TP n° 9 : 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 TP, on s'intéresse à la recherche (naïve) d'un mot dans un texte puis d'un élément dans un tableau.

⚠ Dans ce TP, les boucles inconditionnelles (boucles `for`) sont interdites. On utilisera exclusivement des boucles conditionnelles (boucles `while`).

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)`.

L'élève Marius propose :

PYTHON
```
def recherche(mot, texte)
mot in texte
```

1. Corriger la fonction ci-dessus.

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 [1].

2. Écrire une fonction `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 n'utilisera pas directement une tranche ici, l'idée étant de l'implémenter « à la main ». On rappelle que l'on ne peut pas utiliser de boucles `for` mais uniquement des boucles `while`.

1. 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 sous-linéaire, mais tout ceci est largement hors du champ du programme.

3. Justifier la terminaison et la correction. Donner la complexité en fonction de `m`.
4. En utilisant la fonction précédente, écrire la fonction `recherche` demandée.
5. Justifier la terminaison et la correction. Donner la complexité en fonction de `n` et `m`.
6. Écrire 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.

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.

7. Écrire une fonction `appartient(element, tableau)` qui vérifie si un élément appartient à une liste. Rappel : pas de boucle `for`.
8. Justifier la terminaison, la correction et donner la complexité de cette fonction.
9. Modifier la fonction pour renvoyer l'indice de la première (puis de la dernière) occurrence de l'élément dans le tableau, ou `None` si l'élément n'apparaît pas.

3 Recherche dans un tableau trié

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

10. Modifier la fonction précédente pour arrêter la recherche dès que possible.
11. La complexité au pire est-elle améliorée ? Et la complexité dans le meilleur cas ?

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.

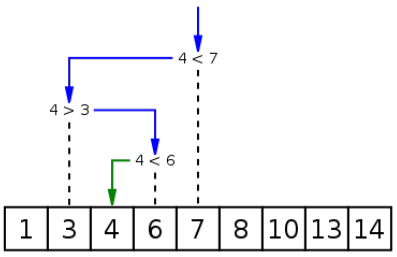

FIGURE 1 – Recherche par dichotomie de la valeur 4 dans un tableau.

12. Écrire une fonction `recherche_dichotomique` qui réalise cette idée.
13. Justifier la terminaison et la correction de votre programme. On admet, pour l'instant, que l'on obtient une complexité logarithmique en la taille du tableau (l'idée est que l'on divise par deux l'espace de recherche à chaque étape, donc il faut seulement un nombre logarithmique d'étapes).