Lista 1 - PMR3201

Thiago Martins, Fabio G. Cozman

2019

Exercícios

1. (PSUB 2018) Seja a seguinte função, $f(x)$ definida de inteiros positivos para inteiros positivos: Se x é par, $f(x) = x/2$. Se x é ímpar, $f(x) = 3x + 1$. Considere a sequência $\{a_0, a_1, \ldots, a_n, \ldots\}$ tal que $a_{i+1} = f(a_i)$. A conjectura de Collatz propõe que, não importa qual o valor de a_0 (desde que inteiro positivo), existe *sempre* um valor n finito tal que $a_n = 1$. A veracidade da conjectura de Collatz ainda é um problema em aberto na matemática.

   a) Escreva uma função que, para um dado a_0, calcula o menor valor de n tal que $a_n = 1$. Use a seguinte assinatura:

      ```
      def collatz_n(a):
      ```

      Onde `a` é um inteiro positivo com o valor a_0 de sua sequência. Sua função deve retornar o primeiro n tal que $a_n = 1$.

   b) O código que você escreveu corresponde a um algoritmo? Discuta.

2. (P1 2017) Uma sequência $A = \{a_0, \ldots a_{2n-1}\}$ 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. A sua função deve ter complexidade linear $\mathcal{O}(N)$. Explique o comportamento da sua função no caso de uma sequência nula.

3. (P1 2017) Seja $A = \{a_0, \ldots a_{2n-1}\}$ 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) 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.

   b) 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 `n` $\leq$ `len(a)`):

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

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

4. (Adaptado de PSUB 2016) Uma sequência A é dita *subsequência* de outra sequência B se é possível transformar B em A removendo-se elementos mantendo-se a ordem original. Por exemplo, a sequência $\{2, 5, 1\}$ é subsequência da sequência $0, 2, 1, 1, 5, 2, 1$. Por outro lado, a sequência $\{1, 2, 2\}$ *não é* subsequência da sequência $2, 0, 1, 2$. Escreva uma função em Python que determina se uma sequência é subsequência de outra. Use a seguinte assinatura:

```
def checa_subsequencia(x, y):
```

Esta função deve retornar `True` se o vetor `x` é subsequência do vetor `y` e `False` caso contrário. O seu algoritmo deve ter complexidade *linear*, ou seja, $\mathcal{O}(N)$, onde N é o tamanho do vetor `y`.

5. (Adaptado da PSUB-2015) O algoritmo de Euclides Extendido é um dos algoritmos necessários para a geração de chaves de criptografia RSA. Como o nome sugere, ele é uma *extensão* do clássico algoritmo de Euclides para calcular o máximo divisor comum. Além de calcular, como o algoritmo de Euclides, o máximo divisor comum r de a e b, ele também calcula um par de inteiros x, y tal que:

$$r = ax + by$$

A listagem a seguir mostra uma implementação do Algoritmo de Euclides Extendido. Ela retorna a tupla r, x, y.

```
1  def egcd(a, b):
2      xa, ya = 1, 0
3      xb, yb = 0, 1
4      while b !=0 :
5          q = a // b      # Quociente
6          r = a - q * b   # Resto
7          a = b;
8          b = r;
9          xa, xb = xb, xa - q * xb
10         ya, yb = yb, ya - q * yb
11     return a, xa, ya
```

Este algoritmo supõe sempre que $a > b$. Seja a_i, b_i, x_{ai}, y_{ai}, x_{bi}, y_{bi} os valores das variáveis `a`, `b`, `xa`, `ya`, `xb`, `yb` ao final da i-gésima iteração do laco iniciado na linha 4, e a_0 ,b_0 , x_{a0}, y_{a0}, x_{b0}, y_{b0} os valores destas variáveis antes da entrada no laço. Da análise do Algoritmo de Euclides tradicional feita em sala são conhecidos os seguintes fatos:

- O algoritmo termina em tempo finito (note que o controle de fluxo deste algoritmo é idêntico ao algoritmo convencional).
- $\text{gcd}\,(a_0, b_0) = \text{gcd}\,(a_i, b_i)$.
- $b_n = 0 \Rightarrow \text{gcd}\,(a_0, b_0) = a_n$

Dado o exposto acima, demonstre que a implementação do algoritmo de Euclides Extendido está *correta*.

Sugestão: Mostre como vale em toda iteração

$$a_i = x_{ai}a_0 + y_{ai}b_0$$
$$b_i = x_{bi}a_0 + y_{bi}b_0$$

6. (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.

7. (PREC 2017) Seja S uma progressão aritmética de razão unitária inciada em 0 da qual um elemento foi removido.

Exemplos:

- $\{0, 1, 2, 3, 5, 6\}$: O elemento 4 foi removido.
- $\{1, 2, 3, 4, 5, 6\}$: O elemento 0 foi removido.
- $\{0, 1, 2, 3, 4, 5\}$: O elemento 6 foi removido.

Considere o código a seguir:

```
def elemento_faltante(a):
    def elemento_faltante_rec(esquerda, direita):
        if esquerda >= direita:
            return esquerda
        else:
            m = (esquerda+direita)//2
            if a[m] > m:
                return elemento_faltante_rec(esquerda, m)
            else:
                return elemento_faltante_rec(m+1, direita)
    return elemento_faltante_rec(0, len(a))
```

A função `elemento_faltante(a)` retorna o elemento faltante da sequência armazenada no vetor `v`.

a) Mostre que o algoritmo está correto, ou seja, efetivamente retorna o elemento faltante.

b) Calcule a complexidade do algoritmo.

8. (P1 2018) Em uma sequência de inteiros $A = \{a_1, \ldots a_n\}$ 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) 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!

b) Qual a complexidade do seu algoritmo?

9. (Adaptado da P1 2015) Considere o código abaixo para calcular a exponenciação x^a:

```
1 def exp(x, a):
2     r = 1
3     while a!=0:
4         if (a%2)!=0: # Verdadeiro se a impar
5             r *= x
6             a -= 1
7         a //= 2
8         x = x*x
9     return r
```

Para este algoritmo são relevantes as seguintes propriedades da exponenciação:

- Se a é par então $x^a = \left(x^2\right)^{\lfloor a/2 \rfloor}$
- Para a positivo vale $x^a = x \cdot x^{a-1}$

a) Moste que o algoritmo termina em tempo finito para a não-negativo.

b) Mostre que o algoritmo acima efetivamente retorna x^a (sugestão: Considere como varia a cada iteração o valor de $\mathtt{r} \cdot \mathtt{x}^{\mathtt{a}}$ após a execução da linha 8).

10. (PREC 2015) Considere o código abaixo para converter um inteiro maior do que zero em uma String com sua representação decimal:

```
def convert(x):
    r = ""
    while(x!=0):
        d = x % 10
        # Adiciona o digito de d
        # a esquerda de r
        r = chr(ord('0') + d) + r
        x //= 10
    return r
```

a) Mostre que o algoritmo termina em tempo finito para `x` maior do que zero.

b) Mostre que o algoritmo acima está correto, ou seja, ele efetivamente retorna a representação decimal de x.

Sugestão: seja i o número da iteração do laço interno. Considere ao final de cada iteração do laço os valores do número representado pela String armazenada em `r` e de $\mathtt{x} \cdot 10^i$.

11. (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:

```
def atoi(s):
    r = 0
    for i in range(len(s)):
        r *= 10
        r += ord(s[i]) - ord('0')
    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.

12. Considere o problema de se calcular o valor de um polinômio:

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

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

```
def polinomio(a, x):
    i = len(a)-1
    p = 0
    while i>= 0:
        p *= x
        p += a[i]
        i -= 1
    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) Mostre que o algoritmo está correto.

b) Calcule a complexidade do algoritmo.

13. (Adaptada da P1-2015) O produto de duas matrizes $\boldsymbol{A} = \boldsymbol{X} \cdot \boldsymbol{Y}$ quadradas $n \times n$ é definido como:

$$A_{i,j} = \sum_{k=1}^{n} X_{i,k} Y_{k,j}$$

Em Python, matrizes podem ser representadas por listas de linhas, as linhas por sua vez, listas de pontos flutuantes. Assim, por exemplo, se a variável `a` é uma referência a uma matriz $\boldsymbol{A}$, então o elemento $A_{i,j}$ pode ser acessado por `a[i][j]`.

a) Com a representação acima, escreva uma função em Python que recebe 3 matrizes quadradas, `x`, `y` e `a` *de tamanhos idênticos*. A função deve escrever na matriz `a` o resultado do produto matricial de `x` e `y`. Use a seguinte assinatura:

```
def matrixmult(x, y, a):
```

b) Escreva em função de N , onde N é o número de linhas/colunas das matrizes quadradas, a ordem de complexidade da função escrita.

14. (PREC 2018) Uma matriz bidimensional em Python pode ser representada por uma sequência de sequências. Assim, a matriz

$$X = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

pode ser escrita em Python como:

```
X = [[1, 2], [3, 4]]
```

O elemento $X_{1,2}$ da matriz acima, cujo valor é 2, pode ser acessado através da variável `x` acima declarada com a seguinte sintaxe: `x[0][1]` (note a convenção em Python de se utilizar índices baseados em 0).

Seja uma matriz quadrada $N \times N$ cujos coeficientes são 0 ou 1. Em cada linha da matriz, um 0 *jamais* antecede um 1. Por exemplo:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Escreva em Python uma função que encontra com complexidade $\mathcal{O}(N)$ o índice (baseado em zero) da linha da matriz com a *maior* quantidade de 0's. Utilize a seguinte assinatura:

```python
def encontra_mais_zeros(x):
```

onde `x` é uma referência à sequência de sequências que representa a matriz acima descrita. Sua função deve retornar o índice da sequência com a linha que contém a maior quantidade de 0's.

15. Escreva uma rotina recursiva que resolve com custo $\mathcal{O}\left(2^N\right)$ o seguinte problema: Dado um conjunto de N inteiros a_1 a a_N, e um inteiro K maior que todos os A_i, descubra se existe um subconjunto dos inteiros A_1 a A_N cuja soma é exatamente K. (Nota: Esse problema é chamado Problema da Soma de Subconjuntos; não existe nenhuma solução conhecida para esse problema que seja polinomial em N. A descoberta de uma solução polinomial para esse problema é um dos problemas abertos mais importantes em matemática, e o problema mais importante em computação teórica.)

16. (P1 2017) 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 (vide problema 15). 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 2):

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

a) Seja N o tamanho da sequência `a`. Qual a complexidade do Algoritmo em função de N?

b) 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`.

17. O problema 15 pode ser resolvido com complexidade *pseudo*-polinomial $\mathcal{O}(kN)$, onde k é o inteiro que deseja-se obter com uma soma e N é o tamanho do conjunto de inteiros (note que isso *não* é o mesmo que complexidade quadrática!). Encontre essa solução por programação dinâmica. Suponha por simplicidade que $k \geq 0$ e $a_i > 0$.

    Sugestão: Seja $S[k, A_{1:n}]$ a solução do problema. Suponha que você tenha todas as soluções $S[i, A_{1:n-1}]$ para $0 \leq i \leq k$. Você consegue com estas calcular $S[k, A_{1:n}]$ em complexidade constante?

18. (P1 2019) 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:

```python
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 $\mathcal{O}(k)$?

19. (PREC 2015) O código a seguir mostra uma implementação em python do algoritmo *Bubblesort* para ordenação em ordem crescente de vetores de inteiros:

```
def bubblesort(x):
    ultimo = len(x)-1
    while ultimo > 0:
        ultimo_alterado = 0
        for j in range(1, ultimo + 1):
            if x[j]<x[j-1]:
                temp = x[j]
                x[j] = x[j-1]
                x[j-1] = temp
                ultimo_alterado = j
        ultimo = ultimo_alterado - 1;
```

O princípio do algoritmo é comparar a ordem relativa de elementos contíguos em um vetor e realizar a troca de posição sempre que necessário. Note que ao final do laço interno, todos os elementos entre as posições apontadas por `ultimo` e `ultimo_alterado` estão em ordem crescente e são maiores ou iguais a todos os elementos que os precedem. Como `ultimo` é inicializado com a última posição do vetor, e a cada iteração do laço externo é modificado para a posição anterior a `ultimo_alterado`, o algoritmo termina em tempo finito e ordena efetivamente o vetor.

a) Calcule em notação *Big Oh* a ordem de complexidade do algoritmo Bubblesort no seu *pior caso*.

b) Calcule em notação *Big Oh* a ordem de complexidade do algoritmo Bubblesort no seu *melhor caso*.

20. Prove que o seguinte algoritmo ordena corretamente um arranjo `a` entre índices `i` e `j` (inclusive), e obtenha a complexidade do algoritmo em notação BigOh.

```
def ordena(a, i, j):
    if a[i] > a[j]:
        a[i], a[j] = a[j], a[i]
    if (i+1) >= j:
        return
    k = (j-i+1)//3
    ordena(a,i,j-k)
    ordena(a,i+k,j)
    ordena(a,i,j-k)
```

21. (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 $\mathcal{O}(N \log N)$.

22. (P1 2015) O fator h é uma métrica que mede a produção de um pesquisador, que combina quantidade de publicações com frequência na qual estas são citadas. Define-se fator h de um pesquisador como o número *máximo* n de publicações que ele produziu que foram citadas ao menos n vezes cada uma. Por exemplo, um pesquisador que possui 5 publicações que foram citadas respectivamente $\{5, 4, 3, 2, 1\}$ vezes possui um fator h de 3, visto que as 3 publicações mais citadas foram citadas ao menos 3 vezes cada e nenhuma das subsequentes foi citada 4 vezes. Já um pesquisador com 10 publicações que foram citadas $\{11, 10, 6, 5, 4, 3, 2, 1, 1, 1\}$ vezes possui um fator h de 4 (verifique!).

a) Escreva uma função em Python que, dado um vetor com a quantidade de citações que cada uma das publicações de um pesquisador recebeu *ordenado em ordem decrescente*, calcula o fator h. A função deve possuir a seguinte assinatura:

```
def fatorh(citacoes):
```

Nota: há uma solução trivial com complexidade $\mathcal{O}(N)$ mas o aluno deve ser capaz de encontrar uma com complexidade $\mathcal{O}(\log N)$.

b) Explique como, dado o seu conhecimento de algoritmos, é possível calcular o fator h a partir de uma lista *desordenada* com complexidade $\mathcal{O}(N \log N)$.

23. (P1 2015) A listagem a seguir mostra uma implementação do algoritmo de transformada rápida de Fourier segundo o algoritmo de Cooley-Tuckey:

```python
import math
import cmath

def fft(x):
    # Funcao recursiva
    def fft_rec(x, y, start, n, s):
        if n==1: return

        fft_rec(y, x, start, n//2, s*2)
        fft_rec(y, x, start+s, n//2, s*2)

        for k in range(n//2):
            arg = -2*math.pi*(k/n)
            t = y[start+s*(2*k+1)]*cmath.rect(1, arg)
            x[start+k*s] = y[start+k*2*s] + t
            x[start+(n//2+k)*s] = y[start+k*2*s]- t

    # aux recebe uma copia de x
    # ATENCAO: Esta operacao tem complexidade O(len(x))!
    aux = list(x)
    # Chama a funcao recursiva
    fft_rec(x, aux, 0, len(x), 1)
```

Este algoritmo trabalha *exclusivamente* com vetores cujo tamanho é uma potência de 2. O seu ponto de entrada é a função `fft`, que por sua vez chama a função recursiva `fft_rec`.

a) Mostre que o algoritmo *sempre* termina para um vetor com tamanho igual a uma potência de 2.

b) Escreva a equação de recorrência que dá a ordem, em notação *big Oh*, do tempo de execução deste algoritmo para uma entrada de tamanho N. Considere que todas as operações com números complexos têm complexidade constante $\mathcal{O}(1)$.

c) Resolva a equação e obtenha a ordem em notação *big Oh* do tempo de execução do algoritmo.

24. (PSUB 2015) Considere o problema de se determinar se um núumero está ou não presente em uma matriz na qual todas as colunas e linhas estão ordenadas em ordem crescente. Exemplo:

$$\begin{pmatrix} 1 & 3 & 10 & 23 \\ 4 & 7 & 18 & 36 \\ 7 & 12 & 26 & 44 \end{pmatrix}$$

Uma maneira de se resolver este problema é dividir a matriz em quatro quadrantes, noroeste, nordeste, sudeste e sudoeste. Em seguida, o valor na posição no *centro* da matriz é verificada. Caso ele seja maior do que o núumero procurado, o quadrante "sudeste" (inferior direito) é removido da busca (acrescido da coluna e linha centrais) e a busca prossegue nos 3 quadrantes seguintes. Caso ele seja menor, o quadrante "noroeste" (acrescido da coluna e linha centrais) é eliminado da busca e ela prossegue nos 3 quadrantes restantes. A listagem a seguir contém uma implementação em Java deste algoritmo:

```python
def procura (v, a):
    def procura_rec(o, e, n, s):
        if o > e or n > s : return False
```

```
        x = ( o + e )//2  # Coordenadas do
        y = ( n + s )//2  # centro
        t = a [y][x]
        if v == t : return True  # Achou
        elif v > t:   # Descarta canto NO
            return procura_rec(x + 1, e, n, y ) or\
            procura_rec(x + 1, e, y + 1, s) or\
            procura_rec(o, x, y + 1, s)
        else:  # Descarta canto SE
            return procura_rec(x, e, n, y - 1) or\
            procura_rec(o, x - 1, y ,s) or\
            procura_rec(o, x - 1, n , y - 1)

    return procura_rec(0 , len(a[0])-1, 0 , len(a)-1)
```

A matriz é representada por um vetor de linhas, passado no parâmetro `a`. O parâmetro `v` contém o valor a ser procurado. A função `procura(v, a)` chama a função recursiva `procura_rec(o, e, n, s)`, na qual os parâmetros `o`, `e`, `n` e `s` são os limites da matriz a oeste (esquerda), leste (direita), norte (acima) e sul (abaixo).

a) Escreva a equação de recorrência da ordem de complexidade deste algoritmo em função de N, o número total de elementos da matriz.

b) Resolva a equação de recorrência do item anterior e compare este algoritmo com uma busca sequencial na matriz.

25. (P1 2018) Há como resolver a questão 8 com complexidade $\mathcal{O}(N \log N)$ com uma extensão do algoritmo de ordenação mergesort. Considere a listagem:

```
def mergesort(v):
    temp = [None]*len(v)  # Complexidade O(n)
    def merge(e, m, d):
        i = e
        j = m
        k = e
        inv = 0
        while i < m and j < d:
            if v[i] <= v[j]:
                temp[k] = v[i]
                i += 1
            else:  # O elemento à direita é menor do que i, conte inversões
                inv += m - i
                temp[k] = v[j]
                j += 1
            k += 1
        # Completa com os elementos restantes
        while i<m:
            temp[k] = v[i]
            i += 1
            k += 1
        while j<d:
            temp[k] = v[j]
            j += 1
            k += 1
        # Copia o resultado do vetor temporário
        # de volta em v
        for k in range(e, d):
            v[k] = temp[k]
        return inv

    def merge_recursiva(e, d):
        if d - e <= 1: return 0
        inve = merge_recursiva(e, (e+d)//2)
        invd = merge_recursiva((e+d)//2, d)
        invm = merge(e, (e+d)//2, d)
        return inve+invd+invm

    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.

26. (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 ($\mathcal{O}\left(N\right)$).

27. Resolva a seguinte equação, resultado da análise de um algoritmo recursivo:

$$T(N) = aT(N/b) + N \log N,$$

para:

(a) `a` igual a 3 e `b` igual a 2.

(b) `a` igual a 2 e `b` igual a 2.

(c) `a` igual a 2 e `b` igual a 3.

Assuma que N é uma potência na base mais conveniente e que $T(1) = 1$.

28. (P1 2015) O código a seguir mostra uma implementação em Python do algoritmo de ordenação Quicksort para vetores de inteiros entre os índices a e b:

```
def quicksort(s, a, b):
    if a>=b:
        return
    p = s[b]      # pivot
    l = a
    r = b-1
    while l<=r :
        while l<=r and s[l]<=p:
            l = l+1
        while l<=r and s[r]>=p:
            r = r - 1
        if l<r:
            temp = s[l]
            s[l] = s[r]
            s[r]=temp
    s[b] = s[l]
    s[l] = p
    quicksort(s, a, (l-1))
    quicksort(s, (l+1), b)
```

Esta implemenação escolhe como pivô o último elemento do vetor. O vetor é dividido em dois sub-vetores, um de a a $l-1$ e outro de $l+1$ a b de modo que no primeiro há somente elementos menores ou iguais ao pivô e no segundo somente elementos maiores ou iguais ao pivô.

a) Em sua tese de doutorado em 1975, Robert Sedgewick propôs uma alteração no algoritmo: A *primeira* chamada recursiva do algoritmo deve ser feita sobre o *menor* dos sub-vetores. Reescreva o algoritmo acima de modo a comparar o tamanho relativo dos subvetores após a divisão e fazer as chamadas recursivas de acordo com o proposto por Sedgewick.

b) Reescreva o algoritmo do item (b) de modo a transformar a chamada de cauda (ou seja, a *última* chamada da função) em um laço.

c) O objetivo da modificação de Sedgewick é reduzir a profundidade máxima da árvore de chamadas e consequentemente o uso de memória de pilha pelo algoritmo. Qual a ordem em notação *big Oh* da profundidade máxima da árvore de chamadas do algoritmo produzido no item (c) para um vetor de N posições?

29. (P1 2017) Uma sequência $A = \{a_0, \dots a_{n-1}\}$ 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 seqûencia 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) Seja A_0 uma sequência com número *par* de elementos, e A_1 o resultado da operação de *redução de pares* (vide questão 3) 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.

b) 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.

c) 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`.

Soluções

Exercício 1

a) Uma solução simples é aplicar o procedimento descrito até que a condição $a = 1$ seja atingida e contar as iterações:

```
def collatz_n(a):
    i = 0
    while a != 1:
        if a%2:
           a = 3*a + 1
        else:
           a //= 2
        i += 1
    return i
```

b) O término em tempo finito da função acima implica na veracidade da conjectura de Collatz, o que pelo enunciado ainda é um problema em aberto. Assim, não se sabe se o código representa ou não um algoritmo.

Exercício 2

```
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.

Exercício 3

a) 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) A solução consiste em verificar que o resultado pode ser escrito no vetor original enquanto o mesmo está sendo lido (afinal, a posição a ser lida é sempre maior ou igual à posição de escrita no resultado).

```
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) A complexidade desta implementação, com um laço simples, é $\mathcal{O}(N)$.

Exercício 4

A solução é manter dois índices, um para os elementos em `x`, outro para os elementos em `y`. Os elementos em `y` são inspecionados sequencialmente. A cada vez que o elemento em `x` é igual ao elemento em `y`, o índice de `x` é incrementado. Se por este processo chega-se ao final da sequência em `x`, retorna-se verdadeiro, falso caso contrário.

```
static boolean checaSubsequencia(int[] x, int[] y) {
    int i, j;
    for(i = j = 0; i<y.length && j < x.length; i++)
        if(x[j]==y[i]) j++;
    return j==x.length;
}
```

Exercício 5

Do sabido sobre o algoritmo de Euclides, mostra-se que o algoritmo termina em tempo finito e que o valor final obtido, a_n (sendo n a última iteração do laco iniciado na linha 4) é efetivamente o valor de gcd(a_0, b_0). Resta provar que $a_n = x_{an}a_0 + y_{an}b_0$.

Sabe-se pelas linhas de 8 a 18 do algoritmo que

$$
\begin{align}
a_{i+1} &= b_i \tag{1}\\
q_{i+1} &= \left\lfloor \frac{a_i}{b_i} \right\rfloor \tag{2}\\
b_{i+1} &= a_i - q_{i+1}b_i \tag{3}\\
x_{bi+1} &= x_{ai} - q_{i+1}x_{bi} \tag{4}\\
y_{bi+1} &= y_{ai} - q_{i+1}y_{bi} \tag{5}\\
x_{ai+1} &= x_{bi} \tag{6}\\
y_{ai+1} &= y_{bi} \tag{7}
\end{align}
$$

A relação

$$
\begin{aligned}
a_i &= x_{ai}a_0 + y_{ai}b_0\\
b_i &= x_{bi}a_0 + y_{bi}b_0
\end{aligned}
$$

é trivialmente verdadeira para $i = 0$, visto que $x_{a0} = y_{b0} = 1$ e $y_{a0} = x_{b0} = 0$.
Ora, mas por indução finita, supondo que a relação é válida para i, segue-se que:

$$
\begin{aligned}
x_{bi+1}a_0 + y_{bi+1}b_0 &= (x_{ai} - q_{i+1}x_{bi})\,a_0 + (y_{ai} - q_{i+1}y_{bi})\,b_0\\
&= \overbrace{x_{ai}a_0 + y_{ai}b_0}^{a_i} - q_{i+1}\overbrace{(x_{bi}a_0 + y_{bi}b_0)}^{b_i}\\
&= a_i - q_{i+1}b_i\\
&= b_{i+1}
\end{aligned}
$$

e

$$
x_{ai+1}a_0 + y_{ai+1}b_0 = x_{bi}a_0 + y_{bi}b_0 = b_i = a_{i+1}
$$

Logo a relação é verdadeira em todas as iterações do algoritmo.

Exercício 6

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 $\mathcal{O}\left(\min\left(N, M\right)\right)$.

Exercício 7

a) Seja a sequência $A = \{a_0, a_1, \ldots a_{n-1}\}$.

Se o índice do elemento removido é r, então a expressão para o i-gésimo elemento da sequência é:

$$a_i = \begin{cases} i & \text{para } i < r \\ i+1 & \text{para } i \geq r \end{cases}$$

Assim, $a_m > m \Rightarrow m \geq r$ e $a_m \leq m \Rightarrow m < r$.

Sejam e, d, m os valores das variáveis `esquerda`, `direita` e `m` respectivamente.

O algoritmo descrito é uma função recursiva que procura o elemento faltante entre os índices e e $d - 1$.

O caso base do algoritmo é $e \geq d$. Este caso representa uma subsequência de comprimento zero iniciada na posição e. Neste caso, o único índice possível é e e o algoritmo retorna a resposta correta.

Nos demais casos, o algoritmo calcula o índice

$$m = \left\lfloor \frac{e+m}{2} \right\rfloor$$

e chama a si mesmo ou com os índices (e, m) ou $m + 1, d$. De todo modo, vale $m < d$ e $m + 1 > e$, o que mostra que o caso base é *sempre* atingido, de modo que ele sempre termina em tempo finito.

Ora, pelas propriedades expostas anteriormente, se $e \leq r \leq d$ e $a_m > m$ então vale $e \leq r \leq m$ e a chamada na linha 8 retorna a resposta correta do problema. Do mesmo modo, se $a_m \leq m$ então vale $m + 1 \leq r \leq d$ e a chamada na linha 10 retorna a resposta correta do problema.

b) Cada sub-sequência tem no máximo *metade* do tamanho da sequência original. O algoritmo faz operações em tempo constante chama a si mesmo uma vez com metade da entrada. Assim a complexidade é $\mathcal{O}(\log N)$.

Exercício 8

a) 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:

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

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

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

Naturalmente, a resposta a este item depende da particular solução encontrada para o item a).

Exercício 9

a) Seja i o número de iterações realizadas pelo laço da linha 3, a_i, r_i e x_i o valor das variáveis `a`, `r` e `x` respectivamente *após* a execução da linha 8 na i-gésima iteração. Assim, pelas linhas 6 e 7, $a_{i+1} < a_i$. No entanto, $a_i \geq 0$, pois a variável `a` só é submetida a operações de divisão por 2 e decremento *quando é par não-nulo*. Conclui-se daí que a_i é estritamente decrescente mas limitado inferiormente por 0. Como a_i é uma sequência inteira, há um número finito possível de iterações.

b) Há dois caminhos que o algoritmo pode seguir, dependendo da paridade da variável a_i:

a_i é par:

Neste caso:

$$a_{i+1} = \left\lfloor \frac{a_i}{2} \right\rfloor = \frac{a_i}{2}$$
$$r_{i+1} = r_i$$
$$x_{i+1} = {x_i}^2$$

Neste caso,

$$r_{i+1}{x_{i+1}}^{a_{i+1}} = r_i\left({x_i}^2\right)^{\frac{a_i}{2}} = r_i{x_i}^{a_i}$$

a_i é ímpar:

Neste caso:

$$a_{i+1} = \left\lfloor \frac{a_i - 1}{2} \right\rfloor = \frac{a_i - 1}{2}$$
$$r_{i+1} = r_i x_i$$
$$x_{i+1} = {x_i}^2$$

Neste caso,

$$r_{i+1}{x_{i+1}}^{a_{i+1}} = r_i x_i\left({x_i}^2\right)^{\frac{a_i-1}{2}} = r_i x_i {x_i}^{a_i - 1} = r_i {x_i}^{a_i}$$

Seja como for, vale $r_{i+1}{x_{i+1}}^{a_{i+1}} = r_i{x_i}^{a_i}$, ou seja, o valor $r_i{x_i}^{a_i}$ é *invariante* no algoritmo. Ora, mas para $i = 0$, vale $r_i{x_i}^{a_i} = {x_0}^{a_0}$, ou seja, o valor x^a procurado. Por outro lado, se n é a iteração final na qual $a_n = 0$, então $r_n{x_n}^{a_n} = r_n$, que é o valor retornado pelo algoritmo. Deste modo, conclui-se que $r_n = {x_0}^{a_0}$, ou seja, o algoritmo efetivamente retorna o valor x^a.

Exercício 10

a) Seja x_i o valor da variável `x` ao final da execução da linha 8, na i-gésima iteração do laço iniciado na linha 3. Pela linha 8, $x_{i+1} = \lfloor x_i/10 \rfloor$ e assim $x_{i+1} < x_i$. Por outro lado, $x_i \geq 0 \Rightarrow x_{i+1} \geq 0$. Assim, a sequência x_i é inteira, estritamente decrescente e limitada inferiormente por 0. Existe portanto um valor finito n para o qual $x_n = 0$, condição de encerramento do laço.

b) Seja r_i o inteiro representado pelo i-gésimo dígito da cadeia em `r`, da direita para a esquerda e com o índice do primeiro dígito igual a 1 (atenção!). Pela linha 7, a i-gésima iteração adiciona à cadeia exatamente o i-gésimo dígito, mantendo os anteriores inalterados. Seja R_i o valor representado pela cadeia em `r`. Seja n o índice do dígito final da cadeia em `r`, que é também o número da iteração do algoritmo, quando vale $x_n = 0$.

Diz-se que a representação final em `r` está correta se e somente se

$$0 <= r_i <= 9 \tag{8}$$

e

$$R_n = \sum_{i=1}^{n} r_i \cdot 10^{i-1} = x_0 \tag{9}$$

Pelo algoritmo,

$$r_{i+1} = x_i \bmod 10 \tag{10}$$
$$x_{i+1} = \left\lfloor \frac{x_i}{10} \right\rfloor \tag{11}$$

A condição (8) é trivialmente satisfeita por (10). Sabe-se que

$$R_{i+1} = R_i + r_{i+1} \cdot 10^i \tag{12}$$

Por outro lado,

$$\left\lfloor \frac{x_i}{10} \right\rfloor = \frac{x_i - (x_i \bmod 10)}{10} = \frac{x_i - r_{i+1}}{10}$$

Assim,

$$x_{i+1} \cdot 10^{i+1} = \frac{x_i - r_{i+1}}{10} 10^{i+1} = x_i \cdot 10^i - r_{i+1} \cdot 10^i \tag{13}$$

Somando-se (12) e (13), chega-se a:

$$R_{i+1} + x_{i+1} \cdot 10^{i+1} = R_i + x_i \cdot 10^i$$

Ou seja, o valor $R_i + x_i \cdot 10^i$ é *invariante* no algoritmo. No início do algoritmo, $R_0 = 0$. No final, $x_n = 0$. Assim, $R_n = x_0$, o que comprova o funcionamento do algoritmo.

Exercício 11

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 $\mathcal{O}(N)$.

Exercício 12

a) 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:

$$i_{j+1} = i_j - 1 \tag{14}$$

$$p_{j+1} = xp_j + a_{i_j} \tag{15}$$

Segue-se trivialmente de (14) que

$$i_j = n - j - 1 \tag{16}$$

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:

$$p_j = \sum_{k=n}^{n-(j-1)} a_k x^{k-(n-(j-1))} \tag{17}$$

De fato, isso é trivialmente verdadeiro para p_0. Porém, se existe p_j para o qual vale (17), então por (15),

$$\begin{aligned} p_{j+1} &= xp_j + a_{i_j} \\ &= x \sum_{k=n}^{n-(j-1)} a_k x^{k-(n-(j-1))} + a_{i_j} \\ &= \sum_{k=n}^{n-(j-1)} a_k x^{k-(n-j)} + a_{i_j} x^0 \\ &= \sum_{k=n}^{n-j} a_k x^{k-(n-j)} \end{aligned} \tag{18}$$

Ou seja, se (17) vale para algum p_j, então também vale para p_{j+1}, o que por indução finita mostra a validade de (17) para todo j finito. As propriedades (16, 17) 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) Como visto no item anterior, a complexidade do algoritmo é $\mathcal{O}(N)$, onde N é a quantidade de coeficientes do polinômio.

Exercicio 13

a) Note que embora a matriz `a` esteja pré-alocada, *seu conteúdo é indeterminado*.

```
def matrixmult(x, y, a):
    for i in range(len(a)):
        for j in range(len(a[i])):
            total = 0
            for k in range(len(y)):
                total += x[i][k] * y[k][j]
            a[i][j] = total
```

b) Há 3 laços, cada um com N iterações. Assim a complexidade é $\mathcal{O}\left(N^3\right)$.

Exercício 14

Este problema aparentemente demanda uma solução quadrática. No entanto, note que se uma determinada linha possui 0's até a j-ésima coluna, é possivel ignorar todas as linhas seguintes que possuam 1's até a posição j.

```
def encontra_mais_zeros(x):
    n = len(x)  # Matriz quadrada
    # primeira linha
    j = 0
    while j < n and x[0][j] != 1:
        j += 1
    maior_linha = 0
    maior_sequencia = j
    for i in range(len(x)):
        while j < n and x[i][j] != 1:
            j += 1
        if j > maior_sequencia:
            maior_linha = i
            maior_sequencia = j
    return maior_linha
```

Exercício 15

Seja o conjunto A representado pelo vetor $a = \{a_0, a_1, \dots, a_n\}$. Temos 3 possibilidades:

1. Existe um subconjunto cuja soma é exatamente k que contém a_n.
2. Existe um subconjunto cuja soma é exatamente k que *não* contém a_n.
3. *Não* existe subconjunto cuja soma é exatamente a

Ao menos *uma* destas 3 opções deve ser verdadeira. Assim, testando-se os casos 1 e 2 é possível determinar se o subconjunto existe ou não. De fato, se existe um subcojunto de A que contém a_n cuja soma é exatamente k, então existe um subconjunto de $A \setminus \{a_n\} = \{a_0, a_1, \ldots, a_{n-1}\}$ cuja soma é exatamente $k - a_n$. Por outro lado, se há um subconjunto de A que *não* contém a_n cuja soma é exatamente k, então evidentemente há um subconjunto de $\{a_0, a_1, \ldots, a_{n-1}\}$ cuja soma é exatamente k. Deste modo, é possível resolver o problema para um vetor de tamanho n resolvendo dois problemas com vetores de tamanho $n - 1$, testando as duas possibilidades.

O caso base é o do vetor nulo, para o qual a resposta só é verdadeira se $k = 0$.

```
def ExisteSubSoma(a,k):
    def ExisteSubSomaRec(ultimo, k):
        if ultimo < 0: return k==0
        return ExisteSubSomaRec(ultimo-1, k-a[ultimo]) or ExisteSubSomaRec(ultimo-1,k)

    return ExisteSubSomaRec(len(a)-1, k)
```

A lei de recorrência que rege a complexidade deste algoritmo é:

$$T(N) = 2T(N-1) + \mathcal{O}(1)$$

$$T(1) = \mathcal{O}(1)$$

Cuja solução é $T(N) = \mathcal{O}\left(2^N\right)$ (nota-se que esta equação de recorrência, embora trivial, *não* é solucionável pelo *master theorem*).

Nota: é possível obter também os componentes da sequência desejada:

```
def EncontraSubSeq(a,k):
    def EncontraSubSeqRec(ultimo, k):
        if ultimo < 0:
            if k==0: return True, []
            else: return False, None
        Existe, seq = EncontraSubSeqRec(ultimo-1, k-a[ultimo])
        if Existe: return True, seq + [a[ultimo]]
        Existe, seq = EncontraSubSeqRec(ultimo-1, k)
        if Existe: return True, seq
        return False, None

    return EncontraSubSeqRec(len(a)-1, k)
```

Exercício 16

a) 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) 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 $x_0 =$ `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 2. 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:

(a) Existe uma subsequência S qualquer de $A_{0:n-i}$ cuja soma é exatamente x_i:

i. $x_i < a_{n-i}$: 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 $x_i = x_{i+1}$

ii. $x_i \geq a_{n-i}$: 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 $x_i - a_{n-i} = x_{i+1}$

(b) 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 $x_n = 0$.

Exercício 17

A sugestão do enunciado dá o caminho para a solução. Seja $S\left[k, A_{1:n}\right]$ verdadeiro se existe subconjunto de $A_{1:n}$ cuja soma seja exatamente k e falso caso contrário.

Vamos construir a solução por programação dinâmica "incrementando" o conjunto um elemento por vez e a meta para a soma k de um em um.

De fato, note que existe soma de subconjunto de $A_{1:n}$ exatamente igual a k então evidentemente esta soma inclui ou não o elemento a_n. Assim, existe soma de subconjunto $A_{1:n}$ exatamente igual a k se e somente se existe soma de subconjunto $A_{1:n-1}$ igual a k ou a $k - a_n$.

Deste modo,

$$S\left[k, A_{1:n}\right] = S\left[k, A_{1:n-1}\right] \vee S\left[k - a_n, A_{1:n-1}\right]$$

Assim, se são conhecidos todos os valores possíveis de $S\left[i, A_{1:n-1}\right]$ com $0 \leq i \leq k$, então é possível calcular os valores de $S\left[i, A_{1:n}\right]$.

A listagem a seguir implementa esta idéia:

```
def ExisteSubSoma(a,k):
    s = [False]*(k+1)
    s[0] = True
    s_a = list(s)
    for v in a:
        for i in range(k+1):
            s_a[i] = s[i]
            s[i] = s_a[i] or (i>=v and s_a[i-v])
    return existe[-1]
```

Note que é necessário preservar o valor anterior das variáveis `s` até que a atualização esteja completa.

Esta solução tem complexidade $\mathcal{O}(kN)$ onde k é o valor meta da soma e N é a quantidade de elementos do conjunto. Note que esta *não* é uma solução com complexidade verdadeiramente quadrática, pois o valor k não é proporcional ao *tamanho* do problema. Assim, diz-se que este algoritmo opera em tempo *pseudo*-polinomial.

Exercício 18

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
```

Exercício 19

a) Seja u_i o valor da variável `ultimo` ao final da execução da linha 11, na i-gésima iteração do laço externo (iniciado na linha 3). O laço na linha 5 vai executar um total de u_i passos. Seja a_i o valor da variável `ultimo_alterado` ao final da execução da linha 10, na i-gésima iteração do laço interno. O valor máximo de a_i é $(u_i - 1)$, quando o último elemento a ser inspecionado pelo laço interno é modificado (isso ocorre por exemplo com uma entrada em ordem inversa, na qual o laço interno transporta o primeiro elemento até a posição u_i e mantém a ordem relativa dos demais elementos inalterada). Assim, o valor máximo que u_i pode assumir é $(u_{i-1} - 1)$. Ou, seja, no pior caso, u_i segue uma sequência aritmética decrescente de razão -1. O número total de iterações do laço interno no pior caso para uma entrada com n elementos é dado por:

$$\sum_{i=n-1}^{1} i = \frac{n^2 - n}{2} = \mathcal{O}\left(n^2\right)$$

b) Do mesmo modo, o valor *mínimo* que a_i pode assumir na primeira iteração é 0, quando *nenhuma* posição é modificada (como por exemplo com uma entrada ordenada). Neste caso a complexidade é $\mathcal{O}\left(n\right)$.

Exercício 20

A idéia básica é dividir o vetor em 3 regiões, uma no início, uma no centro e uma no final, de modo que a região central é maior que as outras duas. O algoritmo é chamado recursivamente nas duas primeiras regiões concatenadas, depois nas duas finais e finalmente, nas duas primeiras novamente. Mostra-se que após a 2a chamada a 3a região contém os maiores elementos do vetor de forma ordenada. Segue a prova detalhada:

Trata-se de um algoritmo recursivo. Sejam i, j os valores das variáveis `i`, `j` em uma dada chamada da função (como o algoritmo é recursivo, estes valores são específicos de um dado escopo). Seja $k = \lfloor j - i \rfloor /3$. Seja $A = \{d_i, \ldots, d_j\}$ o valor da variável `a` *antes* da chamada a `ordena(a, i, j)` e A^* o valor da mesma variável *após* a chamada.

As pós-condições desejáveis são:

1. $a^*_{l+1} \geq a^*_l \,\forall\, i \leq l < j$, ou seja, o vetor A^* está em ordem crescente.
2. $\exists\, \pi : \mathbb{N}^{[i,j]} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}^{[i,j]} \mid a_l = a^*_{\pi(l)}$, ou seja, existe uma *permutação* (representada como a função bijetora π) de A em A^*.
3. $a^*_l = a_l \,\forall\, i < l \vee l > j$, ou seja, os eventuais elementos de A fora do intervalo de índices i, j permanecem *inalterados*.

Trata-se de um algoritmo recursivo cujo caso-base é $i - j \leq 2$, ou seja o vetor a ser ordenado tem no máximo duas posições. O algoritmo chama a si próprio 3 vezes, com índices $(i, j - k)$, $(i + k, j)$ e $(i, j - k)$ novamente.

Ora, mas é fácil mostrar que fora do caso-base, $k = \lfloor j - i + 1 \rfloor /3 \geq 1$. Isso significa que diferença $i - j$ em cada nível de chamada da função é um valor inteiro, *estritamente* decrescente e limitado inferiormente. Deste modo o caso-base é *sempre* atingido.

No caso-base, o algoritmo troca as eventuais duas posições do vetor caso elas estejam fora de ordem (ou não faz nada se o vetor é unitário). Deste modo, as duas pós-condições são trivialmente satisfeitas.

Ora, mas se as 3 sub-chamadas efetivamente ordenam os respectivos sub-vetores, então, seja $A^{(1)}$ o conteúdo do vetor `a` *após* a 1a chamada. Neste caso vale:

$$\begin{aligned}
& a^{(1)}_{l+1} \geq a^{(1)}_l \,\forall\, i \leq l \leq (j - k) \\
& \exists\, \rho^{(1)} : \mathbb{N}^{[i,j-k]} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}^{[i,j-k]} \mid a_l = a_{\rho^{(1)}(l)} \\
& a^1_l = a_l \,\forall\, (j - k) < l \leq j
\end{aligned}$$

Ou seja, o intervalo entre índices $i, j - k$ da sequência $A^{(1)}$ (denotado $A^{(1)}_{i:j-k}$) está ordenado em ordem crescente e os valores neste intervalo são obtidos de uma permutação dos valores de A *no mesmo intervalo*. Os demais valores permanecem inalterados, ou seja, $A^{(1)}_{j-k+1:j} = A_{j-k+1:j}$ (logo há uma permutação $\pi^{(1)} : \mathbb{N}^{[i,j]} \rightarrowtail \twoheadrightarrow \mathbb{N}^{[i,j]}$ tal que $a_l = a_{\pi^{(1)}(l)}$).

A propriedade de ordenação em particular garante que:

$$a^{(1)}_m \leq a^{(1)}_n \,\forall\, i \leq m < (i+k) \leq n \leq (j-k)$$

Isso significa que se há $a^{(1)}_o < a_m$ com $i \leq m < (i+k)$ e $(k+1) \leq o \leq j$ então necessariamente $o > (j-k)$. Assim, há no máximo k elementos na subsequência $A^{(1)}_{i+k:j}$ que podem ser menores do que algum elemento de $A^{(1)}_{i:i+k-1}$.

Se $A^{(2)}$ é o valor do vetor a depois da segunda chamada, então:

$$\begin{aligned}
&a^{(2)}_{l+1} \geq a^2_l \,\forall\, (i+k) \leq l \leq j \\
&\exists\, \rho^{(2)} : \mathbb{N}^{[i+k,j]} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}^{[i+k,j]} \mid a^{(2)}_l = a^{(1)}_{\rho^{(2)}(l)} \\
&a^{(2)}_l = a^{(1)}_1 \,\forall\, i \leq l < i+k
\end{aligned}$$

Do mesmo modo, a propriedade de ordenação garante que:

$$a^{(2)}_m \leq a^{(2)}_n \,\forall\, (i+k) \leq m \leq (j-k) < n \leq j$$

Mostra-se, por absurdo, que além disso, *todos* os elementos de $A^{(2)}_{j-k+1:j}$ são maiores do que qualquer elemento de $A^{(2)}_{i,i+k-1}$. De fato, sabe-se devido à existência de $\rho^{(2)}$ e às propriedades da primeira chamada que há no máximo k elementos em $A^{(2)}_{i+k,j}$ que podem ser menores do que algum elemento de $A^{(2)}_{i:i+k-1}$. Por outro lado, se ouvesse algum elemento em $A^{(2)}_{j-k+1:j}$ menor do que algum elemento de $A^{(2)}_{i:i+k-1}$, então a propriedade de ordenação exigiria que ouvesse ao menos $j-i-2k$ também menores. Ora, mas:

$$j-i-2k = j-i-2\left\lfloor \frac{j-i}{3} \right\rfloor > j-i-2\frac{j-i}{3} = \frac{j-i}{3} > \left\lfloor \frac{j-i}{3} \right\rfloor = k$$

Isso é uma contradição de modo que não pode existir tal elemento. Assim, combinando esta propriedade com a de ordenação mostra-se que todos os elementos em $A^{(2)}_{j-k+1:j}$ são maiores do que todos os elementos em $A^{(2)}_{i:j-k}$.

A terceira chamada completa a ordenação, pois após esta,

$$\begin{aligned}
&a^{(3)}_{l+1} \geq a^{(3)}_l \,\forall\, (i) \leq l \leq (j-k) \\
&\exists\, \rho^{(3)} : \mathbb{N}^{[i,j-k]} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}^{[i,j-k]} \mid a^{(3)}_l = a^{(2)}_{\rho^{(3)}(l)} \\
&a^{(3)}_l = a^{(2)}_1 \,\forall\, (j-k) < l \leq j
\end{aligned}$$

Deste modo, os elementos em $A^{(3)}_{i:j-k}$ estão ordenados. Ora, mas também o estão os elementos de $A^{(3)}_{j-k+1,j}$ (que não foram modificados). Além disso, todos os elementos em $A^{(3)}_{i:j-k}$ são menores ou iguais aos elementos em $A^{(3)}_{j-k+1,j}$. Assim, o vetor $A^{(3)}$ está completamente ordenado. É trivial mostrar que ele é uma permutação do vetor original. Como $A^{(3)} = A^*$, as pós-condições originais são atendidas.

Quanto à complexidade, ele faz operações em tempo constante e em seguida chama a si próprio 3 vezes com entradas com $2/3$ do tamanho original.

Na notação do Teorema-Mestre,

$$T(N) = 3T\left(\frac{N}{3/2}\right) + \mathcal{O}\left(N^0\right)$$

Deste modo,

$$T(N) = \mathcal{O}\left(N^{\log_{3/2} 2}\right) \approx \mathcal{O}\left(N^{1{,}71}\right)$$

Exercício 21

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 22

a) Uma solução trivial com complexidade $\mathcal{O}(N)$:

```
def fatorh(citacoes):
    i=0
    while i<len(citacoes) and citacoes[i]>=(i+1): i += 1
    return i
```

A versão com complexidade $\mathcal{O}(\log N)$ usa as mesmas idéias básicas de uma busca binária:

```
def fatorh(citacoes):
    a = 0
    b = len(citacoes)
    best = 0
    while a<=b:
        ref = (a+b)//2
        if citacoes[ref]>=(ref+1): # Procura na metade superior
            best = ref
            a = ref + 1
        else: # Procura na metade inferior
            b = ref - 1
    return best+1
```

É possível ainda omitir completamente a variável `best`:

```
def fatorh(citacoes):
    a = 0
    b = len(citacoes)
    while(a<=b):
        ref = (a+b)//2
        if citacoes[ref]>=(ref+1): # Procura na metade superior
            a = ref+1
        else : # Procura na metade inferior
            b = ref - 1
    return a
```

Finalmente há uma versão recursiva bastante compacta:

```python
def fatorh(citacoes):
    def busca(a, b):
        if a>b: return a
        ref = (a+b)//2
        if citacoes[ref]>=(ref+1): return busca(ref+1, b)
        return busca(a, ref-1)
    return busca(0, len(citacoes)-1)
```

b) O algoritmo de ordenação *mergesort* ordena um vetor com complexidade $\mathcal{O}(N \log N)$. Assim, a complexidade das operações conjuntas ordenar por *mergesort* e calcular o fator h é de $\mathcal{O}(N \log N + N) = \mathcal{O}(N \log N)$.

Exercício 23

a) O algoritmo é um algoritmo recursivo cujo caso base é o parâmetro `n` igual a 1. Caso contrário, o algoritmo chama a si mesmo duas vezes, passando `n/2` como novo parâmetro `n`. Se o valor inicial de `n`, o tamanho inicial do vetor, for exatamente uma potência de 2, as sucessivas divisões por 2 farão com que o caso base seja atendido.

b) Seja $T(N)$ a ordem de complexidade do algoritmo. O algoritmo chama a si mesmo duas vezes com $N/2$ como parâmetro e realiza $N \cdot \mathcal{O}(1)$ operações sobre os vetores. A equação é:

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

c) Na notação do *Master Theorem*, temos $A = 2$, $B = 2$ e $L = 1$. Neste caso, tem-se exatamente $A = B^L$ donde a solução é:

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

Exercício 24

a) Resposta: O algoritmo divide a matriz em 4 quadrantes, cada um com aproximadamente $1/4$ do tamanho da matriz original. Em seguida, chama a si mesmo em 3 destes quadrantes. A operação de divisão e escolha de quadrantes a rejeitar tem complexidade constante O(1). A equação recursiva é:

$$T(N) = 3T\left(\frac{N}{4}\right) + \mathcal{O}(1)$$

b) Na notação do teorema Mestre, tem-se $A = 3$, $B = 4$ e $L = 0$. Neste caso, naturalmente, $A > B^L$ e a solução é $N^{\log_B A}$, ou seja,

$$T(N) = \mathcal{O}\left(N^{\log_4 3}\right) \approx \mathcal{O}\left(N^{0{,}7925}\right)$$

Exercício 25

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 8, o número de inversões I é dado por

$$I = \sum_{i=0}^{n-1} \xi_i \tag{19}$$

onde é o total de elementos a_j com $j > i$ tais que $a_i < a_j$. Tomando-se a função $H(n) = 1$ para $n \geq 0$ e $H(n) = 0$ para $n < 0$ (esta é a Função de Heviside), então pode-se escrever

$$\xi_i = \sum_{j=i+1}^{n-1} 1 - H(a_j - a_i) \tag{20}$$

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

$$\xi_i = \overbrace{\sum_{j=i+1}^{m-1} 1 - H(a_j - a_i)}^{\epsilon_{i,m}} + \overbrace{\sum_{j=m}^{n-1} 1 - H(a_j - a_i)}^{\delta_{i,m}} \tag{21}$$

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_i é a soma da quantidade de elementos maiores encontrados na primeira partição $\epsilon_{i,m}$ e a quantidade de elementos na segunda partição $\delta_{i,m}$. Observa-se ademais que quando $i > m$ então $\epsilon_{i,m} = 0$.

Substituindo-se em (19), temos:

$$I = \sum_{i=0}^{n-1} \epsilon_{i,m} + \sum_{i=0}^{n-1} \delta_{i,m} = \overbrace{\sum_{i=0}^{m-1} \epsilon_{i,m}}^{I_e} + \overbrace{\sum_{i=m}^{n-1} \delta_{i,m}}^{I_d} + \overbrace{\sum_{i=m}^{n-1} \epsilon_{i,m}}^{I_m} \tag{22}$$

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.

Seja $I_{i,j}$ o valor da variável `inv` quando `i` $= i$, `j` $= j$ ao final da execução da linha 25. Então vale:

$$I_{i,j} = \sum_{k=0}^{i-1} \epsilon_{i,m} + \sum_{k=m}^{j} 1 - H(a_j - a_i) \tag{23}$$

Exercício 26

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 ($\mathcal{O}(N)$).

Exercício 27

A equação a ser resolvida é

$$T\left(N\right) = aT\left(\frac{N}{b}\right) + N\log N$$

Esta é uma equação recursiva, na qual o termo de uma ordem ($T(N)$) está expresso em função de N e do termo de ordem imediatamente anterior ($T(N/b)$). Uma maneira de resolvê-la é escrever o termo de ordem imediatamente anterior ($T(n/b)$) em função de seu antecessor ($T(N/b^2)$), usando para isso a própria equação recursiva[1]:

$$T\left(\frac{N}{b}\right) = aT\left(\frac{N}{b^2}\right) + \frac{N}{b}\log\left(\frac{N}{b}\right)$$

Substituindo-se na equação anterior,

$$T\left(N\right) = a^2T\left(\frac{N}{b^2}\right) + \frac{a}{b}N\log\left(\frac{N}{b}\right) + N\log N$$

Prosseguindo agora, escrevemos $T(N/b^2)$ em função de $T(N/b^3)$,

$$T\left(\frac{N}{b^2}\right) = aT\left(\frac{N}{b^3}\right) + \frac{N}{b^2}\log\left(\frac{N}{b^2}\right)$$

e

$$T\left(N\right) = a^3T\left(\frac{N}{b^3}\right) + \frac{a^2}{b^2}n\log\left(\frac{N}{b^2}\right) + \frac{a}{b}N\log\left(\frac{N}{b}\right) + N\log N$$

Repetindo-se este processo k passos, tem-se:

$$T\left(N\right) = a^kT\left(\frac{N}{b^k}\right) + \sum_{i=0}^{k-1}\frac{a^i}{b^i}N\log\left(\frac{N}{b^i}\right)$$

Se $N = b^k$,

$$\begin{aligned}
T\left(N\right) &= a^k\overbrace{T(1)}^{\mathcal{O}(1)} + \sum_{i=0}^{k-1}\left(\frac{a}{b}\right)^i b^k\log\left(\frac{b^k}{b^i}\right)\\
&= a^k\mathcal{O}(1) + \sum_{i=0}^{k-1} b^k\left(\frac{a}{b}\right)^i\left(\log b^k - \log b^i\right)\\
&= a^k\mathcal{O}(1) + b^k\log b\left(k\sum_{i=0}^{k-1}\left(\frac{a}{b}\right)^i - \sum_{i=0}^{k-1} i\left(\frac{a}{b}\right)^i\right)\\
&= a^k\mathcal{O}(1) + b^k\log\left(b^k\right)\left(\sum_{i=0}^{k-1}\left(\frac{a}{b}\right)^i - \frac{1}{k}\sum_{i=0}^{k-1} i\left(\frac{a}{b}\right)^i\right)
\end{aligned}$$

Existem formas fechadas para as duas séries do lado direito, que dependem da razão (a/b). Se $a = b$, então $a/b = 1$ e trivialmente,

$$T\left(N\right) = a^k\mathcal{O}(1) + b^k\log\left(b^k\right)k$$

Como $N = b^k$ e $k = \log_b N$,

$$T\left(N\right) = \mathcal{O}\left(N\left(\log N\right)^2\right)$$

Para $a \neq b$ usa-se resultados conhecidos de somas de séries. Com $x \neq 1$, vale[2]:

$$\sum_{i=0}^{k-1} x^i = \frac{x^k - 1}{x - 1}$$

[1]Esta é a maneira pela qual o *Master Theorem* é demonstrado no curso

[2]Note que o segundo resultado é a derivada do primeiro multiplicada por x

$$\sum_{i=0}^{k-1} i x^i = \frac{x}{(x-1)^2} + x^k \left(\frac{k}{x-1} - \frac{x}{(x-1)^2} \right)$$

Então, substituindo-se $x = a/b$ e descartando-se os termos de menor ordem em k,

$$\sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i - \frac{1}{k} \sum_{i=0}^{k-1} i \left(\frac{a}{b}\right)^i = \mathcal{O}\left(\frac{{(a/b)^{k+1}}/{k} - 1}{(a/b) - 1)} \right)$$

Retomando-se $N = b^k$ e $k = \log_b N$,

$$T(N) = N^{\log_b a} \mathcal{O}(1) + N\mathcal{O}\left(\frac{(a/b)^{\log_b N + 1} - \log_b N}{(a/b) - 1} \right)$$

Estamos então diante de 2 possibilidades:

1. $a/b > 1$. Neste caso, $(a/b)^{\log_b N} \gg \log_b N$ e
$$T(N) = \mathcal{O}\left(N^{\log_b a}\right)$$

2. $a/b < 1$. Neste caso, $(a/b)^{\log_b N} \ll \log_b N$ e
$$T(N) = \mathcal{O}\left(N \log N\right)$$

Resumindo, tem-se:

$$T(N) = \begin{cases} \mathcal{O}\left(N \log N\right) & \text{para } a < b \\ \mathcal{O}\left(N \left(\log N\right)^2\right) & \text{para } a = b \\ \mathcal{O}\left(N^{\log_b a}\right) & \text{para } a > b \end{cases}$$

Em particular, para:

(a) $a = 3, b = 3$:
$$T(N) = \mathcal{O}\left(N^{\log_b a}\right) \approx N^{1{,}585}$$

(b) $a = 2, b = 2$:
$$T(N) = \mathcal{O}\left(N \left(\log N\right)^2\right)$$

(c) $a = 2, b = 2$:
$$T(N) = \mathcal{O}\left(N \log N\right)$$

Exercício 28

a) A quantidade de chamadas subsequente é limitada pelo tamanho do vetor de entrada. Assim, naturalmente, a maior quantidade de chamadas ocorrerá quando um dos sub-vetores pós-divisão for o maior possível. Isso ocorre por exemplo em um vetor pré-ordenado, quando o primeiro sub-vetor tem tamanho $N - 1$. Neste caso a profundidade máxima é $\mathcal{O}(N)$.

b) O resultado é idêntico à listagem do enunciado até a linha 17:

```
1 def quicksort(s, a, b):
2     if a>=b:
3         return
4     p = s[b]      # pivot
5     l = a
6     r = b-1
7     while l<=r :
```

```
        while l<=r and s[l]<=p:
            l = l+1
        while l<=r and s[r]>=p:
            r = r - 1
        if l<r:
            temp = s[l]
            s[l] = s[r]
            s[r]=temp
    s[b] = s[l]
    s[l] = p
    if (l-1-a)<(b-l-1):
        quicksort(s, a, (l-1))
        quicksort(s, (l+1), b)
    else:
        quicksort(s, (l+1), b)
        quicksort(s, a, (l-1))
```

c) Note que é impossível retirar *ambas* as chamadas recursivas (ao menos sem uma estrutura de dados auxiliar (que por sua vez utilizaria uma quantidade de memória comparável à da pilha do programa).

```
def quicksort(s, a, b):
    while a < b:
        p = s[b]      # pivot
        l = a
        r = b-1
        while l<=r :
            while l<=r and s[l]<=p:
                l = l+1
            while l<=r and s[r]>=p:
                r = r - 1
            if l<r:
                temp = s[l]
                s[l] = s[r]
                s[r]=temp
        s[b] = s[l]
        s[l] = p
        if (l-1-a)<(b-l-1):
            quicksort(s, a, (l-1))
            a = l + 1
        else:
            quicksort(s, (l+1), b)
            b = l -1
```

d) A profundidade máxima ocorre quando o sub-vetor passado para a chamada recursiva tem o tamanho máximo a cada iteração. Ora, este sub-vetor tem no máximo o tamanho da metade do vetor original (visto que ele é menor ou igual ao outro sub-vetor). Finalmente, é possível dividir um vetor em 2 no máximo $\log_2 N$ vezes, de modo que a profundidade máxima é de ordem $\mathcal{O}(\log N)$.

Exercício 29

a) 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) 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:

(a) A sequência recebida é nula, situação em que a função retorna `False`.

(b) 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 $\lfloor \mathtt{len}(\mathtt{v})/2 \rfloor$. 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) 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:

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

Cuja solução é:

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

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:

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

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