Prova 1 — PMR3201 Computação para Automação 1o. semestre 2019

Prof. Thiago de Castro Martins

Instruções: Coloque seu nome e NUSP na primeira folha de papel almaço. Numere as páginas de sua prova e coloque o número total de páginas na primeira. Quando escrever código em Python use sempre linhas verticais para explicitar as delimitações de escopo. As questões podem ser resolvidas em qualquer ordem. Use apenas instruções de Python com complexidade constante.

1. (2,5 pontos) Sejam duas sequências de inteiros $A = \{a_0, \ldots, a_{n-1}\}$ e $B = \{b_0, \ldots, b_{m-1}\}$, cada uma com ao menos um elemento. Escreva em python uma função que retorna 1 se todos os elementos em A são maiores que todos os elementos em B, -1 se todos os elementos em B são maiores do que todos os elementos em A e 0 se nenhuma dessas condições se aplica. Use a seguinte assinatura:

```
def compara_sequencias(a, b):
```

onde `a` é a sequência A e `b` é a sequência B. Sua função deve ter complexidade $\mathcal{O}(N + M)$ onde N e M são os tamanhos das sequências A e B respectivamente. Piores complexidades valem 1,5 pontos.

Resposta: Todos os elementos de A são maiores do que todos os elementos de B se e somente se o menor elemento de A é maior do que o maior elemento de B. Do mesmo modo, todos os elementos de B são maiores do que todos os elementos de A se e somente se o menor elemento de B é maior do que o maior elemento de A. Estes elementos podem ser obtidos com complexidade linear. A solução se segue:

```
def compara_sequencias(a, b):
    maior_a = menor_a = a[0]
    for i in range(1, len(a)):
        if a[i]<menor_a:
            menor_a = a[i]
        elif a[i] > maior_a:
            maior_a = a[i]
    maior_b = menor_b = b[0]
    for i in range(1, len(b)):
        if b[i]<menor_b:
            menor_b = b[i]
        elif b[i] > maior_b:
            maior_b = b[i]
    if maior_b < menor_a:
        return 1
    if maior_a < menor_b:
        return -1
    return 0
```

2. (2,5 pontos) Seja uma sequência de números de ponto flutuante $A = \{a_0, \ldots, a_{n-1}\}$ ordenada em ordem estritamente crescente, ou seja, $a_{i+1} > a_i$. Considere o problema de, dado um número x, encontrar o valor de a_i que minimiza o valor $\|x - a_i\|$, ou seja, o elemento *mais próximo* de x. O código abaixo resolve este problema:

```
1  def mais_proximo(a, x):
2      e = 0
3      d = len(a)-1
4      while (d-e)>1:
5          m = (e+d)//2
```

```
6           if a[m] > x:
7               d = m
8           else:
9               e = m
10      return a[e] if x - a[e] < a[d] - x else a[d]
```

Onde `a` é a sequência de números de ponto flutuante A e `x` contém o número x que deseja-se aproximar.

(a) (1.5 pontos) Mostre que o código está correto, ou seja, efetivamente retorna o valor de A mais próximo de x.

Resposta: Sejam e_i, m_i e d_i o valor das variáveis `e`, `m` e `e` ao final da i-gésima iteração do laço da linha 4, com $e_0 = 0$ e $d_0 = N - 1$. Vale a seguinte lei de recorrência para o laço:

$$m_{i+1} = \left\lfloor \frac{e_i + d_i}{2} \right\rfloor$$

$$\{e_{i+1}, d_{i+1}\} = \begin{cases} \{m_{i+1}, d_i\} \text{ para } a_{m_{i+1}} > x \\ \{e_i, m_{i+1}\} \text{ para } a_{m_{i+1}} \leq x \end{cases}$$

A condição de permanência no laço é $d_i - e_i > 1$. Ora, mas para $d_i - e_i > 1$ vale $e_i < m_{i+1} < d_i$. Como m_{i+1} substitui ou e_i ou d_i na iteração seguinte, vale $d_{i+1} - e_{i+1} < d_i - e_i$. Deste modo, o algoritmo termina necessariamente em tempo finito.

Seja p o índice do elemento de A que é mais próximo de x. Vale o seguinte invariante:

$$e_i \leq p \leq d_i$$

De fato, ele é trivialmente verdadeiro para $i = 0$, pois o intervalo $[e_0, d_0]$ inclui todos os índices possíveis de A. Porém se existe algum índice q tal que $a_q > x$ então necessariamente $p \leq q$, pois numa sequência ordenada, todos os elementos de A posteriores a a_q são mais distantes de x. Do mesmo modo, se $a_q \leq x$ então necessariamente $p \geq q$. Aplicando-se estas propriedades às leis de recorrência (com m_{i+1} fazendo o papel do índice q), verifica-se que o invariante se mantém.

Ora, na condição de saída do laço tem-se um intervalo de no máximo dois índices ao qual pertence o índice desejado. Basta verificar qual dos dois índices apresenta um elemento mais próximo de x.

(b) (1.0 pontos) Obtenha a complexidade em notação *Big Oh* em função do comprimento da sequência N.

Resposta: O algoritmo divide a cada iteração do laço o intervalo de índices $[e, d]$ pela metade. Ele pode fazer isso no máximo $\log_2 N$ vezes. Assim a complexidade é $\mathcal{O}(\log N)$.

3. (2,5 pontos) Considere o código abaixo:

```
1  def particao(a, e, d):
2      p = a[d-1]
3      i = e
4      j = d-2
5      continua = True
6      while continua:
7          while i <= j and a[i] <= p: i += 1
8          while i <= j and a[j] >= p: j -= 1
```

```
9           if i<=j:
10              a[i], a[j] = a[j], a[i]
11          else:
12              continua = False
13      a[d-1] = a[i]
14      a[i] = p
15      return i if i>e else i+1
```

A função `particao` recebe uma sequência de m inteiros $A = \{a_0, \ldots, a_{m-1}\}$ em `a` e dois índices e e d em `e` e `d` respectivamente, com $0 \leq e < d \leq m$. A função rearanja os elementos de A entre os índices e e $d-1$ (ou seja, d indica a primeira posição não-modificada) de modo que exista algum índice i tal que $e < i < d$ e que todos os elementos de A entre e e $i-1$ seja menores ou iguais a todos os elementos de A de i a $d-1$. A função retorna este índice i. Esta operação é realizada com complexidade $\mathcal{O}(d-e)$. Seja o problema de se encontrar o elemento de ordem n em uma sequência de inteiros não ordenada.

Define-se elemento de ordem n o *menor* elemento de A para o qual haja n elementos menores ou iguais a ele. Por exemplo, na sequência $\{1, 3, 5, 6, 3\}$ o elemento de ordem 0 é 1, o elemento de ordem 1 é 3, o elemento de ordem 2 é 3, o elemento de ordem 4 é 5 e o elemento de ordem 5 é 6 (verifique!). O código abaixo resolve este problema:

```
1  def busca_enesimo(a, n):
2      a = list(a) # Copia a, complexidade O(N)
3      def busca_recursiva(e, d):
4          if d-e <=1:
5              return a[e]
6          k = particao(a, e, d)
7          if k > n:
8              return busca_recursiva(e, k)
9          else:
10             return busca_recursiva(k, d)
11     return busca_recursiva(0, len(a))
```

(a) (1,5 pontos) Mostre que o código está correto. Você pode usar todas as propriedades da função `particao` do enunciado.

Resposta: A função chama a função `busca_recursiva`. Sejam e, d e k os valores das variáveis `e`, `d` e `k` respectivamente na função `busca_recursiva`. O caso base da função é $e - d \leq 1$. Ora mas a função chama a si mesma ou com os limites e, k ou com k, d. Sabe-se das propriedades de `particao` que $e < k < d$, ou seja, o caso base é `sempre` atingido e o algoritmo termina em tempo finito.

Mostra-se que todo elemento de $A0 : e-1$ é menor ou igual a todo elemento de $A_{e:d-1}$. Estes, por sua vez, são menores ou iguais a todos elementos de $A_{d:m-1}$. De fato, isso é trivialmente verdadeiro na chamada inicial a `busca_recursiva`. Por outro lado, pelas propriedades de `particao`, todos os elementos de $A_{e:k-1}$ são menores ou iguais aos elementos de $A_{k,d-1}$. Assim, por indução finita, a propriedade se mantém, seja e ou d substituído por k no próximo nível de chamada recursivo.

Mostra-se também que o índice p do elemento de ordem n está sempre entre e e d. Isso é trivialmente verdadeiro no primeiro nível, quando $e = 0$ e $d = m$. Ora, mas se $k > n$, então tem-se necessariamente $p < k$ pois todos os elementos em $A_{k:d-1}$ são maiores ou iguais aos elementos em $A_{e:k}$. Por outro lado, existem mais do que n elementos em $A_{0:k}$, então existe necessariamente algum valor neste intervalo maior ou igual a ao menos n elementos deste mesmo. Por outro lado, se $k \leq n$, então tem-se necessariamente $p \geq n$, pois existem em $A_{0,k-1}$

ao menos n elementos, e *todos* estes são menores ou iguais aos elementos de $A_{k,d-1}$. Deste modo, a propriedade também é mantida.

Se estas duas pré-condições se aplicam a *todas* as chamadas de `busca_recursiva`, então elas se aplicam também ao caso base. Mas no caso base, o único valor possível para o índice p é e. Assim o algoritmo retorna a resposta correta.

(b) (1,0 pontos) Qual é a complexidade assintótica da função `busca_enesimo` em função do comprimento da sequência M?

Resposta: A análise é similar a do Quicksort. A operação de partição na pior das hipóteses divide o vetor com comprimento N em uma partição com comprimento 1 e outra com comprimento $N-1$. Novamente, no pior dos casos, o algoritmo persiste na maior das partições. Assim a complexidade total é dada por:

$$\sum_{i=N}^{1} \mathcal{O}(i) = \mathcal{O}(N^2)$$

4. (2,5 pontos) Seja $A = \{a_1, \ldots, a_n\}$ uma sequência de inteiros. Uma *subsequência crescente* de A é uma sequência $\{a_{i_1}, \ldots, a_{i_m}\}$ tal que $1 \leq i_j < i_{j+1} \leq n$ e $a_{i_j} < a_{i_{j+1}}$, ou seja, uma subsequência com os elementos de A, não necessariamente contígua e estritamente crescente. Por exemplo, considere a sequência $\{4, 1, 2, 5, 2, 3, 6\}$. São suas subsequências crescentes $\{4, 5, 6\}$, $\{1, 2, 3\}$, $\{1, 2, 5, 6\}$ entre outras. Escreva uma função em Python que, dada uma encontrar o comprimento da mais longa subsequência crescente. Use a seguinte assinatura:

```
def max_subsec_cresc(a):
```

onde `a` é um vetor de inteiros com a sequência A. Sua função deve retornar o tamanho da maior subsequência crescente de A. Sua função deve ter complexidade $\mathcal{O}\left(N^2\right)$ onde N é o comprimento da sequência A. Complexidades piores valem 1 ponto.

Sugestão: Suponha que para um índice k, são conhecidos os tamanhos das máximas subsequências de A que acabam em cada elemento a_i, com $i \leq k$. Você é capaz de encontrar o tamanho da máxima subsequência que acaba em a_{k+1} com complexidade $\mathcal{O}(k)$?

Resposta: A maior subsequência crescente terminada na posição $k+1$ tem comprimento 1 + a maior das subsequências que acabam em um elemento a_i com $i \leq k$ tal que $a_i < a_{k+1}$. Como há no máximo k tais subsequências, este teste pode ser feito com complexidade $\mathcal{O}(k)$. A complexidade total do algoritmo é $\mathcal{O}(N^2)$.

```
def max_subsec_cresc(a):
    max_i = [1]*len(a)
    max_sub = 1
    for i in range(1,len(a)):
        for j in range(i):
            if a[j]<a[i] and max_i[i] < max_i[j]+1:
                max_i[i] = max_i[j]+1
        if max_i[i] > max_sub:
            max_sub = max_i[i]
    return max_sub
```

Formulário

Somas de sequências:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}, \quad \sum_{i=0}^{n-1} i^2 = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}, \quad \sum_{i=0}^{n-1} i^3 = \frac{n^4}{4} - \frac{n^3}{2} + \frac{n^2}{4},$$

$$\sum_{i=0(a\neq 1)}^{n-1} a^i = \frac{1-a^n}{1-a}, \quad \sum_{i=0(a\neq 1)}^{n-1} i a^i = \frac{a - na^n + (n-1)a^{n+1}}{(1-a)^2}.$$

Solução de equações de recorrência pelo *Master Theorem*:

A equação:

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + cN^L,$$

com $T(1) = \mathcal{O}(1)$, tem solução

$$T(N) = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L \end{cases}$$

Equivalência de recursão por *tail call* a laço:

function $F(\boldsymbol{X})$
  if $C(\boldsymbol{X})$ **then**
    return $E(\boldsymbol{X})$
  else
    return $F(G(\boldsymbol{X}))$
  end if
end function

Versão recursiva

function $F(\boldsymbol{X})$
  while NOT $C(\boldsymbol{X})$ **do**
    $\boldsymbol{X} \leftarrow G(\boldsymbol{X})$
  end while
  return $E(\boldsymbol{X})$
end function

Versão iterativa com laço