MC102 – Aula 26
Recursão

Instituto de Computação – Unicamp

1 de Junho de 2012

Roteiro

1. Recursão – Indução
2. Recursão
3. Fatorial
4. O que acontece na memória
5. Recursão × Iteração
6. Soma em um Vetor
7. Números de fibonacci

Recursão – Indução

- Devemos criar uma 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 básicos;
  - Em seguida, definimos como resolver o problema para um caso geral, utilizando-se de soluções para instâncias menores do problema.

Recursão – Indução

- Devemos criar uma 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 básicos;
  - Em seguida, definimos como resolver o problema para um caso geral, utilizando-se de soluções para instâncias menores do problema.

Recursão – Indução

- Devemos criar uma 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 básicos;
  - 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 onde 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. **Passo 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 que T é válido para $n - 1$ devemos PROVAR que T é válido para n.

Indução

- **Indução:** Técnica de demonstração matemática onde 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. **Passo 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 que T é válido para $n - 1$ devemos PROVAR que T é válido para n.

Indução

- **Indução:** Técnica de demonstração matemática onde 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. **Passo 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 que T é válido para $n - 1$ devemos PROVAR que T é válido para n.

Indução

- **Indução:** Técnica de demonstração matemática onde 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. **Passo 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 que T é válido para $n - 1$ devemos PROVAR que T é válido para n.

Indução

- **Indução:** Técnica de demonstração matemática onde 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. **Passo 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 que T é válido para $n - 1$ devemos PROVAR que T é válido para n.

Indução

- **Indução:** Técnica de demonstração matemática onde 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. **Passo 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 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 valido 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 $n(n+1)/2$

Prova.

Base: Para $n = 1$ devemos mostrar que $n(n+1)/2 = 1$, o que é claramente verdade.

Hip. de Indução: Vamos assumir que é válido para um valor n, ou seja, $S(n) = n(n+1)/2$.

Exemplo

Teorema

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

Prova.

Base: Para $n = 1$ devemos mostrar que $n(n+1)/2 = 1$, o que é claramente verdade.

Hip. de Indução: Vamos assumir que é válido para um valor n, ou seja, $S(n) = n(n+1)/2$.

□

Exemplo

Prova.

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

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

□

Recursão

- Definições recursivas de funções funcionam como o *princípio matemático da indução* que vimos anteriormente.
- A idéia é que a solução de um problema pode ser expressa da seguinte forma:
  - Definimos a solução para os casos básicos;
  - Definimos como resolver o problema geral utilizando soluções do mesmo problema só que para casos menores.

Fatorial

Problema: Calcular o fatorial de um número ($n!$).
Qual o caso base e o passo da indução?

- Se n é igual a 1, então o fatorial é 1.

Qual seria o passo indutivo?

- Temos que expressar a solução para $n > 1$, 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

Problema: Calcular o fatorial de um número ($n!$).
Qual o caso base e o passo da indução?

- Se n é igual a 1, então o fatorial é 1.

Qual seria o passo indutivo?

- Temos que expressar a solução para $n > 1$, 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

Problema: Calcular o fatorial de um número ($n!$).
Qual o caso base e o passo da indução?

- Se n é igual a 1, então o fatorial é 1.

Qual seria o passo indutivo?

- Temos que expressar a solução para $n > 1$, 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 $n = 1$ 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
    }
}
```

Fatorial

- 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.

Fatorial

- 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.

O que acontece na memória

- Precisamos entender como é feito o controle sobre os dados 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, seus dados locais são armazenados no topo da pilha.
- Quando uma função termina a sua execução, seus dados locais são removidos da pilha.

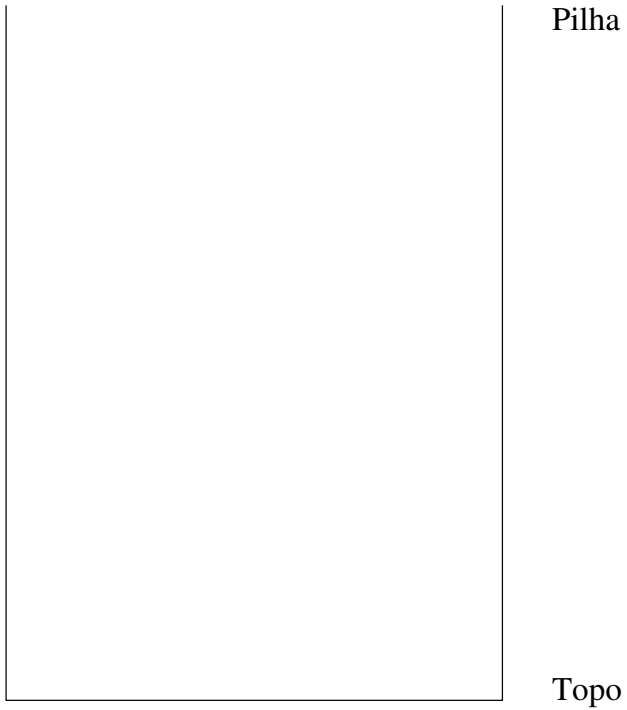
Considere o exemplo:

```
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;
}
```

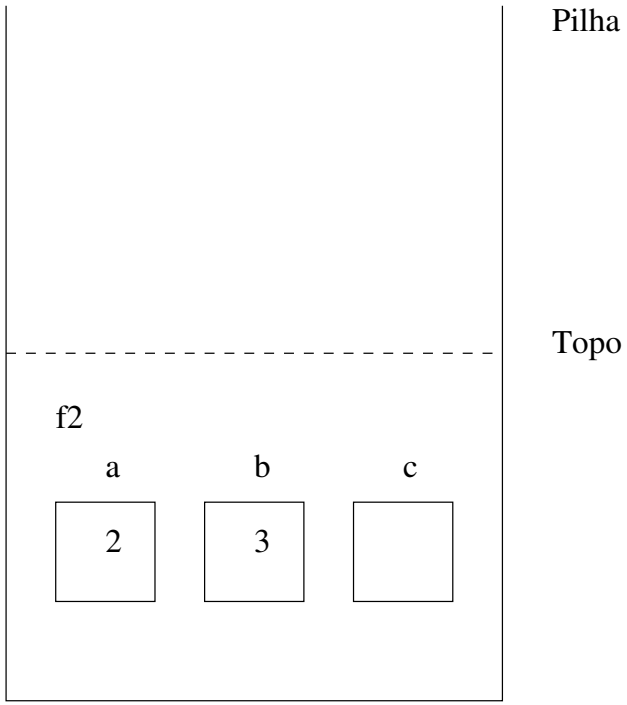
O que acontece na memória

Inicialmente a pilha está vazia.



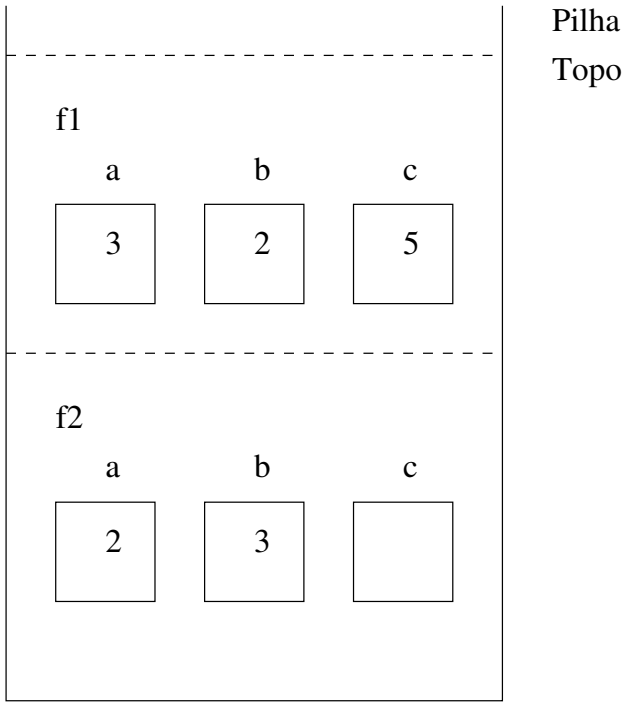
O que acontece na memória

Quando **f2(2,3)** é invocada, seus dados locais são alocados no topo da pilha.



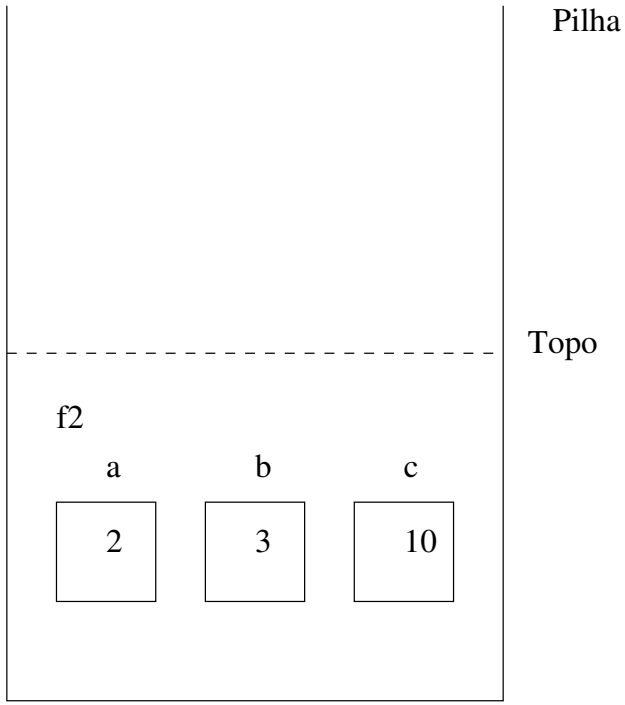
O que acontece na memória

A função **f2** invoca a função **f1(b,a)** e os dados locais desta são alocados no topo da pilha sobre os de **f2**.



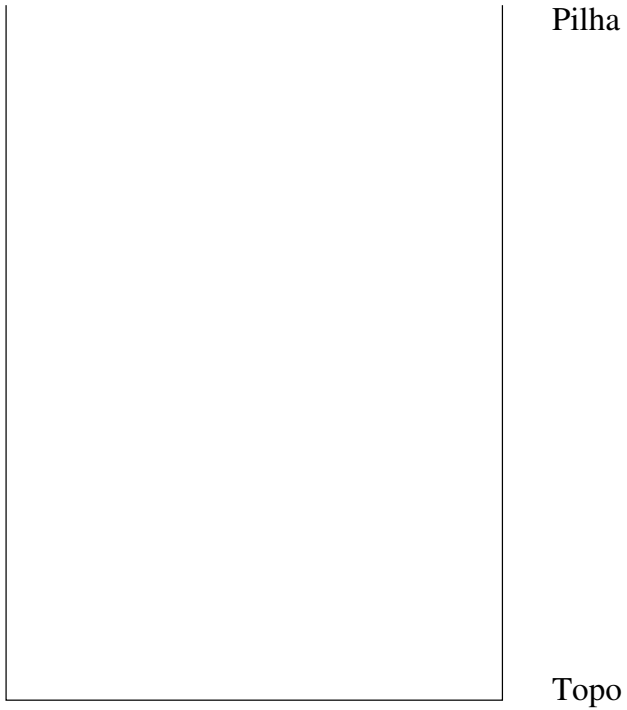
O que acontece na memória

A função **f1** termina, devolvendo 10. Os dados locais de **f1** são removidos da pilha.



O que acontece na memória

Finalmente **f2** termina a sua execução devolvendo 10. Seus dados locais são removidos 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 dados locais alocados na pilha, que está no topo, corresponde aos dados da última chamada da função.
- Quando termina a execução de uma chamada da função, os dados locais desta são removidos 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 novos dados locais de mesmo nome (n, x, r) .
- Portanto, vários dados (variáveis e parâmetros) n, x e r podem existir em um dado momento.
- Em um dado instante, o nome n (ou x ou r) refere-se ao dado local da função que está sendo executada naquele instante.

O que acontece na memória

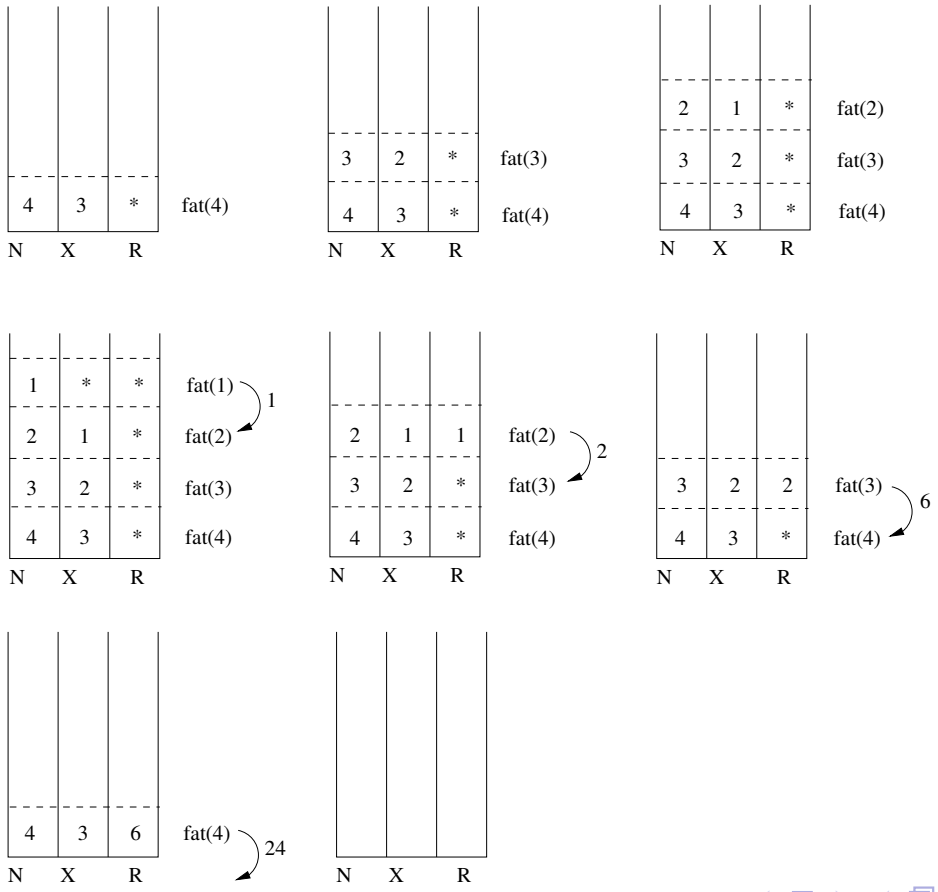
- Cada chamada da função *fatr* cria novos dados locais de mesmo nome (n, x, r) .
- Portanto, vários dados (variáveis e parâmetros) n, x e r podem existir em um dado momento.
- Em um dado instante, o nome n (ou x ou r) refere-se ao dado local da função que está sendo executada naquele instante.

O que acontece na memória

- Cada chamada da função *fatr* cria novos dados locais de mesmo nome (n, x, r) .
- Portanto, vários dados (variáveis e parâmetros) n, x e r podem existir em um dado momento.
- Em um dado instante, o nome n (ou x ou r) refere-se ao dado local 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

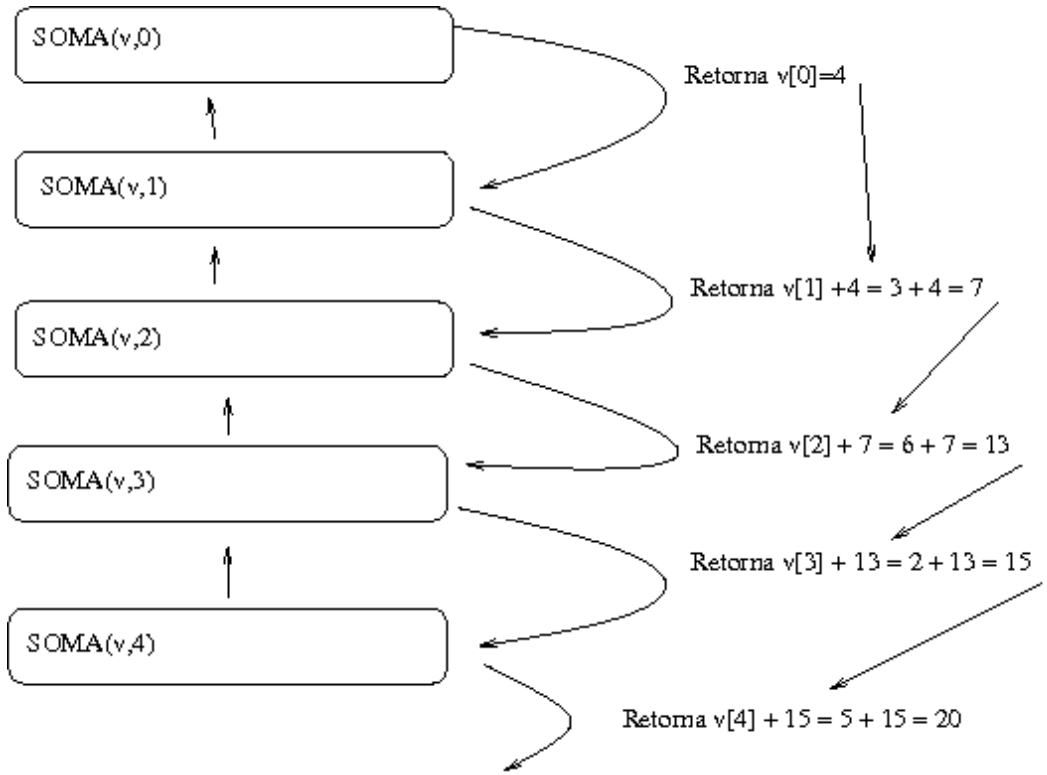
- Suponha que temos um vetor v de inteiros de tamanho $n+1$, e queiramos saber a soma dos seus elementos da posição 0 até n.
- Como podemos descrever este problema de forma recursiva? Isto é, como podemos descrever este problema em função de si mesmo?
- Vamos denotar por $S(n)$ a soma dos elementos das 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.
$V = (4, 3, 6, 2, 5)$



Soma do vetor recursivo

- O método recursivo sempre termina:
  - Existência de um caso base.
  - A cada chamada recursiva do método temos um valor menor de n.

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;
}
```

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 fibo$(n) = 1$.
- Sabendo casos anteriores podemos computar fibo(n) como:
  - fibo$(n) =$ fibo$(n-1) +$ 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 × eficiência do algoritmo.