

Fiche méthode : while, dichotomie, récursivité

Boucle while – Terminaison – Dichotomie – Récursivité

1. La boucle while

On l'utilise quand on **ne connaît pas à l'avance le nombre de tours** : on s'arrête dès qu'on a trouvé, ou quand une quantité devient assez petite.

```
def premier_indice(L: list, v) -> int:
    '''Renvoie l'indice de la première occurrence de v dans L, -1 si v est absent.'''
    i = 0                                # 1. initialisation
    while i < len(L) and L[i] != v:      # 2. condition pour CONTINUER
        i = i + 1                        # 3. progression
    if i < len(L):
        return i
    return -1
```

● Méthode

- Écrire la condition de **poursuite**, pas la condition d'arrêt : on continue *tant que* on n'est pas au bout et pas trouvé.
- Dans `i < len(L) and L[i] != v`, l'ordre compte : si `i < len(L)` est faux, Python n'évalue pas `L[i]`.
- Vérifier que le corps de la boucle **fait progresser** la condition vers la sortie.

2. Justifier la terminaison : le variant

● Méthode

Un **variant** est une quantité **entière, positive ou nulle** tant que la boucle continue, qui **diminue strictement** à chaque tour. Comme une suite d'entiers naturels strictement décroissante ne peut pas être infinie, la boucle s'arrête.

Rédaction type : « La quantité ... est un entier naturel tant que la condition est vraie ; elle diminue d'au moins 1 à chaque tour car ... ; donc la boucle termine. »

Boucle	Variant
<code>while i < len(L) ... : avec <code>i = i + 1</code></code>	<code>len(L) - i</code>
dichotomie <code>while g <= d:</code>	<code>d - g + 1</code> (taille de la zone)
<code>while n > 0: avec <code>n = n // 10</code></code>	<code>n</code>
<code>while len(dist) != 0: avec une clé supprimée par tour</code>	<code>len(dist)</code>
fonction récursive sur <code>L[1:]</code>	<code>len(L)</code>

Un **invariant** est une propriété vraie avant chaque tour de boucle ; il sert à prouver que le résultat est correct (par exemple : « `s` contient la somme des `i` premiers éléments »).

3. La dichotomie

Dans une liste **triée**, on compare **v** à l'élément du milieu de la zone de recherche et on élimine la moitié qui ne peut pas le contenir.

```
def est_present(L: list, v) -> bool:
    '''L triée par ordre croissant. Renvoie True si v est dans L.'''
    g = 0
    d = len(L) - 1          # zone de recherche : indices g à d
    while g <= d:
        m = (g + d) // 2    # milieu (division entière)
        if L[m] == v:
            return True
        elif L[m] < v:
            g = m + 1        # v ne peut être qu'à droite de m
        else:
            d = m - 1        # v ne peut être qu'à gauche de m
    return False
```

● Méthode

- La taille de la zone est au moins divisée par 2 à chaque tour : au plus environ $\log_2 n$ tours, complexité $O(\log n)$ (contre $O(n)$ pour une recherche séquentielle).
- Sur un intervalle de réels $[a, b]$ que l'on coupe en deux jusqu'à une largeur ε : après k tours, la largeur vaut $\frac{b-a}{2^k}$; il faut $k \geq \log_2 \frac{b-a}{\varepsilon}$.
- Pour compléter une dichotomie à trous, **dérouler la boucle à la main** sur une liste de 5 ou 6 éléments.

4. La récursivité

Une fonction récursive s'appelle elle-même sur un **problème plus petit**.

<pre>def fact(n: int) -> int: '''n >= 0. Renvoie n!.''' if n == 0: # cas de base return 1 return n * fact(n - 1) # appel plus petit</pre>	<pre>fact(3) = 3 × fact(2) = 3 × 2 × fact(1) = 3 × 2 × 1 × fact(0) = 3 × 2 × 1 × 1 = 6</pre>
---	--

● Méthode

1. **Cas de base** : le cas si simple qu'on connaît la réponse (liste vide, $n == 0$, case hors du camp adverse...).
2. **Cas général** : supposer que l'appel sur le problème plus petit renvoie le bon résultat, et construire la réponse à partir de lui.
3. **Terminaison** : l'argument (n , la longueur, un indice) diminue strictement à chaque appel et finit par atteindre le cas de base.

5. Diviser pour régner : reconnaître le tri fusion

Un algorithme *diviser pour régner* découpe le problème en sous-problèmes (souvent deux moitiés), les résout récursivement, puis combine les résultats. Le tri fusion en est l'exemple type, et les sujets demandent souvent de le **reconnaître**.

```
def tri_fusion(L: list) -> list:
    '''Renvoie une copie triée de L.'''
    if len(L) <= 1:
        return L
    m = len(L) // 2
    G = tri_fusion(L[:m])
    D = tri_fusion(L[m:])
    return fusion(G, D)
```

Signes qui permettent de le reconnaître : découpage en `len(L) // 2`, **deux** appels récursifs, fonction auxiliaire qui combine deux listes **triées** en comparant leurs éléments. Complexité : $O(n \log n)$.

6. Les pièges

● Attention !

1. Oublier la progression dans un `while` : boucle infinie.
2. Écrire `L[i] != v and i < len(L)` : `IndexError` en fin de liste.
3. Dans une dichotomie, écrire `m = (g + d) / 2` : `m` est un flottant et ne peut pas servir d'indice.
4. Oublier le cas de base, ou faire un appel récursif qui ne rapproche pas du cas de base : `RecursionError`.
5. Écrire `fact(n - 1)` sans `return` ni utilisation du résultat : la valeur calculée est perdue.