Devoir Surveillé 4

11 janvier 2024

INFORMATIQUE MPI*

DURÉE DE L'ÉPREUVE : **4 heures**

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Ce sujet comporte huit pages numérotées de 1/17 à 17/17

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Tournez la page S.V.P.

Vue d'ensemble du sujet

Ce sujet est composé de 3 parties composant la majorité de l'épreuve Informatique 1 du Concours Commun Mines-Ponts de 2023 en langage C, puis d'une 4ème partie traitant de la résolution du jeu du taquin en OCaml à l'aide de l'algorithme A*, qui est complètement indépendante du reste du sujet.

Préliminaires

Présentation du sujet

L'épreuve est composée d'un problème unique comportant 32 questions divisées en trois sections. L'objectif du problème est de construire des *listes à accès direct* : une liste à accès direct est un type de donnée abstrait qui permet, d'une part, d'empiler et de dépiler efficacement un élément en tête de liste et, d'autre part, d'accéder efficacement au k^{e} élément de la liste, pour n'importe quel indice k valide.
Dans la première section (page 1), nous étudions un système de numération : la représentation binaire gauche des entiers naturels. Dans la deuxième section (page 4), nous étudions les arbres binaires parfaits. Dans la troisième section (page 6), nous implémentons le type liste à accès direct par la structure de données concrète *liste gauche*, que nous construisons en exploitant les résultats obtenus dans les sections précédentes.
Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractère différentes désignera la même entité, mais du point de vue mathématiques avec la police en italique (par exemple n) et du point de vue informatique avec celle en romain avec espacement fixe (par exemple `n`).

Travail attendu

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. Des rappels de programmation sont faits en annexe et peuvent être utilisés directement.
Selon les questions, il faudra coder des fonctions à l'aide du langage de programmation C exclusivement, en reprenant le prototype de fonction fourni par le sujet, ou en pseudo-code (c-à-d. dans une syntaxe souple mais conforme aux possibilités offertes par le langage C). Il est inutile de rappeler que les entêtes <assert.h>, <stdbool.h>, etc. doivent être inclus.
Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites. On suppose que le type **int** n'est jamais sujet à des débordements.
Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle. Lorsqu'une réponse en pseudo-code est permise, seule la logique de programmation est évaluée, même dans le cas où un code en C a été fourni en guise de réponse.

1 Représentation binaire gauche des entiers naturels

1.1 Mise en place

Soient m un entier naturel et N un entier naturel non nul. Il est classique d'appeler *représentation binaire standard de l'entier m sur N chiffres*, ou plus simplement *représentation standard*, toute suite finie $b = (b_n)_{0\leq n<N}$ de longueur N telle que, pour tout indice n compris entre 0 et $N-1$, le chiffre b_n appartient à $\{0,1\}$ et l'égalité suivante est vérifiée

$$m = \sum_{n=0}^{N-1} b_n 2^n.$$

Définition : Nous appelons *représentation binaire gauche de l'entier m sur N chiffres* toute suite finie $g = (g_n)_{0\leq n<N}$ de longueur N telle que,

(i) pour tout indice n compris entre 0 et $N-1$, le chiffre g_n appartient à $\{0,1,2\}$,

(ii) l'égalité suivante est satisfaite

$$m = \sum_{n=0}^{N-1} g_n(2^{n+1}-1),$$

(iii) le chiffre « 2 » n'apparaît qu'au plus une fois parmi les chiffres $(g_n)_{0\leq n<N}$,

(iv) s'il existe une position p telle que le chiffre g_p est 2, alors, pour tout indice n compris entre 0 et $p-1$, le chiffre g_n est nul.

De manière plus courte, nous parlons simplement de *représentation gauche*.
La figure 1 ci-dessous donne une représentation standard et une représentation gauche sur quatre chiffres des seize premiers entiers. Conformément à l'usage habituel, nous écrivons toute représentation, qu'elle soit standard ou gauche, sous la forme d'un mot $b_{N-1}\cdots b_0$ ou $g_{N-1}\cdots g_0$ dans lequel les chiffres de poids faibles sont écrits à droite.

Entier	Repr. standard	Repr. gauche
0	0000	0000
1	0001	0001
2	0010	0002
3	0011	0010
4	0100	0011
5	0101	0012
6	0110	0020
7	0111	0100

Entier	Repr. standard	Repr. gauche
8	1000	0101
9	1001	0102
10	1010	0110
11	1011	0111
12	1100	0112
13	1101	0120
14	1110	0200
15	1111	1000

FIGURE 1 – Représentations standard et gauche des seize premiers entiers

☐ 1 – Soit c un entier. Donner la représentation standard de l'entier dont une représentation gauche est $10\cdots0$ (avec c chiffres nuls) en justifiant sommairement. Faire de même avec l'entier dont une représentation gauche est $20\cdots0$ (avec c chiffres nuls).

Réponse 1. L'entier dont la représentation gauche est $10\cdots0$ (avec c chiffres nuls) est $1\times(2^{c+1}-1)=\sum_{i=0}^{c+1}2^i$ donc sa représentation standard est $11\cdots1$ (avec $c+1$ chiffres 1).

L'entier dont la représentation gauche est $20\cdots0$ (avec c chiffres nuls) est $2\times(2^{c+2}-1)=\sum_{i=1}^{c+2}2^i$ donc sa représentation standard est $11\cdots10$ (avec $c+1$ chiffres 1).

☐ 2 – Déterminer, en justifiant, le plus grand entier naturel M_N qui admet une représentation gauche sur N chiffres. Préciser la représentation gauche de cet entier.

Réponse 2. On montre par récurrence sur $N\geq0$ que $M_N=2^{N+1}-2$, et sa représentation gauche est $20\cdots0$ (avec $N-1$ chiffres nuls).

- La figure 1 montre que cette propriété est vraie pour N allant de 0 à 3.
- La représentation gauche de M_{N+1} ne peut avoir que 3 formes possibles : $20\cdots0$, ce qui donne $2^{N+2}-2$, $1g_N\cdots g_0$, ou $0g_N\cdots g_0$, ce qui donne au plus $2^{N+1}-1+2^{N+1}-2=2^{N+2}-3$ par récurrence.

Définition : Soit n_0 un indice compris entre 0 et $N-1$. On dit que l'indice n_0 est la *position du chiffre de plus fort poids* d'une représentation $g_{N-1}\cdots g_0$ si l'indice n_0 est le plus petit entier tel que, pour tout indice $n>n_0$, le chiffre g_n est nul. On appelle le chiffre g_{n_0} le *chiffre de plus fort poids*.

☐ 3 – Soient N un entier naturel non nul, $g=g_{N-1}\cdots g_0$ et $h=h_{N-1}\cdots h_0$ deux représentations gauches d'un même entier m. Démontrer que les chiffres de plus fort poids de g et de h sont de même valeur et à la même position.

Réponse 3. Notons n_g la position du chiffre de plus fort poids de g et n_h celle de h. Sans perte de généralité, on peut supposer que $(n_g,g_{n_g})\geq(n_h,h_{n_h})$ (on utilise l'ordre lexicographique).
Supposons que $(n_g,g_{n_g})>(n_h,h_{n_h})$, alors

$$(g_{n_g}-h_{n_g})(2^{n_g+1}-1)=\sum_{i=0}^{n_g-1}(h_i-g_i)(2^{i+1}-1).$$

Avec $(g_{n_g} - h_{n_g}) \geq 1$, et $(h_i - g_i) \leq h_i$, on a donc

$$2^{n_g+1} - 1 \leq \sum_{i=0}^{n_g-1} h_i(2^{i+1} - 1),$$

ce qui est impossible d'après la réponse à la question précédente car $h_{n_g-1} \cdots h_0$ est une notation gauche sur n_g chiffres donc représente un entier valant au plus $2^{n_g+1} - 2$.
On a donc bien $(n_g, g_{n_g}) = (n_h, h_{n_h})$.

☐ 4 – Démontrer que tout entier appartenant à l'intervalle $[\![0, M_N]\!]$, où l'entier M_N a été introduit à la question 2, ne possède au plus qu'une seule représentation gauche sur N chiffres.

Réponse 4. Par récurrence sur N :
- 0 ne possède qu'une seule représentation gauche sur 0 chiffres.
- Soit $g = g_N \cdots g_0$ et $h = h_N \cdots h_0$ deux représentations gauches d'un même entier appartenant à $[\![0, M_{N+1}]\!]$. Par la question précédente, on a $g_N = h_N$ et par hypothèse de récurrence, on a $g_{N-1} \cdots g_0 = h_{N-1} \cdots h_0$, car ce sont deux représentations gauches d'un même entier appartenant à $[\![0, M_N]\!]$.

Indication C : L'entier N est déclaré comme constante globale. Nous utilisons la structure C déclarée comme suit pour écrire la représentation gauche sur N chiffres d'un entier $g_{N-1} \cdots g_0$:

```
#define N 8

struct RepGauche {
  int position;
  bool chiffres[N];
};
typedef struct RepGauche rg;
```

Le champ position repère la position éventuelle du chiffre 2 ; il vaut -1 au cas où le chiffre 2 n'apparaît pas. Pour tout indice n compris entre 0 et $N-1$, la n^{e} case du champ chiffres vaut true si le chiffre g_n est non-nul et vaut false sinon.
Par exemple, les entiers 15 et 21 ont pour représentation gauche les variables entier_15 et entier_21 suivantes :

```
rg entier_15 = { .position = -1,
                 .chiffres = {false, false, false, true,
                              false, false, false, false}};
rg entier_21 = { .position = 1,
                 .chiffres = {false,  true, false, true,
                              false, false, false, false}};
```

☐ 5 – Formaliser sous la forme d'un ou de plusieurs invariants le fait qu'une valeur C de type rg est la représentation gauche d'un entier.

Réponse 5. $(\text{g.position} = -1) \vee (\text{g. chiffres [g.position]} \wedge (\forall i.\text{g.position} > i \rightarrow \neg \text{g. chiffres [i]}))$

☐ 6 – Écrire une fonction C **int** rg_to_int(rg g), qui renvoie l'entier dont g est la représentation gauche. On supposera que l'invariant de la question 5 est satisfait.

Réponse 6.

```
int rg_to_int(rg g) {
  int tmp = 0, acc = 0;
  for (int i = 0; i < N; i = i + 1) {
    if (g.chiffres[i]) {
      acc = acc + tmp;
    }
    if (g.position == i) {
      acc = acc + tmp;
    }
```

```
    tmp = 2 * tmp + 1;
  }
  return acc;
}
```

1.2 Incrémentation et décrémentation

Nous proposons l'algorithme suivant :

> ALGORITHME MYSTÈRE :
> *Entrée :* Représentation gauche $g = g_{N-1} \cdots g_0$ d'un certain entier m.
> *Effet :*
> — Si aucun des chiffres $(g_n)_{0 \leq n < N}$ ne vaut 2, changer le chiffre g_0 en $g_0 + 1$.
> — Sinon, en notant p la position du chiffre 2, changer le chiffre g_p en 0 et le chiffre g_{p+1} en $g_{p+1} + 1$.
> Nous notons m' l'entier dont la représentation gauche est g après exécution de l'algorithme.

FIGURE 2 – Un algorithme

☐ 7 – Vérifier que l'invariant de la question 5 n'est pas rompu par l'algorithme mystère (*cf. figure 2*). Avec les notations m et m' de la figure 2, caractériser, en fonction de l'entier m, la valeur de l'entier m'.

Réponse 7.

— Si g.position $= -1$, alors g_0 est changé en $g_0 + 1$, avec $g_0 = 0 \vee g_0 = 1$ au début, donc à la fin, il y a au plus un seul chiffre 2, en position 0, et il n'y a donc pas de 1 à un indice plus petit.
— Sinon, si g.position $= p$, alors $\forall i. p > i \rightarrow \neg$g.chiffres[i] au début, et après avoir changé g_p en 0, tous les chiffres à un indice plus petit que $p + 1$ sont des 0. Or, g_{p+1} est le seul chiffre qui peut devenir un 2 après exécution de l'algorithme.

☐ 8 – Écrire une fonction C bool rg_incr(rg *s) dont la spécification suit :
Précondition : La variable s est un pointeur vers la représentation gauche d'un entier m.
Effet : La valeur pointée par s est modifiée afin de représenter l'entier $m + 1$.
Valeur de retour : Booléen true si l'incrémentation de m peut avoir lieu et false si un débordement se produit car $m + 1$ n'est pas représentable sur le même nombre de chiffres.

Réponse 8.

```
bool rg_incr(rg *s) {
  if (s->position == N - 1) {
    return false;
  }
  if (s->position != -1) {
    s->chiffres[s->position] = false;
  }
  if (s->chiffres[s->position+1]) {
    s->position = s->position+1;
  }
  else {
    s->chiffres[s->position+1] = true;
    s->position = -1;
  }
  return true;
}
```

☐ 9 – Calculer la complexité en temps dans le pire des cas de la fonction rg_incr en fonction de N. Comparer avec la complexité de la même opération sur la représentation standard.

Réponse 9. Dans le pire des cas, avec le code proposé, nous effectuons un nombre constant d'opérations : $O(1)$. Avec une représentation standard, on peut avoir une retenue qui se propage et avoir une complexité dans le pire des cas en $O(N)$, si on incrémente par exemple le nombre $2^{N-1}-1$. L'incrémentation est donc nettement plus efficace dans le pire des cas en utilisant la représentation gauche des entiers.

☐ 10 – Écrire une fonction C bool rg_decr(rg *s) dont la spécification suit :
Précondition : Le pointeur s désigne la représentation gauche d'un entier m.
Effet : La valeur pointée par s est modifiée afin de représenter l'entier $m-1$.
Valeur de retour : Booléen true si la décrémentation de m peut avoir lieu et false si un débordement se produit car m est nul.
Il est recommandé d'expliquer son intention avant de donner son code.

Réponse 10. Soit i le plus petit indice d'un chiffre non nul. Si $i=0$, il suffit de décrémenter le chiffre à l'indice 0. Sinon, il faut décrémenter le chiffre à l'indice i, et mettre le chiffre à l'indice $i-1$ à 2.

C

```
bool rg_decr(rg *s) {
  if (s->position != -1) {
    s->chiffres[s->position - 1] = true;
    s->position = s->position - 1;
  }
  else {
    int i = 0;
    while (!s->chiffres[i]) {
    i = i + 1;
      if (i == N) {
        return false;
      }
    }
    s->chiffres[i] = false;
    s->chiffres[i-1] = true;
    s->position = i-1;
  }
}
```

2 Arbres binaires parfaits

2.1 Opérations sur les arbres binaires

Indication C : Nous représentons les arbres binaires à valeurs entières au moyen du type C arb suivant, qui est un pointeur vers une structure contenant la valeur entière dans le champ valeur et les deux fils dans les champs fils_g et fils_d. L'arbre vide se représente par le pointeur NULL.

```
typedef struct Noeud *arb;
struct Noeud {
  int valeur;
  arb fils_g;
  arb fils_d;
}
```

L'arbre vide est, par convention, de hauteur -1.

☐ 11 – Écrire une fonction C **int** hauteur(arb a) qui calcule la hauteur de l'arbre a.

Réponse 11.

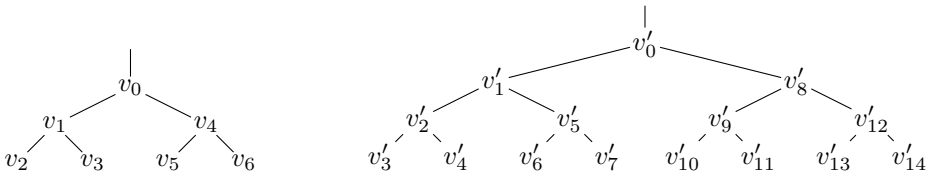

FIGURE 3 – Arbres binaires parfaits de hauteur 2 et de hauteur 3.

C

```c
int hauteur(arb a) {
  if (a == NULL) {
    return -1;
  }
  int hg = hauteur(a->fils_g);
  int hd = hauteur(a->fils_d);
  if (hg > hd) {
    return hg+1;
  }
  return hd+1;
}
```

☐ 12 – Écrire une fonction C arb noeud(**int** v, arb ag, arg ad) qui construit un arbre dont la racine a pour étiquette v, dont le fils gauche est a_g et le fils droit est a_d. Dans cette question, il est demandé de se défendre, par le truchement d'une assertion, de toute erreur liée à un échec d'allocation dynamique de mémoire.

Réponse 12.

C

```c
arb noeud(int v, arb ag, arg ad) {
  arb a = malloc(sizeof(struct Noeud));
  assert(a != NULL);
  a->valeur = v;
  a->fils_g = ag;
  a->fils_d = ad;
  return a;
}
```

2.2 Arbres parfaits

Définition : Un *arbre binaire parfait*, ou simplement *arbre parfait*, est un arbre binaire dont tous les nœuds internes ont exactement deux fils, dont toutes les feuilles sont à la même profondeur et dont les nœuds sont étiquetés par des valeurs entières.
Des exemples d'arbres binaires parfaits sont donnés figure 3.

☐ 13 – Démontrer que tout arbre binaire parfait de hauteur n possède $2^{n+1} - 1$ nœuds.

Réponse 13. Par induction structurelle sur les arbres binaires.
- L'arbre vide est un arbre parfait de hauteur -1 possédant 0 nœuds.
- Un arbre binaire parfait non vide est de hauteur $n \neq 0$, et toutes ses feuilles sont à profondeur n. Il possède un sous-arbre gauche et un sous-arbre droit, chacun d'eux possède des nœuds internes qui ont exactement deux fils et toutes leurs feuilles sont à profondeur $n-1$, ce sont donc deux arbres binaires parfaits de hauteur $n-1$, qui possède $2^n - 1$ nœuds par hypothèse de récurrence, et l'arbre binaire parfait de hauteur $n \neq 0$ possède donc $2(2^n - 1) + 1 = 2^{n+1} - 1$ nœuds.

☐ 14 – Écrire une fonction C bool est_parfait(arb a, **int** n) dont la spécification suit :

Précondition : Le pointeur a désigne la racine d'un arbre binaire (dont les nœuds sont bien tous distincts).
Valeur de retour : Booléen true si l'arbre pointé est parfait de hauteur n et false sinon.

Réponse 14.

C

```
bool est_parfait(arb a, int n) {
  if (a == NULL) {
    return (n == -1);
  }
  return (n != -1 && est_parfait(a->fils_g, n-1) && est_parfait(a->fils_d, n-1));
}
```

☐ 15 – Calculer la complexité en temps dans le pire des cas de l'exécution de est_parfait(a, n) en fonction de l'entier n.

Réponse 15. Le pire des cas est une instance positive. On ne connaîtra alors le résultat qu'après avoir parcouru tous les nœuds de l'arbre, soit une complexité en $O(2^n)$.

2.3 Opérations sur les arbres parfaits

Définition : Soient S une structure de données qui permet de stocker une collection d'entiers et t le cardinal de S. Nous disons que la structure de données S est à *accès direct* s'il existe une manière systématique de numéroter chaque élément de S entre 0 et $t-1$ et s'il existe deux primitives, `acces` et `modif`, qui permettent respectivement de consulter et de modifier n'importe quel élément de S en temps logarithmique en t à partir seulement de son numéro.
Nous souhaitons vérifier qu'un arbre parfait est une structure de données à accès direct. Nous numérotons les éléments d'un arbre parfait en utilisant l'ordre préfixe de gauche à droite de l'arbre (comme illustré dans la figure 3).
Rappel de programmation en C : l'expression 1 << n représente le décalage de la valeur 1 sur n bits : elle a pour valeur l'entier 2^n.

☐ 16 – Écrire une fonction C arb arb_trouve(arb a, **int** n, **int** k) ainsi spécifiée :
Précondition : Le pointeur a désigne la racine d'un arbre binaire parfait de hauteur n. L'entier k satisfait les inégalités $0 \leq k < 2^{n+1} - 1$.
Valeur de retour : Pointeur vers le k^{e} nœud de l'arbre dans l'ordre préfixe.

Réponse 16.

```
arb arb_trouve(arb a, int n, int k) {
  if (k == 0) {
    return a;
  }
  int pivot = 1 << n;
  if (k < pivot) {
    return arb_trouve(a->fils_g, n-1, k-1);
  }
  return arb_trouve(a->fils_d, n-1, k-pivot);
}
```

Il est rappelé que la racine d'un arbre est à profondeur 0. Nous donnons la formule de sommation, valable pour tout réel $a \neq 1$ et tout entier t,

$$\sum_{k=0}^{t} k \cdot a^k = \frac{((a-1)t-1)a^{t+1}+a}{(a-1)^2}$$

☐ 17 – Nous appelons P la profondeur d'un nœud choisi aléatoirement et uniformément parmi les $2^{n+1}-1$ nœuds d'un arbre binaire parfait de hauteur n. Calculer l'espérance $\mathbb{E}(P)$ de la variable aléatoire P.

Réponse 17. On montre par récurrence que pour tout $0 \leq k \leq n$, il y a 2^k nœuds de profondeur k dans un arbre binaire parfait de hauteur n.

— Seule la racine est à profondeur 0.
— Chacun des 2^k nœuds à profondeur k possède 2 fils, donc il y a bien 2^{k+1} nœuds à profondeur $k+1$.
En regroupant ensemble les termes pour les nœuds de même profondeur, comme chaque nœud est choisi avec probabilité $\frac{1}{2^{n+1}-1}$, on a

$$\mathbb{E}(P) = \frac{1}{2^{n+1}-1}\sum_{k=0}^{n} k\cdot 2^k = \frac{((2-1)n-1)2^{n+1}+2}{(2^{n+1}-1)(2-1)^2} = \frac{(n-1)2^{n+1}+2}{2^{n+1}-1}$$

☐ 18 – Déterminer la complexité moyenne en temps de l'instruction arb_trouve(a, n, k) lorsque la hauteur n et l'arbre a sont fixés et le numéro k est choisi aléatoirement et uniformément dans l'intervalle $[\![0, 2^{n+1}-2]\!]$.

Réponse 18. À chaque appel, la fonction termine ou bien effectue un seul appel récursif en partant d'un nœud dont la profondeur est strictement plus grande. En plus de cet appel, on effectue seulement un nombre constant d'opérations, donc la complexité est linéaire en la profondeur du nœud recherché de numéro k. Choisir k uniformément dans l'intervalle $[\![0, 2^{n+1}-2]\!]$ revient à choisir aléatoirement et uniformément un nœud, on peut donc utiliser le résultat de la question précédente : la complexité de cette instruction est $O(n)$.

☐ 19 – Écrire une fonction C **int** arb_acces(arb a, **int** n, **int** k) qui renvoie la k^{e} valeur de l'arbre a de hauteur n ainsi qu'une fonction C **void** arb_modif(arb a, **int** n, **int** k, **int** v) qui remplace la k^{e} valeur de l'arbre a de hauteur n par la valeur v. Dire finalement si la structure de données arbre binaire parfait est à accès direct.

Réponse 19.

C

```
int arb_acces(arb a, int n, int k) {
  arb b = arb_trouve(a, n, k);
  return b->valeur;
}

void arb_modif(arb a, int n, int k, int v) {
  arb b = arb_trouve(a, n, k);
  b->valeur = v;
}
```

Dans les deux cas, la complexité est dominée par l'instruction arb_trouve(a, n, k), dont la complexité dans le pire des cas est $O(n)$, car le nœud le plus profond est à profondeur n. Or le cardinal d'un arbre binaire parfait est $t = 2^{n+1}-1$, donc $n = O(\log(t)$, et la complexité de ces deux fonctions est logarithmique en t.
La structure de données arbre binaire parfait est à accès direct.

3 Listes gauches

Les arbres binaires parfaits seuls ne permettent de représenter que des collections d'entiers dont le cardinal est de la forme $2^{n+1}-1$ où n est entier naturel. Afin de représenter des collections de cardinal quelconque, nous introduisons des suites d'arbres parfaits, aux hauteurs strictement croissantes et savamment choisies.
Définition : Nous appelons *liste binaire gauche de cardinal m sur N arbres* la structure de données constituée de $N+1$ arbres binaires parfaits $((a_n)_{0\leq n<N}, e)$ comme suit. Nous décomposons l'entier m selon sa représentation binaire gauche sur N chiffres : $g_{N-1}\cdots g_0$. Pour tout indice n compris entre 0 et $N-1$, si le chiffre g_n n'est pas nul, l'arbre binaire parfait a_n est un arbre de hauteur n, sinon il s'agit de l'arbre vide. Si le chiffre 2 apparaît parmi les chiffres de la représentation gauche de m et si l'indice p est sa position, alors l'arbre e est un arbre binaire parfait de hauteur p, sinon l'arbre e est l'arbre vide. De manière plus courte, nous parlons simplement de *liste gauche*.
Une liste binaire gauche de cardinal m sur N arbres permet de stocker une collection de m éléments entiers. Il apparaît qu'avec N arbres, une liste binaire gauche peut stocker jusqu'à M_N entiers (où l'entier M_N a été introduit à la question 2) et que m est inférieur à M_N.
Indication C : Nous adoptons le type C suivant :

```c
struct ListeGauche {
  int hauteur_e;
  arb extra;
  int nb_arbres;
  arb *arbres;
};
typedef struct ListeGauche lg;
```

Le champ hauteur_e contient la hauteur p de l'arbre exceptionnel e. Le champ extra désigne la racine de l'arbre exceptionnel e. Le champ nb_arbres contient l'entier N. Enfin, le champ arbres désigne un tableau de N pointeurs vers les arbres $(a_n)_{0\leq n<N}$ qui a été alloué dynamiquement.

3.1 Opérations simples sur les listes gauches

☐ 20 – Écrire une fonction C lg lg_init(**int** N) qui renvoie une liste gauche de cardinal nul sur N arbres.

Réponse 20.

C

```c
lg lg_init(int N) {
  lg l;
  l.hauteur_e = -1;
  l.extra = NULL;
  l.nb_arbres = N;
  l.arbres = malloc(N*sizeof(arb));
  return l;
}
```

☐ 21 – Écrire une fonction C **int** lg_card(lg l) qui calcule le cardinal m de la liste gauche ℓ.

Réponse 21. On reprend le squelette de la réponse à la question 6.

C

```c
int lg_card(lg l) {
  int tmp = 0, acc = 0;
  for (int i = 0; i < l.nb_arbres; i = i + 1) {
    if (l.arbres[i] != NULL) {
      acc = acc + tmp;
    }
    if (l.hauteur_e == i) {
      acc = acc + tmp;
    }
    tmp = 2 * tmp + 1;
  }
  return acc;
}
```

Définition : Afin de numéroter l'ensemble des éléments d'une liste gauche $\ell = ((a_n)_{0\leq n<N}, e)$, nous parcourons d'abord les éléments de l'arbre exceptionnel e dans l'ordre préfixe, puis nous parcourons les éléments des arbres $a_0, a_1, \ldots, a_{N-1}$ dans l'ordre préfixe. Les numéros sont attribués aux valeurs rencontrées par ordre de première rencontre.

☐ 22 – Écrire une fonction C arb lg_trouve(lg l, **int** k) qui renvoie le k^{e} nœud de la liste gauche ℓ. On supposera que k est un indice valide ($0 \leq k < m$).

Réponse 22. On sépare le cas où il existe un arbre exceptionnel, ou non. Dans le cas où il existe, soit on y cherche l'élément, soit on se ramène au cas où il n'y a pas d'arbre exceptionnel, en renumérotant. Si il n'y a pas d'arbre exceptionnel, on peut dénombrer les éléments des i premiers arbres, jusqu'à dépasser la position recherchée. On sait alors dans quel arbre chercher, et à quelle position.

C

```c
arb lg_trouve(lg l, int k) {
  if (l.hauteur_e != -1) {
    int taille_e = (1 << (l.hauteur_e + 1)) - 2;
    if (k < taille_e) {
      return arb_trouve(l.extra, l.hauteur_e, k);
    }
    else {
      l.hauteur_e = -1; l.extra = NULL; // Ne modifie pas la liste (appel par valeur)
      return lg_trouve(l, k - taille_e);
    }
  }
  int i = -1, tmp = 0, acc = 0;
  while (acc <= k) {
    i = i + 1;
    if (l.arbres[i] != NULL) {
      acc = acc + tmp;
    }
    tmp = 2 * tmp + 1;
  }
  return arb_trouve(l.arbres[i], i, k - acc + tmp);
}
```

☐ 23 – Calculer la complexité en temps dans le pire des cas de lg_trouve(l, k) en fonction de la capacité maximale M_N de la liste gauche ℓ.

Réponse 23. Le pire des cas est lorsque l'on cherche le dernier élément. On doit alors parcourir la liste d'arbres avant de trouver le bon arbre, puis parcourir tout l'arbre avant de trouver le bon élément. Chacun de ces parcours s'effectue en $O(N)$. Notons que la présence d'un arbre exceptionnel ne rajoute qu'un nombre constant d'opérations si l'élément recherché n'est pas dedans, et que dans tous les cas, on ne parcours qu'un seul arbre binaire parfait. Étant donné que $M_N = 2^{N+1} - 2$, on a une complexité totale en $O(\log(M_N))$.

☐ 24 – Écrire une fonction C **int** lg_acces(lg l, **int** k) qui renvoie la k^{e} valeur de la liste gauche ℓ ainsi qu'une fonction C **void** lg_modif(lg l, **int** k, **int** v) qui remplace la k^{e} valeur de la liste gauche ℓ par la valeur v.

Réponse 24.

C

```c
int lg_acces(lg l, int k) {
        arb a = lg_trouve(l, k);
        return a->valeur;
}

void lg_modif(lg l, int k, int v) {
        arb a = lg_trouve(l, k);
        a->valeur = v;
}
```

3.2 Ajout et suppression en tête de liste gauche

☐ 25 – Soient v une valeur entière et $\ell = ((a_n)_{0\leq n<N}, e)$ une liste gauche de cardinal m, avec $m < M_N$, qui contient les éléments $v_1, \ldots, v_m$ dans cet ordre. Décrire, en fonction de la liste gauche ℓ, la liste gauche $\ell' = ((a'_n)_{0\leq n<N}, e')$ de cardinal $m+1$ dont les éléments sont $v, v1, \ldots, v_m$ dans cet ordre. En déduire le principe d'une fonction C bool lg_empile(**int** v, lg l) réalisant l'insertion de la valeur v en tête de la liste gauche ℓ où le booléen résultat vaut true si l'ajout a eu lieu et vaut false si un débordement de capacité de la liste se produit. On ne demande pas le code complet de cette fonction.

Réponse 25.
- Si e est non vide, soit p sa hauteur. Soit a^* l'arbre binaire parfait de hauteur $p+1$ de racine d'étiquette v, de sous-arbre gauche e et de sous-arbre droit a_p. Si a_{p+1} est vide, $a'_{p+1} = a^*$ et e' est vide. Sinon, $e' = a^*$. Les autres indices restent inchangés.
- Si e est vide. Soit a^* l'arbre binaire parfait de hauteur 0 de racine d'étiquette v. Si a_0 est vide, $a'_0 = a^*$. Sinon, $e' = a^*$. Les autres indices restent inchangés.

La fonction bool lg_empile(**int** v, lg l) fonctionne comme décrit ci-dessus, et renvoie true, sauf si (hauteur_e == l->nb_arb auquel cas elle renvoie false.

☐ 26 – Déterminer la complexité en temps dans le pire des cas de la fonction lg_empile.

Réponse 26. Dans tous les cas, la fonction lg_empile n'effectue qu'un nombre constant de tests et d'appels à la fonction noeud, qui se termine en temps constant, et ne modifie qu'au plus deux valeurs. On a donc une complexité en temps dans le pire des cas en $O(1)$.

☐ 27 – Donner le principe d'une fonction C bool lg_depile(**int** *w, lg l) réalisant le retrait de l'élément de tête de la liste gauche ℓ et son affectation à l'adresse w. Le booléen résultat vaut true si le retrait a eu lieu et vaut false si l'opération a échoué en raison d'une liste vide. On ne demande pas le code complet.

Réponse 27.
- Si e est non vide, soit p sa hauteur. On affecte à l'adresse w l'étiquette de sa racine. De plus, on affecte à a'_{p-1} son fils droit, et à e' son fils gauche si $p > 0$. On renvoie true.
- Si e est vide, on parcourt les arbres $a_0, \dots, a_{N-1}$ jusqu'à trouver a_i non vide, avec i minimal. Si un tel i n'existe pas, on renvoie false, sinon on affecte à l'adresse w l'étiquelle de sa racine. De plus, on affecte à e' son fils gauche, et à a'_{i-1} son fils droit si $i > 0$. On affecte ensuite l'arbre vide à a'_i et on renvoie true.

☐ 28 – Donner la complexité en temps dans le pire des cas de la fonction lg_depile.

Réponse 28. Dans le pire des cas, en plus d'un nombre constant de tests, et d'affectations, la fonction lg_depile doit parcourir la liste des arbres pour trouver le premier arbre non vide. On a donc une complexité en temps dans le pire des cas en $O(N)$.

☐ 29 – Discuter la possibilité d'obtenir une complexité plus faible à la question 28, quitte à modifier légèrement la définition du type lg.

Réponse 29. La complexité de la fonction lg_depile est dominée par le parcours de la liste des arbres.
L'approche intuitive pour résoudre ce problème serait d'ajouter un champ au type lg indiquant la position du premier arbre non vide s'il existe, ou -1 dans le cas contraire. Ceci permettrait d'éviter de parcourir la liste des arbres, et d'obtenir ainsi une complexité en temps dans le pire des cas en $O(1)$ pour cette fonction. Ceci suppose que l'on modifie les fonctions déjà établies pour maintenir à jour cette information, ce qui peut se faire en temps constant :
- lg_init peut initialiser cette valeur à -1.
- lg_empile peut incrémenter cette valeur lorsque la liste gauche donnée en entrée possède un arbre exceptionnel non vide, ou lorsque l'on insère un élément dans une liste gauche vide.
- Les autres fonctions définies précédemment de modifient pas cette valeur.

Cependant, pour mettre à jour ce champ dans la fonction lg_depile, on a besoin de parcourir la liste à nouveau si la liste gauche donnée en entrée possède un arbre exceptionnel e vide et un arbre a_0 non vide (e.g., lorsque l'on retire le dernier élément).
La complexité en temps dans le pire des cas reste donc $O(N)$.

☐ 30 – Soit N un entier et $\ell = ((a_n)_{0,\leq n<N}, e)$ une liste gauche. Discuter la possibilité de modifier en place et avec une faible complexité en temps la liste gauche ℓ de sorte que le nombre d'arbres N devienne $N+1$ et que les mêmes éléments deumeurent dans la liste.

Réponse 30. En l'état, il suffit de réallouer un tableau possédant une case de plus et d'y recopier le contenu du champ arbres, ainsi que d'incrémenter le champ nb_arbres. On peut noter que la fonction realloc existe pour ce genre d'opérations en C. La complexité de cette opération est $O(N)$, aussi, il peut être avantageux d'augmenter N de plus que 1 pour avoir recours à cette opération le moins souvent possible. Il s'agit d'un compromis entre le temps et la mémoire.

L'autre possibilité, si on veut que cette opération soit en temps constant, est de remplacer l'utilisation du tableau par une liste chaînée, mais cela ralentirait les accès à tous les arbres...
Enfin, on pourrait essayer de stocker la liste d'arbres avec une liste gauche.
Toutes ces possiblités offrent des compromis en terme de mémoire utilisée, et de complexité en temps des différentes opérations.

* *
*

Les listes binaires gauches, ou *skew binary lists*, ont été inventées par Eugene Myers en 1983.

Partie IV. Taquin (OCaml) – 30 pts

Le jeu du taquin consiste à déplacer des tuiles adjacentes à la case vide jusqu'à arriver à mettre tous les nombres dans l'ordre. Voici un exemple de positions pour un taquin de dimension 3 :

3	1	2
7	5	
4	6	8

Position initiale

	1	2
3	4	5
6	7	8

Position finale

On représente une position de taquin en OCaml par un int array, en représentant la case vide par 0, et on généralise le problème à une grille de dimensions $n \times n$. On suppose pour toute cette section qu'on a définit une variable globale n qui contient la dimension de la grille. La case numéro i représente le nombre en position $(i/n, i \mod n)$. Une position est dite *valide* si chaque nombre entre 0 et $n^2 - 1$ apparaît une et une seule fois dans le tableau. La position gagnante correspond à une position valide triée.

☐ 31 – Écrire une fonction est_finale : int_array -> bool qui détermine si une position correspond à la position finale.

Réponse 31.

```
let est_finale pos =
  let flag = ref true in
  for i = 0 to Array.length pos - 1 do
    if pos.(i) <> i then flag := false
  done; !flag
```

☐ 32 – Écrire une fonction case_vide : int array -> int qui renvoie l'indice de la case vide.

Réponse 32.

```
let case_vide pos =
  let nn = Array.length pos in
  let rec loop i =
    if i >= nn then raise Not_found
    else if pos.(i) = 0 then i
    else loop (i+1)
  in loop 0
```

☐ 33 – Écrire une fonction est_valide : int array -> bool qui détermine si une position est valide ou non et qui possède une complexité dans le pire des cas $O(n)$, où n est la taille du tableau.

Réponse 33.

```
let est_valide pos =
  let nn = Array.length pos in
  let effectif = Array.make nn 0 in
  for i = 0 to nn - 1 do
    effectif.(pos.(i)) <- effectif.(pos.(i)) + 1
  done;
  let flag = ref true in
  for i = 0 to nn - 1 do
    if effectif.(i) <> 1 then flag := false
  done;
  !flag
```

☐ 34 – Écrire une fonction distance : int −> int −> int qui à partir de deux nombres i et j calcule le nombre de déplacements horizontaux ou verticaux nécessaires pour aller de la position i à la position j. On rappelle que l'on peut utiliser la variable globale n pour avoir directement la dimension de la grille.

Réponse 34. On suppose l'existence d'une variable globale n donnant la dimension de la grille (les tableaux sont de taille $n \times n$).

```
let distance i j =
  let xi, yi = i mod n, i / n in
  let xj, yj = j mod n, j / n in
  abs (xi - xj) + abs (yi - yj)
```

On cherche maintenant à trouver la solution la plus courte au problème, c'est-à-dire en déplaçant le moins de tuiles possible.

☐ 35 – Soit h la fonction qui à partir d'une position, donne le nombre de cases qui sont mal placées.
Montrer que la fonction h est une heuristique admissible.

Réponse 35. $h(p) = 0$ si p est la position finale. Chaque coup déplace une seule tuile, donc le nombre de coups à jouer pour terminer est nécessairement supérieur ou égal au nombre de tuiles mal placées.

☐ 36 – Proposer une autre heuristique admissible f qui domine l'heuristique h, c'est-à-dire que pour toute position x, $f(x) \geq h(x)$.

Réponse 36. Si la tuile i se trouve en position j, il faut au moins distance i j déplacements pour amener cette tuile à sa place. On peut donc choisir comme heuristique f la somme des distance i j pour toutes les tuiles i. On remarque que $f(x) \geq h(x)$ car si distance i j vaut 0, alors la tuile est bien placée.

☐ 37 – Écrire une fonction heuristique : int array −> int qui calcule l'heuristique f.

Réponse 37.

```
let heuristique pos =
  let nn = Array.length pos in
  assert (nn = n * n);
  (* On pourrait recalculer n mais ce n'est
     pas la partie interessante de l'exercice *)
  let dist = ref 0 in
  for i = 0 to nn - 1 do
    dist := !dist + distance i pos.(i)
  done; !dist
```

☐ 38 – Écrire une fonction voisins : int array −> int array list qui à partir d'une position du taquin, renvoie la liste des positions atteignables à partir de celle-ci en un déplacement. On prendra soin de créer des copies du tableau initial à l'aide de la fonction Array.copy : int array −> int array.

Réponse 38.

```
let voisins pos =
  let nn = Array.length pos in
  let z = case_vide pos in
  let res = ref [] in
  let copy_and_change i j =
    let tab = Array.copy pos in
    tab.(i) <- pos.(j); tab.(j) <- pos.(i); tab
```

```
  in
  if z > 0 then res := copy_and_change z (z-1)::!res;
  if z < nn-1 then res := copy_and_change z (z+1)::!res;
  if z >= n then res := copy_and_change z (z-n)::!res;
  if z < nn-n then res := copy_and_change z (z+n)::!res;
  !res
```

On suppose disposer d'une file de priorité de type ('a, 'b) file_prio disposant des fonctions suivantes :

- creer_FP : unit -> ('a, 'b) file_prio : ne prend pas d'argument et crée une file de priorité vide ;
- ajouter_FP : ('a, 'b) file_prio -> 'a -> 'b -> unit : prend en argument une file de priorité, un élément et une priorité et ajoute l'élément dans la file de priorité avec la priorité donnée, ou met à jour sa priorité si l'élément est déjà dans la file ;
- extraire_FP : ('a, 'b) file_prio -> 'a : prend en argument une file de priorité et extrait l'élément de priorité maximale (de plus petite valeur) et renvoie sa valeur.

On rappelle aussi l'existence du module Hashtbl contenant les fonctions suivantes :

- create : int -> ('a, 'b) t : prend en argument un entier n et crée une table de hachage vide de capacité n (redimensionnée automatiquement si nécessaire) ;
- mem : ('a, 'b) t -> 'a -> bool : prend en argument une table de hachage h et une clé k et renvoie **true** si et seulement si h possède une association avec la clé k ;
- add : ('a, 'b) t -> 'a -> 'b -> unit : prend en argument une table de hachage h, une clé k et une valeur v et ajoute l'association (k, v) à h ;
- find : ('a, 'b) t -> 'a -> 'b : prend en argument une table de hachage h et une clé k et renvoie la valeur v telle que h contient l'association (k, v). Lève l'exception Not_found si aucune telle valeur n'existe.

☐ 39 – Écrire une fonction solve : int array -> int array list qui à partir d'une position du taquin, renvoie la liste des positions successives permettant de résoudre le problème en utilisant l'algorithme A*.
On pourra utiliser la fonction Array.to_list : 'a array -> 'a list.

Réponse 39.

```
(* Il faut une structure de donnees persistante pour servir de cle dans une table de hacha
let cle pos = Array.to_list pos

let solve pos =
  let nn = Array.length pos in
  let dist = Hashtbl.create 1 in
  let parent = Hashtbl.create 1 in
  let f = creer_FP () in
  let rec ajoute_liste f d p = function
    | [] -> ()
    | t::q -> ajouter_FP f (t,d,p) (d+heuristique t); ajoute_liste f d p q
  and traite (x,d,p) =
    if not (Hashtbl.mem parent (cle x)) then
      begin
        Hashtbl.add dist (cle x) d;
        Hashtbl.add parent (cle x) p;
        if est_finale x then raise Exit;
        ajoute_liste f (d+1) (Some x) (voisins x)
      end
  and chemin x = match Hashtbl.find parent (cle x) with
    | None -> [x]
    | Some p -> x::chemin p
  in
  ajouter_FP f (pos, 0, None) 0;
  try
    while true do
```

```
      traite (extraire_FP f)
    done; assert false
  with
    Exit -> List.rev (chemin (Array.init nn (fun x -> x)))
```

Fin de l'épreuve