Linguagens de Programação

CENTRO UNIVERSITÁRIO NEWTON PAIVA

Prof. Dr. João Paulo Aramuni

Newton

Quem se prepara, não para.

Newton
Quem se prepara, não para.

Linguagens de Programação

3º período

Prof. Dr. João Paulo Aramuni

Sumário

- Java
  - Recursão

Sumário

- Java
  - **Recursão**

Java

Newton
Quem se prepara, não para.

- Recursão
  - Em Java, um método pode chamar a si mesmo. Esse processo se chama recursão e dizemos que um método é recursivo quando chama a si próprio.
  - Em geral, recursão é o processo em que algo é definido a partir de si mesmo e é um pouco parecido com uma definição circular. O componente-chave do método recursivo é a instrução que executa uma chamada a ele próprio.
  - A recursão é um mecanismo de controle poderoso.

Fonte: SCHILDT, 2015.

- Recursão
  - Vantagens da programação recursiva:
    - **Legibilidade**: Em certos casos, o código recursivo pode ser mais fácil de entender e ler, especialmente quando a lógica do problema é naturalmente recursiva.
    - **Abstração**: A recursão pode ajudar a simplificar a solução de problemas complexos, permitindo que você se concentre na definição do problema e divida-o em problemas menores.
    - **Redução de código**: Em muitos casos, a solução recursiva pode levar a menos linhas de código em comparação com soluções iterativas.

- Recursão
  - Vantagens da programação recursiva:
    - **Solução elegante**: Em algumas situações, a recursão pode levar a uma solução mais elegante, que é mais fácil de implementar e manter.

    - **Algoritmos dividir e conquistar**: A recursão é frequentemente usada para implementar algoritmos de "dividir e conquistar", que são eficientes para resolver problemas como ordenação, busca binária e outras tarefas complexas.

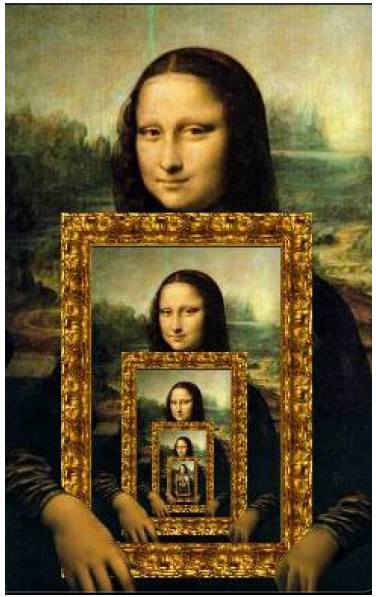
- Recursão
  - Desvantagens da programação recursiva:
    - **Consumo de memória**: Em algumas situações, a recursão pode levar a um alto consumo de memória, especialmente quando há muitas chamadas recursivas aninhadas. Isso pode resultar em estouro da pilha de chamadas (stack overflow).

    - **Desempenho**: A abordagem recursiva pode ser menos eficiente em termos de desempenho em comparação com uma solução iterativa, devido ao alto número de chamadas de função.

- Recursão
  - Desvantagens da programação recursiva:
    - **Dificuldade de depuração**: O processo de depuração pode ser mais complicado com funções recursivas, pois as chamadas podem ser aninhadas e mais difíceis de rastrear.
    - **Limite de recursão**: Alguns ambientes de programação têm limites para a profundidade máxima de recursão, o que pode limitar a capacidade de resolver problemas muito grandes.
    - **Complexidade da implementação**: Em certos casos, a recursão pode exigir uma lógica mais complexa para garantir que o algoritmo termine corretamente.

Java

- Recursão
  - Exemplos em C



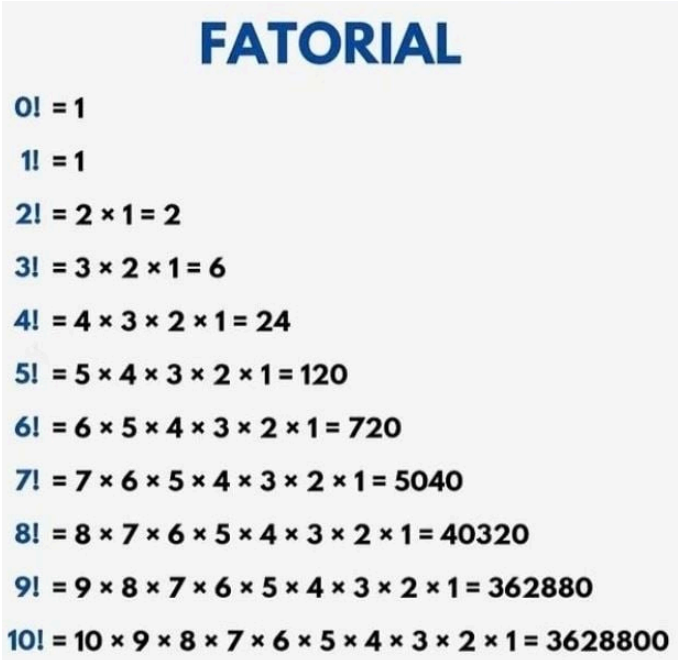
c / Aramuni.c ↑ Top

Code | Blame | 204 lines (175 loc) · 4.71 KB | Raw

```c
int palindrome(char str[100], int tam) {
return (tam <= 1) || (*str == str[tam - 1] && palindrome(str + 1, tam - 2));
}

int strLen(char *orig, int cont) {
return (*orig++) ? cont++, strLen(orig, cont) : cont;
}

void strCpy(char *str1, char *str2) {
return (*str1) ? *str2 = *str1, str1++, str2++, strCpy(str1, str2) : 0;
}

void strCat(char *str1, char *str2, int p) {
return (*str2) ? str1[p] = *str2, p++, str2++, strCat(str1, str2, p) : 0;
}

void strInverte(char *str1, char *str2, int p, int i, int j) {
return (p != 0) ? (str1++, p--, strInverte(str1, str2, p, i, j)): (i++, (j++ != i) ? (i--, *str2 = *str1, str1--, str2++, strInverte(str1, str2, p, i, j)): 0);
}

void strLimpa(char *str1) {
return (*str1) ? *str1 = ' ', str1++, strLimpa(str1) : 0;
}

int contLetra(char palavra[100], int cont, char letra, int i) {
return (palavra[i] == letra) ? contLetra(palavra, ++cont, letra, ++i): (palavra[i] ? contLetra(palavra, cont, letra, ++i) : cont);
}

int fatorial(int c) {
return (c == 0 || c == 1) ? 1 : (c * fatorial(c - 1));
}

int fibonacci(int n) {
return (n == 0) ? 0: ((n == 1) ? 1 : (fibonacci(n - 1) + fibonacci(n - 2)));
}

int mdc(int a, int b) {
return (a == 0) ? b : ((b == 0) ? a : mdc(b, a % b));
}

int powExpo(int base, int expo) {
return (expo == 0) ? 1 : (expo == 1 ? base : (base* powExpo(base, expo - 1)));
}
```

Java

- Recursão
  - Exemplo em Java



Fonte: SCHILDT, 2015.

ExemploRecursao.java

```java
package meuapp;

import java.util.Scanner;

public class ExemploRecursao {

    // Um exemplo simples de recursão.
    // Esta é uma função recursiva.
    private static int factR(int n) {
        int result;
        if (n == 1) {
            return 1;
        }
        // Executa a chamada recursiva a factR( ).
        result = factR(n - 1) * n;
        return result;
    }

    public static void main(String[] args) {
        Scanner ler = new Scanner(System.in);
        System.out.println("Digite um numero para calcular o fatorial: ");
        int num = ler.nextInt();
        ler.close();
        System.out.println(factR(num));
    }
}
```

Console | Problems | Javadoc | Declaration | Search | Breakpoints

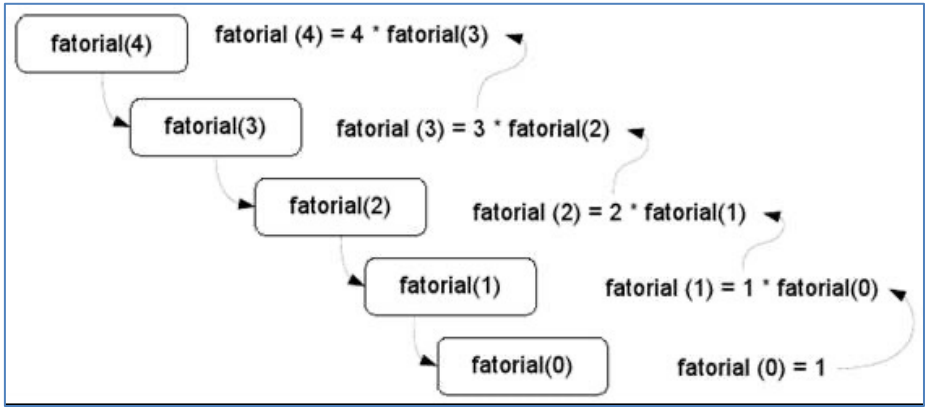
```
<terminated> ExemploRecursao [Java Application] /Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Hom
Digite um numero para calcular o fatorial:
5
120
```

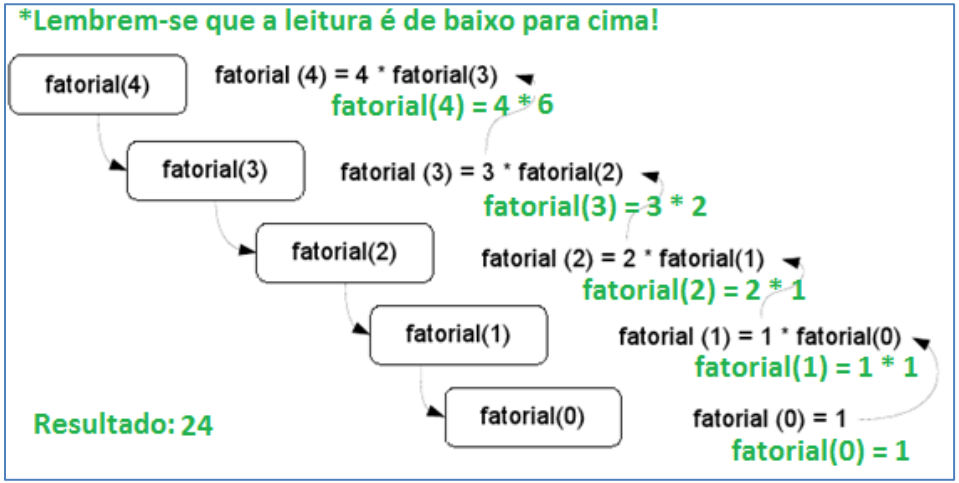
- Recursão
  - A operação do método recursivo factR() é mais complexa em relação à um método comum.
  - Quando factR() é chamado com um argumento igual a 1, ele retorna 1; caso contrário, retorna o produto de facR(n – 1)*n.
  - Para avaliar essa expressão, factR() é chamado com n-1. Esse processo se repete até n ser igual a 1 (caso base) e as chamadas ao método começarem a retornar.

Fonte: SCHILDT, 2015.

Recursão

Newton
Quem se prepara, não para.





- Recursão
  - Quando um método chama a si próprio, é alocado espaço de armazenamento na pilha para novas variáveis e parâmetros locais e o código do método é executado com essas novas variáveis desde o início.

  - Uma chamada recursiva **não faz uma nova cópia do método**. Só os argumentos são novos. À medida que cada chamada recursiva retorna, as variáveis e parâmetros locais antigos são removidos da pilha e a execução é retomada no ponto de chamada dentro do método.

  - Poderíamos dizer que os métodos recursivos "**empilham-se**" para frente e para trás.

Fonte: SCHILDT, 2015.

Java

- Recursão
  - Outros exemplos em Java

```
public static void main(String[] args) {
    Scanner ler = new Scanner(System.in);
    System.out.println("Digite um numero para calcular o fatorial: ");
    int num = ler.nextInt();
    System.out.println(factR(num));

    System.out.println("Digite dois numeros para o MDC: ");
    int num1 = ler.nextInt();
    int num2 = ler.nextInt();
    System.out.println(mdc(num1, num2));

    System.out.println("Digite um numero para contar os pares: ");
    int num3 = ler.nextInt();
    System.out.println(contaPares(num3));

    System.out.println("Digite um numero para calcular a potência: ");
    int num4 = ler.nextInt();
    System.out.println("Digite a potência: ");
    int num5 = ler.nextInt();
    System.out.println(potencia(num4, num5));

    System.out.println("Digite um numero para calcular a Fibonacci: ");
    int num6 = ler.nextInt();
    System.out.println(fibo(num6));

    ler.close();

}
```

Fonte: SCHILDT, 2015.

Java

- Recursão
  - Outros exemplos em Java

```java
private static int factR(int n) {
    int result;
    if (n == 1) {
        return 1;
    }
    // Executa a chamada recursiva a factR( ).
    result = factR(n - 1) * n;
    return result;
}

private static int mdc(int a, int b) {
    if (b == 0) {
        return a;
    }
    return mdc(b, (a % b));
}

private static int contaPares(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    if (n % 2 == 0) {
        return contaPares(n - 1) + 1;
    } else {
        return contaPares(n - 1);
    }
}

public static double potencia(int n, int pot) {
    if (pot == 1) {
        return n;
    }
    return n * potencia(n, pot - 1);
}

public static int fibo(int n) {
    if (n < 2) {
        return n;
    }
    return fibo(n - 1) + fibo(n - 2);

}
```

Fonte: SCHILDT, 2015.

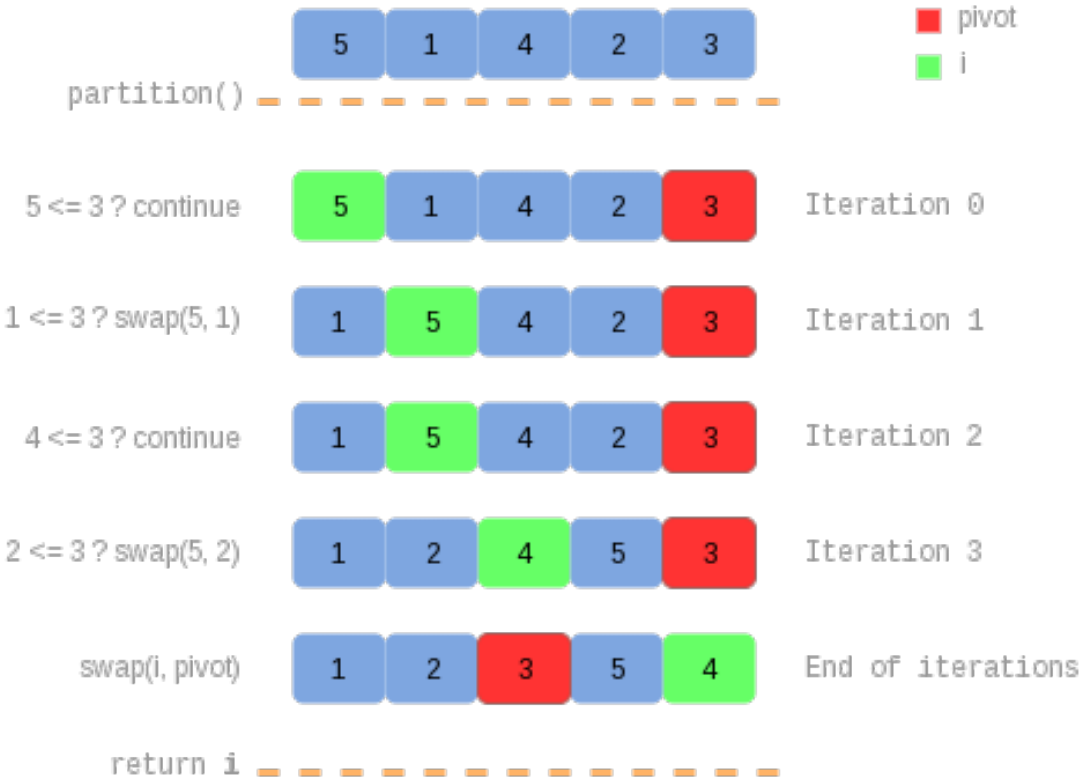
- Recursão
  - Exemplo QuickSort
    - O algoritmo **Quicksort** utiliza o paradigma de programação *Dividir para Conquistar*. Esse paradigma é uma abordagem **recursiva** em que a entrada do algoritmo é ramificada múltiplas vezes a fim de quebrar o problema maior em problema menores da mesma natureza.

    - No caso do Quicksort o método ***partition()*** é responsável por implementar o paradigma. Vamos entender mais sobre o seu funcionamento e papel dentro do algoritmo de ordenação Quicksort.

Fonte: SCHILDT, 2015.

Java

- Exemplo QuickSort



Fonte: SCHILDT, 2015.

Java

Newton
Quem se prepara, não para.

- Exemplo QuickSort

QuickSort	
Elementos	Caso médio (nlogn)
100	0 ms
1.000	0 ms
10.000	39 ms
100.000	43 ms
200.000	50 ms

https://www.youtube.com/watch?v=WP7KDljG6IM
https://www.youtube.com/watch?v=tIYMCYooo3c

Java

Newton
Quem se prepara, não para.

- Exemplo QuickSort

```java
public class QuickSort {

    public static void main(String[] args) {
        int quantidade = 100;
        int[] vetor = new int[quantidade];

        for (int i = 0; i < vetor.length; i++) {
            vetor[i] = (int) (Math.random() * quantidade);
            System.out.println(vetor[i]);
        }

        long tempoInicial = System.currentTimeMillis();

        quickSort(vetor, 0, vetor.length - 1);

        long tempoFinal = System.currentTimeMillis();

        System.out.println("Executado em = " + (tempoFinal - tempoInicial) + " ms");
        System.out.println("\nVetor Ordenado:");
        for (int i = 0; i < vetor.length; i++) {
            System.out.println(vetor[i]);
        }

    }
```

Java

Newton
Quem se prepara, não para.

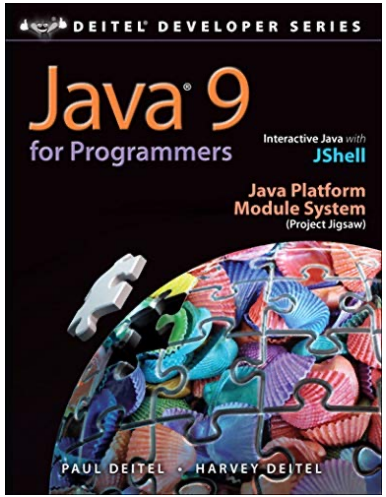
- Exemplo QuickSort

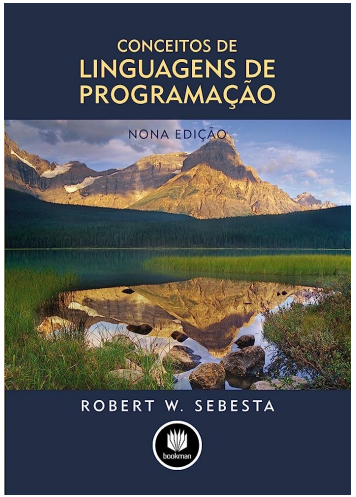
Recursão

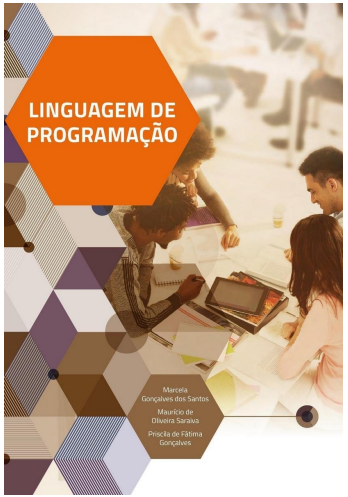
```
28    private static void quickSort(int[] vetor, int inicio, int fim) {
29        if (inicio < fim) {
30            int posicaoPivo = separar(vetor, inicio, fim);
31            quickSort(vetor, inicio, posicaoPivo - 1);
32            quickSort(vetor, posicaoPivo + 1, fim);
33        }
34    }
35
36    private static int separar(int[] vetor, int inicio, int fim) {
37        int pivo = vetor[inicio];
38        int i = inicio + 1, f = fim;
39        while (i <= f) {
40            if (vetor[i] <= pivo)
41                i++;
42            else if (pivo < vetor[f])
43                f--;
44            else {
45                int troca = vetor[i];
46                vetor[i] = vetor[f];
47                vetor[f] = troca;
48                i++;
49                f--;
50            }
51        }
52        vetor[inicio] = vetor[f];
53        vetor[f] = pivo;
54        return f;
55    }
56 }
```

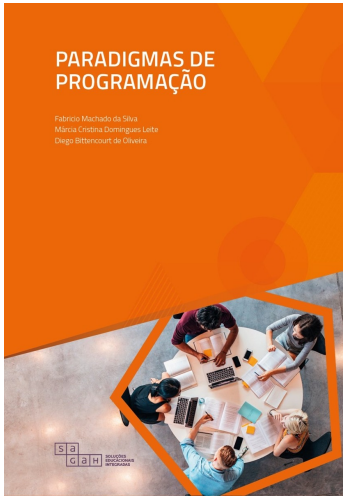
Referências

Newton
Quem se prepara, não para.









Obrigado!

joao.aramuni@newtonpaiva.br