AJ1E 的板子

图论

最小生成树

我们定义无向连通图的 **最小生成树**（Minimum Spanning Tree，MST）为边权和最小的生成树。

注意：只有连通图才有生成树，而对于非连通图，只存在生成森林。

Kruskal

维护一个森林，查询两个结点是否在同一棵树中，连接两棵树。抽象一点地说，维护一堆 **集合**，查询两个元素是否属于同一集合，合并两个集合。其中，查询两点是否连通和连接两点可以使用并查集维护。

如果使用 $O(mlogm)$的排序算法，并且使用 $O(mlogm)$并查集，就可以得到时间复杂度为 $O(mlogm)$的 Kruskal 算法。

```
struct Edge {
    int u, v, w;
    bool operator < (const Edge& rh) const {
        return w < rh.w;
    };
};

void solve() {
    int n, m;
    cin >> n >> m;
    vector<Edge> e(m);
    for (int i = 0; i < m; i ++ )
        cin >> e[i].u >> e[i].v >> e[i].w;
    vector<int> p(n + 1);
    for (int i = 1; i <= n; i ++ )
        p[i] = i;
    function<int(int)> find = [&] (int x) {
        return x == p[x] ? x : p[x] = find(p[x]);
    };
    int ans = 0, cnt = 0;
    function<void()> kruskal = [&] () {
        sort(e.begin(), e.end());
        for (auto [u, v, w] : e) {
            u = find(u), v = find(v);
            if (u == v)
                continue;
            ans += w, cnt ++;
            p[v] = u;
            if (cnt == n - 1) break;
        }
    };
    kruskal();
    if (cnt == n - 1) cout << ans << "\n";
    else cout << "orz\n";
}
```

最小生成树唯一性

考虑最小生成树的唯一性。如果一条边 **不在最小生成树的边集中**，并且可以替换与其 **权值相同、并且在最小生成树边集** 的另一条边。那么，这个最小生成树就是不唯一的。

对于 Kruskal 算法，只要计算为当前权值的边可以放几条，实际放了几条，如果这两个值不一样，那么就说明这几条边与之前的边产生了一个环（这个环中至少有两条当前权值的边，否则根据并查集，这条边是不能放的），即最小生成树不唯一。

寻找权值与当前边相同的边，我们只需要记录头尾指针，用单调队列即可在$O(a(m))$ (m 为边数）的时间复杂度里优秀解决这个问题（基本与原算法时间相同）。

```
// undo
```

Kruskal 重构树

首先新建 n 个集合，每个集合恰有一个节点，点权为 0。

每一次加边会合并两个集合，我们可以新建一个点，点权为加入边的边权，同时将两个集合的根节点分别设为新建点的左儿子和右儿子。然后我们将两个集合和新建点合并成一个集合。将新建点设为根。

不难发现，在进行$n-1$轮之后我们得到了一棵恰有 n个叶子的二叉树，同时每个非叶子节点恰好有两个儿子。这棵树就叫 Kruskal 重构树。

可以用Kruskal重构树计算路径上最小边最大（最大边最小）问题， 只需要构建最小生成树（最大生成树）求 u 和 v 两点的LCA权值。

```
struct Edge {
    int u, v, w;
    bool operator < (const Edge& rh) const {
        return w < rh.w;
    }
};

void solve() {
    int n, m;
    cin >> n >> m;
    vector<int> p(n << 1);
    for (int i = 1; i <= n; i ++ )
        p[i] = i;
    function<int(int)> find = [&] (int x) {
        return x == p[x] ? x : find(p[x]);
    };
    vector<Edge> e(m);
    for (int i = 0; i < m; i ++ )
        cin >> e[i].u >> e[i].v >> e[i].w;
    vector g(n << 1, vector<int> ());
    vector<int> val(n << 1), siz(n << 1), dep(n << 1);
    vector<int> top(n << 1), son(n << 1), ff(n << 1);
    vector<bool> vis(n << 1);
    int idx = n;
    function<void(int, int)> dfs1 = [&] (int u, int fa) {
        siz[u] = 1; vis[u] = true;
        for (auto v : g[u]) {
            if (v == fa) continue;
```

```
            dep[v] = dep[u] + 1, ff[v] = u;
            dfs1(v, u);
            siz[u] += siz[v];
            if (siz[v] > siz[son[u]]) son[u] = v;
        }
    };
    function<void(int, int)> dfs2 = [&] (int u, int tp) {
        top[u] = tp;
        if (!son[u]) return ;
        dfs2(son[u], tp);
        for (auto v : g[u]) {
            if (v == son[u] || v == ff[u])
                continue;
            dfs2(v, v);
        }
    };
    function<void()> kruskal = [&] () {
        sort(e.begin(), e.end());
        reverse(e.begin(), e.end());
        for (auto [u, v, w] : e) {
            u = find(u), v = find(v);
            if (u == v) continue;
            val[++ idx] = w;
            p[idx] = p[u] = p[v] = idx;
            g[idx].push_back(u); g[u].push_back(idx);
            g[idx].push_back(v); g[v].push_back(idx);
        }
        for (int i = 1; i <= idx; i ++ )
            if (!vis[i]) {
                int u = find(i); // 不能直接替换 u = find(u)
                dfs1(u, 0), dfs2(u, u);
            }
    };
    function<int(int, int)> lca = [&] (int u, int v) {
        while (top[u] != top[v]) {
            if (dep[top[u]] > dep[top[v]]) u = ff[top[u]];
            else v = ff[top[v]];
        }
        return dep[u] < dep[v] ? u : v;
    };
    kruskal();
    int q; cin >> q;
    while (q -- ) {
        int u, v;
        cin >> u >> v;
        if (find(u) != find(v)) cout << "-1\n";
        else cout << val[lca(u, v)] << "\n";
    }
}
```

割点/割边

割点

对于一个无向图，如果把一个点删除后这个图的极大连通分量数增加了，那么这个点就是这个图的割点（又称割顶）。

```cpp
void solve() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> e(n + 1);
    for (int i = 1; i <= m; i ++ ) {
        int u, v;
        cin >> u >> v;
        e[u].push_back(v);
        e[v].push_back(u);
    }
    int dfn = 0, cnt = 0;
    vector<int> num(n + 1), low(n + 1), iscut(n + 1);
    function<void(int, int)> dfs = [&](int u, int fa) {
        num[u] = low[u] = ++ dfn;
        int son = 0;
        for (auto v : e[u]) {
            if (!num[v]) {
                son ++;
                dfs(v, u);
                low[u] = min(low[u], low[v]);
                if (low[v] >= num[u] && u != fa) // 割点是 low[v] >= num[u], 割边 low[v] > num[u]
                    cnt += !iscut[u], iscut[u] = true;
            }
            else if (num[v] < num[u] && v != fa)
                low[u] = min(low[u], num[v]); // 用 num[v] 去更新， 否则会出现走了重复的路
        }
        if (u == fa && son >= 2)
            cnt += !iscut[u], iscut[u] = true;
    };
    for (int i = 1; i <= n; i ++ )
        if (!num[i])
            dfs(i, i);
    cout << cnt << "\n";
    for (int i = 1; i <= n; i ++ )
        if (iscut[i])
            cout << i << " ";
    cout << "\n";
}
```

割边

做法和割点差不多，只要改一处：$low\ v\ > num\ u$ 就可以了，而且不需要考虑根节点的问题。

割边是和是不是根节点没关系的，原来我们求割点的时候是指点v是不可能不经过父节点u为回到祖先节点（包括父节点），所以顶点u是割点。如果 $low\ v\ = num\ u$ 表示还可以回到父节点，如果顶点v不能回到祖先也没有另外一条回到父亲的路，那么 $u-v$这条边就是割边。

基环树

```
vector<vector<int>> g(N);
vector<int> d(N);
vector<int> vis(N);

void solve() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i ++ ) {
        int u, v;
        cin >> u >> v;
        g[u].push_back(v);
        g[v].push_back(u);
        d[u] ++, d[v] ++;
    }
    queue<int> q;
    for (int i = 1; i <= n; i ++ )
        if (d[i] == 1) vis[i] = 1, q.push(i);
    while (q.size()) {
        auto u = q.front();
        q.pop();
        for (auto v : g[u]) {
            if (vis[v]) continue;
            d[v] --;
            if (d[v] == 1) vis[v] = 1, q.push(v);
        }
    }
    for (int i = 1; i <= n; i ++ )
        if (!vis[i])
            cout << i << " ";
}
```

字符串

KMP

Knuth–Morris–Pratt 算法（简称 KMP 算法）用$O(n+m)$的时间以及$O(n)$的内存解决了问题。

```
void solve() {
    string s1, s2;
    cin >> s1 >> s2;
    int n = s1.size(), m = s2.size();
    s1 = " " + s1, s2 = " " + s2;
    vector<int> ne(m + 1);
    for (int i = 2, j = 0; i <= m; i ++ ) {
```

```
        while (j && s2[j + 1] != s2[i]) j = ne[j];
        if (s2[j + 1] == s2[i]) j ++;
        ne[i] = j;
    }
    for (int i = 1, j = 0; i <= n; i ++ ) {
        while (j && s2[j + 1] != s1[i]) j = ne[j];
        if (s2[j + 1] == s1[i]) j ++;
        if (j == m) {
            cout << i - m + 1 << endl;
            j = ne[j];
        }
    }
    for (int i = 1; i <= m; i ++ )
        cout << ne[i] << " ";
    cout << endl;
}
```

```
import java.math.BigInteger;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int x = input.nextInt();
        int y = input.nextInt();
        String z = input.next();
        if (z.equals("0")) {
            System.out.println("0");
        }
        else {
            BigInteger res = new BigInteger("0");
            int len = z.length();
            BigInteger t = new BigInteger("1");
            for (int i = len - 1; i >= 0; i -- ) {
                int temp = 0;
                if (z.charAt(i) >= '0' && z.charAt(i) <= '9') {
                    temp = z.charAt(i) - '0';
                }
                if (z.charAt(i) >= 'a' && z.charAt(i) <= 'z') {
                    temp = z.charAt(i) - 'a' + 36;
                }
                if (z.charAt(i) >= 'A' && z.charAt(i) <= 'Z') {
                    temp = z.charAt(i) - 'A' + 10;
                }
                res = res.add(t.multiply(BigInteger.valueOf(temp)));
                t = t.multiply(BigInteger.valueOf(x));
            }
            String ans = "";
            while (!res.equals(BigInteger.valueOf(0))) {
                BigInteger temp = res.mod(BigInteger.valueOf(y));
                Character c = '0';
                if (temp.intValue() >= 0 && temp.intValue() <= 9) {
                    c = (char) (temp.intValue() + '0');
                }
                if (temp.intValue() >= 10 && temp.intValue() <= 35) {
```

```
                    c = (char) (temp.intValue() - 10 + 'A');
                }
                if (temp.intValue() >= 36 && temp.intValue() <= 61) {
                    c = (char) (temp.intValue() - 36 + 'a');
                }
                ans += c;
                res = res.divide(BigInteger.valueOf(y));
            }
            int n = ans.length();
            for (int i = n - 1; i >= 0; i -- ) {
                System.out.print(ans.charAt(i));
            }
            System.out.println();
        }

    }
}

```