密码锁

一种显然的做法是，枚举所有 10^5 种可能的密码，然后枚举 n 种状态，然后枚举转移过去的方式，判断是否合法。只有当 n 种状态都存在转移过去的方式时才是一个合法的密码。

一种复杂度可能更优秀的做法是，枚举 n 种状态，枚举转移过去的方式，给原密码 +1 ，然后输出权值为 n 的密码的个数。

消消乐

首先考虑如何判断一个子串是否可消除。

可以发现，消除的顺序并不重要，能被消除当且仅当贪心消可以把一整个序列消完。通过简单的调整法即可证明。

因此我们可以选择从左往右贪心：维护一个栈，每次新增一个元素时如果和栈顶相等就消掉，否则 push 进栈里。合法当且仅当加入所有元素之后栈恰好为空。

不难发现这个做法和右端点在哪并没有太大关系，因此枚举左端点之后往右枚举右端点即可做到 $O(n^2)$ 。

考虑如何加速这个做法。我们并不那么关心栈里的具体状态，只关心栈什么时候为空。

对于一个左端点 l ，我们希望求出第一个右端点 r 使得 $[l,r]$ 合法，那么以 l 为左端点的方案数就是以 $r+1$ 为左端点的方案数加一。

设 $nxt_{i,j}$ 表示栈里当前只有一个元素 j ，那么从 i 开始加入元素，什么时候能把 j 消掉。转移分为两种：

- 如果 $s_i=j$ ，显然 $nxt_{i,j}=j$ 。
- 否则，s_i 会被 push 进栈里，因此需要先把 s_i 消掉才能消 j ，因此有 $nxt_{i,j}=nxt_{nxt_{i+1,s_i}+1,j}$ 。

直接转移即可做到 $O(26n)$ 。

```
int n;
char s[sz];
int nxt[sz][26];
int match[sz],dep[sz];

int main() {
    read(n);
    cin>>s;
    rep(j,0,25) nxt[n][j]=nxt[n+1][j]=-1;
    ll ans=0;
    drep(i,n-1,0) {
        match[i]=nxt[i+1][s[i]-'a'];
        if (match[i]==-1) match[i]=n;
        else dep[i]=dep[match[i]+1]+1,ans+=dep[i];
        rep(j,0,25) nxt[i][j]=nxt[match[i]+1][j];
        nxt[i][s[i]-'a']=i;
    }
    cout<<ans<<'\n';
    return 0;
}
```

结构体

C

```
map <string, ll> siz = {{"byte", 1}, {"short", 2}, {"int", 4}, {"long", 8}};
map <string, ll> mx = {{"byte", 1}, {"short", 2}, {"int", 4}, {"long", 8}};
map <string, map <string, ll> > start;
map <string, map <string, string> > Str;
map <string, string> str;
map <string, ll> point;
```

用 map <string, ll> siz; 来维护每个 type 的字节数

map<string, ll> mx; 维护每个 type 的对齐

map<string, map <string, ll> > start; 表示 type A 中 name 为 B 的元素的启示位置

map<string, map<string, string> > Str; 表示 type A 中 name 为 B 的元素的 type

map<string, string> str; 表示最外层元素 x 的 type

map<string, ll> point; 表示最外层元素 x 的起始位置

操作 1

通过除法上取整来对齐，同时维护 Strat, Str

```
if(op == 1) {
    string S;
    int k;
    cin >> S >> k;
    ll sz = 0, mxx = 0;
    for(int i = 0; i < k; i++) {
        string T, A;
        cin >> T >> A;
        sz = sz ? ((sz - 1) / mx[T] + 1) * mx[T] : 0;
        start[S][A] = sz;
        Str[S][A] = T;
        sz += siz[T];
        mxx = max(mxx, mx[T]);
    }
    sz = ((sz - 1) / mxx + 1) * mxx;
    siz[S] = sz;
    mx[S] = mxx;
    cout << sz << ' ' << mxx << '\n';
}
```

操作 2

注意对齐，同时维护 str, point

```
if(op == 2) {
    string T, A;
    cin >> T >> A;
    pos = pos ? ((pos - 1) / mx[T] + 1) * mx[T] : 0;
    str[A] = T;
    point[A] = pos;
    pos += siz[T];
}
```

操作 3

一个一个依次求位置，再相加

```
if(op == 3) {
    string S;
    cin >> S;
    int len = S.length();
    string cur = "";
    ll sum = 0;
    bool first = 1;
    string type;
    for(int i = 0; i <= len; i++) {
        if(i < len && S[i] != '.') cur += S[i];
        else {
            if(first) first = 0, sum = point[cur], type = str[cur];
            else sum += start[type][cur], type = Str[type][cur];
            cur = "";
        }
    }
    cout << sum << '\n';
}
```

操作 4

利用维护的信息，一层一层化简为子问题求解，注意判断 ERR 的情况

```
bool work(ll addr, string type) {
    if(type == "") {
        ll po = -1;
        string si = "";
        for(auto [s, p] : point)
        if(p <= addr && p > po) po = p, si = s;
        if(po == -1) {
            return false;
        }
        ANS += si + ".";
        return work(addr - po, str[si]);
    }
    else {
        if(siz[type] < addr) return false;
        ll po = -1;
        string si = "";
        for(auto [s, p] : start[type])
        if(p <= addr && p > po) po = p, si = s;
```

```
        ANS += si;
        type = Str[type][si];
        if(!cc[type]) {
            ANS += ".";
            return work(addr - po, type);
        }
        if(addr - po < siz[type]) {
            ANS += '\n';
            return true;
        }
    }
    return false;
}
```

种树

直接二分答案，只需要判断能否在时间 t 之前完成任务。

因为长高的长度和 1 取了 max ，所以每棵树会有一个时间 t_i ，只有它在 t_i 之前被种下去，就可以在 t 之前长到 a_i 。这个 t_i 可以直接算也可以二分，二分的细节会少一点，但有 TLE 的风险。

二分 t_i 的时候有一个细节是要计算 $\sum_{x=t_i}^{t}\max(b_i+c_ix,1)$ 。这是个分段一次函数求和，因此在分段点切开之后两边分别算即可。有一点细节但还好。另一个需要注意的点是这东西可以取到 $(10^9)^3$ ，小心爆 long long 。

得到 t_i 之后就需要确定种树的顺序。显然的想法是先种 t_i 较小的树，但因为有父亲种了才能种儿子的限制，因此 t_i 较小的树可能现在并不能种。

但是注意到父亲的 t_i 比儿子的 t_i 大是没有意义的：如果儿子需要在 t_i 之前种下去，那么父亲肯定需要在 t_i-1 之前种下去。

因此可以设 $t'_i=\min(t_i,\min_v(t_v-1))$ ，那么 t'_i 就一定满足父亲比儿子小了。

此时直接按 t'_i 排序之后贪心种即可。合法当且仅当排序之后 $t'_i\geq i$ 。

```
int n;
vector<int>V[sz];
ll a[sz],b[sz],c[sz];
ll ti[sz];

void dfs(int x,int fa) {
    for (auto v:V[x]) if (v!=fa) dfs(v,x),chkmin(ti[x],ti[v]-1);
}

ll P;
lll sum(ll l,ll r,ll b,ll c) {
    assert(l<=r);
    if (l==r) return 0;
    if (l<P&&P<r) return sum(l,P,b,c)+sum(P,r,b,c);
    if (b+c*r<=0) return r-l;
    return b*(r-l)+c*lll(r*(r+1)/2-l*(l+1)/2);
}

int check(ll t) {
    rep(i,1,n) {
        if (b[i]<=0&&c[i]<=0) P=t+1;
```

```
        else if (b[i]<=0) P=(-b[i])/c[i],assert(b[i]+c[i]*P<=0&&b[i]+c[i]*(P+1)>0);
        else if (c[i]==0) P=t+1;
        else if (c[i]<0) P=(b[i]-1)/(-c[i]),assert(b[i]+c[i]*P>0&&b[i]+c[i]*(P+1)<=0);
        else P=t+1;
        ll l=1,r=t,pos=-1;
        while (l<=r) {
            ll m=(l+r)>>1;
            lll tt=sum(m-1,t,b[i],c[i]);
            if (tt>=a[i]) pos=m,l=m+1;
            else r=m-1;
        }
        if (pos==-1) return 0;
        ti[i]=pos;
    }
    dfs(1,0);
    sort(ti+1,ti+n+1);
    rep(i,1,n) if (ti[i]<i) return 0;
    return 1;
}

int main() {
    file();
    read(n);
    rep(i,1,n) read(a[i],b[i],c[i]);
    int x,y;
    rep(i,1,n-1) read(x,y),V[x].push_back(y),V[y].push_back(x);
    ll l=0,r=1e9,ans=r;
    while (l<=r) {
        ll mid=(l+r)>>1;
        if (check(mid)) ans=mid,r=mid-1;
        else l=mid+1;
    }
    cout<<ans<<'\n';
    return 0;
}
```