MP2I – DS5 2023-2024 – 4h – sujet B

Pour chaque question, le langage imposé par l'énoncé doit être respecté. La rédaction doit être soignée.

Les fonctions doivent être documentées et commentées à bon escient. Le code doit être indenté et d'un seul tenant (pas de flèche pour dire qu'un bout de code se situe ailleurs par exemple). Le code illisible ne sera pas corrigé.

Sauf précision contraire, les réponses doivent être justifiées.

Tout document interdit. Calculatrice interdite. Téléphone éteint et rangé dans le sac. Pas de montre connectée.

Ce qui est illisible ou trop sale ne sera pas corrigé.
Soulignez ou encadrez vos résultats.

Correction :
Cet énoncé est une combinaison d'une adaptation d'un extrait de l'épreuve d'option indormatique du concours Centrale 2018 et d'un extrait de l'épreuve d'option informatique du concours X/ENS 2017.
Le sujet A était un extrait du sujet B (le début de chaque partie), je mets donc ici la correction du sujet B, il faudra faire un effort de translation sur les numéros de questions de la deuxième partie pour ceux qui avaient le sujet A.

Toute fonction demandée dans une question peut être utilisée dans une question ultérieure, même si elle n'a pas été implémentée. Vous pouvez à tout moment choisir d'introduire des fonctions intermédiaires non explicitement demandées, mais elles doivent être précisément documentées.

On considère le jeu *Lights Out*[1] qui se joue sur une grille carrée de côté N pour un certain entier $N > 3$, chaque case contenant une ampoule. Le site Wikipedia décrit ce jeu de la manière suivante : "Quand le jeu commence, un nombre aléatoire [...] de ces lumières s'allument. Appuyer sur l'une des lumières basculera la position des lumières adjacentes à celle-ci [ainsi que la lumière concernée]. Le but du jeu est d'éteindre toutes les lumières, de préférence avec le moins de coups possibles."
Une *configuration* de *Lights Out* est la donnée des cases allumées et éteintes de la grille. La *grille finale* est la configuration correspondant à une grille où toutes les lumières sont éteintes.

PREMIÈRE PARTIE : TABLES DE HACHAGE

Cette partie est à traiter intégralement en `OCaml`.
Pour rappel, le module `Array` possède une fonction `make : int -> 'a -> 'a array` qui permet de créer un tableau dont toutes les cases ont la même valeur, en spécifiant sa longueur et cette valeur.

On cherche dans cette partie à créer une structure permettant de conserver la liste des configurations déjà obtenues lors d'une partie de *Lights Out*, avec pour chacune le nombre minimal de coups pour l'atteindre à partir de la configuration initiale, de sorte à ce que les opérations de recherche, d'ajout et de mise à jour soient rapides.
On va utiliser à cette fin un dictionnaire. Pour rappel, une structure de dictionnaire est un ensemble de couples (clef, élément), les clefs (nécessairement distinctes) appartenant à un même ensemble K et les éléments à un ensemble E.
Dans l'exemple de *Lights Out*, les clefs sont les configurations du jeu et les éléments sont des entiers contenant la longueur d'une suite de coups allant de la configuration initiale à la grille finale.

Dans la suite, un dictionnaire sera réalisé à l'aide d'une table de hachage possédant w alvéoles et dont les collisions sont résolues par chaînage. On appelle w la *largeur de la table* et on note $h_w : K \to [\![0, w-1]\!]$ la fonction de hachage associée à la table.
On suppose dorénavant que N (la longueur du côté de la grille) est donné par une variable entière `n`.

1. Et je ne peux pas faire un sujet sur ce jeu sans remercier Balthazar qui m'a involontairement inspirée pour ce choix.

1 Tables de hachage de largeur fixée

Dans cette sous-section, on fixe une largeur de hachage w. On définit le type suivant :

```
type ('a, 'b) table_hachage = {
    hache: 'a -> int;
    donnees: ('a * 'b) list array;
    largeur: int
  }
```

1.1 Quelques fonctions utilitaires

Correction :
Cette première partie n'aurait dû poser de problème à personne : on a déjà écrit maintes fois ces fonctions. Il y en a pourtant encore quelques-uns qui n'y arrivent pas...

Question 1 : Écrire une fonction `mem1 x q` de signature `'a -> ('a * 'b) list -> bool` testant l'existence d'un couple dont le premier élément est `x` dans la liste `q`, de complexité temporelle dans le pire des cas linéaire en la longueur de la liste (à justifier brièvement).

Correction :
Pour rappel, on ne peut pas utiliser la valeur d'une variable dans un motif de filtrage.

```
1 let rec mem1 x = function
2   | [] -> false
3   | (k, _) :: suite -> x = k || mem1 x suite
```

Si on note $T(n)$ le nombre d'opérations élémentaires pour un appel sur une liste de longueur n, on a la relation $T(n) \leq 1 + T(n-1)$. Par conséquent, la complexité temporelle est au plus linéaire dans le pire des cas. Cette complexité est atteinte si on cherche un élément qui n'appartient pas à la liste car l'inégalité précédente est dans ce cas une égalité.

Question 2 : Prouver que la fonction `mem1` de la question précédente possède une correction totale.

Correction :
Montrons par récurrence sur la longueur de la liste fournie en argument que la fonction `mem1` possède une correction totale.
cas de base : Si la liste est vide, alors l'appel se termine et renvoie `false`, ce qui est la bonne réponse.
hérédité : Supposons que tout appel se termine et est correct pour une liste de longueur $n \geq 0$ pour un certain n. Soit une liste `lst` de longueur $n+1$. Cette liste n'est pas vide. Si `x` est la valeur de son premier maillon, alors l'appel se termine par le test de la ligne 3 car `||` est paresseux, et renvoie la bonne réponse. Sinon, la valeur `x` apparaît dans `lst` si et seulement si elle apparaît dans `suite`. L'appel récursif se termine avec la bonne réponse par hypothèse de récurrence.
Par le principe de récurrence, la fonction `mem1` possède donc une correction totale.

Question 3 : Écrire une fonction `assoc x q` de signature `'a -> ('a * 'b) list -> 'b` renvoyant, s'il existe, l'élément `y` du premier couple de la forme `(x,y)` appartenant à la liste `q`, et levant l'exception `Not_found` sinon.

Correction :

```
1 let rec assoc x = function
2   | [] -> raise Not_found
3   | (k, e) :: _ when k = x -> e
4   | _ :: suite -> assoc x suite
```

Question 4 : Écrire une fonction `suppr x q` de signature `'a -> ('a * 'b) list -> ('a * 'b) list` qui s'évalue en une liste contenant les mêmes éléments que `q`, excepté le premier couple de la forme `(x, y)` de la liste `q` s'il en existe.

Correction :

```
let rec suppr x = function
  | [] -> []
  | (k, _) :: suite when x = k -> suite
  | _ :: suite -> suppr x suite
```

1.2 Implantation de la structure de dictionnaire

Question 5 : Écrire une fonction `creer_table h w : ('a -> int) -> int -> ('a, 'b) table_hachage` telle que `creer_table h w` renvoie une nouvelle table de hachage vide de largeur `w` et munie de la fonction de hachage `h`.

Correction :

```
let creer_table_dyn h =
  {hache = h; taille = 0; donnees = Array.make 1 []; largeur = 1}
```

Question 6 : Écrire une fonction `recherche t k : ('a, 'b) table_hachage -> 'a -> bool` renvoyant un booléen indiquant si la clé `k` est présente dans la table `t`.

Correction :
Il faut bien sûr utiliser les fonctions déjà écrites dans la partie précédente plutôt que de tout faire : ça va plus vite et on fait moins d'erreurs. Je ne remets pas cette remarque pour les questions suivantes, mais elle reste valide.

```
let recherche (t : ('a, 'b) table_hachage) k = mem1 k t.donnees.(t.hache k)
```

Question 7 : Écrire une fonction `element t k : ('a, 'b) table_hachage -> 'a -> 'b` renvoyant l'élément `e` associé à la clef `k` dans la table `t`, si cette clef est présente dans la table, et levant l'exception `Not_found` sinon.

Correction :

```
let element (t : ('a, 'b) table_hachage) k = assoc k t.donnees.(t.hache k)
```

Question 8 : Écrire une fonction `ajout t k e : ('a, 'b) table_hachage -> 'a -> 'b -> unit` ajoutant l'entrée `(k, e)` à la table de hachage `t`. On n'effectuera aucun changement si la clef est déjà présente.

Correction :

```
let ajout (t : ('a, 'b) table_hachage) k e =
  if not (recherche t k) then
    let i = t.hache k in
    t.donnees.(i) <- (k, e) :: t.donnees.(i)
```

Question 9 : Écrire une fonction `suppression t k : ('a, 'b) table_hachage -> 'a -> unit` supprimant l'entrée de la clef `k` dans la table `t`. On n'effectuera aucun changement si la clef n'est pas présente.

Correction :

```
let suppression (t : ('a, 'b) table_hachage) k =
  if recherche t k then
```

```
3       let i = t.hache k in
4       t.donnees.(i) <- suppr k t.donnees.(i)
```

Question 10 : Écrire une fonction `mise_a_jour t k e : ('a, 'b) table_hachage -> 'a -> 'b -> unit` modifiant la valeur associée à `k` dans la table de hachage `t`, de sorte que cette valeur soit `e`. Si la clef n'est pas présente, le couple `(k, e)` sera ajouté à la table de hachage.

Correction :

```
1 let mise_a_jour t k e =
2   suppression t k; ajout t k e
```

1.3 Étude de la complexité de la recherche d'un élément

Nous étudions ici la complexité de la recherche d'une clef dans une table de hachage où les collisions sont gérées par chaînage.

Question 11 : Montrer que dans le pire des cas, la complexité de recherche d'une clef dans une telle table de hachage est en $\Theta(n)$, où n est le nombre de couples stockés dans la table de hachage.

Correction :
Dans le pire des cas, le complexité de recherche d'une clef se fait en temps linéaire en la longueur de la liste chaînée correspondant à cette clef : on doit parcourir cette liste et on peut être amené à la parcourir intégralement, par exemple si cette clef n'est pas présente. Si la fonction de hachage est telle que toutes les clefs correspondent à la même liste chaînée, alors la complexité est linéaire en le nombre de clefs stockées dans la table.

Si la fonction de hachage h_w est bien choisie, on peut espérer que les clefs vont se répartir de façon relativement uniforme dans les alvéoles, ce qui donnera une complexité bien meilleure.
Nous faisons donc ici l'hypothèse de *hachage uniforme simple* : pour une clef donnée, la probabilité d'être hachée dans l'alvéole i est $1/w$, indépendante des autres clefs. On note n le nombre de clefs stockées dans la table et on appelle $\alpha = n/w$ le *facteur de remplissage* de la table. On suppose de plus, que le calcul de la valeur hachée d'une clef se fait en temps constant.

Question 12 : On se donne une clef k non présente dans la table. Montrer que la complexité en moyenne de la recherche de k dans la table est en $O(1+\alpha)$.

Correction :
Le calcul de la valeur hachée se fait en temps $O(1)$. Le parcours de la liste chaînée associée à la clef se fait en temps $O(\alpha)$ en moyenne.

Question 13 : On prend au hasard une clef présente dans la table ; toutes les clefs sont équiprobables. Montrer qu'alors la recherche de la clef se fait en $O(1+\alpha)$, en moyenne sur toutes les clefs présentes.

Correction :
Attention : ici il ne s'agit pas du résultat vu en cours. On veut une moyenne sur les clefs présentes.
Le calcul de la valeur hachée se fait en temps $O(1)$. Si la clef est présente, sa recherche se fait en temps proportionnel à la place de cette clef dans la liste chaînée correspondante. On note n le nombre de clefs présentes ; le nombre d'élément dans chaque liste est α par hypothèse.

On a donc :

$$\begin{aligned}
\text{indice moyen d'une clef} &= \frac{1}{n}\sum_{c \text{ clef présente}} \text{indice de } c \text{ dans sa liste chaînée} \\
&= \frac{1}{n}\sum_{i=0}^{w-1}\sum_{c \text{ dans la liste } i} \text{position de } c \text{ dans sa liste} \\
&= \frac{1}{n}\sum_{i=0}^{w-1}\frac{\alpha(\alpha+1)}{2} \\
&= \frac{w}{n}\frac{\alpha(\alpha+1)}{2} \\
&= \frac{\alpha+1}{2}
\end{aligned}$$

d'où le résultat.

1.4 Une fonction de hachage h_w pour le jeu *Lights Out*

Construisons un fonction de hachage h_w, pour une liste de couples de $[\![0, N-1]\!]^2$, cette liste correspondant aux coordonnées des cases allumées dans une configuration de *Lights Out*, en lisant les cases de gauche à droite et de haut en bas (comme quand on lit un texte en Français). Un hachage naturel d'une liste comportant les couples $(a_i, b_i)_{0\leq i<p}$ avec $0 \leq a_i, b_i < N-1$ est donné par :

$$P_w(N) = \left(\sum_{i=0}^{p-1}(a_i + b_i N)N^{2i}\right) \text{ modulo } w$$

On suppose que le type `int` d'`OCaml` est suffisant pour représenter tous les entiers manipulés dans l'énoncé.

Question 14 : Écrire une fonction récursive `hachage_liste w q : int -> (int * int) list -> int` calculant la quantité précédente. Cette fonction doit être linéaire en la longueur de la liste passée en argument.

Correction :

```
1 let hachage_liste w q =
2   let rec hach acc nPuissance2i = function  (* calcule la somme sans le modulo *)
3     | [] -> acc
4     | (ai, bi) :: suite -> hach (acc + (ai + bi*n)*nPuissance2i) (nPuissance2i*n*
    n) suite
5   in (hach 0 1 q) mod w
```

2 Tables de hachage dynamique

Les questions 12 et 13 montrent que l'on peut assurer une complexité moyenne constante pour la recherche dans une table de hachage, sous réserve que le facteur de remplissage α soit borné. Il en va de même des opérations d'insertion et de suppression, pour peu que les clefs à ajouter/supprimer vérifient des hypothèses d'indépendance. Bien souvent, et cela va être le cas dans notre problème, on ne sait pas à l'avance quel sera le nombre de clefs à stocker dans la table, et on préfère ne pas surestimer ce nombre pour garder un espace mémoire linéaire en le nombre de clefs stockées. Ainsi, il est utile de faire varier la largeur w de la table de hachage : si le facteur de remplissage devient trop important, on réarrange la table sur une largeur plus grande (de même, on peut réduire la largeur de la table lorsque le facteur de remplissage devient petit). On parle alors de tables de hachage *dynamiques* pour ces tables à largeur variable.
À une table de hachage dynamique est associée une *famille de fonctions de hachage* $(h_w)_w$.

On définit le type suivant :

```
type ('a,'b) table_dyn = {
    hache: int -> 'a -> int;
    mutable taille: int;
    mutable donnees: ('a * 'b) list array;
    mutable largeur: int
```

```
}
```

On notera trois différences par rapport au type précédent :
- la fonction `hache` possède un paramètre supplémentaire qui est la largeur de hachage, elle correspond maintenant à la famille de fonctions de hachage $(h_w)_w$;
- on a rendu les champs `donnees` et `largeur` mutables ;
- un champ `taille` (mutable) est rajouté, il contient à tout moment le nombre de clefs présentes dans la table.

Question 15 : Écrire une fonction `creer_table_dyn h` permettant de créer une table de hachage dynamique initialement vide, avec la famille de fonctions de hachage `h` et la largeur initiale 1.

Correction :

```
let creer_table_dyn h =
  {hache = h; taille = 0; donnees = Array.make 1 []; largeur = 1}
```

On admet avoir écrit deux fonctions `recherche_dyn t k` et `element_dyn t k`, variantes des fonctions `recherche` et `element` précédentes, basées sur le même principe. On va maintenant développer une stratégie pour maintenir à tout moment un facteur de remplissage borné.

Question 16 : Écrire une fonction `rearrange_dyn t w2` prenant en entrée une table de hachage dynamique et une nouvelle largeur de hachage `w2`, qui réarrange la table sur une largeur `w2`. En supposant que le calcul des valeurs de hachage se fasse en temps constant, la complexité doit être en $O(n+w+w_2)$ où n est le nombre de clefs présentes dans la table (sa taille), w est l'ancienne largeur de la table, w_2 la nouvelle.

Correction :

```
(* insere dans le tableau de listes donnees2 les éléments
qui sont dans la liste lst qui vient de t *)
let rec insere t donnees2 lst w2 =
  match lst with
  | [] -> ()
  | (k, v) :: suite -> begin
      if not (recherche_dyn t k)
      then let i = t.hache w2 k in
           donnees2.(i) <- (k, v) :: donnees2.(i);
           insere t donnees2 suite w2
    end


let rearrange_dyn t w2 =
  let donnees2 = Array.make w2 [] in
  for i = 0 to t.largeur - 1 do
    insere t donnees2 t.donnees.(i) w2
  done;
  t.donnees <- donnees2;
  t.largeur <- w2
```

Une stratégie heuristique simple pour garantir que le facteur de remplissage reste borné, tout en garantissant une bonne répartition des clefs dans le cas des listes de couples à valeurs dans $[\![0, N-1]\!]$ avec $N = 16$, est d'utiliser les puissances de 3 comme largeurs de hachage. Après ajout d'un élément à la table, si celle-ci est de taille strictement supérieure à trois fois sa largeur w, on la réarrange sur une largeur $w' = 3w$.

Question 17 : Écrire une fonction `ajout_dyn t k e` ajoutant le couple `(k, e)` à la table de hachage (si la clef `k` n'est pas présente), en réarrangeant si nécessaire la table, en suivant le principe ci-dessus.

Correction :

```
let ajout_dyn (t : ('a, 'b) table_dyn) k e =
  if not (recherche_dyn t k)
  then begin
      let i = t.hache t.largeur k in
      t.donnees.(i) <- (k, e) :: t.donnees.(i);
      t.taille <- t.taille + 1;
      if t.taille > 3*t.largeur then rearrange_dyn t  (3*t.largeur)
    end
```

Question 18 : Dans l'hypothèse que chaque ajout qui ne provoque pas d'agrandissement de la table se fait en temps $O(1+\alpha)$, où α est le facteur de remplissage de la table, montrer que la complexité amortie des ajouts dans une table initialement vide est constante.

Correction :
On peut utiliser la méthode du banquier en prévoyant 3 jetons pour chaque insertion.

DEUXIÈME PARTIE : JEUX À UN JOUEUR ET SOLUTIONS OPTIMALES

Cette partie est à traiter intégralement en C.
On suppose qu'on dispose des types suivants :

- `configuration` pour représenter une configuration du jeu ;
- `ensemble` pour représenter un ensemble (au sens mathématique du terme : aucun doublon possible) de configurations.

On suppose également disposer des fonctions suivantes pour gérer les ensembles de configuration :

```
/** renvoie un ensemble vide */
ensemble ensemble_vide();
/** teste si l'ensemble e est vide */
bool est_vide(ensemble e);
/** ajoute la configuration c dans l'ensemble e */
void ajout(ensemble *e, configuration c);
/** teste si la configuration c appartient à l'ensemble e */
bool appartient(ensemble e, configuration c);
/** ajoute les éléments de l'ensemble e2 à l'ensemble e1 */
void union(ensemble *e1, ensemble e2);
/** renvoie une configuration de e s'il est non vide en la supprimant de e,
* NULL sinon */
configuration *un_element(ensemble *e);
/** libère la mémoire associée à l'ensemble e */
void liberation(ensemble e);
```

L'objectif de cette partie est d'étudier différents algorithmes pour trouver des solutions minimales en nombre de coups pour le jeu *Lights Out*.
La section 3 introduit la notion de jeu à un joueur et un premier algorithme pour trouver une solution optimale. Les section 4 et 5 proposent d'autres algorithmes, plus efficaces.

3 Jeu à un joueur, parcours en largeur (BFS : *Breadth-First Search*)

Un jeu à un joueur est la donnée d'un ensemble non vide E, d'un élément $e_0 \in E$, d'une fonction $s : E \to \mathscr{P}(E)$, où $\mathscr{P}(E)$ désigne l'ensemble des parties de E, et d'un sous-ensemble F de E. L'ensemble E représente les configurations possibles du jeu. L'élément e_0 est la configuration initiale. Pour une configuration e, l'ensemble $s(e)$ représente toutes les configurations atteignables en un coup à partir de e. Enfin, F est l'ensemble des configurations gagnantes du jeu. On dit qu'une configuration e_p est à la *profondeur* p s'il existe une séquence finie de $p+1$ configurations

$$e_0\ e_1\ \ldots\ e_p$$

avec $e_{i+1} \in s(e_i)$ pour tout $0 \leq i < p$. Si par ailleurs $e_p \in F$, une telle séquence est appelée une *solution* du jeu, de profondeur p. Une solution *optimale* est une solution de profondeur minimale. On notera qu'un même configuration peut être à plusieurs profondeurs différentes.
Voici un exemple de jeu :

$$\begin{aligned} E &= \mathbb{N}^* \\ e_0 &= 1 \\ s(n) &= \{2n, n+1\} \end{aligned} \tag{1}$$

Question 19 : Donner une solution optimale pour le jeu (1) lorsque F = {42} (sans avoir à justifier son optimalité).

Correction :
L'énoncé demandant explicitement de ne pas justifier, ***ce n'est pas la peine de justifier****, ça prend du temps et ça ne rapporte rien...*
$1 \to 2 \to 4 \to 5 \to 10 \to 20 \to 21 \to 42$

Parcours en largeur. Pour chercher une solution optimale pour un jeu quelconque, on peut utiliser un parcours en largeur. Un pseudo-code pour un tel parcours est donné par l'algorithme 1.

Algorithme 1 BFS : Parcours en largeur d'un jeu d'ensemble de configurations E et de fonction de succession $s : E \to \mathcal{P}(E)$

entrée : configuration initiale $e_0 \in E$, ensemble de configurations gagnantes $F \subseteq E$
sortie : teste l'existence d'une solution *via* un parcours en largeur des configurations du jeu

```
1 : A ← {e_0}
2 : tant que A ≠ ∅ faire
3 :     B ← ∅
4 :     pour tout x ∈ A faire
5 :         si x ∈ F alors
6 :             renvoyer vrai
7 :         B ← B ∪ s(x)
8 :     A ← B
9 : renvoyer faux
```

Question 20 : Montrer que le parcours en largeur renvoie vrai si et seulement si une configuration gagnante est atteignable à partie de e_0 ~~(c'est-à-dire que l'algorithme~~ 1 ~~possède une correction totale)~~.

Correction :
L'indication entre parenthèse était fausse (ce n'était pas voulu), car l'algorithme peut tout à fait ne pas s'arrêter. Je suis d'autant plus désolée qu'Anaïs me l'a signalé pendant l'épreuve mais je n'ai pas percuté sur le moment.
Montrons qu'on a l'invariant suivant pour la boucle de la ligne 2 : **(I)** "A contient exactement les configurations atteignables en i coups à partir de e_0" (i représente le numéro de l'itération)
Avant la boucle, $i = 0$ et A contient e_0, **(I)** est donc vérifié.
Supposons qu'au début de l'itération i, **(I)** soit vérifié. Alors, juste avant la ligne 8, B contient exactement les configurations atteignables en un coup à partir d'une configuration de A, soit exactement les configurations atteignables en $i+1$ coups à partir de e_0, par hypothèse sur A. Donc **(I)** est vérifié en fin d'itération.
Le parcours en largeur renvoie vrai si et seulement s'il pase par la ligne 6, donc si et seulement si A contient à un moment une configuration gagnante atteignable à partir de e_0 en i coups pour un certain i.

Question 21 : On se place dans le cas particulier du jeu (1) pour un ensemble F arbitraire pour lequel le parcours en largeur donné par l'algorithme 1 termine. Montrer alors que la complexité en temps et en espace est exponentielle en la profondeur p de la solution trouvée. On demande de montrer que la complexité est bornée à la fois inférieurement et supérieurement par deux fonctions exponentielles en p.

Correction :
Dans le cadre du jeu (1), le cardinal de l'ensemble A est au plus doublé à la fin de chaque itération de la boucle ligne 2 car l'image d'un élément par s est de cardinal 2. Comme on parcourt tous les éléments de A avant de passer à l'itération suivante (si on y passe), on a parcouru au plus $\sum_{i=0}^{p-} 2^i = 2^{p+1} - 1$ configurations au moment où l'algorithme arrive à une configuration gagnante et on a stocké au plus 2^p configuration. Les complexités en

temps et en espace sont donc en $O(2^p)$ (en supposant que l'opération d'union est en temps constant).
Par ailleurs, à chaque itération, on récupère tous les doubles des éléments de A en appliquant $n \mapsto 2n$, soit au moins autant d'éléments que dans A. Ces doubles sont tous pairs. On récupère également un certain nombre d'entiers impairs en prenant les entiers pairs de A et en leur appliquant $n \mapsto n+1$. Or il y a au moins la moitié des entiers de A qui sont pairs (par une récurrence immédiate), on a donc au moins multiplié le cardinal de A par $\frac{3}{2}$. Les complexités temporelles et spatiales sont donc en $\Omega((\frac{3}{2})^n)$.

Question 22 : On suppose qu'on dispose d'une fonction `ensemble s(configuration c)` qui donne toutes les configuration atteignables en un coup à partir de `c`. Écrire une fonction

```
int bfs(configuration e0, ensemble F)
```

qui effectue un parcours en largeur à partir de la configuration initiale `e0` et renvoie la profondeur de la première solution trouvée dans `F`. Lorsqu'il n'y a pas de solution, le comportement de cette fonction pourra être arbitraire.

Correction :
Je vous la laisse à faire.

Question 23 : Montrer que la fonction `bfs` renvoie toujours une profondeur optimale lorsqu'une solution existe.

Correction :
Je fais ça sur l'algo du coup. On l'a en fait déjà prouvé en montré la correction partielle de l'algorithme puisque les configurations atteignables sont visitées par distance croissante à e_0 et qu'on renvoie un résultat dès qu'on trouve une configuration gagnante.

Correction :
Je vous laisse la fin à faire tous seuls car ça ressemble à la suite du cours.

4 Parcours en profondeur (DFS : *Depth-First Search*)

Comme on vient de le montrer, l'algorithme BFS permet de trouver une solution optimale mais il peut consommer un espace important pour cela, comme illustré dans le cas particulier du jeu (1) qui nécessite un espace exponentiel. On peut y remédier, on utilise un parcours en profondeur. L'algorithme 2 contient le pseudo-code d'une fonction DFS effectuant un parcours en profondeur à partir d'une configuration e de profondeur p, sans dépasser une profondeur maximale m donnée.

Algorithme 2 DFS : Parcours en profondeur, limité par une profondeur maximale, d'un jeu d'ensemble de configurations E, d'ensemble de configurations gagnantes $F \subseteq E$ et de fonction de succession $s : E \to \mathscr{P}(E)$

entrée : configuration $e \in E$, entiers p et m
sortie : teste l'existence d'une solution à partir de e de profondeur p, *via* un parcours en profondeur des configurations du jeu, sans dépasser la profondeur maximale m

```
1 : si p > m alors
2 :     renvoyer faux
3 : si e ∈ F alors
4 :     renvoyer vrai
5 : pour tout x ∈ s(e) faire
6 :     si DFS(x, p+1, m) est vrai alors
7 :         renvoyer vrai
8 : renvoyer faux
```

Question 24 : Montrer que $\text{DFS}(e_0, 0, m)$ renvoie vrai si et seulement si une solution de profondeur inférieure ou égale à m existe.

Recherche itérée en profondeur. Pour trouver une solution optimale, une idée simple consiste à effectuer un parcours en profondeur avec $m = 0$, puis avec $m = 1$, puis avec $m = 2$, etc., jusqu'à ce que $\text{DFS}(e_0, 0, m)$ renvoie vrai.

Question 25 : Écrire une fonction

```
int ids(configuration e0, ensemble F);
```

qui effectue une recherche itérée en profondeur et renvoie la profondeur d'une solution optimale. Lorsqu'il n'y a pas de solution, cette fonction ne termine pas.

Question 26 : Montrer que la fonction `ids` renvoie toujours une profondeur optimale lorsqu'une solution existe.

Question 27 : Comparer les complexités en temps et en espace du parcours en largeur et de la recherche itérée en profondeur dans les deux cas particuliers suivants :

1. il y a exactement une configuration à chaque profondeur p ;
2. il y a exactement 2^p configurations à la profondeur p.

On demande de justifier les complexités qui seront données.

5 Parcours en profondeur avec horizon

On peut améliorer encore la recherche d'une solution optimale en évitant de considérer successivement toutes les profondeurs possibles. L'idée consiste à introduire une fonction $h : E \to \mathbb{N}$ qui, pour chaque configuration, donne un minorant du nombre de coups restant à jouer avant de trouver une solution. Lorsqu'une configuration ne permet pas d'atteindre une solution, cette fonction peut renvoyer n'importe quelle valeur.
Commençons par définir la notion de *distance* entre deux configurations. S'il existe une séquence de $k+1$ configurations $x_0 \ \dots \ x_k$ avec $x_{i+1} \in s(x_i)$ pour tout $0 \leq i < k$, on dit qu'il y a un *chemin de longueur k* entre x_0 et x_k. Si de plus k est minimal, on dit que la *distance* entre x_0 et x_k est k.
On dit alors que la fonction h est *un horizon admissible* si elle ne surestime jamais la distance entre une configuration et une solution, c'est-à-dire que pour toute configuration e, il n'existe pas de configuration $f \in F$ située à une distance de e strictement inférieure à $h(e)$.
On procède alors comme pour la recherche itérée en profondeur, mais pour chaque configuration e considérée à la profondeur p on s'interrompt dès que $p + h(e)$ dépasse la profondeur maximale m (au lieu de s'arrêter simplement lorsque $p > m$). Initialement, on fixe m à $h(e_0)$. Après chaque parcours en profondeur infructueux, on donne à m la plus petite valeur $p + h(e)$ qui a dépassé m pendant ce parcours, le cas échéant, pour l'ensemble des configurations e rencontrées dans ce parcours. L'algorithme 3 donne le pseudo-code d'un tel algorithme appelé IDA*, où la variable globale min est utilisée pour retenir la plus petite valeur ayant dépassé m.

Algorithme 3 IDA* : Parcours IDA*d'un jeu d'ensemble de configurations E, d'ensemble de configurations gagnantes $F \subseteq E$, de fonction de succession $s : E \to \mathcal{P}(E)$ et d'horizon admissible $h : E \to \mathbb{N}$

entrée : configuration initiale $e_0 \in E$
sortie : teste l'existence d'une solution à partir de e_0 *via* un parcours en profondeur avec horizon

```
m ← h(e_0)
tant que m ≠ ∞ faire
    min ← ∞
    si DFS*(e_0, 0, m, min) est vrai alors
        renvoyer vrai
    m ← min
renvoyer faux
```

Algorithme 4 DFS* : Parcours DFS*d'un jeu d'ensemble de configurations E, d'ensemble de configurations gagnantes $F \subseteq E$, de fonction de succession $s : E \to \mathcal{P}(E)$ et d'horizon admissible $h : E \to \mathbb{N}$

entrée : configuration $e \in E$, entiers p et m, entier (éventuellement infini) min
sortie : teste l'existence d'une solution à partir de e de profondeur p, *via* un parcours en profondeur avec horizon des configurations du jeu, sans dépasser la profondeur maximale m, modifie la valeur de min

```
c ← p + h(e)
si c > m alors
    si c < min alors
        min ← c
    renvoyer faux
si e∈F alors
    renvoyer vrai
pour tout x ∈ s(e) faire
    si DFS*(x, p + 1, m, min) est vrai alors
        renvoyer vrai
renvoyer faux
```

Question 28 : Écrire une fonction

```
int idastar(configuration c, ensemble F);
```

qui réalise l'algorithme IDA*et renvoie la profondeur de la première solution rencontrée. Lorsqu'il n'y a pas de solution, le comportement de cette fonction pourra être arbitraire. Il est suggéré de décomposer le code en plusieurs fonctions. On précisera le type et la valeur retenue pour représenter ∞.

Question 29 : Proposer un horizon admissible h pour le jeu (1), non constant, en supposant que l'ensemble F est un singleton $\{t\}$ avec $t \in \mathbb{N}^*$. On demande de justifier que h est un horizon admissible.

Question 30 : Montrer que si h est un horizon admissible, la fonction `idastar` renvoie toujours une profondeur optimale lorsqu'une solution existe.