2019-3-9模拟赛解题报告——*Trump*

1. Robot *字符串*

简简单单的一道字符串，或者说是数学题，大概就是读入命令后去模拟就好了，但需要注意的是，当T过大，需要执行的命令次数过多的时候如果只是完全一个一个读入的话大概是会TLE的，所以可以把完整地执行一遍命令后的数据变化量读出来，然后操作数直接除以一串命令的数量乘上变化量就好了，但不要忘了最后剩下的部分记得一个一个去走，上代码：

```
#include <cstring>
#include <iostream>
#include <cstdio>
using namespace std;
int main(){
    std::ios::sync_with_stdio(false);
    cout.tie(0);
    freopen("robot.in","r",stdin);
    freopen("robot.out","w",stdout);
    string com;
    int X,Y,t,times,len;
    X=Y=0;
    cin>>com; //读入字符串
    len=com.length(); //字符串长度
    for(int i=0;i<len;i++){//先走一遍确定以执行一遍命令的变化量
        if(com[i]=='W') X--;
        if(com[i]=='E') X++;
        if(com[i]=='N') Y++;
        if(com[i]=='S') Y--;
    }
    cin>>t; //操作数
    times=t/len; //算出完整命令地执行次数
    t%=len; //最后剩下的个数
    X*=times;Y*=times; //执行完整命令的直接变化量乘上次数
    for(int i=0;i<t;i++){ //对于剩下地再一步一步操作
        if(com[i]=='W') X--;
        if(com[i]=='E') X++;
        if(com[i]=='N') Y++;
        if(com[i]=='S') Y--;
    }
    cout<<X<<" "<<Y<<endl;
}
```

2. seq *矩阵加速*

矩阵加速，这是个学过马上就会做，没学过绝对不会做的玩意儿，原理是利用矩阵的乘法，从我自己十分卑微的角度理解，并且说得玄学一点，矩阵乘法实际上是一种面向数的变化，将某几个数通过一种固有的规则以及玄学的变幻变为另几个数，这种变换通常是以加法和乘法为基础的（估计是我菜不知道更多的），显然就是这道题要求的，与之类似的还有斐波那契数列之类的递推数列，那么问题来了。

矩阵到底是嘛玩意儿？好吧这个不在我，这个蒟蒻可以解释的范围之内了，自行百度百科或者去看洛谷模板的题解大佬们吧(P1939)，我还是就题论题好了

首先通过题目要求的条件`a[x]=a[x-3]+a[x-1]`可以~~轻而易举~~地写出变换用的3*3矩阵：

```
0 1 0          1
0 0 1          1
1 0 1          1
```

嗯就是上面左边（令作A）这玩意儿，右边（令作B）的自然是初始时的元矩阵。也就是说我们只要用左边这个数列疯狂地去乘右边那玩意儿以及乘出来的新的3*1矩阵就可以得到题目所要求的了，当然不要忘了$A^n * B$中的第三项得到的是整个数列的第n+3项，因此就只需要再用一下下快速幂就可以A掉啦，所以上代码：

```
#include <iostream>
#include <cstdio>
#include <cstring>
#define rep(i,l,r) for(int i=(l);i<=(r);i++)
#define per(i,l,r) for(int i=(l);i>=(r);i--)
#define MOD 1000000007
using namespace std;
typedef long long ll;
struct Mat{ //把矩阵用结构体存储
    ll cont[5][5]; //这里其实只需要用到3*3
    int w,h; //矩阵的列数w和行数h
    Mat operator*(const Mat &b) const{ //重新定义矩阵乘法
        Mat c;
        c.h=this->h;
        c.w=b.w;
        rep(i,1,c.h)
            rep(j,1,c.w){
                rep(k,1,this->w)
                    c.cont[i][j]+=(this->cont[i][k]*b.cont[k][j])%MOD;
                    //MOD可以直接在运算过程中处理好了
                c.cont[i][j]%=MOD;
            }
        return c;
    }
    Mat(){
        w=h=0;
        memset(cont,0,sizeof(cont));
    }
};
Mat change;
Mat qPow(Mat base, int q){ //快速幂
    Mat ans;
    ans.w=ans.h=base.h;
    rep(i,1,base.h) ans.cont[i][i]=1;
    if(q==0) return ans;
```

```
    while(q>0){
        if(q&1) ans=ans*base;
        base=base*base;
        q>>=1;
    }
    return ans;
}
int main(){
    std::ios::sync_with_stdio(false);
    cout.tie(0);
    freopen("seq.in","r",stdin);
    freopen("seq.out","w",stdout);
    Mat e; //定义文中的矩阵B
    int n,T;
    e.w=1;e.h=3;
    e.cont[1][1]=e.cont[2][1]=e.cont[3][1]=1; //上面说的右边的那玩意儿（矩阵B）
    change.w=change.h=3;
    change.cont[1][2]=change.cont[2][3]=change.cont[3][1]=change.cont[3][3]=1;
    //change是变换矩阵
    cin>>T;
    rep(i,1,T){
        Mat t;
        cin>>n;
        if(n==1||n==2||n==3) {cout<<1<<endl;continue;}
        //这里一个小小的特判，前三个的情况
        t=qPow(change,n-3)*e; //快速幂，记得n-3
        cout<<t.cont[3][1]%MOD<<endl;
    }
}
```

3.holes *图——最短路*

其实吧就是一个最短路问题，边权的计算比较花里胡哨而已，***问题不是特别大，真的***，对于题中的每个洞的两个状态（黑和白），其实可以直接看成两个不同的互相连接的洞，而这两个洞之间的边权实际上也就是飞船停留时的消耗问题，然后就可以按照题目地要求开始建图了。

建图的时候需要相当的仔细，因为这道破题的限制条件是真的多，鬼知道哪里一个符号打错了就打死都调不出来（留下了调了两节课的泪水）。建图的时候我们要分清楚需要连起来的到底是洞的哪个状态，事实上如果有一条从u->v的单向路径，u的当前状态要连接的其实是v的下一个状态，因为黑白洞是会转化的，也就是说假设u和v都是黑洞，那么在图上就应该把u的黑洞状态连接的v的白洞状态，因为当飞船从u飞到v的时候，v已经转换成了白洞，想清楚这个之后题目其实就明了了，至于是否要加减delta只要看u和v的初始状态是否相同就好了，当然不要忘了每个洞的两个状态相连，因为飞船是可以停留的嘛。

建完图之后就可以走一遍最短路了，至于最短路的算法用SPFA或者Dijkstra都可以，因为这道题飞船燃料消耗的下限是0，不涉及到负权边，最后只要取第n个洞的两个状态的消耗最小值就好了（因为到达目的地时第n个洞可能是黑的也可能是白的），so先放上程序看看吧：

```
//SPFA版本
#include <iostream>
#include <cmath>
#include <cstdio>
#include <vector>
```

```cpp
#include <queue>
#include <cstring>
#define rep(i,l,r) for(int i=(l);i<=(r);i++)
#define per(i,l,r) for(int i=(l);i>=(r);i--)
#define NMAX 5000
#define MMAX 30000
using namespace std;
typedef long long ll;
int n;
struct hole{
    int w;
    int dis;
    hole(){
        w=0;
        dis=1000000007; //初始化dis（到源点距离）为无穷大
    }
} h[2*NMAX+5]; //定义黑白洞，范围到2*NMAX是因为一个洞有两个状态，也就算两个洞
struct edge{ //用结构体来存储边了
    int to;
    int w;
};
vector<edge> e[2*NMAX+5];//建图嘛我用的是vector写的伪邻接表
inline void createEdge(int a, int b, int s){ //a到b建一条长为s的有向边
    edge c;
    c.to=b;
    c.w=s;
    e[a].push_back(c);
}

void spfa(int root){ //spfa算法
    queue<int> q; //stl是个好东西....
    bool isInQ[2*n+5]={}; //用来判断某个点是否已在队列中
    h[root].dis=0;
    q.push(root);
    isInQ[root]=1;
    while(!q.empty()){
        root=q.front();q.pop();
        isInQ[root]=0;
        rep(i,0,(int)e[root].size()-1){ //更新每个点连向的节点
            int to=e[root][i].to;
            int w=e[root][i].w;
            if(h[root].dis+w<h[to].dis){ //可以缩边就索
                h[to].dis=h[root].dis+w;
                if(!isInQ[to]){
                    q.push(to);
                    isInQ[to]=1;
                }
            }
        }
    }
}
int main(){
    std::ios::sync_with_stdio(false);
```

```
    cout.tie(0);
    freopen("holes.in","r",stdin);
    freopen("holes.out","w",stdout);
    bool isBlack[NMAX+5]={};
    int m,stay;
    int u,v,k,delta;
    cin>>n>>m;
    rep(i,1,n)
        cin>>isBlack[i]; //读入初始状态
    rep(i,1,n){
        cin>>h[2*i-1].w;
        h[2*i]=h[2*i-1]; //读入点的质量，其实这句有点多余，但是没什么关系..
    }
    rep(i,1,n){
        cin>>stay;
        createEdge(2*i-1,2*i,stay); //2*i-1为洞i的黑洞状态，2*i是白洞状态
        createEdge(2*i,2*i-1,0); //黑->白停留需要消耗stay，白->黑不用
    }
    rep(i,1,m){
        cin>>u>>v>>k;
        if(isBlack[u]==isBlack[v]){
            //如果u和v初始状态相同就不用care delta了
            createEdge(2*u-1,2*v,k);
            createEdge(2*u,2*v-1,k);
        } else { //u和v初始状态不同
            delta=abs(h[2*u].w-h[2*v].w);
            //我就是这里的2*u漏了调了半天，后悔自己为什么要用结构体
            createEdge(2*u-1,2*v-1,k+delta);
            //u的黑洞状态连到v的白洞状态的下一状态（v的黑洞状态），边权增加delta
            createEdge(2*u,2*v,(k-delta>0 ? k-delta : 0));
            //u的白洞状态连到v的黑洞状态的下一状态（v的白洞状态），边权减少delta
            //注意一下0的问题
        }
    }
    int start=isBlack[1]?1:2; //第一个洞的黑白决定最短路的源点是哪个
    spfa(start);
    cout<<min(h[2*n].dis,h[2*n-1].dis)<<endl;//输出洞n的两个状态中小的那个
}
```

再放上来一个Dijkstra的版本吧（函数），别的地方一样：

```
void dijkstra(int root){
    priority_queue<int,vector<int>,greater<int> > q;
    //优先队列优化Dijkstra
    bool vis[2*n+5]={};//Dijkstra中每个点是只要访问一次就够的来着
    h[root].dis=0;
    q.push(root);
    while(!q.empty()){
        root=q.top();q.pop();
        if(vis[root]) continue; //访问过就跳过
        vis[root]=true;
        rep(i,0,(int)e[root].size()-1){
            int to=e[root][i].to;
```

```
            int w=e[root][i].w;
            if(h[root].dis+w<h[to].dis){
                h[to].dis=h[root].dis+w;
                q.push(to);
            }
        }
    }
}
```

OVER~ 最后的最后白色情人节祝各位脱单呐