前缀 差分 位运算 快速幂 倍增

前缀

二维前缀和: $sum[i][j] = sum[i][j - 1] + sum[i - 1][j] - sum[i - 1][j - 1] + a[i][j]$

计算一个区间的和(x1 , y1) (x2 , y2) $ans = sum[x2][y2] - sum[x2][y1 - 1] - sum[x1 - 1][y2] + sum[x1 - 1][y1 - 1]$给一个区间加上x $sum[x1][y1] + x sum[x2 + 1][y1] - x sum[x1][y2 + 1] - x sum[x2 + 1][y2 + 1] + x$

差分

差分指的是两个数的差

a1 a2 - a1 a3 - a2 a4 - a3

差分数组的前缀和就是该数的值 例如a4 = b4 + b3 + b2 + b1

这样计算出差分数组之后如果要对区间[L , R]进行想加操作的话 只要对 $d[L] + x , d[R + 1] - x$ 即可

位运算

1.取某位k位 x &(1 << k - 1)

2.取右数的第k位 x >> (k - 1) & 1

3.判断奇偶 x & 1

4.取最低位的1 lowbit(x) = x & (-x)

快速幂

```
long long qpow(long long a , long long b , long long p){
    long long res = 1;
    while(b){
        if(b & 1) res = res * a % p;
        a = a * a % p;
        b >>= 1;
    }
    return res;
}
```

```
(a / c) % p = a * c ^ (p - 2) % p
```

倍增

```
#include <iostream>
#include <algorithm>
#include <cmath>
using namespace std;
int st[100001][22];
int N, M;
inline int read(){ //快速读取
    int x = 0, f = 1; char ch = getchar();
    while (ch < '0' || ch > '9') {if (ch == '-') f = -1; ch = getchar();}
    while (ch >= '0' && ch <= '9') {x = x * 10 + ch - 48; ch = getchar();}
    return x * f;
}
//__lg是快速log2 时间为O(1)
void ST(){
    for (int i = 1; i <= __lg(N); i++)
        for (int j = 1; j + (1 << i) - 1 <= N; j++)
            st[j][i] = max(st[j][i - 1], st[j + (1 << (i - 1))][i - 1]);
}
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    N = read();
    M = read();
    for (int i = 1; i <= N; i++) st[i][0] = read();
    ST();
    for (int i = 0; i < M; i++){
        int l = read(), r = read();
        int k = __lg(r - l + 1);
        cout << max(st[l][k], st[r - (1 << k) + 1][k]) << "\n";
    }
    return 0;
}
```

洛谷3865模板

寻找区间内的最大或者最小值

先进行预处理找到从这个数到1 2 4 8 16等位置的最值

查找：　区间为(l , r) 那我只要找到log2（r - l + 1）这个范围 两个范围相加就是这个区间

贪心

最优解包含贪心选择的证明可用反证法，证明如下： 假设最优解不含贪心选择，即最优解 的某个活动 所选择的会场 不是结束时间最早的会场， 由于 的结束时间小于， 能在 中进行，说明 必定能在 中进行， 因此用 代替 的结果 仍是一个最优解， 即最优解包含贪心选择，与假设矛盾。

[FatMouse'Trade](FatMouse' Trade - HDU 1009 - Virtual Judge (vjudge.net))

```
struct node {
    double j, f, val;
    bool operator<(const node &T) const {
        return val > T.val;
    }
} a[N];
// bool cmp(node a,node b){
// return a.val>b.val;
// }
int n, m;
void solve() {
    while (cin >> m >> n) {
        if (m == -1 && n == -1) {
            break;
        }
        for (int i = 1; i <= n; ++i) {
            cin >> a[i].j >> a[i].f;
            a[i].val = (1.0 * a[i].j) / a[i].f;
        }
        sort(a + 1, a + 1 + n);
        double ans = 0, now = m;
        for (int i = 1; i <= n; ++i) {
            if (now > a[i].f) {
                ans += a[i].j;
                now -= a[i].f;
            } else {
                ans += a[i].val * now;
                break;
            }
        }
        // cout<<fixed<<setprecision(3)<<ans<<endl;
        printf("%.3f\n", ans);
    }
}
```

[独木舟](独木舟 - 51Nod 1432 - Virtual Judge (vjudge.net))

```
int a[N];
void solve() {
    int n, w;
    cin >> n >> w;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }
    sort(a + 1, a + 1 + n);
    int l = 1, r = n;
    int ans = 0;
    while (l < r) {
        if (a[l] + a[r] > w) {
            r--;
```

```
        } else {
            r--, l++;
        }
        ans++;
    }
    if (l == r) {
        ans++;
    }
    cout << ans << endl;
}
```

[拼数](拼数 - 计蒜客 T2144 - Virtual Judge (vjudge.net))

```
string s[N];
bool cmp(string a, string b) {
    return (a + b > b + a);
}
void solve() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; ++i) {
        cin >> s[i];
    }
    sort(s + 1, s + 1 + n, cmp);
    for (int i = 1; i <= n; ++i) {
        cout << s[i];
    }
    cout << endl;
}
```

[今年寒假不AC](今年暑假不AC - HDU 2037 - Virtual Judge (vjudge.net))

```
struct node {
    int st, ed;
    bool operator<(const node T) const {
        return ed < T.ed;
    }
} a[N];
void solve() {
    int n;
    while (cin >> n) {
        if (n == 0) {
            break;
        }
        for (int i = 1; i <= n; ++i) {
            cin >> a[i].st >> a[i].ed;
        }
        sort(a + 1, a + 1 + n);
        int now = a[1].ed, cnt = 1;
```

```
        for (int i = 2; i <= n; ++i) {
            if (a[i].st >= now) {
                now = a[i].ed;
                ++cnt;
            }
        }
        cout << cnt << endl;
    }
}
```

区间选点&&最大不想交区间数量

```
struct node {
    int l, r;
    bool operator<(const node T) {
        return r < T.r;
    }
} a[N];
void solve() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i].l >> a[i].r;
    }
    sort(a + 1, a + 1 + n);
    int now = a[1].r, cnt = 1;
    for (int i = 2; i <= n; ++i) {
        if (a[i].l > now) {
            cnt++;
            now = a[i].r;
        }
    }
    cout << cnt << endl;
}
```

区间分组

```
struct node {
    int l, r;
    bool operator<(const node T) {
        return l < T.l;
    }
} a[N];
void solve() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i].l >> a[i].r;
    }
```

```
    sort(a + 1, a + 1 + n);
    int cnt = 0;
    priority_queue<int, vector<int>, greater<int>> q;
    for (int i = 1; i <= n; ++i) {
        if (q.size() == 0 || q.top() >= a[i].l) {
            cnt++;
        } else {
            q.pop();
        }
        q.push(a[i].r);
    }
    cout << cnt << endl;
}
```

区间覆盖

```
struct node {
    int l, r;
    bool operator<(const node T) {
        return l < T.l;
    }
}a[N];
void solve() {
    int st, ed, n;
    cin >> st >> ed >> n;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i].l >> a[i].r;
        Rock and Lever
    }
    sort(a + 1, a + 1 + n);
    int ok = 0, ans = 0;
    for (int i = 1; i <= n; i++) {
        int pos = i, r = -2e9;
        while (pos <= n && a[pos].l <= st) {
            r = max(r, a[pos].r);
            pos++;
        }
        if (r < st) {
            ans = -1;
            break;
        }
        ans++;
        if (r >= ed) {
            ok = 1;
            break;
        }
        st = r;
        i = pos - 1;
    }
    if (!ok) {
        ans = -1;
```

```
    }
    cout << ans << endl;
}
```

[rock and Lever](Rock and Lever - CodeForces 1420B - Virtual Judge (vjudge.net))

```
#define int ll
int a[N], cnt[N];
void solve() {
    int n;
    cin >> n;
    for (int i = 1, x; i <= n; ++i) {
        cin >> x;
        cnt[(int)log2(x)]++;
    }
    int ans = 0;
    for (int i = 0; i <= 40; ++i) {
        ans += ((cnt[i] * (cnt[i] - 1) / 2));
        cnt[i] = 0;
    }
    cout << ans << endl;
}
```

合并果子

```
void solve() {
    int n;
    cin >> n;
    priority_queue<int, vector<int>, greater<int>> q;
    for (int i = 1, x; i <= n; ++i) {
        cin >> x;
        q.push(x);
    }
    int ans = 0;
    while (q.size() != 1) {
        int num1 = q.top();
        q.pop();
        int num2 = q.top();
        q.pop();
        q.push(num1 + num2);
        ans += (num1 + num2);
    }
    cout << ans << endl;
}
```

Work Scheduling G

二分 三分

二分查找

二分查找 也叫折半搜索 是用来在一个有序数组中查找某一元素的算法

时间复杂度：logn

例题：洛谷P2249

模板1

```
bool check(int mid) {
    /*……*/
}
int main() {
    int l = 1, r = n;
    while (l < r) {
        int mid = l + r >> 1; // 区间[l, r]被划分成[l, mid]和[mid + 1, r]:
        if (check(mid))
            l = mid + 1; // check()判断mid是否满足性质
        else
            r = mid;
    }
    //[1,l-1]为check函数返回为true的部分， [l,r]为check函数返回为false的部分
    cout << l - 1 << endl; // 自己需要调边界
}
```

模板2

```
int main() {
    int l = 1, r = n;
    while (l <= r) {
        int mid = l + r >> 1;
        if (check(mid))
            l = mid + 1;
        else
            r = mid - 1;
    }
}
```

浮点二分

```
double eps = 1e-5;
while (fabs(r - l) > eps) // 需要一个精度保证
{
    double mid = (l + r) / 2;
    if (check(mid))
        l = mid;
    else
        r = mid;
}
```

需要一个精度作为保证

三分查找

通常用来求单峰问题(先升后降 先降后升)

例题：洛谷P3382

```
#include <iostream>
#include <cmath>
using namespace std;
double a[20] , n;
double way(double x){
    double ans = 0;
    for(int i = 0 ; i <= n ; i++){
        ans = ans * x + a[i];
    }
    return ans;
}
int main(){
    double l , r , eqs = 1e-5;
    cin >> n >> l >> r;
    for(int i = 0 ; i <= n ; i++) cin >> a[i];
    while(fabs(l - r) >= eqs){
        double mid = (l + r) / 2;
        if(way(mid - eqs) < way(mid + eqs)) l = mid;
        else r = mid;
    }
    cout << r << endl;
}
```

模板1

```
// l表示区间左端点，r表示区间右端点
while (fabs(r - l) > eps) {
    double mid1 = (r - l) / 3 + l;
    double mid2 = (r - l) / 3 * 2 + l;
    if (f(mid1) < f(mid2))
```

```
        l = mid1;
    else if (f(mid2) < f(mid1))
        r = mid2;
    else
        break;
}
```

模板二

```
// l表示区间左端点，r表示区间右端点
while (fabs(r - l) > eps) {
    double mid = (l + r) / 2;
    if (f(mid - eps) < f(mid + eps))
        l = mid;
    else
        r = mid;
}
```

另外也可以执行1000次来保证精度

```
// l表示区间左端点，r表示区间右端点
int cnt = 1000;
while (cnt--) {
    double mid = (l + r) / 2;
    if (f(mid - eps) < f(mid + eps))
        l = mid;
    else
        r = mid;
}
```

二分答案

通过二分来枚举答案 从而减小时间复杂度使n的复杂度变为logn

例题：洛谷 P2678 跳石头 P1824 进击的奶牛

```
#include<iostream>
#include<algorithm>
using namespace std;
int num[100010];
int n , m , L;
bool check(int mid){
    int last = 0 , cnt = 0;
    for(int i = 0 ; i <= n ; i++){
        if(num[i] - last >= mid) last = num[i];
        else cnt++;
    }
```

```
    if(cnt > m) return false;
    else return true;
}
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin >> L >> n >> m;
    for(int i = 0 ; i < n ; i++)
        cin >> num[i];
    num[n] = L;
    sort(num , num + n + 1);
    int l = 1 , r = L;
    while(l <= r){
        int mid = (l + r) >> 1;
        if(check(mid)) l = mid + 1;
        else r = mid - 1;
    }
    cout << r << endl;
}
```

```
#include<iostream>
#include<algorithm>
using namespace std;
int num[100010];
int n , m;
bool check(int mid){
    int last = num[0] , cnt = 0;
    for(int i = 1 ; i < n ; i++){
        if(num[i] - last >= mid) last = num[i];
        else cnt++;
    }
    if(cnt > n - m) return false;
    else return true;
}
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cin >> n >> m;
    for(int i = 0 ; i < n ; i++)
        cin >> num[i];
    sort(num , num + n);
    int l = 1 , r = num[n - 1] - num[0];
    while(l <= r){
        int mid = (l + r) >> 1;
        if(check(mid)) l = mid + 1;
        else r = mid - 1;
    }
    cout << r << endl;
}
```

二分交互

例题：洛谷P1733

```
#include<iostream>
using namespace std;
int main(){
    int l = 1 , r = 1e9;
    while(1){
        int mid = (l + r) >> 1;
        cout << mid << endl;
        int k;
        cin >> k;
        if(k == 0) break;
        else if(k == 1) r = mid - 1;
        else if(k == -1) l = mid + 1;
    }
    return 0;
}
```

二分函数

```
#include <algorithm> // 二分函数头文件
int main() {
    vector<int> a;
    int b[N];
    set<int> s;
    // 找升序数组中d的位置
    lower_bound(a.begin(), a.end(), d);
    lower_bound(b + 1, b + 1 + n, d);
    s.lower_bound(d);
    // 升序数组中第一个大于等于d的元素的位置，如果没找到就是a.end()
    upper_bound(); // 升序数组中第一个大于d的元素的位置
}
```

并查集 最小生成树

并查集模板

```
#include <iostream>
using namespace std;
const int N = 1e6;
int n, m, f[N];
void INIT() {
    for (int i = 1; i <= n; i++) f[i] = i;
```

```
}
int find(int x) {
    return x == f[x] ? x : f[x] = find(x);
}
int merge(int x, int y) {
    int xx = find(x);
    int yy = find(y);
    f[yy] = xx;
}
int main() {
}
```

例题: [Problem-C - codefores](Problem - C - Codeforces)

有三个排列，其中给 序列和 序列，对于第 个位置来说，ci = ai or bi; 问 序列最多有多少种组成方法 做法: 首先我们先考虑序列是以 排列 的形式给出，排列的特征使得如果，那么对于一个数字来说我最多可能出现在 的两个不同的位置 假设我们对于一个数字，他在第 位和第 位都有出现，假设我第 位选了，那么第 位我就选另外一个数字，反之同理，并且我们可以发现位置和位置之间都有相互牵制制约的关系，假设我们把这个位置看成边，两个数字看成点，那么这种关系形成了若干个环那么如果是只有一个点的连通块，只有一种选择，我们只计算一次答案，如果是多个点的是存在两个选择那么我们就乘以 即可

最小生成树

概念

图是一个二元组 G = (V(G) , E(G))，其中 V(G) 为点的集合，E(G) 为边的集合 对于 V 中的每个元素，我们可以称它为顶点或者节点 一个没有固定根结点的树称为 **无根树**（unrooted tree）。无根树有几种等价的形式化定义：

1. 有 n 个结点，n - 1 条边的连通无向图
2. 无向无环的连通图
3. 任意两个结点之间有且仅有一条简单路径的无向图
4. 任何边均为桥的连通图
5. 没有圈，且在任意不同两点间添加一条边之后所得图含唯一的一个圈的图

在无根树的基础上，指定一个结点称为 根，则形成一棵 **有根树**（rooted tree）。 **生成树**（spanning tree）：一个连通无向图的生成子图，同时要求是树。也即在图的边集中选择条，将所有顶点连通。 那么 **最小生成树** 就是边权和最小的生成树 注意：只有连通图才有生成树，而对于非连通图，只存在生成森林。

Kruskal算法

该算法的基本思想是从小到大加边，是个**贪心算法** 那么显而易见的我们首先对这些若干条边按照边权从小到大排序；然后进行加边操作 那加边怎么加边呢？ 首先我们对于当前**需要加边**的两个点来说，他们一定不属于同一棵树内，他们一定属于两个森林 两个森林相当于两个集合，维护集合我们只需要使用并查集即可 如果两个点属于同一个集合，那么这两个就在同一棵树当中，我们就不需要往里面进行加边

```
#include <iostream>
#include <array>
#include <algorithm>
using namespace std;
const int N = 1e6;
int n, m, f[N];
void INIT() {
    for (int i = 1; i <= n; i++) f[i] = i;
}
int find(int x) {
    return x == f[x] ? x : f[x] = find(f[x]);
}
array<int, 3> edge[N];
int ans, num;
void kru() {
    sort(edge + 1, edge + 1 + m);
    INIT();
    ans = 0;
    num = 0;
    for (int i = 1; i <= m; i++) {
        auto [w, u, v] = edge[i];
        int fu = find(u);
        int fv = find(v);
        if (fu != fv) {
            num++;
            ans += w;
            f[fu] = fv;
        }
    }
}
int main() {
}
```

Prim算法

Prim算法的基本思想是从一个节点开始，不断加点，具体来说就是每次选择一个还没有被使用过的距 离最小的节点然后去用这个节点新的边去更新其他节点的距离 那么这个算法实际上就是把一个点集划分成两个集合，一个集合是已经形成最小生成树的点的集合，另外一个集合是还没有形成最小生成树的点的集合 那么我们在具体写的时候可以先假定 点已经在这个集合中了，然后我们在这个基础上进行如下的操作：

1. 选取距离生成树集合最近的点，加入集合，并且将这个最近的距离加入权值
2. 若找到了 条边，则停止，否则继续进行上述操作

```
A:0 in
B:inf 6 5 5 5 in
C:inf 1 in
D:inf 5 5 2 in
E:inf inf 6 6 6 3 in
F:inf inf 4 in
```

```cpp
template <typename T>
inline bool chkmin(T &a, const T &b) {
    return a > b ? a = b , 1 : 0;
}
#define int long long
const int maxn = 1e2 + 10;
const int inf = 1e18;
int mp[maxn][maxn];
int dis[maxn], vis[maxn], n, m;
void prim() {
    for (int i = 1; i <= n; ++i) {
        dis[i] = inf, vis[i] = 0;
    }
    dis[1] = 0;
    for (int i = 1; i <= n - 1; ++i) {
        int x = 0;
        for (int j = 1; j <= n; ++j) {
            if (!vis[j] && (x == 0 || dis[j] < dis[x])) x = j;
        }
        vis[x] = 1;
        for (int j = 1; j <= n; ++j) {
            if (!vis[j]) {
                chkmin(dis[j], mp[x][j]);
            }
        }
    }
}
```

Kruskal一般用于疏密图上， 如果是稠密图的话可以考虑Prim

搜索 拓扑排序

DFS(深度优先搜索)

该算法讲解时常常与 BFS并列，但两者除了都能遍历图的连通块以外，用途完全不同，很少有能混用两种算法的情况。

DFS常常用来指代用递归函数实现的搜索，但实际上两者并不一样。有关该类搜索思想请参阅 DFS（搜索）.

过程

DFS 最显著的特征在于其 递归调用自身。同时与 BFS 类似，DFS 会对其访问过的点打上访问标记，在遍历图时跳过已打过标记的点，以确保 每个点仅访问一次。符合以上两条规则的函数，便是广义上的 DFS。

模板

伪代码

```
DFS(v) // v 可以是图中的一个顶点，也可以是抽象的概念，如 dp 状态等。
  在 v 上打访问标记
  for u in v 的相邻节点
    if u 没有打过访问标记 then
      DFS(u)
    end
  end
end
```

```
#include<iostream>
using namespace std;
int n , k;
bool vis[9];
string str[9];
int ans , now;
void dfs(int x , int y){
    if(y == 0){//达成条件 退出
        ans++;
        return;
    }
    for(int i = x ; i < n ; i++){
        for(int j = 0; j < n; j++){
            if(str[i][j] == '#' && vis[j] != 1){
                vis[j] = 1;//经过该点 给该点打上标记
                dfs(i + 1 , y - 1);//从该点继续往下遍历
                vis[j] = 0;//撤回标记 方便下次遍历
            }
        }
    }
}
int main(){
    while(cin >> n >> k , n != -1 && k != -1){
        ans = 0;
        for(int i = 0 ; i < n ; i++){
            cin >> str[i];
        }
        dfs(0 , k);
        cout << ans << endl;
    }
}
```

优化

常用的优化有三种， 记忆化搜索 最优性剪枝 可行性剪枝

记忆性剪枝

因为在搜索中 相同的传入值往往会带来相同的结果，那我们就可以用数组来记忆

```
int g[MAXN]; // 定义记忆化数组
int ans = 最坏情况, now;
void dfs f(传入数值) {
    if (g[规模] != 无效数值) return;                    // 或记录解，视情况而定
    if (到达目的地) ans = 从当前解与已有解中选最优; // 输出解，视情况而定
    for (遍历所有可能性)
        if (可行) {
            进行操作;
            dfs(缩小规模);
            撤回操作;
        }
}
int main() {
    // ...
    memset(g, 无效数值, sizeof(g)); // 初始化记忆化数组
                                    // ...
}
```

最优性剪枝

在搜索中导致运行慢的原因还有一种，就是在当前解已经比已有解差时仍然在搜索，那么我们只需要判断一下当前解是否已经差于已有解。

```
int ans = 最坏情况, now;

void dfs(传入数值) {
    if (now比ans的答案还要差) return;
    if (到达目的地) ans = 从当前解与已有解中选最优;
    for (遍历所有可能性)
        if (可行) {
            进行操作;
            dfs(缩小规模);
            撤回操作;
        }
}
```

可行性剪枝

在搜索过程中当前解已经不可用了还继续搜索下去也是运行慢的原因。

```
int ans = 最坏情况, now;

void dfs(传入数值) {
    if (当前解已不可用) return;
    if (到达目的地) ans = 从当前解与已有解中选最优;
    for (遍历所有可能性)
        if (可行) {
            进行操作;
```

```
            dfs(缩小规模);
            撤回操作;
        }
}
```

BFS

所谓宽度优先。就是每次都尝试访问同一层的节点。如果同一层都访问完了，再访问下一层。

这样做的结果是，BFS 算法找到的路径是从起点开始的 最短合法路径。换言之，这条路径所包含的边数最小。

在 BFS 结束时，每个节点都是通过从起点到该点的最短路径访问的。

算法过程可以看做是图上火苗传播的过程：最开始只有起点着火了，在每一时刻，有火的节点都向它相邻的所有节点传播火苗。

```
#include <iostream>
#include <algorithm>
#include <queue>
using namespace std;
int a, b, c, flag;
int X, Y, Z, x2, y2, z2, min1;
int vis[40][40][40], viss[40][40][40];
int u[6][3] = {{1, 0, 0}, {-1, 0, 0}, {0, 1, 0}, {0, -1, 0}, {0, 0, 1}, {0, 0,
-1}};
string str[40][40];
struct node {
    int x, y, z;
};
void dfs(int x, int y, int z, int step) {
    queue<node> que;
    node aa;
    aa.x = x;
    aa.y = y;
    aa.z = z;
    que.push(aa);
    while (!que.empty()) {
        node bb = que.front();
        que.pop();
        for (int i = 0; i < 6; i++) {
            int xx = bb.x + u[i][0];
            int yy = bb.y + u[i][1];
            int zz = bb.z + u[i][2];
            if (xx >= 0 && xx < a && yy >= 0 && yy < b && zz >= 0 && zz < c &&
vis[xx][yy][zz] == 0 && str[xx][yy][zz] == '.') {
                if (xx == x2 && yy == y2 && zz == z2) flag = 0;
                vis[xx][yy][zz] = vis[bb.x][bb.y][bb.z] + 1;
                node aa;
                aa.x = xx;
                aa.y = yy;
                aa.z = zz;
```

```
                que.push(aa);
            }
        }
    }
}
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    while (cin >> a >> b >> c, a != 0 && b != 0 && c != 0) {
        min1 = 0x3f3f3f;
        flag = 1;
        for (int i = 0; i < a; i++) {
            for (int j = 0; j < b; j++) {
                cin >> str[i][j];
                for (int k = 0; k < c; k++) {
                    viss[i][j][k] = 0x3f3f3f3f;
                    vis[i][j][k] = 0;
                    if (str[i][j][k] == 'S') {
                        X = i;
                        Y = j;
                        Z = k;
                    } else if (str[i][j][k] == 'E') {
                        x2 = i;
                        y2 = j;
                        z2 = k;
                        str[i][j][k] = '.';
                    }
                }
            }
        }
        dfs(X, Y, Z, 0);
        if (flag == 0)
            cout << "Escaped in " << vis[x2][y2][z2] << " minute(s)." << endl;
        else
            cout << "Trapped!" << endl;
    }
}
```

拓扑排序

Kahn 算法

过程

初始状态下，集合S装着所有入度为0的点,L是一个空列表。

每次从S中取出一个点u（可以随便取）放入 L,然后将 u 的所有边 (u,vi) 删除。对于边(u,v)若将该边删除后点 v 的入度变为 0，则将 v 放入 S 中。

不断重复以上过程，直到集合S为空。检查图中是否存在任何边，如果有，那么这个图一定有环路，否则返回L,L中顶点的顺序就是拓扑排序的结果。

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
int n, vis[101], ans[101];
vector<int> vec[101];
queue<int> que;
void bfs() {
    int k = 1;
    for (int i = 1; i <= n; i++) {
        if (vis[i] == 0) que.push(i);
    }
    while (!que.empty()) {
        int t = que.front();
        que.pop();
        cout << t << " ";
        for (int i = 0; i < vec[t].size(); i++) {
            vis[vec[t][i]]--;
            if (vis[vec[t][i]] == 0) que.push(vec[t][i]);
        }
    }
}
int main() {
    cin >> n;
    for (int i = 1, num; i <= n; i++) {
        while (cin >> num, num != 0) {
            vec[i].push_back(num);
            vis[num]++;
        }
    }
    bfs();
}
```

#](https://csacademy.com/app/graph_editor/)

最短路

两种存图方式

1、邻接矩阵：

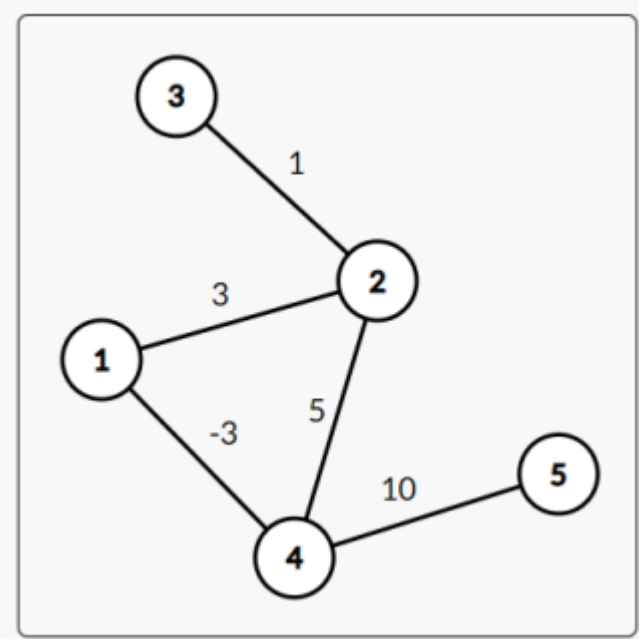
edge[i][j]： i 点与 j 点间的边权， 空间复杂度O(n^2)

2、邻接表：

vector> adj[i]： 保存从 i 出发的每条边的信息

也可以用链式前向星

空间复杂度 O(n+m)



如：这张无向图的邻接表可以表示为

adj[1]	(2, 3)	(4, -3)	
adj[2]	(1, 3)	(3, 1)	(4, 5)
adj[3]	(2, 1)		
adj[4]	(1, -3)	(2, 5)	(5, 10)
adj[5]	(4, 10)		

代码实现

```
vector<pair<int, int>> adj[N];
void add(int u, int v, int w) {
    adj[u].push_back({v, w});
}
```

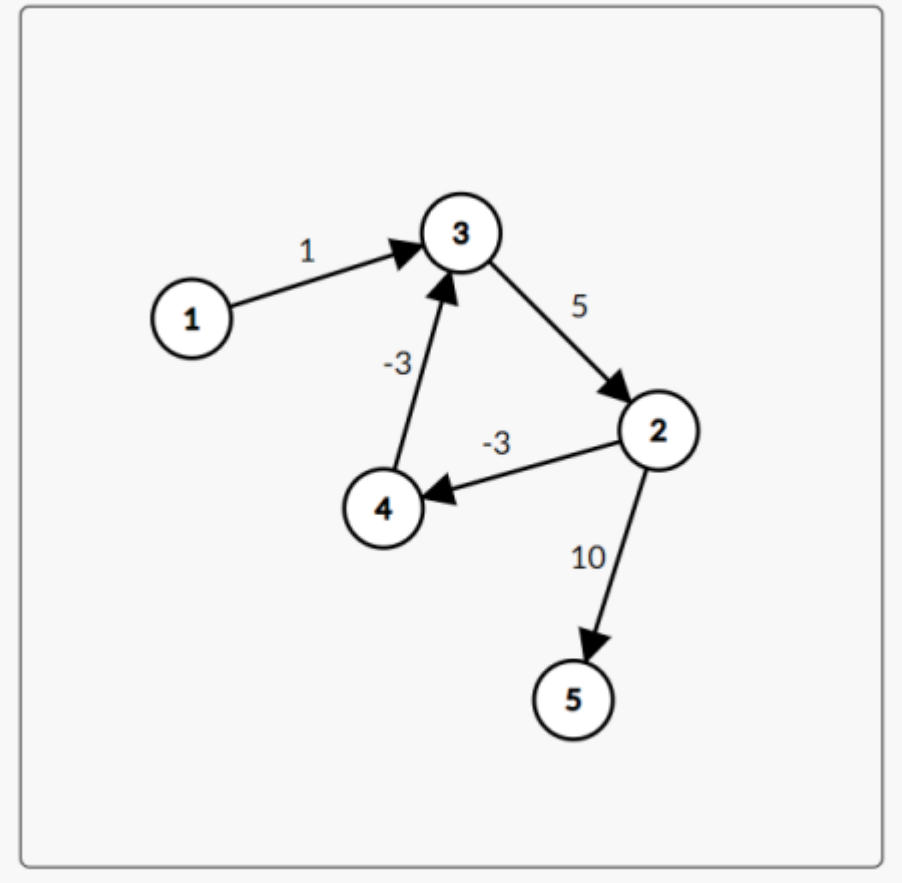
邻接矩阵的优缺点：

优点：访问更快

缺点：占用空间大

最短路存在条件

u, v 间存在最短路 $\iff$ 从 u 出发到 v 的所有路径中，不存在一个负环



单源最短路

源：起点。

指定一个起点， 求出到其他点的最短距离

bellman-ford算法

该算法的流程如下：

d[u]：从起点 st 出发， 到 u 的最短路

1 令 $d[i]= \begin{cases} 0, i = st \ INF, else \end{cases}$

2 扫描每一条边，对于边 (u, v, w), 若 d[v] > d[u] + w ， 则令d[v] = d[u] + w

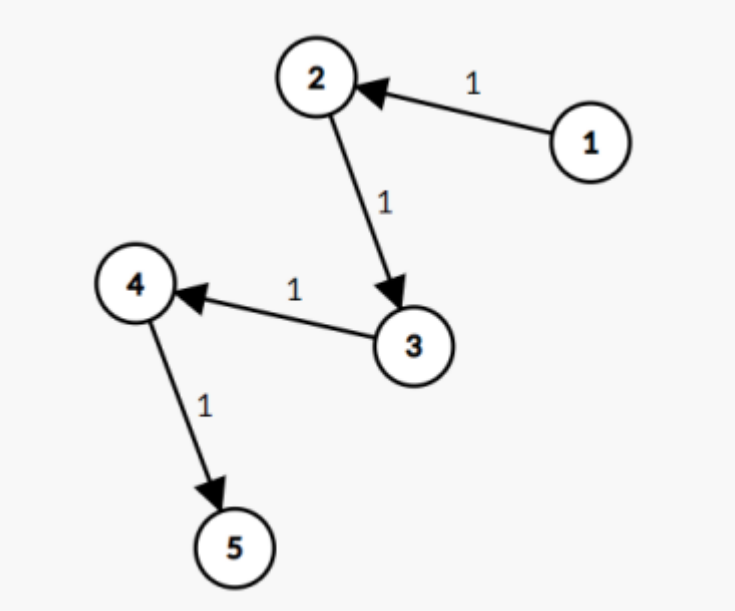
3 重复上述步骤 n-1 次，则可得单源最短路

bellman-ford算法演示

对于一张 n 个点， m 条边的图而言

性质：任意一条最短路最多经过 n 个点

反证法：否则，必然有一点 p 经过两次，则第一次和第二次间的所有点构成一个负环



主代码

```
for (int i = 1; i < n; i++) {
    for (int u = 1; u <= n; u++) {
        for (int j = 0; j < adj[u].size(); j++) {
            int v = adj[u][j].first, w = adj[u][j].second;
            if (d[v] > d[u] + w) {
                d[v] = d[u] + w;
            }
        }
    }
}
```

利用bellman-ford算法判负环

```
for (int i = 1; i <= n; i++) {
    for (int u = 1; u <= n; u++) {
        for (int j = 0; j < adj[u].size(); j++) {
            int v = adj[u][j].first, w = adj[u][j].second;
            if (d[v] > d[u] + w) {
                if (i == n) return true; // 有负环
                d[v] = d[u] + w;
            }
        }
    }
}
return false;
```

时间复杂度 $O(nm)$

spfa

全名 shortest path fast algorithm。

算法流程：

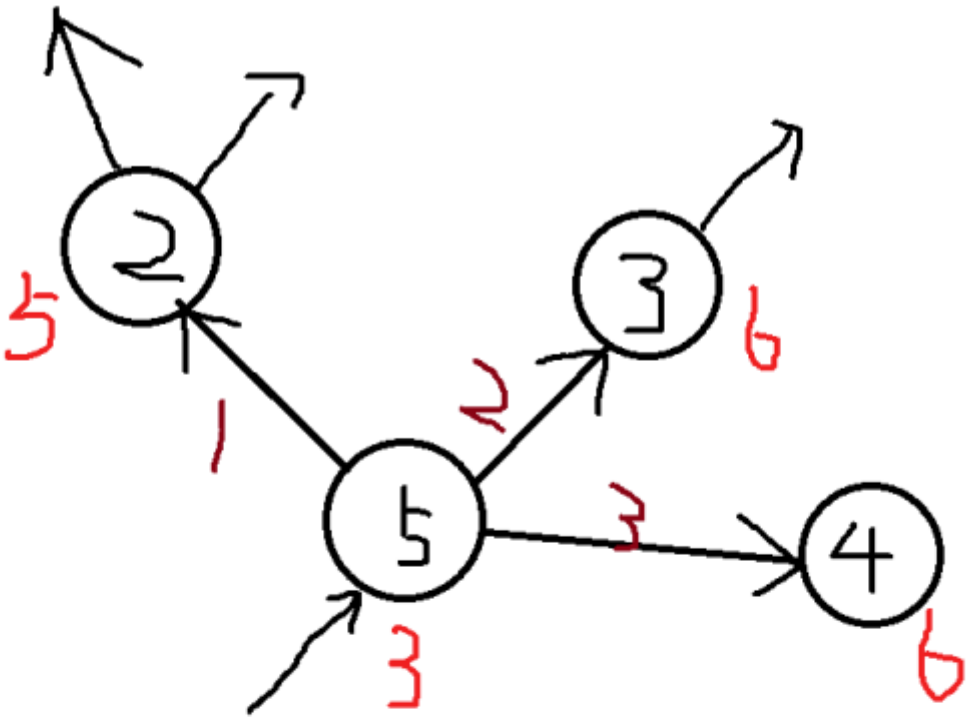
1、建立一个队列，最初队列中只含有起点1。

2、取出队头节点 u ，扫描他的所有出边 (u, v, w) ，若 d[v] > d[u] + w, 则 d[v] = d[u] + w 。同时，若 v 不在队列，则把 v 入队。

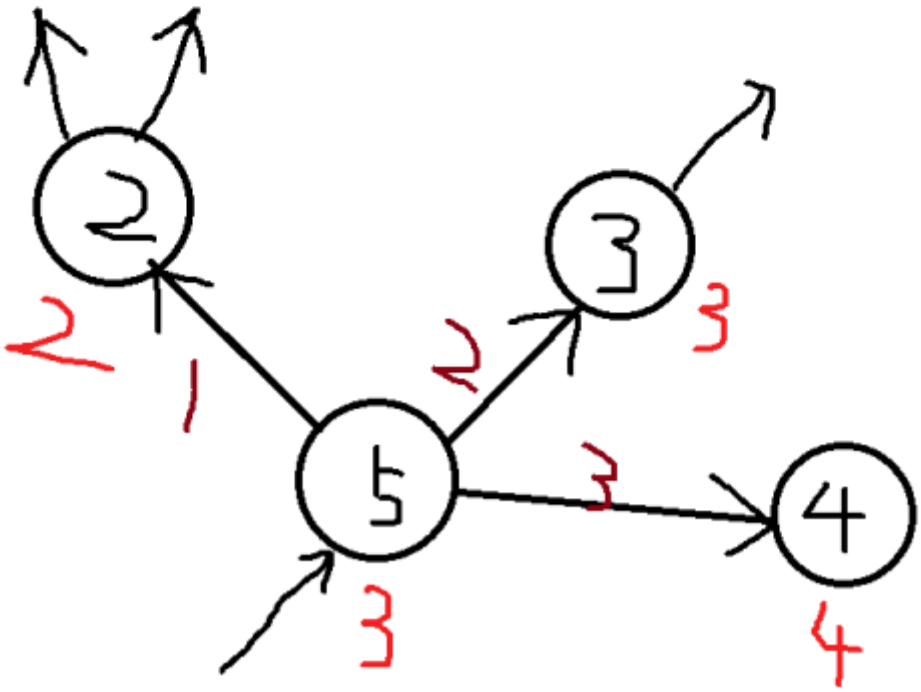
3、重复上述步骤，直到队列为空

解释

遍历从5号点出发的所有边



(5,2)， (5,3)的访问有效， (5, 4)无效



所有访问无效

spfa代码

```
int vis[N], cnt[N]; // cnt表示最短路径所含边数， cnt >= n时有负环
bool spfa(int st) { // 有负环， return true
    memset(cnt, 0, sizeof cnt); //
    for (int i = 1; i <= n; i++)
        d[i] = INF;
    d[st] = 0;
    queue<int> que;
    que.push(st);
    memset(vis, 0, sizeof vis);
    vis[st] = 1;

    while (que.size()) {
        int u = que.front(); que.pop();
        vis[u] = 0;
        for (auto [v, w] : adj[u]) { // 结构化绑定要求c++17标准
            if (d[v] > d[u] + w) {
                d[v] = d[u] + w;
                cnt[v] = cnt[u] + 1;
                if (cnt[v] >= n) return true;
                if (!vis[v]) {
                    que.push(v);
                    vis[v] = 1;
                }
            }
        }
    }
    return false;
}
```

时间复杂度$ O(km)$ 。在随机图上，k是一个较小的常数；但在特殊构造的图上，该算法会退化成$O(nm)$ 。

dijkstra 算法

算法流程：

1、令

$d[i]= \begin{cases} 0, i = st \ INF, else \end{cases}$

2、找出一个未被标记的， d[u] **最小**的节点 u, 然后标记 u

3、扫描 u 的所有出边 (u, v, w)， 若 d[v] > d[u] + w, 则令 d[v] = d[u] + w

4、重复 2~3 两个步骤，直到所有节点都被标记

dijkstra算法演示

性质：在**非负权**图上，对于被标记的节点 u， d[u] 不可能在后续操作中再被更改

朴素dij代码

```
for (int i = 1; i <= n; i++) {
    int u = 0;
    for (int j = 1; j <= n; j++) {
        if (!vis[j] && (u == 0 || d[j] < d[u])) u = j;
    }
    vis[u] = 1;
    for (auto [v, w] : adj[u]) {
        d[v] = min(d[v], d[u] + w);
    }
}
```

时间复杂度$O(n^2 + m)$

找最小值的过程可以用优先队列优化

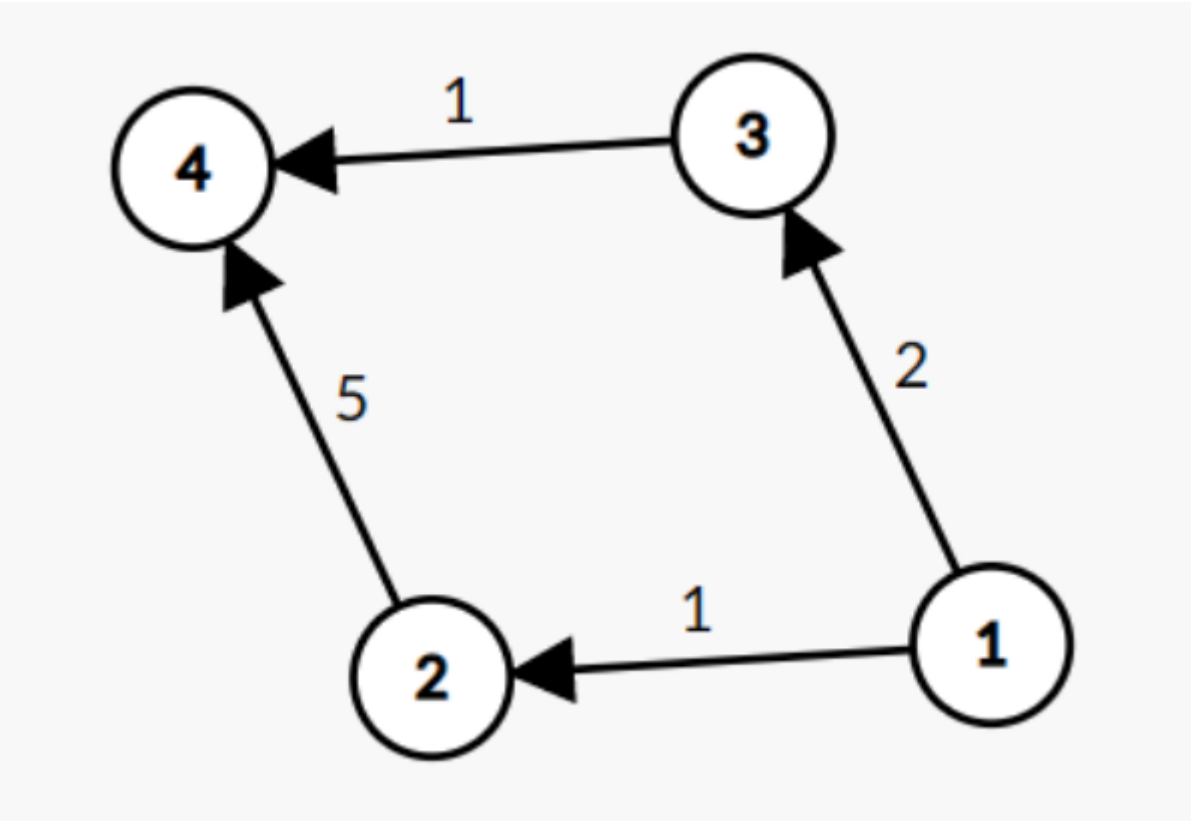
堆优化dij代码

```
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>
que; // 小根堆
que.push({0, st}); // 存pair对 (d[u], u) 。不能反过来，第一维作为排序依据

while (que.size()) {
    int u = que.top().second; que.pop();
    if (vis[u]) continue; // 如果被标记过了，则直接跳过
    vis[u] = 1;
    for (auto [v, w] : adj[u]) {
        if (d[v] > d[u] + w) {
            d[v] = d[u] + w;
            que.push({d[v], v});
        }
    }
}
```

时间复杂度$O(mlogm)$

注意：稠密图上朴素的dij算法更快



对于vis标记的解释

注意：dij只能用于非负权图！！！

任意两点间最短路

floyd算法

设 d[k, u, v] 表示经过若干个编号不超过 k 的节点（不包括u, v）， 从 u 到 v 的最短路

可以列出 dp 方程 $d[k,u,v] = min(d[k-1,u,v], d[k-1,u,k]+d[k-1,k,v]) $ 对于该方程的解释是： u经过前k个点到v，有两种可能

1、u不经过第k个点，而是经过前k-1个到 v

2、u经过前k个点，且第k个点必经过

对于第二种可能，我们从路径中删去k， 则最短路分成u->k, 和 k->v两段，且中间经过的点必然 < k

另外可以发现，dp方程第一维是可以优化的 （Floyd的题点的个数一般为300~500, 不优化空间会炸掉）

floyd代码

```
for (int k = 1; k <= n; k++) {
    for (int u = 1; u <= n; u++) {
        for (int v = 1; v <= n; v++) {
            d[u][v] = min(d[u][v], d[u][k] + d[k][v]);
        }
    }
}
```

时间复杂度 $O(n^3)$

floyd计算传递闭包

传递性：对于定义在集合 S 上的二元关系 I, 若对于 \forall a, b, c \in S, 只要有 a Ib 且 bIc ， 则有aIc。

例如：图论中的**可达**关系

```
for (int k = 1; k <= n; k++) {
   for (int u = 1; u <= n; u++) {
       for (int v = 1; v <= n; v++) {
           d[u][v] |= d[u][k] & d[k][v];
       }
   }
}
```

特殊的最短路

有向无环图 -> 拓扑排序 O(n)

边权为 1 的图 -> bfs O(n)

边权为 0 或 1 的图 -> 双端队列bfs O(n)

总结

做最短路题时，要根据数据范围选择合适的方法

对于一张 n 个点， m 条边的图而言（图论一般题 $n, m \le 1e5$ ）

floyd 复杂度为 $O(n^3)$ ， $n \le500$ 可以考虑， $n > 500$ 一般不考虑

dijkstra 复杂度稳定在$O(mlogm)$ ， 几乎可以适用所有权值≥ 0 的图

spfa 复杂度 $O(km)$ ， 按复杂度上界算会TLE， 但也有概率能过题。对于负权图，实在没办法的时候可以试试。

例题

acwing 342. 道路与航线 - AcWing题库)

工具

Graph Editor (csacademy.com)

动态规划（Dynamic Programming, DP）

DP大致简述

能用动态规划解决的问题，需要满足三个条件：最优子结构，无后效性和子问题重叠。

最优子结构，举个例子，最短路,下图边权均为1

如图:q->t的最短路满足最优子结构，他由q->s的最短路转移过来

但是最长路并不是最优子结构，例如q->t的最长路实际上是由q->s的最短路转移过来的，不满足最优子结构，因为q->s的最长路显然是q->r->t->s

DP用图论的概念我们可以抽象成一张有向无环图(DAG)，边相当于决策方案，求解的过程相当于在做一个拓扑排序

无后效性，就是已经求解的子问题，不会再受到后续决策的影响，从DAG中的理解就是无环，例如状态a可以推出b，b可以推出c，c可以推出a，这个就不满足无后效性,因为后续的决策c会对a产生影响

子问题重叠性,大量的重叠子问题，我们可以用空间将这些子问题的解存储下来，避免重复求解相同的子问题，从而提升效率。

DP能做什么

dp能写的东西其实很多，一些贪心的题目，你用DP也能够去解决，因为某种意义上dp相当于做了最优的决策

然后还有一大类DP的题目，比较常见的有LCS(最长公共子序列),LIS(最长上升子序列),LCP(最长公共子串),记忆化搜索，背包DP，区间DP，树形DP，状压DP，数位DP，计数DP，概率/期望DP等，涉及范围较广。

除此之外还有一些DP的优化算法，本质是利用DP函数的单调性等优化决策转移的速度

比赛中出现的频率也是很高的那种，签到到牌区的题都有。

今天我们介绍下几个简单的例子入门LCS,LIS,LCP,背包DP

LCS,LIS,LCP

首先区分下两个概念，子序列和子串。举个例子，例如一个字符串abcdefghijk

我们从头和尾部删除一部分，留下中间连续的一部分内容，这部分被称之为子串

然后我们按照顺序依次拿出 p_1

LCS(最长公共子序列)

```
abcfbc
abfcab
```

按照这个例子，我们可以发现最长公共子序列是abfc

我们可以设计DP方程dp[i][j]代表第一个字符串的[1,i]的子串和第二个字符串[1,j]的子串的LCS，当$s_1[i]$和$s_2[j]$位置相同时，我们可以产生一个贡献，他由dp[i-1][j-1]转移，如果两者不同，我们要从最大的那个转移，就是$max\left{dp[i-1][j],dp[i][j-1]\right}$,例如dp[3][2]这个状态，由于$s_1[3]$和$s_2[2]$不同，我们要么选择$s_1$少一个或者$s_2$少一个的状态转移。

$$ dp[i][j]=\begin{cases} dp[i-1][j-1]+1，s_1[i]=s_2[j] \ \max\left{dp[i-1][j],dp[i][j-1]\right}， s_1[i]\neq s_2[j]\end{cases} $$

初始态显然是dp[0][0]=0

LCP(最长公共子串)

子串和子序列的区别实际上就是必须要连续，所以在$s_1[i]\neq s_2[j]$时，贡献会被清零，我们定义dp[i][j]代表第一个字符串的[1,i]的子串和第二个字符串[1,j]的子串的LCP

$dp[i][j]=\begin{cases} dp[i-1][j-1]+1，s_1[i]=s_2[j] \ 0， s_1[i]\neq s_2[j]\end{cases}$

从转移方程可以看出，dp[n][m]$不一定是最优解，所以要对全部的状态取

$max(dp[i][j] 1<= i <= n , 1 <= j <= m)$

LIS(最长上升子序列)

我们定义dp[i]为已i结尾的LCS，那么我们dp[i]如果要从dp[j]转移的话，首先要满足上升子序列的这一个性质(基于题目，可以取到等于，看是否是严格递增)，a_i>a_j满足上升这一个性质，然后此时能够转移的状态应该有很多位置满足这一个性质，这时候要让当前状态最优，就要取一个最大的dp[j]进行转移，所以我们需要用O(n)的时间便利数组实现转移

$dp[i] = max(dp[i][j] (1 <= j <= i - 1) (ai > aj) + 1$

背包DP

背包DP有三种比较简单的模型

01背包

一般的模型就是有n个物品，每个物品重w_i，有v_i的价值，你有一个背包容量为V，求最大价值，每个物品只有取或者不取两种状态

我们一般会设置dp[i][j]为当前考虑到第i件物品，背包容量为j的最大价值，那么对于第i个物品我们有两种决策方式，第一种取，那么如果要取第i件物品，第i-1的状态时，他的体积必须是j-w_i才能够放下这件物品，并且我们获得了v_i的额外价值，如果不取，那么直接从dp[i-1][j]转移即可，然后对于两种决策我们要取到一个max，达到最优决策，所以dp方程就比较容易理解为

$dp[i][j]=max\left{dp[i-1][j],dp[i-1][j-w_i]+v_i \right}$

然后讲个比较简单的优化，也是常用到的，假如n=10000，V=20000,那么此双重循环时间实际上是够的，但是2e8的空间一般是开不下的，然后我们发现上述的dp方程，其实第一维并不是很重要，我们可以有两种方式把他优化掉

第一种，直接舍弃掉他，然后再做循环枚举的时候要注意顺序，假如我们从小开始枚举，我们把前面的数组更新了就会对后面的产生影响，所以要从后往前更新，因为这个式子大的体积的答案并不会影响小的

第二种，我们可以使用滚动数组优化，因为当前状态只和前一个状态有关，更早之前的其实并不影响，所以只需要记录前一个数组的信息即可

完全背包

完全背包和01背包的本质区别就是，每个物品是无限的

我们设计dp[i]代表容量为i时的最大价值，那么当前如果选择放置重w_j，j价值为v_j的物品，那么就要从dp[i-w[j]]转移过来，显然，这里有一个条件是w[i]\leq i，然后此时你可以选择放置n个物品中的任意一个，你有n种决策方式，从中选择最优解即可

所以，$dp[i]=max(dp[i-w[j]]+v[j]|(1 \leq j \leq n) (w[j] \leq i))$

多重背包

现在给你一个容量为V的背包，有N个物品，其中第i件物品的重量为w_i，价值为v_i，第i件物品一共有s_i件，问在有限的容量内，最多可以拿到多少价值的物品。这是一个比较典型的多重背包例子，和01背包的区别就是每个物品多了一个件数的条件。

一般来说他会把\sum s_i的总和开的很大，如果你按照01背包的做法时间就会变成$O(V*\sum s_i)$会被卡tle

二进制拆分优化

假如当前物品有s_i件，我们可以根据二进制可以表示任意数字的思想，将其捆绑成整体做01背包

特殊地，若s_i+1不是2的整数次幂，则需要在最后添加一个由$s_i-2^{\lfloor log_2(s_i+1) \rfloor-1}$个捆绑的物品

举个例子

7=1+2+4这个就是刚好的

6=1+2+3这个最后补4就多了，所以吧剩余的3补上

18=1+2+4+8+3

31=1+2+4+8+16

```
index = 0;
for (int i = 1; i <= m; i++) {
  int c = 1, p, h, k;
  cin >> p >> h >> k;
  while (k > c) {
    k -= c;
    list[++index].w = c * p;
    list[index].v = c * h;
    c *= 2;
  }
  list[++index].w = p * k;
  list[index].v = h * k;
}
```

在二进制拆分之后你只需要把拆分后的数组做01背包即可，总的时间复杂度被优化到了$O(V*\sum log(s_i))$

DP路径记录

dp的路径记录一般是利用前驱数组记录的，因为一个状态可能会转移给很多状态，但是由于最优决策，只有一个最优的前驱会向当前状态转移，所以我们只需要在转移时记录其前驱即可，然后用递归的方式还原路径

```
int pre[N];//前驱数组
vector<int>ansid;//路径记录
void getans(int pos){//递归记录路径
    if(pre[pos])getans(pre[pos]);
    ansid.push_back(pos);
    return;
}
```

LCA(最近公共祖先)

离线

先把所有询问读进来，每个节点开一个 vector < int >，记录这是第 几个询问。同时开一个 ans 数组，ans[i] 代表第 i 次询问的答案 从根节点进行一次 dfs，设一个变量，遇到特殊节点之后就将变量 的值改成当前最近遇到的特殊节点，dfs 出来时再变回原来的版本 遍历每个节点的 vector，将最近一次遇到的特殊节点的编号赋值给对应的 ans[i] 遍历 ans 数组进行输出 时间复杂度 O(n + q)

受上述算法启发，我们是不是也能先按照本身的性质，对询问进行 回答，最后按序输出呢 我们对 LCA 进行一个变换，两点的 LCA 也可以看成找一个深度最 大的节点，使得询问的两点均在以该节点为根的子树中 我们对 LCA 进行分类讨论，即 LCA 为其中一节点的情况（换句话 说，其中一个节点为另一个节点的祖先节点），或 LCA 为另外一节 点的情况 不知道会先遍历到询问的哪个点，所以询问要挂在两个点上。对于 第一种情况，只需要标记另外一个节点是否已经被遍历过就行了 对于第二种情况，也要对遍历过的节点打标记，同时发现需要把先 遍历过的点往上提，提到最近一个还没有退出 dfs 的节点上来。这 里用并查集实现就很方便了，注意合并方向 时间复杂度 O(n + q · α(n))

```
#include <iostream>
#include <vector>
using namespace std;
const int N = 5e5 + 10;
vector<int> vec[N];
vector<pair<int, int>> Q[N];
int f[N], vis[N], ans[N];
int n, m, s;
void INIT() {
    for (int i = 1; i <= n; i++) f[i] = i;
}
int find(int x) {
    return f[x] == x ? x : f[x] = find(f[x]);
}
void merge(int x, int y) {
    int xx = find(x);
    int yy = find(y);
    if (xx == yy) return;
    f[yy] = xx;
```

```
}
void dfs(int u, int fa) {
    vis[u] = 1;
    for (auto [qid, v] : Q[u]) {
        if (!vis[v]) continue;
        ans[qid] = find(v);
    }
    for (auto v : vec[u]) {
        if (v == fa) continue;
        dfs(v, u);
        merge(u, v);
    }
}
int main() {
    cin >> n >> m >> s;
    INIT();
    for (int i = 1, u, v; i < n; i++) {
        cin >> u >> v;
        vec[u].push_back(v);
        vec[v].push_back(u);
    }
    for (int i = 1, u, v; i <= m; i++) {
        cin >> u >> v;
        Q[u].push_back({i, v});
        Q[v].push_back({i, u});
    }
    dfs(s, 0);
    for (int i = 1; i <= m; i++) {
        cout << ans[i] << endl;
    }
}
```

在线倍增

如果需要按照顺序来回答，我们可以采用倍增的方法来实现 对于 LCA 为其中一点的情况，只需要用深度较大的节点往上跳， 找到其祖先节点中，深度最小的节点，满足其深度不小于询问的另 外一个节点。此时答案就是该节点 对于 LCA 为另外一节点的情况，我们参考第一种情况的处理方式， 此时两个节点的深度相同，我们需要两个节点同时往上跳，找到第 一次相遇的节点。但倍增只能找到符合或者不符合条件的最后一个 节点，所以我们在跳的过程中找最后一对不相等的节点，该节点的 父亲节点就是我们要的 LCA 时间复杂度 $O((n + q) \cdot \log(n))$

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
typedef long long ll;
const int N = 5e5 + 7;
int n, m, rt;
int par[N][20], dep[N];
vector<int> G[N];
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
```

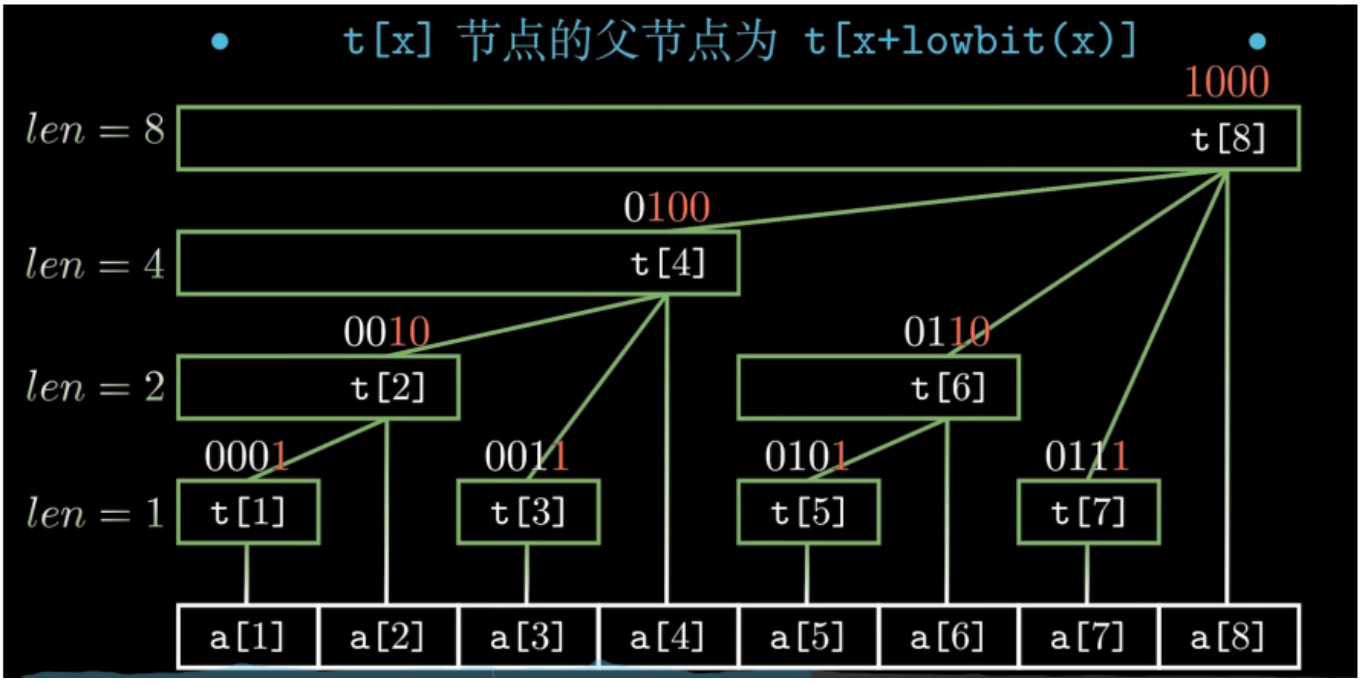
```
    par[u][0] = fa;
    for (int i = 1; i < 20; ++i) {
        par[u][i] = par[par[u][i - 1]][i - 1];
    }
    for (auto &v : G[u]) {
        if (v == fa) continue;
        dfs(v, u);
    }
}
int getLca(int u, int v) {
    if (dep[u] < dep[v]) swap(u, v);
    for (int i = 19; i >= 0; --i) {
        if (dep[par[u][i]] >= dep[v]) u = par[u][i];
    }
    if (u == v) return u;
    for (int i = 19; i >= 0; --i) {
        if (par[u][i] != par[v][i]) {
            u = par[u][i], v = par[v][i];
        }
    }
    return par[u][0];
}
void solve() {
    cin >> n >> m >> rt;
    for (int i = 1, u, v; i < n; ++i) {
        cin >> u >> v;
        G[u].emplace_back(v);
        G[v].emplace_back(u);
    }
    dfs(rt, 0);
    for (int i = 1, u, v; i <= m; ++i) {
        cin >> u >> v;
        cout << getLca(u, v) << '\n';
    }
}
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
    cout << fixed << setprecision(20);
    int t = 1;
    while (t--) solve();
    return 0;
}
```

树状数组

lowbit

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

计算在二进制下最后一个1对应的数



1. t数组就是所谓的树状数组
2. 每个节点$t[x]$表示以x为根的子树中叶节点值的和
3. 把每个节点$t[x]$中的下标x转化为二进制的形式，我们会发现每一层的末尾的 0 的数量是相同的（也就是说同一层的$lowbit(x)$返回值相同）
4. $t[x]$表示的是$\sum^{x}_{i=x-lowbit(x)+1}{a_i}$，它的父节点是$t[x+lowbit(x)]$
5. 整棵树的深度为$log(n) + 1$

操作

修改

```
void add(int pos , int val){
    while(pos <= n){
        c[pos] += val;
        pos += lowbit(pos);
    }
}
```

前缀求和

```
int getSum(int x){
    int ans = 0;
    while(x >= 1){
        ans += c[x];
        x -= lowbit(x;)
    }
}
```

区间求和

```
int query(int l , int r){
    return getSum(r) - getSum(l - 1);
}
```

区间修改

$\sum^{r}_{i = 1}{a_i}$

$=\sum^{r}*{i=1}\sum^{i}*{j=1}{b_j}$

$=\sum^{r}_{i=1}{b_i}*(r-i+1)$

$=(r+1)\sum^{r}*{i=1}{b_i}-\sum^{r}*{i=1}{b_i*i}$

```
#include <iostream>
using namespace std;
const int N = 1e6;
long long t1[N], t2[N];
int n, q;
long long lowbit(long long x) {
    return x & -x;
}
void add(int x, long long val) {
    long long val1 = val;
    long long val2 = val * x;
    while (x <= n) {
        t1[x] += val1;
        t2[x] += val2;
        x += lowbit(x);
    }
}
long long getsum(int x) {
    long long sum1 = 0;
    long long sum2 = 0;
    int xx = x;
    while (x > 0) {
        sum1 += t1[x];
        sum2 += t2[x];
```

```
            x -= lowbit(x);
        }
        return sum1 * (xx + 1) - sum2;
    }
    int main() {
        cin >> n >> q;
        long long pre = 0;
        for (int i = 1; i <= n; i++) {
            long long num;
            cin >> num;
            add(i, num - pre);
            pre = num;
        }
        while (q--) {
            int k, l, r;
            cin >> k >> l >> r;
            if (k == 1) {
                int val;
                cin >> val;
                add(l, val);
                add(r + 1, -val);
            } else {
                cout << getsum(r) - getsum(l - 1) << endl;
            }
        }
    }
```

线段树

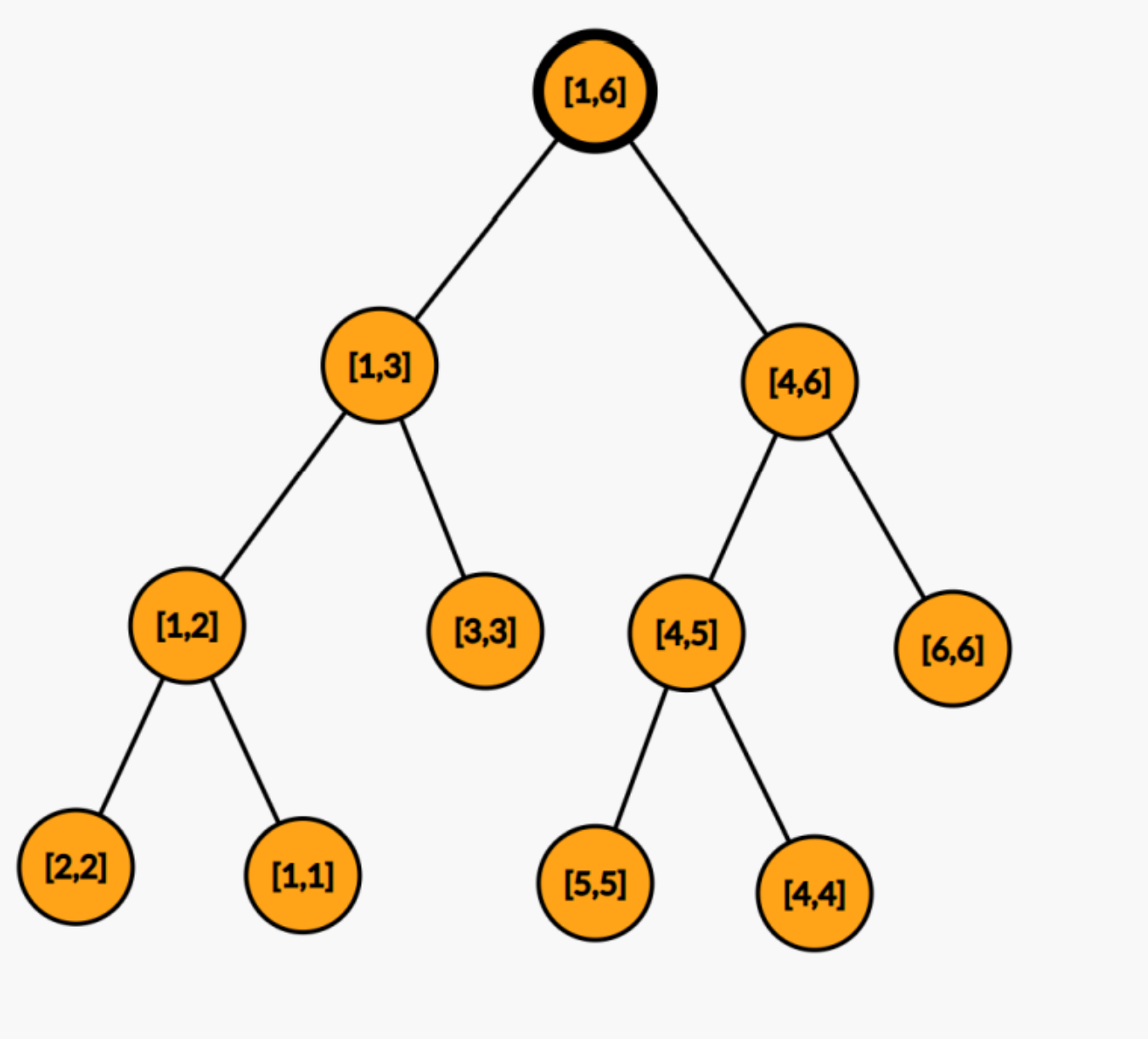
线段树是一种在算法竞赛中非常常见的数据结构。它是一种基于分治思想的 **二叉树** 结构，常用于在区间上进行信息统计。并且跟 **树状数组** 相比，其适用 **场景更多** ， 但 **码量较大** ， **常数** 也 **较大。**

线段树的每个节点都代表了序列上的一个区间。

特别的，根节点代表总区间，如$[1 , N]$，叶子节点代表某个长度为1的特定位置，如$[x ,x]$。

线段树上除 **叶子节点** 以外的每个节点都有 **有且只有两个** 儿子： 左儿子和右儿子。

如果这个节点代表的区间是$[l,r]$，那么它的左儿子代表的区间是$[l,\lfloor\frac{l+r}{2}\rfloor]$，右儿子代表的区间$[\lfloor \frac{l+r}{2}+1\rfloor,r]$。举例来说，区间$[1,6]$构造的线段树如下图所示：

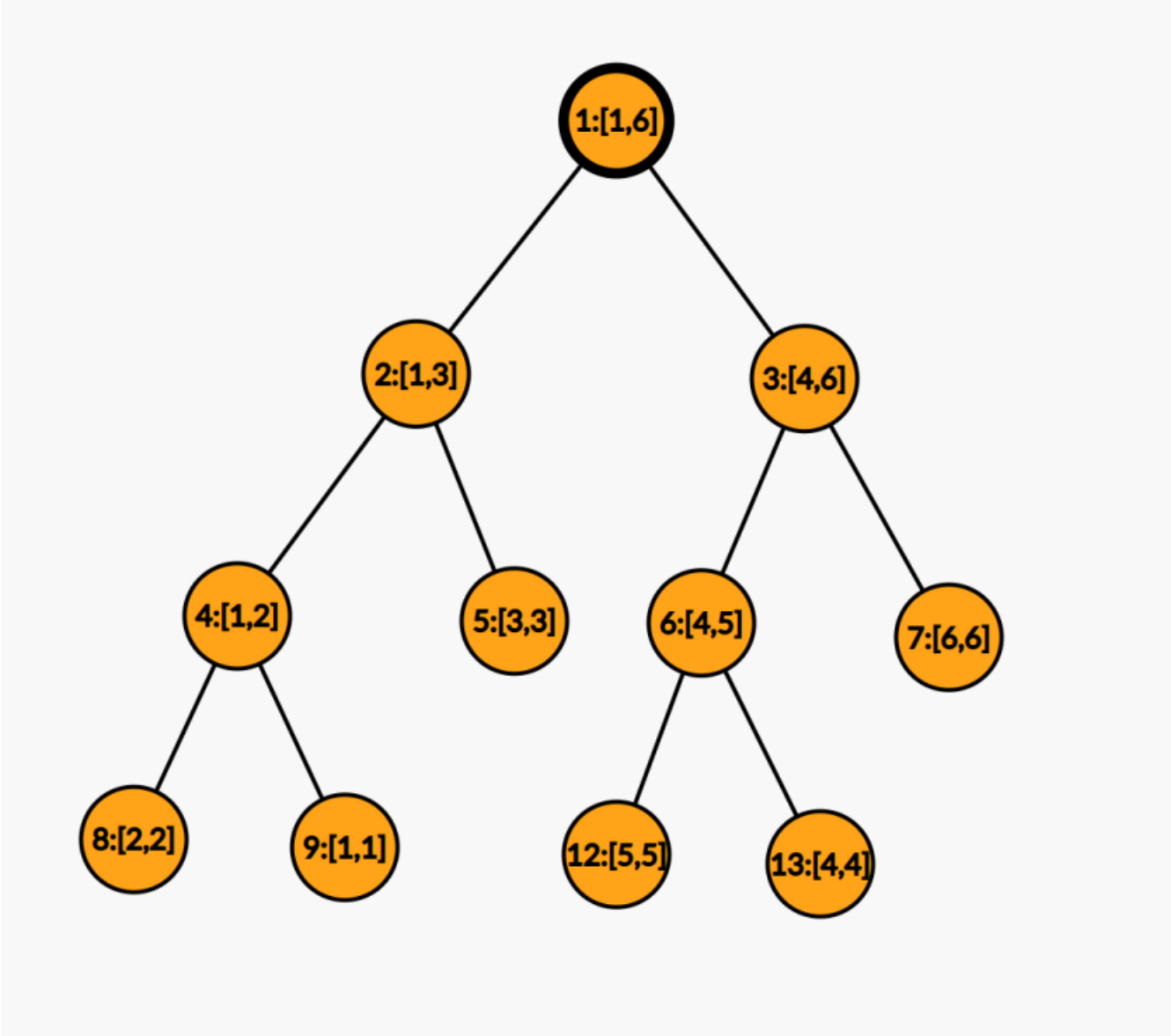


注意：上面的数值代表的是在数组当中对应的下标。

因为线段树除了叶子节点外，其他节点都 **有且只有两个** 儿子，所以我们在存储的时候不使用邻接表或者邻接矩阵。我们直接对各个节点进行编号。具体的编号规则如下：

1. 根节点编号为1.
2. 编号为x的结点的左节点编号为$2\times x$ ，右节点编号为 $2\times x+1$。

这样一来，我们就能简单地使用一个$struct$数组来保存线段树。当然，树的最后一层在数组中保存的位置是不连续的，我们直接空出那些位置就可以。上面的例子编号之后:



在理想情况下， N个叶节点的满二叉树有 $N + \frac{N}{2} +\frac{N}{4}+...+2+1=2N-1$ 个节点。因为在上述存储方式下，最后一行产生了空余，所以保存线段树的 **数组长度要不小于$4N$才保证不会越界。**

显然，树的高度是$O(logn)$的。

建树

```
#include <iostream>
using namespace std;
const int N = 1e5;
int n, m, a[N];
struct node {
    int l, r;
    int sum;
} seg[N << 2];
void up(int id) {
    seg[id].sum = seg[id << 1].sum + seg[id << 1 | 1].sum;
}
void build(int id, int l, int r) {
    seg[id].l = l;
```

```
    seg[id].r = r;
    if (l == r) {
        seg[id].sum = a[l];
        return;
    }
    int mid = (l + r) >> 1;
    build(id << 1, l, mid);
    build(id << 1 | 1, mid + 1, r);
    up(id);
}
```

单点修改

```
void change(int id, int pos, int val) {
    int l = seg[id].l;
    int r = seg[id].r;
    if (l == r) {
        seg[id].sum += val;
        return;
    }
    int mid = (l + r) >> 1;
    if (pos <= mid)
        change(id << 1, pos, val);
    else
        change(id << 1 | 1, pos, val);
    up(id);
}
```

区间询问

```
int query(int id, int ql, int qr) {
    int l = seg[id].l;
    int r = seg[id].r;
    if (ql > r || qr < l) return 0;
    if (ql <= l && r <= qr) return seg[id].sum;
    return query(id << 1, ql, qr) + query(id << 1 | 1, ql, qr);
}


int query(int id, int ql, int qr) {
    int l = seg[id].l;
    int r = seg[id].r;
    if (ql <= l && r <= qr) return seg[id].sum;
    int ans = 0;
    int mid = (l + r) >> 1;
    if (ql <= mid) ans += query(id << 1, ql, qr);
    if (qr > mid) ans += query(id << 1 | 1, ql, qr);
    return ans;
}
```

```
int query(int id, int ql, int qr) {
    int l = seg[id].l;
    int r = seg[id].r;
    if (ql <= l && r <= qr) {
        return seg[id].sum;
    }
    int mid = (l + r) >> 1;
    if (qr <= mid) {
        return query(id << 1, ql, qr);
    } else if (ql > mid) {
        return query(id << 1 | 1, ql, qr);
    } else {
        return query(id << 1, ql, qr) + query(id << 1 | 1, ql, qr);
    }
}
query(1, ql, qr) // 调用入口
```

区间修改

对于区间修改，我们自然不可能用单点修改的方法暴力修改每一个区间，这个时候强大而伟大的 **懒标记** 登场了！ 简单来说，就是要对一个区间做修改，我们可以先把对这个区间的修改暂存在某一个祖先节点上。在需要的时候，再将影响落实到其子孙节点。 考虑要实现操作 ，我们在线段树的每个节点 i 上再增加一个变量 A_i，表示给节点 i 所代表的区间中的所有数都增加了 A_i 。

现在有一个操作 $3 $，需要给$[L,R]$ 区间内的每个数都加上 x 。我们还是跟区间询问一样，先在线段树上定位这个区间，然后给这个区间打上懒标记 x，具体来讲就是 A_i 把 x 加上 并更新这个节点的区间和（加上区间大小乘上 ），然后 向上更新它的所有父节点的信息。 不难发现，对于线段树中的任意一点，只有当它的所有祖先节点的标记都是 0 的时候，它所记录的信息才是正确的（只要有一个祖先节点的标记不为 ，说明某一次的区间修改的影响只是暂存在了某个祖先节点上，还没有落实到儿子上）。因此无论是修改操作还是询问操作，访问到一个节点的时候，都应该将这个节点的标记下传到它的儿子中。给第 i 个节点进行标记下传相当于给它的左儿子和右儿子打上懒标记 A_i ，并把 A_i 清零。 时间复杂度和区间询问一样，是 $O(logn)$ 的。

```
struct node {
    int l, r;
    int sum, lazy;
} seg[N << 2];
// 对标号为 id 的节点打上 tag 的懒标记
void settag(int id, int tag) {
    // 先更新节点维护的信息
    seg[id].sum += (seg[id].r - seg[id].l + 1) * tag;
    // 再把原来的懒标记和新增的懒标记合并
    seg[id].lazy += tag;
}
// 下放懒标记
void down(int id) {
    // 本身没有懒标记 直接返回
    if (seg[id].lazy == 0) return;
```

```
    // 对左右儿子分别打上与自己相同的标记
    settag(id << 1, seg[id].lazy);
    settag(id << 1 | 1, seg[id].lazy);
    // 将自己的标记清除
    seg[id].lazy = 0;
}
void modify(int id, int ql, int qr, int val) {
    int l = seg[id].l;
    int r = seg[id].r;
    if (ql > r || qr < l) return;
    // 到达包含的区间，直接在该节点上打上懒标记
    if (ql <= l && r <= qr) {
        settag(id, val);
        return;
    }
    // 定位区间过程中，对遇到的节点的懒标记都要先下放
    down(id);
    int mid = (l + r) >> 1;
    // 继续定位左右儿子
    modify(id << 1, ql, qr, val);
    modify(id << 1 | 1, ql, qr, val);
    // 更新完所有底层信息后，从下往上再更新其他祖先的信息
    up(id);
}
modify(1, ql, qr, val) // 调用入口
// 注意：  若要支持区间修改 则区间询问的函数要改成这样：
int query(int id, int ql, int qr) {
    int l = seg[id].l;
    int r = seg[id].r;
    if (ql > r || qr < l) return 0;
    if (ql <= l && r <= qr) return seg[id].sum;
    // 定位区间过程中，对遇到的节点的懒标记都要先下放
    down(id);
    return query(id << 1, ql, qr) + query(id << 1 | 1, ql, qr);
}
```

数论

最大公约数

```
int gcd(int a , int b){
    return b == 0 ? a : gec(b , a % b);
}
__gcd();
gcd(a , b) * lcm(a , b) = a * b
```

求 $ax + by = gcd(a , b)$ 的一组可行解

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

埃氏筛和欧拉筛

```
埃氏筛
const int maxn = 10000000;
bool not_prime[maxn + 5];
int prime[maxn + 5], tot = 0;
void get_prime(int n) {
    for (int i = 2; i <= n; i++) {
        if (!not_prime[i]) {
            prime[++tot] = i;
            for (int j = 2 * i; j <= n; j += i) {
                not_prime[j] = 1;
            }
        }
    }
}
```

```
欧拉筛
const int maxn = 10000000;
bool not_prime[maxn + 5];
int prime[maxn + 5], tot = 0;
void get_prime(int n) {
    for (int i = 2; i <= n; i++) {
        if (!not_prime[i]) prime[++tot] = i;
        for (int j = 1; j <= tot && i * prime[j] <= n; j++) {
            not_prime[i * prime[j]] = 1;
            if (i % prime[j] == 0) break;
        }
    }
}
```

分解质因数

求n的所有因子

```
int low[N];
void getPrime() // low数组记录n的最大质因子,low[i]=i为质数。
{
    for (int i = 2; i < N; i++) {
        low[1] = 1;
        if (!low[i]) {
            for (int j = i; j < N; j += i) {
                low[j] = i;
            }
        }
    }
}
int cnt, p;
while (n != 1) {
    cnt = 0, p = low[n];
    while (n % p == 0) {
        ++cnt, n /= p;
    }
}
```

KMP Hash

KMP

求一个字符串在长度为k的子字符串内最长的前后相同子串的长度

abdab 在5的长度中为2

```
void get_Next() {
    Next[0] = -1;
    int j = 0, k = -1;
    while (j < s2.length()) {
        if (k == -1 || s2[j] == s2[k])
            Next[++j] = ++k;
        else
            k = Next[k];
    }
}
```

Hash

将一个字符串转换为不一样的hash数

```
一维hash
unsigned long long Hash1[N], Hash2[N], xp[N];
void Init() {
```

```
    xp[0] = 1;
    for (int i = 1; i < N; i++) {
        xp[i] = xp[i - 1] * Ha;
    }
}
//计算整串的Hash
void make_Hash(string s, unsigned long long Hash[]) {
    int len = s.length();
    Hash[len] = 0;
    for (int i = len - 1; i >= 0; i--) Hash[i] = (Hash[i + 1] * Ha + (s[i] - 'A' +
1));
}
//得到n到n+len的Hash
int get_Hash(int n, int len, unsigned long long Hash[]) {
    return (Hash[n] - Hash[n + len] * xp[len]);
}
```

二维

```
#include <iostream>
#include <cstring>
using namespace std;
typedef long long ll;
const int N = 1e6 + 10;
const int mod1 = 3145739;
const int mod2 = 6291469;
struct Hash {
    ll has1[N], has2[N];
    ll bas1[N], bas2[N];
    ll p1, p2;
    void init(char s[]) {
        p1 = 2333;
        p2 = 17;
        has1[0] = has2[0] = 0;
        bas1[0] = bas2[0] = 1;
        int len = strlen(s + 1);
        for (int i = 1; i <= len; ++i) {
            bas1[i] = (bas1[i - 1] * p1) % mod1;
            bas2[i] = (bas2[i - 1] * p2) % mod2;
        }
        for (int i = 1; i <= len; ++i) {
            has1[i] = (has1[i - 1] * p1 + s[i]) % mod1;
            has2[i] = (has2[i - 1] * p2 + s[i]) % mod2;
        }
    }
    ll gethash1(int r, int len) {
        return ((has1[r] - has1[r - len] * bas1[len]) % mod1 + mod1) % mod1;
    }
    ll gethash2(int r, int len) {
        return ((has2[r] - has2[r - len] * bas2[len]) % mod2 + mod2) % mod2;
    }
};
```

树形DP

树的直径

定义：树上两点之间最远的距离 我们考虑用树形 求解，$dp[u]$ 代表以 u 点可以到达他子树中的最远距离，在 dp 中，我们先递归获取了 u 点中以 v_i 的子树信息，对于经过点 u 和 v 的最大距离，肯定是 $dp[u] + dp[v] + edge[u , v]$ ，对全部的这个取 max 就是直径了，同时，我们也需要向上合并 dp 信息，对于$dp[u]$ 来说，他的最大值一定是在本身 $dp[u]$ 和 $dp[v] + edge[u , v]$ 中取 max 所以只要这样维护即可

```
int ans = 0;
vector<pair<int, int>> G[N];
void dfs(int u, int fa) {
    for (auto &[v, w] : G[u])
        if (fa != v) {
            dfs(v, u);
            ans = max(ans, dp[u] + dp[v] + w);
            dp[u] = max(dp[u], dp[v] + w);
        }
}
```

树的重心

定义：在树中找到一个点，以他为根时，他的各个子树大小中的最大值最小

```
int ans = 0;
vector<int> G[N];
int sz[N], maxx[N], ans = 2e9, ansu;
void dfs(int u, int fa) {
    sz[u] = 1;
    maxx[u] = 0;
    for (auto &v : G[u])
        if (fa != v) {
            dfs(v, u);
            sz[u] += sz[v];
            maxx[u] = max(maxx[u], sz[v]);
        }
    maxx[u] = max(maxx[u], n - sz[u]);
    if (maxx[u] < ans) ans = maxx[u], ansu = u;
}
```

树的中心

定义：在树中找到一个点，他到其他所有点中的最远距离最近 我们可以吧距离分成两部分， 点子树内到他的最远距离，和子树外的最远距离，两者取 max 对于子树内的最远距离很容易 dp 得到 $dp[u] = max(dp[v] + edge[u , v]) $

对于子树外的的点来说，我们可以从上向下 dp 得到，对于 v 点，他子树外的最大距离，要么是父节点 u 加上 u 向下最长的距离，或者向上的最长距离，但是如果 v 刚好在 u 向下的最长路径上，那么就会发生问题，所以要额外记录次长距离，在 在最长路径上时，次长距离可能大于往上的最长距离。所以总的思路就是做两次 dfs ，第一次先把每个先到子树的最长距离和次长距离记录。在第二次的时候计算子树外的最长距离

```
vector<pair<int, int>> G[N];
int dp[N][3];
void dfs(int u, int fa) {
    dp[u][0] = 0;
    dp[u][1] = 0;
    for (auto &[v, w] : G[u])
        if (fa != v) {
            dfs(v, u);
            if (dp[v][0] + w >= dp[u][0]) {
                dp[u][1] = dp[u][0];
                dp[u][0] = dp[v][0] + w;
            } else if (dp[v][0] + w >= dp[u][1]) {
                dp[u][1] = dp[v][0] + w;
            }
        }
}
void dfs2(int u, int fa) {
    for (auto &[v, w] : G[u])
        if (fa != v) {
            if (w + dp[v][0] == dp[u][0])
                dp[v][2] = max(dp[u][2] + w, dp[u][1] + w);
            else
                dp[v][2] = max(dp[u][2] + w, dp[u][0] + w);
            dfs2(v, u);
        }
}
```

树上背包

树上背包一般是树上的 类似背包一样的效果，如题目[P2014 CTSC1997] 选课 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn) 主要是复杂度的证明分析，代码比较简单 首先设置 dp ，$dp[i][j]$ 代表在 i 点的子树内选了 j 门课的最大贡献，显然选课需要有前置，所以在 i 点子树内选一门课只能选自己，$dp[i][1]$ 的分数就是本身分数，然后选 0 门课我们需要给他负无穷的贡献，因为需要达成依赖的关系，这样在下方不选课的时候，该处贡献永远无法产生。

```
vector<int> G[N];
int dp[N][N];
int temp_dp[N];
void dfs(int u, int fa) {
    dp[u][0] = -1e9;
    dp[u][1] = a[u];
    for (auto &v : G[u]) {
        dfs(v, u);
```

```
        for (int i = 0; i <= m + 1; i++) temp_dp[i] = dp[u][i];
        for (int i = 0; i <= m + 1; i++) {
            for (int j = 0; j <= m + 1; j++) {
                temp_dp[i + j] = max(temp_dp[i + j], dp[u][i] + dp[v][j]);
            }
        }
        for (int i = 0; i <= m + 1; i++) dp[u][i] = temp_dp[i];
    }
}
```

这个方法是 $O(n * m^2)$

可以进行优化

首先一个比较简单的优化,我们发现第一重循环不需要从 $m + 1$ 开始，因为子树大小不一定有这么大,然后第二重循环是枚举 这个子树大小的，我们也可以对其进行一定的限制

```
vector<int> G[N];
int dp[N][N], sz[N];
int temp_dp[N];
void dfs(int u, int fa) {
    dp[u][0] = -1e9;
    dp[u][1] = a[u];
    sz[u] = 1;
    for (auto &v : G[u]) {
        dfs(v, u);
        for (int i = 0; i <= min(m + 1, sz[u]); i++) temp_dp[i] = dp[u][i];
        for (int i = min(m + 1, sz[u]) + 1; i <= min(m + 1, sz[u] + sz[v]); i++)
temp_dp[i] = 0;
        for (int i = 0; i <= min(m + 1, sz[u]); i++) {
            for (int j = 0; j <= min(m + 1 - i, sz[v]); j++) {
                temp_dp[i + j] = max(temp_dp[i + j], dp[u][i] + dp[v][j]);
            }
        }
        sz[u] += sz[v];
        for (int i = 0; i <= min(m + 1, sz[u]); i++) dp[u][i] = temp_dp[i];
    }
}
```

这个方法是 $O(n*m)$

dfs序 树上差分

树上倍增

例题

给定一棵 n 个点的树，每条边上有边权。 q 次询问，每次给出两个点 u v ，求 u 到 v 的树上简单路径所经过的所有边的最小值。 其中 $n , q <= 2 *10^5$

```
void dfs(int u, int f) {
    dep[u] = dep[f] + 1;
    for (auto &[v, x] : g[u]) {
        if (v == f) continue;
        // 处理往上一个点的情况
        par[v][0] = u;
        val[v][0] = x;
        dfs(v, u);
    }
}
// 在求解lca的过程中额外维护我们要的信息
int query(int u, int v) {
    int ans = 1 << 30;
    if (dep[u] > dep[v]) swap(u, v);
    int d = dep[v] - dep[u];
    for (int j = 30; j >= 0; j--)
        if (d & (1 << j)) {
            ans = min(ans, val[v][j]);
            v = par[v][j];
        }
    if (u == v) return ans;
    for (int j = 30; j >= 0; j--)
        if (par[u][j] != par[v][j]) {
            ans = min({ans, val[u][j], val[v][j]});
            u = par[u][j];
            v = par[v][j];
        }
    ans = min({ans, val[u][0], val[v][0]});
    return ans;
}
int main() {
    cin >> n >> q;
    for (int i = 1, u, v, w; i < n; i++) {
        // u 到 v 的权值为 w 的边
        cin >> u >> v >> w;
        g[u].push_back({v, w});
        g[v].push_back({u, w});
    }
    dfs(1, 0);
    for (int j = 1; j <= 30; j++) {
        for (int u = 1; u <= n; u++) {
            par[u][j] = par[par[u][j - 1]][j - 1];
            val[u][j] = min(val[u][j - 1], val[par[u][j - 1]][j - 1]);
        }
    }
    for (int i = 1; i <= q; i++) {
        int u, v;
        cin >> u >> v;
        cout << query(u, v) << endl;
    }
}
```

只要信息可以在倍增过程中 **快速合并**。并且问题中 **不带修改**，我们都可以使用倍增来快速询问一些关于 **树上路径** 的信息。

树上差分

点差分

给定一棵 n 个点的树，每个点初始点权都是 0 。你要执行 q 次操作，每次操作形如： u v x 表示对 u 到 v 路径上的所有点点权都增加 x 。 在 q 次操作之后，你需要输出所有点的点权。 是否有一点点熟悉的味道？它是不是很像这样一个题目： 给定一个长度为 n 的数组 a。你要执行 q 次操作，每次操作形如： l r x表示对区间 $[l,r]$ 的每个数增加 x 。

在 q 次操作之后，你需要输出数组中每个数的值。 对于这个序列的版本，我们已经知道可以使用差分，把区间修改的操作转化到原数组的差分数组上，然后通过前缀和就可以复原数组，获得每一个数的值了，时间复杂度 $O(q+n)$ 。 那同样的，原题只不过是把这个背景从序列上搬到了树上。我们显然也可以使用类似的思想来解决这个问题。 假设我们对下图中的 s 到 t 结点路径上的点的点权都增加 1 。

其他点的 cnt 都是 0 。我们考虑如何复原，考虑在序列上的前缀和在树上的表现形式是什么，发现就是 **子树和** 。所以每个点的点权只要求出它为根的子树的差分数组的点权和即可。这个过程可以在 dfs 的时候回溯完成，类似一个极其简单的树形 dp 。 这个差分的过程其实也是十分自然的，我们肯定要对 u 和 v 的差分数组上 $+1$ ，所以考虑转折点 $lca(u,v)$，它把两个点都包括在了自己的子树内，但是它只被经过了一次，所以要在 $lca(u,v)$ 的差分数组上 -1 ，但是这还没有结束，因为现在你发现 $lca(u,v)$ 的所有祖先因为都包含了 $lca(u,v)$ ，所以在计算时都是 1 ，但显然他们的值是 0 ，所以我们还要在 $lca(u,v)$ 的最近的祖先处的差分数组上 -1 。显然这个点就是它的父亲。 这样，点差分就结束了。

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
using namespace std;
typedef long long ll;
const int N = 5e5 + 7;
int n, m, rt;
int par[N][20], dep[N], cnt[N], val[N];
vector<int> G[N];
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    par[u][0] = fa;
    for (int i = 1; i < 20; ++i) {
        par[u][i] = par[par[u][i - 1]][i - 1];
    }
    for (auto &v : G[u]) {
        if (v == fa) continue;
        dfs(v, u);
    }
}
int LCA(int u, int v) {
    if (dep[u] < dep[v]) swap(u, v);
    for (int i = 19; i >= 0; --i) {
        if (dep[par[u][i]] >= dep[v]) u = par[u][i];
```

```
        }
        if (u == v) return u;
        for (int i = 19; i >= 0; --i) {
            if (par[u][i] != par[v][i]) {
                u = par[u][i], v = par[v][i];
            }
        }
        return par[u][0];
    }
    void dfs1(int u, int fa) {
        val[u] = cnt[u];
        for (auto &v : G[u]) {
            if (v == fa) continue;
            dfs1(v, u);
            val[u] += val[v];
        }
    }
    int main() {
        freopen("maxflow.in", "r", stdin);
        freopen("maxflow.out", "w", stdout);
        cin >> n >> m;
        for (int i = 1, u, v; i < n; i++) {
            cin >> u >> v;
            G[u].push_back(v);
            G[v].push_back(u);
        }
        dfs(1, 0);
        while (m--) {
            int x, y;
            cin >> x >> y;
            cnt[x]++;
            cnt[y]++;
            cnt[LCA(x, y)]--;
            cnt[par[LCA(x, y)][0]]--;
        }
        dfs1(1, 0);
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            ans = max(ans, val[i]);
        }
        cout << ans << endl;
    }
```

边差分

给定一棵 n 个点的树，每条边的边权都是 0 。你要执行 q 次操作，每次操作形如： u v $x:$表示对 u 到 v 路径上的所有边边权都增加 x 。 在 q 次操作之后，你需要输出所有边的边权。 假设我们对下图中的 s 到 t 结点路径上的边的边权都增加 1 ：

这次的不同在于所有的操作都是对于边而言的。那么我们是否可以把边转化为点呢？ 我们钦定这样一种对边标记的方法，设有一条边的两个点端点为 u v。并且满足 $dep[u] > dep[v]$ $dep[i]$，代表 i 点的深度。

我们就把这条边认为是 u 的。这样一来由于树的每个节点有且仅有一个父亲，所以每个点只会被分到一条边。其中根节点没有分到任何一条边。这样一来， 上面的图就变成了这样:

接下来问题就回归了点差分。并且这个东西更简单了，因为它相当于对 s 到 t 的路径上除了 $lca(u,v)$的点点权加 1 。 显然我们只需要对 $cnt[s] +1,cnt[t]+1,cnt[lca(s,t)]-2$ 即可。 这样，边差分也就结束了。

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
using namespace std;
typedef long long ll;
const int N = 5e5 + 7;
int n, m, rt;
int par[N][20], dep[N], cnt[N], val[N], qaq[N], ans[N];
vector<int> g[N];
pair<int, int> G[N];
void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    par[u][0] = fa;
    for (int i = 1; i < 20; ++i) {
        par[u][i] = par[par[u][i - 1]][i - 1];
    }
    for (auto &v : g[u]) {
        if (v == fa) continue;
        dfs(v, u);
    }
}
int LCA(int u, int v) {
    if (dep[u] < dep[v]) swap(u, v);
    for (int i = 19; i >= 0; --i) {
        if (dep[par[u][i]] >= dep[v]) u = par[u][i];
    }
    if (u == v) return u;
    for (int i = 19; i >= 0; --i) {
        if (par[u][i] != par[v][i]) {
            u = par[u][i], v = par[v][i];
        }
    }
    return par[u][0];
}
void dfs1(int u, int fa) {
    val[u] = cnt[u];
    for (auto &v : g[u]) {
        if (v == fa) continue;
        dfs1(v, u);
        val[u] += val[v];
    }
    ans[qaq[u]] = val[u];
}
int main() {
    ios::sync_with_stdio(false);
```

```
    cin.tie(0);
    cout.tie(0);
    cin >> n;
    for (int i = 1, u, v; i < n; i++) {
        cin >> u >> v;
        g[u].push_back(v);
        g[v].push_back(u);
        G[i] = {u, v};
    }
    dfs(1, 0);
    for (int i = 1; i < n; i++) {
        int u = G[i].first;
        int v = G[i].second;
        if (dep[u] < dep[v]) swap(u, v);
        qaq[u] = i;
    }
    cin >> m;
    while (m--) {
        int x, y;
        cin >> x >> y;
        cnt[x]++;
        cnt[y]++;
        cnt[LCA(x, y)] -= 2;
    }
    dfs1(1, 0);
    for (int i = 1; i < n; i++) {
        cout << ans[i] << " ";
    }
}
```

总结

在序列上的区间修改，在树上的表现形式就是路径修改。 在序列上的前缀和，在树上的表现形式就是子树和。对 u 和 v 路径上做 **点差分** 等价于 $cnt[u]+1,cnt[v]+1,cnt[lca(u,v)]-1,cnt[fa(lca(u,v))]-1$

对 u 和 v 路径上做 **边差分** 等价于 $cnt[u]+1,cnt[v]+1,cnt[lca(u,v)]-2$ 。

dfs序

给定一棵 n 个点的树，每个点初始点权都是 0 。你要支持两种操作： 1 u v $x:$表示对 u 到 v 路径上的所有点点权都增加 x 。 2 $u:$表示输出点 u 的点权。 坏了，这下修改和查询会穿插进行了。我们还是先来考虑原来序列上的版本： 给定一个长度为 n 的数组 a 。你要支持两种操作： 1 l r $x:$表示对区间 $[l,r]$ 的每个数增加 x 。 2 $d:$表示输出下标为 d 的数。 显然，这个序列上的版本是普通的差分前缀和是无法完成的。我们需要使用 **树状数组或线段树** 维护差分或 **线段树打** 来完成。我们考虑前者，其实前者的本质还是差分和前缀和，差分还是可以直接做，只不过这个前缀和是动态的，所以我们无法直接获得，要使用 **树状数组** 维护。

```
int L[N], R[N], tot;
// L[i] 即为 i 点为根的子树的最小点的编号
// R[i] 即为 i 点为根的子树的最大点的编号
```

```
void dfs(int x, int fa) {
    L[x] = ++tot;
    for (auto u : g[x]) {
        if (u == fa) continue;
        dfs(u, x);
    }
    R[x] = tot;
}
```

<<<<<<< HEAD

=======

单调栈 单调队列

单调栈

例题引入 给定一个 n 个数的数组 a ，定义 $f(i)$ 是 $a[i]$ 左边第一个小于 $a[i]$ 的元素的下标，若不存在，则 $f(i)=0$ ，试求出所有数的 $f(i)$

假如有两个数 $a[i]$ 和 $a[j]$ ， $a[i] <a[j]$ 并且 $i>j$ ，意思就是 $a[i]$ 在 $a[j]$ 的右边并且还比 $a[j]$ 小。那么 $a[j]$ 对于后面的数来说，必然不会作为答案，相当于 $a[j]$ 这个值没有用了，那么我们就可以想象栈。对于每个数进来，因为维护的相当于是前面的数的一个集合，所以我们可以把前面的比他大的数都弹出去。那么停止的时候， 就是遇到了第一个小于 $a[i]$ 的数，其实用单调栈的思想可以看作只是暴力的优化或者说减枝，因为栈里维护的也是前面的集合，或者说是可行的答案的集合，因为每加进来一个数，都会弹出前面不可能作为答案的数。显然每个数最多只能被加进来一次，弹出去一次，那么时间复杂度显然是 $O(n)$ 的。那么就比前面的方法优秀很多。

```
void solve() {
    int n;
    cin >> n;
    stack<int> st; // 栈里面存放的是下标
    vector<int> a(n + 1), f(n + 1);
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
        while (!st.empty() && a[st.top()] >= a[i]) {
            // 不同题目的符号不同，注意分辨
            st.pop();
        }
        f[i] = (st.size() == 0 ? 0 : st.top()); // f[i] = (st.size() == 0 ? n + 1
: st(top());
        st.push(i);                             // 注意栈里面放进去的是下标
    }
    for (int i = 1; i <= n; ++i) {
        cout << f[i] << " \n"[i == n];
    }
}
```

单调队列

单调队列 是一种主要用于解决**滑动窗口**类问题的数据结构在长度为 n 的序列中，求每个长度为 m 的区间的区间最值。它的时间复杂度是 n ，在这个问题中比线段树或者 st 表优秀。

1. 思考一下如何用普通队列做，那么只要用一个队列来维护当前窗口里的所有值，然后线性扫描一下队列，每次新加进来一个数，就把队尾加一个数，队头弹一个数出去，很显然时间复杂度是 $O(nk)$
2. 类似于单调栈的想法，观察一下哪些数永远不可能作为答案了，也就是说没有用了。

所以假如要求区间最小值，有一个数如果更靠后并且更小，那么前面的数字就没有用了。所以新加进来一个数，可以把靠右的比他大的数弹出去，然后因为区间往右边挪了，维护的数的左端点也要合法，所以弹出左侧不合法的下标，因为停止的条件是碰到了一个比他小的数，所以容易想到队列里面的元素排列是递增的，所以叫单调队列，因为维护的都是可能的答案加上单调递增。所以处理完之后，最小值就是队头的元素，就可以 $O(1)$ 得到。因为需要左侧，右侧，可以联想到 $depue$ 这个数据结构

```
#include <iostream>
#include <algorithm>
#include <deque>
#include <vector>
using namespace std;
int n, m;
void solve() {
    cin >> n >> m;
    vector<int> a(n + 1), ans(n + 1);
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }
    deque<int> q, q1;
    for (int i = 1; i <= n; ++i) {
        while (!q.empty() && i - q.front() >= m) {
            q.pop_front();
        }
        while (!q1.empty() && i - q1.front() >= m) {
            q1.pop_front();
        }
        while (!q.empty() && a[q.back()] >= a[i]) {
            q.pop_back();
        }
        while (!q1.empty() && a[q1.back()] <= a[i]) {
            q1.pop_back();
        }
        q.push_back(i);
        q1.push_back(i);
        if (i >= m) {
            cout << a[q.front()] << " \n"[i == n];
            ans[i] = a[q1.front()];
        }
    }
    for (int i = m; i <= n; i++) cout << ans[i] << " ";
}
```

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    solve();
}
```