Travaux pratiques de mathématiques appliquées

Algorithmes itératifs et récursifs

Durée prévue : 3 H

1. Introduction

1.1. Objectifs

Les objectifs principaux de ce laboratoire sont :

- mettre en évidence la différence qui existe entre une méthode de résolution itérative et une méthode de résolution récursive ;
- concevoir des algorithmes récursifs d'une difficulté croissante ;
- appliquer l'approche récursive à la résolution du jeu « Le compte est bon ».

Ce laboratoire utilise et met en pratique les notions expliquées dans le chapitre 7 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-REC.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. Exercices de base

2.1. Somme des n premiers entiers impairs (20 min)

Considérons la somme des n premiers entiers impairs positifs ($n > 0$).
On peut démontrer que la valeur de cette somme est égale à :

$1 + 3 + 5 + \cdots + (2n - 1) = n^2$ (§ 7.1.3.3 – Exercice 1).

On peut également calculer la valeur de cette somme de manière itérative ou récursive.

Exercice 2.1 – Somme des nombres impairs

- Dans la classe « `SommeNombresImpairs` » du package « `sommeImpairs` », complétez les méthodes suivantes :

  `public static int` **`sommeNombresImpairsIteratif`**`(int n)`

  → Calcule la somme des n premiers entiers impairs, de manière itérative.

  `public static int` **`sommeNombresImpairsRecursif`**`(int n)`

  → Calcule la somme des n premiers entiers impairs, de manière récursive.
- Vérifiez que le paramètre n fourni lors de l'appel est valide. 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 « `SommeNombresImpairsIteratifTest` » et « `SommeNombresImpairsRecursifTest` ».

2.2. Termes d'une suite (20 min)

Considérons la suite définie, de manière récurrente, par :

$$\begin{cases} u_1 = 3 \\ u_{n+1} = 2u_n + 4 \ \ (n \in \mathbb{N}_0) \end{cases}$$

Comment peut-on exprimer, ***de manière générale***, le n^{ième} terme de cette suite ?

Dans cet exercice, nous souhaitons déterminer et afficher les n termes de la suite décrite ci-dessus de manière itérative puis de manière récursive.

Exercice 2.2 – Termes d'une suite

- Dans la classe « `TermesSuite` » du package « `suite` », complétez les méthodes suivantes :

  `public static long` **`termesSuiteIteratif`**`(int n)`

  → Calcule et affiche les n premiers termes de la suite, de manière itérative.

  `public static long` **`termesSuiteRecursif`**`(int n)`

  → Calcule et affiche les n premiers termes de la suite, de manière récursive.

- Vérifiez que le paramètre n fourni lors de l'appel est valide. 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 « `TermesSuiteIteratifTest` » et « `TermesSuiteRecursifTest` ».

2.3. Amorce d'une chaîne de caractères (20 min)

Dans cet exercice, nous souhaitons déterminer si une chaîne de caractères commence par une amorce donnée, d'abord de manière itérative et ensuite de manière récursive.
Remarque : cette analyse ne sera pas sensible à la casse.

Exemples : `amorceIdentique("Chateau", "chat")` → true
`amorceIdentique("chateau", "choc")` → false

Exercice 2.3 – Amorce d'une chaîne de caractères

- Dans la classe « `AmorceIdentique` » du package « `amorce` », complétez les méthodes suivantes :

  `public static boolean` **`amorceIdentiqueIteratif`**`(String chaine, String amorce)`

  → Détermine si la chaîne `chaine` commence par `amorce`, de manière itérative.

  `public static boolean` **`amorceIdentiqueRecursif`**`(String chaine, String amorce)`

  → Détermine si la chaîne `chaine` commence par `amorce`, de manière récursive.

- Vérifiez que les paramètres `chaine` et `amorce` fournis lors de l'appel sont valides. 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 « `AmorceIdentiqueIteratifTests` » et « `AmorceIdentiqueRecursifTests` ».

2.4. Calcul du PGCD de deux nombres naturels (20 min)

Le PGCD (**Plus Grand Commun Diviseur**) de deux nombres naturels, *quand au moins l'un des deux n'est pas égal à zéro*, est le plus grand entier qui divise en même temps chacun de ces nombres.

Exemples

- Le PGCD de 18 et 24 est 6.
- Le PGCD de 84 et 275 est 1 ! (84 *et* 275 *sont premiers entre eux*)
- Le PGCD de 675 et 375 est 75.

Ce dernier exemple montre que chercher tous les diviseurs communs à deux nombres peut être long. Une autre méthode, classique, est l'**algorithme d'Euclide** ; on procède par divisions euclidiennes successives. La version récursive vous a été présentée lors du cours théorique.

La méthode de détermination du PGCD de deux nombres naturels que nous allons implémenter dans ce laboratoire est l'utilisation de **l'algorithme des soustractions successives**.
Cette méthode repose sur le principe suivant : « *si un nombre est un diviseur de deux nombres p et q tels que $p > q$, alors il est aussi diviseur de la différence $p - q$* » (voir syllabus 3.3.2).
Son intérêt est qu'il remplace le calcul du PGCD de deux nombres par le calcul du PGCD de deux nombres plus petits. On peut alors calculer un PGCD en utilisant plusieurs fois d'affilée ce principe (=> méthode des soustractions successives).

Exemple : calcul du PGCD de 64 et 36 :

$$64 - 36 = 28 => PGCD(64\,;36) = \text{PGCD}(36\,;\,28)$$
$$36 - 28 = 8 \;\; => PGCD(36\,;28) = \text{PGCD}(28\,;\,8)$$
$$28 - 8 = 20 \;\; => PGCD(28\,;8) = \text{PGCD}(20\,;\,8)$$
$$20 - 8 = 12 \;\; => PGCD(20\,;8) = \text{PGCD}(12\,;\,8)$$
$$12 - 8 = 4 \;\;\;\; => PGCD(12\,;8) = \text{PGCD}(8\,;\,4)$$
$$8 - 4 = 4 \;\;\;\;\; => PGCD(8\,;4) = \text{PGCD}(4\,;\,4)$$
$$et\ PGCD(4\,;4) = 4$$
$$=> PGCD(64\,;36) = 4$$

Cette méthode est assez simple d'un point de vue du calcul (uniquement des soustractions).
Elle peut cependant être assez longue dans certains cas (Exemple : P$GCD\,(100\,;\,96)$?!) ; l'algorithme d'Euclide est plus rapide.

Ainsi, pour calculer le PGCD de deux nombres naturels de manière ***itérative***, et en utilisant **l'algorithme des soustractions successives**, il faut :

1. Si p est nul, alors renvoyer q ($PGCD(p\,;0) = p$) ;
2. Si q est nul, alors renvoyer p ($PGCD(0\,;q) = q$) ;
3. Calculer la différence $p - q$;
4. Tant que p et q sont différents, recommencer le processus avec $p = p - q$ si $q < p$ ou $q = q - p\ si\ q > p$;
5. Si $p = q$, alors $PGCD\,(p\,;q) = p\ (= q)$.

Un organigramme est présenté à la Figure 1.

Cette méthode peut aussi s'écrire de manière ***récursive*** :

$$PGCD\,(p\,;\,q) = \begin{cases} q & si\ p = 0 \\ p & si\ q = 0 \\ p & si\ p = q \\ PGCD\,(p - q\,;\,q) & si\ q < p \\ PGCD\,(p\,;\,q - p\,) & sinon \end{cases}$$

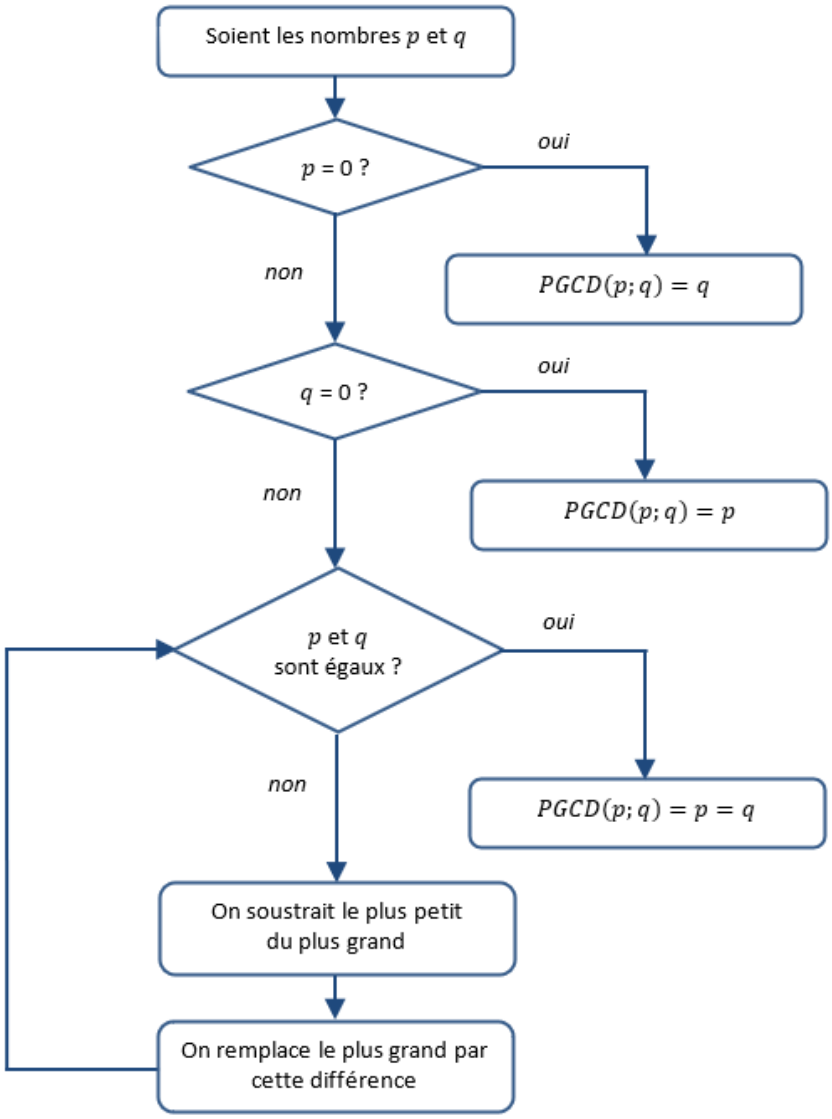


Figure1. Organigramme de l'algorithme du calcul du PGCD de deux naturels par soustractions successives.

Exercice 2.3 – Calcul du PGCD de deux nombres naturels

- Dans la classe « `Pgcd` » du package « `pgcd` », complétez les méthodes suivantes :

  `public static int` **`pgcdIteratif`**`(int p, int q)`

  → Calcule le PGCD de p et de q , de manière itérative.

  `public static int` **`pgcdRecursif`**`(int p, int q)`

  → Calcule le PGCD de p et de q , de manière récursive.

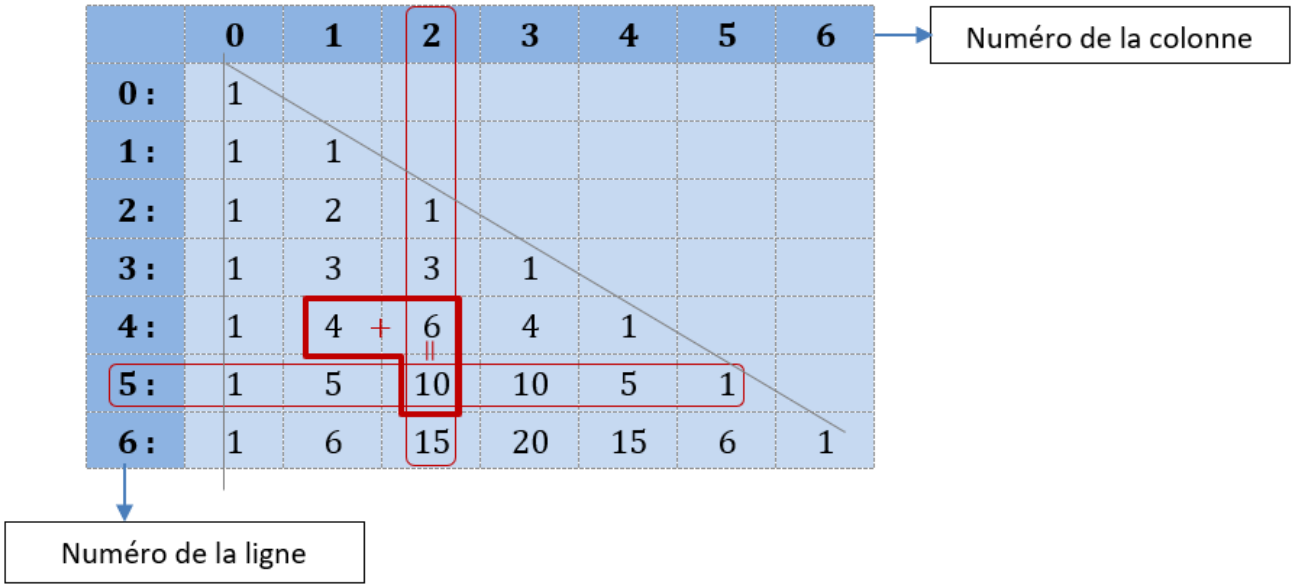
- Vérifiez que les paramètres p et p fournis lors de l'appel sont valides. A défaut, la méthode doit lever une exception du type « `IllegalArgumentException` » au moyen de l'instruction « `throw new IllegalArgumentException(`*`<message>`*`)` ».

- Testez que vos méthodes fonctionnent correctement au moyen des plans de test JUnit « `PgcdIteratifTest` » et « `PgcdRecursifTest` ».

3. Triangle de Pascal (45 min)

Le triangle de Pascal est le tableau des coefficients qui sont utilisés pour le développement de certaines expressions comme : $(a+b)^2$ ou $(a+b)^n$ (= binôme de Newton) ; ces coefficients s'appellent les "coefficients binomiaux".

Ce triangle est le suivant :



Le numéro de la ligne de ce triangle est la puissance à laquelle $a+b$ est élevée ; par exemple,

$(a+b)^0 = 1$; $(a+b)^2 = 1.a^2 + 2.a.b + 1.b^2$;

$(a+b)^5 = 1.a^5 + 5.a^4.b + 10.a^3.b^2 + 10.a^2.b^3 + 5.a.b^4 + 1.b^5$

On obtient chaque coefficient en additionnant le nombre qui est situé au-dessus de lui dans la même colonne *(ligne-1)* et celui qui est situé au-dessus de lui dans la colonne de gauche (*ligne-1 colonne-1*).

Exemple : le nombre se trouvant à l'intersection de la ligne 5 et de la colonne 2, soit 10, est égal à la somme du nombre se trouvant à l'intersection de la ligne 4 et de la colonne 2, soit 6, et du nombre se trouvant à l'intersection de la ligne 4 et de la colonne 1, soit 4.

On remarque que la première colonne (*colonne 0*) est toujours à 1, ainsi que la diagonale.

Dans cet exercice, nous souhaitons écrire une méthode qui retourne le coefficient binomial correspondant à une ligne et à une colonne donnée, d'abord de manière itérative et ensuite de manière récursive. Par exemple, si l'on souhaite connaître le coefficient qui se trouve à l'intersection de la ligne 4 et de la colonne 2, la méthode retourne le nombre 6.

Ensuite, le programme affichera en console les 17 premières lignes (0-16) du triangle de Pascal. Voici les 2 dernières lignes :

```
1     15    105    455    1365   3003   5005   6435   6435   5005   3003   1365   455    105    15     1
1     16    120    560    1820   4368   8008   11440  12870  11440  8008   4368   1820   560    120    16     1
```

Indications :

- la version ***itérative*** peut implémenter la formule suivante :

$$C(n,p) = n.\frac{n-1}{2}.\frac{n-2}{3}.\ldots.\frac{n-p+2}{p-1}.\frac{n-p+1}{p}$$

- la version ***récursive*** s'appuie sur la définition mathématique récursive suivante :

$$C(n,p) = \begin{cases} 1 & si\ p = 0\ ou\ p = n \\ C(n-1,p) + C(n-1,p-1) & sinon \end{cases}$$

Dans ces formules, n désigne le numéro de la ligne et p est le numéro de la colonne.

Exercice 3 – Triangle de Pascal

- Dans la classe « `TrianglePascal` » du package « `trianglePascal` », complétez les méthodes suivantes :

  `public static int` **`coefficientPascalIteratif`**`(int ligne, int colonne)`

  → Calcule le coefficient binomial correspondant à la ligne et à la colonne données, de manière itérative.

  `public static int` **`coefficientPascalRecursif`**`(int ligne, int colonne)`

  → Calcule le coefficient binomial correspondant à la ligne et à la colonne données, de manière récursive.

- Vérifiez que les paramètres fournis lors de l'appel sont valides. 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 « `TrianglePascalIteratifTests` » et « `TrianglePascalRecursifTests` ».
- Complétez la méthode « `main()` » de la classe de manière à afficher dans la console les dix-sept premières lignes du triangle de Pascal (voir résultat ci-dessus), une première fois en appelant la méthode itérative de calcul des coefficients binomiaux, et une seconde fois en appelant la méthode récursive de calcul des coefficients binomiaux.

4. Le compte est bon (1 heure 15)

4.1. Introduction

Le **compte est bon** est un jeu mathématique issu de l'émission « Des chiffres et des lettres » (créé par A. Jammot en 1972). Son but est d'obtenir un nombre (de 101 à 999) à partir des quatre opérations élémentaires (addition, soustraction, multiplication et division) et de six nombres tirés au hasard parmi les 24 suivants : les nombres de 1 à 10 présents deux fois, 25, 50, 75 et 100. Tous les résultats des calculs intermédiaires doivent être des *entiers naturels*. Chacun des nombres (nombres de départ et résultats intermédiaires) ne peut être utilisé qu'une seule fois. Si le résultat ne peut être atteint, il faut s'en approcher le plus possible. (*Remarque : cette règle ne sera pas implémentée dans notre version du jeu*).
(Référence : Des chiffres et des lettres — Wikipédia).

Dans cet exercice, nous allons implémenter un solveur basé sur une recherche récursive. L'algorithme mis en place utilisera une approche de *backtracking* pour explorer toutes les combinaisons possibles et trouver la solution exacte. Il proposera la première solution adéquate. Note : ce n'est donc pas nécessairement la solution la plus rapide (nombre d'étapes minimum).

Exemples

1) Obtenir 123 avec 4, 5, 6, 7, 8 et 9 :
   => 3 opérations : 8+5 = 13 ; 13 x 9 = 117 ; 117 + 6 = 123.
   ou
   => 5 opérations : 4+5 = 9 ; 6 x 7 = 42 ; 8 x 9 = 72 ; 9 + 42 = 51 ; 51 + 72 = 123

2) Obtenir 952 avec 3, 6, 25, 50, 75 et 100
   => 5 opérations : 100 + 3 = 103 ; 103 x 6 = 618 ; 618 x 75=46350 ; 46350 / 50927 ;
   927+25 = 952.

4.2. Algorithme de résolution

L'algorithme que nous allons mettre en œuvre pour résoudre un « **compte est bon** » est basé sur une recherche récursive.

Le principe général est de tester toutes les paires de nombres que l'on peut composer à partir de la liste de 6 nombres et d'effectuer toutes les opérations (+, -, *, / si exact) avec ces 2 nombres ; si le résultat d'une opération est un entier strictement positif, on remplace, dans la liste, les 2 nombres par le résultat obtenu et on recommence avec la nouvelle liste (plus courte d'un élément !), et ainsi de suite, jusqu'à obtenir une liste d'un seul nombre. Si ce nombre correspond au nombre recherché, alors c'est gagné ; sinon, on n'a pas trouvé de solution.

La méthode de résolution proposée fournit les nombres que l'on peut utiliser dans un tableau.

L'algorithme peut être décrit de la manière suivante :

- *[Cas de base] :* le tableau comporte un seul nombre → si ce nombre correspond au nombre recherché, alors c'est gagné et on renvoie la chaîne de caractères « Le compte est bon ! » ; sinon, on n'a pas trouvé de solution et on renvoie *null* ;
- *[Règle de récurrence]* :
  - pour chaque paire de nombres, on effectue toutes les opérations arithmétiques possibles ; pour limiter les opérations inutiles, on commencera par ordonner les nombres de la paire traitée (a > b) ;
  - si le résultat de l'opération arithmétique effectuée est un entier strictement positif, on appelle la méthode récursive en lui fournissant un nouveau tableau contenant :
    - les nombres du tableau précédent sauf les nombres constituant la paire testée et
    - le résultat de l'opération arithmétique qui vient d'être réalisée.

    et si la chaîne de caractères renvoyée par la méthode récursive est différente de *null*, on renvoie la chaîne de caractères décrivant les opérations arithmétiques qui conduisent à la solution proposée ;
  - si toutes les solutions ont été testées et qu'aucune solution n'a été trouvée, l'algorithme renvoie *null*.

Exercice 4 – Le compte est bon

- Dans la classe « `LeCompteEstBon` » du package « `jeu` », complétez les méthodes suivantes :

  `public static String` **`solveur`**`(int[] tabNombres, int nombreRecherche)`

  → Recherche de manière récursive une solution pour atteindre le nombre recherché en combinant les nombres donnés avec les opérations arithmétiques de base (+, -, *, /).

  `private static int` **`calculer`**`(int nombre1, int nombre2, char operation)`

  → Applique l'opération arithmétique choisie aux deux nombres et renvoie le résultat ou -1 si l'opération n'est pas permise.

  NB : la méthode `calculer` recevra nombre 1 > nombre2.

  `private static int[]` **`tirageNombres`**`(int[] listeNombre, int nombre)`

  → Sélectionne aléatoirement `nombre` nombres distincts dans la liste fournie.

- La classe « `LeCompteEtBon` » définit la constante OPERATIONS→ +, -, *, / ;pensez à l'utiliser !

- Complétez la méthode « `main()` » de la classe de manière à afficher dans la console :
  - les six nombres résultant du tirage aléatoire ;
  - le nombre à atteindre (résultant d'un tirage aléatoire ; doir être compris entre 101 et 999)
  - la solution proposée (étapes de calculs) si une solution existe, ou « Pas de solution » sinon.

  Voici des exemples d'exécution :

```
Nombres de départ : [3, 6, 25, 50, 75, 100]
Nombre recherché : 952
solution trouvée :
75 * 3 = 225
100 + 6 = 106
225 * 106 = 23850
23850 - 50 = 23800
23800 / 25 = 952
Le compte est bon !
```

```
Nombres de départ : [2, 4, 3, 2, 7, 9]
Nombre recherché : 999
Pas de solution
```

Bon travail !