

XI – Algorithmes de recherche séquentielle

Dans ce chapitre, nous présentons quelques algorithmes classiques de recherche dans une liste ou tableau unidimensionnel, faisant appel à une ou plusieurs boucles imbriquées.

1/ Boucles simples

Les algorithmes de cette section sont tous de complexité linéaire (dans le pire des cas). En effet, les boucles contiennent un nombre fixé d'opérations, et le nombre d'itérations est au plus égal à la longueur de la liste considérée.

1.1) Recherche d'un élément

En Python, le test `a in L` permet de savoir si un élément a appartient à une liste L . Néanmoins, il est bon de savoir programmer un tel test. Voici l'algorithme, puis le code d'une fonction `Recherche(a,L)` basée sur cet algorithme.

```
a, L : objet, liste
POUR e DANS L
    SI e=a ALORS
        Renvoyer 'Vrai'
        Interrompre l'algorithme
Renvoyer 'Faux'
```

```
def Recherche(a,L):
    for e in L:
        if e==a:
            return True
    return False
```

1.2) Recherche du minimum et du maximum

Python dispose déjà des fonctions `min()` et `max()`. L'algorithme sous-jacent a été croisé au cours de nos précédentes aventures. On l'utilise ici pour créer une fonction `MinMax(L)`.

```
L : liste de nombres
m ← L[0], M ← L[0]
POUR e DANS L
    SI e<m ALORS
        m ← e
    SINON SI e>M ALORS
        M ← e
Renvoyer (m,M)
```

```
def MinMax(L):
    m,M = L[0],L[0]
    for e in L:
        if e<m:
            m=e
        elif e>M:
            M=e
    return m,M
```

On peut obtenir les indices des extrema en modifiant la fonction ci-dessus de sorte qu'elle parcoure la liste à l'aide de ses indices (en exercice).

1.3) Recherche des deux plus grands éléments

On adapte l'algorithme de recherche du maximum en gardant en mémoire les deux plus grandes valeurs distinctes rencontrées.

<pre> L : liste de nombres M₁ ← L[0], M₂ ← L[1] SI M₁ < M₂ ALORS échanger M₁ et M₂ POUR e DANS L SI e > M₁ ALORS M₂ ← M₁, M₁ ← e SINON SI M₂ < e < M₁ ALORS M₂ ← e Renvoyer (M₁, M₂) </pre>	<pre> def deuxMax(L): M1, M2 = L[0], L[1] if M1 < M2: M1, M2 = M2, M1 for e in L: if e > M1: M1, M2 = e, M1 elif M2 < e < M1: M2 = e return M1, M2 </pre>
--	---

1.4) Comptage des éléments d'une liste

On peut dénombrer les occurrences des éléments d'une liste et relever leurs indices en utilisant un dictionnaire construit de la manière suivante :

- 1) ses clefs sont les éléments **distincts** de la liste
- 2) ses valeurs sont des listes [n,L], où *n* est le nombre d'occurrences de la clef associée et L la liste des occurrences de celles-ci.

Cette idée est à la base du **tri par comptage**.

<pre> L : liste de nombres D : dictionnaire vide POUR i ALLANT DE 0 A len(L)-1 e ← L[i] SI e DANS D ALORS D[e][0] ← D[e][0]+1 Ajouter i à la fin de D[e][1] SINON D[e] ← [1,[i]] Renvoyer D </pre>	<pre> def Comptage(L): D=dict() for i in range(len(L)): e=L[i] if e in D: D[e][0]=D[e][0]+1 D[e][1].append(i) else: D[e]=[1,[i]] return D </pre>
--	--

2/ Boucles imbriquées

Les algorithmes de cette section sont tous de complexité quadratique car ils comprennent deux boucles imbriquées contenant un nombre fixé d'opérations. En effet, les nombres de passages dans la boucle interne pour ces trois algorithmes sont respectivement

$$\sum_{j=1}^{n-1} j = \frac{(n-1)n}{2} \qquad \sum_{i=0}^{n-2} (n-i-1) = \frac{(n-1)n}{2} \qquad (n+1-p)p$$

2.1) Recherche des deux valeurs les plus proches dans une liste

Soit L une liste de longueur n . On va déterminer les indices i et j vérifiant $0 \leq i < j < n$ qui minimisent la valeur absolue de $L[i] - L[j]$.

<pre> L : liste de nombres n : longueur de L m,p,q ← L[0]-L[1] ,0,1 POUR j ALLANT DE 1 A n-1 POUR i ALLANT DE 0 A j-1 d ← L[i]-L[j] SI d < m ALORS m,p,q ← d,i,j Renvoyer p,q </pre>	<pre> def valeursProches(L): n=len(L) m,p,q = abs(L[0]-L[1]),0,1 for j in range(1,n): for i in range(j): d=abs(L[i]-L[j]) if d<m: m,p,q = d,i,j return p,q </pre>
--	--

Remarque : on obtiendra de la sorte le premier couple réalisant ce minimum.

2.2) Le tri à bulles

Cet algorithme permet de trier très simplement une liste de nombres L de longueur n .

- On parcourt la liste $L=L[0:n]$ en comparant les couples d'éléments successifs. Lorsqu'ils ne sont pas rangés dans l'ordre croissant, on les échange. Après ce premier passage, le plus grand élément se trouve alors en fin de liste.
- On réitère le procédé avec $L[0:n-1]$, puis avec $L[0:n-2]$, $L[0:n-3]$, etc...
- Après $n-1$ passages, on est assurés d'avoir une liste triée.

Voici sa traduction en Python.

```

def triBulles(L):
    n=len(L)
    for i in range(n-1):
        for j in range(n-i-1):
            if L[j]>L[j+1]:
                L[j],L[j+1]=L[j+1],L[j]

```

Remarque : nous verrons une version améliorée en exercice.

2.3) Recherche d'un mot dans une chaîne de caractères

Soient s une chaîne de caractères de longueur n et m un mot de longueur p non nulle. On veut savoir si la chaîne contient le mot et, si oui, à quelle(s) position(s).

Pour que ce mot m soit placé dans la chaîne s à l'indice $k \in \llbracket 0 ; n - p \rrbracket$, il faut que

$$\forall i \in \llbracket 0 ; p - 1 \rrbracket \quad m[i] = s[k + i]$$

ce qui s'écrit aussi $m = s[k : k + p]$. On va donc passer en revue les valeurs successives de k et ne conserver que celles qui satisfont la condition ci-dessus.

<pre> s, m : chaînes de caractères n ← taille(s), p ← taille(m) positions : liste vide POUR k ALLANT DE 0 A n-p i ← 0 TANT QUE i < p ET m[i] = s[k+i] i ← i+1 SI i = p ALORS Ajouter k à positions Renvoyer positions </pre>	<pre> def rechercheMot(s,m): n,p = len(s),len(m) positions=[] for k in range(n+1-p): i=0 while i<p and m[i]==s[k+i]: i=i+1 if i==p: positions.append(k) return positions </pre>
---	--

Remarque : la boucle WHILE n'est pas indispensable ici et on peut utiliser le test $m == s[k : k+p]$ à sa place. Elle permet néanmoins de rappeler que la complexité de ce test est en $\mathcal{O}(p)$.

2.4) Une idée à retenir

Lors de la recherche d'un mot dans une chaîne de caractères, on a rencontré une idée intéressante qu'il faut retenir. La voici : pour démontrer qu'une proposition $\mathcal{P}(i)$ est vraie pour tout entier $i \in \llbracket 0 ; p - 1 \rrbracket$, on exécute les instructions suivantes

```

i=0
while i<p and P(i):
    i=i+1
    
```

Si la variable i vaut p en sortie de boucle, la propriété est vérifiée. Sinon, i contient le premier rang pour lequel la proposition $\mathcal{P}$ est fausse.

Variante

On peut procéder plus simplement et créer à l'aide d'une boucle FOR une fonction qui renvoie True si la proposition est toujours vraie et False sinon.

```

def Test():
    for i in range(p):
        if not P(i):
            return False
    return True
    
```

En effet, la boucle ne va à son terme que si la propriété est vérifiée.

Exercices

Exercice 1.

On dispose d'un élément `a` et d'une liste `L`. Écrire les deux fonctions suivantes :

- 1) la fonction `Occurrences(a,L)` renvoie le nombre d'occurrence de `a` dans `L`.
- 2) la fonction `OccurrencesListe(a,L)` renvoie la liste des positions de `a` dans `L`.

Exercice 2.

Adapter la fonction `MinMax` vue en cours afin qu'elle renvoie également les indices du minimum et du maximum.

Exercice 3.

On reprend la fonction `deuxMax` vue en cours.

- 1) Modifier cette fonction afin qu'elle renvoie les indices de `M1` et `M2`.
- 2) Comment faire pour autoriser l'égalité de `M1` et `M2` ?

Exercice 4.

Comment modifier la fonction `triBulles` vue en cours pour éviter les parcours inutiles ?

