枚举优化与搜索

洛谷网校
基础衔接提高计划
吾王美如画

洛谷
www.luogu.com.cn

枚举优化

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。

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：$1 \le n \le 7$

对于 80% 的数据：$1 \le n \le 10$

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

P1118 [USACO06FEB] Backward Digit Sums G/S

- 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。

P1118 [USACO06FEB] Backward Digit Sums G/S

- 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$(0 \le i < n)$，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。

P1118 [USACO06FEB] Backward Digit Sums G/S

```
int tmp[30][30],f[30];
int n,sum;
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();
    tmp[0][0]=1;
    //计算杨辉三角
    for(int i=1;i<n;i++){
        tmp[i][0]=1;
        for(int j=1;j<=i;j++)
            tmp[i][j]=tmp[i-1][j]
                      +tmp[i-1][j-1];
    }
    for(int i=0;i<n;i++)
        f[i]=tmp[n-1][i];//最后一行即为系数
    dfs(0,0);
    return 0;
}
```

P8809 [蓝桥杯 2022 国 C] 近似 GCD

给定一个长度为 n 的数组 A，以及一个正整数 g。

如果最多更改数组中的一个元素之后，数组的最大公约数为 g，那么称 g 为这个数组的近似 GCD。

求数组 A 有多少个长度大于等于 2 的子数组满足近似 GCD 的值为 g。

$2 \leq n \leq 10^5, 1 \leq g, a_i \leq 10^9$

P8809 [蓝桥杯 2022 国 C] 近似 GCD

- 首先考虑最朴素的枚举方法。我们枚举 a 的所有子数组，不妨用 $[l,r]$ 表示，枚举修改的位置，再计算其它位置的 $\gcd$ 。
- 但是我们发现其实没必要计算 $\gcd$ 。一个子数组的近似 $\gcd[l,r]=\mathrm{g}$ 当且仅当 $[l,r]$中不是 g 的倍数的数不超过一个。若只有一个，不妨设其为 a_i ，我们直接将 a_i 改为 g 即可。若均为倍数，任选一个数将其改为 g 即可。这样我们统计 $[l,r]$ 中有多少个数不是 g 的倍数，如果不超过一个则计入答案。这样我们需要 $O(N^3)$ 时间。

P8809 [蓝桥杯 2022 国 C] 近似 GCD

- 进一步的，我们发现没必要先完全确定子数组再统计。对于所有左端点相同的子数组，随着右端点 r 不断右移，不是 g 的倍数的数一定是单调增加的。于是我们只需枚举左端点 l，不断右移右端点 r 并一直记录其中不为 g 倍数的个数 sum 直到其大于 1。不妨设 $[l,r]$ 是最后一个 sum 不超过 1 的子数组，则从 $[l,l+1],[l,l+2],\ldots,[l,r-1],[l,r]$ 这 $r-l$ 个子数组都是合法的子数组。时间复杂度 $O(N^2)$

P8809 [蓝桥杯 2022 国 C] 近似 GCD

- 接着上一步，我们考虑双指针的做法。在上述过程中，我们其实不仅知道了 $[l,l+1],[l,l+2],\ldots,[l,r-1],[l,r]$ 这 $r-l$ 个子数组都是合法的子数组，还能发现 $[l+1,r]$ 必然也是合法的，故当我们计算左端点为 $l+1$ 的最大合法子数组时，可以直接从 $r+1$ 开始尝试右端点，也即意味着我们枚举的这个右端点位置是单调不下降的。总的来说一共只会枚举 $O(n)$ 个右端点，同样左端点固定为 n 次枚举，总共 $O(n)$ 复杂度。

P8809 [蓝桥杯 2022 国 C] 近似 GCD

```cpp
int a[110000],n,g,b[110000];
int main(){
    cin>>n>>g;
    for(int i=1;i<=n;i++)
        cin>>a[i],
        b[i]=!((a[i]%g)==0);//如果不是g的倍数则为1，否则为0
    int sum = 0, r = 0;// 滑动窗口的右端
    ll ans=0;
    for(int l=1;l<n;l++){
        while(r+1<=n&&sum+b[r+1]<=1)//如果右移右端点之后仍然合法
            sum+=b[r+1],r++;//右移右端点
        ans+=r-l;//统计[l,l+1],[l,l+2],...,[l,r]的合法子数组
        sum-=b[l];// 左端点即将右移，更新sum
    }
    cout<<ans<<endl;
    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：$1 \le X_i \le n \le 125$，$1 \le m \le 200$

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

P2119 [NOIP2016 普及组] 魔法阵

- 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$O(N^2)$

P2119 [NOIP2016 普及组] 魔法阵

```
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;
        //枚举a,b并统计各数作为ab的次数
        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;
        //枚举c,d
        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

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

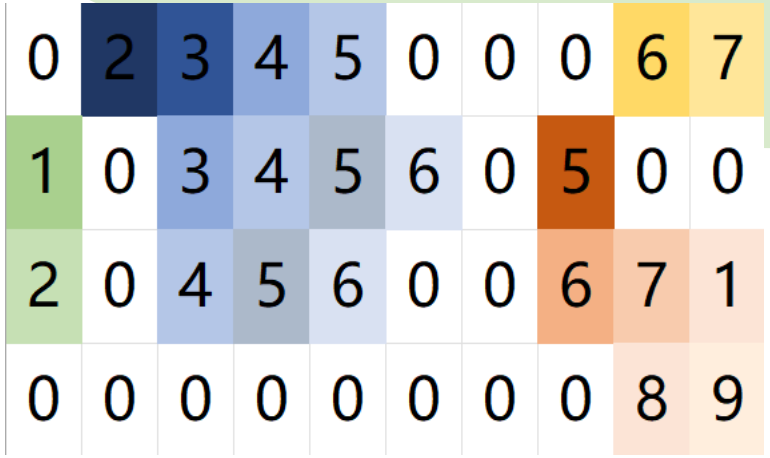
一矩形阵列由数字 0 到 9 组成，数字 1 到 9 代表细胞，细胞的定义为沿细胞数字上下左右若还是细胞数字则为同一细胞，求给定矩形阵列的细胞个数。

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

P1451 求细胞数量

- $Floodfill$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。

0	2	3	4	5	0	0	0	6	7
1	0	3	4	5	6	0	5	0	0
2	0	4	5	6	0	0	6	7	1
0	0	0	0	0	0	0	0	8	9



P1451 求细胞数量

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;
}
```

P1451 求细胞数量

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。

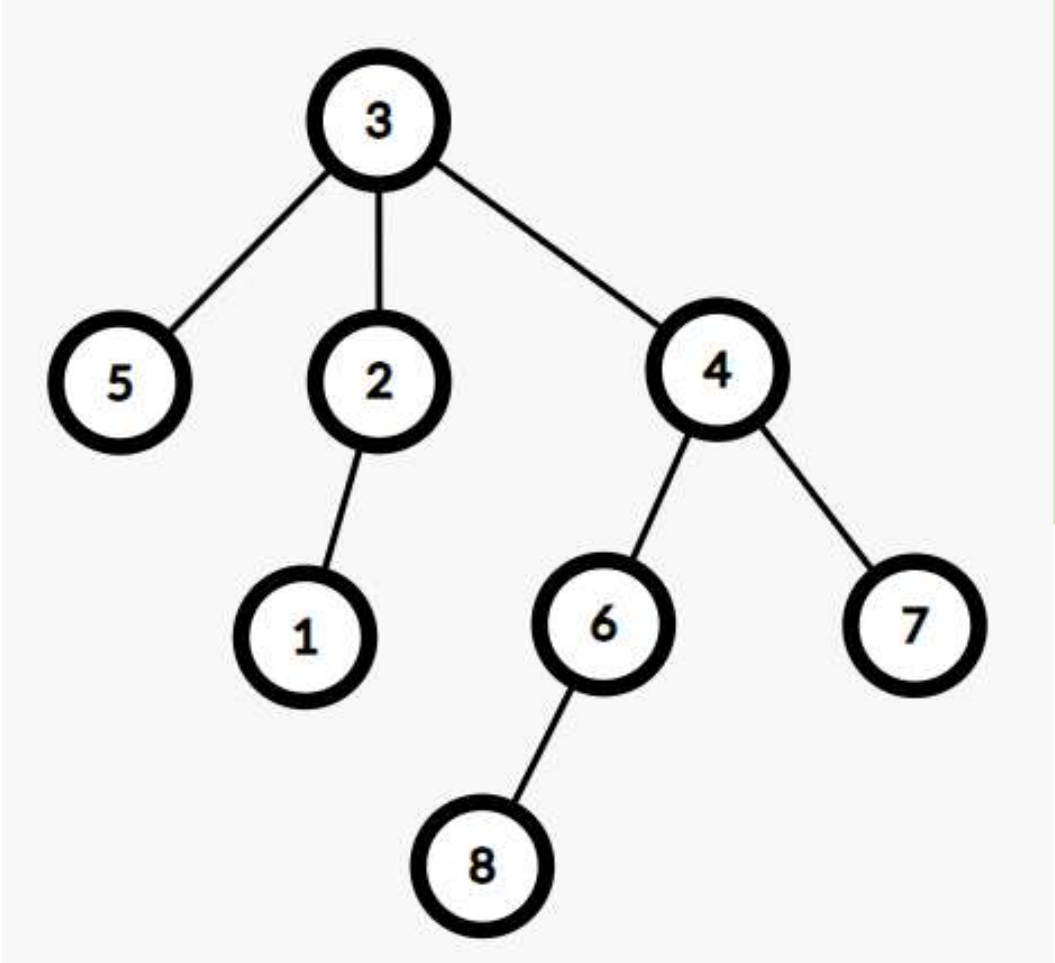
```
void bfs(int x,int y){
    queue<pair<int,int>> q;
    q.push(make_pair(x,y)); vis[x][y]=1;
    while(!q.empty()){
        pair<int,int>now=q.front(); q.pop();
        int u_x=now.first,u_y=now.second;
        for(int i=0;i<4;i++){
            int v_x=u_x+dx[i],v_y=u_y+dy[i];
            if (v_x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

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.

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

（选学）CF1328E Tree Queries

- 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

（选学）CF1328E Tree Queries

- 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$dfn[x], dfn[x]+1, \ldots, dfn[x]+size[x]-1$
- 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。

洛谷

```
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。

$1 \leq N, M \leq 500000$

P3379 【模板】最近公共祖先（LCA）

- 考虑求 u,v $(dfn[u] \le dfn[v])$两点的最近公共祖先w，我们分两种情况讨论
  1. u是v的祖先，则u必是u,v的最近公共祖先
  2. 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。

P3379 【模板】最近公共祖先（LCA）

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。

P3379 【模板】最近公共祖先（LCA）

```
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

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

P10490 Missile Defence System

现有 n 个导弹按顺序袭来，每个导弹有一个高度 h_i 。为了拦截所有导弹，你可以建造多套防御系统。一套防御系统拦截的导弹高度要么一直严格单调上升要么一直严格单调下降，问至少要多少套防御系统

$1 \leq n \leq 50$

P10490 Missile Defence System

- 由于 n 十分的小，我们考虑使用搜索解决。
- 首先考虑最朴素的搜索，对于当前处理到的一个导弹，我们直接枚举使用之前的某个防御装置拦截其，或者为其新建一个防御装置，直到所有导弹都被处理，求其中装置最小的方案。
- 但是由于 n 有 50 的大小，这样做无法通过全部数据，我们考虑进行一些优化。

P10490 Missile Defence System

- 首先是最简单的一个最优化剪枝。我们将当前的最小答案记录下来，如果某个分支使用的装置数已经超过了最小答案，其显然不可能变得更小，从而必然不可能导致更优解，应当剪除。
- 另一方面我们考虑如果现在有两个升序防御装置 a,b，其拦截的最后一个导弹高度分别为 h_a,h_b，并且 $h_a < h_b$。当前导弹高度为 h_i，且 $h_i \geq h_a,h_b$。显然我们使用装置 b 拦截该导弹的方案一定不弱于使用装置 a 拦截。这启发我们如果要用之前的升序防御装置拦截某导弹，选择其中**可行的末位导弹高度最高的一定最优**（使用降序装置同理，选择最低的即可）。这也是一种最优化剪枝

P10490 Missile Defence System

- 更进一步的，我们发现当决定使用升序装置拦截一个导弹时，我们新建一个装置当且仅当**之前的装置无法拦截它**。比如存在一个之前的升序装置 a，其末位为 h_a。此时的导弹为 h_i 且 $h_i > h_a$。如果新建一个升序装置，我们得到了末位分别为 h_i 和 h_a 的两个装置。而将 h_i 用 a 拦截，则只需一个末位为 h_i 的装置，必然更优。（降序同理）这样就足够通过本题

洛谷

```
int n,h[60],ans = 100,incr[60],decr[60];
void dfs(int x,int cnt1,int cnt2){
    if (cnt1+cnt2>=ans) return;//当前已不最优
    if (x==n+1){//更新答案
        ans=min(ans,cnt1+cnt2); return;
    }
    int id,cop=-1e9,tag=0; //使用升序装置
    for(int i=1;i<=cnt1;i++)
        if (incr[i]<h[x]){
            tag=1;//存在现存装置可用
            if (incr[i]>cop) id=i,cop=incr[i];
        }
    if (tag){
        incr[id]=h[x];//更新现存装置
        dfs(x+1,cnt1,cnt2);
        incr[id]=cop;//还原
    }
    else {          //如果没有现存装置可用
        incr[cnt1+1]=h[x];
        dfs(x+1,cnt1+1,cnt2);
    }
    cop=1e9,tag=0; //使用降序装置
    for(int i=1;i<=cnt2;i++)
        if (decr[i]>h[x]){
            tag=1;//存在现存装置可用
            if (decr[i]<cop)id=i,cop=decr[i];
        }
    if (tag){
        decr[id]=h[x];
        dfs(x+1,cnt1,cnt2);
        decr[id]=cop;
    }
    else {
        decr[cnt2+1]=h[x];
        dfs(x+1,cnt1,cnt2+1);
    }
}
int main(){
    while(cin>>n&&n){
        ans = 100;
        for(int i=1;i<=n;i++) cin>>h[i];
        dfs(1,0,0); cout<<ans<<endl;
    }
    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
$1\sim2$	991	991	所有人的起点等于自己的终点，即 $\forall i,\ s_i=t_i$
$3\sim4$	992	992	所有 $w_j=0$
5	993	993	无
$6\sim8$	99994	99994	$\forall i\in[1,n-1]$，i 与 $i+1$ 有边。即树退化成 $1,2,\ldots,n$ 按顺序连接的链
$9\sim12$	99995	99995	所有 $s_i=1$
$13\sim16$	99996	99996	所有 $t_i=1$
$17\sim19$	99997	99997	无
20	299998	299998	无

P1600 [NOIP2016 提高组] 天天爱跑步

- $1\sim2$：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。

- $3\sim4$：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)

P1600 [NOIP2016 提高组] 天天爱跑步

```
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]);
}
```

P1600 [NOIP2016 提高组] 天天爱跑步

- 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 。

P1600 [NOIP2016 提高组] 天天爱跑步

```
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]);
}
```

P1600 [NOIP2016 提高组] 天天爱跑步

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();
```