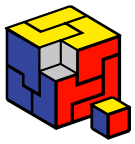
Prologin 2011



Concours National d'Informatique

Questionnaire de sélection

Correction des questions d'algorithmique

Benoît Zanotti Jill-Jênn Vie *

13 juin 2011

1 Les « 101 » Dalmatiens - Partie I

1.1 Énoncé

On vous donne un tableau de bits représentant une image en noir et blanc, vous devez implémenter une fonction qui calcule le négatif de cette image.

1.2 Solutions proposées par les candidats

Cet exercice étant très simple, il n'y avait pas réellement plusieurs implémentations possibles. L'idée était bien entendu de parcourir chaque pixel et d'échanger son état (de 0 vers 1 ou de 1 vers 0). On aura ainsi des algorithmes d'une complexité en $O(n^2)$ et ceci n'était pas simplifiable. Cependant, il existe d'autres critères que la complexité générale d'un algorithme.

En effet, on peut également s'appliquer à diminuer le coefficient caché par cette complexité ou encore à réduire la mémoire prise. Ici, les moins bons (et hélas les plus répandus) de vos algorithmes étaient trop gourmands en mémoire, car ils étaient en $O(n^2)$.

Nous sommes bien conscients que le code que nous fournissions avait déjà cette complexité en mémoire, mais il était bien plus intéressant de s'affranchir de ce code (qui n'est qu'une aide et n'est en rien obligatoire).

De même, nombreux sont ceux qui ont utilisé une structure conditionnelle pour inverser les valeurs en entrée. Sur un exercice aussi simple, nous attendions une optimisation plus poussée permettant de réduire au maximum les cycles processeurs utilisés.

Nous vous rappelons que le but du concours n'est **pas** de vous faire utiliser les fonctions des différentes bibliothèques (standard ou non) du langage que vous utilisez. Il était ainsi peu adapté (et très mal vu) d'utiliser la fonction **imagefilter** de PHP qui est beaucoup plus poussée (et donc moins efficace dans notre cas) que ce que nous vous demandions.

Ce que nous voulons voir avant tout, c'est votre raisonnement, pas que vous soyez capable d'utiliser des fonctions déjà implémentées par des personnes n'ayant rien à prouver à ce niveau-là.

1.3 Solution

Comme nous vous l'avons indiqué dans le paragraphe précédent, il y avait deux points à améliorer :
- la mémoire utilisée, que nous allons réduire de $O(n^2)$ à $O(1)$ en traitant « à la volée » les pixels ;
- la structure conditionnelle, que nous allons remplacer par un simple calcul, moins coûteux et tout aussi efficace.

Pour le premier point, il était en effet inutile de stocker l'intégralité de l'image pour ensuite la modifier. Vous pouviez très bien modifier et renvoyer le nouveau pixel immédiatement, en ne le stockant que temporairement. On utilise ainsi une seule variable pour stocker l'ensemble des pixels, tour à tour (N.B. – cette variable peut ne pas être explicite comme dans la correction qui va suivre).

*Pour l'équipe des correcteurs 2011 : Pierre Bourdon, Adrien Conrath, Maximin Coste, Nicolas Hureau, Sylvain Laurent, Victor Lenoir, Franck Michea, Pierre-Marie de Rodat, Alexis Rolland, Yves Stadler, Jill-Jênn Vie, Benoît Zanotti.

Pour le second point, nous avons trouvé deux solutions (parmi vos réponses) nous convenant tout à fait :

- la première est un XOR par rapport à 1. En effet, on a $1 \oplus 1 = 0$ et $0 \oplus 1 = 1$. De plus, au niveau de la machine, il ne s'agit que d'un simple masque ce qui est très rapide à exécuter ;
- la deuxième possibilité est la suivante : $sortie = 1 - entrée$. On remarque aisément que l'on a bien une inversion pour les valeurs 0 et 1.

Listing 1 – Solution de l'exercice 1 en C++

```
int N << std::cin;
int M << std::cin;
for (int i = 0; i < N; i++) {
  for(int j = 0; j < M; j++)
    std::cout << 1 - std::cin;
    // Inversion sans stockage du pixel
  std::cout << std::endl;
}
```

2 Les « 101 » Dalmatiens - Partie II

2.1 Énoncé

On vous donne un tableau de bits représentant une image en noir et blanc, vous devez dire si cette image admet un axe de symétrie vertical au milieu de l'image.

2.2 Solutions proposées par les candidats

Pour ce problème, il fallait déjà avoir la bonne définition d'un axe de symétrie vertical (et pas horizontal comme certains l'ont fait) : si le nombre de colonnes est impair, il est tout à fait **possible** qu'il y ait un axe de symétrie. Pour être exact, s'il existe, ce sera la colonne du milieu.

Nous avons ensuite eu plusieurs gestions des deux bornes, certains préférant utiliser deux variables pour ne pas se perdre, mais cela n'était pas réellement nécessaire (mais tout à fait correct).

N'oubliez pas qu'il est aussi bien plus intelligent d'arrêter vos boucles une fois que l'on a notre réponse (dès que l'on a un contre-exemple, on sait que l'image n'est pas symétrique). Il est **inutile** de continuer (de nombreux candidats ont bien entendu compris cela).

De plus, encore une fois la mémoire utilisée par nombre de candidats était bien trop importante car en $O(n^2)$.

2.3 Solution

Outre la solution évidente (on parcourt chaque moitié de ligne et on vérifie l'égalité), nous attendions deux principales optimisations (encore une fois, les deux premiers exercices étaient très faciles, nous attendions donc une certaine optimisation).

La première est, encore une fois, sur la mémoire. Nous n'avons ici besoin que d'une ligne à la fois, et non de l'intégralité de l'image. Ce simple fait nous permet de descendre à une utilisation de mémoire en $O(n)$ ce qui est bien plus acceptable.

La deuxième (si tant est que l'on considère cela comme une optimisation) est de s'arrêter **dès** que l'on a notre réponse. Il convient alors d'adapter son code (suivant le langage que l'on utilise) pour éviter un nombre important de calculs inutiles. On peut ainsi réduire la complexité *minimum* de l'algorithme en $O(n)$ (il faut récupérer l'intégralité ou du moins la moitié d'une ligne pour commencer notre vérification), même si le cas où l'image est symétrique reste en $O(n^2)$.

Listing 2 – Solution de l'exercice 2 en C++

```
std::vector<int> ligne;
ligne.resize(M);
for (int i = 0; i < N; i++)
{
  for (int j = 0; j < M/2; j++)
    std::cin >> ligne[j];
  if (M%2)
    std::cin.ignore(1);
  // On ignore le pixel central si M impair
  for (int j = 1; j <= M/2; j++) {
    int k << std::cin;
    if (ligne[M/2 - j] != k)
      return 0;
  }
}
return 1;
```

3 Pince os

3.1 Énoncé

On vous donne un tableau de bits représentant une image en noir et blanc, ProloGIMP 0.4.2β ne dispose que d'un pinceau en forme de « + » (3×3 pixels). Chaque application du pinceau sur l'image (le centre du pinceau devant être dans l'image, sinon le logiciel plante !) noircit les 5 pixels affectés. Vous devez implémenter une fonction déterminant s'il est possible de dessiner l'image donnée en entrée avec le pinceau, en partant d'une image dont tous les pixels sont blancs.

3.2 Solution

Pour commencer, nous allons apporter une solution générale au problème des bords. En effet, si l'on veut éviter de tester si un pixel hors de l'image est noir ou blanc, il faut d'abord tester que ce pixel existe bien.

La solution que nous vous proposons est moins lourde dans le code, plus rapide (surtout quand l'image peut être tracée, on parle de solution « optimiste ») mais aussi un peu plus gourmande en mémoire. Nous allons en effet étendre l'image initiale d'une ligne sur les 4 bords. Nos boucles ne parcourront pas ces lignes, mais permettront d'éviter un grand nombre de tests.

Cette solution est discutable. En effet, cela dépend de ce que vous voulez réduire : le temps d'exécution ou la mémoire utilisée. Nous avons ici fait le choix du temps car c'est une solution que nous avons peu observée parmi vos prestations.

De plus, nous allons, au lien d'en recréer une, utiliser l'image initiale et la modifier. L'idée est donc de partir de l'image initiale et de voir si on peut l'effacer entièrement à coups de pinceau (reproduire l'image ou effacer l'existante étant exactement la même chose). Ainsi, pour chaque pixel noir rencontré, s'il est un centre, nous appliquerons un coup de pinceau (avec une valeur différente de 1) sur l'image. Si, à la fin, l'image contient un pixel à 1, alors il n'était pas possible de la dessiner.

Attention, dans le code que nous vous proposons, les 0 et les 1 sont inversés afin de profiter de la fonction **calloc** du C qui, en plus de nous allouer de la mémoire, nous l'initialise également à 0. Si cela vous dérange, il était bien évidemment correct de conserver les conventions de l'énoncé. Autre astuce, nous voulons des valeurs différentes de 1, donc différentes d'un nombre impair, une comparaison bit à bit est donc aussi efficace et plus rapide.

On aurait également pu factoriser les 5 valeurs que l'on teste pour éviter d'avoir un code peu lisible, mais l'avantage est que nous n'allons faire ici que les tests utiles et nous arrêter dès qu'un pixel est à 1 (donc est blanc).

Nous vous rappelons également que l'on a une image de taille $(N+2) \times (M+2)$ pour gérer simplement les bords.

Listing 3 – Une solution de l'exercice 3 en C

```c
int estDessinable(int N, int M, int** m) {
  // Attention aux bornes, on est sur une image de taille (N+2)*(M+2)
  for (int i = 1; i < N+1; i++) {
    for (int j = 1; j < M+1; j++)
    {
      // Si une case vaut 1 alors elle est blanche
      if (m[i][j] != 1 && m[i-1][j] != 1 && m[i+1][j] != 1 &&
          m[i][j-1] != 1 && m[i][j+1] != 1) {
        m[i][j] = m[i-1][j] = m[i+1][j] = m[i][j-1] = m[i][j+1] = 2;
      }
    }
  }
  for (int i = 1; i < N+1; i++) {
    for (int j = 1; j < M+1; j++) {
      if (!m[i][j]) {
        return (0);
      }
    }
  }
  return (1);
}
```

Une autre solution consistait à tester, pour chaque pixel noir, s'il pouvait appartenir à un coup de pinceau. Cette solution avait l'avantage de s'arrêter dès qu'une anomalie était découverte.

Nous avons choisi de ne pas représenter cette solution étant donné qu'elle est très naïve et intuitive (et ne nécessitait donc pas réellement de correction).

3.3 Solutions proposées par les candidats

Cet exercice est le premier où il existe deux méthodes vraiment différentes pour le résoudre. Il y avait également quelques points gênants lors de la mise en place de l'une de ces méthodes, notamment la gestion des limites de l'image, qui pouvait ajouter un grand nombre de tests inutiles. Parmi les candidats ayant réussi, vous avez majoritairement utilisé les deux versions dont nous vous avons proposé une solution.

Pour la deuxième version évoquée, très peu de gens ont factorisé leur code, ce qui est très dommage. On se retrouvait ainsi avec des codes très longs et parfois pleins d'erreurs d'inattention.

Pour la version « optimiste », vous avez été très nombreux à utiliser une **autre** matrice pour marquer vos coups de pinceaux avant de tester la différence avec la matrice en entrée. C'est dommage, car la solution avec une unique matrice n'était pas particulièrement plus compliquée, mais tellement moins gourmande en mémoire !

4 Corruption

4.1 Énoncé

On vous donne un tableau de bits représentant une image en noir et blanc, vous disposez d'un pinceau en forme de « + » (3×3 pixels). Chaque application du pinceau sur l'image (le centre du pinceau devant être dans l'image) inverse la couleur (les bits) des 5 pixels affectés. Vous devez implémenter une fonction déterminant s'il est possible de dessiner l'image donnée en entrée avec le pinceau, en partant d'une image dont tous les pixels sont blancs (à 0).

4.2 Solutions

4.2.1 Une première approche

La solution que nous allons vous présenter ici est loin d'être la meilleure, mais c'était le minimum que nous attendions des candidats pour espérer glaner quelques points à cette question.

Il ne s'agit en fait que d'un *brute-force* amélioré. En effet, beaucoup l'auront remarqué, la valeur maximum de la largeur M du tableau était très limitée (car inférieure à 15). On pouvait donc, en restant dans les temps, effectuer un *brute-force* sur la première ligne (exclusivement).

Mais cela ne suffisait pas, il fallait également s'apercevoir que seule la disposition de la première ligne importait, et qu'il suffisait ensuite de corriger chaque ligne en effectuant un coup de pinceau bien placé sur la ligne du dessous (en se servant donc de la partie supérieure du pinceau pour corriger le pixel souhaité). La dernière ligne dépendait ainsi de la correction à apporter à l'avant-dernière ligne, de même pour celle-ci avec l'antépénultième. On pouvait ainsi remonter jusqu'à la première ligne, ce qui démontre bien que seule sa disposition à elle importait, et qu'il suffisait de toutes les générer pour avoir notre réponse.

Ainsi, une première solution était donc de tester, pour toutes les premières lignes possibles, si on pouvait bien annuler l'image de départ avec cette méthode. Notez bien que si l'on peut annuler l'image de départ, on peut bien évidemment la reconstruire puisqu'un coup de pinceau **inverse** les pixels.

Sachant que M est limité par 15, on choisit de représenter tous les bits d'une même ligne par un seul entier. Les bits de 0 à $M-1$ de cet entier représenteront les M pixels de chacune des N lignes. On pourra ainsi utiliser un XOR bit à bit pour simuler un coup de pinceau (en effet, le fait d'inverser des pixels entre 0 et 1 correspond au XOR binaire avec 1, puisque $0 \oplus 1 = 1$ et $1 \oplus 1 = 0$).

Listing 4 – Une première solution de l'exercice 4 en C

```c
int solve(int N, int M, int *tmp)
{
  for (int i = 0; i < N-1; i++)
  {
    tmp[i+1] ^= (tmp[i] ^ (tmp[i]<<1) ^ (tmp[i]>>1)) & ((1<<M) - 1);
    tmp[i+2] ^= tmp[i];
    // Inutile de modifier tmp[i], il ne sert plus dans la suite
  }
  return (!tmp[N-1]);
}

int estDessinableCorruption(int N, int M, int* matrix)
{
  int *tmp = calloc(N+1, sizeof(int));
  for (int i = 0; i < 1<<M; i++)
  {
    for (int j = 0; j < N; j++)
      tmp[j] = matrix[j];
    tmp[0] = (tmp[0] ^ i ^ (i<<1) ^ (i>>1)) & ((1<<M) - 1);
    tmp[1] = tmp[1] ^ i;
    if (solve(N,M,tmp))
      return (1);
  }
  return (0);
}
```

4.2.2 Les mathématiques à la rescousse !

Le problème de la solution du Listing 4 est sa complexité. En effet, elle est de $O(2^M NM)$. On comprend donc aisément que cette solution est viable **seulement** grâce à la faible valeur de M_{max}. En effet, une matrice 100×100 n'est pas envisageable, puisqu'elle demanderait de l'ordre de $2^{100} \times 100^2$ calculs. Sur un processeur à 1GHz, il faudrait plusieurs milliers de fois l'âge de l'univers pour obtenir une réponse !

Il est donc évident que, si cette solution était acceptable dans les conditions données, il nous faut chercher un meilleur algorithme.

Une meilleure solution nous vient alors des mathématiques : une simple résolution d'équations !

Considérons l'ensemble des entiers modulo 2 : $\{0, 1\}$, muni des opérations suivantes :

$+$	0	1
0	0	1
1	1	0

$\times$	0	1
0	0	0
1	0	1

On le note $\mathbb{Z}/2\mathbb{Z}$ [1].

Soit $x_{i,j}$ une variable valant 1 si le pinceau a été appliqué une fois en (i, j), 0 sinon. On ne s'intéresse pas aux autres cas, vu que l'appliquer 2 fois revient à ne pas l'appliquer du tout. En fait, tout ce qui compte, c'est de savoir s'il a été appliqué un nombre pair de fois (0) ou un nombre impair de fois (1), d'où le « modulo 2 » [2].

$x_{0,0}$	$x_{1,0}$			$x_{0,n-1}$
$x_{0,1}$			$x_{i-1,j}$	
		$x_{i,j-1}$	$x_{i,j}$	$x_{i,j+1}$
$x_{m-1,0}$			$x_{i+1,j}$	

Les applications du pinceau pouvant changer l'état de la case $(0, 0)$ sont celles en $(0, 0)$, $(0, 1)$, $(1, 0)$. Donc l'état de $(0, 0)$ après avoir effectué toutes les applications est donné par l'expression $x_{0,0} + x_{0,1} + x_{1,0}$. Si par exemple on applique le pinceau non en $(0, 0)$ mais en $(0, 1)$ et en $(1, 0)$, l'état de $(0, 0)$ sera : $0 + 1 + 1 = 0$.

À présent, notons $g_{i,j}$ l'état de la grille que l'on souhaite obtenir pour la case (i, j).

On pose $X = (x_{0,0}, \ldots, x_{m-1,n-1})$ et $G = (g_{0,0}, \ldots, g_{m-1,n-1})$. Les $x_{i,j}$ (resp. les $g_{i,j}$) seront appelées les *composantes* de X (resp. de G).

Pour une grille de taille 2×2, nous obtenons donc le système d'équations suivant :

$$\begin{cases} x_{0,0} + x_{0,1} + x_{1,0} = g_{0,0} \\ x_{0,0} + x_{0,1} + x_{1,1} = g_{0,1} \\ x_{0,0} + x_{1,0} + x_{1,1} = g_{1,0} \\ x_{0,1} + x_{1,0} + x_{1,1} = g_{1,1} \end{cases}$$

ou, sous forme matricielle [3] :

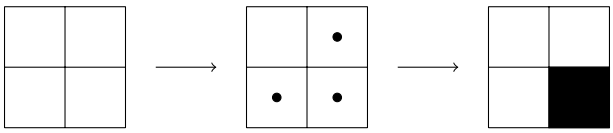
$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \underbrace{\begin{pmatrix} x_{0,0} \\ x_{0,1} \\ x_{1,0} \\ x_{1,1} \end{pmatrix}}_{X} = \underbrace{\begin{pmatrix} g_{0,0} \\ g_{0,1} \\ g_{1,0} \\ g_{1,1} \end{pmatrix}}_{G}$$

dont les inconnues sont les $x_{i,j}$.

Connaître X revient à connaître les endroits où il faut appliquer le pinceau.

1. Notez qu'on peut interpréter $\mathbb{Z}/2\mathbb{Z}$ comme l'ensemble des booléens, où $+$ joue le rôle de XOR et $\times$ le rôle de AND.
2. Vous voyez, tout est lié.
3. http://fr.wikipedia.org/wiki/Produit_matriciel

Exemple. Si $\begin{pmatrix} g_{0,0} & g_{0,1} \\ g_{1,0} & g_{1,1} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$, on peut voir que $\begin{pmatrix} x_{0,0} & x_{0,1} \\ x_{1,0} & x_{1,1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ est solution, comme indiqué sur la figure suivante.



Dans le cas général, sauf effets de bord, l'expression de l'équation de la case (i, j) est :
$x_{i-1,j-1} + x_{i-1,j} + x_{i,j} + x_{i,j+1} + x_{i+1,j+1} = g_{i,j}$.

La matrice correspondante est sous la forme

$$A = \begin{pmatrix} 1 & 1 & & 1 & & \\ 1 & \ddots & * & & \ddots & \\ & * & \ddots & * & & 1 \\ 1 & & * & \ddots & * & \\ & \ddots & & * & \ddots & 1 \\ & & 1 & & 1 & 1 \end{pmatrix}$$

où les vides correspondent à des 0 et les étoiles correspondent à des 1 sauf sur les bords de gauche et droite, lorsque la case n'a pas de voisin de gauche ou de voisin de droite.

La question est : est-ce que $AX = G$ admet une solution ?

Pour le savoir, nous allons bricoler les matrices A et G pour donner à A l'une des formes suivantes :

$$A' = \begin{pmatrix} 1 & * & * & * & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots & * \\ & & & \ddots & \ddots & * \\ & & & & \ddots & * \\ & & & & & 1 \end{pmatrix} \text{ ou } \begin{pmatrix} 1 & * & * & * & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots & * \\ & & & 1 & * & * \\ & & & & & \\ & & & & & \end{pmatrix}.$$

Il restera à résoudre $A'X = G'$.

Premier cas. Il n'y a aucune ligne de 0 dans A'.
Dans ce cas, n'importe quelle grille admet une solution ! En effet, la dernière ligne du système a donc pour équation $x_{m-1,n-1} = g'_{m-1,n-1}$ [4], elle dicte la valeur de $x_{m-1,n-1}$. La ligne précédente permet de dicter la valeur de l'avant-dernière composante de X, et ainsi de suite jusqu'à la première, qui dicte la valeur de $x_{0,0}$.

Deuxième cas. Les k dernières lignes de A' ne contiennent que des 0.
Les k dernières équations sont donc de la forme :

$$\begin{cases} 0x_{0,0} + \ldots + 0x_{m-1,n-1} = 0 = g''_1 \\ \vdots \\ 0 = g''_k \end{cases}$$

où $(g''_1, \ldots, g''_k)$ représentent les dernières composantes de G' [5].
Là, si un élément, ne serait-ce qu'un seul, parmi les g''_k, est égal à 1, on a **perdu**. Sinon, il suffit de choisir des valeurs pour les k dernières composantes de X, puis les équations dicteront les $mn - k$ premières valeurs, comme expliqué dans le premier cas.

4. Pour comprendre ceci, écrivez $A'X = G'$ sous forme de système.

5. Si vous vous demandez pourquoi j'ai noté le dernier élément g''_k au lieu de $g'_{m-1,n-1}$, demandez-vous comment j'aurais noté l'avant-dernier élément. $g'_{m-1,n-2}$? Mais si $n = 1$? Bref, vous avez votre réponse.

Les opérations que l'on peut effectuer pour bricoler la matrice A sont les suivantes :

- *la permutation de deux lignes* : en effet, changer l'ordre des équations ne change pas la solution ; on pourra donc intervertir les lignes i et i' de la matrice si l'on intervertit également g_i et g'_i [6] ;
- *l'ajout d'une ligne à une autre* : vous avez déjà utilisé au collège cette méthode sous le nom de méthode par combinaison, avec les systèmes de 2 équations à 2 inconnues. Si on sait que $x_{0,0}+x_{0,1}+x_{1,0} = g_{0,0}$ et que $x_{0,0}+x_{0,1}+x_{1,1} = g_{0,1}$, on peut en déduire que $x_{1,0}-x_{1,1} = g_{0,0}-g_{0,1}$. Comme $-1 = 1$ [7], on peut remplacer les $-$ par des $+$. Dans notre implémentation, la somme de 2 lignes sera effectuée avec des XOR.

Exemple. On note :

- $a_{i,i'}$ l'opération d'ajout à la ligne i de la ligne i' ;
- et $p_{i,i'}$ la permutation des lignes i et i'.

$$\begin{pmatrix}1&1&1&0\\1&1&0&1\\1&0&1&1\\0&1&1&1\end{pmatrix} \xrightarrow[a_{2,0}]{a_{1,0}} \begin{pmatrix}1&1&1&0\\0&0&1&1\\0&1&0&1\\0&1&1&1\end{pmatrix} \xrightarrow{p_{1,2}} \begin{pmatrix}1&1&1&0\\0&1&0&1\\0&0&1&1\\0&1&1&1\end{pmatrix} \xrightarrow{a_{3,1}} \begin{pmatrix}1&1&1&0\\0&1&0&1\\0&0&1&1\\0&0&1&0\end{pmatrix} \xrightarrow{a_{3,2}} \begin{pmatrix}1&1&1&0\\0&1&0&1\\0&0&1&1\\0&0&0&1\end{pmatrix}$$

Stocker toute la matrice demanderait trop de mémoire, et serait idiot car elle contient beaucoup de 0. En réalité, il suffit de stocker les éléments à distance $2n$ de la diagonale. Cela a pour conséquence quelques changements d'index que vous n'aurez pas de mal à remarquer [8].

Cette solution en $O(mn^2)$ en mémoire et en $O(mn^3)$ en temps n'est **pas** optimisée. Il est tout à fait possible de diminuer encore sa complexité en partant à la chasse aux opérations coûteuses. Pour les plus curieux, vous pouvez passer voir l'algorithme de Wiedemann.

Listing 5 – Une solution de l'exercice 4 en C utilisant le pivot de Gauss

```
1  #define NB_DIRS 4
2
3  struct dir
4  {
5    int lig;
6    int col;
7  } dirs[NB_DIRS] = {{0, -1}, {1, 0}, {0, 1}, {-1, 0}};
8
9  int m, n;
10 int grille[100][15];
11 int taille;
12 int matrice[1500][61];
13
14 int id(int i, int j)
15 {
16   return n * i + j; // Permet d'associer un identifiant unique a (i, j)
17 }
18
19 void remplirCase(int i1, int j1, int i2, int j2)
20 {
21   if(i1 >= 0 && i1 < m && j1 >= 0 && j1 < n
22      && i2 >= 0 && i2 < m && j2 >= 0 && j2 < n)
23     matrice[id(i1, j1)][id(i2, j2) + 2 * n - id(i1, j1)] = 1;
```

6. Quand on intervertit les premiers membres de deux équations, il faut intervertir leurs seconds membres aussi.
7. Bienvenue dans $\mathbb{Z}/2\mathbb{Z}$.
8. Tellement ils sont moches.

```
24 }
25
26 void remplirLigne(int i, int j)
27 {
28   int dir;
29   remplirCase(i, j, i, j);
30   for(dir = 0 ; dir < NB_DIRS ; dir++) // On ajoute les voisins a la matrice
31     remplirCase(i, j, i + dirs[dir].lig, j + dirs[dir].col);
32 }
33
34 void permuterLignes(int i1, int i2)
35 {
36   int t, j;
37   for(j = 0 ; j + i2 - i1 < 4 * n + 1 ; j++)
38   {
39     t = matrice[i2][j];
40     matrice[i2][j] = matrice[i1][j + i2 - i1];
41     matrice[i1][j + i2 - i1] = t;
42   }
43   t = grille[i2 / n][i2 % n]; // Ne pas oublier de permuter le second membre aussi
44   grille[i2 / n][i2 % n] = grille[i1 / n][i1 % n];
45   grille[i1 / n][i1 % n] = t;
46 }
47
48 int gaussBinaire()
49 {
50   int i, j, k;
51   for(k = 0 ; k < m * n ; k++)
52   {
53     if(matrice[k][2 * n] == 0) // Zut, il est nul, on cherche plus bas un element non nul
54     {
55       i = k + 1;
56       while(i <= k + n && i < m * n && matrice[i][k + 2 * n - i] == 0)
57         i++;
58       if(i == k + n + 1 || i == m * n)
59         return k; // S'il n'y en a pas, ca signifie qu'il ne reste plus que des lignes de 0
60       else
61         permuterLignes(k, i);
62     }
63     for(i = k + 1 ; i <= k + n && i < m * n ; i++) // Apres k + n, les elements sont nuls
64     {
65       if(matrice[i][k + 2 * n - i]) // Element non nul repere
66       {
67         for(j = 0 ; j + i - k < 4 * n + 1 ; j++)
68           matrice[i][j] ^= matrice[k][j + i - k]; // Les 1 sous la diagonale disparaissent
69         grille[i / n][i % n] ^= grille[k / n][k % n];
70       }
71     }
72   }
73   return m * n;
74 }
75
76 int estDessinable()
```

```
77 {
78   int i, j, k, rang;
79   for(i = 0 ; i < m ; i++)
80     for(j = 0 ; j < n ; j++)
81       remplirLigne(i, j);
82   taille = m * n;
83   rang = gaussBinaire(); // Rang a partir duquel il n'y a que des 0
84   for(k = rang ; k < taille ; k++)
85     if(grille[k / n][k % n]) // Si un second membre est non nul, ce n'est pas dessinable
86       return 0;
87   return 1;
88 }
```

4.3 Solutions proposées par les candidats

Pour ceux qui se sont attaqués à l'exercice, vous avez majoritairement utilisé une méthode naïve et itérative (le *brute-force*), plus ou moins optimisé. Nous avons donc un bon nombre de copies de personnes qui ont bien compris que seule la première ligne importait. Il faut cependant relativiser ce résultat étant donné que cette solution a circulé en partie sur le forum. D'autres ont repéré diverses solutions mathématiques (plus ou moins poussées) à ce problème. Nous tenons à les féliciter pour avoir réussi cet exercice en un temps polynomial.

Merci à tous d'avoir participé ! Si vous avez des questions, n'hésitez pas à nous contacter à l'adresse info@prologin.org. Nous espérons vous revoir lors de l'édition 2012, la 20^{e} édition !