A Кольцевая линия

Так как в графе N вершин и N ребер, то ответ всегда YES, потому что максимальное количество ребер при которых не будет цикла равно N-1 (когда граф является деревом)

Решение Python

```
print("YES")
```

B Транспорт на Новый год

В этой задаче нам дается ориентированный граф и спрашивают, достижима ли конкретная вершина из вершины 1. Это можно решить, запустив поиск в глубину, начиная с вершины 1. Поскольку каждая вершина имеет не более одного исходящего ребра, можно записать DFS в виде простого цикла.

Решение Python

```
a, b = map(int, input().split())
c = list(map(int, input().split()))
d = 0
while d < b - 1:
    d += c[d]
if d == b - 1:
    print("YES")
else:
    print("NO")
```

Решение C++

```
#include <cstdio>

using namespace std;

int a[1234567];

int main() {
  int n, t;
  scanf("%d %d", &n, &t);
  for (int i = 1; i < n; i++) {
    scanf("%d", a + i);
  }
  int x = 1;
  while (x < t) {
    x += a[x];
  }
  puts(x == t ? "YES" : "NO");
  return 0;
}
```

C Обход графа

Достаточно применить любой известный обход графа (в ширину, глубину или другой)

Решение C++

```
int mas[111][111];
int u[111];
int res = 0;
int n;

void go(int v)
{
    if (u[v]) return;
    u[v] = 1;
    res++;
    for (int i = 1; i <= n; i++)
        if (mas[v][i])
            go(i);
}


int main()
{

    int m;
    cin >> n >> m;

    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        mas[a][b] = mas[b][a] = 1;
    }

    go(1);

    cout << res;

    return 0;
}
```

D Две кнопки

Самое простое решение — просто запустить обход в ширину. Построим граф, в котором вершины — числа, а ребра ведут из одного числа в другое, если можно получить второе из первого за одну операцию. Можно заметить, что первое число никогда не выгодно делать больше, чем 2*m*, поэтому в графе будет не больше $2 \cdot 10^4$ вершин и $4 \cdot 10^4$ ребер, и BFS сработает очень быстро.

Однако, есть решение гораздо быстрее. Развернем задачу: изначально у нас есть число *m*, и мы хотим получить число *n*, пользуясь операциями "прибавить к числу 1" и "поделить число на 2, если оно четно".

Пускай мы в какой-то момент применили две операции типа 1, и потом операцию типа 2; но эти три операции можно заменить на две: сначала операция типа 2, а потом одна операция типа 1, и количество операций от этого уменьшится. Отсюда следует, что применять больше одной операции типа 1 имеет смысл только тогда, когда операций типа 2 больше не будет, то есть, когда *n* меньше *m* и осталось сделать несколько прибавлений, чтобы их сравнять. На самом деле, теперь существует ровно одна последовательность операций, удовлетворяющая таким требованиям: если *n* меньше *m*, прибавляем единички пока нужно, иначе если *n* четно, делим на 2, а если нечетно, прибавляем 1 и потом делим на 2. Количество операций в такой последовательности можно найти за время $O(\log n)$.

Challenge: рассмотрим обобщенную задачу: мы хотим получить число *n* из числа *m* с помощью операций двух типов "отнять *a*" и "умножить на *b*". Обобщите решение исходной задачи и научитесь находить минимальное количество операций за время $O(\log n)$ в случае, если *a* и *b* взаимно просты. Справитесь ли вы, если у *a* и *b* могут быть нетривиальные общие делители?

Решение Python

```
nums = str(input()).split()
fr = int(nums[0])
t = int(nums[1])

def diff(a,b):
    if b<=a:
        return a-b
    elif b%2==0:
        return diff(a,b/2)+1
    else:
        return diff(a,(b+1)/2)+2

print(int(diff(fr,t)))
```

Решение C++

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

```
ll cnt(ll a, ll b)
{
    if (a >= b)
        return (a - b);
    else if (b % 2 == 0)
        return 1 + cnt(a, b / 2);
    else
        return 2 + cnt(a, (b + 1) / 2);
}

int main()
{
    ll a, b;
    cin >> a >> b;
    cout << cnt(a, b);
    return 0;
}
```

E Гастролеры

Решение

Здесь имеется граф в виде дерева (связный неориентированный граф без циклов). И нужно для каждого запроса найти путь и увеличить значение в вершинах. Это можно сделать одним из стандартным алгоритмов – обходом в глубину

http://e-maxx.ru/algo/dfs

или ширину

http://e-maxx.ru/algo/bfs

То есть находим путь и восстанавливаем ответ.

Для тех, кто хорошо умеет писать обход в глубину можно сразу искать путь и обновлять значения в вершинах.

То есть написать функцию, которая будет возвращать ответ 1, если мы нашли путь в требуемую вершину и 0 в противном случае. Ну и соответственно если нашли, то обновляем значение в вершине.

Тут было важно еще как мы храним граф. Если матрицей смежности, то каждый обход в худшем случае был бы за O(N*N), что неприемлемо.

Нужно хранить списком смежных вершин, который работает за O(E), где E – количество вершин в графе (для дерева оно равно N)

Итоговая асимптотика O(N*M)

```
#include <iostream>
#include <cstdio>
#include <cmath>
#include <string>
#include <algorithm>
#include <vector>
#include <queue>
#include <stack>
#include <vector>
#include <set>
```

```
#include <map>
#include <sstream>
#include <memory.h>
#include <cassert>
#include <time.h>
using namespace std;

typedef vector<int> vi;
#define sz(a) int((a).size())
#define all(c) (c).begin(),(c).end()


const int MAXN = 200100;
vi mas[MAXN]; // граф храним списком смежных вершин
int add, need;
int res[MAXN];

// возвращаем 1 если нашли путь, 0 - если нет
// передаем текущую вершину и ее предка
int go(int v, int pr)
{
	int r=0;
	if (v == need)
	{
		res[v]+=add;
		return 1;
	}

	for (int i=0;i<sz(mas[v]);i++)
	if (mas[v][i]!=pr)
	{
		r = go(mas[v][i],v);
		if (r==1) // если нашли путь
		{
			res[v]+=add;
			return 1;
		}

	}

	return 0;
}


void solve()
{
	int n;
	scanf("%d",&n);
	assert(n>=1 && n<=100000);
	for (int i=0;i<MAXN;i++)
		mas[i].clear();
	memset(res,0,sizeof(res));

	for (int i=1;i<n;i++)
	{
		int a,b;
		scanf("%d%d",&a,&b);
		mas[a].push_back(b);
		mas[b].push_back(a);
		assert(a>=1 && a<=n);
		assert(b>=1 && b<=n);
		assert(a!=b);
	}
```

```
	int m;
	scanf("%d",&m);
	assert(m>=0 && m<=100000);

	for (int i=0;i<m;i++)
	{
		int a;
		scanf("%d%d%d",&a,&need,&add);
		assert(a>=1 && a<=n);
		assert(need>=1 && need<=n);
		go(a,-1);
	}

	for (int i=1;i<=n;i++)
		printf("%d\n",res[i]);

	return ;

}

int main()
{
	solve();

	return 0;
}
```

F Строим дорогу

Так как m<n/2, то существует минимум одна вершина из которой нет ребер. Используем топологию звезда. То есть ставим одну вершину в центр и из нее все остальные.

Решение Python

```
n,m = list(map(int,input().split()))
avail = set(range(1,n+1))
for i in range(m):
    a,b = list(map(int,input().split()))
    avail = avail-{a,b}
center = list(avail)[0]
print(n-1)
for i in range(1,n+1):
    if i!=center:
        print(center,i)
```

Решение C++

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


int main()
{
	int i,j,n,k,H[10000];

	scanf("%d %d",&n,&k);
	for(i = 1; i <= k; i++)
	{
		int a, b;
```

```
            scanf("%d %d",&a,&b);
            H[b] = H[a]=1;
        }

        i=0;
        while(H[++i]);

        printf("%d\n",n-1);

        for(j = 1; j <= n; j++)
            if(j!=i) printf("%d %d\n",j,i);
}
```

G Вечеринка

Ясно, что хотя бы один человек (тот, у кого меньше всех знакомых) должен уйти. Докажем, что должны уйти хотя бы двое. Действительно, пусть ушел только один человек, и у него d знакомых. Тогда у остальных до его ухода было больше d знакомых, а после его ухода стало меньше, чем $d + 1$, то есть не больше d. Значит, его уход повлиял на количество знакомых у каждого из оставшихся, то есть он знаком со всеми: $d = N - 1$. Но у него меньше всех знакомых — противоречие.

Итак, ответ не больше $N - 2$. Покажем, что $N - 2$ человека могли остаться (если, конечно, $N > 1$). Организуем граф знакомств следующим образом: возьмем полный граф на N вершинах и удалим одно ребро. Тогда степени двух вершин равны $N - 2$, а степени остальных $N - 1$. После удаления первых двух вершин остается полный граф на $N - 2$ вершинах, и все степени равны $N - 3$, поэтому больше никто не уйдет.

Решение Python

```
import sys
input=sys.stdin.readline
t=int(input())
for i in range(t):
    print(max(0,int(input())-2))
```

Решение C++

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

int t,n;

signed main()
{
        cin>>t;
        while (t--)
        {
                cin>>n;
                cout<<max(0ll,n-2)<<endl;
        }
        return 0;
}
```

H Граф максимального диаметра

Построим граф следующим образом. Возьмем все вершины с $a_i > 1$ и сложим их в цепочку. Разумеется, все, кроме крайних из них, будут иметь степень 2, диаметр станет равен количеству вершин минус 1.

Можео показать, что построение графа любым другим образом не увеличит диаметр.

Как можно распределить остальные вершины? Две из них можно использовать для увеличения диаметра. А остальные не повлияют на ответ, их можно соединить с любыми из вершин со свободными степенями. Если не добавлять циклов, то и диаметр не изменится — путь, который был самым длинным, не станет короче.

Все эти факты подразумевают, что граф должен быть деревом, а сумма a_i должна быть не меньше $2n - 2$.

Асимптотика решения: $O(n)$.

Решение Python

```
n = int(input())
l = list(map(int, input().split()))

one,many = [],[]
a,b,c = 0,0,0
for i,d in enumerate(l):
    if d==1:
        one.append(i+1)
        a+=1
    else:
        many.append((i+1,d))
        b+=1
        c+=d

if c-2*b+2<a:
    print("NO")
    exit()

print("YES",b+ min(a, min(a,2))-1)
print(n-1)

left = []
dia  = []
if one:
    dia.append(one.pop())
for i,d in many:
    dia.append(i)
    if d>2:
        left.append((i,d-2))
if one:
    dia.append(one.pop())
for i in range(len(dia)-1):
    print(dia[i], dia[i+1])
while one:
    i,d = left.pop()
    print(i, one.pop())
    if d>1:
        left.append((i,d-1))
```

Решение C++

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

#define forn(i, n) for (int i = 0; i < int(n); i++)

using namespace std;

const int N = 1000 + 7;

int n;
int a[N];

int main() {
	scanf("%d", &n);
	forn(i, n)
		scanf("%d", &a[i]);

	int sum = 0;
	forn(i, n)
		sum += a[i];

	if (sum < 2 * n - 2){
		puts("NO");
		return 0;
	}

	vector<int> ones;
	forn(i, n) if (a[i] == 1){
		a[i] = 0;
		ones.push_back(i);
	}

	int t = ones.size();
	int dm = (n - t) - 1 + min(2, t);
	printf("YES %d\n%d\n", dm, n - 1);

	int lst = -1;
	if (!ones.empty()){
		lst = ones.back();
		ones.pop_back();
	}

	forn(i, n){
		if (a[i] > 1){
			if (lst != -1){
				--a[lst];
				--a[i];
				printf("%d %d\n", lst + 1, i + 1);
			}
			lst = i;
		}
	}

	for (int i = n - 1; i >= 0; --i){
		while (!ones.empty() && a[i] > 0){
			--a[i];
			printf("%d %d\n", i + 1, ones.back() + 1);
			ones.pop_back();
		}
	}
	return 0;
}
```

I Треугольники

Фактически, представим что весь полный граф просто покрашен — каждое ребро или красное, или синее. Нас интересует количество одноцветных треугольников (где все три ребра одного цвета).

Будем искать обратное — количество разноцветных треугольников (где не все три ребра одного цвета). Зафиксируем одну вершину разноцветного треугольника, причём ту, где рёбра из неё разных цветов. Тогда, если из зафиксированной вершины выходит cnt красных рёбер, то ответ для этой вершины = cnt * (N — 1 — cnt) треугольников.

Если мы посчитаем сумму всех этих значений для всех вершин, то получим, что каждый разноцветный треугольник мы посчитали дважды, потому что у каждого разноцветного треугольника ровно 2 вершины, из которых рёбра разных цветов. Делим на 2.

Вот и всё — количество всего треугольников мы знаем: C(N,3), и остаётся отнять от него поделённую на 2 сумму. Тоже O(M + N).

Решение C++

```
#include<iostream>
using namespace std;
const int N=1e6;
long long d[1000000],ans,n,m,sum;

int main()
{
    int x,y;
    scanf("%I64d%I64d",&n,&m);
    sum=n*(n-1)/2*(n-2)/3;
    while(m--)
    {
            scanf("%d%d",&x,&y);
            d[x]++;
            d[y]++;
    }
    for(int i=1;i<=n;i++)
        ans=ans+(d[i]*(n-1-d[i]));
    ans/=2;
    printf("%I64d\n",sum-ans);
    return 0;
}
```

J Алгоритм Дейкстры

Задача на применение алгоритма Дейкстры с кучей с восстановлением ответа. Для восстановления ответа достаточно для каждой вершины хранить номер вершины, из которой мы пришли. В конце, чтобы получить путь, нужно пройти по ссылкам из конечной вершины в начальную. Дейкстра с кучей/сетом применяется для разреженного графа и имеет асимптотику O(E*log(E)), где E – количество ребер в графе. Для полного графа или при задании графа матрицей смежности нужно использовать обычный проход по всем вершинам вместо сета (в этом случае асимптотика $O(N^2)$).

Решение Python

```
INF = 1 << 60
n, m = map(int, input().split())
g = [[] for _ in range(n)]
d = [0] + [INF] * n
p = [-1] * n
for _ in range(m):
   u, v, t = map(int, input().split())
   g[u-1].append((t, v-1))
   g[v-1].append((t, u-1))
from heapq import *
q = [(0, 0)]
while q:
   u = heappop(q)[1]
   for e in g[u]:
          t = d[u] + e[0]
          v = e[1]
          if t < d[v]:
                 d[v] = t
                 p[v] = u
                 heappush(q, (d[v], v))
if d[n-1] == INF:
   print(-1);
else:
   x = n - 1
   ans = []
   while x != -1:
          ans.append(x + 1)
          x = p[x]
   ans.reverse()
   print(' '.join(map(str, ans)))
```

Решение C++

```
// очередь с приоритетом (расстояние, вершина)
priority_queue < pair<int, int> > pq;
//список смежных вершин и расстояний до них
vector < pair <int ,int> > mas[1001];
```

```
int dijkstra2(int s, int t) //кратчайший путь из s в t
{
	for (int i=1;i<=n;i++) d[i]=oo;		// сначала заполняем оо
	memset(u,0,sizeof(u));				// помечаем все 0
	d[s]=0;							// расстояние до s == 0
	pq.push(make_pair(0,s));			// заносим вершину 0 в хип

	while (!pq.empty())					// пока очередь не пуста
	{
		s = pq.top().second; // берем из кучи ближайшую вершину
		pq.pop();
		if (u[s]) continue; //эта вершина уже обработана!!!
		u[s]=1;			// помечаем ее (кратчайшее расстояние до нее
d[s])
		if (s==t) break;

		for (int i=0;i<sz(mas[s]);i++) if (!u[mas[s][i].first])
		if (d[s]+mas[s][i].second<d[mas[s][i].first])
		{	// обновляем расстояния и кидаем вершину в хип
			d[mas[s][i].first] = d[s]+mas[s][i].second;
			pq.push( make_pair(- d[mas[s][i].first],mas[s][i].first )
);
		}
	}
	return d[t];				//кратчайшее расстояние из s в t
}
```