Trabalho 1 - Análise de Algoritmos

Aluno: André de Oliveira Barbosa

April 23, 2024

- Segue o link para o repositorio no github com os códigos das questoes: clique aqui

1 Busca Sequencial

- Matematicamente se denotarmos $T(n)$ como o tempo de execução da função, com n sendo o número de elementos do array, podemos expressar isso como:

  $$T(n) = T(n-1) + c$$

  onde c é uma constante que representa o tempo necessário para executar o restante do código da função em cada chamada. No caso base, quando o elemento que buscamos está na primeira posição do array, a função retorna imediatamente, então $T(1) = 1 + c$.

  Expandindo a recorrência, temos:

  $$\begin{aligned} T(n) &= T(n-1) + c \\ &= T(n-2) + c \\ &= T(n-3) + c \\ &\vdots \\ &= T(n-k) + c \\ &= T(1) + kc \end{aligned}$$

  Onde k simboliza o número de chamadas recursivas.

  Note que:

  $$\begin{aligned} T(n-k) &= T(1) \\ n-k &= 1 \\ k &= n-1 \end{aligned}$$

Portanto:

$$T(n) = T(1) + kc = (1 + c) + (n - 1)c$$

Deprezando a constante c, temos:

$$T(n) = 1 + n - 1$$
$$T(n) = n \Rightarrow T(n) = O(n)$$

Logo, a complexidade temporal no pior caso da busca sequencial é $O(n)$.

- A complexidade no pior caso da versão recursiva é a mesma da versão iterativa, pois precisaremos percorrer o vetor por completo para, se existir, acharmos o item desejado.

2 Busca Binária

- A complexidade temporal no melhor caso da busca binária iterativa é $T(1) = 1$, que é a mesma da versão recursiva, pois em ambas as implementações o melhor caso é quando achamos o item desejado na primeira chamada.

  Agora calcularemos a complexidade temporal no pior caso da busca binária:

  Se denotarmos $T(n)$ como o tempo de execução da função, com n sendo o número de elementos do array, podemos expressar isso como:

$$T(n) = T(\frac{n}{2}) + c$$

  onde c é uma constante que representa o tempo necessário para executar o restante do código da função em cada chamada. No caso base, quando encontramos o elemento que buscamos na primeira chamada, a função retorna imediatamente, então $T(1) = 1 + c$.

  Expandindo a recorrência, temos:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + c \\ &= T(\frac{n}{4}) + c \\ &= T(\frac{n}{8}) + c \\ &\vdots \\ &= T(\frac{n}{k}) + c \end{aligned}$$

$$= T(1) + kc$$

Onde k simboliza o número de vezes em que o loop foi executado.

Note que podemos reescrever as funçoes como potências de 2:

$$
\begin{aligned}
T(n) &= T(\frac{n}{2^1}) + c \\
&= T(\frac{n}{2^2}) + c \\
&= T(\frac{n}{2^3}) + c \\
&\vdots \\
&= T(\frac{n}{2^k}) + c \\
&= T(1) + kc
\end{aligned}
$$

Observe que:

$$
\begin{aligned}
T(\frac{n}{2^k}) &= T(1) \\
\frac{n}{2^k} &= 1 \\
n &= 2^k \\
\log_2 n &= \log_2 2^k \\
\log_2 n &= k
\end{aligned}
$$

Portanto:

$$
\begin{aligned}
T(n) &= T(1) + kc \\
&= T(1) + (\log_2 n) \cdot c \\
&= 1 + (\log_2 n) \cdot c
\end{aligned}
$$

Deprezando a constante c, temos:

$$T(n) = 1 + \log_2 n \Rightarrow T(n) = O(\log_2 n)$$

Logo, a complexidade temporal no pior caso da busca binária é $O(\log_2 n)$.

- A complexidade espacial de uma busca binária **iterativa** é $O(1)$, ou seja, constante. Isso acontece porque ela não requer espaço adicional proporcional ao tamanho dos dados de entrada.

  Já a complexidade espacial da busca binária **recursiva** é $O(\log n)$. Isso ocorre devido ao uso da pilha de chamadas recursivas, que adiciona uma nova camada à medida que a recursão avança.

3 Vetor ordenado em ordem crescente

- O melhor caso ocorre quando o vetor está ordenado, pois o algoritmo ainda precisa verificar cada par de elementos adjacentes para garantir que cada elemento é menor ou igual ao próximo. Portanto, mesmo no melhor caso, o algoritmo executa $n-1$ comparações, onde n é o tamanho do vetor.

$$T(n) = n - 1 \Rightarrow O(n)$$

- O pior caso ocorre quando o os ultimos números do vetor nao estão em ordem, e o algoritmo precisa percorrer o vetor inteiro para verificar. Neste caso o algoritmo executa $n-1$ comparações, ou seja:

$$T(n) = n - 1 \Rightarrow O(n)$$

- O algoritmo usa um número constante de variáveis locais (uma variável de índice i, por exemplo), independentemente do tamanho do vetor. Não são utilizadas estruturas de dados adicionais que cresçam com o tamanho da entrada. Assim, a complexidade espacial é constante:

$$O(1) = 1$$

4 Fibonacci

- Calculando a complexidade temporal da função iterativa:

  Matematicamente se denotarmos $T(n)$ como o tempo de execução da função, com n sendo o número que queremos retornar da sequência de Fibonacci, podemos expressar isso como:

$$T(n) = T(n-1) + c$$

  onde c é uma constante que representa o tempo necessário para executar o restante do código da função em cada chamada. No caso base, quando o elemento que buscamos é o próprio número na sequência, a função retorna imediatamente, então $T(1) = 1 + c$.

  Expandindo a recorrência, temos:

$$\begin{aligned} T(n) &= T(n-1) + c \\ &= T(n-2) + c \\ &= T(n-3) + c \\ &\vdots \\ &= T(n-k) + c \end{aligned}$$

$$= T(1) + kc$$

Onde k a quantidade de vezes que o loop rodou.

Note que:

$$T(n-k) = T(1)$$
$$n - k = 1$$
$$k = n - 1$$

Portanto:

$$T(n) = T(1) + kc = (1 + c) + (n-1)c$$

Deprezando a constante c, temos:

$$T(n) = 1 + n - 1$$
$$T(n) = n \Rightarrow T(n) = O(n)$$

Logo, a complexidade temporal no pior caso para buscar um número da sequência de Fibonacci de forma iterativa é $O(n)$.

- Calculando a complexidade temporal da função recursiva:

  Matematicamente se denotarmos $T(n)$ como o tempo de execução da função, com n sendo o número que queremos retornar da sequência de Fibonacci, temos:

  No caso base, para $n = 0$ ou $n = 1$, o algoritmo retorna n diretamente sem fazer chamadas recursivas. Nesse caso, $T(0) = T(1) = 1$.

  Para $n > 1$, o algoritmo faz duas chamadas recursivas para calcular $fibonacci(n-1) e fibonacci(n-2)$, além da chamada atual da função. Portanto, o número total de chamadas recursivas é a soma do número de chamadas para $n - 1$ e $n - 2$.

$$T(n) = T(n-1) + T(n-2) + T(1) + c$$

  onde c é uma constante que representa o tempo necessário para executar o restante do código da função em cada chamada.

  Temos que:

$$T(n) = T(n-1) + T(n-2)$$

  A equação caracteristica para essa função é:

$$x^2 - x - 1 = 0$$

Resolvendo pela formula quadrática, temos:

$$x_1 = (\frac{1+\sqrt{5}}{2})$$
$$x_2 = (\frac{1-\sqrt{5}}{2})$$

Sabemos que a solução de uma função recursiva linear é dada como:

$$T(n) = T(\alpha_1)^n + T(\alpha_2)^n$$

onde α_1 e α_2 são as raizes das equação caracteristica. Logo, temos que:

$$T(n) = T(\alpha_1)^n + T(\alpha_2)^n$$
$$T(n) = T(\frac{1+\sqrt{5}}{2})^n) + T(\frac{1-\sqrt{5}}{2})^n$$
$$T(n) = O(\frac{1+\sqrt{5}}{2})^n) + O(\frac{1-\sqrt{5}}{2})^n$$

Usando as propriedades de notação Big O, onde eliminamos o termo de ordem inferior, temos:

$$T(n) = O(\frac{1+\sqrt{5}}{2})^n)$$
$$T(n) \approx O(1.618^n)$$
$$T(n) \approx O(2^n)$$

Logo, podemos ver que a complexidade desta função recursiva de fibonacci é $O(2^n)$, onde podemos ver que é mais custosa do que a versao iterativa, que possui complexidade $O(n)$.

- É possível implementar uma versão recursiva mais eficiente em termos de complexidade temporal, uma dessas formas é armazenar os números de fibonacci calculados a cada chamada.

  Nessa abordagem a complexidade é $O(n)$, pois utilizamos um loop para calcular os números de fibonacci.