Devoir surveillé d'informatique

Consignes

- Les programmes demandés doivent être écrits en C ou en OCaml suivant l'exercice. Dans le cas du C, on suppose que les librairies standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.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 : *Question de cours*

1. Rappeler 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.

2. Rappeler 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 :
$$\begin{cases} h(A) = -1 & \text{si } A \text{ est vide} \\ h(A) = 1 + \max(h(g), h(d)) & \text{si } A = (g, a, d) \end{cases}$$

3. Soit a un arbre binaire de taille n et de hauteur h. Prouver que $h + 1 \leqslant n \leqslant 2^{h+1} - 1$

La preuve s'effectue par récurrence forte sur la taille $n(A)$ de l'arbre binaire A, on note $P(n)$, la propriété, « *pour tout arbre binaire de taille n et de hauteur h, on a, $h + 1 \leqslant n \leqslant 2^{h+1} - 1$* »

— pour $n = 0$, A est l'arbre vide et on a $h(A) = -1$, donc $P(0)$ est vraie.

— Soit $n \in \mathbb{N}$, tel que $P(k)$ est vraie pour tout $k \in [\![0; n]\!]$, on considère un arbre A ayant $n + 1$ noeuds. A n'est pas vide, on note $A = (g, v, d)$. Et par définition :

— $n(A) = n + 1 = 1 + n(g) + h(d)$, g et d ont au plus n noeuds on peut donc utiliser l'HR et $n(g) \geqslant h(g) + 1$, $n(d) \geqslant h(d) + 1$. Donc, $n(A) \geqslant 3 + h(g) + h(d)$. Or, par définition de la hauteur on a $h(d) \geqslant -1$ donc $n(A) \geqslant h(g) + 2$ et de la même façon $n(A) \geqslant h(d) + 2$ c'est à dire $n(A) \geqslant 2 + \max(h(g), h(d))$ et donc $n(A) \geqslant 1 + h(A)$ ce qui prouve la première inégalité.

— En appliquant l'HR sur la deuxième partie de l'inégalité, on a :
$n(A) = 1 + n(g) + n(d) \leqslant 1 + 2^{h(g)+1} - 1 + 2^{h(d)+1} - 1$, or $1 + h(g) \leqslant h(A)$ et $1 + h(d) \leqslant h(A)$ donc,
$n(A) \leqslant 2 \times 2^{h(A)} - 1$ c'est à dire $n(A) \leqslant 2^{h(A)+1} - 1$ ce qui prouve la deuxième inégalité.

La propriété $P(n)$ est donv vérifiée pour tout $n \in \mathbb{N}$.

❑ **Exercice 2** : *Nombres de Hamming*

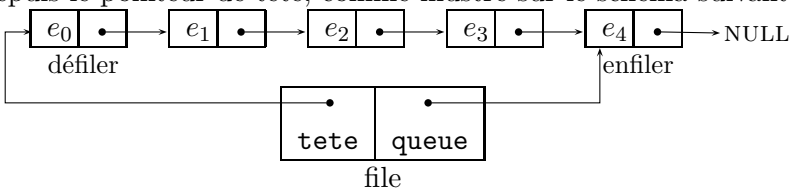
Les programmes de cet exercice doivent être écrits en langage C.

■ **Partie I** : Implémentation d'une file en C

Dans cette partie, on implémente une file en C à l'aide d'une liste chainée en conservant un pointeur sur le premier élément de la liste (à partir duquel on défile) et un autre pointeur sur le dernier élément de la liste (à partir duquel on enfile). Pour cela on définit un type structuré `maillon` contenant une valeur et un pointeur sur le maillon suivant, et une file est alors implémentée comme deux pointeurs (un vers le début et l'autre vers la fin de la liste chainée.) :

```c
/* Implementation d'une file avec accès au premier et au dernier*/
struct maillon
{
    int valeur;
    struct maillon *suivant;
};
typedef struct maillon maillon;
struct file
{
    maillon *tete;
    maillon *queue;
};
typedef struct file file;
```

⚠ On notera bien que dans cette implémentation, l'ajout d'un élément s'effectue sur le pointeur de queue et le retrait d'un élément depuis le pointeur de tête, comme illustré sur le schéma suivant :



La création d'une file vide s'effectue en initialisant les deux pointeurs à `NULL` :

```c
file cree_file()
{
    file f = {.tete = NULL, .queue = NULL};
    return f;
}
```

1. Dans l'implémentation suivante de defiler, ajouter une instruction `assert` permettant de vérifier que la file n'est pas vide avant de défiler. Sans cette instruction, quel sera le comportement de `defiler` sur une file vide ?

```c
int defiler(file *f)
{
    int v = f->tete->valeur;
    maillon *old = f->tete;
    f->tete = f->tete->suivant;
    return v;
}
```

On insère `assert f->tete != NULL;` entre les lignes 2 et 3. Sans cette instruction, dans le cas d'une file vide, l'instruction de la ligne 3 : `int v = f->tete->valeur;` est le déréférencement d'un pointeur NULL ce qui est un comportement indéfini en C.

2. Un problème subsiste dans l'implémentation proposée pour la fonction `defiler`, lequel et comment le corriger ?

   La maillon qui contenait la valeur défilée n'est pas libéré, cette fonction provoque donc une fuite mémoire car on n'a plus de référence sur ce maillon et donc il n'est plus libérable. On peut corriger le problème en ajoutant une instruction `free(old);`. D'autre part la fonction `defile` ne traite pas le cas où on défile le dernier élément, dans ce cas on devrait aussi mettre à jour la queue de la liste.

3. Ecrire la fonction `prochain` qui renvoie le prochain élément qui sera défilé mais *sans le retirer de la file*. On traitera par un `assert` le cas de la file vide.

```
int prochain(file f)
{
    assert (f.tete != NULL);
    return (f.tete)->valeur;
}
```

4. Ecrire la fonction permettant d'enfiler un élement de prototype `void enfiler(file *f, int val)`

```
void enfiler(file *f, int val)
{
    maillon *nm = malloc(sizeof(maillon));
    nm->valeur = val;
    nm->suivant = NULL;
    if (f->queue == NULL)
    {
        f->queue = nm;
    }
    else
    {
        f->queue->suivant = nm;
        f->queue = nm;
    }
    if (f->tete == NULL)
    {
        f->tete = nm;
    }
}
```

■ **Partie II** : Nombre de Hamming

On appelle *nombre de Hamming* un entier naturel strictement positif n'ayant pas d'autres diviseurs premiers que 2, 3 et 5. Les premiers nombres de Hamming, par ordre croissant, sont donc 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, ...

1. Donner (sans justification) les six nombres de Hamming suivants.

   15, 16, 18, 20, 24, 25

2. Ecrire une fonction `est_hamming` de signature `bool est_hamming(int n)` permettant de tester si l'entier passé en paramètre est un nombre de Hamming.

```
bool est_hamming(int n)
{
    if (n==1) {return true;}
    while (n%2==0)
    { n=n/2;}
    while (n%3==0)
    { n=n/3;}
    while (n%5==0)
    { n=n/5;}
    return (n==1);
}
```

3. Donner, en la justifiant, la complexité de votre fonction `est_hamming`.

> La fonction `est_hamming` effectue au plus $\log_k n$ division pour $k = 2, 3, 5$, elle est donc en $O(\log(n))$.

Pour générer les n premiers nombres de Hamming de *façon efficace* et par ordre croissant, on propose l'algorithme suivant :

— On initialise trois files `f2`, `f3` et `f5` contenant au départ un 1 ;

— A chaque étape :

- On note `x = min {x2, x3, x5}` où `x2` (resp. `x3`, resp. `x5`) est le prochain élément à être défilé de `f2` (resp. `f3`, resp. `f5`)
- `x` est le prochain nombre de Hamming, on l'enlève de la file qui le contient (il est forcément le prochain à être défilé)
- On enfile `2x` à `f2`, `3x` à `f3` et `5x` à `f5`

4. Simuler l'exécution de cet algorithme pour `n=8` en donnant l'évolution des 3 files.

f2	f3	f5	[]
1	1	1	[]
2	3 ; 1	5 ; 1	[1]
2; 2	3 ; 3	5; 5 ; 1	[1; 1]
2; 2; 2	3 ; 3 ; 3	5; 5; 5	[1; 1; 1]
4; 2; 2	6; 3 ; 3 ; 3	10; 5; 5; 5	[1; 1; 1; 2]
4; 4; 2	6; 6; 3 ; 3 ; 3	10; 10; 5; 5; 5	[1; 1; 1; 2; 2]
4; 4; 4	6; 6; 6; 3 ; 3 ; 3	10; 10; 10; 5; 5; 5	[1; 1; 1; 2; 2; 2]
6; 4; 4; 4	9; 6; 6; 6; 3 ; 3	15; 10; 10; 10; 5; 5; 5	[1; 1; 1; 2; 2; 2; 3]
6; 6; 4; 4; 4	9; 9; 6; 6; 6; 3 ; 3	15; 15; 10; 10; 10; 5; 5; 5	[1; 1; 1; 2; 2; 2; 3; 3]

5. Implémenter cet algorithme en écrivant une fonction `hamming` de signature `int * hamming(int n)` qui renvoie le tableau des `n` premiers nombres de Hamming. On supposera déjà écrite la fonction `min3` qui renvoie le plus petit des trois entiers passés en paramètres. Votre fonction devra utiliser l'implémentation des files de la partie 1 et ses fonctions `enfiler` et `defiler`

> Dans la correction de la question suivante on enlève `xm` (le minimum) seulement de l'une des files

6. Un même nombre peut être présent dans plusieurs des files. Ecrire une nouvelle version de la fonction `hamming` ne présentant plus ce problème.

```c
int * hamming(int n)
{
    file f2 = cree_file();
    file f3 = cree_file();
    file f5 = cree_file();
    int x2, x3, x5, xm;
    enfiler(&f2, 1);
    enfiler(&f3, 1);
    enfiler(&f5, 1);
    int *ham = malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++)
    {
        x2 = prochain(f2);
        x3 = prochain(f3);
        x5 = prochain(f5);
        if (x2 == min3(x2, x3, x5))
        {
            xm = defiler(&f2);
        }
        if (x3 == min3(x2, x3, x5))
            {
                xm = defiler(&f3);
            }
        if (x5 == min3(x2, x3, x5))
        {
            xm = defiler(&f5);
        }
        enfiler(&f2, 2 * xm);
        enfiler(&f3, 3 * xm);
        enfiler(&f5, 5 * xm);
        ham[i] = xm;
    }
    return ham;
}
```

❑ **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 prénom sont des chaines de caractères, les autres sont des entiers.

1. 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.

2. Expliquer ce qu'affiche la requête suivante :

```sql
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.

3. Ecrire les requêtes permettant d'afficher les informations suivante :
   a) La liste des titres des articles parus en 2022 listé par ordre alphabétique.

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

   b) Les noms des revues listé par ordre alphabétique, sans répetition.

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

   c) Les noms et prénoms des auteurs qui ont publié dans la revue "Nature" en 2000.

```sql
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
```

   d) 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
```

   e) 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"
```

   f) 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** : *Piles et notation polonaise inverse*

Les programmes de cet exercice doivent être écrits en OCaml.

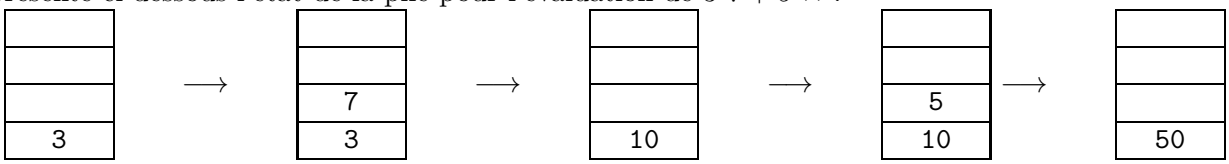
La *notation polonaise inverse* (NPI) est une méthode d'écriture des expressions mathématiques qui n'utilise pas de parenthèses et qui de plus se calcule sans règles de priorité. Prenons un exemple, l'expression $(3 + 7) \times 5$, s'écrit en notation polonaise inversée : $3\ 7 + 5\ \times$. C'est à dire qu'on donne *d'abord les deux opérandes puis l'opération*. Le but de l'exercice est d'écrire une fonction OCaml évaluant une expression en NPI passée en paramètre à l'aide d'une pile, on utilisera la méthode suivante :

① Parcourir l'expression de gauche à droite

② Si on rencontre un nombre l'empiler

③ Si on rencontre une opération effectuer cette opération entre les deux valeurs situées au dessus de la pile et empiler le résultat

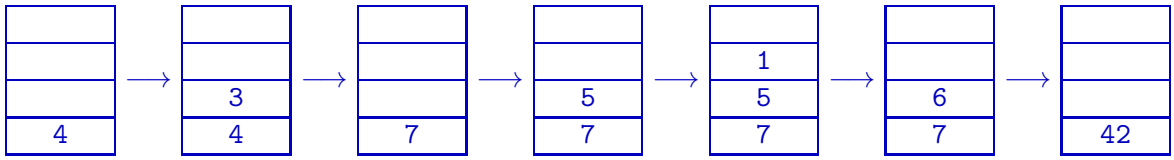
On représente ci-dessous l'état de la pile pour l'évaluation de $3\ 7 + 5\ \times$:



1. Ecrire l'expression $(4 + 3) \times (5 + 1)$ en NPI.

   $4\ 3 + 5\ 1 + \times$

2. Donner les étapes de l'évaluation de cette expresssion en faisant figurer à chaque étape le contenu de la pile.



Afin de représenter les éléments (c'est à dire soit des `int` soit une opération) d'une expression en NPI, on défini le type union suivant en OCaml :

```
type elt = Add | Sub | Mult | Div | Nombre of int
```

Ainsi, l'expression $3\ 7 + 5\ \times$ est représentée par la liste `[Nombre 3; Nombre 7; Add; Nombre 5; Mult]`.

D'autre part, on utilise le module `Stack` de OCaml afin de disposer d'une structure de pile *mutable*. On rappelle ci-dessous les fonctions principales de ce module :

— `Stack.create` de signature `() -> 'a t` qui crée une pile vide d'éléments de type `'a`. Par exemple `let mapile = Stack.create ()`

— `Stack.push` de signature `'a 'a t -> ()` qui empile un élément. Par exemple `Stack.push 5 mapile` empile l'entier 5 sur `mapile` (le type option 'a est alors le type `int`).

— `Stack.pop` de signature `'a t -> 'a` qui renvoie l'élément situé au sommet de la pile en le dépilant.

3. Ecrire une fonction `evalue` de signature `elt list -> int` et qui renvoie l'évaluation d'une expression en `npi` donné sous la forme d'une liste de type `elt`. Par exemple, `evalue [Nombre 3; Nombre 7; Sub; Nombre 5; Mult]` doit renvoyer 50.

```
let evalue npi =
  let rec aux npi pile =
    match npi with
    | [] -> Stack.pop pile
    | h::t -> (match h with
      | Add -> Stack.push (Stack.pop pile + Stack.pop pile) pile
      | Sub -> Stack.push (Stack.pop pile -Stack.pop pile) pile
      | Mult -> Stack.push (Stack.pop pile * Stack.pop pile) pile
      | Div -> Stack.push (Stack.pop pile / Stack.pop pile) pile
      | Nombre x -> Stack.push x pile );
     aux t pile
  in
    let temp = Stack.create () in
    aux npi temp;;
```

❑ **Exercice 5** : *Représentations classiques d'ensembles* 🎓 d'après CCSE 2021 - MP (Partie 2)

Les programmes de cet exercice doivent être écrits en OCaml.

On s'intéresse dans cet exercice à des structures de données représentant des ensembles d'entiers naturels. On implémente dans cet exercice des ensembles par des structures connues et, on notera $|E|$ le cardinal d'un ensemble E.

■ **Partie I** : Avec une liste d'entiers triés

Dans cette partie uniquement, on implémente un ensemble d'entiers positifs par la liste des ses éléments rangés *dans l'ordre croissant.*

1. Ecrire une fonction `succ_list` de signature `int list -> int -> int` prenant en arguments une liste d'entiers *distincts* dans l'ordre croissant et un entier x et renvoyant le successeur de x dans la liste, c'est à dire le plus petit entier strictement supérieur à x de la liste (-1 si cela n'existe pas.).

```
let rec succ_list entiers x =
  match entiers with
  | [] -> -1
  | h::t -> if h>x then h else succ_list t x
```

2. Donner la complexité de cette fonction dans le pire des cas.

Les appels récursif sont en temps constant et la taille de la liste diminue de 1 à chaque appel donc la complexité est linéaire en fonction de la taille de la liste.

■ **Partie II** : Avec un tableau trié

Soit N un entier naturel strictement positif, fixé pour toute cette partie. On choisit de représenter un ensemble d'entiers E de cardinal $n \leq N$ par un tableau de taille $N+1$ dont la case d'indice 0 indique le nombre n d'éléments de E et les cases d'indices 1 à n contiennent les éléments de E *rangés dans l'ordre croissant*, les autres cases étant non significatives. Par exemple, le tableau `[| 3; 2; 5; 7; 9; 1; 14 |]` représente l'ensemble $\{2, 5, 7\}$.

1. Pour une telle implémentation d'un ensemble E, décrire brièvement des méthodes permettant de réaliser chacune des opérations ci-dessous (on ne demande pas d'écrire des programmes) et donner leurs complexités dans le pire cas :
   — déterminer le maximum de E,
   — tester l'appartenance d'un élément x à E
   — ajouter un élément x dans E (on suppose que $x \notin E$ et que la taille du tableau est suffisante)

On note `tab` le tableau représentant l'ensemble d'entiers
- Pour déterminer le maximum de E, il suffit de renvoyer `tab[tab[0]]` car les éléments sont dans l'ordre croissant et leur indice vont de 1 à `tab[0]`. C'est donc une opération en temps constant.
- On doit parcourir le tableau entre les éléments d'indice 1 et `tab[0]` (complexité linéaire), ou alors (puisque le tableau est trié) effectuer une recherche dichotomique (complexité logarithmique).
- On incrémente `tab[0]` et on place l'élément `x` à l'indice `tab[0]`, ensuite, pour que le tableau reste trié, on peut par exemple échanger cet élément avec son voisin tant qu'il lui est inférieur (et qu'on a pas atteint l'indice 1). L'insertion est alors en complexité linéaire.

2. Par une méthode dichotomique, écrire une fonction `succ_vect` de signature `int array -> int -> int` prenant en arguments un tableau `t` codant un ensemble E comme ci-dessus et un entier x et renvoyant le successeur de x dans E (-1 si cela n'existe pas.)

```
let succ_vect entiers x =
  if x >= entiers.(entiers.(0)) then -1 else (
  let imin = ref 1 in
  let imax = ref entiers.(0) in
  let found = ref false in
  let imid = ref 0 in
  while (!imax - !imin >=0 && not !found) do
    imid := (!imax + !imin)/2;
    if (entiers.(!imid)=x) then found:=true else
      if (entiers.(!imid)<x) then imin := !imid + 1  else imax := !imid-1
    done;
    if (!found) then entiers.(!imid+1) else entiers.(!imin))
  ;;
```

3. Calculer la complexité dans le pire cas de la fonction `succ_vect` en fonction de n.

> A chaque passage dans la boucle `while` la taille de l'intervalle $[\![imin; imax]\!]$ est divisée par 2. Cet intervalle étant de taille $|E|$, il faut au plus $\log(|E|)$ division avant de quitter la boucle, la fonction est donc de complexité logarithmique en la taille de l'ensemble.

4. Ecrire une fonction `union_vect` de signature `int array -> int array -> int array` prenant en arguments deux tableaux `t_1` et `t_2`, de taille N, codant deux ensembles E_1 et E_2 et renvoyant le tableau correspondant à $E_1 \cup E_2$. On supposera que $|E_1 \cup E_2| \leq N$.

```
let union t1 t2 =
  (* size est la variable N de l'énoncé*)
  let t = Array.make size 0 in
  let size1 = t1.(0) in
  let size2 = t2.(0) in
  let i1 = ref 1 in
  let i2 = ref 1 in
  let i = ref 1 in
  while (!i1<=size1 || !i2<=size2) do
    Printf.printf "i1 = %d, i2 = %d, i = %d\n" !i1 !i2 !i;
    (* prendre dans t1 si t2 est vide OU si il reste des éléments dans les 2 et le
    ↪  plus petit est dans t1*)
    if (!i2>size2) || (!i1 <= size1 && !i2 <=size2 && t1.(!i1) <= t2.(!i2)) then
      (
        t.(!i) <- t1.(!i1);
        (* en cas d'égalité on avance aussi dans t2*)
        if (!i2<=size2 && t1.(!i1) = t2.(!i2)) then (i2 := !i2 +1);
        i1 := !i1 +1;
      ) else
    (* sinon prendre dans t2 *)
      (
        t.(!i) <- t2.(!i2);
        (* en cas d'égalité on avance aussi dans t1*)
        if (!i1<=size1 && t1.(!i1) = t2.(!i2)) then (i1 := !i1 +1);
        i2 := !i2 +1;
      );
      i := !i + 1;
      done;
      t.(0) <- !i-1;
      t;;
```

■ **Partie III** : Avec une table de hachage

Soit K un entier naturel strictement positif. On choisit de représenter un ensemble d'entiers E de cardinal n par une table de hachage de taille K avec résolution des collisions par chainage. La fonction de hachage est $h(i) = i \mod K$.

1. Dans le cas où $K = 10$, représenter la table de hachage qui correspond à l'ensemble $\{2, 5, 7, 15\}$.

2. A quelle condition, portant sur K et sur n, la fonction h génère-t-elle forcément des collisions ?

Il y a K alveoles donc dès que $n > K$, on a forcément des collisions (principe des tiroirs)

3. Décrire brièvement (on ne demande pas d'écrire un programme) une fonction permettant de renvoyer le maximum d'un ensemble E représenté par une table de hachage. Donner sa complexité.

On parcourt les listes chainées contenues dans chaque alvéole, en mettant à jour une variable contenant le maximum. On a donc une complexité en $O(|E|)$.

4. Peut-on améliorer la complexité de la fonction précédente si on suppose que les listes chainées contenues dans chacune des alvéoles de la table de hachage sont maintenues triées par ordre décroissant ?

Dans ce cas, on doit simplement regarder la tête de chaque liste, on a donc une complexité en $O(K)$.