



DESCRIÇÃO

Conceitos introdutórios às definições recursivas, tais como sequências, funções, conjuntos; Implementação da recursividade; Recursividade aninhada e Recursão indireta; Comparação entre a iteração x recursividade e Exemplos de funções recursivas.

PROPÓSITO

Apresentar os conceitos básicos para o entendimento das definições recursivas, diferenciar algoritmos interativos dos recursivos e identificar os problemas que podem ocorrer ao se empregar algoritmos recursivos.

PREPARAÇÃO

Antes de iniciar o conteúdo, tenha à mão um livro de Matemática do Ensino Médio que apresente os conceitos de funções matemáticas, como fatorial, somatórios, teoria de conjuntos, sequências e séries. Preferencialmente, também tenha um computador com um ambiente para compilar e executar programas escritos na linguagem de programação C. . Para que você possa acompanhar melhor os exemplos, baixe os arquivos com os exemplos apresentados clicando [aqui](#).

OBJETIVOS

MÓDULO 1

Definir recursividade

MÓDULO 2

MÓDULO 3

Identificar a utilização de funções recursivas através de exemplos

INTRODUÇÃO

A recursividade é uma ferramenta poderosa para resolução de diversos problemas na computação. Alguns, que seriam solucionados com algoritmos complexos e difíceis de entender, conseguem ser resolvidos de forma elegante e bem mais simples.

A partir do momento que se entende a definição recursiva de um algoritmo, a sua implementação se torna bastante simples. Alguns pontos, porém, devem ser considerados para a implementação de algoritmos recursivos, por exemplo, a grande utilização da pilha da memória do computador, levando a estouros da capacidade de memória. É fundamental, portanto, testar os limites dos valores a serem inseridos nesses algoritmos e entender em que tipos de aplicações devemos utilizar funções recursivas ou programas iterativos.

Diversos problemas complexos da computação são resolvidos utilizando programação dinâmica e/ou paradigmas de programação funcional. Essas técnicas e paradigmas são todas baseadas em programação recursiva, o que torna obrigatório o conhecimento adequado da recursividade.

MÓDULO 1

① Definir recursividade

DEFININDO RECURSIVIDADE

Em algumas situações da Matemática, é bastante complexo definir o valor de um objeto explicitamente, sendo mais fácil defini-lo **recursivamente**, de forma que a solução inicial faça parte para descobrir a próxima solução de um item, e assim por diante.

Para uma formalização de definições recursivas, podemos considerar o seguinte:

Na definição recursiva, também chamada de **indutiva**, o item que está sendo definido aparece como parte da definição. Esse procedimento funciona porque as definições recursivas são formadas por duas partes:

1. UMA BASE, QUE DETERMINA EXPLICITAMENTE ALGUNS CASOS SIMPLES DO ITEM QUE ESTÁ SENDO DEFINIDO.



2. UM PASSO INDUTIVO OU RECURSIVO, NO QUAL OUTROS CASOS DO ITEM QUE ESTÁ SENDO DEFINIDO SÃO DETERMINADOS EM TERMOS DOS CASOS

ANTERIORES.

Segundo Oliveira (2020), a recursão pode ser usada para definir sequência de objetos, funções de objetos e conjuntos.

SEQUÊNCIAS

De acordo com Vidal (2020), uma sequência é uma lista de objetos, enumerados, segundo uma ordem. O $S(K)$ denota o késimo elemento da sequência. Uma sequência recursiva explicita seu primeiro valor (ou primeiros valores) e define outros valores na sequência em termos dos valores iniciais.

Utilizando definição:

Passo 1 - o elemento base indica que $S(1) = 2$

Passo 2 - o passo recursivo 2 é apresentado com $S(2) = S(1) * 2$

Passo 3 - o passo recursivo 3 é apresentado como $S(3) = S(2) * 2$

A seguir, alguns exemplos de sequências recursivas:

SEQUÊNCIA DE POTÊNCIAS DE 2:

$$a^n = 2^n, \text{ para } n = 0; 1; 2; 3; 4$$

No entanto, uma sequência também pode ser definida de forma recursiva.

A partir do 1º termo e de uma regra para encontrar um termo da sequência a partir do anterior, ou seja:

$$a^0 = 1$$

$a^{n+1} = 2 \cdot a_n, \text{ para } n = 0; 1; 2; 3$ gerando a seguinte sequência:

$$S(0) = 1$$

$$S(1) = 2 \cdot S(0) = 2$$

$$S(2) = 2 \cdot S(1) = 4$$

$$S(3) = 2 \cdot S(2) = 8$$

SEQUÊNCIA ARITMÉTICA T:

$$T(1) = 1$$

$$T(n) = T(n - 1) + 3 \text{ para } n \geq 2$$

Esta sequência é recursiva, ou seja, a sequência dos termos é obtida a partir do 1º termo.

$$T(1) = 1$$

$$T(2) = T(1) + 3 = 1 + 3 = 4$$

$$T(3) = T(2) + 3 = 4 + 3 = 7$$

$$T(4) = T(3) + 3 = 7 + 3 = 10$$

$$T(5) = T(4) + 3 = 10 + 3 = 13$$

📣 ATENÇÃO

É importante ressaltar que quando definimos uma sequência recursivamente, podemos usar indução para provar resultados sobre ela. Nos exemplos anteriores, foi utilizada, como recurso, a indução para se obter o próximo valor da sequência recursiva.

FUNÇÕES

A definição recursiva de uma função cujo domínio é o conjunto dos inteiros não negativos consiste em duas etapas semelhantes a todas as definições recursivas:

PASSO BÁSICO

PASSO RECURSIVO

PASSO BÁSICO:

Especificar o valor da função em zero.

PASSO RECURSIVO:

Fornecer uma regra para encontrar o valor da função em um inteiro a partir dos seus valores em inteiros menores.

Esta definição, segundo Freitas (2020), é chamada de recursiva ou ainda de indutiva.

Exemplo 1:

Suponha que f é definida recursivamente por:

$$f(0) = 3$$

$$f(n + 1) = 2 \cdot f(n) + 3$$

Encontre $f(1)$, $f(2)$, $f(3)$ e $f(4)$.

Solução:

$$f(1) = 2f(0) + 3 = 2 \cdot 3 + 3 = 9$$

$$f(2) = 2f(1) + 3 = 2 \cdot 9 + 3 = 21$$

$$f(3) = 2f(2) + 3 = 2 \cdot 21 + 3 = 45$$

$$f(4) = 2f(3) + 3 = 2 \cdot 45 + 3 = 93$$

Exemplo 2:

Encontre $f(1)$, $f(2)$ e $f(3)$ se $f(n)$ é definida recursivamente por: $f(0) = 1$; para $n = 0, 1, 2, 3, 4$ para a função: $f(n + 1) = 2^{f(n)}$

Solução:

$$f(0) = 1$$

$$f(1) = 2^{f(0)} = 2^1 = 2$$

$$f(2) = 2^{f(1)} = 2^4 = 16$$

$$f(3) = 2^{f(2)} = 2^{16} = 65536$$

$$f(4) = 2^{f(3)} = 2^{45} = 35184372088832$$

Exemplo 3:

Encontre $f(1)$, $f(1)$ e $f(3)$ se $f(n)$ é definida recursivamente por: $f(0) = 1$; para $n = 0, 1, 2, 3, 4$ para a função:

$$f(n + 1) = f(n)^2 + f(n) + 1$$

Solução:

$$f(0) = 1$$

$$f(1) = f(0)^2 + f(0) + 1 = 1^2 + 1 + 1 = 3$$

$$f(2) = f(1)^2 + f(1) + 1 = 3^2 + 3 + 1 = 13$$

$$f(3) = f(2)^2 + f(2) + 1 = 13^2 + 13 + 1 = 183$$

$$f(4) = f(3)^2 + f(3) + 1 = 183^2 + 183 + 1 = 33673$$

EXEMPLOS DE FUNÇÕES COMO DEFINIÇÕES RECURSIVAS

A função factorial é um exemplo clássico da definição recursiva, pela própria definição do factorial de um número **N** ser o factorial do número (n)

multiplicado pelo factorial do número menos

$$1 (n - 1)$$

, e assim, sucessivamente.

Pela definição matemática:

O factorial do inteiro 5 é calculado da seguinte forma:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120.$$

Utilizando a definição recursiva para funções, temos o seguinte resultado:

Solução:

Definição recursiva:

$$F(0) = 1$$

$$F(n + 1) = (n + 1)F(n)$$

Avaliando para o caso

$$F(5) = 5! :$$

$$F(5) = 5 \cdot F(4)$$

$$F(4) = 5 \cdot 4 \cdot F(3)$$

$$F(3) = 5 \cdot 4 \cdot 3 \cdot F(2)$$

$$F(2) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot F(1)$$

$$F(1) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot F(0)$$

$$F(0) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 120$$

INDUÇÃO MATEMÁTICA

O princípio da indução matemática garante que funções definidas recursivamente ficam bem definidas.

Considerando que, para todo inteiro positivo, o valor da função neste inteiro é determinado de forma não ambígua, ou seja, obtemos o mesmo valor qualquer que seja o modo de aplicar as duas partes da definição.

Segundo Freitas (2020), em algumas definições, os valores da função nos primeiros k inteiros positivos são especificados. E então é fornecida uma regra para determinar o valor da função em inteiros maiores a partir dos seus valores em alguns ou todos os inteiros que o precedem.

O princípio da indução forte garante que tais definições produzem funções bem definidas. Nos exemplos do fatorial para funções e sequências, são fornecidas as regras que levam a indução para os restantes dos valores aplicados à função.

O PROBLEMA DE FIBONACCI

Os números de Fibonacci formam uma sequência infinita de números naturais, sendo que, a partir do terceiro, os números são obtidos através da soma dos dois números anteriores.

De acordo com Costa (2020), a sequência definida por Fibonacci inicia-se com 1 e 1, mas por convenção, pode-se definir $F(0) = 0$, isto é, que a sequência começa em 0 e 1.

Em termos matemáticos, a sequência é definida recursivamente pela fórmula a seguir, sendo o primeiro termo $f_1 = 1$:

$$f_n = f_{n-1} + f_{n-2}$$

Atenção! Para visualização completa da equação utilize a rolagem horizontal

★ EXEMPLO

Encontre os números de Fibonacci f_2 ; f_3 ; f_4 ; f_5 e f_6 .

Solução: Seguindo a definição:

$$f_2 = f_1 + f_0 = 1 + 0 = 1$$

$$f_3 = f_2 + f_1 = 1 + 1 = 2$$

$$f_4 = f_3 + f_2 = 2 + 1 = 3$$

$$f_5 = f_4 + f_3 = 3 + 2 = 5$$

$$f_6 = f_5 + f_4 = 5 + 3 = 8$$

SOMATÓRIO

Peterelli (2020), explica que o somatório é a adição de uma sequência de quaisquer tipos de números, chamados parcelas ou somando; o resultado é sua soma ou total. O somatório de uma sequência de um único elemento tem como resultado o próprio elemento, enquanto o de uma sequência vazia – sem elementos – resulta, por convenção, em 0.

Frequentemente, os elementos de uma sequência são definidos através de padrões regulares, como uma função de sua posição na sequência.

O somatório é denotado por meio da notação:

$$\sum_{i=m}^n x_i = x_m + x_{m+1} + x_{m+2} + \dots + x_{x-1} + x_n$$

- **Atenção!** Para visualização completa da equação utilize a rolagem horizontal

Em que:

i é o índice do somatório.

x_i é uma variável indexada que representa cada termo do somatório.

m é o índice inicial (ou limite inferior).

n é o índice final (ou limite superior).

Segundo Peterelli (2020), a expressão $i = m$ sob o símbolo de somatório significa que o índice i começa igual a m . Este é incrementado em uma unidade a cada termo subsequente, terminando quando $i = n$.

Número de termos do somatório:

É uma forma de se descobrir a quantidade de itens que serão executados na definição recursiva.

$$\sum_{i=m}^n x_i$$

- **Atenção!** Para visualização completa da equação utilize a rolagem horizontal

O número de termos da expressão resultante será dado por $t = n + 1 - m - r$, onde:

t é o número de termos do somatório expandido.

n é o índice final (ou limite superior).

m é o índice inicial (ou limite inferior).

r é o número de restrições as quais o intervalo $[m,n]$ está submetido.

O número de termos da expressão resultante terá:

$t = n + 1 - m - r = 5 + 1 - 1 - 0$, ou seja, 5 termos:

$$\sum_{i=1}^5 2i = 2(1) + 2(2) + 2(3) + 2(4) + 2(5) = 2 + 4 + 6 + 8 + 10$$

- **Atenção!** Para visualização completa da equação utilize a rolagem horizontal

CONJUNTOS

Os conjuntos são uma estrutura que possui elementos finitos e desordenados.

As sequências e as funções possuem uma sequência de objetos ordenados.

De acordo com Vidal (2020), certos conjuntos podem ser definidos recursivamente, por exemplo, os ancestrais de um ser vivo:

Os pais de um ser vivo

são seus ancestrais.



Todo pai de um ancestral

também é um ancestral.

As definições recursivas de conjuntos também têm duas partes:

PASSO BÁSICO

PASSO RECURSIVO

PASSO BÁSICO:

Uma coleção inicial de elementos é especificada.

PASSO RECURSIVO:

Regras para formar novos elementos a partir daqueles que já se sabe que estão no conjunto.

Definições recursivas também podem incluir uma regra de extensão que estipula que um conjunto definido recursivamente não contém nada mais do que:

Os elementos especificados no passo básico.

Ou elementos gerados por aplicações/execuções do passo indutivo, assumiremos que esta regra sempre vale.

★ EXEMPLO

Exemplo: Considere o subconjunto S dos inteiros definido por:

Passo básico: $2 \in S$

Passo indutivo: Se $x \in S$ e $y \in S$, então $x + y \in S$

Elementos que estão em S:

Passo básico: 2

Aplicando o passo indutivo/recursivo:

$2 + 2 = 4$ (primeira aplicação do passo indutivo)

$2 + 4 = 4 + 2 = 6$ e $4 + 4 = 8$ (segunda aplicação passo indutivo), e assim por diante.

CONJUNTOS STRINGS

Definições recursivas são muito importantes no estudo de strings (Uma cadeia de caracteres) , pois estas são composições de outras strings. Uma string sobre um alfabeto \mathbf{c} forma uma sequência finita de símbolos de \sum :

O conjunto \sum^* : de strings sobre o alfabeto \sum pode ser definido por:

Passo básico: $\lambda \in \sum^*$ (contém a string vazia).

Passo recursivo: Se $w \in \sum^*$ e $x \in \sum$, então $wx \in \sum^*$.

O passo recursivo estabelece que:

Novas strings são produzidas pela adição de um símbolo de \sum ao final das strings já em \sum^* .

A cada aplicação do passo recursivo, são geradas strings contendo um símbolo a mais.

Exemplo: se $\sum = f\{0; 1\}$:

\sum^* é o conjunto de todas as strings de bits.

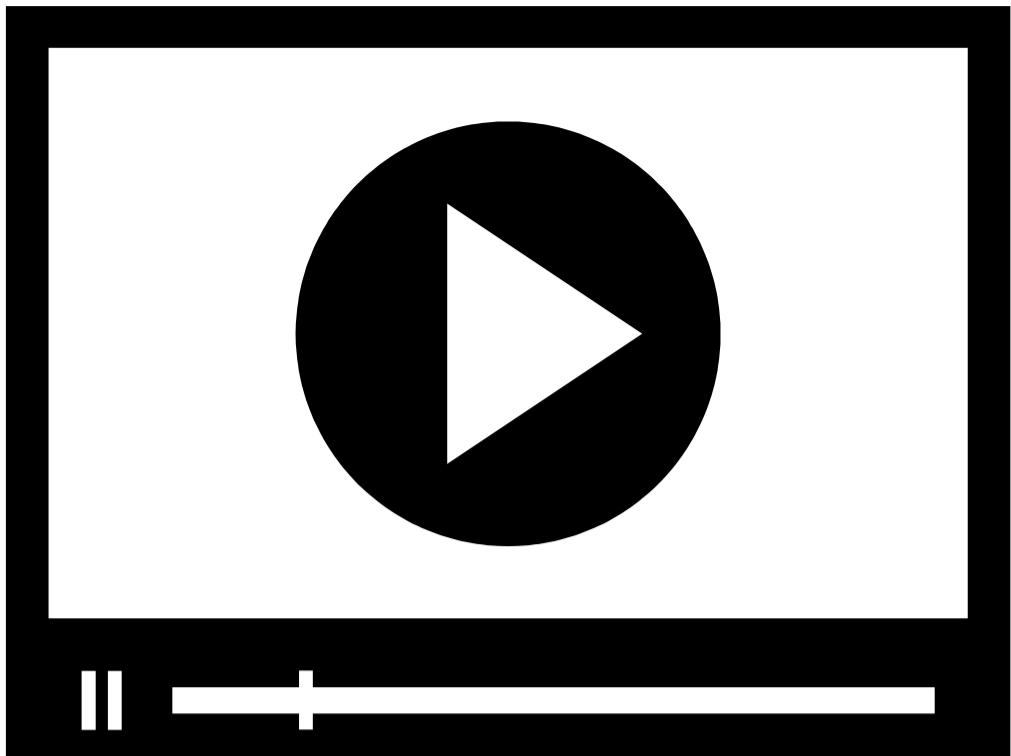
Strings que estão em \sum^* :

λ ;

0 e 1 (primeira aplicação do passo indutivo).

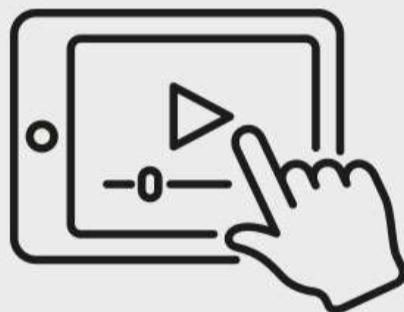
00, 01, 10, 11 (segunda aplicação passo indutivo).

UMA CADEIA DE CARACTERES



ENTENDENDO A RECURSIVIDADE

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. A DEFINIÇÃO RECURSIVA É MUITO IMPORTANTE PARA A RESOLUÇÃO DE DIVERSOS PROBLEMAS MATEMÁTICOS. QUAIS AFIRMAÇÕES A SEGUIR ESTÃO CORRETAS EM RELAÇÃO A UMA DEFINIÇÃO RECURSIVA?

I. OS MATEMÁTICOS TAMBÉM DENOMINAM DE DEFINIÇÃO INDUTIVA.

II. UM PASSO INDUTIVO OU RECURSIVO, NO QUAL OUTROS CASOS DO ITEM QUE ESTÁ SENDO DEFINIDO SÃO DETERMINADOS EM TERMOS DOS NOVOS CASOS INSERIDOS PELO USUÁRIO.

III. UMA BASE PARA DETERMINAÇÃO DE CASOS SIMPLES DO ITEM.

A) Apenas a I

B) Apenas a II

C) Apenas a I e a II.

D) Apenas a I e a III.

E) Apenas a III

2. O ALGORITMO É EXECUTADO UTILIZANDO A SEGUINTE FÓRMULA RECURSIVA: $f_n = f_{n-1} + f_{n-2}$. CONSIDERANDO QUE $f(0) = 0$ E $f(1) = 1$, INDIQUE QUAL SÉRIE ESTÁ CORRETA PARA $f(7)$:

- A) 0, 1, 1, 2, 3, 5, 8
- B) 0, 1, 1, 2, 3, 5, 8, 13
- C) 1, 2, 3, 5, 8, 13
- D) 0, 2, 3, 8, 13, 21
- E) 1, 2, 3, 5, 7, 11

GABARITO

1. A definição recursiva é muito importante para a resolução de diversos problemas matemáticos. Quais afirmações a seguir estão corretas em relação a uma definição recursiva?

I. Os matemáticos também denominam de definição indutiva.

II. Um passo indutivo ou recursivo, no qual outros casos do item que está sendo definido são determinados em termos dos novos casos inseridos pelo usuário.

III. Uma base para determinação de casos simples do item.

A alternativa "D" está correta.

A II não retrata a estratégia da definição recursiva que executa em termos dos valores passados inicialmente para a função.

2. O algoritmo é executado utilizando a seguinte fórmula recursiva: $f_n = f_{n-1} + f_{n-2}$. Considerando que $f(0) = 0$ e $f(1) = 1$, indique qual série está correta para $f(7)$:

A alternativa "B" está correta.

$$f(0) = 0$$

$$f(1) = 1$$

$$f(2) = f(1) + f(0) = 1 + 0 = 1$$

$$f(3) = f(2) + f(1) = 1 + 1 = 2$$

$$f(4) = f(3) + f(2) = 2 + 1 = 3$$

$$f(5) = f(4) + f(3) = 3 + 2 = 5$$

$$f(6) = f(5) + f(4) = 5 + 3 = 8$$

$$f(7) = f(6) + f(5) = 8 + 5 = 13$$

Portanto, a sequência correta é: 0, 1, 1, 2, 3, 5, 8, 13.

MÓDULO 2

-
- Identificar as situações de uso da recursividade

RECURSÃO

A recursividade é uma característica encontrada em diversas linguagens de programação, como: C, Java, Python, C++, entre outras.

Uma função recursiva é adequada para resolver problemas matemáticos com definições recursivas, conforme apresentado anteriormente, tais como: Fatorial, série de Fibonacci, problemas com conjuntos e sequência de valores.

Atualmente, a recursividade se torna bastante importante, pois é a base para a programação dinâmica e para linguagens com paradigma funcional, que são úteis para resolução de problemas computacionais bastante complexos.

Segundo Costa (2020), as funções recursivas são, em sua maioria, soluções mais elegantes e simples, se comparadas a funções tradicionais ou iterativas, pois executam tarefas repetitivas sem utilizar nenhuma estrutura de repetição – como **for** ou **while**.

Entretanto, essa elegância e simplicidade têm um preço que requer muita atenção em sua implementação.

O QUE É RECURSÃO?

Recursão é um método de resolução que envolve quebrar um problema em subproblemas menores e menores, até chegar a um pequeno o suficiente para que possa ser resolvido trivialmente.

Normalmente, recursão envolve uma função que chama a si mesma. Conforme explicado por Miller e Ranum (2020), essa estratégia é chamada de dividir para conquistar e é utilizada em boa parte dos algoritmos.

Isso porque a recursão permite escrever soluções elegantes para problemas que, de outra forma, podem ser muito difíceis de programar.

● COMENTÁRIO

Os problemas computacionais, com definições recursivas, são resolvidos de forma bem mais simples que de forma iterativa.

DIFERENÇA ENTRE UM ALGORITMO RECURSIVO E ITERATIVO

Consideremos o algoritmo para resolução do fatorial. A solução do fatorial se define matematicamente da seguinte forma:

$$fatorial(n) = \begin{cases} 1 & \text{se } n = 0 \\ n \times fatorial(n - 1) & \text{se } n > 0 \end{cases}$$

□ **Atenção!** Para visualização completa da equação utilize a rolagem horizontal

Figura 1 – Fórmula matemática do fatorial. Fonte: EnsineMe.

Pela definição de fatorial, o cálculo do valor de $N!$ deve ser obtido pelo cálculo de $N \cdot (N - 1) \cdot (N - 2) \cdot (N - 3) \cdots (N - N)!$, que será o caso de parada do cálculo do fatorial.

O algoritmo recursivo é apresentado a seguir:

- (1) função fatorial(n: inteiro): inteiro
- (2) inicio
- (3) se n=0 então
- (4) fatorial = 1

(5) senão

(6) fatorial = n * fatorial(n – 1)

(7) fim se

(8) fim

Figura 2 – Implementação factorial recursivo. Fonte: EnsineMe.

Na linha (3) é definida a condição base para realizar a parada das chamadas recursivas da função. Conforme a definição matemática, o valor do factorial de 0 ($0!$) é definida como 1.

Se o algoritmo não chegou na condição de parada, a função será chamada novamente na linha (5) para resolução de um problema com valor menor ($n - 1$), e assim, sucessivamente.

O cálculo do factorial do número 5, por exemplo, divide-se no cálculo do factorial do número 4, do número 3, até chegar ao cálculo do valor do factorial de zero, cujo valor é 1.

RESOLUÇÃO ITERATIVA DO FATORIAL

Conforme explica Costa (2020), para a mesma definição matemática de factorial mencionada anteriormente, pode-se utilizar uma implementação iterativa, ou seja, utilizando comandos de repetição como: Repita, enquanto e para.

A seguir, é apresentada uma versão iterativa para o algoritmo factorial.

função factorial (n: inteiro): inteiro

var n, aux : inteiro

inicio

aux = 1

para i de 1 até n faça

aux = aux * n

fim_para

fatorial = aux

fim

Figura 3 – Implementação factorial iterativo. Fonte: EnsineMe.

Na linha (4), é criada uma variável aux para o armazenamento dos valores do factorial. Na linha (5), é criado o loop para calcular o factorial a partir do valor 1 até o n.

Esse valor é armazenado na variável aux. Na linha (8), o factorial recebe o valor da variável aux. Comparando as duas versões, percebe-se que o algoritmo recursivo é mais intuitivo, pois implementa diretamente a definição recursiva do factorial.

Outro ponto importante, para se indicar a utilização de recursividade, é quando há necessidade de guardar em pilha os estados intermediários para resolução de um problema, como visitar os nós de uma árvore binária ou um algoritmo de ordenação.

AS TRÊS LEIS DA RECURSAO

De acordo com Miller e Ranum (2020), para se implementar um algoritmo recursivo completo, devem ser obedecidas três leis importantes:

LEI 1

Um algoritmo recursivo deve ter um caso básico.



LEI 2

Um algoritmo recursivo deve mudar o seu estado e se aproximar do caso básico.



LEI 3

Um algoritmo recursivo deve chamar a si mesmo, recursivamente.

LEI 1

Um caso básico é o ponto onde a função será encerrada, e é geralmente um limite superior ou inferior da regra geral da definição recursiva da função. Esse limite é um problema suficientemente pequeno para resolver diretamente. No caso do fatorial, o caso básico é valor do fatorial do número 0, que é definido como 1.

LEI 2

Para se atender à segunda lei, deve-se haver mudança de estados para que aproxime o algoritmo do caso básico apresentado na lei 1. Para haver a mudança, os dados devem ser alterados para reduzir o problema de alguma forma, utilizando a estratégia do dividir para conquistar, conforme mencionado anteriormente. No caso do fatorial, a progressão natural do problema será diminuir o valor do número a ser calculado o fatorial. Então, o valor será diminuído em uma unidade ($n - 1$).

LEI 3

É a chamada do algoritmo a si mesmo, considerando a própria definição. No caso do fatorial, é chamado da função fatorial com uma unidade a menos (fatorial ($n - 1$)), até a finalização do caso básico. A princípio, parece que o algoritmo entrará em loop infinito, no entanto, o problema tem um ponto de parada e é resolvido de forma mais direta.

TIPOS DE RECURSIVIDADE

Existem alguns tipos mais complexos de recursividade para a resolução de problemas mais elaborados cuja recursão simples não suporta.

RECURSAO EM CAUDA

Uma recursão é chamada **em cauda** quando é o último comando a ser executado em uma função recursiva.

Por exemplo, na Figura 2, a função recursiva implementada para o fatorial não é em cauda, pois o retorno da função é multiplicado por n para ser atribuída ao fatorial.

Para ficar mais clara a recursão em cauda, vamos apresentar, na Figura 4, a função fatorial utilizando a estratégia:

```
função fatorial (n:intero, valor_atual: intero) : intero
  inicio
    se n = 0 então
```

```
valor_atual
senão
fatorial = fatorial (n – 1, valor_atual * n)
fim se
fim
```

Figura 4 – Função factorial utilizando recursão em cauda. Fonte: EnsineMe.

É importante analisar a pilha de execução para verificar a diferença da recursão em cauda para a recursão comum. Supondo a execução do factorial de 5:

```
fatorial(5, 1)
fatorial(4, 1 * 5)//i = 5
fatorial(3, 5 * 4)//i = 4
fatorial(2, 20 * 3)//i = 3
fatorial(1, 60 * 2)//i = 2
fatorial(1, 120 * 1)//i = 1
return 120 //i=0
```

Figura 5 – Execução da função factorial com recursão em cauda. Fonte: EnsineMe.

A diferença fundamental é que a própria função armazena o valor do resultado final da execução.

Para isso, na Figura 4, na linha (6), a própria função é passada para a próxima chamada, mudando a função para ter 2 parâmetros:

O primeiro possui o valor do índice para se chegar ao caso básico.



O segundo possui o valor atual do fatorial.

● COMENTÁRIO

Segundo o Programmerinterview.com (2020), uma função é considerada recursiva em cauda se o resultado final da chamada recursiva, no caso 120, for também o resultado final da execução da função. Dessa forma, a última chamada já possui o resultado da função.

RECURSÃO INDIRETA

As funções podem ser recursivas (invocar a si próprias) indiretamente, fazendo isso através de outras funções: "P" pode chamar "Q", que chama "R", e assim por diante, até que "P" seja novamente invocada.

Costa (2020), explica que um exemplo é a análise de expressões matemáticas realizadas, principalmente, por compiladores.

Suponha que exista um analisador sintático para cada tipo de subexpressão, e tenha uma expressão " $3 + (2 * (4 + 4))$ ". A função que processa expressões "+" chamaria uma segunda função que processaria expressões "*", que, por sua vez, chamaria novamente a primeira função que processa "+" para resolução da sua parte recebida na execução.

Seguem, na Figura 6, exemplos de chamada indireta em duas funções recursivas:

```
Função par(x) : booleano
x: inteiro
inicio
Se x = 0 então
    retorno (verdadeiro)
Senão
```

```

Se x = 1 então
    retorno (falso)
senão
    retorno(impar(x-1))
fim se
fim se
fim

Função impar(x) : booleano
x: inteiro
inicio
Se x = 0 então
    retorno (falso)
Senão
Se x = 1 então
    retorno (verdadeiro)
senão
    retorno(par(x-1))
fim se
fim se
fim

```

Figura 6 – Recursão indireta. Fonte: EnsineMe.

A recursão indireta acontece na linha (10) da função par, invocando a função ímpar, e na linha (10) da função ímpar é feita uma invocação para a função par. Nesse caso, ainda se torna mais problemático o tamanho da pilha na memória, pois são duas funções em paralelo.

RECURSÃO ANINHADA

Uma função recursiva pode receber um argumento que inclui outra função recursiva.

Um exemplo é a função de Ackermann, que por ser aninhada, cresce de forma exponencial, pois internamente ela é chamada novamente como parâmetro de uma chamada, conforme a Figura 7.

```

Função ackermann(n:inteiro, m:inteiro)
inicio
se n = 0
    retorno m + 1
senão se n > 0 e m = 0 então
    ackermann(n - 1,m)
senão
    ackermann(n - 1, ackermann(n , m-1))
fim se
fim

```

Figura 7 – Recursão aninhada. Fonte: EnsineMe.

Na linha (8), é realizada a recursão aninhada dentro da chamada ackermann (n - 1, ackermann(n , m - 1)). Segundo Abreu (2020), algumas vezes também pode ser chamada de recursão dupla.

QUANDO NÃO EMPREGAR RECURSIVIDADE

● COMENTÁRIO

A grande questão, portanto, é saber quando utilizar a recursão ou a iteração. Solucionar alguns problemas através da iteração pode ser bem trabalhoso e demorado, enquanto a recursão exige menos esforço de codificação e lógica.

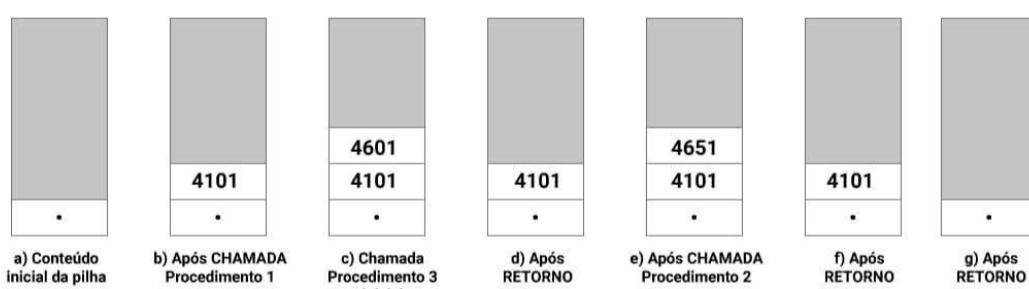
Por outro lado, em termos de eficiência, a iteração é mais rápida em muitos casos. Isso porque, em sua forma mais simples e genérica, ela consiste na reexecução de um conjunto de instruções, enquanto que cada chamada recursiva necessita da criação de uma nova sub-rotina em memória – que pode conter endereços de parâmetros e endereço de retorno. Portanto, os valores dos parâmetros não podem ser perdidos e a recursão deve funcionar adequadamente para cada chamada como se fosse uma nova função totalmente à parte da que está em execução. Por isso, uma pilha é utilizada em **procedimentos reentrantes**.

“

UM PROCEDIMENTO REENTRANTE "É AQUELE EM QUE É POSSÍVEL TER VÁRIAS CHAMADAS ABERTAS AO MESMO TEMPO.

UM PROCEDIMENTO RECURSIVO (AQUELE QUE CHAMA A SI MESMO) É UM EXEMPLO DO USO DESSE RECURSO."

(STALLINGS, 2010).



Fonte: EnsineMe

● Figura 8 – Empilhamento em memória de procedimentos chamados. Fonte: EnsineMe.

Percebe-se, na Figura 8, que a recursão utiliza bastante memória e tempo de processamento, pois para cada chamada para uma função recursiva, deve-se armazenar o estado da chamada anterior à função até que o caso particular aconteça.

💡 DICA

Lembre-se de que memória é consumida cada vez que o computador faz uma chamada a uma função. Portanto, com funções recursivas, a memória do computador pode se esgotar rapidamente.

A vantagem de se utilizar funções recursivas é que elas deixam o código mais legível e mais fácil de entender, com uma estrutura mais parecida com a sua definição.

Para exemplificar o problema com chamadas recursivas, vamos analisar a chamada a um programa que calcula a sequência de Fibonacci, conforme programa apresentado na Figura 9:

```
//cálculo da sequência de Fibonacci com uso da recursividade.
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
//protótipo de função Fibonacci
```

```
int fibonacci(int);
```

```
int main()
```

```
{
```

```
int n, i;
```

```
printf("Digite a quantidade de termos da série ");
```

```
scanf("%d", &n);
```

```
printf("Sequência de Fibonacci e: \n");
```

```
for(i=0; i < n; i++)
```

```
printf("%d", fibonacci(i+1));
```

```
}
```

```
//função recursiva de Fibonacci
```

```
int fibonacci(int num)
```

```
{
```

```
if (num == 1|| num == 2)
```

```
return 1;
```

```
else
```

```
return fibonacci(num - 1) + fibonacci(num - 2);
```

```
}
```

Figura 9 – Programa para calcular a sequência de Fibonacci com recursividade. Fonte: EnsineMe.

Cada chamada à função Fibonacci que não corresponde a um dos casos base resulta em mais duas chamadas recursivas. Esse número cresce rapidamente, mesmo para números bem pequenos da série.

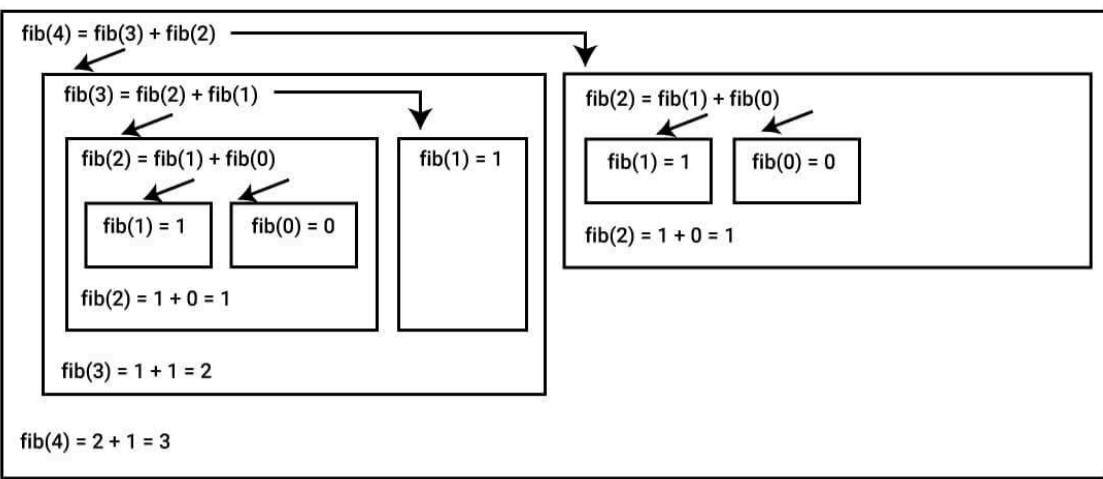
Por exemplo, o Fibonacci de 31 exige 4.356.617 chamadas à função, enquanto o Fibonacci de 32 exige 7.049.155, ou seja, 2.692.538 chamadas adicionais à função.

Logo, é interessante evitar programas recursivos no estilo de Fibonacci que resultam em uma “explosão” exponencial de chamadas. Isso porque, a cada chamada recursiva da função, ela é adicionada a uma pilha; da mesma forma, a cada término da função, ela é desempilhada.

Esse empilhamento e desempilhamento repetitivo consome tempo, fazendo com que o mesmo programa implementado iterativamente venha a ser mais rápido que o recursivo (embora, por outro lado, provavelmente não seja de compreensão e manutenção tão simples).

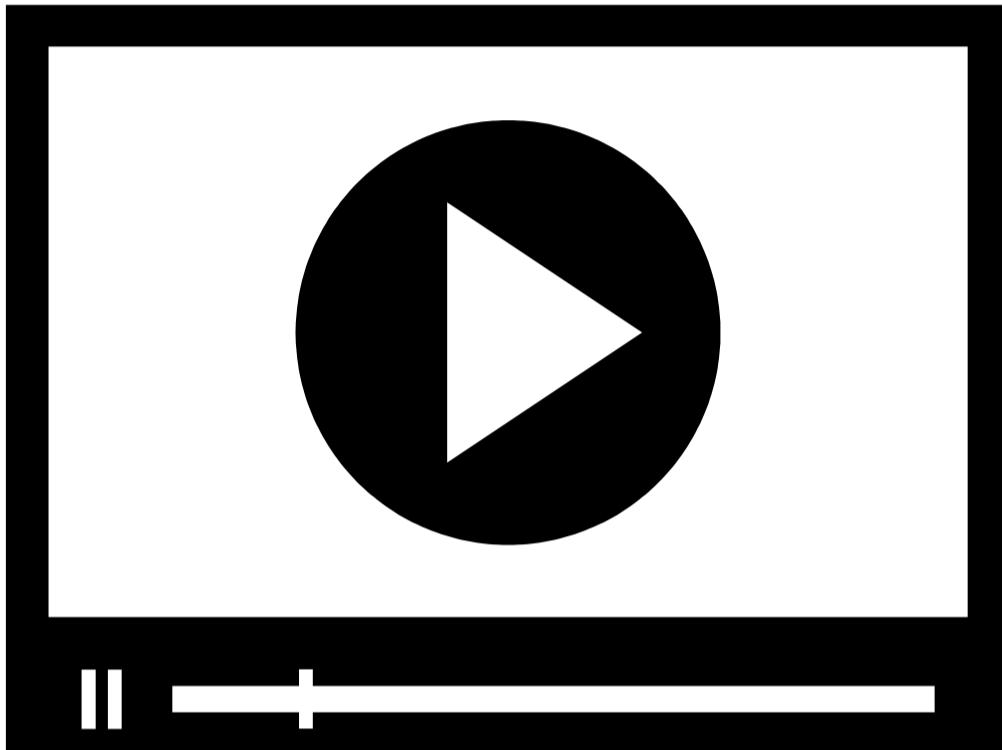
Além disso, a pilha de funções apresenta um tamanho **máximo**, limitando, assim, o uso de recursões que mantenham muitas funções na pilha (como o Fibonacci de um número grande).

A Figura 10 descreve essa “explosão” exponencial de chamadas para o cálculo de Fibonacci (4):



Fonte: EnsineMe

Figura 10 – Cálculo de Fibonacci. Fonte: EnsineMe.



CHAMADAS RECURSIVAS NA MEMÓRIA

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. SUPONHA QUE VOCÊ PRECISA ESCRIVER UMA FUNÇÃO RECURSIVA PARA CALCULAR O FATORIAL DE UM NÚMERO. A FUNÇÃO FAT (N) RETORNA $N * N - 1 * N - 2 * \dots$, NO QUAL O FATORIAL DE ZERO É DEFINIDO COMO 1. QUAL SERIA O CASO BÁSICO MAIS APROPRIADO?

A) $n == 0$

B) $n == 1$

C) $n \geq 0$

D) $n \leq 1$

E) $n = 2$

2. (FCC – 2015 – DPE/SP – PROGRAMADOR) O USO DA RECURSIVIDADE GERALMENTE PERMITE UMA DESCRIÇÃO MAIS CLARA E CONCISA DOS ALGORITMOS. EM RELAÇÃO AOS CONCEITOS E UTILIZAÇÃO DE RECURSIVIDADE, É CORRETO AFIRMAR:

- A) Um compilador implementa um procedimento recursivo por meio de um deque, no qual são armazenados os dados usados em cada chamada de um procedimento que ainda não terminou de processar.
- B) Uma exigência fundamental é que a chamada recursiva a um procedimento P esteja sujeita a uma condição B, que não deve ser satisfeita em nenhum momento da execução.
- C) Algoritmos recursivos são apropriados quando o problema a ser resolvido ou os dados a serem tratados são definidos em termos recursivos, pois isso garante sempre a melhor solução para resolver o problema.
- D) Apenas os dados não globais vão para o deque de controle, pois o estado corrente da computação deve ser registrado, para que possa ser recuperado de uma nova ativação de um procedimento recursivo.
- E) Na prática, é necessário garantir que o nível mais profundo de recursão seja finito e que também possa ser mantido pequeno, pois em cada ativação recursiva de um procedimento P, uma parcela de memória é requerida.

GABARITO

1. Suponha que você precisa escrever uma função recursiva para calcular o factorial de um número. A função fat (n) retorna $n * n - 1 * n - 2 * \dots$, no qual o factorial de zero é definido como 1. Qual seria o caso básico mais apropriado?

A alternativa "A" está correta.

O caso básico para a função factorial é para o cálculo do factorial 0, portanto, o caso básico será para quando n for igual a 0, e a função deve estar preparada para responder com valor 1.

2. (FCC – 2015 – DPE/SP – Programador) O uso da recursividade geralmente permite uma descrição mais clara e concisa dos algoritmos. Em relação aos conceitos e utilização de recursividade, é correto afirmar:

A alternativa "E" está correta.

É vital que a quantidade de chamadas recursivas seja finita, pois funções recursivas utilizam bastante memória.

MÓDULO 3

- ④ Identificar a utilização de funções recursivas através de exemplos

EXEMPLIFICANDO RECURSAO

Neste módulo, serão apresentados alguns exemplos de algoritmos que utilizam procedimentos recursivos para a resolução de problemas computacionais.

O primeiro problema que vamos tratar é o da determinação da soma dos n primeiros números:

$$\text{Sigma}(n) = 1 + 2 + 3 + 4 + \dots + (n - 1) + n$$

Isso poderia ser feito pelo trecho algoritmo na Figura 11, no qual todas as variáveis são supostamente declaradas como inteiros.

início

soma <- 0;

ind <- 1;

enquanto ind <= n faça

início

soma <- soma + ind;

ind <- ind + 1;

fim;

sigma <- soma;

fim.

Figura 11 – Declaração de variáveis. Fonte: EnsineMe.

Esse mesmo resultado poderia ser obtido como nos trechos do algoritmo principal e do procedimento recursivo denominado de **calcSigma**.

Primeiramente, vamos à descrição do programa principal na Figura 12:

início

soma <- 0;

ind <- 1;

execute calcSigma;

sigma <- soma;

fim.

Figura 12 – Programa principal calcSigma. Fonte: EnsineMe.

A seguir, o procedimento recursivo calcSigma na Figura 13:

proc calcSigma

Se i <= n

Então

início

soma <- soma + 1;

ind <- ind + 1;

execute calcSigma;

fim

Senão

retorne;

fim-se;

Figura 13 – Programa calcSigma. Fonte: EnsineMe.

Desta forma, teremos um cálculo iterativo executado por um procedimento recursivo, o que pode ser feito de maneira geral.

Vejamos um exemplo, apresentado na Figura 14, para o cálculo da raiz quadrada. Neste procedimento, o que se faz para achar a raiz de x é partir de um número qualquer r, menor do que x, como aproximação inicial do valor da raiz.

Se r^2 é igual a x (a menos de uma tolerância indicada), então r é uma boa aproximação da raiz. Senão, repete-se o processo, usando como nova aproximação a média aritmética entre r e $\frac{x}{r}$.

proc raiz : real(x : real, r : real, tol : real)

Se $|x - r^2| \leq tol$

Então

```
retorne r;
```

```
Senão;
```

```
retorne raiz(x, (x / r + r) / 2, tol );
```

```
fim-se;
```

Figura 14 – Cálculo da raiz quadrada. Fonte: EnsineMe.

O procedimento, apresentado na Figura 15, faz a extração de uma subcadeia:

```
proc subcadeia : cadeia (c: cadeia, p : int, n: int)
```

```
Se n = 0
```

```
Então
```

```
retorne “ ”
```

```
Senão;
```

```
Se p > 1
```

```
Então
```

```
retorne subcadeia (cont c, p – 1, n)
```

```
Senão
```

```
retorne subcadeia( cont c, 1, n – 1);
```

```
fim-se;
```

```
fim-se;
```

Figura 15 – Cálculo da subcadeia. Fonte: EnsineMe.

A seguir, um exemplo, apresentado na Figura 16, de procedimento recursivo para realizar a soma de elementos de um vetor, onde n <=n úmero de elementos do vetor v:

```
proc somavet : int (v: vetor, n: int)
```

```
Se n = 0
```

```
Então
```

```
retorne 0;
```

```
Senão;
```

```
retorne v [ n ] + somavet (v, n – 1)
```

Figura 16 – Soma de elementos de um vetor. Fonte: EnsineMe.

O exemplo, apresentado na Figura 17, mostra a pesquisa de valor em um vetor por pesquisa binária.

```
proc pesqbin : log (v : vet [1...10] : real, r : real)
```

```
var
```

```
i, j : int;
```

```
início
```

```
retorne pesqrec (1, 10);
```

```
fim
```

Figura 17 – Busca de valor em um vetor por pesquisa binária. Fonte: EnsineMe.

O procedimento, apresentado na Figura 18, busca r no trecho de v entre i e j .

```
proc pesrec : log (i : int, j : int)
```

```
Se i = j
```

```
Então
```

```
Se v [ i ] = r
```

```
Então
```

```
retorne V
```

```
Senão
```

```
retorne F
```

```
Senão
```

retorne pesqrec(i, (i + j) div 2) ou

retorne pesqrec ((i + j) div 2 + 1, j)

Figura 18 – Busca de valor em um vetor por pesquisa binária. Fonte: EnsineMe.

EXEMPLOS DE EMPREGO DOS ALGORITMOS RECURSIVOS

Analisemos o exemplo clássico do Jogo da Torre da Hanói: Neste jogo, temos três hastes, que chamamos de Origem, Destino e Temporária, e um número qualquer de discos de tamanhos diferentes posicionados inicialmente na haste Origem.

Os discos são dispostos em ordem de tamanho, de forma que o maior disco fique embaixo, em seguida o segundo maior, e assim por diante, conforme a Figura 19.

O objetivo do jogo é movimentar um a um os discos da haste Origem para a haste Destino utilizando a haste Temporária como auxiliar.

Nenhum disco pode ser colocado sobre um disco menor.



Fonte: shutterstock.com

Figura 19 – Torre de Hanói. Fonte: shutterstock.com

A função que será escrita recebe o número de discos e o nome das hastes como argumento e exibe a solução.

Para tanto, vamos considerar os seguintes passos:

PASSO 1

Mover **n-1** discos da haste Origem para a haste Temporária.

PASSO 2

Mover o disco **n** da haste Origem para a haste Destino.

PASSO 3

Mover **n-1** discos da haste Temporária para a haste Destino.

► ATENÇÃO

A função **mover()**, apresentada na Figura 20, é duplamente recursiva e foi escrita seguindo os três passos que acabamos de apresentar.

Veja, a seguir, a função **mover()** escrita na linguagem de programação C.

```
// Resolve o jogo da Torre de Hanói utilizando recursão
```

```
#include <stdio.h>
```

```

#include <stdlib.h>
// protótipo da função recursiva
void mover (int n, char Orig, char Temp, char Dest);

int main()
{
    mover (3, 'O', 'T','D');
    system("pause");
    return 0;
}

void mover (int n, char Orig, char Temp, char Dest)
{
    if (n == 1)
        printf("\nMova o disco 1 da haste %c", Orig, " para a haste %c\n", Dest);
    else
    {
        mover(n - 1, Orig, Dest, Temp);
        printf("\nMova o disco %d", n, "da haste %c", Orig, " para a haste %c\n", Dest);
        mover(n - 1, Temp, Orig, Dest);
    }
}

```

Figura 20 – Função mover(). Fonte: EnsineMe.

A saída para o programa, conforme a Figura 21, será a seguinte:

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

Mova o disco 1 da haste O
Mova o disco 2
Mova o disco 1 da haste D
Mova o disco 3
Mova o disco 1 da haste T
Mova o disco 2
Mova o disco 1 da haste O
[Done] exited with code=0 in 0.092 seconds

```

Fonte: EnsineMe.

Figura 21 – Saída do programa Torre de Hanói. Fonte: EnsineMe.

Analisemos, então, o problema de calcular o fatorial de um número empregando a linguagem C.

Na Figura 22, é apresentado um programa em C sem a utilização da recursividade para calcular o fatorial de um número:

```

// cálculo do fatorial de um número sem o uso da recursividade
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int fat, n;
    printf("Insira um valor para o qual deseja calcular seu fatorial: ");
    scanf("%d", &n);
    for(fat = 1; n > 1; n--)
        fat = fat * n;
}

```

```
printf("\nFatorial calculado: %d", fat);
return 0;
}
```

Figura 22 – Programa para cálculo do fatorial sem recursividade. Fonte: EnsineMe.

A saída para o programa, considerando o cálculo para o fatorial do número 5:

```
Insira um valor para o qual deseja calcular seu fatorial: 5
Fatorial calculado: 120
...Program finished with exit code 0
Press ENTER to exit console.
```

Fonte: EnsineMe.

Figura 23 – Saída do programa para cálculo do fatorial sem recursividade. Fonte: EnsineMe.

Vejamos o cálculo de um fatorial utilizando uma função que seja recursiva, conforme o código apresentado na Figura 24.

Note que, enquanto n não for igual a 0, a função **fat()** chama a si mesma, cada vez com um valor menor.

Dessa forma, n = 0 é critério de parada para essa função.

```
//cálculo do fatorial de um número com o uso da recursividade
```

```
#include <stdlib.h>
```

```
//protótipo da função recursiva fat()
```

```
int fat(int);
```

```
int main()
```

```
{
```

```
int n;
```

```
printf("\n\n Digite um valor para n:")
```

```
scanf("%d", &n);
```

```
printf("\nO fatorial de %d e' %d, n, fat(n));
```

```
return 0;
```

```
}
```

```
//função recursiva fat()
```

```
int fat(int n)
```

```
{
```

```
if(n)
```

```
return n * fat(n-1);
```

```
else
```

```
return 1;
```

```
}
```

Figura 24 – Programa para cálculo do fatorial com recursividade. Fonte: EnsineMe.

Segue a saída, na Figura 25, para o mesmo cálculo do fatorial de 5, agora utilizando a função recursiva **fat()**:

```
Digite um valor para n: 5
O fatorial de 5 e' 120
...Program finished with exit code 0
Press ENTER to exit console.
```

Fonte: EnsineMe.

Figura 25 – Saída do programa para cálculo do fatorial com recursividade. Fonte: EnsineMe.

A seguir, o cálculo da sequência de Fibonacci sem a utilização da recursividade, conforme código apresentado na Figura 26:

```
// cálculo da sequência de Fibonacci sem o uso da recursividade
#include <stdio.h>
void main()
{
int a, b, auxiliar, i, n;
a = 0;
b = 1;
printf("Digite um número: ");
scanf("%d", &n);
printf("Série de Fibonacci:\n");
printf("%d\n", b);
for(i = 1; i < n; i++)
{
auxiliar = a + b;
a = b;
b = auxiliar;
printf("%d\n", auxiliar);
}
}
```

Figura 26 – Programa para cálculo da sequência de Fibonacci sem recursividade. Fonte: EnsineMe.

A saída para o cálculo dos dez primeiros elementos da série de Fibonacci é mostrada na Figura 27:

Digite um número: 10

Série de Fibonacci:

```
1
1
2
3
5
8
13
21
34
55
```

Figura 27 – Saída do programa para cálculo da sequência de Fibonacci sem recursividade. Fonte: EnsineMe.

Na Figura 28, um exemplo de um programa para apresentar a sequência dos 10 primeiros termos de Fibonacci utilizando recursividade. Note que o laço for contido na função principal main() chama a função Fibonacci(), que calcula os valores retornando o valor 1 quando a posição da sequência for igual a 1 ou 2. E posteriormente calcula o restante dos números, sempre somando as duas posições anteriores para obter o resultado atual.

```
// cálculo da sequência de Fibonacci com o uso da recursividade
#include <stdio.h>
#include <stdlib.h>

// protótipo da função Fibonacci
int fibonnaci(int);

int main()
```

```

{
int n, i;

printf("Digite a quantidade de termos da sequência de Fibonacci: ");
scanf("%d", &n);

printf("\nA sequência de Fibonacci é: \n");
for(i=0; i < n; i++)
printf("%d ", fibonacci(i+1));
}

// função recursiva de Fibonacci
int fibonacci(int num)
{
if(num == 1 || num == 2)
return 1;
else
return fibonacci(num - 1) + fibonacci(num - 2);
}

```

Figura 28 – Programa para cálculo da sequência de Fibonacci com recursividade. Fonte: EnsineMe.

A saída para o cálculo dos dez primeiros elementos da série de Fibonacci utilizando a recursividade é mostrada na Figura 29:

```

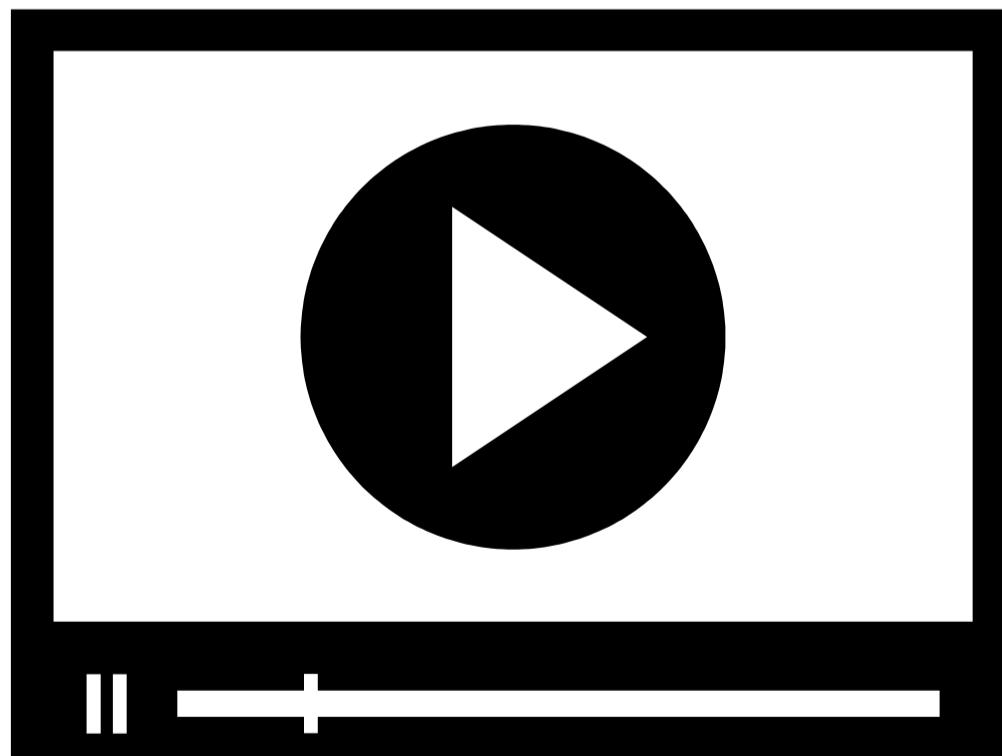
Digite a quantidade de termos da série: 10
A sequência de Fibonacci é:
1 1 2 3 5 8 13 21 34 55

...Program finished with exit code 0
Press ENTER to exit console. []

```

Fonte: EnsineMe.

Figura 29 – Saída do programa para cálculo da sequência de Fibonacci sem recursividade. Fonte: EnsineMe.



ANALISANDO O DESEMPENHO DOS ALGORITMOS RECURSIVOS

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. (FGV – 2010 – DETRAN/RN – PROGRAMADOR DE COMPUTADOR) SOBRE O PROCEDIMENTO DE RECURSIVIDADE, ANALISE OS ALGORITMOS A SEGUIR E ASSINALE A ALTERNATIVA CORRETA:

ALGORITMO I

FUNÇÃO FATORIAL(I)

FAT(I) := SE I ≤ 1 ENTÃO 1 SENÃO I X FAT(I-1)

ALGORITMO II

FAT[0]:= 1

PARA J := 1, ..., N FAÇA

FAT[J] := J X FAT[J-1]

A) Os algoritmos I e II são recursivos.

B) Somente o algoritmo I é recursivo.

C) Somente o algoritmo II é recursivo.

D) O algoritmo I não é recursivo e o II é orientado a objeto.

E) Os algoritmos I e II não são recursivos.

2. (FAPERJ – 2015 – SEMAE – ANALISTA EM TECNOLOGIA DA INFORMAÇÃO) UMA BOA LÓGICA DE PROGRAMAÇÃO É FUNDAMENTAL PARA QUE OS ALGORITMOS SEJAM BEM DESENVOLVIDOS E, CONSEQUENTEMENTE, OS PROGRAMAS BEM IMPLEMENTADOS, CLARO QUE SE AGREGANDO O CONHECIMENTO DA SINTAXE DA LINGUAGEM DE PROGRAMAÇÃO ESCOLHIDA. DESSA FORMA, PENSANDO-SE EM ESTRUTURAS DE ALGUMAS IMPLEMENTAÇÕES, CONSIDERE O SEGUINTE TRECHO DE CÓDIGO:

```
INT FIBONACCI (INT N) {  
    IF (N <= 1)  
        RETURN N;  
    ELSE  
        RETURN (FIBONACCI(N - 1) + FIBONACCI(N - 2));  
}
```

PODE-SE AFIRMAR, A PARTIR DO CÓDIGO ANTERIORMENTE APRESENTADO, QUE:

- A)** Existe uma estrutura de repetição.
- B)** Existe uma estrutura de desvio múltipla.
- C)** Existe um processo recursivo.
- D)** Existe uma chamada de função com passagem de parâmetros por referência.
- E)** Existe uma chamada de função não recursiva.

GABARITO

1. (FGV – 2010 – DETRAN/RN – Programador de computador) Sobre o procedimento de recursividade, analise os algoritmos a seguir e assinale a alternativa correta:

Algoritmo I

função fatorial(i)

fat(i) := se i ≤ 1 então 1 senão i x fat(i-1)

Algoritmo II

fat[0]:= 1

para j := 1, ..., n faça

fat[j] := j x fat[j-1]

A alternativa "B" está correta.

Veja que o algoritmo II não é recursivo, pois **j** é o índice da estrutura de repetição **para**. Portanto, somente o algoritmo I é recursivo

2. (FAPERJ – 2015 – SeMAE – Analista em Tecnologia da Informação) Uma boa lógica de programação é fundamental para que os algoritmos sejam bem desenvolvidos e, consequentemente, os programas bem implementados, claro que se agregando o conhecimento da sintaxe da linguagem de programação escolhida. Dessa forma, pensando-se em estruturas de algumas implementações, considere o seguinte trecho de código:

```
int fibonacci (int N) {  
    if (N <= 1)  
        return N;  
    else  
        return (fibonacci(N - 1) + fibonacci(N - 2));  
}
```

Pode-se afirmar, a partir do código anteriormente apresentado, que:

A alternativa "C" está correta.

Analizando o código apresentado, podemos perceber que no bloco de comandos do **else** existe uma chamada da função **Fibonacci()**, o que caracteriza que existe um processo recursivo.

CONCLUSÃO

CONSIDERAÇÕES FINAIS

A recursividade se apresenta como uma ferramenta poderosa para resolução de problemas computacionais, tanto acadêmicos quanto comerciais. Por isso, vários algoritmos de ordenação e de estrutura de dados são baseados em programas recursivos.

Portanto, o entendimento e o domínio da aplicação da recursividade são obrigatórios para o desenvolvimento de softwares mais complexos e com um desempenho adequado para a resolução de problemas.

A evolução das linguagens de programação aponta para a utilização em larga escala de linguagens funcionais, como F# e Haskell. Como o próprio nome indica, elas são baseadas em funções e, principalmente, funções recursivas. Inclusive, na utilização de linguagens para aplicações comerciais como C#, é bastante comum a chamada de rotinas funcionais F# para resolução de alguns problemas de forma otimizada.

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.



REFERÊNCIAS

ABREU, J. **Dia C: Recursão.** In: VidaGeek. Publicado em: 09 mar. 2007.

COSTA, M. **Estrutura de dados II – Notas de aula.** Março de 2019, 20 f.

FREITAS, D. **Fundamentos de Matemática Discreta para a Computação.** Notas de aula. Consultado em meio eletrônico em: 16 nov. 2020.

MILLER, B.; RANUM, D. **Aprendendo com Python:** Edição interativa. Recursão. Tradução de C.

H. Morimoto, J. C. de Pina Jr., J. A. Soares. In: Panda. IME. Consultado em meio eletrônico em: 16 nov. 2020.

OLIVEIRA, A. **Notações sobre definições recursivas.** In: CIn-UFPE. Consultado em meio eletrônico em: 16 nov. 2020.

OLIVEIRA, R. **Algoritmos e programação de computadores.** Notas de aula. Consultado em meio eletrônico em: 16 nov. 2020.

PETERNELLI, L. **Capítulo 1 – Conceitos introdutórios.** In: Estatística I. DPI-UFV. Consultado em meio eletrônico em: 16 nov. 2020.

PROGRAMMERINTERVIEW. **What is Tail Recursion?** Provide an example and a simple explanation. In: Programmerinterview. Consultado em meio eletrônico em: 16 nov. 2020.

QCONCURSOS. **Questões de concursos.** In: QConcursos. Consultado em meio eletrônico em: 16 nov. 2020.

SILVA, A. **Em programação, quais as desvantagens de se usar uma função recursiva?** In: Quora. Consultado em meio eletrônico em: 16 nov. 2020.

STALLINGS, W. **Arquitetura e organização de computadores.** 10. ed. Londres: Pearson, 2000.

VIDAL, A. **Recursão.** Consultado em meio eletrônico em: 16 nov. 2020.

EXPLORE+

Para saber mais sobre os assuntos tratados neste tema, leia:

Algorithms, Sanjoy Dasgupta, Christos H. Papadimitriou e Umesh Vazirani, para mais informações sobre programação dinâmica e sua importância na solução de problemas.

Functional thinking: Paradigm over syntax, Neal Ford, são introduzidas e apresentadas diversas estratégias para utilização da programação funcional.

CONTEUDISTA

Marcelo Nascimento Costa

🔗 CURRÍCULO LATTES