Prova 1 — PMR3201 Computação para Automação 1o. semestre 2017

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) Uma sequência $A = \{a_0, \dots a_{2n-1}\}$ é dita *supercrescente* se:

$$a_i > \sum_{j=0}^{i-1} a_j$$

$$a_i > 0$$

Ou seja, cada elemento é *positivo* e *estritamente maior* que a soma dos seus anteriores. Por exemplo, a sequência $[1, 2, 3]$ *não* é supercrescente, enquanto que a sequência $[2, 3, 6]$ é. Escreva em Python uma função que verifica se uma dada seqûencia é supercrescente. Use a seguinte assinatura: `supercrescente(a)`, onde `a` é a sequência. A função deve retornar `True` se `a` é supercrescente e `False` caso contrário. Explique o comportamento da sua função no caso de uma sequência nula. Uma função com complexidade $\mathcal{O}(N)$, onde N é o tamanho da sequência, vale 2,5 pontos, complexidades piores valem 1,5.

Resposta:

```
def supercrescente(a):
    soma = 0
    for i in a:
        if soma > i:
            return False
        soma += i
    return a[0]>0
```

Note a necessidade de se verificar se o último elemento é positivo ao final. Esta função retorna `True` para sequências nulas.

2. (2,5 pontos) Seja $A = \{a_0, \dots a_{2n-1}\}$ uma sequência com número *par* de elementos. Chamemos de operação de *redução de pares* a operação em que cada par de elementos *consecutivos* (a_i, a_{i+1}) é substituído por a_i se $a_i = a_{i+1}$ ou é *eliminado* se $(a_i \neq a_{i+1})$. Por exemplo, esta operação aplicada à sequência $[1, 2, 2, 1, 1, 1, 3, 3, 3, 1]$ transforma-a na sequência $[1, 3]$ (verifique!).

   (a) (0,5 pontos) Seja N o tamanho original da sequência. O tamanho *máximo* da sequência obtida pela operação de redução de pares é $N/2$, e o tamanho mínimo é 0 (uma sequência nula). Forneça exemplos de ambos os casos com sequências de 6 elementos.

   Resposta: O tamanho máximo é obtido quando todos os pares de elementos são formados por dois elementos iguais e são reduzido a um elemento, o que produz uma lista com tamanho $N/2$. Exemplo: $[1, 1, 2, 2, 3, 3]$, com comprimento 6, que produz a sequência $[1, 2, 3]$.

   O tamanho mínimo é obtido quando todos os pares de elementos são formados por dois elementos diferentes e são eliminado, o que produz uma sequência nula. Exemplo: $[1, 2, 3, 4, 5, 6]$

   (b) (1,0 pontos) Escreva em Python a função `reduz` que recebe uma sequência com um número *par* de elementos e transforma-a (substituindo os elementos originais!) pela sequência resultante da

operação de redução de pares *sem uso de espaço auxiliar*. Você deve *modificar* a sequência recebida, inclusive no seu tamanho e só deve usar um número fixo de variáveis escalares. Use a seguinte assinatura:

```
def reduz(a):
```

Onde `a` é a sequência sobre a qual deve-se fazer a operação. *Lembrete:* Em Python, o tamanho de um vetor `a` pode ser *reduzido* a um tamanho `n` em tempo constante com o comando (com $\texttt{n} \leq \texttt{len(a)}$):

```
del a[n:]
```

Resposta:

```
def reduz(a):
    n = 0
    for i in range(0, len(a), 2): # dois em dois
        if a[i]==a[i+1]:
            a[n] = a[i]
            n += 1
    del a[n:] # Mantem somente os n primeiros elementos
```

(c) (1,0 pontos) Escreva em notação *Big Oh* a complexidade do seu algoritmo em função do tamanho da sequência original N.

Resposta: A complexidade desta implementação é $\mathcal{O}(N)$.

3. (2,5 pontos) O problema da Soma de Subconjuntos é o problema de se determinar se em uma sequência $A = \{a_0, \ldots a_{2n-1}\}$ existe uma subsequência cuja soma é exatamente um dado valor x. Por exemplo, existe uma subsequência em $[1, 3, 6, 2]$ cuja soma é exatamente 5 e *não existe* subsequência em $[1, 3, 6, 4]$ que some exatamente 12. Este é um problema computacional notoriamente difícil, para o qual não se conhecem algoritmos com complexidade sub-exponencial. Existe no entanto uma forma simples de resovê-lo quando a sequência A é *supercrescente* (vide questão 1):

```
1 def ExisteSubSoma(a,x):
2     for v in reversed(a): # Do fim para o começo
3         if x >= v: x -= v
4     return x==0
```

(a) (0,5 pontos) Seja N o tamanho da sequência `a`. Qual a complexidade do Algoritmo em função de N?

Resposta: O único laço do algoritmo percorre a sequência `a` em ordem reversa, consequentemente faz exatamente N iterações. Assim a complexidade do algoritmo é $\mathcal{O}(N)$.

(b) (2,0 pontos) Mostre que o algoritmo está correto, ou seja, ele retorna `True` se e somente se há uma subsequência de `a` cuja soma é exatamente `x`.

Resposta: O algoritmo trivialmente termina em tempo finito (vide item anterior). Seja $A = \{a_0, \ldots a_n\}$ a sequência em `a` (note que neste caso $n = N - 1$) e $A_{i:j} = \{a_i, \ldots a_j\}$ $i \leq j$ a subsequência de `a` iniciada no índice i e terminada no índice j. Seja n o *último* indice válido de

a (ou seja, $N-1$). Seja x_i o valor da variável a na i-gésima iteração do laço, *após* a execução da linha 3, com $x_0 = \texttt{x}$. A lei de recorrência do algoritmo é:

$$x_{i+1} = \begin{cases} x_i & \text{se, } x_i < a_{n-i} \\ x_i - a_{n-i} & \text{se, } x_i \geq a_{n-i} \end{cases}$$

Mostra-se que vale *sempre* a propriedade:

Lema 3.1 *Existe subsequência de A que soma exatamente x_0 se e somente se existe subsequência de $A_{0:(n-i)}$ que soma exatamente x_i.*

Esta propriedade é trivialmente verdadeira para $i = 0$. Vamos dividir esta afirmação em 3 sub-casos:

1. Existe uma subsequência S qualquer de $A_{0:n-i}$ cuja soma é exatamente x_i:
   (a) $x_i < a_{n-i}$: Evidentemente $a_{n-i} \notin S$ (pois $a_i > 0$, assim qualquer soma que inclua a_{n-i} será maior que x_i). e assim existe uma subsequência de $A_{0:n-(i+1)}$ cuja soma é exatamente $x_i = x_{i+1}$
   (b) $x_i \geq a_{n-i}$: Neste caso $a_{n-i} \in S$, pois, pela definição de supercrescente, qualquer soma de subsequência que não contém este valor será menor do que a_{n-1}. Novamente, existe uma subsequência de $A_{0:n-(i+1)}$ cuja soma é exatamente $x_i - a_{n-i} = x_{i+1}$
2. Não existe subsequência S qualquer de $A_{0:n-i}$ cuja soma é exatamente x_i: Então não pode existir subsequência de $A_{0:n-i}$ cuja soma seja exatamente x_i nem $x_i - a_{n-i}$ (caso contrário seria trivial construir a sequência desejada). Assim, não existe subsequência de $A_{0:n-i}$ seja x_{i+1}.

Ora, mas na condição de saída, $i = n$ e $A_{0:n-i+1} = \{\}$, cuja soma é 0. Assim, existe subsequência de A cuja soma é exatamente x_0 se e somente se $x_n = 0$.

4. (2,5 pontos) *Nota:* O enunciado publicado na prova contém um erro na listagem abaixo. Aqui está publicada a versão correta. Infelizmente este erro afeta a resposta do item (c). Vide solução deste item para mais detalhes.

   Uma sequência $A = \{a_0, \ldots a_{n-1}\}$ possui um *elemento dominante* se exite um elemento que se repete em mais da metade das posições de A (ou seja, existe x tal que $|\{i \in \mathbb{N} | a_i = x\}| > n/2$). Por exemplo, a lista $\{1, 2, 3, 2\}$ *não* possui elemento dominante (o elemento 2 se repete apenas duas vezes e a seqüencia possui 4 elementos). Já a lista $\{1, 2, 2\}$ possui elemento dominante (o elemento 2 se repete duas vezes e a sequência possui 3 elementos).

   (a) (0,5 pontos) Seja A_0 uma sequência com número *par* de elementos, e A_1 o resultado da operação de *redução de pares* (vide questão 2) aplicada a A_0 . Vale a seguinte propriedade: se x é elemento dominante de A_0, então x também é elemento dominante de A_1. A implicação reversa, no entanto, *não* é válida, ou seja, existem sequências resultantes A_1 que possuem um elemento dominante x que *não* é elemento dominante de A_0. Forneça um exemplo deste último caso, ou seja, uma sequência que *não* possui elemento dominante, mas que, submetida à operação de redução, produz uma sequência com elemento dominante.

   Resposta: A sequência $[1, 2, 3, 3]$, sob a operação de redução, gera a sequência $[3]$. 3 é trivialmente elemento dominante da segunda sequência, mas *não* é da primeira.

   (b) (1,0 pontos) Considere a função:

```
def ExisteElementoDominante(a):
    v = list(a) # Cópia:  Complexidade O(len(a))
    n = len(a)
    def TestaRecursivamente():
        if len(v) == 0: # Não há mais candidato!
            return False
        if len(v)%2!=0: # se v tem numero ímpar de elementos...
            # Verifica se o último elemento é dominante
            if v.count(v[-1])*2 > len(v): # Contagem:  Complexidade O(len(v))
                return a.count(v[-1])*2 > n # Complexidade O(len(a))
            v.pop()    # Retira o último elemento:  Complexidade O(1)
        reduz(v) # Presuma aqui complexidade O(len(v))
        return TestaRecursivamente()
    return TestaRecursivamente()
```

Esta é uma função recursiva que usa internamente uma função `reduz(a)`, similar à descrita na questão 2. Mostre que a função `ExisteElementoDominante(a)` termina em tempo finito e retorna `True` se a sequência em `a` possui elemento dominante, `False` caso contrário.

Sugestão: Você pode usar as propriedades do item (a) da questão 2 e do item (a) desta mesma questão. Note que é preciso mostrar que a chamada a `reduz(v)` é *sempre* válida.

Resposta: A função recursiva verifica se a sequência é nula e retorna verdadeiro neste caso. Em seguida ela verifica se a sequência tem um número *ímpar* de elementos. Caso tenha, ela verifica se o seu último elemento é dominante da sequência original. Finalmente ela chama `reduz` sobre a sequência e a si mesma.

Esta função recursiva tem dois casos-base:

1. A sequência recebida é nula, situação em que a função retorna `False`.
2. A sequência recebida tem número *ímpar* de elementos e seu último elemento é elemento dominante da sequência `v`, situação em que a função retorna `True` caso o elemento também seja, `False` caso contrário.

O primeiro caso base é correto, pois pela propriedade do item (a), se alguma sequência `v` não tem elemento dominante, a sequência original `a` também não pode ter. O segundo caso-base também é, pois se o último elemento de `v` é dominante em `v`, ele é o único candidato possível a dominância de `a`.

A chamada a `reduz` é *sempre* válida, pois se o vetor `v` tem inicialmente número ímpar de elementos, o seu último elemento é retirado. Ora, mas a chamada a `reduz` garante que o tamanho do vetor `v` na próxima chamada estará entre 0 e $\lfloor \texttt{len(v)}/2 \rfloor$. Assim, algum caso-base é *sempre* atingido.

Ora, mas a chamada recursiva retorna o valor correto, então necessariamente a função retorna o valor correto (de fato, trata-se de um *tail call*).

(c) (1,0 pontos) Escreva a equação de recorrência da complexidade da chamada a `ExisteElementoDominante(a)` em função do comprimento da sequência `a` N. Resolva a equação e obtenha a complexidade da função.

Atenção: Aqui você deve adotar uma complexidade da chamada à função `reduz(v)` $\mathcal{O}(N)$, onde N é o comprimento da sequência `v`.

Resposta: O pior caso da função recursiva é o no qual o vetor `v` é sempre *ímpar* e não possui elemento dominante. Neste caso a função recursiva faz $\mathcal{O}(N)$ operações e chama a si mesma com no máximo $N/2$ elementos, onde N é o tamanho de `v`. Assim, a equação recursiva é:

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

Cuja solução é:

$$T(N) = \mathcal{O}(N)$$

O enunciado original não faz a verificação final sobre o vetor `a`, o que torna o algoritmo *incorreto*. Durante a prova pediu-se aos alunos que substituissem o teste `v.count(v[-1])*2 > len(v)` por `a.count(v[-1])*2 > len(v)`. Este teste tem complexidade `len(a)`, que *não* depende do tamanho de `v`. A equação resultante é:

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

Cuja solução é $\mathcal{O}(\log N\,(1 + \mathcal{O}(\texttt{len(a)}))) = \mathcal{O}(N \log N)$ Em virtude do erro no enunciado, *ambas* as respostas serão consideradas corretas.

Formulário

Somas de sequências:

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

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

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(X)
    if C(X) then
        return E(X)
    else
        return F(G(X))
    end if
end function
```

Versão recursiva

```
function F(X)
    while NOT C(X) do
        X ← G(X)
    end while
    return E(X)
end function
```

Versão iterativa com laço