Recursão

Emílio Bergamim Júnior

Instituto de Geociências e Ciências Exatas - UNESP

2023

Conteúdo da aula

- Definição e exemplos de funções recursivas
- Passagem de parâmetros, empilhamento de parâmetros, endereço de retorno e informações relevantes para recursão
- Algoritmos recursivos versus iterativos

Considere a função fatorial

$$f(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1. \tag{1}$$

Note então que

$$f(n-1) = (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 2 \cdot 1 \tag{2}$$

o que permite escrever a função fatorial em termos de si mesma:

$$f(n) = n \cdot f(n-1) \quad ou \quad n! = n \cdot (n-1)! \tag{3}$$

Isso é um exemplo de *função recursiva*: uma função que chama a si mesma durante sua execução.

Função recursiva

Uma função é dita recursiva caso faça uma chamada a si mesma de forma direta ou indireta durante sua execução.

```c
// Recursão direta
void F()
{
    //Operações inicias
    F(); //Chamada recursiva
    //Operações finais
}

//Recursão indireta
float A()
{
    //Operações iniciais
    B(); //Chama uma segunda função
    //Operações finais
}

//Chamada indireta de A via B
int B()
{
    //Operações inicias
    A(); //Chama a função A
    //Operações finais
}

```

Figura: Exemplos de recursão direta e indireta.

Divisão e conquista

O paradigma de dividir para conquistar

Determinados problemas podem ser compreendidos como um conjunto do mesmo subproblema. Nesse caso, diz-se que o problema possui uma *estrutura recursiva*.

- **Divisão**
  - O problema é particionado em subproblemas que podem ser resolvidos usando o mesmo algoritmo.
  - A partição é feita até atingir o menor subproblema que pode ser resolvido (este é dito o *critério de parada*)
- **Conquista**
  - A solução dos subproblemas é combinada para solucionar o problema principal

Recursão

Na implementação recursiva de uma função, precisamos então de

- **Casos base**: são aqueles que fornecem o critério de parada e podem ser resolvidos trivialmente. Isto é, sem uma chamada recursiva. Estes são os menores subproblemas que podem ser resolvidos.
- **Passo recursivo:** invocação da própria função de forma a resolver um subproblema, o qual é necessário para a solução do problema principal.
  - Nessa etapa, o problema é particionado até atingir o caso base.
  - Algebricamente, corresponde ao *passo indutivo*.

O algoritmo recursivo do fatorial

- A etapa de divisão consiste em notar que o cálculo de $n!$ passa pelo cálculo de $(n-1)!$, que passa pelo cálculo de $(n-2)!$ e assim por diante.
- Como o fatorial é calculado somente para inteiros não negativos, os casos base podem ser entendidos como $1!$ ou $0! = 1$.
- Assim, para $n \geq 0$, podemos definir

$$n! = \begin{cases} n \cdot (n-1)!, & \text{se n>1} \\ 1 & \text{caso contrário.} \end{cases} \tag{4}$$

Um detalhe da implementação do fatorial

- Na linguagem C, o tipo *int* possui um armazenamento de 4 textitbytes. Lembrando que

$$1\text{byte} = 8\text{bits}. \tag{5}$$

- Assim, a faixa de valores que uma variável tipo *int* pode armazenar é de -2.147.483.648 até 2.147.483.647.
- Quando um dos limites é excedido, a implementação do tipo é **circular**, significando que extrapolar um dos limites faz a operação retornar ao outro limite.
- De forma a calcular o fatorial de números maiores, recomenda-se a utilização do tipo *unsigned long int*, cujos limites são de 0 até 18.446.744.073.709.551.615 (8 *bytes*)
- O tipo long int tem limites de -9.223.372.036.854.775.808 até 9.223.372.036.854.775.807. (8 *bytes*)

Recursão e a memória

- Na chamada da função,
  - Inicialização dos parâmetros formais a partir dos argumentos da função
  - Armazenamento do endereço da instrução a ser executada após término da execução da função atual
- Como o computador diferencia entre os argumentos da chamada inicial e aqueles das chamadas seguintes?

A pilha (*stack*) I

Pilha (*stack*)

Em computação, chama-se de pilha uma estrutura dotada de uma dinâmica da forma **primeiro a entrar, último a sair**. No inglês, chama-se de *First-in, Last out* (FILO).

- A cada chamada de função, cria-se um **registro de ativação** (RA) (*stack frame*), os quais são armazenados na memória.
- Em uma região da memória denominada pilha, os RAs são **empilhados**.
- No registro de ativação são armazenados
  - Argumentos de entrada
  - Variáveis locais
  - Endereço da instrução a ser executada após fim da execução
  - Valor de retorno
  - Vínculo dinâmico (endereço para o RA de quem fez a chamada)

A pilha (*stack*) II

- O tempo de vida dos RAs é apenas durante a chamada da função.
- O RA da função *main* fica ativo durante toda a execução do programa.
- O espaço para os RAs é alocado na entrada da função e desalocado na saída.
- O primeiro (aquele que está no topo da pilha) RA empilhado é aquele da função em execução. O último, por consequência, é o da função *main*.
- A pilha contém um ponteiro para o endereço do topo da pilha, que é atualizado a cada novo RA que é empilhado.
- Vamos examinar o que acontece na pilha com uma chamada da função *fatorial(5)*

A pilha dos RAs

Por simplicidade, considere um registro de ativação que armazena as variáveis de entrada e o valor de retorno. As dinâmicas de retorno serão expressas por setas.

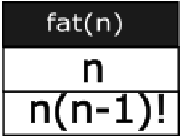

Figura: Forma de RA a ser utilizada na análise da pilha de RAs.

A pilha dos RAs

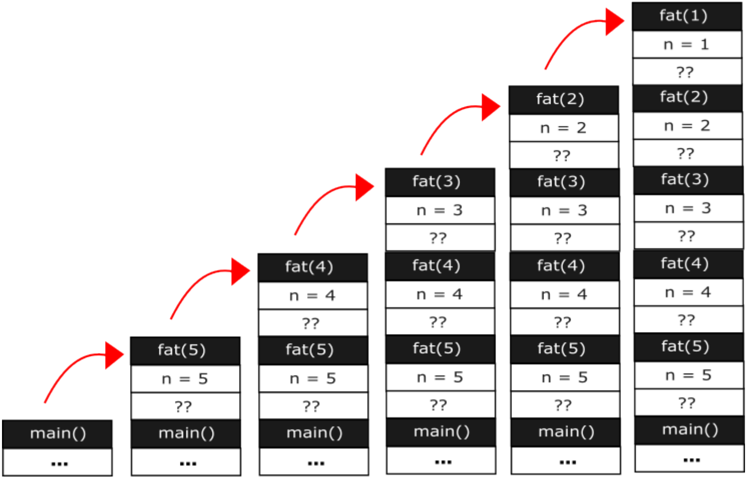

Figura: Exemplo dos RAs que são gerados durante a execução da função fatorial.

A pilha dos RAs

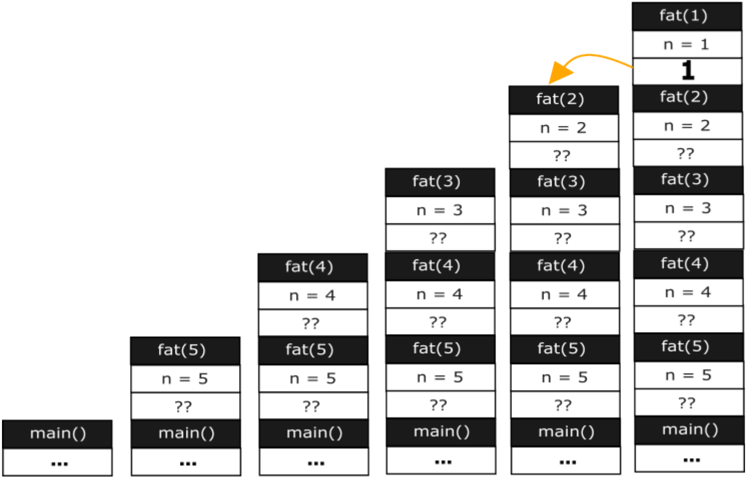

Figura: Uma vez que chega-se ao caso básico, os resultados dos subproblemas são agregados. A função no topo da pilha passa seu resultado para aquela imediatamente abaixo.

A pilha dos RAs

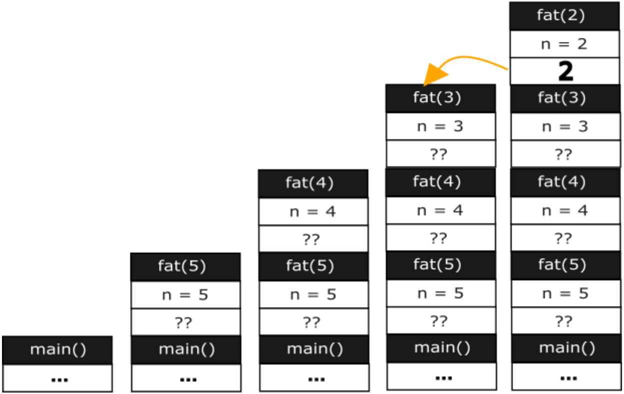

Figura: Uma vez que chega-se ao caso básico, os resultados dos subproblemas são agregados. A função no topo da pilha passa seu resultado para aquela imediatamente abaixo.

A pilha dos RAs

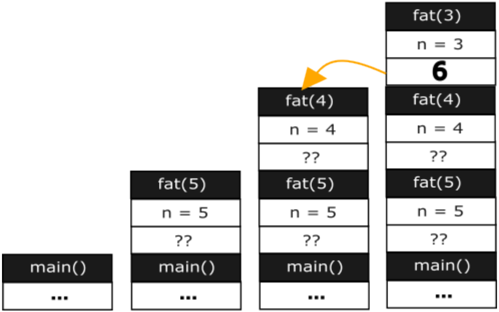

Figura: Uma vez que chega-se ao caso básico, os resultados dos subproblemas são agregados. A função no topo da pilha passa seu resultado para aquela imediatamente abaixo.

A pilha dos RAs

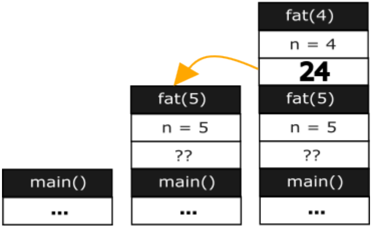

Figura: Uma vez que chega-se ao caso básico, os resultados dos subproblemas são agregados. A função no topo da pilha passa seu resultado para aquela imediatamente abaixo.

A pilha dos RAs

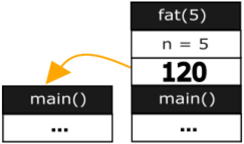

Figura: Uma vez que chega-se ao caso básico, os resultados dos subproblemas são agregados. A função no topo da pilha passa seu resultado para aquela imediatamente abaixo.

Figura: Ao final das chamadas recursivas da função fatorial, a função *main* segue sua execução.

Solução de uma equação não-linear

- O modelo Ising foi proposto no início do século passado como uma descrição de um certo tipo de magnetismo em sólidos. Posteriormente, o modelo foi generalizaado e encontrou aplicações em outras áreas, como modelagem de epidemias e aprendizado de máquina.
- Uma quantia relevante do modelo é sua magnetização que, em determinadas situações, é expressa por

$$m = \tanh(\beta m) \tag{6}$$

  onde $\beta \geq 0$ é um parâmetro do modelo relacionado à temperatura.
- Para resolver essa equação, pode-se utilizar um algoritmo iterativo

$$m_t = \tanh(\beta m_{t-1}) \tag{7}$$

Solução de uma equação não-linear

- Partindo de uma condição inicial m_0 e um dado valor de β, itera-se a expressão 7 até um limite de iterações t_{max}.
- Numa implementação recursiva, t_{max} denota a **profundidade de recursão**, ou seja, o número de chamadas recursivas que serão realizadas.
- Para um valor suficientemente grande de t_{max}, é possível ultrapassar o limite de memória da pilha.
- A utilização na prática desse algoritmo usualmente envolve um critério de parada em termos de **precisão numérica** e valores tão altos de t_{max} dificilmente são um problema. Costuma-se utilizar um critério como

$$\left| m_t - m_{t-1} \right| < \epsilon \tag{8}$$

sendo $\epsilon > 0$ a precisão desejada.

A sequência de Fibonacci

A sequência de Fibonacci é dada por

$$F(n) = \begin{cases} 0, & \text{se } n = 0, \\ 1, & \text{se } n = 1, \\ F(n-1) + F(n-2) & \text{em caso contrário.} \end{cases} \tag{9}$$

onde n é um inteiro não-negativo.

- Os casos $n = 0$ e $n = 1$ são triviais e correspondem aos casos base do problema.
- O passo recursivo consiste de duas chamadas de F. Note que esta etapa contém um certo excesso em termos de operações uma vez que, exceto nos casos triviais, $F(n-1)$ demandará uma chamada de $F(n-2)$ que já será feita por $F(n)$.

A sequência de Fibonacci

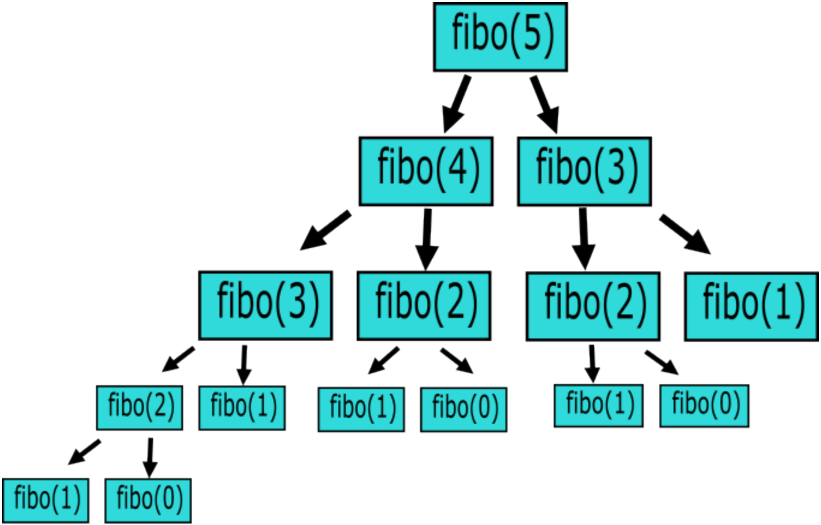

Figura: Árvore de recursão da função de Fibonacci.

A sequência de Fibonacci

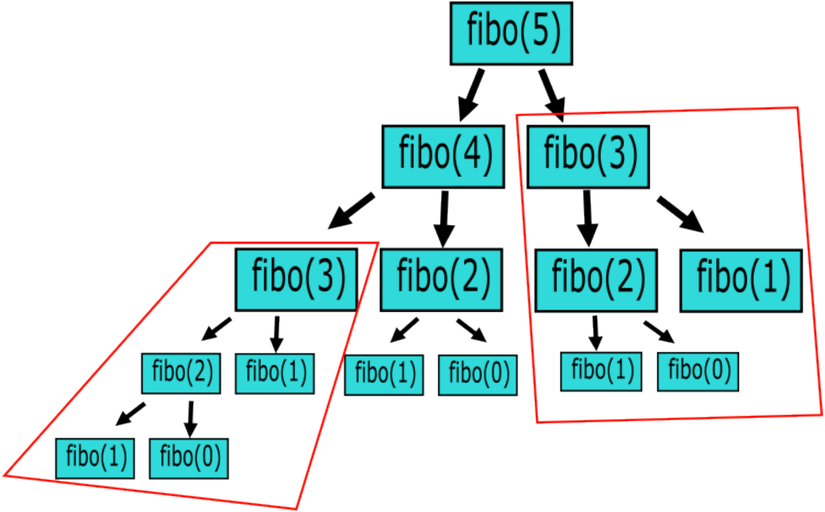

Figura: Árvore de recursão da função de Fibonacci. Em destaque duas regiões que executam exatamente os mesmos comandos.

A sequência de Fibonacci

- A performance da versão iterativa da função de Fibonacci é muito superior à da versão recursiva
- Isso ocorre porque, como visto na árvore de recursão, as chamadas sucessivas de *fibo* levam a execuções repetidas de um mesmo conjunto de operações em ramos distintos da árvore.

O problema das Torres de Hanoi consiste em transportar n discos de tamanhos distintos empilhados em ordem decrescente de tamanho em uma haste para uma segunda haste sem nunca colocar um disco maior sobre um disco menor. Para isso, conta-se com o auxílio de uma terceira haste.

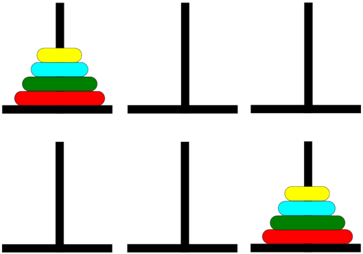

Figura: Exemplo do problema das Torres de Hanoi para $n = 4$.

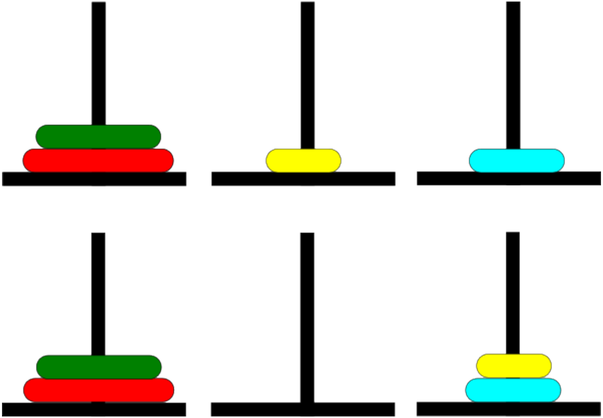

Figura: Os três primeiros movimentos para resolver o problema das torres de Hanoi.

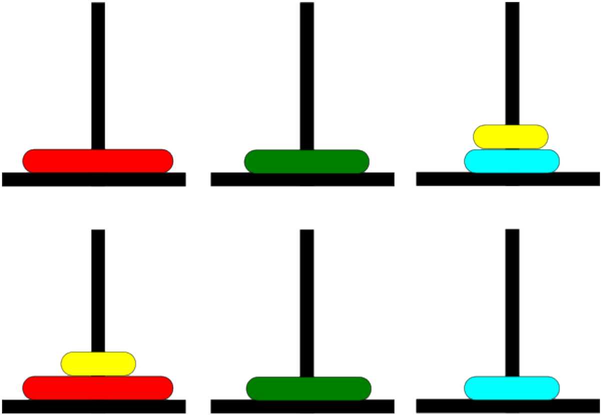

Figura: Quarto e quinto movimentos para resolver o problema das torres de Hanoi.

Torres de Hanoi

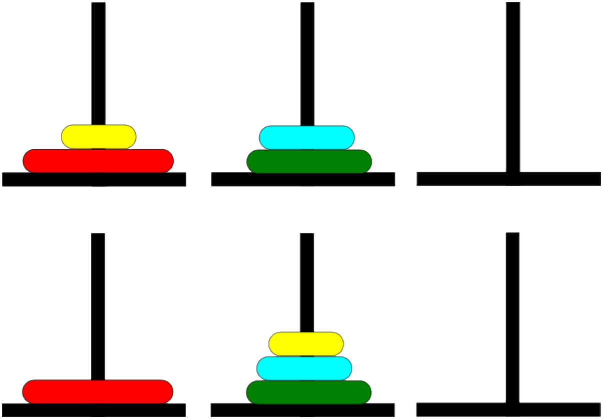

Figura: Sexto e sétimo movimentos para resolver o problema das torres de Hanoi.

Torres de Hanoi

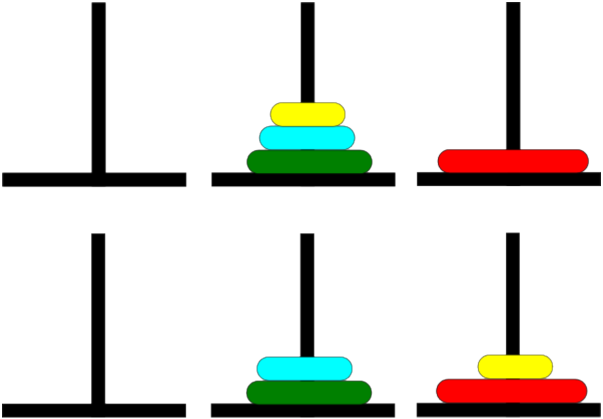

Figura: Oitavo e nono movimentos para resolver o problema das torres de Hanoi.

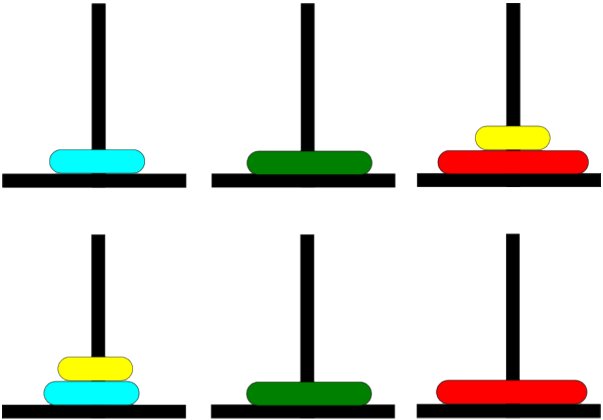

Figura: Décimo e décimo primeiro movimentos para resolver o problema das torres de Hanoi.

Torres de Hanoi

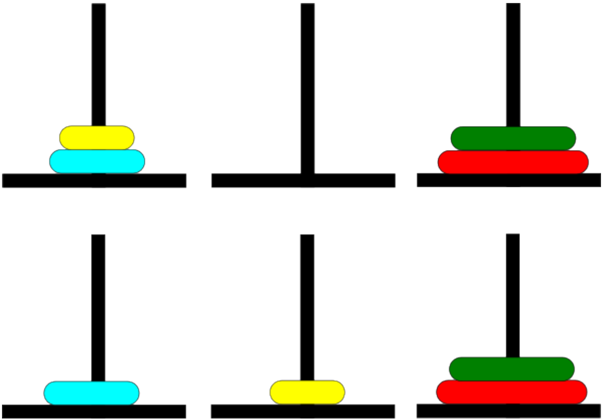

Figura: Décimo segundo e décimo terceiro movimentos para resolver o problema das torres de Hanoi.

Torres de Hanoi

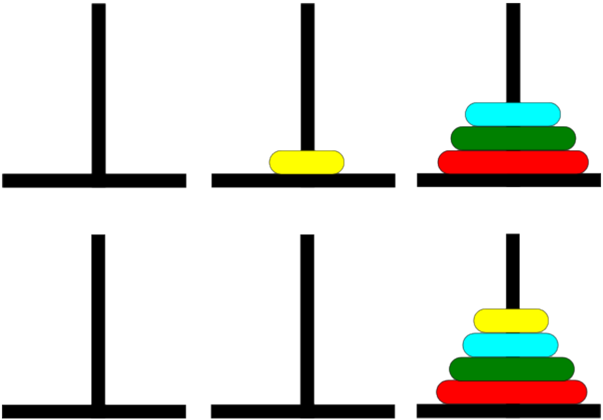

Figura: Décimo quarto e décimo quinto movimentos para resolver o problema das torres de Hanoi.

Torres de Hanoi - Resumo

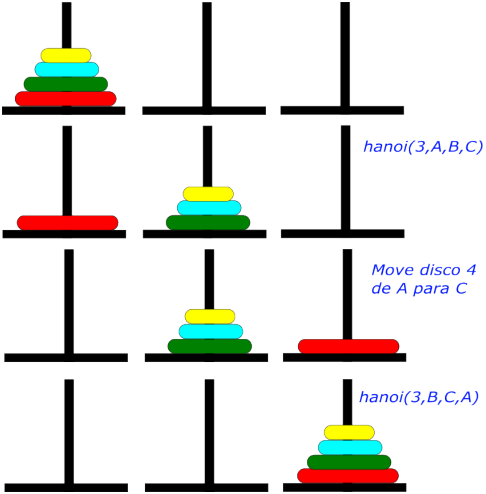

Figura: Síntese do algoritmo recursivo para solução do problema.

Algoritmos recursivos *versus* iterativos

Todo problema que possui uma solução recursiva possui também uma solução iterativa. Algumas **vantagens** de soluções recursivas são

- Elegância
- Código mais compacto
- Pode ser mais simples pensar em uma solução recursiva
- Alguns problemas são definidos de forma recursiva

enquanto as **desvantagens** são

- Maior consumo de memória
- Risco de extrapolar o limite de armazenamento da pilha
- Desempenho pode ser muito inferior ao da solução iterativa
- Dificuldade de depuração

Atividade em sala

Números primos

Um número inteiro positivo é dito **primo** se for divisível apenas pelo número 1 e por si mesmo. Em particular, basta checar se existe um divisor de n entre 2 e $\sqrt{n}$.

Implemente duas funções: uma que determina se um número é primo recursivamente e outra que faz o mesmo, mas de forma iterativa.

Para tirar 11: é possível testar uma quantidade ainda menor de números para determinar se um número é primo ou não notando que o único primo par é o número 2. Implemente-a.

Referências

- Deitel, Harvey; Deitel, Paul. *C: How to program*. Pearson Education, 2004.
- Prinz, Peter; Crawford, Tony. *C in a Nutshell*. O'Reilly Media, Inc., 2005.
- Feofillof, P. *Algoritmos em linguagem C*. Elsevier, 2009.

Agradecimentos especiais ao Prof. Dr. Denis Salvadeo, cujos *slides* serviram de inspiração para estes.