Estrutura de Dados – 1º semestre de 2021

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.

$$A \rightarrow B$$

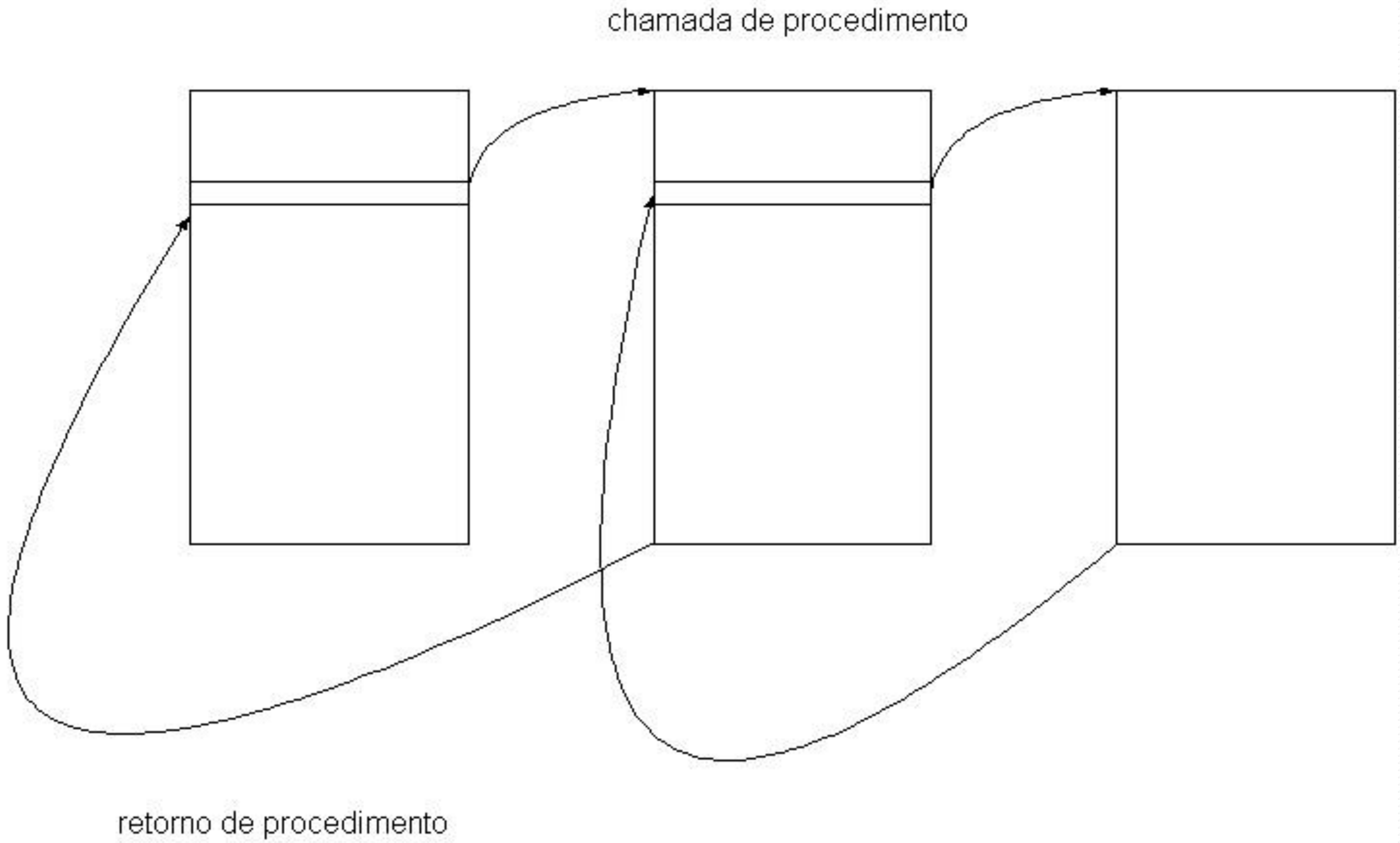
$$B \rightarrow A$$

Eventos

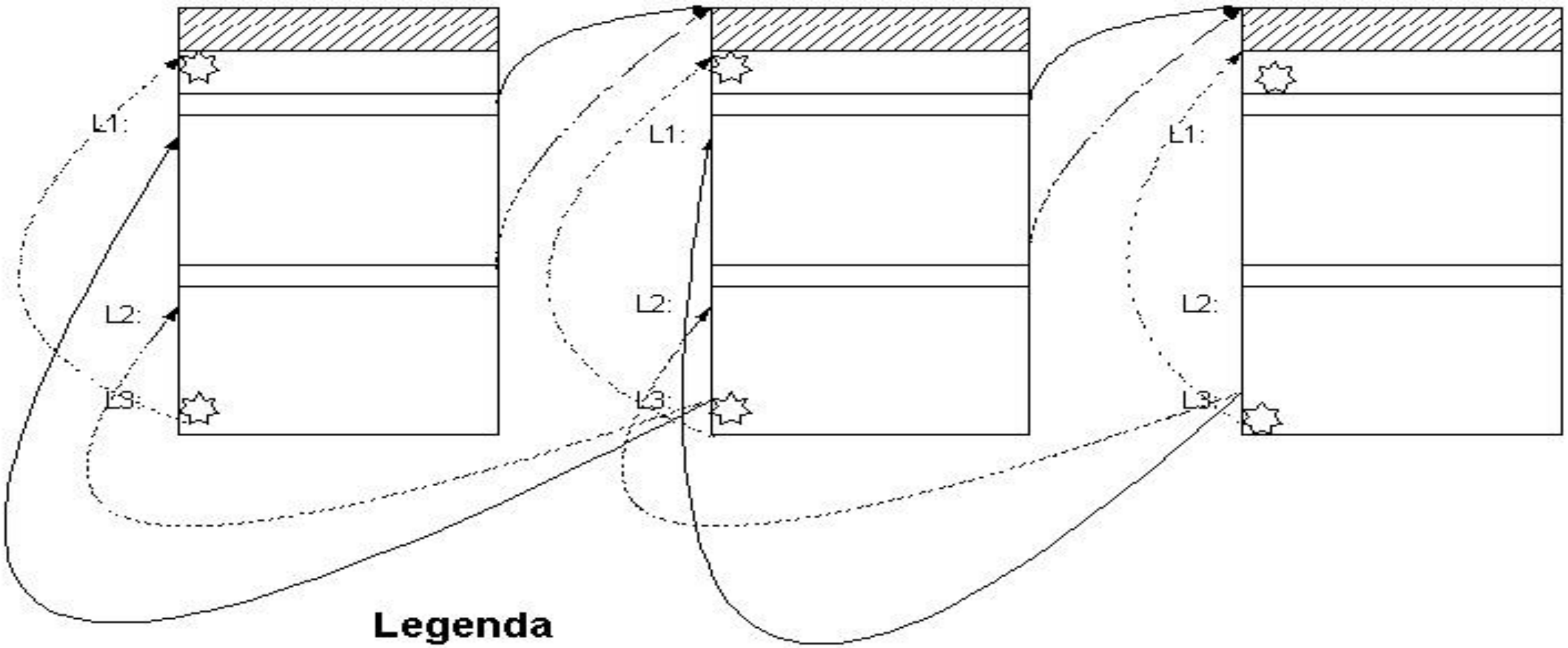
- **Na chamada de procedimentos**
- Passagem de argumentos
- Alocação e inicialização de variáveis locais
- Transferência de controle para a função

- **No retorno do procedimento**
- Recuperação do endereço de retorno
- Liberação da área de dados
- Desvio para o endereço de retorno

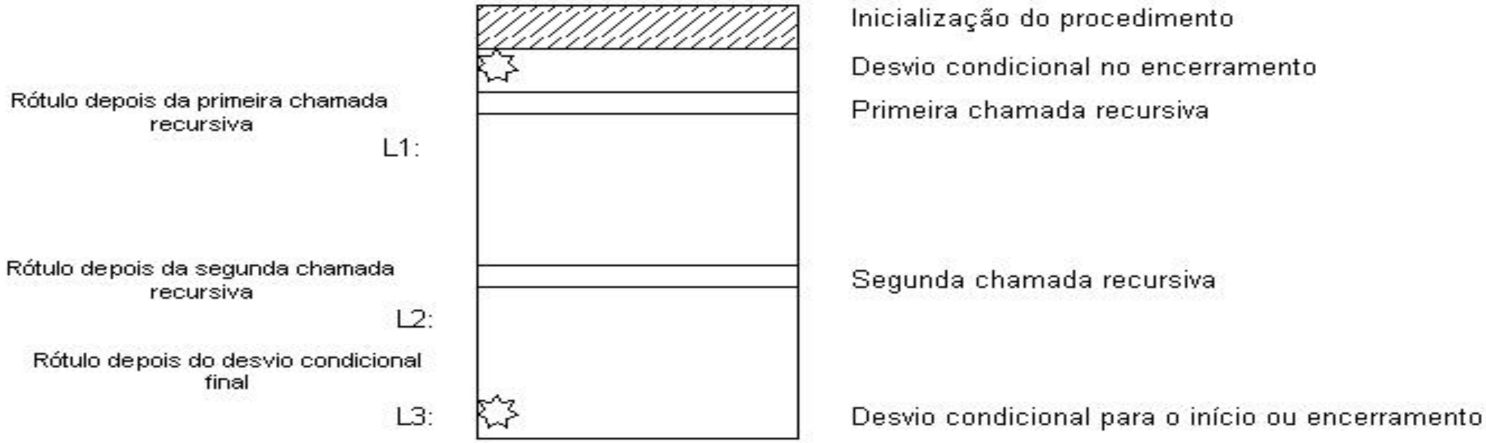
Procedimentos não recursivos



Chamadas de funções recursivas



Legenda



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)* > o 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:

$n! = n * (n-1) * (n-2) * (n-3) *....* 1$
$(n-1)! = (n-1) * (n-2) * (n-3) *....* 1$

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.

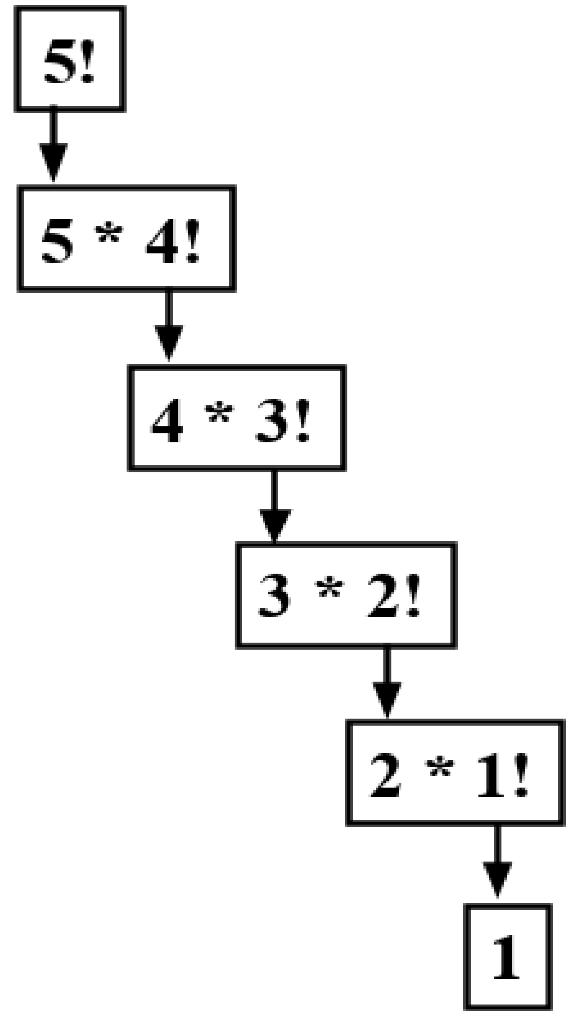
$1! = 1$

$2! = 2 * 1$

$3! = 3 * 2 *1$

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

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



```
fact(n) = n * fact(n - 1)
      se n=o·
          então
              fact(0) = 1
          senão
              x = n-1
             y = fact(x)
              fact(n) = n * y
      fim do se
```

n	**x**	**y**

Inicialmente

3	ND	ND
n	**x**	**y**

fact(3)

2	ND	ND
3	2	ND
n	**x**	**y**

fact(2)

1	ND	ND
2	1	ND
3	2	ND
n	**x**	**y**

fact(1)

0	ND	ND
1	0	ND
2	1	ND
3	2	ND
n	**x**	**y**

fact(0)

1	0	1
2	1	ND
3	2	ND
n	**x**	**y**

y = fact(0)

2	1	1
3	2	ND
n	**x**	**y**

y = fact(1)

3	2	2
n	**x**	**y**

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!

-Pior caso: quando os itens estiverem no início do vetor. Nesse caso, seria melhor utilizar busca seqüencial. Mas como saber quando o ítem 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,Inicio, Fim);
Inteiro: meio
Início
        meio ← div((inicio + fim), 2)
        Se fim < inicio então
                escreva (‘Elemento Não Encontrado’)
        Senão se (v[meio]) = x então
                escreva (‘Elemento está na posição ’,meio)
                senão
                        se v[meio] < x então
                                inicio ←  meio +1;
                                Busca_Binaria (x, inicio, fim);
                        senão
                                fim ←  meio - 1;
                                Busca_Binaria (x, inicio, 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);
    }
}
```

Estratégias para problemas tratáveis

- **Estruturas de Dados**
  - Use uma estrutura adequada
- **Espaço por Tempo**
  - Gaste mais espaço para economizar tempo
- **Algoritmos Probabilísticos**
  - Use aleatoriedade para conseguir eficiência
- **Dividir para conquistar (top-down)**
  - Divida em subproblemas semelhantes e disjuntos, resolva e combine
- **Programação Dinâmica (bottom-up)**
  - Comece com subproblemas e componha um maior, reusando solução de subproblemas compartilhados
- **Algoritmos Gulosos**
  - Sempre pegue o melhor

Forma geral da recursão

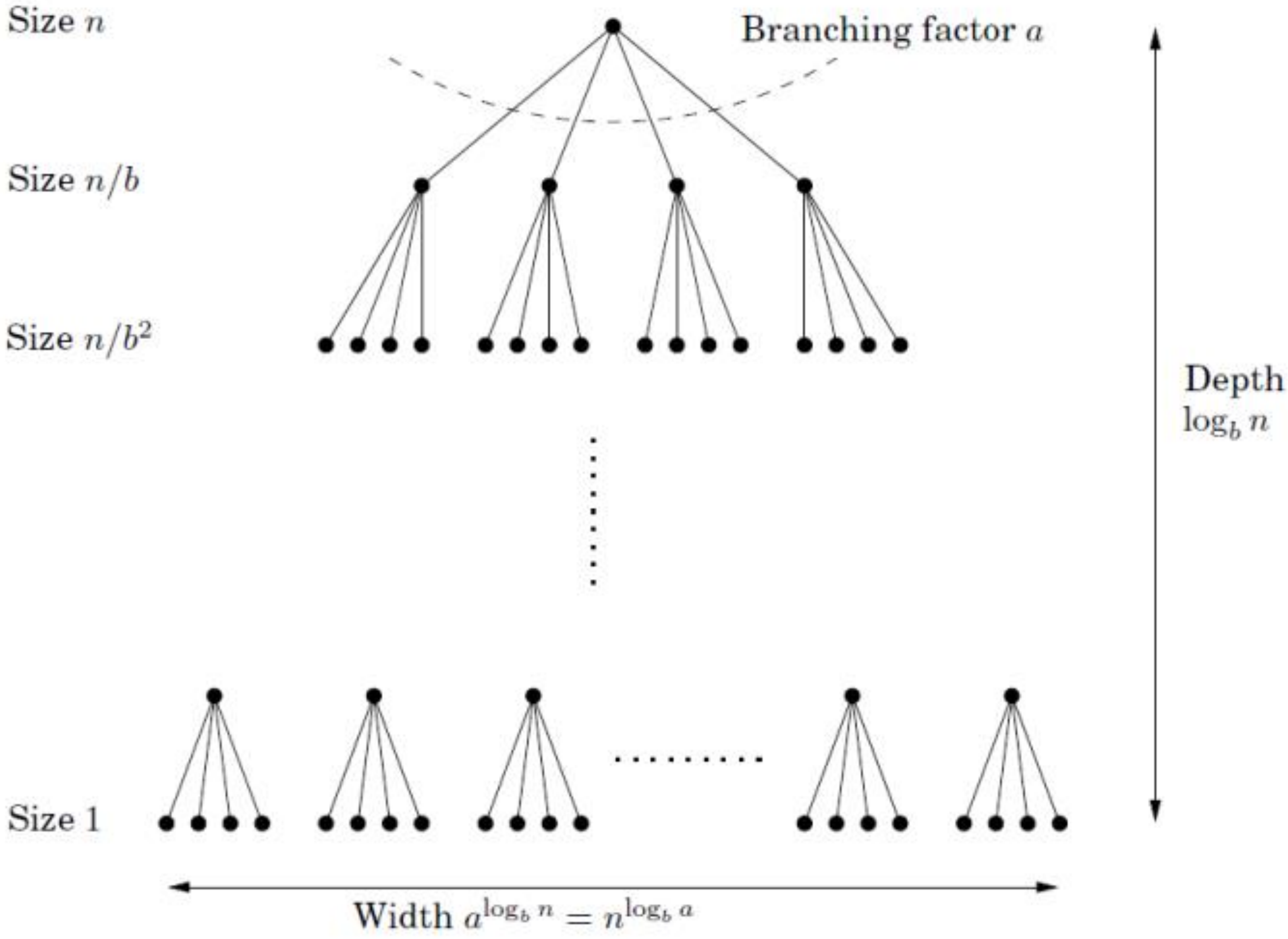
$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Número de subproblemas

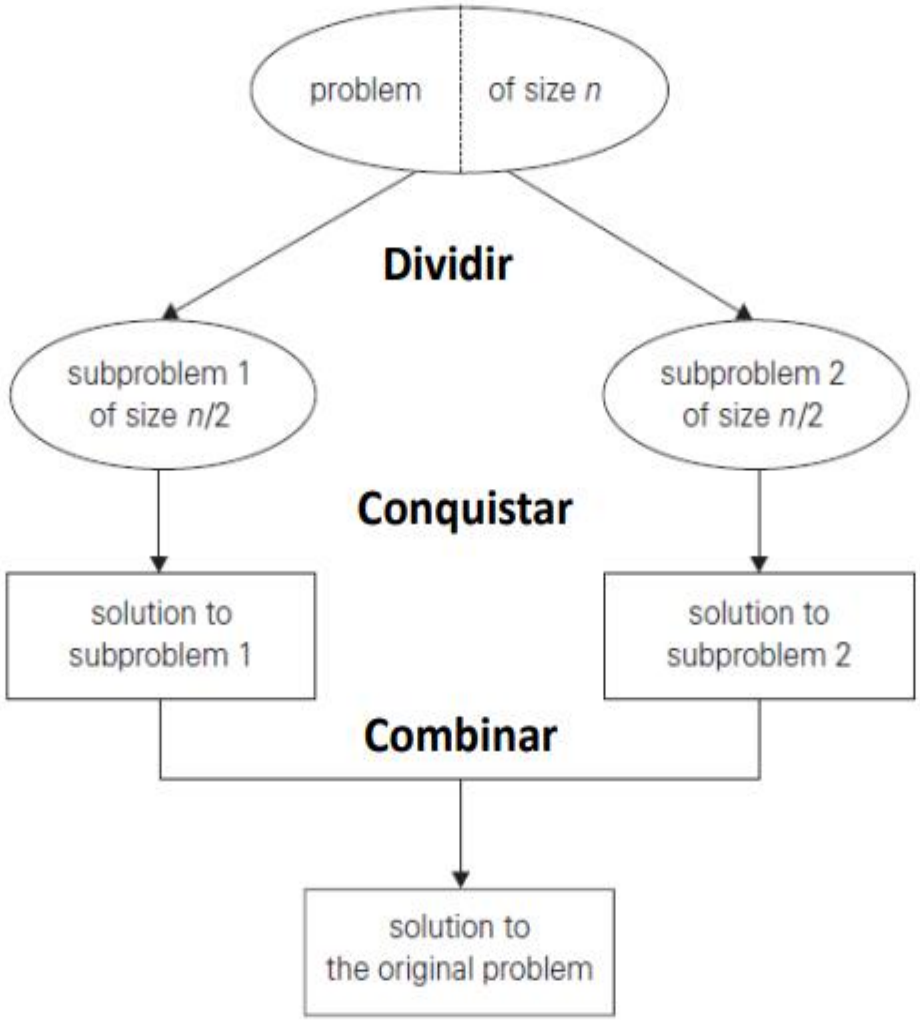
Tamanho dos subproblemas

Custo local da função

Forma geral da recursão



Forma geral da recursão



Divisão e Conquista

- Algoritmos baseados em divisão e conquista são, em geral, recursivos.
- A maioria dos algoritmos de divisão e conquista divide o problema em a subproblemas da mesma natureza, de tamanho n/b
- **Vantagens**:

✓ Requerem um número menor de acessos à memória

✓ São altamente paralelizáveis. Se existem vários processadores disponíveis, a estratégia propicia eficiência

Divisão e Conquista

- Diminuir complexidade (em geral de linear para logarítmico) de algoritmos polinomiais
  - Busca, Ordenação, Multiplicação e Exponenciação
- Apresentar melhor função de complexidade de pior caso
  - Mínimo/máximo

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, importante para se obter uma boa eficiência temporal
- **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.
- **Algoritmos adequados para processamento paralelo**

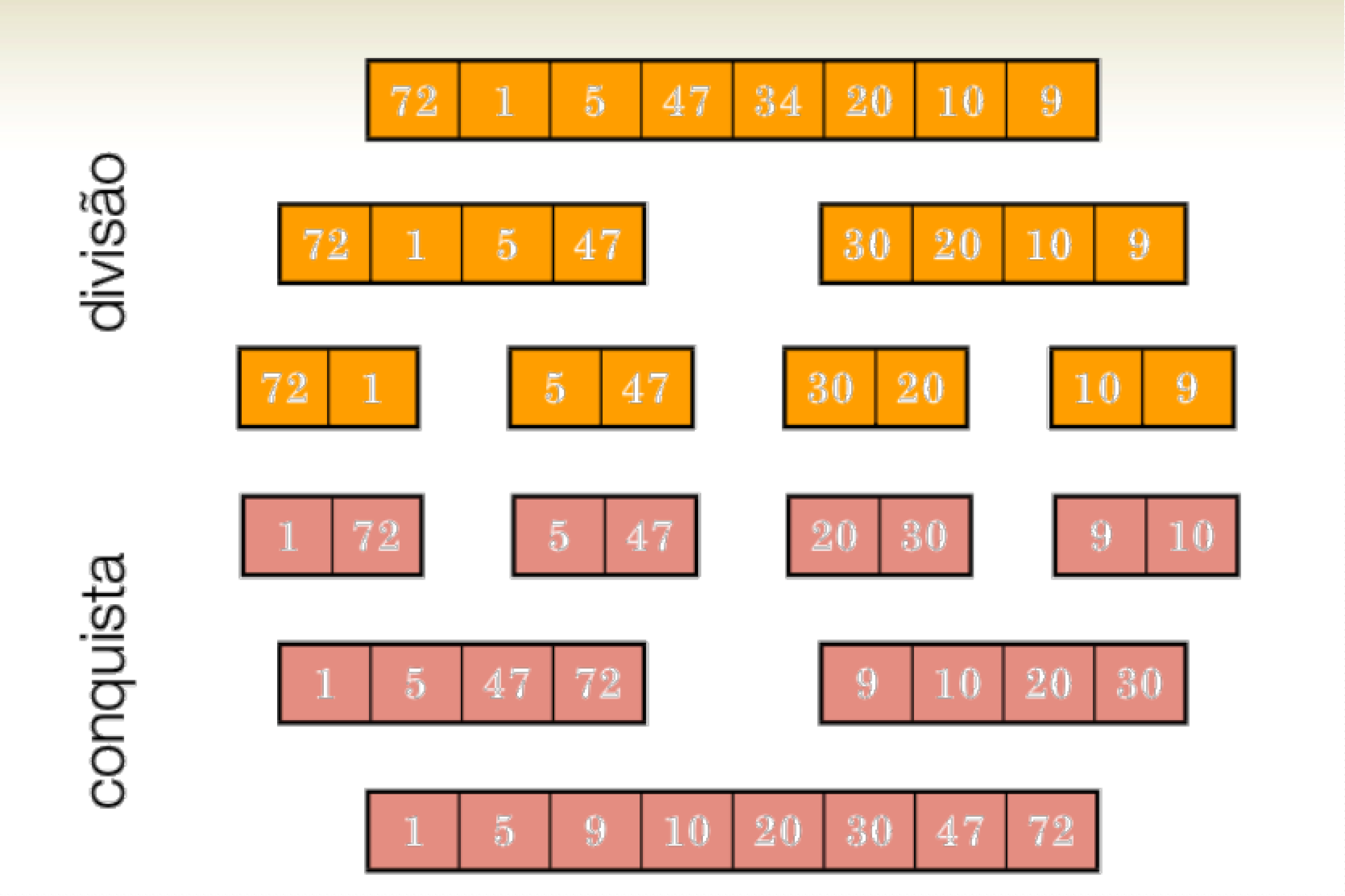
Divisão e Conquista

- **A busca binária recursiva utiliza essa técnica?**
- **Dividir**
  - Divide o problema em subproblemas?
- **Conquistar**:
  - Resolve os subproblemas recursivamente?
- **Combinar**:
  - Forma a solução final a partir da combinação das soluções dos subproblemas?
- **Nesse caso, a etapa de combinar tem custo zero, pois o resultado do subproblema já é o resultado do problema maior.**

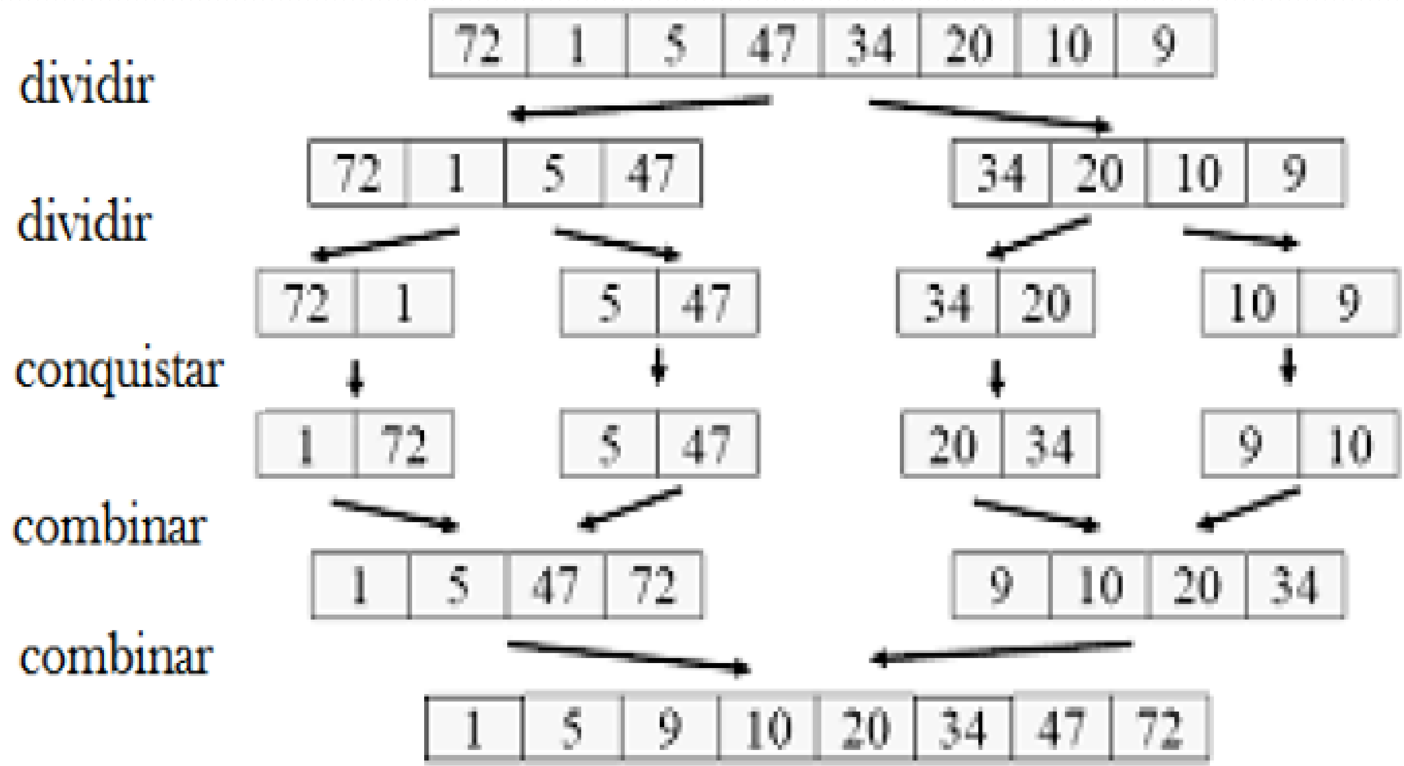
Divisão e Conquista

- Problemas menores (fração), **independentes e não-sobrepostos**
- **Divisão e combinação são partes não-recursivas**
  - Algoritmos tendem a tornar uma das duas mais complicada
  - Mergesort, Quicksort
- Importante definir o que é pequeno
  - **Parada de recursão (multiplicação de números)**

Exemplo: Algoritmo Merge Sort



Exemplo: Algoritmo Merge Sort



Divisão e Conquista - Vantagens

- Resolução de problemas difíceis, como a Torre de Hanói
- Pode gerar algoritmos eficientes
- Ótima ferramenta para busca de algoritmos eficientes, com forte tendência a complexidade logarítmica
- Paralelismo
- Facilmente paralelizável na fase de conquista

Divisão e Conquista - Desvantagens

- Recursão ou Pilha explícita
- Tamanho da Pilha
- Número de chamadas recursivas e/ou armazenadas na pilha pode ser um inconveniente
- Dificuldade na seleção dos casos bases
- Repetição de subproblemas
- Situação que pode ser resolvida através do uso de memorização

Exemplo

```
def divisao_e_conquista(x):
    if x é pequeno ou simples:
        return resolve(x)
    else:
        decompor x em n conjuntos menores x0,x1,...,xn-1
        for i in [0,1,...,n-1]:
            yi = divisao_e_conquista(xi)
        combinar y0,y1,...,yn-1 em y
        return y
```

Exemplo – Maior valor de um vetor

- É possível aplicar Divisão em Conquista para encontrar o maior valor em um vetor?

- Opção 1:

```
int maxVal1(int A[], int n) {
    int max = A[0];
    for (int i = 1; i < n; i++) {
        if( A[i] > max ) max = A[i];
    }
    return max;
}
```

  - Melhor alternativa??

Exemplo – Maior valor de um vetor

- Opção 2:

```
int maxVal2(int A[], int init, int end) {
    if (end - init <= 1)
        return max(A[init], A[end]);
    else {
        int m = (init + end)/2;
        int v1 = maxVal2(A,init,m);
        int v2 = maxVal2(A,m+1,end);
        return max(v1,v2);
    }
}
```

- E agora? Melhorou?

Exemplo – Exponenciação

```
int pow1(int a, int n) {
    int p = 1;
    for (int i = 0; i < n; i++)
        p = p * a;
    return p;
}
```

```
int pow2(int a, int n) {
    if (n == 0)
        return 1;
    if (n % 2 == 0)
        return pow2(a,n/2) * pow2(a,n/2);
    else
        return pow2(a,(n-1)/2) * pow2(a,(n-1)/2) * a;
}
```

Contatos

- Email: fabio.silva321@fatec.sp.gov.br
- Linkedin: https://br.linkedin.com/in/b41a5269
- Facebook: https://www.facebook.com/fabio.silva.56211