Programação Estruturada

Prof. Rodrigo Hausen
http://progest.compscinet.org

Recursão

O QUE É?

Definição recursiva é aquela que define os elementos de um conjunto em função de outros elementos desse mesmo conjunto.

Exemplo: sequência de Fibonacci

$$F_0, F_1, F_2, \ldots, F_n, \ldots$$

é um conjunto infinito definido por:

$$F_0 := 0$$
$$F_1 := 1$$
$$F_n := F_{n-1} + F_{n-2}, \text{ se } n \geq 2$$

Exemplo: sequência de Fibonacci

$F_0 := 0$

$F_1 := 1$

$F_n := F_{n-1} - F_{n-2}$, se $n \geq 2$

Pela definição:

$F_0 = 0$

$F_1 = 1$

$F_2 = F_1 + F_0 = 1$

$F_3 = F_2 + F_1 = 2$

$F_4 = F_3 + F_2 = 3$

$F_5 = F_4 + F_3 = 5$

$F_6 = F_5 + F_4 = 8$

Calcule F_9.

FUNÇÕES RECURSIVAS

Função recursiva é aquela definida usando-se a própria função.

Exemplo: considere a função

$$f : \mathbb{N} \to \mathbb{N}$$

$$n \mapsto f(n) := \begin{cases} 0, \text{ se } n \text{ é } 0 \\ 1, \text{ se } n \text{ é } 1 \\ f(n-1) + f(n-2), \text{ se } n \geq 2 \end{cases}$$

Observe que, pela forma como definimos a função, $f(n)$ equivale a F_n.

FUNÇÕES RECURSIVAS em C

Podemos implementar funções de maneira recursiva em C.

Exemplo:

```
unsigned int fibonacci(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

FUNÇÕES RECURSIVAS em C

Podemos implementar funções de maneira recursiva em C.

Exemplo:

```
unsigned int fibonacci(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

casos base

chamadas recursivas

Outro exemplo

Fatorial de um número definido recursivamente:

$$0! := 1$$
$$n! := (n-1)! \times n$$

Compare com a definição não recursiva:

$$n! := \begin{cases} 1, \text{ se } n \text{ é } 0 \\ \\ n \cdot (n-1) \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1, \text{ se } n \geq 1 \end{cases}$$

Implemente a função fatorial de maneira recursiva e não recursiva em C.

```
// implementação recursiva
unsigned int fatorial(unsigned int n) {
  if (n == 0) return 1;
  return n*fatorial(n-1);
}
```

```
// implementação não recursiva
unsigned int fatorial(unsigned int n) {
  unsigned int produto = 1;
  unsigned int i;
  for(i = n; i >= 1; --i) {
    produto *= i;
  }
  return produto;
}
```

```
// implementação recursiva
unsigned int fatorial(unsigned int n) {
  if (n == 0) return 1;  ← caso base
  return n*fatorial(n-1);
}                    ↖
                       chamada recursiva
```

```
// implementação não recursiva
unsigned int fatorial(unsigned int n) {
  unsigned int produto = 1;
  unsigned int i;
  for(i = n; i >= 1; --i) {
    produto *= i;
  }
  return produto;
}
```

LEMBRE-SE

Em C, **toda** função recursiva **sempre** deve ter:

- → pelo menos 1 chamada recursiva (óbvio, senão não seria uma função recursiva)
- → pelo menos 1 caso base (caso contrário a recursão não terá fim, ou seja, a execução nunca termina)

Verifique **sempre** se os casos base são atingidos para quaisquer valores de entrada e se **todos** os casos base foram considerados.

FALTA DE CASO BASE: consequências

Nesta função recursiva, o programador esqueceu de considerar o caso base:

```
unsigned int fatorial(unsigned int n) {
  if (n == 0) return 1;
  return n*fatorial(n-1);
}
```

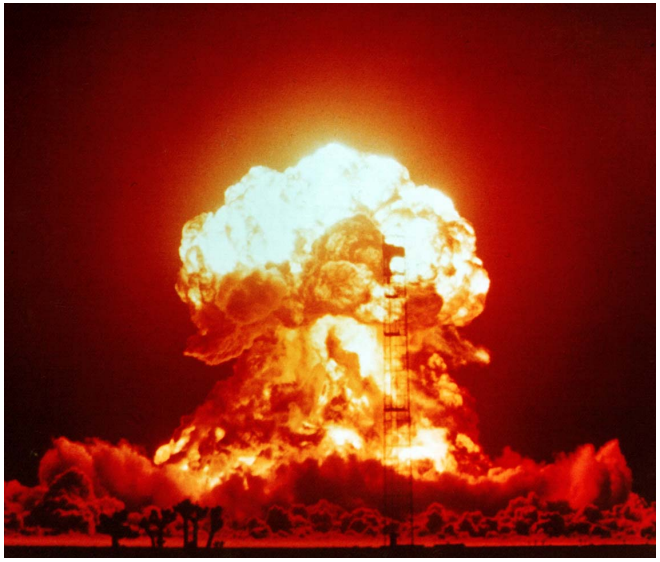
```
[cling]$ .L fatorial.c
[cling]$ fatorial(2)
```

FALTA DE CASO BASE: consequências

Nesta função recursiva, o programador esqueceu de considerar o caso base:

```
unsigned int fatorial(unsigned int n) {
  if (n == 0) return 1;
  return n*fatorial(n-1);
}
```

```
[cling]$ .L fatorial.c
[cling]$ fatorial(2)
Falha de segmentação
```



FALTA DE CASO BASE: consequências

```
#include <stdio.h>
unsigned int fatorial(unsigned int n) {
  printf("fatorial(%u)\n", n);
  if (n == 0) return 1;
  return n*fatorial(n-1);
}
```

```
[cling]$ fatorial(2)
```

FALTA DE CASO BASE: consequências

```
#include <stdio.h>
unsigned int fatorial(unsigned int n) {
  printf("fatorial(%u)\n", n);
  if (n == 0) return 1;
  return n*fatorial(n-1);
}
```

```
[cling]$ fatorial(2)
fatorial(2)
fatorial(1)
fatorial(0)
fatorial(4294967295)
fatorial(4294967294)
…
Falha de segmentação
```

FALTA DE CASO BASE: consequências

```
#include <stdio.h>
unsigned int fatorial(unsigned int n) {
  printf("fatorial(%u)\n", n);
  if (n == 0) return 1;
  return n*fatorial(n-1);
}
```

```
[cling]$ fatorial(2)
fatorial(2)
fatorial(1)
fatorial(0)
fatorial(4294967295)
fatorial(4294967294)
…
Falha de segmentação
```

underflow da variável *n*

falta de caso base faz com que não exista critério de parada. função executa indefinidamente

FALTA DE CASO BASE: consequências

Cada chamada a uma função ocupa espaço na região de memória chamada pilha.

Quando a função termina de ser executada, o espaço alocado à função é liberado

Em função recursiva, se o caso base nunca é atingido, a função chama a si própria mas nunca termina!

Cada chamada ocupa mais espaço na pilha.

Uma hora, o espaço na pilha acaba, resultando em *falha de segmentação*.

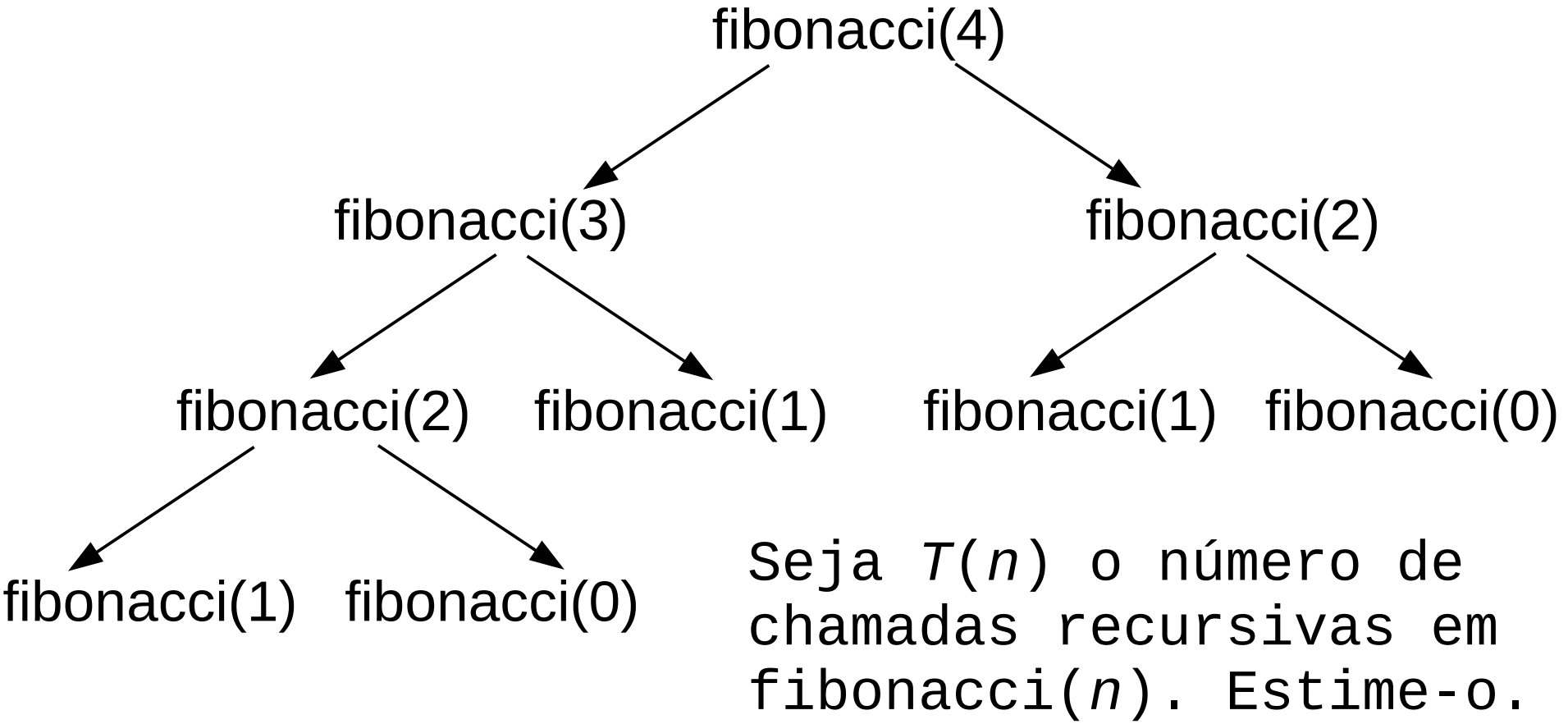
ESTOURO DE PILHA

Quando acaba a memória disponível para a pilha, temos uma condição chamada **estouro de pilha** ou **transbordamento de pilha** (*stack overflow*).

Um dos motivos para o esturo de pilha é a execução de função recursiva que não atinge nenhum caso base.

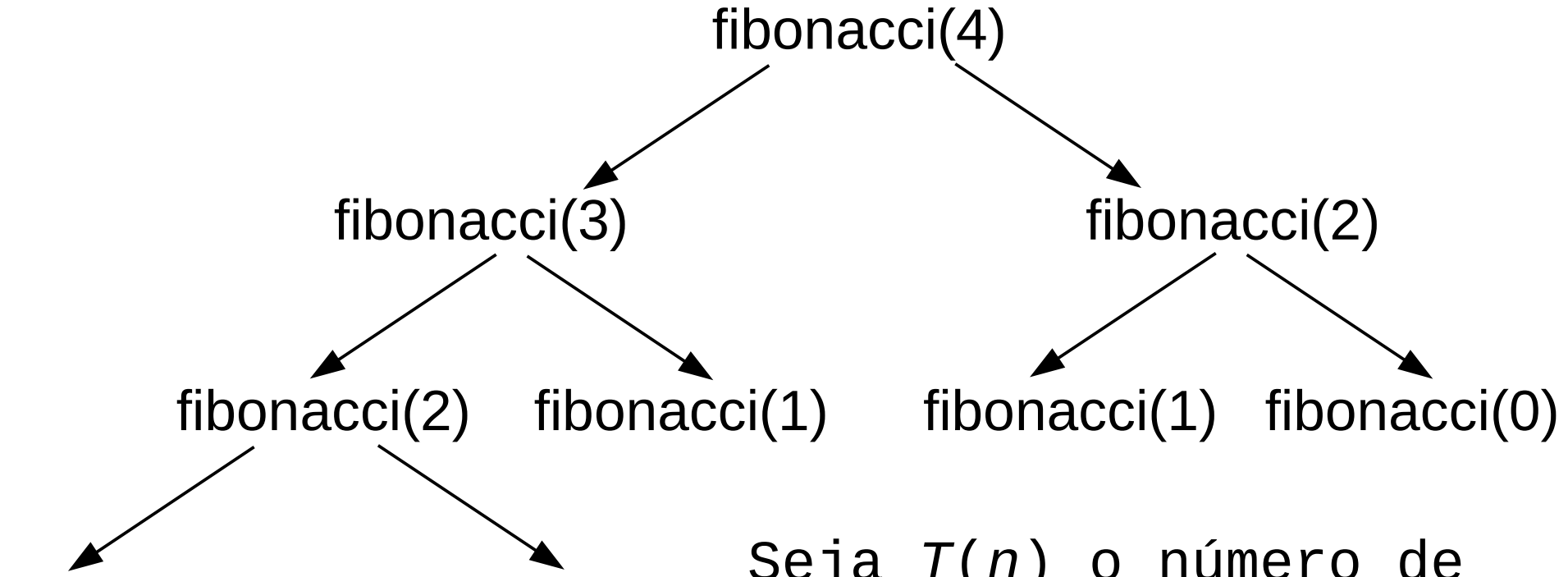
Porém, mesmo funções recursivas com casos base bem definidos podem estourar a pilha caso o número de chamadas recursivas cresça muito rapidamente.

GRAFO DE CHAMADAS de fibonacci(4)



Seja $T(n)$ o número de chamadas recursivas em fibonacci(n). Estime-o.

GRAFO DE CHAMADAS de fibonacci(4)



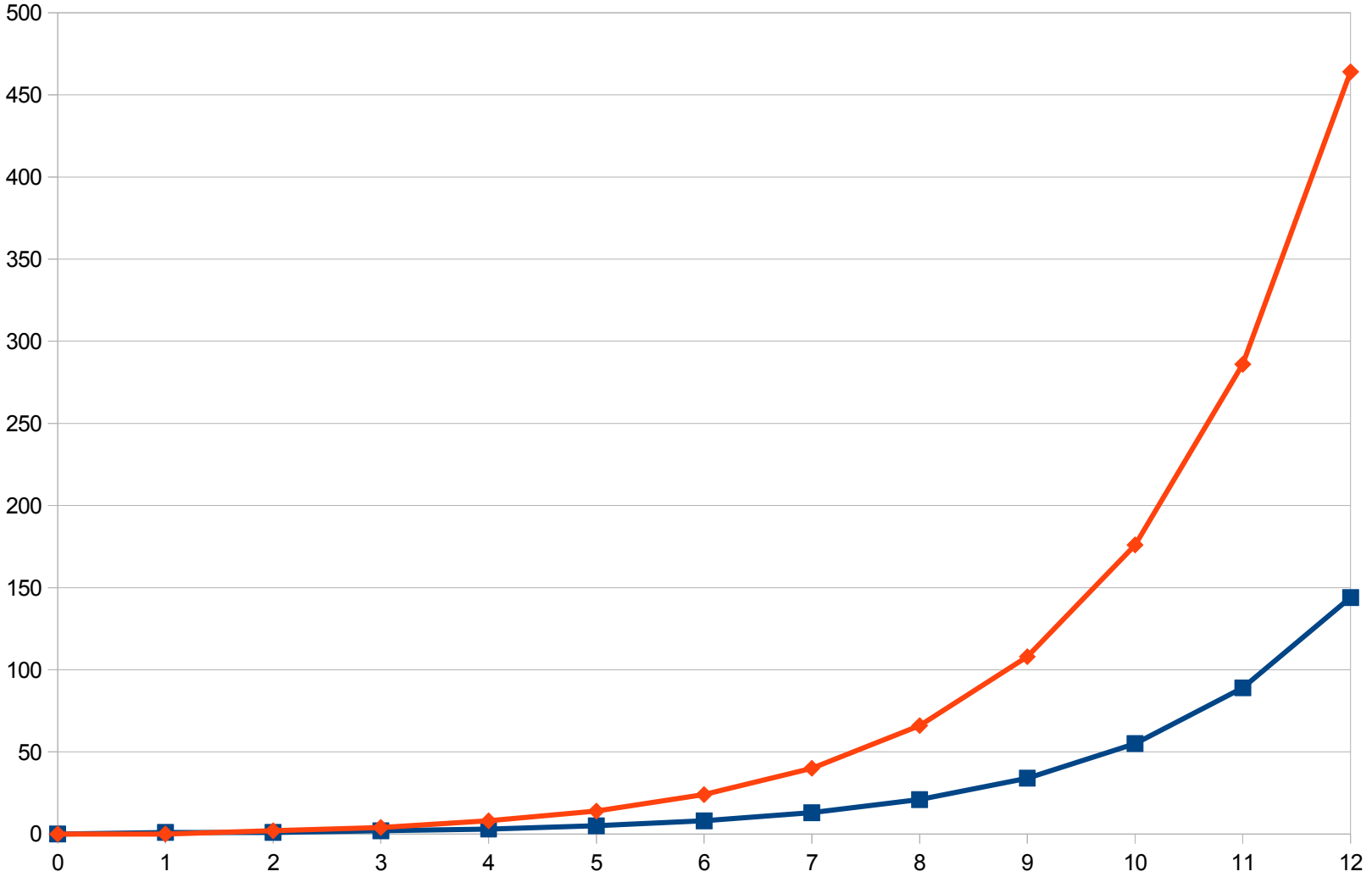
Seja $T(n)$ o número de chamadas recursivas em fibonacci(n). Estime-o.

$T(0) = 0$, $T(1) = 1$,

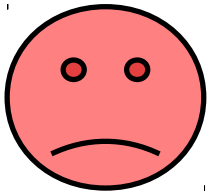
$T(n) = 2 + T(n-1) + T(n-2)$, $n \geq 2$

CHAMADAS RECURSIVAS em fibonacci(n)

$T(n) > F_n$ para $n \geq 2$



QUANDO **NÃO SE DEVE** USAR RECURSÃO?

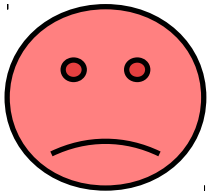


→ Quando o número de chamadas recursivas cresce muito rapidamente (risco de estouro de pilha para entradas muito pequenas)

→ Quando há soluções iterativas simples

```
unsigned int fatorial(unsigned int n) {
  if (n == 0) return 1;
  return n*fatorial(n-1);
}
```

QUANDO **NÃO SE DEVE** USAR RECURSÃO?



→ Quando o número de chamadas recursivas cresce muito rapidamente (risco de estouro de pilha para entradas muito pequenas)

→ Quando há soluções iterativas simples

```
unsigned int fatorial(unsigned int n) {
  int i, produto = 1;
  for (i=n; i>=1; --i) {
    produto *= n;
  }
  return produto;
}
```

QUANDO **NÃO SE DEVE** USAR RECURSÃO?

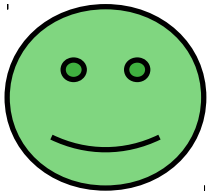
Transforme a implementação de fibonacci em uma solução iterativa.

```
unsigned int fibonacci(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

QUANDO **NÃO SE DEVE** USAR RECURSÃO?

```
unsigned int fibonacci(unsigned int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  int i, fn;
  int fnmenos1 = 1, fnmenos2 = 0;
  for (i = 2; i<= n; ++i) {
    fn = fnmenos1 + fnmenos2;
    fnmenos2 = fnmenos1;
    fnmenos1 = fn;
  }
  return fn;
}
```

QUANDO **É VANTAJOSO** USAR RECURSÃO?



→ Quando o número de chamadas cresce muito devagar

→ Quando o problema é definido de maneira recursiva e implementação é viável (p.ex. para algumas estruturas de dados, tipo árvores binárias)

Às vezes, uma solução recursiva pode ser ponto de partida para uma solução iterativa.

ESTUDOS DE CASO

→ Máximo Divisor Comum

→ Busca binária

MÁXIMO DIVISOR COMUM

Máximo Divisor Comum (MDC) entre dois números naturais a e b, tais que $a \neq 0$:

→ maior inteiro M tal que M é divisor de a e de b.

Exemplo: o MDC entre 84 e 72 é 12, pois 12 é o maior inteiro que divide ambos.

Veja que:
→ $\text{mdc}(a, 0) = a$
→ Se $b \neq 0$, $\text{mdc}(a, b) = \text{mdc}(b, a \bmod b)$

Onde $a \bmod b$ é o resto da divisão de a por b.

MÁXIMO DIVISOR COMUM

```
// cálculo do máximo divisor comum para
// números naturais a, b tais que a ≠ 0
unsigned int mdc(unsigned int a,
                 unsigned int b) {
  if (b == 0) return a;
  return mdc(b, a % b);
}
```

MÁXIMO DIVISOR COMUM

```
// cálculo do máximo divisor comum para
// números naturais a, b tais que a ≠ 0
unsigned int mdc(unsigned int a,
                 unsigned int b) {
  if (b == 0) return a;
  return mdc(b, a % b);
}

[cling]$ mdc(84, 72)
(unsigned int) 12
```

MÁXIMO DIVISOR COMUM

```
// cálculo do máximo divisor comum para
// números naturais a, b tais que a ≠ 0
unsigned int mdc(unsigned int a,
                 unsigned int b) {
  if (b == 0) return a;
  return mdc(b, a % b);
}

[cling]$ mdc(84, 72)
(unsigned int) 12
[cling]$ mdc(144, 90)
(unsigned int) 18
```

MÁXIMO DIVISOR COMUM

```
// cálculo do máximo divisor comum para
// números naturais a, b tais que a ≠ 0
unsigned int mdc(unsigned int a,
                 unsigned int b) {
  if (b == 0) return a;
  return mdc(b, a % b);
}

[cling]$ mdc(84, 72)
(unsigned int) 12
[cling]$ mdc(144, 90)
(unsigned int) 18
[cling]$ mdc(13, 5)
(unsigned int) 1
```

MÁXIMO DIVISOR COMUM

```
// cálculo do máximo divisor comum para
// números naturais a, b tais que a ≠ 0
unsigned int mdc(unsigned int a,
                 unsigned int b) {
  if (b == 0) return a;
  return mdc(b, a % b);
}

[cling]$ mdc(144, 90)
(unsigned int) 18
[cling]$ mdc(13, 5)
(unsigned int) 1
[cling]$ mdc(21, 0)
(unsigned int) 21
```

MÁXIMO DIVISOR COMUM

```
// cálculo do máximo divisor comum para
// números naturais a, b tais que a ≠ 0
unsigned int mdc(unsigned int a,
                 unsigned int b) {
  if (b == 0) return a;
  return mdc(b, a % b);
}

[cling]$ mdc(13, 5)
(unsigned int) 1
[cling]$ mdc(21, 0)
(unsigned int) 21
[cling]$ mdc(0, 0)
(unsigned int) 0
```

← *o resultado deveria ser "indeterminado" pois 0 não divide 0. Mas não temos como representar isso...*

MÁXIMO DIVISOR COMUM

Para casa:

→ implemente a função mdc de maneira iterativa

→ como você alteraria na função mdc para que ela funcionasse para inteiros quaisquer (com sinal)