ACM 模板

钱智煊，黄佳瑞，车昕宇

2024 年 9 月 24 日

目录

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：

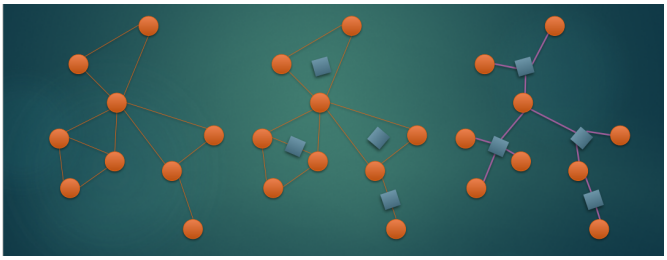
- 若 u 是根节点，当至少存在 2 条边满足 $low(v) \geq dfn(u)$ 则 u 是割点。
- 若 u 不是根节点，当至少存在 1 条边满足 $low(v) \geq dfn(u)$ 则 u 是割点。

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

```
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 平衡树合并

注意这里的合并是指合并两个值域相交的集合，并且这个操作应该只有 treap 支持。代码如下：

```
int join(int x1, int x2) {
    if (!x1 || !x2) return x1 + x2;
    if (pri(x1) > pri(x2)) swap(x1, x2);
    int ls, rs;
    split(x2, val(x1), ls, rs);
    ls(x1) = join(ls(x1), ls);
    rs(x1) = join(rs(x1), rs);
    return pushup(x1), x1;
}
```

可以证明，以上操作的总时间复杂度为 $O(n\log n\log V)$，其中 n 为元素总个数，V 为值域。而在特定的场景下，时间复杂度会更优。（例如若只支持合并与分裂操作，则时间复杂度为 $O(n\log n)$。）

2.1.3 Splay

```
// 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

```
// 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

2.4.1 修改与询问

```
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 {
    #define mid ((l + r) >> 1)
    #define ls (k << 1)
    #define rs ((k << 1) | 1)
    line f[N << 2];
    void insert(int k, int l, int r, int x, int y, line g) {
        if (x <= l && r <= y) {
            if (cmp(f[k](mid), g(mid))) swap(f[k], g);
            if (cmp(f[k](l), g(l))) insert(ls, l, mid, x, y, g);
```

```
            if (cmp(f[k](r), g(r))) insert(rs, mid + 1, r, x, y, g);
            return;
        }
        if (x <= mid) insert(ls, l, mid, x, y, g);
        if (y > mid) insert(rs, mid + 1, r, x, y, g);
    } // 插入 O(log^2n) : 定位到 O(logn) 个区间，每个区间 O(logn) 递归到叶子。
    int query(int k, int l, int r, int x) {
        double res = f[k](x);
        if (l == r) return res;
        if (x <= mid) return max(res, query(ls, l, mid, x));
        else return max(res, query(rs, mid + 1, r, x));
    }
} smt;
```

2.4.2 合并

```
// 声明 line 类
struct SMT {
    // 定义 mid
    struct node {
        line f;
        int ch[2];
        #define f(x) nod[x].f
        #define ls(x) nod[x].ls
        #define rs(x) nod[x].rs
    } nod[N];
    // 实现 insert 与 query
    int merge(int x, int y, int l, int r) {
        if (!x || !y) return x | y;
        if (l < r) {
            ls(x) = merge(ls(x), ls(y), l, mid);
            rs(x) = merge(rs(x), rs(y), mid + 1, r);
        }
        insert(x, l, r, f(y));
        // 注意此处的 insert 为全局插入。
        return x;
    } // 理论上李超树的合并是 O(nlog^2n) 的，但是我并不知道怎么证明。
}
```

2.5 二维树状数组

```
struct BIT {
    int sm[N][N][4]; // sm 是 sum 的缩写。
    int lowbit(int x) { return x & -x; }
    void add(int x, int y, int k) {
       int a = k, b = k * x, c = k * y, d = k * x * y;
       for (int i = x; i <= n; i += lowbit(i)) {
            for (int j = y; j <= m; j += lowbit(j)) {
               sm[i][j][0] += a;
               sm[i][j][1] += b;
               sm[i][j][2] += c;
               sm[i][j][3] += d;
          }
        }
    }
    int query(int x,int y) {
       int ret = 0, a = x * y + x + y + 1, b = y + 1, c = x + 1, d = 1;
       for (int i = x; i; i -= lowbit(i)) {
            for(int j = y; j; j -= lowbit(j)) {
               ret += sm[i][j][0] * a;
               ret -= sm[i][j][1] * b;
               ret -= sm[i][j][2] * c;
               ret += sm[i][j][3] * d;
           }
        }
       return ret;
    }
} bit;

// [a, c] * [b, d] + x : add(a, b, x), add(a, d + 1, -x), add(c + 1, b, -x), add(c + 1, d + 1, x);
// sum of [a, c] * [b, d] : query(c, d) - query(a - 1, d) - query(c, b - 1) + query(a - 1, b - 1);
```

2.6 虚树

```
int dfn[N];
vector<int> v, w; // v 为关键点，w 为虚树上的点。
bool cmp(int x, int y) { return dfn[x] < dfn[y]; }
void build() {
    sort(v.begin(), v.end(), cmp);
    w.push_back(v[0]);
    for (int i = 1; i < v.size(); i++) {
        w.push_back(lca(v[i - 1], v[i]));
        w.push_back(v[i]);
    } // 提取虚树中的所有点。
    sort(w.begin(), w.end(), cmp);
    w.erase(unique(w.begin(), w.end()), w.end());
    for (int i = 1; i < w.size(); i++) {
        connect(lca(w[i - 1], w[i]), w[i]);
    } // 构建虚树。即对于每一个虚树中的非根节点，向其父亲连边。
}
```

2.7 左偏树

```
struct heap {
    struct node {
        int val, dis, ch[2];
        #define val(x) nod[x].val
        #define ls(x) nod[x].ch[0]
        #define rs(x) nod[x].ch[1]
        #define dis(x) nod[x].dis
    } nod[N]; // dis : 节点到叶子的最短距离。
    int merge(int x, int y) {
        if (!x || !y) return x | y;
        if (val(x) > val(y)) swap(x, y);
        rs(x) = merge(rs(x), y);
        if (dis(ls(x)) < dis(rs(x))) swap(ls(x), rs(x));
        dis(x) = dis(rs(x)) + 1;
    } // 若根的 dis 为 d ，则左偏树至少包含 O(2^d) 个节点，因此 d=O(logn) 。
} hp;
```

特别的，若要求给定节点所在左偏树的根，须使用并查集。对于每个节点维护 rt[] 值，查找根时使用函数：

```
int find(int x) { return rt[x] == x ? x : rt[x] = find(rt[x]); }
```

在合并节点时，加入：

```
rt[x] = rt[y] = merge(x, y);
```

在弹出最小值时加入：

```
rt[ls(x)] = rt[rs(x)] = rt[x] = merge(ls(x), rs(x));
```

另外，删除过的点是不能复用的，因为这些点可能作为并查集的中转节点。

2.8 吉司机线段树

- 区间取 min 操作：通过维护区间次小值实现，即将区间取 min 转化为对区间最大值的加法，当要取 min 的值 v 大于次小值时停止递归。时间复杂度通过标记回收证明，即将区间最值视作标记，这样每次多余的递归等价于标记回收，总时间复杂度为 $O(m \log n)$。
- 区间历史最大值：通过维护加法标记的历史最大值实现。

```cpp
void Max(auto &x, auto y) { x = max(x, y); }
void Min(auto &x, auto y) { x = min(x, y); }
struct SMT {
    #define ls (k << 1)
    #define rs (k << 1 | 1)
    #define mid ((l + r) >> 1)
    struct node {
        int sm, mx, mx2, c, hmx, ad, ad2, had, had2;
        // 区间和，最大值，次大值，最大值个数，历史最大值，最大值加法标记，其余值加法标记，最大值的历史最大加法标记，其余值的历史最大加法标记。
        node operator + (const node &x) const {
            node t = *this;
            t.sm += x.sm, Max(t.hmx, x.hmx);
            if (t.mx > x.mx) Max(t.mx2, x.mx);
            else if (t.mx < x.mx) t.mx2 = max(t.mx, x.mx2), t.mx = x.mx, t.c = x.c;
            else Max(t.mx2, x.mx2), t.c += x.c;
            t.ad = t.ad2 = t.had = t.had2 = 0;
            return t;
        }
    } nod[N << 2];
    #define sm(x) nod[x].sm
    #define mx(x) nod[x].mx
    #define mx2(x) nod[x].mx2
    #define c(x) nod[x].c
    #define hmx(x) nod[x].hmx
    #define ad(x) nod[x].ad
    #define ad2(x) nod[x].ad2
    #define had(x) nod[x].had
    #define had2(x) nod[x].had2
    void pushup(int k) { nod[k] = nod[ls] + nod[rs]; }
    void add(int k, int l, int r, int ad, int ad2, int had, int had2) {
        Max(had(k), ad(k) + had), Max(had2(k), ad2(k) + had2), Max(hmx(k), mx(k) + had);
        sm(k) += c(k) * ad + (r - l + 1 - c(k)) * ad2, mx(k) += ad, ad(k) += ad, ad2(k) += ad2, mx2(k) += ad2;
    }
    void pushdown(int k, int l, int r) {
        int mx = max(mx(ls), mx(rs));
        if (mx(ls) == mx) add(ls, l, mid, ad(k), ad2(k), had(k), had2(k));
        else add(ls, l, mid, ad2(k), ad2(k), had2(k), had2(k));
        if (mx(rs) == mx) add(rs, mid + 1, r, ad(k), ad2(k), had(k), had2(k));
        else add(rs, mid + 1, r, ad2(k), ad2(k), had2(k), had2(k));
        ad(k) = ad2(k) = had(k) = had2(k) = 0;
    }
    void build(int k, int l, int r) {
        if (l == r) return nod[k] = {a[l], a[l], -inf, 1, a[l]}, void();
        build(ls, l, mid), build(rs, mid + 1, r);
        pushup(k);
    }
    void add(int k, int l, int r, int x, int y, int v) {
        if (x <= l && r <= y) return add(k, l, r, v, v, v, v);
        pushdown(k, l, r);
        if (x <= mid) add(ls, l, mid, x, y, v);
        if (y > mid) add(rs, mid + 1, r, x, y, v);
        pushup(k);
    }
    void Min(int k, int l, int r, int x, int y, int v) {
        if (v >= mx(k)) return;
        if (x <= l && r <= y && v > mx2(k)) return add(k, l, r, v - mx(k), 0, v - mx(k), 0), void();
        pushdown(k, l, r);
        if (x <= mid) Min(ls, l, mid, x, y, v);
        if (y > mid) Min(rs, mid + 1, r, x, y, v);
        pushup(k);
    }
    node query(int k, int l, int r, int x, int y) {
        if (x <= l && r <= y) return nod[k];
        pushdown(k, l, r);
        if (y <= mid) return query(ls, l, mid, x, y);
        else if (x > mid) return query(rs, mid + 1, r, x, y);
        else return query(ls, l, mid, x, y) + query(rs, mid + 1, r, x, y);
    }
} smt;
```

2.9 树分治

熟知序列分治的过程是选取恰当的分治点并考虑所有跨过分治点的区间。而树分治的过程也是类似的，以点分治为例，每一次选择当前联通块的重心作为分治点，然后考虑所有跨越分治点的路径，并对分割出的联通块递归。

若要处理树上邻域问题，可以考虑建出点分树。处理点 x 的询问时，只需考虑 x 在点分树上到根的路径，每一次加上除开 x 所在子树的答案即可。

```cpp
int siz[N], rt, tot, dfa[N], mxp[N], vis[N], mxd;
// rt: 当前重心，tot: 联通块大小，dfa: 点分树父亲，mxp: 最大子树大小，mxd: 最大深度。
void find(int x, int fa) {
    siz[x] = 1, mxp[x] = 0;
    for (int i = head[x]; i; i = e[i].next)
    {
        if (e[i].b != fa && !vis[e[i].b])
        {
            find(e[i].b, x);
            siz[x] += siz[e[i].b];
            mxp[x] = max(mxp[x], siz[e[i].b]);
        }
    }
    mxp[x] = max(mxp[x], tot - siz[x]);
    rt = mxp[rt] > mxp[x] ? x : rt;
}
void get_dis(int x, int fa, int dep) {
    mxd = max(mxd, dep);
    for (int i = head[x]; i; i = e[i].next)
        if (e[i].b != fa && !vis[e[i].b])
            get_dis(e[i].b, x, dep + 1);
}
// bit[x][0]: 存储了以 x 为重心时 x 的子树内的信息。
// bit[x][1]: 存储了以 dfa[x] 为重心时 x 的子树内的信息。
void divide(int x, int lim) {
    vis[x] = 1, mxd = 0, get_dis(x, 0, 0);
    bit[x][0].build(mxd);
    if (dfa[x]) bit[x][1].build(lim);
    for (int i = head[x]; i; i = e[i].next) {
        if (!vis[e[i].b]) {
            rt = 0, tot = siz[e[i].b] < siz[x] ? siz[e[i].b] : lim - siz[x];
            find(e[i].b, x);
            dfa[rt] = x, divide(rt, tot);
        }
    }
}
void modify(int ct, int v) {
```

```
    int x = ct;
    while (x) {
        bit[x][0].add(dis(x, ct), v); // dis(x, y): 返回 x 到 y 的距离。
        if (dfa[x]) bit[x][1].add(dis(dfa[x], ct), v);
        x = dfa[x];
    }
} // 将点 ct 的值加上 v 。
int query(int ct, int k) {
    int x = ct, res = 0, lst = 0;
    while (x) {
        int d = dis(x, ct);
        if (d <= k) {
            res += bit[x][0].query(k - d);
            if (lst) res -= bit[lst][1].query(k - d);
        }
        lst = x, x = dfa[x];
    }
    return res;
}
void init() {
    // dis 的预处理。
    mxp[rt = 0] = tot = n, find(1, 0), divide(rt, tot);
}
```

```
        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 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;
```

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;
    }
}
```

4 多项式

4.1 FFT

```
struct comp {
    double x, y;
    comp(double X = 0, double Y = 0) { x = X, y = Y; }
    comp operator + (const comp &rhs) const { return comp(x + rhs.x, y + rhs.y); }
    comp operator - (const comp &rhs) const { return comp(x - rhs.x, y - rhs.y); }
    comp operator * (const comp &rhs) const { return comp(x * rhs.x - y * rhs.y, x * rhs.y + y * rhs.x); }
    comp operator / (const comp &rhs) const {
        double t = rhs.x * rhs.x + rhs.y * rhs.y;
        return {(x * rhs.x + y * rhs.y) / t, (y * rhs.x - x * rhs.y) / t};
    }
};
const double pi = acos(-1);
int lim, tr[N];
void adjust(int n) { // n: 多项式的次数。
```

```
    lim = 1;
    while (lim <= n) lim <<= 1;
    for (int i = 0; i < lim; i++) tr[i] = (tr[i >> 1] >> 1) | ((i & 1) ? lim >> 1 : 0);
} // 准备蝶形变换。
void fft(comp *f, int op) { // op: 1 为 dft , -1 为 idft 。
    for (int i = 0; i < lim; i++) if (tr[i] < i) swap(f[tr[i]], f[i]);
    for (int l = 1; l < lim; l <<= 1) {
        comp w1(cos(2 * pi / (l << 1)), sin(2 * pi / (l << 1)) * op);
        for (int i = 0; i < lim; i += l << 1) {
            comp w(1, 0);
            for (int j = i; j < i + l; j++, w = w * w1) {
                comp x = f[j], y = f[j + l] * w;
                f[j] = x + y, f[j + l] = x - y;
            }
        }
    }
    if (op == -1) {
        for (int i = 0; i < lim; i++) f[i].x /= lim;
    }
}
```

4.2 NTT

```
const int mod = 998244353, G = 3, iG = 332748118;
int tr[N], lim;
void adjust(int n) { // n: 多项式的次数。
    lim = 1;
    while (lim <= n) lim <<= 1;
    for (int i = 0; i < lim; i++) tr[i] = (tr[i >> 1] >> 1) | ((i & 1) ? lim >> 1 : 0);
} // 准备蝶形变换。
void ntt(int *f, int op) { // op: 1 为 dft , -1 为 idft 。
    for (int i = 0; i < lim; i++) if (tr[i] < i) swap(f[tr[i]], f[i]);
    for (int l = 1; l < lim; l <<= 1) {
        int w1 = qpow(op == 1 ? G : iG, (mod - 1) / (l << 1)); // qpow: 快速幂。
        for (int i = 0; i < lim; i += l << 1) {
            for (int j = i, w = 1; j < i + l; j++, (w *= w1) %= mod) {
                int x = f[j], y = f[j + l] * w % mod;
                f[j] = (x + y) % mod, f[j + l] = (x - y) % mod;
            }
        }
    }
    if (op == -1) {
        int iv = qpow(lim); // 算逆元。
        for (int i = 0; i < lim; i++) (f[i] *= iv) %= mod;
    }
}
```

4.3 集合幂级数

4.3.1 并卷积、交卷积与子集卷积

集合并等价于二进制按位或，因此并卷积的计算实际上就是做高维前缀和以及差分，也被称作莫比乌斯变换。

```
void fmt(int *f, int op) { // op: 1 为 dft , -1 为 idft 。
    for (int l = 1; l < lim; l <<= 1) {
        for (int i = 0; i < lim; i += l << 1) {
            for (int j = i; j < i + l; j++) {
                f[j + l] += f[j] * op;
            }
        }
    }
}
```

而集合交卷积则对应后缀和。

```
void fmt(int *f, int op) { // op: 1 为 dft , -1 为 idft 。
    for (int l = 1; l < lim; l <<= 1) {
        for (int i = 0; i < lim; i += l << 1) {
            for (int j = i; j < i + l; j++) {
                f[j] += f[j + l] * op;
            }
        }
    }
}
```

子集卷积则较为特殊，为了使得产生贡献的集合没有交集，考虑引入代表集合大小的占位符。这样只需做 n 次 FMT，再枚举长度做 n^2 次卷积。因为 FMT 具备线性性，所以最后只需做 n 次 iFMT 即可。

```
void multi(int *f, int *g, int *h) { // 对 f 和 g 做子集卷积，答案为 h 。
    static int a[n][lim], b[n][lim], c[n][lim]; // lim = 1 << n
    memset(a, 0, sizeof a);
    memset(b, 0, sizeof b);
    memset(c, 0, sizeof c);
    for (int i = 0; i < lim; i++) a[pcnt[i]][i] = f[i]; // pcnt[i] = popcount(i)
    for (int i = 0; i < lim; i++) b[pcnt[i]][i] = g[i];
    for (int i = 0; i <= n; i++) fmt(a[i], 1), fmt(b[i], 1);
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= i; j++)
            for (int k = 0; k < lim; k++)
                c[i][k] += a[j][k] * b[i - j][k];
    for (int i = 0; i <= n; i++) fmt(c[i], -1);
    for (int i = 0; i < lim; i++) h[i] = c[pcnt[i]][i];
}
```

特别的，子集卷积等价于 n 元保留到一次项的线性卷积。

4.3.2 对称差卷积

集合对称差等价于按位异或，而异或卷积则等价于 n 元模 2 的循环卷积，因此，FWT 实质上和 n 元 FFT 没有什么区别。

```
void fwt(int *f, int op) { // op: 1 为 dft , -1 为 idft 。
    for (int l = 1; l < lim; l <<= 1) {
        for (int i = 0; i < lim; i += l << 1) {
            for (int j = i; j < i + l; j++) {
                int x = f[j], y = f[j + l];
                f[j] = x + y, f[j + l] = x - y;
                if (op == -1) f[j] /= 2, f[j + l] /= 2;
                // 模意义下改成乘逆元。
            }
        }
    }
}
```

4.4 多项式全家桶

```
const int N = 5000005;
const long long mod = 998244353;
#define int long long
namespace poly {
    const long long G = 3, iG = 332748118;
    int tr[N], lim;
```

```
    int qpow(int a, int b = mod - 2) {
        int res = 1;
        while (b) {
            if (b & 1) (res *= a) %= mod;
            (a *= a) %= mod, b >>= 1;
        }
        return res;
    }
    void adjust(int n) {
        lim = 1;
        while (lim <= n) lim <<= 1;
        for (int i = 0; i < lim; i++) tr[i] = (tr[i >> 1] >> 1) | ((i & 1) ? lim >> 1 : 0);
    } // 准备蝶形变换
    void copy(int *f, int *g, int n) { for (int i = 0; i <= n; i++) f[i] = g[i]; }
    void clear(int *f, int n) { for (int i = 0; i <= n; i++) f[i] = 0; }
    void erase(int *f, int l, int r) { for (int i = l; i <= r; i++) f[i] = 0; }
    void reverse(int *f, int n) { for (int i = 0; i <= n / 2; i++) swap(f[i], f[n - i]); }
    void integral(int *f, int n) {
        for (int i = n + 1; i >= 1; i--) f[i] = f[i - 1] * qpow(i) % mod;
        f[0] = 0;
    } // 对 n 次多项式 f 求积分，默认常数项为 0 。
    void dif(int *f, int n) {
        for (int i = 0; i < n; i++) f[i] = f[i + 1] * (i + 1) % mod;
        f[n] = 0;
    } // 对 n 次多项式 f 求导。
    void ntt(int *f, int op) { // op: 1 为 dft , -1 为 idft 。
        for (int i = 0; i < lim; i++) if (tr[i] < i) swap(f[tr[i]], f[i]);
        for (int l = 1; l < lim; l <<= 1) {
            int w1 = qpow(op == 1 ? G : iG, (mod - 1) / (l << 1)); // qpow: 快速幂。
            for (int i = 0; i < lim; i += l << 1) {
                for (int j = i, w = 1; j < i + l; j++, (w *= w1) %= mod) {
                    int x = f[j], y = f[j + l] * w % mod;
                    f[j] = (x + y) % mod, f[j + l] = (x - y) % mod;
                }
            }
        }
        if (op == -1) {
            int iv = qpow(lim); // 算逆元。
            for (int i = 0; i < lim; i++) (f[i] *= iv) %= mod;
        }
    }
    void multiply(int *h, int *f, int n, int *g, int m) {
        static int a[N], b[N];
        copy(a, f, n), copy(b, g, m);
        adjust(n + m);
        ntt(a, 1), ntt(b, 1);
        for (int i = 0; i < lim; i++) h[i] = a[i] * b[i] % mod;
        ntt(h, -1);
        clear(a, lim - 1), clear(b, lim - 1);
    } // 计算 f 与 g 的积，存放在 h 中，f 与 g 不变。
    void inverse(int *f, int *g, int n) {
        static int t[N];
        if (!n) return f[0] = qpow(g[0]), void();
        inverse(f, g, n >> 1);
        adjust(2 * n), copy(t, g, n);
        ntt(t, 1), ntt(f, 1);
        for (int i = 0; i < lim; i++) f[i] = f[i] * (2 - f[i] * t[i] % mod) % mod;
        ntt(f, -1), erase(f, n + 1, lim - 1), clear(t, lim - 1);
    } // 计算 g 的 n 次逆，存放在 f 中，g 不变。不要让 f 和 g 为同一个数组。
    void ln(int *f, int *g, int n) {
        static int t[N];
        copy(t, g, n), inverse(f, g, n), dif(t, n);
        adjust(n * 2), ntt(t, 1), ntt(f, 1);
        for (int i = 0; i < lim; i++) (f[i] *= t[i]) %= mod;
        ntt(f, -1), integral(f, n);
        erase(f, n + 1, lim - 1), clear(t, lim - 1);
    } // 要求 g[0] = 1 。
    void exp(int *f, int *g, int n) {
        static int t[N];
        if (!n) return f[0] = 1, void();
        exp(f, g, n >> 1), ln(t, f, n);
        for (int i = 0; i <= n; i++) t[i] = (g[i] - t[i]) % mod;
        t[0]++;
        adjust(n * 2), ntt(f, 1), ntt(t, 1);
        for (int i = 0; i < lim; i++) (f[i] *= t[i]) %= mod;
        ntt(f, -1), clear(t, lim - 1), erase(f, n + 1, lim - 1);
    } // 要求 g[0] = 0 。
    void pow(int *f, int *g, int n, int k) {
        static int t[N];
        ln(t, g, n);
        for (int i = 0; i <= n; i++) (t[i] *= k) % mod;
        exp(f, t, n);
    } // 要求 g[0] = 1 。
    void divide(int *q, int *r, int *f, int n, int *g, int m) {
        static int a[N], b[N], c[N];
        copy(a, f, n), copy(b, g, m);
        reverse(a, n), reverse(b, m);
        inverse(c, b, n - m);
        multiply(q, a, n - m, c, n - m);
        reverse(q, n - m);
        multiply(a, g, m, q, n - m);
        for (int i = 0; i < m; i++) r[i] = (f[i] - a[i] + mod) % mod;
    } // 多项式带余除法，其中 q 为商，r 为余数。
}
using namespace poly;
```