

Estrutura de Dados – 2º semestre de 2019

Professor Mestre Fabio Pereira da Silva

Recursividade

- Se um problema pode ser resolvido facilmente
 - resolva o problema;
- Se o problema é grande,
 - elabore uma solução menor do problema,
 - relacione com o problema maior,
 - resolva o problema menor,
 - volte ao problema inicial.

Recursividade

- Um objeto é dito recursivo se ele consistir parcialmente ou for definido em termos de si mesmo.
- Uma função recursiva é uma função que faz uma chamada a si mesma.
- Uma função recursiva é definida em termos dela mesma
- Exemplos
 - Números naturais, Função fatorial, Árvore

Recursividade Direta ou Indireta

- Se uma função A contiver uma chamada explícita a si mesma, essa função é dita diretamente recursiva.

$$A \rightarrow A$$

- Se uma função A contiver uma chamada a uma função B , que por sua vez contenha uma chamada a função A , a função A é dita indiretamente recursiva.

$$A \rightarrow B$$

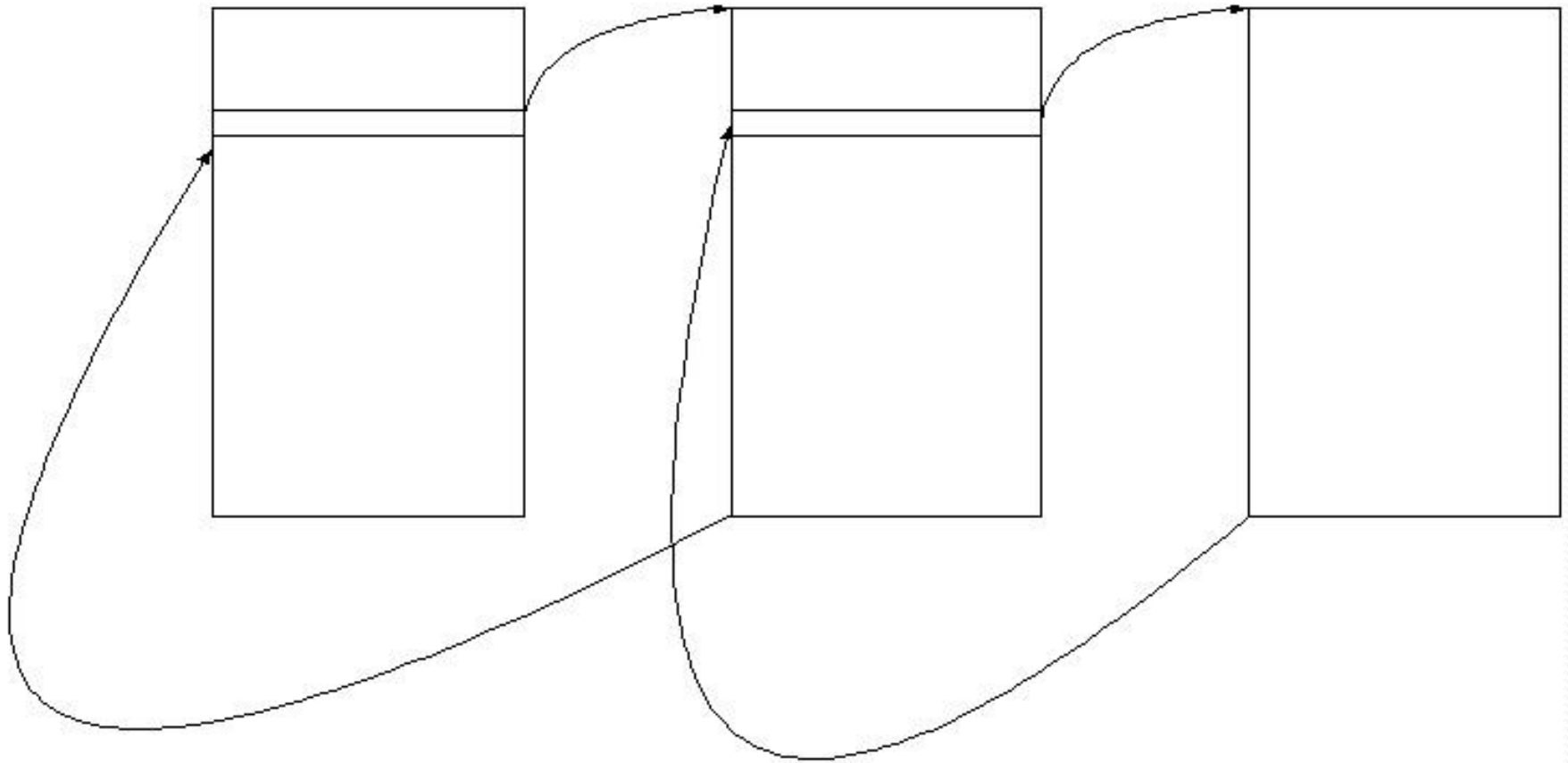
$$B \rightarrow A$$

Eventos

- **Na chamada de procedimentos**
 - Passagem de argumentos
 - Alocação e inicialização de variáveis locais
 - Transferência de controle para a função
-
- **No retorno do procedimento**
 - Recuperação do endereço de retorno
 - Liberação da área de dados
 - Desvio para o endereço de retorno

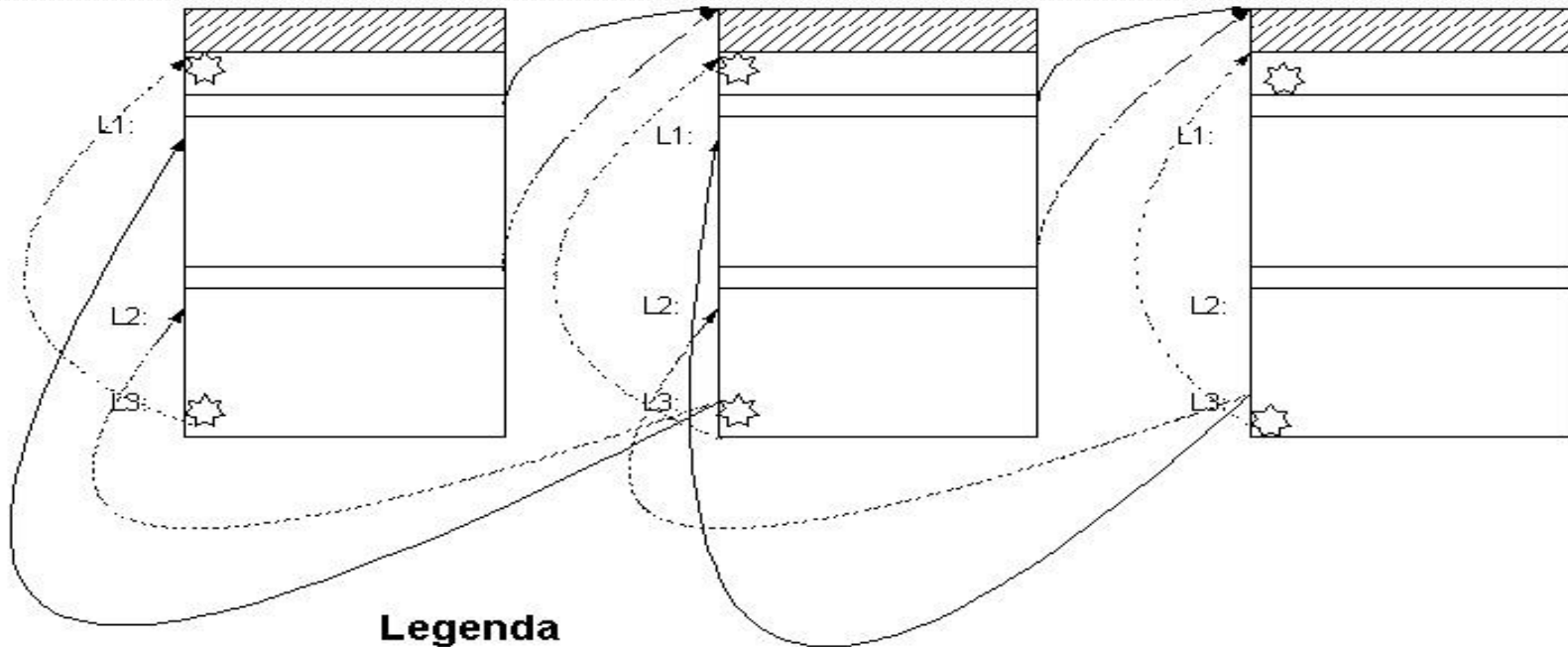
Procedimentos não recursivos

chamada de procedimento



retorno de procedimento

Chamadas de funções recursivas



Legenda

Rótulo depois da primeira chamada recursiva

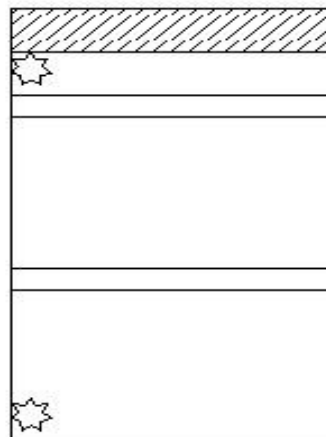
L1:

Rótulo depois da segunda chamada recursiva

L2:

Rótulo depois do desvio condicional final

L3:



Inicialização do procedimento

Desvio condicional no encerramento

Primeira chamada recursiva

Segunda chamada recursiva

Desvio condicional para o início ou encerramento

Implementação de procedimentos recursivos

- Procedimentos recursivos só podem ser implementados em alto nível de abstração.
- As máquinas não executam procedimentos recursivos.
- Cabe ao “software” simular procedimentos recursivos.
- A simulação de recursão utilizará uma pilha com os seguintes atributos gravados:
 - Parâmetros
 - Variáveis
 - Valor da função (se for o caso)
 - Endereço de retorno

Condição de parada

- Nenhum programa nem função pode ser exclusivamente definido por si
 - Um programa seria um loop infinito
 - Uma função teria definição circular
- Condição de parada
 - Permite que o procedimento pare de se executar
 - $F(x) > 0$ onde x é decrescente

Implementação de procedimentos recursivos

- A recursividade é uma estratégia que pode ser utilizada sempre que o cálculo de uma função para o valor n , pode ser descrita a partir do cálculo desta mesma função para o termo anterior $(n-1)$.

Exemplo – Função fatorial:

$$n! = n * (n-1) * (n-2) * (n-3) * \dots * 1$$

$$(n-1)! = (n-1) * (n-2) * (n-3) * \dots * 1$$

logo:

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

Fluxo de Execução

- Internamente, quando qualquer chamada de função é feita dentro de um programa, é criado um **Registro de Ativação** na **Pilha de Execução** do programa
- O registro de ativação armazena os parâmetros e variáveis locais da função bem como o “ponto de retorno” no programa ou subprograma que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função

Fluxo de Execução

- Sempre que há uma chamada de função (recursiva ou não) os parâmetros e as variáveis locais são empilhadas na pilha de execução.
- No caso da função **recursiva**, para cada chamada é criado um ambiente local próprio. (As variáveis locais de chamadas recursivas são independentes entre si, como se fossem provenientes de funções diferentes).

Fatorial de um número.

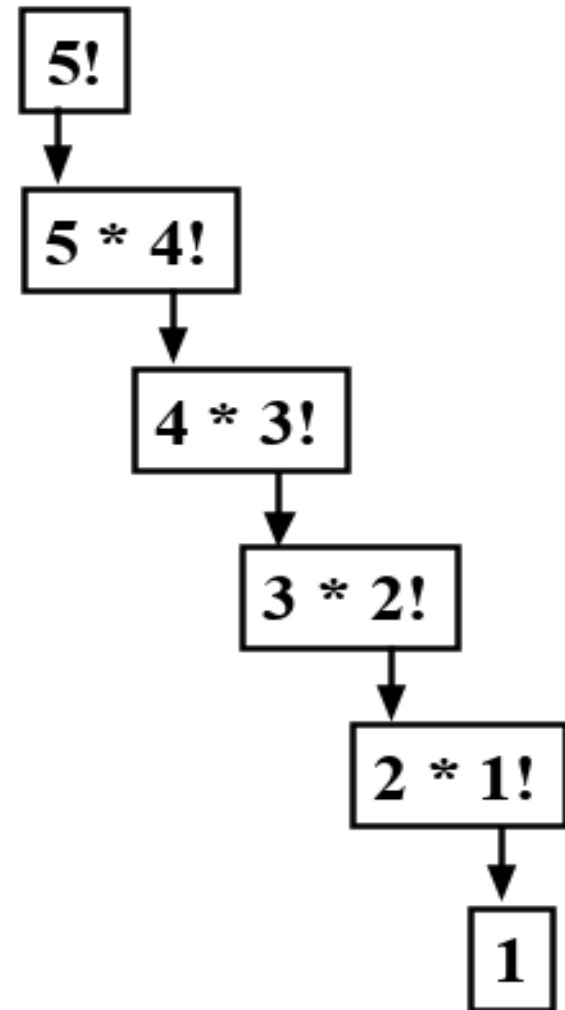
$$1! = 1$$

$$2! = 2 * 1$$

$$3! = 3 * 2 * 1$$

$$4! = 4 * 3 * 2 * 1$$

$$5! = 5 * 4 * 3 * 2 * 1$$





```
fact(n) = n * fact(n - 1)
```

```
se n=0
```

```
então
```

```
fact(0) = 1
```

```
senão
```

```
x = n-1
```

```
y = fact(x)
```

```
fact(n) = n * y
```

```
fim do se
```

												1	ND	ND
								2	ND	ND		2	1	ND
				3	ND	ND		3	2	ND		3	2	ND
n	x	y		n	x	y		n	x	y		n	x	y
<u>Inicialmente</u>				<u>fact(3)</u>				fact(2)				fact(1)		

0	ND	ND												
1	0	ND		1	0	1								
2	1	ND		2	1	ND		2	1	1				
3	2	ND		3	2	<u>ND</u>		3	2	<u>ND</u>		3	2	2
n	x	y		n	x	y		n	x	y		n	<u>x</u>	<u>y</u>
<u>fact(0)</u>				y = fact(0)				y = fact(1)				y = fact(2)		

n	x	y	
fact(3)			

```
public class Factorial {  
    public static void main(String[] args) {  
        int input = Integer.parseInt(args[0]);  
        double result = factorial(input);  
        System.out.println(result);  
    }  
    public static double factorial(int x) {  
        if (x<0) return 0.0;  
        else if (x==0) return 1.0;  
        else return x*factorial(x-1);  
    }  
}
```



```
public static int fib(int n)
{
    .
    int x,y;
    if (n <= 1) return 1;
    else {
        x = fib(n-1);
        y = fib(n-2);
        return x + y;
    }
}
```

Busca Binária

- Divide seu vetor em duas metades
- Três condições
 1. Se o item for igual ao item que está na metade do vetor, o item foi encontrado
 2. Se for menor, procure na primeira metade
 3. Se for maior procure na segunda metade

Busca Binária

Procurar por R

1	2	3	4	5	6	7	8	9	10
A	C	E	H	L	M	P	R	T	V
					I		X		F

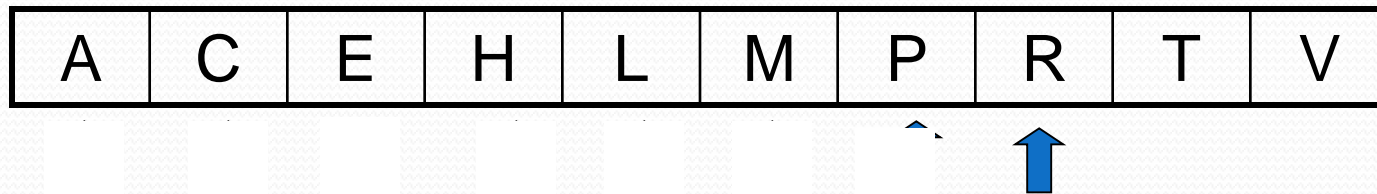
-2 Comparações!

-Casos piores: quando os itens estiverem no início do vetor.
Nesse caso, seria melhor utilizar busca seqüencial. Mas como saber quando o item está no início do vetor?

Busca Sequencial

Procurar por R

A	C	E	H	L	M	P	R	T	V
---	---	---	---	---	---	---	---	---	---



- 8 Comparações!
- Se estivermos procurando o item V, o número de comparações seria a quantidade de elementos no vetor
- O ideal seria dividir o vetor pela metade para então procurar (Busca Binária)

Busca Binária

Procedimento Busca_Binária(inteiro: x, Início, Fim);

Inteiro: meio

Início

meio \leftarrow div((início + fim), 2)

Se fim < início **então**

escreva ('Elemento Não Encontrado')

Senão se (v[meio]) = x **então**

escreva ('Elemento está na posição ', meio)

senão

se v[meio] < x **então**

 início \leftarrow meio + 1;

 Busca_Binaria (x, início, fim);

senão

 fim \leftarrow meio - 1;

 Busca_Binaria (x, início, fim);

fim se

fim se

fim se

Fim

Percurso em lista encadeada

```
void mostra_lista_recursivo2 (Lista* p) {  
    if (p!=NULL) {  
        printf ("%d\t", p->info);  
        mostra_lista_recursivo(p->prox);  
    }  
}
```

Divisão e Conquista

- Construção incremental
- Consiste em, inicialmente, resolver o problema para um subconjunto dos elementos da entrada e, então adicionar os demais elementos um a um.
- Em muitos casos, se os elementos forem adicionados em uma ordem ruim, o algoritmo não será eficiente.
- Ex: Calcule $n!$, recursivamente

Divisão e Conquista

- Dividir o problema em determinado número de subproblemas.
- Conquistar os subproblemas, resolvendo os recursivamente.
- Se o tamanho do subproblema for pequeno o bastante, então a solução é direta.
- Combinar as soluções fornecidas pelos subproblemas, a fim de produzir a solução para o problema original.

Contatos

- Email: fabio.silva321@fatec.sp.gov.br
- LinkedIn: <https://br.linkedin.com/in/b41a5269>