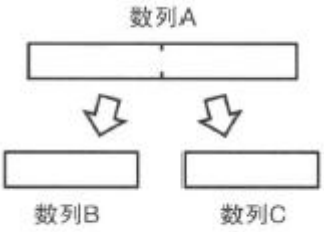
分治法

一、问题引入

求数列的逆序对（即$i<j$且$a_i>a_j$的数对(i,j)）数是一个经典问题，它可以使用树状数组等数据结构求得。而除了使用数据结构外，这个问题也可以在归并排序的基础上统计额外信息来求得，而实际上这个利用归并排序的方法也是一种分治法的体现。

二、算法流程



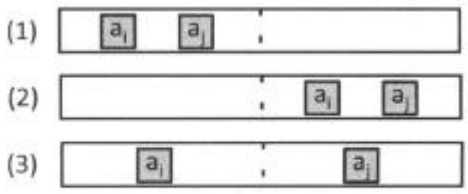
数列的划分

假设我们现在要统计数列 A 中逆序对的个数。如图所示，我们可以将数列 A 平均分成两半得到数列 B 和数列 C，于是，数列 A 中的逆序对(i,j)必然是下面三种之一：

(1) i,j 都属于数列 B 的逆序对(i,j)

(2) i,j 都属于数列 C 的逆序对(i,j)

(3) i 属于数列 B 而 j 属于数列 C 的逆序对(i,j)



三种逆序对

因此，我们只要分别统计这三种逆序对，再把结果加起来就好了。对于(1)和(2)，它们的总数即为数列 B 与数列 C 的逆序对总数，这是一个子问题，我们递归就可以解决。而对于(3)，我们可以对数列 C 中的每个数字x，统计在数列 B 中比x大的数字的个数，再把所有的这样的个数加起来，得到的结果就是(3)的逆序对总数。这个步骤可以像下面这样在归并排序的同时进行统计而完成。

```
1.  // 统计数列 A 的区间[l,r)内的逆序对数
```

```
long long merge_count(int l, int r) {
  if (l + 1 == r) return 0;
  int mid = l + r >> 1;
  long long cnt = 0;
  // 递归到两个子数列求解
  cnt += merge_count(l, mid);
  cnt += merge_count(mid, r);

  static int b[N], c[N];
  int b_n = 0, c_n = 0;
  for (int i = l; i < mid; ++i) b[b_n++] = a[i];
  for (int i = mid; i < r; ++i) c[c_n++] = a[i];

  int bi = 0, ci = 0;
  for (int i = l; i < r; ++i) {
    if (bi < b_n && (ci == c_n || b[bi] <= c[ci])) {
      a[i] = b[bi++];
    } else {
      // 此时有 b[bi]>c[ci]，由于 b 中[0,b_n)这个区间内的数已经有序，因此[bi,b_n)中的所有数均比 c[ci]来得大，也就是我们新找到了 b_n-bi 个逆序对。
      a[i] = c[ci++];
      cnt += b_n - bi;
    }
  }
  return cnt;
}
```

每次递归数列长度都会减半，所以递归深度为 $O(\log n)$，而每一层总的操作都是 $O(n)$，所以总的时间复杂度为 $O(n\log n)$。

在这类问题中，我们把问题分割成更小的子问题递归求解，再处理不同子问题之间的部分，这种算法设计方法就是分治法。

三、主定理

在分治法中，我们常常会将问题分成若干个规模大小一样的子问题并递归求解，之后再利用一定的时间进行合并处理，而类似这样的递归算法在分析复杂度时，我们有专门的结论可以使用，而这个结论称之为主定理。

主定理可以简单描述为：

若规模为 n 的问题通过分治，得到 a 个规模为 $\frac{n}{b}$ 的子问题，每次递归带来的额外计算（也

就是除了子问题之外进行的计算工作）量为 cn^d，则处理规模为 n 的问题的时间 $T(n)$ 有：

$$T(n) = aT(\frac{n}{b}) + cn^d$$

这个递归式的解为：

(1) 若 $a = b^d$ 则 $T(n) = O(cn^d \log n)$；

(2) 若 $a < b^d$ 则 $T(n) = O(cn^d)$；

(3) 若 $a > b^d$ 则 $T(n) = O(cn^{\log_b a})$.

像上述的分治算法，它的递归式为 $T(n) = 2T(\frac{n}{2}) + n$，那么其 $a = 2, b = 2, d = 1$，它符合(1)，因此 $T(n) = O(n^d \log n) = O(n \log n)$.

又例如 $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$，它的 $a = 2, b = 4, d = \frac{1}{2}$，也满足(1)，因此

$$T(n) = O(n^d \log n) = O(\sqrt{n} \log n).$$

四、例题

Ultra-QuickSort (POJ 2299)

题目大意：

求给定数列的逆序对数。

分析：

模板题。

代码：

```
#include <cstdio>

inline int read() {
  static char ch;
  while ((ch = getchar()) < '0' || ch > '9');
  int res = ch - 48;
  while ((ch = getchar()) >= '0' && ch <= '9')
    res = res * 10 + ch - 48;
  return res;
}

const int N = 5e5 + 3;
```

```
int n, a[N];

long long merge_count(int l, int r) {
  if (l + 1 == r) return 0;
  int mid = l + r >> 1;
  long long cnt = 0;
  cnt += merge_count(l, mid);
  cnt += merge_count(mid, r);

  static int b[N], c[N];
  int b_n = 0, c_n = 0;
  for (int i = l; i < mid; ++i) b[b_n++] = a[i];
  for (int i = mid; i < r; ++i) c[c_n++] = a[i];

  int bi = 0, ci = 0;
  for (int i = l; i < r; ++i) {
    if (bi < b_n && (ci == c_n || b[bi] <= c[ci])) {
      a[i] = b[bi++];
    } else {
      a[i] = c[ci++];
      cnt += b_n - bi;
    }
  }
  return cnt;
}

int main() {
  while (scanf("%d", &n), n > 0) {
    for (int i = 0; i < n; ++i) a[i] = read();
    printf("%lld\n", merge_count(0, n));
  }
  return 0;
}
```

五、平面上的分治法

问题引入

上面提到的是序列上的分治法，有时我们对于平面上的问题，还可以对平面进行分治。

仍然考虑一个经典问题：给定平面上的 n 个点，求距离最近的两个点的距离。 n <= 10^5.

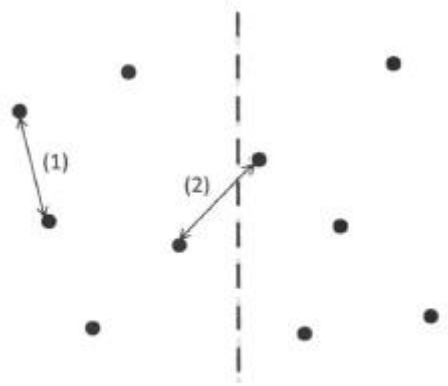
解决这个问题所运用的就是平面上的分治法。

算法流程

假设我们把所有点按 x 坐标平均分成了左右两个部分，那么最近点对 (p,q) 的距离就是下面二者的最小值：

(1) 两点 p,q 同属于左半边或者右半边时的最近点对距离

(2) 两点 p,q 属于不同区域时的最近点对距离



两种点对

对于(1)我们可以递归计算。而对于(2)我们没有很好的办法能直接处理，但如果我们已经知道了(1)部分的最小值，不妨记为 d ，那么我们再考虑下面的(2')：

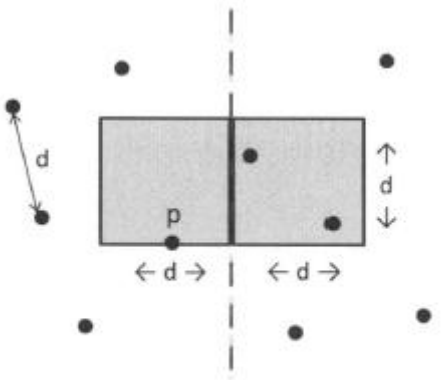
(2') 两点 p,q 属于不同区域时，距离小于 d 的点对 (p,q) 的最小距离。

由于我们已经求解出了(1)，所以我们可以对(2)加上距离小于 d 的限制，若不能满足这个限制(2)就是无用的。（我们已经找到了距离为 d 的点对，不需要再考虑距离大等于 d 的点对，它们肯定不能更新答案），所以加上这个限制后(2')仍然能保证答案的正确性。

有了这个条件，我们所要考虑的点对数目就减小了。首先我们考虑 x 坐标。假设将点划分为左右两半的直线为 l，其 x 坐标为 x_0，那么根据(2')，到直线 l 的距离大等于 d 的点就没有必要考虑了。因此我们只要考虑 x 坐标满足 $x_0-d<x<x_0+d$ 的点。

接下来我们考虑 y 坐标。对于每个点，我们都只需考虑与那些 y 坐标不比自己大的点所组成的点对。另外，也没有必要考虑那些 y 坐标相差大等于 d 的点。也就是说对于 y 坐标为 y_p 的点，我们只需考虑与 y 坐标满足 $y_p-d<y\le y_p$ 的点所组成的点对就足够了。

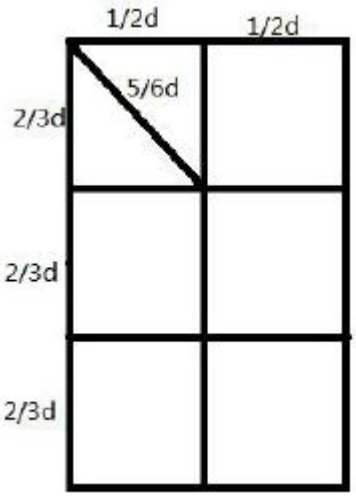
综合以上两点可知，我们要检查的点都在 $x_0-d<x<x_0+d$ 且 $y_p-d<y\le y_p$ 的矩形区域内。实际上这个矩形内的点的个数最多只有 6 个。如下图所示：



对于点p所需考虑的点的范围

因为 d 是(1)中的最小值，所以同属左半边或同属右半边的点其距离均不小于 d 。因此把待考虑矩形分成左右两个正方形后，每个正方形内至多只包含三个点。因此该矩形区域内至多只含有 6 个点。

从下图中也能看出，一个小矩形内最多只有一个点，而这样的小矩形有 6 个，因此需要考虑的点不超过 6 个。



为了实现按 x 坐标划分，我们可以在开始时先将所有点按 x 坐标排序。另一方面，为了更高效地检查对应矩形内的所有点，在处理(2')之前，我们可以先把所有要考虑的点按 y 坐标排序。而这步可以通过在递归的同时做 y 坐标的归并排序来实现。

每次递归会将点集平均分为两部分，因此递归的深度是 $O(\log n)$ ，而每层有 $O(n)$ 的操作，因此总复杂度为 $O(n\log n)$ 。

六、例题

Quoit Design (HDU 1007)

题目大意：

给定平面上的 n 个点，求距离最近的两个点的距离的一半。 n <= 10^5.

分析：

模板题。

代码：

```
#include <cmath>
#include <cstdio>
#include <algorithm>

void chkmin(double &x, const double &y) { if (x > y) x = y; }

const int N = 1e5 + 3;
int n;
struct point {
  double x, y;
  point() {}
  point(const double &_x, const double &_y) :
    x(_x), y(_y) {}
  friend inline point operator - (const point &lhs, const point &rhs) {
    return point(lhs.x - rhs.x, lhs.y - rhs.y);
  }
  inline double norm() const {
    return sqrt(x * x + y * y);
  }
  inline bool operator < (const point &rhs) const {
    return x < rhs.x;
  }
} p[N];

inline double solve(int l, int r) {
  if (l + 1 == r) return 1e20;
  int mid = l + r >> 1;
  double x0 = (p[mid - 1].x + p[mid].x) / 2.0;
  double d = std::min(solve(l, mid), solve(mid, r));

  static point a[N], b[N], c[N];
  int b_n = 0, c_n = 0;
  int L = l, R = mid;
  for (int i = l; i < r; ++i) {
    if (L < mid && (R == r || p[L].y < p[R].y)) {
```

```
      a[i] = p[L++];
      if (x0 - d < a[i].x) b[b_n++] = a[i];
    } else {
      a[i] = p[R++];
      if (a[i].x < x0 + d) c[c_n++] = a[i];
    }
  }
  for (int i = l; i < r; ++i) p[i] = a[i];

  // 扫描矩形内的点
  for (int i = 0, j = 0; i < b_n || j < c_n; ) {
    if (i < b_n && (j == c_n || b[i].y < c[j].y)) {
      for (int k = j - 1; k >= 0; --k) {
        if (b[i].y - d >= c[k].y) break;
        chkmin(d, (c[k] - b[i]).norm());
      }
      ++i;
    } else {
      for (int k = i - 1; k >= 0; --k) {
        if (c[j].y - d >= b[k].y) break;
        chkmin(d, (b[k] - c[j]).norm());
      }
      ++j;
    }
  }

  return d;
}

int main() {
  while (scanf("%d", &n), n > 0) {
    for (int i = 0; i < n; ++i) scanf("%lf%lf", &p[i].x, &p[i].y);
    std::sort(p, p + n);
    printf("%.2f\n", solve(0, n) / 2.0);
  }
  return 0;
}
```

练习题

HDU 5696 ; CF 480E

CF 97B ; BZOJ 2458

树分治

一、问题引入

经典问题：给定一棵含有 n 个顶点的树，第 i 条边连接着顶点 u_i 与 v_i，它的长度为 w_i。现在要求统计，最短距离不超过 k 的顶点的对数。

上面我们介绍的两种分治算法，分别应用于序列上与平面上。实际上，我们也能将分治算法用于树结构，这样的分治算法就叫做树分治。

上面所提到的问题求的是顶点对数，而实际上一对顶点就代表着树中的一条路径，而树分治就是用来解决一类与树的路径有关的高效算法。

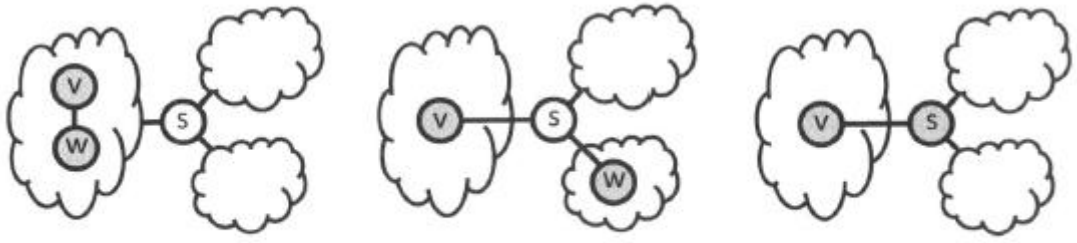
二、算法介绍

回到上面所提到的经典问题，假设我们按重心把树分成了若干子树，那么符合条件的顶点对有以下三种：

(1) 顶点 v,w 属于同一子树的顶点对 (v,w)

(2) 顶点 v,w 属于不同子树的顶点对 (v,w)

(3) 顶点 s 与其他顶点 v 组成的顶点对 (s,v)



三种顶点对

对于情况(1)，它所包含的合法顶点对就是这棵子树中所有合法的顶点对，这就是一个与原问题相同的子问题，我们可以递归求解。

对于情况(2)，从顶点 v 到顶点 w 的路径必然经过了顶点 s。记 d_u 表示结点 u 到 s 的路径长度，那么我们要统计的就是满足 $d_u+d_v\le k$ 且 u,v 来自不同子树的 (u,v) 个数。这个值等于满足 $d_u+d_v\le k$ 的 (u,v) 个数减去满足 $d_u+d_v\le k$ 且 u,v 来自相同子树的 (u,v) 个数。

而这两个部分，都是要求出满足 $d_u + d_v \le k$ 的 (u,v) 个数。将 d 值排序后，我们利用单调性很容易能得出一个 $O(\text{结点个数})$ 的算法。

对于情况(3)，我们将 s 看做是一个到 s 的路径长度为 0 的结点，那么就转化为了情况(2)，将其与情况(2)一起处理即可。

在递归的每一层我们都做了排序，复杂度是 $O(n\log n)$，而递归的深度为 $O(\log n)$（下面我们会证明），所以总的复杂度为 $O(n\log^2 n)$.

三、算法分析

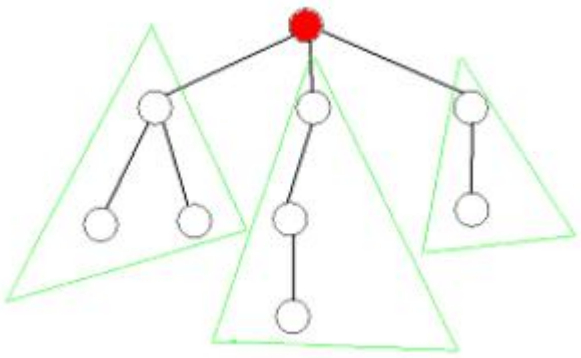
在上面的例子中，顶点对实际上就对应着一条树中的路径，而顶点对的两种情况（情况(3)与(2)的本质是一样的），实际上对应着这么一个关系：一条路径它要么过重心 s，要么在 s 的一棵子树中。这也就是树分治处理树上路径问题的核心：处理过重心的路径，而在子树中的路径则递归即可。

有时在处理过重心的路径时，我们需要合并两棵子树的信息，且合并的复杂度常常为深度较大的那棵子树的深度，因此为了保证复杂度，我们需要将所有的儿子按照其所在子树的深度排序，按顺序合并，这样就能使得合并的复杂度正确。

事实上树分治有两种常用的形式：

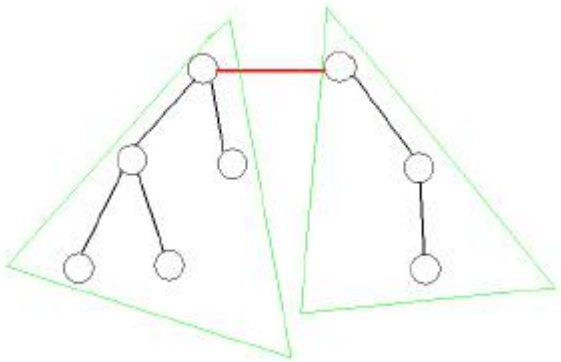
基于点的分治：

首先选取一个点将无根树转化为有根树，处理过根结点的路径，再递归处理每一棵以根结点某个儿子为根的子树。这种树分治我们通常称它为点分治。



基于边的分治：

在树中选取一条边，将原树分成两棵不相交的树，处理过这条边的路径，再递归处理两棵子树。这种树分治我们通常称它为边分治。



在点分治中，我们选取的点通常设为这棵树的重心（在树中将其删去后，结点最多的子树的结点个数最小），因为它的每个子树的大小都不会超过原树的一半（可以根据重心的性质得到），因此每次递归，结点个数都会减半，因此我们的递归深度不超过 $O(\log n)$ 。

在漆子超的论文中有关于选取点部分复杂度的证明介绍：

定理：存在一个点使得分出的子树的结点个数均不大于 $n/2$

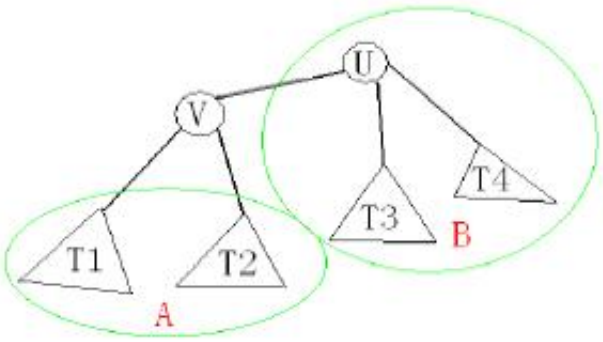
证明：假设 U 是树的重心，它与 $V_1,V_2,\cdots,V_K$ 相邻，记 $size(x)$ 表示以 x 为根的子树的结点个数。记 V 为 $V_1,V_2,\cdots,V_K$ 中 $size$ 值最大的点。

我们采取反证法，即假设 $size(V)>n/2$ ，那么我们考虑如果选取 V 作为根结点的情况，记 $size'(x)$ 表示此时以 x 为根的子树的结点个数。

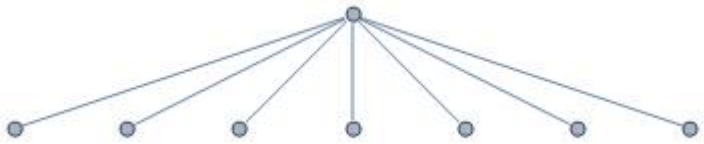
如下图，对于 A 部分，显然 $size'(T_i)<size(V)$.

对于 B 部分， $size'(u)=n-size(V)<n/2<size(V)$ 这与我们假设矛盾。

定理得证。



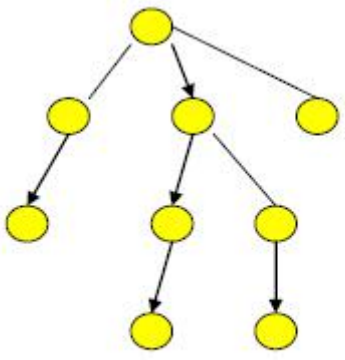
而对于边分治，我们考虑“扫把图”，那么不管怎么选取边进行递归，递归的深度总是会达到$O(n)$，每次的子问题规模也只会减小 1，因此若要使得边分治的复杂度正确，需要对原图进行特殊的转化处理，这个处理过程比较麻烦，因此本文就不予介绍。在实际应用时，我们也常常只考虑使用点分治。



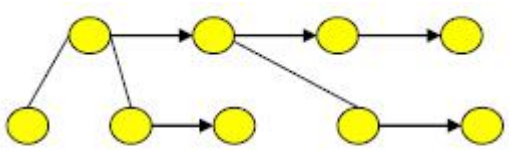
四、树链剖分与树分治的关系

由于树链剖分与树的分治均可以解决树中路径的问题，我们不禁疑问，它们是否有着千丝万缕的联系呢?答案是肯定的。

我们首先画出一棵树及其剖分（如图 a），我们似乎看不出有什么特殊，我们不妨把它的样子稍加改变（如图 b），按照点到根结点路径上的轻边个数分层摆放。



图(a)



图(b)

这时我们就能明显的看出树链剖分与树的分治两种算法其实有着惊人的相似，我们上一节中介绍了分治的两种方式：删除一个点，或一条边，而树链剖分相当于每次删除了一条链！所以树链剖分算法可以看做是基于链的分治，而且这种分治还有一个特点，每次被删除结点的儿子必将作为下一次删除的链的头结点。这样，就给我们的维护带来了方便。而每个点到根结点路径上的轻边个数是 $O(\log n)$ 的，这样算法的时间复杂度便有了保证。

五、例题

Tree (POJ 1741)

题目大意：

给定一棵含有 n 个顶点的树，第 i 条边连接着顶点 u_i 与 v_i，它的长度为 w_i。现在要求统计，最短距离不超过 k 的顶点的对数。 $n \leq 10000$

分析：

树分治模板题，根据之前的分析实现即可。

代码：

```
#include <cstdio>
#include <algorithm>

const int N = 1e4 + 5;
int n, K, fa[N], sze[N], son[N], dis[N];
int ecnt, adj[N], nxt[N * 2], go[N * 2], len[N * 2];
long long ans;
bool vis[N];

inline void addEdge(int u, int v, int w) {
  nxt[++ecnt] = adj[u], adj[u] = ecnt, go[ecnt] = v, len[ecnt] = w;
  nxt[++ecnt] = adj[v], adj[v] = ecnt, go[ecnt] = u, len[ecnt] = w;
}

void init() {
```

```
  ans = ecnt = 0;
  for (int i = 1; i <= n; ++i) adj[i] = vis[i] = 0;
}

int calcG(int sv) {
  static int qn, que[N];
  int u, v, e;
  int mx = n, G;
  que[qn = 1] = sv, fa[sv] = 0;
  for (int ql = 1; ql <= qn; ++ql) {
    sze[u = que[ql]] = 1, son[u] = 0;
    for (e = adj[u]; e; e = nxt[e]) {
      if (vis[v = go[e]] || v == fa[u]) continue;
      fa[v] = u, que[++qn] = v;
    }
  }
  for (int ql = qn; ql >= 1; --ql) {
    u = que[ql], v = fa[u];
    if (qn - sze[u] > son[u]) son[u] = qn - sze[u];
    if (son[u] < mx) G = u, mx = son[u];
    if (!v) break;
    sze[v] += sze[u];
    if (sze[u] > son[v]) son[v] = sze[u];
  }
  return G;
}

inline long long calc(int sv, int L) {
  static int qn, que[N], d[N];
  int u, v, e, d_n = 0;
  que[qn = 1] = sv, dis[sv] = L, fa[sv] = 0;
  for (int ql = 1; ql <= qn; ++ql) {
    d[d_n++] = dis[u = que[ql]];
    for (e = adj[u]; e; e = nxt[e]) {
      if (vis[v = go[e]] || v == fa[u]) continue;
      fa[v] = u, dis[v] = dis[u] + len[e], que[++qn] = v;
    }
  }
  long long cnt = 0;
  std::sort(d, d + d_n);
  int l = 0, r = d_n - 1;
  while (l < r) {
    if (d[l] + d[r] <= K) cnt += r - l++;
    else --r;
```

```
  }
  return cnt;
}

void solve(int u) {
  int G = calcG(u);
  vis[G] = true;
  ans += calc(G, 0);
  for (int e = adj[G]; e; e = nxt[e])
    if (!vis[go[e]]) ans -= calc(go[e], len[e]);
  for (int e = adj[G]; e; e = nxt[e])
    if (!vis[go[e]]) solve(go[e]);
}

int main() {
  while (scanf("%d%d", &n, &K), n || K) {
    init();
    for (int i = 1; i < n; ++i) {
      int u, v, w;
      scanf("%d%d%d", &u, &v, &w);
      addEdge(u, v, w);
    }
    solve(1);
    printf("%lld\n", ans);
  }
  return 0;
}
```

练习题

SPOJ FTOUR2 ; BZOJ 2152 ; BZOJ 1758

CDQ 分治

一、简介

CDQ 分治是一种特殊的分治方法，在 OI 界初见于陈丹琦 2008 年的集训队作业中，因此被称为 CDQ 分治。

CDQ 分治通常用来解决一类“修改独立，允许离线”的数据结构题。实际上它的本质是按时间分治，即若要处理时间$[l,r]$上的修改与询问操作，就先处理$[l,mid]$上的修改对$[mid+1,r]$上的询问的影响，之后再递归处理$[l,mid]$与$[mid+1,r]$，根据问题的不同，这几个步骤的顺序有时也会不一样。

CDQ 分治会使得我们考虑的问题的思维难度与代码难度大大减小，通常利用 CDQ 分治能使得一个树套树实现的题目，能够去掉外层的树，改为用分治来进行求解。

二、算法描述

首先，在数据结构题中，我们称要求我们作出回答的操作为“询问”，称会对询问操作的答案造成影响的操作为“修改”。（有时一个操作既是询问也是修改）

CDQ 分治适用于满足以下两个条件的数据结构题：

(1) 修改操作对询问的贡献独立，修改操作之间互不影响效果。

(2) 题目允许使用离线算法。

我们不妨假设我们需要（按顺序）完成的操作序列为S，考虑将整个操作序列等分为前后两个部分，那么现在我们可以发现以下两个性质：

(1) 显然，后一半操作序列中的修改操作对前一半操作序列中的询问的结果不会产生任何影响。

(2) 后一半操作序列中的询问操作只受两方面影响：一是前一半操作序列中的所有修改操作；二是后一半操作序列中，在该询问操作之前的修改操作。

容易发现，因为后一半操作序列的修改操作完全不会影响前一半操作序列中的询问结果，因此前一半操作序列的查询实际是与后一半操作序列完全独立的，是与原问题完全相同的子问题，可以递归处理。

接下来我们来考虑后一半操作序列中的询问操作。我们发现，影响后一半操作序列询问的答案的因素中，第二部分“后一半操作序列中，在该询问操作之前的修改操作”也是与前一半序列完全无关的（因为我们前面已经假定题目中的修改操作互相独立互不影响，而询问操作更不会影响修改操作了）。因此，这部分因素也是与原问题完全相同的完全独立的子问题，也可以递归处理。

我们还可以发现，影响后一半操作序列询问答案的因素的第一部分“前一半操作序列中的所有修改操作”虽然与前一半序列密切相关，但它有一个非常好的性质，就是现在后一半操作序列的询问都是在前一半操作序列的所有修改完成之后执行的，那么对于影响后半部分询问的前半部分修改，它们对于任意一个后半部分询问的影响都是一模一样的。这时，原问题的动态修改操作便不再存在了，而被转化为了离线的、与原问题同样规模的“一开始给出所有修改”然后“回答若干询问”的更简单的问题，从而能化简算法。

不妨设“解决无动态修改操作的原问题”的复杂度为$O(f(n))$，那么由主定理，我们知道这样分治的总时间复杂度将是$O(f(n)\log n)$。

因此，只要数据结构题满足我们上文假定的两个要求：修改独立，允许离线，就可以以一个$\log$的代价，将原问题中的动态修改去掉，变为没有动态修改的简化版问题，极大简化我们的思维与代码难度。

这种对时间分治的方法要求操作之间是互相独立的，因此如果有形如“撤销某次操作”这样的操作，就不能直接按上面的方法来解决问题，因为现在的插入操作可能会被后面的删除操作撤销掉，修改操作并不是完全独立的。

我们依然把操作序列等分为前后两半。我们考虑前一半操作中的询问操作，它们的答案显然与后半部分的插入与删除操作无关，因此它们仍然可以直接递归。

考虑后一半操作中的询问操作，它们的答案只与两部分内容有关：

(1) 前一半操作序列中的所有插入操作中在该询问之前未被删除的部分；

(2) 后一半操作序列中，在该询问操作之前的且未被删除的插入操作。

我们发现，因为后一半序列中的插入操作显然不会在前一半序列中被删除，因此“后一半操作序列中，在该询问操作之前的且未被删除的插入操作”这一部分与前一半操作序列完全无关，也是与原问题完全相同的子问题，可以递归求解。

因此现在，我们实际要处理的内容是：

“前一半操作序列中的所有插入操作中未被删除的部分”对“后一半操作序列的询问操作”的贡献。

因此，我们的实际任务是初始时给定一些插入操作，然后现在有个操作序列，每个操作会是删除一个插入操作，或是一个询问操作。

这个问题直接解决还是非常困难，但我们发现现在问题中只有删除没有插入。于是一个经典的解决方法就可以派上用场了，那就是“时光倒流”。

因为我们使用的是离线算法，我们完全可以预知转化后的问题中“初始的插入操作”，“给定的删除操作与询问操作”等所有需要的信息，那么我们能事先求出到最后都没有被删除掉的那些初始的插入操作，接着我们逆序处理操作序列，这样询问操作没有变，而删除操作变为了插入操作。利用时光倒流，我们再次把问题转化为了与之前相同的问题。

现在，只要插入操作贡献独立，插入操作之间互不影响，且题目允许离线，即使有删除或者变更操作（变更操作实际等价于先删除原操作，再插入新操作），我们也可以利用 CDQ 分治与时光倒流，以 2 个 $\log$ 的时间复杂度为代价，把题目简化为没有动态插入，没有动态删除，没有动态变更的完全静态版问题。

三、例题

陌上花开 (BZOJ 3262)

题目大意：

有 n 个元素，每个元素有三种属性 a_i,b_i,c_i，元素 i 比元素 j 要大定义为 i 的三个属性值均大等于 j 对应的属性值。现在对于每个元素求有多少个元素比它小。$n \le 10^5$。

分析：

将所有元素按 a_i 排序，之后按顺序处理每个元素。现在相当于把 (b_i,c_i) 看成二维平面上的坐标，那么每个元素是个询问操作，询问其左下角点的个数。同时询问过后，它又是一个插入操作，在对应的 (b_i,c_i) 的位置插入一个点。

套用上面 CDQ 分治算法的流程即可。

代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

const int N = 1e5 + 3, M = 2e5 + 3;
int n, m, s, head, tail, ans[N], f[M];
struct node {
    int x, y, z, cnt, sum;
    inline bool operator == (const node &b) const {
        return x == b.x && y == b.y && z == b.z;
    }
    inline bool operator < (const node &b) const {
        if (x != b.x) return x < b.x;
        if (y != b.y) return y < b.y;
        return z < b.z;
    }
    inline bool operator > (const node &b) const {
        if (y != b.y) return y < b.y;
        return z <= b.z;
    }
} t[N], a[N];

inline void Init(int x) {
    for (; x <= s; x += x & -x)
        if (f[x] != 0) f[x] = 0;
        else    break;
    return ;
}

inline void Insert(int x, const int &v) {
    for (; x <= s; x += x & -x) f[x] += v;
    return ;
}

inline int Query(int x) {
    int res = 0;
    for (; x; x -= x & -x) res += f[x];
    return res;
}

```

```
inline void CdqSolve(const int &l, const int &r) {
    if (l == r) return ;
    int mid = l + r >> 1, idx1 = l, idx2 = mid + 1;
    CdqSolve(l, mid); CdqSolve(mid + 1, r);
    for (int i = l; i <= r; ++i) {
        if (idx2 > r || idx1 <= mid && a[idx1] > a[idx2]) {
            t[i] = a[idx1++];
            Insert(t[i].z, t[i].cnt);
        }
        else {
            t[i] = a[idx2++];
            t[i].sum += Query(t[i].z);
        }
    }
    for (int i = l; i <= r; ++i) {
        a[i] = t[i]; Init(a[i].z);
    }
    return ;
}

char ch;
inline int read() {
    while (ch = getchar(), ch < '0' || ch > '9');
    int res = ch - 48;
    while (ch = getchar(), ch >= '0' && ch <= '9') res = res * 10 + ch - 48;

    return res;
}

int main() {
    m = read(); s = read();
    for (int i = 1; i <= m; ++i)
        t[i].x = read(), t[i].y = read(), t[i].z = read();
    sort(t + 1, t + m + 1);
    head = 1; n = 0;
    while (head <= m) {
        tail = head + 1;
        while (tail <= m && t[tail] == t[head]) ++tail;
        a[++n] = t[head]; a[n].cnt = tail - head;
        head = tail;
    }
    CdqSolve(1, n);
    for (int i = 1; i <= n; ++i)
        ans[a[i].sum + a[i].cnt - 1] += a[i].cnt;
```

```
    for (int i = 0; i < m; ++i)
        printf("%d\n", ans[i]);
    return 0;
}
```

练习题

BZOJ 1176 ; BZOJ 3295 ; BZOJ 2716

上述内容参考：

1.挑战程序设计竞赛（第二版）

2.2013 年国家集训队论文，浅谈数据结构题的几个非经典解法，许昊然

3.2014 年国家集训队论文，线段树在一类分治问题上的应用