Lista de Linguagens de Programação – 16

Nome: _____ Matrícula: _____

1. Abaixo existem duas implementações diferentes to Crivo de Erastótenes, um algoritmo bastante conhecido para a listagem de números primos:

```
def sievePrimes0(candidates):
  """Removes every non-prime number from the ordered list l."""
  for i in range(0, int(len(candidates) ** 0.5 + 1)):
    currentPrime = candidates[i]
    if currentPrime != 0:
      for j in range(i + currentPrime, len(candidates), currentPrime):
        if candidates[j] % candidates[i] == 0:
          candidates[j] = 0
  return filter(lambda x : x != 0, candidates)


def sievePrimes1(l):
  """Removes every non-prime number from the ordered list l."""
  if l[0] * l[0] <= l[len(l) - 1]:
    l[1:] = sievePrimes1(filter(lambda x: x % l[0] != 0, l))
  return l
```

   (a) Um destes algoritmos foi escrito segundo um estilo mais imperativo de programação, enquanto o outro foi escrito em um estilo mais declarativo. Qual é o algoritmo imperativo? Por que?

   (b) Qual destes algoritmos é, em sua opinião, o mais legível? Esta pergunta, obviamente, é subjetiva, e diferentes pessoas podem escolher diferentes respostas. Justifique a sua resposta, explicando quais razões te levam a preferir uma sintáxe em relação à outra.

(c) Qual implementação do crivo é mais eficiente? Tome medidas de tempo das duas implementações. Para isto, você pode usar a função `timeSieve` abaixo. Plote um gráfico com medidas de tempo para crivos de tamanho 10, 50, 100, 500, 1.000, 5.000, 10.000, 50.000, 100.000 e 500.000.

```
def timeSieve(maxPrime, numTimes, sieveFunc):
  """Times the sieve implementations."""
  t0 = 0
  for numTries in range(0, numTimes):
    taux = time.clock()
    primes0 = sieveFunc(range(2, maxPrime))
    t0 = t0 + time.clock() - taux
  print (t0 / numTimes)
```

(d) Quais razões poderiam explicar a diferença de eficiência observada na questão anterior? Existe alguma diferença entre a complexidade assimptótica dos dois algoritmos?

(e) Escreva uma função `testPrimes`, que torne mais interativo o processo de teste de eficiência. Esta função deve ler dois valores da entrada: um número `numTimes` de vezes que a função crivo deve ser chamada, e um número `maxPrime` que determina o tamanho do crivo gerado. A função `timeSieve` deverá ser chamada duas vezes, uma com `sievePrimes0` e outra com `sievePrimes1`, usando os parâmetros lidos da entrada. Dê uma olhada nas funções `raw_input` e `int`. Você pode querer usá-las neste exercício.

2. Python é uma das poucas linguagens de programação modernas em que a indentação é parte da sintáxe. A partir desta observação, responda as perguntas abaixo:

   (a) O que queremos dizer quando afirmamos que a indentação é parte da sintaxe da linguagem?

   (b) A maior parte das linguagens de programação mais conhecidas separa indentação de sintaxe. Esta não separação, em Python, foi uma decisão polêmica. Qual(is) a(s) vantagem(ns) e desvantagem(ns) da abordagem de Python?

3. A sintaxe de Python provê uma cláusula `else` para laços. Em geral, linguagens da família BCPL (C, C++, Java, C#) não contêm nada semelhante. O programa abaixo ilustra o uso deste tipo de `else`. O que a função `mystery` faz?

```
def mystery(limit):
  nums = []
  for n in range(2, limit):
    for x in range(2, n):
      if n % x == 0:
        break
    else:
      nums.append(n)
  return nums
```

4. Tal qual em SML, em Python nós não declaramos os tipos das variáveis manipuladas pelos programas Estas linguagens, contudo, lidam com tipos de maneira muito diferente.

   (a) SML é uma linguagem estaticamente tipada, ao contrário de Python, que é tipada dinamicamente. Qual a diferença entre tipagem estática e tipagem dinâmica?

   (b) Podemos fazer um punhado de estripulias em linguagens dinamicamente tipadas. Por exemplo, escreva uma função `pydiv(n, d)` em Python, que retorna o resultado da divisão do número real `n` pelo número real `d`. Entretanto, se `d` for zero, então o valor especial `None` deverá ser retornado. Note que `None` é um valor pré-definido em Python; de fato, é o valor retornado por todas as funções que não "retornam nada".

   (c) Como poderíamos implementar algo parecido em SML? Usando tipos algébricos, talvez? Escreva uma função `smldiv`, cujo tipo seja `real * real -> ???` que simule da forma mais fidedigna possível a função `pydiv`.

5. Escreva uma função `perm(n, l)`, que produza todas as permutações de `n` elementos da lista `l`. Por exemplo:

```
>>> perm(2, ["a", "b", "c"])
[['a', 'b'], ['a', 'c'], ['b', 'a'], ['b', 'c'], ['c', 'a'], ['c', 'b']]
>>>
>>> perm(3, [1, 2, 3])
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

6. Considere as seguintes sequências de números:

   - $\{0163, 0613, 1063\}$
   - $\{1487, 4817, 8147\}$

   Estas sequências são especiais por quatro razões.

   (a) Estes números são permutações de quatro dígitos distintos.

   (b) Estes números são primos.

   (c) Estes números são todos menores que 9999.

   (d) Cada sequência forma uma progressão aritmética. A primeira possui razão 450, e a segunda possui razão 3330.

   Existe uma outra sequência de três números que preenche estes quatro requisitos. Escreva um programa em Python que encontre tal sequência. Talvez você queria reutilizar a função `perm` do exercício anterior. Você não precisa re-escrever o código de `perm` em seu arquivo. Python provê um mecanismo de reúso de código via a primitiva `import`. Por exemplo, supondo que `perm` está em um arquivo `ex.py`, você pode torná-la visível em seu escopo local com a comando `from ex import perm`. Este problema é uma variação do problema 49 do Projeto Euler. Você pode querer dar uma olhadinha naquele problema, assim que resolver este aqui.