Dividir & Conquistar

Análise de Algoritmos

Recursão

Para resolvermos um dado problema podemos utilizar a seguinte idéia: *dividi-lo em subproblemas similares ao problema principal, resolvê-los recursivamente e então combinar as respostas de modo a obter a solução para o problema original*.

Ex.:

Fatorial de um número *n* inteiro positivo:

$$\text{Fatorial}(n) = n * \text{Fatorial}(n-1)$$

Paradigma da Divisão e Conquista

O paradigma da divisão e conquista envolve três passos em cada nível da recursão. Sendo eles:

- **Dividir** o problema em um número de subproblemas.
- **Conquistar** ou resolver os subproblemas recursivamente. Quando o problema tem uma solução trivial, resolver em tempo constante.
- **Combinar** as soluções encontradas para formar uma solução para o problema original.

Análise da Divisão e Conquista

- Para descrevermos o tempo de execução de algoritmos que possuem chamadas recursivas, geralmente utilizamos recorrências.
- Uma *recorrência* é uma equação ou inequação que descreve uma função em termos da mesma função para entradas menores.

Recorrências

- A recorrência para expressar o tempo de execução da divisão e conquista é baseada nos três passos do paradigma.
- Seja $T(n)$ o tempo de execução de um problema de tamanho n.
  - Se n é pequeno ($\leq c$, onde c é uma constante), a solução é trivial e portanto $T(n) = O(1)$.
  - Suponha que dividimos o problema em a subproblemas de tamanho $1/b$ do tamanho original.
  - Se gastamos $D(n)$ para dividir o problema em subproblemas e $C(n)$ para combinar as soluções dos subproblemas, temos que:

$$T(n) = \begin{cases} 1 & , n \leq c \\ aT(n/b) + D(n) + C(n) & , n > c \end{cases}$$

Exemplo

/ ordenação usando recursão */*

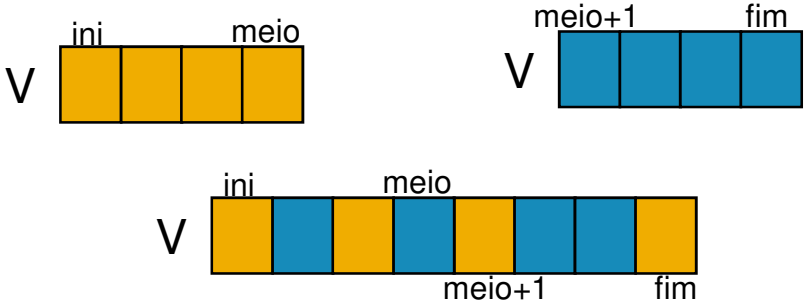
```
MergeSort(V, ini, fim)
{
(1)  se (ini < fim) então
(2)     meio ← ⌊ (ini + fim) / 2 ⌋
(3)     MergeSort(V, ini, meio)
(4)     MergeSort(V, meio + 1, fim)
(5)     Intercala(V, ini, meio, fim)
}
```

Exemplo

```
/* ordenação usando recursão */

MergeSort(V, ini, fim)
{
(1)  se (ini < fim) então
(2)     meio ← ⌊ (ini + fim) / 2 ⌋
(3)     MergeSort(V, ini, meio)
(4)     MergeSort(V, meio + 1, fim)
(5)     Intercala(V, ini, meio, fim)
}
```

A rotina *Intercala* tem a tarefa de intercalar o vetor já ordenado V[*ini*...*meio*] com o vetor V[*meio*+1...*fim*] também ordenado, colocando o resultado desta intercalação em V[*ini*...*fim*]



Exemplo

```
/* ordenação usando recursão */

MergeSort(V, ini, fim)
{
(1)  se (ini < fim) então
(2)     meio ← ⌊ (ini + fim) / 2 ⌋
(3)     MergeSort(V, ini, meio)
(4)     MergeSort(V, meio + 1, fim)
(5)     Intercala(V, ini, meio, fim)
}
```

- **Dividir**:

o problema original, ordenação do vetor de tamanho n, é dividido em subproblemas de ordenação, ou seja, na ordenação de 2 vetores menores com $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$ elementos cada.

Exemplo

```
/* ordenação usando recursão */

MergeSort(V, ini, fim)
{
(1)  se (ini < fim) então
(2)     meio ← ⌊ (ini + fim) / 2 ⌋
(3)     MergeSort(V, ini, meio)
(4)     MergeSort(V, meio + 1, fim)
(5)     Intercala(V, ini, meio, fim)
}
```

- **Conquistar**:

Ou resolver os subproblemas recursivamente, neste caso, realizando 2 chamadas recursivas, uma para cada subproblema.

Quando o problema tem uma solução trivial (*ini* ≥ *fim*), resolver em tempo constante, isto é, não há chamadas recursivas

Exemplo

/ ordenação usando recursão */*

```
MergeSort(V, ini, fim)
{
(1)  se (ini < fim) então
(2)     meio ← ⌊ (ini + fim) / 2 ⌋
(3)     MergeSort(V, ini, meio)
(4)     MergeSort(V, meio + 1, fim)
(5)     Intercala(V, ini, meio, fim)
}
```

- **Combinar**:

  este passo constitui na intercalação dos dois sub-vetores já ordenados para formar a solução original (em cada passo da recursão)

Exemplo

```
/* ordenação usando recursão */

MergeSort(V, ini, fim)
{
(1)  se (ini < fim) então
(2)     meio ← ⌊ (ini + fim) / 2 ⌋
(3)     MergeSort(V, ini, meio)
(4)     MergeSort(V, meio + 1, fim)
(5)     Intercala(V, ini, meio, fim)
}
```

- Seguindo o método geral para encontrar a recorrência da divisão e conquista, temos que:

$$T(n) = \begin{cases} 1 & ,n \leq c \\ aT(n/b) + D(n) + C(n) & ,n > c \end{cases}$$

- onde:

  $c = 1$ ... é o tamanho da entrada para a solução trivial

  $a = 2$ ... é a quantidade de subproblemas

  $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$ ... são os tamanhos de cada subproblemas

  $D(n) = O(1)$ ... tempo para realizar a divisão

  $C(n) = O(n)$ ... tempo para intercalar os vetores de resposta

Exemplo

```
/* ordenação usando recursão */

MergeSort(V, ini, fim)
{
(1)  se (ini < fim) então
(2)     meio ← ⌊ (ini + fim) / 2 ⌋
(3)     MergeSort(V, ini, meio)
(4)     MergeSort(V, meio + 1, fim)
(5)     Intercala(V, ini, meio, fim)
}
```

- Então teremos:

$$T(n) = \begin{cases} O(1) & , n \leq 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(1) + O(n) & , n > 1 \end{cases}$$

Obs: Como os subproblemas podem ter tamanhos diferentes, então representaremos separadamente as duas chamadas recursivas na recorrência.

$c = 1$ ... é o tamanho da entrada para a solução trivial

$a = 2$ ... é a quantidade de subproblemas

$\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$ ... são os tamanhos de cada subproblemas

$D(n) = O(1)$ ... tempo para realizar a divisão

$C(n) = O(n)$ ... tempo para intercalar os vetores de resposta

Resolvendo as recorrências

- Método Iterativo:
  Este método para resolver recorrências consiste em expandir a recorrência e expressá-la como um somatório dependente de *n*.
- Método Indutivo ou da Substituição:
  Devemos começar com um bom "chute"sobre a solução e logo após aplicar indução para provar que a solução é verdadeira.
- Mudança de Variáveis:
  Algumas vezes, a mudança de variáveis pode fazer uma recorrência desconhecida recair em uma conhecida.

Método iterativo

- Tomando como exemplo nossa expressão do *Merge Sort*, temos:

  Façamos inicialmente algumas simplificações, substituindo O(1) por 1, O(n) por n e desprezaremos a diferença entre $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$, então:

$$T(n) = \begin{cases} 1 & , n \leq 1 \\ 2T(n/2) + 1 + n & , n > 1 \end{cases}$$

Método iterativo

$= 2(T(n/2)) + n + 1$
$= 2(2T(n/4) + n/2 + 1) + n + 1$
$= 4T(n/4) + 2n + 3$
$= 4(2T(n/8) + n/4 + 1) + 2n + 3$
$= 8T(n/8) + 3n + 7$
$= 8(2T(n/16) + n/8 + 1) + 3n + 7$
$= 16T(n/16) + 4n + 15$
... após $i = log_2 n$ passos ...
$= 2log_2 nT(n/2log_2 n) + nlog_2 n + 2log_2 n - 1$
$= n + nlog_2 n + n - 1$
$= O(nlog_2 n)$