Prova Substitutiva — PMR3201 1o. semestre 2018
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. Não referencie elementos de uma classe iniciados pelo caractere "_" a menos que o seu código faça parte da implementação desta classe. A menos que expressamente instruído ao contrário, as funções que você criar não devem modificar os parâmetros passados. Você não pode empregar nenhum método do *runtime* python que tenha complexidade pior do que constante.

Questões

1. (2,0 pontos) Uma fila é uma estrutura de dados na qual o elemento a ser retirado é o mais antigo ainda presente. A classe `fila_circular` implementa uma fila utilizando a estratégia de "sequência circular". O campo `_dados` contém uma sequência com os dados enfileirados. Posições não-utilizadas são preenchidas com `None`. O campo `_indiceVaiSair` armazena o índice do próximo elemento a ser retirado da fila, caso exista algum. O campo `_indiceEntrou` armazena o índice do último elemento a ser inserido na fila, caso exista algum. O campo `_tamanho` contém a quantidade de elementos enfileirados. O método `enqueue(self, x)` adiciona o elemento `x` na fila. Caso não haja espaço na sequência atual para a inserção do elemento, o método `_duplique_seq(self)` é invocado. Este método cria uma nova sequência com o dobro da referenciada por `_dados`. Os elementos são inseridos na nova sêequncia pela ordem da fila, a partir da primeira posição. Finalmente, a operação de adição é concluída. O método `dequeue(self)` remove um elemento da fila e retorna-o. O método `top(self)` retorna uma cópia do valor do próximo elemento a ser removido na pilha, sem modificá-la. A posição onde este elemento estava é preenchida por `None`.

   Considere que em um dado momento, um objeto `f` desta classe contém os seguintes valores:

   `_dados`: $\{$`None`$, 2, 3, 4\}$

   `_indiceVaiSair`: 1

   `_indiceEntrou`: 3

   `_tamanho`: 3

   Mostre o conteúdo do objeto após a sequência de operações:

   ```
   f.enqueue(5)
   f.enqueue(6)
   f.dequeue()
   ```

   Resposta:

   1. `f.enqueue(5)`

      `_dados`: $\{5, 2, 3, 4\}$

      `_indiceVaiSair`: 1

      `_indiceEntrou`: 0

      `_tamanho`: 4

2. `f.enqueue(6)`

   Note que esta inserção duplica a sequência subjacente.

   `_dados`: $\{2, 3, 4, 5, 6, \texttt{None}, \texttt{None}, \texttt{None}\}$

   `_indiceVaiSair`: 0

   `_indiceEntrou`: 4

   `_tamanho`: 5

3. `f.dequeue()`

   Note que esta inserção duplica a sequência subjacente.

   `_dados`: $\{\texttt{None}, 3, 4, 5, 6, \texttt{None}, \texttt{None}, \texttt{None}\}$

   `_indiceVaiSair`: 1

   `_indiceEntrou`: 4

   `_tamanho`: 4

2. (2,0 pontos) Seja a seguinte função, $f(x)$ definida de inteiros positivos para inteiros positivos: Se x é par, $f(x) = x/2$. Se x é ímpar, $f(x) = 3x + 1$. Considere a sequência $\{a_0, a_1, \ldots, a_n, \ldots\}$ tal que $a_{i+1} = f(a_i)$. A conjectura de Collatz propõe que, não importa qual o valor de a_0 (desde que inteiro positivo), existe *sempre* um valor n finito tal que $a_n = 1$. A veracidade da conjectura de Collatz ainda é um problema em aberto na matemática.

   (a) (1,5 pontos) Escreva uma função que, para um dado a_0, calcula o menor valor de n tal que $a_n = 1$. Use a seguinte assinatura:

   `def collatz_n(a):`

   Onde `a` é um inteiro positivo com o valor a_0 de sua sequência. Sua função deve retornar o primeiro n tal que $a_n = 1$.

   Resposta:

```
def collatz_n(a):
    i = 0
    while a != 1:
        if a%2:
           a = 3*a + 1
        else:
           a //= 2
        i += 1
    return i
```

   (b) (0,5 pontos) O código que você escreveu corresponde a um algoritmo? Discuta.

   Resposta: O término em tempo finito da função acima implica na veracidade da conjectura de Collatz, o que pelo enunciado ainda é um problema em aberto. Assim, não se sabe se o código representa ou não um algoritmo.

3. (2,0 pontos) Escreva uma função em Python que permuta os elementos de um vetor de inteiros de modo que exista um índice i neste vetor tal que todos os elementos anteriores a este índice sejam negativos e nenhum elemento deste índice em diante o seja. A complexidade de sua função deve ser *linear*.

Use a seguinte assinatura:

`def arruma(v):`

Onde `v` é o vetor de inteiros em questão. Sua função deve modificar o conteúdo de `v`.

Resposta: A idéia é usar um procedimento similar ao do particionamento do quicksort:

```
def arruma(v):
    i = 0
    j = len(v)-1
    while i<=j:
        while i<=j and v[i]<0:
            i += 1
        while i<=j and v[j]>=0:
            j -= 1
        if i<j:
            v[i], v[j] = v[j], v[i]
```

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. 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.

   Escreva em Python uma função que, dadas duas árvores binárias de busca contendo inteiros, exibe o conteúdo da *interseccão* destas, ou seja, os inteiros que estão armazenados em *ambas*. Sua função deve ter complexidade $\mathcal{O}(M + N)$, onde M é a quantidade de elementos em uma árvore e N é a quantidade elementos em outra. Complexidades piores valem 1 ponto.

   Use a seguinte assinatura:

   `def mostra_interseccao(a, b):`

   Onde `a` e `b` são os nós raiz da classe `NoArvoreBinaria` das árvores binárias de busca que contém inteiros. Mostre o conteúdo comum usando a função `print()`.

Resposta: Uma enumeração em ordem *interior* de uma árvore binária de busca retorna uma ordenação crescente dos seus elementos. É possível, por exemplo, inserir em uma fila como a da classe `fila_circular` os elementos de uma árvore em ordem interior. Depois, enumera-se a outra árvore comparando-se os valores desta com os presentes na fila.

```
def mostra_interseccao(a, b):
    def enfileira(r, p):
        if r:
            enfileira(r.e, p)
            p.enqueue(r.x)
            enfileira(r.d, p)

    def mostra(r, p):
        if r:
            mostra(r.e, p)
            while len(p) and p.top() < r.x:
                p.dequeue()
```

```
            if len(p) and p.top()==r.x:
                print(r.x)
            mostra(r.d, p)

    p = fila_circular()
    enfileira(a, p)
    mostra(b, p)
```

5. (2,0 pontos) Uma sequência $v = \{v_0, v_1, \ldots, v_{n-1}\}$ é dita *unimodal* se exitem dois índices s e d com $s \leq d$ tal que a sequência é *crescente* de 0 a d (ou seja, $v_i \geq v_{i_1}$ para $i \leq s$) e decrescente de s a $n-1$, ou seja (ou seja, $v_i \geq v_{i_1}$ para $i \leq s$). Por exemplo, a sequência $\{0, 1, 2, 5, 8, 8, 7, 6, 6, 2\}$ é unimodal ($s = 4$, $d = 5$), enquanto que $\{0, 1, 2, 5, 8, 5, 7, 6, 6, 2\}$ não é.

   A função `max_unimodal(v, a, b):` procura por um máximo de uma sequência unimodal:

```
1  def max_unimodal(v, a, b):
2      l = b - a
3      if l <= 1:
4          return max(v[a], v[b])
5      if l == 2:
6          return max(v[a], v[a+1], v[b])
7      c = b - int((b-a)*r)
8      d = a + int((b-a)*r)
9      if v[c] > v[d]:
10         return max_unimodal(v, a, d)
11     else:
12         return max_unimodal(v, c, b)
```

   O parâmetro `v` contém uma sequência unimodal e os parâmetros `a` e `b` contém os índices entre os quais existe um máximo.

   A constante `r` é um número de ponto flutuante igual a 3/4.

   (a) (1.0 pontos) Mostre que o algoritmo está correto.

> **Resposta:** Esta é uma versão simplificada do algoritmo *Golden Search*. Mostra-se na versão completa que ao menos um dos índices c, d é reutilizado, o que *não* é feito aqui. Sejam a, b, c, d, r os valores das variáveis `a`, `b`, `c`, `d` e `r` respectivamente. O algoritmo busca por um máximo entre os índices a e b. O caso base é o caso em que $a - b \leq 2$. Neste caso, trivialmente, a resposta correta é o maior entre v_a, v_{a+1} e v_b. Nos outros casos o algoritmo calcula os índices c e d como:
>
> $$c = b - \left\lfloor (b-a)\frac{\sqrt{5}-1}{2} \right\rfloor$$
>
> $$d = a + \left\lfloor (b-a)\frac{\sqrt{5}-1}{2} \right\rfloor$$
>
> Para $a - b > 2$, $a < c < d < b$ Assim os intervalos (a, d) e (c, b) cobrem (com sobreposição) todos os índices da sequência. Ora, mas se $v_c > v_d$ então a sequência é necessariamente decrescente a partir de um determinado índice i com $i \leq d$. Assim, o ótimo está necessariamente entre

a e d e a chamada na linha 8 retorna a resposta correta. Caso $b = c$ (quando $a - b = 2$) Do mesmo modo, se $v_c <= v_d$, então a sequência é necessariamente crescente até um índice i com $i \geq c$. Neste caso, o ótimo está necessariamente entre c e b e a chamada na linha 10 retorna efetivamente o valor correto do problema.

(b) (1.0 pontos) Calcule a complexidade do algoritmo.

Resposta: O algoritmo chama a si mesmo em um intervalo com 3/4 do tamanho do original. Assim a complexidade é $\mathcal{O}(\log N)$.

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 `fila_circular`

```
class fila_circular:
    def __init__(self):
        self._dados = [None]*4
        self._indiceVaiSair = 0
        self._indiceEntrou=len(self._dados)-1
        self._tamanho = 0

    def __len__(self):
        return self._tamanho

    def _incrementa(self, indice):
        indice += 1
        if indice == len(self._dados):
            indice = 0
        return indice

    def _duplique_seq(self):
        novo = [None]*2*len(self._dados)
        n = self._tamanho
        for i in range(n):
            novo[i] = self.dequeue()
        self._dados = novo
        self._indiceVaiSair = 0
        self._indiceEntrou = n - 1;
        self._tamanho = n

    def enqueue(self, x):
       if self._tamanho == len(self._dados):
          self._duplique_seq()
       self._indiceEntrou = self._incrementa(self._indiceEntrou)
       self._dados[self._indiceEntrou] = x
       self._tamanho += 1

    def dequeue(self):
       if self._tamanho == 0:
          raise IndexError("Fila␣vazia")
       self._tamanho -= 1
       x = self._dados[self._indiceVaiSair]
       self._dados[self._indiceVaiSair] = None
       self._indiceVaiSair = self._incrementa(self._indiceVaiSair)
       return x

    def top(self):
       if self._tamanho == 0:
          raise IndexError("Fila␣vazia")
       return self._dados[self._indiceVaiSair]
```

Classe `NoArvoreBinaria`

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