FUNDAÇÃO GETÚLIO VARGAS
ESCOLA DE MATEMÁTICA APLICADA
MESTRADO 2015.1
ESTRUTURA DE DADOS E SEUS ALGORITMOS
Prof Alexandre Rademaker

Divisão e Conquista: Implementações

GRUPO:
KIZZY TERRA
OTTO TAVARES
VIVIAN TOMÉ

RIO DE JANEIRO
ABRIL DE 2015

1 Exercício 2.17

1.1 Utilizando Array

Implementação 1 Exercício 2.17

```
def exercicio17(L):
    n = len(L)
    if n==1:
        if L[0] == 0:
            print 'true'
    else:
        if L[n/2] == n/2:
            print 'true'
        else:
            if L[n/2] > n/2:
                exercicio17(L[:n/2])
            else:
                exercicio17(L[n/2:]-n/2)
```

1.2 Utilizando Lista Encadeada

Implementação 2 Implementação de uma Lista Encadeada

```
def Node(value, next):
    return [value, next]

def emptyList():
    return []

#insere um valor no início da lista
def insert(value, head):
    return Node(value, head)

# retorna uma nova lista sem o nó correspondente a variável value
def remove(value, lst):
    current = lst
    next = lst[1]
    if current[0] == value:
        lst = current[1]
        return lst
    else:
        while next:
            if next[0] == value:
                current[1] = next[1]
                return lst
            else:
                current = next
                next = next[1]

Exemplo de lista encadeada:
    list = Node(1, Node(0, Node(3, [])))
    list = insert(4, list)
    list = remove(3, list)
```

Analisando as operações implementadas para a lista e o algoritmo proposto para a solução do exercício 2.17 fica claro que não é possível implementá-lo em O(log n). Para chegar ao meio da lista, por exemplo, precisamos percorrer todos os elementos da lista até o centro, essa operação

tem custo k*n/2 = O(n). Desta forma, no caso de uma lista encadeada a solução mais simples para este exercício é visitar elemento a elemento da lista para verificar se A[i] = i e retornar verdadeiro na primeira ocorrência. Segue a implementação:

Implementação 3 Exercício 2.17 - Lista

```
def exercicio17_lista(lst):
    index = 0
    value = lst[0]
    next = lst[1]
    while index != value:
        if next:
            index +=1
            value = next[0]
            next = next[1]
        else: return 'false'
    return 'true'
```

A função acima recebe uma lista encadeada

2 Exercício Resolvido 1 (capítulo 5 - Eva Tardos)

2.1 Utilizando Array

A função implementada deve receber um vetor unimodal.

Implementação 4 Exercício 1

```
def exercicio1(L):
    n = len(L)
    if n < 3:
        if L[0] > L[1]:
            return L[0]
        else:
            return L[1]
    else:
        if L[n/2] > L[n/2 + 1]:
            if L[n/2] > L[n/2 - 1]:
                return L[n/2]
            else:
                return exercicio1(L[:n/2])
        else:
            return exercicio1(L[n/2:])
```

2.2 Utilizando Lista Encadeada

Assim como no exercício anterior, se utilizarmos lista encadeada para implementar este algoritmo a complexidade passa a ser O(n) devido aos acessos aleatórios que são utilizados no algoritmo. Considerando a complexidade O(n) podemos implementar um outro algoritmo utilizando uma lista duplamente encadeada ao invés de utilizar uma lista encadeada, para faciitar a comparação entre um nó, seu nó antecessor e seu nó sucessor. O novo algoritmo consiste simplesmente de percorrer a lista até encontrar o nó que satisfaz a propriedade desejada (nó i: tal que A[i] > A[i-1] e A[i] > A[i+1]).

Implementação 5 Implementação de uma Lista Duplamente Encadeada

```
class DNode:
    def __init__(self):
        self.value = None
        self.next = None
        self.previous = None


class DLList:
    def __init__(self):
        self.head_node = DNode()
        self.tail_node = DNode()

    def add_node_head(self, value):
        new_node = DNode()
        new_node.value = value
        if not (self.head_node.next and self.tail_node.previous):
            self.head_node.next = new_node
            self.tail_node.previous = new_node
        else:
            if self.head_node.next:
                new_node.next = self.head_node.next
                self.head_node.next = new_node
                new_node.next.previous = new_node

    def add_node_tail(self, value):
        new_node = DNode()
        new_node.value = value
        if not (self.head_node.next and self.tail_node.previous):
            self.head_node.next = new_node
            self.tail_node.previous = new_node
        else:
            if self.tail_node.previous:
                new_node.previous = self.tail_node.previous
                self.tail_node.previous = new_node
                new_node.previous.next = new_node

    def print_(self):
        node = self.head_node.next
        while node:
            print node.value
            node = node.next

Exemplo de lista duplamente encadeada unimodal:
    listDup = DLList()
    listDup.add_node_tail(8)
    listDup.add_node_tail(9)
    listDup.add_node_tail(10)
    listDup.add_node_tail(11)
    listDup.add_node_tail(12)
    listDup.add_node_tail(7)
    listDup.add_node_tail(6)
    listDup.add_node_tail(5)
    listDup.add_node_tail(4)
    listDup.add_node_tail(3)
    listDup.print_()
```

A função a seguir deve receber um objeto do tipo DLList (lista duplamente encadeada implementada acima) unimodal.

Implementação 6 Exercício 1 - Lista

```
def exercicio1_lista(L):
    node = L.head_node.next
    while node.value < node.next.value and node.next:
        node = node.next
    return node.value
```