Correção e projeto de algoritmos recursivos

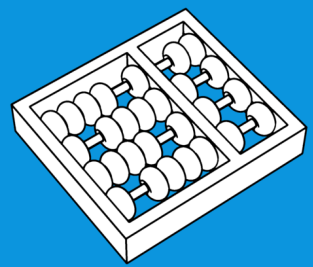
MO417 - Complexidade de Algoritmos I

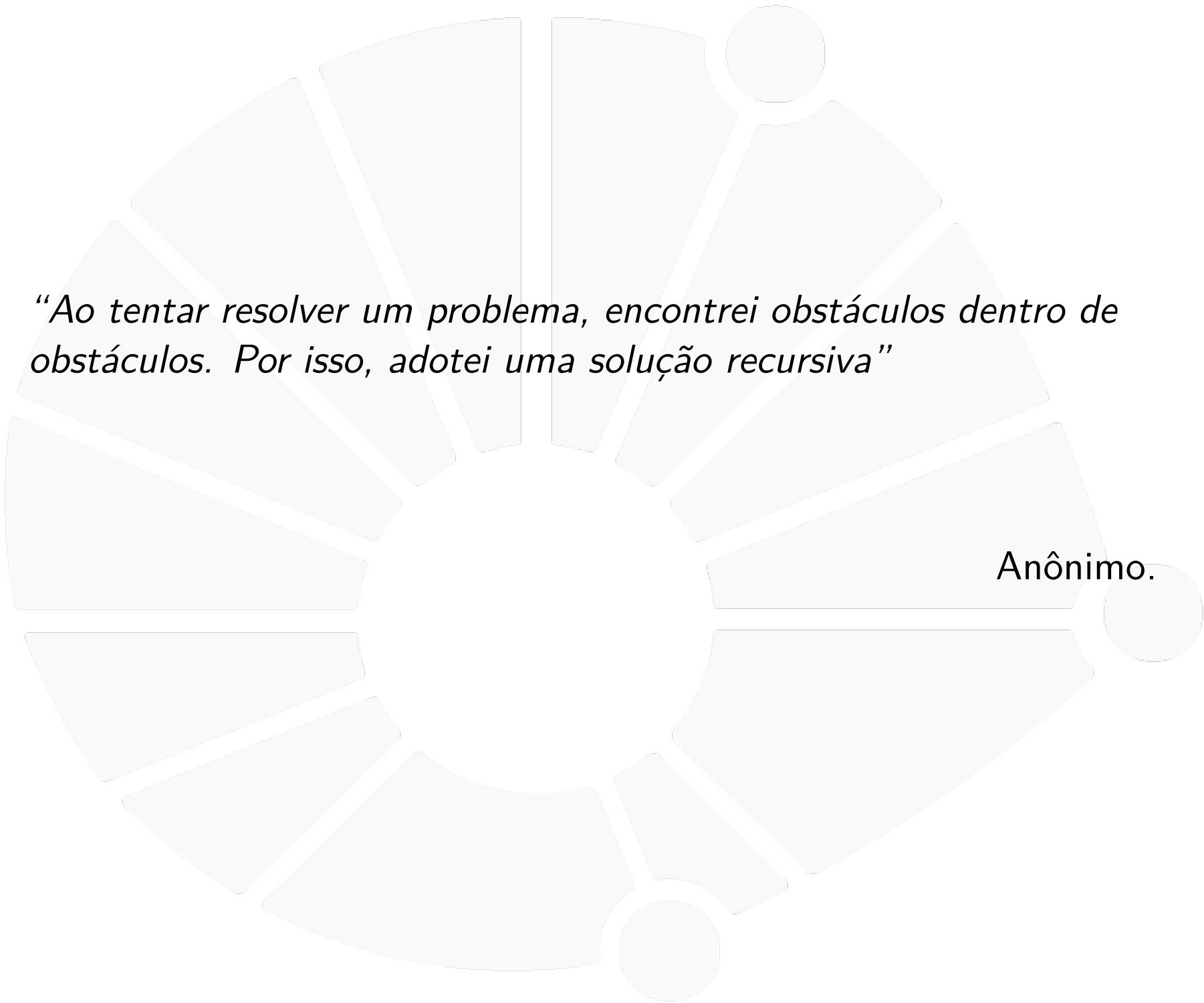
03/24

7

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP

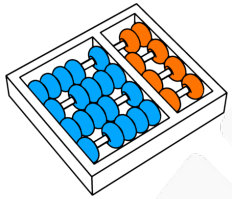




"Ao tentar resolver um problema, encontrei obstáculos dentro de obstáculos. Por isso, adotei uma solução recursiva"

Anônimo.

Última aula



Teorema Master

Teorema (Teorema Master)

Sejam $a \geq 1$ e $b > 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida para os inteiros não negativos pela relação de recorrência:

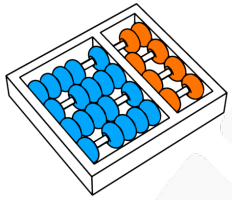
$$T(n) = aT(n/b) + f(n).$$

Então, $T(n)$ tem o seguinte limitante assintótico:

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

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

3. *Se $f(n) \in \Omega(n^{\log_b a + \epsilon})$ para alguma constante $\epsilon > 0$ e se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e todo n suficientemente grande, então $T(n) \in \Theta(f(n))$.*



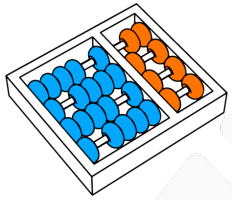
Exemplos de Recorrências

Exemplo (Aplicação do Teorema Master)

- *Caso 1:*
  $T(n) = 9T(n/3) + n.$
  $T(n) = 4T(n/2) + n \log n.$

- *Caso 2:*
  $T(n) = T(2n/3) + 1.$
  $T(n) = 2T(n/2) + (n + \log n).$

- *Caso 3:*
  $T(n) = T(3n/4) + n \log n.$

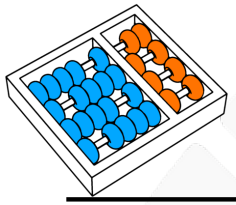


Exemplos de Recorrências

Exemplo (NÃO aplicação do Teorema Master)

- $T(n) = T(n-1) + n$.
- $T(n) = T(n-a) + T(a) + n$, *(*$a \geq 1$ *inteiro).*
- $T(n) = T(\alpha n) + T((1-\alpha)n) + n$, *(*$0 < \alpha < 1$*).*
- $T(n) = T(n-1) + \log n$.
- $T(n) = 2T(\frac{n}{2}) + n \log n$.

Correção de algoritmos recursivos



Correção de MergeSort

Algoritmo: MergeSort(A, p, r)

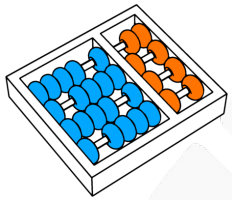
```
1 se p < r
2     q ← ⌊(p + r)/2⌋
3     MergeSort(A, p, q)
4     MergeSort(A, q + 1, r)
5     Intercala(A, p, q, r)
```

Como demonstrar a correção de um algoritmo recursivo?

- Podemos usar **INDUÇÃO** diretamente!
- Precisamos verificar as demais sub-rotinas.

Correção de MergeSort

- Queremos mostrar que MergeSort ordena $A[p \ldots r]$.
- Basta usar indução em $\mathbf{n = r - p + 1}$.



Revendo INTERCALA

Algoritmo: INTERCALA(A, p, q, r)

```
1  para i ← p até q
2      B[i] ← A[i]
3  para j ← q + 1 até r
4      B[r + q + 1 − j] ← A[j]
5  i ← p
6  j ← r
7  para k ← p até r
8      se B[i] ≤ B[j]
9          A[k] ← B[i]
10         i ← i + 1
11     senão
12         A[k] ← B[j]
13         j ← j − 1
```

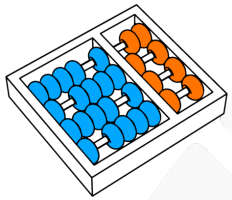
Exemplo (Invariante)

Em cada iteração da linha 7, vale:

1. $A[p \ldots k-1]$ *está ordenado,*
2. $A[p \ldots k-1]$ *contém itens de* $B[p \ldots i-1]$ *e de* $B[j+1 \ldots r]$,
3. $B[i] \geq A[k-1]$ *e* $B[j] \geq A[k-1]$.

Exercício: demonstre

- A invariante.
- A correção de INTERCALA.



Correção de MergeSort

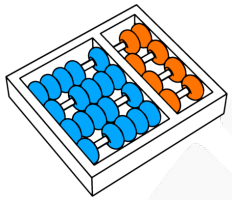
Base da indução:

- Para um vetor de tamanho 0 ou 1, o vetor não é alterado.
- Assim MergeSort está correto nesses casos.

Caso geral:

- Considere um vetor de tamanho $n \geq 2$.
- Suponha que MergeSort ordena vetores menores.
- Daí, após as chamadas recursivas, os subvetores $A[p \ldots q]$ e $A[q+1 \ldots r]$ estão ordenados.
- Como Intercala está correto, o vetor $A[p \ldots r]$ estará ordenado no final do algoritmo.

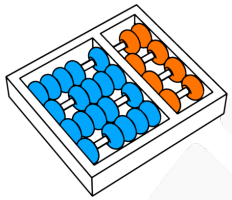
Projeto de algoritmos recursivos



Algoritmos recursivos

Resolvendo um problema com recursão

1. **Instâncias pequenas:** resolver diretamente.
2. **Instâncias grandes:**
   - Construir uma instância menor do mesmo problema.
   - Resolver essa subinstância recusivamente.
   - Encontrar uma solução para a instância original.



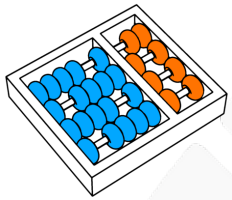
Projeto de algoritmos recursivos

Uma estratégia é:

1. Encontrar e definir um **SUBPROBLEMA** adequado.
2. Supor que sabemos resolver instâncias menores.
3. Construir um algoritmo recursivo para o subproblema.

O subproblema escolhido:

- Não precisar ser o problema original.
- Costuma ser uma variante ligeiramente **MAIS GERAL**.



Recursão e indução

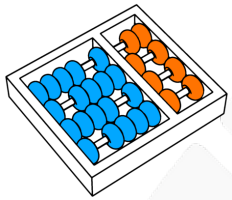
Este processo é análogo a **DEMONSTRAÇÕES POR INDUÇÃO**:

- **Caso básico:** instâncias pequenas.
- **Caso geral:** instâncias grandes:
  - A chamada recursiva corresponde à **HIPÓTESE DE INDUÇÃO**.
  - A construção da solução corresponde ao **PASSO DA INDUÇÃO**.

Algumas vantagens:

- A correção vem do próprio processo indutivo.
- A complexidade é expressa numa recorrência.

Alguns exemplos



Cálculo de polinômios

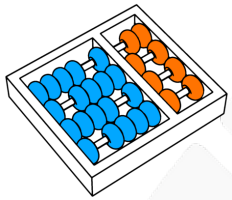
Problema (Problema)

Entrada: *Uma sequência de números reais* $a_n, a_{n-1}, \ldots, a_1, a_0$ *e um número real* x*.*

Saída: *O valor do polinômio:*

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- Este é um problema bem simples.
- Mas queremos fazer poucas adições e multiplicações.

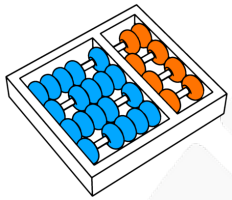


Primeira tentativa

Teorema (Hipótese de indução)

Dada uma sequência de números reais $a_{n-1}, \ldots, a_1, a_0$ *e um número real* x*, sabemos calcular o valor de* $P_{n-1}(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$.

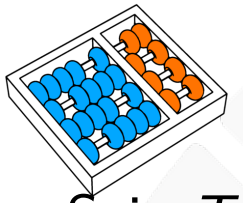
- **Caso base**: $n = 0$.
  - Devolvemos a_0.
- **Caso geral**: $n > 0$.
  - Calculamos $P_{n-1}(x)$ recursivamente.
  - Somamos $a_n x^n$ ao resultado obtido.



Algoritmo

Algoritmo: CALCULO-POLINOMIO(A, x)

```
1  se n = 0
2      y ← a_0
3  senão
4      A' ← a_{n−1}, ..., a_1, a_0
5      y' ← CALCULO-POLINOMIO(A', x)
6      xn ← 1
7      para i ← 1 até n
8          xn ← xn · x
9      y ← y' + a_n · xn
10 devolva y
```



Complexidade da primeira tentativa

Seja $T(n)$ o número de operações aritméticas realizadas:

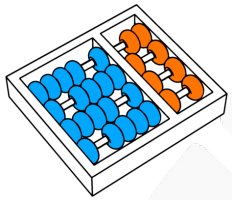
$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + (n+1) \text{ multiplicações} + 1 \text{ adição}, & n > 0. \end{cases}$$

Utilizando o método da iteração:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} [(i+1) \text{ multiplicações} + 1 \text{adição}] \\ &= (n+3)n/2 \text{ multiplicações} + n \text{ adições}. \end{aligned}$$

Observações:

- Pode-se diminuir o número de multiplicações.
- Para isso, calculamos x^n com exponenciação rápida.



Segunda solução indutiva

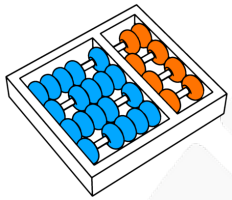
Melhorando:

- No primeiro algoritmo, recalculamos potências de x.
- Podemos reaproveitar o cálculo de x^{n-1}.
- Para isso, **REFORÇAMOS** a hipótese de indução.

Teorema (Hipótese de indução reforçada)

Sabemos calcular $P_{n-1}(x) = a_{n-1}x^{n-1} + \cdots + a_1x + a_0$ *e também o valor de* x^{n-1}.

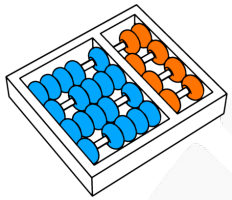
- Podemos fazer hipóteses mais fortes sobre a recursão.
- Mas agora precisamos também devolver o valor de x^n.
- Precisamos resolver um problema **MAIS GERAL**.



Segundo algoritmo

Algoritmo: CALCULO-POLINOMIO(A, x)

```
1 se n = 0
2     y ← a_0
3     xn ← 1
4 senão
5     A' ← a_{n−1}, ..., a_1, a_0
6     y', x' ← CALCULO-POLINOMIO(A', x)
7     xn ← x · x'
8     y ← y' + a_n · xn
9 devolva y, xn
```



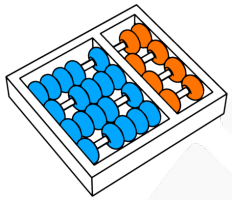
Complexidade do segundo algoritmo

Para essa versão temos:

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + 2 \text{ multiplicações} + 1 \text{ adição}, & n > 0. \end{cases}$$

A solução da recorrência é:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} (2 \text{ multiplicações} + 1 \text{ adição}) \\ &= 2n \text{ multiplicações} + n \text{ adições}. \end{aligned}$$



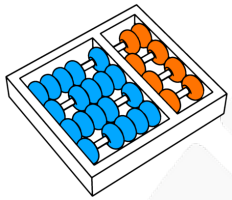
Terceira solução indutiva

Existem várias escolhas para a definição do subproblema:

Teorema (Outra hipótese de indução)

Sabemos calcular $P'_{n-1}(x) = a_n x^{n-1} + a_{n-1} x^{n-2} \cdots + a_1$.

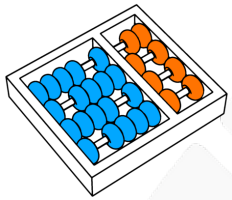
- Essa versão é mais simples.
- E precisamos de menos operações.



Algoritmo melhorado

Algoritmo: CALCULO-POLINOMIO(A, x)

```
1 se n = 0
2     y ← a_0
3 senão
4     A' ← a_{n-1}, ..., a_1
5     y' ← CALCULO-POLINOMIO(A', x)
6     y ← x · y' + a_0
7 devolva y
```



Complexidade do algoritmo melhorado

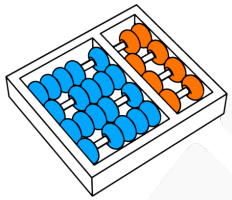
Finalmente, temos:

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + 1 \text{ multiplicação} + 1 \text{ adição}, & n > 0. \end{cases}$$

A solução é:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} (1 \text{ multiplicação} + 1 \text{ adição}) \\ &= n \text{ multiplicações} + n \text{ adições}. \end{aligned}$$

Esse algoritmo chamado de **regra de Horner**.



Fator de balanceamento em árvores binárias

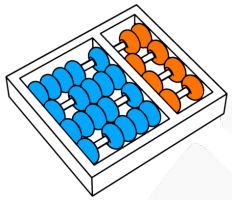
Vamos relembrar árvore binárias de busca:

- Cada nó v tem uma chave.
- A subárvore esquerda tem chaves menores.
- A subárvore direita tem chaves maiores.

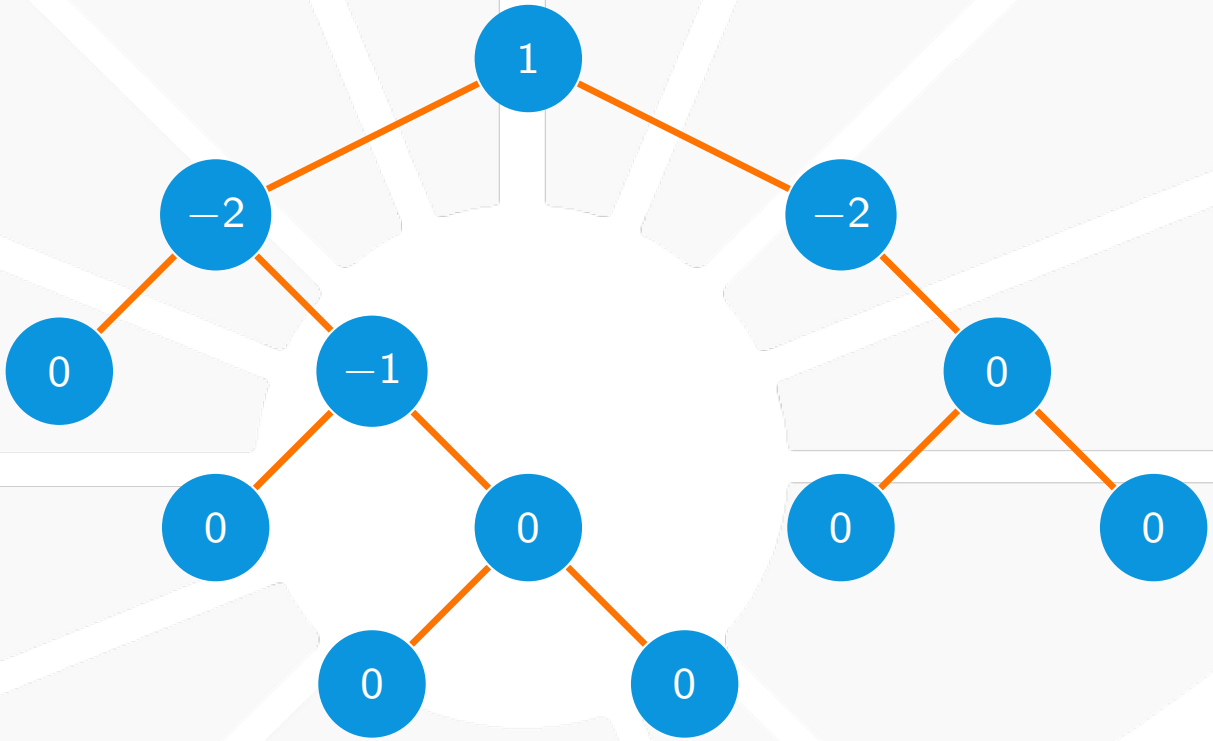
Definição

*O **fator de balanceamento** (f.b.) de um nó v é a diferença entre as alturas das subárvores esquerda e direita.*

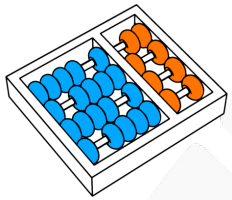
- A árvore é **balanceada** se todo nó tem f.b. limitado.
- Em uma árvore AVL, todo nó tem f.b. $-1, 0$ ou $+1$.
- Uma árvore vazia tem f.b. igual a zero.



Exemplo



Uma árvore e o f.b. de cada nó.



Calculando o fator de balanceamento

Problema

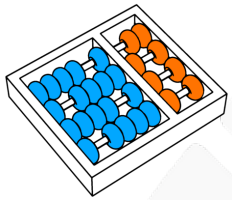
Entrada: *Uma árvore binária A com n nós.*

Saída: *Fatores de balanceamento de cada nó de A.*

Vamos projetar o algoritmo indutivamente.

Teorema (Hipótese de indução)

Sabemos como calcular fatores de balanceamento de árvores com menos que n nós.



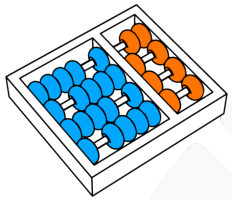
Projeto recursivo

Uma dificuldade:

- A definição de f.b. **NÃO** é recursiva!
- O f.b. de um nó depende das alturas das subárvores.
- Mas sabemos apenas o f.b. das subárvores.
- Conclusão: é necessária uma h.i. mais forte!

Teorema (Nova hipótese de indução)

Para um $n > 0$ qualquer, sabemos como calcular fatores de balanceamento e alturas de árvores com menos que n nós.

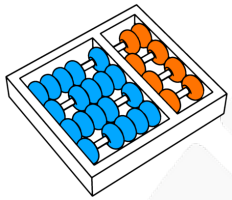


Encontrando a solução do subproblema

1. Se $n = 0$, a árvore a árvore é vazia, então:
   - O f.b. é igual 0.
   - A altura é igual 0.
2. Se $n > 0$, a árvore tem pelo menos um nó:
   - Recursivamente, obtemos a altura da árvore esquerda h_e.
   - Do mesmo modo, obtemos a altura da árvore direita h_d.
   - Por definição, f.b. é $h_e - h_d$.
   - A altura é $\max(h_e, h_d) + 1$.

Observações:

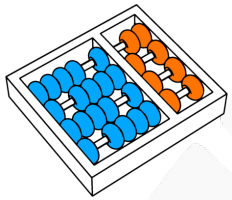
- Repare que não precisamos de f.b. das subárvores.
- Mas obtemos a altura recursivamente.
- De novo, fortalecer a h.i. simplificou o projeto do algoritmo!



Pseudocódigo do algoritmo

Algoritmo: FATOR-ALTURA(A)

```
1 se n = 0
2     f ← 0
3     h ← 0
4 senão
5     f_e, h_e ← FATOR-ALTURA(A_e)
6     f_d, h_d ← FATOR-ALTURA(A_d)
7     f ← h_e − h_d
8     h ← max(h_e, h_d) + 1
9 devolva f, h
```

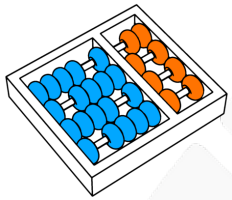


Complexidade

Seja $T(n)$ o número de operações executadas no pior caso,

$$T(n) = \begin{cases} \Theta(1), & n = 0 \\ T(n_e) + T(n_d) + \Theta(1), & n > 0, \end{cases}$$

onde n_e, n_d são os números de nós das subárvores.



Estimando o pior caso

- O pior caso parece acontecer se $n_e = 0$ e $n_d = n - 1$
- Nesse caso, temos

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(1) \\ &= T(n-2) + 2T(0) + 2\Theta(1) \\ &\vdots \\ &= T(0) + nT(0) + n\Theta(1) \\ &= (n+1)\Theta(1) + n\Theta(1) = \Theta(n). \end{aligned}$$

Exercício: argumente que este é, de fato, um pior caso. Qual seria um melhor caso?

Correção e projeto de algoritmos recursivos

MO417 - Complexidade de Algoritmos I

03/24

7

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP