

# ACM Template

DUT ACM Lab

2021 年 10 月 30 日

# 目录

<b>第一章 STL 使用</b>	<b>1</b>
1.1 set-multiset-map	1
1.1.1 set	1
1.1.2 multiset	1
1.1.3 map	1
1.2 unordered STL	1
<b>第二章 图论</b>	<b>2</b>
2.1 二分图	2
2.1.1 二分图最大匹配	2
2.2 树上问题	3
2.2.1 树的重心	3
2.2.2 点分治	4
2.2.3 边分治	6
2.2.4 点分树	6
2.2.5 树上启发式合并	6
2.3 网络流	8
2.3.1 最大流算法	8
2.3.2 费用流算法	10
2.3.3 上下界网络流	10
2.3.4 其余变形	10
2.4 生成树	10
2.4.1 最小生成树	10
2.4.2 生成树计数	10
<b>第三章 数论</b>	<b>11</b>
3.1 逆元	11
3.1.1 逆元的一些实现	11
3.1.2 逆元存在性	12
3.2 筛法	12
3.2.1 埃氏筛（素数）	12
3.2.2 欧拉筛（素数，积性函数）	12
3.3 MR 素性检验	14
3.4 狄利克雷卷积	14

<b>第四章 多项式</b>	<b>15</b>
4.1 快速傅里叶变换 . . . . .	15
<b>第五章 数据结构</b>	<b>16</b>
5.1 树状数组 . . . . .	16
<b>第六章 组合统计</b>	<b>17</b>
6.1 常见的恒等式与组合结论 . . . . .	17
6.1.1 结论 . . . . .	17
6.1.2 恒等式 . . . . .	17
<b>第七章 典型题型</b>	<b>18</b>
7.1 最长上升子序列 . . . . .	18
7.1.1 做法 . . . . .	18
7.2 多重限制的问题 . . . . .	18
7.2.1 例题 . . . . .	18
<b>第八章 杂项</b>	<b>19</b>
8.1 位运算 . . . . .	19
8.2 莫队算法 . . . . .	19
8.2.1 普通莫队 . . . . .	19
8.2.2 回滚莫队 . . . . .	20
8.3 cdq 分治 . . . . .	20

# 第一章 STL 使用

## 1.1 set-multiset-map

### 1.1.1 set

```
1 set<T> e;  
2 e.clear();  
3 e.insert();  
4 e.size();  
5 e.lower_bound(); // iter  
6 e.upper_bound(); // iter  
7 e.find();
```

### 1.1.2 multiset

```
1 multiset<T> e;  
2 // the same to up  
3 e.equal_range(); // pair  
4 e.count();s
```

### 1.1.3 map

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

## 1.2 unordered 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  $\Theta(nm)$

```
1  const int mx_n = 1005;
2  bool mp[mx_n][mx_n];
3  bool vis[mx_n];
4  int pre[mx_n];
5
6  bool dfs(cint loc) {
7      for(int i=n+1; i<=n+m; i++) {
8          if(mp[loc][i] && !vis[i]) {
9              vis[i] = 1;
10             if(!pre[i] || dfs(pre[i])) {
11                 pre[i] = loc;
12                 return 1;
13             }
14         }
15     }
16     return 0;
17 }
18
19 int main() {
20     int ans = 0;
21     for(int i=1; i<=n; i++) {
22         memset(vis, 0, sizeof vis);
23         ans += dfs(i);
24     }
25     cout << ans << endl;
26     return 0;
27 }
```

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。

```

1 void dfs(cint loc, cint fa) {
2     int pre = 0;
3     son[loc] = 1;
4     for(int v: to[loc]) {
5         if(v != fa) {
6             dfs(v, loc);
7             pre = max(pre, son[v]);
8             son[loc] += son[v];
9         }
10    }
11    pre = max(pre, n-son[loc]);
12    if(pre < st) {
13        st = pre; # 最大儿子的最小值
14        ans = loc; # 最大儿子的编号
15    }
16 }

```

#### 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

```
1 int h[10010], nx[20020], to[20020], w[20020], cnt_; // 链式前向星数组
2 int son[10010]; // 经过处理后的每个点的儿子个数
3 bool vis[10010]; // 该点是否在分治时作为子树的根
4 int id; // 当前所处理的树的重心
5 int snode; // 当前所处理树的节点数量
6
7 void add(cint f, cint t, cint co) {
8     nx[++cnt_] = h[f];
9     h[f] = cnt_;
10    to[cnt_] = t;
11    w[cnt_] = co;
12 }
```

统计以某点为根且不跨越其余重心的子树大小

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

```

4         if(to[i] != fa && !vis[to[i]]) {
5             sum += gsiz(to[i], loc);
6         }
7     return sum;
8 }

```

寻找树的重心

```

1 void gp(cint loc, cint fa) {
2     int pre = 0;
3     son[loc] = 1;
4     for(int i=h[loc]; i; i=nx[i])
5         if(to[i] != fa && !vis[to[i]]) {
6             gp(to[i], loc);
7             pre = max(pre, son[to[i]]);
8             son[loc] += son[to[i]];
9             if(id) return;
10        }
11    pre = max(pre, snode - son[loc]);
12    // 树的重心可能有两个，此处任取了一个
13    if(pre <= snode/2) {
14        id = loc;
15        return;
16    }
17 }

```

统计跨越重心的答案

```

1 void check(cint loc, cint fa) {
2     // 统计跨越重心的答案
3     for(int i=h[loc]; i; i=nx[i])
4         if(to[i] != fa && !vis[to[i]]) {
5             check(to[i], loc);
6         }
7 }

```

合并子树

```

1 void update(cint loc, cint fa) {
2     // 合并子树
3     for(int i=h[loc]; i; i=nx[i]) {
4         if(to[i] != fa && !vis[to[i]]) {
5             update(to[i], loc);
6         }
7     }
8 }

```

解决问题



```

1 void sol(cint loc) {
2     vis[loc] = 1;
3     // 初始化计算答案与合并子树时需要用到的东西
4     for(int i=h[loc]; i; i=nx[i]) {
5         if(!vis[to[i]]) {
6             check(to[i], loc);
7             update(to[i], loc);
8         }
9     }
10    for(int i=h[loc]; i; i=nx[i]) {
11        if(!vis[to[i]]) {
12            snode = gsiz(to[i], loc);
13            id = 0;
14            gp(to[i], loc);
15            sol(id);
16        }
17    }
18 }

```

主函数里的一点东西

```

1 int main() {
2     snode = n;
3     gp(1, 1);
4     sol(id);
5 }

```

### 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

```

1 vector<int> to[100100]; // 邻接表
2 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);
```

```

9     }
10  }

```

合并子树

```

1  void ins(cint r, cint x) {
2      // do somethings
3  }
4
5  void update(cint loc, cint fa) {
6      // ins
7      for(int v: to[loc]) {
8          if(v != fa) update(v, loc);
9      }
10 }

```

主函数的一些部分

```

1  int main() {
2      fd_son(1, 1);
3      sol(1, 1);
4  }

```

## 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 $\Theta(m\sqrt{n})$

```

1  const int mxn = 202; // 最大点数
2  const int mxm = 5005; // 最大边数
3  struct dinic {
4      int s, t; // 源, 汇
5      ll w[mxm*2];
6      int h[mxn], nx[mxm*2], to[mxm*2], cnt=1; // 链式前向星, cnt=1
           方便找到反边
7      int cur[mxn]; // 当前弧优化辅助数组
8      int label[mxn]; // 分层图标号
9      void clear_map() { cnt = 1; for(int i=1; i<=mxn; i++) h[i] = 0;
           }
10     void add(cint f, cint t, cint co) {
11         nx[++cnt] = h[f]; h[f] = cnt; to[cnt] = t; w[cnt] = co;
12         nx[++cnt] = h[t]; h[t] = cnt; to[cnt] = f; w[cnt] = 0;
13     }
14     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

```
1 ll ksm(ll m, int c) {
2     ll ans = 1;
3     while(c) {
4         if(c&1) ans = (ans*m) % mod;
5         c >>= 1;
6         m = (m*m) % mod;
7     }
8     return ans;
9 }
10
11 ll inv(ll x) { return ksm(x, mod-2); }
```

##### 3.1.1.2 阶乘的线性逆元

```
1 ll ksm(ll m, int c) {
2     ll ans = 1;
3     while(c) {
4         if(c&1) ans = (ans*m) % mod;
5         c >>= 1;
6         m = (m*m) % mod;
7     }
8     return ans;
9 }
10 void sol_inv() {
11     fac[0] = 1;
12     for(int i=1; i<=mx_n; i++) fac[i] = fac[i-1] * i % mod;
13     inv[mx_n] = ksm(fac[mx_n], mod-2);
14     for(int i=mx_n-1; i; i--) inv[i] = inv[i+1] * (i+1) % mod;
```

15 }

**3.1.1.3 1 到 n 的线性逆元**

```

1 inv[1] = 1;
2 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  $b^{\varphi(p)-1} \pmod{p}$

**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

```

1 const int mx_n = 100000000;
2 int vis[mx_n+1000];
3 int prim[mx_n+1000], cnt;
4
5 void liner_sieve(cint x) {
6     int rt = 0;
7     for(int i=2; i<=x; i++) {
8         if(!vis[i]) {
9             prim[++cnt] = i;
10            vis[i] = i;

```

```

11     }
12     for(int j=1; j<=cnt; j++) {
13         if(1ll*prim[j]*i > x) break;
14         if(prim[j] > vis[i]) break;
15         vis[prim[j]*i] = prim[j];
16     }
17 }
18 }

```

#### 省空间的写法

```

1  const int mx_n = 100000000;
2  int n, q;
3  bool vis[mx_n+1000];
4  int prim[mx_n+1000], cnt;
5
6  void liner_sieve(cint x) {
7      int rt = 0;
8      for(int i=2; i<=x; i++) {
9          if(!vis[i]) {
10             prim[++cnt] = i;
11         }
12         for(int j=1; j<=cnt; j++) {
13             if(1ll*prim[j]*i > x) break;
14             vis[prim[j]*i] = 1;
15             if(!(i%prim[j])) break;
16         }
17     }
18 }

```

#### 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

```

1  // 求欧拉函数
2  void liner_sieve(cint x) {
3      int rt = 0;
4      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

## 第五章 数据结构

### 5.1 树状数组

```
1  int bnode[mx_n];
2
3  int lowbit(int &x) { return x&-x; }
4
5  void add(int x, cint co) {
6      while(x <= mx_n) {
7          bnode[x] += co;
8          x += lowbit(x);
9      }
10 }
11
12 int query(int x) {
13     int ans = 0;
14     while(x) {
15         ans += bnode[x];
16         x -= lowbit(x);
17     }
18     return ans;
19 }
```

注意，树状数组无法直接处理 0，需要处理一下

## 第六章 组合统计

### 6.1 常见的恒等式与组合结论

#### 6.1.1 结论

1.  $K_n$  的生成数的个数为  $n^{n-2}$
2.  $n$  个点的凸多边形对角线的交点最多为  $C_n^4$ ，其中不超过两条对角线在内部的任何一点相交

#### 6.1.2 恒等式

1.  $\sum_{i=1}^n C_i^x = C_{n+1}^{x+1}$

## 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

```
1 // Nope
```

第二种：

贪心, 复杂度  $\Theta(N \log N)$  , 优点是复杂度低

```
1 int mx[mx_n];
2 int r = 0;
3 memset(mx, 0x3f, sizeof mx);
4 mx[0] = 0;
5 for(int i=1; i<=m; i++) {
6     int id = lower_bound(mx, mx+r+1, c[i]) - mx;
7     mx[id] = c[i];
8     r = max(r, id);
9 }
10 // 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  $block\_siz = \frac{n}{\sqrt{m}}$

```
1 int block_siz; // 块大小
2 struct query {
3     int l, r; // 询问区间
4     int bl, br; // 区间端点所属的块的编号
5     int id;
6     int ans;
7     void init(cint x) {
8         id = x;
9         cin >> l >> r;
10        bl = l / block_siz;
11        br = r / block_siz;
12    }
13 } b[200200];
```

主要部分

```
1 for(int i=1; i<=q; i++) {
2     while(r < b[i].r) ++r, add(r);
3     while(l > b[i].l) --l, add(l);
4     while(r > b[i].r) --r, dele(r+1);
5     while(l < b[i].l) ++l, dele(l-1);
6     b[i].ans = siz;
7 }
```

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