Universidade Estadual de Maringá – Departamento de Informática

6889/2 – Projeto e análise de algoritmos – Ciência da computação

Prof. Marco Aurélio

Aluno(a): Matheus Augusto Schiavon Parise.  
RA: 107115.

# Avaliação 01

1. Considere o seguinte problema:

**Entrada**: Uma sequência de 𝑛 elementos 𝐴 = ⟨𝑎1,𝑎2,…,𝑎𝑛⟩ e um valor 𝑣.

**Saída**: Uma permutação ⟨𝑎′1,𝑎′2,…𝑎′𝑛⟩ de 𝐴 e um índice 𝑘 tal que os elementos 𝑎′1,𝑎′2,…,𝑎′𝑘 sejam menores que 𝑣 e os elementos de 𝑎′𝑘+1,…,𝑎′𝑛 sejam maiores ou iguais a 𝑣.

E o seguinte algoritmo que resolve o problema:

particao(𝐴,𝑣)

* 1. 𝑖 = 1
  2. 𝑗 = 𝐴.*length*
  3. **while** 𝑖 ≤ 𝑗
  4. **if** 𝐴[𝑖] < 𝑣
  5. 𝑖 = 𝑖 + 1
  6. **elseif** 𝐴[𝑗] ≥ 𝑣
  7. 𝑗 = 𝑗 − 1
  8. **else**
  9. troca(𝐴,𝑖,𝑗) **//** troca os valores 𝐴[𝑖] e 𝐴[𝑗]
  10. 𝑖 = 𝑖 + 1
  11. 𝑗 = 𝑗 − 1
  12. **if** 𝐴[𝑖] < 𝑣
  13. **return** 𝑖 + 1
  14. **else**
  15. **return** 𝑖
  16. (2,0) Faça a análise do tempo de execução do algoritmo. Como deve estar a entrada para que o laço execute o mínimo de vezes? Como deve estar a entrada para que o laço execute o número máximo de vezes?

Analisando o algoritmo percebemos que as linhas 1, 2, 12, 13, 14, 15 possuem tempo constante c. o laço de repetição definido pelas linhas 3 á 11 é executado até n vezes. Então temos como tempo:

T(n) = n + c = O(n).

a entrada para que o laço execute o mínimo de vezes é quando ele tem que fazer o maior número de permutações possíveis, ou seja, todos os elementos do índice 1 até k são maiores que v e todos os elementos de k+1 até n são menores que v pois então o problema é reduzido 2 vezes ou seja i aumenta 1 vez e j diminui uma.

a entrada para que o laço execute o número máximo de vezes é quando que todos os elementos de índice 1 até k sejam menores que v e todos de k+1 até n sejam maiores que v, dessa forma toda vez que executar um laço o problema é reduzido uma vez ou i aumenta em 1 ou j diminui em 1.

* 1. (2,0) Use a invariante a seguir e mostre que o algoritmo é correto. Invariante: os subarranjos 𝐴[1..𝑖 − 1] e 𝐴[𝑗+1..𝑛], onde 𝑛 = 𝐴.*length*, contêm os elementos inicialmente em 𝐴[1..𝑖−1] e 𝐴[𝑗+1..𝑛] mas rearranjados de forma que os elementos de 𝐴[1..𝑖 − 1] são menores que 𝑣 e os elementos de 𝐴[𝑗 + 1..𝑛] são maiores ou iguais a 𝑣.

De fato, está correto pois é possível verificar que a cada iteração que isso se mantem através da manutenção por exemplo antes dele começar o primeiro laço i = 1 e j = n, então 𝐴[1..𝑖 − 1] => 𝐴[1..1 − 1] => 𝐴[1 − 1] = 𝐴[0] ou seja nenhum elemento foi rearranjado de forma que eles são menores que v pois a sequência começa em 𝐴[1], o mesmo vale para 𝐴[𝑗+1..𝑛] => 𝐴[n+1..𝑛], n+1 está fora da sequencia ou seja nenhum elemento maior que v foi rearranjado, durante o laço ocorre a manutenção dessa invariante pois se 𝐴[𝑖] < 𝑣 então avança i uma posição pois ele está certo ou se ele 𝐴[𝑗] ≥ 𝑣 avança j pois ele é maior que v e no ultimo caso se os 2 estiverem errados, troca a posição deles e aumenta i e diminui j mantendo a invariante correta no começo do próximo laço.

1. Considere o seguinte problema:

**Entrada**: Uma sequência de 𝑛 elementos 𝐴 = ⟨𝑎1,𝑎2,…,𝑎𝑛⟩.

**Saída**: A modificação da 𝐴 tal que 𝐴 = ⟨𝑎2,…,𝑎𝑛,𝑎1⟩

* 1. (2,0) Projete um algoritmo *top-down* recursivo para resolver o problema e faça a análise do tempo de execução.

# Entrada: Uma sequência de 𝑛 elementos 𝐴 = ⟨𝑎1,𝑎2,…,𝑎𝑛⟩.  
# Saída: A modificação da 𝐴 tal que 𝐴 = ⟨𝑎2,…,𝑎𝑛,𝑎1⟩  
sequencia = [5,4,3,2,1]  
  
def ex\_2a(lst: list):  
 l = len(lst) - 1  
 v = lst[0]  
 def ex2a(lst2: list, l, n=0):  
 if l <= 1:  
 return lst2  
 elif l == n:  
 lst2[l] = v  
 if l > n:  
 lst2[n] = lst2[n+1]  
 ex2a(lst2, l, n+1)  
  
 ex2a(lst, l)  
  
ex\_2a(sequencia)

* 1. (2,0) Projete um algoritmo *bottom-up* iterativo para resolver o problema e enuncie a invariante (não é preciso mostrar que a invariante é válida).

# Entrada: Uma sequência de 𝑛 elementos 𝐴 = ⟨𝑎1,𝑎2,…,𝑎𝑛⟩.  
# Saída: A modificação da 𝐴 tal que 𝐴 = ⟨𝑎2,…,𝑎𝑛,𝑎1⟩  
sequencia = [1,2,3,4,5]  
  
def ex\_2b(lst: list):  
 i = 0  
 v = 0  
 l = len(lst)-1  
 while i <= l:  
 if i == 0:  
 v = lst[i]  
 elif i == l:  
 lst[l] = v  
 lst[i] = lst[i + 1]  
 i += 1  
  
ex\_2b(sequencia)

1. (2,0) Escolha apenas um dos problemas a seguir e projete um algoritmo com tempo de execução 𝑂(lg𝑛) que resolva o problema. Argumente que o seu algoritmo é correto e faça a análise do tempo de execução.
   1. **Entrada**: Uma sequência de 𝑛 = 2𝑘+1 elementos ordenados, onde 𝑘 é um número natural, tal que na sequência aparecem 𝑘 + 1 elementos distintos, sendo que 𝑘 elementos aparecem duas vezes.

**Saída**: O valor do elemento que aparece apenas uma vez. (Por exemplo, para a sequência de entrada ⟨4,4,7,7,8,9,9⟩ a resposta é 8.)

A questão escolhida foi 3) a)

# Entrada: Uma sequência de 𝑛 = 2𝑘 +1 elementos ordenados, onde 𝑘 é um número natural, tal que na sequência  
# aparecem 𝑘 + 1 elementos distintos, sendo que 𝑘 elementos aparecem duas vezes.  
# Saída: O valor do elemento que aparece apenas uma vez. (Por exemplo, para a sequência de entrada  
# ⟨4, 4, 7, 7, 8, 9, 9⟩ a resposta é 8.)  
# Considerando o índice inicial 0  
sequencia = [4, 4, 7, 7, 8, 9, 9]  
sequencia2 = [3, 4, 4, 7, 7, 8, 8, 9, 9]  
  
def Achar\_unico(lst: list, a=0) -> int:  
 if len(lst) == 1:  
 return lst[0]  
 else:  
 if lst[a] == lst[a+1]:  
 return Achar\_unico(lst, a+2)  
 else:  
 return lst[a]

* 1. **Entrada**: Uma sequência de 𝑛 > 0 números 𝐴 = ⟨𝑎1,𝑎2,…,𝑎𝑘,𝑎𝑘+1 …,𝑎𝑛−1,𝑎𝑛⟩, tal que os elementos 𝑎1,𝑎2,…,𝑎𝑘 não são positivos e os elementos 𝑎𝑘+1,…,𝑎𝑛−1,𝑎𝑛 são positivos.

**Saída**: O índice 𝑘. Note que 𝑘 pode ser qualquer valor entre 0 e 𝑛. (Por exemplo, para sequência de entrada ⟨−3,−1,0,−2,6,2,1, a resposta é 4.)