ITC – cours n°3

Algorithmes de parcours à boucles imbriquées

I. Parcours sur des paires d'indices

Considérons le cas suivant : on souhaite déterminer, dans un tableau de nombres, la plus petite différence non nulle que l'on peut trouver entre deux éléments. On doit donc parcourir des paires d'éléments distincts du tableau, ou des paires d'indices distincts. Voici une proposition :

```
def plus_petit_écart(tab: list) -> int:
    # on suppose que la liste fournie est de taille 2 au moins
    n = len(tab)
    d = float('inf')
    for i in range(n):           # coût dans la boucle a x (n-i-1)
        for j in range(i+1, n):  # coût dans la boucle constant : a
        # (i, j) est une paire d'indices valides
        candidat = abs(tab[i] - tab[j])
        if candidat != 0 and candidat < d:
            d = candidat
    # si tous les éléments sont égaux entre eux,
    if d == float('inf'):
        d = 0
    return d
```

On retrouve les éléments classiques d'analyse : initialisation, parcours et finalisation. Cependant, la propagation doit se faire sur l'ensemble des paires d'indices distincts : autrement dit, il faut générer l'ensemble des paires (i, j) telles que

$$i \in [\![0; n[\![\quad ; \quad j \in [\![0; n[\![\quad ; \quad j > i$$

C'est à garantir ce parcours sur les paires d'indices que l'on utilise deux boucles imbriquées :

- pour une valeur de i donnée, on répète l'opération pour tout $j \in [\![i+1; n[\![$;
- on répète cette boucle pour tous les $i \in [\![0; n[\![$.

Discutons de la complexité C_n : la boucle intérieure coûte un temps constant a et est répétée $n-i-1$ fois, donc la complexité de cette fonction (en notant les temps constants hors boucles b) s'écrit :

$$C_n = \sum_{i=0}^{n-1} a(n-i-1) + b = a\sum_{i=0}^{n-1}(n-1) - a\sum_{i=0}^{n-1} i + b$$

$$C_n = an(n-1) - a\frac{n(n-1)}{2} + b = a\frac{n(n-1)}{2} + b$$

donc le terme dominant quand n devient très grand est proportionnel à n^2 et

$$C_n = \mathcal{O}(n^2)$$

II. Parcours de sous-listes

On peut être amené à parcourir toutes les sous-listes `tab` d'une liste `tab` . Prenons l'exemple de la recherche de la position d'une séquence dans une liste, par exemple

```
In [1]: liste = [0, 3, 4, 1, 3, 2, 9]
In [2]: recherche_séquence(liste, [1, 3, 2])
Out[1]: 3
In [3]: recherche_séquence(liste, [1, 1, 1])
Out[2]: -1  # signifie que la séquence n'est pas trouvée
```

Dans ce cas, on va partir d'une proposition qui teste l'égalité de deux listes, la second étant plus petite que la première par hypothèse :

```
def listes_égales(l1: list, l2: list) -> bool:
    n2 = len(l2)
    i2 = 0
    while i2 < n2 and l1[i2] == l2[i2]:
        i2 += 1
    if i2 == n2:  # (1)
        return True
    else:
        return False
    # (1) proposition alternative pour la fin : return i2 == n2
```

Le choix de la boucle while permet de s'arrêter dès la première case différente entre `l1` et `l2` ; on utilise alors la condition de sortie utilisée pour déterminer ce qui a arrêté la boucle (une case différente ou le fait d'avoir tout parcouru).

En ce qui concerne la complexité D_{n_2} : on comprend, puisque i augmente de 1 à chaque tour, que dans le pire cas ce sera la condition $i < n_2$ qui deviendra fausse et arrêtera la boucle, soit $D_{n_2} = an_2 = \mathcal{O}(n_2)$.

Utilisons à présent cette idée et imbriquons-là pour **chaque** case de la liste de départ : cela conduit à la proposition :

```
def recherche_séquence(tab: list, seq: list) -> int:
    # renvoie l'indice où commence la séquence si elle est présente,
    # -1 sinon
    n, nseq = len(tab), len(seq)
    for i in range(n-nseq+1):  # boucle faisant n - nseq tours
        # on essaie élément par élément de faire coller la séquence
        iseq = 0
        while iseq < nseq and tab[i+iseq] == seq[iseq]:  # (1)
            iseq += 1
        if iseq == nseq:  # si iseq == nseq, trouvé en position i
            return i
    # si on arrive au bout de la boucle for, on a pas trouvé la séquence
    return -1
```

La boucle (1) est l'équivalent de la fonction `listes_égales`, elle est répétée au plus n_{seq} fois et permet de tester si la liste `seq` est égale à la liste `tab`, **décalée de i**, d'où le choix d'indice dans le test `tab[i+iseq] == seq[iseq]`. On commence donc par chercher la séquence au départ de $i = 0$, puis au départ de $i = 1$ et ainsi de suite jusqu'à $i = n - n_{\text{seq}}$. On peut donc écrire la complexité de l'ensemble :

$$C_{n,n_{\text{seq}}} = D_{n_{\text{seq}}} \times (n - n_{\text{seq}}) = an_{\text{seq}}(n - n_{\text{seq}}) = \mathcal{O}(nn_{\text{seq}})$$

si on admet que n_{seq} reste très petit devant n.

III. Modification de liste : exemple du tri insertion

Nous avons vu dans un TD précédent une procédure pour insérer en place un élément dans une liste préalablement triée :

```
def insertion(tab, x):
    tab.append(x)
    j = len(tab)-1  # position candidate pour insérer x
    while j > 0 and x < tab[j-1]:
        tab[j] = tab[j-1]
        j -= 1
    tab[j] = x
```

On peut réutiliser cette idée sur le sous-tableau `tab[0:1]`, puis `tab[0:2]` ...pour que le sous-tableau trié soit de plus en plus grand, jusqu'à atteindre le tableau entier. Il s'agit du tri par insertion :

```
def tri_insertion(tab):
    n = len(tab)
    for i in range(n):
        # on insère l'élément i parmi tab[0:i] déjà trié
        j = i  # représente la position courante de l'insertion
        x = tab[j]
        while j > 0 and x < tab[j-1]:
            tab[j] = tab[j-1]  # déplacement de l'élément j-1 en posi-
tion j
            j -= 1
        tab[j] = x
        # tab[0:j+1] est trié
    # tab est trié
```

Estimons la complexité :

- dans le pire cas, le tableau est trié en ordre inverse et la boucle while fait systématiquement i tours, donc

$$C_n = \sum_{i=0}^{n-1} ai = a\frac{n(n-1)}{2} = \mathcal{O}(n^2)$$

- dans le meilleur cas, le tableau est déjà trié et la boucle while ne fait aucun tour, donc la boucle for a un coût constant et

$$C_n = \sum_{i=0}^{n-1} a' = a'n = \mathcal{O}(n)$$

En pratique, le tri par insertion est en moyenne en $\mathcal{O}(n^2)$, ce qui n'en fait pas un très bon tri pour les grands tableaux (nous verrons mieux plus tard) ; il est revanche très performant pour des petits tableaux ($n \lesssim 50$) et pour des tableaux presque triés, dans lesquels la boucle while fera toujours un petit nombre de tours.

ITC – cours n°3

Exercices

Parcours sur des paires d'indices

1 – Dans un tableau t, on définit les inversions comme les paires d'indices (i, j) telles que $i < j$ et $t[i] > t[j]$. Écrire une fonction `nb_inversions(tab: list) -> int` qui calcule et renvoie le nombre d'inversions dans un tableau fourni.

2 – Écrire une fonction `deux_égaux(tab: list) -> (int, int)` qui prend en paramètre une liste python et renvoie une paire d'indices i et j tels que $tab[i] = tab[j]$; si aucune paire satisfaisante n'est trouvée, la fonction renvoie $-1, -1$.

3 – Écrire une fonction `toutes_les_paires(tab: list) -> list` qui prend en paramètre une liste Python et renvoie une liste contenant toutes les paires d'entiers (i, j) telles que $tab[i] = tab[j]$.

Recherche de proche en proche

4 – Dans un tableau t, on appelle une corrélation de longueur ℓ une séquence $[i : i + \ell]$ telle que pour tout j dans cette séquence,

$$|t[i] - t[j]| < S$$

avec S un seuil à fixer. Écrire une fonction

```
longueur_corrélation(tab: list, seuil: int) -> int
```

qui calcule la longueur de la plus longue corrélation dans un tableau `tab` .

Modification de liste

5 – Recoder le tri par insertion.

6 – Écrire une fonction `indice_minimum(tab:list, i:int)` qui trouve l'indice de la position du minimum dans le sous-tableau `tab[i:]` .

7 – En s'inspirant de la fonction précédente, écrire une fonction `tri_sélection` qui trie le tableau en répétant l'opération de sélectionner le plus petit élément dans le sous-tableau `tab[i:]` et en le permutant avec le i^{e} élément.