其他

分数规划

给出 a_i 和 b_i，求一组 $w_i \in \{0,1\}$，最小化或最大化

$$\frac{\sum\limits_{i=1}^{n} a_i \times w_i}{\sum\limits_{i=1}^{n} b_i \times w_i}$$

二分 mid:

$$\begin{aligned}&\frac{\sum a_i \times w_i}{\sum b_i \times w_i} > mid \\ \Longrightarrow &\sum a_i \times w_i - mid \times \sum b_i \cdot w_i > 0 \\ \Longrightarrow &\sum w_i \times (a_i - mid \times b_i) > 0\end{aligned}$$

```
int a[maxn], b[maxn];
double c[maxn];
bool check(double m)
{
    for(int i=1;i<=n;i++) c[i] = a[i]-m*b[i];
    sort(c+1, c+1+n, greater<double>());
    return accumulate(c+1, c+1+k, 0.0)>0;
}
double l=0, r=1e5;
while(r-l>eps)
{
    double mid = (l+r)/2;
    if(check(mid)) l = mid;
    else r = mid;
}
```

例：分母至少为 W

```
double dp[maxw];
bool check(double m)
{
    fill(dp+1, dp+1+W, -1e9);
    for(int i=1;i<=n;i++)
    {
        for(int j=W;j>=0;j--)
        {
            int k = min(W, j+b[i]);
            dp[k] = max(dp[k], dp[j]+a[i]-m*b[i]);
        }
    }
```

```
    return dp[W]>0;
}
```

extc++(pd_ds & rope)

哈希表

用法同 `std::unordered_map`

```
__gnu_pbds::cc_hash_table<K, V> h; // 拉链法
__gnu_pbds::gp_hash_table<K, V> h; // 探测法
```

平衡树

```
__gnu_pbds::tree<pii, __gnu_pbds::null_type, less<pii>,
    __gnu_pbds::rb_tree_tag, __gnu_pbds::tree_order_statistics_node_update> rbt;
// $1: key type
// $2: val type(allow null_type)
// $3: comp
// $4: which tree (rbt, splay, ov)
// $5: node updater
```

```
rbt.insert(make_pair(x,i)); // insert, use pair to unique(let i>0 and unique)
rbt.erase(rbt.lower_bound(make_pair(x,0))); // remove
rbt.order_of_key(make_pair(x,0))+1; // query order(1-index) by number
rbt.find_by_order(x-1)->first; // query number by order(1-index)
rbt.find_by_order(rbt.order_of_key(make_pair(x,0))-1)->first; // query prev
rbt.find_by_order(rbt.order_of_key(make_pair(x+1,0)))->first; // query next

rbt.join(t); // merge t into rbt
rbt.split(x,t); // split elements greater than x to t;
```

Trie

```
__gnu_pbds::trie<string, null_type, __gnu_pbds::trie_string_access_traits<>,
    __gnu_pbds::pat_trie_tag, __gnu_pbds::trie_prefix_search_node_update> t;
// $1: key type
// $2: val type(allow null_type)
// $3: access_trait
// $4: recommand pat
// $5: node updater
```

堆

```
__gnu_pbds::priority_queue<int, less<>, __gnu_pbds::pairing_heap_tag> pq;
// $1: val type
// $2: less: 大根堆
// $3: 见下图
```

rope

比较暴力的长 `std::string`

- `operator+()` 与 `operator+=()`，拼接
- `operator-()` 与 `operator-=()`，剪切
- `operator<()` 与 `operator==()`，比较

`rope` 暴力可持久化数组

	push	pop	modify	erase	join
std::priority_queue	$\Theta(n)/\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(n \ \lg n)$	$\Theta(n \ \lg n)$	$\Theta(n \ \lg n)$
__gnu_pbds::pairing_heap_tag	$O(1)$	$\Theta(n)/\Theta(\lg n)$	$\Theta(n)/\Theta(\lg n)$	$\Theta(n)/\Theta(\lg n)$	$O(1)$
__gnu_pbds::binary_heap_tag	$\Theta(n)/\Theta(\lg n)$	$\Theta(n)/\Theta(\lg n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
__gnu_pbds::binomial_heap_tag	$\Theta(\lg n)/O(1)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$
__gnu_pbds::rc_binomial_heap_tag	$O(1)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$
__gnu_pbds::thin_heap_tag	$O(1)$	$\Theta(n)/\Theta(\lg n)$	$\Theta(\lg n)/O(1)$	$\Theta(n)/\Theta(\lg n)$	$O(n)$

Figure 1: pb_ds_heap

```cpp
int n,m;
cin>>n>>m;
vector<__gnu_cxx::rope<int>> w(1);
w.front().push_back(0);
for(int i=1;i<=n;i++)
{
    int x;
    cin>>x;
    w.front().push_back(x);
}
for(int v=1;v<=m;v++)
{
    int r,o,p;
    cin>>r>>o>>p;
    w.emplace_back(w[r]);
    if(o==1)
    {
        int x;
        cin>>x;
        w[v].mutable_reference_at(p) = x;
    }
    else cout<<w[v][p]<<'\n';
}
```

rope 暴力文艺平衡树

```cpp
__gnu_cxx::rope<int> a,b;
int n,m;
cin>>n>>m;
for(int i=1;i<=n;i++) a.push_back(i), b.push_back(n-i+1);
while(m--)
{
    int l,r;
    cin>>l>>r;
    l--;
    auto p = a.substr(a.begin()+l, a.begin()+r);
    a = a.substr(a.begin(), a.begin()+l)+b.substr(b.begin()+(n-r), b.begin()+(n-l))+ \
        a.substr(a.begin()+r, a.end());
    b = b.substr(b.begin(), b.begin()+(n-r))+p+b.substr(b.begin()+(n-l), b.end());
}
for(auto i : a) cout<<i<<' ';
cout<<endl;
```

动态规划

LIS

$a\{n\}$

- 设计状态：$dp[i]$ 代表数列 $a\{n\}$ 的前 i 个数的最长不下降子序列长度。
- 初始状态：$dp[1]=1$
- 转移方程：$dp[i]=\max(dp[i],dp[j]+1)\ (j<i\ \ \&\ \ a[i]\geq a[j])$
- 结果：$dp[n]$

贪心 ＋ 二分

```
int lis(const vector<int>& v)    // 最长不下降子序列
{
    vector<int> d = {v.front()};
    for(size_t i=1;i<v.size();i++)
    {
        if(v[i]>=d.back()) d.push_back(v[i]);
        else *upper_bound(d.begin(), d.end(), v[i]) = v[i];
    }
    return d.size();
}
```

LCS

普通

- 设计状态：$dp[i][j]$ 代表数列 $a\{n\}$ 的前 i 个数以及数列 $b\{n\}$ 的前 j 个数的 LCS 长度。
- 初始状态：$dp[0][0]=0$
- 转移方程：

$$dp[i][j]=\begin{cases}\max(dp[i][j],dp[i-1][j-1]+1) & a_i=b_j\\ \max(dp[i-1][j],dp[i][j-1]) & a_i\neq b_j\end{cases}$$

- 结果：$dp[n][m]$

全排列

$a\to A[1]\ b\to A[2]\ c\to A[3]\ d\to A[4]\ e\to A[5]$

{a}: a b c d e {b}: c b a d e

LCS 长度没有发生变化。P 是 a 与 b 的 LCS，P 一定既是 a 的子序列也是 b 的子序列。P 一定递增。最长的 $P=>b$ 的最长上升子序列。

博弈论

巴什博奕

有 n 个石子，两个人轮流从中取走石子，每次最多取 m 个，最少取 1 个，取完者获胜，问先手方有没有必胜的策略。

```
n%(m+1)?"YES":"NO"
```

威佐夫博弈

有两堆石子，石子数量分别为 a, b。两人轮流取石子，取法有两种：

- 取走一堆中任意个石子；
- 从两堆中取走相同数目的石子。取完所有石子的一方获胜，问先手方有没有必胜的策略。

```
if(a>b) swap(a,b);
auto t = (sqrt(5)+1)/2;
if(int(t*(b-a))==a) cout<<"NO"<<'\n';
else cout<<"YES"<<'\n';
```

Nim 游戏

有 n 堆石子 $a_1, a_2, \cdots, a_n$，两人轮流从任意一组取若干个石子，谁取完谁赢，问先手有没有必胜策略。

```
if(accumulate(v.begin(), v.end(), 0, [](int a,int b) { return a^b; }))
    cout<<"YES"<<'\n';
else cout<<"NO"<<'\n';
```

SG 定理

$$\operatorname{mex}(S) = \min\{x\} \quad (x \notin S, x \in N)$$

$$\operatorname{SG}(x) = \operatorname{mex}\{\operatorname{SG}(y_1), \operatorname{SG}(y_2), \ldots, \operatorname{SG}(y_k)\}$$

对于一个由 n 个有向图组成的游戏，当且仅当下式成立时，这个游戏是先手必胜的：

$$\operatorname{SG}(s_1) \oplus \operatorname{SG}(s_2) \oplus \ldots \oplus \operatorname{SG}(s_n) \neq 0$$

图论

bfs/dfs 版 SPFA 判负环 (用来卡常)

```
ll dis[maxn];
int cnt[maxn];
bool spfa(double k)
{
    memset(dis, 0x3f, sizeof(dis));
    queue<int> q;
    for(int i=1;i<=n;i++) q.push(i);
    while(!q.empty())
    {
        int u = q.front();
        q.pop();
        for(auto&& [v,w] : G[u])
        {
            auto t = dis[u]+w;
            if(t<dis[v])
            {
                dis[v] = t;
                cnt[v] = cnt[u]+1;
                if(cnt[v]==n) return true;
                q.push(v);
            }
        }
    }
    return false;
}
bool spfa(int u)
{
    vis[u] = true;
    for(int i=Head[u];~i;i=Edge[i].next)
    {
        int v = Edge[i].to;
        if(dis[v]>dis[u]+Edge[i].w)
        {
            dis[v] = dis[u]+Edge[i].w;
            if(vis[v]||spfa(v)) return true;
        }
    }
    vis[u] = false;
    return false;
}
```

二分图最大匹配

- 一个图没有奇环当且仅当是一个二分图

匈牙利算法

- 枚举每一个左部结点 u;
  - 枚举 u 结点的邻接结点 v, 对于每个结点 v
    * 如果 v 没有被匹配过, 那么直接让 u, v 匹配;
    * 否则让原来匹配 v 的结点 u' 去尝试匹配其他右部结点;
      · 如果 u' 匹配到了其他结点 v', 那么 u, v 匹配, u', v' 匹配;
      · 否则 u 失配。

```
vector<int> G[maxn];
int vis[maxn], match[maxn];
bool dfs(int u,int tag)
{
    if(vis[u]==tag) return false;
    vis[u] = tag;
    for(auto v : G[u])
    {
        if(!match[v]||dfs(match[v],tag))
        {
            match[v] = u;
            return true;
        }
    }
    return false;
}
for(int i=1;i<=n;i++) if(dfs(i,i)) ans++;
```

强连通分量

```
vector<vector<int>> G(n+1);
int tim = 0;
vector<int> dfn(n+1), low(n+1);
vector<bool> vis(n+1);
stack<int> st;
int cnt = 0;
vector<int> scc(n+1);  // output: {scc: scc id}
function<void(int)> tarjan = [&](int u)
{
    low[u] = dfn[u] = ++tim;
    st.push(u);
    vis[u] = true;
    for(auto v : G[u])
    {
        if(!dfn[v])
        {
            tarjan(v);
            low[u] = min(low[u], low[v]);
        }
        else if(vis[v]) low[u] = min(low[u], dfn[v]);
    }
    if(dfn[u]==low[u])
    {
        cnt++;
        int v;
        do
        {
            v = st.top();
```

```
            st.pop();
            scc[v] = cnt;
            vis[v] = false;
        }
        while(v!=u);
    }
};
for(int i=1;i<=n;i++) if(!dfn[i]) tarjan(i);
//* 以下为缩点，注意直接这样会有重边
vector<vector<int>> H(cnt+1);
for(int u=1;u<=n;u++)
    for(auto v : G[u])
        if(scc[u]!=scc[v])
            H[scc[u]].push_back(scc[v]);
//! 缩点后结点排列顺序是逆拓扑序，不需要再次拓排
for(int u=cnt;u>=1;u--) // do something...(按拓扑序处理)
```

点双连通分量与割点

```
vector<vector<int>> G(n+1);
int tim = 0;
vector<int> dfn(n+1), low(n+1);
stack<int> st;
vector<bool> cut(n+1); // output: {cut: is_cut_vertex}
vector<vector<int>> bcc; // output: {bcc: nodes in bcc_i}
function<void(int,int)> tarjan = [&](int u,int f)
{
    low[u] = dfn[u] = ++tim;
    st.push(u);
    if(!f&&G[u].empty()) // 孤立点
    {
        bcc.emplace_back(1,u);
        return;
    }
    int s = 0;
    for(auto v : G[u])
    {
        if(!dfn[v])
        {
            tarjan(v,u);
            low[u] = min(low[u], low[v]);
            if(low[v]>=dfn[u])
            {
                s++;
                if(f||s>1) cut[u] = true;
                bcc.emplace_back();
                int w;
                do
                {
                    w = st.top();
                    st.pop();
                    bcc.back().push_back(w);
                }
                while(w!=v);
                bcc.back().push_back(u);
            }
```

```
        }
        else low[u] = min(low[u], dfn[v]);
    }
};
//! 注意：请保证图中不存在自环
for(int i=1;i<=n;i++) if(!dfn[i]) tarjan(i,0);
```

边双连通分量与桥

```
typedef pair<int,int> pii;
vector<vector<pii>> G(n+1); //! edge: <to, id>
int tim = 0;
vector<int> dfn(n+1), low(n+1);
vector<bool> cut(m+1); // output: {cut: is_cut_edge}
// 求桥
function<void(int,int)> tarjan = [&](int u,int f)
{
    low[u] = dfn[u] = ++tim;
    for(auto [v,id] : G[u])
    {
        if(!dfn[v])
        {
            tarjan(v,u);
            low[u] = min(low[u], low[v]);
            if(low[v]>dfn[u]) cut[id] = true;
        }
        else if(v!=f) low[u] = min(low[u], dfn[v]);
    }
};
// 求边双连通分量
int cnt = 0;
vector<int> bcc(n+1); // output: {bcc: bcc id}
function<void(int)> dfs = [&](int u)
{
    bcc[u] = cnt;
    for(auto [v,id] : G[u])
    {
        if(bcc[v]||cut[id]) continue;
        dfs(v);
    }
};
for(int i=1;i<=n;i++) if(!dfn[i]) tarjan(i,0);
for(int i=1;i<=n;i++) if(!bcc[i]) cnt++, dfs(i);
```

字符串

前缀函数与 KMP

```
vector<size_t> prefix_function(const string& s)
{
    vector<size_t> nxt(s.length());
    for(size_t i=1,j=0;i<s.length();i++)
    {
        while(j>0&&s[j]!=s[i]) j = nxt[j-1];
        if(s[j]==s[i]) nxt[i] = ++j;
    }
    return nxt;
}
// split 为任意不在字符集中的字符
vector<size_t> kmp(const string& match, const string& pattern, char split = '#')
{
    vector<size_t> res;
    auto s = pattern+split+match;
    auto nxt = prefix_function(s);
    for(size_t i=0;i<s.length();i++)
        if(nxt[i]==pattern.length())
            res.push_back(i-(pattern.length()<<1));
    return res;
}
```

数据结构

ST 表

一维

```
vector<int> a(n+1);
// init a;
vector<vector<int>> st(20, vector<int>(n+1));
for(int i=1;i<=n;i++) st[0][i] = a[i];
for(int i=1;(1<<i)<=n;i++)
    for(int j=1;j+(1<<i)-1<=n;j++)
        st[i][j] = max(st[i-1][j], st[i-1][j+(1<<(i-1))]);
auto query = [&](int l,int r)
{
    int k = log2(r-l+1);
    return max(st[k][l], st[k][r-(1<<k)+1]);
};
```

二维

```
vector<vector<int>> a(n+1, vector<int>(m+1));
// init a
vector<vector<vector<int>>> st(10, vector<vector<int>>(n+1, vector<int>(m+1)));
for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++)
        st[0][i][j] = a[i][j];
for(int k=1;k<=20;k++)
    for(int i=1;i+(1<<k)-1<=n;i++)
        for(int j=1;j+(1<<k)-1<=m;j++)
            st[k][i][j] = min({
                st[k-1][i][j],
                st[k-1][i+(1<<(k-1))][j],
                st[k-1][i][j+(1<<(k-1))],
                st[k-1][i+(1<<(k-1))][j+(1<<(k-1))]
            });
auto query = [&](int r,int c,int s)  // (r, c) 为左上角的 s*s 方阵
{
    int k = log2(s);
    return min({
        st[k][r][c],
        st[k][r+s-(1<<k)][c],
        st[k][r][c+s-(1<<k)],
        st[k][r+s-(1<<k)][c+s-(1<<k)]
    });
};
```

单调队列

```
int a[maxn];
vector<int> solve(int n, int k, auto rule=less<int>())   //? less: min, greater: max
{
    vector<int> res; deque<int> q;
    for(int i=1;i<=n;i++)
    {
        if(!q.empty()&&q.front()+k==i) q.pop_front();
        while(!q.empty()&&rule(a[i],a[q.back()])) q.pop_back();
        q.push_back(i);
        if(i>=k) res.push_back(a[q.front()]);
    }
    return res;
}
```

单调栈

```
int a[maxn];
vector<int> solve(int n)
{
    vector<int> res; stack<int> st;
    for(int i=n;i>=1;i--) //? 反向遍历代表第 i 个元素之“后”
    {
        while(!st.empty()&&a[st.top()]<=a[i]) st.pop(); //? 小于等于号代表第一个“大于” a_i 的元素的下标
        res.push_back(st.empty()?0:st.top());
        st.push(i);
    }
    reverse(res.begin(), res.end());
    return res;
}
```

树状数组

```
int n; ll c[maxn];
int lowbit(int x) { return x&-x; }
void modify(int p,int x) { for(int i=p;i<=n;i+=lowbit(i)) c[i] += x; }
int query(int p)    // [1,p] 之和
{
    ll sum = 0;
    for(int i=p;i;i-=lowbit(i)) sum += c[i];
    return sum;
}
```

线段树分裂合并

```
struct Node { int l,r; ll val; } sgt[maxn*40];      //? 40 = 2*maxm*log2(maxn)
int cnt, root[maxn];
inline void pushup(int k) { sgt[k].val = sgt[sgt[k].l].val + sgt[sgt[k].r].val; }
void modify(int l,int r,int &k,int p,int x)      // 单点修改: p 位置的值加上 x, 空间复杂度 O(logn)
{
    if(!k) k = ++cnt;      // 如果到了 NULL 结点就新建一个
    sgt[k].val += x;     // 单点修改的加法直接一条线上全部加上 x 即可
    if(l==r) return;
    int m = (l+r)>>1;
```

```
    if(p<=m) modify(l, m, sgt[k].l, p, x);
    else modify(m+1, r, sgt[k].r, p, x);
}
void merge(int &x,int y)         // 把 y 子树的内容合并到 x 子树上, 此写法不消耗空间
{
    if(!(x&&y)) x |= y;            // 如果二者有 NULL 结点
    else
    {
        sgt[x].val += sgt[y].val;    // 维护加法, 直接加就是了
        merge(sgt[x].l, sgt[y].l);  // 递归合并两结点的左子树
        merge(sgt[x].r, sgt[y].r);  // 递归合并两结点的右子树
    }
}
int split(int l,int r,int &k,int x,int y)    // 从 k 子树中分离出 [x,y] 区间并返回新结点编号, 空间复杂度 O(2logn)
{
    int n = ++cnt;
    if(x<=l&&y>=r)       // 如果 k 结点维护的区间在 [x,y] 中
    {
        sgt[n] = sgt[k];     // 直接拿过来便是
        k = 0;               // 置为 NULL, 断掉联系
        return;
    }
    int m = (l+r)>>1;
    if(x<=m) sgt[n].l = split(l, m, sgt[k].l, x, y);
    if(y>m)  sgt[n].r = split(m+1, r, sgt[k].r, x, y);
    pushup(k); pushup(n);
    return n;
}
ll query(int l,int r,int k,int x,int y)
{
    if(x<=l&&y>=r) return sgt[k].val;
    int m = (l+r)>>1;
    ll sum = 0;
    if(x<=m) sum += query(l,m,sgt[k].l,x,y);
    if(y>m)  sum += query(m+1,r,sgt[k].r,x,y);
    return sum;
}
```

线段树分裂合并解决多次区间正反排序问题

```
struct Node { int ch[2],val; } sgt[70*maxn];
int cnt;
inline int& ls(int k) { return sgt[k].ch[0]; }
inline int& rs(int k) { return sgt[k].ch[1]; }
int modify(int l,int r,int p)       // 单点修改, 固定修改 p 位置为 1
{
    int n = ++cnt;
    sgt[n].val = 1;
    if(l!=r)
    {
        int m = (l+r)>>1;
        if(p<=m) ls(n) = modify(l, m, p);
        else rs(n) = modify(m+1, r, p);
    }
    return n;
}
void merge(int &x, int y)            // 经典 merge, 略
```

```
{
    if(!x||!y) x |= y;
    else
    {
        sgt[x].val += sgt[y].val;
        merge(ls(x), ls(y));
        merge(rs(x), rs(y));
    }
}
int split(int k, int kth, bool rev)     // 把前 kth 个数字之外的部分分裂出去并返回结点编号
{
    if(sgt[k].val==kth) return 0;     // 如果不够 kth 个无法分裂（不会出现 < 的情况所以 == 相当于 <=）
    int n = ++cnt;
    sgt[n].val = sgt[k].val-kth;         // 分裂出去的部分的数量为总数减 kth 个
    sgt[k].val = kth;                    // 剩下 kth 个
    int val = sgt[sgt[k].ch[rev]].val;  // 靠前子树值，如果正序那么就是判断左子树数字数量，如果倒序那么就是判断右子树数字数量
    if(val>=kth)                         // 如果在靠前子树
    {
        sgt[n].ch[rev] = split(sgt[k].ch[rev], kth, rev);   // 分裂靠前子树
        sgt[n].ch[!rev] = sgt[k].ch[!rev];      // 靠前子树只剩 kth 个，靠后子树自然是要归给新结点（新结点是剩余部分嘛）
        sgt[k].ch[!rev] = 0;                    // 与分出去的结点断开联系
    }
    else sgt[n].ch[!rev] = split(sgt[k].ch[!rev], kth - val, rev);  // 如果在靠后子树，直接分裂即可，因为靠前子树还是应该属于老树
    return n;
}
typedef tuple<int, bool, int> tp3;  // < 维护区间的左端点，是否倒序排序，根结点编号 >
// 只以存放的数字个数来去重，不写这个仿函数就会因为继续比较 tuple 的另两个元素而导致未去重
struct Cmp { bool operator() (const tp3& t1, const tp3& t2) const { return get<0>(t1) < get<0>(t2); } };
set<tp3, Cmp> rt;
void data(int l, int r, int k, vector<int>& v)  // 取序列
{
    if(!k) return;
    if(l==r) v.push_back(l);
    int m = (l+r)>>1;
    data(l, m, ls(k), v);
    data(m+1, r, rs(k), v);
}
vector<int> print(int l, int r)
{
    vector<int> ret;
    for(auto t : rt)
    {
        vector<int> v;
        data(l, r, get<2>(t), v);
        if(get<1>(t)) ret.insert(ret.end(), v.rbegin(), v.rend());
        else ret.insert(ret.end(), v.begin(), v.end());
    }
    return ret;
}
auto split(int p)   // 使得 rt 中存在以 p 为区间左端点的元素
{
    auto it = rt.lower_bound(tp3(p, false, 0));
    if(get<0>(*it)==p) return it;   // 满足条件
    it--;   // 否则 p 一定在前一个元素中
    int l, n; bool rev;
    tie(l, rev, n) = *it;
```

```
    // 给剩下 p-l 个元素，其余的分裂成新树，这样 p 就会是其所在线段树维护区间的左端点了
    return rt.insert(tp3(p, rev, split(n, p-l, rev))).first;
}
void solve(int l, int r, bool rev)
{
    if(l>r) return;
    auto itl = split(l), itr = split(r+1);
    int n = 0;
    for(auto it = itl; it!=itr; ++it) merge(n, get<2>(*it)); // 为使他们具有相同的排序顺序，合并起来再塞回去
    rt.erase(itl, itr);                        // 从 rt 中删去老信息
    rt.insert(tp3(l, rev, n));                 // 塞回去
}
int n,m;
cin>>n>>m;
for(int i=1;i<=n;i++)
{
    int p;
    cin>>p;
    rt.insert(tp3(i, 0, modify(1,n,p)));
}
while(m--)
{
    int opt,l,r;
    cin>>opt>>l>>r;
    solve(l,r,opt); // opt 为 0 是升序排序
}
for(auto i : print(1, n)) cout<<i<<' ';
```

带修主席树（树状数组套值域线段树）

```
const int maxn = 1e5+5;
vector<int> o;
int getid(int x) { return lower_bound(o.begin(), o.end(), x)-o.begin()+1; };
struct Node { int l,r,val; } sgt[maxn*400];      // O(nlog^2n)
int cnt, root[maxn];
int& ls(int k) { return sgt[k].l; }
int& rs(int k) { return sgt[k].r; }
void modify(int l,int r,int p,int x,int& k)
{
    if(!k) k = ++cnt;
    sgt[k].val += x;
    if(l==r) return;
    int m = (l+r)>>1;
    if(p<=m) modify(l,m,p,x,ls(k));
    else modify(m+1,r,p,x,rs(k));
}
int query(int l,int r,vector<int>& L,vector<int>& R,int k)  // 区间 k 小
{
    if(l==r) return l;
    int m = (l+r)>>1;
    int sum = 0;
    for(auto i : R) sum += sgt[ls(i)].val;
    for(auto i : L) sum -= sgt[ls(i)].val;
    if(k<=sum)
    {
        transform(L.begin(), L.end(), L.begin(), ls);
```

```
        transform(R.begin(), R.end(), R.begin(), ls);
        return query(l,m,L,R,k);
    }
    else
    {
        transform(L.begin(), L.end(), L.begin(), rs);
        transform(R.begin(), R.end(), R.begin(), rs);
        return query(m+1,r,L,R,k-sum);
    }
}
int n;
int lowbit(int x) { return x&-x; }
void modify(int p,int q,int x)
{
    for(int i=p;i<=n;i+=lowbit(i)) modify(1,o.size(),q,x,root[i]);
}
int query(int l,int r,int k)
{
    vector<int> L,R;
    for(int i=r;i;i-=lowbit(i)) R.push_back(root[i]);
    for(int i=l-1;i;i-=lowbit(i)) L.push_back(root[i]);
    return query(1,o.size(),L,R,k);
}

int m;
cin>>n>>m;
vector<int> v(n+1);
for(int i=1;i<=n;i++) cin>>v[i];
o.insert(o.end(), v.begin()+1, v.end());
vector<tuple<int,int,int>> w;
for(int i=0;i<m;i++)
{
    char opt;
    cin>>opt;
    if(opt=='Q')
    {
        int l,r,k;
        cin>>l>>r>>k;
        w.emplace_back(l,r,k);  // 查 [l,r] k 小
    }
    else
    {
        int p,x;
        cin>>p>>x;
        w.emplace_back(p,x,-1); // p 位置改为 x
        o.push_back(x);
    }
}
sort(o.begin(), o.end());
o.erase(unique(o.begin(), o.end()), o.end());
for(int i=1;i<=n;i++) modify(i, getid(v[i]), 1);
for(auto [x,y,z] : w)
{
    if(~z) cout<<o[query(x,y,z)-1]<<'\n';
    else
    {
```

```
        modify(x, getid(v[x]), -1);
        v[x] = y;
        modify(x, getid(v[x]), 1);
    }
}
```

数论

快速幂

```
ll qpow(ll a,ll k,ll p)
{
    ll res = 1;
    for(;k;k>>=1,a=a*a%p)
        if(k&1) res=res*a%p;
    return res;
}
```

扩展欧几里得

解 $ax+by=\gcd(a,b)$ 的一组可行解。解保证 $|x|\le b,\ |y|\le a$

```
ll exgcd(ll a,ll b,ll& x,ll& y)
{
    if(!b) { x=1; y=0; return a; }
    ll d = exgcd(b, a%b, y, x);
    y -= (a/b)*x;
    return d;
}
```

乘法逆元

```
ll inv(ll a, int p)
{
    ll x,y;
    ll d = exgcd(a,p,x,y);
    return d==1?(x+p)%p:-1;
}
```

$\Theta(n)$ 求 $1\sim n$ 的逆元

```
int n=1e6, p=998244353;
vector<ll> res(n+1);
res[1] = 1;
for(int i=2;i<=n;i++) res[i] = (ll)(p-p/i)*res[p%i]%p; // 结果: res
```

$\Theta(n)$ 求任意 n 个数的逆元

```
vector<int> v;  // 所给 n 个数
int n=v.size(), p=998244353;
vector<ll> s(n+1), sv(n+1), res(n);
s[0] = 1;
for(int i=1;i<=n;i++) s[i] = s[i-1]*v[i-1]%p;
sv[n] = inv(s[n], p);
for(int i=n;i>=1;i--) sv[i-1] = sv[i]*v[i-1]%p;
for(int i=0;i<n;i++) res[i] = sv[i+1]*s[i]%p; // 结果: res
```

欧拉函数

$\varphi(x)$ 代表小于等于 x 的与 x 互质的数的个数，$\varphi(1) = 1$

```
ll phi(ll x)
{
    ll res = x;
    for(ll i=2;i*i<=x;i++)
    {
        if(x%i) continue;
        res = res/i*(i-1);
        while(x%i==0) x /= i;
    }
    if(x>1) res = res/x*(x-1);
    return res;
}
```

欧拉定理/扩展欧拉定理

$$\gcd(a,m) = 1 \to a^{\varphi(m)} \equiv 1 \pmod m$$

$$a^b \equiv \begin{cases} a^{b \bmod \varphi(m)}, & \gcd(a,m) = 1, \\ a^b, & \gcd(a,m) \neq 1, b < \varphi(m), \\ a^{(b \bmod \varphi(m)) + \varphi(m)}, & \gcd(a,m) \neq 1, b \geq \varphi(m). \end{cases} \pmod m$$

扩展欧拉定理求 $a^k \bmod p$，k 非常大

```
ll exeuler(ll a, const string& k, int p)
{
    ll phip = phi(p);
    ll t = 0;
    bool flag = false;
    for(auto c : k)
    {
        t = t*10+c-'0';
        if(t>phip)
        {
            t %= phip;
            flag = true;
        }
    }
    if(flag) t += phip;
    return qpow(a, t, p);
}
```

线性筛

```
int n = 1e6;
vector<bool> vis(n+1);
vector<int> pri;    // n 以内质数
vector<int> phi(n+1);   // 1~n 欧拉函数
phi[1] = 1;
for(int i=2;i<=n;i++)
{
    if(!vis[i])
```

```
    {
        pri.push_back(i);
        phi[i] = i-1;
    }
    for(auto j : pri)
    {
        if((ll)i*j>n) break;
        vis[i*j] = true;
        if(i%j==0)
        {
            phi[i*j] = phi[i]*j;
            break;
        }
        else phi[i*j] = phi[i]*(j-1);
    }
}
```

中国剩余定理

$$\begin{cases} x & \equiv a_1 \pmod{r_1} \\ x & \equiv a_2 \pmod{r_2} \\ & \vdots \\ x & \equiv a_k \pmod{r_k} \end{cases}$$

保证模数 r 两两互质，求解 x

```
ll crt(const vector<ll>& a, const vector<ll>& r)
{
    int k = a.size();
    ll n=1, ans=0;
    for(int i=0;i<k;i++) n = n*r[i];
    for(int i=0;i<k;i++)
    {
        ll m = n/r[i], b, y;
        exgcd(m, r[i], b, y);
        ans = (ans+a[i]*m*b%n)%n;
    }
    return (ans%n+n)%n;
}
```

数论分块

$$\sum_{i=1}^{n} f(i) \left\lfloor \frac{n}{i} \right\rfloor$$

```
ll sum = 0;
for(ll l=1,r;l<=n;l=r+1)
{
    r = n/(n/l);
    // 显然若没有 f(i) 即 f(i)=1，把 pre[r]-pre[l-1] 替换为 r-l+1
    sum += (pre[r]-pre[l-1])*(n/l);
}
cout<<sum<<endl;
```

树

树的直径

树上两个结点之间最长的一条简单路径是树的直径。

两次 dfs

1. 从任意结点 x 开始进行第一次 dfs，找到距离 x 最远的结点，记作 y；
2. 从 y 结点开始进行第二次 dfs，找到距离 y 最远的结点，记作 z。

答案（直径）为：$y \rightsquigarrow z$

局限性：树上边权非负。

```
int fur, dep[maxn];
void dfs(int u,int f)
{
    dep[u] = dep[f]+1;
    for(int i=Head[u];~i;i=Edge[i].next)
    {
        int v = Edge[i].to;
        if(v==f) continue;
        dfs(v,u);
        if(dep[v]>dep[fur]) fur = v;
    }
}
dfs(1,0); dfs(fur,0);
ans = dep[fur]-1;
```

树上 dp

任一根结点（1 号）

- 设计状态：$dp[u]$ 代表以 1 号结点为根结点时，从结点 u 往其子树走能够到达的最远距离。
- 初始状态：$dp[l] = 0$
- 状态转移方程：

$$dp[u] = \max_{v \in son[u]} (dp[v] + w(u,v))$$

经过 u 结点的最长链的长度，记作 $L[u]$，有：

$$\begin{aligned} L[u] &= \max_{v_1,v_2 \in son[u]} (dp[v_1] + w(u,v_1) + w(u,v_2) + dp[v_2]) \\ &= \max_{v_1 \in son[u]} (dp[v_1] + w(u,v_1) + \max_{v_2 \in son[u]} (dp[v_2] + w(u,v_2)) \qquad (v_1 \neq v_2) \\ &= \max_{v_1,v_2 \in son[u]} (dp[v_1] + w(u,v_1) + dp[u]) \end{aligned}$$

答案（直径）为：$\max(L[u])$

```
int ans, dp[maxn];
void dfs(int u,int f)
{
    for(int i=Head[u];~i;i=Edge[i].next)
    {
        int v = Edge[i].to;
        if(v==f) continue;
        dfs(v,u);
        ans = max(ans, dp[v]+1+dp[u]);
        dp[u] = max(dp[u],dp[v]+1);
    }
}
dfs(1,0);
```

树的重心

不带点权

```
int n,rt,siz[maxn],maxp[maxn];
void dfs1(int u,int f)
{
    siz[u] = 1;
    for(int i=Head[u];~i;i=Edge[i].next)
    {
        int v = Edge[i].to;
        if(v==f) continue;
        dfs1(v,u);
        siz[u] += siz[v];
        maxp[u] = max(maxp[u], siz[v]);
    }
    maxp[u] = max(maxp[u],n-siz[u]);
    if(maxp[u]<maxp[rt]) rt = u;
}
ll ans;
void dfs2(int u,int f,int dep)
{
    ans += dep;
    for(int i=Head[u];~i;i=Edge[i].next)
    {
        int v = Edge[i].to;
        if(v==f) continue;
        dfs2(v,u,dep+1);
    }
}
maxp[rt] = n;
dfs1(1,0); dfs2(rt,0,0);
```

带点权

解决方法：树上 DP

有如下定义：

1. 权值：$v[u]$
2. 深度：$dep[u]$ （根结点为 0）
3. 子树大小（权值和）：$siz[u]$

- 设计状态：$dp[u]$ 代表以 u 作为根结点时的答案（总距离）

- 初始状态:

$$dp[rt]=\sum_{u\in V} v[u]\times dep[u]$$

- 状态转移: $dp[u]=dp[f]-siz[u]+(siz[1]-siz[u])$
- 结果:

$$\min_{u\in V}(dp[u])$$

```
int v[maxn], siz[maxn];
ll dp[maxn];
void dfs1(int u,int f=0,int dep=0)
{
    siz[u] = v[u];
    for(int i=Head[u];~i;i=Edge[i].next)
    {
        int v = Edge[i].to;
        if(v==f) continue;
        dfs1(v,u,dep+1);
        siz[u] += siz[v];
    }
    dp[1] += v[u]*dep;
}
ll ans = inf;
void dfs2(int u,int f=0)
{
    ans = min(ans, dp[u]);
    for(int i=Head[u];~i;i=Edge[i].next)
    {
        int v = Edge[i].to;
        if(v==f) continue;
        dp[v] = dp[u]-siz[v]+siz[1]-siz[v];
        dfs2(v,u);
    }
}
dfs1(1); dfs2(1);
```

树上随机游走

设 $f(u)$ 代表 u 结点走到其父结点 p_u 的期望距离，则有:

$$f(u)=\sum_{(u,t)\in E} w(u,t)+\sum_{v\in son_u} f(v)$$

初始状态为 $f(leaf)=1$。当树上所有边的边权都为 1 时，上式可化为:

$$f(u)=d(u)+\sum_{v\in son_u} f(v)$$

即 u 子树的所有结点的度数和，也即 u 子树大小的两倍 -1。

设 $g(u)$ 代表 p_u 结点走到其子结点 u 的期望距离，则有:

$$g(u)=\sum_{(p_u,t)\in E} w(p_u,t)+g(p_u)+\sum_{s\in sibling_u} f(s)$$

初始状态为 $g(root)=0$。当树上所有边的边权都为 1 时，上式可化为:

$$g(u) = d(p_u) + g(p_u) + \sum_{s \in sibling_u} f(s)$$

```
int d[maxn], siz[maxn], ss[maxn], dp[maxn]; // 预处理 d 为结点度数
void dfs1(int u,int f)
{
    siz[u] = 1;
    for(auto v : G[u])
    {
        if(v==f) continue;
        dfs1(v,u);
        siz[u] += siz[v];
        ss[u] += 2*siz[v]-1;
    }
}
void dfs2(int u,int f)
{
    if(u!=1) dp[u] = dp[f]+d[f]+ss[f]-(2*siz[u]-1);
    for(auto v : G[u])
    {
        if(v==f) continue;
        dfs2(v,u);
    }
}
dfs1(1,0); dfs2(1,0);
//? f(u) = siz[u]*2-1, g[u] = dp[u]
```

树上背包

$dp[i][j]$ 表示 i 子树中在 j 的容量范围内，最大可以获得多少收益。

答案显然 $dp[root][W]$

```
function<void(int,int)> dfs = [&](int u,int f)
{
    for(auto v : G[u])
    {
        if(v==f) continue;
        dfs(v,u);
        for(int j=m;j>=0;j--)    // 当前子树使用多少容量
            for(int k=0;k<=j;k++)    // 给 v 子树分配 k 容量
                dp[u][j] = max(dp[u][j], dp[u][j-k]+dp[v][k]+v[u]);
    }
}
```

有依赖的树上背包

如果选择一个物品，则必须选择它的父结点

```
function<void(int)> dfs = [&](int u,int f)
{
    for(int i=w[u];i<=W;i++) dp[u][i] = v[u];    // 因为要选儿子必选自己所以先把自己选上
    for(auto v : G[u])
    {
        if(v==f) continue;
        dfs(v,u);
        for(int j=W;j>=w[u];j--)
```

```cpp
            for(int k=0;k<=j-w[u];k++)
                dp[u][j] = max(dp[u][j], dp[u][j-k]+dp[v][k]);
    }
};
```

也可以利用 dfs 序 $O(n^2)$ 快速 dp 出来

```cpp
int tim = 0;
vector<int> nfd(n+1), siz(n+1), pre(n+1);
function<void(int)> dfs = [&](int u)
{
    nfd[tim++] = u;
    siz[u] = 1;
    for(auto v : G[u])
    {
        pre[v] = pre[u]+w[u];
        dfs(v);
        siz[u] += siz[v];
    }
};
dfs(1);
vector<vector<int>> dp(tim+1, vector<int>(W+1));
for(int i=0;i<tim;i++)
{
    for(int j=pre[nfd[i]];j<=W-w[nfd[i]];j++)    // 选当前结点
        dp[i+1][j+w[nfd[i]]] = max(dp[i+1][j+w[nfd[i]]], dp[i][j]+v[nfd[i]]);
    for(int j=pre[nfd[i]];j<=W;j++)              // 不选当前结点
        dp[i+siz[nfd[i]]][j] = max(dp[i+siz[nfd[i]]][j], dp[i][j]);
}
cout<<dp[tim][W]<<endl;
```

组合数学

排列组合

球盒模型

n 个小球放到 m 个盒子里:

- 球相同，盒子不同，不能有空盒: $\binom{n-1}{m-1}$
- 球相同，盒子不同，可以有空盒: $\binom{n+m-1}{n}$
- 球不同，盒子不同，可以有空盒: m^n
- 球不同，盒子相同，不能有空盒: $S(n, m)$，S 为第二类斯特林数（见后文）
- 球不同，盒子不同，不能有空盒: $m! \times S(n, m)$
- 球不同，盒子相同，可以有空盒: $\sum_{i=1}^{m} S(n, i)$
- 球相同，盒子相同，可以有空盒:

  设 $f[n][m]$ 为 n 个球放到 m 个盒子里的方案数

  如果只有一个盒子或者没有小球，方案数自然为 1

  如果小球比盒子要少，小球肯定是放不满盒子的，由于盒子相同，可以得到转移 $f[i][j] = f[i][i]$

  如果小球比盒子要多，就分为将盘子放满和没放满两种情况，即 $f[i][j] = f[i-j][j] + f[i][j-1]$

  ```
  for(int i=0;i<=n;i++)
  {
      for(int j=1;j<=m;j++)
      {
          if(i==0||j==1) f[i][j] = 1;
          else if(i<j) f[i][j] = f[i][i];
          else if(i>=j) f[i][j] = f[i-j][j]+f[i][j-1];
      }
  }
  ```

- 球相同，盒子相同，不能有空盒:

  假设在每一个盒子里都放上了一个球，就跟上面的情况一样了。结果: $f[n-m][m]$

组合数性质

- $\sum_{i=0}^{n} \binom{n}{i}^2 = \binom{2n}{n}$
- $\sum_{i=0}^{n} i\binom{n}{i} = n2^{n-1}$
- $\sum_{i=0}^{n} i^2\binom{n}{i} = n(n+1)2^{n-2}$
- $\sum_{i=0}^{n} \binom{i}{m} = \binom{n+1}{m+1}$
- $\sum_{i=0}^{n} \binom{n-i}{i} = F_{n+1}$，$F$ 为斐波那契数列

二项式定理

$$(a+b)^n=\sum_{i=0}^{n}\binom{n}{i}a^{n-i}b^i$$

- $\sum_{i=0}^{n}\binom{n}{i}=2^n$
- $\sum_{i=0}^{n}(-1)^i\binom{n}{i}=[n=0]$

错位排列

前几项: $0,1,2,9,44,265$

对于 $1\sim n$ 的排列 P, 如果满足 $P_i\neq i$, 则称 P 是 n 的错位排列。

$$D_n=(n-1)(D_{n-1}+D_{n-2})$$

$$D_n=nD_{n-1}+(-1)^n$$

$$D_n=\left\lfloor\frac{n!}{e}\right\rfloor$$

圆排列

n 个物品选 m 个排成一个环的方案数

$$Q_n^m=\frac{A_n^m}{m}=\frac{n!}{r\times(n-r)!}$$

其他

n 个完全相同的元素, 要求将其分为 m 组, 要求每组至少有 a_i 个元素 ($\sum a_i\leq n$), 一共有 $\binom{n-\sum a_i+k-1}{n-\sum a_i}$ 种分法

$1\sim n$ 中选 m 个, 这 m 个数中任何两个数都不相邻的组合有 $\binom{n-m+1}{m}$ 种。

斯特林数

第二类斯特林数

$$S(n,m)$$

$$\left\{\begin{matrix}n\\m\end{matrix}\right\}$$

表示将 n 个两两不同的元素, 划分为 m 个互不区分的非空子集的方案数。

递推:

$$\left\{\begin{matrix}n\\m\end{matrix}\right\}=\left\{\begin{matrix}n-1\\m-1\end{matrix}\right\}+m\left\{\begin{matrix}n-1\\m\end{matrix}\right\}$$

$$\left\{\begin{matrix}n\\0\end{matrix}\right\}=[n=0]$$

通项:

$$\begin{Bmatrix} n \\ m \end{Bmatrix} = \sum_{i=0}^{m} \frac{(-1)^{m-i} i^n}{i!(m-i)!}$$

第一类斯特林数

$$s(n, m)$$

$$\begin{bmatrix} n \\ m \end{bmatrix}$$

表示将 n 个两两不同的元素，划分为 m 个互不区分的非空轮换的方案数。

一个轮换就是一个首尾相接的环形排列。我们可以写出一个轮换 $[A, B, C, D]$，并且 $[A, B, C, D] = [B, C, D, A] = [C, D, A, B] = [D, A, B, C]$，即，两个可以通过旋转而互相得到的轮换是等价的。注意，两个可以通过翻转而相互得到的轮换不等价，即 $[A, B, C, D] \neq [D, C, B, A]$。

$$\begin{bmatrix} n \\ m \end{bmatrix} = \begin{bmatrix} n-1 \\ m-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ m \end{bmatrix}$$

$$\begin{bmatrix} n \\ 0 \end{bmatrix} = [n = 0]$$

卡特兰数

前几项：$1, 1, 2, 5, 14, 42, 132$

1. 有 $2n$ 个人排成一行进入剧场。入场费 5 元。其中只有 n 个人有一张 5 元钞票，另外 n 人只有 10 元钞票，剧院无其它钞票，问有多少种方法使得只要有 10 元的人买票，售票处就有 5 元的钞票找零？
2. 一位大城市的律师在她住所以北 n 个街区和以东 n 个街区处工作。每天她走 $2n$ 个街区去上班。如果他从不穿越（但可以碰到）从家到办公室的对角线，那么有多少条可能的道路？
3. 在圆上选择 $2n$ 个点，将这些点成对连接起来使得所得到的 n 条线段不相交的方法数？
4. 对角线不相交的情况下，将一个凸多边形区域分成三角形区域的方法数？
5. 一个栈（无穷大）的进栈序列为 $1, 2, 3, \cdots, n$ 有多少个不同的出栈序列？
6. n 个结点可构造多少个不同的二叉树？

$$H_n = \frac{\binom{2n}{n}}{n+1} (n \geq 2, n \in \mathbf{N}_+)$$

$$H_n = \begin{cases} \sum_{i=1}^{n} H_{i-1} H_{n-i} & n \geq 2, n \in \mathbf{N}_+ \\ 1 & n = 0, 1 \end{cases}$$

$$H_n = \frac{H_{n-1}(4n-2)}{n+1}$$

$$H_n = \binom{2n}{n} - \binom{2n}{n-1}$$

贝尔数

前几项: $1,1,2,5,15,52,203$

$$B_n$$

基数为 n 的集合的划分方法的数目。例如 3 个元素的集合 $\{a,b,c\}$ 有 5 种不同的划分方法: $\{\{a\},\{b\},\{c\}\},\{\{a\},\{b,c\}\},\{\{b\},\{a,c\}\},\{\{c\},\{a,b\}\},\{\{a,b,c\}\}$

$$B_{n+1}=\sum_{k=0}^{n}\binom{n}{k}B_k$$

$$B_n=\sum_{k=0}^{n}S(n,k)$$

贝尔三角形

用以下方法构造一个三角矩阵 (形式类似杨辉三角形):

- 第一行第一项为 1 $(a_{1,1}=1)$;
- 对于 $n>1$, 第 n 行第一项等于第 $n-1$ 行的第 $n-1$ 项 $(a_{n,1}=a_{n-1,n-1})$;
- 对于 $m,n>1$, 第 n 行的第 m 项等于它左边和左上角两个数之和 $(a_{n,m}=a_{n,m-1}+a_{n-1,m-1})$

每行的首项是贝尔数。可以利用这个三角形来递推求出 Bell 数。

卢卡斯定理

$$\binom{n}{m} \bmod p=\binom{\lfloor n/p\rfloor}{\lfloor m/p\rfloor}\cdot\binom{n \bmod p}{m \bmod p} \bmod p$$

```
ll fac[maxn];   //* 预处理阶乘到 p-1
ll C(ll n,ll m,ll p)
{
    return n>=m?fac[n]*qpow(fac[m],p-2,p)%p*qpow(fac[n-m],p-2,p)%p:0;
}
ll lucas(ll n,ll m,ll p)    // C(n,m) mod p
{
    return m?lucas(n/p, m/p, p)*C(n%p, m%p, p)%p:1;
}
```

康托展开

求一个排列的排名, 可用于哈希

```
//* include 树状数组
ll fac[maxn];   //* 预处理阶乘到 n
ll cantor(const vector<int>& v)
{
    for(int i=1;i<=n;i++) modify(i, 1);
    ll sum = 1;
    for(int i=1;i<=n;i++)
    {
        modify(v[i-1], -1);
        sum = (sum+fac[n-i]*query(v[i-1]))%mod;
    }
    return sum;
}
```

逆康托展开

以第 38 名长度为 5 的排列为例：

1. $37 \div 4! = 1 \cdots 13$，故首位为 2；
2. $13 \div 3! = 2 \cdots 1$，故第二位为 4（前面已有一个 2）；
3. $1 \div 2! = 0 \cdots 1$，故第三位为 1；
4. $1 \div 1! = 1$，故第四位为 5（前面已有 $1, 2, 4$）；
5. 故第五位为 3，原排列为 $2, 4, 1, 5, 3$。

可以使用线段树维护，方法与康托展开相似。

$n \times m$ 网格中矩形数量

$$\frac{nm(n+1)(m+1)}{4}$$

计算几何

极角排序

```cpp
struct Point { double x,y; };
double cross(double x1,double y1,double x2,double y2)
{
    return (x1*y2-x2*y1);
}
double compare(const Point& a,const Point& b,const Point& c)
{
    return cross((b.x-a.x),(b.y-a.y),(c.x-a.x),(c.y-a.y));
}
bool cmp(const Point& p,const Point& q)
{
    Point O = {0,0};
    if(compare(O,p,q)==0) return p.x<q.x;
    else return compare(O,p,q)>0;
}
int quadrant(const Point& p)
{
    if(p.x>0&&p.y>=0) return 1;
    if(p.x<=0&&p.y>0) return 2;
    if(p.x<0&&p.y<=0) return 3;
    if(p.x>=0&&p.y<0) return 4;
}
bool operator<(const Point& p, const Point& q)
{
    if(quadrant(p)==quadrant(q)) return cmp(p,q);
    return quadrant(p)<quadrant(q);
}
// atan2 法
// bool operator<(const Point& p,const Point& q)
// {
//     if(atan2(p.y,p.x)!=atan2(q.y,q.x)) return atan2(p.y,p.x)<atan2(q.y,q.x);
//     else return p.x<q.x;
// }
```

欧拉公式

顶点数-棱长数 $+$ **表面数** $=2$