Test de rentrée — corrigé

Manipulation de listes

1. La moyenne n'est définie que si la liste est non vide, ce qui constitue l'hypothèse à faire sur `T` et l'assertion à poser.

```
def moyenne(T: list[float]) -> float:
    '''
    Entree : T, liste non vide de flottants
    Sortie : la moyenne des elements de T
    '''
    assert len(T) > 0
    s = 0
    for x in T:
        s = s + x
    return s / len(T)
```

   Toute variante par indice (`for i in range(len(T))`) est acceptée.

2. L'idée du parcours unique : on maintient *deux* variables, le maximum `m1` et le second maximum `m2` des éléments déjà vus, et on les met à jour à chaque nouvel élément.

```
def second_max(T: list[float]) -> float:
    '''
    Entree : T, liste d'au moins 2 flottants deux a deux distincts
    Sortie : la deuxieme plus grande valeur de T
    '''
    assert len(T) >= 2
    if T[0] > T[1]:
        m1 = T[0]
        m2 = T[1]
    else:
        m1 = T[1]
        m2 = T[0]
    for i in range(2, len(T)):
        if T[i] > m1:
            m2 = m1
            m1 = T[i]
        elif T[i] > m2:
            m2 = T[i]
    return m2
```

   Sur l'exemple, `second_max(T)` renvoie `17.1` (le maximum étant `17.3`).

3. La boucle `for` effectue $n-2$ tours et chaque tour a un coût constant (au plus deux comparaisons et deux affectations) ; les opérations hors boucle sont en nombre constant. La complexité dans le pire cas est donc linéaire, en $O(n)$.

4. On parcourt la chaîne ; pour chaque caractère, s'il est déjà une clé du dictionnaire on incrémente sa valeur, sinon on crée l'association clé valeur.

```
def effectifs(M: str) -> dict[str, int]:
    '''
    Entree : M, chaine de caracteres
    Sortie : le dictionnaire associant a chaque caractere de M
             son nombre d'occurrences dans M
    '''
    d = {}
```

```
    for c in M:
        if c in d:
            d[c] = d[c] + 1
        else:
            d[c] = 1
    return d
```

Sur l'exemple : `effectifs(M)` vaut

```
{'S': 5, 'N': 4, 'P': 3}
```

Correction et terminaison

5. Trace de `recherche(Ttri, 12.4)` :

tour de boucle	a	b	m	Ttri[m]
1	0	11	5	10.2
2	6	11	8	15.8
3	6	7	6	12.4
4	(pas de quatrième tour)			

6. `recherche(Ttri, 12.0)` renvoie `-1`.
7. Posons $v = b - a$. Montrons que v est un variant de boucle : c'est un entier, et tant que la boucle tourne on a $a \leqslant b$, donc $v \geqslant 0$. À chaque tour où la fonction ne renvoie pas de valeur, comme $a \leqslant m \leqslant b$:
   - si `L[m] < x`, alors a devient $m + 1 > a$, donc v diminue strictement ;
   - si `L[m] > x`, alors b devient $m - 1 < b$, donc v diminue strictement.

   Une suite strictement décroissante d'entiers positifs est finie : la boucle effectue un nombre fini de tours. L'appel termine donc toujours, soit par un `return m`, soit par la sortie de boucle et le `return -1`.
8. (a) Il faut que `L` soit *triée par ordre croissant*. Cette hypothèse est utilisée pour justifier les éliminations : si `L[m] < x`, le tri garantit que tous les éléments d'indice $\leqslant m$ sont $< x$, donc que x ne peut se trouver qu'à droite de m (et symétriquement).

   (b) *Initialisation* : avant la boucle, $a = 0$ et $b = n - 1$: $[\![a; b]\!]$ contient tous les indices de `L`, donc P est vraie.
   Hérédité : supposons P vraie en début de tour.
   - Si `L[m] < x` : la liste étant triée, `L[i]` $\leqslant$ `L[m]` $< x$ pour tout $i \leqslant m$; si x figure dans `L`, c'est donc à un indice de $[\![m + 1; b]\!]$, qui est le nouvel intervalle $[\![a; b]\!]$: P reste vraie.
   - Si `L[m] > x` : symétriquement, x ne peut figurer qu'à un indice de $[\![a; m - 1]\!]$, nouvel intervalle : P reste vraie.

   P est donc un invariant de la boucle.

   (c) Si la fonction renvoie -1, c'est que la boucle s'est arrêtée avec $a > b$: l'intervalle $[\![a; b]\!]$ est vide. L'invariant P est encore vrai en sortie de boucle ; si x figurait dans `L`, il figurerait à un indice d'un intervalle vide, ce qui est absurde. Donc x ne figure pas dans `L`.
9. *Correction partielle* : le résultat est correct *lorsque* l'algorithme s'arrête. *Correction totale* : correction partielle *et* terminaison. La question 6 établit la terminaison et la question 7 la correction partielle : on a donc établi la correction totale.

Complexité

10. `occurrence(M, "PS")` renvoie `6` : dans `"SSNNPPPSNNSS"`, la première occurrence du facteur `"PS"` commence à l'indice 6 (le `P` d'indice 6 suivi du `S` d'indice 7).
    La fonction renvoie l'indice de la première occurrence de la chaîne `m` comme facteur de la chaîne `s`, et -1 si `m` n'est pas un facteur de `s`.

11.

(a) La boucle `for` effectue au plus $n-p+1$ tours. Pour chaque valeur de `i`, la boucle `while` effectue au plus p tours (elle s'arrête au plus tard lorsque $k=p$).

(b) Chaque tour de `while` a un coût constant, d'où un coût au plus $(n-p+1)\,p$: la complexité dans le pire cas est en $O(np)$. Pour p de l'ordre de $n/2$, ce coût est quadratique en n.

(c) Le pire cas est atteint lorsque presque chaque tentative est longue avant d'échouer, par exemple : `s = "AAAAAAAA"` (n fois `A`) et `m = "AAAB"` ($p-1$ fois `A` puis un `B`) : pour chaque `i`, la boucle `while` parcourt les $p-1$ premiers caractères avec succès et n'échoue que sur le dernier.

Questions de cours

12. (a) Les flottants sont représentés en base 2 sur un mot de taille fixe (signe, exposant, mantisse). Les nombres décimaux 10.2 et 2.2 n'ont pas d'écriture finie en base 2 : ils sont représentés par des valeurs approchées, et la somme calculée `10.2 + 2.2` vaut exactement `12.399999999999999`, qui est différente du flottant représentant 12.4. Le test d'égalité de la dichotomie échoue donc à chaque comparaison, et la fonction élimine peu à peu tout l'intervalle : elle renvoie -1. On ne teste jamais l'égalité stricte de deux flottants : on teste leur proximité à une tolérance près, par exemple `abs(x - y) < 1e-12` (tolérance absolue, à adapter à l'ordre de grandeur des valeurs manipulées).

13. Sur 8 bits, 5 s'écrit `0000 0101`. On complémente bit à bit : `1111 1010`, puis on ajoute 1 :

$$-5 \longmapsto \texttt{1111 1011}$$

14. (a) Représentation par listes d'adjacence :

```
G = {'A': ['B', 'C'],
     'B': ['A', 'C', 'D'],
     'C': ['A', 'B', 'E'],
     'D': ['B', 'E'],
     'E': ['C', 'D']}
```

(L'ordre à l'intérieur de chaque liste est libre ; le graphe étant non orienté, chaque arête apparaît dans les deux listes.)
Degrés : $d(A)=2$, $d(B)=3$, $d(C)=3$, $d(D)=2$, $d(E)=2$.

(b) Le graphe est connexe : le chemin $A-B-D-E-C$ passe par tous les sommets, donc tout couple de sommets est relié par un chemin. Il contient des cycles, par exemple $A-B-C-A$ (ou $B-D-E-C-B$).

(c) Un parcours en profondeur depuis A, en visitant les voisins dans l'ordre alphabétique : A, B, C, E, D. (Tout ordre compatible avec un parcours en profondeur est accepté, par exemple A, C, E, D, B.)