2020 年 ACM/CCPC 程序设计竞赛校内赛(网赛) 正式赛 题解

A. Yet Another Find Your Miku

题目要求 字符画 0 和 字符画 1 的个数，那么我们怎么让程序自动识别 0 和 1 的字符画呢？观察字符画 0 和 1，不难发现每个字符画 0 中间都有一个空洞，而 1 却没有，且不管图形怎样拉伸，这一特性都不会改变。因此我们考虑对所有边缘使用一次种子填充（floodfill），这等于将背景刷上了别的颜色。接下来我们遍历全图，寻找黑色连通块，将黑色连通块的个数计入 sum1 以后，也将黑色连通块进行 floodfill 填充成背景色。最后我们再遍历一遍全图，寻找白色连通块进行 floodfill 填充成背景色，同时将白色连通块个数计入 sum0 即可。最后的坐标即为 (sum0,sum1-sum0)。

代码：

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
typedef pair<int,int> P;
#define _for(i,a,b) for(register int i = (a);i < b;i ++)
#define _rep(i,a,b) for(register int i = (a);i > b;i --)
#define INF 0x3f3f3f3f
#define ZHUO 11100000007
#define SHENZHUO 1000000007
#define pb push_back
#define debug() printf("Miku Check OK!\n")
#define maxn 1039

int n, m;
char in[maxn][maxn];
bool vis[maxn][maxn];
int sum0 = 0, sum1 = 0;
bool valid(int x,int y)
{
    return x>=0 && x<n && y>=0 && y<m;
}
//eight directions
const int dx[] = {-1,1,0,0};
const int dy[] = {0,0,1,-1};
//flood fill
void ff1(int x,int y)
{
    queue<P> q;
    q.push({x,y});
```

```
        vis[x][y] = true;
        in[x][y] = 39;
        while(!q.empty())
        {
            x = q.front().first;
            y = q.front().second;
            q.pop();
            _for(i,0,4)
            {
                int nx = x+dx[i];
                int ny = y+dy[i];
                if(valid(nx,ny) && !vis[nx][ny] && in[nx][ny] == '.')
                {
                    vis[nx][ny] = true;
                    in[nx][ny] = 39;
                    q.push({nx,ny});
                }
            }
        }
}

void ff2(int x,int y)
{
        queue<P> q;
        q.push({x,y});
        vis[x][y] = true;
        in[x][y] = 39;
        while(!q.empty())
        {
            x = q.front().first;
            y = q.front().second;
            q.pop();
            _for(i,0,4)
            {
                int nx = x+dx[i];
                int ny = y+dy[i];
                if(valid(nx,ny) && !vis[nx][ny] && in[nx][ny] == '*')
                {
                    vis[nx][ny] = true;
                    in[nx][ny] = 39;
                    q.push({nx,ny});
                }
            }
        }
```

```
}
int main()
{
	scanf("%d%d",&n,&m);
	_for(i,0,n)
	_for(j,0,m)
	scanf(" %c",&in[i][j]);

	memset(vis,0,sizeof(vis));
	_for(i,0,n)
	_for(j,0,m)
	if( (!i || !j || i==n-1 || j==m-1)
			&& !vis[i][j] && in[i][j]=='.' )
		ff1(i,j);

	memset(vis,0,sizeof(vis));
	_for(i,0,n)
	_for(j,0,m)
	if(!vis[i][j] && in[i][j]=='*')
		sum1 ++,ff2(i,j);

	memset(vis,0,sizeof(vis));
	_for(i,0,n)
	_for(j,0,m)
	if(!vis[i][j] && in[i][j]=='.')
		sum0 ++,ff1(i,j);

	sum1 -= sum0;
	printf("%d %d",sum0,sum1);
	return 0;
}
```

B. ISLANDERS

读题后不难发现，因为需要将建筑物全都摆放在岛上，因此最终得分只与摆放的顺序有关。我们首先考虑最暴力的解法，即对于一共 $a+b+c$ 次放置，每一次放置都选择其中一种建筑，计算得分后再考虑下一次放置。可以考虑使用记忆化剪枝，即当前已存在的三种建筑物数量完全一致时，若当前得分没有已有方案中的最大得分大，则可以回溯，时间复杂度为$O(3^{(a+b+c)})$ 。剪枝后可大幅度优化时间，但对于题目最大数据仍无法通过。

代码：

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

```
#define _for(i,a,b) for(register int i = (a);i < b;i ++)
#define _rep(i,a,b) for(register int i = (a);i > b;i --)
#define INF 0x3f3f3f3f
#define ZHUO 1000000007
#define pb push_back
#define maxn 100
#define debug() printf("Miku Check OK!\n")
int Get[5][5];
int ms[maxn][maxn][maxn];
int a, b, c;
void dfs(int ai,int bi,int ci,int val)
{
    //记忆化剪枝
    if(val < ms[ai][bi][ci])
    return ;
    ms[ai][bi][ci] = val;
    //考虑三种放置可能
    if(ai+1 <= a)
    dfs(ai+1,bi,ci,val+ai*Get[0][0]+bi*Get[0][1]+ci*Get[0][2]);
    if(bi+1 <= b)
    dfs(ai,bi+1,ci,val+ai*Get[1][0]+bi*Get[1][1]+ci*Get[1][2]);
    if(ci+1 <= c)
        dfs(ai,bi,ci+1,val+ai*Get[2][0]+bi*Get[2][1]+ci*Get[2][2]);
}
int main()
{
    // freopen("data.in","r+",stdin);
    // freopen("bf.out","w+",stdout);
    memset(ms,-0x3f,sizeof(ms));
    scanf("%d%d%d",&a,&b,&c);
    _for(i,0,3)
    _for(j,0,3)
    scanf("%d",&Get[i][j]);
    dfs(0,0,0,0);
    printf("%d\n",ms[a][b][c]);
    return 0;
}
```

继续考虑刚才的记忆化剪枝，当已经摆放的三种建筑数量确定时，实际上我们只关心此时的最大得分，而对于某种三种建筑物数量都确定的情况，事实上只有在上一步放了一个A or B or C 三种可能，因此我们不难得出状态转移方程：

dp[ai][bi][ci] 表示当前已放置了 ai 个 A 建筑，bi 个 B 建筑，ci 个 C建筑的最大得分。

初值：dp[0][0][0] := 0,其余为 -INF。

转移方程：dp[ai][bi][ci] = max(dp[ai-1][bi][ci] + affectA(ai-1,bi,ci)

, dp[ai][bi-1][ci] + affectB(ai,bi-1,ci)

, dp[ai][bi][ci-1] + affectC(ai-1,bi,ci-1))

其中affectA(x,y,z)代表放置一个建筑A且已有x个A，y个B，z个C时的得分，affectB与affectC同理。

结果：dp[a][b][c]

于是我们不难得出动态规划解法：

代码：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef pair<ll, ll> P;
#define _for(i,a,b) for(register int i = (a);i < b;i ++)
#define _rep(i,a,b) for(register int i = (a);i > b;i --)
#define INF 0x3f3f3f3f
#define ZHUO 1000000007
#define pb push_back
#define maxn 100
#define debug() printf("Miku Check OK!\n")
int Get[5][5];
int dp[maxn][maxn][maxn];
int main()
{
    // freopen("data.in","r+",stdin);
    // freopen("sol1.out","w+",stdout);
    memset(dp, -0x3f, sizeof(dp));
    dp[0][0][0] = 0;
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    _for(i, 0, 3)
        _for(j, 0, 3)
        scanf("%d", &Get[i][j]);
    int tot = a + b + c;
    _for(i, 1, tot + 1)
        _for(ai, 0, a + 1)
        _for(bi, 0, min(b + 1, i - ai + 1))
    {
        int ci = i - ai - bi;
        if (ci<0 || ci>c)
            continue;
        //三种转移
```

```
            if (ai)
                dp[ai][bi][ci] = max(dp[ai][bi][ci],
                    dp[ai - 1][bi][ci] + Get[0][0] * (ai - 1) + Get[0][1] * bi + Get[0][2] * ci);
            if (bi)
                dp[ai][bi][ci] = max(dp[ai][bi][ci], dp[ai][bi - 1][ci] + Get[1][0] * ai + Get[1][1] *
(bi - 1) + Get[1][2] * ci);
            if (ci) dp[ai][bi][ci] = max(dp[ai][bi][ci], dp[ai][bi][ci - 1] + Get[2][0] * ai + Get[2][1] *
bi + Get[2][2] * (ci - 1));
        }
        printf("%d\n", dp[a][b][c]);
        return 0;
}
```

时间复杂度为$O((a+b+c)^3)$，实际上远远达不到这个上界，对于题目的数据范围完全够用。

于是我们就能愉快的Accept了，可是这样足够了吗？

我们换个角度思考一下，房屋到底是按照什么顺序放置的？如果对于一组数据，我们假设建筑物B的数量为0，也就是目前只有AC种建筑，那么放置ACACA的分数到底是怎么计算的呢？不难得出，

$$point(ACACA) = affectC(1,0,0) + affectA(1,0,1) + affectC(2,0,1) + affectA(2,0,2) = affectA(3,0,3) + affectC(3,0,1)$$

又因为

$$point(AAACC) = affectA(3,0,0) + affectC(6,0,1)\ point(CCAAA) = affectA(3,0,6) + affectC(0,0,1)$$

我们惊奇的发现参数之和居然是一样的！而且自己对自己建筑物的影响是固定的，也就是当题目中　　固定时，$affectA()$ 的第一个参数是恒定的，而当影响函数的两个参数都固定时，$affectA(x,x,y)$ 与 $affectC(y,x,x)$ 的大小可以通过比较题目中所给矩阵中的值得出！因此有

$$\text{point(ACACA)} \le \max(\text{point(AAACC)}, \text{point(CCAAA)})$$

推广一下可知若建筑物B的数量不为0同理

因此只需要考虑放置建筑物的**种类**即可，我们可以枚举建筑物种类放置的全排列，因为题目中种类数量恒等于 3，因此该贪心算法时间复杂度为$O(3!) = O(6) = O(1)$

代码：

```
#include <bits/stdc++.h>


using namespace std; typedef long long ll; typedef pair<ll, ll> P;
#define _for(i,a,b) for(register int i = (a);i < b;i ++) #define _rep(i,a,b) for(register int i = (a);i > b;i
--) #define INF 0x3f3f3f3f
#define ZHUO 1000000007
#define pb push_back #define maxn 100
#define debug() printf("Miku Check OK!\n")
```

```
int Get[5][5]; int type[5];

int main()
{
	//	freopen("data.in","r+",stdin);
	// freopen("sol2.out","w+",stdout);


	_for(i, 0, 3)
		scanf("%d", &type[i]);
	_for(i, 0, 3)
		_for(j, 0, 3)
		scanf("%d", &Get[i][j]);


	int rnt = -INF; int base = 0;
	//固定不变的价值
	base += ((type[0] * (type[0] - 1)) / 2)*Get[0][0];
	base += ((type[1] * (type[1] - 1)) / 2)*Get[1][1];
	base += ((type[2] * (type[2] - 1)) / 2)*Get[2][2];


	int circle[4] = { 0,1,2 }; do
	{
		int trnt = base;
		trnt += type[circle[1]] * type[circle[0]] * Get[circle[1]][circle[0]]; trnt += type[circle[2]]
* type[circle[0]] * Get[circle[2]][circle[0]]; trnt += type[circle[2]] * type[circle[1]] *
Get[circle[2]][circle[1]]; rnt = max(rnt, trnt);
	} while (next_permutation(circle, circle + 3));


	printf("%d\n", rnt); return 0;
}
```

C. POP TEAM shoryu

读题后不难发现，虽然是顺序打压黑恶势力，但是其实你可以任意挑选黑恶势力来打压，因为最后算的是总名望值，并且黑恶势力毫无还手之力，所以只要不断打下去一定会被全部打败，自然每个黑恶势力也就遍历得到。

对于某个黑帮一开始自然可以混合双打，直到该黑帮被打到 只要 PIPI 美来一下，POP 子来一下，它就会瓦解的程度，然后你看看 PIPI 美能不能过来把它瓦解掉，如果不能，其实就是要让 POP 子住手然后 PIPI 美上去再 A 一下。所以我们可以计算出对于每个黑帮，PIPI 美让 POP 子停手几次才能获得名望值。用刚才的策略就可以获得最小的消耗。

每个黑帮有一个消耗值和一个名誉值，你有一个总消耗次数... 典型 0-1 背包问题。模板一套，AC 拿到。

代码：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef double db;
#define _for(i,a,b) for(int i = (a);i < b;i ++)
#define _rep(i,a,b) for(int i = (a);i > b;i --)
#define INF 0x3f3f3f3f
#define ZHUO 11100000007
#define MOD 1000000007
#define MIKUNUM 39
#define pb push_back
#define debug() printf("Miku Check OK!\n")
#define maxn 200039
#define X first
#define Y second

int n, a, b, k;
int p[maxn];
int w[maxn];
struct Pack
{
    int dp[maxn];
    int V;
    void ZeroOnePack(int cost,int weight)
    {
        for(int i=V; i >= cost; i--)
            dp[i]=max(dp[i],dp[i-cost]+weight);
    }
}P;
int main()
{
    ios::sync_with_stdio(false);
    cin >> n >> a >> b >> k;
    _for(i,1,n+1)
    {
        cin >> p[i];
        int t1 = p[i]/(a+b);
        p[i] -= max(0,t1)*(a+b);
        if(!p[i])
            p[i] = a+b;
```

```
            if(a>=p[i] || !(p[i]%a))
                p[i] = max(0,p[i]/a-1);
            else
                p[i] /= a;
        }
        _for(i,1,n+1)
            cin >> w[i];
        P.V = k;
        _for(i,1,n+1)
            P.ZeroOnePack(p[i],w[i]);
        printf("%d\n",P.dp[k]);
        return 0;
}
```

D. Shakugan no Shana

题意很好理解，给定一张图，每次删掉其中的一个节点，求每次删掉之后剩余的联通块中点权和的最大值

首先既然有联通块，还是变化的，每次 dfs 很明显会超时，所以需要使用并查集，但是删除却不是一个好的操作，并查集并没有简单的拆集合方法。所以需要考虑倒序，把删除点看成添加一个点，然后根据关系进行 merge 操作。

所以我们利用一个 map<int,set<int>>来记录边，方便添加点时执行 merge 操作，同时使用一个 bool ext[]数组来记录点是否已经被添加（排除 merge 时还“不存在”的点）

每次合并时将当前点权加上所有跟此点有关系的联通块的点权和，然后更新 max

然后将每次添加后的点权和最大值存入数组，最后倒序输出即可

代码：

```
#include <iostream>

#include<map>

#include<set>

#include<vector>

#include<algorithm>

using namespace std;


#define Design ios::sync_with_stdio(0),cin.tie(0),cout.tie(0)


const int maxn = 200010;

int fa[maxn];
```

```
map<int, set<int>>edge;//朋友关系
bool ext[maxn];//是否存在
vector<int>ans;//ans
int maxp;//当前最大
int que[maxn];//击杀顺序
int sum[maxn];//存储以i为根节点的联通块战斗力，只有i为根节点时才有意义

int find(int x)
{
    if (fa[x] == x) return x;
    fa[x] = find(fa[x]);
    return fa[x];
}

void merge(int x, int y)
{
    int fx = find(x);
    int fy = find(y);
    fa[fx] = fy;
    sum[fy] += sum[fx];
}

bool check(int a, int b)
{
    return find(a) == find(b);
}

int main()
```

```
{
    Design;
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        int t;
        cin >> t;
        //并查集初始化
        fa[i] = i;
        sum[i] = t;
    }
    for (int i = 1; i <= n; i++)
    {
        cin >> que[i];
    }
    int m;
    cin >> m;
    for (int i = 1; i <= m; i++)
    {
        int a, b;
        cin >> a >> b;
        edge[a].insert(b);
        edge[b].insert(a);
    }
    for (int i = n; i > 0; i--)
    {
        ans.emplace_back(maxp);
        int x = que[i];
```

```
            ext[x] = true;
            for (auto it : edge[x])
            {
                if (ext[it] && !check(it, x))
                {
                    merge(it, x);
                }
            }
            maxp = max(maxp, sum[find(x)]);
        }
        for (int i = ans.size() - 1; i >= 0; i--)
        {
            cout << ans[i] << endl;
        }
}
```

E. daiko!!

根据题意，而对于每个总得分数，对应的圆圈数就是小于或等于它的最大的 2 的 n 次方。比如总得分数为 3，那么圆圈数就是$(\text{int})\log_2 3 = 1$，所以根据这个和数据范围得到，圆圈数的范围为[1,15]（$2^{16} = 65536 > 5 \times 10^4$），这样问题就转化成了对于每个圆圈数有一个初始全为 0 的数组，p 操作即[l,r]区间加 1，q 操作即查询[l,r]区间的数字总和，由此可以得到本题的正确做法：建 15 棵线段树，分别对应圆圈数 1-15，然后对于每个 p，q 操作给定的 a 找到相应的线段树，进行修改或查询即可。需要注意的是查询时的 a 表示圆圈数，虽然范围有[1,50000]但实际上大于 15 都只要输出 0 就可以了

代码：

```
#include<iostream>
#include<algorithm>
#include<fstream>
#include<string>
#include<ctime>
#include<vector>
using namespace std;

const int maxn = 50010;
const int INF = 0x3f3f3f3f;
```

```
#define lc(x) x<<1
#define rc(x) x<<1|1
#pragma warning(disable:4996)

struct tree
{
	long long a;
	int l, r;
	long long tag;
	tree()
	{
		tag = 0;
		l = 0;
		r = 0;
		a = 0;
	}
}node[20][maxn << 2];

int a[maxn];
#define lc(x) x<<1
#define rc(x) x<<1|1

void build(int x, int l, int r, tree* node)
{
	node[x].l = l;
	node[x].r = r;
	if (l == r)
	{
		node[x].a = a[l];
		return;
	}
	int mid = (l + r) >> 1;
	build(lc(x), l, mid, node);
	build(rc(x), mid + 1, r, node);
	node[x].a = node[lc(x)].a + node[rc(x)].a;
}
inline void push_down(int x, tree* node)
{
	node[lc(x)].tag += node[x].tag;
	node[lc(x)].a += node[x].tag * (node[lc(x)].r - node[lc(x)].l + 1);
	node[rc(x)].a += node[x].tag * (node[rc(x)].r - node[rc(x)].l + 1);
	node[rc(x)].tag += node[x].tag;
	node[x].tag = 0;
}
```

```
void update(int left, int right, int x, int k, tree* node)
{
	if (left <= node[x].l && node[x].r <= right)
	{
		node[x].a += k * (node[x].r - node[x].l + 1);
		node[x].tag += k;
		return;
	}
	push_down(x, node);
	int mid = (node[x].l + node[x].r) >> 1;
	if (left <= mid) update(left, right, lc(x), k, node);
	if (right > mid) update(left, right, rc(x), k, node);
	node[x].a = node[lc(x)].a + node[rc(x)].a;
}
long long query(int left, int right, int x, tree* node)
{
	long long res = 0;
	if (left <= node[x].l && node[x].r <= right) return node[x].a;
	int mid = (node[x].l + node[x].r) >> 1;
	push_down(x, node);
	if (left <= mid) res += query(left, right, lc(x), node);
	if (right > mid) res += query(left, right, rc(x), node);
	return res;
}

int getnum(int x)    //获取(int)logx的值
{
	int pro = 1;
	int res = -1;
	while (pro <= x)
	{
		pro *= 2;
		res++;
	}
	return res;
}

int main()
{
	for (int i = 1; i <= 15; i++)
	{
		build(1, 1, 50000, node[i]);
	}
	int m;
```

```
    cin >> m;
    for (int i = 1; i <= m; i++)
    {
        char t;
        cin >> t;
        int a, l, r;
        cin >> a >> l >> r;
        if (t == 'q')
        {
            if (a > 15) cout << 0 << endl;
            else cout << query(l, r, 1, node[a]) << endl;
        }
        else
        {
            update(l, r, 1, 1, node[getnum(a)]);
        }
    }
}
```

F. :)'s necklace

一道有难度的题，首先题目要求相邻且形状不同的珠子颜色必须不同,但形状相同的珠子颜色也可以不同。所以我们可以考虑用 1,2 交替上色。这样可以保证相邻珠子颜色全不同。如果第 n 个与第 1 个珠子不相邻的话。那么，这样就解决了。但是 n 与 1 相邻,所以我们需要考虑 1,2,1,2,1,….1,2,1 这种情况,可以看出只有当 n 为奇数时,第 n 个珠子才染成 1,偶数则位 2。那么我们,现在只需要考虑 n 为奇数的情况,怎样才能让 1,2 交替的情况下。使第 n 个染成 2 呢？我们考虑减去一个珠子或者加上一个珠子就可以使 n 变为偶数。因为相邻的形状相同的珠子可以上相同的颜色。所以如果我们把两个相邻且形状相同的珠子染上相同的颜色。那么我们就相当于将 n 减去了 1 个,这样一来第 n 个就被染成了 2。所以最后,如果 n 个珠子全部相同那么 k = 1。否则，如果 n 为偶数 k = 2，如果 n 为奇数分为两种情况一种是有连续相同形状的珠子(**注意：**当然第 n 个与第一个相同形状也算连续)k = 2，否则 k = 3.

代码：

```
#include<bits/stdc++.h>
using namespace std;
#define rep(i, a, n) for(int i = a;i < n;i++)
#define per(i, a, n) for(int i = n - 1;i >= a;i--)
#define SZ(x) ((int)(x).size())
#define INF 0x3f3f3f3f
typedef vector<int> VI;
typedef long long ll;
const int N = 2e5 + 10;
```

```
int t[N];
int c[N];
int QAQ[N];
int main() {
    int _;
    scanf("%d", &_);
    while (_--) {
        memset(QAQ, 0, sizeof(QAQ));
        int n;
        cin >> n;
        int f1 = 0;
        int f2 = 0;
        rep(i, 1, n + 1) {
            scanf("%d", &t[i]);
            if (t[i] == t[i - 1])
                f2 = 1;
            if (!QAQ[t[i]]) {
                f1++;
                QAQ[t[i]] = 1;
            }
        }
        if (t[n] == t[1])
            f2 = 1;
        if (f1 == 1) {
            cout << 1 << endl;
            rep(i, 1, n + 1)
                cout << 1 << " ";
            cout << endl;
            continue;
        }
        if (n % 2 == 0) {
            cout << 2 << endl;
            rep(i, 0, n / 2) {
                cout << "1 2 ";
            }
            cout << endl;
        }
        else {
            if (f2) {
                cout << 2 << endl;
                int cnt = 1, flag1 = 1;
                rep(i, 1, n + 1) {
                    if (cnt % 2 == 0) {
                        if (t[i] == t[i - 1] && flag1)
```

```
                    cout << "1 ", flag1 = 0;
                else
                    cout << "2 ", cnt++;
            }
            else {
                if (t[i] == t[i - 1] && flag1)
                    cout << "2 ", flag1 = 0;
                else
                    cout << "1 ", cnt++;
            }
        }
    }
    else {
        cout << 3 << endl;
        int cnt = 1;
        rep(i, 1, n) {
            if (cnt % 2 == 0) {
                cout << "2 ", cnt++;
            }
            else {
                cout << "1 ", cnt++;
            }
        }
        cout << 3;
    }
    cout << endl;
  }
 }
 return 0;
}
```

G. Overfly

本题是本次比赛的压轴题

首先，根据题目，如果两个城镇可以互相到达（即构成了环），那么他们之间的运输上限是无限，而运输费用是 0，所以对于环，可以视为同一个点，而这个点入边和出边是其中所有点的所有出点和入点不属于这个环的边，所以由此需要使用 tarjan 或 kosaraju 算法进行缩点，将环变成一个点，然后就是很经典的问题，从一个地方运东西到另一个地方，每条路都有一个运量和一个单位运量费用，典型的最小费用最大流问题，题目数据范围不大，用 spfa 或 dijstra 找增广路的最小费用最大流都可以通过本题

代码：

```
#include<iostream>
#include<vector>
```

```
#include<set>
#include<iomanip>
#include<cmath>
#include<algorithm>
#include<queue>
#include<stack>
#include<map>
#include<string>
#include<fstream>
#pragma warning(disable:4996)
#pragma comment(linker, "/STACK:200000000,200000000")
using namespace std;

#define Design ios::sync_with_stdio(0),cin.tie(0),cout.tie(0)

const int maxn = 50010;
const int maxm = 100010;
int prep[maxn];
int pree[maxm];
int dis[maxn];
int st;
int ed;
int cnt;

struct MCMF
{
	int n;
	int cnt;
	int h[maxn];

	struct Edge
	{
		int v, nxt;
		int w, f;
	}e[maxm];

	int head[maxn];

	inline void init()
	{
		memset(head, -1, sizeof(head));
		memset(h, 0, sizeof(h));
		memset(e, 0, sizeof(e));
		cnt = 1;
```

```
}

inline void add(int u, int v, int w, int f)
{
	e[++cnt].v = v;
	e[cnt].nxt = head[u];
	head[u] = cnt;
	e[cnt].f = f;
	e[cnt].w = w;
	e[++cnt].v = u;
	e[cnt].nxt = head[v];
	head[v] = cnt;
	e[cnt].f = -f;
	e[cnt].w = 0;
}

pair<int, int> dijstra()
{
	int maxf = 0;
	int minc = 0;
	while (1)
	{
		priority_queue<pair<int, int>>heap;
		memset(dis, 0x3f, sizeof(dis));
		dis[st] = 0;
		heap.push(make_pair(0, st));
		while (!heap.empty())
		{
			pair<int, int> x = heap.top();
			heap.pop();
			if (-x.first != dis[x.second]) continue;
			if (x.second == ed) break;
			for (int i = head[x.second]; i != -1; i = e[i].nxt)
			{
				int now = e[i].f + h[x.second] - h[e[i].v];
				if (e[i].w > 0 && dis[e[i].v] > dis[x.second] + now)
				{
					dis[e[i].v] = dis[x.second] + now;
					heap.push(make_pair(-dis[e[i].v], e[i].v));
					prep[e[i].v] = x.second;
					pree[e[i].v] = i;
				}
			}
		}
```

```
            if (dis[ed] >= 0x3f3f3f3f) break;
            for (int i = 0; i <= n; i++) h[i] += dis[i];
            int now = 0x3f3f3f3f;
            for (int i = ed; i != st; i = prep[i])
                now = min(now, e[pree[i]].w);
            for (int i = ed; i != st; i = prep[i])
            {
                e[pree[i]].w -= now;
                e[pree[i] ^ 1].w += now;
            }
            maxf += now;
            minc += now * h[ed];
        }
        return make_pair(maxf, minc);
    }
}mc;



struct Edge
{
    int v, nxt;
    int w, f;
}e[maxm];

int head[maxn];

inline void init()
{
    memset(head, -1, sizeof(head));
}

void add(int u, int v, int w, int f)
{
    e[++cnt].v = v;
    e[cnt].nxt = head[u];
    head[u] = cnt;
    e[cnt].w = w;
    e[cnt].f = f;
    //add_dinic(v, u);
}

int dfn[maxn]; //第i个点被dfs到次序
int low[maxn];      //二叉搜索树上i所在子数上仍在栈中的最小dfn，low[i]相等的点在一个
```

```
强连通分量中
bool vis_tarjan[maxn];
stack<int>s_tarjan;
int tot_tarjan;
int cnt_tarjan;
int res_tarjan[maxn];       //储存强连通分量，res[i]表示点i属于第res[i]个强连通分量

void tarjan(int x)
{
    dfn[x] = low[x] = ++cnt_tarjan;
    s_tarjan.push(x);
    vis_tarjan[x] = true;
    for (int i = head[x]; ~i; i = e[i].nxt)
    {
        int v = e[i].v;
        if (!dfn[v])
        {
            tarjan(v);
            low[x] = min(low[x], low[v]);
        }
        else if (vis_tarjan[v])
        {
            low[x] = min(low[x], dfn[v]);
        }
    }
    if (low[x] == dfn[x])
    {
        tot_tarjan++;
        while (!s_tarjan.empty() && s_tarjan.top() != x)
        {
            int t = s_tarjan.top();
            res_tarjan[t] = tot_tarjan;
            vis_tarjan[t] = false;
            s_tarjan.pop();
        };
        s_tarjan.pop();
        res_tarjan[x] = tot_tarjan;
        vis_tarjan[x] = false;
    }
}

int dnt;
int viss[maxn];
map<int, int>sn;
```

```
map<int, int>nums;
int S = 20;
int T = 20;


int main()
{
	init();
	int n, m;
	cin >> n >> m;
	for (int i = 1; i <= m; i++)
	{
		int u, v;
		int w, f;
		cin >> u >> v >> w >> f;
		add(u, v, w, f);
	}
	for (int i = 1; i <= n; i++)
	{
		if (!dfn[i]) tarjan(i);
	}
	mc.init();
	for (int i = 1; i <= n; i++)
	{
		//cout << i << " " << res_tarjan[i] << endl;
		if (!viss[res_tarjan[i]])
		{
			viss[res_tarjan[i]] = true;
			dnt++;
			sn[i] = dnt;
			nums[res_tarjan[i]] = dnt;
		}
		else
		{
			sn[i] = nums[res_tarjan[i]];
		}
	}
	mc.n = dnt;
	st = sn[1];
	ed = sn[n];
	for (int i = 1; i <= n; i++)
	{
		for (int x = head[i]; ~x; x = e[x].nxt)
		{
```

```
                int v = e[x].v;
                if (sn[i] != sn[v])
                {
                    mc.add(sn[i], sn[v], e[x].w, e[x].f);
                }
            }
        }
        pair<int, int>ans = mc.dijstra();
        cout << ans.first << " " << ans.second << endl;
}
```

H. Low of Cycles

签到

代码：

```
#include<iostream>
#include<cmath>
using namespace std;

int main()
{
    const double PI = 3.1415926535;
    double a, b;
    double x, y;
    double ans = sqrt(pow(a - x, 2) + pow(b - y, 2)) * 2 * PI;
    printf("%.2lf", ans);
}
```

I. No repetition

这是一道签到题。字符串 s 只由连续的”wtcl”子串连接而成。所以 s 的长度应该为 4 的倍数。|s| / 4 即为”wtcl”出现的次数,记 n。

禁言时间 T = 0 + (n – 1) * 5.

代码：

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

int main() {
    int q;
    cin >> q;
    while (q--) {
        string s;
        cin >> s;
```

```
            cout << (s.length() / 4 - 1) * 5 << endl;
        }
        return 0;
}
```

J. Cirno's perfect math class

对于给定数列 $a_1, a_2 \dots a_n$和给定数字b，首先列出n次多项式

$$f(x) = k_1 x^n + k_2 x^{n-1} + \cdots + k_n x + k_{n+1}。$$

我们令 x 取值从 1 到 n+1：

$$f(1) = k_1 + k_2 + \cdots + k_n + k_{n+1} = a_1$$

$$f(2) = k_1 \times 2^n + k_2 \times 2^{n-1} + \cdots + k_n \times 2 + k_{n+1} = a_2$$

$$\cdots$$

$$f(n+1) = k_1 \times (n+1)^n + k_2 \times (n+1)^{n-1} + \cdots + k_n \times (n+1) + k_{n+1} = b$$

k_1到k_{n+1}共有$n+1$个未知数代表各项系数，可列出非线性方程组：

$$\begin{pmatrix} 1 & 1 & \cdots & 1 & 1 & a_1 \\ 2^n & 2^{n-1} & \cdots & 2 & 1 & a_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ n^n & n^{n-1} & \cdots & n & 1 & a_n \\ (n+1)^n & (n+1)^{n-1} & \cdots & n+1 & 1 & b \end{pmatrix}$$

为范德蒙矩阵的增广矩阵。

通过高斯消元法解得方程组的解即为多项式各项系数。

（高斯消元法可以查阅资料找模板，本题相关代码是优化过且有删减，比如忽略方程组无解等情况）

代码：

```cpp
#include<cstdio>
#include<cmath>
#include<algorithm>
using namespace std;

const int maxn = 10;
double matrix[maxn][maxn];
int n;

int Pow(int b, int t){//快速幂
    int ret = 1;
    while (t)
    {
        if (t & 1)
            ret *= b;
        b *= b;
        t >>= 1;
    }
    return ret;
}

int main(){
    scanf("%d", &n);
    ++n;     //因为本题构造的是n+1阶矩阵，此处n自增1方便计算
    for (int i = 1; i <= n; ++i)
        scanf("%lf", &matrix[i][n + 1]);
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            matrix[i][j] = Pow(i, n - j);
    for (int i = 1; i <= n; ++i){
        int maxc = i;
        for (int j = i + 1; j <= n; ++j){
            if (fabs(matrix[j][i]) > fabs(matrix[maxc][i]))
                maxc = j;
        }
        for (int j = 1; j <= n + 1; ++j)
            swap(matrix[i][j], matrix[maxc][j]);
        for (int j = 1; j <= n; ++j){
            if (j != i){
                double temp = matrix[j][i] / matrix[i][i];
                for (int k = i + 1; k <= n + 1; ++k)
                    matrix[j][k] -= matrix[i][k] * temp;
            }
        }
    }
    for (int i = 1; i <= n; ++i)
        printf("%.6lf\n", matrix[i][n + 1] / matrix[i][i]);
    return 0;
}
```

K. Searching for HRNA

本题的本意是前缀匹配，可以想到 tire 树，但是 tire 树空间极为浪费，虽然可以通过本题，但是本题还有其它更简单的方法。首先对所有字符串进行排序，可以得到：

如果一个串是另一个串的母串，那么这个母串一定在子串的后面，如：

ACB

ACBD

ACBDE

而若一个串不是另一个串的母串，那如果这个串存在母串，它一定排在那个串的母串之后，如：

ACB

ACBD

ACC

所以可以得知：根据字典序排序之后只需要找一个字符串的下一个字符串，就可以判断这个串是否为 HRNA 了。

代码：

```
#include<iostream>
#include<string>
#include<algorithm>
using namespace std;

bool no[1010];
string s[1010];

int main()
{
	int n;
	cin >> n;
	for (int i = 1; i <= n; i++)
	{
		cin >> s[i];
	}
	sort(s + 1, s + 1 + n);
	for (int i = 1; i < n; i++)
	{
		//cout << " " << s[i] << endl;
		if (s[i + 1].find(s[i]) == 0) no[i] = true;
	}
	for (int i = 1; i <= n; i++)
```

```
    {
        if (!no[i]) cout << s[i] << endl;
    }
}
```

L. “<<” and “>>” operation

道有趣的题。因为<<与>>运算会用 0 补位。所以如果 1101 ….. 0000 左移之后右移之后,变成 0101 …. 0000 如果 0000 …. 1101 右移之后左移,变成 0000 …. 1100 可以发现我们可以通过<<与>>运算将某个二进制位以左或以右的 1 变成 0. 所以我们将 x 转成二进制后,除去前缀 0 和后缀 0 剩下的二进制数，如果是 y 转成二进制后的子串,那么我们就可以通过<<与>>使 y 变成 x，需要特判一下 x 为 0 时总是 YES,y 为 0 且 x 不为 0 时总是 NO。

本题使用 cin/cout 或 java 语言必须进行输入输出优化，同时匹配字符串需要用 string.find()或 kmp 算法，否则会时间超限

代码（KMP）：

```
#include<bits/stdc++.h>
#define ll long long
using namespace std;
char a[70];
char b[70];
int nxt[70];
int tran(char c[], ll x) {
    int cnt = 0;
    while (x % 2 == 0) {
        x /= 2;
    }
    while (x != 0) {
        ll tmp = x % 2;
        c[cnt++] = '0' + tmp;
        x /= 2;
    }
    return cnt;
}
void preKMP(char x[], int m, int nxt[]) {
    memset(nxt, 0, sizeof(nxt));
    int j = 0, k = -1;
    nxt[0] = -1;
    while (j < m) {
        while (k != -1 && x[j] != x[k]) k = nxt[k];
        if (x[++j] == x[++k]) nxt[j] = nxt[k];
        else nxt[j] = k;
```

```
    }
}
bool judge(char x[], int n, char y[], int m, int nxt[]) {
    preKMP(y, m, nxt);
    int i = 0, j = 0;
    while (i < n) {
        while (j != -1 && x[i] != y[j]) j = nxt[j];
        i++; j++;
        if (j >= m) {
            return true;
        }
    }
    return false;
}
int main() {
    int t;
    scanf("%d", &t);
    while (t--) {
        memset(a, 0, sizeof(a));
        memset(b, 0, sizeof(b));
        ll x, y;
        scanf("%lld %lld", &x, &y);
        if (x == 0) {
            printf("YES\n");
            continue;
        }
        if (y == 0) {
            printf("NO\n");
            continue;
        }
        int len1 = tran(a, y);
        int len2 = tran(b, x);
        if (judge(a, len1, b, len2, nxt)) {
            printf("YES\n");
        }
        else {
            printf("NO\n");
        }
    }
    return 0;
}
```