M1.23.3 Algorithmique

Travaux dirigés et pratiques du troisième semestre

Jean-François Remm
Jean-François Berdjugin
Jean-François Dufayard

M1.23.3 Algorithmique: Travaux dirigés et pratiques du troisième semestre

par Jean-François Remm, Jean-François Berdjugin, et Jean-François Dufayard

Table des matières

Liste des tableaux

Liste des exemples

Chapitre 1. Travaux Pratiques

Dans cette série de trois TPs (Travaux Pratiques), vous aller, principalement, coder, en *java* les algorithmes vus en TDs. L'ensemble des TPs doit être fait.

Début

Dans cette première partie nous allons écrire notre premier programme *java* qui affiche

```
Hello World
```

à l'écran.

Configuration de *Windows*

Pour travailler nous utiliserons le JDK Java Development Kit, l'environnement dans lequel le code *Java* est compilé pour être transformé en *bytecode* afin que la JVM (Machine Virtuelle de Java) puisse l'interpréter. Le JDK est aussi appelé Java 2 Software Development Kit.

Le JDK contient un ensemble d'outils parmi lesquels nous trouvons :

javac — le compilateur java,

jar — l'archiveur, qui met sous forme d'un paquetage les bibliothèques de classes relatives au projet fusionné en un fichier *jar*,

javadoc — le générateur de documentation, qui génère automatiquement de la documentation à partir des *annotations* du code source,

jdb — le débogueur,

jre — un ensemble d'outils permettant l'exécution de programmes Java sur toutes les plates-formes supportées et qui contient la machine virtuelle java.

Vous devez avoir d'installé sur votre machine, le jdk, il doit se trouver dans `C:\Program Files\Java`. Mais pour l'utiliser, il vous faut définir un ensemble de variables d'environnement :

PATH — qui contient la liste des dossiers dans lesquels Windows ira chercher les commandes par défaut.

CLASSPATH — qui permet lors de la compilation ou de l'exécution d'un programme à l'aide de Java, de spécifier l'emplacement des bibliothèques utilisées.

Pour ces TP, nous allons commencer par définir la variable `path` pour quelle contienne la valeur du répertoire contenant les exécutables *javac* et *java*. Il vous faut suivre la procédure suivante pour la définir :

1. clic droit sur « poste de travail »,
2. afficher les propriétés,
3. choisir l'onglet « avancé »,
4. cliquer sur « variables d'environnement »,
5. éditer ou créer la variable path en lui affectant le répertoire contenant javac et java, le séparateur est le point-virgule.

Tester en tapant dans une invite de commande **javac**.

Démonstration

Le langage java peut-être utilisé dans le domaine du Web pour exécuter du code sur le client :

applet	du code inclus dans une page Web
Java Web Start (JWS)	un moyen de déploiement d'applications via le web.

Le langage java peut aussi être utilisé pour s'exécuter sur les serveurs (Web dynamique) :

EJB	Enterprise Java Bean composant Java, réutilisable, destiné à être déployé sur des serveurs d'applications.
JSP	Java Server Page, technologie Java de création de pages dynamique en XHTML
Servlet	Le coeur, le programme java qui s'exécute sur le serveur.

Pour tester ces différentes technologies vous pouvez consulter les urls suivantes :

- `http://java.sun.com/products/javawebstart/demos.html`
- `http://meteo.fr`

Création des répertoires et premier programme

Nous allons structurer notre travail en répertoire, reproduisez en utilisant l'archive fournie, le modèle suivant sur votre partage (`Z:`) :

```
|-code
 |-bin
 |-boucles
 |-conditionnelles
 |-exemples
 |-perso
 |-tests
```

Placez-vous, à l'aide du ligne de commande (**cmd**), dans le répertoire `code/tests`, nous allons créer notre premier programme. Il est de tradition de commencer l'apprentissage d'un langage par la production d'un programme écrivant un texte sur la sortie standard. Écrire le fichier `Hello.java` suivant :

```
//Mon premier programme Java
//JFB

public class Hello
{
  public static void main (String[] args)
   { System.out.println("Hello World");
   }
}
```

Le compiler :

```
javac Hello.java
```

Charger le chier *Hello.class* avec le JDK

```
java Hello
```

Attention : notez bien que le chier obtenu s'appelle Hello.class, mais qu'on lance bien la commande java Hello.

Pour bien comprendre la distinction entre le code source et le bytecode, nous allons placer le bycode dans un autre répertoire en utilisant la ligne de commande suivante :

```
javac -d ../bin Hello.java
```

Ouvrer une autre invite de commande et rendez-vous dans `code/bin` puis taper

```
java Hello
```

. Dans la suite nous continuerons avec cette approche de séparation entre le code et le *bytecode*.

Premiers élements de syntaxe

Dans cette partie, nous allons apprendre à déclarer des variables, et à faire des affectations.

Commentaires

Les commentaires peuvent être en ligne // ou en bloc /* */. Tout ce qui suit le signe // ou qui est dans le bloc /* */ est considéré comme du commentaire c'est à dire que le compilateur n'en tient pas compte. Dans ces TP vous mettrez systématiquement un commentaire portant votre nom et un résume de votre programme.

Déclaration

En Java *TOUTES* les variables doivent être déclarées. Une déclaration précise un type et comporte une ou plusieurs variables de ce type, par exemple :

Exemple 1.1. Exemple de déclarations

```
int mini, maxi ; // mini et maxi sont deux entiers.
char c ; // c est un caractère
final double tva = 0.196 ; // tva est un réel dont la valeur (non modifiable) est 0.196
```

On peut répartir les variables entre les déclarations comme on le désire ; les déclarations précédentes auraient pu s'écrire :

```
int mini ; // un entier pour le mini
int maxi ; // un entier pour le maxi
```

Cette forme développée prend plus de place, mais elle permet d'ajouter un commentaire à chaque déclaration (ce qui facilite les modifications futures). On peut également initialiser les variables au moment où on les déclare.

```
int mini = 0 ;
int maxi = 10 ;
char c = 'c' ;
```

Affectation

Toute variable déclarée, doit être initialisée. Si l'initialisation n'est pas incluse dans la déclaration (comme cela a été fait ci-dessus) , il faut pas exemple ajouter cette initialisation :

```
mini = 0 ;
maxi = mini + 10 ;
c = 'c' ;
```

Lors des affectations, il vous faut respecter le *type* ou faire appel au *transtypage*.

Méthodes

L'élément de structuration en java, comme dans les autres langages orientés *objet* est la *class*. Les classes sont composée d'*attributs* et de *méthodes*. Java est fourni avec une API (Application Programming Interface) qui contient un ensemble de classes. Vous trouverez la documentation de l'API à l'url suivante: `http://docs.pedago.src`.

Ecrire à l'écran

Pour écrire à l'écran nous allons utiliser des méthode de l'API.

En suivant la documentation dans le package java.lang vous trouvez une classe System.

La classe System possède un attribut *static* nommé *out* de type *PrintStream*, un attribut *static* est accessible via le nom de la *class* suivie d'un point puis du nom de l'attribut. Ainsi nous pouvons y accéder via

```
System.out
```

La classe PrintStream possède un ensemble de méthode *static* (de *class*) dont des println(),

```
int a = 4;
System.out.println("Il est "+a+" heure");
```

nous affichera à l'écran,`il est 4 heures`. Tout comme pour l'attribut, l'accès à une méthode se fait avec le point.

Lire au clavier

Pour lire au clavier, nous utiliserons la *class* scanner présente dans le *package* java.util.

Cette classe ne possède pas d'attributs ou de méthode static, il nous faut donc en créer une instance :

```
Scanner clavier; //créer un objet de type scanner
clavier = new Scanner(System.in); //instancie clavier et le lie à l'entrée standard (le clavier)
int a;
a=clavier.nextInt(); //utilisation de la méthode nextInt() pour lire sur l'entrée standard.
```

Un petit programme pour lire et écrire au clavier

Dans le dossier `code/tests`, créer une *class* LireEcrire qui lit un entier puis un double et enfin affiche la somme des deux à l'écran. L'ossature de cette class ressemblera à :

```
import java.util.Scanner;

public class LireEcrire
{
 public static void main(String[] args)
 {

 }
}
```

L'*import* permet au compilateur de savoir où se trouve la *class* Scanner.
N'oubliez pas de compiler dans le répertoire courant en déposant vos bytecode dans code/bin, puis de tester dans code/bin.

Méthode statiques

Pour continuer à prendre des habitudes de programmation, nous allons séparer notre code en deux parties : les méthodes et leur test

Vous aller créer dans /code/conditionnelles une class Tp1 qui contiendra la méthode static static int prixPizza(int a) dont le code est le suivant :

```
public class Tp1
{
//calcul le prix de nb pizza
//sachant que la pizza vaut 10€ et que la dixième est gratuite
//
 static int prixPizza(int nb)
 {
   final int prixU=10;
   return nb*prixU-((nb/10)*prixU);
 }
}
```

Pour ne pas avoir à naviguer entre les répertoire, vous allez vous placer dans `code` compiler `conditionnelles/Tp1.java` et placer le résultat dans `code/bin` :

```
javac -d ../bin Tp1
```

Le programme que nous avons ne peut-être lancé, il ne contient qu'une méthode qui n'est pas un *main*. Nous allons créer une *class TestTp1* dans code/test dont le code est le suivant :

```
public class TestTp1
```

```
{
 public static void main(String[] args)
 {
  System.out.println("10 pizzas => " + Tp1.prixPizza(10));
  System.out.println("20 pizzas => " + Tp1.prixPizza(20));
 }
}
```

La compilation est obtenue en utilisant la ligne de commande suivante :

```
javac -classpath ".;bin" -d bin tests\TestTp1.java
```

Nous voyons apparaître le `classpath` qui indique au compilateur ou trouver Tp1.class. Tester avec :

```
java TestTp1
```

Nous pouvons pour ne pas avoir à spécifier à chaque fois le classpath définir le variable d'environnement `CLASSPATH` de Windows avec la valeur *.;bin*. Puis tester dans un nouveau terminal avec

```
javac -d bin tests\TestTp1.java
```

, le nouveau terminal est nécessaire car les variables d'environnement ne sont prise en compte qu'a l'ouverture.

Modifier la *class TestTp1* pour qu'elle demande une lecture au clavier du nombre des pizzas.

Conditionnelles

Element de syntaxe

La comparaison se fait de la manière suivante :

```
if (testBooléen) // testBooléen est un booléen
                 //ou un test dont le résultat est une valeur booléenne
traitement à effectuer si le test est vérifié
[else traitement à effectuer si le test n'est pas vérifié]
```

Ce qui est entre [ ] est optionnel. On l'utilise si on en a besoin. Exemple :

```
if (maxi > 5) // ici testBooléen est un test
              //dont le résultat est true si maxi est supérieur à 5, false sinon.
System.out.println("maxi est plus grand que 5");
else
System.out.println("maxi est plus petit ou égal que 5");
```

Je vous conseil d'utiliser systématiquement les accolades pour définir un bloc: if (cond) {...} else {...}

Exercices

Remise (Conditionnelle simple)

Écrire une méthode static *remise1* de Tp1 qui reproduise l'algorithme suivant :

```
Methode remise(montant: reel):reel
const tauxRemise <- 0.01 : réel
Si (montant>300) Alors
 montant <- montant * (1-tauxRemise)
Fin Si
Renvoie(montant)
Fin
```

Modifier la *class TestTp1* pour tester la méthode static remise.

Remise (Conditionnelles multiples)

Écrire une méthode static*remise2* de Tp1 qui reproduise l'algorithme suivant :

```
Methode remise2(montant: reel):reel
  var tauxRemise : réel
  var res : réel
  Si (montant>750) Alors
    tauxRemise <- 0.02
  Sinon
    Si (montant>300) Alors
      tauxRemise <- 0.01
    Sinon
      tauxRemise <- 0
    Fin Si
  Fin Si
  res <- montant * (1-tauxRemise)
  renvoie(res)
Fin methode
```

Maximum de trois entiers

Implanter la méthode static suivante (donnée en TD).

```
Methode max(a: entier, b:entier, c:entier):entier
  var max : entier
  max <- a
  Si (b>=max)
   max <- b
  Fin Si
  Si (c>=max)
   max <- c
  Fin Si
  renvoie(max)
Fin methode
```

Estimation du prix de revient d'un véhicule

Il existe un barème pour l'évaluation du prix de revient kilométrique des véhicules. Écrire une méthode *prixRevient* de la *class Tp2* effectuant le calcul de ce prix en fonction de *nbKm*, nombre de kilomètres parcourus . Règles :

Tableau 1.1. Prix de revient du véhicule

nb de km / puissance fiscale	5CV	6CV
jusqu'à 5000	nbKm * 0,43	nbKm * 0,47
de 5001 à 20000	nbKm * 0,23+1000	nbKm * 0,27 + 1000
au delà de 20000	nbKm*0,28	nbKm * 0,32

Ecrire une *class TestTp2* dans `code/tests` et tester.

Convertisseurs Euros/Francs

Vous pouvez tester le corrigé (après avoir téléchargé le cher : Convertisseur.jar)

```
java -jar Convertisseur.jar
```

1. Ecrire une methode qui résolve le problème de la conversion de francs <-> euros
2. Traduire cette méthode (sur papier)
3. Coder puis compiler
4. Tester

Monnayeur

Nous allons coder deux monnayeurs qui rendent des pièces de 5, 2 et 1 €.

Caisse illimitée

Ecrire et tester une méthode static caisseIllimite de Tp2 qui ne renvoie rien et qui affiche le nombre de pièce de 5, de 2 et de 1 à rendre.

Caisse limitée

Ecrire et tester une méthode static caisseLimite de Tp2 qui renvoie true si la monnaie est disponible et qui affiche le nombre de pièce de 5, de 2 et de 1 à rendre.

Comparaison de durées

On considère des durées notées avec des valeurs entières en heures, minutes et secondes (h,m,s). Exemple : d=(3,5,1) : 3 heures, 5 minutes et 1 seconde. Après lecture de deux durées d_1 et d_2 dans la *class TestTp2* :

- Écrire puis implanter une première méthode static (duree1 de Tp2) qui détermine la durée la plus courte, en convertissant les deux durées en secondes.
- Écrire puis implanter une deuxième static méthode (duree2 de Tp2) qui réalise la même action, mais sans convertir en seconde

Les méthodes doivent retourner -1 si la première date est plus courte, 0 si égalité et 1 sinon.

Problèmes du test d'égalité de flottants

Avant de passer au tp suivant consacré à l'étude des boucles, voici quelques exemples qui devraient vous prouver l'intérêt des entiers dans le comptage du nombre d'itérations. Il vous est demandé de compiler et exécuter et de comprendre les programmes suivants :

- Flottant1
- Flottant2
- Flottant3

Ces trois exemples sont tirés d'un point technique disponible sur le site http ://java.sun.com. Des explications complémentaires sont disponible dans la section 4.2.3 des spécifications du langage java et dans les documents IEEE 754 suivants.

Tri

Écrire un méthode static tri de Tp2 qui lit trois valeurs entières (a, b et c) et qui affiche ces trois valeurs dans un ordre croissant.

Boucles

Syntaxe

En java les structures de boucle sont le *while* et le *for*. Nous verrons le *for* plus tard. Le *while* s'utilise de deux manières differentes :

```
while (condition) {
  instructions
}

ou

do{
  instructions
}
while (condition);
```

La boucle est exécutée tant que la condition est vérifiée.

Première Boucle

Vous aller créer une *class* TestTp4 dans `code/tests` dans laquelle vous allez traduire l'algorithme suivant :

```
Algo Boucle
  var a : entier
Début
  a <- 5
  Tant que (a > 0)
    Ecrire(a)
    a <- a - 2
  Fin Tant que
Fin
```

Modier le code, pour pouvoir lire la valeur de a. Faire diérents tests avec a = 1, -1,... Modier la condition ; par exemple a= 0. Tester avec a = 2, 3,... Commentaires ?

Astuce

CTRL+C vous permet de sortir d'une boucle sans fin.

Lecture de caractère

Pour cette nouvelle partie, nous allons créer dans `code/boucles` une *class* Tp3 qui contiendra nos méthodes *static* et continuer avec la *class* TestTp4 dans `code/tests` qui nous permettra de tester nos méthodes.

Commençons, par lire un caractère et l'afficher jusqu'à ce que l'on saisisse le caractère 'y'. Faire deux versions, l'une où ce dernier caractère est affiché à l'écran, l'autre où il ne l'est pas.

La *class Scanner* ne posséde pas de méthode *nextChar()*, nous allons contourner le problème en lisant une chaine de caractères et en recupérant le premier caractère avec le code suivant :

```
Scanner cl = new Scanner(System.in);
char c = cl.next().charAt(0);
```

Lecture avec affichage du 'y' final

La méthode static ne prend pas de paramètres et ne retourne rien.

```
public static void LireChar1()
```

Lecture sans affichage du 'y' final

La méthode static ne prend pas de paramètres et ne retourne rien.

```
public static void LireChar2()
```

Somme des n premiers entier

Ecrire une méthode *static* qui retourne la somme des n premiers entiers.

```
public static int somme(int n)
```

Remboursement d'emprunt

Un emprunt ne peut être remboursé que si le remboursement annuel est supérieur au coût annuel de l'emprunt : emprunt * taux.

Chaqu'année, la valeur de l'emprunt est augmentée de son coût annuel et diminuée du remboursement.

Un emprunt est terminé lorsqu'il n'y a plus rien à rembourser.

Calcul du nombre d'année

Calculer le nombre d'années nécessaires au remboursement d'un emprunt à taux d'intérêt fixe et dont le remboursement annuel est fixe également. (Attention : le remboursement de la première année doit être strictement supérieur à l'intérêt payé la première année).

La méthode *static* devra retournée -1 si l'emprunt ne peut-être remboursé et le nombre d'années nécessaires sinon.

```
public static int remboursement1(double emprunt, double taux, double remboursement)
```

Calcul du coût

Même exercice, mais avec calcul du taux d'intéret effectif à savoir la parts des interêts dans la somme totale payée (somme des intérets divisée par l'emprunt initial).

```
public static double remboursement2(double emprunt, double taux, double remboursement)
```

Calcul de maximum

On lit des entiers jusqu'à lire la valeur -1. Déterminer la valeur maximale des valeurs lues (sans tenir compte du -1).

Exemple 1.2. Exemple séquence

```
4
17
13
-6
-1
Valeur max : 17
```

La fonction ne produira pas d'affichage et retournera le maximum :

```
public static int max()
```

Devine

Écrire deux méthodes qui fassent deviner en un nombre d'essais limités ou illimités un entier *aTrouver*. Pour chaque valeur entrée au clavier par l'utilisateur, le programmme renvoie des indications : trop grand, trop petit, gagné et perdu (dans le cas d'un nombre d'essais limités). Pour tirer un nombre entier au hazard vous pourrez utiliser :

```
int aTrouver = (int) (Math.random()*plage);
```

```
Math.random()
```

permet d'obtenir un nombre réél, compris entre 0 et 1, tiré au hazard.

```
Math.random()*plage
```

permet d'obtenir un nombre réél, compris entre 0 et *plage*, tiré au hazard. Enfin

```
(int) (Math.random()*plage)
```

permet le transtypage vers un entier.

Sans limite

Ecrire une première version avec un nombre d'essais illimité.

```
public static void devine(int plage)
```

Avec un nombre de coup limité

Ecrire une version avec avec `nbEssai` autorisés.

```
public static void devine(int plage, int nbEssai)
```

Décomposition en facteurs premiers

Pour cette nouvelle et dernière partie, nous allons créer dans `code/boucles` une *class* Tp4 qui contiendra nos méthodes *static* et une *class* TestTp4 dans `code/tests` qui nous permettra de tester nos méthodes.

Décomposer un nombre en nombre premiers. Essayer les divisions du nombre par les tous les entiers (à partir de 2) et faire afficher simplement les différent diviseur.

N.B. Pour simplier, on effectue les divisions du nombre par tous les entiers, qu'ils soient premiers ou non, de toute façon, un nombre qui n'est pas premier ne pourrait diviser car tous ses diviseurs (plus petit que lui) auraient précédemment divisé le nombre.

Exemple : pour 20, on exécute les divisions par 2 (oui), puis par 2 (oui), puis par 2 (non), puis par 3 (non), puis par 4 (non), puis par 5 (oui), ... On voit bien qu'on ne peut plus diviser par un nombre non pair (4) parce que le nombre de départ a déjà été divisé par ses diviseur (2 et 2).

La méthode prendra en paramétre le `nombre` à décomposer et ne retournera rien, les affichages se feront dans la méthode.

```
public static void decomposition(int nombre)
```

Monnayeur

Nous allons faire deux versions du monnayeur, la première avec une caisse illimitée et l'autre avec un caisse limitée. Les monnayeur ne connaissent que les pièces de 5, de 2 et de 1. Ils cherchent à rendre en premier les pièces de 5 puis celles de 2 et enfin celles de 1. Les implémentations seront basées sur des boucles.

Avec caisse illimitée

En vous aidant de l'algorithme suivant :

```
Algo Monnayeur
  var somme : entier
  var nb5,nb2,nb1 : entier
Début
  Lire(somme)
  nb5 <- 0 //initialisation
  nb2 <- 0 nb1 <- 0
  Tant que(somme >= 5)
    nb5 <- nb5 + 1
    somme <- somme - 5
  Fin Tant que
  Tant que(somme >= 2)
   nb2 <- nb2 + 1
   somme <- somme - 2
  Fin Tant que
  Tant que(somme >= 1)
   nb1 <- nb1 + 1 // on aurait pu faire une conditionnelle
   somme <- somme - 1
  Fin Tant que
  Ecrire("Il faut rendre : ")
  Ecrire(nb5 + "jetons de 5")
  Ecrire(nb2 + "jetons de 2")
  Ecrire(nb1 + "jetons de 1")
Fin
```

Ecrire une fonction qui prend en paramétres la somme à rendre. C'est fonction affichera, le nombre de pièces de 5, de 2 et de 1 rendues.

```
public static void monnayeur(int somme)
```

Avec caisse limitée

En vous aidant de l'algorithme suivant :

```
Algo Monnayeur
  var somme : entier
  var nb5,nb2,nb1 : entier
  var nb5Dispo, nb2Dispo, nb1Dispo : entier
Début
  Lire(somme)
  Lire(nb5Dispo)
  Lire(nb2Dispo)
  Lire(nb1Dispo)
  nb5 <- 0 //initialisation
  nb2 <- 0
  nb1 <- 0
  Tant que(somme >= 5 et nb5 < nb5Dispo)
    nb5 <- nb5 + 1
    somme <- somme - 5
  Fin Tant que
  Tant que(somme >= 2 et nb2 < nb2Dispo)
    nb2 <- nb2 + 1
    somme <- somme - 2
  Fin Tant que
  Tant que(somme >= 1 et nb1 < nb1Dispo )
    nb1 <- nb1 + 1
    somme <- somme - 1
  Fin Tant que
  Si (somme = 0) Alors
   Ecrire("Il faut rendre : ")
   Ecrire(nb5 + "jetons de 5")
   Ecrire(nb2 + "jetons de 2")
   Ecrire(nb1 + "jetons de 1")
   nb5Dispo <- nb5Dispo - nb5
   nb2Dispo <- nb2Dispo - nb2
   nb1Dispo <- nb1Dispo - nb1
 Sinon
  Ecrire("Impossible")
  Fin Si
Fin
```

Ecrire une fonction qui prend en paramétres la somme à rendre, le nombre de piéces de 5, de 2 et de 1 disponibles qui returne vraie si c'est possible et faux si c'est impossible. C'est fonction affichera, lorsque c'est possible, le nombre de pièces de 5, de 2 et de 1 rendues.

```
public static boolean monnayeur(int somme, int nb5Dispo, int nb2Dispo, int nb1Dispo)
```

Boucle "for"

La boucle *for* et une écriture condensée du *while*.

```
for (init;cond;post-traitement)
{
  traitement
}

équivaut à

init;
while (cond)
{
  traitement
  post-traitement
}
```

L'utilisation la plus courante est la suivante :

```
for (int i = 0; i < limite; i++)
{
    // instructions à exécuter
}
```

Dans cet exemple, `i` prendra les valeurs 0, 1, .., limite-1

Astuce

Il n'est pas nécessaire d'incrémenter la variable de boucle (`i`) dans la boucle

Si l'initialisation ou le post-traitement contiennent plusieurs instructions, il faut les séparer par des virgules.

Affichage des n premiers entier

Ecrire un algorithme à l'aide d'une boucle *for* qui affiche les n premiers entiers (de 1 à n ou de n à 1), le traduire, puis le tester.

```
public static void nPremiersEntier(int n)
```

Somme des n premiers entier

Ecrire un algorithme à l'aide d'une boucle *for* qui ache la somme des n premiers entiers (de 1 à n ou de n à 1), le traduire, puis le tester.

```
public static int sommeNPremiersEntier(int n)
```

Placement

Ecrire l'algorithme puis le programme répondant à la question suivante : Si l'on place `somme` au 1 janvier de l'année `anDepot` à `taux`% (en accumulant les intérêts), quelle va être la somme présente sur le compte le 1 janvier de de l'année `anRetrait` ?. Les variables somme, anDepot, anRetrait et taux sont à passée en paramétres.

```
public static double placement(double somme, int anDepot, int anRetrait, double taux)
```