

Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Segundo Semestre de 2013

Roteiro

- 1 Recursão
- 2 Fatorial
- 3 O que ocorre na memória
- 4 Recursão \times Iteração
- 5 Soma em um vetor
- 6 Números de Fibonacci
- 7 Exercícios



Recursão

- Desejamos criar um algoritmo para resolver um determinado problema.
- Usando o método de recursão/indução, a solução de um problema pode ser expressa da seguinte forma:
 - ▶ Primeiramente, definimos a solução para casos base.
 - ▶ Em seguida, definimos como resolver o problema para um caso geral, utilizando-se de soluções para instâncias menores do problema.

Indução

- *Indução*: Técnica de demonstração matemática em que algum parâmetro da proposição a ser demonstrada envolve números naturais.
- Seja T uma proposição que desejamos provar como verdadeira para todos valores naturais n .
- Ao invés de provar diretamente que T é válido para todos os valores de n , basta provar as duas condições 1 e 3 a seguir:
 - ① *Caso Base*: Provar que T é válido para $n = 1$.
 - ② *Hipótese de Indução*: Assumimos que T é válido para $n - 1$.
 - ③ *Passo de Indução*: Sabendo-se que T é válido para $n - 1$, devemos provar que T é válido para n .

Indução

- Por que a indução funciona? Por que as duas condições são suficientes?
 - ▶ Mostramos que T é válido para um caso simples como $n = 1$.
 - ▶ Com o passo da indução, mostramos que T é válido para $n = 2$.
 - ▶ Como T é válido para $n = 2$, pelo passo de indução, T também é válido para $n = 3$, e assim por diante.

Exemplo

Teorema

A soma $S(n)$ dos primeiros n números naturais é $S(n) = n(n + 1)/2$

Prova.

Base: Para $n = 1$, temos que $S(1) = n(n + 1)/2 = 1(1 + 1)/2 = 1$.

Hipótese de Indução: Vamos assumir que a fórmula é válida para $n - 1$, ou seja, $S(n - 1) = (n - 1)n/2$.

Passo: Devemos mostrar que é válido para n . Por definição, $S(n) = S(n - 1) + n$. Por hipótese $S(n - 1) = (n - 1)n/2$, logo:

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

□

Recursão

- Definições recursivas de funções operam como o *princípio matemático da indução* visto anteriormente, ou seja, a solução é inicialmente definida para os casos base e estendida para o caso geral.

Fatorial

Problema: Calcular o fatorial de um número ($N!$).

Qual é o caso base?

- $0! = 1$

Qual é o passo indutivo?

- Temos que expressar a solução para $N \geq 1$, supondo que já sabemos a solução para algum caso mais simples.
- $N! = N(N - 1)!$.

Este caso é trivial pois a própria definição do fatorial é recursiva.

Fatorial

Portanto, a solução do problema pode ser expressa da seguinte forma:

- Se $N = 0$ então $0! = 1$.
- Se $N \geq 1$ então $N! = N(N - 1)!$.

Note como aplicamos o princípio da indução:

- Sabemos a solução para um caso base: $N = 0$.
- Definimos a solução do problema geral $N!$ em termos do mesmo problema, mas para um caso mais simples.

Implementação em C

```
long int fatorial(int N) {  
    int X;  
    long int Y;  
  
    if (N == 0) /* caso base */  
        return 1;  
    else {  
        X = N - 1;  
        Y = fatorial(X);  
        return N * Y;  
    }  
}
```

Fatorial

- Para solucionar o problema, faz-se uma chamada para a própria função, por isso, esta função é chamada *recursiva*.
- Recursividade geralmente permite uma descrição mais clara e concisa dos algoritmos, especialmente quando o problema é recursivo por natureza.

O que ocorre na memória

- Devemos entender como é feito o controle sobre as variáveis locais em chamadas recursivas.
- A memória de um sistema computacional é dividida em três partes:
 - ▶ *Espaço Estático*: contém as variáveis globais e código do programa.
 - ▶ *Heap*: para alocação dinâmica de memória.
 - ▶ *Pilha*: para execução de funções.

O que ocorre na memória

- Toda vez que uma função é invocada, suas variáveis locais são armazenadas no topo da pilha.
- Quando uma função termina a sua execução, suas variáveis locais são removidas da pilha.

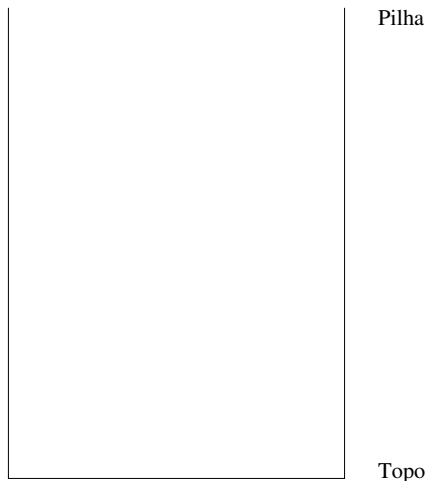
```
int f1(int a, int b) {  
    int c = 5;  
    return (c + a + b);  
}
```

```
int f2(int a, int b) {  
    int c;  
    c = f1(b, a);  
    return c;  
}
```

```
int main() {  
    f2(2, 3);  
  
    return 0;  
}
```

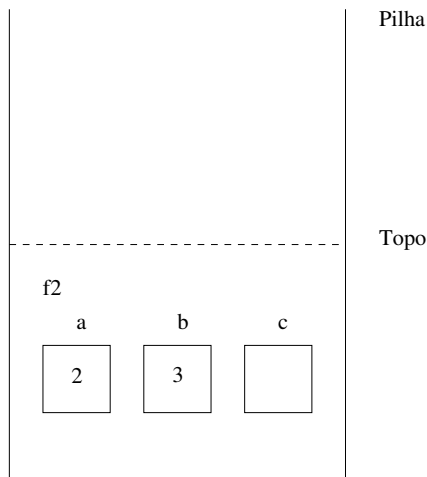
O que ocorre na memória

O programa começa a execução pela função `main`, que não aloca nenhuma variável. Sendo assim, inicialmente, a pilha está vazia.



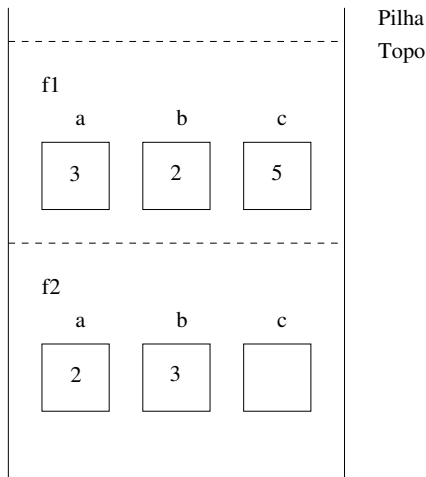
O que ocorre na memória

Quando $f2(2,3)$ é invocada, suas variáveis locais são alocadas no topo da pilha.



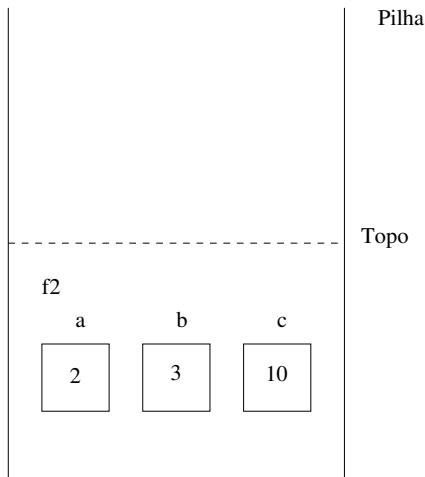
O que ocorre na memória

A função f_2 invoca a função $f_1(b, a)$ e as variáveis locais desta são alocadas no topo da pilha sobre as de f_2 .



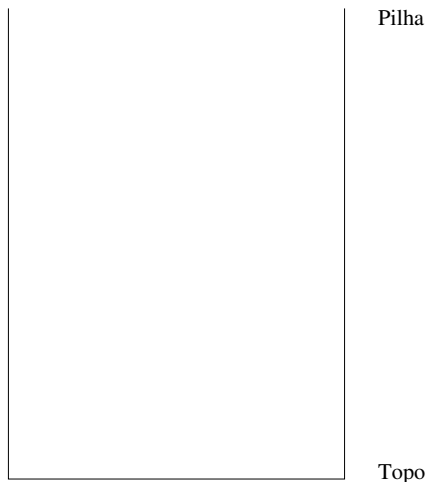
O que ocorre na memória

A função `f1` termina, devolvendo 10. As variáveis locais de `f1` são removidas da pilha.



O que ocorre na memória

Finalmente, f2 termina a sua execução devolvendo 10. Suas variáveis locais são removidas da pilha.



O que ocorre na memória

No caso de chamadas recursivas é como se cada chamada correspondesse a uma chamada de uma função distinta.

- As execuções das chamadas de funções recursivas são feitas na pilha, assim como qualquer função.
- O último conjunto de variáveis alocadas na pilha, que está no topo, corresponde às variáveis da última chamada da função.
- Quando termina a execução de uma chamada da função, as variáveis locais desta são removidas da pilha.

Usando recursão em programação

Considere novamente a solução recursiva para se calcular o fatorial e assumamos que seja feita a chamada `fatorial(4)`.

```
long int fatorial(int N) {
    int X;
    long int Y;

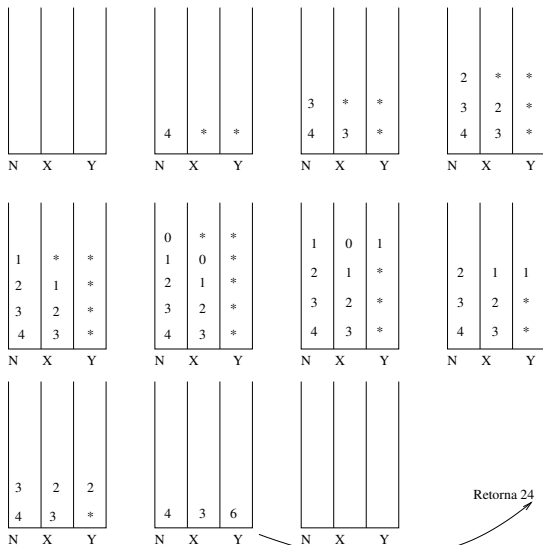
    if (N == 0) /* caso base */
        return 1;
    else {
        X = N - 1;
        Y = fatorial(X);
        return N * Y;
    }
}
```

O que ocorre na memória

- Cada chamada da função *fatorial* cria novas variáveis locais de mesmo nome (N , X e Y).
- Portanto, múltiplas variáveis (N , X e Y) podem existir em um dado momento.
- Em um dado instante, o nome N (ou X ou Y) refere-se à variável local ao corpo da função que está sendo executada naquele instante.

O que ocorre na memória

Estado da pilha de execução para fatorial(4).



O que ocorre na memória

- É claro que as variáveis X e Y são desnecessárias.

```
long int fatorial(int N) {  
  
    if (N == 0) /* caso base */  
        return 1;  
    else  
        return N * fatorial(N - 1);  
}
```


Recursão × Iteração

- Soluções recursivas são geralmente mais concisas do que as iterativas.
- Soluções iterativas em geral consomem menos memória do que as soluções recursivas.
- Cópia dos parâmetros a cada chamada recursiva é um custo adicional para as soluções recursivas.

Recursão × Iteração

Neste caso, uma solução iterativa é mais eficiente. Por quê?

```
long int fatorial(int n) {  
    long int fat = 1;  
    int i;  
  
    for (i = 1; i <= n; i++)  
        fat = fat * i;  
  
    return fat;  
}
```

Exemplo: soma de elementos de um vetor

- Suponha que temos um vetor v de inteiros de tamanho n e queiramos saber a soma de todos os seus elementos.
- Como podemos descrever este problema de forma recursiva?
- Vamos denotar por $Soma(k)$ a soma k primeiros elementos do vetor. Com isso, temos:
 - ▶ Se $n = 0$ então $Soma(0) = 0$.
 - ▶ Se $n > 0$ então $Soma(n) = Soma(n - 1) + v[n - 1]$.

Implementação em C

```
int soma(int v[], int n) {  
  
    if (n == 0)  
        return 0;  
    else  
        return soma(v, n - 1) + v[n - 1];  
}
```

Soma do vetor recursivo

- O método recursivo sempre termina pois:
 - ▶ Existe um caso base bem definido.
 - ▶ A cada chamada recursiva, usamos um valor menor de n .

Implementação em C

Neste caso, é claro que a solução iterativa também seria melhor (não há criação de variáveis por causa das chamadas recursivas):

```
int soma(int v[], int n) {  
    int soma = 0, i;  
  
    for (i = 0; i < n; i++)  
        soma = soma + v[i];  
  
    return soma;  
}
```

Fibonacci

- A série de Fibonacci é a seguinte:
 - ▶ 1, 1, 2, 3, 5, 8, 13, 21, ...
- Queremos determinar qual é o n -ésimo ($\text{Fibonacci}(n)$) número da série.
- Como descrever o n -ésimo número de Fibonacci de forma recursiva?

Fibonacci

- No caso base temos:
 - ▶ Se $n = 1$ ou $n = 2$ então $\text{Fibonacci}(n) = 1$.
- Conhecendo casos anteriores, podemos computar $\text{Fibonacci}(n)$ como:
 - ▶ $\text{Fibonacci}(n) = \text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2)$.

Implementação em C (versão recursiva)

```
unsigned long int Fibonacci(int n) {  
  
    if (n <= 2)  
        return 1;  
    else  
        return Fibonacci(n - 1) + Fibonacci(n - 2);  
}
```

Implementação em C (versão iterativa)

```
unsigned long int Fibonacci(int n) {
    unsigned long int anterior, atual, temp;
    int i;

    anterior = 1;
    atual = 1;

    for (i = 3; i <= n; i++) {
        temp = atual;
        atual = atual + anterior;
        anterior = temp;
    }

    return atual;
}
```

Fibonacci

- Quantas somas são necessárias para calcular $\text{Fibonacci}(n)$, para $n > 2$, usando uma estratégia iterativa?
 - ▶ São necessárias $n - 2$ somas.
- Quantas somas são realizadas para calcular $\text{Fibonacci}(n)$, para $n > 2$, usando a função recursiva vista anteriormente?
 - ▶ São realizadas $\text{Fibonacci}(n) - 1$ somas.
 - ▶ $\text{Fibonacci}(3)$: 1 soma.
 - ▶ $\text{Fibonacci}(4)$: 2 somas.
 - ▶ $\text{Fibonacci}(5)$: 4 somas.
 - ▶ $\text{Fibonacci}(6)$: 7 somas.
 - ▶ $\text{Fibonacci}(7)$: 12 somas.
 - ▶ $\text{Fibonacci}(8)$: 20 somas.
 - ▶ $\text{Fibonacci}(9)$: 33 somas.
 - ▶ $\text{Fibonacci}(10)$: 54 somas.
 - ▶ $\text{Fibonacci}(20)$: 6.765 somas.
 - ▶ $\text{Fibonacci}(30)$: 832.040 somas.
 - ▶ $\text{Fibonacci}(40)$: 102.334.155 somas.
 - ▶ $\text{Fibonacci}(50)$: 12.586.269.025 somas.

Fibonacci

- Enquanto a versão iterativa, para obter o valor de $\text{Fibonacci}(n)$, calcula uma única vez todos os valores de $\text{Fibonacci}(k)$, para $1 \leq k \leq n$, a versão recursiva calcula várias vezes.
- Por exemplo, para $n = 20$, o número de vezes que a função recursiva calcula $\text{Fibonacci}(k)$, para $1 \leq k \leq n$:

<code>Fibonacci(1): 2584</code>	<code>Fibonacci(11): 55</code>
<code>Fibonacci(2): 4181</code>	<code>Fibonacci(12): 34</code>
<code>Fibonacci(3): 2584</code>	<code>Fibonacci(13): 21</code>
<code>Fibonacci(4): 1597</code>	<code>Fibonacci(14): 13</code>
<code>Fibonacci(5): 987</code>	<code>Fibonacci(15): 8</code>
<code>Fibonacci(6): 610</code>	<code>Fibonacci(16): 5</code>
<code>Fibonacci(7): 377</code>	<code>Fibonacci(17): 3</code>
<code>Fibonacci(8): 233</code>	<code>Fibonacci(18): 2</code>
<code>Fibonacci(9): 144</code>	<code>Fibonacci(19): 1</code>
<code>Fibonacci(10): 89</code>	<code>Fibonacci(20): 1</code>

- Para calcular o valor de $\text{Fibonacci}(n)$, a função recursiva calcula o valor de $\text{Fibonacci}(k)$, para $2 \leq k \leq n$, $\text{Fibonacci}(n-k+1)$ vezes.

Relembrando

- Recursão é uma técnica para se criar algoritmos em que:
 - ① Devemos descrever soluções para casos base.
 - ② Assumindo a existência de soluções para casos mais simples, mostramos como obter solução para o caso mais complexo.
- Algoritmos recursivos geralmente são mais claros e concisos.
- Deve-se avaliar a clareza de código \times eficiência do algoritmo.

Exercício

O que será impresso pelo programa abaixo?

```
#include <stdio.h>

void imprime(int v[], int i, int n);

int main() {
    int vet[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    imprime(vet, 0, 10);
    printf("\n");
}

void imprime(int v[], int i, int n) {
    if (i < n) {
        printf("%d ", v[i]);
        imprime(v, i + 1, n);
    }
}
```

Saída:

1 2 3 4 5 6 7 8 9 10

Exercício

O que será impresso pelo programa abaixo?

```
#include <stdio.h>

void imprime(int v[], int i, int n);

int main() {
    int vet[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    imprime(vet, 0, 10);
    printf("\n");
}

void imprime(int v[], int i, int n) {
    if (i < n) {
        imprime(v, i + 1, n);
        printf("%d ", v[i]);
    }
}
```

Saída:

10 9 8 7 6 5 4 3 2 1