Prova 1 — PMR3201 Computação para Automação 1o. semestre 2018

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.

1. (2,5 pontos) Escreva uma função em python que retorna o *segundo* maior elemento de uma sequência de inteiros com ao menos 2 elementos. Use a seguinte assinatura:

   ```
   def segundomaior(a):
   ```

   onde `a` é uma sequência de inteiros com ao menos 2 elementos.

   Resposta:

   Define-se *segundo maior* elemento de uma sequência $A = \{a_1, a_2, \dots a_n\}$, $a_n \geq 2$ como o elemento a_s em $\{a_p, a_s\}$ tal que $s \neq p$, $a_p \geq a_s$ e $a_s \geq a_i \ \forall i \neq p$. Uma solução é adaptar o algoritmo que busca pelo maior elemento para que este trate do *par* de maiores elementos, mantendo-o ordenado:

   ```
   def segundomaior(a):
       if a[0]>a[1]:
           p, s = a[0], a[1]
       else:
           p, s = a[1], a[0]
       for i in range(2, len(a)):
           if a[i]> s:
               if a[i]>p:
                   s = p
                   p = a[i]
               else:
                   s = a[i]
       return s
   ```

2. (2,5 pontos) Em uma sequência de inteiros $A = \{a_1, \dots a_n\}$ uma *inversão* é uma ocorrência de pares de índices (i, j) com $i < j$ e $a_i > a_j$, ou seja, um par de elementos tal que o maior antecede o menor. Por exemplo, a sequência $(1, 2, 3)$ não contém nenhuma inversão (como toda sequência ordenada), enquanto que a sequência $(3, 2, 1)$ contém 3 inversões (os pares $(3, 1)$, $(3, 2)$ e $(2, 1)$).

   (a) (1,5 pontos) Escreva uma função em python implementando um algoritmo *não-recursivo* que retorna o número de inversões em uma sequência. O seu algoritmo deve usar memória *constante* com o tamanho da entrada. Use a seguinte assinatura:

   ```
   def inversoes(a):
   ```

   onde `a` é a sequência cujas inversões devem ser contadas. *Sugestão*: Cuidado para não contar a mesma inversão mais de uma vez!

   Resposta: Pela definição, o total de inversões é igual ao total, para cada elemento a_i, do número de elementos à sua direita que são menores do que ele, ou seja:

   $$\texttt{inversoes}\,(a_1, \dots, a_n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \begin{cases} 0 \,\text{se}\, a_j \geq a_i \\ 1 \,\text{se}\, a_j < \ a_i \end{cases}$$

Esta soma se traduz trivialmente no algoritmo abaixo:

```
def inversoes(a):
    inversoes = 0
    for i in range(len(a)):
        for j in range(i+1, len(a)):
            if a[j] < a[i]:
                inversoes += 1
    return inversoes
```

(b) (1,0 pontos) Qual a complexidade do seu algoritmo?

Resposta: Seja n o tamanho da seqência. Então a complexidade é dada pelo total de iterações do laço interno, ou seja,

$$\sum i = 1^n \sum j = 1 + 1^n \mathcal{O}(1) = \mathcal{O}\left(n^2\right)$$

3. (2,5 pontos) : Considere o problema de se calcular o valor de um polinômio:

$$p\left(x\right) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

Este polinômio pode ser representado computacionalmente por uma sequência de seus $n+1$ coeficientes $A = \{a_0, \ldots a_n\}$. Considere o algoritmo abaixo:

```
1  def polinomio(a, x):
2      i = len(a)-1
3      p = 0
4      while i>= 0:
5          p *= x
6          p += a[i]
7          i -= 1
8      return p
```

Este algoritmo recebe uma sequência com os coeficientes de $p(x)$ e um valor x e retorna o valor de $p(x)$.

(a) 2,0 Mostre que o algoritmo está correto.

Resposta: Seja $n+1$ o tamanho da sequência. Seja x o valor (constante) do parâmetro `x`. Seja j o número de vezes que as linhas 4-7 foram executadas. Seja i_j e p_j o valor das variáveis `i` e `p` ao *final* da execução pela j-ésima vez da linha 7, com $p_0 = 0$ e $i_0 = n$. As leis de recorrência, derivadas das transformações nas linhas 5, 6 e 7 são:

$$i_{j+1} = i_j - 1 \tag{1}$$
$$p_{j+1} = x p_j + a_{i_j} \tag{2}$$

Segue-se trivialmente de (1) que

$$i_j = n - j - 1 \tag{3}$$

A condição de permanência no laço na linha 4 é $i_j \geq 0$, de modo que a última iteração é a na qual $j = n + 1$. Deste modo, o programa termina necessariamente em tempo finito. Vale

também:

$$p_j = \sum_{k=n}^{n-(j-1)} a_k x^{k-(n-(j-1))} \tag{4}$$

De fato, isso é trivialmente verdadeiro para p_0. Porém, se existe p_j para o qual vale (4), então por (2),

$$\begin{aligned} p_{j+1} &= xp_j + a_{i_j} \\ &= x \sum_{k=n}^{n-(j-1)} a_k x^{k-(n-(j-1))} + a_{i_j} \\ &= \sum_{k=n}^{n-(j-1)} a_k x^{k-(n-j)} + a_{i_j} x^0 \\ &= \sum_{k=n}^{n-j} a_k x^{k-(n-j)} \end{aligned} \tag{5}$$

Ou seja, se (4) vale para algum p_j, então também vale para p_{j+1}, o que por indução finita mostra a validade de (4) para todo j finito. As propriedades (3, 4) são os *invariantes* deste algoritmo. O valor retornado pelo algoritmo é p_{n+1}, que vale:

$$p_{n+1} = \sum_{k=n}^{0} a_k x^k$$

Que é o valor desejado. Isso demonstra que o algoritmo está correto.

(b) 0,5 Calcule a complexidade do algoritmo.

Resposta: Como visto no item anterior, a complexidade do algoritmo é $\mathcal{O}(N)$, onde N é a quantidade de coeficientes do polinômio.

4. (2,5 pontos) Há como resolver a questão 2 com complexidade $\mathcal{O}(N \log N)$ com uma extensão do algoritmo de ordenação mergesort. Considere a listagem:

```
1  def mergesort(v):
2      temp = [None]*len(v) # Complexidade O(n)
3      def merge(e, m, d):
4          i = e
5          j = m
6          k = e
7          inv = 0
8          while i < m and j < d:
9              if v[i] <= v[j]:
10                 temp[k] = v[i]
11                 i += 1
12             else: # O elemento à direita é menor do que i, conte inversões
13                 inv += m - i
14                 temp[k] = v[j]
15                 j += 1
16             k += 1
```

```
17          # Completa com os elementos restantes
18          while i<m:
19              temp[k] = v[i]
20              i += 1
21              k += 1
22          while j<d:
23              temp[k] = v[j]
24              j += 1
25              k += 1
26          # Copia o resultado do vetor temporário
27          # de volta em v
28          for k in range(e, d):
29              v[k] = temp[k]
30          return inv
31
32      def merge_recursiva(e, d):
33          if d - e <= 1: return 0
34          inve = merge_recursiva(e, (e+d)//2)
35          invd = merge_recursiva((e+d)//2, d)
36          invm = merge(e, (e+d)//2, d)
37          return inve+invd+invm
38
39      return merge_recursiva(0, len(v))
```

Esta listagem é muito similar à implementação de mergesort estudada no curso. De fato, assim como no mergesort original, este algoritmo ordena o vetor `v` com complexidade $\mathcal{O}(N \log N)$. Esta implementação, no entanto, também retorna a quantidade total de inversões no vetor original. Para tanto, ela soma os valores retornados pelas duas chamadas recursivas e o valor retornado pela função `merge`. Mostre que esta implementação está correta. *Sugestões*:

- Naturalmente, você pode usar qualquer propriedade do mergesort aplicável (por exemplo, o algoritmo termina, os sub-vetores após cada chamada recursiva estão ordenados, etc.).
- Observe que o total de inversões é o total, para cada elemento, de outros elementos menores do que si que o sucedem. O que é igual ao total, para cada elemento, de outros elementos maiores do que si que o precedem.

Resposta: Como trata-se de uma adição de um contador ao *mergesort*, este algoritmo necessariamente termina em tempo finito. O caso base do algoritmo é o do vetor unitário ou nulo, para o qual este retorna o valor correto de zero inversões. Resta mostrar que se as duas chamadas recursivas nas linhas 34 e 35 efetivamente retornam o número de inversões nos seus respectivos intervalos, a função também retorna o valor correto. Como visto na questão 2, o número de inversões I é dado por

$$I = \sum_{j=0}^{n-1} \xi_j \tag{6}$$

onde ξ_j é o total de elementos a_i com $i < ji$ tais que $a_i > a_j$. Escrevendo-se a função $C(n) = 1\, n > 0$ e $C(n) = 0\, n \leq 0$, então pode-se escrever

$$\xi_j = \sum_{i=0}^{j-1} C(a_i - a_j) \tag{7}$$

Ora, mas para qualquer índice m tal que $0 \leq m \leq n-1$, vale

$$\xi_j = \overbrace{\sum_{i=0}^{m-1} C(a_i - a_j)}^{\epsilon_{j,m}} + \overbrace{\sum_{i=m}^{j-1} C(a_i - a_j)}^{\delta_{j,m}} \tag{8}$$

Ou seja, se particiona-se a sequência em $e = \{a_0, \ldots, a_{m-1}\}$ e $d = \{a_m, \ldots, a_{n-1}\}$, então trivialmente, o valor ξ_j é a soma da quantidade de elementos maiores encontrados na primeira partição $\epsilon_{j,m}$ e a quantidade de elementos na segunda partição $\delta_{j,m}$. Observa-se ademais que quando $j < m$ então $\delta_{j,m} = 0$.

Substituindo-se em (6), temos:

$$I = \sum_{j=0}^{n-1} \epsilon_{j,m} + \sum_{j=0}^{n-1} \delta_{j,m} = \overbrace{\sum_{j=0}^{m-1} \epsilon_{j,m}}^{I_e} + \overbrace{\sum_{j=m}^{n-1} \delta_{j,m}}^{I_d} + \overbrace{\sum_{j=m}^{n-1} \epsilon_{i,m}}^{I_m} \tag{9}$$

Por hipótese, I_e seria o valor retornado na linha 34, ou seja, a quantidade de inversões na primeira sub-sequência. Do mesmo modo, I_d seria o valor retornado na linha 35, ou seja a quantidade de inversões na segunda sub-sequência. Assim, se provarmos que a chamada a `merge` efetivamente retorna I_m, comprovaremos a correção do algoritmo.

Em primeiro lugar, lembremos que os sub-vetores estão *ordenados*, ou seja, $a_i \leq a_{i+1}$ para $0 \leq i < m-1$ e , $a_j \leq a_{j+1}$ para $m \leq j < n-1$. Seja `i` $= i$, `j` $= j$ no final da execução da linha 25. Vale

$$a_k \leq a_j \, \forall \, 0 \leq k < i \tag{10}$$

Isso é trivialmente verdadeiro quando $i = 0$. Qualquer incremento de j não afeta esta propriedade, visto que o sub-vetor à direita está ordenado. Os incrementos de i ocorrem na linha 11, quando $a_{i-1} \leq a_j$. Como o sub-vetor à esquerda está ordenado, estes também não afetam a propriedade.

Seja I o valor da variável `inv` ao final da execução da linha 25. Então vale:

$$I = \sum_{k=m}^{j-1} \epsilon_{j,m} \tag{11}$$

De fato, isso é trivialmente verdadeiro quando $j = m$ e $I = 0$. Quando j é incrementado na linha 24, tem-se $a_k > a_{j-1} \forall \, i \leq k < m-1$, pois o vetor é ordenado. Por outro lado, por (10), estes são os *únicos* elementos no sub-vetor à esquerda maiores do que a_{j-1}. Segue-se que $\epsilon j - 1, m = m - 1$ e a atualização de I na linha 13 não altera a propriedade (11).

O laço na linha 8 sai por ou $i = m$ ou $j = n-1$. No segundo caso, (11) nos leva a $I = I_m$. Por outro lado, se $i = m$ então (10) implica que $\epsilon_{k,m} = 0 \, \forall \, j \leq k < n$, ou seja, do mesmo modo, $I = I_m$.

Assim, o valor retornado pela função `merge` é efetivamente I_m e o algoritmo está correto.

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