

M415 : Compléments d'algorithmique

Support ici :

<http://www-sop.inria.fr/members/Stephane.Perennes/TD1/cours1.pdf>

S. Perennes (stephane.perennes@inria.fr)

DUT INFO - IUT Nice Côte d'Azur

19 janvier 2022

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



- 1 Introduction
 - Définition du mot algorithme
 - Objectifs du cours
 - Algorithme et Preuves, Modèles abstraits
- 2 Exemples
 - Exemple 1 : Recherche d'un élément dans une séquence
 - Exemple 2 : Recherche d'un élément dans une séquence triée
- 3 Algorithmes de tri
 - Tri par insertion

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 écrit 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)

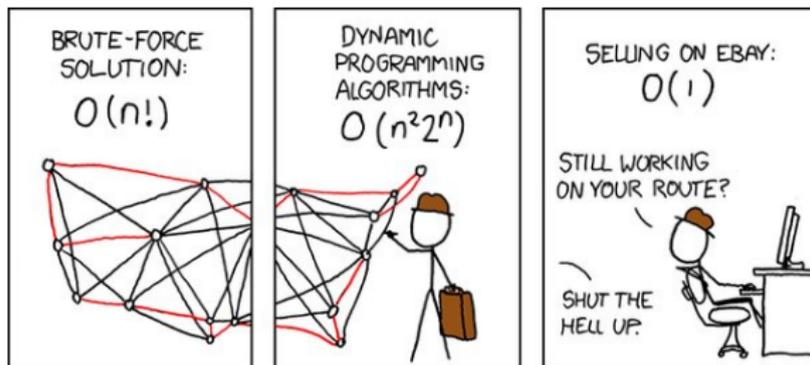
Contenus :

- 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



La crise des Mathématiques, Hilbert, Godel

Mathématiques & Logique

- Programme de Hilbert : Formaliser les mathématiques, une preuve est un processus logique. Objectif. : Système **complet** : Toute proposition est **décidable**, non contradictoire.
- Contradictions dans la théorie des ensembles.
- Dizième problème de Hilbert : Que peut on faire des équation diophantiennes :
 $Z^4 - 2XY - 3Y^2 + X + XY = 0.$

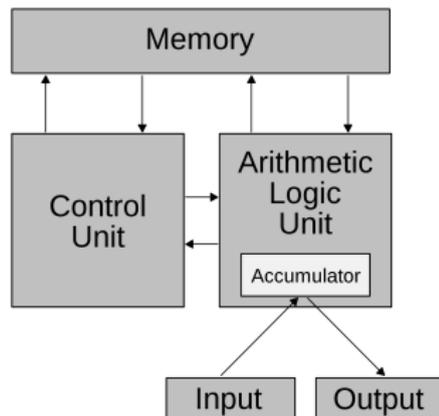
Avancées : Formalisation du processus de preuve. Un algorithme est une preuve e tune preuve est un algorithme.

Informatique

- Théorème de Godel (1931).
- Premiers Systèmes de calcul abstraits : Lambda Calcul (1936), Machine de Turing (1936) .

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.

These de Church (Alonso Church).

- Très nombreux modèles de calculs proposés.
- Tout ces modèles sont *équivalents*.
- Autrement une Machine de Turing peut simuler une PRAM et vice versa.

Le modèle de Turing est considéré comme universel.

Thèse de Church

Les Fonctions calculables *physiquement* sont celles calculées par une machine de Turing.

Depuis les années 2000, les modèles de calcul quantique remettent cette thèse en cause.

En pratique Les ordinateurs correspondent tous au modèle de Turing et sont proches des PRAM.

Impossibilité de la certification des programmes.

Mauvaises nouvelles :

- Les théorèmes de Godel et Turing affirment qu'il n'existe pas de procédure systématique (i.e. d'algorithme) décidant si un programme va s'arreter iou non.
- Impossible donc de savoir systématiquement ce que calcule un programme.

Mais

- Certification d'un programme particulier "à la main" .
- Les langages très structurés (OObjet, Java) simplifient cette tâche.
- Tests bien pensés assurant que le programme effectue bien le calcul souhaité
- developement modulaire : i.e on est (presque) sur qu'une boite noire (bibliothèque) effectue correctement un certain calcul.

Exemple 1 : Recherche d'un élément dans une séquence

Données :

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

Résultat :

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

Exemple

Recherche de 6 dans $\{7, 10, 4, 1, 6, 3\} \rightarrow 4$

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

Principes d'un algorithme possible :

- Parcourir la séquence en comparant e à a_0 , puis à a_1 , puis à a_2, \dots
- 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 :

```
1 class TableauEntier {
2     /** méthode pour rechercher l'indice d'un élément
3         dans un tableau */
4     public static int recherche(int[] tab, int e) {
5         int i=0;
6         while(i<tab.length && e!=tab[i])
7             i++;
8         if (i==tab.length)
9             return -1;
10        else
11            return i;
12    }
```

Traduction en Python :

```
1 def linear_search_raw(e, l):
2     i = 0
3     while i < len(l):
4         if e == l[i]:
5             return i
6         else:
7             i += 1
8     return False
```

Si on utilise les méthodes internes de Python `in` et `index()` comme ici (voir plus tard <https://wiki.python.org/moin/TimeComplexity>), en fait on fait deux fois la recherche alors qu'on a écrit 2 lignes en apparence.

```
1 def linear_search(e, l):
2     if e in l:
3         return l.index(e)
4     else:
5         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 ?

```

1 def linear_search_raw(e, l):
2     '''Search e in list l
3     Preconditions:
4         l is a list of unsorted distinct elements
5         e is an element
6     Postconditions:
7         return i if l[i] equals e, False otherwise
8     '''
9     i = 0
10    while i < len(l):
11        if e == l[i]:
12            # A1 : e is at index i in l
13            return i
14        else:
15            # A2 : for 0 <= j <= i, l[j]!=e
16            i += 1
17    # A3 : is not in l
18    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 $l[i] \neq e$ 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 $\forall 0 \leq j \leq \text{len}(l) \quad l[j] \neq e$
donc e n'est pas dans le tableau.
- A_1 est vraie.
Si $i < \text{len}(l)$ 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 ?

```
1     i = 0
2     while i < len(l):
3         if e == l[i]:
4             return i
5         else:
6             i += 1
7     return False
```

au pire, i croît de 0 à $\text{len}(l)$ 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 $\text{len}(l)$ 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, \dots, 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, \dots, a_{n-1}
- j si $e = a_j$

Exemple

Recherche de 6 dans $\{3, 4, 6, 10, 34\} \rightarrow 2$

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

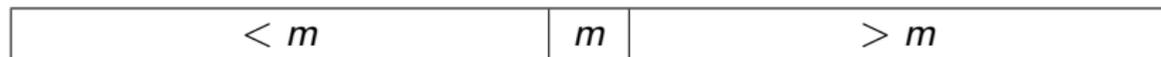
Solutions possibles

- **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 .



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 :

```
1 public int rechercheVite(int[] tab, int e) {
2     int gauche = 0; int droite = tab.length - 1; int
      milieu;
3     while (gauche <= droite) {
4         // A1 : pour j < gauche tab[j]!=e
5         //       pour j > droite tab[j]!=e
6         milieu = (gauche + droite) / 2 ;
7         if (e==tab[milieu])
8             // A2 : e est à l'indice milieu
9             return milieu;
10        if (e<tab[milieu]) droite = milieu - 1;
11        else gauche = milieu + 1;
12    }
13    // A3 : e n'est pas dans le tableau
14    return -1;
15 }
```

Traduction en Python :

```
1 def binary_search(e, l):
2     left = 0
3     right = len(l) - 1
4     while left <= right:
5         # A1 : for j < left l[j]!=e
6         #       for j > right l[j]!=e
7         mid = (left + right) // 2
8         if e == l[mid]:
9             # A2 : e is at index mid
10            return mid
11        elif e < l[mid]:
12            right = mid - 1
13        else:
14            left = mid + 1
15        # A3 : e is not in l
16    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 ?

Tri via une structure de donnée triée S

On souhaite trier un ensemble de n Elements

$$A = \{a_0, a_1, \dots, a_i, a_{n-1}\}.$$

- S Structure de donnée triée (liste, tableau, arbre, etc).
 $S = \emptyset$
- Pour i allant de 0 à $n - 1$ insérer i dans S
- **Assertion** : Après l'étape i $|S| = i$ et S contient $\{a_0, \dots, a_i\}$.
- Temps de calcul ? $\sum_{i=0}^{n-1} I(i)$ ou $I(i)$ est le temps pour insérer dans une structure de taille i .
- Coût de l'insertion :
 - Liste : recherche en au plus i , insertion en temps constant.
 - Tableau : recherche en au plus $\log_2(i)$, insertion en temps i (il faut déclarer le tableau).

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

```
1 public static void triInsertion(int[] tab) {  
2     for (int i=1; i<tab.length;i++) {  
3         //A1: les éléments de tab de 0 à i-1 sont triés  
4         // par ordre croissant et tab contient les mêmes  
5         // valeurs qu'à l'état initial  
6     }  
7 }
```

Étapes $i = 1$ et $i = 6$:

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

