18.9.20 模板

目录

[Tarjan 强连通分量 2](#_Toc525223082)

[Tarjan 双联通分量，桥 5](#_Toc525223083)

[LCA 倍增 8](#_Toc525223084)

[LCA RMQ 11](#_Toc525223085)

[LCA Tarjan离线 14](#_Toc525223086)

[KMP 17](#_Toc525223087)

[AC自动机 19](#_Toc525223088)

[凸包 22](#_Toc525223089)

[旋转卡壳求对踵点 25](#_Toc525223090)

[平板电视红黑树 28](#_Toc525223091)

[笛卡尔树 30](#_Toc525223092)

[归并树求区间第k大 O（n(logn)^2） 33](#_Toc525223093)

[平方分割求区间第k大 O（nlogn + m\*sqrt(n)\*(logn)^2） 36](#_Toc525223094)

[线段树 39](#_Toc525223095)

[欧拉路 42](#_Toc525223096)

[2-sat 47](#_Toc525223097)

[树上点分治 52](#_Toc525223098)

## Tarjan 强连通分量

#include <cstdio>

#include <cstring>

#include <stack>

using namespace std;

const int MAXN = 110;

const int MAXM = MAXN \* MAXN;

struct edge {

int v, nt;

} e[MAXM];

int head[MAXN], low[MAXN], dfn[MAXN], in[MAXN], out[MAXN], color[MAXN];

int cnte, cntc, idx, n;

stack<int> s;

void init()

{

cnte = cntc = idx = 0;

memset(head, 0, sizeof head);

memset(dfn, 0, sizeof dfn);

memset(color, 0, sizeof color);

memset(in, 0, sizeof in);

memset(out, 0, sizeof out);

}

void add (int u, int v)

{

cnte ++;

e[cnte].v = v;

e[cnte].nt = head[u];

head[u] = cnte;

}

void tarjan(int u)

{

dfn[u] = low[u] = ++ idx;

s.push(u);

for (int i = head[u]; i; i = e[i].nt) {

int v = e[i].v;

if (!dfn[v]) {

tarjan(v);

low[u] = min(low[u], low[v]);

} else if (!color[v]) {

low[u] = min(low[u], dfn[v]);

}

}

if (dfn[u] == low[u]) {

++ cntc;

while (true) {

int now = s.top(); s.pop();

color[now] = cntc;

if (now == u) {

break;

}

}

}

}

int main()

{

while (~scanf("%d", &n)) {

init();

for (int i = 1; i <= n; i ++) {

int v;

while (scanf("%d", &v), v) {

add(i, v);

}

}

for (int i = 1; i <= n; i ++) {

if (!dfn[i]) {

tarjan(i);

}

}

for (int u = 1; u <= n; u ++) {

for (int i = head[u]; i; i = e[i].nt) {

int v = e[i].v;

if (color[u] != color[v]) {

in[color[v]] ++;

out[color[u]] ++;

}

}

}

int in0 = 0, out0 = 0;

for (int i = 1; i <= cntc; i ++) {

if (in[i] == 0) ++ in0;

if (out[i] == 0) ++ out0;

}

printf("%d\n", in0);

if (cntc == 1) {

puts("0");

} else {

printf("%d\n", max(in0, out0));

}

}

return 0;

}

## Tarjan 双联通分量，桥

#include <iostream>

#include <cstdio>

#include <algorithm>

#include <cstring>

#include <stack>

#include <queue>

using namespace std;

const int MAX = 100010;

struct edge {

int v, nt, used;

} e[MAX << 2];

int n, m, q, cnte, idx, cntb;

int head[MAX], dfn[MAX], low[MAX];

int isbridge[MAX], father[MAX];

void init()

{

father[1] = cnte = idx = cntb = 0;

memset(head, -1, sizeof head);

memset(dfn, 0, sizeof dfn);

memset(isbridge, 0, sizeof isbridge);

}

void add(int u, int v)

{

e[cnte].v = v;

e[cnte].nt = head[u];

e[cnte].used = 0;

head[u] = cnte;

++ cnte;

}

void tarjan(int u)

{

dfn[u] = low[u] = ++ idx;

for (int i = head[u]; i != -1; i = e[i].nt) {

if (e[i].used)

continue;

e[i].used = 1;

e[i ^ 1].used = 1;

int v = e[i].v;

if (!dfn[v]) {

father[v] = u;

tarjan(v);

if (low[v] > dfn[u]) {

// v无法通过回边或者通过子女到达比u点更靠前的点，

// 那么我们只需要标记v点即可表明割边（桥）

// 桥就是u到v的这条边

isbridge[v] = 1;

++ cntb;

}

low[u] = min(low[u], low[v]);

} else {

low[u] = min(low[u], dfn[v]);

}

}

}

void lca(int u, int v)

{

while (u != v) {

while (dfn[u] > dfn[v]) {

if (isbridge[u]) {

isbridge[u] = 0;

-- cntb;

}

u = father[u];

}

while (dfn[v] > dfn[u]) {

if (isbridge[v]) {

isbridge[v] = 0;

-- cntb;

}

v = father[v];

}

}

//printf("lca : %d\n", u);

}

int main()

{

int tc = 1;

while (~scanf("%d%d", &n, &m), n || m) {

init();

int u, v;

for (int i = 1; i <= m; i ++) {

scanf("%d%d", &u, &v);

add(u, v);

add(v, u);

}

tarjan(1);

scanf("%d", &q);

printf("Case %d:\n", tc ++);

while (q --) {

int u, v;

scanf("%d%d", &u, &v);

lca(u, v);

printf("%d\n", cntb);

}

puts("");

}

return 0;

}

## LCA 倍增

// 倍增求LCA

#include <iostream>

#include <cstdio>

#include <cstring>

#include <algorithm>

#include <queue>

using namespace std;

const int MAX = 1e4 + 10;

// 最大深度对二取对数

const int MAXH = 16;

queue<int> q;

struct edge {

int v, nt;

} e[MAX];

int n, isroot[MAX], head[MAX], cnte, dep[MAX];

// anc[i][j]表示第i个节点向上跳2^j层后的节点

// anc[i][0]就是父亲节点

// 如果跳2^j层后超过了root，那么anc[i][j] = root

int anc[MAX][MAXH];

void init()

{

cnte = 0;

memset(head, 0, sizeof head);

memset(isroot, -1, sizeof isroot);

}

void add(int u, int v)

{

++ cnte;

e[cnte].v = v;

e[cnte].nt = head[u];

head[u] = cnte;

}

// x向上跳h层后的节点编号

int swim(int x, int h)

{

int ret = x;

// 从二进制角度看，如6=110，那么先跳2层，再跳4层

for (int i = 0; h; i ++, h >>= 1) {

if (h & 1) {

ret = anc[ret][i];

}

}

return ret;

}

// 遍历整个树，打出anc表

void bfs(int root)

{

dep[root] = 1;

q.push(root);

for (int i = 0; i < MAXH; i ++) {

anc[root][i] = root;

}

while (!q.empty()) {

int u = q.front(); q.pop();

for (int i = head[u]; i; i = e[i].nt) {

int v = e[i].v;

if (v != anc[u][0]) {

dep[v] = dep[u] + 1;

anc[v][0] = u;

for (int i = 1; i < MAXH; i ++) {

// 倍增

anc[v][i] = anc[anc[v][i - 1]][i - 1];

}

q.push(v);

}

}

}

}

int lca(int x, int y)

{

if (dep[x] < dep[y]) {

swap(x, y);

}

// 先把较深的跳到较浅的同一高度

x = swim(x, dep[x] - dep[y]);

if (x == y) {

return x;

}

// 后一次跳的高度一定比前一次跳的高度还小

// 可以用反证法证明

for (int i = MAXH - 1; i >= 0; i --) {

if (anc[x][i] != anc[y][i]) {

x = anc[x][i];

y = anc[y][i];

}

}

// 循环结束后，anc[x][0] = anc[y][0] = lca

return anc[x][0];

}

int main()

{

int t;

scanf("%d", &t);

while (t --) {

init();

scanf("%d", &n);

int u, v;

for (int i = 1; i < n; i ++) {

scanf("%d%d", &u, &v);

add(u, v);

isroot[v] = 0;

}

int root = -1;

for (int i = 1; i <= n; i ++) {

if (isroot[i]) {

root = i;

break;

}

}

bfs(root);

scanf("%d%d", &u, &v);

printf("%d\n", lca(u, v));

}

return 0;

}

## LCA RMQ

/\*\*

\* 通过记录dfs序，找到lca

\* 思想是找u和v的lca，手动模拟dfs可以发现，lca一定在dfs的路线上

\* 而在这个区间中，lca一定是深度最小的那个

\* 因此，可以维护区间最小值，O(log n)处理一个请求

\* 预处理的时间是O(n log n)的

\*/

#include <iostream>

#include <algorithm>

#include <cstdio>

#include <cstring>

using namespace std;

const int MAX = 1e4 + 10;

struct edge {

int v, nt;

} e[MAX];

int n, isroot[MAX], head[MAX], cnte;

// in记录dfs过程中第一次找到该节点时的时间戳

// order记录每个时间戳对应的是哪个节点

// dep记录每个时间戳所对应的节点的深度

// 注意，时间戳的最大值是节点个数的2倍减1

int in[MAX], order[MAX << 1], dep[MAX << 1], idx;

// st表，维护dep区间最小值，最多有时间戳个数的dep

int st[MAX << 1][20], id[MAX << 1][20];

void init()

{

cnte = 0;

memset(head, 0, sizeof head);

memset(isroot, -1, sizeof isroot);

}

void add(int u, int v)

{

++ cnte;

e[cnte].v = v;

e[cnte].nt = head[u];

head[u] = cnte;

}

void dfs(int u, int d)

{

in[u] = ++ idx;

order[idx] = u;

dep[idx] = d;

for (int i = head[u]; i; i = e[i].nt) {

int v = e[i].v;

dfs(v, d + 1);

++ idx;

order[idx] = u;

dep[idx] = d;

}

}

void build()

{

for (int i = 1; i <= idx; i ++) {

st[i][0] = dep[i];

id[i][0] = order[i];

}

for (int j = 1; j < 20; j ++) {

for (int i = 1; i + (1 << j) <= idx + 1; i ++) {

int halfloc = i + (1 << (j - 1));

if (st[i][j - 1] < st[halfloc][j - 1]) {

st[i][j] = st[i][j - 1];

id[i][j] = id[i][j - 1];

} else {

st[i][j] = st[halfloc][j - 1];

id[i][j] = id[halfloc][j - 1];

}

}

}

}

// 返回的是区间dep最小的节点编号

int query(int l, int r)

{

int k = 0;

while (r - l + 1 >= (1 << (k + 1))) {

++ k;

}

if (st[l][k] < st[r - (1 << k) + 1][k]) {

return id[l][k];

} else {

return id[r - (1 << k) + 1][k];

}

}

int lca(int u, int v)

{

return query(min(in[u], in[v]), max(in[u], in[v]));

}

int main()

{

int t;

scanf("%d", &t);

while (t --) {

init();

scanf("%d", &n);

for (int i = 1; i < n; i ++) {

int u, v;

scanf("%d%d", &u, &v);

// 题目保证了u是v的父亲

add(u, v);

isroot[v] = 0;

}

idx = 0;

for (int i = 1; i <= n; i ++) {

if (isroot[i]) {

dfs(i, 1);

break;

}

}

build();

int u, v;

scanf("%d%d", &u, &v);

printf("%d\n", lca(u, v));

}

return 0;

}

## LCA Tarjan离线

// tarjan lca模板

/\*\*

\* 核心思想：如果要求u，v的lca，有一个点a，

\* uv分别在a的左右子树，那么a就是uv的lca

\* 用到了并查集

\*/

#include <bits/stdc++.h>

using namespace std;

const int MAX = 4e4 + 10;

struct edge {

int v, w, nt;

} e[MAX << 1];

// 离线算法，需要存储所有查询

struct Query {

int id, v;

Query(int vv, int idd): id(idd), v(vv) {}

};

int head[MAX], father[MAX], vis[MAX];

// 用来存储询问的节点和答案

int lca[MAX], ulca[MAX], vlca[MAX];

long long dis[MAX];

int cnte, n, m;

// 存储所有询问

vector<Query> query[MAX];

void init()

{

cnte = 0;

memset(head, 0, sizeof head);

memset(vis, 0, sizeof vis);

for (int i = 1; i <= n; i ++) {

father[i] = i;

}

}

int find(int x)

{

while (x != father[x]) {

father[x] = father[father[x]];

x = father[x];

}

return x;

}

// 并查集合并，注意tarjan求lca的时候，需要把孩子节点

// 并到父亲节点上，因此合并的时候谁并到谁的次序不能错！

// 把y并到x

void merge(int x, int y)

{

x = find(x);

y = find(y);

father[y] = x;

}

void add(int u, int v, int w)

{

++ cnte;

e[cnte].v = v;

e[cnte].w = w;

e[cnte].nt = head[u];

head[u] = cnte;

}

// 核心部分

void tarjan(int u, int fa)

{

// dfs

for (int i = head[u]; i; i = e[i].nt) {

int v = e[i].v;

if (v != fa) {

dis[v] = dis[u] + e[i].w;

tarjan(v, u);

}

}

// 可以把这些查询看作图的深度优先搜索树上的非树边

for (int i = 0; i < query[u].size(); i ++) {

int v = query[u][i].v;

int id = query[u][i].id;

// 如果v被访问过了，这条边如果相当于前向边（祖先指向孩子）

// 此时find(v) = u

// 或者如果是交叉边（无祖先关系）

// 此时find(v)是使uv在其不同子树的节点

if (vis[v]) {

lca[id] = find(v);

}

}

// 当该节点及其子树所有节点都访问过，才把当前节点并到父节点上

// 并且这时才置访问标记

merge(fa, u);

// 第一次访问到该节点就置访问标记也对，

// 不过会使uv有直接祖先关系的查询查询两次

vis[u] = 1;

}

int main()

{

int t;

scanf("%d", &t);

while (t --) {

scanf("%d%d", &n, &m);

init();

for (int i = 1; i < n; i ++) {

int u, v, w;

scanf("%d%d%d", &u, &v, &w);

add(u, v, w);

add(v, u, w);

}

for (int i = 1; i <= m; i ++) {

int u, v;

scanf("%d%d", &u, &v);

ulca[i] = u;

vlca[i] = v;

query[u].push\_back(Query(v, i));

query[v].push\_back(Query(u, i));

}

dis[1] = 0;

tarjan(1, 0);

for (int i = 1; i <= m; i ++) {

printf("%lld\n", dis[ulca[i]] + dis[vlca[i]] - 2 \* dis[lca[i]]);

}

}

return 0;

}

## KMP

#include <iostream>

#include <algorithm>

#include <cstring>

#include <cstdio>

using namespace std;

const int MAX = 1e6 + 10;

char pat[MAX];

char tar[MAX];

int nt[MAX], ans;

void getNext()

{

int lenp = strlen(pat);

int lent = strlen(tar);

memset(nt, 0, sizeof nt);

nt[0] = -1;

int i = 0;

int j = -1;

while (i < lenp) {

if (j == -1 || pat[i] == pat[j]) {

nt[ ++ i] = ++ j;

} else {

j = nt[j];

}

}

}

void kmp()

{

ans = 0;

int i = 0, j = 0;

int lent = strlen(tar);

int lenp = strlen(pat);

while (i < lent) {

if (tar[i] == pat[j] || j == -1) {

i ++; j ++;

} else {

j = nt[j];

}

if (j == lenp) {

ans ++;

j = nt[j];

}

}

}

int main()

{

int t;

scanf("%d", &t);

while (t --) {

scanf(" %s %s", pat, tar);

getNext();

kmp();

printf("%d\n", ans);

}

}

## AC自动机

#include <bits/stdc++.h>

using namespace std;

const int MAX = 1e6 + 10;

int n, ans;

struct acm {

static int tot;

int trie[MAX][26], sum[MAX], fail[MAX];

void init() {

memset(sum, 0, sizeof(int) \* (tot + 1));

memset(fail, 0, sizeof(int) \* (tot + 1));

for (int i = 0; i <= tot; i ++) {

for (int j = 0; j < 26; j ++) {

trie[i][j] = 0;

}

}

fail[0] = 0;

tot = 0;

}

void add(char \*s, int len) {

int x = 0;

for (int i = 0; i < len; i ++) {

int id = s[i] - 'a';

if (!trie[x][id]) {

trie[x][id] = ++ tot;

}

x = trie[x][id];

if (i == len - 1) {

sum[x] ++;

}

}

}

int getfail(int x, int k) {

if (trie[x][k]) return trie[x][k];

if (x == 0) return 0;

return getfail(fail[x], k);

}

void makefail() {

queue<int> q;

q.push(0);

while (!q.empty()) {

int now = q.front(); q.pop();

for (int i = 0; i < 26; i ++) {

if (trie[now][i]) {

if (now == 0) {

fail[trie[now][i]] = 0;

} else {

fail[trie[now][i]] = getfail(fail[now], i);

}

q.push(trie[now][i]);

}

}

}

}

void match(char \*s, int len) {

int x = 0;

for (int i = 0; i < len; i ++) {

int id = s[i] - 'a';

while (x && !trie[x][id]) x = fail[x];

x = trie[x][id];

int temp = x;

while (temp) {

if (sum[temp]) {

ans += sum[temp];

sum[temp] = 0;

}

temp = fail[temp];

}

}

}

} ac;

int acm::tot = MAX;

int main()

{

int t;

scanf("%d", &t);

while (t --) {

ans = 0;

char s[MAX];

ac.init();

scanf("%d", &n);

for (int i = 1; i <= n; i ++) {

scanf(" %s", s);

ac.add(s, strlen(s));

}

ac.makefail();

scanf(" %s", s);

ac.match(s, strlen(s));

printf("%d\n", ans);

}

return 0;

}

## 凸包

#include <iostream>

#include <algorithm>

#include <cstring>

#include <cstdio>

#include <stack>

using namespace std;

const int MAX = 110;

const int INF = 4e4 + 10;

const double EPS = 1e-6;

struct Point {

double x, y;

Point() {}

Point(double xx, double yy): x(xx), y(yy) {}

Point operator+(const Point& rhs) const {

return Point(x + rhs.x, y + rhs.y);

}

Point operator-(const Point& rhs) const {

return Point(x - rhs.x, y - rhs.y);

}

Point operator\*(double rhs) const {

return Point(x \* rhs, y \* rhs);

}

} p[MAX];

int n;

stack<Point> s;

// 叉积，两向量角度为p1到p2逆时针旋转的角度，180度以内叉积为正，否则为负

double det(const Point& p1, const Point& p2)

{

return (p1.x \* p2.y - p1.y \* p2.x);

}

// 极角排序，atan2(y, x)求极角

bool cmp(const Point& p1, const Point& p2)

{

double a = atan2(p1.y - p[1].y, p1.x - p[1].x);

double b = atan2(p2.y - p[1].y, p2.x - p[1].x);

if (a != b) {

return a < b;

} else {

return p1.x < p2.x;

}

}

double dis(const Point& p1, const Point& p2)

{

return sqrt( (p1.x - p2.x) \* (p1.x - p2.x) + (p1.y - p2.y) \* (p1.y - p2.y));

}

void graham()

{

while (!s.empty()) {

s.pop();

}

s.push(p[1]);

s.push(p[2]);

for (int i = 3; i <= n; i ++) {

while (s.size() >= 2) {

Point mid = s.top();

s.pop();

Point last = s.top();

if (det(mid - p[i], mid - last) > 0) {

s.push(mid);

break;

}

}

s.push(p[i]);

}

}

int main()

{

while (~scanf("%d", &n), n) {

Point bottom;

bottom.x = INF;

bottom.y = INF;

int id = 0;

for (int i = 1; i <= n; i ++) {

scanf("%lf%lf", &p[i].x, &p[i].y);

if (p[i].x < bottom.x || (p[i].x == bottom.x && p[i].y < bottom.y)) {

bottom = p[i];

id = i;

}

}

if (n == 1) {

printf("0.00\n");

continue;

} else if (n == 2) {

printf("%.2f\n", dis(p[1], p[2]));

continue;

}

swap(p[1], p[id]);

sort(p + 2, p + n + 1, cmp);

graham();

// 求凸包周长

double ans = dis(s.top(), p[1]);

Point last = s.top();

s.pop();

while (!s.empty()) {

Point now = s.top();

s.pop();

ans += dis(now, last);

last = now;

}

printf("%.2f\n", ans);

}

return 0;

}

## 旋转卡壳求对踵点

#include <iostream>

#include <cstdio>

#include <cstring>

#include <algorithm>

#include <vector>

#include <cmath>

using namespace std;

const int MAX = 5e4 + 10;

const int INF = 1e5 + 10;

const double EPS = 1e-6;

struct Point {

double x, y;

Point() {}

Point(double xx, double yy): x(xx), y(yy) {}

Point operator+(const Point& rhs) const {

return Point(x + rhs.x, y + rhs.y);

}

Point operator-(const Point& rhs) const {

return Point(x - rhs.x, y - rhs.y);

}

Point operator\*(double rhs) const {

return Point(x \* rhs, y \* rhs);

}

bool operator<(const Point& rhs) const {

if (x != rhs.x) {

return x < rhs.x;

} else {

return y < rhs.y;

}

}

} p[MAX];

int n;

vector<Point> s;

bool cmp(const Point& p1, const Point& p2)

{

double a = atan2(p1.y - p[0].y, p1.x - p[0].x);

double b = atan2(p2.y - p[0].y, p2.x - p[0].x);

if (a != b) {

return a < b;

} else {

return p1.x < p2.x;

}

}

double dis2(const Point& p1, const Point& p2)

{

return (p1.x - p2.x) \* (p1.x - p2.x) + (p1.y - p2.y) \* (p1.y - p2.y);

}

double det(const Point& p1, const Point& p2)

{

return (p1.x \* p2.y - p1.y \* p2.x);

}

void graham()

{

s.clear();

s.push\_back(p[0]);

s.push\_back(p[1]);

for (int i = 2; i < n; i ++) {

while (s.size() >= 2) {

Point mid = s.back();

s.pop\_back();

Point last = s.back();

if (det(mid - p[i], mid - last) > 0) {

s.push\_back(mid);

break;

}

}

s.push\_back(p[i]);

}

}

double rotation()

{

int size = s.size();

if (size == 1) {

return 0;

} else if (size == 2) {

return dis2(s[0], s[1]);

}

int bg = 0;

int ed = 0;

for (int k = 1; k < size; k ++) {

if (s[k] < s[bg]) {

bg = k;

}

if (s[ed] < s[k]) {

ed = k;

}

}

int i = bg, j = ed;

double dis = -1;

while (i != ed || j != bg) {

dis = max(dis, dis2(s[i], s[j]));

if (det(s[(i + 1) % size] - s[i], s[(j + 1) % size] - s[j]) < 0) {

i = (i + 1) % size;

} else {

j = (j + 1) % size;

}

}

return dis;

}

int main()

{

while (~scanf("%d", &n)) {

Point bottom;

int id;

bottom.x = bottom.y = INF;

for (int i = 0; i < n; i ++) {

scanf("%lf%lf", &p[i].x, &p[i].y);

if (p[i].x < bottom.x || (p[i].x == bottom.x && p[i].y < bottom.y)) {

bottom = p[i];

id = i;

}

}

swap(p[id], p[0]);

sort(p + 1, p + n, cmp);

graham();

printf("%.0f\n", rotation());

}

return 0;

}

## 平板电视红黑树

#include <iostream>

#include <algorithm>

#include <cstdio>

#include <cstring>

#include <ext/pb\_ds/assoc\_container.hpp>

using namespace std;

using namespace \_\_gnu\_pbds;

const int MAX = 1e6 + 10;

int vis[MAX], occ[MAX];

int ans[MAX];

int n, m, minn;

struct node {

int value;

int curid;

node() {}

node(int vv, int cc): value(vv), curid(cc) {}

bool operator< (const node& rhs) const {

return curid < (rhs.curid);

}

} nd[MAX];

bool cmp(const node& n1, const node& n2)

{

return n1.curid < n2.curid;

}

tree<node, null\_type, less<node>, rb\_tree\_tag, tree\_order\_statistics\_node\_update> rbt;

int main()

{

scanf("%d%d", &n, &m);

for (int i = 1; i <= n; i ++) {

nd[i].value = nd[i].curid = i;

rbt.insert(nd[i]);

}

int flag = true;

minn = 0;

for (int i = 1; i <= m; i ++) {

int u, v;

scanf("%d%d", &u, &v);

if (!flag) {

continue;

}

// 找第v大

auto it = rbt.find\_by\_order(v - 1);

if (vis[it -> value] && ans[it -> value] != u) {

flag = false;

continue;

}

if (occ[u] && ans[it -> value] != u) {

flag = false;

continue;

}

vis[it -> value] = 1;

occ[u] = 1;

ans[it -> value] = u;

// 插入

rbt.insert(node(it -> value, minn --));

// 删除

rbt.erase(it);

}

if (!flag) {

puts("-1");

return 0;

}

int id = 1;

for (int i = 1; i <= n; i ++) {

if (!ans[i]) {

while (occ[id]) {

++ id;

}

ans[i] = id;

occ[id] = 1;

}

printf("%d ", ans[i]);

}

puts("");

return 0;

}

## 笛卡尔树

// 笛卡尔树，中序遍历得到原数组，从数组元素的值来看是堆

#include <iostream>

#include <cstdio>

#include <algorithm>

#include <cstring>

#include <stack>

using namespace std;

const int MAX = 5e4 + 10;

struct Node {

int idx; // 在数组中的下标

int v; // 数组元素的值

int id; // 输入时的顺序

} node[MAX];

int n;

int l[MAX], r[MAX], p[MAX];

stack<Node> s;

bool cmp(const Node& n1, const Node& n2)

{

return n1.idx < n2.idx;

}

// 单调栈，保存建树过程中的右链，这里是小根堆

// 最后栈底元素就是整棵树的根

void build()

{

while (!s.empty()) {

s.pop();

}

s.push(node[1]);

int id = node[1].id;

l[id] = r[id] = p[id] = 0;

// 由于是按照下标排序，故每次插入一定是在树右链的末端

for (int i = 2; i <= n; i ++) {

Node now; // 当前栈顶节点

Node last; // 最近一次弹出的节点

last.id = 0; // 初始化

// 寻找右链中第一个比待插入节点元素值小的节点

while (!s.empty()) {

now = s.top();

if (now.v < node[i].v) {

break;

}

last = s.top();

s.pop();

}

if (s.empty()) {

// 没有找到比待插入节点更小的元素，待插入的是当前最小的

int curid = node[i].id;

int lastid = last.id;

// 把整棵树链到待插入节点的左儿子上，待插入的成为新的根

p[curid] = r[curid] = 0;

l[curid] = lastid;

p[lastid] = curid;

} else {

// 找到了比待插入元素更小的节点

int curid = node[i].id;

int lastid = last.id;

int fatherid = now.id;

// 把待插入节点成为其右儿子

p[curid] = fatherid;

r[fatherid] = curid;

// 原来的右子树变成待插入节点的左子树

l[curid] = lastid;

r[curid] = 0;

p[lastid] = curid;

}

// 更新右链

s.push(node[i]);

}

}

int main()

{

while (~scanf("%d", &n)) {

for (int i = 1; i <= n; i ++) {

scanf("%d%d", &node[i].idx, &node[i].v);

node[i].id = i;

}

// 按照下标从小到大排序

sort(node + 1, node + 1 + n, cmp);

build();

puts("YES");

for (int i = 1; i <= n; i ++) {

printf("%d %d %d\n", p[i], l[i], r[i]);

}

}

return 0;

}

## 归并树求区间第k大 O（n(logn)^2）

/\*\*

\* 归并树，线段树中每个节点保存归并排序时对应区间内排序好的数列，

\* 即保存的是归并排序的过程

\*

\*/

#include <iostream>

#include <cstdio>

#include <cstring>

#include <algorithm>

#include <vector>

using namespace std;

const int MAX = 1e5 + 10;

struct Tree {

vector<int> v;

int l, r;

} t[MAX << 2];

int n, m, cnta;

int a[MAX];

// 归并排序中的合并

void maintain(int x)

{

int lson = x << 1;

int rson = x << 1 | 1;

vector<int>::iterator i = t[lson].v.begin();

vector<int>::iterator j = t[rson].v.begin();

while (i != t[lson].v.end() && j != t[rson].v.end()) {

if (\*i < \*j) {

t[x].v.push\_back(\*i);

++ i;

} else {

t[x].v.push\_back(\*j);

++ j;

}

}

while (i != t[lson].v.end()) {

t[x].v.push\_back(\*i);

++ i;

}

while (j != t[rson].v.end()) {

t[x].v.push\_back(\*j);

++ j;

}

}

void build(int x, int l, int r)

{

t[x].l = l;

t[x].r = r;

t[x].v.clear();

if (l == r) {

scanf("%d", &a[++ cnta]);

t[x].v.push\_back(a[cnta]);

return ;

}

int mid = (l + r) >> 1;

build(x << 1, l, mid);

build(x << 1 | 1, mid + 1, r);

maintain(x);

}

int query(int x, int l, int r, int value)

{

int ret = 0;

if (l <= t[x].l && t[x].r <= r) {

ret = upper\_bound(t[x].v.begin(), t[x].v.end(), value) - t[x].v.begin();

return ret;

}

int mid = (t[x].l + t[x].r) >> 1;

if (r <= mid) {

ret = query(x << 1, l, r, value);

} else if (l >= mid + 1) {

ret = query(x << 1 | 1, l, r, value);

} else {

ret = query(x << 1, l, mid, value);

ret += query(x << 1 | 1, mid + 1, r, value);

}

return ret;

}

int main()

{

while (~scanf("%d%d", &n, &m)) {

cnta = 0;

build(1, 1, n);

sort(a + 1, a + 1 + n);

// 对于答案a[l]，当前区间内有k个数字小于等于a[l]，且a[l]是这样数字中最小的那个

while (m --) {

int left, right, k;

scanf("%d%d%d", &left, &right, &k);

int l = 1, r = n, mid;

while (l <= r) {

mid = (l + r) >> 1;

int num = query(1, left, right, a[mid]);

if (num <= k - 1) {

l = mid + 1;

} else {

r = mid - 1;

}

}

printf("%d\n", a[l]);

}

}

return 0;

}

## 平方分割求区间第k大 O（nlogn + m\*sqrt(n)\*(logn)^2）

#include <iostream>

#include <cstdio>

#include <algorithm>

#include <cstring>

#include <cmath>

using namespace std;

const int MAX = 1e5 + 10;

int a[MAX];

int sorted[MAX];

int bucket[MAX / 10][MAX / 10];

int n, m;

int num, size;

// 统计区间l到r中小于等于x的数字的个数

int Count(int l, int r, int x)

{

int cnt = 0;

int bound;

// 首先处理区间l到r没有完全覆盖整个桶的部分，这部分暴力的查找

if (l % size != 0) {

bound = (l / size + 1) \* size;

while (l <= r && l < bound) {

if (a[l] <= x) {

++ cnt;

}

++ l;

}

}

if ((r + 1) % size != 0) {

bound = (r / size) \* size;

while (l <= r && r >= bound) {

if (a[r] <= x) {

++ cnt;

}

-- r;

}

}

// 然后处理区间内的每个桶，由于桶内是有序的，故可以二分找到小于等于x的数的个数

if (l <= r) {

int beg = l / size;

int ed = r / size;

for (int i = beg; i <= ed; i ++) {

cnt += upper\_bound(bucket[i], bucket[i] + size, x) - (bucket[i]);

}

}

return cnt;

}

int main()

{

while (~scanf("%d%d", &n, &m)) {

size = sqrt(n \* log2(n));

if (size == 0)

size = 1;

num = n / size;

for (int i = 0; i < n; i ++) {

scanf("%d", &a[i]);

sorted[i] = a[i];

}

sort(sorted, sorted + n);

// 分桶，并在每个桶内排序

for (int i = 0, cnt = 0; i < num; i ++) {

for (int j = 0; j < size; j ++, cnt ++) {

bucket[i][j] = a[cnt];

}

sort(bucket[i], bucket[i] + size);

}

// 核心思想是二分找到区间中 恰有k个数字小于等于sorted[mid] 中最小的那个，这样sorted[mid]就是答案

while (m --) {

int left, right, k;

scanf("%d%d%d", &left, &right, &k);

-- left;

-- right;

int l = 0, r = n - 1, mid, ret = -1;

while (l <= r) {

mid = (l + r) >> 1;

int num = Count(left, right, sorted[mid]);

if (num <= k - 1) {

l = mid + 1;

} else {

r = mid - 1;

}

}

printf("%d\n", sorted[l]);

}

}

return 0;

}

## 线段树

#include <iostream>

#include <cstdio>

#include <algorithm>

#include <cstring>

using namespace std;

const int MAX = 1e5 + 10;

struct Tree {

long long sum;

int l, r, len;

long long tag;

} t[MAX << 2];

int n, q;

void maintain(int x)

{

t[x].sum = t[x << 1].sum + t[x << 1 | 1].sum;

}

void pushdown(int x)

{

if (t[x].tag) {

t[x << 1].sum += t[x].tag \* t[x << 1].len;

t[x << 1].tag += t[x].tag;

t[x << 1 | 1].sum += t[x].tag \* t[x << 1 | 1].len;

t[x << 1 | 1].tag += t[x].tag;

t[x].tag = 0;

}

}

void build(int x, int l, int r)

{

t[x].l = l;

t[x].r = r;

t[x].len = r - l + 1;

if (l == r) {

scanf("%lld", &t[x].sum);

return ;

}

int mid = (l + r) >> 1;

build(x << 1, l, mid);

build(x << 1 | 1, mid + 1, r);

maintain(x);

}

void modify(int x, int l, int r, int del)

{

if (l <= t[x].l && t[x].r <= r) {

t[x].sum += t[x].len \* del;

t[x].tag += del;

return ;

}

pushdown(x);

int mid = (t[x].l + t[x].r) >> 1;

if (r <= mid) {

modify(x << 1, l, r, del);

} else if (l >= mid + 1) {

modify(x << 1 | 1, l, r, del);

} else {

modify(x << 1, l, mid, del);

modify(x << 1 | 1, mid + 1, r, del);

}

maintain(x);

}

long long query(int x, int l, int r)

{

if (l <= t[x].l && t[x].r <= r) {

return t[x].sum;

}

pushdown(x);

int mid = (t[x].l + t[x].r) >> 1;

long long ret = 0;

if (r <= mid) {

ret = query(x << 1, l, r);

} else if (l >= mid + 1) {

ret = query(x << 1 | 1, l, r);

} else {

ret = query(x << 1, l, mid);

ret += query(x << 1 | 1, mid + 1, r);

}

maintain(x);

return ret;

}

int main()

{

while (~scanf("%d%d", &n, &q)) {

build(1, 1, n);

while (q --) {

char c;

int l, r, del;

scanf(" %c", &c);

if (c == 'Q') {

scanf("%d%d", &l, &r);

printf("%lld\n", query(1, l, r));

} else {

scanf("%d%d%d", &l, &r, &del);

modify(1, l, r, del);

}

}

}

return 0;

}

## 欧拉路

#include <iostream>

#include <algorithm>

#include <cstdio>

#include <cstring>

#include <stack>

#include <queue>

using namespace std;

const int MAX = 1e5 + 10;

struct E {

int v, nt, id, vis;

} e[MAX << 2];

int head[MAX], cnte;

int father[MAX], size[MAX], sumodd[MAX], tmp[MAX << 2];

int degree[MAX];

int n, m, ans;

stack<int> s;

queue<int> q;

vector<int> outp;

void init()

{

cnte = 0;

memset(head, -1, sizeof head);

for (int i = 1; i <= n; i ++) {

father[i] = i;

size[i] = 1;

}

memset(degree, 0, sizeof degree);

memset(sumodd, 0, sizeof sumodd);

ans = 0;

}

int find(int x)

{

while (x != father[x]) {

father[x] = father[father[x]];

x = father[x];

}

return x;

}

void merge(int x, int y)

{

x = find(x);

y = find(y);

if (x > y) {

father[y] = x;

size[x] += size[y];

} else if (x < y) {

father[x] = y;

size[y] += size[x];

}

}

void add(int u, int v, int id)

{

e[cnte].v = v;

e[cnte].id = id;

e[cnte].vis = 0;

e[cnte].nt = head[u];

head[u] = cnte;

++ cnte;

}

/\*\*

\* dfs找欧拉路，前提是存在欧拉路

\* 如果都是偶点，或者入度等于出度，则从任意一点开始都可以

\* 如果有两个奇点，则需要从其中一个开始dfs

\* 如果有两个点入度不等于出度，这两个点必须其中一个出度比入度大1，另一个入度比出度大1，

\* 从出度比入度大1的点开始

\* 核心是递归返回时把边压入栈中

\* dfs结束后从栈中依次弹出就是路径

\*/

void dfs(int x)

{

for (int i = head[x]; ~i; i = e[i].nt) {

if (!e[i].vis) {

e[i].vis = 1;

e[i ^ 1].vis = 1;

dfs(e[i].v);

s.push(e[i].id);

}

}

}

int main()

{

while (~scanf("%d%d", &n, &m)) {

init();

for (int i = 1; i <= m; i ++) {

int u, v;

scanf("%d%d", &u, &v);

add(u, v, i);

add(v, u, -i);

merge(u, v);

degree[u] ^= 1;

degree[v] ^= 1;

}

int lastodd;

int oddcnt = 0;

for (int i = 1; i <= n; i ++) {

if (degree[i] & 1) {

int x = find(i);

sumodd[x] += 1;

if (!oddcnt) {

oddcnt ^= 1;

lastodd = i;

} else {

oddcnt ^= 1;

add(i, lastodd, MAX);

add(lastodd, i, MAX);

}

}

}

/\*\*

\* 一个连通分量需要的一笔画的笔数：

\* 1. 孤立点为0

\* 2. 无奇点为1

\* 3. 否则是奇点数目除以2

\*/

for (int i = 1; i <= n; i ++) {

int x = find(i);

if (i == x && size[x] != 1) {

ans += max(1, sumodd[x] / 2);

}

}

printf("%d\n", ans);

for (int i = 1; i <= n; i ++) {

if (i == find(i) && size[i] != 1) {

dfs(i);

int maxcnt = 0;

while (!s.empty()) {

int id = s.top();

if (id == MAX) {

if (maxcnt && outp.size()) {

printf("%d ", outp.size());

for (int i = 0; i < outp.size(); i ++) {

printf(i==(outp.size()-1)?"%d\n":"%d ", outp[i]);

}

outp.clear();

}

maxcnt ++;

} else {

if (maxcnt == 0) {

q.push(id);

} else {

outp.push\_back(id);

}

}

s.pop();

}

int ts = q.size() + outp.size();

if (ts) {

printf("%d ", ts);

for (int i = 0; i < outp.size(); i ++) {

if (i == outp.size() - 1) {

if (i == ts - 1) {

printf("%d\n", outp[i]);

} else {

printf("%d ", outp[i]);

}

} else {

printf("%d ", outp[i]);

}

}

outp.clear();

while (!q.empty()) {

printf("%d", q.front());

q.pop();

if (!q.empty()) {

putchar(' ');

} else {

puts("");

}

}

}

}

}

}

return 0;

}

## 2-sat

/\*\*

\* 2-sat 是可满足性问题，化成合取范式之后，每个子句中文字个数不超过2

\* 对一个子句(a V b)，若a为假则b必须为真，若b为假则a必须为真

\* 即!a为真，b必须为真，!b为真，a必须为真

\* 如此，一个文字分为两个节点a与!a，可以连两条有向边，一条是!a -> b，一条是!b -> a

\* 若a !a出现在同一强连通分量中，则一定无解

\* 否则有解，看a !a所处的强连通分量的拓扑序，若!a在a之前，则a为真，否则a为假

\*

\* 当面对一个点只有两种状态，两种状态非此即彼的时候，可以考虑2-sat

\*/

#include <iostream>

#include <algorithm>

#include <cstdio>

#include <cstring>

#include <stack>

#include <queue>

using namespace std;

const int MAXN = 2e3 + 10;

const int MAXM = 4e6 + 10;

struct edge {

int v, nt;

} e[MAXM];

int head[MAXN], cnte;

int dfn[MAXN], low[MAXN], idx, color[MAXN], cntc;

stack<int> s;

// tsort

queue<int> q;

int head2[MAXN], cnte2;

edge e2[MAXM];

int indeg[MAXN], corder[MAXN];

int n;

int beg[MAXN], ed[MAXN], len[MAXN];

void add(int h[MAXN], edge ed[MAXM], int &cnt, int u, int v)

{

++ cnt;

ed[cnt].v = v;

ed[cnt].nt = h[u];

h[u] = cnt;

}

bool overlap(int x1, int y1, int x2, int y2)

{

return (x2 < x1 && x1 < y2) ||

(x2 < y1 && y1 < y2) ||

(x1 < x2 && x2 < y1) ||

(x1 < y2 && y2 < y1) ||

(x1 == x2) ||

(y1 == y2);

}

void tarjan(int u)

{

dfn[u] = low[u] = ++ idx;

s.push(u);

for (int i = head[u]; i; i = e[i].nt) {

int v = e[i].v;

if (!dfn[v]) {

tarjan(v);

low[u] = min(low[u], low[v]);

} else if (!color[v]) {

low[u] = min(low[u], dfn[v]);

}

}

if (low[u] == dfn[u]) {

++ cntc;

while (true) {

int now = s.top();

s.pop();

color[now] = cntc;

if (now == u) {

break;

}

}

}

}

void tsort()

{

int ord = 0;

for (int i = 1; i <= cntc; i ++) {

if (indeg[i] == 0) {

q.push(i);

}

}

while (!q.empty()) {

int u = q.front();

q.pop();

corder[u] = ++ ord;

for (int i = head2[u]; i; i = e2[i].nt) {

int v = e2[i].v;

indeg[v] --;

if (!indeg[v]) {

q.push(v);

}

}

}

}

int main()

{

while (~scanf("%d", &n)) {

int h, m;

for (int i = 0; i < n; i ++) {

scanf("%d:%d", &h, &m);

beg[i] = 60 \* h + m;

scanf("%d:%d", &h, &m);

ed[i] = 60 \* h + m;

scanf("%d", &len[i]);

}

cnte = 0;

memset(head, 0, sizeof head);

// 拆点，分别为i, i + n

for (int i = 0; i < n; i ++) {

for (int j = 0; j < i; j ++) {

if (overlap(beg[i], beg[i] + len[i], beg[j], beg[j] + len[j])) {

// !(a && b) == !a V !b

add(head, e, cnte, i, j + n);

add(head, e, cnte, j, i + n);

}

if (overlap(beg[i], beg[i] + len[i], ed[j] - len[j], ed[j])) {

// !(a && !b) == !a V b

add(head, e, cnte, i, j);

add(head, e, cnte, j + n, i + n);

}

if (overlap(ed[i] - len[i], ed[i], beg[j], beg[j] + len[j])) {

// !(!a && b) == a V !b

add(head, e, cnte, i + n, j + n);

add(head, e, cnte, j, i);

}

if (overlap(ed[i] - len[i], ed[i], ed[j] - len[j], ed[j])) {

// !(!a && !b) == a V b

add(head, e, cnte, i + n, j);

add(head, e, cnte, j + n, i);

}

}

}

// tarjan

memset(dfn, 0, sizeof dfn);

idx = 0;

memset(color, 0, sizeof color);

cntc = 0;

for (int i = 0; i < 2 \* n; i ++) {

if (!dfn[i]) {

tarjan(i);

}

}

int ok = true;

for (int i = 0; i < n; i ++) {

if (color[i] == color[i + n]) {

ok = false;

break;

}

}

if (!ok) {

puts("NO");

} else {

puts("YES");

memset(indeg, 0, sizeof indeg);

memset(head2, 0, sizeof head2);

cnte2 = 0;

for (int u = 0; u < 2 \* n; u ++) {

for (int i = head[u]; i; i = e[i].nt) {

if (color[u] != color[e[i].v]) {

indeg[color[e[i].v]] ++;

add(head2, e2, cnte2, color[u], color[e[i].v]);

}

}

}

// tsort

tsort();

for (int i = 0; i < n; i ++) {

int b, e;

if (corder[color[i]] > corder[color[i + n]]) {

// !a在a之前，a为真

b = beg[i];

e = beg[i] + len[i];

} else {

// !a在a之后，a为假

b = ed[i] - len[i];

e = ed[i];

}

printf("%02d:%02d %02d:%02d\n", b / 60, b % 60, e / 60, e % 60);

}

}

}

return 0;

}

## 树上点分治

/\*\*

\* 分治常常能把一个n的复杂度降到log n，数列上的分治常常比较好实现，但是树上怎么办呢？

\* 树上如果随便找一个点的话，可能会退化，因此需要每次从重心分割然后分治

\*

\* 本题的思路是点对

\* (1) 在同一颗子树内，是一个子问题，那么就递归求解

\* (2) 在不同子树内，需要求出每个点到重心的距离，然后合并之。

\* 考虑把所有点到重心的距离都放到一个vector里面，排好序，

\* 这样两个指针一个从头一个从尾扫一扫就能求出所有距离和小于k的点对数。

\* 不过这样会把同一棵子树的点对也算进去，需要再减出来，方法同上

\* (3) 一个在子树一个在重心，可以把重心当作一个距离重心为0的点

\*

\* 分治共log n层，每一层都需要对所有节点排序，所以总复杂度是n log^2 n的

\*/

#include <iostream>

#include <cstdio>

#include <cstring>

#include <algorithm>

#include <vector>

using namespace std;

const int MAX = 1e4 + 10;

struct edge {

int v, l, nt;

} e[MAX << 1];

int head[MAX], cnte;

int n, k, ans;

int centriod, censz;

int sz[MAX];

int vis[MAX]; // 标记这个点是否已经被作为重心去掉了

int dis[MAX]; // 标记这个点到其重心的距离

vector<int> disvec; // 记录点到重心的距离

void add(int u, int v, int l)

{

++ cnte;

e[cnte].v = v;

e[cnte].l = l;

e[cnte].nt = head[u];

head[u] = cnte;

}

// 求重心

void getCentriod(int u, int fa, int component)

{

sz[u] = 1;

int maxx = 0;

for (int i = head[u]; i; i = e[i].nt) {

int v = e[i].v;

if (v == fa || vis[v]) {

continue;

}

getCentriod(v, u, component);

sz[u] += sz[v];

maxx = max(maxx, sz[v]);

}

maxx = max(maxx, component - sz[u]);

if (maxx < censz) {

censz = maxx;

centriod = u;

}

return ;

}

// 求各个点到重心的距离，同时求重心分割后各个子树的size

void getdis(int u, int fa)

{

disvec.push\_back(dis[u]);

sz[u] = 1;

for (int i = head[u]; i; i = e[i].nt) {

int v = e[i].v;

if (vis[v] || v == fa) {

continue;

}

dis[v] = dis[u] + e[i].l;

getdis(v, u);

sz[u] += sz[v];

}

return ;

}

/\*\*

\* u传入重心，那么就是在计算

\* 被分割的不同子树中所有到重心距离和不超过k的点对个数（包含同一子树的）

\* 此时initDis传入0

\*

\* u传入的是重心分割后子树的根，

\* 那么就是在计算这一棵子树中所有到重心距离和不超过k的点对个数

\* 此时initDis传入重心到子树根的距离

\*/

int getLessK(int u, int initDis)

{

disvec.clear();

dis[u] = initDis;

getdis(u, -1);

sort(disvec.begin(), disvec.end());

int ret = 0;

for (int i = 0, j = disvec.size() - 1; i < j;) {

if (disvec[i] + disvec[j] <= k) {

ret += j - i;

++ i;

} else {

-- j;

}

}

return ret;

}

void solve(int cen)

{

vis[cen] = 1;

ans += getLessK(cen, 0);

for (int i = head[cen]; i; i = e[i].nt) {

int v = e[i].v;

if (vis[v]) {

continue;

}

// 之前算的时候会把在同一子树的点对也算进去，这是不对的，要减出来

ans -= getLessK(v, e[i].l); // 此时已经将对应子树的size求出来了，就是sz[v]

censz = MAX;

getCentriod(v, cen, sz[v]);

solve(centriod);

}

}

int main()

{

while (~scanf("%d%d", &n, &k), n || k) {

memset(head, 0, sizeof head);

cnte = 0;

for (int i = 1; i < n; i ++) {

int u, v, l;

scanf("%d%d%d", &u, &v, &l);

add(u, v, l);

add(v, u, l);

}

memset(vis, 0, sizeof vis);

censz = MAX;

getCentriod(1, -1, n); // 获取最初重心

ans = 0;

solve(centriod); // 分治

printf("%d\n", ans);

}

return 0;

}