Fundamentos da Programação

TÉCNICO LISBOA

Solução do Exame

1 de Fevereiro de 2019

09:00–11:00

1. Usando palavras suas e, no máximo, em cinco linhas responda às seguintes questões. Respostas dadas através de exemplos serão classificadas com zero valores.

   (a) **(0.5)** Diga o que é um algoritmo e quais são as suas características (sem descrever as características).
   Resposta:
   Um algoritmo é uma sequência finita de instruções bem definidas e não ambíguas cada uma das quais pode ser executada mecanicamente num período de tempo finito e com uma quantidade de esforço finito. Um algoritmo é rigoroso, eficaz e deve terminar.

   (b) **(0.5)** Qual a relação entre um programa e um algoritmo?
   Resposta:
   Um programa corresponde a um algoritmo escrito numa linguagem de programação.

   (c) **(0.5)** Qual a relação entre um processo e um programa.
   Resposta:
   Um processo corresponde ao conjunto de ações tomadas por um computador durante a execução de um programa.

2. **(1.0)** Escreva a função `apenas_pares` que recebe um número inteiro positivo e devolve o número composto apenas pelos seus algarismos pares. Não pode recorrer a cadeias de caracteres nem a listas. A sua função deve verificar a correção do argumento. Por exemplo,

```
>>> apenas_pares(543765092)
4602
>>> apenas_pares(5.3)
ValueError: O argumento deve ser um inteiro > 0
```

   Resposta:

```
def apenas_pares(n):
    if isinstance(n, int) and n > 0:
        res = 0
        mult = 1
        while n > 0:
```

```
            dig = n % 10
            n = n // 10
            if dig % 2 == 0:
                res = res + mult * dig
                mult = mult * 10
        return res
    else:
        raise ValueError('O argumento deve ser um inteiro > 0')
```

3. Um número *abundante* é um número natural para o qual a soma dos seus divisores próprios é maior que o próprio número. Um divisor próprio de um número inteiro n é um divisor de n que seja diferente de n. O inteiro 12 é o primeiro número abundante. Seus divisores próprios são 1, 2, 3, 4 e 6 cuja soma é 16.

   (a) **(1.0)** Escreva o predicado `eh_abundante` que recebe como argumento um inteiro positivo e devolve verdadeiro apenas se o seu argumento é um número abundante. Não é necessário validar o argumento. Por exemplo,

   ```
   >>> eh_abundante(6)
   False
   >>> eh_abundante(12)
   True
   ```

   Resposta:

   ```
   def eh_abundante(n):
       soma = 0
       for i in range(1, n):
           if n % i == 0:
               soma = soma + i
       return soma > n
   ```

   (b) **(1.0)** Usando o predicado `eh_abundante` da alínea anterior, escreva a função `n_primeiros_abundantes` que recebe como argumento um inteiro positivo, `n` e devolve a lista dos `n` primeiros números abundantes. Não é necessário validar o argumento. Por exemplo,

   ```
   >>> n_primeiros_abundantes(5)
   [12, 18, 20, 24, 30]
   ```

   Resposta:

   ```
   def n_primeiros_abundantes(n):
       res = []
       i = 1
       while n != 0:
           if eh_abundante(i):
               res = res + [i]
               n = n -1
           i = i + 1
       return res
   ```

4. A *cifra de César* (com este nome porque foi usada por Júlio César) contém 26 cifras de substituição, uma para cada letra do alfabeto. A Figura 1 apresenta uma destas cifras. Para cifrar ou decifrar uma mensagem, apenas temos que saber qual a letra, no círculo interior, que corresponde ao `A` no círculo exterior. No exemplo da

Figure 1: Cifra de César.

Figura 1, esta letra é o `T`. A partir daí, cada letra é substituída pela letra que se encontra à distância igual à distância entre o `A` e o `T`. O algoritmo apenas considera as letras maiúsculas que aparecem na mensagem original. A mensagem cifrada é em maiúsculas, sem espaços nem pontuação, sendo-lhe **adicionada**, na última posição, a letra que define a cifra usada. No nosso exemplo, seria adicionada a letra `T` ao final da mensagem cifrada. Para tornar a mensagem ainda mais difícil de decifrar, após cada letra cifrada é adicionada uma letra maiúscula gerada aleatoriamente. Por exemplo,

```
>>> cifra_cesar('UM TESTE', 'T')
'NEFMMLXWLFMIXBT'
```

(a) **(1.5)** Escreva a função `cifra_cesar` que recebe uma cadeia de caracteres e a letra correspondente ao `A` e devolve a mensagem cifrada. A sua função não tem que validar o argumento.
Resposta:

```
def cifra_cesar(frase, cifra):
    from random import random
    desloc = ord(cifra) - ord('A')
    res = ''
    for i in range (len(frase)):
        if 'A' <= frase[i] <= 'Z':
            res = res + \
                chr(ord('A') + \
                    (ord(frase[i]) - ord('A') + desloc) % 26) + \
                chr(ord('A') + int(random() * 26) + 1)
    return res + cifra
```

(b) **(1.5)** Escreva a função `decifra_cesar` que recebe uma mensagem cifrada e devolve a mensagem original. A sua função não tem que validar o argumento. Por exemplo,

```
>>> decifra_cesar('NEFMMLXWLFMIXBT')
'UMTESTE'
```

Resposta:

```
def decifra_cesar(frase):
    cifra = frase[-1]
    dist = ord(cifra) - ord('A')
    res = ''
    for i in range (0, len(frase)-1, 2):
        res = res + \
            chr(ord('A') + \
                (ord(frase[i]) - ord('A') - dist) % 26)
    return res
```

5. **(1.5)** Escreva a função `calc_valor` que calcula o valor aproximado da série para um determinado valor de x:

$$\sum_{n=0}^{\infty} \frac{x^n}{n!}$$

O cálculo do valor aproximado da série deverá terminar quando o termo a adicionar for inferior a um certo δ que deverá ser escolhido por si. A sua função não pode utilizar as funções potência nem fatorial. Não é necessário validar os argumentos. SUGESTÃO: Veja como cada termo é calculado a partir do termo anterior. Por exemplo,

```
>>> calc_valor(3.14)
23.10383988830576
```

Resposta:

```
def calc_valor(x):
    res = 0
    termo = 1
    n = 0
    while termo > 0.0001:
        res = res + termo
        n = n + 1
        termo = termo * (x / n)
    return res
```

6. Considere a função, definida para inteiros não negativos, do seguinte modo:

$$f(n) = \begin{cases} 1 & \text{se } n = 0 \\ 2 \cdot f(n-1) & \text{se } n \text{ é par} \\ 3 \cdot f(n-1) & \text{se } n \text{ é impar} \end{cases}$$

Escreva esta função (sem verificar a validade do seu argumento):

(a) **(1.0)** Usando iteração linear.
Resposta:

```
def fi(n):
    res = 1
    for i in range(1, n+1):
        if i % 2 == 0:
            res = res * 2
        else:
            res = res * 3
    return res
```

(b) **(1.0)** Usando recursão com operações adiadas.
Resposta:

```
def fr(n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return 2 * fr(n-1)
    else:
        return 3 * fr(n-1)
```

(c) **(1.0)** Usando recursão de cauda.
Resposta:

```
def fc(n):

    def fc_aux(n, res):
        if n == 0:
            return res
        elif n % 2 == 0:
            return fc_aux(n-1, 2 * res)
        else:
            return fc_aux(n-1, 3 * res)

    return fc_aux(n, 1)
```

7. **(1.0)** Usando um ou mais dos funcionais sobre listas (`filtra`, `transforma`, `acumula`, ou as funções embutidas `filter`, `map`, `reduce`), escreva a função `soma_quadrados` que recebe um inteiro positivo n e devolve a soma

$$1^2 + 2^2 + 3^2 + \ldots + n^2$$

Não é necessário verificar o argumento. O corpo da sua função apenas pode ter uma instrução, a instrução `return`.

Resposta:

```
def soma_quadrados(n):
    return acumula(transforma(list(range(1,n+1)), \
                              lambda x : x * x), \
                   lambda x, y : x + y)
```

8. Considere a seguinte gramática em notação BNF, na qual o símbolo inicial é ⟨A⟩:

⟨A⟩ ::= a ⟨B⟩ b | a ⟨A⟩ b

⟨B⟩ ::= c ⟨B⟩ d | c ⟨C⟩ d

⟨C⟩ ::= e

(a) **(0.5)** Indique os símbolos terminais e os símbolos não terminais da gramática.
Símbolos terminais:
Resposta:
a, b, c, d, e
Símbolos não terminais:
Resposta:
⟨A⟩, ⟨B⟩, ⟨C⟩

(b) **(0.5)** Indique, justificando no caso de não pertencer, quais das seguintes expressões pertencem ou não pertencem ao conjunto de frases da linguagem definida pela gramática:

i. `aacedbb`
Resposta:
Pertence.

ii. `aacdbb`
Resposta:
Não pertence pois falta um `e`.

iii. `abc`
Resposta:
Não pertence pois `a` e `b` não podem aparecer seguidos.

iv. `ABC`
Resposta:
Não pertence pois `ABC` não são símbolos terminais.

v. `acedb`
Resposta:
Pertence.

(c) **(1.5)** Escreva a função `reconhece`, que recebe como argumento uma cadeia de caracteres e devolve verdadeiro se o seu argumento corresponde a uma frase da linguagem definida pela gramática e falso em caso contrário. Não é necessário validar os argumentos.
Resposta:

```
def reconhece(frase):
    i = 0
    f = len(frase) - 1
    if frase[0] != 'a' or frase[-1] != 'b':
        return False
    while frase[i] == 'a' and frase[f] == 'b' and i < f:
        i = i + 1
        f = f - 1
    if frase[i] != 'c' or frase[f] != 'd' or i >= f:
        return False
    while frase[i] == 'c' and frase[f] == 'd' and i < f:
        i = i + 1
        f = f - 1
    return frase[i] == 'e' and i == f
```

9. **(1.5)** Escreva a função `junta` que recebe dois dicionários, cujos valores associados às chaves correspondem a listas, e devolve o dicionário que contém todas as chaves contidas em pelo menos um dos dicionários e o valor associado a cada chave corresponde à lista obtida pela "união" (no sentido de conjuntos) das listas correspondendo às chaves existentes nos dicionários. Não é necessário validar os argumentos. Por exemplo,

```
>>> d1 = {'g': [12, 1], 'z': [5], 'a':[98, 32]}
>>> d2 = {'f':[3], 'g':[33, 44]}
>>> junta(d1, d2)
{'a': [98, 32], 'f': [3], 'g': [12, 1, 33, 44], 'z': [5]}
```

Resposta:

```
def junta(d1, d2):
    res = {}
    # res = d1, sem destruir d1
    for c in d1:
        res[c] = d1[c]
    for c in d2:
        if c in res:
            lst = res[c]
            for el in d2[c]:
                if el not in lst:
                    lst = lst + [el]
            res[c] = lst
        else:
            res[c] = d2[c]
    return res
```

10. Uma *fila de prioridades* é uma estrutura de dados composta por um certo número de filas, cada uma das quais associada a uma determinada prioridade.

    Considere uma fila de prioridades com duas prioridades, `urgente` e `normal`. Nesta fila, novos elementos são adicionados, indicando a sua prioridade, e são colocados no fim da fila respetiva. Os elementos são removidos da fila através da remoção do elemento mais antigo da fila `urgente`. Se a fila `urgente` não tiver elementos, a operação de remoção remove o elemento mais antigo da fila `normal`. Existe uma operação para aumentar a prioridade, a qual remove o elemento mais antigo da fila `normal` e coloca-o como último elemento da `fila urgente`. As operações básicas para o tipo fila de prioridades (com prioridades `urgente` e `normal`) são as seguintes:

    - *Construtores:*
      - $nova\_fila\_2p : \{\} \mapsto fila\_2p$
        $nova\_fila\_2p()$ tem como valor uma fila de duas prioridades sem elementos.
    - *Seletores:*
      - *inicio* : $fila\_2p \mapsto elemento$
        inicio$(fila)$ tem como valor o elemento que se encontra no início da fila de prioridade $urgente$ da *fila*; se a fila de prioridade $urgente$ da *fila* não tiver elementos, tem como valor o elemento que se encontra no início da fila de prioridade $normal$ da *fila*. Se as filas de prioridade $urgente$ e $normal$ não tiverem elementos, o valor desta operação é indefinido.
      - $comprimento\_2p : fila\_2p \times \{urgente, normal\} \mapsto \mathbb{N}_0$
        $comprimento\_2p(fila, tipo)$ tem como valor o número de elementos da fila de prioridade *tipo* da *fila*.
    - *Modificadores:*
      - $coloca\_2p : fila\_2p \times \{urgente, normal\} \times elemento \mapsto fila\_2p$
        $coloca\_2p(fila, tipo, elm)$ altera de forma permanente a *fila* para a fila que resulta em inserir $elem$ no fim da fila de prioridade *tipo* da $fila$. Devolve a fila resultante.
      - $retira\_2p : fila\_2p \mapsto fila\_2p$
        $retira\_2p(fila)$ altera de forma permanente a *fila* para: (1) a fila que resulta em remover o elemento que se encontra no início da fila de prioridade

$urgente$ da *fila*; (2) a fila que resulta em remover o elemento que se encontra no início da fila de prioridade $normal$ da *fila*, se a fila de prioridade $urgente$ da *fila* não tiver elementos. Se as filas de prioridade $urgente$ e $normal$ não tiverem elementos, o valor desta operação é indefinido. Devolve a fila resultante.

- $aumenta\_prioridade\_2p$: *fila_2p* $\mapsto$ *fila_2p*
  $aumenta\_prioridade\_2p(fila)$ altera de forma permanente a *fila* para a fila que resulta em remover o elemento que se encontra no início da fila de prioridade $normal$ da *fila* e coloca-o no final da fila de prioridade $urgente$ da *fila*. Se a fila de prioridade $normal$ não tiver elementos, esta operação não altera a *fila*. Devolve a fila resultante.

- *Reconhecedores:*
  - $e\_fila\_2p : Universal \mapsto logico$
    $e\_fila\_2p(arg)$ tem o valor *verdadeiro*, se *arg* é uma fila de prioridades com as prioridades *urgente* e *normal*, e tem o valor *falso*, em caso contrário.
  - *fila_2p_vazia* : *fila* $\times \{urgente, normal\} \mapsto logico$
    fila_2p_vazia(*fila*, *tipo*) tem o valor *verdadeiro*, se a fila de prioridade *tipo* da *fila* é a fila vazia, e tem o valor *falso*, em caso contrário.

- *Testes:*
  - *filas_2p_iguais* : *fila_2p* $\times$ *fila_2p* $\mapsto$ *logico*
    filas_2p_iguais($fila_1$, $fila_2$) tem o valor *verdadeiro*, se $fila_1$ é igual a $fila_2$, e tem o valor *falso*, em caso contrário.

(a) **(2.0)** Defina a classe `fila_2_p` com prioridades `urgente` e `normal`, escolhendo uma representação externa para as instâncias da classe.
Resposta:

```
class fila_2_p:

    def __init__(self):
        self.urgente = []
        self.normal = []

    def inicio(self):
        if len(self.urgente) != 0:
            return self.urgente[0]
        elif len(self.normal) != 0:
            return self.normal[0]
        else:
            raise ValueError('inicio: filas vazias')

    def comprimento(self, tipo):
        if tipo == 'urgente':
            return len(self.urgente)
        elif tipo == 'normal':
            return len(self.normal)
        else:
            raise ValueError('comprimento: tipo desconhecido')

    def coloca(self, tipo, el):
        if tipo == 'urgente':
            self.urgente = self.urgente + [el]
```

```
            return self
        elif tipo == 'normal':
            self.normal = self.normal + [el]
            return self
        else:
            raise ValueError('coloca: tipo desconhecido')

    def retira(self):
        if len(self.urgente) != 0:
            del(self.urgente[0])
            return self
        elif len(self.normal) != 0:
            del(self.normal[0])
            return self

        else:
            raise ValueError('inicio: filas vazias')

    def aumenta_prioridade(self):
        if len(self.normal) != 0:
            primeiro = self.normal[0]
            del(self.normal[0])
            self.urgente = self.urgente + [primeiro]
        return self

    def fila_vazia(self, tipo):
        if tipo == 'urgente':
            return self.urgente == []
        elif tipo == 'normal':
            return self.normal == []
        else:
            raise ValueError('fila_vazia: tipo desconhecido')

    def fila_para_listas(self):
        return (self.urgente, self.normal)

    def filas_iguais(self, outro):
        fp = outro.fila_para_listas()
        return self.urgente == fp[0] and self.normal == fp[1]

    def __repr__(self):
        def rep_fila(f):
            r = '< '
            for e in f:
                r = r + str(e) + ' '
            r = r + '<'
            return r
        return 'urgente: ' + rep_fila(self.urgente) + \
               '; normal: ' + rep_fila(self.normal)
```

(b) **(1.0)** Escreva comandos, bem como os valores devolvidos pelo Python, para:
Criar uma instância da classe `fila_2_p`, e guardá-la numa variável
Inserir nessa instância o valor 5 na fila `urgente`
Inserir nessa instância o valor 3 na fila `urgente`
Inserir nessa instância o valor 7 na fila `normal`
Inserir nessa instância o valor 9 na fila `normal`

Inserir nessa instância o valor 11 na fila `normal`
Retira dessa instância um elemento
Aumentar a prioridade
Resposta:

```
>>> f = fila_2_p()
>>> f.coloca('urgente', 5)
urgente: < 5 <; normal: < <
>>> f.coloca('urgente', 3)
urgente: < 5 3 <; normal: < <
>>> f.coloca('normal', 7)
urgente: < 5 3 <; normal: < 7 <
>>> f.coloca('normal', 9)
urgente: < 5 3 <; normal: < 7 9 <
>>> f.coloca('normal', 11)
urgente: < 5 3 <; normal: < 7 9 11 <
>>> f.retira()
urgente: < 3 <; normal: < 7 9 11 <
>>> f.aumenta_prioridade()
urgente: < 3 7 <; normal: < 9 11 <
```