ACM Template

DUT ACM Lab

2022 年 4 月 17 日

目录

第一章　STL 使用

1.1　set-multiset-map

1.1.1　set

```
set<T> e; // 定义 set
e.clear(); // 清空 set
e.insert(); // 插入 val , 返回指向插入点的迭代器
e.size(); // 返回 set 大小
e.lower_bound(); // iter , 大于等于
e.upper_bound(); // iter , 大于
e.find(); // iter
```

1.1.2　multiset

```
mlutiset<T> e;
// the same to up
e.equal_range(); // pair<iter,iter>
e.count();
```

1.1.3　map

```
map<T1, T2> e;
// the same to up
e[T1] = T2;
```

1.2　unrodered STL

第二章　图论

2.1　二分图

2.1.1　二分图最大匹配

2.1.1.1　实现

第一种做法

匈牙利算法，复杂度 $\Theta(nm)$

```
const int mx_n = 1005;
bool mp[mx_n][mx_n];
bool vis[mx_n];
int pre[mx_n];

bool dfs(cint loc) {
    for(int i=n+1; i<=n+m; i++) {
        if(mp[loc][i] && !vis[i]) {
            vis[i] = 1;
            if(!pre[i] || dfs(pre[i])) {
                pre[i] = loc;
                return 1;
            }
        }
    }
    return 0;
}

int main() {
    int ans = 0;
    for(int i=1; i<=n; i++) {
        memset(vis, 0, sizeof vis);
        ans += dfs(i);
    }
    cout << ans << endl;
    return 0;
}
```

第二种做法

转化为网络流，复杂度依赖选择

2.1.1.2 性质

最大独立集 = n - 最大匹配
最小点覆盖 = n - 最大独立集

2.2 树上问题

2.2.1 树的重心

2.2.1.1 实现

对于树上的每一个点，计算其所有子树中最大的子树节点数，这个值最小的点就是这棵树的重心。

```
void dfs(cint loc, cint fa) {
    int pre = 0;
    son[loc] = 1;
    for(int v: to[loc]) {
        if(v != fa) {
            dfs(v, loc);
            pre = max(pre, son[v]);
            son[loc] += son[v];
        }
    }
    pre = max(pre, n-son[loc]);
    if(pre < st) {
        st = pre; # 最大儿子的最小值
        ans = loc; # 最大儿子的编号
    }
}
```

2.2.1.2 性质

1. 一棵树最多有两个重心，且相邻
2. 以树的重心为根时，所有子树的大小都不超过整棵树大小的一半
3. 在一棵树上添加或删除一个叶子，那么它的重心最多只移动一条边的距离
4. 把两棵树通过一条边相连得到一棵新的树，那么新的树的重心在连接原来两棵树的重心的路径上
5. 树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样

2.2.1.3 一点证明

首先证明如果树有两个重心，则它们必相邻
设两点分别为 a ，b ，且它们之间的简单路径经过的点的个数不为 0 ，即它们不相邻

不妨认为 a 在树中的深度大于 b

那么，点 a 向上的子树大小一定**大于**点 b 向上的子树，同时**大于**点 b 向下不经过点 a 的子树

同理，点 b 向下经过点 a 的子树一定**大于**点 a 向下的子树

所以，最大值仅由这两棵子树决定，而只要两点不相邻，上述总是成立的

不难发现，这两种子树，在 a 或 b 沿着两点间简单路径移动时会减小，不符合重心的定义

再证最多只有两个重心

显然，如果树的重心大于两个，至少有一对重心无法相邻

2.2.2 点分治

2.2.2.1 思路

如果在统计树上信息时，可以将子树内的信息单独统计，将多个子树的信息合并统计，那么可以考虑树分治。

如果对于每一次分治，复杂度为 $\Theta(N)$ ，那么 k 次递归的复杂度就是 $\Theta(kN)$

如果能保证递归次数为 log 级别，那么复杂度就会是 $\Theta(N\log N)$ ，而从重心分治就可以保证最多递归 $\log N$ 次

同时可以发现,在递归时保存所有以重心为根的子树的信息的空间复杂度也是 $\Theta(N\log N)$ 的

不会受到影响的信息有简单路径

会受到影响的信息有 LCA

如果题目需要求的信息会受到根节点选取的影响，还是不要使用点分治为好

2.2.2.2 实现

预定义部分

```
int h[10010], nx[20020], to[20020], w[20020], cnt_; // 链式前向星数组
int son[10010]; // 经过处理后的每个点的儿子个数
bool vis[10010]; // 该点是否在分治时作为子树的根
int id; // 当前所处理的树的重心
int snode; // 当前所处理树的节点数量

void add(cint f, cint t, cint co) {
    nx[++cnt_] = h[f];
    h[f] = cnt_;
    to[cnt_] = t;
    w[cnt_] = co;
}
```

统计以某点为根且不跨越其余重心的子树大小

```
int gsiz(cint loc, cint fa) {
    int sum = 1;
    for(int i=h[loc]; i; i=nx[i])
```

```
        if(to[i] != fa && !vis[to[i]]) {
            sum += gsiz(to[i], loc);
        }
    return sum;
}
```

寻找树的重心

```
void gp(cint loc, cint fa) {
    int pre = 0;
    son[loc] = 1;
    for(int i=h[loc]; i; i=nx[i])
        if(to[i] != fa && !vis[to[i]]) {
            gp(to[i], loc);
            pre = max(pre, son[to[i]]);
            son[loc] += son[to[i]];
            if(id) return;
        }
    pre = max(pre, snode - son[loc]);
    // 树的重心可能有两个，此处任取了一个
    if(pre <= snode/2) {
        id = loc;
        return;
    }
}
```

统计跨越重心的答案

```
void check(cint loc, cint fa) {
    // 统计跨越重心的答案
    for(int i=h[loc]; i; i=nx[i])
        if(to[i] != fa && !vis[to[i]]) {
            check(to[i], loc);
        }
}
```

合并子树

```
void update(cint loc, cint fa) {
    // 合并子树
    for(int i=h[loc]; i; i=nx[i]) {
        if(to[i] != fa && !vis[to[i]]) {
            update(to[i], loc);
        }
    }
}
```

解决问题

```
void sol(cint loc) {
    vis[loc] = 1;
    // 初始化计算答案与合并子树时需要用到的东西
    for(int i=h[loc]; i; i=nx[i]) {
        if(!vis[to[i]]) {
            check(to[i], loc);
            update(to[i], loc);
        }
    }
    for(int i=h[loc]; i; i=nx[i]) {
        if(!vis[to[i]]) {
            snode = gsiz(to[i], loc);
            id = 0;
            gp(to[i], loc);
            sol(id);
        }
    }
}
```

主函数里的一点东西

```
int main() {
    snode = n;
    gp(1, 1);
    sol(id);
}
```

2.2.3 边分治

2.2.4 点分树

2.2.5 树上启发式合并

2.2.5.1 思路

当子树可以单独处理，且子树信息转移到父节点较为容易时可以考虑

任意一条路径上轻边个数不超过 $\log N$

每一条轻边连接的轻子树会额外访问子树中所有的点一次，那么每个点至多被额外访问 $\Theta(\log N)$ 次

理论复杂度 $\Theta(N \log N)$

2.2.5.2 实现

预定义部分

```
vector<int> to[100100]; // 邻接表
int son[100100]; // 子树大小
```

```
3 int bson[100100]; // 重儿子
```

寻找重儿子

```
1  void fd_son(cint loc, cint fa) {
2      son[loc] = 1;
3      for(int v: to[loc]) {
4          if(v != fa) {
5              fd_son(v, loc);
6              son[loc] += son[v];
7              if(son[v] > son[bson[loc]]) bson[loc] = v;
8          }
9      }
10 }
```

递归主体

```
1  void clear() {
2      // do somethings
3  }
4
5  void sol(cint loc, cint fa) {
6      for(int v: to[loc]) {
7          if(v != fa && v != bson[loc]) {
8              sol(v, loc);
9              clear(); // 清空函数
10         }
11     }
12     if(bson[loc]) sol(bson[loc], loc);
13     for(int v: to[loc]) {
14         if(v != fa && v != bson[loc]) {
15             check(v, loc, a[loc]);
16             update(v, loc);
17         }
18     }
19     // 此处注意插入当前节点
20 }
```

统计答案

```
1  void cacu(cint r, cint x) {
2      // do somethings
3  }
4
5  void check(cint loc, cint fa, cint co) {
6      // cacu
7      for(int v: to[loc]) {
8          if(v != fa) check(v, loc, co);
```

```
    }
}
```

合并子树

```
void ins(cint r, cint x) {
    // do somethings
}

void update(cint loc, cint fa) {
    // ins
    for(int v: to[loc]) {
        if(v != fa) update(v, loc);
    }
}
```

主函数的一些部分

```
int main() {
    fd_son(1, 1);
    sol(1, 1);
}
```

2.3 网络流

2.3.1 最大流算法

2.3.1.1 增广路算法

2.3.1.1.1 dinic 复杂度为 $\Theta(n^2m)$ ，特别地，对于二分图匹配复杂度为 $\Theta(m\sqrt{n})$

```
const int mxn = 202; // 最大点数
const int mxm = 5005; // 最大边数
struct dinic {
    int s, t; // 源，汇
    ll w[mxm*2];
    int h[mxn], nx[mxm*2], to[mxm*2], cnt=1; // 链式前向星，cnt=1 方便找到反边
    int cur[mxn]; // 当前弧优化辅助数组
    int label[mxn]; // 分层图标号
    void clear_map() { cnt = 1; for(int i=1; i<=mxn; i++) h[i] = 0; }
    void add(cint f, cint t, cint co) {
        nx[++cnt] = h[f]; h[f] = cnt; to[cnt] = t; w[cnt] = co;
        nx[++cnt] = h[t]; h[t] = cnt; to[cnt] = f; w[cnt] = 0;
    }
    bool bfs() {
```

```
15          memset(label, 0x3f, sizeof label); // 初始化标号
16          memcpy(cur, h, sizeof h); // 当前弧优化
17          queue<int> q;
18          q.push(s);
19          label[s] = 0;
20          while(!q.empty()) {
21              int r = q.front();
22              q.pop();
23              for(int i=h[r]; i; i=nx[i])
24                  if(w[i] && label[to[i]] > label[r]+1) {
25                      label[to[i]] = label[r] + 1;
26                      q.push(to[i]);
27                  }
28          }
29          return label[t] < label[mxn-1];
30      }
31      ll dfs(cint loc, ll cap) {
32          if(loc == t) return cap;
33          ll sum = 0;
34          for(int i=cur[loc]; i; i=nx[i]) {
35              cur[loc] = i;
36              if(label[to[i]] == label[loc] + 1) {
37                  ll d = dfs(to[i], min(cap, w[i]));
38                  if(d) { w[i] -= d; w[i^1] += d; cap -= d; sum += d;
                        }
39                  if(!cap) break;
40              }
41          }
42          return sum;
43      }
44      ll max_flow(cint s, cint t) {
45          this->s = s;
46          this->t = t;
47          ll flow = 0;
48          while(bfs()) { flow += dfs(s, inf_ll); }
49          return flow;
50      }
51  };
```

2.3.1.2 预流推进算法

2.3.2 费用流算法

2.3.3 上下界网络流

2.3.4 其余变形

2.4 生成树

2.4.1 最小生成树

2.4.1.1 相关

1. 最小生成树是原图中边权和最小的生成树
2. 最小生成树中任意两点的路径边权的最大值一定是该两点间所有路径中边权最大值最小的

2.4.1.2 kruskal 算法（避圈法）

2.4.1.3 prim 算法（边割法）

2.4.2 生成树计数

2.4.2.1 prufer 序列

2.4.2.2 cayley 定理

K_n 的生成树个数为 n^{n-2}

2.4.2.3 扩展 cayley 定理

n 个点组成的 s 棵树的森林的生成数个数为 $F(n,s)=sn^{n-s-1}$

2.4.2.4 完全二部图的生成数个数

K_{n_1,n_2} 的生成树的个数为 $n_1^{n_2-1}n_2^{n_1-1}$

第三章　数论

3.1　逆元

3.1.1　逆元的一些实现

注意，不是所有时候都有逆元

3.1.1.1　单个数的逆元

```
ll ksm(ll m, int c) {
    ll ans = 1;
    while(c) {
        if(c&1) ans = (ans*m) % mod;
        c >>= 1;
        m = (m*m) % mod;
    }
    return ans;
}

ll inv(ll x) { return ksm(x, mod-2); }
```

3.1.1.2　阶乘的线性逆元

```
ll ksm(ll m, int c) {
    ll ans = 1;
    while(c) {
        if(c&1) ans = (ans*m) % mod;
        c >>= 1;
        m = (m*m) % mod;
    }
    return ans;
}
void sol_inv() {
    fac[0] = 1;
    for(int i=1; i<=mx_n; i++) fac[i] = fac[i-1] * i % mod;
    inv[mx_n] = ksm(fac[mx_n], mod-2);
    for(int i=mx_n-1; i; i--) inv[i] = inv[i+1] * (i+1) % mod;
```

```
}
```

3.1.1.3 1 到 n 的线性逆元

```
inv[1] = 1;
for(int i=2; i<=n; i++) inv[i] = (mod-mod/i) * inv[mod%i] % mod;
```

3.1.2 逆元存在性

3.1.2.1 p 与 b 不互质

逆元不存在

3.1.2.2 模数 p 为质数

根据费马小定理，数 b 的逆元为 b^{p-2}

3.1.2.3 模数 p 不为质数

如果 p 与 b 互质，数 b 的逆元为 $b^{\varphi(p)-1}(mod\ p)$

3.2 筛法

3.2.1 埃氏筛（素数）

复杂度 $\Theta(N \log \log N)$

3.2.2 欧拉筛（素数，积性函数）

复杂度 $\Theta(N)$

3.2.2.1 素数

每一个合数可以被唯一的分解为一个最小质数和另一个合数的乘积
正确性证明每一个数仅被分解到一次且每一个数都被分解到就好
同时得到每个数的最小因数

```
const int mx_n = 100000000;
int vis[mx_n+1000];
int prim[mx_n+1000], cnt;

void liner_sieve(cint x) {
    int rt = 0;
    for(int i=2; i<=x; i++) {
        if(!vis[i]) {
            prim[++cnt] = i;
            vis[i] = i;
```

```
        }
        for(int j=1; j<=cnt; j++) {
            if(1ll*prim[j]*i > x) break;
            if(prim[j] > vis[i]) break;
            vis[prim[j]*i] = prim[j];
        }
    }
}
```

省空间的写法

```
const int mx_n = 100000000;
int n, q;
bool vis[mx_n+1000];
int prim[mx_n+1000], cnt;

void liner_sieve(cint x) {
    int rt = 0;
    for(int i=2; i<=x; i++) {
        if(!vis[i]) {
            prim[++cnt] = i;
        }
        for(int j=1; j<=cnt; j++) {
            if(1ll*prim[j]*i > x) break;
            vis[prim[j]*i] = 1;
            if(!(i%prim[j])) break;
        }
    }
}
```

3.2.2.2 积性函数

对于积性函数 f ，有 $f(1)=1$ 且当 $\gcd(a,b)=1$ 时有 $f(ab)=f(a)f(b)$
用欧拉筛筛积性函数大概有以下几个步骤

1. 对于质数 p ，求出 $f(p)$
2. 对于 $\gcd(p,q)=1$ 的情况，求出 $f(pq)=f(p)f(q)$
3. 对于 $\gcd(p,q)\neq 1$ 的情况，求出 $f(pq)$ 的值（对于完全积性函数，这一步可以归到 2 中）

   或者说对于质数 p ，求出 $f(p^k)$ 的值

```
// 求欧拉函数
void liner_sieve(cint x) {
    int rt = 0;
    for(int i=2; i<=x; i++) {
```

```
5         if(!vis[i]) {
6             prim[++cnt] = i;
7             phi[i] = i-1; // 1
8         }
9         for(int j=1; j<=cnt; j++) {
10            if(1ll*prim[j]*i > x) break;
11            vis[prim[j]*i] = 1;
12            if(!(i%prim[j])) {
13                phi[i*prim[j]] = phi[i] * prim[j]; // 3
14                break;
15            }
16            else phi[i*prim[j]] = phi[i] * phi[prim[j]]; // 2
17        }
18    }
19 }
```

3.3 MR 素性检验

3.4 狄利克雷卷积

第四章　多项式

4.1　快速傅里叶变换

在算法竞赛中，FFT 基本被用来进行多项式乘法的加速，所以无论什么如果发现暴力计算速度不够，就可以考虑写成形式幂级数然后套 FFT（这不就和生成函数很有关系了）

因为多项式相乘，每一项前都是两个数列的卷积，所以可以拆项凑卷积

任意模数多项式乘法使用 MTT ，不使用 CRT + NTT

```
1  namespace FFT {
2      typedef long long ll;
3      typedef long double db;
4
5      struct Complex {
6          db x, y;
7          Complex(db _x = 0.0, db _y = 0.0) { x = _x; y = _y; }
8          Complex conj() { return Complex(x, -y); }
9          Complex operator!() { return Complex(x, -y); }
10         Complex operator-(const Complex &b) const { return Complex(
               x - b.x, y - b.y); }
11         Complex operator+(const Complex &b) const { return Complex(
               x + b.x, y + b.y); }
12         Complex operator*(const db&b) const { return Complex(x*b, y
               *b); }
13         Complex operator*(const Complex &b) const { return Complex(
               x * b.x - y * b.y, x * b.y + y * b.x); }
14     };
15
16     /* PI 的值 */
17     const db PI = acos(-1.0);
18     /* 最大长度 ，保证补充到了 2 的次幂 */
19     const int max_le = (1<<19)+1;
20
21     /*
22      * 进行 FFT 和 IFFT 前的反置变换
23      * 位置 i 和 i 的二进制反转后的位置互换
24      * len 必须为 2 的幂
25      */
```

```
26      void change(Complex y[], int len) {
27          int i, j, k;
28          for (i = 1, j = len / 2; i < len - 1; i++) {
29              if(i < j) { swap(y[i], y[j]); }
30              k = len / 2;
31              while (j >= k) { j = j - k; k = k / 2; }
32              if(j < k) { j += k; }
33          }
34      }
35
36      /*
37       * 做 FFT
38       * len 必须是 2^k 形式
39       * on == 1 时是 DFT, on == -1 时是 IDFT
40       */
41      void fft(Complex y[], int len, int on) {
42          change(y, len);
43          for(int h = 2; h <= len; h <<= 1) {
44              /* 使用预处理的 sin 和 cos */
45              // Complex wn = prewn[h];
46              // wn.y *= on;
47
48              /* 不使用预处理的 sin 和 cos */
49              Complex wn(cos(2 * PI / h), sin(2 * PI / h) * on);
50
51              for(int j = 0; j < len; j += h) {
52                  Complex w(1, 0);
53                  for (int k = j; k < j + h / 2; k++) {
54                      Complex u = y[k];
55                      Complex t = w * y[k + h / 2];
56                      y[k] = u + t;
57                      y[k + h / 2] = u - t;
58                      w = w * wn;
59                  }
60              }
61          }
62          /* 这里将实部和虚部都从点值转换成了系数 */
63          if(on == -1) { for(int i = 0; i < len; i++) { y[i].x /= len
                ; y[i].y /= len; } }
64      }
65
66      /* FFT过程辅助数组 */
67      Complex c1[max_le], c2[max_le];
68
```

```
69      /*
70       * 做 FFT
71       * x1,x2为输入数组 , 其长度需要都不小于最近的 2 的次幂
72       * len1 , len2 为输入数组长度 , 注意下标从 0 开始 , 所以按 [0,
             len) 计算
73       * ans 为答案数组 , flen 为最后长度
74       * ans 只会在过程最后被修改
75       */
76      void solve_fft(int x1[], int x2[], int len1, int len2, int ans
             [], int&flen) {
77          int len = 1;
78          while(len < (len1+len2)) { len *= 2; }
79          flen = len;
80          for(int i=0; i<len; i++) {
81              if(i < len1) { c1[i] = Complex(x1[i], 0); }
82              else { c1[i] = Complex(); }
83              if(i < len2) { c1[i] = Complex(x2[i], 0); }
84              else { c2[i] = Complex(); }
85          }
86          fft(c1, len, 1);
87          fft(c2, len, 1);
88          for(int i=0; i<len; i++) { c1[i] = c1[i] * c2[i]; }
89          fft(c1, len, -1);
90          for(int i=0; i<len; i++) { ans[i] = int(c1[i].x + 0.5); }
91      }
92
93      /* FFT 加上三次变两次优化 */
94      void solve_fft2(int x1[], int x2[], int len1, int len2, int ans
             [], int&flen) {
95          int len = 1;
96          while(len < (len1+len2)) { len *= 2; }
97          flen = len;
98          for(int i=0; i<len; i++) {
99              if(i < len1 && i < len2) { c1[i] = Complex(x1[i], x2[i
                     ]); }
100             else if(i < len1) { c1[i] = Complex(x1[i], 0); }
101             else if(i < len2) { c1[i] = Complex(0, x2[i]); }
102             else c1[i] = Complex(0, 0);
103         }
104         fft(c1, len, 1);
105         for(int i=0; i<len; i++) { c1[i] = c1[i] * c1[i]; }
106         fft(c1, len, -1);
107         for(int i=0; i<len; i++) { ans[i] = int(c1[i].y/2 + 0.5); }
108     }
```

```
109
110     /* MMT过程辅助数组 */
111     Complex a[max_le], b[max_le], c[max_le], d[max_le];
112
113     /*
114      * 做 MTT
115      * x1, x2 为两个多项式系数
116      * len1 , len2 为输入数组长度 , 注意下标从 0 开始 , 所以按 [0,
             len) 计算
117      * ans 为输出的答案数组
118      * flen 为调整后的长度
119      * p 模数
120      */
121     void solve_mtt(int x1[], int x2[], int len1, int len2, int ans
            [], int &flen, cint p) {
122         int len = 1;
123         const int mv = 15;
124         while (len < (len1+len2)) { len *= 2; }
125         flen = len;
126         for(int i=0; i<len; i++) {
127             if(i < len1) { a[i] = Complex((x1[i] & ((1<<mv)-1)), (
                    x1[i] >> mv)); }
128             else { a[i] = Complex(); }
129             if(i < len2) { b[i] = Complex((x2[i] & ((1<<mv)-1))), (
                    x2[i] >> mv)); }
130             else { b[i] = Complex(); }
131         }
132         fft(a, len, 1); fft(b, len, 1);
133         for(int i=0; i<len; i++) {
134             int j = (i==0?0:len-i);
135             c[j] = Complex(a[i].x+a[j].x,a[i].y-a[j].y)*0.5*b[i];
136             d[j] = Complex(a[i].y+a[j].y,a[j].x-a[i].x)*0.5*b[i];
137         }
138         fft(c, len, 1); fft(d, len, 1);
139         for(int i=0; i<len; i++) {
140             ll x=ll(round(c[i].x/len))%p;
141             ll y=ll(round(c[i].y/len))%p;
142             ll u=ll(round(d[i].x/len))%p;
143             ll v=ll(round(d[i].y/len))%p;
144             ans[i] = (x+((y+u)<<mv)%p+(v<<(mv<<1))%p)%p;
145         }
146     }
147 };
```

4.2 快速数论变换

对于形式为 $r \times 2^k + 1$ 的质数模数，可以利用原根取代单位根，从而避免精度损失

```
namespace NTT {
    typedef long long ll;

    /* 常见原根表 */
    // 998244353 -> 3
    const int P = 998244353, G = 3;

    /* 最大长度 , 保证补充到了 2 的次幂 */
    const int max_le = (1<<21)+1;

    int w[max_le], inv[max_le];

    int mod(int x) { return x >= P ? x - P : x; }

    ll ksm(ll bs, int x) {
        ll ans = 1;
        while(x) {
            if(x & 1) { ans = ans * bs % P; }
            bs = bs * bs % P;
            x >>= 1;
        }
        return ans;
    }

    void init(int len) {
        inv[1] = 1;
        for(int i=2;i<=len;i++) { inv[i] = mod(P - 1ll * (P / i) *
           inv[P%i] % P); }
        for(int i=1;i<len;i<<=1) {
            int wn = ksm(G, (P - 1) / (i<<1));
            for(int j=0, ww=1; j<i; j++, ww=1ll*ww*wn%P) { w[i+j] =
                ww; }
        }
    }

    void change(ll y[], int len) {
        int i, j, k;
        for (i = 1, j = len / 2; i < len - 1; i++) {
            if(i < j) { swap(y[i], y[j]); }
            k = len / 2;
            while (j >= k) { j = j - k; k = k / 2; }
```

```
            if(j < k) { j += k; }
        }
    }

    void ntt(ll f[], int len, int op) {
        change(f, len);
        for(int i=1;i<len;i<<=1){
            for(int j=0;j<len;j+=i<<1) {
                for(int k=0;k<i;k++) {
                    int x = f[j+k], y = 1ll * f[i+j+k]* w[i+k]%P;
                    f[j+k] = mod(x+y);
                    f[i+j+k] = mod(x-y+P);
                }
            }
        }
        if(op == -1) {
            reverse(&f[1],&f[len]);
            for(int i=0;i<len;i++) { f[i] = 1ll * f[i] * inv[len] %
                P; };
        }
    }
    /* NTT 过程辅助数组 */
    ll a[max_le], b[max_le];

    void solve(ll x1[], ll x2[], int len1, int len2, ll ans[]) {
        int len = 1;
        while(len < (len1 + len2)) { len <<= 1; }
        init(len);
        for(int i=0; i<len; i++) {
            if(i < len1) { a[i] = x1[i]; }
            else { a[i] = 0; }
            if(i < len2) { b[i] = x2[i]; }
            else { b[i] = 0; }
        }
        ntt(a, len, 1); ntt(b, len, 1);
        for(int i=0; i<len; i++) { a[i] = a[i] * b[i] % P; }
        ntt(a, len, -1);
        for(int i=0; i<len; i++) { ans[i] = a[i]; }
    }
};
```

原根表

```
常用素数:
P = 1004535809  ====>  pr = 3
P = 998244353  =====>  pr = 3
```

prime	r	k	g
3	1	1	2
5	1	2	2
17	1	4	3
97	3	5	5
193	3	6	5
257	1	8	3
7681	15	9	17
12289	3	12	11
40961	5	13	3
65537	1	16	3
786433	3	18	10
5767169	11	19	3
7340033	7	20	3
23068673	11	21	3
104857601	25	22	3
167772161	5	25	3
469762049	7	26	3
1004535809	479	21	3
2013265921	15	27	31
2281701377	17	27	3
3221225473	3	30	5
75161927681	35	31	3
77309411329	9	33	7
206158430209	3	36	22
2061584302081	15	37	7
2748779069441	5	39	3
6597069766657	3	41	5
39582418599937	9	42	5
79164837199873	9	43	5
263882790666241	15	44	7
1231453023109121	35	45	3
1337006139375617	19	46	3
3799912185593857	27	47	5
4222124650659841	15	48	19
7881299347898369	7	50	6
31525197391593473	7	52	3
180143985094819841	5	55	6
1945555039024054273	27	56	5
4179340454199820289	29	57	3

4.3 分治 FFT

如果某一个项的值和前面的项有关，可以考虑使用分治 FFT 优化

基本思路为分治，对于 $[l,r]$ ，先求出 $[l,mid]$ 的值，再计算 $[l,mid]$ 对 $[mid+1,r]$ 的贡献，再去算 $[mid+1,r]$ 内的贡献

```
void cdq(int l, int r) {
    if(l == r) { return; }
    int mid = (l+r) >> 1;
    cdq(l, mid);
    solve_contri();
    for(int i=mid+1; i<=r; i++) { add_contri();  }
    cdq(mid+1, r);
}
```

注意，分治到 $[a,b]$ 的时候，应该只用 $[a,b]$ 这一段做多项式乘法，而不是 $[1,b]$

举个例子

```
void cdq(int l, int r) {
    if(l == r) { return; }
    int mid = (l+r) >> 1;
    cdq(l, mid);
    NTT::solve(ans+l, a, mid+1-l, r-l+1, pre);
    for(int i=mid+1; i<=r; i++) { (ans[i] += pre[i-l]) %= mod;  }
    cdq(mid+1, r);
}
```

4.4 多项式乘法逆

采用倍增的思想

首先有 $[x^0]f^{-1}(x)$ 为 $[x^0]f(x)$ 在模数下的逆元

不妨假定多项式长度为 2 的幂

如果现在，我们算出了 $f_0^{-1}(x)$ ，满足 $f(x)\cdot f_0^{-1}\equiv 1\ (mod\ x^{\frac{n}{2}})$ ，那么如果答案为 $f^{-1}(x)$ ，则有

$$f^{-1}(x)-f_0^{-1}(x)\equiv 0\ (mod\ x^{\frac{n}{2}})$$

两边平方并同乘 $f(x)$ ，整理则有

$$f^{-1}(x)\equiv f_0^{-1}(x)(2-f(x)f_0^{-1}(x))\ (mod\ x^n)$$

直接计算即可

需要保证多项式常数项在模数意义下有逆元存在

```
void solve() {
    // a 为系数数组 , mod 为模数 , pre 为辅助数组
    inv[0] = get_inv(a[0], mod);
    for(int i=2; i<n*2; i<<=1) {
        FFT(a, inv, i, i/2, pre, mod);
        pre[0] -= 2;
        for(int j=0; j<i; j++) { pre[j] = -pre[j]; }
        FFT(inv, pre, i/2, i, inv, mod);
```

```
9       }
10      for(int i=0; i<n; i++) { if(inv[i] < 0) { inv[i] += mod; } }
11      for(int i=0; i<n; i++) { cout << inv[i] << '␣'; }
12      cout << '\n';
13  }
```

4.5 多项式求导与积分

4.6 多项式 ln 与 exp

第五章　组合计数与生成函数

第六章　数据结构

6.1　树状数组

```
int bnode[mx_n];

int lowbit(int &x) { return x&-x; }

void add(int x, cint co) {
    while(x <= mx_n) {
        bnode[x] += co;
        x += lowbit(x);
    }
}

int query(int x) {
    int ans = 0;
    while(x) {
        ans += bnode[x];
        x -= lowbit(x);
    }
    return ans;
}
```

注意，树状数组无法直接处理 **0** ，需要处理一下

第七章　典型题型

7.1　最长上升子序列

7.1.1　做法

第一种：

dp，复杂度 $\Theta(N^2)$ ，优点是可以记录子序列

```
// Nope
```

第二种：

贪心，复杂度 $\Theta(N \log N)$ ，优点是复杂度低

```
int mx[mx_n];
int r = 0;
memset(mx, 0x3f, sizeof mx);
mx[0] = 0;
for(int i=1; i<=m; i++) {
    int id = lower_bound(mx, mx+r+1, c[i]) - mx;
    mx[id] = c[i];
    r = max(r, id);
}
// 最后 mx 数组合法的最大的下标就是答案
```

7.2　多重限制的问题

7.2.1　例题

> 阿强得到一份地图。这个地图包含 n 个点 m 条边的无重边无自环的无向图。每个节点都有一个正权值，每条路径也都含有一个正权值。他希望评估任意两点间的难度是多少。两点间的难度定义为：所有这两点间的路径中能得到的最小的路径最大点权乘以路径最大边权。

考虑固定一个条件，比如说经过的点权不大于某个值，这样最短路径就是一些点的导出子图的最小生成树上的路径

本质上，如果两点之间的最短路的最大点权是 c 的话，那么所有点权小于等于 c 的点的生成子图的最小生成树一定包含这条最短路，所以枚举了点权。

第八章　杂项

8.1　位运算

1. $a+b=a\&b+a|b$
2. 奇数个数的**与和**一定小于等于其**异或和**，偶数个数的与和为 1 的位置，异或和在该位置一定为 0

8.2　莫队算法

8.2.1　普通莫队

8.2.1.1　普通莫队

一些定义，其中 $block\_siz=\frac{n}{\sqrt{m}}$

```
int block_siz; // 块大小
struct query {
    int l, r; // 询问区间
    int bl, br; // 区间端点所属的块的编号
    int id;
    int ans;
    void init(cint x) {
        id = x;
        cin >> l >> r;
        bl = l / block_siz;
        br = r / block_siz;
    }
} b[200200];
```

主要部分

```
for(int i=1; i<=q; i++) {
    while(r < b[i].r) ++r, add(r);
    while(l > b[i].l) --l, add(l);
    while(r > b[i].r) --r, dele(r+1);
    while(l < b[i].l) ++l, dele(l-1);
    b[i].ans = siz;
}
```

奇偶化排序优化

```
1 bool cmp1(const query&a, const query&b) {
2     return a.bl == b.bl ? ( a.bl&1 ? a.r < b.r : a.r > b.r) : a.bl
          < b.bl;
3 }
```

8.2.2 回滚莫队

8.3 cdq 分治