Prova de Recuperação — PMR3201 1o. semestre 2018

Escreva o nome e o número USP na folha de papel almaço. Numere cada página. Indique o total de páginas na primeira página. Você não pode empregar nenhum método do *runtime* python que tenha complexidade pior do que constante.

1. (2,0 pontos) Uma Pilha é uma estrutura de dados na qual o elemento a ser removido é o *mais recente* a ser inserido. Suponha que em uma pilha é inserida a sequência $\{1, 2, 3, 4, 5\}$, *nesta ordem*. A ordem relativa entre inserções e remoções dos elementos é variável, mas as remoções são feitas de modo que a pilha ao final está vazia. A ordem relativa entre inserções e remoções pode produzir diversas sequências de elementos removidos. Aponte entre as seguir quais são possíveis e quais não o são:

   a) $1, 2, 3, 4, 5$
   b) $5, 4, 3, 2, 1$
   c) $1, 2, 3, 5, 4$
   d) $2, 1, 3, 4, 5$
   e) $5, 4, 1, 2, 3$
   f) $2, 3, 5, 4, 1$
   g) $4, 5, 2, 3, 1$
   h) $3, 4, 2, 1, 5$

   Dica: A *maioria* das sequências acima é possível, mas não todas!

   Resposta: Como os elementos foram inseridos em ordem crescente, quando um determinado elemento é retirado, todos os menores já foram inseridos. Do mesmo modo, uma ordem crescente só se apresenta na remoção quando um elemento é retirado antes do seu sucessor ser inserido.

   Assim, a sequência e) não é possível, pois o 5 só seria retirado se 1, 2 e 3 já estivessem na pilha. Deste modo é impossível que os elementos 1, 2 e 3 apareçam na remoção nesta ordem.

   O mesmo argumento vale para a sequência g) onde a remoção do elemento 4 antecede a sequência 2, 3.

   Todas as outras são possíveis.

2. (2,0 pontos) Algoritmos de Bresenham são empregados em computação gráfica para determinar quais pontos em uma tela pertencem a uma determinada curva. O problema de se traçar um arco de circunferência de raio $r \geq 1$, centrado na coordenada $(0, 0)$, do seu ponto mais alto até 45^o em sentido horário, pode ser visto como a busca da sequência $Y = \{y_0, \ldots y_n\}$ tal que a entrada y_x corresponde ao ponto (x, y). Vale para esta sequência as seguintes propriedades:

$$y_x = max\left\{y \in \mathbb{N} \mid y^2 + x^2 \leq r^2\right\} \tag{1}$$

$$y_i \geq i \tag{2}$$

   A propriedade (1) significa que cada ponto y_x é o ponto com *maior* coordenada inteira y dentro da circunferência de raio r^2 centrada em $(0, 0)$ sobre a reta vertical com coordenada x. Em particular, nota-se que $y_0 = r$. A propriedade (2) limita a curva a 45^o.

   Por exemplo, para $r = 10$, tem-se a sequência $\{10, 9, 9, 9, 9, 8, 8, 7\}$.

   A listagem a seguir implementa um algoritmo que produz esta sequência:

```
1 def bresenham(r):
2     out = []
3     y = r
4     x = 0
5     r2 = r*r
6     y2 = r*r
7     x2 = 0
8     while y >= x:
9         out.append(y)
```

```
10          x2 += x + x + 1
11          x += 1
12          if y2 + x2 > r2:
13              y2 -= y + y + 1
14              y -= 1
15      return out
```

onde `r` é o raio da circunferência. Note que a exceção da linha 5, este algoritmo não faz nenhuma multiplicação. Mostre que o algoritmo está correto.

Resposta: Seja y_i, x_i os valores das variáveis `y`, `x` ao final da i-gésima iteração do laço iniciado na linha 8, com $y_0 = r$ e $x_0 = 0$.

Seja $\hat{y}_i, \hat{x}_i$ os valores das variáveis `y2`, `x2` ao final da i-gésima iteração do laço iniciado na linha 8, com $\hat{y}_0 = r^2$ e $\hat{x}_0 = 0$.

A lei de transição é:

$$
\begin{aligned}
x_{i+1} &= x_i + 1 \\
\hat{x}_{i+1} &= \hat{x}_i + 2x_i + 1 \\
y_{i+1} &= \begin{cases} y_i & \text{para } \hat{y}_i + \hat{x}_{i+1} \leq r^2 \\ y_i - 1 & \text{para } \hat{y}_i + \hat{x}_{i+1} > r^2 \end{cases} \\
\hat{y}_{i+1} &= \begin{cases} \hat{y}_i & \text{para } \hat{y}_i + \hat{x}_{i+1} \leq r^2 \\ \hat{y}_i - 2y_i - 1 & \text{para } \hat{y}_i + \hat{x}_{i+1} > r^2 \end{cases}
\end{aligned}
$$

Trivialmente, vale $x_i = i$. Mais importante, $\hat{x}_i = {x_i}^2$. De fato, isso é verdadeiro para $i = 0$. Porém, se existe algum i para o qual isso é verdadeiro, então pela lei de transição vale:

$$
\begin{aligned}
\hat{x}_{i+1} &= \hat{x}_i + 2x_i + 1 \\
&= {x_i}^2 + 2x_i + 1 \\
&= (x_i + 1)^2 \\
&= {x_{i+1}}^2
\end{aligned}
$$

O mesmo argumento mostra que $\hat{y}_i = {y_i}^2$ (note que as condições são as mesmas para a atualização tanto de y_i quanto de $\hat{y}_i$).

Finalmente, mostra-se que $y_i = max\left\{y \in \mathbb{N} \mid y^2 + {x_i}^2 \leq r^2\right\}$.

Em primeiro lugar, note-se que

$$max\left\{y \in \mathbb{N} \mid y^2 + {x_i}^2 \leq r^2\right\} \leq max\left\{y \in \mathbb{N} \mid y^2 + {x_{i+1}}^2 \leq r^2\right\}$$

A proposição é verdadeira para y_0.

Por outro lado, se existe i para o qual $y_{i+1} \leq x_{i+1}$ e $y_i = max\left\{y \in \mathbb{N} \mid y^2 + {x_i}^2 \leq r^2\right\}$, então se ${y_i}^2 + {x_{i+1}}^2 \leq r$ tem-se necessariamente $max\left\{y \in \mathbb{N} \mid y^2 + {x_{i+1}}^2 \leq r^2\right\} = y_i$.

Por outro lado, seja o caso em que ${y_i}^2 + {x_{i+1}}^2 > r$. Neste caso, evidentemente $y_i > max\left\{y \in \mathbb{N} \mid y^2 + {x_i}^2 \leq r^2\right\}$.

No entanto, substituindo-se a expressão de x_{i+1} em função de x_i, chega-se a ${y_i}^2 + {x_i}^2 + 2x_i + 1 > r$. Porém, como vale $y_i > x_i$, então $(y_i - 1)^2 + {x_{i+1}}^2 = {y_i}^2 - 2y_i - 1 + {x_i}^2 + 2x_i + 1 < {y_i}^2 + {x_i}^2 \leq r^2$. Ora, o *primeiro* número inteiro menor do que y_i é $y_i - 1$ de modo que $max\left\{y \in \mathbb{N} \mid y^2 + {x_{i+1}}^2 \leq r^2\right\} = y_i - 1 = y_{i+1}$.

Assim mostra-se que o algoritmo está correto.

3. (2,0 pontos) Uma matriz bidimensional em Python pode ser representada por uma sequência de sequências. Assim, a matriz

$$X = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

pode ser escrita em Python como:

```
X = [[1, 2], [3, 4]]
```

O elemento $X_{1,2}$ da matriz acima, cujo valor é 2, pode ser acessado através da variável `x` acima declarada com a seguinte sintaxe: `x[0][1]` (note a convenção em Python de se utilizar índices baseados em 0).

Seja uma matriz quadrada $N \times N$ cujos coeficientes são 0 ou 1. Em cada linha da matriz, um 0 *jamais* antecede um 1. Por exemplo:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Escreva em Python uma função que encontra com complexidade $\mathcal{O}(N)$ o índice (baseado em zero) da linha da matriz com a *maior* quantidade de 0's.[1] Utilize a seguinte assinatura:

```
def encontra_mais_zeros(x):
```

onde `x` é uma referência à sequência de sequências que representa a matriz acima descrita. Sua função deve retornar o índice da sequência com a linha que contém a *maior* quantidade de 0's.

Funções que desempenhem corretamente o descrito acima com complexidade pior do que $\mathcal{O}(N)$ valem 0,5 pontos.

Resposta: Este problema aparentemente demanda uma solução quadrática. No entanto, note que se uma determinada linha possui 0's até a j-ésima coluna, é possivel ignorar todas as linhas seguintes que possuam 1's até a posição j.

```
def encontra_mais_zeros(x):
    n = len(x)  # Matriz quadrada
    # primeira linha
    j = 0
    while j < n and x[0][j] != 1:
        j += 1
    maior_linha = 0
    maior_sequencia = j
    for i in range(len(x)):
        while j < n and x[i][j] != 1:
            j += 1
        if j > maior_sequencia:
            maior_linha = i
            maior_sequencia = j
    return maior_linha
```

4. (2,0 pontos) Uma árvore binária de busca é uma árvore binária na qual todos os elementos à esquerda de um dado nó são *menores* do que o nó e todos os elementos à direita do nó são *maiores*. A classe `NoArvoreBinaria` implementa um nó de árvore binária.

[1] Robert Sedgewick e Kevin Wayne. *Algorithms*. 4th. sec. 1.4. Addison-Wesley Professional, 2011.

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

O campo `e` aponta para a sub-árvore esquerda, ou `None` se esta não existe. O campo `d` aponta para a sub-árvore direita, ou `None` se esta não existe. O campo `x` contém o valor armazenado no nó.

Um sistema armazena valores em ponto flutuante nos nós de uma árvore binária *perfeitamente balanceada* (ou seja, todos os níveis exceto o último estão completamente preenchidos). Deseja-se implementar um método que retorna o valor armazenado *mais próximo* a um valor dado.

Use a seguinte assinatura:

```
def encontra_mais_proximo(r, x):
```

Onde `r` é o nó raiz da arvore balanceada que armazena os números e `x` é o número de ponto flutuante que deseja-se aproximar. Sua função deve ter complexidade logarítmica, ou seja, $\mathcal{O}(\log N)$ onde N é a quantidade de elementos armazenados na árvore balanceada.

5. (2,0 pontos) A base de dados da questão 4 realiza, além da operação de busca pelo elemento mais próximo, operações de inserção de novos elementos e remoção de elementos preexistentes. Espera-se que todas as operações ocorram com frequência similar. Um programador observa que árvores auto-balanceadas, como árvores AVL, podem implementar as operações de inserção e remoção em complexidade $\mathcal{O}(\log N)$. Outro programador sugere que tabelas de espalhamento (*hash tables*) seriam ainda melhores do que árvores auto-balanceadas para esta base de dados, pois as operações de inserção e remoção podem ser feitas com complexidade *constante*. Discuta os argumentos e aponte qual estrutura de dados é mais adequada para esta base de dados, árvores auto-balanceadas ou tabelas de espalhamento.