杂项

结论

子系列相关

如果一个数列长度为$n*2$,尝试构造出一个数组，使其所有长度为n的的子序列，对该子序列而言，其中所有元素的乘积等于其补集的所有元素的和，其构造出来是$n*2-1$个-1和一个n，
$n==1||n==2$ 须分类讨论，也包含全0的情况

等边三角形

不存在三个顶点都是整数的等边三角形，原因如下：

由皮克定理知，三个顶点都是整数的三角形其面积必定是整数或者整数+0.5的形式，而等边三角形的面积是无理数因此不存在三个顶点都是整数的等边三角形

数论相关

结论1：如果a,b均为正整数且互质，那么由$ax+by,x>=0,y>=0$不能凑出来的最大数是$ab-a-b$.

证明：

首先证明出$ab-a-b$不能被$ax+by,x>=0,y>=0$表示出

反证法，假设$ab-a-b=ax+by$,那么$ab=a(x+1)+b(y+1)$,由于

$a|ab,a|a(x+1)$,所以$a|b(y+1)$,由于a,b互质，所以$a|(y+1)$,由于$y>=0$,所以$a<=y+1$,所以$b(y+1)>=ab$,同理可得$a(x+1)>=ab$,所以$a(x+1)+b(y+1)>=2ab>ab$,矛盾。

结论二：这个题正解的复杂度是对的，虽然我也是看了官方题解之后才知道的。 利用的关键性质是这样的：对于两个素数 $p_1,p_2\leq 10^9$, $|\ p_1-p_2\ |<300$. 实际上，最大的差值为282. 同时，我们可以发现对于一个小于三百的偶数，一定能被表示成两个素数的和

结论三：哥德巴赫猜想： **（1）任何一个大于2的偶数都可以表示为两个素数之和； （2）任何一个大于5的奇数是3个素数之和**

结论四：$gcd(gcd(a,b),gcd(b,c))=gcd(a,b,c)$ 感性理解即可, 在这个结论的基础上如果判断一个数是否能成为当前数组的某两个数的最大公因数，那么可以直接判断这个数的所有倍数即可，因为首先这个数所有倍数的最大公因数 肯定不会比当前数小，其次，如果出现两个数的最大公因数是当前数的话，那么整个倍数的最大公因数肯定也是当前数。

结论五：给定x个数都是k的约数，那么这x个数的最小公倍数至少不超过k,从分解质因子的角度更好理解。

前缀和差分

二维前缀和

建议：直接抽象为函数 传入 $x1,y1,x2,y2$

```
void add(int x,int y)
{
    s[x][y] = s[x-1][y]+s[x][y-1]-s[x-1][y-1]+a[x][y];
}

ll sum(int x1,int y1,int x2,int y2)
{
    ll ans = 1ll*(s[x2][y2]-s[x1-1][y2]-s[x2][y1-1]+s[x1-1][y1-1]);
    return ans;
}
```

二维差分

```
void sub(int i,int j)
{
    c[i][j] = a[i][j]-a[i-1][j]-a[i][j-1]+a[i-1][j-1];
}

void insert(int x1,int y1,int x2,int y2,int s)
{
    c[x1][y1] += s;
    c[x2+1][y1] -= s;
    c[x1][y2+1] -= s;
    c[x2+1][y2+1] += s;
}
```

分数规划

分数规划用来求一个分式的极值。

形象一点就是，给出a_i和b_i，求一组$w_i \in 0,1$,最小化或最大化$\Sigma_1^n ai*wi/\Sigma_1^n bi*wi$

另外一种描述：每种物品有两个权值和$ai*bi$，选出若干个物品使得$\Sigma_1^n ai/\Sigma_1^n bi$最小/最大。

一般分数规划问题还会有一些奇怪的限制，比如分母至少为W

求解：

二分法

分数规划问题的通用方法是二分。

假设我们要求最大值。二分一个答案mid然后推式子（为了方便少写了上下界）：

推出$\Sigma_1^n wi*(a_i-mid*bi)>0$要求$a_i-mid*bi>0$

那么只要求出不等号左边的式子的最大值就行了。如果最大值比0要大，说明mid是可行的，否则不可行。

求最小值的方法和求最大值的方法类似。

代码：

```
const int N = 1e5+10;

int a[N],b[N];
double c[N];
int n,k; // 可以不选k个课程 也就是说选n-k个 来使答案最大
```

```
bool check(double mid)
{
    for(int i=1;i<=n;i++)
        c[i] = a[i]-mid*b[i]; // 计算加排序
    sort(c+1,c+n+1);
    double sum = 0;
    rep(i,k+1,n){
        sum += c[i];
    }
    return sum >= 0;
}

int main()
{
    while(cin >> n >> k && (n||k)){
        rep(i,1,n)
            cin >> a[i];
        rep(i,1,n)
            cin >> b[i];
        double l=0,r=1;
        while(fabs(r-l)>1e-4){ // 看精度要求 如果精度是1e-2 则这里为 1e-4 加2就行
            double mid = (r+l)/2;
            if(check(mid)){
                l = mid;
            }
            else{
                r = mid;
            }
        }
        int ans = (int)(100*l+0.5);
        cout << ans << endl;
    }

    return 0;
}
```

扫描线

重点：区间映射

将[l,r] 这段 映射到 [l,r+1] 这段里面，导致 最后一个值没有被用于建树上。查询也是同理，l,r 映射到 l,r-1 这段区间。

```
// 离散化的版本
#define lc(u) u<<1
#define rc(u) u<<1|1

const int N = 1e5+10;

int n;
int s[200010];

struct line{
    int x1,x2,y;
    int mark;
    bool operator<(const line &t){
```

```
        return y<t.y;
    }
}line[200010];

struct node{
    int l,r;
    int cnt,len;
}tr[N*8];

void build(int u,int l,int r)
{
    tr[u] = {l,r,0,0};
    if(l == r) return ;
    int mid = l+r>>1;
    build(lc(u),l,mid);
    build(rc(u),mid+1,r);
}

void pushup(int u)
{
    int l=tr[u].l,r=tr[u].r;
    if(l == r){
        if(tr[u].cnt) tr[u].len = s[r+1]-s[l];
        else tr[u].len = 0;
        return ;
    }
    if(tr[u].cnt) tr[u].len = s[r+1]-s[l];
    else tr[u].len = tr[lc(u)].len+tr[rc(u)].len;
}

void modify(int u,int l,int r,int mark)
{
    if(l<=tr[u].l && tr[u].r <= r){
        tr[u].cnt += mark;
        pushup(u);
        return ;
    }
    int mid = tr[u].l+tr[u].r>>1;
    if(l<=mid) modify(lc(u),l,r,mark);
    if(r>mid) modify(rc(u),l,r,mark);
    pushup(u);
}

int main()
{
    IOS;
    cin >> n;
    int x1,y1,x2,y2;
    rep(i,1,n){
        cin >> x1 >> y1 >> x2 >> y2;
        s[i] = x1;
        s[n+i] = x2;
        line[i] = {x1,x2,y1,1};
        line[i+n] = {x1,x2,y2,-1};
    }

    n <<= 1;
    sort(line+1,line+n+1);
```

```
    sort(s+1,s+n+1);
    int tot = unique(s+1,s+n+1)-s-1;
    build(1,1,tot-1);
    ll ans = 0;
    for(int i=1;i<n;i++){
        int l=lower_bound(s+1,s+tot+1,line[i].x1)-s;
        int r=lower_bound(s+1,s+tot+1,line[i].x2)-s;
        modify(1,l,r-1,line[i].mark);
        ans += 1ll*tr[1].len*(line[i+1].y-line[i].y);
    }
    cout << ans << endl;


    return 0;
}


// 值域连续的版本
#define lc(u) u<<1
#define rc(u) u<<1|1

const int N = 2e5+10;

int n;
int pre[N],nxt[N];
int a[N];

struct line{
    int x1,x2,y;
    int mark;
    bool operator<(const line &t){
        return y<t.y;
    }
}line[N*2];

struct node{
    int l,r;
    int cnt,len;
}tr[N*8];

void build(int u,int l,int r)
{
    tr[u] = {l,r,0,0};
    if(l == r) return ;
    int mid = l+r>>1;
    build(lc(u),l,mid);
    build(rc(u),mid+1,r);
}

void pushup(int u)
{
    int l=tr[u].l,r=tr[u].r;
    if(l == r){
        if(tr[u].cnt) tr[u].len = r+1-l;
        else tr[u].len = 0;
        return ;
    }
    if(tr[u].cnt) tr[u].len = r+1-l;
```

```
    else tr[u].len = tr[lc(u)].len+tr[rc(u)].len;
}

void modify(int u,int l,int r,int mark)
{
    if(l<=tr[u].l && tr[u].r <= r){
        tr[u].cnt += mark;
        pushup(u);
        return ;
    }
    int mid = tr[u].l+tr[u].r>>1;
    if(l<=mid) modify(lc(u),l,r,mark);
    if(r>mid) modify(rc(u),l,r,mark);
    pushup(u);
}

void solve()
{
    cin >> n;
    rep(i,1,n)
        cin >> a[i];
    map<int,int> mp;
    build(1,1,n);

    rep(i,1,n){
        if(mp.count(a[i])) pre[i] = mp[a[i]];
        else pre[i] = 0;
        mp[a[i]] = i;
    }
    mp.clear();
    per(i,n,1){
        if(mp.count(a[i])) nxt[i] = mp[a[i]];
        else nxt[i] = n+1;
        mp[a[i]] = i;
    }

    int x1,x2,y1,y2;
    rep(i,1,n){
        x1 = pre[i]+1;
        x2 = i;
        y1 = i;
        y2 = nxt[i]-1;
        line[i] = {x1,x2,y1,1};
        line[i+n] = {x1,x2,y2+1,-1};
    }
    sort(line+1,line+n*2+1);
    int j = 1;
    ll ans = 0;
    rep(i,1,n){
        while(j<=n*2 && line[j].y<=i){
            modify(1,line[j].x1,line[j].x2,line[j].mark);
            j++;
        }
        ans += tr[1].len;
    }
    if(ans == 1ll*(n+1)*n/2) cout << "non-boring" << endl;
    else cout << "boring" << endl;
}
```

```
int main()
{
    IOS;
    int t;
    cin >> t;
    while(t--)
        solve();

    return 0;
}

/*
2
5
1 2 3 4 5
5
1 1 1 1 1
*/
```

倍增

基于倍增的 ST 表

区间 GCD

只支持静态查询。

```
const int N = 1e5+10;
const int bit = 20;

int n;
int st[N][bit];

void init()
{
    for(int j=0;j<bit;j++){
        for(int i=1;i+(1<<j)-1<=n;i++){
            if(!j) st[i][j] = a[i];
            else st[i][j] = gcd(st[i][j-1],st[i+(1<<(j-1))][j-1]);
        }
    }
}

int query(int l,int r)
{
    int len = r-l+1;
    int k = log(len)/log(2);
    return gcd(st[l][k],st[r-(1<<k)+1][k]);
}
```

三分

枚举子集、超集

搜索

剪枝

迭代加深搜索

定义：迭代加深是一种 **每次限制搜索深度的** 深度优先搜索。

解释：迭代加深搜索的本质还是深度优先搜索，只不过在搜索的同时带上了一个深度d，当d达到设定的深度时就返回，一般用于找最优解。如果一次搜索没有找到合法的解，就让设定的深度加一，重新从根开始。

既然是为了找最优解，为什么不用 BFS 呢？我们知道 BFS 的基础是一个队列，队列的空间复杂度很大，当状态比较多或者单个状态比较大时，使用队列的 BFS 就显出了劣势。事实上，迭代加深就类似于用 DFS 方式实现的 BFS，它的空间复杂度相对较小。

当搜索树的分支比较多时，每增加一层的搜索复杂度会出现指数级爆炸式增长，这时前面重复进行的部分所带来的复杂度几乎可以忽略，这也就是为什么迭代加深是可以近似看成 BFS 的。

题目描述：给定数字n,要求数列第一个数字为1,最后一个数字为n,序列是严格单调递增的，并且每一个数都是由前面的两个数字之和所构成的。

分析：这道题用bfs会比较复杂，而分析后发现n为100时，搜索的深度不会太大，因为我可以前面全为二进制，最多搜索也就10层左右，因此可以利用迭代加深来代替bfs,在处理第u位时，采用倒着枚举的方式，优化复杂度。

代码：

```
const int N = 110;

int n;
int path[N];

bool dfs(int u,int k)
{
    if(u==k)
        return path[u-1] == n;

    bool st[N] = {0};
    for(int i=u-1;i>=0;i--){
        for(int j=i;j>=0;j--){
            int s=path[i]+path[j];
            if(s>n || s<=path[u-1] || st[s])
                continue;
            st[s] = true;
            path[u] = s;
            if(dfs(u+1,k))
                return true;
        }
    }

    return false;
}

int main()
{
    path[0] = 1;
```

```
    while(cin >> n,n){
        int k = 1;
        while(!dfs(1,k))
            k++;
        rep(i,0,k-1)
            cout << path[i] << " ";
        cout << endl;
    }

    return 0;
}
```

折半搜索

折半搜索的一个很重要的特点是我们搜索2^n的方案是会TLE的，但我们搜索两次$2^{n/2}$的方案则是不会TLE的，求解的关键是我们如何将原有的任务分成两部分,以及分成两部分之后如何合并的问题。

例题：达达帮翰翰给女生送礼物，翰翰一共准备了 N个礼物，其中第i个礼物的重量是 $g[i]$。达达的力气很大，他一次可以搬动重量之和不超过g的任意多个物品。达达希望一次搬掉尽量重的一些物品，请你告诉达达在他的力气范围内一次性能搬动的最大重量是多少。

数据范围：$1 <= N <= 46, 1 <= g[i] <= 2^{31} - 1$

分析：本题明显是一个背包问题，但第二维体积过大导致我们开不下这么大的数组，看到N的范围之后可以考虑爆搜，但直接搜索2^{46}肯定会超时，因此考虑折半搜索，我们先搜索出前2^{23}中的选择方案，将其存在一个数组里面，再搜索后一半数组，得到一个sum值，我们只需要在前一个数组里面二分出小于等于$m - sum$的最大值即可。复杂度为$O(2^{n/2} + 2^{n/2}log(2^{n/2}))$,这个复杂度是可以接受的（考虑优化的话就是将前面的数组变大一点，后面数组小一点，复杂度还会减少一些）

代码：

```
const int N = 50;
int n,m,k;
int w[N],cnt;
ll weight[1<<25];
ll ans;

void dfs1(int u,ll sum)
{
    if(u==k+1){
        weight[++cnt] = sum;
        return ;
    }

    dfs1(u+1,sum);
    if(sum+w[u]<=m)
        dfs1(u+1,sum+w[u]);
}

void dfs2(int u,ll sum)
{
    if(u == n+1){
        int l = 0, r = cnt;
        while(l<r){
            int mid = l+r+1>>1;
            if(weight[mid]<=m-sum)
                l = mid;
```

```
            else
                r = mid-1;
        }
        ans = max(ans,sum+weight[l]);
        return ;
    }

    dfs2(u+1,sum);
    if(sum+w[u]<=m)
        dfs2(u+1,sum+w[u]);
}

int main()
{
    cin >> m >> n;
    rep(i,1,n)
        cin >> w[i];
    sort(w+1,w+n+1);
    reverse(w+1,w+n+1); // 优化搜索顺序，从分支节点较小的搜索树开始搜索

    k = n/2;
    dfs1(1,0);
    sort(weight+1,weight+cnt+1);

    cnt = unique(weight+1,weight+cnt+1)-weight-1;
    dfs2(k+1,0);
    cout << ans << endl;

    return 0;
}
```

DLX

A*

本质上是BFS的改进,在搜索过程中引入估价函数,定义起点s，终点t，从起点（初始状态）开始的距离函数$g(x)$，每个点到终点的估价函数$f(x)$。定义$h(x)=g(x)+f(x)$,每次从优先队列中取出一个$h(x)$最小的值，然后更新相邻的状态，$A*$算法的正确性必须保证当前距离加估计距离小于等于真实距离，这样才会保证算法正确，当$f=0$,时$A*$算法变成$Dijkstra$;当$h=0$并且边权为1时变成BFS.$A*$算法的关键是其估价函数的设计。

例题：八数码

结论：起始状态逆序对为奇数的必定无解，因为移动左右并不改变逆序对的数量，而移动上下要么逆序对加2，要么减2要么不变。其估价函数就是当前点离终点的曼哈顿距离。

代码：

```
int f(string str)
{
    int res = 0;
    for(int i=0;i<str.size();i++){
        if(str[i] != 'x'){
            int t= str[i]-'1';
            res += abs(i/3-t/3)+abs(i%3-t%3);
        }
    }
```

```
    return res;
}

string bfs(string start)
{
    int dx[4]={-1,0,1,0};
    int dy[4]={0,1,0,-1};
    char op[4]={'u','r','d','l'};

    string end = "12345678x";
    unordered_map<string,int> dist;
    unordered_map<string,pair<string,char>> prev;

priority_queue<pair<int,string>,vector<pair<int,string>>,greater<pair<int,string
>>> heap;

    heap.push({f(start),start});
    dist[start] = 0;

    while(heap.size()){
        auto t=heap.top();
        heap.pop();

        string state = t.second;
        if(state == end)
            break;
        int step = dist[state];
        int x,y;
        for(int i=0;i<state.size();i++){
            if(state[i] == 'x'){
                x = i/3;
                y = i%3;
                break;
            }
        }

        string source = state;
        for (int i = 0; i < 4; i ++ )
        {
            int a = x + dx[i], b = y + dy[i];
            if (a >= 0 && a < 3 && b >= 0 && b < 3)
            {
                swap(state[x * 3 + y], state[a * 3 + b]);
                if (!dist.count(state) || dist[state] > step + 1)
                {
                    dist[state] = step + 1;
                    prev[state] = {source, op[i]};
                    heap.push({dist[state] + f(state), state});
                }
                swap(state[x * 3 + y], state[a * 3 + b]);
            }
        }
    }

    string res;
    while(end != start){
        res += prev[end].second;
        end = prev[end].first;
```

```
    }
    reverse(res.begin(),res.end());
    return res;
}

int main()
{
    string g,c,seq;

    while(cin >> c){
        g += c;
        if(c != "x")
            seq += c;
    }

    int t=0;
    for(int i=0;i<seq.size();i++){
        for(int j=i+1;j<seq.size();j++){
            if(seq[i]>seq[j])
                t++;
        }
    }

    if(t%2)
        puts("unsolvable");
    else
        cout << bfs(g) << endl;

    return 0;
}
```

应用：k短路

见图论相关章节。

IDA*

$IDA*$其实就是迭代加深与$A*$算法的组合，在搜索过程中，如果当前深度与估价得到的深度之和大于max_depth时，就直接返回，否则继续进行，本质上是对迭代加深搜索的一种剪枝。

参考代码：

```
const int N = 20;

int n;
int q[N];
int w[6][N];

int f()
{
    int cnt = 0;
    for (int i = 0; i + 1 < n; i ++ )
        if (q[i + 1] != q[i] + 1)
            cnt ++ ;
    return (cnt + 2) / 3;
}
```

```
bool check()
{
    for (int i = 0; i + 1 < n; i ++ )
        if (q[i + 1] != q[i] + 1)
            return false;
    return true;
}

bool dfs(int u,int max_depth)
{
    if(u+f()>max_depth) // 不同题的估价函数不一样，但搜索过程中的剪枝是一样的。
        return false;
    if(check())
        return true;

    for(int len=1;len<=n;len++){
        for(int l=0;l+len-1<n;l++){
            int r=l+len-1;
            for(int k=r+1;k<n;k++){
                memcpy(w[u], q, sizeof q);
                int x, y;
                for (x = r + 1, y = l; x <= k; x ++, y ++ )
                    q[y] = w[u][x];
                for (x = l; x <= r; x ++, y ++ )
                    q[y] = w[u][x];
                if (dfs(u + 1, max_depth))
                    return true;
                memcpy(q, w[u], sizeof q);
            }
        }
    }

    return false;
}

int main()
{
    int t;
    cin >> t;
    while(t--){
        cin >> n;
        rep(i,0,n-1)
            cin >> q[i];
        int k=0;
        while(k<5 && !dfs(0,k))
            k++;
        if(k>=5)
            puts("5 or more");
        else
            cout << k << endl;
    }

    return 0;
}
```

分治

树上分治

点分治

前置知识点：树的重心

用途：点分治适合处理大规模的树上路径信息问题，我们首先要找一遍树的重心，而后以该重心为根来解决问题，之所以要找树的重心来当根节点是为了保证时间复杂度，可以证明的是以树的重心为根的话，其分割出来的最大子树的$size$不会超过$n/2$，而这就能保证时间复杂度。难点在于在不同子树中如何合并解的过程。

时间复杂度：$O(nlogn)$

模板题：给定一个有 N 个点（编号 $0,1,\ldots,N-1$）的树，每条边都有一个权值（不超过 1000）。

树上两个节点 x与 y 之间的路径长度就是路径上各条边的权值之和。

求长度不超过 K 的路径有多少条。

思路：不妨设此时树的重心为u，我们把树上的路径分为两种，一种是不经过树的重心的，那这种情况我们可以通过递归的方式来解决，另一种是经过树的重心的，而经过树的重心的又可以分为两种，一种是树的重心不是其中一个端点的，另一种是端点的，其中如果树的重心是端点的话好处理，只需要记录一下其余所有节点到重心的距离即可，而如果树的重心不是端点的话，我们可以这样处理，获得所有节点到重心的距离，而后就是在这里面求任意两个数满足$a[i]+a[j]<=m$的情况，但这样会有重复计算的情况，比如说在同一子树内有两点的距离到重心的距离之和$s<=m$，而这两点必然能走到与重心所相连的那个点，由于边权是$>=0$的，那么在这个子树内该两点距离肯定是要小于等于S，那这种情况在我们分治求每颗子树的时候毫无疑问已经被算过了，因此需要减去，避免重复计算。

示例代码：

```
const int N = 1e4+10,M = N*2;

int n,m;
int h[N],e[M],w[M],ne[M],idx;
bool st[N];
int p[N],q[N];

void add(int a,int b,int c)
{
    e[idx] = b;
    w[idx] =  c;
    ne[idx] = h[a];
    h[a] = idx;
    idx++;
}

int get_size(int u,int fa) // 求出来每颗子树的大小，用于求每颗子树的重心
{
    if(st[u])
        return 0;
    int res = 1;
    for(int i=h[u];i!=-1;i=ne[i]){
        int j=e[i];
        if(j == fa)
            continue;
        res += get_size(j,u);
    }
    return res;
```

```
}

int get_wc(int u,int fa,int tot,int& wc) // 求子树的大小，将重心存在wc中
{
    if(st[u])
        return 0;
    int sum = 1,ms = 0;
    for(int i=h[u];i!=-1;i=ne[i]){
        int j=e[i];
        if(j == fa)
            continue;
        int t=get_wc(j,u,tot,wc);
        ms = max(ms,t);
        sum += t;
    }
    ms = max(ms,tot-sum);
    if(ms <= tot/2) // 其实并不需要真正求得重心，只需要保证分解后其最大子树的大小是小于等于n/2即可，这样可以保证时间复杂度。
        wc = u;
    return sum;
}

void get_dist(int u,int fa,int dist,int &qt) // 获得子树到重心的距离
{
    if(st[u])
        return ;
    q[qt++] = dist;
    for(int i=h[u];i!=-1;i=ne[i]){
        int j=e[i];
        if(j == fa)
            continue;
        get_dist(j,u,dist+w[i],qt);
    }
}

int get(int a[],int k) // 在数组中任意选择两个数，满足a+b<=m 排序后采用双指针算法
{
    sort(a,a+k);
    int res = 0;
    for(int i=k-1,j=-1;i>=0;i--){
        while(j+1<i && a[j+1]+a[i]<=m)
            j++;
        j = min(j,i-1);
        res += j+1;
    }
    return res;
}

int calc(int u) // 计算以u为根的子树中答案是多少
{
    if(st[u])
        return 0; // 如果已经被删除过了就返回0
    int res = 0;
    get_wc(u,-1,get_size(u,-1),u); //获取该子树的重心
    st[u] = true; // 删除重心
    int psize = 0;
    for(int i=h[u];i!=-1;i=ne[i]){
        int j=e[i],qsize=0;
```

```
        get_dist(j,-1,w[i],qsize);
        res -= get(q,qsize); // 减去子树中的相关数量
        for(int k=0;k<qsize;k++){
            if(q[k] <= m) // 单独判断其到子树的距离是否小于等于m
                res++;
            p[psize++] = q[k];
        }
    }
    res += get(p,psize); // 获取总的数量

    for(int i=h[u];i!=-1;i=ne[i])
        res += calc(e[i]);
    return res;
}

int main()
{
    while(scanf("%d%d",&n,&m),n||m){
        memset(h,-1,sizeof h);
        memset(st,false,sizeof st);
        idx = 0;
        rep(i,1,n-1){
            int a,b,c;
            scanf("%d%d%d",&a,&b,&c);
            add(a,b,c);
            add(b,a,c);
        }
        printf("%d\n",calc(0));
    }

    return 0;
}
```

边分

动态点分治（点分树）

cdq分治

主要是用来解决三维偏序问题，统计具有三维属性 (a,b,c) 的、满足一定的比较关系 data

有多少对，优化一些数据结构，算法。

首先我们对原数按照三关键字进行排序，这样就能保证 第一维的关系满足了，然后我们采用分治的思想处理第二维，对于在两个子区间内部的分治即可，一个点在子区间而另一个在另一个区间的采用双指针算法，而第三维则是在采用双指针算法的时候用树状数组来维护，这样就能处理了

需要注意的小细节：排完序我们需要将原数组进行去重的操作。

时间复杂度：$O(nlog(n))$

模板代码：

```
const int N = 2e5+10;

int n,m;
int ans[N],tr[N];
```

```
struct node{
    int a,b,c,s,res;
    bool operator<(const node&t) const{
        if(a != t.a)
            return a<t.a;
        if(b != t.b)
            return b<t.b;
        return c<t.c;
    }
    bool operator==(const node &t) const{
        return a==t.a && b == t.b && c == t.c;
    }
}q[N],w[N];

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

void add(int x,int c)
{
    for(int i=x;i<N;i+=lowbit(i))
        tr[i] += c;
}

int query(int x)
{
    int ans = 0;
    for(int i=x;i;i-=lowbit(i))
        ans += tr[i];
    return ans;
}

void merge_sort(int l,int r)
{
    if(l>=r)
        return ;
    int mid=l+r>>1;
    merge_sort(l,mid);
    merge_sort(mid+1,r);
    int i=l,j=mid+1,k=0;
    while(i<=mid && j<=r){
        if(q[i].b <= q[j].b){
            add(q[i].c,q[i].s);
            w[k++] = q[i++];
        }
        else{
            q[j].res += query(q[j].c);
            w[k++] = q[j++];
        }
    }

    while(i<=mid)
        add(q[i].c,q[i].s),w[k++] = q[i++];
    while(j<=r)
        q[j].res += query(q[j].c),w[k++] = q[j++];
    for(i=l;i<=mid;i++)
        add(q[i].c,-q[i].s);
```

```
    for(i=l,j=0;i<=r;i++,j++)
        q[i] = w[j];
}

int main()
{
    IOS;
    cin >> n >> m;
    rep(i,1,n){
        int a,b,c;
        cin >> a >> b >> c;
        q[i] = {a,b,c,1};
    }
    sort(q+1,q+n+1);

    int k=2;
    for(int i=2;i<=n;i++){
        if(q[i] == q[k-1])
            q[k-1].s++;
        else
            q[k++] = q[i];
    }
    k--;
    merge_sort(1,k);
    rep(i,1,k)
        ans[q[i].res+q[i].s-1] += q[i].s;
    rep(i,0,n-1)
        cout << ans[i] << endl;

    return 0;
}
```

动态逆序对

给定原序列，每次删除一个数然后求其逆序对。

简单思路，把操作的时间戳看成第三维然后进行分治。

模板代码：

```
const int N = 1e5+10;

int n,m;
struct node{
    int a,t,res;
}q[N],w[N];

int tr[N],pos[N];
ll ans[N];

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

void add(int x,int c)
{
```

```
    for(int i=x;i<N;i+=lowbit(i))
        tr[i] += c;
}

int query(int x)
{
    int res=0;
    for(int i=x;i;i-=lowbit(i))
        res += tr[i];
    return res;
}

void merge_sort(int l,int r)
{
    if(l>=r)
        return ;
    int mid=l+r>>1;
    merge_sort(l,mid),merge_sort(mid+1,r);
    int i=mid,j=r;
    while(i>=l && j>mid){
        if(q[i].a > q[j].a)
            add(q[i].t,1),i--;
        else
            q[j].res += query(q[j].t-1),j--;
    }
    while(j>mid)
        q[j].res += query(q[j].t-1),j--;
    for(int k=mid;k>i;k--)
        add(q[k].t,-1);

    i=l,j=mid+1;
    while(i<=mid && j<=r){
        if(q[i].a > q[j].a)
            add(q[j].t,1),j++;
        else
            q[i].res += query(q[i].t-1),i++;
    }
    while(i <= mid)
        q[i].res += query(q[i].t - 1),i++;
    for(int k = mid + 1; k < j; k++)
        add(q[k].t, -1);

    i=l,j=mid+1;
    int k=0;
    while(i<=mid && j<=r){
        if(q[i].a < q[j].a)
            w[k++] = q[i++];
        else
            w[k++] = q[j++];
    }
    while(i<=mid)
        w[k++] = q[i++];
    while(j<=r)
        w[k++] = q[j++];
    for(i=l,j=0;i<=r;i++,j++)
        q[i] = w[j];
}
```

```cpp
int main()
{
    IOS;
    cin >> n >> m;
    rep(i,1,n){
        cin >> q[i].a;
        pos[q[i].a] = i;
    }
    for(int i=1,j=n;i<=m;i++){
        int a;
        cin >> a;
        q[pos[a]].t = j--;
    }

    for(int i=1,j=n-m;i<=n;i++){
        if(q[i].t)
            continue;
        q[i].t = j--;
    }
    merge_sort(1,n);
    rep(i,1,n)
        ans[q[i].t] = q[i].res;
    for(int i=2;i<=n;i++)
        ans[i] += ans[i-1];
    for(int i=n;i>n-m;i--)
        cout << ans[i] << endl;

    return 0;
}
```

随机

爬山

模拟退火

遗传算法

对拍

常数优化

读入优化

关闭同步/解除绑定

`std::ios::sync_with_stdio(false)`

这个函数是一个「是否兼容 $stdio$」的开关，C++ 为了兼容 C，保证程序在使用了 `printf` 和 `std::cout` 的时候不发生混乱，将输出流绑到了一起。同步的输出流是线程安全的。

这其实是 C++ 为了兼容而采取的保守措施，也是使 `cin` / `cout` 速度较慢的主要原因。我们可以在进行 IO 操作之前将 $stdio$ 解除绑定，但是在这样做之后要注意不能同时使用 `std::cin` 和 `scanf`，也不能同时使用 `std::cout` 和 `printf`，但是可以同时使用 `std::cin` 和 `printf`，也可以同时使用 `scanf` 和 `std::cout`。

`tie`

tie 是将两个 stream 绑定的函数，空参数的话返回当前的输出流指针。

在默认的情况下 `std::cin` 绑定的是 `std::cout`，每次执行 `<<` 操作符的时候都要调用 `flush()` 来清理 stream buffer，这样会增加 IO 负担。可以通过 `std::cin.tie(0)`（0 表示 NULL）来解除 `std::cin` 与 `std::cout` 的绑定，进一步加快执行效率。

但需要注意的是，在解除了 `std::cin` 和 `std::cout` 的绑定后，程序中必须手动 `flush` 才能确保每次 `std::cout` 展现的内容可以在 `std::cin` 前出现。这是因为 `std::cout` 被 buffer 为默认设置。

代码实现：

```
+std::ios::sync_with_stdio(false);
std::cin.tie(0);
// 如果编译开启了 C++11 或更高版本，建议使用 std::cin.tie(nullptr);
```

常用的read和write函数(只适用于整数)

输入优化

`scanf` 和 `printf` 依然有优化的空间，这就是本章所介绍的内容——读入和输出优化。

- 注意，本页面中介绍的读入和输出优化均针对整型数据，若要支持其他类型的数据（如浮点数），可自行按照本页面介绍的优化原理来编写代码。

众所周知，`getchar` 是用来读入 1 byte 的数据并将其转换为 `char` 类型的函数，且速度很快，故可以用「读入字符——转换为整型」来代替缓慢的读入

每个整数由两部分组成——符号和数字

整数的 '+' 通常是省略的，且不会对后面数字所代表的值产生影响，而 '-' 不可省略，因此要进行判定

10 进制整数中是不含空格或除 0~9 和正负号外的其他字符的，因此在读入不应存在于整数中的字符（通常为空格）时，就可以判定已经读入结束

C 和 C++ 语言分别在 $ctype.h$ 和 $cctype$ 头文件中，提供了函数 `isdigit`，这个函数会检查传入的参数是否为十进制数字字符，是则返回 **true**，否则返回 **false**。对应的，在下面的代码中，可以使用 `isdigit(ch)` 代替 `ch >= '0' && ch <= '9'`，也可以使用 `!isdigit(ch)` 代替 `ch <'0' || ch> '9'`

代码实现：

```
int read() {
  int x = 0, w = 1;
  char ch = 0;
  while (ch < '0' || ch > '9') {  // ch 不是数字时
    if (ch == '-') w = -1;        // 判断是否为负
    ch = getchar();               // 继续读入
  }
  while (ch >= '0' && ch <= '9') {  // ch 是数字时
    x = x * 10 + (ch - '0');  // 将新读入的数字「加」在 x 的后面
```

```
    // x 是 int 类型，char 类型的 ch 和 '0' 会被自动转为其对应的
    // ASCII 码，相当于将 ch 转化为对应数字
    // 此处也可以使用 (x<<3)+(x<<1) 的写法来代替 x*10
    ch = getchar();  // 继续读入
  }
  return x * w;  // 数字 * 正负号 = 实际数值
}

/*
下方是删除了注释相关的东西，可以直接使用
*/

template <class T> inline T read()
{
    T x=0,w=1;char ch=0;
    while(ch<'0' || ch>'9'){ if (ch=='-') {w=-1;} ch=getchar();}
    while(ch>='0' && ch<='9'){ x=(x<<1)+(x<<3)+(ch^48);ch=getchar();}
    return x*w;
}

template <class T> inline void write(T x)
{
    if(x<0){ x=-x; putchar('-');}
    if(x>9) write(x/10);
    putchar(x%10+'0');
}

举例：
   num = read();
```

输出优化

原理

同样是众所周知，`putchar` 是用来输出单个字符的函数

因此将数字的每一位转化为字符输出以加速

要注意的是，负号要单独判断输出，并且每次 %（mod）取出的是数字末位，因此要倒序输出

代码实现：

```
void write(int x) {
  if (x < 0) {  // 判负 + 输出负号 + 变原数为正数
    x = -x;
    putchar('-');
  }
  if (x > 9) write(x / 10);  // 递归，将除最后一位外的其他部分放到递归中输出
  putchar(x % 10 + '0');  // 已经输出（递归）完 x 末位前的所有数字，输出末位
}
```

但是递归实现常数是较大的，我们可以写一个栈来实现这个过程

```
inline void write(int x) {
  static int sta[35];
  int top = 0;
  do {
```

```
    sta[top++] = x % 10, x /= 10;
  } while (x);
  while (top) putchar(sta[--top] + 48);  // 48 是 '0'
}

/*
下方是注释过的可以直接使用
*/
inline void write(int x){
  static int sta[35];
  int top=0;
  do {
    sta[top++]=x%10;
    x/=10;
  }while(x);
  while(top)
    putchar(sta[--top]+48);
}
```

#