Estruturas de Dados I

Recursão

Prof. Bruno Azevedo

Instituto Federal de São Paulo

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Catanduva

O Princípio da Indução

- Suponha que desejamos provar que uma propriedade se mantém verdadeira para todos os números naturais 0, 1, 2, 3, ...
- Pode parecer difícil, ou impossível, em muitos casos, provar tal propriedade.
- Entretanto, esta percepção falha em capturar o fato que os números naturais ocorrem em uma sequência onde qualquer um dos números podem ser obtido a partir do número zero e somando 1, um número suficiente de vezes.
- O inteiro $n + 1$ é o sucessor do inteiro n. Deste modo, se começarmos com o inteiro 0 e construir seu sucessor, e assim em diante, eventualmente alcançaremos qualquer inteiro positivo.
- Esta ideia é capturada pelo **princípio da indução**.

O Princípio da Indução

- Indução é um método para provar que uma proposição P(n) é verdadeira para todo número natural n.
- Ou seja, que para os infinitos casos P(0), P(1), P(2), ..., P(n), a proposição se mantém verdadeira.
- Uma prova por indução consiste em dois casos.
  - O primeiro, o **caso base**, prova a afirmação para $n = 0$ sem assumir qualquer conhecimento de outros casos.
  - O segundo caso, o **passo de indução**, prova que se a afirmação vale para um caso dado $n = k$, então também deve valer para o próximo caso $n = k + 1$.
- Esses dois passos estabelecem que a afirmação vale para todos os números naturais n.
- O caso base não necessariamente começa com $n = 0$, mas frequentemente com $n = 1$, e possivelmente com qualquer número natural fixo $n = N$, estabelecendo a verdade da afirmação para todos os números naturais $n \geq N$.

Exemplo da Aplicação do Princípio da Indução

Teorema

A soma $S(n)$ dos primeiros n números naturais é $S(n) = \dfrac{n(n+1)}{2}$.

Prova

Base: Para n $=$ 0, temos que S(0) $=$ n(n $+$ 1)/2 $=$ 0(0 $+$ 1)/2 $=$ 0.
Ou seja, S(0) é verdadeiro.
Passo de indução: Precisamos mostrar que para um k qualquer. Se S(k) for verdadeiro, então S(k $+$ 1) também será verdadeiro.

Exemplo da Aplicação do Princípio da Indução

Prova (continuação)

Vamos assumir que k = n - 1. Por hipótese, $S(n-1) = \frac{n-1(n-1+1)}{2}$.
Por definição temos que: $S(n) = S(n-1) + n$. Substituindo:

$$\begin{aligned} S(n) &= \frac{n-1(n-1+1)}{2} + n. \\ &= \frac{n-1(n)}{2} + n. \\ &= \frac{n^2-n}{2} + n. \\ &= \frac{n^2+n}{2}. \\ &= \frac{n(n+1)}{2}. \end{aligned}$$

- Ou seja, assumimos ser verdade para k = n - 1, e demonstramos ser verdade para o caso genérico *n*, provando a validade da proposição para todos os números naturais.

Recursão

- A recursão é um conceito fundamental na computação, onde uma função chama a si mesma para resolver problemas de forma iterativa.
- Definições recursivas de funções operam de acordo com o princípio matemático da indução.
- Ou seja, a solução é inicialmente definida para o(s) caso(s) base e estendida para o caso geral.
- A recursividade geralmente permite uma descrição clara e concisa dos algoritmos, especialmente quando o problema é recursivo por natureza.
- Vamos ver um exemplo que possui essa característica.

Recursão

Problema

Problema: calcular o fatorial de um número natural positivo n.

- O fatorial de um número natural positivo n, denotado por $n!$, é o produto de todos os números naturais positivos menores que ou iguais a n.

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 3 \times 2 \times 1$$

- Por exemplo, $4! = 4 \times 3 \times 2 \times 1 = 24$.
- Alternativamente, o fatorial de n é igual ao produto de n pelo próximo menor fatorial, ou seja, $n! = n \times (n-1)!$.

Recursão

Problema

Problema: calcular o fatorial de um número natural positivo n.

- Considerando o princípio da indução, o caso base é: $1! = 1$.
- O passo de indução é: $n! = n \times (n-1)!$
- Portanto, a solução do problema é:
  - Se n $= 1$, $1! = 1$.
  - Se n > 1, então $n! = n \times (n-1)!$
- Notem a **recursão** ocorrendo quando calculamos $n!$.
- $n! = n \times (n-1)!$, e $(n-1)! = (n-1) \times (n-2)!$.
- Isso será feito **recursivamente** até chegarmos ao caso base.
- Ou seja, o caso base determina o **critério de parada** da recursão.

Recursão

Problema

Problema: calcular o fatorial de um número natural positivo n.

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

- Em nossa solução, fizemos uma chamada para a **própria** função. Por isso, ela é chamada de **função recursiva**.

Recursão

- Vamos entender em detalhe como essa função executa.
- Para isso, precisamos compreender como ocorre o gerenciamento da memória durante sua execução.
    - Toda vez que uma função é chamada, suas variáveis locais são empilhadas no topo da pilha.
    - Quando uma função termina a sua execução, suas variáveis locais são desempilhadas do topo da pilha.
- Vamos calcular o fatorial de 4 utilizando a nossa **função recursiva**.
- Mas antes, iremos adicionar uma variável auxiliar ao código para facilitar a visualização dos valores.

Recursão

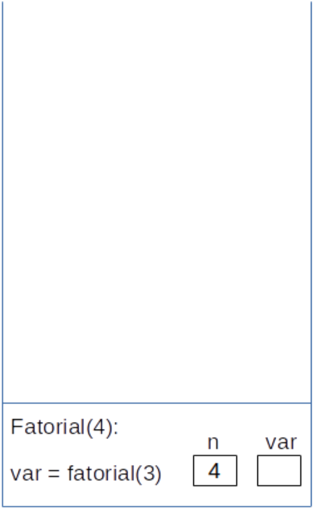
```c
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
```

Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```

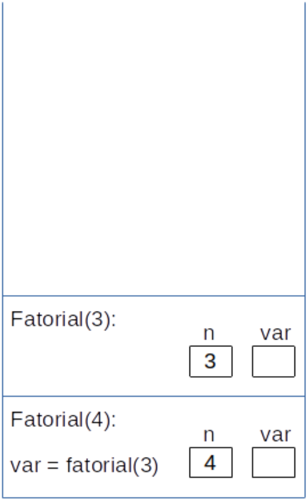
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



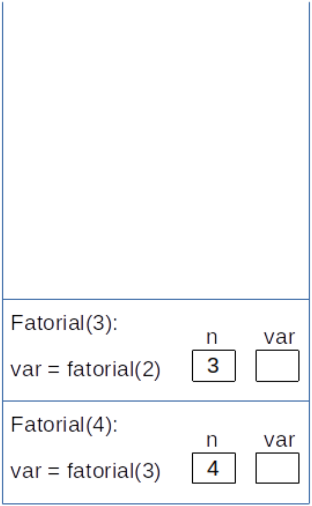
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



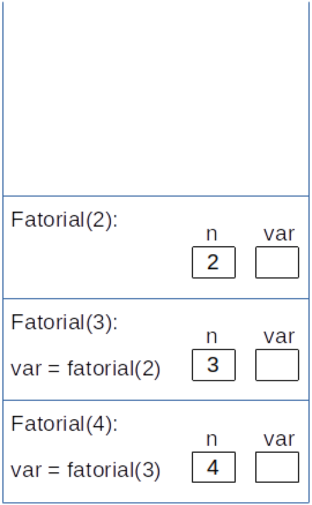
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



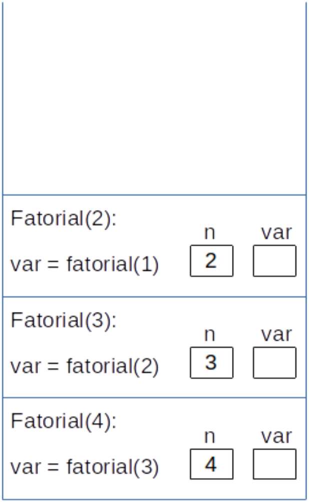
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



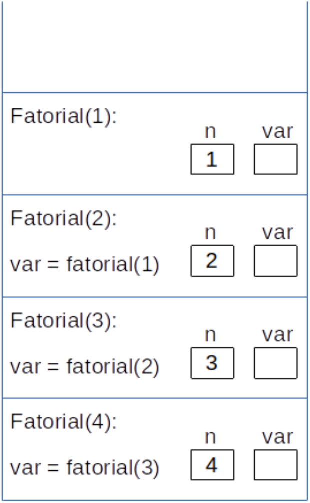
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



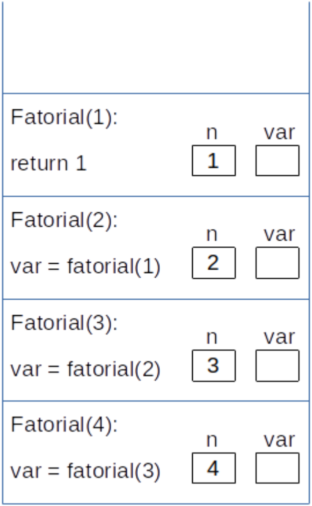
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



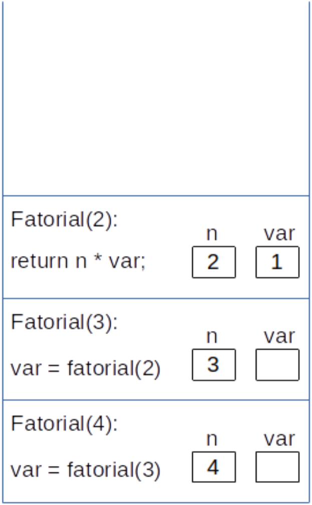
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



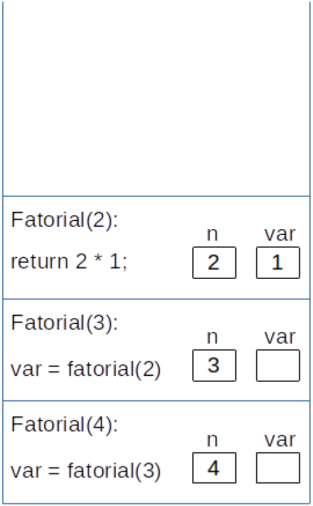
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



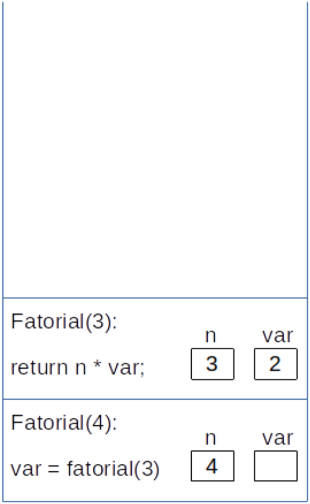
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



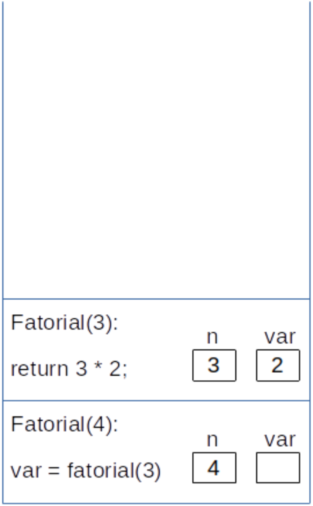
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



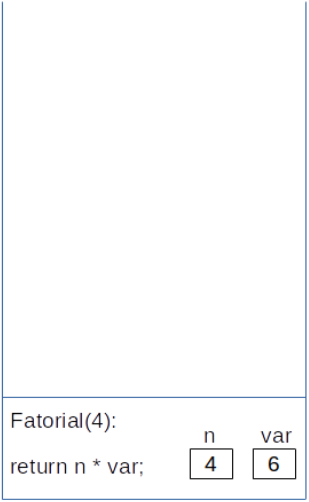
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



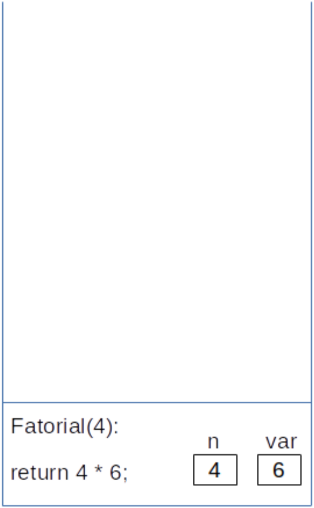
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



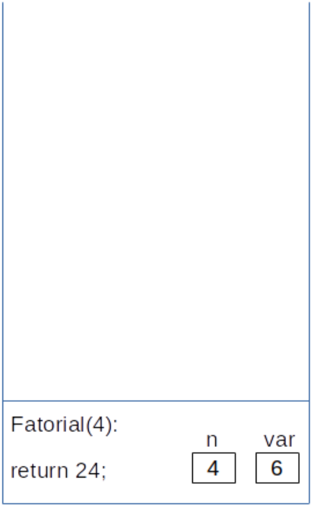
Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```



Recursão

```
int fatorial(int n) {
    if(n == 1)
        return 1;
    else {
        var = fatorial(n - 1);
        return n * var;
    }
}
int main() {
    cout << fatorial(4);
}
```

Recursão

- Notem que a variável var é desnecessária e foi utilizada apenas para facilitar a ilustração da execução do programa.
- O else também é desnecessário.
- E podemos tratar o caso de n $==$ 0, que é definido como 0! $=$ 1.

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

Solução Iterativa

- Também podemos implementar a função fatorial de modo iterativo.
- Uma possível implementação.

```
int fatorial(int n) {
    int resultado = 1;
    if((n == 0) || (n == 1))
        return 1;
    for(int i = 2; i <= n; ++i)
        resultado *= i;
    return resultado;
}
```

- Isso vale uma discussão.

Recursão $\times$ Iteração

- Em geral, soluções recursivas são mais concisas do que as iterativas.
- Podem ser mais intuitivas de serem implementadas, especialmente quando lidando com problemas que já possuem uma natureza recursiva, como o cálculo do fatorial.
- Entretanto, em geral, soluções iterativas consomem menos memória.
- Existe também um custo computacional com as cópias dos parâmetros (mas isso não é tão significativo).

Recursão × Iteração

- O que nos interessa de fato é a **eficiência** de nosso algoritmo.
- E isso é o que devemos avaliar quando criando soluções usando recursividade ou iteratividade.
- Lembrando, eficiência se relaciona com escalabilidade, ou seja, *como o tempo de processamento cresce quando a quantidade de dados do problema aumenta*.
- Vamos abordar outro problema e comparar as eficiências das soluções obtidas.

Sequência de Fibonacci

- A Sequência de Fibonacci é uma sequência em qual cada número é a soma dos dois números que o precedem.
- Números que são parte da sequência são chamados de números de Fibonacci.
- A sequência geralmente inicia com 0 e 1, mas alguns autores iniciam de 1 e 1, e Fibonacci iniciava em 1 e 2.
- Começando de 0 e 1, os 10 primeiros números são: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.
- Ou seja, $1 = 0 + 1$, $2 = 1 + 1$, $3 = 2 + 1$, $5 = 3 + 2$, $8 = 5 + 3$, $13 = 8 + 5$, $21 = 13 + 7$ e $34 = 21 + 13$

Sequência de Fibonacci

- Suponha que queremos calcular o n-ésimo número da sequência de Fibonacci.
- Denotaremos este número por Fibonacci(n).
- Temos os seguintes casos base:
  - Se n = 1 então Fibonacci(n) = 0
  - Se n = 2, então Fibonacci(n) = 1.
- Para o passo de indução, podemos computar Fibonacci(n) do seguinte modo:
  - Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
- Vamos implementar esta função de dois modos: recursivo e iterativo.

Sequência de Fibonacci

- Uma implementação recursiva da função para calcular o n-ésimo número da sequência de Fibonacci.

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

- Uma implementação iterativa da função para calcular o n-ésimo número da sequência de Fibonacci.

```
int fibonacci(int n) {
    int fib[n];
    fib[0] = 0;
    fib[1] = 1;
    for(int i = 2; i < n; i++)
        fib[i] = fib[i - 1] + fib[i - 2];
    return fib[n - 1];
}
```

Sequência de Fibonacci

- Uma implementação recursiva da função para calcular o n-ésimo número da sequência de Fibonacci.

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

- Uma implementação iterativa da função para calcular o n-ésimo número da sequência de Fibonacci.

```
int fibonacci(int n) {
    int fib[n];
    fib[0] = 0;
    fib[1] = 1;
    for(int i = 2; i < n; i++)
        fib[i] = fib[i - 1] + fib[i - 2];
    return fib[n - 1];
}
```

- São igualmente eficientes?

Avaliando a Eficiência das Implementações

```
int fibonacci(int n) {
    int fib[n];
    fib[0] = 0;
    fib[1] = 1;
    for(int i = 2; i < n; i++)
        fib[i] = fib[i - 1] + fib[i - 2];
    return fib[n - 1];
}
```

- Para n $>$ 2, quantas somas são necessárias para calcular Fibonacci(n) usando a estratégia iterativa?

Avaliando a Eficiência das Implementações

```
int fibonacci(int n) {
    int fib[n];
    fib[0] = 0;
    fib[1] = 1;
    for(int i = 2; i < n; i++)
        fib[i] = fib[i - 1] + fib[i - 2];
    return fib[n - 1];
}
```

- Para n $>$ 2, quantas somas são necessárias para calcular Fibonacci(n) usando a estratégia iterativa?
- Serão feitas (n-2) somas, afinal o nosso laço for soma até n - 1 vezes, iniciando em 2.

Avaliando a Eficiência das Implementações

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

- Para n $>$ 2, quantas somas são necessárias para calcular Fibonacci(n) usando a estratégia recursiva?

Avaliando a Eficiência das Implementações

- Antes de fazer uma análise, vamos simplesmente modificar o código e deixar ele nos dar o resultado.
- Criaremos uma variável soma para acompanharmos quantas adições serão feitas durante a execução do código.

```
#include <iostream>
using namespace std;
int soma = 0;
int fibonacci(int n) {
    if(n == 1)
        return 0;
    if(n == 2)
        return 1;
    soma++;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
int main() {
    int fib = fibonacci(20);
    cout << soma << endl;
    return 0;
}
```

- Na abordagem iterativa, sabemos que se buscamos o n-ésimo número, sempre faremos n - 2 somas.
- Ou seja, para calcular o vigésimo número iterativamente, o algoritmo iterativo fará 18 operações de adição.

Avaliando a Eficiência das Implementações

```
#include <iostream>
using namespace std;
int soma = 0;
int fibonacci(int n) {
    if(n == 1)
        return 0;
    if(n == 2)
        return 1;
    soma++;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
int main() {
    int fib = fibonacci(20);
    cout << soma << endl;
    return 0;
}
```

- Mas e no algoritmo recursivo, quantas somas serão feitas com n = 20?

Avaliando a Eficiência das Implementações

```
#include <iostream>
using namespace std;
int soma = 0;
int fibonacci(int n) {
    if(n == 1)
        return 0;
    if(n == 2)
        return 1;
    soma++;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
int main() {
    int fib = fibonacci(20);
    cout << soma << endl;
    return 0;
}
```

- Se executarem o código acima, descobrirão que ele fez 6764 somas (!!).
- Se quiserem calcular o quinquagésimo número (n = 50), terão que esperar um pouco para ter o resultado... afinal, serão 12.586.269.024 operações de adição.
- O que está acontecendo aqui é que a função está recalculando os mesmos valores várias vezes.

Avaliando a Eficiência das Implementações

- Por exemplo, para calcular fibonacci(20), calculamos fibonacci(19) e fibonacci(18). Mas para calcular fibonacci(19) também calculamos fibonacci(18).
- São geradas ramificações com cálculos idênticos.
- Esse algoritmo não é eficiente. Sua complexidade é exponencial (!!);
- Isso ocorre porque a função faz duas chamadas recursivas para fibonacci(n - 1) e fibonacci(n - 2) em cada chamada, exceto nos casos bases.
- Essas chamadas recursivas se ramificam exponencialmente à medida que n aumenta.
- Portanto, a complexidade de tempo para este código é exponencial em relação a n, tornando-o ineficiente para valores grandes de n.

Avaliando a Eficiência das Implementações

- Conclusão, usem a recursividade com o cuidado necessário.
- Devemos avaliar a relação entre clareza de código, intuitividade de implementação, versus a eficiência do algoritmo.
- Mas isso não se aplica apenas a recursividade, também podemos ter abordagens com complexidades exponenciais ou piores em soluções iterativas.

Exercícios

- Escreva uma função recursiva que efetue a operação de potenciação. Por exemplo, pot(7, 2) deve retornar 7^2. Dica: qual o caso base? Qual o passo de indução em uma operação de potenciação?
- Escreva uma função recursiva que efetue a soma de dois números inteiros não negativos usando apenas incrementos e decrementos unitários.
- Escreva uma função recursiva que efetue a multiplicação de dois números inteiros positivos usando apenas somas e subtrações.
- Escreva uma função recursiva que efetue a soma de todos os inteiros positivos pares menores ou iguais a um valor inteiro n.
- Escreva uma função recursiva que efetue a soma dos dígitos de um número inteiro não negativo. Dica: usem a operação de módulo (%) para obter cada dígito.
- Escreva uma função recursiva que, dado um número inteiro positivo n, retorne a representação binária de n.