ACM 模板

钱智煊，黄佳瑞，车昕宇

2024 年 9 月 20 日

目录

1 图论 **3**

1.1 连通性相关 3

1.1.1 tarjan 3

1.1.2 tarjan 求 LCA 3

1.1.3 割点、割边 3

1.1.4 圆方树 3

1.2 同余最短路 3

2 数据结构 **4**

2.1 平衡树 4

2.1.1 无旋 Treap 4

2.1.2 Splay 4

2.2 动态树 4

2.3 珂朵莉树 5

2.4 李超线段树 5

3 字符串 **5**

3.1 后缀数组（与后缀树） 5

3.2 AC 自动机 6

3.3 回文自动机 6

3.4 Manacher 算法 6

3.5 KMP 算法与 border 理论 7

3.6 Z 函数 7

1 图论

1.1 连通性相关

1.1.1 tarjan

```
void tarjan(int x)
{
    dfn[x]=low[x]=++Time,sta[++tp]=x,ins[x]=true;
    for(int i=hea[x];i;i=nex[i])
    {
        if(!dfn[ver[i]]) tarjan(ver[i]),low[x]=min(low[x],low[ver[i]]);
        else if(ins[ver[i]]) low[x]=min(low[x],dfn[ver[i]]);
    }
    if(dfn[x]==low[x])
    {
        do
        {
            x=sta[tp],tp--,ins[x]=false;
        } while (dfn[x]!=low[x]);
    }
}
```

1.1.2 tarjan 求 LCA

实现均摊 $O(1)$。就是用 tarjan 按照顺序遍历子树的特点加上并查集即可。

```
inline void add_edge(int x,int y){ ver[++tot]=y,nex[tot]=hea[x],hea[x]=tot; }
inline void add_query(int x,int y,int d)
    { qver[++qtot]=y,qnex[qtot]=qhea[x],qhea[x]=qtot, qid[qtot]=d; }
int Find(int x){ return (fa[x]==x)?x:(fa[x]=Find(fa[x])); }
void tarjan(int x,int F)
{
    vis[x]=true;
    for(int i=hea[x];i;i=nex[i])
    {
        if(ver[i]==F) continue;
        tarjan(ver[i],x),fa[ver[i]]=x;
    }
    for(int i=qhea[x];i;i=qnex[i])
    {
        if(!vis[qver[i]]) continue;
        ans[qid[i]]=Find(qver[i]);
    }
}
for(int i=1;i<=n;i++) fa[i]=i;
for(int i=1,x,y;i<n;i++) x=rd(),y=rd(),add_edge(x,y),add_edge(y,x);
for(int i=1,x,y;i<=m;i++)
    x=rd(),y=rd(),add_query(x,y,i),add_query(y,x,i);
tarjan(s,s);
for(int i=1;i<=m;i++) printf("%d\n",ans[i]);
```

1.1.3 割点、割边

注意这里的 dfn 表示**不经过父亲**，能到达的最小的 dfn 。割点：

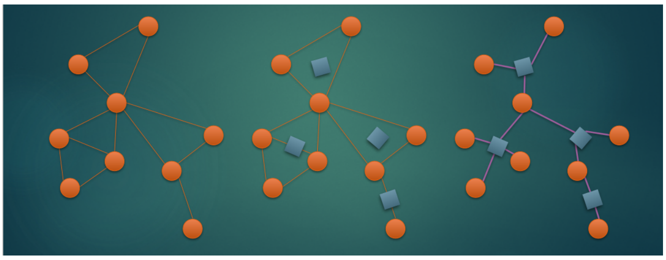
- 若 u 是根节点，当至少存在 2 条边满足 $low(v) \geq dfn(u)$ 则 u 是割点。
- 若 u 不是根节点，当至少存在 1 条边满足 $low(v) \geq dfn(u)$ 则 u 是割点。

割边：

- 当存在一条边条边满足 $low(v) > dfn(u)$ 则边 i 是割边。

注意：记录上一个访问的边时要记录边的编号，不能记录上一个过来的节点（因为会有重边）！！！或者在加边的时候特判一下，不过注意编号问题。（用输入顺序来对应数组中位置的时候，重边跳过，但是需要 tot+=2）

1.1.4 圆方树



圆方树会建立很多新的点，所以不要忘记**给数组开两倍**！

```
void tarjan(int u)
{
    dfn[u]=low[u]=++Time,sta[++tp]=u;
    for(int v:G[u])
    {
        if(!dfn[v])
        {
            tarjan(v),low[u]=min(low[u],low[v]);
            if(low[v]==dfn[u])
            {
                int hav=0; ++All;
                for(int x=0;x!=v;tp--) x=sta[tp],T[x].pb(All),T[All].pb(x),hav++;
                T[u].pb(All),T[All].pb(u),;
                siz[All]=++hav;
            }
        }
        else low[u]=min(low[u],dfn[v]);
    }
}
```

1.2 同余最短路

形如：

- 设问 1 ：给定 n 个整数，求这 n 个整数在 $h(h \leq 2^{63}-1)$ 范围内 **能拼凑出多少的其他整数（整数可以重复取）**。
- 设问 2 ：给定 n 个整数，求这 n 个整数 **不能拼凑出的最小（最大）的整数**。

设 x 为 n 个数中最小的一个，令 $ds[i]$ 为只通过增加其他 $n-1$ 种数能够达到的最低楼层 p ，并且满足 $p \equiv i \pmod{x}$ 。
对于 $n-1$ 个数与 x 个 $ds[i]$ ，可以如下连边：

```
for(int i=0;i<x;i++) for(int j=2;j<=n;j++) add(i,(i+a[j])%x,a[j]);
```

之后进行最短路，对于：

- 问在 h 范围内能够到达的**点的数量**：答案为（加一因为 i 本身也要计算）：

$$\sum_{i=0}^{x-1}[d[i] \leq h] \times \frac{h-d[i]}{x}+1$$

- 问不能达到的**最小的数**：答案为（i 一定时最小表示的数为 $d[i] = s \times x + i$ ，则 $(s-1) \times x + i$ **一定不能**被表示出来）：

$$\min_{i=1}^{x-1} \{d[i] - x\}$$

注意：ds 与 h 范围相同，一般也要开 long long！

2 数据结构

2.1 平衡树

2.1.1 无旋 Treap

```cpp
mt19937 mt(chrono::system_clock::now().
    time_since_epoch().count());
int rt;
struct FHQ {
    struct node {
        int val, pri, siz, ch[2], rv;
        #define val(x) nod[x].val
        #define pri(x) nod[x].pri
        #define siz(x) nod[x].siz
        #define ls(x) nod[x].ch[0]
        #define rs(x) nod[x].ch[1]
        #define rv(x) nod[x].rv
    } nod[N];
    int cnt;
    int create(int val) { return nod[++cnt] = {val, mt(),
        1}, cnt; }
    void pushup(int x) { siz(x) = siz(ls(x)) + siz(rs(x))
        + 1; }
    void reverse(int x) { rv(x) ^= 1, swap(ls(x), rs(x));
        }
    void pushdown(int x) { if (rv(x)) reverse(ls(x)),
        reverse(rs(x)), rv(x) = 0; }
    void split(int x, int siz, int &x1, int &x2) {
        if (!x) return x1 = x2 = 0, void();
        pushdown(x);
        if (siz(ls(x)) + 1 <= siz) x1 = x, split(rs(x),
            siz - siz(ls(x)) - 1, rs(x), x2);
        else x2 = x, split(ls(x), siz, x1, ls(x));
        pushup(x);
    } // x1 中存放了前 siz 个元素，x2 中存放了其余的元素。
    int merge(int x1, int x2) {
        if (!x1 || !x2) return x1 | x2;
        pushdown(x1), pushdown(x2);
        if (pri(x1) < pri(x2)) return rs(x1) =
            merge(rs(x1), x2), pushup(x1), x1;
        else return ls(x2) = merge(x1, ls(x2)),
            pushup(x2), x2;
    }
    int kth(int k) {
        int x = rt;
        while (x) {
            pushdown(x);
            if (siz(ls(x)) + 1 == k) return val(x);
            else if (k <= siz(ls(x))) x = ls(x);
            else k -= siz(ls(x)) + 1, x = rs(x);
        }
        return -1;
    } // 寻找第 k 位的元素。
} fhq;
```

2.1.2 Splay

```cpp
// 洛谷 P3391
// 理论上 splay 的每次操作以后都应该立刻做 splay 操作以保证
    均摊时间复杂度，但是某些地方又不能立刻做，例如下面的 kth
    。
int rt;
struct Splay {
    struct node {
        int val, siz, fa, ch[2], rv;
        #define ls(x) nod[x].ch[0]
        #define rs(x) nod[x].ch[1]
        #define val(x) nod[x].val
        #define siz(x) nod[x].siz
        #define fa(x) nod[x].fa
        #define rv(x) nod[x].rv
    } nod[N];
    int cnt;
    bool chk(int x) { return x == rs(fa(x)); }
    void reverse(int x) { rv(x) ^= 1, swap(ls(x), rs(x));
        }
    void pushup(int x) { siz(x) = siz(ls(x)) + 1 +
        siz(rs(x)); }
    void pushdown(int x) { if (rv(x)) reverse(ls(x)),
        reverse(rs(x)), rv(x) = 0; }
    void connect(int x, int fa, int son) { fa(x) = fa,
        nod[fa].ch[son] = x; }
    void rotate(int x) {
        int y = fa(x), z = fa(y), ys = chk(x), zs =
            chk(y), u = nod[x].ch[!ys];
        connect(u, y, ys), connect(y, x, !ys), connect(x,
            z, zs), pushup(y), pushup(x);
    }
    void pushall(int x) { if (fa(x)) pushall(fa(x));
        pushdown(x); }
    void splay(int x, int to) {
        pushall(x);
        while (fa(x) != to) {
            int y = fa(x);
            if (fa(y) != to) rotate(chk(x) == chk(y) ? y
                : x);
            rotate(x);
        }
        if (!to) rt = x;
    } // 将 x 伸展为 to 的儿子。
    void append(int val) {
        if (!rt) nod[rt = ++cnt] = {val, 1};
        else {
            int x = rt;
            while (rs(x)) pushdown(x), x = rs(x);
            splay(x, 0), nod[rs(x) = ++cnt] = {val, 1,
                x}, pushup(x);
        }
    }
    int kth(int k) {
        int x = rt;
        while (x) {
            pushdown(x);
            if (siz(ls(x)) + 1 == k) return x;
            else if (k <= siz(ls(x))) x = ls(x);
            else k -= siz(ls(x)) + 1, x = rs(x);
        }
        return -1;
    } // kth 做完以后不能立刻 splay ，因为需要提取区间。
    void reverse(int l, int r) {
        splay(kth(r + 2), 0), splay(kth(l), rt);
        reverse(rs(ls(rt))), pushup(ls(rt)), pushup(rt);
    } // 这里添加了前后两个哨兵，以避免额外的分类讨论。
} spl;
```

2.2 动态树

```cpp
// 洛谷 P3690
struct LCT {
```

```
    struct node {
        int rv, ch[2], fa, sm, val;
        #define ls(x) nod[x].ch[0]
        #define rs(x) nod[x].ch[1]
        #define fa(x) nod[x].fa
        #define sm(x) nod[x].sm
        #define rv(x) nod[x].rv
        #define val(x) nod[x].val
    } nod[N];
    // 根节点的父亲：链顶节点的树上父亲。
    // 其余节点的父亲：splay 中的父亲。

    bool chk(int x) { return rs(fa(x)) == x; }
    bool isroot(int x) { return nod[fa(x)].ch[chk(x)] != x; }
    void pushup(int x) { sm(x) = sm(ls(x)) ^ val(x) ^ sm(rs(x)); }
    void reverse(int x) { rv(x) ^= 1, swap(ls(x), rs(x)); }
    void pushdown(int x) {
        if (rv(x)) reverse(ls(x)), reverse(rs(x)), rv(x) = 0;
    }
    void connect(int x, int fa, int son) { fa(x) = fa, nod[fa].ch[son] = x; }
    void rotate(int x) {
        int y = fa(x), z = fa(y), ys = chk(x), zs = chk(y), u = nod[x].ch[!ys];
        if (isroot(y)) fa(x) = z;
        else connect(x, z, zs);
        connect(u, y, ys), connect(y, x, !ys), pushup(y), pushup(x);
    }
    void pushall(int x) { if (!isroot(x)) pushall(fa(x)); pushdown(x); }
    void splay(int x) {
        pushall(x);
        while (!isroot(x)) {
            if (!isroot(fa(x))) rotate(chk(x) == chk(fa(x)) ? fa(x) : x);
            rotate(x);
        }
    }
    void access(int x) { for (int y = 0; x; y = x, x = fa(x)) splay(x), rs(x) = y, pushup(x); }
    void makeroot(int x) { access(x), splay(x), reverse(x); }
    int findroot(int x) { access(x), splay(x); while (ls(x)) pushdown(x), x = ls(x); return splay(x), x; }
    void link(int x, int y) { makeroot(y); if (findroot(x) != y) fa(y) = x;}
    void split(int x, int y) { makeroot(y), access(x), splay(x); }
    void cut(int x, int y) { split(x, y); if (ls(x) == y) ls(x) = fa(y) = 0, pushup(x); }
    void modify(int x, int val) { splay(x), val(x) = val, pushup(x); }
    int sum(int x, int y) { split(x, y); return sm(x); }
    // 任何操作过后都应该立即 splay 以保证均摊复杂度。
} lct;
```

2.3 珂朵莉树

```
// 珂朵莉树的本质是颜色段均摊。若保证数据随机，可以证明其期望时间复杂度为 O(nlogn) 。
struct ODT {
    struct node {
        int l, r;
        mutable int val;
        node(int L, int R, int V) { l = L, r = R, val = V; }
        bool operator < (const node &rhs) const { return l < rhs.l; }
    };
    set<node> s;
    auto split(int x) {
        auto it = s.lower_bound(node(x, 0, 0));
        if (it != s.end() && it->l == x) return it;
        it--;
        int l = it->l, r = it->r, val = it->val;
        s.erase(it), s.insert(node(l, x - 1, val));
        return s.insert(node(x, r, val)).first;
    }
    void assign(int l, int r, int val) {
        // 此处须先 split(r + 1) 。因为若先 split(l) ，则后来的 split(r + 1) 可能致使 itl 失效。
        auto itr = split(r + 1), itl = split(l);
        s.erase(itl, itr), s.insert(node(l, r, val));
    }
    void perform (int l, int r) {
        auto itr = split(r + 1), itl = split(l);
        while (itl != itr) {
            // do something...
            itl++;
        }
    }
} odt;
```

2.4 李超线段树

```
const double eps = 1e-9;
struct line {
    double k, b;
    double operator () (const double &x) const {
        return k * x + b;
    }
};
bool cmp(double x, double y) {
    return y - x > eps;
}
struct SMT {
    line f[N << 2];
    void insert(int k, int l, int r, int x, int y, int k) {
        if (x <= l && r <= y) {

        }
    }
};
```

3 字符串

3.1 后缀数组（与后缀树）

```
const int N = 1e6 + 5;

char s[N];
int sa[N], rk[N], n, h[N];
// 后缀数组。h[i] = lcp(sa[i], sa[i - 1])
int rt, ls[N], rs[N], fa[N], val[N];
// 后缀树。实际上就是 height 数组的笛卡尔树。
// val[x] : x 与 fa[x] 对应的子串等价类的大小之差，也就是 x 贡献的本质不同子串数。

struct suffix {
    int k1[N], k2[N << 1], cnt[N], mx, stk[N], top;
    void radix_sort() {
        for (int i = 1; i <= mx; i++) cnt[i] = 0;
        for (int i = 1; i <= n; i++) cnt[k1[i]]++;
        for (int i = 1; i <= mx; i++) cnt[i] += cnt[i - 1];
```

```
        for (int i = n; i >= 1; i--) sa[cnt[k1[k2[i]]]--] = k2[i];
    } // 基数排序
    void sort() {
        mx = 'z';
        for (int i = 1; i <= n; i++) k1[i] = s[i], k2[i] = i;
        radix_sort();
        for (int j = 1; j <= n; j <<= 1) {
            int num = 0;
            for (int i = n - j + 1; i <= n; i++) k2[++num] = i;
            for (int i = 1; i <= n; i++) if (sa[i] > j) k2[++num] = sa[i] - j;
            radix_sort();
            for (int i = 1; i <= n; i++) k2[i] = k1[i];
            k1[sa[1]] = mx = 1;
            for (int i = 2; i <= n; i++) k1[sa[i]] = k2[sa[i]] == k2[sa[i - 1]] && k2[sa[i] + j] == k2[sa[i - 1] + j] ? mx : ++mx;
        }
    } // 后缀排序
    void height() {
        for (int i = 1; i <= n; i++) rk[sa[i]] = i;
        int k = 0;
        for (int i = 1; i <= n; i++) {
            if (k) k--;
            if (rk[i] == 1) continue;
            int j = sa[rk[i] - 1];
            while (i + k <= n && j + k <= n && s[i + k] == s[j + k]) k++;
            h[rk[i]] = k;
        }
    } // 计算 height 数组
    void build() {
        if (n == 1) return rt = 1, void();
        ls[2] = n + 1, rs[2] = n + 2, fa[ls[2]] = fa[rs[2]] = rt = stk[++top] = 2;
        for (int i = 3; i <= n; i++) {
            while (top && h[stk[top]] > h[i]) top--;
            int p = stk[top];
            if (top) ls[i] = rs[p], fa[rs[p]] = i, rs[p] = i, fa[i] = p;
            else ls[i] = rt, fa[rt] = i, rt = i;
            rs[i] = n + i, fa[rs[i]] = i, stk[++top] = i;
        }
        for (int i = 2; i <= n + n; i++) val[i] = (i > n ? n - sa[i - n] + 1 : h[i]) - h[fa[i]];
    } // 构建后缀树
} SA;
```

3.2 AC 自动机

```
struct ACAM {
    int ch[N][26], cnt, fail[N], vis[N];
    queue <int> q;
    void insert(char *s, int n, int id) {
        int x = 0;
        for (int i = 1; i <= n; i++) {
            int nw = s[i] - 'a';
            if (!ch[x][nw]) ch[x][nw] = ++cnt;
            x = ch[x][nw];
        }
        vis[x]++;
    }
    void build() {
        for (int i = 0; i < 26; i++) {
            if (ch[0][i]) q.push(ch[0][i]);
        }
        while (!q.empty()) {
            int x = q.front(); q.pop();
            for (int i = 0; i < 26; i++) {
                if (ch[x][i]) {
                    fail[ch[x][i]] = ch[fail[x]][i];
                    q.push(ch[x][i]);
                } else ch[x][i] = ch[fail[x]][i];
            }
        }
    }
    void match(char *s, int n) {
        int x = 0;
        for (int i = 1; i <= n; i++) {
            int nw = s[i] - 'a';
            x = ch[x][nw];
        }
    }
} AC;
```

3.3 回文自动机

```
struct PAM {
   int fail[N], ch[N][26], len[N], s[N], tot, cnt, lst;
    // fail : 当前节点的最长回文后缀。
    // ch : 在当前节点的前后添加字符，得到的回文串。
   PAM() {
      len[0] = 0, len[1] = -1, fail[0] = 1;
      tot = lst = 0, cnt = 1, s[0] = -1;
   }
   int get_fail(int x) {
      while (s[tot - 1 - len[x]] != s[tot]) x = fail[x];
      return x;
   }
   void insert(char c) {
      s[++tot] = c - 'a';
      int p = get_fail(lst);
      if(!ch[p][s[tot]]) {
         len[++cnt] = len[p] + 2;
         int t = get_fail(fail[p]);
         fail[cnt] = ch[t][s[tot]];
         ch[p][s[tot]] = cnt;
      }
      lst=ch[p][s[tot]];
   }
} pam;
```

3.4 Manacher 算法

```
char s[N], t[N];
// s[] : 原串，t[] : 加入分割字符的串，这样就只需考虑奇回文串了。
int mxp, cen, r[N], n;
// mxp : 最右回文串的右端点的右侧，cen : 最右回文串的中心，r[i] : 以位置 i 为中心的回文串半径，即回文串的长度一半向上取整。

void manacher() {
    t[0] = '~', t[1] = '#';
    // 在 t[0] 填入特殊字符，防止越界。
    int m = 1;
    for (int i = 1; i <= n; i++) {
        t[++m] = s[i], t[++m] = '#';
    }
    for (int i = 1; i <= m; i++) {
        r[i] = mxp > i ? min(r[2 * cen - i], mxp - i) : 1;
        // 若 i (cen, mxp)，则由对称性 r[i] 至少取 min(r[2 * cen - i], mxp - i)。否则直接暴力扩展。
        while (t[i + r[i]] == t[i - r[i]]) r[i]++;
        if (i + r[i] > mxp) mxp = i + r[i], cen = i;
    }
}
```

3.5 KMP 算法与 border 理论

```
char s[N], t[N];
int nex[N], n, m;

void kmp() {
    int j = 0;
    for (int i = 2; i <= n; i++) {
        while (j && s[j + 1] != s[i]) j = nex[j];
        if (s[j + 1] == s[i]) j++;
        nex[i] = j;
    }
}

void match() {
    int j = 0;
    for (int i = 1; i <= m; i++) {
        while (j && s[j + 1] != t[i]) j = nex[j];
        if (s[j + 1] == t[i]) j++;
        if (j == n) {
            //match.
            j = nex[j];
        }
    }
}
```

字符串的 border 理论：以下记字符串 S 的长度为 n 。

- 若串 S 具备长度为 m 的 border，则其必然具备长度为 $n-m$ 的周期，反之亦然。
- 弱周期性引理：若串 S 存在周期 p 、q ，且 $p+q \le n$ ，则 S 必然存在周期 $\gcd(p,q)$ 。
- 引理 1：若串 S 存在长度为 m 的 border T，且 T 具备周期 p ，满足 $2m-n \ge p$ ，则 S 同样具备周期 p 。
- 周期性引理：若串 S 存在周期 p 、q ，满足 $p+q-\gcd(p,q) \le n$ ，则串 S 必然存在周期 $\gcd(p,q)$ 。
- 引理 2：串 S 的所有 border 的长度构成了 $O(\log n)$ 个不交的等差数列。更具体的，记串 S 的最小周期为 p ，则其所有长度包含于区间 $[n \bmod p + p, n)$ 的 border 构成了一个等差数列。
- 引理 3：若存在串 S 、T ，使得 $2|T| \ge n$ ，则 T 在 S 中的所有匹配位置构成了一个等差数列。
- 引理 4：PAM 的失配链可以被划分为 $O(\log n)$ 个等差数列。

3.6 Z 函数

Z 函数用于求解字符串的每一个后缀与其本身的 lcp 。其思路和 manacher 算法基本一致，都是维护一个扩展过的最右端点和对应的起点，而当前点要么暴力扩展使最右端点右移，要么处在记录的起点和终点间，从而可以利用已有的信息快速转移。

```
char s[N];
int z[N];

void zfunc() {
    z[1] = n;
    int j = 0;
    for (int i = 2; i <= n; i++) {
        if (j && j + z[j] - 1 >= i) z[i] = min(z[i - j + 1], j + z[j] - i);
        while (i + z[i] <= n && s[i + z[i]] == s[1 + z[i]]) z[i]++;
        if (i + z[i] > j + z[j]) j = i;
    }
}
```