IngeSUP - Cours 12 - Recherche dans une liste et complexité

Sommaire

Objectifs

- Implémenter un algorithme de recherche séquentielle dans une liste non triée ;
- Programmer un algorithme de recherche séquentielle dans une liste triée ;
- Implémenter un algorithme de recherche par dichotomie ;
- Programmer un algorithme de recherche d'extremums ;
- Comprendre la notation de 'Grand O' (O) ;
- Appréhender les consquences de la compléxité algorithmique ;
- Déterminer la complexité d'algorithmes simples.

Algorithmes de recherche

Une des utilisations les plus communes de l'informatique est le stockage de collections de données présentant des caractéristiques communes, et la recherche parmi ces données, d'éléments satisfaisants certains critères.

Si le nombre de données est important, comme c'est souvent le cas, les opérations de recherche, de tris et de stockage ne doivent pas être réalisées en consommant beaucoup de temps.

Il existe des algorithmes de recherche qui sont naïfs et qui prennent en moyenne plus de temps et ceux qui sont un peu plus "intelligents".

Apprendre à savoir faire la différence entre un algorithme "naïf" et un algorithme "intelligent" est la clé de ce chapitre.

Introduction

On appelle **recherche associative** le fait que le critère de recherche ne porte que sur la valeur de la clé de l'élément recherché.

La recherche est qualifée de :

- **Positive** : lorsque la clé recherchée est présente dans la collection de données ;
- **Négative** : lorsque la clé recherchée est absente de la collection de données.

Recherche séquentielle dans une liste non triée

Le premier cas abordé est le cas d'une recherche séquentielle dans une liste non triée. On se place sur le premier élément de la liste.

Tant qu'il reste des éléments, et que l'élément courant n'est pas x, on avance à l'élément suivant.

Si la liste a été parcourue entièrement et qu'on a pas trouvé l'élément, la recherche est négative.

Elle est positive sinon. L'élément sur lequel on s'est arrêté est celui que l'on cherchait.

Voyons l'exemple de la recherche d'un élément `x` dans une liste `L` :

Entrée [1]:

```
def recherche_sequentielle_LNT(L,x):

    for element in L: # on itère directement sur la liste

        if element == x:
            return True

    return False # arrivé à la fin de la liste, l'élément n'est pas présent

print(recherche_sequentielle_LNT([5, 3, 9, 17, 8, 11, 13, 4, 17, 6],13))
```

```
True
```

Recherche séquentielle dans une liste triée

La recherche séquentielle peut être améliorée lorsque la liste est **triée** : il est inutile de continuer la recherche si la valeur cherchée a été dépassée.

On se place sur le premier élément de la liste.

Tant qu'il reste des éléments, et que l'élément courant n'a pas dépassé x, on avance à l'élément suivant.

On a trouvé l'élément si on n'a pas parcouru toute la liste et si l'élément sur lequel on s'est arrêté est x.

Voyons l'exemple de la recherche d'un élément `x` dans une liste `L` :

Entrée [2]:

```
def recherche_sequentielle_LT(L,x):

    for element in L: # on itère directement sur la liste

        if element == x:
            return True
        if element > x:
            return False

    return False # arrivé à la fin de la liste, l'élément n'est pas présent

print(recherche_sequentielle_LT([1, 3, 5, 7, 8, 10, 13, 14, 17, 19],13))
```

```
True
```

Recherche par dichotomie

La **recherche séquentielle** impose dans le **pire des cas** le parcours de **l'ensemble des éléments de la liste** pour identifier si une valeur y est présente.

Si l'on manipule une **liste triée**, il existe plusieurs algorithmes de recherche plus efficaces que la recherche séquentielle.

Parmi eux, nous aborderons ce semestre **l'algorithme de recherche par dichotomie**.

Soit une liste `L` , un élément `x` recherché et `m` le milieu de la liste `L` :

- si *x = ième(L, m),* la recherche est positive ;
- si *x < ième(L, m),* on poursuit donc la recherche sur la **moitié inférieure** de la liste L ;
- si *x > ième(L, m),* on poursuit donc la recherche sur la **moitié supérieure** de la liste L ;
- si la recherche se termine sur une liste vide, la **recherche est négative**.

Voyons le code de la recherche dichotomique d'un élément `x` dans une liste triée `L` :

Entrée [3]:

```
def recherche_dichotomique(L, x):

    g = 0
    d = len(L) - 1

    while g <= d:
        m = (g + d) // 2

        if L[m] == x:
            return True

        if x < L[m]:
            d = m - 1

        else:
            g = m + 1

    return False

print(recherche_dichotomique([1, 3, 5, 7, 8, 10, 13, 14, 17, 19],13))
```

```
True
```

Recherche d'extremums

La recherche d'un **minimum** ou d'un **maximum** dans une liste revient à récupérer les bornes inférieures et supérieures de la liste, à savoir les **extremums** de la liste.

Nous regarderons donc ici la recherche d'un minimum ou d'un maximum dans le cas d'une liste non triée. La démarche est identique à la recherhe séquentielle vue précédement.

Chaque nouvel élément visité est comparé à l'élément le plus petit (respectivement le plus grand) identifié jusqu'à présent.

Voyons le code de la recherche d'extremums dans une liste `L` :

Entrée [4]:

```
def recherche_min(L):

    mini = L[0]
    for element in L:

        if element < mini:
            mini=element

    return mini
print(recherche_min([5, 3, 9, 17, 8, 11, 13, 4, 2, 6]))
```

```
2
```

Entrée [ ]:

```
def recherche_max(L):

    maxi = L[0]
    for element in L:

        if element > maxi:
            maxi=element

    return maxi
print(recherche_max([5, 3, 9, 17, 8, 11, 13, 4, 2, 6]))
```

Complexité d'un algorithme

L'algorithmique est la science qui s'intéresse non seulement à l'écriture des algorithmes, mais également à leur étude et analyse. Dans ce chapitre, nous abordons la notion de **complexité algorithmique**.

C'est une mesure de l'« efficacité » d'un algorithme. Nous nous intéressons donc non seulement à l'écriture d'algorithmes qui produisent des résultats corrects, mais également à la **vitesse à laquelle ils résolvent le problème**.

> **Information**
>
> Contrairement à ce que le nom suggère, ***la complexité*** n'est pas une mesure de si un algorithme est « simple » ou « complexe » d'un point de vue humain.
>
> C'est en fait bien souvent l'inverse : un algorithme simple aura généralement une complexité plus élevée (il « prend plus de temps »), qu'un algorithme ingénieux, qui aura une faible complexité (plus « rapide ») !

Complexité temporelle

L'objectif d'un calcul de **complexité algorithmique temporelle** est de pouvoir comparer l'efficacité d'algorithmes résolvant le même problème.

Dans une situation donnée, cela permet donc d'établir lequel des algorithmes disponibles est le **plus optimal**.

> **Note**
>
> Pour des données volumineuses, la différence entre les durées d'exécution de deux algorithmes ayant la même finalité, mais des complexités différentes peut être de l'ordre de plusieurs jours, voire même de plusieurs années !

Réaliser un calcul de complexité en temps revient à compter le nombre d'opérations élémentaires (affectation, calcul arithmétique ou logique, comparaison…) effectuées par l'algorithme.

Puisqu'il s'agit seulement de comparer des algorithmes, les règles de ce calcul doivent être indépendantes :

- du langage de programmation utilisé ;
- du processeur de l'ordinateur sur lequel sera exécuté le code ;
- de l'éventuel compilateur employé.

Par soucis de simplicité, on fera l'hypothèse que toutes les opérations élémentaires sont à égalité de coût, soit 1 « unité » de temps.

> *Exemple : Pour a = b * 3 . On a 1 multiplication + 1 affectation = 2 « unités » de temps.*

La complexité en temps d'un algorithme sera exprimé par une fonction, notée T (pour Time), qui dépend :

- De la taille des données passées en paramètres : plus ces données seront volumineuses, plus il faudra d'opérations élémentaires pour les traiter. On notera ***n*** le nombre de données à traiter.
- De la donnée en elle même et de la façon dont sont réparties les différentes valeurs qui la constituent.

Par exemple, si on effectue une recherche séquentielle d'un élément dans une liste non triée, on parcourt un par un les éléments jusqu'à trouver, ou pas, celui recherché.

Ce parcours peut s'arrêter dès le début si le premier élément est « le bon ». Mais on peut également être amené à parcourir la liste en entier si l'élément cherché est en dernière position, ou même n'y figure pas.

Cette remarque nous conduit à préciser un peu notre définition de la complexité en temps. On peut en effet distinguer deux formes de complexité en temps :

- **La complexité dans le meilleur des cas** : c'est la situation la plus favorable, *par exemple : recherche d'un élément situé à la première position d'une liste.*
- **La complexité dans le pire des cas** : c'est la situation la plus défavorable, *par exemple : recherche d'un élément dans une liste alors qu'il n'y figure pas.*

> **Note**
>
> On calculera le plus souvent la complexité dans le pire des cas, car elle est la plus pertinente. Il vaut mieux en effet toujours envisager le pire !

Ordre de grandeur

Pour comparer des algorithmes, il n'est pas nécessaire d'utiliser la fonction T, mais seulement l'ordre de grandeur asymptotique, noté O (« grand O »).

Une fonction $T(n)$ est en $O(f(n))$ (en grand O de f(n)) si :

$$\exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}^+, n \geq n_0 \Rightarrow |T(n)| \leq c|f(n)|$$

> **Note**
>
> Autrement dit :
>
> $T(n)$ est en $O(f(n))$ s'il existe un seuil n_0 à partir duquel la fonction T est toujours dominée par la fonction f, à une constante multiplicative fixée c près.

Considérons maintenant plusieurs expressions communes de $f(n)$:

- **Constant** : Pour un algorithme *en temps constant*, nous avons $t = O(1)$. Cela veut dire que le temps de calcul sera *indépendant* de la taille du problème n ;
- **Polynomial** : Pour un algorithme *en temps polynomial*, nous avons :

$$t = O(n^k)$$

où $k \geq 1$ est une constante (pas nécéssairement entière).

Les cas usuels sont :

- $O(n)$: Complexité linéaire ;
- $O(n^2)$: Complexité quadratique ;
- $O(n^3)$: Complexité cubique.

- $O(\log n)$: Complexité logarithmique ;
- $O(n \log n)$: Complexité quasi-linéaire ;
- $O(c^n)$, où $c \geq 1$: Complexité exponentielle.

Déterminer la complexité d'un algorithme

Pour déterminer la complexité d'un algorithme, il suffit seulement de compter le nombre d'opérations effectuées par l'algorithme.

import Par exemple, si l'on considère un tableau `x` de longueur n que l'on multiplie par un réel a :

Entrée [ ]:

```
import numpy as np

n = 100000
x = np.random.rand(n)

a = 10.0
for i in range(n):
    x[i] = a*x[i]
```

Le coût de l'opération `x[i] = a*x[i]` est $O(1)$ pour chaque `i` , et cela est répété n fois, donc au final le coût est $O(n)$.

Exercices de TD

Vous pouvez maintenant vous exercer à partir du notebook TD 12 - Recherche dans une liste et complexité (../TD/TD%2012%20-%20Recherche%20dans%20une%20liste%20et%20complexit%C3%A9.ipynb).