MC102 – Aula 25
Recursão I

Instituto de Computação – Unicamp

12 de Junho de 2012

Roteiro

Solução Iterativa

- Considere o problema de calcular a soma dos números inteiros no intervalo $[m, n]$, onde $m \leq n$. Para m $= 2$ e n $= 15$ temos:

```
2 + 3 + 4 + 5 + ... + 14 + 15
```

```
m + (m + 1) + (m + 2) + (m + 3) + ... + (n - 1) + n
```

Solução Iterativa

- Considere o problema de calcular a soma dos números inteiros no intervalo $[m, n]$, onde $m \leq n$.

```
m + (m + 1) + (m + 2) + (m + 3) + ... + (n - 1) + n
```

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

Veja o exemplo em `soma_iterativa.c`.

Solução Recursiva

A solução de um problema é dita recursiva quando ela é escrita em função de si própria para instâncias menores do mesmo problema.

Solução Recursiva

Exemplos:

Recursão Crescente

$Soma(m, n) = m + Soma(m + 1, n)$

.

Recursão Decrescente

$Soma(m, n) = Soma(m, n - 1) + n$

- A solução recursiva está intimamente ligada ao conceito de indução matemática.
- A indução matemática fraca, por exemplo, usa como hipótese que a solução de um problema de tamanho t pode ser obtida a partir de sua solução de tamanho $t - 1$.

Componentes da Recursão

Toda recursão é composta por

- Um caso base, uma instância do problema que pode ser solucionada facilmente. Por exemplo, é trivial fazer a soma de uma lista com um único elemento.
- Uma ou mais chamadas recursivas, onde o objeto define-se em termos de sí próprio, tentando convergir para o caso base. A soma de uma lista de n elementos pode ser definida a partir da lista da soma de $n - 1$ elementos.

Solução Recursiva

Recursão Crescente

$Soma(m, n) = m + Soma(m + 1, n)$

Solução Crescente

```
int soma(int m, int n) {

 if (m == n)
  return (n);
 else
  return (m + soma(m + 1, n));
}
```

Veja o exemplo em `soma_rec_crescente.c`.

Solução Recursiva

Recursão Decrescente

$Soma(m, n) = Soma(m, n-1) + n$

Solução Decrescente

```
int soma(int m, int n) {

 if (m == n)
  return (m);
 else
  return (soma(m, n-1) + n);
}
```

Veja o exemplo em `soma_rec_decrescente.c`.

Solução Recursiva

- Para solucionar o problema, é feita 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.

Solução Recursiva

- Para solucionar o problema, é feita 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.

Exemplo: Fatorial

A solução do problema pode ser expressa da seguinte forma:

- Se $n = 1$ então $n! = 1$.
- Se $n > 1$ então $n! = n * (n - 1)!$.

Note como aplicamos o princípio da indução:

- Sabemos a solução para um caso base: $n = 1$.
- Definimos a solução do problema geral $n!$ em termos do mesmo problema só que para um caso menor $((n - 1)!)$.

Fatorial em C

```
long fatr(long n){
    long x,r;
    if(n == 1) //Passo Básico
      return 1;
    else{
      x = n-1;
      r = fatr(x); //Sabendo o fatorial de (n-1)
      return (n* r);  //calculamos o fatorial de n
    }
}
```

O que acontece na memória

- Precisamos 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 acontece na memória

O que acontece na pilha:

- 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.

Considere o exemplo:

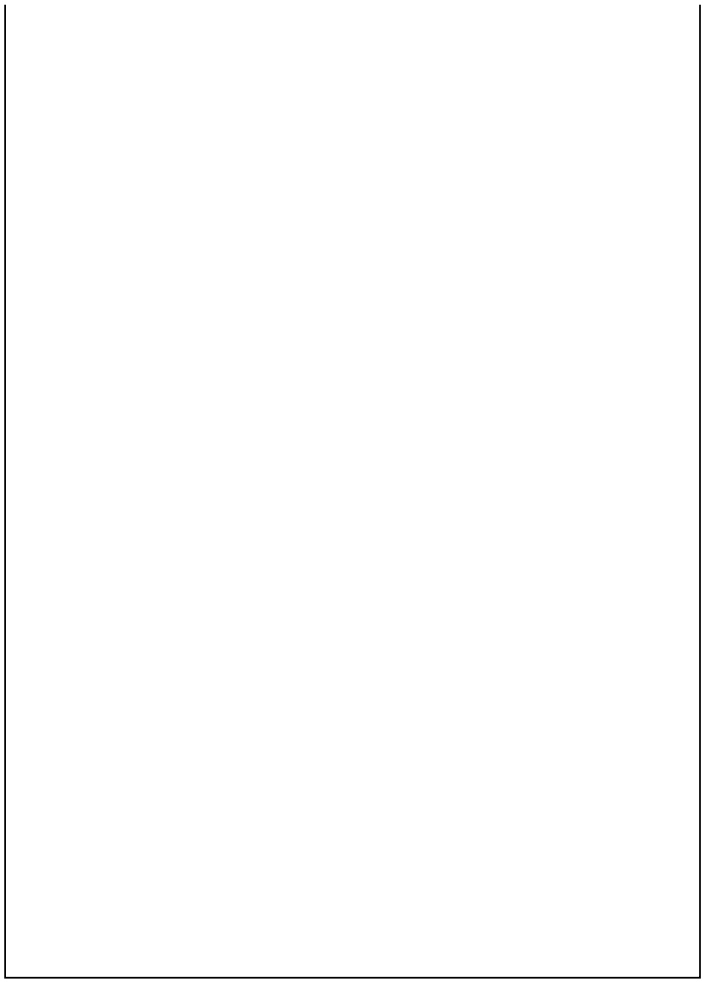
```
int f1(int a, int b){
   int c=5;
   return (c+a+b);
}

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

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

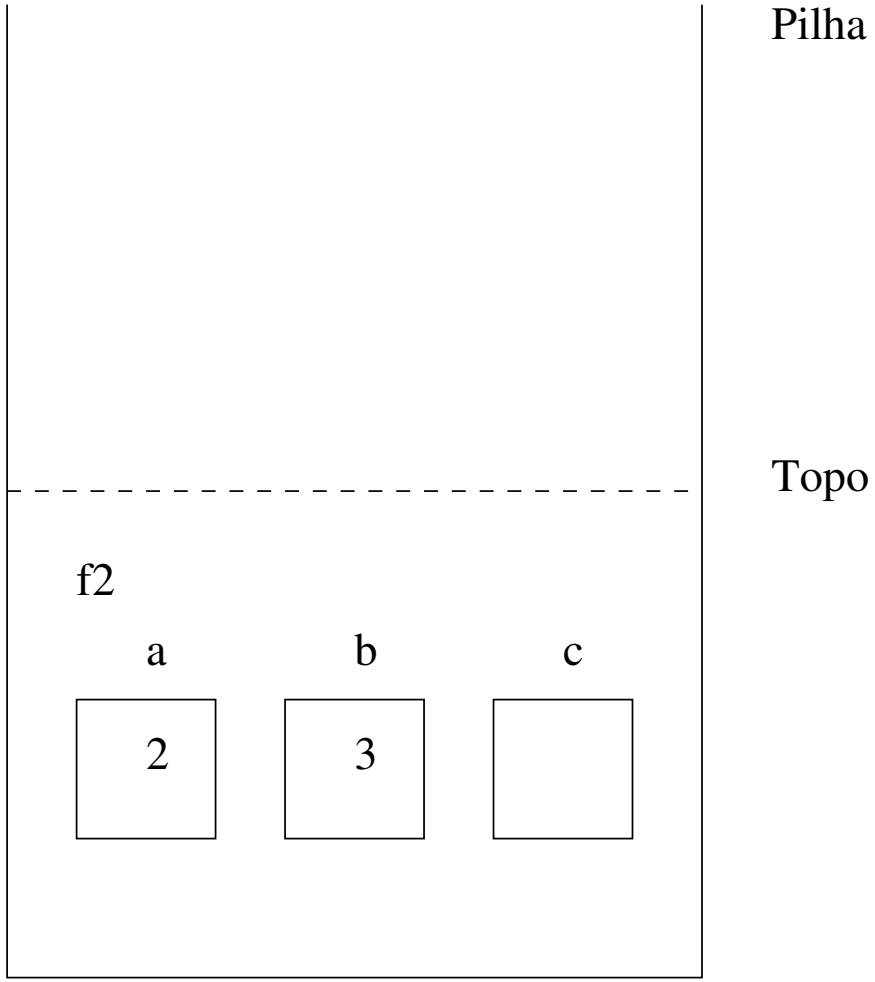
O que acontece na memória

Inicialmente a pilha está vazia.



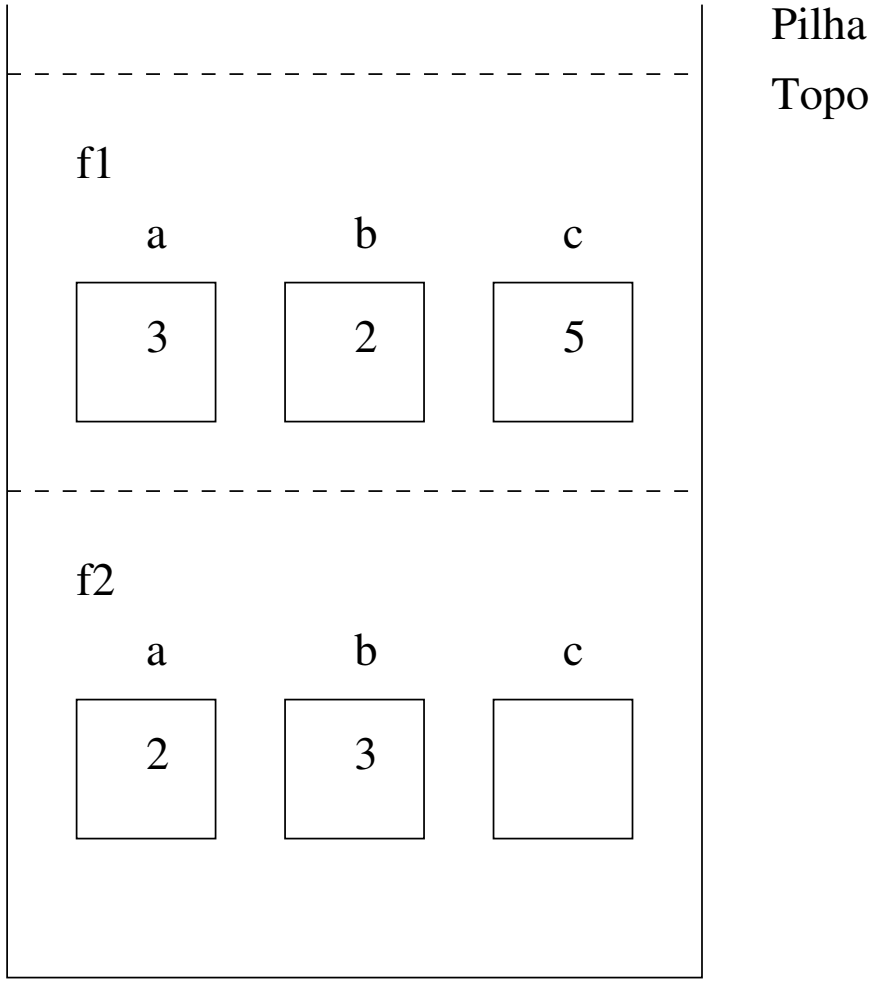
O que acontece na memória

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



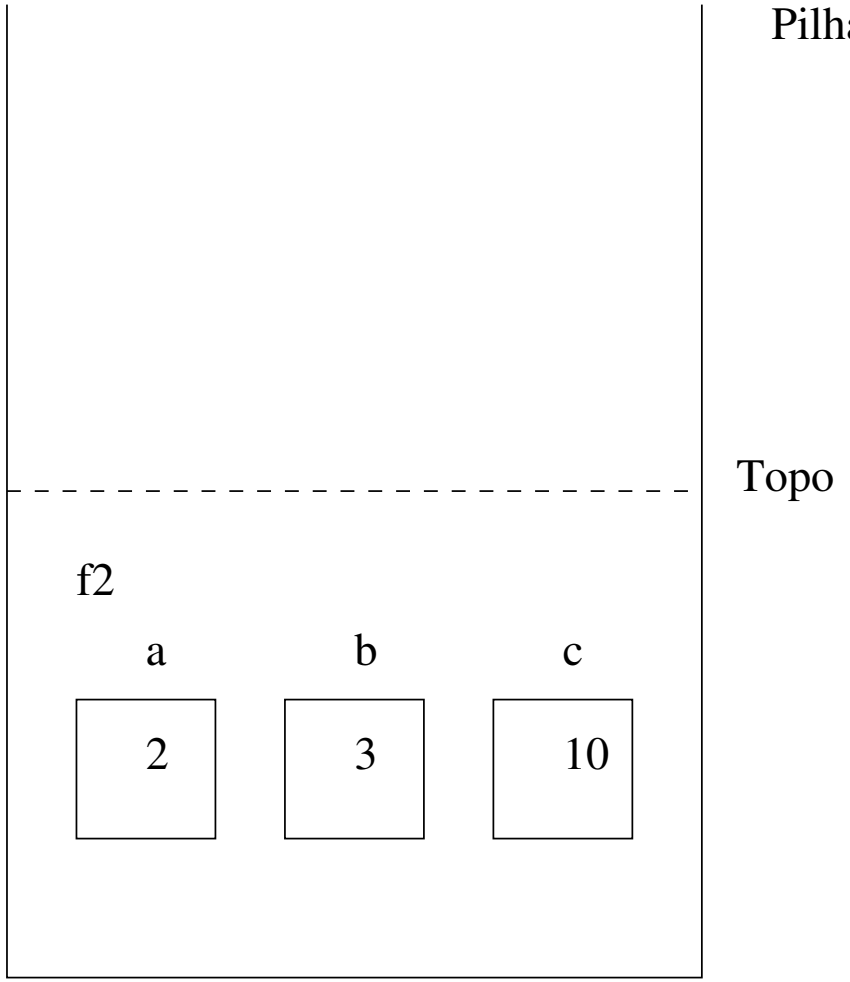
O que acontece na memória

A função **f2** invoca a função **f1(b,a)** e as variáveis locais desta são alocadas no topo da pilha sobre as de **f2**.



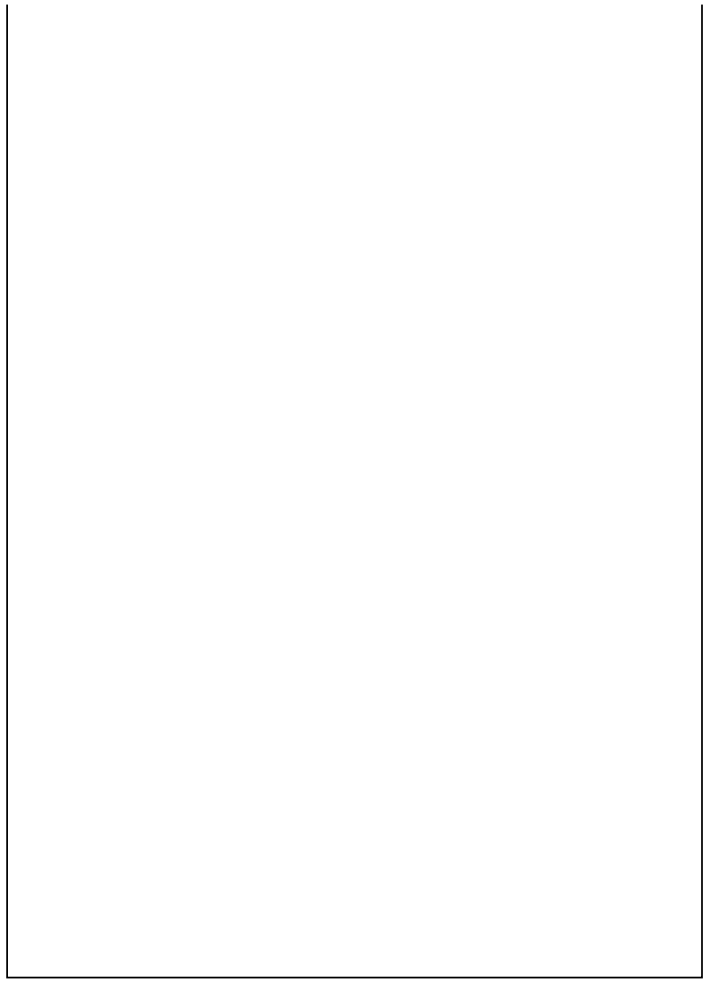
O que acontece na memória

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



O que acontece na memória

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



O que acontece na memória

No caso de chamadas recursivas para uma mesma função, é como se cada chamada correspondesse a 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 assuma que seja feito a chamada **fatr(4)**.

```
long fatr(long n){
    long x,r;
    if(n == 1) //Passo Básico
      return 1;
    else{
      x = n-1;
      r = fatr(x); //Sabendo o fatorial de (n-1)
      return (n* r);  //calculamos o fatorial de n
    }
}
```

O que acontece na memória

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

O que acontece na memória

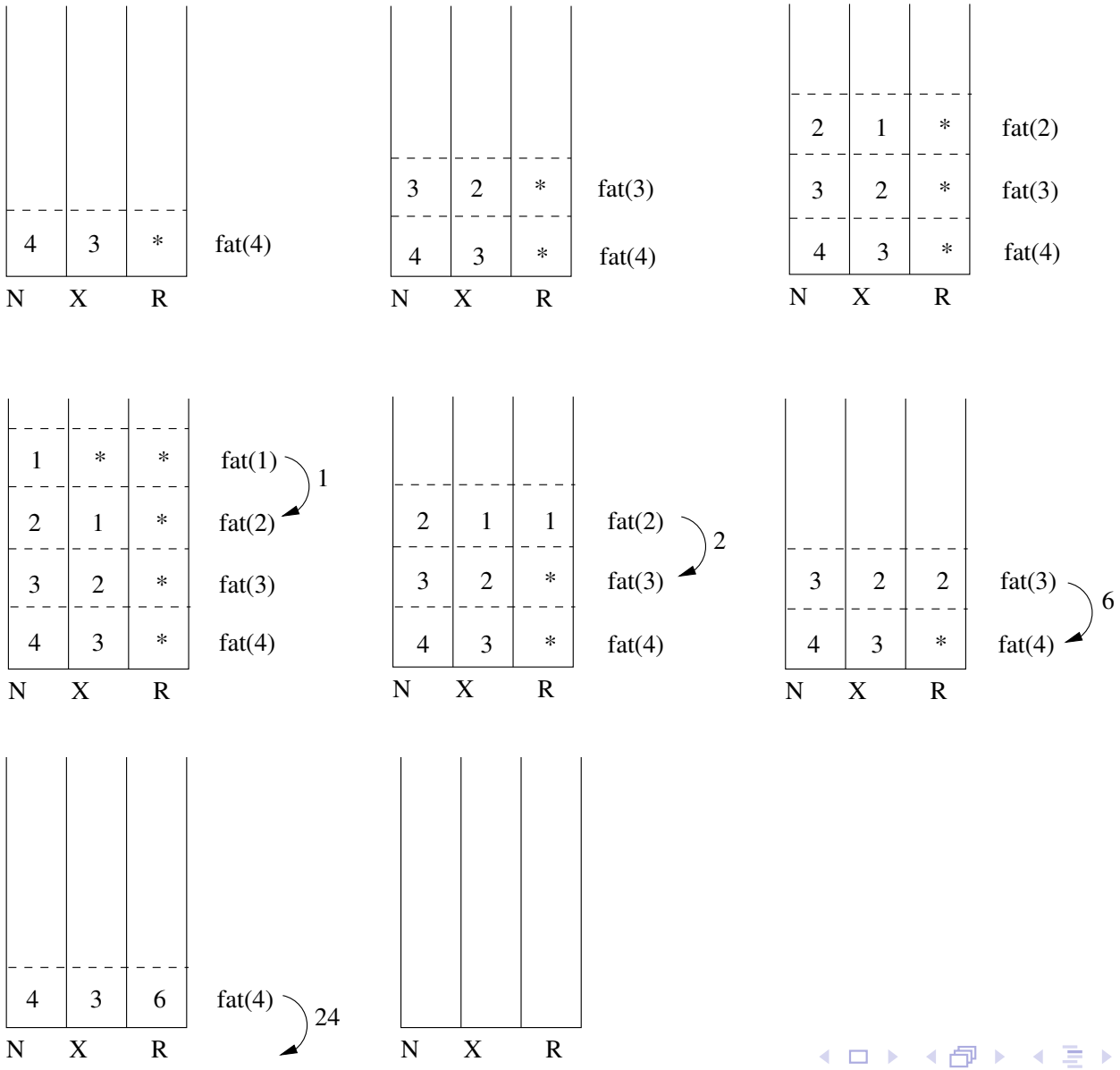
- Cada chamada da função *fatr* cria novas variáveis locais de mesmo nome (n, x, r) .
- Portanto, várias variáveis n, x e r podem existir em um dado momento.
- Em um dado instante, o nome n (ou x ou r) refere-se à variável local ao corpo da função que está sendo executada naquele instante.

O que acontece na memória

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

O que acontece na memória

Estado da Pilha de execução para *fatr(4)*.



O que acontece na memória

- É claro que as variáveis x e r são desnecessárias.
- Você também deveria testar se n não é negativo!

```
long fatr (long n){
    if(n <= 1) //Passo Básico
      return 1;
    else  //Sabendo o fatorial de (n-1)
          //calculamos o fatorial de n
      return (n* fatr(n-1));
}
```

Recursão × Iteração

- Soluções recursivas são geralmente mais concisas que as iterativas. Programas mais simples.
- Soluções iterativas em geral têm a memória limitada enquanto as recursivas, não.
- 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 fat(long n)
{
   long r = 1;

   for(int i = 1; i <= n; i++)
      r = r * i;

   return r;
}
```

Exemplo: Soma de elementos de um vetor

- Suponha que temos um vetor v de inteiros de tamanho $n+1$, e queiramos saber a soma dos seus elementos da posição 0 até n.
- Como podemos descrever este problema de forma recursiva? Isto é, como podemos descrever este problema em função de si mesmo?
- Vamos denotar por $S(n)$ a soma dos elementos das posições 0 até n do vetor. Com isso temos:
  - Se $n = 0$ então a soma é igual a $v[0]$.
  - Se $n > 0$ então a soma é igual a $v[n] + S(n-1)$.

Exemplo: Soma de elementos de um vetor

- Suponha que temos um vetor v de inteiros de tamanho $n+1$, e queiramos saber a soma dos seus elementos da posição 0 até n.
- Como podemos descrever este problema de forma recursiva? Isto é, como podemos descrever este problema em função de si mesmo?
- Vamos denotar por $S(n)$ a soma dos elementos das posições 0 até n do vetor. Com isso temos:
  - Se $n = 0$ então a soma é igual a $v[0]$.
  - Se $n > 0$ então a soma é igual a $v[n] + S(n-1)$.

Exemplo: Soma de elementos de um vetor

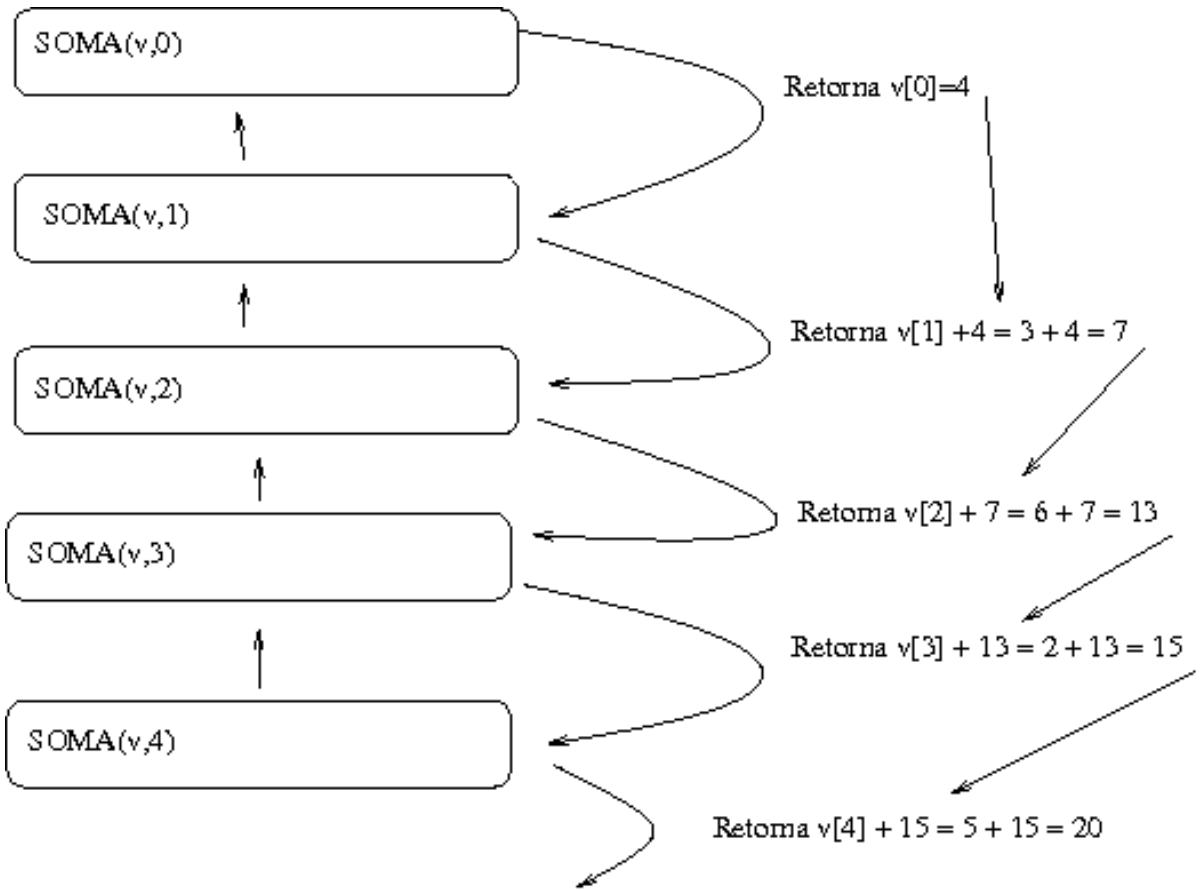
- Suponha que temos um vetor v de inteiros de tamanho $n+1$, e queiramos saber a soma dos seus elementos da posição 0 até n.
- Como podemos descrever este problema de forma recursiva? Isto é, como podemos descrever este problema em função de si mesmo?
- Vamos denotar por $S(n)$ a soma dos elementos das posições 0 até n do vetor. Com isso temos:
  - Se $n = 0$ então a soma é igual a $v[0]$.
  - Se $n > 0$ então a soma é igual a $v[n] + S(n-1)$.

Algoritmo em C

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

Exemplo de execução

É claro que o você deve se certificar de usar valores válidos de n.
$V = (4, 3, 6, 2, 5)$



Algoritmo em C

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

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

Cálculo de Potências

Suponha que queiramos calcular x^n para n inteiro positivo. Como calcular de forma recursiva?
x^n é:

- 1 se $n = 0$.
- xx^{n-1} caso contrário.

Cálculo de Potências

Suponha que queiramos calcular x^n para n inteiro positivo. Como calcular de forma recursiva?
x^n é:

- 1 se $n = 0$.
- xx^{n-1} caso contrário.

Cálculo de Potências

```
long pot(long x, long n){
  if(n == 0)
    return 1;
  else
    return x*pot(x,n-1);
}
```

Cálculo de Potências

Neste caso a solução iterativa é mais eficiente.

```
long pot(long x, long n){
    long p = 1, i;
    for( i=1; i<=n; i++)
       p = p * x;
    return p;
}
```

- O laço é executado n vezes.
- Na solução recursiva são feitas n chamadas, mas tem-se o custo adicional para criação/remoção de variáveis locais na pilha.

Cálculo de Potências

Mas e se definirmos a potência de forma diferente:
x^n é:

- Caso básico:
  - Se $n = 0$ então $x^n = 1$.
- Caso Geral:
  - Se $n > 0$ e é par, então $x^n = (x^{n/2})^2$.
  - Se $n > 0$ e é ímpar, então $x^n = x(x^{(n-1)/2})^2$.

Note como no caso geral definimos a solução do caso maior em termos de casos menores.

Cálculo de Potências

Mas e se definirmos a potência de forma diferente:
x^n é:

- Caso básico:
  - Se $n = 0$ então $x^n = 1$.
- Caso Geral:
  - Se $n > 0$ e é par, então $x^n = (x^{n/2})^2$.
  - Se $n > 0$ e é ímpar, então $x^n = x(x^{(n-1)/2})^2$.

Note como no caso geral definimos a solução do caso maior em termos de casos menores.

Cálculo de Potências

Este algoritmo é mais eficiente do que o iterativo. Por que? Quantas chamadas recursivas o algoritmo pode fazer?

```
long pot(long x, long n){
    if(n == 0)
       return 1;

    else if(n%2 == 0){ //se n é par
      double aux = pot(x, n/2);
      return aux * aux;
    }

    else{ //se n é impar
      double aux = pot(x, (n-1)/2);
      return x*aux*aux;
    }
}
```

Cálculo de Potências

Este algoritmo é mais eficiente do que o iterativo. Por que? Quantas chamadas recursivas o algoritmo pode fazer?

```
long pot(long x, long n){
    if(n == 0)
       return 1;

    else if(n%2 == 0){ //se n é par
      double aux = pot(x, n/2);
      return aux * aux;
    }

    else{ //se n é impar
      double aux = pot(x, (n-1)/2);
      return x*aux*aux;
    }
}
```

Cálculo de Potências

Este algoritmo é mais eficiente do que o iterativo. Por que? Quantas chamadas recursivas o algoritmo pode fazer?

```
long pot(long x, long n){
    if(n == 0)
       return 1;

    else if(n%2 == 0){ //se n é par
      double aux = pot(x, n/2);
      return aux * aux;
    }

    else{ //se n é impar
      double aux = pot(x, (n-1)/2);
      return x*aux*aux;
    }
}
```

Cálculo de Potências

- No algoritmo anterior a cada chamada recursiva, o valor de n é dividido por 2. Ou seja, a cada chamada recursiva, o valor de n decai para pelo menos a metade.
- Usando divisões inteiras faremos no máximo $\lceil(\log_2 n)\rceil + 1$ chamadas recursivas.
- Enquanto isso, o algoritmo iterativo executa o laço n vezes.

Recursão com várias chamadas

- Não há necessidade da função recursiva ter apenas uma chamada para si própria.
- A função pode fazer várias chamadas para si própria.
- A função pode ainda fazer chamadas recursivas indiretas. Neste caso a função 1, por exemplo, chama uma outra função 2 que por sua vez chama a função 1.

Fibonacci

- A série de fibonacci é a seguinte:
  - $1, 1, 2, 3, 5, 8, 13, 21, \ldots$.
- Queremos determinar qual é o n-ésimo (fibo(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 $\text{fibo}(n) = 1$.
- Sabendo casos anteriores podemos computar $\text{fibo}(n)$ como:
  - $\text{fibo}(n) = \text{fibo}(n-1) + \text{fibo}(n-2)$.

Algoritmo em C

A definição anterior é traduzida diretamente em um algoritmo em C:

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

Relembrando

- Recursão é uma técnica para se criar algoritmos onde:
  1. Devemos descrever soluções para casos básicos.
  2. Assumindo a existência de soluções para casos menores, mostramos como obter solução para o caso maior.
- Algoritmos recursivos geralmente são mais claros e concisos.
- Implementador deve avaliar clareza de código $\times$ eficiência do algoritmo.

Exercício

Mostre a execução da função recursiva **imprime** abaixo:
O que será impresso?

```
#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, 9);
  printf("\n");
}



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

}
```

Exercício

- Mostre o estado da pilha de memória durante a execução da função **fibo** com a chamada **fib(5)**.
- Qual versão é mais eficiente para se calcular o *n*-ésimo número de fibonacci? A recursiva ou iterativa?

Exercício

- Defina de forma recursiva a busca binária.
- Escreva um algoritmo recursivo para a busca binária.