: Complexidade e Projeto de Algoritmos **versão: 5/6/24**

Tópico 2 — Algoritmos de divisão e conquista

Professor: Vinícius Dias *Autor: Vinícius Dias*

Sumário

2.1 Algoritmos de Divisão e Conquista: Árvore de recursão, Método da substituição, Método Mestre, Método Akra-Bazzi, Problemas resolvidos por Divisão e Conquista

2.2 Introdução sobre algoritmos de divisão e conquista

2.3 Resolvendo recorrências usando expansão de termos

2.4 Resolvendo recorrências usando árvores de recursão

2.5 Resolvendo recorrências usando indução matemática (método da substituição)

2.6 Resolvendo recorrências usando o Teorema Mestre

Usado para recorrências do tipo

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{se não-trivial} \\ \Theta(1) & \text{se trivial} \end{cases}$$

O teorema consiste em comparar $f(n)$ com a grandeza $n^{\log_b a}$ e o que dominar assintoticamente é a solução da recorrência. A função $f(n)$ representa o custo associado às operações que são efetuadas dentro de uma única chamada recursiva, ignorando o custo adicional das outras chamadas recursivas (conquista). Por outro lado, $n^{\log_b a}$ representa o contrário, o custo associado às chamadas recursivas:

- se $f(n) \succ n^{\log_b a}$: significa que o custo dominante da recorrência se concentra nas raízes das (sub)árvores, isto é, o custo dos níveis da árvore de recursão vai progressivamente diminuindo.
- se $f(n) \prec n^{\log_b a}$: significa que o custo dominante da recorrência acontece nos filhos das (sub)árvores, isto é, o custo dos níveis da árvore de recursão vai progressivamente aumentando.
- se $f(n) \equiv n^{\log_b a}$: significa que cada nível contribui igualmente para o custo.

Caso 1. Se $f(n) \in O(n^{\log_b a+\epsilon})$ para $\epsilon > 0$, então $T(n) \in \Theta(n^{\log_b a})$

Caso 2. Se $f(n) \in \Theta(n^{\log_b a})$, então $T(n) \in \Theta(n^{\log_b a} \log n)$

Caso 3. Se $f(n) \in \Omega(n^{\log_b a-\epsilon})$ para $\epsilon > 0$ e atender uma condição de regularização $af(n/b) \leq cf(n)$ para $c > 0$ e a partir de $n \geq n_0$, então $T(n) \in \Theta(f(n))$

2.7 Resolvendo recorrências usando o Akra-Bazzi

Se trata de uma generalização do teorema mestre, possibilitando a solução de recorrências do tipo:

$$T(n) = \begin{cases} c_0 & n = x_0 \\ \sum_{i=1}^{k} a_i T(b_i n) + f(n) & n > x_0 \end{cases}$$

Onde: $c_0 \in \mathbb{R}$, $x_0 \in \mathbb{N}$, $a_i > 0$, $0 < b_i < 1$, $f(n)$ é uma função assintoticamente positiva. Além disso, seja p a única raíz real da equação:

$$\sum_{i=1}^{k} a_i b_i^p = 1$$

O teorema de Akra-Bazzi conclui um limite assintótico justo para $T(n)$:

$$T(n) \in \Theta\left(n^p + n^p \int_{n_0}^{n} \frac{f(x)}{x^{p+1}} dx\right)$$

2.8 Estudo de caso: ordenação por intercalação (merge sort)

```
MERGE-SORT(A)
if A.length = 1 or A.length = 0 return
mid ← ⌊A.length/2⌋
left ← A[1···mid]
right ← A[mid+1···A.length]
MERGE-SORT(left)
MERGE-SORT(right)
MERGE(left, right, A)
```

```
MERGE(left, right, A)
i ← 1; j ← 1; k ← 1
while i ≤ left.length and j ≤ right.length
    if left[i] < right[j]
        A[k] ← left[i]
        i ← i+1
    else
        A[k] ← right[j]
        j ← j+1
    k ← k+1
while i ≤ left.length
    A[k] ← left[i]
    i ← i+1; k ← k+1
while j ≤ right.length
    A[k] ← right[j]
    j ← j+1; k ← k+1
```

2.9 Estudo de caso: ordenação usando quick sort

```
QUICK-SORT(A, l, r)
if l ≥ r return
p ← PARTITION(A, l, r)
QUICK-SORT(A, l, p−1)
QUICK-SORT(A, p+1, r)
```

```
PARTITION(A, l, r)
pivot ← A[r]
i ← l−1
for j ← 1 to r−1
    if A[j] ≤ pivot
        i ← i+1
        swap(A, j, i)
swap(A, i+1, r)
return i+1
```

2.10 Estudo de caso: subarranjo de soma máxima

O subarranjo de soma máxima pode estar: (1) na metade esquerda; (2) na metade direita; (3) cruzando o meio. Observe que, se estamos no caso (3), a restrição nos ajuda a encontrar o melhor subarranjo: a partir do meio, basta adicionarmos elementos à esquerda rastreando em que posição a soma fica máxima e o mesmo à direita. Assim, (1) e (2) são resolvidos diretamente pela recursão durante a conquista dos subproblemas e (3) é resolvido pelo procedimento MAX-SUM-SUBARRAY-CROSSING-MID de complexidade linear $\Theta(n)$, onde n é o tamanho do arranjo.

```
MAX-SUM-SUBARRAY(a, l, r)
if l = r return (l, r, a[l])
mid ← ⌊(l + r)/2⌋
(l1, r1, s1) ←MAX-SUM-SUBARRAY(a, l, mid)
(l2, r2, s2) ←MAX-SUM-SUBARRAY(a, mid + 1, r)
(l3, r3, s3) ←MAX-SUM-SUBARRAY-CROSSING-MID(a, l, r, mid)
if s1 > s2 and s1 > s3 return (l1, r1, s1)
else if s2 > s3 return (l2, r2, s2)
else return (l3, r3, s3)
```

```
MAX-SUM-SUBARRAY-CROSSING-MID(a, l, r, mid)
s1 ← −∞
s ← 0
for i ← mid downto l
    s ← s + a[i]
    if s > s1
        s1 ← s
        l1 ← i
s2 ← −∞
s ← 0
for i ← mid + 1 to r
    s ← s + a[i]
    if s > s2
        s2 ← s
        r1 ← i
return (l1, r1, s1 + s2)
```

2.11 Estudo de caso: número de inversões em um arranjo