SCC0201 – ICC2

Recursão

Definição

- Uma função é dita *recursiva* quando é definida em seus próprios termos, direta ou indiretamente
  - Dicionário Michaelis: *ato ou efeito de recorrer*
    - Recorrer: *Correr de novo por; tornar a percorrer. Repassar na memória; evocar.*
- É uma função como qualquer outra

Recursividade

- Utiliza-se uma função que permite chamar a si mesma (direta ou indiretamente)
- Exemplo: soma dos primeiros N inteiros
  - S(5) = 1 + 2 + 3 + 4 + 5
  - = 5 + S(4)
  - S(4) = 4 + S(3)
  - S(3) = 3 + S(2)
  - S(2) = 2 + S(1)
  - S(1) = 1 (solução trivial)

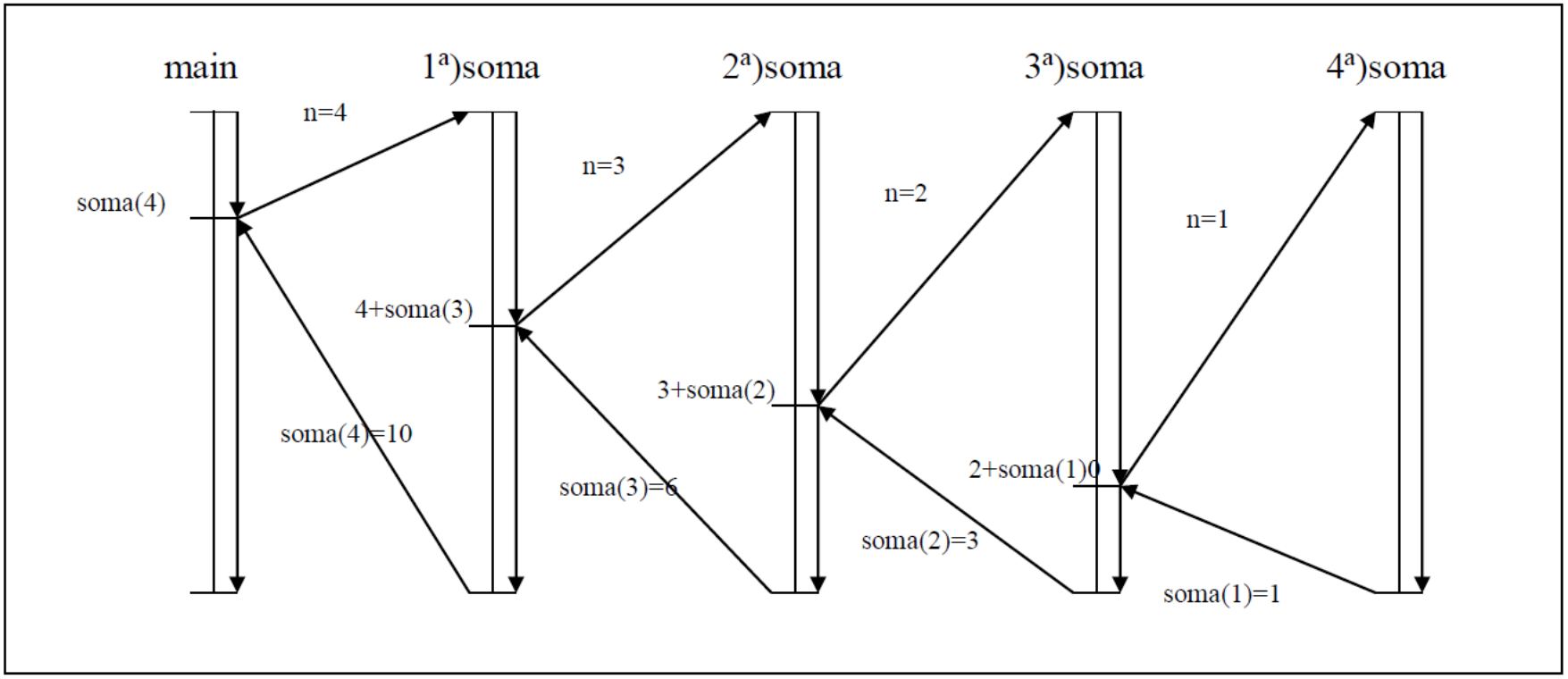
Recursividade

- **2 passos:**
  - Definir uma função recursiva, que decresce até alcançar a solução mais simples (trivial)
    - Ex: S(n) = n + S(n-1)
  - Definir a condição de parada (solução trivial)
    - Ex.: S(1) = 1

Recursividade

```
main( )
{
   int n;
   scanf("%d", &n);
   printf("%d", soma(n));
}
int soma(int n)
{
   if (n == 1)   return (1);
   else return (n + soma(n - 1));
}
```

Recursividade



Exemplo

- Função que imprime os elementos de um vetor

```
void imprime(int v[], int tamanho) {
    int i;
    for (i=0;i<tamanho;i++)
        printf("%d ",v[i]);
}
```

Exercício

- Faça a versão recursiva da função

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

Efeitos da recursão

- A cada chamada
  - Empilham-se na memória os dados locais (variáveis e parâmetros) e o endereço de retorno
    - A função corrente só termina quando a função chamada terminar
  - Executa-se a nova chamada (que também pode ser recursiva)
  - Ao retornar, desempilham-se os dados da memória, restaurando o estado antes da chamada recursiva

Exercício

- Simule a execução da função para um vetor de tamanho 3 e mostre a situação da memória a cada chamada recursiva

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

Alternativa: tem o mesmo efeito?

```
void imprime0(int v[], ...) {
        printf("%d ",v[0]);
        imprime1(v,...);
    }
}

void imprime1(int v[], ...) {
        printf("%d ",v[1]);
        imprime2(v,...);
    }
}
```

```
void imprime2(int v[], ...) {
        printf("%d ",v[2]);
        imprime3(v,...);
    }
}

...
```

Alternativa: tem o mesmo efeito?

```
void imprime0(int v[], ...) {
        printf("%d ",v[0]);
        imprime1(v,...);
    }
}

void imprime1(int v[], ...) {
        printf("%d ",v[1]);
        imprime2(v,...);
    }
}
```

```
void imprime2(int v[], ...) {
        printf("%d ",v[2]);
        imprime3(v,...);
    }
}

...
```

Mesmo resultado, com diferença de haver duplicação de código

Efeitos da recursão

- *Mesmo resultado, com diferença de haver duplicação de código*
  - O que isso quer dizer? Funções recursivas são sempre melhores do que funções não recursivas?

Efeitos da recursão

- *Mesmo resultado, com diferença de haver duplicação de código*
  - O que isso quer dizer? Funções recursivas são sempre melhores do que funções não recursivas?
    - Depende do problema, pois nem sempre a recursão é a melhor forma de resolver o problema, já que pode haver uma versão simples e não recursiva da função (que não duplica código e não consome mais memória)

Recursão

- Quando usar: quando o problema pode ser definido recursivamente de forma natural

- Como usar
  - 1º ponto: definir o problema de forma recursiva, ou seja, em termos dele mesmo
  - 2º ponto: definir a condição de término (ou *condição básica*)
  - 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término (caso mais simples)
    - Caso contrário, qual o problema?

Recursão

- Problema do fatorial
  - 1º ponto: definir o problema de forma recursiva
    - n! = n * (n-1)!
  - 2º ponto: definir a condição de término
    - n=0
  - 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término
    - A cada chamada, n é decrementado, ficando mais próximo da condição de término

Recursão vs. iteração

- Quem é melhor?

```
//versão recursiva
int fatorial(int n) {
    int fat;

    if (n==0) fat=1;
    else fat=n*fatorial(n-1);

    return(fat);
}
```

```
//versão iterativa
int fatorial(int n) {
    int i, fat=1;

    for (i=2;i<=n;i++)
        fat=fat*i;

    return(fat);
}
```

Exercício

- Implemente uma função recursiva para calcular o enésimo número de Fibonacci
  - 1º ponto: definir o problema de forma recursiva
  - 2º ponto: definir a condição de término
  - 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término

Exercício

- Implemente uma função recursiva para calcular o enésimo número de Fibonacci
  - 1º ponto: definir o problema de forma recursiva
    - f(0)=0, f(1)=1, f(n)=f(n-1)+f(n-2) para n>=2
  - 2º ponto: definir a condição de término
    - n=0 e/ou n=1
  - 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término
    - n é decrementado em cada chamada

Recursão vs. iteração

- Quem é melhor? Simule a execução

```
//versão recursiva
int fib(int n) {
    int resultado;

    if (n<2) resultado=n;
    else resultado=fib(n-1)+fib(n-2);

    return(resultado);
}
```

```
//versão iterativa
int fib(int n) {
    int i=1, k, resultado=0;

    for (k=1;k<=n;k++) {
        resultado=resultado+i;
        i=resultado-i;
    }

    return(resultado);
}
```

Recursão vs. iteração

- Quem é melhor? Simule a execução

```
//versão recursiva
int fib(int n) {
    int resultado;

    if (n<2) resultado=n;
    else resultado=fib(n-1)+fib(n-2);

    return(resultado);
}
```

Certamente mais elegante, mas duplica muitos cálculos!

```
//versão iterativa
int fib(int n) {
    int i=1, k, resultado=0;

    for (k=1;k<=n;k++) {
        resultado=resultado+i;
        i=resultado-i;
    }

    return(resultado);
}
```

Recursão vs. iteração

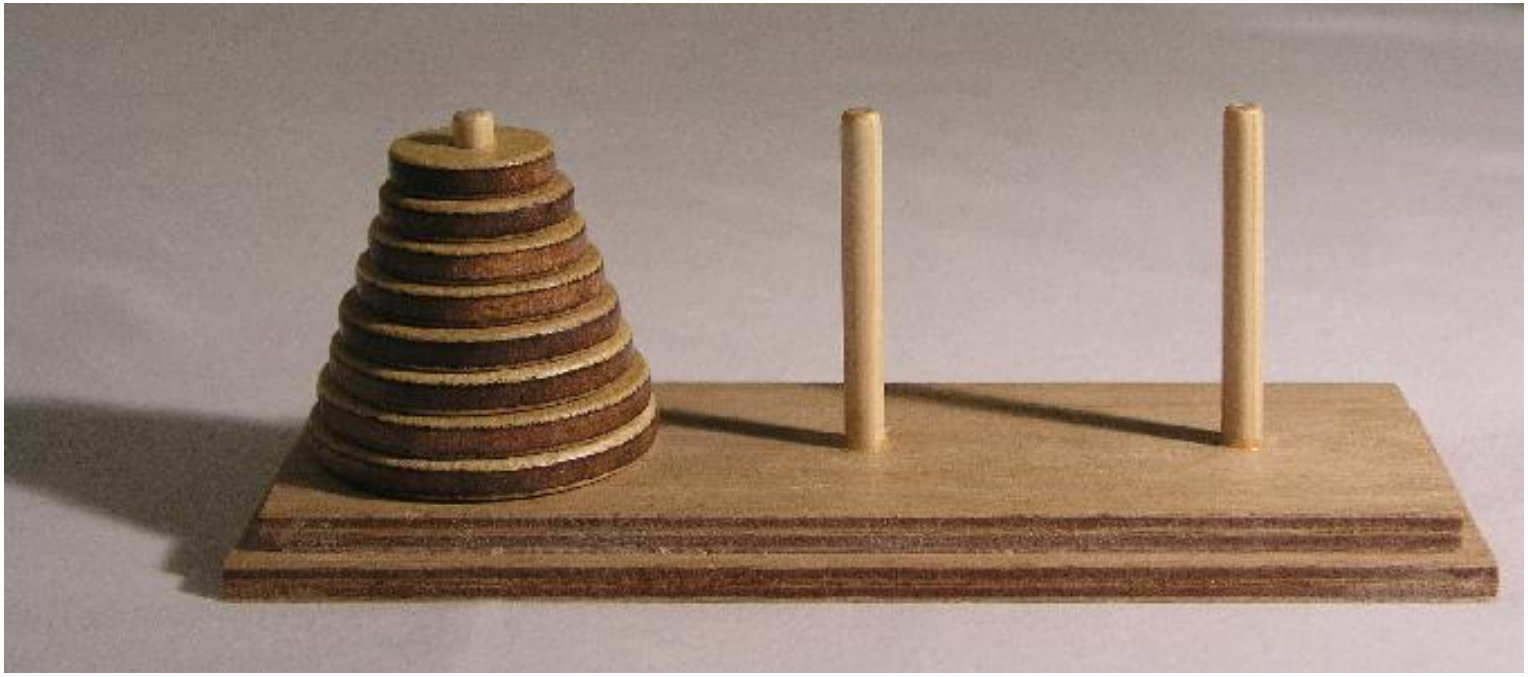
- Quem é melhor?
  - Estimativa de tempo para Fibonacci (Brassard e Bradley, 1996)

n	***10***	***20***	***30***	***50***	***100***
Recursão	8 ms	1 s	2 min	21 dias	10^9 anos
Iteração	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms

Recursão vs. iteração

- Programas recursivos que possuem chamadas ao final do código são ditos terem **recursividade de cauda**
  - São mais facilmente transformáveis em programas iterativos
    - A recursão pode virar uma condição

Torres de Hanoi



Torres de Hanoi

- Jogo
  - Tradicionalmente com 3 hastes: Origem, Destino, Temporária
  - Número qualquer de discos de tamanhos diferentes na haste Origem, dispostos em ordem de tamanho: os maiores embaixo
  - Objetivo: usando a haste Temporária, movimentar um a um os discos da haste Origem para a Destino, sempre respeitando a ordem de tamanho
    - Um disco maior não pode ficar sobre um menor!

Torres de Hanoi

- Implemente uma função recursiva que resolva o problema das Torres de Hanoi

Torres de Hanoi

- Implemente uma função recursiva que resolva o problema das Torres de Hanoi
  - Mover n-1 discos da haste Origem para a haste Temporária
  - Mover o disco n da haste Origem para a haste Destino
  - Recomeçar, movendo os n-1 discos da haste Temporária para a haste Destino

Torres de Hanoi

```
#include <stdio.h>

void mover(int, char, char, char);

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

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

Recursão

- Muito útil para lidar com estruturas de dados mais complexas
  - Listas sofisticadas, árvores, etc.

Imprimir uma lista (recursivo)

```
1.  /*
2.   * Recursive C program to display members of a linked list
3.   */
4.  #include <stdio.h>
5.  #include <stdlib.h>
6.
7.  struct node
8.  {
9.      int a;
10.     struct node *next;
11. };
```

Imprimir uma lista (recursivo)

```
49.  void display(struct node *head)
50.  {
51.      printf("%d    ", head->a);
52.      if (head->next == NULL)
53.      {
54.          return;
55.      }
56.      display(head->next);
57.  }
```

Imprimir lista de forma reversa?

- Sem inverter a lista

Imprimir lista de forma reversa?

```c
/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};
```

```c
/* Function to reverse the linked list */
void printReverse(struct Node* head)
{
    // Base case
    if (head == NULL)
       return;

    // print the list after head node
    printReverse(head->next);

    // After everything else is printed, print head
    printf("%d  ", head->data);
}
```

Exercício

- Escreva as funções recursivas que unem dois (arrays), sem elementos repetidos, classificadas considerando que as duas listas não têm elementos em comum