Concours National d'Informatique

Questionnaire de sélection

Correction des questions d'algorithmique et rapport des correcteurs

Thomas Deniau*

14 janvier 2007

1 Bissextile...

1.1 Énoncé

Écrire une fonction qui prend une année (un entier) en argument et retourne 1 si elle est bissextile, 0 sinon.

1.2 Corrigé

Toute bonne encyclopédie donne la définition d'une année bissextile de nos jours : il s'agit d'une année divisible par 4, mais qui n'est pas divisible par 100, sauf si elle est divisible par 400.

Cet exercice très simple visait simplement à voir si les candidats savaient manipuler des structures de contrôle simples comme le test et des opérations arithmétiques de base comme la division et le modulo.

En C, par exemple, l'on peut obtenir le reste de la division euclidienne de p par q à l'aide de l'opérateur modulo noté %; p%q donne, pour p et q positifs, le reste r cherché : $p \equiv r \pmod{q}$.

On peut alors simplement traduire la définition, toujours en C ici :

```
int bissextile(int annee)
{
        if ((annee%4)!=0) return 0;
        if ((annee%400) == 0) return 1;
        if ((annee%100) == 0) return 0;
        return 1;
}
```

Ce qui peut s'abréger en :

```
int bissextile(int annee)
{
        return (((annee%4)==0) &&
                ((annee%100 != 0) || (annee%400 == 0)));
}
```

*Pour l'équipe des correcteurs de Prologin 2007 : Julien Guertault, Alban Lefebvre, Raphaël Marinier, Thomas Deniau.

Notons l'élégante solution de Nicolas Morey, qui propose d'utiliser un Ou exclusif : en effet, l'année est bissextile si et seulement si un nombre impair de ces trois restes sont nuls.

```
int bissextile(int annee)
{
        return ((annee%4)==0)
                ^ ((annee%100) == 0)
                ^ ((annee%400) == 0));
}
```

1.3 Solutions proposées par les candidats

Ce problème a bien évidemment été très bien réussi dans l'ensemble. Seuls quelques candidats ne se sont pas renseignés sur la définition précise d'une année bissextile. En revanche, certains ne connaissaient pas l'opérateur modulo de leur langage et ont tenté de vérifier ces divisibilités à la main, en comparant le quotient de la division euclidienne et celui de la division en virgule flottante.

Certains candidats ont poussé le vice jusqu'à tester si l'année était bissextile dans le cadre du calendrier julien. Ce niveau de détail n'était pas attendu des correcteurs !

2 Hauteur de jetons

2.1 Énoncé

On donne une grille de Puissance 4 : un tableau de taille N par M, de 0 et de 1, où les 1 sont les jetons, de couleur indifférenciée, et les 0 les trous ; vous devez trouver la hauteur maximale atteinte par les jetons.

2.2 Solution naïve

La solution naïve consistait en un parcours complet du tableau, qui comptait le nombre de 1 dans chaque colonne, et repérait au moment de ce comptage la colonne comportant le plus de 1. Elle a une complexité $O(NM)$.

L'on pouvait trouver une solution de même complexité (avec un facteur constant inférieur en moyenne) en remarquant que, puisque les jetons sont soumis à la gravité, il suffisait de parcourir le tableau ligne par ligne en partant d'en haut à gauche et de s'arrêter au premier 1 trouvé.

Il fallait alors, dans les deux cas, bien faire attention à la manière dont le tableau était représenté en mémoire pour ne pas faire d'erreurs d'indices. Par exemple, dans le deuxième cas, si i était le numéro de la ligne en cours, il fallait retourner $N - i$ puisque la 0^e ligne était la ligne la plus haute du tableau...

2.3 Solutions optimales

Nous attendions cependant une solution de complexité inférieure. Il y avait deux solutions optimales.

La première consistait à faire un parcours "en escalier". En partant du bas à gauche, on "monte" dans la première colonne tant que l'on trouve des 1. Au

premier zéro trouvé, on se décale vers la droite jusqu'à retomber sur un 1, puis on remonte jusqu'à trouver un 0, etc. Cette solution a une complexité $O(N+M)$: elle a l'avantage de ne pas parcourir toutes les cases dont on sait déjà qu'elles sont en dessous ou à la même hauteur qu'un autre 1.

Nous pouvons la programmer très simplement :

```
int hauteur(int **grille, int N, int M)
{
        int i=0; // ligne en cours
        for (int j=0; j<M; j++)
                while ((i<N) && grille[N-i-1][j]) i++;
        return i;
}
```

Une autre solution consistait, en remarquant encore que les jetons étaient soumis à la gravité, à se servir du fait que chaque colonne était composée d'un nombre donné de 1 suivi d'un nombre donné de 0 pour faire une recherche dichotomique du sommet de la colonne de 1. On obtient alors une solution de complexité $O(M.\log N)$.

2.4 Solutions proposées par les candidats

Nous avons rencontré un problème de correction sur cette question. En effet, sur le site d'entraînement de Prologin, les candidats devaient également récupérer le tableau sur l'entrée standard. On ajoute alors nécessairement aux complexités précédentes un $O(NM)$ de lecture des données, et l'on ne pouvait donc pas faire échouer les programmes non-optimaux sur certains tests afin d'inciter les candidats à rechercher de meilleures solutions.

Certains candidats ont par ailleurs profité de ce qu'ils maîtrisaient la lecture des données pour appliquer le deuxième algorithme naïf et s'arrêter même de lire l'entrée standard dès qu'ils lisaient un 1. Cet algorithme bat bien sûr en pratique toutes les solutions proposées plus haut, mais le but de l'exercice était bien de faire chercher au candidat un algorithme efficace une fois le tableau entré... Nous n'avons cependant que très peu pénalisé ces candidats astucieux par rapport à ceux qui avaient trouvé la solution optimale.

En revanche, le fait que les algorithmes naïfs passaient tous les tests n'a pas encouragé la plupart des candidats à chercher les algorithmes optimaux, et la grande majorité des copies proposaient d'utiliser un des deux algorithmes naïfs.

Enfin, seuls certains candidats, qui n'avaient manifestement pas utilisé le site d'entraînement, ont fait des erreurs d'indiçage.

3 Sudoku

3.1 Énoncé

Le Sudoku est un jeu qui est devenu très populaire récemment. Il se présente sous la forme d'un tableau de trois grilles par trois, elles-mêmes composées de trois cases par trois. Le but du jeu consiste à remplir les cases de chiffres allant de 1 à 9 sans qu'un même chiffre apparaisse plus d'une fois par ligne, colonne, et grille.

On vous donne donc un tableau de taille 9 par 9, rempli d'entiers allant de 1 à 9. Vous devez écrire une fonction qui retourne 1 si ce tableau est un jeu de Sudoku correctement rempli (et 0 sinon).

3.2 Solutions naïves

La première solution naïve consistait, pour chaque chiffre rencontré, et pour chaque "zone" (ligne, colonne, carré), à itérer sur le reste de la zone pour vérifier que le chiffre n'apparaissait pas une deuxième fois. Cette solution, outre le fait d'avoir une complexité importante (si le Sudoku était de côté n^2 et non de côté 9, elle aurait une complexité $O(n^6)$), demande un code long et dangereux, car beaucoup d'erreurs d'indices peuvent s'y glisser.

La deuxième solution naïve consistait à trier les chiffres de chaque zone pour vérifier, en temps $O(n^2 \log n)$ si l'on considère n^2 chiffres, que chaque chiffre apparaissait bien une seule fois. On a alors un algorithme en $O(n^4 \log n)$, ce qui est déjà beaucoup mieux...

3.3 Solutions optimales : avec des tableaux de booléens

La première solution optimale demandait l'utilisation d'un tableau de booléens. Supposant ce tableau ayant initialement toutes ses cases mises à FAUX, on parcourt une zone en marquant les cases correspondant aux chiffres rencontrés d'un VRAI. Si le chiffre en cours a déjà sa case marquée VRAI, c'est qu'il apparaît deux fois dans la zone et que le Sudoku est invalide. Si l'on a réussi à parcourir toute la zone sans tomber sur ce cas, c'est que chaque chiffre n'est présent qu'une seule fois, et, par bijectivité, que chaque chiffre y est une seule fois (puisqu'on a autant de cases que de chiffres).

Une manière élégante d'écrire cette solution, qui évitait des manipulations fastidieuses d'indices, était d'utiliser plusieurs tableaux de booléens pour les différentes zones afin de tout faire "d'un coup".

Voici une manière de le faire, en C99 (attention aux indices ; si l'on utilisait directement l'élément (i, j) de la grille pour se repérer dans les tableaux, il fallait des tableaux de 10 cases et non de 9 ; nous soustrayons ici 1 aux indices pour éviter cet écueil) :

```
int col[9][9];
int row[9][9];
int square[3][3][9];

bool test_sudoku(int **sudoku)
{
  // nous passons l'etape d'initialisation
  // des tableaux a "false", triviale

  for(int i=0; i<9; i++)
  {
    for(int j = 0; j<9; j++)
      {
        if (col[j][sudoku[i][j]-1]) return false;
          col[j][sudoku[i][j]-1]= true;
```

```
            if (row[i][sudoku[i][j]-1]) return false;
              row[i][sudoku[i][j]-1]= true;

            if (square[i/3][j/3][sudoku[i][j]-1]) return false;
              square[i/3][j/3][sudoku[i][j]-1]= true;
        }
    }
    return true;
}
```

On ne parcourt ici qu'une seule fois le Sudoku et on a donc une complexité $O(n^4)$. Malheureusement, on a maintenant également un $O(n^4)$ en mémoire. L'utilisation d'un seul tableau de booléens remis à Faux à chaque vérification de zone fait descendre cette utilisation mémoire à un $O(n^2)$, mais rend le code plus difficilement lisible.

Puisque n était explicitement fixé à 3 dans l'énoncé, ces considérations sont cependant accessoires.

Il y avait une optimisation que peu de candidats ont vue : si toutes les lignes et colonnes convenaient, il n'était pas nécessaire de vérifier les neuf carrés mais seulement quatre d'entre eux. Si l'on vérifie les deux carrés en haut à gauche par exemple, le troisième carré de la première ligne est automatiquement correcte puisque la vérification des trois premières lignes impose la présence de trois fois chaque chiffre dans ce groupe de trois lignes...

3.4 Solutions optimales : avec une somme de contrôle

L'on pouvait utiliser le même principe en utilisant non un tableau de booléens, mais une variable entière pour chaque zone, qui pouvait mesurer diverses sommes de contrôle.

L'adaptation la plus évidente de l'algorithme précédent est de remplacer le tableau de 9 booléens par un entier que l'on manipule bit à bit. Il s'agit alors de vérifier à la sortie de la boucle que l'on a bien neuf 1 dans l'écriture binaire du nombre, soit que le nombre vaut bien 511.

Une autre somme de contrôle qui fonctionnait était d'associer un nombre premier à chaque entier entre 1 et 9, et de vérifier la valeur du produit en sortie de boucle.

3.5 Solutions proposées par les candidats

Ce problème a bien été réussi dans l'ensemble ; la différence se faisait ici sur l'élégance du code... Certains candidats, qui n'ont ainsi pas réussi à caractériser mathématiquement les petits carrés à l'aide de divisions euclidiennes par 3, ont ainsi traité chaque petit carré un à un...

Trop de candidats ont cependant utilisé l'une des deux solutions naïves ; mais, encore une fois, le site d'entraînement ne pouvait pas faire la différence puisque la taille du Sudoku était explicitement fixée à 9 par 9.

En outre, beaucoup de ceux qui se sont essayés à des sommes de contrôle moins évidentes que celles citées plus haut se sont trompés.

Ainsi beaucoup ont juste fait la somme de chaque région pour vérifier qu'elle valait bien 45. Cela ne fonctionne tout bonnement pas (prendre une grille remplie de 5 par exemple).

D'autres ont testé le produit, affirmant de manière péremptoire en commentaires que si le produit était bien égal à 9 ! cela suffisait, ou encore qu'il suffisait de vérifier somme et produit (non : cf $1 \times 2 \times 4 \times 4 \times 4 \times 5 \times 7 \times 9 \times 9 = 9!$ et $1 + 2 + 4 + 4 + 4 + 5 + 7 + 9 + 9 = 45$). Cela était cependant plus acceptable car les multiples contraintes sur lignes, colonnes, etc., empêchaient peut-être l'apparition de ces exemples. Mais c'est dans ce genre de cas que l'on apprécie les démonstrations et non les simples affirmations !

4 L'héritière informaticienne

4.1 Énoncé

Le problème tel que posé dans le questionnaire est : "Un vieil homme qui a beaucoup d'enfants (N) veut choisir lequel sera son héritier. Il les dispose en cercle, les numérote de 0 à N-1, et se met à en éliminer un sur K jusqu'à qu'il n'en reste qu'un... à quelle position doit se placer l'informaticienne de la famille pour être celle qui est choisie ?".

Cet énoncé est une version déguisée d'un problème historique intitulé *problème de Josephus* : en 67 de notre ère, Flavius Josèphe s'était réfugié avec les derniers résistants juifs à l'attaque romaine de la ville de Jotapat dans une caverne. Les troupes de Vespasien arrivèrent et les Juifs décidèrent de se suicider plutot que de devenir esclaves (comme le feront leurs successeurs à Massada quelques années plus tard...), en suivant le schéma décrit dans l'énoncé. Flavius Josèphe fut heureusement parmi les deux derniers et il réussit à convaincre l'autre de ne pas s'entretuer. Cette histoire est rapportée dans *La Guerre des Juifs*.

Ce que l'histoire oublie, c'est que dans le processus expliqué dans *La Guerre des Juifs*, l'ordre des participants dans le cercle était tiré au sort... et que Flavius Josèphe a donc eu plus de chance que de réelle intelligence :-)

4.2 Solution naïve

La solution la plus simple consiste en la création d'un tableau de taille n de booléens : faux si l'individu correspondant n'a pas encore été éliminé, vrai sinon, et d'itérer sur les cases du tableau, en comptant les éliminations, jusqu'à qu'il ne reste plus qu'un individu non encore éliminé. Comme plus on élimine de gens, plus il faut itérer sur un grand nombre de cases avant de trouver le suivant, cette solution n'est clairement pas optimale.

```
int naive(int n, int k)
{
  bool *tab = (bool*)malloc(n*sizeof(bool));
  bzero(tab, n*sizeof(bool));

  int killed = 0;
  int i=0;
  int counted = 0;
```

```
  while (killed < (n-1))
    {
      if (! tab[i])
        {
          counted++;
          if (counted == k)
            {
              killed++;
              tab[i]=true;
              counted=0;
            }
        }
      if ((++i)==n) i=0;
    }
  while (tab[i]) i=(i+1)%n;
  return i;
}
```

4.3 Solution avec des listes circulaires doublement chaînées

Une liste circulaire doublement chaînée est une structure de données dont chaque élément contient un pointeur vers l'élément précédent et l'élément suivant, et où le dernier et le premier élément sont reliés. Elle convient particulièrement lorsqu'il faut supprimer un élément puisque l'on peut accéder facilement à partir d'un élément à celui qui le suit et à celui qui le précède. Le fait qu'elle soit circulaire modélise bien la structure du problème.

Nous construisons donc une liste circulaire doublement chaînée où chaque nœud contient un entier correspondant à son numéro initial dans le cercle, et nous supprimons les nœuds au fur et à mesure. Comme les éliminés ne font plus partie de la liste, nous ne perdons pas de temps à itérer sur des gens déjà éliminés et l'algorithme a une complexité $O(nk)$ bien plus acceptable.

Notre implémentation des listes circulaires doublement chaînées est dans notre cas assez simple : il nous faut seulement une fonction capable de supprimer un nœud (en renvoyant le suivant) et une capable de créer une liste chaînée de n nœuds numérotés de 0 à $n-1$:

```
typedef struct _elem
{
  int k;
  struct _elem *next;
  struct _elem *prev;
} elem;

elem* makeList(int n)
{
  elem *first = (elem*)malloc(sizeof(elem));
  first->k = 0;
  elem *last = first;
  for (int i=1; i<n;i++)
```

```
    {
      elem *e = (elem*)malloc(sizeof(elem));
      e->k = i;
      e->prev = last;
      last->next = e;
      last = e;
    }
  last->next = first;
  first->prev = last;
  return first;
}

elem* delete(elem *e)
{
  e->prev->next = e->next;
  e->next->prev = e->prev;
  elem *r = e->next;
  free(e);
  return r;
}
```

L'implémentation de la solution du problème de Josephus, une fois les listes chaînées implémentées, devient plus simple que la solution précédente :

```
int linkedList(int n, int k)
{
  elem* list = makeList(n);
  int i=1;
  while (list->next != list)
    {
      if (i==k)
        {
          list = delete(list);
          i=1;
        }
      else
        {
          i++;
          list = list->next;
        }
    }
  return list->k;
}
```

4.4 Une solution en $O(n)$

Malheureusement, cette version occupe encore beaucoup de mémoire : elle utilise une quantité de mémoire proportionnelle au nombre de personnes initialement "vivantes" N. Peut-on s'affranchir de cette représentation en mémoire de chaque individu ? Il nous faudrait pour cela trouver de manière mathématique la solution, au lieu de simuler bêtement le processus comme précédemment.

On peut dans cette optique imaginer une renumérotation constante des personnes membres du cercle. Par exemple, dans le cas $n = 5$ et $k = 3$, on élimine la personne numérotée 2. Tout se passe alors comme dans le cas $n = 4$ en établissant une nouvelle numérotation : la 3^{e}devient la 0^{e}, la 4^{e} devient la 1^{re}, la 0^{e} devient la 2^{e}, la 1^{re} devient la 3^{e}. On peut en fait montrer assez facilement que $i' + k \equiv i \pmod{n}$ où i' est le nouveau numéro.

En notant $J(n, k)$ le numéro de la dernière personne en jeu, on obtient donc immédiatement $J(n, k) \equiv (J(n - 1, k) + k) \pmod{n}$ avec la condition aux limites $J(1, k) = 0$ (puisqu'il ne reste plus qu'une personne).

Cela nous fournit la solution qui est à la fois la plus simple et la plus rapide de ce corrigé (des solutions plus rapides existent pour des valeurs de k particulières) :

```
int compute(int n, int k)
{
  if (n==1) return 0;
  return (compute(n-1,k)+k)%n;
}
```

Le seul inconvénient de cette fonction est qu'elle n'est pas récursive terminale ; ainsi, la limite de la taille de la pile du système limitera la validité de notre programme aux petites valeurs de n. On risque un plantage pour les grandes valeurs de n (à environ 200000 sur notre machine de test).

On peut passer outre cette limitation en écrivant une version récursive terminale ou une version itérative de notre solution. Voici par exemple la solution itérative :

```
int computeIter(int n, int k)
{
        int r = 0;
        for (int i=2; i<=n; i++)
        {
                r = (r+k)%i;
        }
        return r;
}
```

4.5 Solutions proposées par les candidats

Environ la moitié des candidats ont utilisé la version naïve, en gardant tout le tableau en mémoire, la version liste chaînée, ou encore un hybride de ces deux méthodes en implémentant la liste chaînée à l'aide d'un tableau (où le tableau mémorise l'index de l'enfant non encore éliminé suivant).

L'autre moitié des candidats ont trouvé la solution optimale (itérative ; personne n'a proposé de version récursive terminale, et seuls quelques-uns ont rendu la version récursive non terminale) ; on peut cependant se demander s'ils l'ont trouvée eux-mêmes, où s'ils l'ont trouvée sur Internet (le problème de Josephus est un classique), au vu de leurs autres réponses et de la rédaction de leur réponse !

Ainsi, nous avons trouvés des étranges $r = (r + (k\%i))\%i$, alors que toute personne familière avec l'utilisation de l'opérateur modulo sait que $(k\%i)\%i =$

$k\%i$ et que $(a+b)\%i = (a\%i + b\%i)\%i$... Les candidats ayant donc justifié leur algorithme (explicitant la récurrence sur $J(n,k)$ et son origine), ou ayant eu la franchise de dire que l'algorithme n'était pas d'eux, ont donc été récompensés.

Enfin, rappelons que comme chaque année, les candidats ayant rendu un code correctement indenté, correctement commenté, et facilement compréhensible, ont été récompensés...