TD

qkzk

Terminaison

Exercice 1

On dit qu'un programme *termine* s'il s'exécute correctement jusqu'au bout et s'arrête ou s'il plante (*lève une exception*).

Il ne **termine pas** s'il entre dans une boucle infinie.

Les programmes suivants terminent-ils ?

1. programme 1

```
n = 10
while n < 100:
    n = 2 * n
```

2. programme 2

```
while n < 100:
    n = 2 * n
```

3. programme 3

```
n = 0
while n < 100:
    n = 2 * n
```

4. programme 4

```
capital = 0
while capital < 1000:
    capital = 1.2 * capital
```

5. programme 5

```
capital = 10
while capital < 1000:
    capital = 1.2 * capital
```

Exercice 2

Citer la plage de valeurs pour lesquelles ces fonctions terminent et celle pour lesquelles elles ne terminent pas

*Ces exemples de code sont **volontairement** mal écrits. Ne vous en inspirez pas et ne les exécutez pas !*

1. fonction 1.

```
def compte_voyelle(mot: str) -> int:
    """Renvoie le nombre de voyelles dans mot"""
    voyelles = "aeiouy"
    i = 0
    compteur = 0
    while i < len(mot):
        lettre = mot[i]
        if lettre in voyelles:
            compteur += 1
    return compteur
```

2. fonction 2

```
def compte_a_rebour(n: int) -> None:
    """Affiche un compte à rebours"""
    while n != 0:
        print(n)
        n -= 1
```

3. fonction 3

```
def compte_a_rebour(n: int) -> None:
    """Affiche un compte à rebours"""
    while n > 0:
        print(n)
        n -= 1
```

Annales du baccalauréat

Ce premier sujet revient sur la calculabilité et le problème de l'arrêt et montre qu'il ne suffit pas de chercher après le mot clé "while" dans un code Python pour déterminer qu'une fonction termine. Il aborde aussi l'algorithme de Boyer-Moore... Pas simple !

25-NSIJ1JA1 : 2025, Asie, Jour 1, Exercice 1 (6 points)

Cet exercice porte sur la décidabilité, l'algorithmique et la programmation en Python.

En Python, on peut utiliser le triple guillemet pour écrire une chaîne de caractères sur plusieurs lignes. Par exemple, on peut définir une variable `programme1` qui contient la chaîne de caractères correspondant à un petit programme Python de la manière suivante :

```
programme1 = """
x = 10
y = 10
while x > 0:
    x = x - 1
    y = y + 1
"""
```

De même, on peut définir la variable `programme2` :

```
programme2 = """
def boucle_infinie():
    while True:
        pass  # Ne rien faire
boucle_infinie()
"""
```

1. On suppose que l'on exécute le programme contenu dans la variable `programme1`. Donner les valeurs de `x` et de `y` après exécution de ce programme.
2. Expliquer pourquoi tout programme Python peut être vu comme une chaîne de caractères.

En Python, la fonction `exec` permet d'exécuter le programme correspondant à une chaîne de caractères passée en paramètre.

```
>>> exec("r = 42")
>>> r
42
>>> exec(programme1)
>>> x + y
20
>>> exec(programme2)
[ne termine pas]
```

On considère les quatre variables `programme3`, `programme4`, `programme5`, `programme6` suivants :

```
programme3 = """
x = 10
while x != 0:
    x = x - 2
"""

programme4 = """
x = 10
while x > 0:
    x = x + 2
"""

programme5 = """
x = 10
while x < 0:
    x = x + 4
"""

programme6 = """
x = 10
while x != 0:
    x = x - 4
"""
```

3. On exécute les variables `programme3`, `programme4`, `programme5`, `programme6` avec la fonction `exec`. Déterminer lesquelles terminent et lesquelles ne terminent pas.

On cherche à écrire une fonction `arret` telle que `arret(programme)` renvoie `True` si `exec(programme)` termine et `False` si `exec(programme)` ne termine pas. Cette fonction `arret` doit donc s'arrêter dans tous les cas.

4. Indiquer ce que réalise le programme suivant et s'il permet de répondre au problème posé ci-dessus.

```
def arret_essai1(programme):
    exec(programme)
    return True
```

On suppose disposer d'une fonction `recherche(mot, texte)` qui renvoie `True` si une chaîne de caractères `mot` est présente dans une chaîne de caractères `texte` et `False` sinon.

5. Expliquer succinctement le principe de l'algorithme de Boyer-Moore qui permet d'implémenter cette fonction `recherche`.

Une idée est d'écrire une fonction qui décrète qu'un programme s'arrête s'il ne contient pas de `while` et ne s'arrête pas s'il en contient un.

6. Écrire une fonction `arret_essai2(programme)` qui renvoie `True` si la chaîne de caractères `"while"` n'est pas utilisée dans la chaîne de caractères `programme` et `False` sinon.

7. Montrer qu'il est possible que :

- `arret_essai2(programme)` renvoie `True` alors que le programme ne s'arrête pas ;
- `arret_essai2(programme)` renvoie `False` alors que le programme s'arrête.

Indication : il n'y a pas que les boucles `while` qui peuvent poser des problèmes de non terminaison.

Nos tentatives pour écrire une telle fonction `arret` sont restées vaines. Nous allons montrer qu'il est en réalité impossible d'écrire une telle fonction. On va supposer qu'une telle fonction `arret` existe. On va montrer que cette supposition aboutit à un paradoxe ce qui prouvera que la supposition est fausse.

8. Écrire une fonction `terminaison_inverse` telle que l'appel `terminaison_inverse(programme)` termine si la chaîne de caractères `programme` représente un programme qui ne termine pas et ne termine pas si la chaîne de caractères `programme` représente un programme qui termine. On pourra utiliser la fonction `boucle_infinie` de `programme2` ainsi bien sûr que la fonction `arret` dont on a supposé l'existence.

On considère `"terminaison_inverse(programme_paradoxal)"` qui n'est rien d'autre qu'une chaîne de caractères, et on définit une variable que l'on appelle `programme_paradoxal` à laquelle on affecte cette chaîne de caractères :

`programme_paradoxal = "terminaison_inverse(programme_paradoxal)"`

9. Étudier si le programme paradoxal termine ou non, c'est-à-dire si `exec(programme_paradoxal)` termine ou non.
10. Indiquer ce que l'on peut conclure sur la fonction `arret`.
11. Expliquer si l'impossibilité d'écrire une telle fonction `arret` est due aux limitations du langage Python.

Remarque

Le sujet stipule "il n'y a pas que les boucles while qui peuvent poser des problèmes de non terminaison"

Voici quelques candidats sans while et le résultat obtenu lors d'une exécution avec Python classique (pas certain de ce qui se passe avec d'autres versions de l'interpréteur) :

Récursion infinie

```
def f():
    return f()

f()
```

s'arrête avec `RecursionError: maximum recursion depth exceeded`. Donc elle termine en pratique.

Appels cycliques indirects

```
def a():
    b()

def b():
    a()

a()
```

même résultat : termine avec `RecursionError: maximum recursion depth exceeded`.

Boucles for avec des itérateurs (HP)

```
def inf():
    while True:
        yield 1

for x in inf():
    pass  # Boucle infinie sans "while"
```

Ne termine pas, équivalent à `while True`. Dans l'état ça ne sert à rien.

Variante :

```
from itertools import count

for x in count():
    pass
```

Blocage sur une ressource ou dans une attente :

```
input("Tapez quelque chose : ")  # Attend potentiellement pour toujours
```

Ne termine pas sans intéraction.

```
import time
while True:
    time.sleep(1)
```

La même que `while True: pass` mais va consommer moins de ressources...

Importation circulaire

```
exec('exec("exec(\'...\')")')  # Peut être instrumenté en boucle infinie
```

On peut construire le même genre de programme avec `import`, `eval` ou `exec`...

Ne termine pas en théorie. Pas pris le risque de l'exécuter sur ma machine... désolé.

Multithreading En autorisant plusieurs fils d'exécution... alors il est facile de créer des boucles infinies sans while.