Université de Nice - Sophia Antipolis
Faculté des Sciences

DEUG MIAS MP1

Programmation 2000-01

1. BOUCLES ET COMPLEXITE

A. Le langage de programmation Java

Java est un langage de programmation normalisé. Le mot *Java* est une marque déposée par la firme *Sun Microsystems* [`http://java.sun.com`] qui contrôle le développement du langage, avec l'aide des principaux acteurs du marché [IBM, Hewlett-Packard, etc]. Ce que l'on nomme *Java 2* débute en réalité avec la version 1.2 du langage [vous travaillez au MIPS avec la version 1.3].

Les outils de développement Java [compilateur, run-time] sont inclus dans le JDK [*Java Development Kit*] distribué par Sun pour Windows ou Linux. Sur son site `http://www.apple.com/java`, le constructeur Apple distribue lui-même pour Mac OS-9 son MRJ-SDK [*Macintosh Runtime for Java, Software Development Kit*] en version 1.1.8 ce qui s'avère suffisant pour les programmes du DEUG. Sous Mac OS-X [un Unix avec interface graphique pour les Macintosh], Java 2 est fourni en standard avec tous les outils Unix.

RealJ est un *environnement de programmation* [en anglais IDE : *Integrated Development Environment*] réservé à Windows. Il permet de programmer en Java, gratuitement pour l'éducation, et de manière simple en gérant des *projets*. Un projet n'est autre qu'un ensemble de classes, dépendantes ou pas, dont l'une est la classe principale [*Set As Main*] destinée à être exécutée.

Un CD-ROM contenant les outils Java sous Windows et Linux est disponible pour le prêt au bureau 203. Un CD-ROM pour le Macintosh est également disponible [mailto:roy@unice.fr].

B. Le travail en Java au second semestre MP1

Seules des séances en salles-machines sont prévues, avec des feuilles d'auto-formation contenant des compléments sur le langage, sur l'algorithmique, et des exercices associés. Tous les exercices doivent avoir été compris et programmés pour les examens. Ne prenez pas de retard, vos enseignants attendent de vous un travail régulier, correspondant au niveau que vous souhaitez atteindre. Sachez qu'à l'aube de ce nouveau millénaire, avec une informatisation toujours plus grande de la société, un[e] jeune scientifique ne sachant pas ou mal programmer sera handicapé[e], que ce soit bien clair. Enfin, comme chaque année, nombre d'entre vous souhaiteront bifurquer vers l'informatique à la fin de la seconde année [filière universitaire ou école d'ingénieur]. Ne négligez pas cette possibilité, laissez-vous le maximum de portes ouvertes, beaucoup de vocations sont tardives et le marché de l'emploi a ses réalités...

Au premier semestre, vous avez appris les rudiments du langage et les premiers algorithmes, notamment sur les tableaux. Au second semestre, vous allez approfondir ce travail préliminaire, et le compléter par un apprentissage de la récursivité, des interfaces graphiques et de types de données plus sophistiqués.

La page Internet associée à ce cours se trouve sur `http://deptinfo.unice.fr/~roy` à la rubrique MP1. Consultez-la régulièrement, vous y trouverez des renseignements pratiques, des compléments de cours impossibles à détailler dans ces feuilles, des réponses à vos questions posées en TP, quelques corrections, etc. La consultation de ces pages fait partie intégrante de cet enseignement. Votre travail ne se limitera donc pas aux séances de TP : les feuilles doivent être préparées, si possible sur machine, *avant* les séances, ou prolongées *après* les séances. Il faut qu'il soit bien clair que le seul travail en séance de TP ne donnera que peu de résultats en l'absence d'un cours en amphi, et l'on n'apprend pas à programmer en dilettante…

C. Organisation du travail

Sur votre disquette, créez un répertoire `A:\Java\MP1` de manière à ne pas vous mélanger avec les fichiers du Tronc Commun qui sont dans `A:\Java`. Copiez le fichier `A:\Java\Numerik.java` dans `MP1`. Editez la nouvelle copie, et ne gardez que les méthodes `main(…)`, `randomInt(a,b)` et `randomDouble(a,b)` qui doivent fonctionner parfaitement. Durant le travail en salles–machines, tous vos fichiers seront placés dans le dossier `A:\Java\MP1`.

D. Rappels sur les boucles et notion de complexité

Il y a 3 types de boucle en Java : `while`, `do…while`, et `for`. La plus facile à utiliser est la boucle `for`, surtout lorsqu'on connaît le nombre d'itérations à effectuer, ou lorsqu'on parcourt un tableau [en sortant avec un `break` afin la fin du tableau si nécessaire[1]].

Exercice 1.1 Dans la classe `Numerik`, rajoutez les méthodes de classe suivantes qui nous serviront souvent :

- Une procédure `printIntArray(String str, int[] tab)` affichant le tableau `tab` à l'horizontale. *Si elle est fournie*[2], la chaîne `str` sert de nom au tableau. Exemple : si `tab` est le tableau `{-3,5,2,0,6,8,1}`, alors `Numerik.printIntArray("t",tab)` produira l'affichage :
  `t[0]=-3 t[1]=5 t[2]=2 t[3]=0 t[4]=6 t[5]=8 t[6]=1`
  tandis que `Numerik.printIntArray(tab)` produira seulement les éléments du tableau sans nom ni indices :
  `-3 5 2 0 6 8 1`
  Dans chaque cas, l'affichage se terminera par un retour à la ligne.
- Une fonction `randomIntArray(int n, int a, int b)` prenant un entier `n = 1` et deux entiers `a = b` [ceci est garanti et n'est pas à tester] et dont le résultat est un tableau contenant `n` entiers choisis aléatoirement dans l'intervalle `[a,b]`. Utilisez `randomInt(…)`. Testez cette méthode dans le `main(…)` en utilisant `printIntArray(…)`.

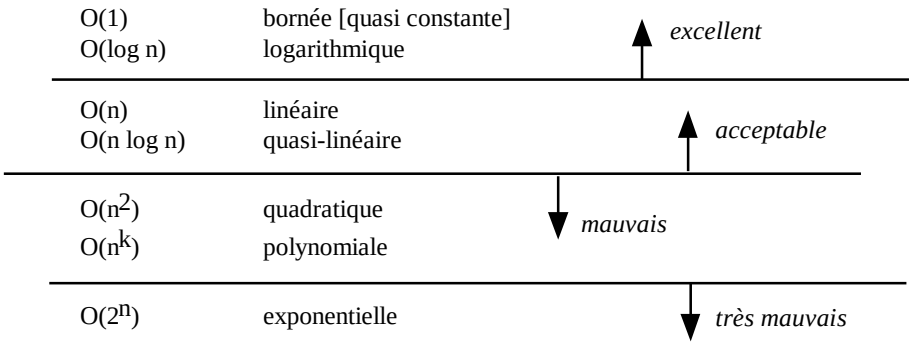
Exercice 1.2 *Rappels du Tronc Commun.* a) Toujours dans la classe `Numerik`, écrivez une fonction `ppdiv(int n)` prenant un entier `n = 2` et retournant le plus petit diviseur [*automatiquement premier*] de `n`. Si l'on ne trouve aucun diviseur jusqu'à $\sqrt{n}$, c'est que `ppdiv(n) == n` et que `n` est premier ! Ex : `ppdiv(45) == 3`, `ppdiv(43) == 43`.

b) En déduire, toujours dans `Numerik`, un prédicat `estPremier(int n)` prenant un entier `n = 2` et retournant `true` si et seulement si `n` est un nombre premier. Exemple : `estPremier(43) == true`.

Remarque : l'algorithme `ppdiv(n)` demandera donc *dans le pire des cas* un nombre de divisions proportionnel à $\sqrt{n}$. On parle alors de complexité en $O(\sqrt{n})$. La complexité se confond souvent mais pas toujours avec le temps de calcul. Ici, la division est l'opération la plus coûteuse. Dans le cas de l'algorithme `ppdiv(n)`, donc aussi dans `estPremier(n)`, le *pire des cas* a lieu lorsque `n` est premier, et le coût est alors de $\sqrt{n}$.

Définition : Par-rapport à un ensemble d'opérations de base, un algorithme portant sur une donnée de taille n a une **complexité** en $O(f(n))$ [ou simplement « est en $O(f(n))$ »], si *dans le pire des cas* il exige un nombre d'opérations de base *proportionnel* à `f(n)`, lorsque n est *grand*. Si dans tous les cas il exige un nombre d'opérations de base proportionnel à `f(n)`, il est dit en $\Theta(f(n))$, lire « Théta de `f(n)` ».
Il est clair que $\Theta(f(n)) \Longrightarrow O(f(n))$, donc qu'un Θ est plus précis qu'un O.

[1] On peut aussi utiliser l'instruction *break* pour sortir brutalement d'une boucle *while* ou *do…while*. Voir sur la page Web.
[2] *Si elle est fournie* : ceci signifie qu'il y aura en réalité deux méthodes de nom *printIntArray*, avec des paramètres distincts. C'est ce que l'on nomme la *surcharge* des méthodes [cf. la page Web].



• Le tableau ci-dessous montre l'évolution de la complexité en fonction de n pour quelques complexités usuelles. Les résultats ne valent que pour leurs comparaisons respectives bien entendu :

n	**log n**	**n log n**	**n^2**	**2^n**
10	0.003µs	0.033µs	0.1µs	1µs
20	0.004µs	0.086µs	0.4µs	1ms
100	0.007µs	0.644µs	10µs	$4x10^{13}$ ans
1 000 000	0.020µs	19.93ms	16.7min	

La définition mathématique précise, utilisée par Maple, de cette notation O(f(n)) est la suivante. Soient f et g deux fonctions positives définies sur ***N****. On dit que f est un O(g) s'il existe une constante k > 0 telle que pour n assez grand, on ait f(n) = k g(n). On note abusivement f = O(g) et on dit que « f est dominée par g ». Par exemple $2n^3 - 15 = O(n^3)$, mais n^2 est aussi un $O(n^3)$. Contrairement aux mathématiciens, les informaticiens diront qu'un algorithme est « en n^2 » et non « en $3n^2$ » ou « en $5n^2$ », en omettant la constante multiplicative. Bien entendu, il est parfois utile de disposer de cette constante pour comparer deux algorithmes en n^2 … Mais un programmeur ne sera vraiment satisfait que s'il passe de n^2 à n log n par exemple, plutôt que de $3n^2$ à $2n^2$.*

Quant à Q(g), on le définit par : f=Q(g) ⇔ f=O(g) et g=O(f) et on dit que « f et g ont le même ordre de grandeur »… Les relations O et Q sont réflexives et transitives, mais seule Θ est symétrique.

• Attention, un algorithme en $O(n^2)$ n'est meilleur qu'un algorithme en O(n log n) qu'à partir d'un certain rang, donc pour n assez grand. Lorsque les données sont de petites tailles [par exemple des tableaux de 10 éléments], les *mauvais* algorithmes sont souvent plus faciles à programmer et aussi efficaces sinon meilleurs. Mais lorsque les données sont grandes, les *mauvais* algorithmes deviennent très vite de *très mauvais* algorithmes !…

Exercice 1.3 A l'aide d'une boucle, calculez la valeur de `n` à partir de laquelle un algorithme dont la complexité exacte est `f(n) = n²/4` devient meilleur qu'un autre algorithme dont la complexité exacte est `g(n) = 200 + n log n`. *Réponse : un peu moins de 40…*

Exercice 1.4 a) Si l'on comptabilise le nombre de multiplications, montrez que la complexité de l'algorithme ci-dessous est *quadratique en n,* c'est-à-dire en $O(n^2)$. Et que calcule-t-il ?

```
int n = Console.readInt("Donnez la valeur de n");
double s = 0 ;
for(int k = 0 ; k < n ; k++)
  {  int f = 1 ;
     for(int i = k ; i > 0 ; i--)
       f = f * i ;
     s = s + 1.0/f ;
  }
return s ;
```

b) Donnez une version *linéaire* de cet algorithme, c'est-à-dire en O(n).