SAE S1.02 – Comparaison d'approches algorithmiques

1. La description officielle au Programme National

Face à un problème algorithmique complexe, l'étudiant doit proposer plusieurs solutions. Chaque solution doit être comparée aux autres sur le plan de l'efficacité en temps d'exécution.

La situation professionnelle est celle du développeur au sein d'une équipe où ce dernier doit résoudre un problème de lenteur d'exécution d'une application.

2. Objectif

L'objectif principal est la mise en œuvre de différents algorithmes pour résoudre un même problème. Par comparaison d'approches algorithmiques distinctes, il est demandé de mettre en évidence par des mesures empiriques la rapidité d'exécution des différents algorithmes.

Il est fortement conseillé de s'inspirer pour cela des techniques de comparaison enseignées dans la ressource R1.01.P2.

3. Le jeu de Grundy avec Intelligence Artificielle (IA) de la machine

Le jeu de Grundy est une variante du jeu de Nim inventée en 1939 par Patrick Grundy. Le jeu se joue à deux joueurs. La position de départ consiste en un unique tas d'objets (des allumettes ou des pions, par exemple). Le seul coup disponible pour les joueurs consiste à séparer un tas d'objets en deux tas de tailles **distinctes**. Les joueurs jouent à tour de rôle, jusqu'à ce que l'un d'entre eux ne puisse plus jouer (c'est le perdant).

Voici un exemple complet où les « bâtons » représentent des allumettes et où les joueurs sont nommés *Jou1* et *Jou2* :

Au départ il y a un seul tas de 7 allumettes : **| | | | | | |**

Jou1 sépare le tas « 7 » en 2 et 5 : **| |** et **| | | | |**

Jou2 sépare le tas « 5 » en 1 et 4 : **| |** et **|** et **| | | |**

Jou1 sépare le tas « 4 » en 1 et 3 : **| |** et **|** et **|** et **| | |**

Jou2 sépare le tas « 3 » en 1 et 2 : **| |** et **|** et **|** et **|** et **| |**

Le jeu est terminé et *Jou2* a gagné car il reste à ce stade 5 tas : 3 de 1 allumette et 2 de 2 allumettes. Il y a donc impossibilité de jouer pour *Jou1* car aucun des tas ne peut être séparé en 2 tas de tailles distinctes. *Jou1* a perdu… *Game Over*…

L'objectif de cette SAÉ est d'écrire un programme java qui permet de jouer des parties du jeu de Grundy entre un joueur et **la machine**.

Le jeu de Grundy que vous allez étudier au niveau de son efficacité dans cette SAE1.02, va se baser sur une version **récursive** qui met en œuvre une théorie qui garantit de jouer en choisissant toujours la solution (décomposition) gagnante si elle existe (sinon, la machine choisira une décomposition en tas d'allumettes au hasard).

Nous savons qu'il existe une version **non récursive** (pas forcément simple à comprendre) qui permet également à la machine de gagner : celle-là **nous ne l'étudierons PAS**.

4. Travail demandé

4.1. Le point de départ

Le point de départ du travail est une version récursive de base (appelée *GrundyRecBrute.java*) qui contient une méthode fondamentale dans un jeu « joueur contre machine avec IA » et qui se nomme : *boolean jouerGagnant (ArrayList<Integer> jeu)*.

Etant donné l'état du jeu passé en paramètre à la méthode (type *ArrayList*), cette méthode renvoie un booléen qui est à vrai si le coup suivant (i.e. la décomposition suivante en plusieurs tas d'allumettes) est gagnant à 100% pour la machine (et dans ce cas le coup gagnant est joué par la machine). Si ce booléen est à faux, la machine jouera au hasard (et dans ce cas à vous d'écrire le code pour avoir un jeu fonctionnel).

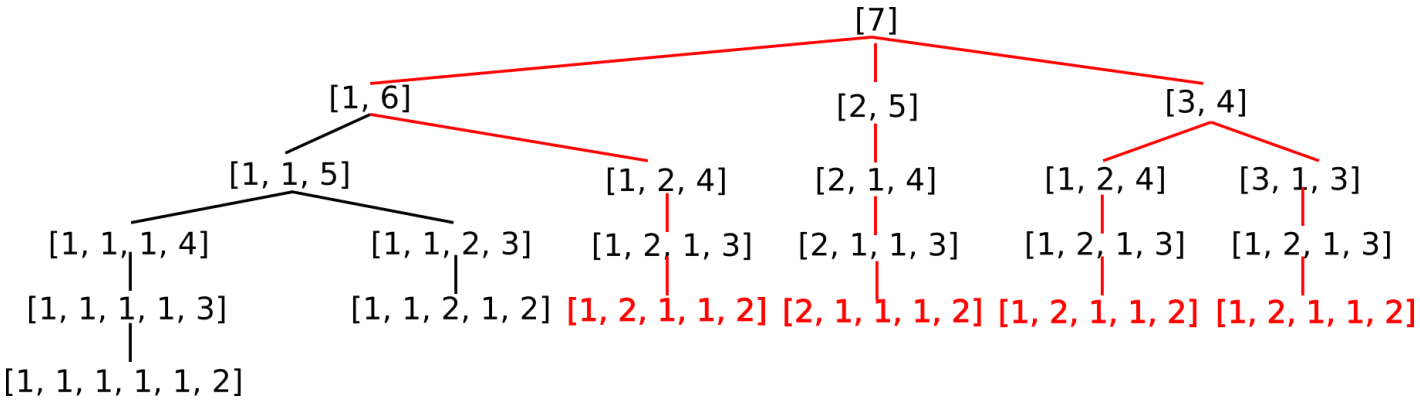
Cette méthode (*jouerGagnant*) se base sur 2 règles **fondamentales** qui proviennent de ce document (http://mathenjeans.free.fr/amej/edition/9903grun/grundy2.html) et qui sont :

1. **Une situation (ou position) est dite perdante** pour la machine (ou le joueur, peu importe) si et seulement si TOUTES ses décompositions possibles (c-à-d TOUTES les actions qui consistent à décomposer un tas en 2 tas inégaux) sont TOUTES gagnantes pour l'adversaire. En d'autres mots, quel que soit l'action mise en œuvre (parmi toutes celles qui sont possibles), AUCUNE n'empêchera l'adversaire de gagner.
2. **Une situation (ou position) est dite gagnante** pour la machine (ou le joueur, peu importe), s'il existe AU MOINS UNE décomposition (c-à-d UNE action qui consiste à décomposer un tas en 2 tas inégaux) perdante pour l'adversaire.

La méthode *jouerGagnant* appelle la méthode *estPerdante* sur toutes les décompositions possibles. Dès qu'une décomposition est perdante (pour l'adversaire), elle est jouée (par la machine), la machine est sûre de gagner et la méthode renvoie vrai. Sinon la méthode renvoie faux (aucune décomposition n'est perdante pour l'adversaire) et la machine n'effectue aucune décomposition (c'est donc à vous dans ce cas de faire une décomposition au hasard pour la machine).

Seule la méthode *estPerdante* est **récursive**.

Voici un exemple de toutes les décompositions possibles à partir d'un tas de 7 allumettes.



Les chemins en rouge indiquent que le tas de 7 allumettes est nécessairement perdant pour le joueur qui démarre (appelé « joueur 1 ») car les 3 décompositions possibles le mènent dans tous les cas à sa perte :

- Si le joueur 1 joue [1, 6], le joueur 2 (qui jouera intelligemment) divise le tas de 6 en [2, 4] pour que le joueur 1 perde à tous les coups. En effet, le joueur 1 ne peut que diviser le tas de 4 (allumettes). La position devient alors [2, 1, 3]. Le joueur 2 divise le tas de 3 et bloque nécessairement le joueur 1 qui a perdu.
- Si le joueur 1 joue [2, 5], le joueur 2 ne peut que diviser le tas de 5 en [1, 4]. Le joueur 1 est alors obligé ensuite de décomposer [4] en [1, 3] ce qui l'amène à nécessairement perdre (attention, dans le schéma ci-dessus, la décomposition [2, 2, 3] à partir de [2, 5] a été oubliée, c'est une erreur mais ça ne change rien au fait que [7] est bien perdant).
- Si le joueur 1 joue [3, 4] :
  - soit le joueur 2 divise le tas de 3. La configuration devient [1, 2, 4]. Le joueur 1 ne peut que diviser le tas de 4, [1, 2, 1, 3]. Le joueur 2 divise le tas de 3. Le joueur 1 a nécessairement perdu.
  - soit le joueur 2 divise le tas de 4. La configuration devient [3, 1, 3]. Le joueur 1 ne peut que diviser un des tas de 3. Le joueur 2 divise l'autre tas de 3. Le joueur 1 a nécessairement perdu.

On notera donc que [1] et [2] sont des tas nécessairement perdants puisque indivisibles en tas inégaux.

Vocabulaire & Structure de données

Vocabulaire :

- on appelle « tas » un simple tas d'allumettes (exemples [4], [10], [15], …)
- on appelle « situation » ou « configuration » ou « position » ou « décomposition » un jeu décomposé en plusieurs tas à un certain moment de la partie (exemples pour un tas initial de [12] allumettes on peut avoir les situations [1, 4, 5, 2], [1, 3, 5, 2, 1], [1, 4, 4, 2, 1], [1, 4, 3, 1, 2, 1], …)

Chaque tas d'allumettes est un entier mémorisé dans un tableau dynamique de type *ArrayList<Integer>*. Regardez dans l'API Java (https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) les méthodes de *ArrayList* à votre disposition et en particulier : *add*, *clear, set* et *get*. Pour afficher le contenu d'une *ArrayList*, il suffit d'écrire l'instruction *System.out.println (maListe.toString())*. Une situation (+sieurs tas) est donc mémorisée dans une *ArrayList<Integer>*.

Efficacité

Pour l'évaluation de l'efficacité, on ne vous demandera jamais de calculer f(n) exacte et d'arriver à une expression de Θ. L'expression de f(n) est trop complexe à obtenir (dans le cadre de cette récursivité).

Pour chaque version (classe) développée, vous devrez impérativement :

- coder une méthode intitulée *void testEstGagnanteEfficacite()* qui, pour chaque *n* (le nombre d'allumettes au départ) comptabilise le nombre d'opérations élémentaires exécutées par la méthode *estGagnante(…)* + le temps d'exécution de cette méthode *estGagnante(…)*,
- commenter vos graphiques (voir + loin) : des résultats sans commentaires valables seront considérés comme inexistants (et cela justifiera donc une évaluation très faible voire nulle),
- vérifier que l'IA donne chaque fois (quand elle le peut) une solution gagnante à 100% pour la machine.

Développement

- A chaque version développée (4 au total) correspond 1 seule classe *NomClasse.java* qui contient nécessairement la méthode *void principal()*, point d'entrée de l'exécution, et qui se lance avec la commande *java Start NomClasse*.

- Pour chaque version développée, votre classe doit au minimum contenir les méthodes (déjà écrites pour la version brute) *boolean jouerGagnant(ArrayList<Integer> jeu)*, *boolean estPerdante(ArrayList<Integer> jeu)* et *boolean estGagnante(ArrayList<Integer> jeu)*. Mais également la méthode *void leJeu()* qui permet à un utilisateur de jouer contre la machine. Vous pouvez développer vos propres méthodes mais toute méthode ajoutée (par rapport à la version de base fournie) doit être nécessairement testée.

4.2. Grundy version0

Après avoir bien compris le fonctionnement de toutes les méthodes de la classe *GrundyRecBrute.java* (qui vous est fournie) et en particulier des méthodes *jouerGagnant* et *estPerdante*, vous devez écrire dans une **nouvelle classe *GrundyRecBruteEff*** :

- La boucle de jeu « joueur contre machine » en utilisant l'IA *jouerGagnant* pour la machine et constater que la machine joue intelligemment (dès qu'une décomposition est 100% gagnante elle la choisit).
- Une étude de l'efficacité de la méthode *estGagnante* de la version0 (brute) en plaçant un compteur (*long cpt* déclaré en variable globale) dans la boucle la + imbriquée (donc dans la méthode récursive). Vous ferez également une mesure du temps d'exécution de la méthode en fonction de *n*, *n* étant le nombre d'allumettes du tas de départ. On commencera avec n = 3 et on incrémentera n de 1 allumette (à un certain moment avec la version brute, le temps de calcul devient excessif et vous arrêtez l'expérience). Ces mesures seront consignées dans un tableau : pour chaque *n* en ligne, vous notez la valeur de *cpt* en colonne 1 et le *temps* d'exécution (préciser l'unité de temps) en colonne 2.

Vous devez en plus montrer dans un (ou +sieurs) graphiques, l'évolution de *cpt* en fonction de *n*, de même que le temps d'exécution en fonction de *n*.

Explication de la méthode « suivant »

La méthode *int suivant (ArrayList<Integer> jeu, ArrayList<Integer> jeuEssai, int ligne)* mérite des explications. Cette méthode donne une décomposition possible du *jeu*, sachant que la dernière décomposition est mémorisée dans *jeuEssai* et que le tas qui a été décomposé se trouve à l'indice « ligne ». La méthode réalise la décomposition suivante, mémorise cette nouvelle décomposition dans *jeuEssai* (écrase donc le contenu de *jeuEssai* passé en paramètre) et renvoie l'indice « ligne » correspondant au tas décomposé. Elle renvoie -1 lorsqu'il n'y a plus aucune décomposition possible.

Exemples (**par convention, le premier tas se trouve en ligne zéro = indice zéro dans *ArrayList***) :

- A partir du *jeu [1, 4, 5, 2]*, du *jeuEssai [1, 3, 5, 2, 1]* et de la ligne 1 (position du tas dans *jeu* qui a été décomposé dans *jeuEssai*), la méthode réalise la décomposition suivante *[1, 4, 4, 2, 1]* et renvoie la valeur 2 (numéro du tas décomposé).
- A partir du *jeu [1, 4, 5, 2]*, du *jeuEssai [1, 4, 4, 2, 1]* et de la ligne 2 (position du tas dans *jeu* qui a été décomposé dans *jeuEssai*), la méthode réalise la décomposition suivante *[1, 4, 3, 2, 2]* et renvoie la valeur 2 (numéro du tas décomposé).
- A partir du *jeu [1, 4, 5, 2]*, du *jeuEssai [1, 4, 3, 2, 2]* et de la ligne 2 (position du tas dans *jeu* qui a été décomposé dans *jeuEssai*), la méthode renvoie la valeur -1 car il n'y a plus de décomposition possible.

4.3. Grundy version1 : conserver les positions perdantes

Pour cette version1, vous devez écrire une **nouvelle classe *GrundyRecPerdantes.java***.

Cette version1, plus efficace que la version0, repose sur une constatation assez évidente : il ne sert à rien de recalculer les décompositions d'un tas (ou situation) que l'on sait (car déjà évalué) de toute

façon perdant(e). Ces tas (situations) perdants sont stockés dans un tableau (*ArrayList*) qu'il faudra simplement consulter pour chaque tas et situations avant même d'entamer une décomposition récursive (qui ne sert à rien sauf à perdre du temps).

Exemple : par décompositions successives on a trouvé que [4] est une situation perdante (car ne peut se décomposer qu'en [1, 3] et [3] est nécessairement gagnant). La situation (tas) [4] est ajoutée dans le tableau des situations perdantes (autre exemple : la situation [3, 6] est également perdante). Ultérieurement, si dans le jeu on tombe sur le tas [5], une décomposition possible est [1, 4] : il est inutile d'examiner les décompositions de [4], puisque l'on sait qu'elle est de toute façon perdante. Dès lors, on gagne du temps et le tableau des situations perdantes se construit au fur et à mesure que le jeu se déroule.

Autres simplifications à apporter au tableau des situations perdantes : il ne conservera que des tas d'allumettes > 2 (car [1] et [2] ne peuvent pas être séparés), une situation perdante n'apparaîtra qu'en un seul exemplaire dans le tableau et chacun des tas qui la composent seront rangés par ordre croissant des valeurs (pour permettre une recherche + facile à mettre en oeuvre).

Aide au codage de cette version1

- La structure qui mémorise les situations perdantes est un tableau de tableaux *i.e. ArrayList<ArrayList<Integer>> posPerdantes = new ArrayList<ArrayList<Integer>>()* (dans chaque case de *posPerdantes* on a UNE situation perdante qui est de type *ArrayList*). Ce tableau *posPerdantes* est une variable **globale**.
- Modifications à apporter à la méthode *estPerdante (jeu)* :
  - Avant même de commencer les décompositions de *jeu*, on vérifie d'abord que *jeu* est une situation perdante connue (appel à *estConnuePerdante (jeu)* voir ci-après). Si oui, la réponse de *estPerdante (jeu)* est immédiate : renvoyer *true*.
  - Si au bout de toutes les décompositions *jeu* s'avère être perdant alors le RAJOUTER (après normalisation c'est-à-dire sans les tas 1 et 2 et trié par ordre croissant des tas) dans le tableau des situations perdantes.
- Coder la méthode *boolean estConnuePerdante (ArrayList<Integer> jeu)* qui renvoie vrai si et seulement si APRES normalisation de jeu, il y a égalité entre *jeu* et une situation connue comme perdante dans le tableau des situations perdantes.

Travail demandé

Comme pour la version 0, mener une étude de l'efficacité de la méthode *estGagnante* de cette version1 en plaçant un compteur (*long cpt* déclaré en variable globale) dans la boucle la + imbriquée (donc dans la méthode récursive). Vous ferez également une mesure du temps d'exécution de la méthode en fonction de *n*, *n* étant le nombre d'allumettes du tas de départ. On commencera avec n = 3 et on incrémentera n de 1 allumette (à un certain moment, le temps de calcul devient excessif et vous arrêtez l'expérience). Ces mesures seront consignées dans un tableau.

Attention : pour chaque nouvelle expérience (i.e. pour chaque nouveau n) votre tableau (*ArrayList*) des solutions perdantes doit être vidé (*posPerdantes.clear()* sinon l'expérience est faussée !).

Vous devez en plus montrer dans un (ou +sieurs) graphiques l'évolution de *cpt* en fonction de *n*, de même que le temps d'exécution en fonction de *n*.

Vous devez comparer (commenter) vos résultats (obtenus avec cette version1) par rapport à ceux obtenus avec la version0.

4.4. Grundy version2 : conserver toutes les positions perdantes et gagnantes

Pour cette version2, vous devez écrire une **nouvelle classe *GrundyRecPerdEtGagn.java***.

Cette version2, plus efficace que la version1, exploitera un tableau des positions perdantes (comme la version1) mais aussi un tableau des positions gagnantes. Ces situations perdantes et gagnantes sont stockées dans 2 tableaux distincts (un perdant, un gagnant) qu'il faudra simplement consulter pour chaque situation avant même d'entamer une décomposition récursive (d'où le gain de temps).

Exemple : par décomposition successives, on a trouvé que les tas [3], [5] et [6] étaient tous gagnants. On les ajoute alors dans le tableau des gagnants. Ultérieurement, si dans le jeu on tombe, par exemple, sur des situations [1, 6] ou [1, 2, 5] ou [1, 1, 2, 3], on sera nécessairement gagnant car les tas de 1 ou 2 allumettes ne se décomposant pas, il reste en réalité les tas [6], [5] et [3] qui sont tous gagnants. Il ne sert donc à rien de poursuivre la décomposition.

Travail demandé :

Comme pour les versions précédentes, mener une étude de l'efficacité de la méthode *estGagnante* de cette version2 en plaçant un compteur (*long cpt* déclaré en variable globale) dans la boucle la + imbriquée (donc dans la méthode récursive). Vous ferez également une mesure du temps d'exécution de la méthode en fonction de *n*, *n* étant le nombre d'allumettes du tas de départ. On commencera avec n = 3 et on incrémentera n de 1 allumette (à un certain moment, le temps de calcul devient excessif et vous arrêtez l'expérience). Ces mesures seront consignées dans un tableau.

Attention : pour chaque nouvelle expérience (i.e. pour chaque nouveau *n*) vos 2 tableaux des solutions perdantes et gagnantes doivent être vidé (sinon l'expérience est faussée !).

Vous devez en plus montrer dans un (ou +sieurs) graphiques l'évolution de *cpt* en fonction de *n*, de même que le temps d'exécution en fonction de *n*.

Vous devez comparer vos résultats (obtenus avec cette version2) par rapport à ceux obtenus avec la version1.

4.5. Grundy version3 : conserver les positions et supprimer les tas perdants

Pour cette version3, vous devez écrire une **nouvelle classe *GrundyRecPerdantNeutre.java***.

Cette version3 se base sur le théorème 3.4 (http://mathenjeans.free.fr/amej/edition/9903grun/grundy2.html) qui est le suivant : la nature d'une situation ne change pas si on lui ajoute ou si on lui enlève un tas perdant (qui joue le rôle de neutre). En d'autres termes :

- situation Perdante (P) + tas Perdant (P) = situation Perdante (P)
- situation Gagnante (G) + tas Perdant (P) = situation Gagnante (G)

Exemple : la situation [3, 4, 20] est équivalente à [3] car : [4] est perdant donc [3] (G) + [4] (P) = [3] (G) et comme [20] est perdant (car supposé déjà évalué et mémorisé comme perdant dans le tableau des perdants) [3] (G) + [20] (P) = [3] (G).

Cette théorie permet de ne pas recalculer des positions déjà connues. Par exemple si [3] est connu comme une position gagnante, quand le programme génère la situation [3, 4, 20], cette situation sera immédiatement connue comme gagnante et on supprimera [4] et [20] (car tous les 2 perdants).

Cette version3 construit les tableaux perdants et gagnants comme pour la version2 ET utilise en plus le théorème 3.4. Elle doit donc être + efficace que la version2.

Travail demandé :

Comme pour les versions précédentes, mener une étude de l'efficacité de la méthode *estGagnante* de cette version3 en plaçant un compteur (*long cpt* déclaré en variable globale) dans la boucle la + imbriquée (donc dans la méthode récursive). Vous ferez également une mesure du temps d'exécution de la méthode en fonction de *n*, *n* étant le nombre d'allumettes du tas de départ. On

commencera avec n = 3 et on incrémentera n de 1 allumette (à un certain moment, le temps de calcul devient excessif et vous arrêtez l'expérience). Ces mesures seront consignées dans un tableau.

Attention : pour chaque nouvelle expérience (i.e. pour chaque nouveau *n*) vos 2 tableaux des solutions perdantes et gagnantes doivent être vidé (sinon l'expérience est faussée !).

Vous devez en plus montrer dans un (ou +sieurs) graphiques l'évolution de *cpt* en fonction de *n*, de même que le temps d'exécution en fonction de *n*.

Vous devez comparer (commenter) vos résultats (obtenus avec cette version3) par rapport à ceux obtenus avec la version2.

4.6. Grundy version4 : conserver les positions, supprimer les tas perdants et simplifier les couples gagnants

Pour cette version4, vous devez écrire une **nouvelle classe *GrundyRecGplusGequalsP.java***.

Cette version4 se base sur les constatations supplémentaires suivantes :

- Règle 1 : tas gagnant1 (G) + tas gagnant2 (G) = tas gagnant (G) si et seulement si gagnant1 et gagnant2 sont **de types différents**.
- Règle 2 : tas gagnant1 (G) + tas gagnant2 (G) = tas perdant (P) si et seulement si gagnant1 et gagnant2 sont **de même type**.

Pour savoir si deux positions (tas) sont de même type, on se base sur le tableau ci-dessous qui vous est donné :

T_0	T_1	T_2	T_3	T_4	T_5
0	3	5	13	18	41
1	6	8	16	21	44
2	9	11	19	24	47
4	12	14	22	27	
7	15	17	25	33	
10	28	29	30	36	
20	31	32		39	
23	34	35		42	
26	37	38		45	
50	40			48	
	43				
	46				
	49				

Chaque colonne T0, T1, T2, T3, T4, T5 contient l'ensemble des tas de même type.

Exemples :

- [2] et [4] sont de même type T0. Mais attention, ils sont en plus tous les 2 perdants donc aucune des 2 règles ne peut s'appliquer.
- [9] et [15] sont de même type T1. Et ils sont en plus tous les 2 gagnants. Donc la situation [9, 15] est perdante (voir la règle ci-dessus). La situation [11, 9, 15] est donc simplifiable en [11] puisque [9, 15] est neutre.
- Le même principe s'applique pour [27] et [39] : [27, 39] est perdant car [27] et [39] sont de même type T4 et tous les deux gagnants.
- Par contre, [3, 5, 13] n'est pas du tout simplifiable car les trois tas ne sont pas de même type ([3] est T1, [5] est T2 et [13] est T3).

L'idée est donc de supprimer les couples de positions gagnantes qui ensemble sont perdantes : par exemple X + 9 + 15 = X.

Le tableau précédent va être stocké de la manière suivante :

```
int[] type = {0, 0, 0, 1, 0, 2, 1, 0, 2, 1, 0, 2, …}
```

Ce qui signifie :

- `type[0]` contient le type du tas de 0 allumette (qui est le type 0)
- `type[1]` contient le type du tas de 1 allumette (qui est aussi le type 0)
- …
- `type[5]` contient le type du tas de 5 allumettes (qui est le type 2)
- …
- `type[48]` contient le type du tas de 48 allumettes (qui est le type 4)

De cette manière, il va être facile de savoir si deux tas sont perdants : soit un tas de « i » allumettes et un tas de « j » allumettes, si `type[i] == type[j]` alors cela signifie que les tas « i » et « j » sont de même type et donc ils sont forcément perdants ensemble (car P + P = P et G + G de même type = P).

Cette version4 construit les tableaux perdants et gagnants comme pour la version2 ET tient compte du théorème 3.4 de la version3 ET met en œuvre la théorie sur les types expliquée ci-dessus.

Travail demandé :

Comme pour les versions précédentes, mener une étude de l'efficacité de la méthode *estGagnante* de cette version4 en plaçant un compteur (*long cpt* déclaré en variable globale) dans la boucle la + imbriquée (donc dans la méthode récursive). Vous ferez également une mesure du temps d'exécution de la méthode en fonction de *n*, *n* étant le nombre d'allumettes du tas de départ. On commencera avec n = 3 et on incrémentera n de 1 allumette (à un certain moment, le temps de calcul devient excessif et vous arrêtez l'expérience). Ces mesures seront consignées dans un tableau.

Attention : pour chaque nouvelle expérience (i.e. pour chaque nouveau *n*) vos 2 tableaux des solutions perdantes et gagnantes doivent être vidé (sinon l'expérience est faussée !).

Vous devez en plus montrer dans un (ou +sieurs) graphiques l'évolution de *cpt* en fonction de *n*, de même que le temps d'exécution en fonction de *n*.

Vous devez comparer (commenter) vos résultats (obtenus avec cette version4) par rapport à ceux obtenus avec la version3.

5. Modalités pratiques et rendu de votre travail

Le projet se déroule en binôme (ou seul mais pas en trinôme) au sein d'un même groupe TD.

Début de la SAE : semaine 48.

Un créneau par semaine en autonomie sur les semaines 48, 49, 50, 51.

Travail à temps plein en semaine 03/2025.

Date limite du rendu : **vendredi 17 janvier 2025 à 23h55 au + tard (attention : -2 points de malus si retard)**.

Votre travail sera rendu sur Moodle sous forme d'une archive *nomBinôme1_nomBinôme2.zip* (pas de *.rar* !) qui contiendra plusieurs fichiers :

- Tous vos sources java correspondant aux versions 0 à 4 (1 classe par version avec la bonne orthographe !). Ces sources java, doivent être soigné, documenté (*javaDoc*). Ils doivent non seulement contenir le code correspondant à la bonne version mais ces sources doivent également contenir les tests d'efficacité de la méthode *estGagnante* (méthode *void testEstGagnanteEfficacite()*). **La javaDoc doit être écrite en anglais**.

- Un document (maximum 15 pages) au format PDF nommé *nomBinôme1_nomBinôme2.pdf*, qui contient :
  - les objectifs du travail en introduction,
  - l'ensemble des courbes obtenues pour chaque version + les tableaux qui reprennent pour chaque *n* la valeur du compteur *cpt* et le temps d'exécution et ceci pour chaque version,
  - un commentaire des résultats obtenus pour chaque version par rapport à la version précédente,
  - une conclusion finale sur l'efficacité de chacun des algorithmes (version 0 à 4).

Evaluation finale de cette SAE :

- 50% : note du projet « Jeu de Grundy »
- 50% : note de contrôle de votre niveau en *Java*