Devoir surveillé d'informatique

Consignes

- Les programmes demandés doivent être écrits en C ou en OCaml. Dans le cas du C, on suppose que les librairies standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.h>`, `<stdassert.h>`, ...) sont déjà importées.
- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

❑ Exercice 1 : *Questions de cours*

Q1– Donner la définition d'un arbre binaire.

Un arbre binaire est une structure de données hiérarchique composée de noeuds définie récursivement, en effet un arbre binaire est :

— soit vide, on le note alors $\varnothing$

— soit un noeud c'est à dire un triplet (g, v, d) où g et d sont deux arbres binaires et v l'étiquette.

Q2– Donner les définitions de la hauteur et de la taille d'un arbre binaire.

- Le nombre de noeuds d'un arbre binaire A, noté $n(A)$, se définit récursivement par :
$$\begin{cases} n(A) = 0 & \text{si } A \text{ est vide} \\ n(A) = 1 + n(g) + n(d) & \text{si } A = (g, a, d) \end{cases}$$
- La hauteur d'un arbre binaire A, noté $h(A)$, se définit récursivement par :
$$\begin{cases} h(A) = -1 & \text{si } A \text{ est vide} \\ h(A) = 1 + \max(h(g), h(d)) & \text{si } A = (g, a, d) \end{cases}$$

Q3– Donner la définition d'un arbre binaire de recherche.

Un arbre binaire de recherche (noté ABR), est un arbre binaire tel que :

— Les étiquettes des noeuds, appelées clés sont toutes comparables entre elles.

— Pour tous les noeuds (g, v, d) l'ensemble des clés présentes dans le sous arbre gauche g (resp. droit d) sont strictement inférieures (resp. supérieures) à v.

— Les clés sont uniques.

Q4– Prouver le parcours infixe d'un arbre binaire de recherche fournit les clés dans l'ordre croissant.
Indication : on pourra raisonner par récurrence sur la taille de l'arbre.

Pour tout $n \in \mathbb{N}$, on note $\mathcal{P}(n)$ la propriété : « le parcours infixe d'un ABR de taille n fournit les clés dans l'ordre croissant ».
• Initialisation : $\mathcal{P}(0)$ est vraie puisque le parcours infixe d'un arbre vide est vide et donc rangé dans l'ordre croissant.
• Hérédité : soit $n \in \mathbb{N}$, tel que pour tout $k \leqslant n$, $\mathcal{P}(k)$ est vrai, montrons alors que $\mathcal{P}(n+1)$ est vraie. Soit un arbre binaire de taille $n+1$, alors cet arbre n'est pas vide et donc c'est un triplet (g, e, d) où g et d sont des arbres binaires de taille inférieure ou égale à n. En notant $p(a)$ le parcours prefixe d'un arbre a, on a par définition du parcours infixe : $p((g, e, d)) = (p(g), e, p(d))$, on considère maintenant x, y deux éléments apparaissant *dans cet ordre* dans $(p(g), e, p(d))$ et on raisonne par disjonction de cas :

— $x \in p(g)$ et $y \in p(g)$, par hypothèse de récurrence le parcours infixe de g est rangé dans l'ordre croissant et donc $x < y$.

— $x \in p(g)$ et $y = e$, par la propriété des ABR les clés du sous arbres gauche sont strictement inférieur à la racine donc $x < y$.

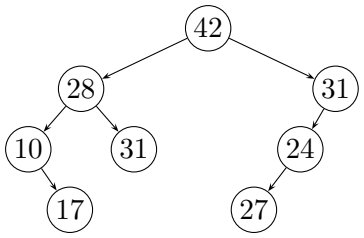
— $x \in p(g)$ et $y \in p(d)$, par la propriété des ABR, $x < e$ et $e < y$, donc $x < y$.

— $x = e$ et $y \in p(d)$, par la propriété des ABR, $y > e$ donc $x < y$.

— $x \in p(d)$ et $y \in p(d)$, par hypothèse de récurrence le parcours infixe de d est rangé dans l'ordre croissant et donc $x < y$.

Donc $P(n+1)$ est vraie.

Q5– Donner l'ordre des noeuds lors des parcours prefixe, infixe et suffixe de l'arbre suivant :



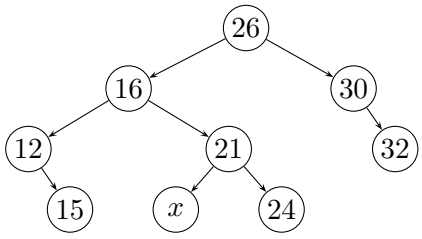
Voici l'ordre des noeuds dans chacun des parcours :

— Prefixe : $42, 28, 10, 17, 31, 31, 24, 27$

— Infixe : $10, 17, 28, 31, 42, 27, 24, 31$

— Suffixe : $17, 10, 31, 28, 27, 24, 31, 42$

Q6– On considère l'arbre binaire suivant :



Donner les valeurs de l'étiquette x pour lesquelles cet arbre est un arbre binaire de recherche.

On peut écrire le parcours infixe de cet arbre : $12, 15, 16, x, 21, 24, 26, 30, 32$ et donc les valeurs de x pour lesquelles cet arbre est un ABR sont celles de l'intervalle $]16, 21[$ en supposant x entier, les valeurs possibles de x sont donc $17, 18, 29$ et 20.

Q7– On implémente les arbres binaires de recherche en OCaml à l'aide du type suivant :

```
type abr =
  | Vide
  | Noeud of  abr * int * abr;;
```

Ecrire une fonction `insere : int -> abr -> abr` qui prend en argument un entier x et un arbre binaire de recherche a et renvoie un arbre binaire de recherche contenant x et tous les éléments de a.

```
let rec insere abr nv =
  match abr with
  | Vide -> Noeud(Vide,nv,Vide)
  | Noeud(g,v,d) -> if nv<v then Noeud(insere g nv, v, d) else Noeud(g, v, insere
  ↪  d nv);;
```

❑ **Exercice 2** : *Valeur plus petite la plus proche*

On considère un tableau d'entiers *positifs* et on s'intéresse au problème de la recherche pour chacun de ces entiers de la valeur plus petite la plus proche située à gauche dans le tableau. Dans le cas où aucune valeur située à gauche dans le tableau n'est plus petite que la valeur considérée alors on renverra -1.
Par exemple dans le tableau $\{2, 1, 7, 9, 8, 3\}$:

- Il n'y a aucune valeur à gauche de 2, donc la valeur plus petite la plus proche est -1,
- Pour 1, aucune valeur située à gauche n'est plus petite, donc on renvoie aussi -1,
- Pour 7, la valeur plus petite la plus proche est 1.
- Pour 9, c'est 7.
- Pour 8 c'est 7.
- Pour 3, c'est 1.

Et donc le tableau des valeurs plus petites les plus proches dans cet exemple est $\{-1, -1, 1, 7, 7, 1\}$

Q8– Donner le tableau des valeurs plus petites les plus proches pour le tableau $\{5, 7, 11, 6, 9, 2\}$

On obtient : $\{-1, 5, 7, 5, 6, -1\}$

Q9– On propose l'algorithme suivant pour résoudre ce problème : pour chaque élément `tab[i]` du tableau on parcourt les valeurs `tab[i-1], ..., tab[0]` *dans cet ordre*, si on trouve un élément strictement inférieur à `tab[i]` alors c'est la valeur plus petite la plus proche, sinon la valeur plus petite la plus proche est -1. Ecrire une implémentation de cet algorithme en C sous la forme d'une fonction de signature `int *vpp_naif(int tab[], int size)` qui prend en argument un tableau d'entiers `tab` ainsi que sa taille `size` et un renvoie un tableau de taille `size` contenant à l'indice `i` la valeur strictement inférieure la plus proche de `tab[i]`.

```c
int *vpp_naif(int tab[], int size)
// Renvoie pour chaque indice i, l'emplacement de plus proche valeur inférieure -1 si inexistant
{
    int *nsv = malloc(sizeof(int) * size);
    int idx;
    nsv[0] = -1;
    for (int i = 1; i < size; i++)
    {
        idx = i - 1;
        while (idx >= 0 && tab[idx] >= tab[i])
        {
            idx--;
        }
        if (idx < 0)
        {
            nsv[i] = -1;
        }
        else
        {
            nsv[i] = tab[idx];
        }
    }
    return nsv;
}
```

Q10– Justifier rapidement que l'algorithme précédent a une complexité quadratique

Pour chaque indice `i` du tableau, on parcourt dans le pire cas, le sous tableau `i-1, ..., 0` en effectuant des opérations élémentaires. On effectue donc au plus $1 + 2 + \ldots (n-1)$ opérations élémentaires. Et donc la complexité est quadratique.

On considère maintenant l'algorithme suivant qui utilise une pile dotée de son interface usuelle (`est_vide, empiler, depiler`) et de la fonction `sommet` qui renvoie la valeur située au sommet de la pile sans la dépiler.

Algorithme : Valeurs plus petites les plus proches

Entrées : Un tableau t d'entiers positifs de taille n
Sorties : Un tableau s d'entiers positifs de taille n tel que $s[i]$ soit la valeur plus petite la plus proche de $t[i]$

```
s ← tableau de taille n
p ← pile de taille maximale n
pour i ← 0 à p − 1 faire
    tant que p n'est pas vide et sommet(p) ⩾ t[i] faire
        depiler(p) ;
    fin
    si p est vide alors
        s[i] ← −1
    fin
    sinon
        s[i] ← sommet(p)
    fin
    empiler t[i] dans p
fin
return s
```

Q11– On fait fonctionncer cet algorithme sur le tableau $\{2, 7, 5, 8, 6, 3\}$. Recopier et compléter le tableau suivant qui indique pour chaque valeur de l'indice `i` de la boucle `for` l'état de la pile et du tableau s

après l'exécution de la boucle pour les valeurs de `i` de 0 à 5 (on note une pile avec les extrémités `|` et `>` pour indiquer le sommet de la pile)

i	État de la pile	État du tableau s
Initialement	`|>`	$\{-1,-1,-1,-1,-1,-1\}$
0	`|2>`	$\{-1,-1,-1,-1,-1,-1\}$
1	`|2, 7>`	$\{-1,2,-1,-1,-1,-1\}$
2	`|2, 5>`	[-1, 2, 2, -1, -1, -1]
3	`|2, 5, 8>`	[-1, 2, 2, 5, -1, -1]
4	`|2, 5, 6>`	[-1, 2, 2, 5, 5, -1]
5	`|2, 3>`	[-1, 2, 2, 5, 5, 2]

Q12– On suppose qu'on a *déjà implémentée* en C une structure de donnée de pile qu'on manipule à l'aide des fonctions suivantes :

— `est_vide` de signature `bool est_vide(pile p)`,

— `empiler` de signature `void empiler(pile *p, int v)`,

— `depiler` de signature `int depiler(pile *p)`.

Ecrire en utilisant ces fonctions une fonction `sommet` de signature `int sommet(pile *p)` qui renvoie le sommet de la pile sans le depiler si la pile n'est pas vide et -1 sinon.

```c
int sommet(pile *p)
{
    if (est_vide(*p))
    {
        return -1;
    }
    int temp = depiler(p);
    empiler(p, temp);
    return temp;
}
```

Q13– Ecrire une implémentation en C de l'algorithme des valeurs plus petites les plus proches donné ci-dessus et utilisant une pile sous la forme d'une fonction de signature `int *vpp_pile(int tab[], int size)` qui renvoie le tableau des valeurs plus petites les plus proches.

```c
int *vpp_pile(int tab[], int size)
{
    pile p = cree_pile(size);
    int *nsv = malloc(sizeof(int) * size);
    for (int i = 0; i < size; i++)
    {
        while (!est_vide(p) && sommet(&p) >= tab[i])
        {
            depiler(&p);
        }
        nsv[i] = sommet(&p);
        empiler(&p, tab[i]);
    }
    return nsv;
}
```

Q14– Prouver que cet algorithme est de complexité linéaire, on pourra vérifier que chaque élément du tableau t est empilé une fois et dépilé au plus une fois.

La boucle `for` ne contient que des instructions depiler, empiler et d'affectation dans le tableau `nsv`, chaque élément n'est empilé qu'une seule fois il y en donc `n` opérations `empiler` en tout, on ne peut donc pas dépiler plus de `n` fois et donc l'algorithme est de complexité linéaire.

❑ **Exercice 3** : *Base de données de publications scientifiques*

On utilise le schéma relationnel suivant afin de modéliser une base de données de publications scientifiques. Chaque article publié ayant un ou plusieurs auteurs.

- **Article** (IdArticle, titre, revue, volume, annee)
- **Auteur** (IdAuteur, nom, prenom)
- **Publie** (#Article,#Auteur)

La clé étrangère #Article de la table **Publie** fait référence à la clé primaire de la table **Article** et la clé étrangère #Auteur de la table **Publie** fait référence à la clé primaire de la table **Auteur**. Les attributs titre, revue, nom et prenom sont des chaines de caractères, les autres sont des entiers.

Q15– Justifier que l'attribut #Article de la table **Publie** seul, ne peut pas servir de clé primaire pour cette table.

L'énonce indique qu'un article peut avoir plusieurs auteurs, par conséquent dans la table publie, plusieurs enregistrements peuvent avoir la même valeur pour le champ Article. Donc cette valeur n'est pas unique pour chaque enregistrement et donc ne peut pas servir de clé primaire.

Q16– Expliquer ce qu'affiche la requête suivante :

```
SELECT nom, prenom
FROM Auteur
JOIN Publie ON Auteur.IdAuteur = Publie.Auteur
WHERE Publie.Article = 42
```

Cet requête affiche les noms et prénoms des auteurs de l'article ayant l'IdArticle 42.

Q17– Ecrire une requête permettant d'obtenir la liste des titres des articles parus en 2022 listé par ordre alphabétique.

```
SELECT titre
FROM Article
WHERE annee = 2022
ORDER BY titre ASC ;
```

Q18– Ecrire une requête permettant d'obtenir les noms des revues listé par ordre alphabétique. On souhaite obtenir cette liste *sans répétition* des noms de revues.

```
SELECT DISTINCT revue
FROM Article
ORDER BY titre ASC ;
```

Q19– Ecrire une requête permettant d'obtenir les noms et prénoms des auteurs qui ont publié dans la revue "Nature" en 2000.

```
SELECT nom, prenom FROM Auteur
JOIN Publie ON Publie.Article = Auteur.IdAuteur
JOIN Article ON Article.IdArticle = Publie.Article
WHERE Article.revue = "Nature" AND Article.annee = 2000
```

Q20– Ecrire une requête permettant d'obtenir les titres et revues des articles écrits (ou co-écrit) par Donald Knuth en 2010.

```sql
SELECT titre, revues FROM Article
JOIN Publie ON Publie.Article = Auteur.IdAuteur
JOIN Article ON Article.IdArticle = Publie.Article
WHERE Auteur.prenom = "Donald" AND Auteur.nom= "Knuth" AND Article.annee = 2010
```

Q21– Ecrire une requête permettant d'obtenir la liste des volumes de la revue "Nature" en 2020 avec le nombre d'article qu'il contient.

```sql
SELECT volume, COUNT(*) FROM Article
GROUPE BY volume
WHERE Article.annee = 2020 and Article.revue = "Nature"
```

Q22– Ecrire une requête permettant d'obtenir pour chaque revue, son nom et l'année de publication de son article le plus ancien.

```sql
SELECT revue, MIN(annee) FROM Article
GROUPE BY revue
```

❑ **Exercice 4** : *Hachage de chaine de caractères*

Le langage d'implémentation dans cet exercice est le langage C, on suppose déjà importées les librairies `<stdint.h>` et `<string.h>` et on s'intéresse aux fonction de hachages sur des chaines de caractères constituées uniquement des lettres de l'alphabet (minuscules ou majuscules), des chiffres de 0 à 9 et des caractères spéciaux `_` et `*`. On remarquera que cela fait un total de **64** caractères possibles.

Q23– Justifier rapidement qu'il est préférable d'utiliser un type entier non signé du langage C (`uint8_t, uint32_t, uint64_t`) comme type de retour de la fonction de hachage.

La fonction de hachage risque de provoquer un dépassement de capacité, sur un type signé c'est un comportement indéfini en C. Il est donc préférable d'utiliser un type non signé sur lequel les dépassements de capacité sont calculés modulo le plus petit entier non représentable sur le nombre de bits du type. Par exemple sur le type `uint8_t` c'est modulo 256

Q24– On suppose que les chaines de caractères ont une longueur fixe de huit caractères. Montrer que si la valeur de hachage est stockée sous la forme du type `uint32_t` du langage C, alors il est certain que deux valeurs distinctes entrent en collision.

Le nombre possible de chaines de caractères de longueur 8 avec 64 possibilités pour chaque caractères est $64^8 = 2^{48}$ cela dépasse 2^{32}, le nombre d'entiers représentable sur un `uint32_t` donc par le principe des tiroirs il est certains que deux chaines distinctes entrent en collision

Q25– Toujours en supposant des chaines de longueur fixe de huit caractères, montrer que si la valeur de hachage est stockée sous la forme du type `uint64_t` du langage C, alors on peut trouver une fonction de hachage ne provoquant aucune collision.

Comme $2^{64} > 2^{48}$, on peut trouver une fonction de hachage donnant une valeur distincte pour chaque chaine, on peut par exemple numéroter dans l'ordre lexicographique les chaines et leur attribuer comme valeur de hachage leur numéro.

Dans la suite de l'exercice, les chaines de caractères à hacher peuvent être de n'importe quelle longueur et on numérote les 64 caractères possibles de la façon suivante :

— les lettres majuscules portent les numéros 1 à 26,

— les lettres minuscules 27 à 52,

— les chiffres portent les numéros 53 à 62

— la caractère `_` a le numéro 63 et `*` le 64.

Et on propose la fonction de hachage suivant pour une chaine s de longueur n constituée des caractères $c_0, \dots, c_{n-1}$:

$$h(s) = \sum_{i=0}^{n-1} N(c_i) \times 32^i$$

où $N(c_i)$ est le numéro du caractère c_i.

On suppose déjà écrite une fonction de signature `int num(char c)` qui renvoie le numéro attribué à un des 64 caractères possibles, par exemple `num('A')` renvoie 1, `num('a')` renvoie 27.

Q26– Expliquer pourquoi l'implémentation suivante de la fonction de hachage est de complexité quadratique en la longueur de la chaine `s` et proposer une correction afin de rendre cette complexité linéaire.

```
1 uint64_t hash(char *s)
2 {
3     uint64_t h = 0;
4     for (int i = 0; i < strlen(s); i++)
5     {
6         h = h * 32 + num(s[strlen(s) - 1 - i]);
7     }
8     return h;
9 }
```

A la ligne 4, chaque passage dans la boucle fait appel à la fonction `strlen` qui est de complexité linéaire en la longueur de la chaine car elle doit parcourir la chaine afin de déterminer la position du caractère sentinelle `\0`. Ce qui donne une complexité quadratique, pour obtenir une complexité linéaire il suffit de calculer la longueur de la chaine une seule fois avant d'entrer dans la boucle.

Q27– Déterminer deux chaines de longueur 2 qui entrent en collision.

On considère deux chaines composées de deux caractères $s = xy$ et $s' = x'y'$ entrant en collision, alors $x + 32y = x' + 32y'$. C'est à dire $x - x' = 32(y' - y)$. C'est à dire que $32|(x - x')$. Comme de plus $x \in [\![1; 64]\!]$ et $x' \in [\![1; 64]\!]$, cela signifie que $x - x' = 32$ et donc $y' = y + 1$, on peut choisir n'importe quelle valeur respectant ces conditions par exemple : $x = 52$, $x' = 20$, $y = 1$ et $y' = 2$ c'est à dire les chaines `'zA'` et 'TB'

Q28– Montrer qu'il est possible de construire des chaines de longueurs arbitraires entrant en collision.

Supposons que s et s' entrent en collision ($s \neq s'$) et soient xy, $x'y'$ deux chaines de longueurs deux entrant en collision tel que déterminé à la question précédente alors en notant sxy la chaine obtenue en concaténant s et xy :

$$h(sxy) = \sum_{i=0}^{n-1} N(c_i) \times 32^i$$

$$h(sxy) = \sum_{i=0}^{n-3} N(c_i) \times 32^i + 32^{n-2}(x + 32y)$$

$h(sxy) = h(s) + 32^{n-2}h(xy)$

$h(sxy) = h(s') + 32^{n-2}h(x'y')$ car s et s' sont en collision et xy et $x'y'$ aussi. Donc $h(sxy) = h(s'x'y')$, c'est à dire que si deux chaines sont en collision alors on peut obtenir une nouvelle collision en concaténant à ces chaines deux chaines de longueur deux entrant en collision.

❑ **Exercice 5** : *Détection de cycle*

Le langage d'implémentation dans cet exercice est OCaml.

Etant donné un entier $N \in \mathbb{N}^*$, on considère la fonction :

$$\begin{array}{rccc} f_N : & \mathbb{Z}^2 & \mapsto & \mathbb{Z}^2 \\ & (x,y) & \longrightarrow & ((x+2y^2) \mod N, (3x^2+y) \mod N) \end{array}$$

Pour un couple $(x_0, y_0) \in \mathbb{Z}^2$, on définit la suite $(u_n)_{n\in\mathbb{N}}$ par $u_0 = (x_0, y_0)$ et $u_{n+1} = f_N(u_n)$.

Q29– Vérifier sur l'exemple $N = 10$ et $(x_0, y_0) = (1, 8)$ en calculant manuellement les premiers termes que la suite contient un cycle. Quelle est la longueur de ce cycle ?

On obtient successivement les couples `(1,8)` `(9,1)` `(1,4)` `(3,7)` `(1,4)` donc un cycle de longueur 2 `(1,4)` `(3,7)` après deux itérations

Q30– Justifier que pour toute valeur de N fixée et tout couple initial (x_0, y_0), la suite (u_n) contient nécessairement un cycle. On pourra remarquer que l'ensemble des couples possibles est fini.

L'ensemble $\mathbb{Z}^2/N\mathbb{Z}^2$ est fini puisqu'il contient exactement N^2 éléments. Comme la suite (u_n) reste dans cet ensemble fini et a une infinité de termes, par le principe des tiroirs, au moins deux éléments de la suite doivent être identiques. Par conséquent, la suite est périodique à partir d'un certain rang.

Q31– Ecrire la fonction f_N en OCaml avec la signature : `f : int -> int -> int-> int * int` en donnant les paramètres dans cet ordre : `x`, `y` et `N`.

```
let f x y n =
  let z = (x+2*y*y) mod n in
  let t = (3*x*x+y) mod n in
  (z,t);;
```

Q32– Ecrire une fonction `appartient : 'a -> 'a list -> bool` qui teste si un élément appartient à une liste. Donner la complexité de cette fonction en fonction de la taille de la liste.

```
let rec appartient elt lst =
  match lst with
  | [] -> false
  | h::t -> elt=h || appartient elt t;;
```

La complexité est linéaire.

Q33– Ecrire une fonction `cycle_naif : int -> int -> int -> int * int` qui prend en argument (dans cet ordre) x_0, y_0 et N et détecte le cycle en maintenant à jour une liste des couples déjà rencontrés, et renvoie le premier couple qui apparaît deux fois.

```
let cycle x y n =
  let rec aux xn yn vus =
    if appartient (xn,yn) vus then (xn,yn) else
      let xn1, yn1 = f xn yn n in
      aux xn1 yn1 ((xn,yn)::vus)
    in
  aux x y [];;
```

Q34– Quelle est la complexité dans le pire cas de la fonction `cycle_naif` en fonction du nombre de termes à parcourir avant de trouver le premier couple apparaissant deux fois ?

La complexité est quadratique.

On propose maintenant d'utiliser une table de hachage afin de stocker les termes déjà rencontrés, les clés sont les couples rencontrés et la valeur associée est l'indice dans la suite de ce couple. Par exemple

si les trois premiers termes de la suite sont $(1,5),(4,3),(7,7)$ alors la table de hachage doit contenir les clés $(1,5),(4,3)$, et $(7,7)$ associées respectivement aux valeurs $0,1$ et 2 (c'est à dire aux indices auxquels sont apparus ces couples). On rappelle ci-dessous les fonctions de manipulation d'une table de hachage en OCaml :

— `Hashtbl.create : int -> ('a,'b) Hashtbl.t` renvoie une table de hachage dont on donne la taille initiale.

— `Hashtbl.mem ('a, 'b) Hashtbl.t -> 'a -> bool` renvoie `true` si la clé donnée en argument apparait dans la table de hachage.

— `Hashtbl.add ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` pour ajouter un couple (clé, valeur) à une table de hachage.

— `Hashtbl.find ('a, 'b) Hashtbl.t -> 'a -> 'b` renvoie la valeur associée à une clé.

Q35– Justifier rapidement que l'utilisation d'une table de hachage pour stocker les couples déjà rencontrés permet d'obtenir une complexité meilleure que celle de la fonction `cycle_naif`.

Le test d'appartenance à la table de hachage est en temps constant.

Q36– En utilisant une table de hachage de OCaml, écrire une version de la fonction de recherche de cycle. Cette fonction devra renvoyé le premier élément rencontré à deux reprises *ainsi que la longueur du cycle*.

```
let cycle_ht x y n =
  let vus = Hashtbl.create 1000 in
  let rec aux xn yn i =
    if Hashtbl.mem vus (xn,yn) then (xn, yn, i- Hashtbl.find vus (xn,yn))  else
      let xn1, yn1 = f xn yn n in
      Hashtbl.add vus (xn,yn) i;
      aux xn1 yn1 (i+1)
    in
  aux x y 0 ;;
```