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

Diminuir e Conquistar

Sara Câmara

24 de abril de 2023

1 Questão

Problema: Implemente o algoritmo de busca binária. E apresentar a análise de complexidade de tempo do algoritmo.

Solução:

a) Implementação:

```
def binary_search(sorted_array, target):
    index_min, index_max = 0, len(sorted_array)-1

    while index_min <= index_max:
        index_med = (index_min + index_max)//2

        if target == sorted_array[index_med]:
            return index_med
        elif target < sorted_array[index_med]:
            index_max = index_med - 1
        else:
            index_min = index_med + 1
    return None
```

b) Operação básica: Comparação para encontrar o elemento procurado.

```
    target == sorted_array[index_med]
```

c) Função de custo:

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

$$C_w(1) = 1$$

d) Cálculo da função de custo:

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:

$$
\begin{aligned}
C_w(2^k) &= C_w(2^{k-1}) + 1 \\
&= [C_w(2^{k-2}) + 1] + 1 = C_w(2^{k-2}) + 2 \\
&= [C_w(2^{k-3}) + 2] + 1 = C_w(2^{k-3}) + 3 \\
&... \\
&= C_w(2^{k-i}) + i \\
&... \\
&= C_w(2^{k-k}) + k = C_w(2^0) + k \\
&= C_w(1) + k = k + 1
\end{aligned}
$$

$$
\begin{aligned}
&C_w(2^k) = k + 1 \\
&C_w(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil
\end{aligned}
$$

e) Eficiência (O ou Θ):

Pior caso: $C_w = \Theta(log_2 n)$

Caso médio: $C_{avg} \approx \Theta(log_2 n)$

Melhor caso: $C = \Theta(1)$

Para provar a afirmação de eficiência para o pior caso por limites, temos que:

$$\lim_{n\to\infty} \frac{log_2(n) + 1}{log_2(n)} = \lim_{n\to\infty} \frac{log_2(n)}{log_2(n)} + \frac{1}{log_2(n)} = \lim_{n\to\infty} 1 + \frac{1}{\cancelto{0}{log_2(\infty)}} = 1$$

O resultado constante 1 implica que $log_2(n) + 1$ tem a mesma ordem de crescimento que $log_2(n)$ e $f(n) \in \Theta g(n)$.

2 Questão

Problema: Implemente o método *Interpolation Search.* E apresentar a análise de complexidade de tempo do algoritmo.

Solução:

a) Implementação:

```
def interpolation_search(arr, t):
    imin, imax = 0, len(arr)-1

    while imin <= imax and arr[imin] <= t <= arr[imax]:
        pos = imin + ((t - arr[imin]) * (imax - imin) // (arr[imax] - arr[imin]))
        if t == arr[pos]:
            return pos
        elif t < arr[pos]:
```

```
            imax = pos - 1
        else:
            imin = pos + 1
    return None
```

b) Operação básica: Comparar se o item atual é o item procurado.

```
target == sorted_array[position]
```

c) Função de custo:

$$C_{avg} = log(log(n))$$

d) Cálculo da função de custo:

Sabendo que a complexidade da busca binária é dada por $\log_2 n$ e calculando para o caso base 1, temos:

$$1 = \frac{n}{2^x} \Rightarrow (2^x)1 = \frac{n}{2^x}(2^x) \Rightarrow 2^x = n \Rightarrow \log_2(2^x) = \log_2(n)$$

$$x\log_2(2) = \log_2(n) \Rightarrow x\cancelto{1}{\log_2(2)} = \log_2(n) \Rightarrow x = \log_2(n) \Leftrightarrow n = 2^x$$

Assim podemos assumir que para o algoritmo de interpolação a constante C base pode ser calculada pela fórmula da posição:

$$x = l + \lfloor\frac{(v - A[l])(r-l)}{A[r] - A[l]}\rfloor$$

Para simplificar reescrevemos a fórmula e substituímos em $1 = C^x n$:

$$1 = (\frac{k-L}{R-L})^x n \Rightarrow (R-L)^x = n(k-L)^x \Rightarrow \log_2(R-L)^x = \log_2(n) + \log_2(k-L)^x$$

$$x\log_2(R-L) - x\log_2(k-L) = \log_2(n) \Rightarrow x[\log_2(R-L) - \log_2(k-L)] = \log_2(n)$$

$$x\log_2(\frac{R-L}{k-L}) = \log_2(n) \Rightarrow x = \frac{1}{\log_2(\frac{R-L}{k-L})}.\log_2(n) \Rightarrow x = \log_{(}\frac{k-L}{R-L}).2.\log_2(n)$$

Com isso:

$$x = \log_C 2.\log_2 n \Rightarrow x = log(log_2(n))$$

e) Eficiência (O ou Θ):

Pior caso: $C_w = \Theta(n)$

Caso médio: $C_{avg} = \Theta(log(log(n))$

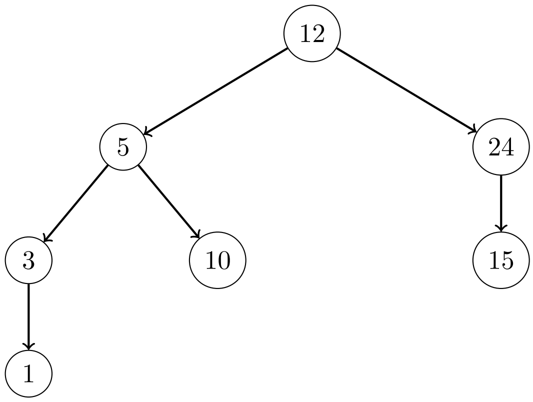
Melhor caso: $C = \Theta(1)$

Provando por definição o caso médio, temos que $\forall\, x > 0,\ \log(x) < x$, e portanto $\forall\, n > 1,\ \log(\log(n)) < \log(n)$. Para os casos pior e melhor que tiveram como resultado uma constante, tem se que, $f(n) \in \Theta g(n)$.

3 Questão

Problema: Implemente a estrutura de dados *Binary Search Tree* e os métodos buscar e inserir.

Para o desenvolvimento e análise do algoritmo de *Binary Search Tree* e os métodos buscar e inserir foi considerado o seguinte grafo:



Solução:

a) Implementação:

```
class Node:
    def __init__(self, value):
        self.left = None
        self.right = None
        self.value = value

    def insert_node(self, key):
        if key < self.value:
            if self.left is None:
                self.left = Node(key)
            else:
                self.left.insert_node(key)
        elif key > self.value:
            if self.right is None:
                self.right = Node(key)
            else:
                self.right.insert_node(key)

    def search_node(self, key):
        if key < self.value:
            if self.left is None:
                return False
            else:
                return self.left.search_node(key)
        elif key > self.value:
            if self.right is None:
                return False
            else:
                return self.right.search_node(key)
        else:
            return True
```

b) Operação básica: Comparação do valor com o valor nó.

```
key < self.value
```

ou

```
key > self.value
```

Como o *insert* e o *search* navegam de forma similar pela árvore binária de busca fazendo comparações, podemos definir as classes de eficiência para os dois métodos em conjunto.

c) Função de custo:

Pior caso (um elemento em cada nível):

$$C_w(n) = (n)$$

Caso médio (árvore balanceada):

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

Melhor caso (nó raiz):

$$C(1) = 1$$

d) Cálculo da função de custo para o caso médio:

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:

$$\begin{aligned}
C_{avg}(2^k) &= C_{avg}(2^{k-1}) + 1 \\
&= [C_{avg}(2^{k-2}) + 1] + 1 = C_{avg}(2^{k-2}) + 2 \\
&= [C_{avg}(2^{k-3}) + 2] + 1 = C_{avg}(2^{k-3}) + 3 \\
&... \\
&= C_{avg}(2^{k-i}) + i \\
&... \\
&= C_{avg}(2^{k-k}) + k = C_{avg}(2^0) + k \\
&= C_{avg}(1) + k = k + 1
\end{aligned}$$

$$\begin{aligned}
&C_{avg}(2^k) = k + 1 \\
&C_{avg}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil
\end{aligned}$$

e) Eficiência (O ou Θ):

Pior caso: $C_w = \Theta(n)$

Caso médio: $C_{avg} \approx \Theta(log_2 n)$

Melhor caso: $C = \Theta(1)$

Para provar a afirmação de eficiência para o caso médio por limites, temos que:

$$\lim_{n\to\infty} \frac{log_2(n) + 1}{log_2(n)} = \lim_{n\to\infty} \frac{log_2(n)}{log_2(n)} + \frac{1}{log_2(n)} = \lim_{n\to\infty} 1 + \frac{1}{\cancelto{0}{log_2(\infty)}} = 1$$

O resultado constante 1 implica que $log_2(n) + 1$ tem a mesma ordem de crescimento que $log_2(n)$ para o caso médio. Para os casos pior e melhor que também tiveram como resultado uma constante, tem se que, $f(n) \ \in \ \Theta g(n)$