Règle du jeu de NIM :
– Deux joueurs
– Un certain nombres d'allumettes
– Au départ, le premier joueur enlève autant d'allumettes qu'il le désire mais il doit en enlever au moins 1 et en laisser au moins 1.
– Par la suite, chaque jour enlève au moins une allumette, et au plus le double du nombre d'allumettes enlevées par l'adversaire au coup précédent.
– Le joueur qui enlève la dernière allumette gagne.

Question 1

Faites 4 parties avec votre voisin , pour N= 8 allumettes : 2 parties où vous commencer, 2 parties votre voisin commence.
Notez le nombre de fois où celui qui commence gagne.

On souhaite écrire un algorithme qui prend en entrée le nombre d'allumettes, et qui retourne

- VRAI si le joueur qui commence gagne toujours, si il joue le mieux possible et quoi que fasse son adversaire
- FAUX si le joueur qui commence perd toujours

Indications :
• On considère un ensemble dont chaque élément est un couple (i,j), qui signifie qu'il reste i allumettes et qu'il est permis d'en enlever jusqu'à j.
• Un coup légal est $(i, j) \rightarrow (i- k, \min (2 k, i- k))$ où $k \in \{1, 2,..., j\}$
• La position de départ est (n,n-1)
• On remplit un tableau G(i,j) pour indiquer si une position est gagnante.

Objectif : Remplir la matrice G[i, j] tel que pour tout $1 <= i <= n$, G[i, j] = VRAI ssi la position < i, j> est gagnante

Question 2

Trouvez la formule de récurrence pour le calcul de G(i,j)

Question 3

Remplissez le tableau pour N=8

Question 4

Ecrire l'algorithme dans le langage de votre choix

Question 5

Est ce que cet algorithme est efficace ? Pourquoi ?

Question 6

Proposer une amélioration pour cet algorithme, dans le langage de votre choix .

Question 2

Une position est perdante si quoi que je joue, je tombe sur une position gagnante pour l'autre

Une position est gagnante si je peux jouer au moins une fois un coup qui amène à une position perdante pour l'autre

Par conséquent :

```
G(i,j) = FAUX si pour tout k de [1,j] , G(i-k,min(2k,i-k)) = VRAI
       = VRAI sinon
```

Question 3

G(1,1), G(2,2), …G(i,i) = VRAI

Pour calculer G(2,1)
k=1 : la position suivante sera G(1,1),
donc G(2,1) = FAUX

Pour calculer G(3,1)
k=1 : la position suivante sera G(2,2)
donc G(3,1) = FAUX

Pour calculer G(3,2)
k = 1 : la position suivante sera G(2,2)
k = 2 : la position suivante sera G(1,1)
donc G(3,2) = FAUX

Pour calculer G(4,1)
k=1 : la position suivante sera G(3,2)
donc G(4,1) = VRAI

Pour calculer G(4,2)
k=1 : la position suivante sera G(3,2)
k=2 : la position suivante sera G(2,2)
donc G(4,2) = VRAI

et ainsi de suite

i/j	1	2	3	4	5	6	7	8
1	VRAI							
2	FAUX	VRAI						
3	FAUX	FAUX	VRAI					
4	VRAI	VRAI	VRAI	VRAI				
5	FAUX	FAUX	FAUX	FAUX	VRAI			
6	VRAI	VRAI	VRAI	VRAI	VRAI	VRAI		
7	FAUX	VRAI	VRAI	VRAI	VRAI	VRAI	VRAI	
8	FAUX	FAUX	FAUX	FAUX	FAUX	FAUX	FAUX	VRAI

Question 4

```
private boolean nim(int n )
{
	boolean[][] G = new boolean[n+1][n+1];

	// On remplit la diagonale
	for(int i=1;i<=n;i++)
	{
		G[i][i] = true;
	}

	// On fait les lignes une par une
	for (int i = 2; i <= n; i++)
	{
		for (int j=1; j< i;j++ )
		{
			G[i][j] = computeValue(G,i,j);
		}
	}

	return G[n][n-1];
}


private boolean computeValue(boolean[][] G, int i, int j)
{
	for (int k = 1; k <= j; k++)
	{
		// On tombe sur une position perdante pour l'autre : on va pouvoir gagner
		if (G[i-k][Math.min(2*k, i-k)]==false)
		{
			return true;
		}
	}

	// On perd, tous les coups nous amenent à une position gagnante pour l'autre
	return false;
}
```

Question 5

Non car par exemple dans le calcul G(15,14) seulement 28 nœuds sont vraiment utiles, mais la programmation dynamique en calcule 121.

Question 6

Il faut combiner les avantages des 2 algorithmes : le récursif ne calcule que des valeurs utiles, le dynamique ne calcule jamais deux fois la même chose.

Il faut donc se rappeler des nœuds qui ont déjà été calculés lors du calcul récursif.

En C, on pourra utiliser un tableau de booléen global Connu[n+1][n+1] , initialisé à FAUX et que l'on positionnera à VRAI chaque fois que G[i][j] sera connu

En Java, on peut remplacer G par un tableau de Boolean, qui ont 3 valeurs : null pour inconnu, TRUE, ou FALSE

Solution en JAVA

```
	private boolean nimRecursif(int n)
	{
		Boolean[][] G = new Boolean[n+1][n+1];

		// On remplit la diagonale
		for(int i=1;i<=n;i++)
		{
			G[i][i] = true;
		}


		boolean res = gagnantRecursif(n,n-1,G);

		// TODO AFFICHER LE TABLEAU

		return res ;
	}

	private boolean gagnantRecursif(int i, int j, Boolean[][] G)
	{
		if (G[i][j]==null)
		{
			G[i][j] = computeValueRecursif(G,i,j);
		}

		return G[i][j];
	}

	private boolean computeValueRecursif(Boolean[][] G, int i, int j)
	{
		for (int k = 1; k <= j; k++)
		{
			// On tombe sur une position perdante pour l'autre : on va pouvoir gagner
			if (gagnantRecursif(i-k,Math.min(2*k, i-k),G)==false)
			{
				return true;
			}
		}

		// On perd, tous les coups nous amenent à une position gagnante pour l'autre
		return false;
	}
```

Si on affiche le tableau en fin de calcul, on obtient

```
i/j 1     2     3     4      5     6     7     8
1 VRAI |
2 null |VRAI |
3 null |FAUX |VRAI |
4 null |VRAI |null |VRAI |
5 null |FAUX |null |FAUX |VRAI |
6 null |VRAI |null |VRAI |null |VRAI |
7 null |VRAI |null |null |null |null |VRAI |
8 null |null |null |null |null |null |FAUX |VRAI |
```

On constate bien que le nombre de calcul est bien plus faible