PROVA 2

MAC0119 INTRODUÇÃO À PROGRAMAÇÃO DE COMPUTADORES (ICB)

2º SEMESTRE DE 2025

Nome completo: _____

NUSP: _____

Instruções:

(*1*) Esta prova é individual.
(*2*) A prova consiste de **4** questões (contando a Questão 0 nesta página). Note que é possível tirar mais de 10 nesta prova `:-)`
(*3*) Para ter nota integral em uma questão, a escrita de sua solução deve estar boa.
(*4*) As respostas devem estar nos locais indicados.
(*5*) Não é permitido o uso de aparelhos eletrônicos de qualquer natureza.
(*6*) Não destaque as folhas deste caderno.
(*7*) Não use folhas avulsas para rascunho. Não é necessário apagar seus rascunhos.
(*8*) Não é permitido consultar colegas.
(*9*) O único material que você pode consultar é seu *cheat sheet*.

Assinatura:

Sua assinatura acima atesta a autenticidade e originalidade de seu trabalho e que você compromete-se a seguir o código de ética da USP em todas as suas atividades, incluindo esta prova.

Boa sorte!

Q	0	1	2	3	**Total**
Nota					

Q0. **[0.5 pontos]** Leia o conteúdo desta página e preencha os itens requisitados. Assine acima, e atente ao significado de sua assinatura.

Data: 2025/12/3, 1:06am

Rascunho

Q1. **[3.5 pontos]** Em cada item desta questão, é dado um trecho de código. Você deve dizer em cada item o que será impresso por esse trecho de código.

(*i*)
```
def power(x, y):
    if y == 0:
        return 1
    return x * power(x, y - 1)

print(power(3, 4))
```

Resposta:

(*ii*)
```
def gc_count(s):
    if s == '':
        return 0
    t = 0
    if s[0] in ['G', 'C']:
        t = 1
    return t + gc_count(s[1:])

print(gc_count('ACTAGGCTC'))
print(gc_count('ATATATTTATTA'))
```

Resposta:

(*iii*)
```
def explode(s):
    if s == '':
        return []
    return [s[0]] + explode(s[1:])

print(explode('MAC0119'))
```

Resposta:

(*iv*)
```
def remove_all(x, l):
    if l == []:
        return []
    if l[0] == x:
        return remove_all(x, l[1:])
    return [l[0]] + remove_all(x, l[1:])

print(remove_all(25, [25, 75, 25, 25, 77, 119]))
print(remove_all(25, [67, "please", 25, [25, "help!"]]))
```

Resposta:

(*v*)
```
def deep_remove_all(x, l):
    if l == []:
        return []
    if type(l[0]) == type([]):  # True se e só se l[0] é lista
        return [deep_remove_all(x, l[0])] + deep_remove_all(x, l[1:])
    t = [l[0]]
    if l[0] == x:
        t = []
    return t + deep_remove_all(x, l[1:])

print(deep_remove_all(25, [25, 67, 25, [24, 25, 26], [25]]))
print(deep_remove_all(55, [25, [1, [55, 48], [55], 55, 51], 55, 52]))
```

Resposta:

Q2. [**4 pontos**] Considere a seguinte função:

```
def no_occurrences(m, s):
    if m == '':
        return 1
    if s == '':
        return 0
    if m[0] != s[0]:
        return no_occurrences(m, s[1:])
    return no_occurrences(m, s[1:]) + no_occurrences(m[1:], s[1:])
```

Se `m` e `s` são dois strings, o valor devolvido por `no_occurrences(m, s)` é o número de ocorrências de `m` em `s` como subsequência. Por exemplo,

```
no_occurrences('ab', 'aabb')
```

vale 4. Entretanto, a função `no_occurrences()` acima é ineficiente: mesmo que `m` e `s` não sejam tão longos, a execução de `no_occurrences(m, s)` pode ser demorada. Por exemplo,

```
no_occurrences('a' * 15, 'a' * 30)
```

demora da ordem de uns dois minutos. Escreva uma versão memoizada da função acima. Sua função deve ter o seguinte cabeçalho:

```
def no_occurrences_memoR(m, s, memo):
```

onde `memo` é um dicionário. Sua função deve funcionar em conjunto com esta função:

```
def no_occurrences_memo(m, s):
    return no_occurrences_memoR(m, s, {})
```

De fato, com sua função `no_occurrences_memoR()`, a execução de

```
no_occurrences_memo('a' * 15, 'a' * 30)
```

deve terminar quase que instantaneamente. O comportamento esperado é como segue:

```
>>> no_occurrences('ab', 'aabb')
4
>>> no_occurrences_memo('ab', 'aabb')
4
>>> no_occurrences('a' * 15, 'a' * 30)
155117520  # depois de uns 2 minutos
>>> no_occurrences_memo('a' * 15, 'a' * 30)
155117520  # praticamente instantâneo
>>>
```

Escreva a seguir sua função `no_occurrences_memoR()` por completo.

Resposta:

Rascunho

Q3. [**4 pontos**] Considere a seguinte função:

```
def subsequence_indices(m, s):
    if m == '':
        return []
    if s == '':
        return 'not subsequence'
    if m[-1] != s[-1]:
        return subsequence_indices(m, s[:-1])
    if subsequence_indices(m[:-1], s[:-1]) == 'not subsequence':
        return 'not subsequence'
    return subsequence_indices(m[:-1], s[:-1]) + [len(s) - 1]
```

Suponha que `m` seja uma subsequência de `s`. A função acima determina uma lista de índices dos caracteres de `s` que determinam uma subsequência de `s` igual a `m`. Seguem exemplos de execuções desta função:

```
>>> subsequence_indices('', 'xyz')
[]
>>> subsequence_indices('abc', 'xaxxbcx')
[1, 4, 5]
>>> subsequence_indices('abc', 'xaaxxbbccx')
[2, 6, 8]
>>> subsequence_indices('abcd', 'xaaxxbbccx')
'not subsequence'
>>>
```

Escreva uma função com cabeçalho

```
def subsequence_align(m, s):
```

que exibe a saída de `subsequence_indices()` de forma mais "visual". Mais precisamente, considere a seguinte função:

```
def subsequence_align_print(m, s):
    print('m =', m)
    print('s =', s)
    print('   ', subsequence_align(m, s))
```

Sua função `subsequence_align()` deve ser tal que, em conjunto com a função `subsequence_align_print()` acima, temos o comportamento ilustrado a seguir:

```
>>> subsequence_align_print('', 'xyz')
m =
s = xyz
    ---
>>> subsequence_align_print('abc', 'xaxxbcx')
m = abc
s = xaxxbcx
    -a--bc-
>>> subsequence_align_print('abc', 'xaaxxbbccx')
m = abc
s = xaaxxbbccx
    --a---b-c-
>>> subsequence_align_print('abcd', 'xaaxxbbccx')
m = abcd
s = xaaxxbbccx
    not subsequence
>>>
```

Escreva a seguir sua função `subsequence_align(m, s)` por completo.

Resposta:

Rascunho:

Rascunho: