如果还能松弛，说明存在负权环
    for (const auto& edge : edges) {
        if (dist[edge.from] != INF &&
            dist[edge.to] > dist[edge.from] + edge.weight) {
            return false; // 存在负权环
        }
    }

    return true;
}
```

15. Dijkstra最短路径

代码块

```
/**
 * Dijkstra算法 - 单源最短路径，不能处理负权边
 * 使用优先队列优化，时间复杂度O((V+E)logV)
 */

int dist[MAXN];  // 最短距离数组
bool visited[MAXN]; // 访问标记

// Dijkstra算法（优先队列优化）
void dijkstra(int start) {
    fill(dist, dist + n, INF);
    fill(visited, visited + n, false);
```

```
    dist[start] = 0;

    // 使用小顶堆：存储{距离, 节点}
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, start});

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        if (visited[u]) continue;
        visited[u] = true;

        for (const auto& edge : graphW[u]) {
            int v = edge.to;
            int weight = edge.weight;

            if (dist[v] > dist[u] + weight) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
}

// 另一种实现：使用邻接表
void dijkstraAdjList(int start) {
    fill(dist, dist + n, INF);
    dist[start] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, start});

    while (!pq.empty()) {
        int u = pq.top().second;
        int d = pq.top().first;
        pq.pop();

        if (d != dist[u]) continue; // 旧的记录，跳过

        for (const auto& edge : graphW[u]) {
            int v = edge.to;
            int w = edge.weight;

            if (dist[v] > dist[u] + w) {
```

```
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
}
```

16. Floyd-Warshall最短路径

代码块

```
/**
 * Floyd-Warshall算法 - 所有点对最短路径
 * 可处理负权边，不能处理负权环，时间复杂度O(V^3)
 */

int floyd[MAXN][MAXN];  // 距离矩阵

// 初始化Floyd矩阵
void initFloyd(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                floyd[i][j] = 0;
            } else {
                floyd[i][j] = INF;
            }
        }
    }
}

// Floyd-Warshall算法
void floydWarshall(int n) {
    // 初始化：直接相连的边
    for (int i = 0; i < n; i++) {
        for (const auto& edge : graphW[i]) {
            floyd[i][edge.to] = min(floyd[i][edge.to], edge.weight);
        }
    }

    // 动态规划：中间节点k
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
```

```
            if (floyd[i][k] == INF) continue;
            for (int j = 0; j < n; j++) {
                if (floyd[i][k] != INF && floyd[k][j] != INF) {
                    floyd[i][j] = min(floyd[i][j], floyd[i][k] + floyd[k][j]);
                }
            }
        }
    }
}

// 检查负权环
bool hasNegativeCycle(int n) {
    for (int i = 0; i < n; i++) {
        if (floyd[i][i] < 0) {
            return true;
        }
    }
    return false;
}
```

17. Kruskal最小生成树

代码块

```
/**
 * Kruskal算法 - 最小生成树（MST）
 * 使用并查集，按边权从小到大选择不形成环的边
 * 时间复杂度：O(E log E)
 */

struct KruskalEdge {
    int from;
    int to;
    int weight;

    bool operator<(const KruskalEdge& other) const {
        return weight < other.weight;
    }
};

vector<KruskalEdge> kruskal_edges;
int parent_kruskal[MAXN];
int rank_kruskal[MAXN];
```

```
20
21  // 并查集查找
22  int findKruskal(int x) {
23      if (parent_kruskal[x] != x) {
24          parent_kruskal[x] = findKruskal(parent_kruskal[x]);
25      }
26      return parent_kruskal[x];
27  }
28
29  // 并查集合并
30  void unionKruskal(int x, int y) {
31      int rootX = findKruskal(x);
32      int rootY = findKruskal(y);
33      if (rootX == rootY) return;
34
35      if (rank_kruskal[rootX] < rank_kruskal[rootY]) {
36          parent_kruskal[rootX] = rootY;
37      } else {
38          parent_kruskal[rootY] = rootX;
39          if (rank_kruskal[rootX] == rank_kruskal[rootY]) {
40              rank_kruskal[rootX]++;
41          }
42      }
43  }
44
45  // Kruskal主算法
46  int kruskalMST(int n, int m) {
47      // 初始化并查集
48      for (int i = 0; i < n; i++) {
49          parent_kruskal[i] = i;
50          rank_kruskal[i] = 0;
51      }
52
53      // 按边权排序
54      sort(kruskal_edges.begin(), kruskal_edges.end());
55
56      int mst_weight = 0;
57      int edges_selected = 0;
58
59      for (const auto& edge : kruskal_edges) {
60          if (edges_selected == n - 1) break;
61
62          int rootFrom = findKruskal(edge.from);
63          int rootTo = findKruskal(edge.to);
64
65          if (rootFrom != rootTo) {
66              unionKruskal(rootFrom, rootTo);
```

```
            mst_weight += edge.weight;
            edges_selected++;
        }
    }

    if (edges_selected != n - 1) {
        return -1; // 图不连通
    }
    return mst_weight;
}
```

18. Prim最小生成树

代码块

```
/**
 * Prim算法 - 最小生成树（MST）
 * 从任意点开始，每次选择连接已选集合和未选集合的最小边
 * 使用优先队列优化，时间复杂度：O(E log V)
 */

int primMST(int n, int start) {
    vector<int> dist(n, INF);     // 到MST的最小距离
    vector<bool> inMST(n, false); // 是否在MST中
    int mst_weight = 0;

    // 小顶堆：存储{权重, 节点}
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    dist[start] = 0;
    pq.push({0, start});

    while (!pq.empty()) {
        int u = pq.top().second;
        int weight = pq.top().first;
        pq.pop();

        if (inMST[u]) continue;
        inMST[u] = true;
        mst_weight += weight;

        for (const auto& edge : graphW[u]) {
```

```
            int v = edge.to;
            int w = edge.weight;

            if (!inMST[v] && w < dist[v]) {
                dist[v] = w;
                pq.push({w, v});
            }
        }
    }

    // 检查是否所有节点都在MST中
    for (int i = 0; i < n; i++) {
        if (!inMST[i]) {
            return -1; // 图不连通
        }
    }

    return mst_weight;
}
```

19. 拓扑排序

代码块

```
/**
 * 拓扑排序 - 有向无环图（DAG）的线性排序
 * Kahn算法：不断删除入度为0的节点
 * 时间复杂度：O(V + E)
 */

vector<int> topologicalSort(int n) {
    vector<int> in_degree(n, 0);
    vector<int> result;
    queue<int> q;

    // 计算每个节点的入度
    for (int i = 0; i < n; i++) {
        for (int neighbor : graph[i]) {
            in_degree[neighbor]++;
        }
    }

    // 将所有入度为0的节点加入队列
```

```cpp
    for (int i = 0; i < n; i++) {
        if (in_degree[i] == 0) {
            q.push(i);
        }
    }

    // Kahn算法
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result.push_back(u);

        for (int v : graph[u]) {
            in_degree[v]--;
            if (in_degree[v] == 0) {
                q.push(v);
            }
        }
    }

    // 检查是否有环
    if (result.size() != n) {
        return {}; // 存在环，无法拓扑排序
    }

    return result;
}

// DFS版本的拓扑排序
vector<int> topo_dfs_result;
bool visited_topo[MAXN];

void dfsTopo(int u) {
    visited_topo[u] = true;
    for (int v : graph[u]) {
        if (!visited_topo[v]) {
            dfsTopo(v);
        }
    }
    topo_dfs_result.push_back(u);
}

vector<int> topologicalSortDFS(int n) {
    fill(visited_topo, visited_topo + n, false);
    topo_dfs_result.clear();

    for (int i = 0; i < n; i++) {
```

```
        if (!visited_topo[i]) {
            dfsTopo(i);
        }
    }

    reverse(topo_dfs_result.begin(), topo_dfs_result.end());
    return topo_dfs_result;
}
```

20. Tarjan算法（强连通分量）

代码块

```
/**
 * Tarjan算法 - 寻找有向图的强连通分量（SCC）
 * 使用DFS和栈，时间复杂度：O(V + E)
 */

int tarjan_index = 0;
int dfn[MAXN];      // DFS序（时间戳）
int low[MAXN];      // 能回溯到的最早时间戳
bool in_stack[MAXN]; // 是否在栈中
stack<int> tarjan_stack;
vector<vector<int>> scc_list; // 存储所有SCC

void tarjanDFS(int u) {
    dfn[u] = low[u] = ++tarjan_index;
    tarjan_stack.push(u);
    in_stack[u] = true;

    for (int v : graph[u]) {
        if (dfn[v] == 0) { // 未访问
            tarjanDFS(v);
            low[u] = min(low[u], low[v]);
        } else if (in_stack[v]) { // 在栈中（后向边）
            low[u] = min(low[u], dfn[v]);
        }
    }

    // 找到SCC的根节点
    if (dfn[u] == low[u]) {
        vector<int> scc;
        while (true) {
```

```
            int v = tarjan_stack.top();
            tarjan_stack.pop();
            in_stack[v] = false;
            scc.push_back(v);
            if (v == u) break;
        }
        scc_list.push_back(scc);
    }
}

// Tarjan主函数
vector<vector<int>> findSCC(int n) {
    // 初始化
    tarjan_index = 0;
    fill(dfn, dfn + n, 0);
    fill(low, low + n, 0);
    fill(in_stack, in_stack + n, false);
    scc_list.clear();
    while (!tarjan_stack.empty()) tarjan_stack.pop();

    // 对每个未访问的节点执行Tarjan
    for (int i = 0; i < n; i++) {
        if (dfn[i] == 0) {
            tarjanDFS(i);
        }
    }

    return scc_list;
}

// Tarjan求割点（无向图）
bool is_cut_vertex[MAXN]; // 是否是割点

void tarjanCutVertex(int u, int parent) {
    dfn[u] = low[u] = ++tarjan_index;
    int children = 0;

    for (int v : graph[u]) {
        if (v == parent) continue;

        if (dfn[v] == 0) {
            children++;
            tarjanCutVertex(v, u);
            low[u] = min(low[u], low[v]);

            // 判断是否是割点
            if (parent != -1 && low[v] >= dfn[u]) {
```

```
                is_cut_vertex[u] = true;
            }
        } else {
            low[u] = min(low[u], dfn[v]);
        }
    }

    // 根节点特殊处理
    if (parent == -1 && children > 1) {
        is_cut_vertex[u] = true;
    }
}
```

21. 欧拉回路

代码块

```
/**
 * 欧拉回路/欧拉路径 - 遍历图中每条边恰好一次
 * 欧拉回路：起点=终点；欧拉路径：起点≠终点
 */

vector<int> euler_path;
int edge_pointer[MAXN]; // 边指针（用于避免重复访问）

// Hierholzer算法（递归版本）
void hierholzerDFS(int u) {
    while (edge_pointer[u] < graph[u].size()) {
        int v = graph[u][edge_pointer[u]++];
        hierholzerDFS(v);
    }
    euler_path.push_back(u);
}

// 检查是否存在欧拉回路/路径
pair<bool, bool> checkEuler(int n) {
    vector<int> in_degree(n, 0), out_degree(n, 0);

    // 计算每个节点的出入度
    for (int u = 0; u < n; u++) {
        out_degree[u] = graph[u].size();
        for (int v : graph[u]) {
            in_degree[v]++;
```

```
        }
    }

    int start_candidates = 0, end_candidates = 0;
    int start_node = -1;

    for (int i = 0; i < n; i++) {
        int diff = out_degree[i] - in_degree[i];
        if (abs(diff) > 1) return {false, false}; // 不存在

        if (diff == 1) {
            start_candidates++;
            start_node = i;
        } else if (diff == -1) {
            end_candidates++;
        }
    }

    // 欧拉回路：所有节点出入度相等
    if (start_candidates == 0 && end_candidates == 0) {
        return {true, true}; // 存在欧拉回路
    }

    // 欧拉路径：恰好一个起点和一个终点
    if (start_candidates == 1 && end_candidates == 1) {
        return {true, false}; // 存在欧拉路径
    }

    return {false, false};
}

// 寻找欧拉回路/路径
vector<int> findEulerPath(int n) {
    auto [exists, is_cycle] = checkEuler(n);
    if (!exists) return {};

    // 找到起点
    int start = 0;
    vector<int> in_degree(n, 0), out_degree(n, 0);
    for (int u = 0; u < n; u++) {
        out_degree[u] = graph[u].size();
        for (int v : graph[u]) {
            in_degree[v]++;
        }
    }

    for (int i = 0; i < n; i++) {
```

```
        if (out_degree[i] > in_degree[i]) {
            start = i;
            break;
        }
    }

    // 初始化边指针
    fill(edge_pointer, edge_pointer + n, 0);
    euler_path.clear();

    hierholzerDFS(start);
    reverse(euler_path.begin(), euler_path.end());

    return euler_path;
}
```

22. 树状数组（Fenwick Tree）

代码块

```
/**
 * 树状数组（Fenwick Tree） - 支持单点更新和前缀查询
 * 时间复杂度：更新O(log n)，查询O(log n)
 */

class FenwickTree {
private:
    vector<int> tree;
    int n;

public:
    FenwickTree(int size) : n(size) {
        tree.resize(n + 1, 0);
    }

    // 低位操作：获取最低位的1
    int lowbit(int x) {
        return x & -x;
    }

    // 单点更新：在位置index增加value
    void update(int index, int value) {
        while (index <= n) {
```

```
            tree[index] += value;
            index += lowbit(index);
        }
    }

    // 查询前缀和：[1, index]的和
    int query(int index) {
        int sum = 0;
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }

    // 区间查询：[left, right]的和
    int rangeQuery(int left, int right) {
        return query(right) - query(left - 1);
    }
};

// 二维树状数组
class FenwickTree2D {
private:
    vector<vector<int>> tree;
    int n, m;

public:
    FenwickTree2D(int rows, int cols) : n(rows), m(cols) {
        tree.resize(n + 1, vector<int>(m + 1, 0));
    }

    void update(int x, int y, int value) {
        for (int i = x; i <= n; i += i & -i) {
            for (int j = y; j <= m; j += j & -j) {
                tree[i][j] += value;
            }
        }
    }

    int query(int x, int y) {
        int sum = 0;
        for (int i = x; i > 0; i -= i & -i) {
            for (int j = y; j > 0; j -= j & -j) {
                sum += tree[i][j];
            }
        }
```

```
        return sum;
    }

    int rangeQuery(int x1, int y1, int x2, int y2) {
        return query(x2, y2) - query(x1 - 1, y2) - query(x2, y1 - 1) +
query(x1 - 1, y1 - 1);
    }
};
```

23. 线段树（Segment Tree）

代码块

```
/**
 * 线段树 - 支持区间查询和区间更新
 * 模板：区间和查询，支持区间加值
 */

class SegmentTree {
private:
    vector<int> tree;   // 线段树数组
    vector<int> lazy;   // 懒标记数组
    int n;

    // 构建线段树
    void build(const vector<int>& arr, int node, int left, int right) {
        if (left == right) {
            tree[node] = arr[left];
            return;
        }

        int mid = (left + right) / 2;
        int left_node = 2 * node + 1;
        int right_node = 2 * node + 2;

        build(arr, left_node, left, mid);
        build(arr, right_node, mid + 1, right);

        tree[node] = tree[left_node] + tree[right_node];
    }

    // 下推懒标记
    void pushDown(int node, int left, int right) {
```

```
        if (lazy[node] != 0) {
            int mid = (left + right) / 2;
            int left_node = 2 * node + 1;
            int right_node = 2 * node + 2;

            // 更新子节点值和懒标记
            tree[left_node] += lazy[node] * (mid - left + 1);
            tree[right_node] += lazy[node] * (right - mid);
            lazy[left_node] += lazy[node];
            lazy[right_node] += lazy[node];

            lazy[node] = 0; // 清除当前节点的懒标记
        }
    }

    // 区间更新
    void updateRange(int node, int left, int right, int l, int r, int value) {
        if (l > right || r < left) return;

        if (l <= left && right <= r) {
            tree[node] += value * (right - left + 1);
            lazy[node] += value;
            return;
        }

        pushDown(node, left, right);
        int mid = (left + right) / 2;
        int left_node = 2 * node + 1;
        int right_node = 2 * node + 2;

        updateRange(left_node, left, mid, l, r, value);
        updateRange(right_node, mid + 1, right, l, r, value);

        tree[node] = tree[left_node] + tree[right_node];
    }

    // 区间查询
    int queryRange(int node, int left, int right, int l, int r) {
        if (l > right || r < left) return 0;

        if (l <= left && right <= r) {
            return tree[node];
        }

        pushDown(node, left, right);
        int mid = (left + right) / 2;
        int left_node = 2 * node + 1;
```

```
78          int right_node = 2 * node + 2;
79
80          int left_sum = queryRange(left_node, left, mid, l, r);
81          int right_sum = queryRange(right_node, mid + 1, right, l, r);
82
83          return left_sum + right_sum;
84      }
85
86  public:
87      SegmentTree(const vector<int>& arr) {
88          n = arr.size();
89          tree.resize(4 * n);
90          lazy.resize
```