# Recursão

## Definição e exemplos: problemas resolvidos

Prof. Edson Alves

2018

Faculdade UnB Gama

# Sumário

# URI 1029 – Fibonacci, quantas chamadas?

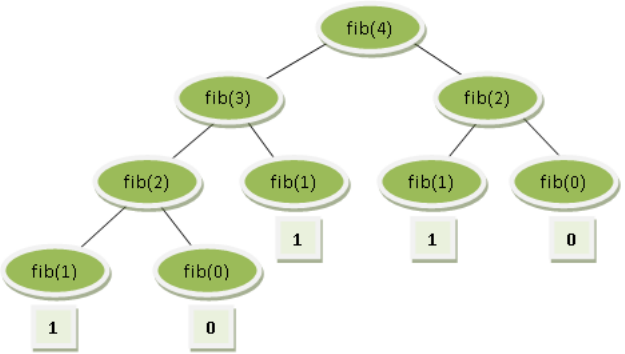
# Problema

Quase todo estudante de Ciência da Computação recebe em algum momento no início de seu curso de graduação algum problema envolvendo a sequência de Fibonacci. Tal sequência tem como os dois primeiros valores 0 (zero) e 1 (um) e cada próximo valor será sempre a soma dos dois valores imediatamente anteriores. Por definição, podemos apresentar a seguinte fórmula para encontrar qualquer número da sequência de Fibonacci:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2);
```

Uma das formas de encontrar o número de Fibonacci é através de chamadas recursivas. Isto é ilustrado a seguir, apresentando a árvore de derivação ao calcularmos o valor `fib(4)`, ou seja o 5º valor desta sequência:

# Problema




Desta forma,

`fib(4) = 1+0+1+1+0 = 3`

Foram feitas 8 calls, ou seja, 8 chamadas recursivas.

# Entrada e saída

**Entrada**

A primeira linha da entrada contém um único inteiro $N$, indicando o número de casos de teste. Cada caso de teste contém um inteiro $X$ $(1 \leq X \leq 39)$.

**Saída**

Para cada caso de teste de entrada deverá ser apresentada uma linha de saída, no seguinte formato: `fib(n) = num_calls calls = result`, aonde `num_calls` é o número de chamadas recursivas, tendo sempre um espaço antes e depois do sinal de igualdade, conforme o exemplo abaixo.

# Exemplo de entradas e saídas

**Exemplo de Entrada**

```
2
5
4
```

**Exemplo de Saída**

```
fib(5) = 14 calls = 5
fib(4) = 8 calls = 3
```

# Solução por força bruta

- Basta acumular o número de chamadas na própria implementação recursiva dos números de Fibonacci
- Este acumulador deve ser ou uma variável global (o que simplifica a codificação) ou um parâmetro passado como referência (o que torna a implementação mais trabalhosa)
- Como para cada valor de $X$ maior do que 1 há duas chamadas, um limite superior para o número de chamadas é $2^X$
- Como $X \leq 39$, e $2^{39}$ é um valor superior à capacidade de uma variável inteira, o mais prudente é utilizar uma variável do tipo **`long long`** para o acumulador

# Solução AC/TLE com complexidade $O(N2^X)$

```cpp
#include <iostream>

int fibonacci(int X, long long& calls)
{
    if (X == 0 || X == 1)
        return X;

    auto A = fibonacci(X - 1, calls);
    auto B = fibonacci(X - 2, calls);

    calls += 2;

    return A + B;
}

```

# Solução AC/TLE com complexidade $O(N2^X)$

```
int main()
{
    int N;
    std::cin >> N;

    while (N--)
    {
        int X;
        std::cin >> X;

        long long calls = 0;
        auto ans = fibonacci(X, calls);

        std::cout << "fib(" << X << ") = " << calls
            << " calls = " << ans << '\n';
    }

    return 0;
}
```

# Solução mais eficiente

- Para diminuir a complexidade assintótica da solução, primeiro é preciso observar que os números de Fibonacci podem ser computados sem o uso de recursão
- Para tal, basta armazenar os valores em um vetor, e computar o próximo valor a partir dos dois últimos valores já armazenados
- Além disso, é preciso observar que o número de chamadas $C(X)$ também é definido por uma recorrência, semelhante à de Fibonacci:

$$C(X) = \begin{cases} 0, & \text{se } X = 0 \text{ ou } X = 1 \\ C(X-1) + C(X-2) + 2, & \text{se } X > 1 \end{cases}$$

- Os valores de $C(X)$ podem ser computados da mesma maneira
- Assim, o tempo de execução passa a ter complexidade $O(NX)$, e memória $O(X)$

# Solução AC com complexidade $O(NX)$

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

using ll = long long;
const int MAX { 40 };

std::vector<ll> fibs(MAX + 1), calls(MAX + 1);

void precomp()
{
    fibs[0] = 0;
    fibs[1] = 1;

    calls[0] = calls[1] = 0;

    for (int i = 2; i <= MAX; ++i)
    {
        fibs[i] = fibs[i - 1] + fibs[i - 2];
        calls[i] = calls[i - 1] + calls[i - 2] + 2;
    }
}
```

# Solução AC com complexidade $O(NX)$

```cpp

int main()
{
    precomp();

    int N;
    std::cin >> N;

    while (N--)
    {
        int X;
        std::cin >> X;

        std::cout << "fib(" << X << ") = " << calls[X]
            << " calls = " << fibs[X] << '\n';
    }

    return 0;
}
```

# UVA 10229 – Modular Fibonacci

# Problema

The Fibonacci numbers (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...) are defined by the recurrence:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \text{ for } i > 1$$

Write a program which calculates $M_n = F_n \bmod 2^m$ for given pair of $n$ and $m$. $0 \leq n \leq 2147483647$ and $0 \leq m < 20$. Note that $a \bmod b$ gives the remainder when $a$ is divided by $b$.

# Entrada e saída

**Input**

Input consists of several lines specifying a pair of $n$ and $m$.

**Output**

Output should be corresponding $M_n$, one per line.

# Exemplo de entradas e saídas

**Exemplo de Entrada**

```
11 7
11 6
```

**Exemplo de Saída**

```
89
25
```

# Solução recursiva

- A implementação recursiva dos números de Fibonacci trazem uma série de problemas
- Primeiramente, a quantidade de chamadas recursivas é demasiadamente alto, o que leva ao TLE
- Além disso, pode acontecer um estouro na pilha de execução, levando a um RTE
- O tipo de variável usada para armazenar os valores pode levar a erros de *overflow*, caso o módulo seja computado apenas ao final
- Para computar o valor exato de $F_n$ para $n \geq 100$ é necessário utilizar aritmética estendida

# Solução WA/TLE com complexidade $O(T2^n)$

```cpp
#include <iostream>

int fib(int n)
{
    if (n < 2)
        return n;

    return fib(n - 1) + fib(n - 2);
}

```

# Solução WA/TLE com complexidade $O(T2^n)$

```cpp
int main()
{
    std::ios::sync_with_stdio(false);

    int n, m;

    while (std::cin >> n >> m)
    {
        int M = (1 << m);
        auto ans = fib(n) % M;

        std::cout << ans << '\n'
    }

    return 0;
}
```

# Solução iterativa

- A implementação iterativa dos números de Fibonacci resolvem dois dos problemas apresentados anteriormente
- Primeiramente, não há mais chamadas recursivas
- Além disso, não há estouro da pilha de execução
- Embora a complexidade seja reduzida para $O(Tn)$, ainda assim a solução não é rápida o suficiente para o tamanho máximo de $n$
- As operações modulares a cada etapa evitam o problema de *overflow*

# Solução TLE com complexidade $O(Tn)$

```cpp
#include <iostream>

int fib(int n, int m)
{
    if (n < 2)
        return n;

    int M = (1 << m);
    int a = 0, b = 1;

    for (int i = 2; i <= n; ++i)
    {
        auto temp = (a + b) % M;
        a = b;
        b = temp;
    }

    return b;
}
```

# Solução TLE com complexidade $O(Tn)$

```cpp
int main()
{
    std::ios::sync_with_stdio(false);

    int n, m;

    while (std::cin >> n >> m)
    {
        auto ans = fib(n, m);

        std::cout << ans << '\n';
    }

    return 0;
}
```

# Forma matricial da recorrência de Fibonacci

- Para uma solução com complexidade inferior à linear, é preciso resolver a recorrência
- Sabemos que $F_{n+2} = F_{n+1} + F_n$ para $n > 1$, e que $F(0) = 0, F(1) = 1$
- Esta relação pode ser representada matricialmente:

$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

- Seja

$$u_n = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \quad \text{e} \quad u_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

- Assim, $u_{n+1} = Au_n$

# Fórmula para os termos de Fibonacci sem recorrência

- Expandindo a recorrência tem-se que

$$\begin{aligned} u_1 &= Au_0 \\ u_2 &= Au_1 = A(Au_0) = A^2u_0 \\ u_3 &= Au_2 = A(A^2u_0) = A^3u_0 \\ &\ldots, \end{aligned}$$

  isto é, $u_n = A^n u_0$
- A expressão acima, que não possui recorrência, permite computar o $n$-ésimo termo da sequência de Fibonacci, se conhecido o valor de $A^n$
- Esta exponenciação matricial pode ser feita, de forma eficiente, se a matriz $A$ possuir autovetores linearmente independentes

## Autovalores de Fibonacci

- O vetor $\vec{x}$ é autovetor de $A$ se existir um $\lambda$ real (denominado autovalor) tal que $A\vec{x} = \lambda\vec{x}$
- Para encontrar os autovalores de $A$, é preciso encontrar os valores $\lambda$ tais que $(A - \lambda I)\vec{x} = \vec{0}$, o que ocorre quando

$$\det \begin{bmatrix} 1-\lambda & 1 \\ 1 & -\lambda \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

  o que leva à expressão

$$p(\lambda) = -\lambda(1-\lambda) - 1 = 0$$

- Os zeros do polinômio $p(\lambda) = \lambda^2 - \lambda - 1$ são os autovetores de Fibonacci, a saber:

$$\lambda_1 = \frac{1+\sqrt{5}}{2} \text{ e } \lambda_2 = \frac{1-\sqrt{5}}{2}$$

# Autovetores de Fibonacci

- Para encontrar os autovetores de Fibonacci, basta levar os autovalores novamente na igualdade anterior
- Para $\lambda_1$ segue que

$$\begin{bmatrix} 1-\lambda_1 & 1 \\ 1 & -\lambda_1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

  cuja solução não trivial é

$$\vec{v}_1 = \begin{bmatrix} \lambda_1 \\ 1 \end{bmatrix},$$

  pois $\lambda_1 - \lambda_1^2 + 1 = 0$, uma vez que $\lambda_1$ é autovalor
- O mesmo vale para $\lambda_2$, obtendo o segundo autovetor

$$\vec{v}_2 = \begin{bmatrix} \lambda_2 \\ 1 \end{bmatrix}$$

# Diagonalização da matrix $A$

- Como os autovalores $\vec{v}_1$ e $\vec{v}_2$ são linearmente independentes, a matriz $A$ é diagonalizável
- Seja $S$ a matriz dos autovetores e $\Lambda$ a matriz diagonal dos autovalores correspondentes, isto é,

$$S = \begin{bmatrix} \lambda_1 & \lambda_2 \\ 1 & 1 \end{bmatrix} \text{ e } \Lambda = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

- A definição de autovetor implica que $AS = S\Lambda$, e como $S$ é invertível, segue que

$$A = S\Lambda S^{-1}$$

# Exponenciação eficiente de matrizes diagonalizadas

- A expressão $A = S\Lambda S^{-1}$ permite computar $A^n$ de forma eficiente, pois

$$
\begin{aligned}
A^2 &= (S\Lambda S^{-1})(S\Lambda S^{-1}) = S\Lambda(S^{-1}S)\Lambda S^{-1} = S\Lambda^2 S^{-1} \\
A^3 &= A^2 A = (S\Lambda^2 S^{-1})(S\Lambda S^{-1}) = S\Lambda^2(S^{-1}S)\Lambda S^{-1} = S\Lambda^3 S^{-1} \\
&\ldots
\end{aligned}
$$

- Assim, $A^n = S\Lambda^n S^{-1}$, com

$$
\Lambda^n = \begin{bmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{bmatrix} \quad \text{e} \quad S^{-1} = \frac{1}{\lambda_1 - \lambda_2} \begin{bmatrix} 1 & -\lambda_2 \\ -1 & \lambda_1 \end{bmatrix}
$$

- A expressão para os números de Fibonacci sem recorrência leva a

$$
u_n = A^n u_0 = (S\Lambda^n S^{-1})u_0 = S\Lambda^n (S^{-1})u_0 = S\Lambda^n \vec{c},
$$

onde o vetor $\vec{c}$ é constante

# Fórmula de Binet

- De fato,

$$\vec{c} = \frac{1}{\lambda_1 - \lambda_2} \begin{bmatrix} 1 & -\lambda_2 \\ -1 & \lambda_1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\lambda_1 - \lambda_2} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{5}} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

- Portanto, como $u_n = S\Lambda^n \vec{c}$,

$$\begin{aligned} \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} &= \frac{1}{\sqrt{5}} \begin{bmatrix} \lambda_1 & \lambda_2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} \\ &= \frac{1}{\sqrt{5}} \begin{bmatrix} \lambda_1 & \lambda_2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \lambda_1^n \\ -\lambda_2^n \end{bmatrix} = \frac{1}{\sqrt{5}} \begin{bmatrix} \lambda_1^{n+1} + \lambda_2^{n+1} \\ \lambda_1^n + \lambda_2^n \end{bmatrix} \end{aligned}$$

- A Fórmula de Binet corresponde à segunda componente do vetor acima, isto é,

$$F_n = \frac{1}{\sqrt{5}} \left[ \left( \frac{1+\sqrt{5}}{2} \right)^n + \left( \frac{1-\sqrt{5}}{2} \right)^n \right]$$

# Solução WA usando a fórmula de Binet

```
#include <iostream>
#include <cmath>

int fib(int n)
{
    auto L = pow((1 + sqrt(5))/2, n);
    auto R = pow((1 - sqrt(5))/2, n);

    return round((L - R)/sqrt(5));
}

```

# Solução WA usando a fórmula de Binet

```
int main()
{
    std::ios::sync_with_stdio(false);

    int n, m;

    while (std::cin >> n >> m)
    {
        int M = (1 << m);
        auto ans = fib(n) % M;

        std::cout << ans << std::endl;
    }

    return 0;
}
```

# Exponenciação rápida de matrizes

- Retornando à igualdade $u_{n+1} = A^n u_0$, é possível computar $A^n$ com complexidade $O(\log n)$ sem o uso de autovetores
- Basta observar que

$$A^n = \begin{cases} A, & \text{se } n = 1 \\ A^{\frac{n}{2}} A^{\frac{n}{2}}, & \text{se n é par} \\ A^{\frac{n}{2}} A^{\frac{n}{2}} A, & \text{caso contrário} \end{cases}$$

- Vale a igualdade

$$A^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix},$$

a qual pode ser provada por indução

# Exponenciação rápida de matrizes

- Para $n = 1$

$$A^1 = A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix}$$

- Suponha que a afirmativa seja verdadeira para $m$ natural. Para $m + 1$ segue que

$$\begin{aligned} A^{m+1} = A^m A &= \begin{bmatrix} F_{m+1} & F_m \\ F_m & F_{m-1} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} F_{m+1} + F_m & F_{m+1} \\ F_m + F_{m-1} & F_m \end{bmatrix} \\ &= \begin{bmatrix} F_{m+2} & F_{m+1} \\ F_{m+1} & F_m \end{bmatrix} \end{aligned}$$

- Portanto a afirmativa é verdadeira para qualquer $n$ natural

# Implementação recursiva da exponencição rápida de matrizes

```
// Definição da classe Matrix, com o operador * implementado

Matrix fast_mod_pow(const Matrix& A, int n, int M)
{
    if (n == 1)
        return A;

    auto B = fast_mod_pow(A, n / 2, M);

    return n & 1 ? B * B * A : B * B;
}
```

# Implementação iterativa da exponencição rápida de matrizes

```cpp
// Definição da classe Matrix, com o operador * implementado

Matrix fast_pow_mod(const Matrix& A, int n, int m)
{
    auto res = Matrix(1, 0, 0, 1, m), base = A;

    while (n)
    {
        if (n & 1)
            res = res * base;

        base = base * base;
        n >>= 1;
    }

    return res;
}
```

# Solução AC com complexidade $O(T \log n)$

```
#include <iostream>

struct Matrix
{
    long long a, b, c, d, m;

    Matrix(int av = 1, int bv = 0, int cv = 0, int dv = 1, int mv = 0)
        : a(av), b(bv), c(cv), d(dv), m(mv) {}

    Matrix operator*(const Matrix& A) const
    {
        long long M = (1 << m);
        auto ra = (a * A.a + b * A.c) % M;
        auto rb = (a * A.b + b * A.d) % M;
        auto rc = (c * A.a + d * A.c) % M;
        auto rd = (c * A.b + d * A.d) % M;

        return Matrix(ra, rb, rc, rd, m);
    }
};

```

# Solução AC com complexidade $O(T \log n)$

```
Matrix fast_pow_mod(const Matrix& A, int n, int m)
{
    auto res = Matrix(1, 0, 0, 1, m), base = A;

    while (n) {
        if (n & 1)
            res = res * base;

        base = base * base;
        n >>= 1;
    }

    return res;
}

int fib(int n, int m)
{
    auto A = Matrix(1, 1, 1, 0, m);
    auto F = fast_pow_mod(A, n, m);
    return F.b;
}
```

# Solução AC com complexidade $O(T \log n)$

```cpp

int main()
{
    std::ios::sync_with_stdio(false);

    int n, m;

    while (std::cin >> n >> m)
    {
        auto ans = fib(n, m);

        std::cout << ans << std::endl;
    }

    return 0;
}
```

# Relação entre números de Fibonacci

- Da igualdade

$$A^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

e do fato de que $A^{m+n} = A^m A^n$ se que

$$A^{n+m} = \begin{bmatrix} F_{m+1} & F_m \\ F_m & F_{m-1} \end{bmatrix} \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

- Daí

$$F_{n+m} = F_{m+1}F_n + F_m F_{n-1}$$

e

$$F_{n+m-1} = F_m F_n + F_{m-1} F_{n-1}$$

# Relação entre números de Fibonacci

- Fazendo $n = 2k$ (isto é, $n$ é par), tem-se que

$$\begin{aligned} F_n = F_{2k} = F_{k+k} &= F_{k+1}F_k + F_kF_{k-1} \\ &= (F_k + F_{k-1})F_k + F_kF_{k-1} \\ &= (2F_{k-1} + F_k)F_k \end{aligned}$$

- Se $n = 2k - 1$, (isto é, $n$ é ímpar),

$$F_n = F_{2k-1} = F_{k+k-1} = F_k^2 + F_{k-1}^2$$

- Estas identidades também permitem computar o $n$-ésimo número de Fibonacci com complexidade $O(n)$, porém com uma codificação mais curta em relação à exponenciação matricial rápida
- Note que o número de valores intermediários a serem computados é inferior a $4 \log n$

# Solução $O(T \log n)$

```cpp
#include <iostream>
#include <map>

using ll = long long;

std::map<ll, ll> fibs;

ll fib(ll n, ll M)
{
    if (fibs.count(n))
        return fibs[n];

    ll k = (n + 1)/2;

    fibs[n] = n & 1 ?
        (fib(k, M)*fib(k, M) + fib(k - 1, M)*fib(k - 1, M)) % M :
        ((2*fib(k - 1, M) + fib(k, M))*fib(k, M)) % M;

    return fibs[n];
}

```

# Solução $O(T \log n)$

```
int main()
{
    std::ios::sync_with_stdio(false);

    int n, m;

    while (std::cin >> n >> m)
    {
        fibs.clear();
        auto M = (1LL << m);

        fibs[0] = 0;
        fibs[1] = 1 % M;

        std::cout << fib(n, M) << '\n';
    }

    return 0;
}
```

# Referências


1. URI 1029 – Fibonacci, Quantas Chamadas?
2. UVA 10229 – Modular Fibonacci
3. **kein_coi_1997**. *An amazing way to calculate* $10^{18}$*-th fibonacci number using 25 lines of code*, acesso em 27/02/2019.[1]
4. Wikipédia. *Fibonacci Number*, acesso em 27/02/2019.[2]


[1] https://codeforces.com/blog/entry/14516

[2] https://en.wikipedia.org/wiki/Fibonacci_number