TD R1.01.P2 – Semaine 46

Objectifs du TD/TP

- Etudes et comparaison d'algorithmes
- Efficacité théorique d'un algorithme
- Complexité des algorithmes et ordres de grandeur

Ce TD dure 1 X 1h30

Exercice 1. Efficacité en temps et efficacité mémoire

Dans cet exercice, on s'intéresse au problème qui consiste à calculer les termes de la suite linéaire d'ordre 2 de *Fibonacci* définie par :

- $u_0 = 0$;
- $u_1 = 1$;
- $u_n = u_{n-2} + u_{n-1}$ pour $n > 1$.

a. Donnez un algorithme « fib1 » dont l'exécution fournit le terme u_n pour tout $n \geq 0$ tout en conservant tous les autres termes qui l'ont précédé dans un tableau. « fib1 » sera une méthode *Java* qui doit renvoyer le tableau contenant tous les termes. A vous de déterminer sa signature.

b. Donnez le nombre d'opérations *f(n)* en fonction de *n* de l'algorithme pour calculer u_n. Pour ce faire, on va considérer que :

   - Une déclaration `int i vaut 1 opération`
   - Une affectation `a = 3 vaut 1 opération`
   - Une addition ou multiplication ou division `vaut 1 opération`
   - Un test de condition (`<=, <, >=, >, !`) `vaut 1 opération`
   - Une auto-incrémentation `i++` vaut `2 opérations (car équivalent à i = i + 1)`
   - Une auto-décrémentation `i--` vaut `2 opérations (car équivalent à i = i - 1)`
   - Accès à un tableau `tab[i] vaut 0 opération`
   - Accès à la taille d'un tableau `tab.length vaut 1 opération`
   - Opération `% (modulo en Java) vaut 1 opération`
   - Un opérateur logique (&&, ||) `vaut 1 opération`
   - Une création de tableau `new int[v] vaut 1 opération`

     Exemple : `tab[i] = tab[i+1] + (i++)` compte pour 5 opérations (2 pour l'auto-incrémentation + 2 additions + 1 affectation).
   - On ne comptabilise PAS ni la déclaration des paramètres, ni l'opération de retour (*return*) si elle existe

Exercice 2. Distinction entre problème et algorithme

Dans cet exercice, on s'intéresse au problème qui consiste à déterminer si un entier naturel est un nombre premier ou non. Ce problème se nomme « test de primalité ». Nous allons montrer en traitant cet exercice qu'à un problème peut être associé plusieurs (une infinité !) algorithmes solutions (chacun présentant des avantages et inconvénients sur le plan de la qualité).

On rappelle qu'un nombre premier est un entier naturel positif (un élément de IN+) qui admet exactement deux diviseurs distincts entiers naturels (qui sont alors 1 et lui-même). Ainsi, 1 n'est pas premier car il n'a qu'un seul diviseur entier positif (lui-même) et zéro non plus car il est divisible par tous les éléments de IN*.

a. Donnez un algorithme *estPremier1* résolvant ce problème. « estPremier1 » sera une méthode Java, qui aura une signature bien précise, à vous de la déterminer.

b. Combien d'instructions sont exécutées par cet algorithme pour un entier « n » dans le meilleur et dans le pire des cas ?

c. Montrez qu'il est possible de n'étudier que les diviseurs de 2 à √n inclus. En déduire un algorithme *estPremier2*. Combien d'instructions sont exécutées par cet algorithme pour un entier « n » dans le meilleur et le pire des cas ?

Exercice 3. Suite de Syracuse

Dans cet exercice nous montrons sur un exemple très simple que le problème de la terminaison d'un algorithme est un problème complexe même pour des algorithmes en apparence triviaux.

En mathématiques, on appelle suite de *Syracuse* une suite d'entiers naturels définie de la manière suivante : on part d'un nombre entier strictement plus grand que 1. S'il est pair, on le divise par 2 et s'il est impair, on le multiplie par 3 et on ajoute 1.

En répétant l'opération, on obtient une suite d'entiers positifs dont chacun ne dépend que de son prédécesseur.

a. Calculez la suite de *Syracuse* pour l'entier 14.

b. Ecrivez une méthode *Java* dont l'exécution calcule et affiche pour un $n > 1$ (à vérifier) passé en paramètre sa suite de *Syracuse* et s'arrête dès que le terme 1 est atteint. Il affichera également le nombre des termes calculés.

c. Calculez le nombre d'opérations effectuées par l'algorithme pour un entier « n » (n > 1) quelconque.

d. Calculez le nombre d'opérations effectuées par votre algorithme pour n = 7 et ensuite pour n = 20. Constatez que la grandeur du nombre de départ n'est pas un critère fiable pour estimer le nombre d'opérations (et donc la longueur de sa séquence de *Syracuse*).

e. Pour certaines valeurs de « n », il est très facile de prouver la terminaison de l'algorithme, lesquelles ? Calculer le nombre d'opérations pour ces valeurs spécifiques de « n ».