Travaux pratiques de mathématiques appliquées

Algorithmes - Arithmétique modulaire

Durée prévue : 2 H

1. Introduction

1.1. Objectifs

Les objectifs principaux de ce laboratoire sont d' :

- implémenter des algorithmes mathématiques, sur base de différentes documentations (organigramme, pseudo-code, …) ;
- appliquer des notions de base de l'arithmétique modulaire vues au cours thorique (Ch. 3).

Ce laboratoire utilise et met en pratique les notions expliquées dans le chapitre 3 du cours théorique.

L'étudiant est supposé avoir révisé la matière de ce chapitre avant le début du laboratoire !

1.2. Evaluation

Les séances de travaux pratiques ont un caractère essentiellement formatif. Les exercices proposés dans cet énoncé ne seront pas notés. Néanmoins, il est fondamental que vous tentiez d'en résoudre un maximum par vous-même, ceci afin d'acquérir une bonne compréhension et une bonne maîtrise de la matière du cours.

Si vous éprouvez des difficultés, n'hésitez pas à solliciter l'aide de votre responsable de laboratoire. Vous pouvez également collaborer avec vos condisciples pour élaborer les solutions. Assurez-vous cependant que vous avez compris les notions mises en pratique. Ne vous contentez pas de recopier une solution sans la comprendre : vous devez être capable de la recréer par vous-même.

Les notions abordées dans ce laboratoire seront évaluées lors d'un examen pratique.

1.3. Consignes générales

Les exercices proposés dans cet énoncé doivent être résolus au moyen du langage Java et de l'environnement de développement Eclipse, en complétant le projet de départ

« **2425-B1MATH-MOD.zip** » mis à votre disposition sur HELMo Learn.

Le projet de départ contient toutes les classes citées dans cet énoncé avec, dans celles-ci, les prototypes des méthodes obligatoires que vous devrez compléter pour la réalisation des différentes tâches. Si nécessaire, vous pouvez enrichir le projet avec vos propres packages, classes et méthodes.

Chaque méthode est précédée d'une Javadoc. Lisez attentivement les informations qui y sont données en complément de l'énoncé.

Le projet de départ contient un plan de test pour la plupart des classes à compléter. Ce plan de test valide les éléments essentiels de votre solution, mais il n'a pas la prétention d'être exhaustif !

2. Algorithme d'Euclide - Algorithme d'Euclide étendu

2.1. Calcul du PDCG de deux naturels – Algorithme d'Euclide (15 min)

En mathématiques, **l'algorithme d'Euclide** permet de calculer le plus grand commun diviseur (PGCD) de deux naturels a et b.
Rappel : on appelle PGCD de a et de b le plus grand naturel qui divise a et b. Il est convenu que $PGCD(0,0) = 0$.

Dans ce laboratoire, nous allons implémenter l'algorithme standard, dit « par division euclidienne ». (Référence : Algorithme d'Euclide — Wikipédia).

Le procédé de calcul est le suivant :

1. On effectue la division euclidienne de a par b.
   Remarque : dans la suite, le reste est noté r (on n'utilisera pas le quotient).
2. On remplace ensuite a par b et b par r.
3. Tant que b est différent de 0, on reprend en 1.

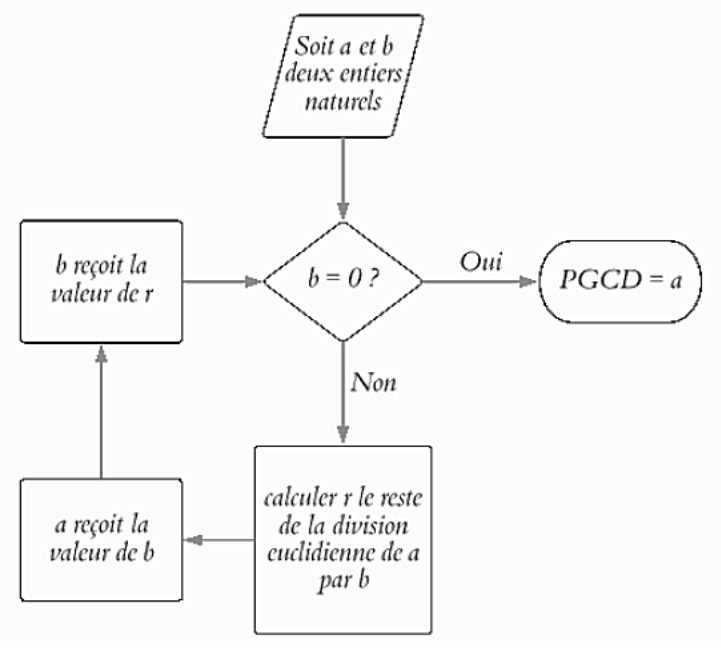
Après un certain nombre d'itérations, b est égal à 0 (ceci marque la fin de l'itération).

Le PGCD de a et b est alors le reste précédent (c'est-à-dire **le dernier reste non nul**).

Une description de l'algorithme d'Euclide appliqué à deux nombres entiers positifs a et b est donc :

- Si $b = 0$, l'algorithme se termine et renvoie la valeur a ;
- Sinon, l'algorithme calcule le reste r de la division euclidienne de a par b, puis recommence avec $a \coloneqq b$ et $b \coloneqq r$.

On peut également présenter cet algorithme sous la forme d'un organigramme de programmation :



Exercice 2.1 – Calcul du PDCG de deux naturels – Algorithme d'Euclide

- Dans la classe « `PgcdEuclide` » du package « `euclide` », complétez la méthode suivante :

  `public static int` **`pgcdEuclide`**`(int a, int b)`

  → Calcule le PGCD de a et de b de manière itérative, en suivant l'algorithme d'Euclide standard (par division euclidienne).
- Vérifiez que les entiers a et b fournis lors de l'appel sont bien des naturels. A défaut, la méthode doit lever une exception du type « `IllegalArgumentException` » au moyen de l'instruction « `throw new IllegalArgumentException(`*`<message>`*`)` ».
- Testez que votre méthode fonctionne correctement au moyen du plan de test JUnit « `PgcdEuclideTests` ».

2.2. Algorithme d'Euclide étendu (30 min)

En mathématiques, **l'algorithme d'Euclide étendu** est une variante de l'algorithme d'Euclide. Comme l'algorithme d'Euclide, il permet de calculer le PGCD de deux naturels a et b. Mais, en plus, il détermine les entiers u et v, solution de de l'équation :

$$a \,.\, u + b \,.\, v = PGCD(a, b) \quad \text{(équation de Bézout)}$$

Rappel : Lorsque a et b sont premiers entre eux, u est l'inverse modulaire de a modulo b.

Dans cet exercice, l'algorithme d'Euclide étendu sera implémenté de manière itérative, en se basant sur le tableau mis en place lors du cours théorique (Ch.3 - § 3.5.3.3).

Exercice 2.2 – Algorithme d'Euclide étendu

- Dans la classe « EuclideEtendu » du package « `euclide` », complétez la méthode suivante :

  `public static int` **`pgcdEuclideEtendu`**`(int a, int b)`

  → Calcule le PGCD de a et de b, de manière itérative, en suivant l'algorithme d'Euclide étendu.
- Dupliquez ensuite votre méthode et modifiez-la de manière à obtenir les 3 valeurs $PGCD(a, b)$, u et v . La signature de votre méthode sera la suivante :

  `public static int[]` **`EuclideEtendu`**`(int a, int b)`

  → Calcule le PGCD de a et de b, ainsi que les entiers u et v solution de l'équation $a \,.\, u + b \,.\, v = PGCD(a, b)$, de manière itérative, en suivant l'algorithme d'Euclide étendu.
- Vérifiez que les entiers a et b fournis lors de l'appel sont positifs. A défaut, la méthode doit lever une exception du type « `IllegalArgumentException` » au moyen de l'instruction « `throw new IllegalArgumentException(`*`<message>`*`)` ».
- Vérifiez que vos méthodes fonctionnent correctement en exécutant les plans de test JUnit « `PgcdEuclideEtenduTests` » et « `EuclideEtenduTests` ».

2.3. Solveur d'équation modulaire (25 min)

Une **équation modulaire** est une équation valide selon une congruence linéaire.

Exemples : $3x + 17 \equiv 8\ (mod\ 5) \implies x = 2$ car $3 . 2 + 17 \equiv 6 + 2 \equiv 8 \equiv 3\ (mod\ 5)$

$2x - 14 \equiv 6\ (mod\ 7) \implies x = 3$ car $2 . 3 - 14 \equiv 6 - 0 \equiv 6\ (mod\ 7)$

$2x + 17 \equiv 8\ (mod\ 6) \implies pas\ de\ solution$!

Dans cet exercice, nous allons nous limiter à des équations plus simples, les équations du type :

1. $\boldsymbol{x + a \equiv b\ (mod\ n)}$ $(n \geq 2)$, et
   Exemple : $x + 12 \equiv 3\ (mod\ 5) \implies x = 1$ car $1 + 12 \equiv 1 + 2 \equiv 3\ (mod\ 5)$
2. $\boldsymbol{ax \equiv b\ (mod\ n)}$ $(a > 0$ et $n \geq 2)$.
   Exemple : $3x \equiv 87\ (mod\ 5) \implies x = 4$ car $3 . 4 \equiv 12 \equiv 2 \equiv 87\ (mod\ 5)$

Remarque : la solution donnée sera dans l'intervalle $[0\ ;\ n[$.

MÉTHODOLOGIE

1. Une équation du type $\boldsymbol{x + a \equiv b\ (mod\ n)}$ peut être résolue simplement :
   $x \equiv b - a\ (mod\ n)$; on s'assure ensuite que : $x \in [0; n[$.
2. La résolution d'une équation du type $\boldsymbol{ax \equiv b\ (mod\ n)}$ est un peu plus compliquée ; nous aurons besoin des algorithmes mis en place aux points précédents.
   Nous allons implémenter une solution permettant la résolution du cas le plus simple ; à savoir lorsque $\boldsymbol{a}$ **et** $\boldsymbol{n}$ **sont premiers entre eux** ($PGCD(a, n) = 1$) car, alors, a possède un inverse modulo n. On peut donc écrire :
   $x \equiv a^{-1}b\ (mod\ n)$; on s'assure ensuite que : $x \in [0; n[$.
   Pour les autres situations, la méthode renverra un code d'erreur.

Exercice 2.3 – Solveur d'équation modulaire

- Dans la classe « `EquationModulaire` » du package « `equationModulaire` », complétez les méthodes suivantes en vous basant sur les éléments fournis ci-dessus :

  `public static int` **`solveurAddition`**`(int a, int b, int n)`

  → Détermine la solution de l'équation modulaire du type $x + a \equiv b\ mod\ n$.

  `public static int` **`solveurMultiplication`**`(int a, int b, int n)`

  → Détermine la solution la plus simple de l'équation modulaire du type $ax \equiv b\ mod\ n$.
  Dans les autres cas, la méthode renvoie la valeur -1.
- Vérifiez que le modulo n fourni lors de l'appel est supérieur à 2, et, pour la deuxième méthode, que le coefficient a est strictement positif. A défaut, la méthode doit lever une exception du type « `IllegalArgumentException` » au moyen de l'instruction « `throw new IllegalArgumentException(`*`<message>`*`)` ».
- Vérifiez que vos méthodes fonctionnent correctement en exécutant les plans de test JUnit.

3. Chiffrement par Modulo (30 min)

Le **chiffrement Modulo** utilise l'arithmétique modulaire sur une série de nombres, pour chiffrer un texte. Les caractères doivent donc être convertis en nombre, par exemple $A = 1$, $B = 2$, … $Z = 26$, mais une autre conversion numérique peut être utilisée.

Exemple : chiffrer « MATH » avec le modulo 26.

1. On convertit chaque lettre en un nombre : MATH $->$ $13, 1, 20, 8$ (avec la convention donnée ci-dessus) ;
2. Pour chaque nombre, on choisit un nombre dont la valeur du modulo est égale à celle du nombre à encoder ; en général, on fixe le nombre de chiffres de ces nombres.
   Ainsi un chiffrement de « MATH », avec des nombres de trois chiffres, peut être $299\ 105\ 358\ 216$, car $299 \equiv 13\ mod\ 26$, $105 \equiv 1\ mod\ 26$, … .

Le **déchiffrement** nécessite de connaître le modulo.

Exemple : déchiffrer $299\ 105\ 358\ 216$ avec le modulo 26.

1. Pour chaque nombre, on calcule le reste de la division euclidienne de ce nombre par le modulo donné. Ainsi le déchiffrement de $299\ 105\ 358\ 216$ est $13, 1, 20, 8$, car $299\ mod\ 26$ est 13, $105\ mod\ 26$ vaut 1 , … ;
2. On convertit alors chaque nombre en une lettre en appliquant la convention choisie : $13, 1, 20, 8$ $->$ MATH.

Exercice 3 – Chiffrement par Modulo

- Dans la classe « `ChiffrementModulo` » du package « `chiffrementModulo` », complétez les méthodes suivantes :

  `public static String` **`chiffrer`**`(String texte, int modulo, int nbChiffres)`

  → Chiffre un texte donné et le renvoie sous la forme d'une chaîne de nombres.

  Remarque : le texte donné ne présentera pas d'espaces entre les mots, ni de caractères spéciaux.

  `public static int` **`nombreCongru`**`(int nombre, int modulo, int nbChiffres)`

  → Génère un nombre aléatoire, de `nbChiffres` chiffres, congru au nombre donné modulo `modulo`.

  `public static int nbChiffres` **`dechiffrer`**`(String message, int modulo, int nbChiffres)`

  → Déchiffre un message donné sous la forme d'une chaîne de nombres.

  Remarque : le texte décodé sera en majuscules et ne présentera pas d'espaces entre les mots.

- Pour la méthode **`chiffrer()`**, vérifiez que le texte fourni lors de l'appel n'est ni *`null`* ni vide, que le modulo `modulo` est positif et que le nombre de chiffres souhaité `nbChiffres` est supérieur à 3. A défaut, la méthode doit lever une exception du type « `IllegalArgumentException` » au moyen de l'instruction « `throw new IllegalArgumentException(`*`<message>`*`)` ».

- Pour la méthode **`dechiffrer()`**, vérifiez que le message fourni lors de l'appel n'est ni *null* ni vide, que le modulo `modulo` est positif et que le nombre de chiffres souhaité `nbChiffres` est supérieur à 3. A défaut, la méthode doit lever une exception du type « `IllegalArgumentException` » au moyen de l'instruction « `throw new IllegalArgumentException(`*`<message>`*`)` ».

- Décodez le message suivant (modulo = 28) :

```
60901 12062 60737 75827 24512 16745 58449 28761 13402 26212 92449 44506 40699
87154 44552 82700 90091 12117 50306 17224 32177 83654 84070 72053 60821 20486
69432 40605 35438 67078 31809 35485 27062 67950 18509 51387 97713 37133 43349
36196 83584 67989 91719 25530 26993 92799 94493 51861 46583 39708 68577 68240
46837 15918 45869 45939 45516 61559 38213 92530 78149 62010 37062 81765 44288
32905 66294 63740 24085 87099 89146 69077 82299 52857 32849 91411 97249 20516
63284 10169 33846 32229 58247 98861 68073 73547 84522 82139 57823 80508 55809
80966 19619 48485 37933 32653 91824 38925 90515 51925 40657 12646 65497 71433
38015 78628 83557 65424 67761 69851 39527 63817 56378 50812 84593 19116 23929
80157 86837 57723 31178 33032 47881 29074 40293 19277 73697 29997 70327 91180
43629 40827 61419 89745 37009 55766 96993 88801 29737 36334 59169 51721 18706
44101 94295 90543 86469 66221 98414 50825 46035 16109 21042 85042 25293 43666
98121 73289 78433 45514 52044 99488 82101 67286 10462 12425 86617 27193 18303
83385 18418 77257 65590 79305 68982 61153 15982 32905 98647 45736 74485 63666
53709 89082 24688 63228 96437 67072 69445 38491 57224 76501 77591 28267 84133
55208 80085 59722 92972 85657 39388 44273 22386 21793 92334 67128 46205 56212
22384 75629 28664 71640 83525 12905 11107 43865 93149 50549 31505 42523 56020
72420 13669 13033 56457 47381 16198
```

4. La serrure modulaire (20 min)

Dans cet exercice, nous allons mettre en place un système de **serrure numérique**, basée sur l'arithmétique modulaire ; celle-ci sera utilisée pour générer et vérifier des codes d'accès.
Pour déverrouiller la serrure, l'utilisateur doit entrer un nombre qui satisfait à une condition modulaire précise, garantissant que seul celui qui connaît le secret (méthode de calcul de la réponse) peut accéder au système protégé par la serrure.
La démarche mise en place dans cet exercice est une variante simplifiée de certains protocoles cryptographiques utilisés en sécurité des systèmes.

PRINCIPE DE FONCTIONNEMENT

1. ***Choix d'un code secret***

   Le propriétaire de la serrure choisit un nombre, et le garde secret.

2. ***Choix d'une clé publique modulaire***

   Un modulo est choisi ; par exemple, un nombre premier.

3. ***Génération d'un code d'accès***

   Un code d'accès valide est obtenu en multipliant le nombre secret par un facteur aléatoire donné par la serrure, puis en lui appliquant le modulo choisi.

4. ***Vérification du code d'accès***

   L'utilisateur entre un code qui est vérifié en appliquant le même calcul modulaire. Si la condition est vérifiée, la serrure s'ouvre ; sinon, elle reste verrouillée !

Exemple :

3. Choix d'un nombre secret : **123** ;
4. Choix d'un modulo : **71** ;
5. Génération du code d'accès valide, si le facteur aléatoire est **13** : **37**
   car : $123 \,.13 \equiv 1599 \equiv 37 \ (mod\ 71)$
6. Vérification par la serrure :
   cas 1 : code d'accès = 37 : $37 \neq 123 \,.13 \equiv 1599 \ (mod\ 71)$ ⟹ Ok ! la serrure s'ouvre !
   cas 2 : code d'accès = 55 : $55 \neq 123 \,.13 \equiv 1599 \ (mod\ 71)$ ⟹ Ko ! la serrure reste verrouillée.

Exercice 4 – La serrure modulaire

- Dans la classe « `SerrureModulaire` » du package « `serrureModulaire` », complétez les méthodes suivantes :

  `public static boolean` **`verifierCode`**`(int nombreSecret, int facteur, int modulo, int codeEntree)`

  → Vérifie si un code donné est correct.

  `public static void` **`main`**`(String[] args)`

  → Fonction principale permettant de tester un code utilisateur en fonction des paramètres de la serrure modulaire.

  Voici des exemples d'exécution :

```
Entrez le nombre secret : 123
Entrez le modulo : 71
Le facteur est : 66

Entrez votre code : 32
Code incorrect. Accès refusé.
Plus que 4 tentatives !
Entrez votre code : 45
Code incorrect. Accès refusé.
Plus que 3 tentatives !
Entrez votre code : 70
Code incorrect. Accès refusé.
Plus que 2 tentatives !
Entrez votre code : 13
Code incorrect. Accès refusé.
Plus que 1 tentatives !
Entrez votre code : 26
Code incorrect. Accès définitivement refusé.
```

```
Entrez le nombre secret : 123
Entrez le modulo : 71
Le facteur est : 12

Entrez votre code : 56
Accès autorisé.
```

```
Entrez le nombre secret : 123
Entrez le modulo : 71
Le facteur est : 5

Entrez votre code : 48
Code incorrect. Accès refusé.
Plus que 4 tentatives !
Entrez votre code : 47
Accès autorisé.
```

Bon travail !