NSI Terminale — Récursivité

Sujet d'entraînement

Durée : 1 h 50 · Partie A sans ordinateur ni calculatrice · Partie A sur papier (≈ 55 min), partie B sur machine (≈ 55 min). Total sur 20 points.

Nom : ..... Prénom : ..... Classe : .....

Les fonctions demandées sont récursives : aucune boucle for ou while. Sur papier, une petite erreur de syntaxe n'est pas pénalisée si l'idée est juste.

PARTIE A — SUR PAPIER (10 POINTS)

Exercice 1 — Dérouler des fonctions récursives (4 points)

1. On considère la fonction suivante. (2,5 points)

```
def motif(n: int) -> None:
    if n == 0:
        return
    motif(n - 1)
    print(n)
    motif(n - 1)
```

a) Quel est le cas de base ? Que se passe-t-il si l'on appelle `motif(-1)` ?

b) Écrire, dans l'ordre, les nombres affichés par `motif(3)`.

c) Pendant l'exécution de `motif(3)`, combien d'appels de `motif`, au plus, sont en cours en même temps dans la pile ?

d) On note $L(n)$ le nombre de lignes affichées par `motif(n)`. Justifier que $L(0)=0$ et $L(n)=2\times L(n-1)+1$. En déduire $L(4)$, puis une formule pour $L(n)$. Quel problème du cours conduit à la même relation ?

2. On considère les deux fonctions suivantes, où n est un entier positif. (1,5 point)

```
def h(n: int) -> int:
    if n < 10:
        return n
    return h(n // 10) + n % 10

def racine(n: int) -> int:
    if n < 10:
        return n
    return racine(h(n))
```

a) Que calcule la fonction h ? Recopier et compléter le tableau, en détaillant le calcul de `racine(98765)`.

n	7	38	999	98765
racine(n)				

b) Pourquoi l'appel `racine(n)` se termine-t-il toujours ?

Exercice 2 — Listes et chaînes (3,5 points)

1. La fonction suivante doit renvoyer la liste privée de toutes les occurrences de `x`. Recopier et compléter les trois lignes. (0,5 point)

```
def supprime(x: int, liste: list) -> list:
    if liste == []:
        return ..........                        # ligne A
    elif liste[0] == x:
        return ..............................    # ligne B
    else:
        return ..............................    # ligne C
```

Exemple : `supprime(2, [2, 5, 2, 2, 7])` renvoie `[5, 7]`.

2. Écrire une fonction récursive `est_palindrome(mot, i, j)` qui renvoie `True` si la partie de `mot` comprise entre les indices `i` et `j` (inclus) se lit de la même façon dans les deux sens. On n'utilise **aucune tranche** (pas de `mot[a:b]`). Quel appel permet de tester le mot entier ? (1 point)

3. Écrire une fonction récursive `maximum(liste, i)` qui renvoie le plus grand élément de la liste parmi ceux d'indice `i` ou plus (on suppose `0 <= i < len(liste)`), sans tranche et sans la fonction `max`. (1 point)

4. Cette fonction devait renvoyer la longueur d'une liste. Expliquer pourquoi elle provoque une `RecursionError` pour toute liste non vide, puis la corriger. (0,5 point)

```
def longueur(liste: list) -> int:
    if liste == []:
        return 0
    return 1 + longueur(liste[0:])
```

5. Pourquoi les versions des questions 2 et 3, qui travaillent avec des indices, sont-elles plus économes que des versions utilisant des tranches comme `mot[1:-1]` ou `liste[1:]` ? (0,5 point)

Exercice 3 — Puissance rapide et tours de Hanoï (2,5 points)

```
def puiss(x: int, n: int) -> int:
    if n == 0:
        return 1
    elif n % 2 == 0:
        return puiss(x * x, n // 2)
    else:
        return x * puiss(x, n - 1)
```

1. Écrire la suite des appels déclenchés par `puiss(3, 13)`. Sans calculer sa valeur, montrer que le résultat est bien 3^{13}. (1 point)

2. Combien d'appels effectue `puiss(3, 13)` ? Combien en effectuerait la version simple, qui renvoie `x * puissance(x, n - 1)` ? Justifier que `puiss` se termine. (1 point)

3. Pour les tours de Hanoï, le nombre de déplacements vérifie $d(1)=1$ et $d(n)=2\times d(n-1)+1$. Combien de déplacements faut-il pour 10 disques ? (0,5 point)

PARTIE B — SUR MACHINE (10 POINTS)

Vous travaillez dans le fichier `recursivite_entrainement_eleve.py`. Complétez les fonctions marquées « À COMPLÉTER » et exécutez le fichier : la fonction `tests()` indique, pour chaque fonction, si les tests sont réussis. Les valeurs relevées et les explications se rédigent sur la copie.

Exercice 4 — Bases de numération et recherche (3,5 points)

1. Écrire la fonction récursive `en_base(n, b)` qui renvoie l'écriture de l'entier positif n en base b (entre 2 et 16), à l'aide de la chaîne `CHIFFRES = "0123456789ABCDEF"` du fichier. Les fonctions `bin`, `hex`, `oct` et `format` sont interdites. (1,5 point)

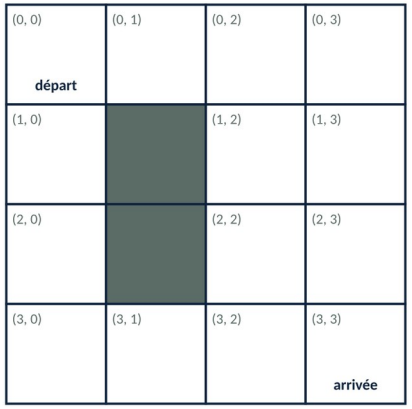
Exemples : `en_base(13, 2)` renvoie `"1101"`, `en_base(255, 16)` renvoie `"FF"`.

2. Écrire la fonction récursive `dicho(tab, x, debut, fin)` : dans la liste triée `tab`, elle renvoie un indice de `x` compris entre `debut` et `fin`, ou `-1`. (1,5 point)

3. Combien d'appels effectue `dicho(list(range(0, 2000, 2)), 1001, 0, 999)` ? Combien en faudrait-il, environ, pour une liste d'un milliard d'éléments ? Justifier. (0,5 point)

Exercice 5 — Chemins avec obstacles (4 points)

Un robot part de la case (0, 0) et ne se déplace que vers la droite ou vers le bas. Certaines cases, les obstacles, lui sont interdites. On note `chemins_obs(l, c, obstacles)` le nombre de chemins de (0, 0) à (l, c) qui évitent les obstacles ; `obstacles` est un ensemble de couples, par exemple `{(1, 1), (2, 1)}` pour la grille ci-dessous.



1. Écrire la fonction récursive `chemins_obs(l, c, obstacles)`. Préciser sur la copie vos cas de base. (1,5 point)

2. Que vaut `chemins_obs(3, 3, {(1, 1), (2, 1)})` ? Décrire chacun de ces chemins par une suite de lettres D (droite) et B (bas). (0,5 point)

3. La fonction fournie `nb_appels_obs` compte les appels. Relever `chemins_obs(10, 10, {(5, 5), (7, 3)})` et le nombre d'appels correspondant. Commenter. (0,5 point)

4. Écrire `chemins_obs_memo(l, c, obstacles, memo)`, qui range chaque résultat calculé dans le dictionnaire `memo`. Vérifier que `chemins_obs_memo(14, 14, {(5, 5), (7, 3)}, {})` répond instantanément. (1,5 point)

Exercice 6 — Arbre à trois branches et listes imbriquées (2,5 points)

1. Écrire la fonction `arbre3(longueur, n)` qui dessine avec `turtle` un tronc, puis, si `n > 0`, trois branches partant de son sommet : à 40° à gauche, tout droit et à 40° à droite. Chaque branche est un arbre de profondeur `n - 1` dont le tronc mesure la moitié de celui de l'arbre. À la fin de chaque appel, la tortue doit être revenue à son point de départ, orientée comme au départ : expliquer pourquoi c'est indispensable. Vérifier avec `dessiner()`. (1 point)

2. Écrire la fonction récursive `nb_segments3(n)`. Combien de segments compte l'arbre de profondeur 4 ? (0,5 point)

3. Écrire la fonction récursive `aplatir(liste)`, qui renvoie la liste des éléments qui ne sont pas des listes, dans l'ordre, quelle que soit la profondeur d'imbrication. (1 point)

Exemple : `aplatir([1, [2, [3, 4]], 5])` renvoie `[1, 2, 3, 4, 5]`.

Récursivité — sujet d'entraînement

Corrigé détaillé et barème — document réservé au professeur

Le fichier corrigé de la partie B est fourni à part : recursivite_entrainement_corrige.py.

Question	Points	Attendus et répartition
A – Ex. 1 – Q1	2,5	a) 0,5. b) 1. c) 0,5. d) 0,5 : relation, L(4) = 15, formule et lien avec Hanoï.
A – Ex. 1 – Q2	1,5	a) 1 : somme des chiffres (0,2) ; 0,2 par valeur, dont le détail de racine(98765). b) 0,5 : h(n) < n pour n ≥ 10.
A – Ex. 2	3,5	Q1 0,5. Q2 1 (cas de base 0,5 ; appel i + 1, j − 1 : 0,5). Q3 1. Q4 0,5. Q5 0,5.
A – Ex. 3	2,5	Q1 1 : sept appels (0,5) + produit égal à 3 puissance 13 (0,5). Q2 1 : 7 et 14 (0,5) + terminaison (0,5). Q3 0,5 : 1 023.
B – Ex. 4	3,5	Q1 1,5 (cas de base 0,5 ; ordre de la concaténation 0,5 ; chiffres hexadécimaux 0,5). Q2 1,5. Q3 0,5 : 11 appels ; environ 30 pour un milliard.
B – Ex. 5	4	Q1 1,5 : obstacle → 0 (0,5) ; départ → 1 (0,5) ; bords gérés (0,5). Q2 0,5 : 5 chemins listés. Q3 0,5. Q4 1,5.
B – Ex. 6	2,5	Q1 1 : trois appels et retour exact (0,75) + justification du retour (0,25). Q2 0,5 : 121. Q3 1 : liste imbriquée traitée par appel récursif sur la tête.
Total	**20**	

PARTIE A

Exercice 1 (4 points)

1. La fonction motif (2,5 points)

a) **Cas de base** : `n == 0`, la fonction ne fait rien.

`motif(-1)` appelle `motif(-2)`, puis `motif(-3)`… : n n'atteint jamais 0. La pile grossit jusqu'à la limite de Python et le programme s'arrête sur une **RecursionError**.

b) `motif(1)` affiche 1 (entre deux appels `motif(0)` qui n'affichent rien).

`motif(2)` affiche ce qu'affiche `motif(1)`, puis 2, puis de nouveau `motif(1)` : 1 2 1.

`motif(3)` affiche `motif(2)`, puis 3, puis `motif(2)` :

1 2 1 3 1 2 1

c) Au plus **4 appels** en même temps : `motif(3)`, `motif(2)`, `motif(1)` et `motif(0)`. La profondeur de la pile ne dépend que de n, pas du nombre total d'appels.

d) `motif(0)` n'affiche rien : $L(0)=0$.

`motif(n)` affiche les lignes de `motif(n - 1)`, une ligne pour n, puis encore les lignes de `motif(n - 1)` :

$L(n)=L(n-1)+1+L(n-1)=2\times L(n-1)+1$

$L(1)=1$

$L(2)=3$

$L(3)=7$

$L(4)=2\times 7+1=15$

L(4) = 15, et plus généralement $L(n)=2^n-1$. C'est la même relation que pour le nombre de déplacements des **tours de Hanoï**, dont la fonction a exactement la même structure : deux appels sur n − 1, séparés par une action.

2. Les fonctions h et racine (1,5 point)

a) `h(n)` est la somme des chiffres de n. Détail de `racine(98765)` :

$9+8+7+6+5=35$, donc `racine(98765)` = `racine(35)`.

$3+5=8$, donc `racine(35)` = `racine(8)` = 8.

n	7	38	999	98765
racine(n)	**7**	**2**	**9**	**8**

Pour 38 : 3 + 8 = 11, puis 1 + 1 = 2. Pour 999 : 9 + 9 + 9 = 27, puis 2 + 7 = 9.

b) Pour un nombre n de deux chiffres ou plus, la somme de ses chiffres est **strictement inférieure à n**. Chaque appel récursif porte donc sur un entier plus petit : on finit par passer sous 10, le cas de base.

Exercice 2 (3,5 points)

1. supprime (0,5 point)

```
        return []                                   # ligne A
        return supprime(x, liste[1:])               # ligne B
        return [liste[0]] + supprime(x, liste[1:])   # ligne C
```

2. Palindrome avec indices (1 point)

```
def est_palindrome(mot: str, i: int, j: int) -> bool:
    if i >= j:
        return True
    if mot[i] != mot[j]:
        return False
    return est_palindrome(mot, i + 1, j - 1)
```

On teste le mot entier avec l'appel `est_palindrome(mot, 0, len(mot) - 1)`.

Le cas de base i >= j couvre à la fois les longueurs paires (i dépasse j) et impaires (i = j).

3. Maximum avec indice (1 point)

```
def maximum(liste: list, i: int) -> int:
    if i == len(liste) - 1:
        return liste[i]
    m = maximum(liste, i + 1)
    if liste[i] > m:
        return liste[i]
    return m
```

4. La fonction longueur (0,5 point)

`liste[0:]` est **la liste entière** : le problème ne diminue jamais et la liste vide n'est jamais atteinte.

Correction : `return 1 + longueur(liste[1:])`.

5. Indices ou tranches (0,5 point)

Une tranche **crée une nouvelle liste** (ou chaîne) à chaque appel, en recopiant presque tous les éléments : de nombreuses copies, en temps et en mémoire. Avec des indices, chaque appel travaille sur la même liste et ne transmet que deux entiers.

Exercice 3 (2,5 points)

1. Suite des appels (1 point)

`puiss(3, 13)` → 13 impair : $3 \times$ `puiss(3, 12)`

`puiss(3, 12)` → `puiss(9, 6)`

`puiss(9, 6)` → `puiss(81, 3)`

`puiss(81, 3)` → $81 \times$ `puiss(81, 2)`

`puiss(81, 2)` → `puiss(6561, 1)`

`puiss(6561, 1)` → $6561 \times$ `puiss(6561, 0)`, qui vaut 1.

Le résultat est donc $3 \times 81 \times 6\,561$. Or $81 = 3^4$ et $6\,561 = 81^2 = 3^8$:

$3 \times 3^4 \times 3^8 = 3^{1+4+8} = 3^{13}$

Le résultat est bien égal à 3^{13}.

2. Nombre d'appels et terminaison (1 point)

7 appels pour `puiss`, contre **14** pour la version simple (n = 13, 12, …, 0).

À chaque appel, n diminue strictement (n `//` 2 ou n `-` 1) en restant positif : il finit par valoir 0, le cas de base.

3. Hanoï (0,5 point)

$d(10) = 2^{10} - 1 = 1\,023$

1 023 déplacements.

PARTIE B

Exercice 4 (3,5 points)

1. en_base (1,5 point)

```
def en_base(n: int, b: int) -> str:
    if n < b:
        return CHIFFRES[n]
    return en_base(n // b, b) + CHIFFRES[n % b]
```

Même principe que pour le binaire : le dernier chiffre est `n % b`, les autres sont ceux de `n // b`. La chaîne `CHIFFRES` fournit les chiffres A à F au-delà de 9 : `en_base(2026, 8)` renvoie **"3752"**.

2. dicho (1,5 point)

```
def dicho(tab: list, x: int, debut: int, fin: int) -> int:
    if debut > fin:
        return -1
    milieu = (debut + fin) // 2
    if tab[milieu] == x:
        return milieu
    elif tab[milieu] < x:
        return dicho(tab, x, milieu + 1, fin)
    else:
        return dicho(tab, x, debut, milieu - 1)
```

3. Nombre d'appels (0,5 point)

11 appels pour 1 000 éléments : chaque appel divise la zone par 2, et $2^{10} = 1024$.

Pour un milliard d'éléments : $10^9 \approx 2^{30}$, car $2^{30} = (2^{10})^3 \approx (10^3)^3$.

Environ 30 appels.

Exercice 5 (4 points)

1. chemins_obs (1,5 point)

```
def chemins_obs(l: int, c: int, obstacles: set) -> int:
    if (l, c) in obstacles:
        return 0
    if l == 0 and c == 0:
        return 1
    total = 0
    if l > 0:
        total = total + chemins_obs(l - 1, c, obstacles)
    if c > 0:
        total = total + chemins_obs(l, c - 1, obstacles)
    return total
```

Cas de base : une case interdite ne peut être atteinte (0 chemin) ; la case de départ est atteinte d'une seule façon.

Avec des obstacles, il n'y a plus un seul chemin le long des bords : un obstacle sur la ligne 0 bloque toute la suite de la ligne. On ne peut donc pas reprendre le cas de base « l == 0 or c == 0 » de la version sans obstacle ; les tests l > 0 et c > 0 évitent de sortir de la grille.

2. Valeur (0,5 point)

5 chemins : DDDBBB, DDBDBB, DDBBDB, DDBBBD et BBBDDD.

Les obstacles bloquent la colonne 1 sur les lignes 1 et 2 : on la franchit soit tout en haut (ligne 0), soit tout en bas (ligne 3).

3. Nombre d'appels (0,5 point)

`chemins_obs(10, 10, {(5, 5), (7, 3)})` = **106 852**, obtenu au prix de **413 927 appels** : les mêmes cases sont recalculées de nombreuses fois, et ce nombre explose avec la taille de la grille.

4. Version mémorisée (1,5 point)

```
def chemins_obs_memo(l, c, obstacles, memo):
    if (l, c) in obstacles:
        return 0
    if l == 0 and c == 0:
        return 1
    if (l, c) not in memo:
        total = 0
        if l > 0:
            total = total + chemins_obs_memo(l - 1, c,
                                             obstacles, memo)
        if c > 0:
            total = total + chemins_obs_memo(l, c - 1,
                                             obstacles, memo)
        memo[(l, c)] = total
    return memo[(l, c)]
```

`chemins_obs_memo(14, 14, {(5, 5), (7, 3)}, {})` renvoie **24 045 480** instantanément : chacune des 225 cases au plus est calculée une seule fois.

Exercice 6 (2,5 points)

1. arbre3 (1 point)

```
def arbre3(longueur: float, n: int) -> None:
    if n == 0:
        forward(longueur)
        backward(longueur)
    else:
        forward(longueur)
        left(40)
        arbre3(longueur * 0.5, n - 1)
        right(40)
        arbre3(longueur * 0.5, n - 1)
        right(40)
        arbre3(longueur * 0.5, n - 1)
        left(40)
        backward(longueur)
```

Les rotations se compensent : +40, −40, −40, +40, soit 0 au total. La tortue retrouve son orientation avant de redescendre le tronc.

Ce retour est indispensable : chaque branche doit partir du sommet du tronc. Si un appel laissait la tortue ailleurs, la branche suivante partirait du mauvais endroit et les décalages s'accumuleraient à chaque niveau.

2. Nombre de segments (0,5 point)

```
def nb_segments3(n: int) -> int:
    if n == 0:
        return 1
    return 1 + 3 * nb_segments3(n - 1)
```

$1 \to 4 \to 13 \to 40 \to 121$

121 segments pour n = 4.

3. aplatir (1 point)

```
def aplatir(liste: list) -> list:
    if liste == []:
        return []
    tete = liste[0]
    if type(tete) == list:
        return aplatir(tete) + aplatir(liste[1:])
    return [tete] + aplatir(liste[1:])
```

Quand la tête est elle-même une liste, on l'aplatit par un appel récursif, puis on aplatit le reste. Ce sont **deux appels récursifs**, comme pour un arbre : une liste imbriquée est une structure récursive.