

# Recursão

Definição e exemplos: problemas resolvidos

---

Prof. Edson Alves

2018

Faculdade UnB Gama

1. URI 1029 – Fibonacci, quantas chamadas?
2. UVA 10229 – Modular Fibonacci

**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:

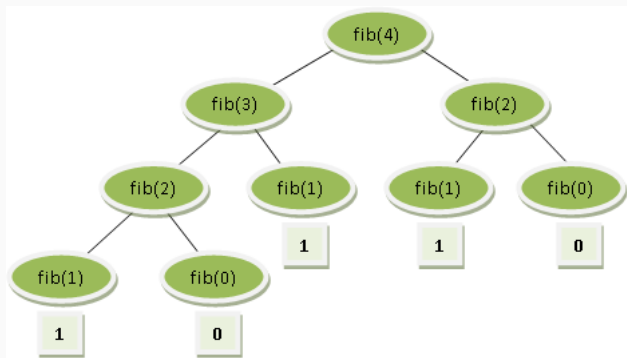
$$\text{fib}(0) = 0$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2);$$

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  $\text{fib}(4)$ , ou seja o 5º valor desta sequência:

# Problema



Desta forma,

$$\text{fib}(4) = 1+0+1+1+0 = 3$$

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 $O(N2^X)$

```
1 #include <iostream>
2
3 int fibonacci(int X, long long& calls)
4 {
5     if (X == 0 || X == 1)
6         return X;
7
8     auto A = fibonacci(X - 1, calls);
9     auto B = fibonacci(X - 2, calls);
10
11     calls += 2;
12
13     return A + B;
14 }
15
```

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

```
16 int main()
17 {
18     int N;
19     std::cin >> N;
20
21     while (N--)
22     {
23         int X;
24         std::cin >> X;
25
26         long long calls = 0;
27         auto ans = fibonacci(X, calls);
28
29         std::cout << "fib(" << X << ") = " << calls
30             << " calls = " << ans << '\n';
31     }
32
33     return 0;
34 }
```

## 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 $O(NX)$

```
1 #include <iostream>
2 #include <vector>
3
4 using ll = long long;
5 const int MAX { 40 };
6
7 std::vector<ll> fibs(MAX + 1), calls(MAX + 1);
8
9 void precomp()
10 {
11     fibs[0] = 0;
12     fibs[1] = 1;
13
14     calls[0] = calls[1] = 0;
15
16     for (int i = 2; i <= MAX; ++i)
17     {
18         fibs[i] = fibs[i - 1] + fibs[i - 2];
19         calls[i] = calls[i - 1] + calls[i - 2] + 2;
20     }
21 }
```

## Solução AC com complexidade $O(NX)$

```
22
23 int main()
24 {
25     precomp();
26
27     int N;
28     std::cin >> N;
29
30     while (N--)
31     {
32         int X;
33         std::cin >> X;
34
35         std::cout << "fib(" << X << ") = " << calls[X]
36             << " calls = " << fibs[X] << '\n';
37     }
38
39     return 0;
40 }
```

## **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  $b$ .

## 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 $O(T2^n)$

```
1 #include <iostream>
2
3 int fib(int n)
4 {
5     if (n < 2)
6         return n;
7
8     return fib(n - 1) + fib(n - 2);
9 }
10
```

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

```
11 int main()
12 {
13     std::ios::sync_with_stdio(false);
14
15     int n, m;
16
17     while (std::cin >> n >> m)
18     {
19         int M = (1 << m);
20         auto ans = fib(n) % M;
21
22         std::cout << ans << '\n'
23     }
24
25     return 0;
26 }
```

- 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 $O(Tn)$

```
1 #include <iostream>
2
3 int fib(int n, int m)
4 {
5     if (n < 2)
6         return n;
7
8     int M = (1 << m);
9     int a = 0, b = 1;
10
11     for (int i = 2; i <= n; ++i)
12     {
13         auto temp = (a + b) % M;
14         a = b;
15         b = temp;
16     }
17
18     return b;
19 }
20
```

## Solução TLE com complexidade $O(Tn)$

```
21 int main()
22 {
23     std::ios::sync_with_stdio(false);
24
25     int n, m;
26
27     while (std::cin >> n >> m)
28     {
29         auto ans = fib(n, m);
30
31         std::cout << ans << '\n';
32     }
33
34     return 0;
35 }
```

# 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

$$u_1 = Au_0$$

$$u_2 = Au_1 = A(Au_0) = A^2u_0$$

$$u_3 = Au_2 = A(A^2u_0) = A^3u_0$$

$$\dots,$$

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:

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

# 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,

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

- 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

$$A^2 = (S\Lambda S^{-1})(S\Lambda S^{-1}) = S\Lambda(S^{-1}S)\Lambda S^{-1} = S\Lambda^2 S^{-1}$$

$$A^3 = A^2 A = (S\Lambda^2 S^{-1})(S\Lambda S^{-1}) = S\Lambda^2(S^{-1}S)\Lambda S^{-1} = S\Lambda^3 S^{-1}$$

...

- 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,

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

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.<sup>1</sup>
4. Wikipédia. *Fibonacci Number*, acesso em 27/02/2019.<sup>2</sup>

---

<sup>1</sup><https://codeforces.com/blog/entry/14516>

<sup>2</sup>[https://en.wikipedia.org/wiki/Fibonacci\\_number](https://en.wikipedia.org/wiki/Fibonacci_number)