Projeto e Análise de Algoritmos (2022-1)

Seleção de exercícios para a primeira avaliação

Prof. Flávio L. C. de Moura

21 de julho de 2022

1. Prove que o algoritmo BubbleSort a seguir é correto.

```
1 for i = 0 to n − 2 do
2 |  for j = n − 1 downto i + 1 do
3 |  |  if A[j] < A[j − 1] then
4 |  |  |  swap A[j] and A[j − 1];
5 |  |  end
6 |  end
7 end
```

Algorithm 1: BubbleSort($A[0..n-1]$)

2. $f(n) = O(g(n))$ se, e somente se $g(n) = \Omega(f(n))$;

3. $T(1) = 1$, $T(n) = 2T(n-1) + 1$, $n \geq 2$

4. Sejam l o número de folhas em uma árvore binária, e h sua altura. Prove que $l \leq 2^h$.

5. O algoritmo de ordenação por inserção binária é uma variação do algoritmo de ordenação por inserção, onde na j-ésima iteração, se insere a chave que ocupa a posição j do vetor de entrada, no subvetor $A[0..j-1]$ previamente ordenado, usando o método de inserção binária ao invés da inserção sequencial.

```
1 for j = 1 to n − 1 do
2 |  pos ← BinarySearchPos(A[0..n − 1], 0, j − 1, A[j]);
3 |  while j > pos do
4 |  |  swap A[j − 1] and A[j];
5 |  |  j ← j − 1;
6 |  end
7 end
```

Algorithm 2: BInsertionSort($A[0..n-1]$)

onde

1.


```
1 for i = 0 to n − 2 do
2 |  for j = n − 1 downto i + 1 do
3 |  |  if A[j] < A[j − 1] then
4 |  |  |  swap A[j] and A[j − 1];
5 |  |  end
6 |  end
7 end
```

Algorithm 1: BubbleSort($A[0..n-1]$)

Invariante p/ o laço interno:

Antes de cada iteração o menor elemento do subvetor $A[j..n-1]$ está na posição j.

Inicialização: Antes da primeira iteração ($j = n-1$) o menor elemento do subvetor $A[n-1]$ está na posição $n-1$. Trivial

```
1  while l ≤ r do
2  |  m = ⌊(l + r)/2⌋;
3  |  if key = A[m] then
4  |  |  return m;
5  |  end
6  |  else
7  |  |  if key < A[m] then
8  |  |  |  r ← m − 1;
9  |  |  end
10 |  |  else
11 |  |  |  l ← m + 1;
12 |  |  end
13 |  end
14 end
15 return l
```

Algorithm 3: BinarySearchPos($A[0..n-1], l, r, key$)

Faça a análise da complexidade do pior caso para este algoritmo.

Manutenção: Suponha que antes da k-ésima iteração ($j = n-k$) o menor elemento do subvetor $A[n-k..n-1]$ está na posição $n-k$. Durante a k-ésima iteração os elementos $A[n-k]$ e $A[n-k-1]$ são comparados. Temos 2 casos:

① $A[n-k] < A[n-k-1]$ (linha 3):

Neste caso, $A[n-k]$ é colocado na posição $n-k-1$, e portanto o menor elemento do subvetor $A[n-k-1..n-1]$ está na posição $n-k-1$.

② $A[n-k] \geq A[n-k-1]$: Neste caso, $A[n-k-1]$ é o menor elemento do subvetor $A[n-k-1..n-1]$ e está posição $A[n-k-1]$.

Em ambos os casos a invariante está satisfeita.

Invariante para o laço externo:

> Antes de cada iteração, o subvetor $A[0..i-1]$ está ordenado com os i menores elementos do vetor A.

Inicialização: Precisamos mostrar que antes da primeira iteração ($i=0$), o subvetor $A[0..i-1]$ está ordenado com os i menores elementos do vetor A. Mas esta afirmação é trivial porque o subvetor $A[0..-1]$ é vazio e $i=0$.

Manutenção: Suponha que antes da k-ésima iteração ($i=k-1$), o subvetor $A[0..k-2]$ está ordenado com os $(k-1)$ menores elementos do vetor A. Durante a k-ésima iteração, temos pela invariante do laço interno que o menor elemento do subvetor $A[k-1..n-1]$ estará na posição $k-1$. Ou seja, o subvetor $A[0..k-1]$ está ordenado com os k menores elementos do vetor A.

Terminação: Ao final da última iteração do laço externo, temos $i = n-1$, e a invariante nos diz que o subvetor $A[0..n-2]$ está ordenado com os $(n-1)$ menores elementos do vetor A. Então $A[n-1]$

Terminação: Ao final da última iteração, temos $j=i$. Concluímos que o menor elemento do subvetor $A[i..n-1]$ está na posição i.

é maior ou igual a todos os outros elementos de A, de forma que A está ordenado com os mesmos elementos do vetor original. Isto é, o algoritmo é correto. ☐

2. $f(n) = O(g(n))$ se, e somente se $g(n) = \Omega(f(n))$;

($\Rightarrow$) Suponha que $f(n) = O(g(n))$, i.e. existem constantes positivas c e n_0 tais que $f(n) \leq c \cdot g(n)$, $\forall n \geq n_0$. Logo, $g(n) \geq \frac{1}{c} \cdot f(n)$, $\forall n \geq n_0$, i.e. $g(n) = \Omega(f(n))$.

($\Leftarrow$) Suponha que $g(n) = \Omega(f(n))$, i.e. existem constantes positivas c e n_0 tais que $g(n) \geq c \cdot f(n)$, $\forall n \geq n_0$. Logo, $f(n) \leq \frac{1}{c} \cdot g(n)$, $\forall n \geq n_0$. Portanto, $f(n) = O(g(n))$. ☐

3. $T(1) = 1$, $T(n) = 2T(n-1) + 1$, $n \geq 2$

$$T(n) = 2 \cdot \underline{T(n-1)} + 1$$
$$= 2 \cdot (2 \cdot T(n-2) + 1) + 1$$
$$= 2^2 \cdot \underline{T(n-2)} + 2 + 1$$
$$= 2^2 \cdot (2 \cdot T(n-3) + 1) + 2 + 1$$
$$= 2^3 \cdot T(n-3) + 2^2 + 2 + 1$$
$$= \cdots = 2^{n-1} \cdot T(1) + 2^{n-2} + 2^{n-3} + \cdots + 2^2 + 2 + 1$$
$$= 2^{n-1} + 2^{n-2} + \cdots + 2 + 1$$
$$= \sum_{i=0}^{n-1} 2^i = 2^n - 1.$$

$\Rightarrow \boxed{T(n) = 2^n - 1}$

Indução em n:

$\boxed{n = 1}$ $T(1) = 2^1 - 1 = 1$ ✓

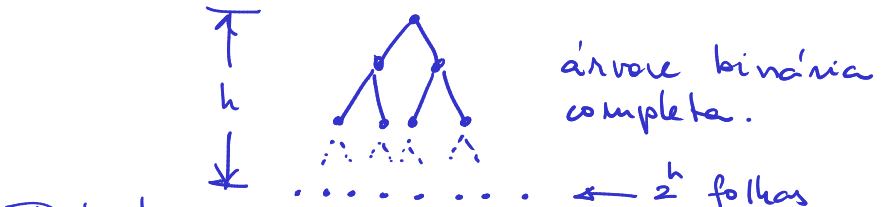
$\boxed{n \geq 2}$ $T(n) = 2 \cdot T(n-1) + 1$
$$\overset{\text{h.i}}{=} 2 \cdot (2^{n-1} - 1) + 1$$
$$= 2^n - 2 + 1 = 2^n - 1.$$

☐

4. Sejam l o número de folhas em uma árvore binária, e h sua altura. Prove que $l \leq 2^h$.

Em uma árvore binária completa temos 2^h folhas na altura h:

h	folhas
0	1
1	2^1
2	2^2
3	2^3
⋮	⋮
h	2^h



Portanto $l \leq 2^h$.

5. O algoritmo de ordenação por inserção binária é uma variação do algoritmo de ordenação por inserção, onde na j-ésima iteração, se insere a chave que ocupa a posição j do vetor de entrada, no subvetor $A[0..j-1]$ previamente ordenado, usando o método de inserção binária ao invés da inserção sequencial.

```
1 for j = 1 to n − 1 do
2 |  pos ← BinarySearchPos(A[0..n − 1], 0, j − 1, A[j]);
3 |  while j > pos do
4 |  |  swap A[j − 1] and A[j];
5 |  |  j ← j − 1;
6 |  end
7 end
```

Algorithm 2: BInsertionSort($A[0..n-1]$)

onde

```
1  while l ≤ r do
2  |  m = ⌊(l + r)/2⌋;
3  |  if key = A[m] then
4  |  |  return m;
5  |  end
6  |  else
7  |  |  if key < A[m] then
8  |  |  |  r ← m − 1;
9  |  |  end
10 |  |  else
11 |  |  |  l ← m + 1;
12 |  |  end
13 |  end
14 end
15 return l
```

Algorithm 3: BinarySearchPos($A[0..n-1], l, r, key$)

Faça a análise da complexidade do pior caso para este algoritmo.

Temos a complexidade do pior caso de BinarySearchPos quando key não ocorre em $A[0..n-1]$. Neste caso, o laço while (linhas 1 – 14) será executado tantas vezes quanto for possível dividir o vetor A por 2, ou seja $T_w(n) = O(\lg n)$.

Portanto, a complexidade de BInsertionSort(A) no pior caso é

$$T_w(n) = \sum_{i=1}^{n-1} \left(O((\lg n) + n) \right) = \sum_{i=1}^{n-1} O(\max(\lg n, n)) =$$

$$\sum_{i=1}^{n-1} O(n) = O((n-1).n) = O(n^2).$$