M415 : Compléments d'algorithmique

M. SYSKA

DUT INFO - IUT Nice Côte d'Azur

13 janvier 2020

Ce chapitre reprend certains supports du cours initial de
H. Collavizza et M. Gaetano, aussi modifié avec N. Stolfi

Qu'est ce qu'un algorithme ?

Algorithme : vient du nom du mathématicien *Al Khuwarizmi* (latinisé au Moyen Âge en *Algoritmi*), qui, au IX^e siècle écrivit le premier ouvrage systématique sur la solution des équations linéaires et quadratiques.

Algorithme : Suite finie, séquentielle de règles que l'on applique à un nombre fini de données, permettant de résoudre des classes de problèmes semblables. Calcul, enchaînement des actions nécessaires à l'accomplissement d'une tâche. *Le Petit Robert.*

Les objectifs du cours : PPN

On en reparle la semaine prochaine, après avoir digéré le hors d'œuvre du premier TD ...

Modules d'ouverture scientifique (OS05) : sensibiliser à la problématique de la complexité des algorithmes

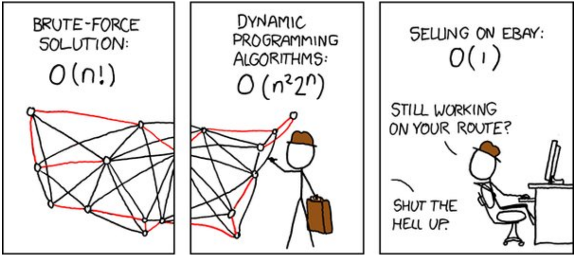
Prérequis : M313 (Algorithmique avancée : Structures arborescentes ; Récursivité ; Structures associatives)
Contenus :

- Sensibilisation au temps d'exécution des algorithmes, complexité
- Preuves d'algorithmes
- Exemples d'algorithmes classiques (tris, etc.)

Mots clés : Algorithmique ; Complexité

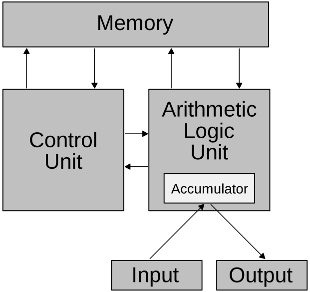
Les objectifs du cours : plan prévisionnel

- Terminaison et algorithmes corrects ?
- Notions élémentaires de complexité
- Techniques algorithmiques : divide-and-conquer, gloutons, heuristiques, programmation dynamique, ...
- Algorithmes de base sur les graphes : parcours, chemins, ...
- Problèmes d'optimisation combinatoire



Modèles abstraits d'ordinateur

- Machine de *Turing* (1936)
- Modèle RAM (Random Access Machine) (1960)
- Architecture de *von Neumann* (1945)
- Autres : P-RAM (Parallel RAM, plusieurs processeurs / cœurs)



Utilité : décrire un traitement de données sur une machine abstraite, dans un langage abstrait, indépendamment d'une implémentation sur une machine réelle : pouvoir calculer le coût du traitement et comparer des algorithmes.

Exemple 1 : Recherche d'un élément dans une séquence

Données :

- une séquence de n entiers distincts $a_0, a_1, \ldots, a_{n-1}$
- un entier particulier e

Résultat :

- -1 si e n'est pas dans la séquence $a_0, a_1, \ldots, a_{n-1}$
- j si $e = a_j$

Exemple

Recherche de 6 dans $\{7, 10, 4, 1, 6, 3\} \longrightarrow 4$
Recherche de 31 dans $\{7, 10, 4, 1, 6, 3\} \longrightarrow -1$

Principes d'un algorithme possible :

- Parcourir la séquence en comparant e à a_0, puis à a_1, puis à a_2, ...
- Si l'on trouve un i tel que $a_i = e$, le résultat est i
- Si l'on parcourt tous les a_i jusqu'à $i = n$, le résultat est -1

Traduction en Java :

```
class TableauEntier {
  /** méthode pour rechercher l'indice d'un élément
       dans un tableau */
  public static int recherche(int[] tab, int e) {
    int i=0;
    while(i<tab.length && e!=tab[i])
      i++;
    if (i==tab.length)
      return -1;
    else
      return i;
    }
}
```

Traduction en Python :

```
def linear_search_raw(e, l):
    i = 0
    while i < len(l):
        if e == l[i]:
            return i
        else:
            i += 1
    return False
```

mais on aurait aussi pu utiliser les méthodes internes de Python `in` et `index()` (voir plus tard https://wiki.python.org/moin/TimeComplexity)

```
def linear_search(e, l):
    if e in l:
        return l.index(e)
    else:
        return False
```

Ça compile et on pense que ça marche mais :

- Question 1 : l'algorithme est-il correct ?
- Question 2 : l'algorithme termine-t-il ?
- Question 3 : quel est l'espace mémoire utilisé ?
- Question 4 : quel est le temps d'exécution ?

L'algorithme est-il correct ?

- On peut faire des tests, et on doit en faire, mais : *Testing shows the presence, not the absence of bugs.* E. W. Dijkstra, 1969.
- On va aussi essayer de s'en convaincre en créant des assertions, mais : *Beware of bugs in the above code; I have only proved it correct, not tried it.* D. E. Knuth, 1977.

L'algorithme est-il correct ?

```
def linear_search_raw(e, l):
    '''Search e in list l
    Preconditions:
        l is a list of unsorted distinct elements
        e is an element
    Postconditions:
        return i if l[i] equals e, False otherwise
    '''
    i = 0
    while i < len(l):
        if e == l[i]:
            # A1 : e is at index i in l
            return i
        else:
            # A2 : for 0 <= j <= i, l[j]!=e
            i += 1
    # A3 : is not in l
    return False
```

Est-ce que les assertions sont vérifiées ?

- A_2 est vraie dès le premier passage dans la boucle (pour $i = j = 0$).
- A_2 est vraie à chaque passage dans la boucle.
  Puisqu'on entre dans ce bloc de la boucle quand `l[i]!=e` et que l'on a incrémenté i.
- A_3 est vraie.
  Puisque A_1 est vraie à chaque étape de la boucle, si $i = \text{len(l)}$ alors $\forall\ 0 \leq j \leq$ `len(l)` `l[j]!=` e
  donc e n'est pas dans le tableau.
- A_1 est vraie.
  Si $i <$ `len(l)` alors on est sorti de la boucle `while` avec la condition `e==l[i]`, donc e est à l'indice i.

L'algorithme termine-t-il ?

```
i = 0
while i < len(l):
    if e == l[i]:
        return i
    else:
        i += 1
return False
```

au pire, i croît de 0 à `len(l)` qui est une valeur finie.

Quelle est la place utilisée dans la mémoire ?

La place nécessaire pour stocker e et l.
Si l contient n éléments, on dit :

Complexité en espace : de l'ordre de n

Quel est le temps d'exécution ?

Temps CPU : dépend de la machine.
Temps de l'algorithme : on fait au plus `len(l)` passages dans la boucle `while`.
Si le tableau/liste contient n éléments :

- Complexité en temps dans le pire des cas : n
- Complexité en temps dans le meilleur des cas : 1
- Complexité en temps en moyenne :

$$p\frac{n+1}{2} + n(1-p)$$

(où p est la probabilité pour que e soit dans le tableau, et on suppose une distribution uniforme des n éléments)

Algorithmique M415 : du Python/Java bien commenté

- **Préconditions** : conditions d'entrée
  - quels types de données sont traités ?
    entiers, chaînes de caractères, réels, tableau d'entiers, ...
  - quelles conditions sur les données ?
    entiers positifs, non nuls, tableau d'entiers triés par ordre croissant, décroissant, ...
- **Postconditions** : que renvoie la méthode (return) ? quelles modifications ont été apportées sur les données ?
  renvoie l'indice de l'élément
  renvoie le maximum des éléments du tableau,
  ordonne les éléments du tableau par ordre croissant,...

- **Assertions** : propriétés liant les données d'entrée et les variables de la méthode.
  Permettent de justifier la correction : si la préconditions est vérifiée, après exécution de la méthode, la postcondition est vérifiée.
  Permettent de justifier la terminaison : si la préconditions est vérifiée, le calcul s'arrêtera.
- **Complexité en temps** : calculée dans le pire des cas, dans le meilleur des cas, ou en moyenne.

Exemple 2 : Recherche d'un élément dans une séquence triée

Données :

- une séquence de n entiers distincts $a_0, a_1, \ldots, a_{n-1}$ ordonnés par ordre croissant
- un entier particulier e

Résultat :

- -1 si e n'est pas dans la séquence $a_0, a_1, \ldots, a_{n-1}$
- j si $e = a_j$

Exemple

Recherche de 6 dans $\{3, 4, 6, 10, 34\} \longrightarrow 2$
Recherche de 31 dans $\{3, 4, 6, 10, 34\} \longrightarrow -1$

Solutions possibles

- **Solution 1** : le même algorithme que dans le cas où les éléments ne sont pas ordonnés
- **Solution 2** : dès que l'on trouve un élément plus grand, on s'arrête
- **Solution 3** : utiliser le fait que les éléments sont ordonnés pour appliquer le paradigme **diviser pour régner**

Quizz !

Principe de la solution 3 : comparer e à l'élément m qui est au milieu de la partie du tableau/liste considérée.

- Si $e = m$, renvoyer l'indice de m.
- Si $e < m$, chercher e dans la partie du tableau à gauche de m.
- Si $e > m$, chercher e dans la partie du tableau à droite de m.
- Si la partie considérée est vide, renvoyer -1.

$< m$	m	$> m$

Avantage : meilleure complexité en temps (cas le pire de l'ordre de $log(n)$ étapes)

Exemple

Recherche de 30

3	6	15	21	30	33	34	40
↑ g			↑ m				↑ d

Comme $30 > 21$

3	6	15	21	30	33	34	40
				↑ g	↑ m		↑ d

Comme $30 < 33$

3	6	15	21	30	33	34	40
				↑ g m d			

30 est trouvé, il est à l'indice 4 du tableau.

Traduction en Java :

```java
public int rechercheVite(int[] tab, int e) {
  int gauche = 0; int droite = tab.length - 1; int
     milieu;
  while (gauche <= droite) {
    // A1 : pour j < gauche tab[j]!=e
    //      pour j > droite tab[j]!=e
    milieu  = (gauche + droite) / 2 ;
    if (e==tab[milieu])
      // A2 : e est à l'indice milieu
      return milieu;
    if (e<tab[milieu]) droite = milieu - 1;
    else gauche = milieu + 1;
  }
  // A3 : e n'est pas dans le tableau
  return -1;
}
```

Traduction en Python :

```python
def binary_search(e, l):
    left = 0
    right = len(l) - 1
    while left <= right:
        # A1 : for j < left l[j]!=e
        #      for j > right l[j]!=e
        mid = (left + right) // 2
        if e == l[mid]:
            # A2 : e is at index mid
            return mid
        elif e < l[mid]:
            right = mid - 1
        else:
            left = mid + 1
    # A3 : e is not in l
    return False
```

Préconditions, postconditions, complexité

- Préconditions : l est un tableau/liste d'entiers distincts ordonnés par ordre croissant, *e* est un entier
- Postconditions : renvoie *i* si `l[i] == e`, *Faux* si *e* n'est pas dans le tableau
- Complexité en temps (pire cas) : $log(n)$

Questions

- Que se passe-t-il si les éléments du tableau ne sont pas distincts dans le cas de la recherche simple ?

21	6	2	21	21	33	21	40

- Que se passe-t-il si les éléments du tableau ne sont pas distincts dans le cas de la recherche dichotomique ?

3	6	21	21	21	33	34	40

- Comment peut-on prendre en compte le fait que les éléments sont triés dans la recherche simple ?

À vous : tri par insertion

- Données : un tableau d'entiers
- Résultat : le tableau est trié par ordre croissant
- Algorithme : respecter l'assertion de boucle A1

```
public static void triInsertion(int[] tab) {
  for (int i=1; i<tab.length;i++) {
    //A1: les éléments de tab de 0 à i-1 sont triés
    // par ordre croissant et tab contient les mêmes
    // valeurs qu'à l'état initial
  }
}
```

6	3	21	62	68	30	33	4	1	40	70

3	6	21	30	62	68	33	4	1	40	70

Un peu d'aide sur le net

- Insert-sort with Romanian folk dance
- FRENCH Computer Science Unplugged - Part 2 Sorting Networks - 2005