



枚举优化与搜索

洛谷网校

基础衔接提高计划

吾王美如画



www.luogu.com.cn

枚举优化

枚举是一种遍历**所有可能**的解来寻找问题答案的方法。但通常一个简单的枚举算法不足以在给定的时间/空间限制内解决问题。

通过观察问题在朴素枚举之外的更多**性质**，我们对枚举算法进行优化，使其能通过更大规模的数据，从而得到更多部分分或直接通过题目。

P3799 小 Y 拼木棒

有 n 根木棒，第 i 根木棍长为 a_i 。从中选取 4 根木棍组成一个正三角形，问有多少种选法？

答案对 $10^9 + 7$ 取模

对于 30% 的数据： $1 \leq n \leq 5 \times 10^3$

对于 100% 的数据： $1 \leq n \leq 10^5$ ， $1 \leq a_i \leq M = 5 \times 10^3$

- 考虑最为暴力的做法，我们直接枚举这四根木棍分别是哪根，再判定是否构成正三角形，时间复杂度为 $O(N^4)$
- 考虑进一步优化，观察到**正三角形**的这个判定条件十分特殊，通过四根木棒构成一个正三角形，必然由两根较长的**长度相等**的木棒构成其两边，而剩下两根木棒拼接起来构成第三边。这样我们便想到可以通过**枚举拼接的两根木棒**，这样便得知较长的两根木棒的长度 l ，设长度为 l 的木棒有 n 根，则对于这对拼接木棒，有 $\binom{n}{2}$ 种选择方法，总时间复杂度为 $O(N^2)$
- 最后我们观察到实际木棒长度范围很小，考虑通过**枚举长度**代替枚举木棒，时间复杂度为 $O(M^2)$

```
ll a[11000], cnt[5100];
ll C_2 (ll n){
    return (n*(n-1)/2)%MOD;
}
int main(){
    int n=read();
    for(int i=1; i<=n; i++)
        a[i]=read(), cnt[a[i]]++;
    ll ans=0;
    for(int x=1; x<=2500; x++){
        ans+=C_2(cnt[x])*C_2(cnt[2*x])%MOD;
        ans%=MOD;
        for(int y=x+1; x+y<=5000; y++){
            ans+=cnt[x]*cnt[y]%MOD*C_2(cnt[x+y])%MOD;
        }
    }
    cout<<ans<<endl;
    return 0;
}
```

P1118 [USACO06FEB] Backward Digit Sums G/S

有这样一个游戏：

写出一个 $1 \sim n$ 的排列 a ，然后每次将相邻两个数相加，构成新的序列，再对新序列进行这样的操作，显然每次构成的序列都比上一次的序列长度少 1，直到只剩下一个数字。

现在想要倒着玩这样一个游戏，如果知道 n ，以及最后得到的数字的大小 sum ，请你求出最初序列

若答案有多种可能，则输出字典序最小的那一个。

对于 40% 的数据： $1 \leq n \leq 7$

对于 80% 的数据： $1 \leq n \leq 10$

对于 100% 的数据： $1 \leq n \leq 12$ ， $1 \leq sum \leq 12345$

- 考虑最为暴力的做法，我们每次枚举下一层的数是如何由上一层合并而来的（例如枚举每一层最左边的数是多少，从而恢复出整层数），而此处 sum 较大，近乎 $O(sum^{12})$ 的复杂度显然无法通过此题。
- 进一步的，我们观察到题目中给出了初始序列一定是一个排列，于是我们直接枚举这个初始排列，并模拟游戏过程得出最后一层的数，时间复杂度为 $O(N^2 \times N!)$ 仍不足以通过全部数据。

- 这时我们几乎很难从枚举的部分获得较大提升，我们考虑优化**模拟**的这一部分。观察游戏过程，发现其与**杨辉三角**的结构极其相似。事实上，对于第一行的第 i 个数 ($0 \leq i < n$)，其对于 sum 的贡献即为 $\binom{n-1}{i} * a_i$ ，通过提前预处理这个系数即可优化掉整个模拟过程，时间复杂度 $O(N!)$ 。
- 虽然此时时间复杂度已经够好，但我们注意到题目对 sum 的限制暗示我们还有很多剪枝的余地。在递归枚举排列的过程中，如果已选择的部分的 sum_{recent} 已经比 sum 大了，则可直接返回。此举可以大幅加快速度。

```
int vis[20], his[20], flag=0;
void dfs(int dep, int s){
    if (dep == n){
        if (s == sum){
            for(int i=0; i<n; i++)
                printf("%d ", his[i]);
            puts("");
            flag=1;
        }
        return;
    }
    for(int i=1; i<=n; i++){
        if (vis[i])
            continue;
        vis[i]=1;
        his[dep]=i;
```

```
        dfs(dep+1, s+i*f[dep]);
        vis[i]=0;
        if (flag)
            return;
    }
}
int main(){
    n=read(), sum=read();
    fac[0]=1;
    for(int i=1; i<=12; i++)
        fac[i]=i*fac[i-1];
    for(int i=0; i<n; i++)
        f[i]=(fac[n-1]/fac[i])/fac[n-1-i];
    dfs(0, 0);
    return 0;
}
```

P2119 [NOIP2016 普及组] 魔法阵

有 m 个物品，每个物品有一个魔法值 X_i 。当且仅当四个魔法物品 a, b, c, d 满足 $X_a < X_b < X_c < X_d$, $X_b - X_a = 2(X_d - X_c)$, $X_b - X_a < (X_c - X_b)/3$ 时称这四个魔法物品形成了一个魔法阵。

现在，大魔法师想要知道，对于每个魔法物品，作为某个魔法阵的 A 物品出现的次数，作为 B 物品的次数，作为 C 物品的次数，和作为 D 物品的次数。

对于 50% 的数据： $1 \leq X_i \leq n \leq 125$, $1 \leq m \leq 200$

对于 100% 的数据： $1 \leq X_i \leq n \leq 15000$, $1 \leq m \leq 40000$

- 直接考虑 $O(N^4)$ 暴力枚举四个物品，可以获得 45~55 分。
- 考虑优化枚举。通常我们都会尝试将**限制**融入枚举过程中，以减少复杂度。此处我们注意到 $X_b - X_a = 2(X_d - X_c)$ 的限制，我们记 $t = (X_d - X_c)$ ，则有 $X_b - X_a = 2t$ ，于是只需枚举 X_a, X_c, t 即可，复杂度 $O(N^3)$
- 更进一步的，由 $X_b - X_a < \frac{X_c - X_b}{3}$ 得 $X_c - X_b > 6t$ 。于是，我们发现对于一对固定的 X_a 与 t ，能确定 $X_c - X_d = t$ 且 $X_c > X_a + 8t$ ，这样我们先枚举 t ，便可**从后往前**累计每对 X_c, X_d 的贡献，从而计算每个数作为 X_a, X_b 的出现次数，反之同理。时间复杂度 $O(N^2)$

```
int cnt[16000],x[41000];
int main(){
    n=read(),m=read();
    for(int i=1;i<=m;i++){
        x[i]=read();
        cnt[x[i]]++;
    }
    for(int t=1;9*t<=n;t++){
        int res=0;
        int a,b,c,d;
        for(a=n-9*t-1;a>0;a--){
            b=a+2*t;c=a+8*t+1;d=a+9*t+1;
            res+=cnt[c]*cnt[d];
            ans[a][0]+=cnt[b]*res;
            ans[b][1]+=cnt[a]*res;
        }
    }
```

```
        res=0;
        for(d=9*t+2;d<=n;d++){
            a=d-9*t-1;b=d-7*t-1;c=d-t;
            res+=cnt[a]*cnt[b];
            ans[c][2]+=cnt[d]*res;
            ans[d][3]+=cnt[c]*res;
        }
    }
    for(int i=1;i<=m;i++){
        for(int j=0;j<4;j++)
            printf("%d ",ans[x[i]][j]);
        puts("");
    }
    return 0;
}
```

DFS与BFS

深度优先搜索（DFS） 是一种用于遍历或搜索图的算法。其思想是从起始节点开始，沿着一个分支一直走到底，再回溯到上一个节点，继续沿下一个分支走到底，直到遍历完所有节点。

广度优先搜索（BFS） 是一种用于遍历或搜索图的算法。其思想是从起始节点开始，首先访问所有相邻节点，然后再逐层向外扩展访问，直到遍历完所有节点。

P1451 求细胞数量

一矩形阵列由数字 0 到 9 组成，数字 1 到 9 代表细胞，细胞的定义为沿细胞数字上下左右若还是细胞数字则为同一细胞，求给定矩形阵列的细胞个数。

对于 100% 的数据，保证 $1 \leq n, m \leq 100$ 。

- *Floodfill*算法，即洪水算法，核心思想为每次从一个**没有被访问过**的节点开始尽可能向四周扩散，并且标记已经访问过的点。扩散过程可以通过*dfs*或*bfs*的方式实现
- 如右图所示，我们的每次扩撒过程是从每个色块中最深的位置开始，向浅色的位置逐步扩散。

[illegible][illegible]

此处我们通过 dfs 实现扩散：

```
int dx[]={-1,1,0,0},dy[]={0,0,1,-1};
void dfs(int x,int y){
    if (vis[x][y]||a[x][y]=='0')
        return;
    if (x<=0||x>n||y<=0||y>m)
        return;
    vis[x][y]=1;
    for(int i=0;i<4;i++)
        dfs(x+dx[i],y+dy[i]);
}
```

```
int main(){
    cin>>n>>m;
    memset(vis,0,sizeof(vis));
    for(int i=1;i<=n;i++)
        cin>>(a[i]+1);
    int ans=0;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
            if (a[i][j]!='0'&&vis[i][j]!=1){
                ans++;
                dfs(i,j);
            }
    cout<<ans<<endl;
    return 0;
}
```

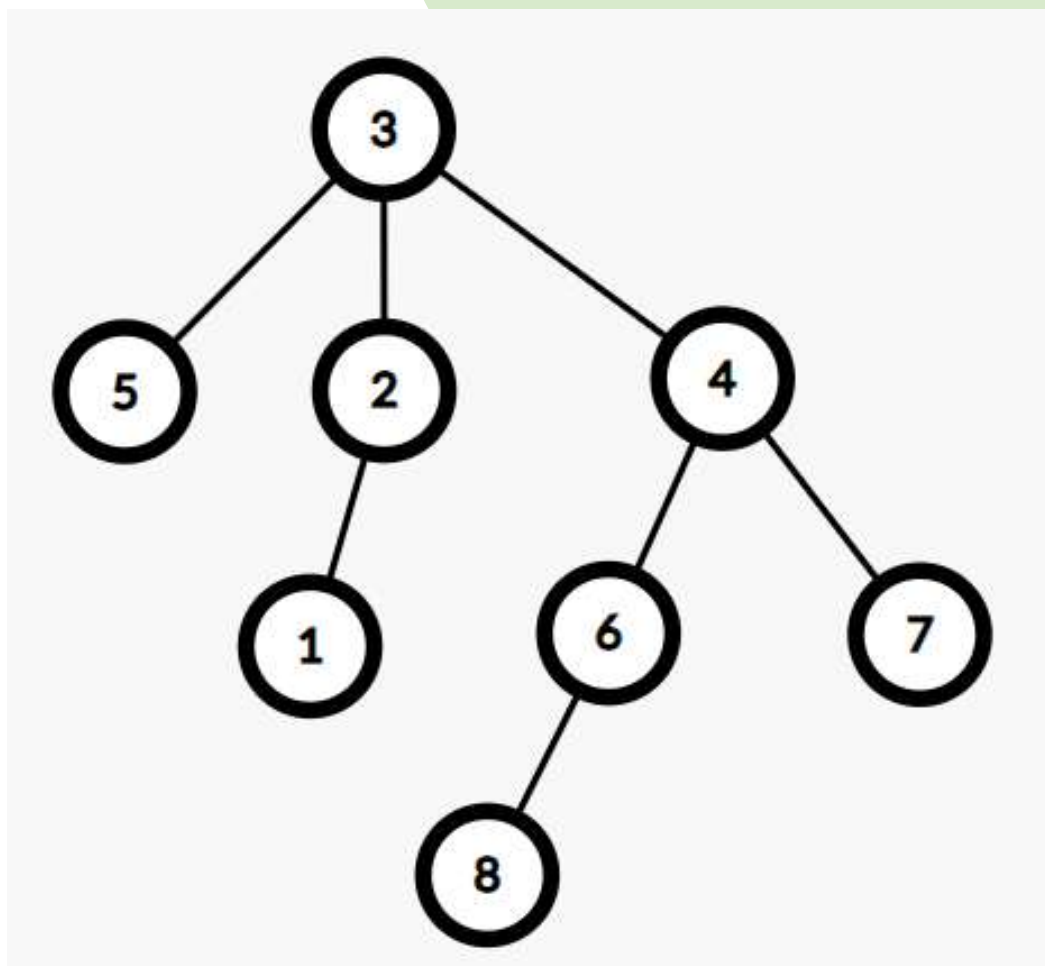
当然，我们也可以使用**bfs**来实现。使用**bfs**的优点在于，其按照离源头的距离一层一层地进行遍历。这样就算有多个源头，我们直接将所有源头一开始全部加入队列并开始遍历，这样第一次访问到的距离便是离所有源头中最短的距离。

```
void bfs(int x,int y){
    queue<pair<int,int>> q;
    q.push(make_pair(x,y)); vis[x][y]=1;
    while(!q.empty()){
        pair<int,int>now=q.front(); q.pop();
        int u_x=now.first,u_y=now.second;
        for(int i=0;i<4;i++){
            int v_x=u_x+dx[i],v_y=u_y+dy[i];
            if (v_x<=0||v_x>n||v_y<=0||v_y>m)
                continue;
            if (vis[v_x][v_y]||a[v_x][v_y]=='0')
                continue;
            vis[v_x][v_y]=1;
            q.push(make_pair(v_x,v_y));
        }
    }
}
```

DFS 序列是指 DFS 调用过程中访问的节点编号的序列。

我们发现，每个子树都对应 DFS 序列中的**连续一段**，这一点帮助我们z将树上的子树信息转移到序列的区间关系上，有利于我们就行处理。

我们考虑以下例子：



(选学) CF1328E Tree Queries

给你一个以 1 为根的有根树.

每回询问 k 个节点, 求出是否有一条以根节点为一端的链使得询问的每个节点到此链的距离均 ≤ 1

只需输出可行性, 无需输出方案.

$$2 \leq n, \sum k_i \leq 2 \times 10^5$$

- 首先观察题目，一点到一条链距离小于等于 1，有三种可能。
 1. 链经过该点
 2. 链经过该点父亲
 3. 链经过该点儿子
- 而由于题设中的链一定是**从根出发**的，故这三种可能都**一定经过该点父亲**，问题即转化为询问点集的父亲是否构成一条从根出发的链

- 考虑我们刚刚学习的 dfs 序的知识，一个节点的子树对应 dfs 序上的一个区间，我们记节点 x 在 dfs 序上的下标为 $dfn[x]$ ，其子树大小为 $size[x]$ ，则其子树对应的区间为 $dfn[x], dfn[x] + 1, \dots, dfn[x] + size[x] - 1$
- 这样，我们对查询点的父亲的 dfn 排序，之后按从小到大的顺序判定是否构成子树关系即可。


```
void dfs(int u,int f){
    dfn[u]=++cnt;fa[u]=f;siz[u]=1;
    for(auto v:g[u]){
        if (v==f) continue;
        dfs(v,u);
        siz[u]+=siz[v];
    }
}
bool cmp(int x,int y){return dfn[x]<dfn[y];}
bool check(int x,int y){
    if (dfn[x]<=dfn[y]&&
        dfn[y]<=dfn[x]+siz[x]-1)
        return 1;
    return 0;
}
int main(){
    n=read(),m=read();
```

```
    for(int i=1;i<n;i++){
        int u=read(),v=read();
        g[u].push_back(v);g[v].push_back(u);
    }
    dfs(1,0); siz[0]=n+1;
    while(m--){
        int k=read();
        for(int i=1;i<=k;i++)
            a[i]=fa[read()];
        sort(a+1,a+1+k,cmp);
        int flag=1;
        for(int i=1;i<k;i++)
            if (!check(a[i],a[i+1])) flag=0;
        if (flag) puts("YES");
        else puts("NO");
    }
    return 0;
}
```

P3379 【模板】最近公共祖先 (LCA)

给定一棵 N 个节点的多叉树， M 次询问，每次询问指定两个点直接最近的公共祖先。

$$1 \leq N, M \leq 500000$$

- 考虑求 u, v ($dfn[u] \leq dfn[v]$) 两点的最近公共祖先 w ，我们分两种情况讨论
 - u 是 v 的祖先，则 u 必是 u, v 的最近公共祖先
 - 否则，考虑 dfs 的过程，必是**先从 w 走向 u** ，走完 u 的子树后再退回 w 并走向 v ，则我们可以知道，在 u 和 v 的 dfs 序之间必存在 w 的一个儿子 s ，其是 v 的祖先。而这个 s ，也必定是 u 和 v 的 dfs 序之间**深度最浅**的节点。故我们考虑用 st 表维护 dfs 序列对应的深度，每次查询相当于询问**区间深度最小值所对应的节点**，即 s 。而 s 的父亲即为所求的 w 。

这里我们稍微复习一下st表的内容。st表以 $f[i][j]$ 的形式记录区间 $[j, j + 2^i - 1]$ 范围内的信息。初始化时从低到高枚举 i 。并将 $f[i-1][j]$ 和 $f[i-1][j + 2^{i-1}]$ 的信息合并为 $f[i][j]$ 。比如我们此处需要维护的信息即为区间内深度最小的节点，我们可以通过右图所示代码进行合并。

```
int x=st[i-1][j];
int y=st[i-1][j+(1<<(i-1))];
if (dep[x]<dep[y]) st[i][j]=x;
else st[i][j]=y;
```

当查询 $[l, r]$ 时，我们发现可以通过合并 $f[k][l]$ 和 $f[k][r - 2^k + 1]$ 的信息来得到。其中 $k = \lfloor \log_2(r - l + 1) \rfloor$ 。合并方式与初始化时一致。

```

void dfs(int u,int f){
    fa[u]=f,siz[u]=1;
    dfn[u]=++cnt;dep[u]=dep[f]+1;
    st[0][dfn[u]]=u;
    for(auto v:g[u]){
        if (v==f) continue;
        dfs(v,u);
        siz[u]+=siz[v];
    }
}

int ask(int l,int r){
    int len=r-l+1,bit=log_2(len);
    int x=st[bit][l],y=st[bit][r-(1<<bit)+1];
    if (dep[x]<dep[y]) return fa[x];
    else return fa[y];
}

int main(){
    n=read(),m=read(),s=read();

```

```

    for(int i=1;i<n;i++){
        int u=read(),v=read();
        g[u].push_back(v);g[v].push_back(u);
    }
    dfs(s,0);
    for(int i=1;i<=20;i++){
        for(int j=1;j+(1<<i)-1<=n;j++){
            int x=st[i-1][j];
            int y=st[i-1][j+(1<<(i-1))];
            if (dep[x]<dep[y]) st[i][j]=x;
            else st[i][j]=y;
        }
    }
    while(m--){
        int u=read(),v=read();
        if (u==v) {printf("%d\n",u);continue;}
        if (dfn[u]>dfn[v]) swap(u,v);
        printf("%d\n",ask(dfn[u]+1,dfn[v]));
    }
    return 0;
}

```

基础剪枝

当谈论到剪枝，我们主要有两大方式（不考虑记忆化）：**最优化剪枝与可行性剪枝**

- 可行性剪枝：在目前状态明显无法通往合法的终状态时结束该搜索分支。
- 最优化剪枝：在目前状态明显无法通往比现有解更加优秀的终止状态时结束该搜索分支

P1731 [NOI1999] 生日蛋糕

制作一个体积为 $N\pi$ 的 M 层生日蛋糕，每层都是一个圆柱体。

设从下往上数第 i 层蛋糕是半径为 R_i ，高度为 H_i 的圆柱。当 $i < M$ 时，要求 $R_i > R_{i+1}$ 且 $H_i > H_{i+1}$

求使得蛋糕外表面（最下一层的下底面除外）的面积 Q 最小的设计方案。

除 Q 外，以上所有数据皆为正整数

$$N \leq 2 \times 10^4, M \leq 15$$

- 首先，我们暴力地自底向上递归枚举每一层的半径和高度。最后再第 M 层枚举之后判定总体积是否合规并更新最小表面积。
- 接着我们考虑剪枝，我们还是从可行性和最优性两方面思考

可行性：注意到总体积恰好等于 N （为了记述方便，后文均省去 π ），其实给了我们两个方向的剪枝：当前**体积过大**；当前**体积过小**。设当前已枚举的层数为 x ，当前总体积为 s ，有以下两条剪枝方法：

1. 当前体积加上后面每层取最小体积(即取 $H_i, R_i = M - i + 1$)，总体积($s + \sum_{k=1}^{M-x} k^3$)仍然大于 N
2. 当前体积加上后面每层取最大体积(为了计算方便，我们可适当放大，取 $H_i = H_x, R_i = R_x$)，总体积($s + R_x^2 H_x (M - x)$)仍小于 N

最优化：当前奶油面积+之后最小奶油面积(可取 $\sum_{k=1}^{M-x} 2 \times k^2$) > 已求最小奶油面积，可提前终止该分支

当然，我们还可以优化枚举过程，例如考虑**从大到小枚举半径和高**，将枚举下限设置为剩余的层数等等

```
int mini=1000000,n,m;
void dfs(int step,int sum,int t,int r,int h){
    a[step]=r;b[step]=h;
    if (t+c[step+1]>n)return;
    if (sum+d[step+1]>mini)return;
    if (t+(r-1)*(r-1)*(h-1)*(m-step)<n)
        return;
    if (step==m){
        if ((sum<mini)&&(t==n)){mini=sum;};
        return;}
    for(int i=r-1;i>=m-step;i--)
        for(int j=h-1;j>=m-step;j--)
            dfs(step+1,sum+2*i*j,t+i*i*j,i,j);
}
```

```
int main(){
    cin>>n>>m;
    c[m+1]=0;d[m+1]=0;
    for(int i=1;i<=m;i++){
        d[m-i+1]=d[m-i+2]+2*i*i;
        c[m-i+1]=c[m-i+2]+i*i*i;
    }
    for(int i=int(sqrt(n));i>=m;i--)
        for(int j=n/(m*m);j>=m;j--)
            dfs(1,2*i*j+i*i,i*i*j,i,j);
    if (mini == 1000000) puts("0");
    else cout<<mini;
    return 0;
}
```

部分分

P1600 [NOIP2016 提高组] 天天爱跑步

给定一棵 n 个节点的树。有 m 个人，第 i 个人起点为 s_i ，终点为 t_i ，从 0 时刻开始，每人以每秒跑一条边的速度，沿自己的路径向终点前进。现树上每一个点 x 在 $w[x]$ 时刻查询该时刻恰好行至该点的人的数量，输出每个点查询的结果。

测试点编号	$n =$	$m =$	约定
1 ~ 2	991	991	所有人的起点等于自己的终点，即 $\forall i, s_i = t_i$
3 ~ 4	992	992	所有 $w_j = 0$
5	993	993	无
6 ~ 8	99994	99994	$\forall i \in [1, n - 1]$, i 与 $i + 1$ 有边。即树退化成 $1, 2, \dots, n$ 按顺序连接的链
9 ~ 12	99995	99995	所有 $s_i = 1$
13 ~ 16	99996	99996	所有 $t_i = 1$
17 ~ 19	99997	99997	无
20	299998	299998	无

- 1~2: 因为起点等于终点, 当 $w[x] = 0$, x 处的询问结果为以 x 为起点的路径数量; 否则询问结果为0。
- 3~4: 因为 $w[x] = 0$, 询问结果即为以 x 为起点的路径数量。
- 5: 数据范围很小, 暴力模拟每个人从起点走向终点的过程。
(其实这一方法可以覆盖前两种情况)

```
int dfs1(int u,int t,int dep,int fa){
    int flag=0;
    if (u==t) flag=1;
    for(auto v:g[u]){
        if (v==fa) continue;
        flag|=dfs1(v,t,dep+1,u);
    }
    if (flag&&w[u]==dep) ans[u]++;
    return flag;
}

void case1(){
    for(int i=1;i<=m;i++)
        dfs1(a[i].s,a[i].t,0,0);
    for(int i=1;i<=n;i++)
        printf("%d ",ans[i]);
}
```


- 6~8: 假设我们只考虑**从左往右跑**, 则我们可以确定 x 节点查询到的一定是从 $x - w[x]$ 出发的那些人。于是我们可以从左到右, 记录从各个节点出发的还在跑的人的数量, 查询时直接输出即可。(从右往左同理)
- 9~12: 由于所有人都是**从根出发**, 可知每个人到达每个节点的时间点都是其**深度** $deep[x]$ (根处深度为0), 则若 $w[x] \neq deep[x]$, 查询结果一定为0。否则, 我们统计有多少人的终点在 x 的子树内, 则这些人必然在 $w[x]$ (即 $deep[x]$) 时跑到了 x 。

```
void case2(){
    for(int i=1;i<=m;i++){
        if (a[i].s<=a[i].t)
            cnt[a[i].s]++,
            upd[a[i].t].push_back(a[i].s);
        for(int i=1;i<=n;i++){
            if (i-w[i]>0) ans[i]+=cnt[i-w[i]];
            for(auto j:upd[i]) cnt[j]--;upd[i].clear();
        }
        memset(cnt,0,sizeof(cnt));
        for(int i=1;i<=m;i++){
            if (a[i].s>a[i].t)
                cnt[a[i].s]++,
                upd[a[i].t].push_back(a[i].s);
        }
        for(int i=n;i;i--){
            if (i+w[i]<=n) ans[i]+=cnt[i+w[i]];
            for(auto j:upd[i]) cnt[j]--;upd[i].clear();
        }
        for(int i=1;i<=n;i++) printf("%d ",ans[i]);
    }
}
```

```
void dfs3(int u,int dep,int fa){
    for(auto v:g[u]){
        if (v==fa) continue;
        dfs3(v,dep+1,u);
        cnt[u]+=cnt[v];
    }
    if (w[u]==dep)
        ans[u]=cnt[u];
}

void case3(){
    for(int i=1;i<=m;i++)
        cnt[a[i].t]++;
    dfs3(1,0,0);
    for(int i=1;i<=n;i++)
        printf("%d ",ans[i]);
}
```

在该题中，我们可以很好地从数据范围直接得出数据点对应的特性，从而选用对应的部分分函数（实现可如下代码所示）。对于一些不直接表明数据对应部分分的题目，我们可能需要进一步对数据进行检查与判断。

```
if (n<1000)
    case1();
else if (m==99994)
    case2();
else if (m==99995)
    case3();
```