MAC 122 - PDA
2. Semestre de 2017 - prof. Marcilio – IME USP BMAC
Segunda Prova – 30 de Novembro de 2017

Questão	1	2	3	4	5	6	Total
Valor	1	1	2	2	2	2	10,0
Nota							

Nome:_____ NUSP:_____

Questão 1 (1 ponto):

Considere uma lista V com 7 elementos. Cada elemento contém um algarismo de seu NUSP. Se seu NUSP tiver 8 dígitos considere apenas os 7 primeiros.

Exemplo: Se o seu NUSP é 8473629 então:

i	0	1	2	3	4	5	6
V[i]	**8**	**4**	**7**	**3**	**6**	**2**	**9**

Diga qual o número de trocas entre os elementos para que esta lista seja classificada em ordem crescente pelos seguintes métodos:

Exemplos para o NUSP : 8473629

a) Método da Bolha (Bubble) – sobe cada elemento até encontrar o seu lugar.

12

b) Método Quick – particiona a lista e aplica o Quick nas 2 partes.

4

c) Usando o método da Bolha, dê uma permutação do seu NUSP que precise de 5 trocas (se houver alguma).

2 7 4 3 6 9 8

d) Para esta mesma permutação do item anterior, quantas trocas serão necessária pelo método Quick?

3

Questão 2 (1 ponto):

O Algoritmo de Boyer-Moore (versão 1) para procurar uma palavra **A** em um texto **B**, verifica qual o próximo elemento de **B** para decidir qual o deslocamento. O algoritmo conta quantas vezes **A** aparece em **B**.
O número de **tentativas** é a quantidade de vezes que **A** é comparada com um **novo trecho** de **B**.

Considere agora o texto **B** como o seu **NUSP** formado por 7 ou 8 dígitos, no qual vamos procurar a palavra **A = "121"**. Mostre na tabela abaixo quais os deslocamentos ou o número de tentativas em que **A** é comparado com **novo trecho** de **B**.

Exemplos para o NUSP : 8473629

seu NUSP	8	4	7	3	6	2	9	
Tentativa 1	1	2	1					
Tentativa 2					1	2	1	
Tentativa 3								
Tentativa 4								
Tentativa 5								
Tentativa 6								

Idem para o Algoritmo de Boyer-Moore (versão 2) para procurar uma palavra **A** em um texto **B**, que verifica a ocorrência de sub-cadeias em **A** iguais a que houve coincidência em **B**.

seu NUSP	8	4	7	3	6	2	9	
Tentativa 1	1	2	1					
Tentativa 2		1	2	1				
Tentativa 3			1	2	1			
Tentativa 4				1	2	1		
Tentativa 5					1	2	1	
Tentativa 6								

Questão 3 (2 pontos):

Mostre o conteúdo de uma tabela hash com as palavras (strings) abaixo. As palavras são inseridas na ordem apresentada:
'amarelo', 'azul', 'branco', 'marron', 'preto', 'verde', 'vermelho'.

A tabela deve ter 11 elementos e a função de hash é a seguinte:

```
def hash(x):
     # dobro do tamanho da string x módulo 11
     return len(x) * 2 % 11
```

```
Valores da função de hash:
'amarelo', 'azul', 'branco', 'marron', 'preto', 'verde', 'vermelho'.
   3         8        1          1        10        10        5
```

1) Tabela de hash simples

índice	conteúdo
0	verde
1	branco
2	marron
3	amarelo
4	
5	vermelho
6	
7	
8	azul
9	
10	preto

2) Tabela de hash duplo com passo 3.

índice	conteúdo
0	
1	branco
2	verde
3	amarelo
4	marron
5	vermelho
6	
7	
8	azul
9	
10	preto

3) Quantas comparações com elementos da tabela serão necessárias para procurar a string **'verde'** nos dois casos acima?

Em ambos os caso encontra o verde na segunda comparação

4) Quantas comparações com elementos da tabela serão necessárias para concluir que a string `'laranja'` não está na tabela?

Em ambos os casos também na segunda comparação, considerando que a segunda comparação é com a casa vazia

Questão 4 (2 pontos):

Considere a classe **`ListaLigada`** como definida abaixo pelo seu método construtor (**`__init__`**) , Cada elemento da lista ligada é um **`_Node`** conforme definição abaixo também.

```
class ListaLigada:
    # classe _Node - interna - nó da LL
    class _Node:
        def __init__ (self, info, prox):
            # inicia os campos
            self._info = info
            self._prox = prox

    # Construtor da classe ListaLigada
    def __init__ (self):
        ''' cria uma LL vazia.'''
        self._inicio = None   # início da LL
        self._tamanho = 0     # quantidade de elementos da LL
```

Escreva uma função **`ComparaLL(L1, L2)`** que compara o conteúdo (campo **`info`**) de 2 listas ligadas **`L1`** e **`L2`** devolvendo: **`0`** se são iguais, **`1`** se **`L1 > L2`** e **`-1`** se **`L1 < L2`**. A comparação é lexicográfica, isto é, o primeiro elemento diferente define qual o maior e qual o menor.

```
def ComparaLL(L1, L2):
    # Compara elemento a elemento
    p, q = L1._inicio, L2._inicio
    while p != None and q != None:
        if p._info > q._info: return 1
        elif p._info < q._info: return -1
        # São iguais - Continua a comparação
        p, q = p._prox, q._prox
    # Se chegou aqui - uma delas ou ambas terminaram
    if p != None: return 1   # L2 terminou e L1 não
    if q != None: return -1  # L1 terminou e L2 não
    # Ambas terminaram - Então são iguais
    return 0
```

Questão 5 (2 pontos):

Supondo que exista uma função **`ProximaSequencia(s, n)`** que devolve a próxima sequência crescente de **`1 2 3 ... n`** na ordem lexicográfica. Lembre-se que para **`n = 3`**, as sequências na ordem lexicográfica são: **`1, 12, 123, 13, 2, 23, 3`**. Exemplos:

```
s = [1]
t = ProximaSequencia(s, 3) # t fica com [1, 2]

s = [1, 3]
t = ProximaSequencia(s, 3) # t fica com [2]

s = [3]
t = ProximaSequencia(s, 3) # t fica com [] – era a última
```

Escreva uma função **`SequenciasSeguidas(n)`** que imprime todas as sequências crescentes de **`1 2 3 ... n`**, com 2 ou mais elementos, constituídas apenas de elementos seguidos. Exemplos:
Para n = 3, as sequências que satisfazem o critério são **`1 2, 2 3 e 1 2 3`**.
Para n = 4, serão: **`1 2, 2 3, 3 4, 1 2 3, 2 3 4, 1 2 3 4`**.

```
def SequenciasSeguidas(n):
    s = [1]
    while True:
        s = ProximaSequencia(s, n)
        # Se chegou ao fim sai do while
        if s == []: break
        # Verifica se s atende a condição
        if len(s) >= 2:
            # Elementos seguidos
            Imprime = True
            for k in range(len(s) - 1):
                if s[k] + 1 != s[k + 1]:
                    Imprime = False
                    break
            # Imprime ou não
            if Imprime: print(s)
```

Questão 6 (2 pontos):

Dada uma tabela **Tab** onde cada elemento **Tab[i]** contém uma string com os dados de um empregado:
Tab[i][0] a Tab[i][29] = nome – 30 posições
Tab[i][30] a Tab[39] = salário – 10 posições

A tabela **Tab** está classificada em ordem crescente de salário e dentro deste em ordem alfabética de nome. Exemplo:

```
Luiz da Silva                  0000123300
Mario Santos                   0000123300
Antonio Ferreira               0000152700
Benito de Paula                0000152700
Roberto Arruda                 0000152700
...
```

Escreva uma função **BuscaBinaria(Tab, Empr)** que procura em **Tab** um elemento igual a **Empr**, devolvendo o índice **k** tal que **Tab[k] == Empr**, ou **-1** se não encontrar. Use o algoritmo de Busca Binária na sua forma não recursiva.

```
def BuscaBinaria(Tab, empr):
    i, f = 0, len(Tab) - 1
    while i <= f:
        m = (i + f) // 2
        # Verifica se é o elemento do meio
        if Tab[m] == empr: return m
        # verifica se está abaixo ou acima
        if empr[30:40] < Tab[m][30:40]: f = m - 1
        elif empr[30:40] > Tab[m][30:40]: i = m + 1
        # Salários iguais - Verifica o nome
        elif empr[:30] < Tab[m][:30]: f = m - 1
        else: i = m + 1
    # Saiu do while - então não encontrou
    return -1
```