Prova de Recuperação — PMR3201 1o. semestre 2017

Prof. Thiago Martins

Instruções: Escreva o nome e o número USP na folha de papel almaço. Numere cada página. Indique o total de páginas na primeira página. Os códigos fornecidos na seção "Códigos-fonte de apoio" podem ser referenciados em qualquer resposta sem necessidade de reprodução.

1. (2,0 pontos) Uma Fila é uma estrutura de dados que armazena dados em uma ordem específica. Os dados são retirados *na mesma ordem* em que são inseridos. A classe `Fila` implementa esta estrutura de dados. O método `enqueue(self, x)` adiciona o objeto `x` à pilha. O método `dequeue(self)` *retira* um objeto da lista e retorna-o, ou, no caso em que a fila esteja vazia, lança uma exceção. O método `tamanho(self)` retorna a quantidade de objetos armazenados na estrutura.

   Escreva uma função em Python que *esvazia* uma pilha que contém inteiros e retorna `True` se os inteiros foram inseridos na pilha em ordem crescente ou `False` caso contrário.

   Use a seguinte assinatura:

   ```
   def ordenada(p):
   ```

   Onde `p` é a pilha a ser esvaziada.

   Resposta:

   ```
   def ordenada(p):
       if p.tamanho()==0:
           return True
       anterior = p.dequeue()
       while p.tamanho():
           atual = p.dequeue()
           if atual < anterior:
               return False
       return True
   ```

2. (2,0 pontos) Uma lista duplamente ligada ou *deque* é uma estrutura de dados composta por uma sequência de nós. Cada nó possui referências para o próximo nó na sequência e o nó anterior. A classe `NoDeque` implementa o nó em tal estrutura. O campo `p` contém uma referência para o nó anterior na sequência, enquanto que o campo `n` contém uma referência para o próximo. O *último* nó tem como valor do seu campo `n` o valor `None`. Do mesmo modo, o *primeiro* nó tem como valor do se campo o valor `None`.

   Escreva em Python uma função que recebe uma referência a um nó em um deque e retorna a fração (em ponto flutuante) de elementos do deque que o *antecedem*, sobre o número total de elementos.

   Utilize a seguinte assinatura:

   ```
   def fracao_anteriores(n):
   ```

   onde `n` é o nó em questão.

   Resposta:

```
def fracao_anteriores(n):
    anteriores = 0
    p = n.p
    while p:
        anteriores = anteriores + 1
        p = p.p
    totais = anteriores
    while n:
        totais = totais + 1
        n = n.n

    return anteriores/totais
```

3. (2,0 pontos) 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:

```
1  def elemento_faltante(a):
2      def elemento_faltante_rec(esquerda, direita):
3          if esquerda >= direita:
4              return esquerda
5          else:
6              m = (esquerda+direita)//2
7              if a[m] > m:
8                  return elemento_faltante_rec(esquerda, m)
9              else:
10                 return elemento_faltante_rec(m+1, direita)
11     return elemento_faltante_rec(0, len(a)-1)
```

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

   (a) (1,0 pontos) Mostre que o algoritmo está correto, ou seja, efetivamente retorna o elemento faltante.

   Resposta: Seja $S = \{s_0, s_1, \ldots, s_n\}$. Existe um índice j entre 0 e n tal que:

$$s_i = i \, \forall i < j \tag{1}$$

$$s_i = i + 1 \, \forall i \geq j \tag{2}$$

   Nota-se que j é também o valor do elemento que foi removido.

   A listagem mostra um algoritmo recursivo que busca o elemento faltante entre os índices e, na variável `esquerda` e d, na variável `direita` (inclusivos). O caso base do algoritmo é naturalmente quando $e = d$, quando a resposta é a única possível, e. Para todos os outros casos, o algoritmo calcula o índice $m = \left\lfloor \frac{e+d}{2} \right\rfloor$. Nota-se que $e \leq m < d$. Por (2), se $s_m > m$, então tem-se necessariamente $e \leq j \leq m$, de modo que a resposta da chamada recursiva na linha 8 é a resposta do problema. De (1) se $s_m \leq m$, então tem-se $m < j \leq d$, de modo que a

resposta da chamada recursiva nda linha 10 é a resposta do problema. Finalmente, verifica-se que como $m < d$, o intervalo considerado na chamada da linha 8 é *estritamente menor* do que o intervalo original. Do mesmo modo, como $m + 1 > e$, o intervalo na chamada da linha 10 também é *estritamente menor*. Porém, como vale $m \geq e$ e $m + 1 \leq d$, o tamanho destes intervalos nunca é nulo. Assim, o tamanho do intervalo em cada nível de chamada recursiva é uma grandeza inteira, estritamente decrescente e limitada inferiormente. Logo o algoritmo termina em tempo finito com a resposta correta.

(b) (1,0 pontos) Calcule a complexidade do algoritmo.

Resposta: Seja $T(N)$ a complexidade de tempo de execução do algoritmo, onde N é o tamanho do vetor original. O algoritmo executa operações em tempo constante e chama a si mesmo em um intervalo com *metade* do tamanho do vetor original. Esta operação é repetida $\lceil \log_2 N \rceil$ vezes. Assim,

$$T(N) = \lceil \log_2 N \rceil \mathcal{O}(N) = \mathcal{O}(\log N)$$

De fato, esta análise é a mesma do algoritmo de busca binária.

4. (2,0 pontos) Um heap binário é uma árvore binária completa (todos os níveis exceto o último cheios) preenchido da esquerda para a direita na qual cada nó é *maior ou igual* a seus filhos. Heaps binários são armazenados em vetores, resultantes da varredura da árvore binária em largura. Dada a rígida estrutura, é possível converter um vetor em árvore e vice-versa.

   Escreva uma função em Python que determina se um vetor de inteiros contém um Heap Binário válido.

   Utilize a seguinte assinatura:

   `def verifica_heap(v):`

   Onde `v` é o vetor em questão. A função deve retornar `True` se o vetor `v` contém um Heap Binário válido ou `False` caso contrário.

Resposta: Basta checar se para cada elemento seus dois filhos são efetivamente menores ou iguais a ele. Note o caso especial no último pai que pode ter um ou dois filhos.

```
def verifica\_heap(v):
    pai = 0
    filho = 1
    while filho < len(v)-1:
        if v[pai] < v[filho] || v[pai] < v[filho+1]:
            return false
        pai = pai+1
        filho = 2*pai+1
    if filho < len(v):
       if v[pai] < v[filho]:
           return False
    return True
```

5. (2,0 pontos) Um vetor de inteiros $A = \{a_0, \ldots a_{n-1}\}$ está *arrumado* se existe uma posição k tal que $i < k \Rightarrow a_i \leq a_k$ e $i > k \Rightarrow a_i \geq a_k$, ou seja, existe ao menos um elemento de A que é *menor ou igual* a todos os seus antecessores e *maior ou igual* a todos os seus sucessores. Por exemplo, o vetor

$2, 4, 1, 4, 6, 8, 4$ está arrumado, pois o elemento na posição de índice 3 (com valor 4) é maior ou igual a seus antecessores e menor ou igual aos seus sucessores. Por outro lado, o vetor $2, 4, 1, 6, 4, 8, 4$ *não está*, pois não há elemento que atenda este critério.

Escreva uma função em Python que determina com complexidade $\mathcal{O}(N)$ se um vetor está arrumado ou não.

Utilize a seguinte assinatura:

```
def arrumado(v):
```

Onde `v` é o vetor em questão.

Resposta: O problema pode ser resolvido por uma modificação do tradicional algoritmo sequencial de busca do maior elemento de um vetor. De fato, seja K o conjunto de todos os índices k que satisfazem a condição do problema. A questão traduz-se em determinar se o conjunto K é vazio ou não. Seja o conjunto M de índices do vetor A definido da seguinte forma:

$$M = \{0\} \cup \{x \in [0 \dots n-1] \mid a_x > a_y \, \forall y < x\}$$

Este é o conjunto de índices considerados por um algoritmo sequencial de busca do maior elemento de A que varre o vetor do índice 0 ao $n-1$. Trivialmente, nota-se que $K \subseteq M$. Uma possível solução aqui é encontrar N, a sequência dos índices de um algoritmo de busca pelo *menor* elemento que opera em ordem reversa, e verificar que $K = M \cap N$, mas existe uma solução ainda mais simples. De fato, observa-se que o conjunto complementar de K em relação a M define-se como:

$$M - K = \{x \in M \mid \exists y > x, a_y < a_x\}$$

Deste modo percebe-se que não é necessário gerar todo o conjunto M, apenas manter durante a varredura do vetor o seu menor elemento que ainda satisfaz as condições do problema. Se durante a varredura é encontrado um índice i tal que $a_i < a_j$, $j \in M \cap [0 \dots i]$, o elemento atual é descartado e o algoritmo procede até encontrar um novo elemento em M ou o vetor se esgotar. A listagem abaixo implementa este algoritmo:

```
def arrumado(v):
    arrumado = True
    maior = v[0]
    candidato = maior
    for i in range(1, len(v)):
        if v[i] > candidato:
            maior = v[i]
            if not arrumado:
                arrumado = True
                candidato = maior
        elseif v[i] < candidato:
            arrumado = False
    return arrumado
```

Formulário

Solução de equações de recorrência pelo *Master Theorem*:

A equação:

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + cN^L,$$

com $T(1) = \mathcal{O}(1)$, tem solução

$$T(N) = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L \end{cases}$$

Códigos-fonte de apoio

Classe `NoListaLigada`

```
class NoListaLigada:
    def __init__(self, x):
        """Inicializa nó isolado"""
        self._x = x
        self.p = None
        self.n = None

    def set_after(self, other):
        """define other como o próximo nó de self"""
        self.n = other
        if other:
            other.p = self

    def set_before(self, other):
        """define other como o nó anterior a self"""
        self.p = other
        if other:
            other.n = self
```

Classe `Pilha`

```
class Pilha:
    """Implementa uma pilha usando listas ligadas"""
    def __init__(self):
        self._e = None
        self._n = 0

    def tamanho(self):
        return self._n

    def push(self, x):
        """Adiciona um elemento à Pilha"""
        novo = NoListaLigada(x)
        novo.set_after(self._e)
        self._e = novo
        self._n += 1

    def pop(self):
        """Remove um elemento da Pilha"""
        if self._e is None:
            raise Exception("Pilha Vazia")
        x = self._e._x
        self._e = self._e.n
        if self._e:
            self._e.p = None
        self._n -= 1
        return x

    def top(self):
        """Retorna o elemento do topo sem alterar a Pilha"""
        if self._e is None:
            raise Exception("Pilha Vazia")
        return self._e._x
```

Classe `Fila`

```
class Fila:
    """Implementa uma fila usando listas ligadas"""
    def __init__(self):
        self._e = None
        self._s = None
        self._n = 0

    def tamanho(self):
        return self._n

    def enqueue(self, x):
        """Adiciona um elemento à fila"""
        if self._e is None: # Fila Vazia
            self._s = self._e = NoListaLigada(x)
        else:
            novo = NoListaLigada(x)
            novo.set_after(self._e)
            self._e = novo
        self._n += 1

    def dequeue(self):
        if self._s is None: # Fila Vazia
            raise Exception("Fila Vazia")
        x = self._s._x
        self._s = self._s.p
        if self._s:
            self._s.n = None
        else: # Fila esvaziada!
            self._e = None
        self._n -= 1
        return x
```

Classe `NoArvoreBinaria`

```
class NoArvoreBinaria:
    def __init__(self, x, e = None, d = None):
        self._x = x
        self.e = e
        self.d = d
```