Lista 1 - *adendo* - PMR3201

Fabio G. Cozman, Thiago Martins

2017

Exercícios

1. (P1 2016) A listagem a seguir mostra o código de uma função que converte uma cadeia de caracteres com a representação decimal de um número em um inteiro com o número em questão:

```
1  def atoi(s):
2      r = 0
3      for i in range(len(s)):
4          r *= 10
5          r += ord(s[i]) - ord('0')
6      return r
```

   A função presume que todos os caracteres na cadeia são dígitos entre "0" e "9".

   São relevantes as seguintes operações sobre uma cadeia de caracteres:

   A função `length(s)` retorna um inteiro com a quantidade de caracteres na cadeia `s`.

   O operador `s[i]` retorna o caractere na i-gésima posição da cadeia `s` (contando a partir de 0).

   A função `ord(c))` retorna o código ascii do caractere `c`. No código ASCII os dígitos de 0 a 9 correspondem a códigos consecutivos.

   a) Mostre que o código acima termina em tempo finito.

   b) Mostre que o código está correto, ou seja, ele efetivamente retorna o número desejado. Sugestão: Seja $S = s_0, s_1, \ldots, s_n$ a sequência de caracteres na cadeia `s`. Observe que o número desejado é

$$\sum_{j=0}^{n} 10^{n-j} s_j$$

   Seja r_i o valor da variável `r` ao final da i-gésima iteração. Escreva a lei de recorrência de r_i e a partir desta encontre a expressão para r_i em função de S e i.

   c) Escreva em notação *big Oh* a ordem de complexidade do algoritmo em função do número de caracteres na cadeia N.

2. (P1 2016) É possível estabelecer uma relação de ordenação entre duas cadeias de caracteres, análoga à ordem alfabética de verbetes em um dicionário. Sejam as cadeias $A = \{a_0, a_1, \ldots, a_n\}$ e $B = \{b_0, b_1, \ldots, b_m\}$ e i o *menor* índice tal que $a_i \neq b_i$ (se houver). Se $a_i < b_i$, então $A < B$ e se $a_i > b_i$, então $A > B$. Se não há tal índice (ou seja, as cadeias são idênticas até a última posição da menor), então se $n < m$ então $A < B$ e se $n > m$ então $A > B$, ou seja, a menor cadeia vem *antes* na ordem estabelecida. Exemplos:

   - se $A =$ "`ab`" e $B =$ "`aa`" então $A > B$.
   - se $A =$ "`aa`" e $B =$ "`aaa`" então $A < B$.
   - se $A =$ "`aaa`" e $B =$ "`aaa`" então $A = B$.

   a) Escreva uma função em Python que recebe duas cadeias de caracteres, `s1` e `s2`, e retorna -1 se `s1` $<$ `s2`, 0 se `s1` $=$ `s2` e 1 se `s1` $>$ `s2`. Use a seguinte assinatura:

```
def CompareStrings(s1, s2):
```

Dos recursos de Python para cadeias de caracteres você deve usar *exclusivamente* as funções `len()`, `ord()` e o operador `[]` (vide questão anterior).

b) Seja N a quantidade de caracteres em `s1` e M a quantidade de caracteres em `s2`. Escreva em função de N e M a quantidade de iterações da função criada.

3. (P1 2016) Define-se como repetição em um vetor a ocorrência de um elemento idêntico a um elemento anterior. Por exemplo, o vetor $\{1, 1, 1\}$ tem duas repetições, o vetor $\{1, 1, 2\}$ tem uma repetição e o vetor $\{1, 2, 3\}$ não tem nenhuma repetição.

a) Escreva uma função em Python que, dado um vetor *ordenado* em ordem crescente, retorna a quantidade de repetições com complexidade *linear* ($\mathcal{O}(N)$). Utilize a seguinte assinatura:

```
Def Repeticoes(a):
```

Onde `a` é o vetor em questão.

b) Explique com o seu conhecimento de algoritmos como é possível obter o número de repetições em um vetor não-ordenado com complexidade $\mathcal{O}(N \log N)$.

4. (P1 2016) Considere o problema de se encontrar a maior subsequência (com ao menos um elemento) de um vetor. O algoritmo abaixo resolve este problema de forma recursiva.

```
1  def maxSubSeqRec(a, e, d):
2      if e==d: return a[e]
3      meio = (e+d)//2
4      s1 = maxSubSeqRec(a, e, meio)
5      s2 = maxSubSeqRec(a, meio+1, d)
6      # Encontra maior subsequência que começa
7      #  à esquerda do centro e acaba à direita
8      # Maior subsequencia do centro a esquerda
9      soma = a[meio]
10     s3e = soma
11     for ee in range(meio-1, e-1, -1):
12         soma +=a [ee]
13         if soma>s3e: s3e = soma
14
15     # Maior subsequencia do centro a direita
16     soma = a[meio+1]
17     s3d = soma
18     for dd in range(meio+2, d+1):
19         soma+=a[dd]
20         if soma>s3d: s3d = soma
21
22     s3 = s3e + s3d
23     # Retorna o maior de s1, s2 e s3
24     if s3 > s1:
25         if s3 > s2: return s3
26         else: return s2
27     elif s2>s1: return s2
28     return s1
```

A função `maxSubSeq(a)` chama `maxSubSeq(a, e, d)` com limites 0 e `len(a)`. Esta função, por sua vez, se o vetor tiver comprimento unitário retorna o único elemento do vetor. Caso contrário, divide o vetor em duas partes de tamanho aproximadamente igual (diferem entre si por no máximo 1), chama a si mesmo em cada uma e armazena o resultado em `s1` e `s2`. Depois, determina qual a maior sequência que começa *antes* da metade do vetor e termina *depois* por busca sequencial simples (linhas de código de 13 a 26) e armazena o resultado em `s3`. Finalmente, retorna o maior valor entre `s1`, `s2` e `s3`. Pede-se:

a) Aponte o caso base do algoritmo recursivo. Mostre que o algoritmo retorna o valor correto neste caso.

b) Mostre que o caso base é *sempre* atingido.

c) Mostre que o algoritmo está correto

d) Escreva a equação de recorrência que dita a ordem de complexidade do algoritmo no seu pior caso, em função do número de elementos do vetor N.

e) Resolva a equação. Compare o desempenho do algoritmo com os vistos em sala por busca direta ($\mathcal{O}\left(N^2\right)$) e por programção dinâmica ($\mathcal{O}\left(N\right)$).

Respostas

Exercício 1

a) O único laço do programa é o iniciado na linha 3 e encerrado na linha 6. A condição de permanência é `i<s.length()`. A variável `i` é inicializada com 0 e é incrementada ao final de cada iteração, enquanto que `s.length()` é imutável. Assim existe um número finito de iterações para o qual a condição de permanência deixa de ser atendida.

b) Se a cadeia é vazia, `s.length()` é 0, o laço nunca é executado e a função retorna 0. Para uma cadeia não-vazia, seja r_i o valor da variável `r` apos a execução da linha 5 na i-gésima iteração (quando a variável `i` vale i). Imediatamente, $r_0 = s_0$. Pelas linhas 4 e 5, a lei de recorrência para o valor de r_i é:

$$r_{i+1} = 10r_i + s_{i+1}$$

Lema 1. $r_i = \sum_{j=0}^{i} 10^{i-j} s_i$

Prova: O lema é trivialmente verdadeiro para $i = 0$, pois $r_0 = s_0$. Mas se existe um i para o qual ele é válido, então, pela lei de recorrência,

$$r_{i+1} = 10 \sum_{j=0}^{i} 10^{i-j} s_i + s_{i+1} = \sum_{j=0}^{i} 10^{(i+1)-j} s_i + s_{i+1} = \sum_{j=0}^{i+1} 10^{(i+1)-j} s_i$$

Demonstrando-se por indução finita o lema.

Ora, mas na condição de parada, $i = n$ e então o valor retornado é $\sum_{j=0}^{n} 10^{n-j} s_j$.

c) O laço executa exatamente N iterações, de modo que a complexidade é $\mathcal{O}\left(N\right)$.

Exercício 2

a) A ideia básica é inspecionar as cadeias um caractere por vez da esquerda para a direita até achar um ponto de discrepância, seja por diferença de caracteres, seja por diferença de comprimento. A dificuldade desta abordagem decorre essencialmente do elevado número de condições de parada do algoritmo. O laço do algoritmo pode parar quando:

1: A cadeia `s1` terminou.

2: A cadeia `s2` terminou.

3: Os caracteres considerado nas cadeias diferem entre si.

Nota-se que é possível que as condições 1 e 2 sejam *simultaneamente* atingidas, caso em que as cadeias são idênticas. Uma possível implementação é fazer um laço que itera sobre um índice e testa as 3 condições acima simultaneamente. Após o término do laço verifica-se qual das condições foi responsável pelo término.

```
def CompareStrings(s1, s2):
    n1 = len(s1)
    n2 = len(s2)
    i = 0 # O valor de i deve ser preservado apos o laco
```

```
while i<n1 and i<n2 and ord(s1[i]) == ord(s2[i]): i+=1
r = 0;
if i>=n1 : # Saiu por termino de s1
    if i<n2: r = -1  # s2 nao terminou, de modo que s1<s2
else:
    if i>=n2: r = 1 # s2 terminou mas s1 nao, de modo que s1>s2
    else: # As cadeias nao terminaram, os caracteres sao diferentes
        if ord(s1[i]) < ord(s2[i]): r=-1
        else: r=1
return r
```

Outra possibilidade menos estruturada é fazer um laço *permanente* (cuja condição de saída é *sempre* verdadeira) e testar as condições, forçando o retorno *imediato* da função quando alguma delas é atingida.

```
def CompareStrings(s1, s2):
    n1 = len(s1)
    n2 = len(s2)
    i = 0
    while True:
        if i>=n1:
            if i>=n2: return 0
            else: return -1
        if i>=n2: return 1
        c1 = ord(s1[i])
        c2 = ord(s2[i])
        if c1<c2: return -1
        if c1>c2: return 1
        i += 1
```

b) O pior caso do algoritmo é quando as cadeias são *idênticas* até o término da menor. Neste caso o número total de iterações é a quantidade de elementos da menor cadeia, ou $\mathcal{O}(\min(N, M))$.

Exercício 3

a) Em um vetor ordenado, cada elemento inspecionado é ou uma repetição do elemento anterior ou um elemento novo que ainda não apareceu. O único caso que deve ser considerado com mais cuidado é o de um vetor com comprimento nulo, mas um teste rápido no início do código resolve o problema:

```
def Repeticoes(a):
    if len(a)==0: return 0
    n = 0
    ultimo = a[0]
    for i in range(1, len(a)):
        if a[i]==ultimo: n += 1
        else: ultimo = a[i]
    return n
```

Este código faz exatamente $N - 1$ iterações, onde N é o tamanho do vetor. Alternativamente é possível comparar os elementos `a[i]` com `a[i-1]` (ou `a[i]` e `a[i+1]`, se o devido cuidado for tomado com os limites para `i`), mas é ligeiramente mais rápido usar uma variável local.

b) O algoritmo *mergesort* ordena vetores com complexidade $\mathcal{O}(N \log N)$. Deste modo a tarefa de se ordenar um vetor e aplicar o algoritmo do item acima é $\mathcal{O}(N \log N + N) = \mathcal{O}(N \log N)$.

Exercício 4

a) O caso base é o de uma sequência na qual o limite à esquerda é igual ao limite à direita, ou seja, o de uma sequência unitária. Naturalmente, a única subsequência (e consequentemente a maior) neste caso é a própria

sequência, e a soma é o único valor contido.

b) São feitas duas chamadas recursivas, uma com limites (`e`, `meio`) e outra com limites (`meio`+1, `d`). Claramente `meio` $<$ `d` e `meio` $+ 1 >$ `e`. Do mesmo modo, `meio` $\geq$ `e` e `meio` $+ 1 \leq$ `d`. Assim, os limites nas chamadas são estritamente decrescentes e limitados inferiormente por uma sequência unitária. Assim há um nível de chamada que leva à sequência unitária.

c) O algoritmo faz duas chamadas recursivas com vetores com metade do tamanho do vetor original. A primeira busca sequencial custa $\mathcal{O}(N/2)$ e a segunda custa $\mathcal{O}(N/2)$. A equação é:

$$T(N) = 2T\left(\frac{N}{2}\right) + \mathcal{O}(N)$$

d) Supondo que as chamadas nas linhas 8 e 9 retornam o valor correto, as variáveis `s1` e `s2` contém, respectivamente, a maior subsequência em (`e`, `meio`) e (`meio`+1, `d`). Por busca sequencial direta, a variável `s3` contém a maior subsequência que começa em (`e`, `meio`) e termina em (`meio`+1, `d`). Ora, mas toda subsequência possível do vetor ou começa e acaba antes da metade, ou começa antes da metade e acaba depois da metade, ou começa e acaba depois da metade. Assim, claramente o maior valor entre `s1`, `s3` e `s2` e contém o valor da maior subsequência.

e) Na notação do *master theorem*, $A = 2$, $B = 2$ e $L = 1$, de modo que $A = B^L$. A solução é assim $\mathcal{O}(N \log N)$. O algoritmo é assim melhor do que o de busca direta ($\mathcal{O}\left(N^2\right)$) e pior do que o por programação dinâmica ($\mathcal{O}(N)$).