A Числа Фибоначчи

Решение C++

```cpp
#include <iostream>
#include <sstream>

using namespace std;


int main() {

    int fib[111];

    int n;
    cin >> n;
    fib[0] = fib[1] = 1;
    for (int i = 2; i <= n; i++)
        fib[i] = fib[i - 1] + fib[i - 2];

    cout << fib[n];

    return 0;
}
```

Решение Python

```python
def fib(n):
    a = 0
    b = 1
    for __ in range(n):
        a, b = b, a + b
    return a

n = int(input())
print(fib(n+1))
```

B Торт - это ложь

Заметим, что какой бы путь вы не выбрали, стоимость пути всегда будет одинаковая. Если вы знаете, что стоимость одинаковая, то посчитать ее несложно. Она равна $n \cdot m - 1$. Поэтому все задача: проверить, равны k и $n \cdot m - 1$ или нет.

Постоянную стоимость можно доказать по индукции от $n + m$: для $n = m = 1$ стоимость равна $1 \cdot 1 - 1 = 0$. Для фиксированных (n, m) же, есть только два возможных последних шага:

- либо из $(n, m - 1)$ стоимости n: общая стоимость будет равна $n \cdot (m - 1) - 1 + n = n \cdot m - 1$
- либо из $(n - 1, m)$ стоимости m: общая стоимость будет равна $(n - 1) \cdot m - 1 + m = n \cdot m - 1$.

Таким образом, какой путь не выбери, стоимость будет постоянна.

Решение C++

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

```c
int n,m,k,t;
int main(){
	scanf("%d",&t);
	while(t--){
		scanf("%d%d%d",&n,&m,&k);
		if(n*m==k+1)
			puts("YES");
		else
			puts("NO");
	}
	return 0;
}
```

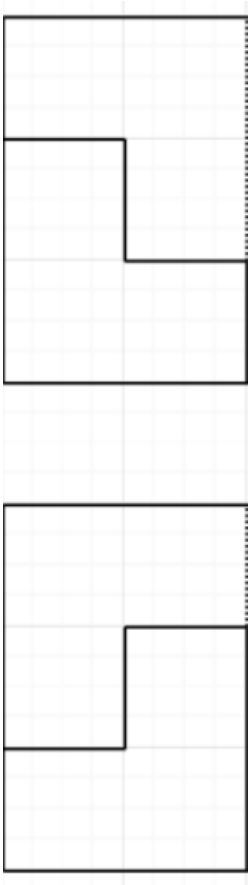
Решение Python

```python
for i in range(int(input())):
    n, m, k = map(int, input().split())
    print("YES" if m * n - 1 == k else "NO")
```

C Заполнение формами

F(n) = F(n-2) * 2

Чтобы посчитать ответ для n клеток рассмотрим количество способов заполнения последних клеток:



Получается всего 2 варианта, итого F(n-2) *2

Либо можно вывести формулу: ответ равен $2^{n/2}$

Решение C++

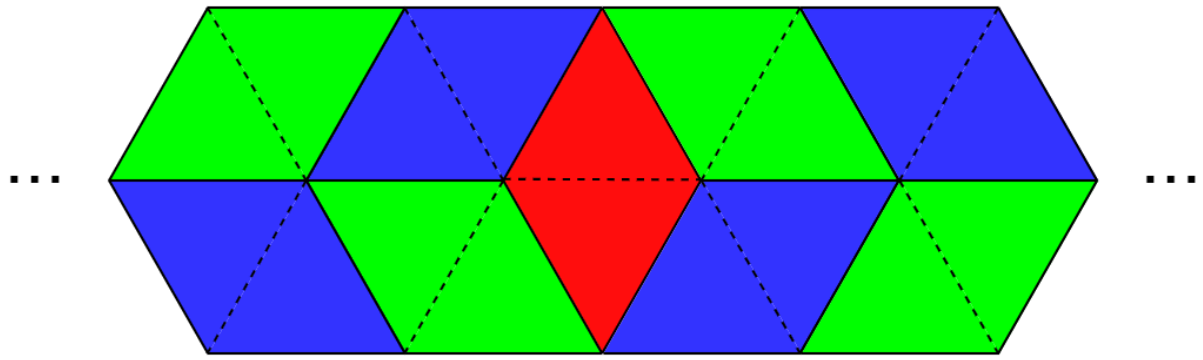
```
#include <stdio.h>
int main(void) {
    int n; scanf("%d", &n);
    if (n % 2 == 0) printf("%d", 1 << (n / 2));
    else printf("0");
    return 0;
}
```

Решение Python

```
    n = int(input())
    print(1 << (n//2) if n % 2 == 0 else 0)
```

D Заполнение ромбами

Куда бы вы ни поместили вертикальный ромб в какой-то момент, все остальные места однозначно размещаются горизонтальными ромбами, как на рисунке ниже.



Есть n мест, где вы можете поставить вертикальный ромб, поэтому ответ будет **n** для каждого теста.

То есть дано n, нужно вывести n

E Своя игра

Пусть у нас n тем

На каждом шаге выбор Тимофея совпадет с выбором игрока с вероятностью 1./n и игра продолжится с n-1 темами. Оставшаяся вероятность 1 - 1./n - в этом случае игра продолжится без Тимофея и с вероятностью 1./(n-1) он выиграет (тема , которую он выбрал будет финальной из тех что остались)

Итого

F(n) = 1./n * F(n-1) + (1-1./n) * (1./(n-1)) = 1./n * F(n-1) + 1./n;

Реализовать можно рекурсивно, либо без рекурсии, заполняя массив по мере увеличения n.

В рекурсии я запоминал уже посчитанные значения, чтобы не пересчитывать каждый раз заново, хотя тут это не обязательно.

Сложность **O(N)**

```
#include <cstdlib>
#include <cstdio>
#include <cassert>

using namespace std;
//#pragma comment(linker, "/STACK:20000000")

int n;
double dp[1001]={0.0};

int main() {
	freopen("jeopardy.in","rt",stdin);  freopen("jeopardy.out","wt",stdout);
	scanf("%d", &n);
	assert(n >1 && n<1001);

	dp[1] = dp[2] = 1.0;

	for (int i = 3; i <= n; ++i){
		dp[i] = 1.0/i + dp[i-1]/i;
	}

	printf("%.9lf\n", dp[i]*100);

	return 0;
}
```

Еще вариант решения с рекурсией

```
double dp[1111];

double go(int left){ // left - сколько осталось тем
	if (left==1) return 1;
	if (dp[left]>-.1) return dp[left]; // если уже посчитано
	double res=0;

	res = 1./left*go(left-1) + (left-1.)/left*1./(left-1);

	return dp[left] = res; // возвращаем результат и сохраняем в массив
}

int main(){
	int n;
	scanf("%d",&n);
	for (int i=0;i<=n;i++)
		dp[i]=-1;

	printf("%.4lf\n",go(n)*100);

	return 0;
}
```

F Баскетбольная зарядка

Это довольно стандартная задача на динамическое программирование. Пусть $dp_{i,1}$ равно максимальному суммарному росту участников команды, если последний взятый школьник имел позицию $(i-1,1)$, $dp_{i,2}$ — то же самое, только последний взятый школьник имел позицию $(i-1,2)$ и $dp_{i,3}$ — опять то же самое, только мы не брали никакого школьника с позиции $i-1$. Переходы довольно простые:

- $dp_{i,1} = max(dp_{i-1,2} + h_{i,1}, dp_{i-1,3} + h_{i,1}, h_{i,1})$;
- $dp_{i,2} = max(dp_{i-1,1} + h_{i,2}, dp_{i-1,3} + h_{i,2}, h_{i,2})$;
- $dp_{i,3} = max(dp_{i-1,1}, dp_{i-1,2})$.

Это динамическое программирование может быть посчитано практически без использования памяти, потому что нам нужен только $i-1$-й ряд для подсчета i-го ряда этого дп. Более того, нам на самом деле не нужно хранить $dp_{i,3}$, если мы добавим переходы $dp_{i,1} = max(dp_{i,1}, dp_{i-1,1})$ и $dp_{i,2} = max(dp_{i,2}, dp_{i-1,2})$. Эти переходы немного изменят наше дп. Теперь $dp_{i,j}$ равно максимальному суммарному росту участников команды, если последний взятый школьник имел позицию $(i-1,1)$ или меньше. То же самое с $dp_{i,2}$. Ответ равен $max(dp_{n,1}, dp_{n,2})$.

Асимптотика решения: $O(n)$.

Решение C++

```
#include<bits/stdc++.h>
using namespace std;
#define ll long long int
int main()
{
    ll i, j, k, m, n;
    cin >> n;
    ll a[n], b[n];
    for (i = 0; i < n; i++)
        cin >> a[i];
    for (i = 0; i < n; i++)
        cin >> b[i];
    ll dp1[n], dp2[n];
    dp1[0] = a[0];
    dp2[0] = b[0];
    for (i = 1; i < n; i++)
    {
        dp1[i] = max(dp1[i - 1], dp2[i - 1] + a[i]);
        dp2[i] = max(dp2[i - 1], dp1[i - 1] + b[i]);
    }
    cout << max(dp1[n - 1], dp2[n - 1]) << endl;
}
```

Решение Python

```
    n=int(input())
    h1=list(map(int,input().split()))
    h2=list(map(int,input().split()))
    dp1=[0]
    dp2=[0]
    for i in range(n):
        dp1.append(max(dp1[-1],dp2[-1]+h1[i]))
        dp2.append(max(dp2[-1],dp1[-2]+h2[i]))
    print(max(dp1[-1],dp2[-1]))
```

G Случайное умножение

Для того чтобы добраться от 1 до числа N мы должны пройти по каким-то его делителям. Если мы попадаем не на делитель числа – то ответ 0.

Для начала найдем все делители числа N. Это делается просто проходом до корня из N. Всего делителей может быть порядка $N^{1/3}$. В нашем случае примерно 1300.

Далее отсортируем их. Например для N = 20 получаем: 1 2 4 5 10 20

Далее считаем вероятность с помощью ДП. Считать будем справа налево. Пусть go(i) – вероятность что мы получим N, начиная с i

go(20) = 1

Для всех остальных перебираем куда мы можем перейти (не более чем в K раз).

go (i) = (go(i) + go(i1) + .. go(im))/ (K)

Тут надо избавиться от рекурсивного вызова самого себя.

Нужно go(i) перенести в левую часть.

Получим go (i) = (go(i1) + .. go(im))/ (K-1)

В общем решение за $O(N^{2/3})$

В своем решении я явно не получал все делители а писал рекурсивное ДП на мэпах.

```
map <long long, double> dp;
long long need,mx;

// Передавал не текущее число, а сколько осталось, то есть N/текущее число
double go(long long left)
{
	if (left==1) return 1;
	if (dp.find(left)!=dp.end())
		return dp[left];

	double res=0;

	//перебираем на сколько домножаем - по сути делители до корня
	for (long long i=1;i<=mx && i*i<=left;i++)
	if (left % i ==0)
	{
		if (i!=1)
		res+=go(left/i);
		if (left/i<=mx && i*i!=left)
		res+=go(i);
	}

	res/=(mx-1);

	return dp[left]=res;
}


void solve(){
	cin >> need >> mx;

	if (mx==1)  {
		printf("0");
		return ;
	}

	printf("%.12lf\n",go(need));
   }
```

H Булочки

Создадим массив dp размера n на m. dp[i][j] будет означать максимальное количество денег, которое мы получим если используем i единиц теста и испечем булочки с начинками типов 1..j.

Изначально dp[i][0] = 0 для всех i.

Пересчитать данную динамику несложно:
dp[i][j] = max{ dp[i-c[j]*k][j-1] + d[j]*k } по всем k от 0 до a[j]/b[j], для которых i-c[j]*k>=0

Ответом будет max{ dp[k][m] + ((n-k)/c0)*d0 } по всем k от 0 до n.

Конечно же, в разборе данной задачи деление везде целочисленное.

Полученное решение работает за O(nma), где a - максимум по a_i.

Решение C++ (нерекурсивное)

```
#include<bits/stdc++.h>
using namespace std;
int dp[1100];
signed main()
{

    int n, m, C, D;
    cin >> n >> m >> C >> D;

    for (int i = 0; i <= n; i++)
    {
        dp[i] = (i / C) * D;
    }
    int ans = 0;
    for (int i = 1; i <= m; i++)
    {
        int a, b, c, d;
        cin >> a >> b >> c >> d;

        for (int j = n; j >= 0; j--)
        {
            for (int k = 1; b * k <= a; k++)
            {
                if (j - c * k >= 0)      dp[j] = max(dp[j], dp[j - c * k] + k * d);
            }
            ans = max(ans, dp[j]);
        }
    }
    cout << ans << endl;

}
```

Решение C++ (рекурсивное)

```
int a[22], b[22], c[22], d[22];
int dp[1111][22];
```

```
int d0, c0;
int m;
int go(int left, int cur)
{
    if (cur == m) return left / c0 * d0;
    if (dp[left][cur] != -1) return dp[left][cur];

    int res = 0;
    for (int i = 0;; i++)
    {
        if (i * c[cur] > left || i * b[cur] > a[cur]) break;
        res = max(res, i * d[cur] + go(left - i * c[cur], cur + 1));

    }

    return dp[left][cur] = res;
}


int main()
{
    // init();

    int n;
    memset(dp, -1, sizeof(dp));
    cin >> n >> m >> c0 >> d0;
    for (int i = 0; i < m; i++)
        cin >> a[i] >> b[i] >> c[i] >> d[i];

    int res = go(n, 0);

    printf("%d\n", res);



    return 0;
}
```

I Теплица

Задача

На прямой даны n точек, каждая одного типа от 1 до m. Мы можем разделить прямую на m - 1 интервалов и переместить какое-то количество точек так, чтобы каждая точка типа i находилась бы внутри i-того интервала, которые пронумерованы от 1 до m слева направо. Нужно найти минимальное количество точек, которые нужно переместить.

Решение

Сперва заметим, что данные координаты не нужны: важен только порядок точек. Пусть мы можем переместить какое-то количество точек, чтобы получить годную перестановку. Тогда все остальные точки остались на своих местах, поэтому их типы должны неубывать слева направо. Поэтому достаточно найти наибольшее количество точек, которые могут остаться на своих местах, что является наибольшей неубывающей последовательностью типов среди данных типов. Если эта длина l, то ответ n - l.

В этой задаче достаточно было реализовать квадратичное решение. Считаем $dp[i][j]$ — длина наидлиннейшей неубывающей последовательности на префиксе $[1;i]$, где j — тип последнего элемента. Переход динамики:

$$dp[i][j] = \begin{cases} 1 + \max_{k=1}^{j} dp[i-1][k], & \text{if } type[i] = j \\ dp[i-1][j], & \text{otherwise} \end{cases}$$

Для лёгкой реализации, хватает завести один массив $dp[j]$, и пропустить обработку второго случая.

Время: $O(n^2)$ / $O(n \log n)$. Память: $O(n^2)$ / $O(n)$.

Решение C++

```
#include<cstdio>
#include<algorithm>
#include<iostream>

using namespace std;

#define MAXN 5005

int n, m, type[MAXN], dp[MAXN];

int main() {
	scanf("%d %d", &n, &m);
	for(int i = 1; i <= n; i++) {
		double x;
		scanf("%d %lf", type+i, &x);
	}

	for(int i = 1; i <= n; i++) {
		int j = type[i];
		for(int k = j; k >= 1; k--) {
			dp[j] = max(dp[j], 1+dp[k]);
		}
	}

	int ans = 0;
	for(int i = 1; i <= n; i++) {
		ans = max(ans, dp[i]);
	}
	printf("%d\n", n-ans);
}
```

Решение Python

```
n,m=map(int,input().split())
l=[]
for i in range(n):
	a,b=input().split()
	l.append(int(a))
dp=[1]*n
for i in range(1,n):
	for j in range(0,i):
		if l[i]>=l[j]:
			dp[i]=max(dp[i],dp[j]+1)
print(n-max(dp))
```

J Холмы

The problem's short statement is: "we allowed to decrease any element and should create at least k local maximums, count the minimum number of operations for all k".

Notice, that any set of positions, where no positions are adjacent could be made to be local maximums — we just need to decrease the neighbouring hills to some value.

Let's introduce the following dynamic programming:

dp[prefix][local_maxs] – the minimum cost if we analyze only given prefix, have the specified number of local maximums ("good hills to build on") and we make a local maximum in the last hill of this prefix.

Th dumb implementation of this leads to $O(n^2)$ states and $O(n^4)$ time — in each state we can brute force the previous position of local maximum (n) and then calculate the cost of patching the segment from previous local maximum to current one.

A more attentive look says that it is, in fact $O(n^3)$ solution — on the segment only first and last elements need to be decreased (possibly first and last elements are same).

To get the full solution full solution in $O(n^2)$ we need to optimize dp a little bit. As we noticed in the previous paragraph, there is one extreme situation, when the first and elements are same, let's handle this transition by hand in $O(1)$ for each state.

Otherwise, funny fact, the cost of the segment strictly between local maximums is the cost of it's left part plus it'cost of it's right part. Seems like something we can decompose, right?

Since our goal is to update state ($prefix$, $local$) now the right part is fixed constant for all such transitions. And we need to select minimum value of $dp[i][local - 1] + cost(i, i + 1)$ where $i <= prefix - 3$.

This can be done by calculating a supplementary dp during the primarily dp calculation — for example we can calculate f[pref][j] = min dp[i][j] + cost(i, i + 1) for $i \le pref$.

Решение C++

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

const int N = 5e3+5;
const int INF = 1e9;
int n, A[N], dp[N][N][2];
bool chk[N][N][2];

int solve(int i, int j, int v) {
    int &z = dp[i][j][v];
    if(chk[i][j][v]) return z;
    if(i == 0 and v == 0) return 0;
    if(i == 0 and v == 1) return max(0, A[j+1] - A[j+2] + 1);
    if(j <= 0) return INF;
    if(v == 0)
        z = min(solve(i, j-1, 0), solve(i-1, j-2, 1) + max(0, A[j+1] - A[j] +
1));
    else
        z = min(solve(i, j-1, 0) + max(0, A[j+1] - A[j+2] + 1),
            solve(i-1, j-2, 1) + max(0, A[j+1] - min(A[j], A[j+2]) + 1));
    chk[i][j][v] = true;
    return z;
}

int main() {
    scanf("%d", &n);
    for(int i = 1; i <= n; ++i) scanf("%d", A+i);
    A[0] = A[n+1] = -INF;
    for(int i = 1; i <= (n+1)/2; ++i)
        printf("%d ", solve(i, n, 0));
}
```