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 $A = \langle a_1, a_2, \ldots, a_n \rangle$ e um valor v.

Saída: Uma permutação $\langle a'_1, a'_2, \ldots a'_n \rangle$ de A e um índice k tal que os elementos $a'_1, a'_2, \ldots, a'_k$ sejam menores que v e os elementos de $a'_{k+1}, \ldots, a'_n$ 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
```

a) (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.

b) (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 $A[j] \geq v$ 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.

2) Considere o seguinte problema:

Entrada: Uma sequência de n elementos $A = \langle a_1, a_2, \ldots, a_n \rangle$.

Saída: A modificação da A tal que $A = \langle a_2, \ldots, a_n, a_1 \rangle$

a) (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)
```

b) (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)
```

3) (2,0) Escolha apenas um dos problemas a seguir e projete um algoritmo com tempo de execução $O(\lg n)$ que resolva o problema. Argumente que o seu algoritmo é correto e faça a análise do tempo de execução.

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 $\langle 4,4,7,7,8,9,9 \rangle$ 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]
```

b) **Entrada**: Uma sequência de $n > 0$ números $A = \langle a_1,a_2,\ldots,a_k,a_{k+1} \ldots,a_{n-1},a_n \rangle$, tal que os elementos $a_1,a_2,\ldots,a_k$ não são positivos e os elementos $a_{k+1},\ldots,a_{n-1},a_n$ são positivos.

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