



1

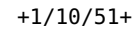
Ens. : O. Lévêque, M. Stojilović
Information, Calcul, Communication - A
Vendredi 19 décembre 2025
Durée : 180 minutes

SCIPER: 0

Salle: INF 1

Attendez le début de l'épreuve avant de tourner la page. Ce document est imprimé recto-verso, il contient 20 pages, les dernières pouvant être vides. Ne pas dégrafer.

- Posez votre carte d'étudiant sur la table.
- Document autorisé pour cet examen : un formulaire constitué de deux pages A4 recto-verso, manuscrites, préparées avec styler+tablette ou à l'ordinateur.
- Seule l'utilisation d'une calculatrice simple (de type TI-30 eco RS) est autorisée pendant l'examen. Tout autre appareil électronique (ordinateur, smartphone/watch, tablette) est interdit.
- L'examen est composé uniquement de questions de type ouvert, valant en tout 40 points.
- Merci d'avance de soigner la présentation de vos réponses !
- Laissez libres les cases à cocher : elles sont réservées aux correcteurs.
- Si une question est erronée, les enseignants se réservent le droit de l'annuler.



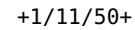
0 .5 1 .5

Fournissez uniquement la réponse demandée ; aucune étape intermédiaire n'est nécessaire. La totalité des points ne sera attribuée que si votre réponse correspond exactement à ce qu'affiche le programme dans le terminal. Donner uniquement la valeur calculée ne suffit pas.

```
['a', 'b', 'e', 'l', 'n', 'r', 't']
['a', 'b', 'e', 'n', 'r', 't', 'u']
['a', 'b', 'n']
```

0 .5 1 .5

Pour votre examen, imprimez de préférence les documents compilés à l'aide de auto-multiple-choice.



{1: 1, 2: 2, 3: -3}

0 .5 1 .5

11
18



Question 8: Cette question est notée sur 8.5 points.

	0			.5		1			.5		2			.5		3			.5		4
	.5		5			.5		6			.5		7			.5		8			.5

De nombreux numéros d'identification (par exemple les numéros de carte de crédit) utilisent une *checksum* pour détecter des erreurs de frappe. L'*algorithme de Luhn* est une méthode simple pour calculer une telle checksum. Dans cet exercice, nous utilisons l'algorithme de Luhn sur une chaîne de chiffres `s`, comme suit :

- On part du chiffre le plus à droite de `s` et on se déplace vers la gauche.
- On numérote les positions à partir de 0 :
 - chiffre le plus à droite : position 0,
 - chiffre suivant vers la gauche : position 1, etc.
- Pour chaque chiffre :
 - si sa position est paire, on conserve le chiffre tel quel ;
 - si sa position est impaire, on double le chiffre ; si le résultat est ≥ 10 , on lui soustrait 9.
- On additionne tous ces nombres. On note cette somme **total**.
- La checksum est simplement le reste de la division entière de **total** par 10.

Vous pouvez supposer que toutes les chaînes ne contiennent que des caractères chiffres ("0"–"9").

Exemple : Soit `s = "1057"`. On numérote les positions en partant de la droite : "7" est en position 0 (paire, on garde le chiffre 7), "5" est en position 1 (impaire, on le double à 10 puis, comme le résultat est ≥ 10 , on lui soustrait 9 et on obtient 1), "0" est en position 2 (paire, on garde le chiffre 0), et "1" est en position 3 (impaire, on le double à 2). La somme vaut donc $7 + 1 + 0 + 2 = 10$, puis la checksum vaut $10 \bmod 10 = 0$; lorsque la checksum est égale à zéro, le numéro est considéré comme *valide*. À titre de comparaison, si l'on numérote les positions en partant de la gauche, on obtient une somme 11 et une checksum $11 \bmod 10 = 1$, ce qui indiquerait (à tort) que le numéro n'est pas valide.

Considérez le dictionnaire suivant `id_numbers`, qui contient les numéros d'identification pour lesquels nous voulons calculer la checksum.

```
1 id_numbers = {
2     "Amelie":    "12344",
3     "Benoit":    "3456780",
4     "Chloe":     "79920",
5     "Dominique": "567891",
6     "Elodie":    "810239",
7 }
```

Pour chaque sous-question où l'on vous demande d'écrire une fonction, vous devez fournir la définition complète de la fonction, et non seulement le corps. Le non-respect des consignes spécifiques d'implémentation indiquées ci-dessous (par exemple l'utilisation des fonctions imposées ou des constructions demandées) entraînera l'attribution d'au plus 50% des points prévus pour la question concernée, voire moins selon le nombre et la nature des écarts par rapport aux consignes.

(a) `string_to_digits`

Écrivez une fonction `string_to_digits` qui prend en argument une chaîne `number` et qui renvoie une liste d'entiers correspondant aux chiffres de `number`.

Vous devez construire cette liste à l'aide d'une compréhension de liste. L'ordre des chiffres dans votre liste doit être le même que dans la chaîne d'origine. Vous ne devez pas utiliser d'indexation du type `number[i]` et vous ne devez pas appeler `list(number)`.

Exemple :

```
1 string_to_digits("12344") # returns [1, 2, 3, 4, 4]
```



(b) luhn_checksum

(b1.) Écrivez une fonction `luhn_checksum` qui prend en argument une chaîne `number`, appelle la fonction `string_to_digits` pour obtenir la liste des chiffres, calcule la Luhn checksum sur cette liste de chiffres, et renvoie la checksum sous la forme d'un entier entre 0 et 9.

Vous devez utiliser la fonction `string_to_digits` à l'intérieur de `luhn_checksum` pour convertir la chaîne en liste de chiffres.

(b2.) Montrez, étape par étape, que votre fonction calcule correctement la checksum du nombre "1057", comme dans le premier exemple.

Exemples :

```
1 luhn_checksum("1057") # returns 0
2 luhn_checksum("12344") # returns 0
3 luhn_checksum("3456780") # returns 3
```

(c) is_valid

Un numéro est *valide* si sa Luhn checksum est égale à 0. Écrivez une fonction `is_valid` qui prend en argument une chaîne `number`, appelle `luhn_checksum` et renvoie `True` si la checksum vaut 0, et `False` sinon.

Vous devez utiliser la fonction `luhn_checksum` à l'intérieur de `is_valid`.

Exemples :

```
1 luhn_checksum("1057") # returns True
2 is_valid("12344") # returns True
3 is_valid("3456780") # returns False
```

(d) fill_checksums

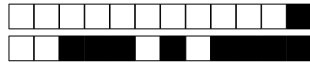
Nous voulons maintenant un second dictionnaire `checksums` qui ait les mêmes clés que `id_numbers`, mais dont les valeurs stockent la checksum et la validité de chaque numéro.

Écrivez une fonction `fill_checksums` qui construit et renvoie un nouveau dictionnaire `checksums` ayant les mêmes clés que `id_numbers`. Pour chaque nom, la valeur associée doit être un couple *checksum, valid* (un *tuple en anglais*) dont la première composante est la checksum et la seconde un booléen (`True` si le numéro est valide, `False` sinon).

Vous devez utiliser les fonctions `luhn_checksum` et `is_valid` à l'intérieur de `fill_checksums`.

Une implémentation correcte de `fill_checksums` doit produire :

```
{
  "Amelie": (0, True),
  "Benoit": (3, False),
  "Chloe": (9, False),
  "Dominique": (0, True),
  "Elodie": (5, False),
}
```



```
def string_to_digits(number):
    """
    (a) Return a list of integers corresponding to the digits of `number`.
    Uses a list comprehension and iteration over characters with `in`.
    """
    return [int(ch) for ch in number]

def luhn_checksum(number):
    """
    (b) Compute the Luhn checksum of the digit string `number`.
    Returns an integer between 0 and 9.
    """
    digits = string_to_digits(number)

    total = 0

    n = len(digits)

    # We want to process digits from right to left.
    # We use `i` to count how far we are from the right:
    # i = 0 for rightmost digit, 1 for next, etc.
    for i in range(n):
        digit = digits[n - 1 - i] # take digits from right to left

        # For Luhn: every second digit from the right is doubled.
        # Here we double when i is odd (1, 3, 5, ...).
        if i % 2 == 1: # odd positions: double and maybe subtract 9
            digit = digit * 2
            # If doubling gives at least 10, subtract 9 (Luhn rule).
            if digit >= 10:
                digit = digit - 9

        # Add the processed digit to the running total.
        total = total + digit

    # The checksum is the last digit of the total (total mod 10).
    return total % 10
```

(b2) Pour number = "1057" :

- Conversion en chiffres:
digits = string_to_digits("1057") = [1, 0, 5, 7]
total = 0, n = 4
- Boucle (on parcourt de droite à gauche avec digit = digits[n-1-i])

i = 0: digit = digits[3] = 7 (position la plus à droite)
i % 2 == 1 ? non → pas doublé
total = 0 + 7 = 7

i = 1: digit = digits[2] = 5
i % 2 == 1 ? oui → doublé : digit = 5 * 2 = 10
digit >= 10 ? oui → digit = 10 - 9 = 1
total = 7 + 1 = 8

i = 2: digit = digits[1] = 0
i % 2 == 1 ? non → pas doublé
total = 8 + 0 = 8

i = 3: digit = digits[0] = 1
i % 2 == 1 ? oui → doublé : digit = 1 * 2 = 2
digit >= 10 ? non → rien à soustraire
total = 8 + 2 = 10

Checksum final: La fonction renvoie total % 10 = 10 % 10 = 0.



```
def is_valid(number):  
    """  
    (c) Return True iff the Luhn checksum of `number` is 0.  
    """  
    # A number is valid if its Luhn checksum is 0.  
    return luhn_checksum(number) == 0  
  
def fill_checksums(id_numbers):  
    """  
    (d) Build and return a new dictionary `checksums` that has the same keys  
    as `id_numbers`. For each name, the value is a tuple (checksum, valid),  
    where:  
        - checksum is the Luhn checksum of the number  
        - valid is True if the number is valid according to Luhn, False otherwise  
    """  
    # Start from an empty dictionary that we will fill and then return  
    checksums = {}  
  
    # Loop over all (name, number) pairs in the id_numbers dictionary  
    for name, number in id_numbers.items():  
        # Compute the checksum of the current number using the helper function  
        cks = luhn_checksum(number)  
  
        # Compute whether the current number is valid using the helper function  
        valid = is_valid(number)  
  
        # Store the pair (checksum, valid) in the checksums dictionary  
        # under the corresponding name. This pair is a tuple in Python.  
        checksums[name] = (cks, valid)  
  
    # Return the fully built checksums dictionary  
    return checksums
```



Question 9: Cette question est notée sur 7 points.

☐	0	☐	.5	☐	1	☐	.5	☐	2	☐	.5	☐	3	☐	.5
☐	4	☐	.5	☐	5	☐	.5	☐	6	☐	.5	☐	7	☐	

On dit qu'une liste est un *palindrome* si elle se lit de la même façon de gauche à droite et de droite à gauche. Par exemple, la liste `[1, 2, 3, 2, 1]` est un palindrome, alors que `[1, 2, 3, 4]` n'en est pas un.

Vous pouvez supposer que les éléments de la liste peuvent être comparés à l'aide de l'opérateur d'égalité `==` ou de l'inégalité `!=`, et que la liste peut contenir un nombre d'éléments pair ou impair.

Pour chaque sous-question où l'on vous demande d'écrire une fonction, vous devez fournir la définition complète de la fonction, et non seulement le corps. Le non-respect des consignes spécifiques d'implémentation indiquées ci-dessous (par exemple l'utilisation des fonctions imposées ou des constructions demandées) entraînera l'attribution d'au plus 50% des points prévus pour la question concernée, voire moins selon le nombre et la nature des écarts par rapport aux consignes.

(a) `is_palindrome_iter`

(a1.) Écrivez une fonction Python *non récursive* `is_palindrome_iter` qui prend en argument une liste `lst` et renvoie `True` si et seulement si `lst` est un palindrome, et `False` sinon.

Vous pouvez utiliser des boucles, l'indexation et des fonctions prédéfinies simples (par exemple `len`, `range`). En revanche, vous ne devez pas utiliser de mécanisme qui renverse directement la liste, comme *list slicing*, la fonction `reversed`, ou d'autres constructions équivalentes qui comparent `lst` à une version renversée d'elle-même. Vous ne devez pas modifier `lst`.

(a2.) Expliquez en détail comment l'exécution de votre fonction se déroule sur l'exemple concret `lst = ['a', 1, 'b', 1, 'a']`. Décrivez quelles opérations sont effectuées et pourquoi le résultat final est `True` ou `False`.

(a3.) Votre fonction fonctionne-t-elle pour une liste contenant un nombre pair d'éléments ? Justifiez votre réponse.

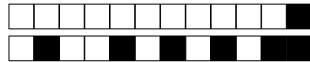
(b) `is_palindrome_recursive`

(b1.) Écrivez maintenant une fonction Python *récursive* `is_palindrome_recursive` qui prend en entrée une liste `lst` et un indice `i` (valeur par défaut `0`) correspondant à la position d'un élément de la liste. La fonction doit renvoyer `True` si et seulement si `lst` est un palindrome, et `False` sinon.

En utilisation normale, la fonction sera appelée sous la forme `is_palindrome_recursive(lst)` ; le paramètre `i` est utilisé en interne par la récursion.

Votre solution ne doit pas utiliser de *list slicing*, ni de boucles, et ne doit pas tenter de construire une copie inversée de la liste. Vous ne devez pas modifier `lst`.

(b2.) Expliquez en détail comment l'exécution de votre fonction se déroule sur le même exemple `lst = ['a', 1, 'b', 1, 'a']`. Décrivez chaque appel de la fonction récursive, comment les arguments évoluent, et à quel moment on atteint un cas de base.



```
def is_palindrome_iter(lst):  
    """  
    Return True if and only if lst is a palindrome, False otherwise.  
    Iterative (non-recursive) version without using reverse operations.  
    """  
    left = 0  
    right = len(lst) - 1  
  
    # Move two pointers towards the centre  
    while left < right:  
        if lst[left] != lst[right]:  
            return False  
        left += 1  
        right -= 1  
  
    # If we never found a mismatch, it's a palindrome  
    return True
```

(a2) La fonction `is_palindrome_iter(lst)` vérifie si une liste est un palindrome, c'est-à-dire si elle est identique lorsqu'on la lit de gauche à droite ou de droite à gauche. Pour faire ça sans inversion de liste, elle utilise deux indices : `left` qui commence au début de la liste (index 0) et `right` qui commence à la fin (index `len(lst)-1`). À chaque tour de boucle, elle compare `lst[left]` et `lst[right]`. Si les deux éléments sont différents, la liste ne peut pas être un palindrome, donc la fonction renvoie immédiatement `False`. Sinon, elle rapproche les deux indices vers le centre (`left += 1` et `right -= 1`) et continue.

Prenons `lst = ['a', 1, 'b', 1, 'a']`. Au départ, `left = 0` et `right = 4` (car la liste a 5 éléments). La condition de boucle est `left < right`, donc `0 < 4` est vrai et on entre dans la boucle. On compare alors `lst[0]` et `lst[4]`, c'est-à-dire `'a'` et `'a'`. Ils sont égaux, donc on ne renvoie pas `False`. On avance ensuite : `left` devient 1 et `right` devient 3.

On vérifie à nouveau la condition : `1 < 3` est vrai, donc on continue. On compare `lst[1]` et `lst[3]`, donc `1` et `1`. Ils sont égaux, donc on continue encore, et on met à jour les indices : `left = 2` et `right = 2`.

À ce stade, la condition `left < right` devient `2 < 2`, ce qui est faux. On sort donc de la boucle. Comme aucune comparaison n'a trouvé de différence, la fonction arrive à la fin et renvoie `True`. Intuitivement, c'est logique : toutes les paires symétriques correspondent (`'a'` avec `'a'`, puis `1` avec `1`) et l'élément central (`'b'`) n'a pas besoin d'être comparé à un autre élément, puisqu'il est « au milieu » et n'affecte pas la symétrie.

(a3) Concernant une liste avec un nombre pair d'éléments, la fonction fonctionne aussi. La seule différence est qu'il n'y a pas d'élément central unique : les deux indices finissent par se croiser après avoir comparé toutes les paires nécessaires. Par exemple, avec `[1, 2, 2, 1]`, on compare d'abord les extrêmes (1 et 1), puis les éléments suivants (2 et 2). Ensuite `left` devient plus grand que `right`, la boucle s'arrête, et comme aucune différence n'a été trouvée, la fonction renvoie `True`. Le critère d'arrêt `left < right` est justement ce qui garantit qu'on compare exactement les paires symétriques, que la longueur soit paire ou impaire.



```
def is_palindrome_recursive(lst, i=0):  
    """  
    Recursively checks if lst is a palindrome.  
    Parameter i is the current index from the left.  
    The symmetric index from the right is j = len(lst) - 1 - i.  
    """  
    # j is the index of the symmetric element from the right  
    j = len(lst) - 1 - i  
  
    # Base case: indices have met or crossed (0 or 1 element left to check)  
    if i >= j:  
        return True  
  
    # If elements at symmetric positions are different, it's not a palindrome  
    if lst[i] != lst[j]:  
        return False  
  
    # Recursive step: move one step inward from the left (i+1)  
    return is_palindrome_recursive(lst, i + 1)
```

(b2) La fonction `is_palindrome_recursive(lst, i=0)` vérifie récursivement si `lst` est un palindrome. L'idée est la même que dans la version itérative, mais au lieu de déplacer deux pointeurs dans une boucle, on déplace un seul index `i` depuis la gauche, et on calcule à chaque appel l'index symétrique `j` à droite avec la formule

$$j = \text{len}(\text{lst}) - 1 - i$$

À chaque appel, on compare `lst[i]` et `lst[j]`. Si ces deux valeurs diffèrent, la fonction renvoie immédiatement `False`. Si elles sont égales, on "avance vers le centre" en appelant la fonction avec `i + 1`. Le moment où la fonction sait qu'elle a fini est géré par le cas de base : quand les indices se rencontrent ou se croisent (`i >= j`), il n'y a plus de paire à comparer, donc on renvoie `True`.

Prenons `lst = ['a', 1, 'b', 1, 'a']`. La longueur vaut 5.

Appel 1 : `is_palindrome_recursive(lst, i=0)`

On calcule $j = 5 - 1 - 0 = 4$.

On teste le cas de base : $i \geq j$? donc $0 \geq 4$ est faux, on n'est pas au cas de base.

On compare `lst[0]` et `lst[4]`, soit 'a' et 'a'. Ils sont égaux, donc on continue.

On fait l'appel récursif suivant : `is_palindrome_recursive(lst, 1)`.

Appel 2 : `is_palindrome_recursive(lst, 1)`

On calcule $j = 5 - 1 - 1 = 3$.

Cas de base : $1 \geq 3$ est faux, donc on continue.

Comparaison : `lst[1]` et `lst[3]` vaut 1 et 1. Ils sont égaux.

Nouvel appel : `is_palindrome_recursive(lst, 2)`.

Appel 3 : `is_palindrome_recursive(lst, 2)`

On calcule $j = 5 - 1 - 2 = 2$.

Cas de base : $i \geq j$? ici $2 \geq 2$ est vrai. On atteint un cas de base : les indices se rencontrent sur l'élément central.

On renvoie donc `True` (il n'y a plus rien à comparer).

Ensuite, ce `True` remonte dans la pile d'appels :

- l'appel 2 recevait le résultat de l'appel 3, donc il renvoie `True`,

- puis l'appel 1 reçoit à son tour `True` et renvoie `True`.

C'est pour ça que le résultat final est `True` : toutes les comparaisons symétriques ont réussi, et on a atteint le cas de base $i \geq j$ sans jamais rencontrer de mismatch.