Algorithmique 1

Licence d'informatique
Licence de mathématiques
2ème année

2015 - 2016

François Denis, Stéphane Grandcolas, Yann Vaxès

Chapitre 1

Introduction.

Ce cours constitue une introduction à l'algorithmique. Il est destiné aux étudiants de deuxième année de la licence d'informatique de l'université d'Aix-Marseille, supposés avoir déjà des bases solides en programmation. Il est également proposé en option aux étudiants de la licence de mathématiques.

1.1 Contenu du cours

Le premier chapitre est une introduction à l'analyse des algorithmes et en particulier, aux notions fondamentales de preuves et de complexité des algorithmes. Le second chapitre est consacré aux structures de données linéaires (tableaux, listes, piles, files et tables de hachage). Le troisième chapitre est dédié aux principaux algorithmes de tris et à l'étude de leur complexité. Le quatrième chapitre est consacré aux arbres binaires de recherche, aux tas et aux arbres lexicographiques. Le cinquième chapitre introduit la notion de programmation dynamique qui permet de résoudre efficacement certains problèmes d'optimisation. Le sixième et dernier chapitre est consacré aux graphes qui interviennent dans la représentation de nombreux problèmes, et pour lesquels un grand nombre d'algorithmes ont été développés, notamment pour le calcul de plus courts chemins. Nous présenterons quelques applications pratiques des graphes, entre autre dans le domaine du routage.

Ce cours est largement inspiré de "Introduction à l'algorithmique" de T. Cormen, C. Leiserson et R. Rivest (éditions DUNOD) qui est la référence pour les cours algorithmique 1 et algorithmique 2 de la licence d'informatique.

Ce support de cours correspond aux cours magistraux ; ils sont complétés

par des travaux dirigés et des travaux pratiques. Les algorithmes sont donnés en pseudo-code, ce qui permet de s'abstraire de détails d'implémentation (voir section suivante), ou en C, langage utilisé dans les travaux pratiques.

1.2 Description des algorithmes en pseudo-code

Un *algorithme*[1] est la description d'un *calcul* effectué sur des *données discrètes*. On distingue les données fournies en *entrée* à l'algorithme et celles qu'il calcule en *sortie*.

Un algorithme est décrit par un *pseudo-code* suffisamment précis pour pouvoir être implémenté dans tout langage de programmation, et en C en particulier.

Conventions retenues pour le pseudo-code :

1. utilisation d'*indentations* pour identifier les blocs d'instructions ;
2. les *tests* usuels $(=, <, >, \leq, \geq)$ sont disponibles ainsi que les connecteurs logiques de base (`ET, OU, NON`)[2] ;
3. les *structures de contrôles conditionnelles* usuelles, `si ...alors ...` et `si ...alors ...sinon ...`, sont disponibles ;
4. les *structures itératives* usuelles `tant que, pour, répéter ... jusqu'à`, sont disponibles ;
5. l'*affectation* de variable est notée $:=$;
6. il est possible de définir et d'utiliser des *fonctions* et *procédures* ; les passages de paramètres se font *par valeur* ; les appels récursifs sont possibles ;
7. les *commentaires* sont précédés du caractère #.

1.3 Premiers exemples d'algorithmes

L'algorithme 1 décrit une méthode simple de tri qui consiste à parcourir un tableau, du second indice jusqu'au dernier, et à insérer l'élément courant parmi les éléments précédents, en préservant leur ordre. L'algorithme 2 réalise l'*insertion* d'un élément dans un tableau trié, en préservant l'ordre. La division de l'algorithme de tri en deux algorithmes rend sa compréhension, et donc sa vérification, plus simple. Remarquez que dans la condition d'arrêt

1. d'après le nom du mathématicien perse Al Khawarizmi (783-850).
2. on supposera, comme c'est le cas en C, que l'évaluation de la proposition `p ET q` (resp. `p OU q`) commence par `p` et s'arrête si `p` est évaluée à `FAUX` (resp. à `VRAI`).

de la boucle de l'algorithme 2, $T[i]$ n'est pas évalué si $i = 0$ puisque dans ce cas, la conjonction est fausse.

Algorithme 1: TriParInsertion

entrée : $T[1, n]$ est un tableau d'entiers, $n \geq 1$.
résultat : les éléments de T sont ordonnés par ordre croissant.
début
    pour $j = 2$ **à** n **faire**
        InsertionOrdonnée(T[j],T[1, j-1])

Algorithme 2: InsertionOrdonnée

entrée : x est un entier ; $T[1, n]$ est un tableau d'entiers trié par ordre croissant.
sortie : $T[1, n + 1]$ contient les éléments de $T \cup \{x\}$ ordonnés par ordre croissant.
début
    $i := n$ # i est l'indice de l'élément courant du tableau T
    tant que $i > 0$ *ET* $T[i] > x$ **faire**
        # l'élément courant est supérieur à x
        $T[i + 1] := T[i]$ # décalage pour laisser de la place à x
        $i := i - 1$ # passage à l'élément précédent
    $T[i + 1] := x$ # insertion de x dans T.

L'algorithme 3 décrit la recherche d'un élément x dans un tableau trié $T[m, n]$: il commence par comparer x à l'élément qui se trouve au centre du tableau puis, selon les cas, arrête la recherche ou la continue sur l'une des deux moitiés du tableau : la première si x est plus petit que l'élément central, et la seconde si x est plus grand. On parle de recherche *dichotomique*. Cet algorithme suit la méthode *diviser pour régner (divide and conquer*), qui consiste à diviser le problème initial en sous-problèmes de tailles inférieures, à résoudre ces sous-problèmes puis à combiner leurs solutions pour trouver la solution du problème initial. C'est un algorithme *récursif* puisqu'il comporte des appels à lui-même.

Algorithme 3: rechercheDichotomique (T, m, n, x)

entrée : $T[m, n]$ est un tableau d'entiers trié par ordre croissant, $1 \leq m \leq n$; x est un entier.
sortie : 0 si $x \notin T$ et i si $T[i]$ est la première occurrence de x dans $T[m, n]$.

```
début
    si m = n alors
        si T[m] = x alors
            retourner m
        sinon
            retourner 0
    sinon
        k := ⌊(m+n)/2⌋ [a] # on a m ≤ k < n
        si T[k] < x alors
            retourner rechercheDichotomique(T, k + 1, n, x)
        sinon
            retourner rechercheDichotomique(T, m, k, x)
```

a. partie entière : voir paragraphe 3.1.