Universidade Federal de Ouro Preto
PCC104 - Projeto e Análise de Algoritmos
Prof. Rodrigo Silva

Dividir e Conquistar

Sara Câmara

8 de maio de 2023

1 Questão

Problema: Implemente um algoritmo de divisão e conquista para encontrar a posição do maior elemento de um arranjo de n elementos.

Para cada implementação, apresentar a análise de complexidade de tempo do algoritmo.

Solução:

- Implementação:

```
def maximo(array, begin, end):
    if end - begin <= 1:
        if array[begin] > array[end]:
            return begin
        else:
            return end
    else:
        mid = begin + end // 2
        max1 = maximo(array, begin, mid)
        max2 = maximo(array, mid+1, end)
        if array[max1] > array[max2]:
            return max1
        else:
            return max2
```

- Operação básica: Comparação para encontrar o maior elemento

```
array[max1] > array[max2]
```

a) Qual será a saída do método quando vários elementos do arranjo tiverem o maior valor?

Fazendo a análise da implementação é possível concluir que o código trará o maior *index* dentre as posições com o maior elemento. O trecho de código em destaque abaixo, ilustra essa interpretação, pois para o caso em análise a segunda opção (*else*) é executada:

```
if array[begin] > array[end]:
    return begin
else:
    return end
```

b) Defina e resolva a relação de recorrência para o método proposto.

- Função de custo:

$$C(n) = C\Big(\Big\lceil\frac{n}{2}\Big\rceil\Big)) + C\Big(\Big\lfloor\frac{n}{2}\Big\rfloor\Big) + 1 \quad \forall\ n > 1,$$
$$C(1) = 0$$

- Cálculo da função de recorrência:

Assumindo que $n = 2^k$, pois também resulta em uma resposta correta sobre a ordem de crescimento para todos os valores de n, temos que:

$$\begin{aligned}
C(2^k) &= 2C(2^{k-1}) + 1 \\
&= 2[2C(2^{k-2}) + 1] + 1 = 2^2C(2^{k-2}) + 2 + 1 \\
&= 2^2[2C(2^{k-3}) + 1] + 2 + 1 = 2^3C(2^{k-3}) + 2^2 + 2 + 1 \\
&... \\
&= C(2^{k-i}) + i \\
&... \\
&= 2^iC(2^{k-i}) + 2^{i-1} + 2^{i-2} + ... + 1 \\
&= 2^kC(2^{k-k}) + 2^{k-1} + 2^{k-2} + ... + 1 \\
&= 2^k - 1 = n - 1
\end{aligned}$$

- Eficiência (O ou Θ):
  - Pior caso: $C(n) = \Theta(n)$
  - Caso médio: $C(n) = \Theta(n)$
  - Melhor caso: $C(1) = \Theta(1)$

Podemos verificar que $C(n) = n - 1$ satisfaz, a recorrência para todo valor de $n > 1$, substituindo-o na equação de recorrência e considerando separadamente os casos par ($n = 2i$) e ímpar ($n = 2i + 1$). Seja $n = 2i$ onde $i > 0$. Então, o lado esquerdo da equação de recorrência é $n - 1 = 2i - 1$. O lado direito é:

$$\begin{aligned}
C(n) &= C\Big(\Big\lceil\frac{n}{2}\Big\rceil\Big) + C\Big(\Big\lfloor\frac{n}{2}\Big\rfloor\Big) + 1 \\
&= C\Big(\Big\lceil\frac{2i}{2}\Big\rceil\Big) + C\Big(\Big\lfloor\frac{2i}{2}\Big\rfloor\Big) + 1 \\
&= 2C(i) + 1 = 2(i - 1) + 1 = 2i - 1
\end{aligned}$$

Seja $n = 2i+1$ onde $i > 0$. Então, o lado direito da equação de recorrência é $n-1 = 2i$. O lado esquerdo é:

$$\begin{aligned} C(n) &= C\Big(\Big\lceil \frac{n}{2} \Big\rceil\Big) + C\Big(\Big\lfloor \frac{n}{2} \Big\rfloor\Big) + 1 \\ &= C\Big(\Big\lceil \frac{2i+1}{2} \Big\rceil\Big) + C\Big(\Big\lfloor \frac{2i+1}{2} \Big\rfloor\Big) + 1 \\ &= C(i+1) + C(i) + 1 = (i+1-1) + (i-1) + 1 = 2i \end{aligned}$$

c) Como este algoritmo se compara com uma solução força bruta?

Por meio da análise dos algoritmos é possível notar que as classes de eficiência dos dois algoritmos é a mesma e assim, dado um $array(n)$ de entrada, os dois algoritmos requerem o mesmo mesmo número de comparações de chaves. Porém no caso n ser impar o valor de operações é acrescido de 1 para o algoritmo de dividir e conquistar e também existe uma sobrecarga associada a chamadas recursivas.

2 Questão

Problema: Implemente o algoritmo *MergeSort*. O *MergeSort* é um algoritmo estável? Explique.

Para cada implementação, apresentar a análise de complexidade de tempo do algoritmo.

Solução:

a) Implementação:

```
def merge_sort(array, begin, end):
    if begin < end:
        mid = (begin + end) // 2
        merge_sort(array, begin, mid)
        merge_sort(array, mid + 1, end)
        merge(array, begin, mid, end)

def merge(array, begin, mid, end):
    i = begin
    j = mid + 1
    aux = []
    while i <= mid and j <= end:
        if array[i] < array[j]:
            aux.append(array[i])
            i += 1
        else:
            aux.append(array[j])
            j += 1
    aux += array[i:mid+1]
    aux += array[j:end+1]
    array[begin:end+1] = aux
```

- Estabilidade do algoritmo *MergeSort*:

A implementação de *MergeSort* feita não é estável.
O *MergeSort* é estável, desde que sua implementação utilize a comparação $\leq$ na mesclagem. Assim, é possivel analisar os seguintes casos tenhamos dois elementos com o mesmo valor nas posições i e j, $i < j$ em uma submatriz antes que suas duas metades (ordenadas) sejam mescladas:

- Se esses dois elementos estiverem na mesma metade da submatriz, sua ordenação relativa permanecerá a mesma após a mesclagem, pois os elementos da mesma metade são processados pela operação de mesclagem no modo *FIFO*.
- Caso em que $A[i]$ está na primeira metade e $A[j]$ está na segunda metade. $A[j]$ é colocado na nova matriz depois que a primeira metade fica vazia (e, portanto, $A[i]$ já foi copiado para a nova matriz) ou depois de ser comparado com alguma chave $K > A[j]$ da primeira metade.
- No último caso, como a primeira metade é classificada antes do início da mesclagem, $A[i] = A[j] < k$ não pode estar entre os elementos não processados da primeira metade. Assim, no momento dessa comparação, $A[i]$ já foi copiado para a nova matriz e, portanto, precederá $A[j]$ após a conclusão da operação de mesclagem.

b) Operação básica: Comparação para encontrar o menor elemento

```
array[i] < array[j]
```

Análise para o caso pior:

c) Função de custo:

$$C_w(n) = 2C_w\left(\frac{n}{2}\right) + n - 1 \quad \forall\ n > 1,$$
$$C_w(1) = 0$$

d) Cálculo da função de recorrência:

Assumindo que $n = 2^k$, pois também resulta em uma resposta correta sobre a ordem de crescimento para todos os valores de n, temos que:

$$\begin{aligned}
C_w(2^k) &= 2C_w(2^{k-1}) + 2^k - 1 \\
&= 2[2C_w(2^{k-2}) + 2^{k-1} - 1] + 2^k - 1 = 2^2C_w(2^{k-2}) + 2\ .\ 2^k - 2 - 1 \\
&= 2^2[2C_w(2^{k-3}) + 2^{k-2} - 1] + 2\ .\ 2^k - 2 - 1 = 2^3C_w(2^{k-3}) + 3\ .\ 2^k - 2^2 - 2 - 1 \\
&... \\
&= 2^iC_w(2^{k-i}) + i2^k - 2^{i-1} - 2^{i-2} - ... - 1 \\
&... \\
&= 2^kC_w(2^{k-k}) + k2^k - 2^{k-1} - 2^{k-2} - ... - 1 \\
&= k2^k - (2^k - 1) = n\ log\ n - n + 1
\end{aligned}$$

Análise para o caso melhor:

c) Função de custo:

$$C_b(n) = 2C_b\left(\frac{n}{2}\right) + \frac{n}{2} \quad \forall\ n > 1,$$
$$C_b(1) = 0$$

d) Cálculo da função de recorrência:

Assumindo que $n = 2^k$, pois também resulta em uma resposta correta sobre a ordem de crescimento para todos os valores de n, temos que:

$$\begin{aligned}
C_b(2^k) &= 2C_b(2^{k-1}) + 2^{k-1} \\
&= 2[2C_b(2^{k-2}) + 2^{k-2}] + 2^{k-1} = 2^2C_b(2^{k-2}) + 2^{k-1} \\
&= 2^2[2C_b(2^{k-3}) + 2^{k-3}] + 2^{k-1} + 2^{k-1} = 2^3C_b(2^{k-3}) + 2^{k-1} + 2^{k-1} \\
&... \\
&= 2^iC_b(2^{k-i}) + i2^{k-1} \\
&... \\
&= 2^kC_b(2^{k-k}) + k2^{k-1} = k2^{k-1} = \frac{1}{2}n\ log\ n
\end{aligned}$$

Análise para o caso médio:

c) Função de custo: Se $n > 1$, o algoritmo copia $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n$ elementos primeiro e depois faz mais n movimentos durante o estágio de mesclagem. Isso produz a seguinte recorrência para o número de movimentos $C_{avg}(n)$:

$$C_{avg}(n) = 2C_{avg}\left(\frac{n}{2}\right) + 2n \quad \forall\ n > 1,$$
$$C_{avg}(1) = 0$$

d) Cálculo da função de recorrência:

De acordo com o Teorema Mestre, sua solução está em $\Theta(n \log n)$ - a mesma classe estabelecida pela análise do número de comparações de chaves.

e) Eficiência (O ou Θ):

- Pior caso: $C(n) = \Theta(n \log n)$
- Caso médio: $C(n) = \Theta(n \log n)$
- Melhor caso: $C(n) = \Theta(n \log n)$

Pelo Teorema Mestre podemos analisar que a função de recorrência têm a seguinte forma (com a, c e k constantes):

$$F(n) = aF\left(\frac{n}{2}\right) + c.n^k$$

Teorema: Sejam a um número natural não-nulo, k um número natural, e c um número real positivo. Seja F uma função que leva números naturais em números reais positivos e satisfaz a recorrência para $n = 2^1, 2^2, 2^3, \ldots$ Suponha que F é assintoticamente não decrescente, ou seja, que existe n_1 tal que $F(n) \leq F(n+1)$ para todo $n \leq n1$. Nessas condições,

- se $\log\ a\ > k$ então $F(n) = \Theta(n^{\log a})$,
- se $\log\ a\ = k$ então $F(n) = \Theta(n^k \log n)$,
- se $\log\ a\ < k$ então $F(n) = \Theta(n^k)$.

Generalizando o Teorema para obter a solução assintótica da recorrência.

$$F(n) = aF\left(\frac{n}{b}\right) + c.n^k$$

- se $\frac{\log\ a}{\log\ b} > k$ então $F(n) = \Theta(n^{\frac{\log\ a}{\log\ b}})$,
- se $\frac{\log\ a}{\log\ b} = k$ então $F(n) = \Theta(n^k \log n)$,
- se $\frac{\log\ a}{\log\ b} < k$ então $F(n) = \Theta(n^k)$.

Como $n = 2^k \Rightarrow k = \log_2 n$ e adotando o caso base $n = 2$, temos $k = 1$. E analisando as funções de custo temos que $a = 2$ e $b = 2$. Assim, para os casos pior, médio e melhor do *MergeSort*:

$$\frac{\log_2\ a}{\log_2\ b} = k \Rightarrow \frac{\log_2\ 2}{\log_2\ 2} = 1 \quad \therefore \ \in \ \Theta(n^k \log n)$$

3 Questão

Problema: Implemente o algoritmo *QuickSort*. O *QuickSort* é um algoritmo estável? Explique.

Para cada implementação, apresentar a análise de complexidade de tempo do algoritmo.

Solução:

a) Implementação:

```
def quick_sort(array, begin, end):
    if begin < end:
        p = partition(array, begin, end)
        quick_sort(array, begin, p-1)
        quick_sort(array, p+1, end)

def partition(array, begin, end):
    pivot = array[end]
    i = begin
    for j in range(begin, end):
        if array[j] <= pivot:
            array[j], array[i] = array[i], array[j]
            i += 1
    array[i], array[end] = array[end], array[i]
    return i
```

- Estabilidade do algoritmo *QuickSort*:

A implementação de *QuickSort* feita não é estável pois utiliza a comparação $\leq$ na troca dos elementos. Assim, é possível analisar que para um vetor de dois elementos de valores iguais nas posições i e j, $i < j$ quando comparados terão suas posições relativas trocadas.

- Operação básica: Comparação para encontrar o menor elemento em relação ao pivô.

```
array[j] <= pivot
```

Análise para o caso melhor:

c) Função de custo:

$$C(n) = C\Big(\left\lceil \frac{1}{2}(n-1) \right\rceil \Big) + C\Big(\left\lfloor \frac{1}{2}(n-1) \right\rfloor \Big) + n - 1$$

d) Cálculo da função de custo:

$$C_b(n) = 2C_b\Big(\frac{n}{2}\Big) + n \quad \forall\, n > 1,$$
$$C_b(1) = 0$$

De acordo com o Teorema Mestre, $C_b(n)\ \in\ (n\ \log_2\ n)$ resolvendo-o exatamente para $n = 2^k$ resulta em $C_b(n)\ =\ (n\ \log_2\ n)$.

e) Eficiência (O ou Θ):

$$C_b(n)\ \in\ \Theta(n\ log_2\ n)$$

Generalizando o Teorema para obter a solução assintótica da recorrências.

Como $n = 2^k \Rightarrow k = \log_2 n$ e adotando o caso base $n = 2$, temos $k = 1$. E analisando as funções de custo temos que $a = 2$ e $b = 2$. Assim, para melhor do *QuickSort*:

$$\frac{\log_2\ a}{\log_2\ b} = k \Rightarrow \frac{\log_2\ 2}{\log_2\ 2} = 1 \quad \therefore\ \in\ \Theta(n^k \log n)$$

Análise para o caso pior:

c) Função de custo:

$$C_w(n)\ =\ C_w(n-1) + n - 1$$

d) Cálculo da função de custo:

$$\begin{aligned}
C_w(n)\ &=\ C_w(n-1) + n - 1\\
&= C_w(n-2)(n-2) + (n-1)\\
&= C_w(n-3) + (n-3) + (n-2) + n - 1\\
&= C_w(n-4) + (n-4) + (n-3) + (n-2) + (n-1)\\
&...\\
&= C_w(0) + 0 + 1 + 2 + ... + (n-2) + (n-1)\\
&= \frac{n(n-1)}{2}
\end{aligned}$$

e) Eficiência (O ou Θ):

$$\frac{n(n-1)}{2} \;\in\; \Theta(n^2)$$

Por definição, para provar esta afirmação, devemos achar constantes $c_1 > 0$, $c_2 > 0$,

tais que: $c_1 n^2 \leq \frac{n(n-1)}{2} \;\in\; (n^2) \leq c_2 n^2 \;\; \forall \; n \; \geq \; 1$

Essa inequação se mantém verdadeira para $c_1 = 1$, $c_2 = 2$ e $n_0 = 1$.

Análise para o caso médio:

c) Função de custo:

$$C_{avg}(n) = \frac{1}{2}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \forall \; n \; > \; 1,$$
$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0$$

d) Cálculo da função de custo:

$$C_{avg}(n) \approx 2n \;\; \ln \;\; n \approx 1.39n \; log_2 \; n$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0$$

Sua solução é mais complicada do que as análises do pior e do melhor caso, porém em média, o *QuickSort* faz apenas 39% mais comparações do que no melhor caso.

e) Eficiência (O ou Θ):

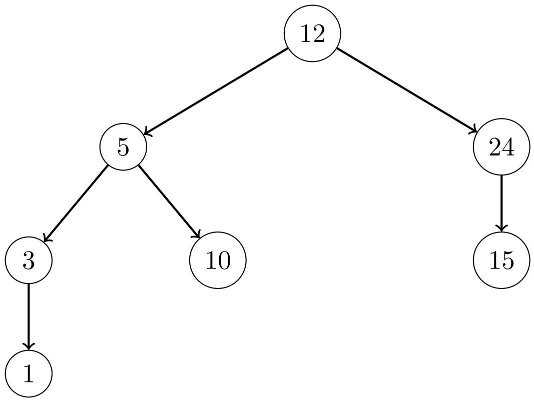
$$C_{avg}(n) \;\in\; \Theta(n \; log_2 \; n)$$

Generalizando o Teorema para obter a solução assintótica da recorrências.

Como $n = 2^k \Rightarrow k = \log_2 n$ e adotando o caso base $n = 2$, temos $k = 1$. E analisando as funções de custo temos que $a = 2$ e $b = 2$. Assim, para melhor do *QuickSort*:

$$\frac{\log_2 \; a}{\log_2 \; b} = k \Rightarrow \frac{\log_2 \; 2}{\log_2 \; 2} = 1 \quad \therefore \; \in \; \Theta(n^k \log n)$$

Para o desenvolvimento e análise do algoritmo recursivo que encontra o tamanho de uma *Binary Tree* e também os caminhamentos *preorder*, *postorder* e *inorder* foi considerada a seguinte árvore binária:



$size\ =\ 7$

$preorder:\ 12,\ 5,\ 3,\ 1,\ 10,\ 24,\ 15$
$postorder:\ 1,\ 3,\ 10,\ 5,\ 15,\ 24,\ 12$
$inorder:\ 1,\ 3,\ 5,\ 10,\ 12,\ 15,\ 24$

---

4 Questão

Problema: Implemente um algoritmo recursivo que encontre o tamanho de uma árvore binária.

Para cada implementação, apresentar a análise de complexidade de tempo do algoritmo.

Solução:

a) Implementação:

```
def leaf_count(self):
    if self.left is None and self.right is None:
        return 1

    if self.left and self.right:
        return 1 + (self.left.leaf_count() + self.right.leaf_count())
    if self.left and not self.right:
        return 1 + self.left.leaf_count()
    if self.right and not self.left:
        return 1 + self.right.leaf_count()
```

- Operação básica: Soma recursiva do numero de folhas.

  ```
  1 + (self.left.leaf_count() + self.right.leaf_count())
  ```

c) Função de custo:

$$A(n(T))\ =\ A(n(T_{left})) + A(n(T_{right})) + 1\ \ \forall\ n\ >\ 1$$
$$A(0)\ =\ 0$$

d) Cálculo da função de custo:

Fazendo a análise da árvore apresentada, temos obtemos o grau de cada nó presente na árvore (quantidade de filhos que ele possui). Na figura, as folhas são os nós 1, 10 e 15. Os nós 3 e 24 têm grau 1, enquanto os nós 5 e 12 possuem grau 2. Os nós com graus 1 e 2 são chamados de nós internos.

Supondo que o número de nós externos x é sempre 1 a mais do que o número de nós internos n:

$$x = n + 1$$

Considerando o número total de nós, tanto internos quanto externos exceto a raiz, tem-se a equação:

$$2n + 1 = x + n$$

que implica na igualdade apresentada anteriormente.

A igualdade também se aplica a qualquer árvore binária completa não vazia e que por definição, cada nó tem zero ou dois filhos. Assim, para o algoritmo *LeafCount*, tem-se o número de operações para verificar a árvore:

$$C(n) = n + x = 2n + 1 \Rightarrow A(n) = n$$

e) Eficiência (O ou Θ):

$$A(n) \in \Theta(n)$$

Por indução, temos que nós internos ≥ 0. A etapa da base é verdadeira porque para $n = 0$ temos a árvore vazia cuja a árvore estendida tem 1 nó externo por definição. Para a etapa indutiva, vamos supor que:

$$x = k + 1$$

para qualquer árvore binária estendida com $0 \leq k < n$ nós internos. Seja T uma árvore binária com n nós internos e que n_L e x_L sejam os números de nós internos e externos na subárvore esquerda de T, e respectivamente, sejam n_R e x_R os números de nós internos e externos na subárvore direita de T. Como $n > 0$, T tem uma raiz, que é seu nó interno e, portanto:

$$n = n_L + n_R + 1$$

Como tanto $n_L < n$ quanto $n_R < n$, podemos usar a igualdade, assumida como correta para a subárvore esquerda e direita de T para obter o seguinte:

$$x = x_L + x_R = (n_L + 1) + (n_R + 1) = (n_L + n_R + 1) + 1 = n + 1$$

5 Questão

Problema: Implemente os caminhamentos *preorder*, *postorder* e *inorder* para árvores binárias.

Para cada implementação, apresentar a análise de complexidade de tempo do algoritmo.

Solução:

a) Implementação:

```
# Preorder traversal: raiz, subarvore esquerda e subarvore direita
def preorder(self):
    print(self.value)
    if self.left:
        self.left.preorder()
    if self.right:
        self.right.preorder()

# Postorder traversal: subarvore esquerda, subarvore direita e raiz
def postorder(self):
    if self.left:
        self.left.postorder()
    if self.right:
        self.right.postorder()
    print(self.value)

# Inorder traversal: subarvore esquerda, raiz e subarvore direita
def inorder(self):
    if self.left:
        self.left.inorder()
    print(self.value)
    if self.right:
        self.right.inorder()
```

a) Operação básica: Verificação se há elementos a esquerda ou direita na árvore.

```
    if self.left:
```

OU

```
    if self.right:
```

c) Função de custo e cálculo:

Para cada um dos caminhamentos implementados, o número de chamadas $C(n)$ feitas pelo algoritmo é igual ao número de nós, internos e externos, na árvore estendida. Portanto, de acordo com a fórmula:

$$C(n) = 2n + 1 \;\Rightarrow A(n) = n$$

d) Eficiência (O ou Θ):

$$A(n) \;\in\; \Theta(n)$$