目录

10.2 模拟赛

1. Lucky Permutation

【问题描述】

给定整数 n 和一个 1~n 的排列 p。

你可以对排列 p 进行下列操作任意次：

- 选择整数 $i, j(1 \leqslant i < j \leqslant n)$，然后交换 pi, pj 的值。

你需要求出至少需要进行上述操作多少次才能使 p 恰有一个逆序对。

每个测试点包含 t 组数据。

【输入格式】

第一行输入一个整数 $t(1 \leqslant t \leqslant 104)$ 表示数据组数，接下来对于每组数据：

第一行输入一个整数 $n(2 \leqslant n, \sum n \leqslant 2 \times 105)$。

接下来输入一行 n 个整数 $p1, p2, \cdots, pn(1 \leqslant pi \leqslant n)$，保证 p 是一个 1~n 的排列。

【输出格式】

对于每组数据：

输出一行一个整数表示使 p 恰有一个逆序对所需的最小操作次数。

可以证明一定存在操作方案使得 p 恰有一个逆序对。

【样例输入】

```
4
2
2 1
2
1 2
4
3 4 1 2
4
2 4 3 1
```

【样例输出】

```
0
1
3
1
```

题目链接：https://www.luogu.com.cn/problem/CF1768D

【题目分析】

给定长度为 n 的排列 p，可以选择其中任意两个数进行交换，使得操作完的后排列 p 只存在一对逆序对。

如果序列不存在逆序对，就是序列从小到大排序，如果只存在一个逆序对，一定是将排序后的相邻两个元素进行交换后产生的，即最终的序列一定是由 1，2，…，n 这个序列进行一次相邻元素操作得到的。问题变成了如何采用最少操作将序列排序。

将每个元素现在的位置与目标位置进行连边，即将 pi 与 i 进行连边，发现整个序列变成了多个环，用并查集或 dfs 判环即可，对于长度为 k 的环，只需要 k-1 次操作就能让序列到达目标状态，将所有环的操作数相加总共操作 ans 次就可以让排列有序。

但可能在操作过程中就已经产生只有一个逆序对的序列，而只会在相邻元素的操作中出现，若存在相邻元素在同一环上，当前环就可以减少一次操作次数，答案为 ans-1；否则只能在排序后交换一对相邻元素，答案为 ans+1。

【参考代码】

```
#include <bits/stdc++.h>
using namespace std;
const int maxn=2e5+10;
int t,n,p[maxn],fa[maxn];
int idx[maxn];//idx[i]:数 i 出现的位置
inline int get(int x)
{
```

```
    if(x==fa[x])return x;
    return fa[x]=get(fa[x]);
}
inline void merge(int x,int y)//将 y 设为 x 的父亲
{
    int f1=get(x),f2=get(y);
    if(f1!=f2)fa[f1]=f2;
}
int main()
{
    scanf("%d",&t);
    while(t--)
    {
        scanf("%d",&n);
        for(int i=1;i<=n;i++)scanf("%d",&p[i]),idx[p[i]]=i;
        for(int i=1;i<=n;i++)fa[i]=i;
        for(int i=1;i<=n;i++)merge(i,p[i]);
        int cnt=0,ans=0;//cnt:环的个数
        for(int i=1;i<=n;i++)if(get(i)==i)cnt++;
        ans=n-cnt;
        bool flag=0;
        for(int i=1;i<n;i++)
        {
            if(get(idx[p[i]])==get(idx[p[i+1]]))//相邻两个数在同一个环中
            {
                flag=1;
                break;
            }
        }
        if(flag==1)printf("%d\n",ans-1);
        else printf("%d\n",ans+1);
    }
    return 0;
}
```

2. Divide by three, multiply by two

【问题描述】

有一个长度为 n 的数列 ai，要求你将这个数列重排成一个排列 pi，使得对于任意 $pi(1\leqslant i<n)$：

- $pi\times 2=pi+1$，或者
- 当 pi 可以被 3 整除时， $pi\div 3=pi+1$。

保证答案存在。

【输入格式】

输入包含两行。

第一行一个整数 $n(2\leqslant n\leqslant 100)$，表示数列中的元素个数。

第二行包含 n 个整数 $a1, a2, \cdots, an(1\leqslant ai\leqslant 10\hat{}18)$，描述这个数列。

【输出格式】

输出应包含 n 个整数 $p1, p2, \cdots, pn$，表示你的答案。

【样例输入】

```
6
4 8 6 3 12 9
```

【样例输出】

```
9 3 6 12 4 8
```

题目链接：https://www.luogu.com.cn/problem/CF977D

【题目分析】

方法一：拓扑排序。根据题目描述可以猜结论，整个序列可以用拓扑排序维护，根据题目要求 ai*2=aj 和 ai/3=aj，建立从 i 到 j 的有向边，最终序列一定可以用拓扑排序维护成一个关系序列。但拓扑排序只能处理有向无环图，所以需要证明这个关系不成环，若存在 2^n/3^m * x = 1/2 * x 或 2^n/3^m * x = 3 * x，则可能成环。

首先证明 2^n/3^m*x=1/2*x 不存在，2^(n+1) 与 3^m 显然不同余，则不可能出现最终为环的情况，接下来证明 2^n/3^m * x = 3 * x 不存在，2^n 与 3^(m+1)同样不可能同余，则该情况也不存在。所以整个序列在建边后不可能出现有环的情况，用拓扑排序维护成立。

方法二：观察数据范围，由于 n 只有 100 的范围，所以可以采用填数爆搜，由于题目保证一定存在序列，则最终序列的数一定是由 2 的因子和 3 的因子组成的，所以在填数的时候判断是否能由当前数扩展到下一个数这种逻辑是可以大大减少情况分支的，接下来判断起点位置，根据逻辑关系，作为 start 的数，若 start 为偶数，数列中不应当存在 start/2 和 start*3，若 start 为奇数，则数列中不应当存在 start*3，根据此关系，可以确定起点。然后从起点往后搜索填数即可。

【参考代码】

```
#include<bits/stdc++.h>
using namespace std;
int n,tot,in[101],ans[101],edge[101][101];
long long a[101];
queue<int> q;
void topo()//拓扑
{
    for(int i=1;i<=n;i++)
        if(!in[i])
            q.push(i);
    while(!q.empty())
    {
        int u=q.front();
        q.pop();
        ans[++tot]=u;//ans 数组存下标
        for(int i=1;i<=n;i++)
            if(edge[u][i])
            {
                in[i]--;
                if(!in[i])
                    q.push(i);
            }
    }
    for(int i=1;i<=n;i++)
        printf("%lld ",a[ans[i]]);
}
int main()
{
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
        scanf("%lld",&a[i]);
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            if((a[i]*2==a[j])||(a[i]%3==0&&a[i]/3==a[j]))//建图
            {
                edge[i][j]=1;
```

```
                in[j]++;
            }
    topo();
    return 0;
}
```

3. Parking Lot

【问题描述】

停车场共有 2*n*-2 个停车位。共有 4 种品牌的汽车，每种汽车的数量都远大于停车位的数量。

该公司首席执行官认为，如果停车场有**恰好** *n* 个连续汽车的品牌相同，则停车场会更漂亮。

给定 n 的值，问有多少的方案使停车场满足条件。

【输入格式】

一行，包含一个整数 *n*（3≤*n*≤30）。

【输出格式】

输出一个整数，即总方案数。

【样例输入】

```
3
```

【样例输出】

```
24
```

题目链接：https://www.luogu.com.cn/problem/CF630I

【题目分析】

利用乘法原理和加法原理统计情况数量，利用快速幂求答案。首先确定汽车放置逻辑，首先确定恰好 n 个连续相同的汽车品牌，共有 4 种情况，然后确定这 n 辆汽车排列的起点和终点位置，由于起点或中点在端点处，与在中间处的方案数量不同，所以分类讨论。

若这 n 辆汽车的起点和终点在中间，则两边都必须放与当前汽车品牌不同的汽车，这种情况下共有 n-3 个起点位置，两边车辆方案为 3^2，余下的 n-4 个位置可以随便摆放车辆，则这种情况下的方案为 4*(n-3)*3^2*4^(n-4)。

若这 n 辆汽车的起点或终点在端点处，即起点在位置 1 或终点在 2n-2 处，则只有一边需要放与当前汽车品牌不同的汽车，则一边的车辆方案为 3，余下的 n-3 个位置可以随便摆放车辆，则这种情况下的方案为 4*2*3*4^(n-3)。

最后利用快速幂求和即可。答案溢出 int 但不会溢出 long long，注意数据范围。

【参考代码】

```
#include <bits/stdc++.h>
using namespace std;
long long n;
long long qpow (long long x, long long y) {
    long long ans = 1;
    while (y > 0) {
        if (y & 1) ans *= x;
        x *= x;
        y >>= 1;
    }
    return ans;
}
int main() {
   cin >> n;
   cout << 24 * qpow(4, n - 3) + 36 * (n - 3) * qpow(4, n - 4) << endl;
   return 0;
}
```

4. Portals

【问题描述】

你在玩一款游戏。

在这个游戏里面，你掌控着一只大军，你的目标是占领全部的 n 个敌方的城堡。

让我们详细的介绍一下这个游戏。

最开始，你掌控着一只有 k 个人的军队，你的敌人有 n 个城堡。

为了占领第 i 个城堡，你需要至少 ai 个士兵（你游戏玩的很好，所以你并不会损失士兵）。

当你占领了第 i 个城堡后，你可以扩军。其中在第 i 个城堡，你可以得到 bi 个士兵。

此外，在你占领了一个城堡后，你可以派至少一个兵驻守。每个城堡有一个重要值 ci，你的总分就是所有你派兵驻守了的城堡的 ci 的和。

有两种方式派兵：

- 你刚占领这个城堡，可以立马派兵驻守。
- 有 m 条小路，其中第 i 条是从 u 通往 v。保证 $u>v$ 。你可以使用这条小路当且仅当你当前在 u，且拥有至少一个士兵可以派往 v 。

显然，如果一个士兵被派去驻守，他就离开了你的军队。

你需要按照从 1 到 n 的顺序依次攻打。

假设你当前在 i，当你攻占 i+1 时，所有在 i 处的小路就不能用了。

如果在游戏中你没有足够的士兵去攻打下一座城堡，你就输了。

你的目标是在胜利的前提下最大化得分。

【样例输入】

```
4 3 7
7 4 17
3 0 8
11 2 0
13 3 5
3 1
2 1
4 3
```

【样例输出】

```
5
```

题目链接：https://www.luogu.com.cn/problem/CF1271D

【题目分析】

核心思想：对于每一个城堡，我们都尽可能在最晚的时间驻守。

方法一：动态规划，观察数据范围，可以发现任何情况下士兵数量都是小于等于 5000 的，因此可以将问题转化为背包。考虑 dp[i][j]表示前 i 个城堡当前军队有 j 人的情况下，控制的最高分数。则在 j>=ai 时，一定有 dp[i][j]=dp[i-1][j-b[i]],对于当前能驻守的每个城堡 k，都有 dp[i][j] = max(dp[i][j],dp[i][j+1]+c[k])。利用滚动数组可以优化掉一维 i，时间复杂度 O(n^2)。

方法二：数据结构优化贪心。考虑这样一个贪心思想：先不驻守任何一个城堡，完成游戏后再按照价值从大到小的顺序尽可能地驻守城堡。由于驻守任何一个城堡需要的士兵都是 1 人。如果留着这个价值较大的城堡不驻守，对后续的影响最多也就是“能多驻守一个价值较小的城堡”，这一定是不划算的。

接下来就需要判断“能否驻守这个城堡”，考虑驻守这个城堡对后续的影响，驻守了这个城堡，后面占领城堡时可用士兵就会少一人。因此我们可以用一个数组记录“占领这个城堡时，可用士兵比必需士兵(a[i])多多少，即盈余士兵”，假设我们想要在 t 时刻驻守一个城堡，那么 t 时刻之后的可用士兵就会少一人，我们就要检查，t 时刻之后的“盈余士兵”是否都大于 0，如果确实大于 0，表示我们可以去驻守这个城堡，并且更新“盈余士兵”，将 t 时刻之后的盈余士兵都减少一人。需要区间减以及区间最小值的操作，可以利用线段树进行实现，时间复杂度 O(nlogn)。

方法三：反悔贪心。反悔贪心同样是基于“驻守任何一个城堡需要的士兵都是 1 人”的条件。它的思路是在每个城堡的“最晚驻守点”，先贪心地把城堡驻守。
这样可能会导致后面的士兵数量不够，游戏失败。此时我们就进行后悔，“要是那个时候没有驻守这个城堡就好了”；而我们要实现“时光倒流”，把当时的操作反悔，也就是减少 c[i]个分数，增加一名士兵。此时我们要进行反悔的城堡，肯定是最不重要的那个，即 c[i]最小的城堡。
我们只需要在驻守城堡的时候，把城堡放到一个堆里，每次要反悔的时候从堆中取出最不重要的城堡进行反悔即可，如果需要反悔时堆是空的，就说明游戏失败。注意在游戏结束时，如果士兵人数为负，则还要进行一次反悔。时间复杂度 O（nlogn）

【参考代码 1】

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;

const int N=3e5+10;

int a[N],b[N],c[N],last[N];
vector<int>contribution[N];

int main() {
    int n,m,k;
    cin>>n>>m>>k;
    for(int i=1;i<=n;i++) scanf("%d%d%d",&a[i],&b[i],&c[i]),last[i]=i;
    int u,v;
    while(m--) scanf("%d%d",&u,&v),last[v]=max(last[v],u);
    for(int i=1;i<=n;i++)
        contribution[last[i]].push_back(c[i]);
    priority_queue<int ,vector<int>, greater<int> >q;
    int sum=k,ans=0;
    for(int i=0;i<=n;i++) {
        sum+=b[i];
        for(auto &v:contribution[i])
            sum--,q.push(v);

        while(!q.empty()&&sum<a[i+1])
            sum++,q.pop();

        if(sum<a[i+1]) printf("-1"),exit(0);
    }
    while(!q.empty())
        ans+=q.top(),q.pop();
    cout<<ans;
}
```

贪心

【参考代码 2】

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;

const int inf=0x3f3f3f3f;
const int N=5e5+100;
```

```
int dp[6000],a[N],b[N],c[N];
vector<int>contribution[N];
int last[N];
int main () {
    int n,m,k;
    cin>>n>>m>>k;
    for(int i=1;i<=n;i++) {
        scanf("%d%d%d",&a[i],&b[i],&c[i]);
        last[i]=i;
    }
    int u,v;
    while(m--) {
        scanf("%d%d",&u,&v);
        last[v]=max(last[v],u);
    }
    for(int i=1;i<=n;i++) contribution[last[i]].push_back(c[i]);
    memset(dp,-1,sizeof dp);
    dp[k]=0;

    for(int i=1;i<=n;i++) {
        for(int j=0;j<a[i];j++) dp[j]=-1;

        for(int j=5500;j>=b[i];j--) dp[j]=dp[j-b[i]];

        for(auto &v:contribution[i])
            for(int j=0;j<=5500;j++)
                if(dp[j+1]!=-1)
                    dp[j]=max(dp[j],dp[j+1]+v);
    }
    cout<<*max_element(dp,dp+5500);
}

dp
```