VOLTAR

Searching...

Recursividade

Apresentar os conceitos de recursividade e alguns exemplos.

NESTE TÓPICO

Definição

Já foi visto que repetições podem ser obtidas escrevendo-se laços, tais como laços for ou laços while. Outra forma de se obter repetição é por meio da recursão.

Um objeto é recursivo se ele é definido parcialmente em termos de si próprio. A recursão é considerada uma técnica poderosa em definições matemáticas. O poder da recursão está na possibilidade de definir elementos com base em versões mais simples deles mesmos.

Algoritmos recursivos são particularmente apropriados quando o problema a ser resolvido, a função a ser computada, ou os dados já estão definidos em termos recursivos ou por indução. Entretanto, isso não significa que tal definição recursiva garanta a melhor solução do problema em termos de eficiência. A única ferramenta necessária para expressar operações recursivamente é o próprio procedimento ou a função, que tem a capacidade de invocar a si mesmo.

Exemplos:

- Potência positiva *(n>=0)* de um número X
  - x^n *= 1 se n=0 (condição de parada ou parte base)*

- $x^n = x * x^{n-1}$ se $n > 0$ *(parte recursiva)*

- Fatorial
  - *0! = 1 (parte base)*
  - *n! = n * (n-1)! (parte recursiva)*

- Seqüência de Fibonacci
  - *fibonacci(0) = 0 (parte base)*
  - *fibonacci(1) = 1 (parte base)*
  - *fibonacci(n) = fibonacci(n-1) + fibonacci(n-2), n>1 (parte recursiva)*

Características

- Uma **condição de parada**, isto é, algum evento que encerre a autochamada consecutiva. No caso do fatorial, isso ocorre quando a função é chamada com parâmetro (*n*) igual a 1. Um algoritmo recursivo precisa garantir que esta condição será alcançada.

- Uma **mudança de "estado"** a cada chamada, isto é, alguma "diferença" entre uma chamada e a próxima. No caso do fatorial, o parâmetro *n* é decrementado a cada chamada.

Padrão de definições recursivas

I. o caso simples é definido explicitamente:

Exemplos:

0! = 1

*a*1 = a*

$x^0 = 1$

II. outros casos são definidos aplicando-se alguma operação que inclua o caso simples na seqüência de operações necessária para resolvê-la.

Exemplos:

*n! = n * (n-1)!*

a b = a * (b-1) + a*

Observação: Nos casos acima a definição recursiva funciona para $n \geq 0$ e $b > 0$.

A seguir, serão mostrados exemplos de funções definidas recursivamente.

A função Fatorial

O fatorial de um inteiro positivo n, denotado n!, é definido como sendo o produto dos inteiros de 1 até *n*. Se $n = 0$, então *n!* é definido como 1 por convenção. Se $n \geq 1$, então $n! = nx(n-1)x(n-2) \ldots 3x2x1$.

Por exemplo, $5! = 5x4x3x2x1 = 120$. Para fazer a ligação com métodos mais clara, será usada a notação *fatorial(n)* para denotar *n!*. A função fatorial pode ser definida de uma forma que sugere uma formulação recursiva. Para entender, observamos que $fatorial(5) = 5 \times (4x3x2x1) = 5 \times fatorial(4)$.

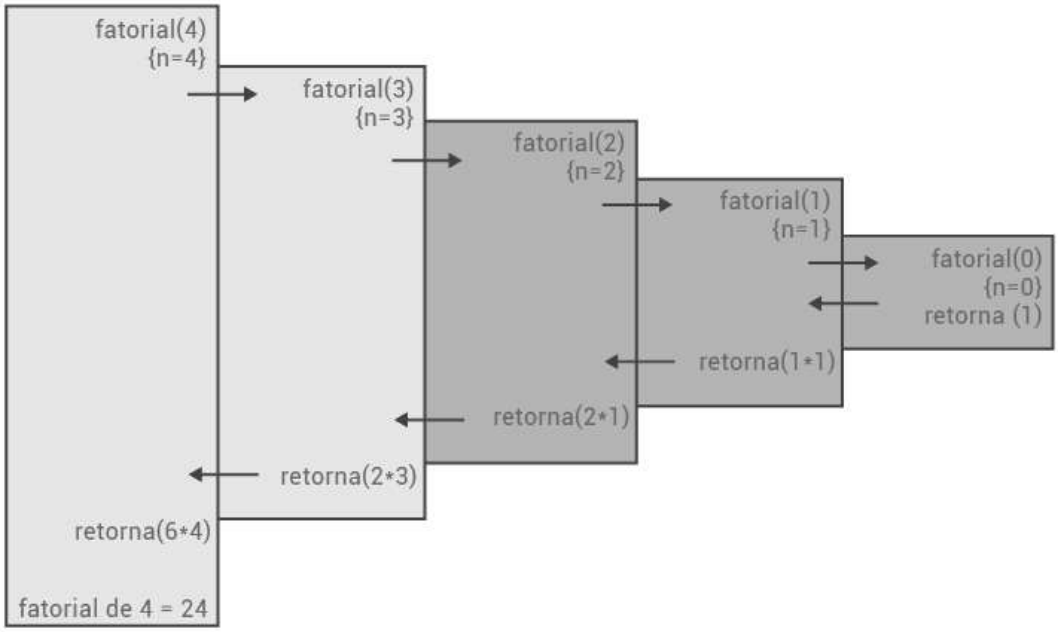
Assim, podemos definir *fatorial(5)* em termos de *fatorial(4)*. Normalmente, para um inteiro positivo *n*, podemos definir *fatorial(n)* como sendo $n \times fatorial(n-1)$. Isso leva a seguinte definição recursiva:

- *0! = 1 (parte base)*
- *n! = n * (n-1)! (parte recursiva)*

Abaixo, segue a função fatorial definida recursivamente.

```
1.  int fatorial (int n){
2.
3.          if (n==0)
4.              return (1);
5.          else return (n*fatorial (n-1));
6.
7.  }
```

Podemos ilustrar a execução da definição recursiva de uma função por meio de um rastreamento recursivo. Cada entrada do rastreamento corresponde a uma chamada recursiva. Cada nova chamada recursiva é indicada por uma seta apontando a nova funçnao ativada. Quando a função retorna, uma seta indicando este retorno é desenhada, e o valor de retorno é indicado na mesma. A Figura a seguir apresenta um exemplo de rastreamento.



Apresentar o Rastreamento recursivo da função Fatorial

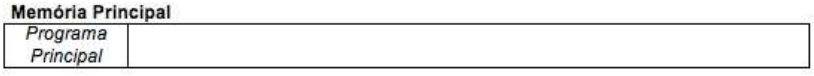
Fonte: Elaborado pelo autor

IMPORTANTE

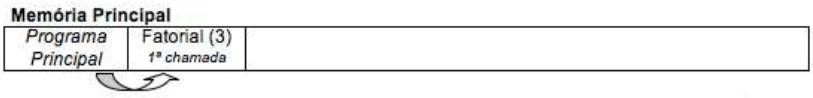
É preciso tomar cuidado ao projetar algoritmos recursivos, para garantir sua terminação. Para isso, é necessário que a chamada recursiva esteja sujeita a uma condição que em algum instante se torna *falsa*, se era inicialmente *verdadeira*, ou se torna *verdadeira*, se era inicialmente *falsa*. A detecção dessa condição deve levar a um caso básico do problema, que possa ser resolvido sem chamar novamente o mesmo procedimento. No exemplo do fatorial, a condição *n=0* indica um caso trivial, que é resolvido simplesmente retornando o valor *1*.

Na utilização de recursão, deve-se levar em conta que cada chamada (ativação) não finalizada de um procedimento (ou função) ocupa espaço na memória para armazenamento das variáveis locais, além de outras informações que serão necessárias para retomar sua execução mais adiante. Por isso, o nível mais profundo de recursão, ou seja, o número de ativações não finalizadas do procedimento, geradas por chamadas recursivas, não deve ser muito grande. O cálculo do fatorial, com recursão, chega a ter *n+1* chamadas não finalizadas da função, ocupando espaço de memória ao mesmo tempo.

A seguir, um esquema da área de memória que é utilizada para essa finalidade é mostrada, a qual pode ser vista como uma seqüência linear de posições de memória. Inicialmente, apenas o programa principal estará ocupando espaço:



Se o programa principal chama a função fatorial, por exemplo, fatorial(3), será alocada uma área de memória para armazenar as informações relativas à execução dessa (primeira) chamada da função.



A execução de *fatorial(3)*, por sua vez, irá gerar uma segunda chamada a essa função, *fatorial(2)*, e assim sucessivamente, até que seja requisitado o cálculo de *fatorial(0)*.

Terminada a execução de *fatorial(0)*, a área de memória associada àquela chamada será liberada e o resultado, igual a 1, será retornado para o cálculo de *fatorial(1)*, o qual, por sua vez, havia gerado a chamada a *fatorial(0)*. Nesse momento, a configuração da memória é:

No sentido contrário ao das chamadas, serão finalizadas, sucessivamente, as execuções de *fatorial(1)*, *fatorial(2)* e *fatorial(3)*. O resultado dessa última, finalmente, retornará para o ponto do programa principal onde o cálculo de *fatorial(3)* havia sido requisitado. A partir daí, a execução do programa principal é retomada e toda a memória que foi usada pela função fatorial estará disponível.

A seqüência de ocupação e desocupação da memória pelas sucessivas chamadas da função tem semelhança com o funcionamento de uma pilha, o que motiva a denominação de "*pilha de recursão*".

Esse consumo de memória pode chegar a ser comprometedor, apontando para a conveniência de substituir o algoritmo recursivo por uma versão não recursiva. Deve-se ressaltar que sempre é possível escrever uma versão não recursiva para um procedimento recursivo, através da utilização de iteração (repetição), embora o código resultante possa ficar mais "obscuro" ou até mesmo inviável devido à complexidade.

A repetição de cálculos idênticos compromete a eficiência do algoritmo, o que fica evidente quando a função é aplicada a números grandes, onde se observa um grande desperdício de tempo e espaço.

Para os exemplos apresentados até aqui, a conclusão é que as soluções recursivas não são recomendáveis. De um modo geral, deve-se evitar o uso de recursão sempre que existir uma solução simples, sendo que a recursão seja substituída por iteração. Por outro lado, e também como regra geral, não se deve evitar o uso de recursividade na fase inicial de projeto de um programa, pois essa estratégia pode levar à solução mais simples e clara, facilitando e dando segurança ao trabalho de programação. Posteriormente, quando já existir uma versão segura do programa, deve-se avaliar se os procedimentos e

funções recursivas podem ser substituídos por versões iterativas com vantagem. Essa substituição pode ser especialmente importante em procedimentos executados com muita freqüência.

A sequência de Fibonacci

A seqüência de Fibonacci é uma série de números inteiros. O *n*-ésimo elemento da série é dado por:

1. Fibonacci (0) = 0, se n=0
2. Fibonacci (1) = 1, se n=1
3. Fibonacci (n) = Fibonacci (n-1) + Fibonacci (n-2), se n>1

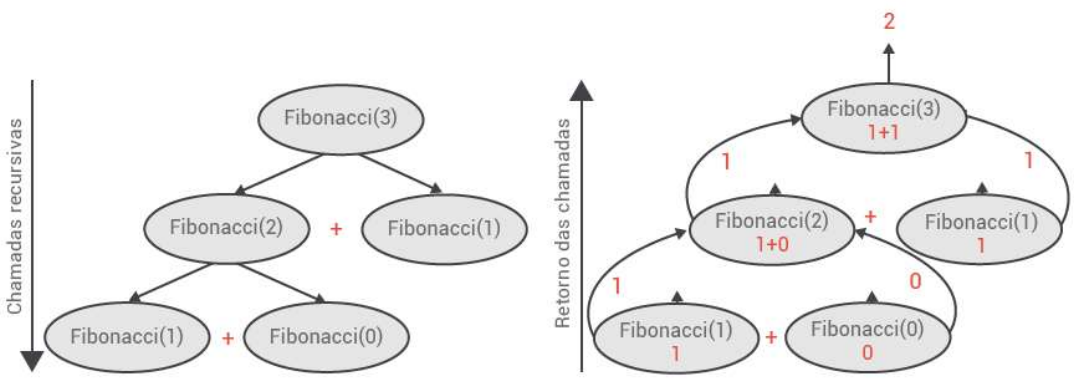
Apesar de ser evidentemente recursiva podemos perceber que, para calcular um dado elemento da seqüência, os dois termos da soma (no caso n>1), cálculos serão repetidos.

A Seqüência de Fibonacci é 0, 1, 1, 2, 3, 5, 8,...

Abaixo, segue a função Fibonacci definida recursivamente.

```
1.  inf Fibonacci (int n){
2.
3.          if (n==0)
4.               return (0);
5.          else if (n==1)
6.                     return (1);
7.                  else return ( Fibonacci(n-1) + Fibonacci(n-2) );
8.
9.  }
10.
```

Outra maneira de visualizar o comportamento de um procedimento recursivo é desenhar a **árvore de recursão**. Trata-se de uma árvore em que cada nó corresponde a uma chamada ao procedimento, tendo como filhos os nós correspondentes às chamadas geradas por ela.



Apresentar a Árvore de recursão para o Finbonacci de 3

Fonte: Do próprio autor

A Função para somar os elementos de um vetor

Supondo, por exemplo, um dado vetor v cujos n inteiros se deseja somar. Podemos resolver este problema usando recursão, observando que a soma de todos os n inteiros em v é igual a v[0], se n = 0, ou a soma dos primeiros n-1 inteiros de v mais o último elemento de v.

Se definir S*(n)* como a soma dos valores de *v* com índices de 1 a n (*v* [*1..n*]), pode-se escrever:

1. *S(0) = 0*
2. *S(n) = s(n-1) + v[n], n>0*

Abaixo, segue a função para somar os elementos de um vetor definida recursivamente.

```
1.  int soma_vetor (int v[], int n){
2.
3.      if (n==0)
4.           return (0);
5.      else return (v[n-1] +  soma_vetor (v,n-1));
6.
7.  }
```

Esquemas não apropriados à recursão

1. Quando existe uma única chamada do procedimento recursivo no fim ou no começo da rotina, o procedimento é facilmente transformado numa iteração simples. É boa política não usar recursão quando existe um algoritmo iterativo igualmente claro que resolva o problema. É o caso do fatorial e da soma de vetores vistos anteriormente.
2. Quando o uso de recursão acarreta num número maior de cálculos

- ***Use recursão quando:***

1. O problema é naturalmente recursivo (clareza) e a versão recursiva do algoritmo não gera ineficiência evidente se comparado com a versão iterativa do mesmo algoritmo.
2. O algoritmo se torna compacto sem perda de clareza ou generalidade.

3. É possível prever que o número de chamadas ou a carga sobre a pilha de passagem de parâmetros não irá causar interrupção do processo.

- *Não use recursão quando:*

1. A solução recursiva causa ineficiência se comparada com uma versão iterativa.
2. A recursão é de cauda.
3. Parâmetros consideravelmente grandes têm que ser passados por valor.
4. Não é possível prever se o número de chamadas recursivas irá causar sobrecarga da pilha de passagem de parâmetros.

Quiz

Exercício

Recursividade

INICIAR

Quiz

Exercício Final

Recursividade

INICIAR

Avalie este tópico

☆☆☆

ANTERIOR
Arquivos

Índice

Biblioteca (https://www.uninove.br/conheca-a-uninove/biblioteca/sobre-a-biblioteca/apresentacao/)
Portal Uninove (http://www.uninove.br)
Mapa do Site

Ajuda? (https://ava.uninove.br/seu/AVA/topico/topico.php?idCurso=)