Algorithmes de recherche

Rechercher un élément dans une collection est l'un des nombreux problèmes fondamentaux de l'informatique. Que l'on pense aux applications de la recherche d'un mot dans une page web, d'une phrase dans un livre ou d'un individu dans une base de données. De nombreuses *structures de données* et algorithmes ont été conçus pour permettre une recherche efficace. Dans ce cours, on s'intéresse à la recherche (naïve) d'un mot dans un texte puis d'un élément dans un tableau.

1 Recherche dans une chaîne de caractères ♥

Dans cette partie, on s'intéresse au problème d'appartenance d'une chaîne de caractères, appelée `mot`, dans une autre appelée `texte`. Attention, ici, un mot désigne n'importe quelle sous-chaîne de caractères et peut contenir des espaces. On cherche donc à écrire une fonction PYTHON `recherche(mot, texte)` qui renvoie `True` s'il existe une occurrence de la sous-chaîne `mot` dans la chaîne `texte` et `False` sinon. On note et on notera `m = len(mot)` et `n = len(texte)`.

En PYTHON, on peut directement écrire :

PYTHON
```python
def recherche(mot, texte):
    """Vérifie si une sous-chaîne est dans une chaîne."""
    return mot in texte
```

On peut admirer la concision (et l'efficacité) de cette fonction. Bien que l'on utilise ici directement un idiome PYTHON, il ne faut pas croire pour autant que cette fonction est de complexité constante. En réalité, elle est de complexité quadratique, en $\Theta(nm)$, dans le pire cas. L'implémentation réelle est très efficace et utilise une combinaison astucieuse des algorithmes de BOYER-MOORE et HORSPOOL. La complexité dans les cas concrets est souvent linéaire, voire même sous-linéaire, mais tout ceci est largement hors du champ du programme.

Écrivons cette fonction « à la main ». On commence par écrire une fonction auxiliaire `occurrence(mot, texte, i)` qui vérifie si un mot apparaît en position `i` d'un texte, c'est-à-dire si `mot == texte[i : i + len(mot)]`. On suppose que $0 \leq i \leq n - m$.

PYTHON
```python
def occurrence(mot, texte, i):
    """Vérifie si une sous-chaîne apparaît en position i
    dans une chaîne."""
    m = len(mot)
    p = 0
    while p < m and mot[p] == texte[i + p]:
        p += 1
    return p == m
```

- **Invariant :** `p <= m and mot[:p] == texte[i : i + p]`
- **Correction :** Si à la fin `p == m` alors `mot[:m] == texte[i : i + m]` ce qui veut dire que l'on a bien une occurrence du mot en position `i`. Sinon, c'est que `p < m` et c'est donc que `mot[p] != texte[i + p]` et on n'a pas d'occurrence position `i`.
- **Variant :** $m - p$
- **Complexité :** $\Theta(m)$ dans le pire cas. $\Theta(1)$ dans le meilleur cas.

On écrit maintenant la fonction recherche en vérifiant successivement s'il existe une position qui correspond à une occurrence.

PYTHON
```python
def recherche(mot, texte):
    """Vérifie si une sous-chaîne est dans une chaîne."""
    n = len(texte)
    m = len(mot)
    i = 0
    while i <= n - m and not occurrence(mot, texte, i):
        i += 1
    return i <= n - m
```

- **Invariant :** $i \leq n - m + 1$ et $\forall k \in [\![0, i-1]\!]$, `mot != texte[i:i + m]`.

- **Correction :** Si à la fin $i \leq n - m + 1$ alors `occurrence(mot, texte, i)` est vrai et on a trouvé une occurrence. Sinon, c'est que $i = n - m + 1$ et l'invariant montre qu'il n'y a pas eu d'occurrence possible avant i, et il ne peut y en avoir en i ou après pour des raisons de taille.
- **Variant :** $n - m + 1 - i$
- **Complexité :** Dans le pire cas : $\Theta(m \times \max(1, n - m)) = O(nm)$. Dans le meilleur cas : $\Theta(\min(m, n - m))$.

On propose ci-dessous une fonction `occurrences(mot, texte)` qui renvoie la liste des occurrences, c'est-à-dire des indices de début des apparitions, d'un mot dans un texte. On peut évidemment directement adapter la fonction ci-dessus, mais on donne ici une version avec une boucle `for` et des tranches.

PYTHON

```
def occurrences(mot, texte):
    """Renvoie la liste des occurrences d'un mot
    dans une chaîne."""
    n = len(texte)
    m = len(mot)
    res = []
    for i in range(n - m + 1):
        if mot == texte[i : i + m]:
            res.append(i)
    return res
```

2 Recherche dans un tableau ♥

Un *tableau* en informatique est une zone contiguë en mémoire découpée en cases de même taille auxquelles on accède à coût constant. Les listes PYTHON sont des tableaux, et, pour l'instant, on peut considérer que « tableau » et « liste PYTHON » sont synonymes. Le contenu de la i^e case d'une liste peut donc être lu ou modifié en temps constant, indépendamment de i ou de la taille de la liste.

Il faut absolument connaître et savoir écrire sans hésiter les trois versions suivantes d'une fonction `appartient(element, tableau)` qui vérifie si un élément appartient à une liste.

PYTHON

```
def appartient(element, tableau):
    """Vérifie si un élément appartient à un tableau."""
    return element in tableau
```

PYTHON

```
def appartient(element, tableau):
    """Vérifie si un élément appartient à un tableau."""
    for autre in tableau:
        if autre == element:
            return True
    return False
```

PYTHON

```
def appartient(element, tableau):
    """Vérifie si un élément appartient à un tableau."""
    n = len(tableau)
    i = 0
    while i < n and element != tableau[i]:
        i += 1
    i < n
```

Pour cette dernière fonction :

- **Invariant :** $i \leq n$ et $\forall k \in [\![0, i-1]\!]$, `element != tableau[k]`.
- **Correction :** Si à la fin `i < n` alors c'est que `element == tableau[i]` et on a trouvé une occurrence. Sinon, c'est que $i = n$ et l'invariant montre qu'il n'y a pas eu d'occurrence possible.
- **Variant :** $n - i$
- **Complexité :** $\Theta(n)$ dans le pire cas. $\Theta(1)$ dans le meilleur cas.

Il est très simple de modifier les deux dernières fonctions pour renvoyer l'indice de la première occurrence de l'élément dans le tableau, ou `None` si l'élément n'apparaît pas.

PYTHON

```python
def indice(element, tableau):
    """Premier indice d'un élément dans un tableau
    ou None s'il n'apparaît pas"""
    n = len(tableau)
    i = 0
    while i < n and element != tableau[i]:
        i += 1
    if i < n:
        return i
    else:
        return None
```

3 Recherche dans un tableau trié ♥

On suppose maintenant que le tableau est trié par ordre croissant.

3.1 Algorithme naïf ♥

Considérons un tableau t de taille $n \in \mathbb{N}$, un indice $i \in [\![0, n-1]\!]$ et un élément x. On remarque que si $x < t[i]$ alors $x \in t \Rightarrow x \in t[:i]$. On peut utiliser cette idée pour améliorer la recherche naïve d'un élément dans un tableau. L'idée est de parcourir le tableau à la recherche du dernier élément plus petit ou égal à l'élément recherché.

PYTHON

```python
def appartient(element, tableau):
    """Vérifie si un élément appartient à
    un tableau trié par ordre croissant."""
    n = len(tableau)
    i = 0
    while i < n and element > tableau[i]:
        i += 1
    return i < n and element == tableau[i]
```

En notant t le tableau et x l'élément, montrons que « $i \leq n$ et $x \notin t[:i]$ est un invariant de la boucle `while` de cette fonction qui en montre la correction.

- **Initialisation :** Avant le début de la boucle, $0 = i \leq n = 0$ et $x \notin t[:0] = \emptyset$.
- **Conservation :** Supposons l'invariant vérifié au début d'un tour de boucle, on a donc « $i \leq n$ et $x \notin t[:i]$ et $i < n$ et $x > t[i]$ » et donc « $i < n$ et $x \notin t[:i+1]$, ce qui, après incrément de i, rétablit l'invariant.
- **Correction :** En sortie de boucle, on a la négation de la condition de boucle et l'invariant : « $i \geq n$ ou $x \leq t[i]$ » et « $i \leq n$ et $x \notin t[:i]$ ».
  - Si $i < n$.
    - ⋆ Si $x = t[i]$, on a trouvé un élément correspondant, il faut renvoyer vrai, ce qui est bien ce que réalise la fonction.
    - ⋆ Si $x \neq t[i]$, alors, $x < t[i]$ et d'après la remarque précédente, puisque $x \notin t[:i]$, c'est que $x \notin t$. Il faut renvoyer faux, ce qui est bien ce que réalise la fonction.
  - Si $i \geq n$. Alors $i = n$ et $x \notin t[:i]$, et donc $x \notin t$. Il faut renvoyer faux, ce qui est bien ce que réalise la fonction.
- **Terminaison :** $n - i$ est un variant. Cette quantité est entière d'après l'invariant $i \leq n$ et décroît de 1 à chaque passage dans la boucle. La fonction termine toujours.
- **Complexité :** Le variant de boucle montre que la boucle `while` s'exécute au plus n fois. Le corps de la boucle, ainsi que le reste est en $\Theta(1)$. La complexité est donc en $O(n)$.
  - Dans le pire cas, l'élément n'est pas présent dans le tableau et la boucle s'exécute exactement n fois. La complexité est donc en $\Omega(n)$ et donc la complexité au pire est en $\Theta(n)$, linéaire en la taille du tableau. On ne fait pas mieux que la recherche dans un tableau non trié.
  - Dans le meilleur cas, l'élément recherché est en première position et la boucle ne s'effectue qu'une seule fois. La complexité est en $\Theta(1)$. On ne fait pas mieux que la recherche dans un tableau non trié.
  - La complexité en mémoire est constante, en $\Theta(1)$.

3.2 Recherche dichotomique ♥

Pensons à la recherche d'un mot dans un dictionnaire (qui est finalement un tableau de mots triés pour l'ordre lexicographique). On ne va pas chercher à partir du début en lisant les mots un par un jusqu'à trouver ou non le mot recherché. L'idée est plutôt la suivante : on considère l'élément situé au milieu du tableau. On détermine si l'élément recherché se situe à gauche ou à droite de cet élément central, puis on répète le processus sur la portion du tableau sélectionnée. C'est ce que l'on appelle la recherche par *dichotomie* dans un tableau trié.

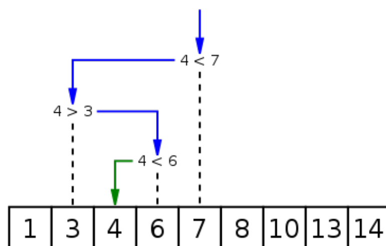

FIGURE 1 – Recherche par dichotomie de la valeur 4 dans un tableau.

Voici une des nombreuses implémentations possibles :

PYTHON

```python
def recherche_dichotomique(element, tableau):
    """Vérifie si un élément appartient à
    un tableau trié croissant de taille non nulle
    par recherche dichotomique."""
    fin = len(tableau) - 1
    debut = 0
    while debut < fin:
        milieu = (debut + fin) // 2
        if element == tableau[milieu]:
            debut = fin = milieu
        elif element < tableau[milieu]:
            fin = milieu - 1
        else:
            debut = milieu + 1
    return debut == fin and element == tableau[debut]
```

En notant t le tableau, n sa taille, x l'élément, d, m, f pour `debut`, `milieu`, `fin`, respectivement, montrons que « $d \leq f+1$ et $x \in t \Rightarrow x \in t[d:f+1]$ est un invariant de la boucle `while` de cette fonction qui en montre la correction. On montre en même temps que la quantité entière $f+1-d$ décroît strictement, ce qui, avec l'invariant $d \leq f+1$ montrera que c'est un variant de boucle.

- **Initialisation :** Avant le début de la boucle, $f+1 = n \geq 0 = d$ et évidemment $x \in t \Rightarrow x \in t[0:n] = t$.
- **Conservation :** Supposons l'invariant vérifié au début d'un tour de boucle, on a alors $d < f$ ainsi que $x \in t \Rightarrow x \in t[d:f+1]$. Distinguons trois cas :
  - Si $x = t[m]$, on a alors $d = m = f$ et donc bien $d \leq f+1$, de plus $x \in t$ est vrai tout comme $x \in t[m:m+1]$ et l'invariant est conservé. On a $f+1-d = 1$ alors qu'on avait $f-d > 0$, donc $f+1-d$ a décru strictement.
  - Si $x < t[m]$, alors puisque le tableau est trié, $x \in t \Rightarrow x \in t[:m]$. On a de plus $d \leq m \leq f$ puisque $d < f$ et que $m = \lfloor (d+f)/2 \rfloor$. Donc comme si $x \in t$ alors $x \in t[:m]$ et $x \in t[d:f+1]$, on a $t \in t[d:m]$. Après affectation $f \leftarrow m-1$ on a donc bien la deuxième partie de l'invariant $x \in t \Rightarrow x \in t[d:f+1]$ ainsi que la première puisque $d \leq m-1+1$.
    Pour le variant, on a, avant affectation, $f \geq m > m-1$ donc $f-d > (m-1)-d$. Après affectation $f-d$, et donc $f+1-d$ a bien décru strictement.
  - Cas symétrique qui se montre de la même manière.
- **Correction :** En sortie de boucle, on a la négation de la condition de boucle et l'invariant ce qui donne « $d \in \{f, f+1\}$ et $x \in t \Rightarrow x \in t[d:f+1]$ autrement dit $x \in t \iff d = f$ et $x = t[d]$ ce qui est bien ce que l'on renvoie.
- **Terminaison :** On a exhibé le variant $f+1-d$ qui montre la terminaison.
- **Complexité :** Le variant de boucle montre que la boucle `while` s'exécute au plus $f+1-d = n$ fois. Le corps de la boucle, ainsi que le reste est en $\Theta(1)$. La complexité est donc en $O(n)$. Mais cette majoration n'est pas très serrée.
  - Montrons que la complexité au pire est en fait en $O(\log n)$. On montre par récurrence sur $k \in \mathbb{N}$ qu'après k itérations de la boucle, $f - d < \frac{n}{2^k}$. Pour $k = 0$, on a $f - d = n - 1 < n$ et le résultat est vérifié. Supposons le acquis pour $k \in \mathbb{N}$. On a alors :

$$f - \left(\left\lfloor \frac{d+f}{2} \right\rfloor + 1 \right) < f - \frac{d+f}{2} = \frac{f-d}{2} < \frac{n}{2 \times 2^k} = \frac{n}{2^{k+1}}$$

    Donc si à la fin du $k+1$ passage on est dans le cas où on réalise l'affectation $d \leftarrow m+1$ on a, après affectation, $f - d < \frac{n}{2^{k+1}}$ et le résultat est acquis pour le rang $k+1$. On montre de même le cas $f \leftarrow m-1$. Dans le dernier cas on a $f - d = 0 < \frac{n}{2^{k+1}}$. Dans tous les cas le résultat est acquis après $k+1$ itérations. On a donc pour $k \geq \lfloor \log_2 n \rfloor$, $f - d < 1$, donc $f \leq d$. Il y a donc au plus $\lfloor \log_2 n \rfloor + 1$ itération de la boucle et donc une complexité au pire en $O(\log n)$. On peut exhiber un pire cas pour montrer que cette borne est serrée, c'est-à-dire que la complexité dans le pire cas est en $\Theta(\log n)$.
  - Dans le meilleur cas, l'élément recherché est en position $\lfloor \frac{n-1}{2} \rfloor$ et la boucle ne s'effectue qu'une seule fois. La complexité est en $\Theta(1)$.
  - La complexité en mémoire est constante, en $\Theta(1)$.