T1

对于前20分，暴力枚举使用药水的时机即可。

考虑使用反悔贪心，当遇到新的怪物时，尽可能使用药水，如果没有药水可以用，则检查之前用过药水的怪物造成伤害的最小值，若比当前怪物造成的伤害少则改为对当前怪物使用药水，血量减去被反悔的怪物的攻击力。暴力枚举时间复杂度为$O(n^2)$。

使用优先队列维护使用过药水的怪物的攻击力可以将时间复杂度压缩至$O(n\log_2 n)$。

也可以先二分答案，假设可以击败前 c 个怪物，那么从贪心的角度，一定会把伤害最高的 k 个怪物免疫，如果提前离散化，每次二分答案的时候桶排，时间复杂度为 $O(n\log_2 n)$

```cpp
#include<bits/stdc++.h>
using namespace std;
void solve() {
    long long n,x,k;
    scanf("%lld%lld%lld",&n,&x,&k);
    priority_queue<long long,vector<long long>,greater<long long> > pq;
    int ans=-1;
    for(int i=1; i<=n; i++) {
        long long in;
        scanf("%lld",&in);
        if(k) {
            k--;
            pq.push(in);
        } else if(pq.size() && pq.top()<in) {
            x-=pq.top();
            pq.pop();
            pq.push(in);
        } else {
            x-=in;
        }
        if(x<=0 && ans==-1) ans=i-1;
    }
    if(ans==-1) ans=n;
    printf("%d\n",ans);
}
int main() {
    int t;
    scanf("%d",&t);
    while(t--) solve();
    return 0;
}

```

T2

对于前50%的数据，可以对每条边两边的点分别进行一次搜索来统计点数，时间复杂度为$O(n^2)$。

随意选一个点作为根，再以这个点出发构造有根树。在进行dfs时，每搜索完一条边，可以通过树的总节点数和子树的节点数快速计算出这条边两边节点个数的差异。时间复杂度为$O(n)$。

```
ans+=abs(n-2*size[to])*val;
```

完整代码

```
#include<bits/stdc++.h>
using namespace std;
struct edge {
    int to;
    long long val;
};
int n;
long long ans=0;
bool used[1000005];
int size[1000005];
vector<edge> begraph[1000005];
vector<edge> aftgraph[1000005];
void maketree() {
    queue<int> deal;
    deal.push(1);
    while(deal.size()) {
        int indx=deal.front();
        used[1]=1;
        deal.pop();
        for(int i=0; i<begraph[indx].size(); i++) if(!used[begraph[indx]
[i].to]) {
                aftgraph[indx].push_back(begraph[indx][i]);
                used[begraph[indx][i].to]=1;
                deal.push(begraph[indx][i].to);
            }
    }
}
int dfs(int lo) {
    if(!aftgraph[lo].size()) {
        size[lo]=1;
        return size[lo];
    }
    for(int i=0; i<aftgraph[lo].size(); i++) {
        size[lo]+=dfs(aftgraph[lo][i].to);
        ans+=abs(n-2*size[aftgraph[lo][i].to])*aftgraph[lo][i].val;
    }
    size[lo]++;
    return size[lo];
}
int main() {
    cin>>n;
    for(int i=1; i<=n-1; i++) {
        int a,b,c;
        scanf("%d%d%d",&a,&b,&c);
        edge indx;
        indx.to=a;
        indx.val=c;
        begraph[b].push_back(indx);
        indx.to=b;
        begraph[a].push_back(indx);
```

```
    }
    maketree();
    dfs(1);
    cout<<ans;
    return 0;
}
```

选家具

- 考点：**DP**，**贪心**。
- 难度：CSP-J T3+

测试点 1~4

留给暴力搜索的部分。

时间复杂度为 $O(3^n)$ 。

测试点 5~10

留给暴力DP的部分。

设 $f_{i,j,k}$ 考虑了 $1\sim i$ 家店，选了 j 个家电， k 个家具，$i-j-k$ 个装饰的最小费用。

有转移：

$$f_{i,j,k}=\min(f_{i-1,j-1,k}+a_i,f_{i-1,j,k-1}+b_i,f_{i-1,j,k}+c_i)$$

最后答案就是 $f_{n,a,b}$

复杂度为 $O(n^3)$ 。

测试点 11~14

同样可以 n^2 DP。

但也有贪心做法。

我们按 b_i-a_i 给店铺排序，前 b 家点选购家具，后 a 家点选购家电。

为什么？因为若 $b_1-a_1<b_2-a_2$，则 $b_1+a_2<b_2+a_1$。

测试点 15~20

我们可以把上面的贪心做法和dp做法结合起来。

设 $f_{i,j}$ 表示虑了 $1\sim i$ 家店，选了 j 个装饰的最小费用，那么剩下的 $i-j$ 个物品就会按贪心的想法优先选择购买家具，然后购买家电。

按 b_i-a_i 给店铺排序，那么就有转移：

$$f_{i,j}=\begin{cases}\min(f_{i-1,j-1}+c_i,f_{i-1,j}+b_i & (i-j\le B)\\ \min(f_{i-1,j-1}+c_i,f_{i-1,j}+a_i & (i-j>B)\end{cases}$$

时间复杂度是 $O(n^2)$

T4

测试点 1~2

从低到高枚举等级暴力搜索即可。

测试点 3~7

可以发现如果第一个买的游戏编号是x，最后一个买的游戏编号是y，那么能且只能确定从x到y之间所有没买的游戏都一定比买了的游戏等级低。

这样就可以用拓扑排序去做，从能确定的低等级游戏到高等级游戏之间连一条边，每个点的最低等级是所有入度里最高的最低等级加1，最后输出最高的最低等级。

时间复杂度是$O(mn^2)$。

测试点 8~10

注意到我们的时间复杂度主要体现在建边上，一轮浏览建边需要的时间复杂度就有$O(n^2)$。

对于每次建边，我们可以建立一个虚点，所有没有被选上的游戏做虚点的入度，选上的游戏做虚点的出度。

这样一轮连边的时间复杂度为$O(n)$，总时间复杂度为O(mn)，要注意虚点不会增加层数。

```cpp
#include<bits/stdc++.h>
using namespace std;
vector<int> graph[2005];
int deep[2005];
int degree[2005];
int main() {
    int n,m;
    scanf("%d%d",&n,&m);
    int ind=n;
    while(m--) {
        int to=0,from=1919810;
        bool ok[1005];
        for(int i=1; i<=n; i++) ok[i]=0;
        int num;
        scanf("%d",&num);
        for(int i=1; i<=num; i++) {
            int indx;
            scanf("%d",&indx);
            to=max(to,indx);
            from=min(from,indx);
            ok[indx]=1;
        }
        ind++;
        for(int i=from; i<=to; i++) {
            if(ok[i]) {
                graph[i].push_back(ind);
                ++degree[ind];
            } else {
                graph[ind].push_back(i);
                ++degree[i];
            }
```

```
        }
    }
    int ans=0;
    queue<int> deal;
    for(int i=1; i<=n; i++) if(!degree[i]) {
            deal.push(i);
            deep[i]=1;
        }
    while(deal.size()) {
        int indx=deal.front();
        ans=max(ans,deep[indx]);
        deal.pop();
        for(int i=0; i<graph[indx].size(); i++) {
            if(graph[indx][i]>n) deep[graph[indx][i]]=max(deep[graph[indx]
[i]],deep[indx]);
            else deep[graph[indx][i]]=max(deep[graph[indx]
[i]],deep[indx]+1);
            --degree[graph[indx][i]];
            if(!degree[graph[indx][i]]) deal.push(graph[indx][i]);
        }
    }
    cout<<ans;
    return 0;
}
```cpp
```
```