Gabarito Primeira Avaliação a Distância – Estrutura de Dados – 2º Semestre de 2018

1. Desenvolva um algoritmo **recursivo** que resolva o seguinte problema: Dada uma lista com n elementos (n>0), na forma de um vetor V[1..n], determinar o maior e o menor elementos desta lista. Calcule quantas chamadas recursivas o seu algoritmo efetua (incluindo a chamada externa). Pode haver elementos repetidos na lista.

R = O Algoritmo abaixo apresenta uma solução possível para o problema proposto no enunciado. Os comentários detalham o fluxo do algoritmo. Ao total são efetuadas n-1 chamadas.

```
def menor_maior_recursivo(lista, n):
    # Assume-se que o primeiro elemento é o unico, portanto, o maior e menor da lista
    saida = [lista[0], lista[0]]

    # A lista é percorrida recursivamente, do ultimo elemento até o primeiro.
    # Quando a quantidade de elementos acessados recursivamente for igual a 2,
    # descobre-se o maaior e menor desta lista
    if n == 2:
        if lista[0] > saida[0]:
            saida[0] = lista[0]
        if lista[1] > saida[1]:
            saida[1] = lista[1]

    else:
        lista_aux = lista[:n - 1]
        print("chamou")
        saida = menor_maior_recursivo(lista_aux, n - 1)

        # Ao descobrir o menor e maior elemento da lista com dois elementos,
        # é feita uma comparação destes valores com o próximo elemento da lista
        # :: Desempilhamento da recursividade
        if (lista[n - 1] > saida[1]):
            saida[1] = lista[n - 1]
        if (lista[n - 1] < saida[0]):
            saida[0] = lista[n - 1]

    return saida

lista = [5,1000,3,-666,9997,-9,5,66,-444]
menor_maior_recursivo(lista, 9)
```

2. Repita o exercício anterior, mas agora desenvolvendo um algoritmo **iterativo** para o problema. Calcule a complexidade do seu algoritmo contando o número de comparações efetuadas. (Uma comparação é uma verificação feita com um par de elementos, para decidir qual é maior e qual é menor, ou se há empate).

R = O algoritmo abaixo apresenta uma solução que realiza aproximadamente 2N comparações, a depender da disposição dos elementos presentes na lista. No exemplo dado, são realizadas 16 comparações, sendo 9 pelo código da linha sete e 7 pelo trecho de código presente linha nove.

```
1   def menor_maior_interativo(lista, n):
2       menor, maior = lista[0], lista[0]
3
4       # A lista é percorrida comparando os elementos um a um até
5       # encontrar o maior e menor.
6       for i in range(n):
7           if lista[i] < menor:
8               menor = lista[i]
9           elif lista[i] > maior:
10              maior = lista[i]
11      return [menor, maior]
12
13  lista = [5,1000,3,-666,9997,-9,5,66,-444]
14  menor_maior_interativo(lista, 9)
15
```

3. Escreva um algoritmo que leia um vetor com n>0 elementos positivos e os classifique os em faixas de 3 em 3 elementos. A resposta deve ser da forma: existem *x* elementos na faixa 1 a 3; *y* elementos na faixa 4 a 6; *z* elementos na faixa 7 a 9 etc. Qual é a complexidade do seu algoritmo?

Exemplo: se o vetor de entrada contém os elementos 9, 5, 3, 2, 6, 17, 4, 10, 11, 12, a resposta será: existem **2** elementos na faixa 1 a 3; **3** elementos na faixa 4 a 6; **1** elemento na faixa 7 a 9; **3** elementos na faixa 10 a 12; **0** elementos na faixa 13 a 15; **1** elemento na faixa 16 a 18.

R = Desconsiderando o loop realizado a partir da linha 26, utilizado apenas para ilustrar o resultado, o algoritmo proposta abaixo apresenta assintóticamente complexidade de N.

```
1   def separa_faixas(lista, n):
2       # Encontra o maior da lista
3       maior = max(lista)
4
5       # Ajusta o maior para ser multiplo de 3
6       while maior % 3 != 0:
7           maior += 1
8
9       # Define a quantidade de grupos
10      qntd_grupos = int(maior / 3)
11
12      # Cada posição do vetor 'contadores' representa um contador
13      # que representará a quantidade de elementos de um grupo
14      contadores = [0] * n
15
16      # Percorre a lista encontrando qual grupo o numero corrente pertence
17      for i in range(n):
18          aux = lista[i]
19          if aux % 3 == 0:
20              indice_grupo = int(aux / 3) - 1
21          else:
22              indice_grupo = int(aux / 3)
23          contadores[indice_grupo] += 1
24
25      # Imprime os resultados
26      for i in range(qntd_grupos):
27          limite_superior = (i + 1) * 3
28          limite_inferior = limite_superior - 2
29          print("Existem", contadores[i], "elementos na faixa de ",
30                limite_inferior,"a", limite_superior)
31
32  lista =  [9, 5, 3, 2, 6, 17, 4, 10, 11, 12]
33  separa_faixas(lista, 10)
```

4. Escreva algoritmos de **busca, inserção e remoção** de um elemento em uma **lista simplesmente encadeada ordenada com nó cabeça**. Para cada algoritmo, determine sua complexidade.

R = Todos os algoritmos apresentados abaixo possuem complexidade assintótica de N.

```python
# Os 3 algoritmos abaixo iniciam o processamento a partir da cabeça da lista
def busca(cabeca, numero):
    pont = cabeca.proximo
    #Enquando o ponteiro não for nulo, avançamos na lista procurando o número em questão
    while pont:
        if pont.valor == numero:
            return pont
        pont = pont.proximo
    return False

def insercao(cabeca, numero):
    pont = cabeca.proximo
    #Enquando o ponteiro não for nulo, avançamos até o ultimo elemento na lista
    while pont.proximo:
        pont = pont.proximo
        #Ao se ter o ponteiro para o ultimo elemento, basta adicionar um novo nó - No()
    pont.proximo = No(numero)

def remocao(cabeca, numero):
    pont = cabeca.proximo
    anterior = None

    # Percorremos alista sempre guardamos a referência do nó anterior ao atual. Ao encontrar o
    # elemento a ser removido, fazemos com que o ponteiro armazenado em 'anterior' passe a
    # apontar para o elemento que o ponteiro a ser removido aponta
    while pont:
        if pont.valor == numero:
            if anterior == None:
                cabeca.proximo = pont.proximo
            else:
                anterior.proximo = pont.proximo
        anterior = pont
        pont = pont.proximo

class No:
    def __init__(self, value):
        self.valor = value
        self.proximo = None
```

5. Escreva um algoritmo que **inverte** uma **lista simplesmente encadeada com nó cabeça**, removendo os elementos repetidos.

Exemplo: se a lista de entrada contém os elementos 3, 5, 7, 3, 8, 3, 5, 9, 1, 2 (nesta ordem), a resposta do algoritmo será a lista contendo os elementos 2, 1, 9, 5, 3, 8, 7 (nesta ordem).

R = Uma solução possível para o problema proposto é apresentada abaixo.

```
def inverte_lista(cabeca, n):
    lista2 = No(None)

    pont_L1 = cabeca.proximo
    pont_L2 = lista2

    pilha = [None] * n
    topo_pilha = 0

    # Cria uma pilha auxiliar com o conteudo da lista
    while pont_L1:
        pilha[topo_pilha] = pont_L1.valor
        topo_pilha += 1
        pont_L1 = pont_L1.proximo

    # Cria a segunda lista
    for i in range(topo_pilha - 1, -1, -1):

        pont_auxiliar = lista2
        adiciona = True

        # Verifica se o item corrente da pilha nao existe na Lista 2.
        while pont_auxiliar:
            if pont_auxiliar.valor == pilha[i]:
                adiciona = False
            pont_auxiliar = pont_auxiliar.proximo

        # Se não existir, adiciona
        if adiciona:
            pont_L2.proximo = No(pilha[i])
            pont_L2 = pont_L2.proximo

class No:
    def __init__(self, value):
        self.valor = value
        self.proximo = None
```

6. Escreva um algoritmo que leia uma sequência de votos onde cada voto tem apenas duas possibilidades (candidato A ou candidato B) e, ao término da leitura, determine qual é o candidato vencedor ou se houve empate. O seu algoritmo **não pode** fazer contagem de votos (por exemplo, usando dois contadores, um para cada candidato, e somando um ao contador correspondente de acordo com o voto lido na sequência). **Sugestão:** use uma **pilha.**

R = Uma solução possível para o problema proposto é apresentada abaixo.

```
def votacao(votos, n):
    pilha = [None] * n
    topo = 0

    # Percorremos a lista de votos comparando se o elemento no topo da pilha é igual a
    # posição corrente ou não. Caso seja, a variavel que indica a posição do topo
    # é incrementada. Caso contrário, é decrementada.
    # Ao final, o topo da pilha indicará o ganhador.
    # Caso o valor do topo seja 0, houve empate na votação.
    for i in range(n):
        voto = votos[i]
        if pilha[topo] == voto or topo == 0:
            topo += 1
            pilha[topo] = voto
        else:
            topo -= 1

    if topo == 0:
        print("Empate")
    elif pilha[topo] == "A":
        print("O candidato A venceu")
    else:
        print("O candidato B venceu")


votos = ["A", "B","A", "A", "B", "B","B","B", "A", "B", "B", "B", "B", "A", "A", "A"]
votacao(votos, len(votos))
```

7. Os clientes chegam a um banco e vão formando uma fila de atendimento (fila "A"). Pessoas com prioridade formam uma fila à parte (fila "B"). A cada minuto chega um novo cliente. Suponha que há apenas um caixa atendendo, e que o atendimento é alternado (atende-se uma pessoa da fila A e depois da fila B, e assim por diante). Desenvolva um algoritmo que leia uma sequência formada por A's e B's (correspondendo a clientes que chegam para as filas A e B) e imprima o estado das filas, sabendo que cada atendimento de um cliente na fila A dura 2 minutos, e o tempo de atendimento de um cliente na fila B dura 3 minutos. Se uma fila fica vazia, pode-se atender dois ou mais clientes da outra fila em sequência.

Exemplo: se a sequência lida é AAABAABBA, o estado das filas será: (o cliente em negrito é o que está sendo atendido)

Tempo 0: **A** --

Tempo 1: **A**A --

Tempo 2: _ **A**A -- (note que o primeiro cliente saiu, e o segundo já começou a ser atendido)

Tempo 3: _ **A**A B

Tempo 4: _ _ AA **B**

Tempo 5: _ _ AAA **B**

Tempo 6: _ _ AAA **B**B

Tempo 7: _ _ **A**AA _ BB

Tempo 8: _ _ **A**AAA _ BB

Tempo 9: _ _ _ AAA _ **B**B

Tempo 12: _ _ _ **A**AA _ _ B

Tempo 14: _ _ _ _ AA _ _ **B**

Tempo 17: _ _ _ _ **A**A _ _ _

Tempo 19: _ _ _ _ _ **A** _ _ _

Tempo 21: _ _ _ _ _ _ _ _ _ (FIM)

R = Algoritmo na próxima página. Obs.: As funções *print_tempo(), inserir_na_fila(),* e *procurar_posicoes_faltantes()*, são funções auxiliares respectivamente utilizadas para:

- Imprimir a saída do algoritmo;

- Inserir um elemento em uma determinada fila

- Receber uma lista com uma sequência numérica e identificar se há alguma elemento faltante. Por exemplo, considerando a lista [0,1,3] a função iria retornar o número 2 (ou seja, o elemento que falta para a sequencia ficar completa).

```python
def filas(clientes, n):

    # Variaveis de controle
    ultimo_cliente_atentido = ""
    posicoes_atendidas = []  # fila contendo as posições atendidas
    tempo = -1
    trecho_A = ""
    trecho_B = ""

    # Enquanto a posição I for menor que N (quantidade de clientes na fila)
    i = 0
    while i < n:

        # Se o cliente ainda nao foi atendido
        if i not in posicoes_atendidas and i >= 0:

            # Faz uma busca entre a posição de i e a posição i+tempo para saber se precisa trocar de fila
            # Ou seja, para saber se vai atender um cliente da fila A ou da fila B
            # É útil a partir da segunda iteração na lista de clientes
            j = i
            troca_fila = False
            while troca_fila == False and j < i + tempo and j < n:

                if clientes[j] != ultimo_cliente_atentido:
                    troca_fila = True
                j += 1

            # Se precisar trocar de fila, o cliente atual é considerado o cliente na posição J
            if troca_fila:
                j = j - 1
                cliente_atual = clientes[j]
                inserir_na_fila(j, posicoes_atendidas)

            # Caso contrario é considerado o cliente da posição i
            else:
                cliente_atual = clientes[i]
                inserir_na_fila(i, posicoes_atendidas)

            # Verifica quem é o cliente atual e define qual o incremento de tempo
            if cliente_atual == "A":
                trecho_A += "A"
                incremento = 2
            else:
                trecho_B += "B"
                incremento = 3

            msg = trecho_A + " " + trecho_B

            print_tempo(tempo, incremento, msg)

            tempo += incremento

            # Verificação para saber qual sera o proximo indice da lista de clientes a ser lido
            # Se houve uma troca de fila, faz uma verificação para saber se algua posição
            ## entre o primeiro e o ultimo cliente foi pulado
            if troca_fila:
                posicoes_puladas = procurar_posicoes_faltantes(posicoes_atendidas)
                if len(posicoes_puladas) > 0:
                    i = posicoes_puladas[0]
                else:
                    i = max(posicoes_atendidas) + 1

            # Caso contrario, apenas incrementa
            else:
                i += 1

pessoas = list("AAABAABBA")
filas(pessoas, 9)
```

8. Escreva um exemplo de entrada com 10 elementos que leve o algoritmo de **ordenação por seleção** ao seu pior caso. Determine quantas comparações entre elementos o algoritmo efetua neste caso. Descreva, passo a passo, todas as trocas entre elementos efetuadas pelo algoritmo para ordenar a entrada.

R= O Algoritmo de ordenação por seleção sempre irá comparar cada elemento do vetor com todos os outros. Considerando isso, podemos afirmar que, independente da entrada, sempre serão executadas todas as comparações, não havendo diferença em complexidade entre o melhor e pior caso. Considerando a sequência abaixo, 45 comparações ocorrerão. Destas, 5 resultam na troca de elementos da lista.

Note que a partir da quinta iteração já não há mais trocas, mesmo assim o algoritmo segue realizando comparações. Uma proposta para contornar esta característica seria adicionar uma variável que contabiliza a quantidade de trocas. A partir da interação que não haja mais nenhuma troca, o loop poderia ser interrompido.

10	9	8	7	6	5	4	3	2	1	Sequência Inicial
1	9	8	7	6	5	4	3	2	10	
1	2	8	7	6	5	4	3	9	10	
1	2	3	7	6	5	4	8	9	10	
1	2	3	4	6	5	7	8	9	10	
1	2	3	4	5	6	7	8	9	10	
1	2	3	4	5	6	7	8	9	10	
1	2	3	4	5	6	7	8	9	10	
1	2	3	4	5	6	7	8	9	10	
1	2	3	4	5	6	7	8	9	10	
1	2	3	4	5	6	7	8	9	10	Sequência Final

9. Repita o exercício anterior, mas agora considerando o algoritmo de **ordenação pelo método da bolha.**

R = O Algoritmo de ordenação bolha consiste em percorrer um vetor e, em cada passagem, empurrar o maior elemento para o final da lista. Um vetor cujos elementos estão organizados em ordem decrescente apresenta o pior caso. Considerando a sequência abaixo, 45 comparações ocorrerão. Destas, todas resultam na troca de elementos na lista.

10	9	8	7	6	5	4	3	2	1	Sequência Inicial
9	10	8	7	6	5	4	3	2	1	
9	8	10	7	6	5	4	3	2	1	
9	8	7	10	6	5	4	3	2	1	
9	8	7	6	10	5	4	3	2	1	
9	8	7	6	5	10	4	3	2	1	
9	8	7	6	5	4	10	3	2	1	
9	8	7	6	5	4	3	10	2	1	
9	8	7	6	5	4	3	2	10	1	
9	8	7	6	5	4	3	2	1	10	
8	9	7	6	5	4	3	2	1	10	
8	7	9	6	5	4	3	2	1	10	
8	7	6	9	5	4	3	2	1	10	
8	7	6	5	9	4	3	2	1	10	
8	7	6	5	4	9	3	2	1	10	
8	7	6	5	4	3	9	2	1	10	
8	7	6	5	4	3	2	9	1	10	
8	7	6	5	4	3	2	1	9	10	
7	8	6	5	4	3	2	1	9	10	
7	6	8	5	4	3	2	1	9	10	
7	6	5	8	4	3	2	1	9	10	
7	6	5	4	8	3	2	1	9	10	
7	6	5	4	3	8	2	1	9	10	
7	6	5	4	3	2	8	1	9	10	
7	6	5	4	3	2	1	8	9	10	
6	7	5	4	3	2	1	8	9	10	
6	5	7	4	3	2	1	8	9	10	
6	5	4	7	3	2	1	8	9	10	
6	5	4	3	7	2	1	8	9	10	
6	5	4	3	2	7	1	8	9	10	
6	5	4	3	2	1	7	8	9	10	
5	6	4	3	2	1	7	8	9	10	
5	4	6	3	2	1	7	8	9	10	
5	4	3	6	2	1	7	8	9	10	
5	4	3	2	6	1	7	8	9	10	
5	4	3	2	1	6	7	8	9	10	
4	5	3	2	1	6	7	8	9	10	
4	3	5	2	1	6	7	8	9	10	
4	3	2	5	1	6	7	8	9	10	
4	3	2	1	5	6	7	8	9	10	
3	4	2	1	5	6	7	8	9	10	
3	2	4	1	5	6	7	8	9	10	
3	2	1	4	5	6	7	8	9	10	
2	3	1	4	5	6	7	8	9	10	
2	1	3	4	5	6	7	8	9	10	
1	2	3	4	5	6	7	8	9	10	Sequência Final

10. Desenvolva um algoritmo que ordene uma lista que só contenha dois tipos de valores, por exemplo 0 e 1. O seu algoritmo deve executar em tempo linear, isto é, O(n) – onde n é o número de elementos a serem ordenados.

Exemplo: se a lista de entrada é 0, 1, 1, 0, 0, 1, 1, a saída será 0, 0, 0, 1, 1, 1, 1.

R = Uma solução possível para o problema proposto é apresentada abaixo.

```
1   def ordenacao_linear(lista, n):
2       indice_auxiliar = 0
3
4       # O algoritmo percorre a lista comparando o indice
5       # auxiliar com a posição corrente
6
7       # O indice auxiliar é incrementado em cada iteração
8       # do loop
9       for i in range(1, n):
10          if lista[indice_auxiliar] == lista[i]:
11              temp = lista[i]
12              lista[i] = lista[indice_auxiliar + 1]
13              lista[indice_auxiliar + 1] = temp
14
15              indice_auxiliar += 1
16
17  ordenacao_linear([0, 1, 1, 0, 0, 1, 1],7)
18
```