Exercice 1

Corrigé Contrôle NSI Terminale

1.1 Fonction inconnue `f1`

1.1.1 1. Cette fonction est-elle récursive ? Pourquoi ?

Oui, cette fonction est récursive car elle s'appelle elle-même dans son corps (ligne `return 2 * f(n - 1)`). Elle possède également un cas de base (`n == 0`) qui permet d'arrêter la récursion.

1.1.2 2. Tableau de traçage pour `f1(4)`

fonction appelée	valeur de retour
`f1(4)`	`2 * f1(3)` = `2 * 8` = **16**
`f1(3)`	`2 * f1(2)` = `2 * 4` = **8**
`f1(2)`	`2 * f1(1)` = `2 * 2` = **4**
`f1(1)`	`2 * f1(0)` = `2 * 1` = **2**
`f1(0)`	**1** (cas de base)

1.1.3 3. Valeur de `f1(4)`

`f1(4) = 16`

Remarque : Cette fonction calcule 2^n (2 puissance n).

1.1.4 4. Nombre d'opérations significatives T(n)

- Cas de base : T(0) = 1 opération
- Cas récursif : T(n) = T(n-1) + 1 multiplication

En résolvant : **T(n) = n + 1**

1.1.5 5. Complexité en notation de Landau

O(n) - Complexité **linéaire**

1.2 Récursivité - fonction `f2`

1.2.1 1. Tableau de traçage pour `f2(['a','b','c','d'])`

fonction appelée	valeur de retour
`f2(['a','b','c','d'])`	`f2(['b','c','d']) + ['a']` = `['d','c','b'] + ['a']` = **['d','c','b','a']**
`f2(['b','c','d'])`	`f2(['c','d']) + ['b']` = `['d','c'] + ['b']` = **['d','c','b']**
`f2(['c','d'])`	`f2(['d']) + ['c']` = `['d'] + ['c']` = **['d','c']**

fonction appelée	valeur de retour
`f2(['d'])`	`f2([]) + ['d'] = [] + ['d']` = **['d']**
`f2([])`	**[]** (cas de base)

1.2.2 2. Que réalise cette fonction ?

Cette fonction **inverse l'ordre des éléments d'une liste** (renverse la liste).

1.2.3 3. Cette fonction termine-t-elle ? Pourquoi ?

Oui, cette fonction termine car : - Elle possède un **cas de base** : `lst == []` qui arrête la récursion - À chaque appel récursif, la taille de la liste **diminue strictement** (`lst[1:]` enlève le premier élément) - On atteint donc nécessairement la liste vide après un nombre fini d'étapes

---

1.3 Complexité - Question de cours

1.3.1 Une relation de récurrence C(n) = C(n/2), quelle est sa complexité ?

Réponse : c. O(log n)

Justification : À chaque étape, on divise n par 2. Le nombre d'étapes nécessaires pour atteindre 1 est $\log_2(n)$.

1.4 Tri par sélection

1.4.1 1. Principe de l'algorithme

Le tri par sélection fonctionne ainsi : - On parcourt la liste de gauche à droite - À chaque étape i, on recherche le plus petit élément parmi les éléments non encore triés (de i à la fin) - On échange cet élément minimum avec l'élément en position i - On recommence avec i+1 jusqu'à la fin de la liste

1.4.2 2. Tableau de suivi du tri pour `[4, 2, 3, 7, 5, 6, 1]`

étape n°	liste au début de l'étape	liste à la fin de l'étape	nombre de comparaisons
1	`[4, 2, 3, 7, 5, 6, 1]`	`[1, 2, 3, 7, 5, 6, 4]`	6
2	`[1, 2, 3, 7, 5, 6, 4]`	`[1, 2, 3, 7, 5, 6, 4]`	5
3	`[1, 2, 3, 7, 5, 6, 4]`	`[1, 2, 3, 7, 5, 6, 4]`	4
4	`[1, 2, 3, 7, 5, 6, 4]`	`[1, 2, 3, 4, 5, 6, 7]`	3
5	`[1, 2, 3, 4, 5, 6, 7]`	`[1, 2, 3, 4, 5, 6, 7]`	2

Explication : - Étape 1 : On trouve 1 (min) et on l'échange avec 4 - Étape 2 : 2 est déjà à la bonne place - Étape 3 : 3 est déjà à la bonne place - Étape 4 : On trouve 4 et on l'échange avec 7 - Étape 5 : 5 est déjà à la bonne place

1.4.3 3. Impact d'une liste triée en sens inverse

Non, cela ne va **pas modifier** le nombre d'opérations effectuées. Le tri par sélection effectue toujours le même nombre de comparaisons quel que soit l'ordre initial de la liste. Il n'est pas adaptatif : il parcourt systématiquement tous les éléments restants pour trouver le minimum, même si la liste est déjà partiellement triée.

1.4.4 4. Complexité asymptotique

Calcul : - Étape 1 : n-1 comparaisons - Étape 2 : n-2 comparaisons - … - Étape n-1 : 1 comparaison

Total : (n-1) + (n-2) + … + 1 = n(n-1)/2 = $(n^2 - n)/2$

Complexité : **$O(n^2)$** - Classe de complexité : **quadratique**

1.4.5 5. Preuve de terminaison

La fonction termine car : - La boucle externe `for i in range(0, n-1)` est bornée : elle s'exécute exactement n-1 fois - La boucle interne `for j in range(i+1, n)` est également bornée - Il n'y a pas de boucle `while` qui pourrait créer une boucle infinie - Les opérations à l'intérieur des boucles sont toutes élémentaires (comparaisons, affectations)

La fonction termine donc toujours en un nombre fini d'étapes, quelle que soit la liste fournie.

1.5 Programmation objet

1.5.1 Classe Etudiant - Complétion

```python
class Etudiant:
    def __init__(self, nom, prenom):
        self.prenom = prenom
        self.nom = nom
        self.notes = []

    def ajouter_note(self, note):
        self.notes.append(note)

    def moyenne(self):
        s = 0
        for note in self.notes:
            s += note
        return s / len(self.notes)
```

1.5.2 Création et utilisation de l'étudiant "Bombeur" "Jean"

```python
>>> etudiant1 = Etudiant("Bombeur", "Jean")
>>> etudiant1.ajouter_note(8)
>>> etudiant1.ajouter_note(14)
>>> print(etudiant1.moyenne())
11.0
```

1.5.3 Classe Parcours - Complétion

```python
class Parcours:
    def __init__(self, code):
        self.code = code
        self.enseignements = []
        self.etudiants = []
        self.recus = []

    def ajouter_etudiant(self, etudiant):
        self.etudiants.append(etudiant)

    def valider(self):
        for etudiant in self.etudiants:
            if etudiant.moyenne() >= 10:
                self.recus.append(etudiant)
```

1.5.4 Utilisation du parcours L1_sciences_ingenieur

```python
>>> L1_sciences_ingenieur = Parcours("L1_SI_2024")
>>> L1_sciences_ingenieur.ajouter_etudiant(etudiant1)
>>> L1_sciences_ingenieur.valider()
>>> print(len(L1_sciences_ingenieur.recus))
1
```

Explication : Jean Bombeur a une moyenne de 11, il est donc ajouté à la liste des reçus lors de l'appel de la méthode `valider()`.

1.6 Points clés à retenir

- **Récursivité** : toujours vérifier le cas de base et la convergence
- **Complexité** : compter les opérations significatives (comparaisons, affectations)
- **Tri par sélection** : $O(n^2)$, non adaptatif, simple mais inefficace pour grandes listes
- **POO** : bien distinguer attributs d'instance et méthodes, utiliser `self`