Devoir Surveillé d'Informatique n°1 : Programmation en OCaml, Complexité

Durée : 4 heures

20 Octobre 2023

Vous pouvez réutiliser des résultats des questions précédentes pour une démonstration ou un code, même si la question n'a pas été traitée.

Vous pouvez introduire des fonctions, variables, types ou notations supplémentaires pour produire vos réponses.

Si, au cours de l'épreuve, vous repérez ce qui vous semble être une erreur d'énoncé, signalez-le sur votre copie et poursuivez votre composition en expliquant les raisons des initiatives que vous seriez amené·e à prendre.

Pour la pagination, vous numéroterez toutes les pages de votre composition en faisant figurer le numéro de la page actuelle à partir de 1, ainsi que le total de pages de votre composition. Vous ferez la pagination durant le temps de l'épreuve, et non après la fin de l'épreuve.

Votre nom devra figurer sur chacune de vos copies qui seront rendue imbriquées les unes dans les autres.

Les questions difficiles sont indiquées par une ou plusieurs étoiles (⋆). Cette difficultés est principalement indicative.

Prenez bien soin de lire les consignes et les indications fournies dans le sujet.

Bon courage, et bonne composition.

On rappelle les syntaxes OCaml suivantes :

- On peut accéder au premier élément d'un couple avec la fonction fst de signature 'a * 'b -> 'a, et au second élément d'un couple avec la fonction snd de signature 'a * 'b -> 'b.
- La fonction sqrt de signature float -> float renvoie la racine carrée d'un nombre passé en argument.
- La syntaxe a mod b permet de calculer le reste de la division euclidienne de a par b.
- La syntaxe p::q permet de construire la liste dont p est la tête et q est la queue.
- La syntaxe ref a permet de créer une référence de valeur a.
- La syntaxe r:=a permet d'affecter à la la référence r la valeur a.
- La syntaxe !r permet d'obtenir la valeur contenue dans la référence r.
- La fonction failwith permet de lever une erreur et d'interrompre le calcul en cours. Cette fonction prend en argument le massage d'erreur à afficher sous la forme d'une chaîne de caractère.
- L'expression Array.make n x crée un nouveau tableau de taille n dont tous les éléments sont égaux à x.
- La fonction Array.length de signature 'a array -> int renvoie la taille d'un tableau passé en argument.
- On peut accéder à la case i dans un tableau t avec la syntaxe t.(i).
- On peut assigner la valeur x à la case i dans un tableau t avec la syntaxe t.(i)<-x.
- L'expression Array.make_matrix n m x crée une nouvelle matrice de taille $n \times m$ dont tous les éléments sont égaux à x.
- La fonction String.length de signature string -> int renvoie la taille d'une chaîne de caractères passée en argument.
- On peut accéder au charactère à la case i d'ans une chaîne de caractère s avec la syntaxe s.[i].
- On peut convertir un entier vers une chaîne de caractère avec la fonction string_of_int de signature int -> string.
- On peut convertir une chaîne de caractère qui représente l'écriture décimale d'un entier vers cet entier avec la fonction int_of_string de signature string -> int.
- Les caractères utilisent les délimiteurs ', et les chaînes de caractère utilisent les délimiteurs ". Ainsi, "c" est une chaîne de caractères, mais 'c' est un caractère.

On rappelle les formules suivantes :

- Binôme de Newton :

$$\forall x, y \in \mathbb{R}^2, \forall n \in \mathbb{N}, (a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

-

$$\forall n, \forall 0 \leqslant k \leqslant n,\ n\binom{n-1}{k-1} = k\binom{n}{k}$$

Exercice 1. Programmation

1. On se donne la suite u_n définie par récurrence suivante :

$$u_0 = 0$$
$$u_1 = 0$$
$$u_2 = 1$$
$$\forall n \in \mathbb{N}^*, u_{n+3} = u_{n+2} + u_{n+1} + u_n$$

   Proposer une fonction u de signature $int->int$ qui calcule u_n où n est son entrée. On prendra soin d'avoir une fonction dont la complexité temporelle est en $O(n)$.

2. Proposer une fonction inverser de signature 'a list -> 'a list qui inverse une liste.

```
inverser [1; 2; 4; 3] (* [3; 4; 2; 1] *)
```

   On veillera à avoir une complexité en $O(n)$ où n est la longueur de la liste.

3. Sans utiliser l'opérateur @, proposer une fonction concatener de signature 'a list -> 'a list -> 'a list qui concatène deux listes en entrée.

```
concatener [1;2;3] [4; 5] (* [1; 2; 3; 4; 5] *)
```

4. Proposer une fonction echanger de signature 'a ref -> 'a ref -> unit qui inverse les valeurs contenues dans deux références passées en argument.
5. Proposer une fonction de signature 'a array -> 'a list qui construit une liste avec les même éléments et dans le même ordre que le tableau en entrée.
6. Proposer une fonction table_multiplication de signature int -> int -> int array array de sorte à ce que table_multiplication n m renvoie une matrice de taille $n \times m$ dont les éléments $m.(i).(j)$ sont égaux à $i \times j$.
7. Proposer une fonction somme de signature int array list -> int qui fait la somme de tous les éléments dans une liste de tableaux d'entiers passée en argument.

```
somme [[|1;2|];[|3;4;5|];[||]] (* 15 *)
```

Correction :

1. On utilise une fonction auxiliaire aux à quatre arguments, de sorte à ce que, pour tout k, lors de l'appel aux a b c k, on ait $a = u_k$, $b = u_{k+1}$ et $c = u_{k+2}$.

   Cette propriété est vraie lors du premier appel aux 0 0 1 0, et on remarque que cette propriété reste vraie si elle l'était lors de l'appel récursif aux b c (a+b+c) (k+1).

```
let u n =
    let rec aux a b c k =
        if k = n then a
                 else aux b c (a+b+c) (k+1)
    in aux 0 0 1 0
```

   Notre fonction s'arrête lorsque k atteint n, auquel cas, on renvoie a qui vaut u_k par récurrence. k augmente à chaque itération de aux et notre fonction termine donc et est correcte. Sa complexité est linéaire en n car il y a au plus n appels à aux de complexité constante.

2. On utilise une fonction auxiliaire qui prend un accumulateur en argument.

```
let inverser l =
    let rec aux l acc = match l with
    |[] -> acc
    | p::q -> aux q (p::acc)
    in aux l []
```

3.
```
let rec concatener l1 l2 = match l1 with
| [] -> []
| p::q -> p::concatener q l2
```

4. On utilise une variable locale temporaire pour se souvenir de la valeur de la première référence.

```
let echanger r1 r2 = let temporaire = !r1 in
    r1 := !r2 ;
    r2 := temporaire
```

5. On utilise une fonction auxiliaire qui itère sur le tableau à l'envers en construisant la liste dans un accumulateur.

```
let convertir t =
    let n = Array.length t in
    let rec aux i acc =
        if i < 0 then acc
        else aux (i-1) (t.(i)::acc)
    in aux (n-1) []
```

```
let table_multiplication n m =
    let resultat = Array.make_matrix n m 0 in
    for i = 0 to n-1 do
        for j = 0 to m-1 do
            t.(i).(j) <- i * j
        done
    done ; resultat
```

6.

7. On utilise deux fonctions auxilaires : la première qui opère sur un tableau, et l'autre qui opére sur une liste :

```
let somme l =
    let aux1 t =
        let n = Array.length t in
        let resultat = ref 0 in
        for i = 0 to n - 1 do
            resultat := !resultat + t.(i)
        done; !resultat in
    let rec aux2 l = match l with
    | [] -> 0
    | p::q -> aux1 p + aux2 q in
    aux2 l
```

Exercice 2. Complexes

On décide de représenter un complexe par un couple de type `float * float`. Le premier élément du couple représente la partie réelle du nombre, et le second élément représente la partie imaginaire du flottant.

Ainsi, le nombre $1 + 2i$ est représenté par le couple `(1., 2.)`.

> Attention !
>
> On utilise des types flottants, il est nécessaire d'utiliser des opérateurs adaptés.

1. Quelle est la représentation avec ce type du nombre 1 ? Quelle est la représentation du nombre i ?
2. À quel nombre correspond la représentation `(2., 1.)` ?
3. Proposer une fonction `module_complexe` de signature `float * float -> float` qui renvoie le module du nombre complexe passé en argument.

```
module_complexe (3., 4.) (* 5.*)
```

On pourra utiliser la fonction `sqrt` de signature `float -> float` qui renvoie la racine carrée d'un flottant passé en argument.

4. Proposer une fonction `ajouter` de signature `float * float -> float * float -> float * float` qui fait la some de deux nombres complexes passés en argument.

```
ajouter (1., 2.) (3., 4.) (* (4., 6.) *)
```

5. Proposer une fonction `multiplier` de signature `float * float -> float * float -> float * float` qui multiplie deux nombres complexes passés en argument.

```
multiplier (1., 2.) (3., 4.) (* (-5., 10.) *)
```

6. Proposer une fonction `conjuguer` de signature `float * float -> float * float` qui renvoie le conjugué d'un nombre complexe passé en argument.

```
conjuguer (2., 3.) (* (2., -3.) *)
```

7. Proposer une fonction `diviser` de signature `float * float -> float * float -> float * float` qui fait la divison de deux nombres complexes passés en argument.

```
diviser (1., 2.) (3., 4.) (* (0.44, 0.08) *)
```

On pourra utiliser le conjugué : $\frac{a+ib}{c+id} = \frac{(a+ib)(c-id)}{(c+id)(c-id)}$. *Il n'est pas nécessaire de renvoyer une erreur spécifique en cas de division par* 0.

Correction :

1. 1 est représenté par `(1.0, 0.0)`, i est représenté par `(0.0, 1.0)`.
2. `(2.0, 1.0)` représente le nombre $2+i$.
3. On utilise la formule $|a+ib| = \sqrt{a^2+b^2}$

```
let norme x, y = sqrt (x *. x +. y *. y)
```

4.
```
let ajouter (x1,y1) (x2,y2) = (x1 +. x2, y1 +. y2)
```

5.
```
let conjugue (x1, y1) = x1, -y1
```

6. On remarque que $(a+ib)(c+id) = (ac-bd)+i(ad+bc)$, et on écrit :

```
let multiplier (x1, y1) (x2, y2) = (x1 *. x2 -. y1 *. y2), (x1 *. y2 +. x2 *. y1)
```

7. On remarque que :

$$\begin{aligned}\frac{a+ib}{c+id} &= \frac{(a+ib)(c-id)}{(c+id)(c-id)} \\ &= \frac{(ac+bd)+i(bc-da)}{c^2+d^2}\end{aligned}$$

On en déduit :

```
let diviser (x1, y1) (x2, y2) = (x1 *. x2 +. y1 *. y2)/.(c*.c + d*.d), (x1 *. y2
    +. x2 *. y1)/.(c*.c + d*.d)
```

Problème : Table de hachage

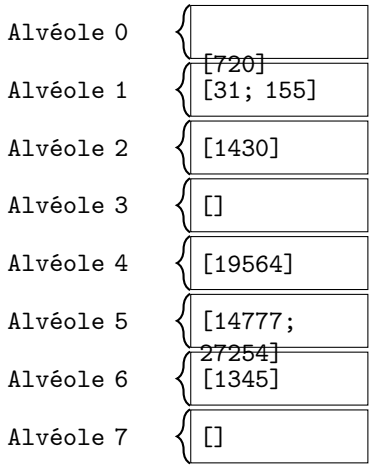
Une table de hachage est une structure de donnée qui permet de stocker des données de sorte à avoir un accès rapide aux donné stockées.

On enregistre les données sous forme d'une table disposant de plusieurs cases. L'idée principale, est de pouvoir, à partir d'un entier, savoir exactement dans quelle case on veut mettre cet élément. Une fonction de hachage, une fonction de signature `int -> int` permet de calculer à partir d'un entier la case dans laquelle il doit se trouver.

Le problème, c'est que plusieurs entiers peuvent avoir la même valeur par la fonction de hachage et cela entraîne une collision. De sorte à pouvoir mettre plusieurs éléments par case, on décide de mettre une liste dans chaque case qui contient l'ensemble des éléments de la table qui doivent être dans cette case. L'ordre des éléments dans chaque liste n'a pas d'importance.

La table sera ici représentée par un tableau de listes, qui contiennent des entiers, et qui est donc de signature `int list array`. Les éléments du tableau, des listes d'entiers, sont appelés les *alvéoles* et contiennent les éléments de la table de hachage.

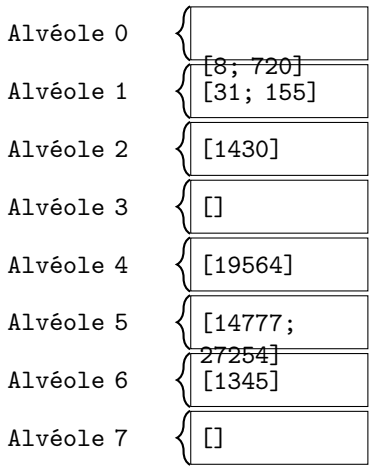
Par exemple, la table de hachage `[|[720]; [31; 155]; [1430]; []; [19564]; [14777; 27254]; [1345]; []|]` peut être représentée ainsi :



Ainsi, quand on cherche un entier n dans une table de hachage, on calcule son image par la fonction de hachage, et on regarde dans la liste à la case donnée par la fonction de hachage.

Par exemple, dans la recherche de 155, la fonction de hachage nous informe que l'élément doit être dans la case 1, et on parcours donc la liste `[31; 155]`.

De la même manière, si on veut ajouter 8, la fonction de hachage nous donne 0, et on doit donc ajouter 8 dans la table à la case 0. On obtient la table suivante après modification :



Cela correspond à la table de hachage `[|[8; 720]; [31; 155]; [1430]; []; [19564]; [14777; 27254]; [1345]; []|]`

Le problème est décomposé en 4 parties : la première s'intéresse à une fonction de hachage ; la seconde s'intéresse à la manipulation de tables de hachage ; la troisième est une étude de complexité ; et enfin la quatrième s'intéresse au problème de la modification de la taille de la table de hachage.

Pour chaque question, il est possible d'utiliser les fonctions définies dans les questions précédentes, même si celles-ci n'ont pas été traitées.

Partie I : Fonction de Hachage

La fonction de hachage est essentielle pour savoir où chercher ou ajouter les éléments dans la table.

On se donne la fonction $H(n)$ définie de la mannière suivante :
— On élève n au carré ;
— On prend les quatres chiffres médians dans l'écriture décimale de n^2 ;
— Dans le cas où n^2 a un nombre impair de chiffres, on prend les quatre chiffres médians les plus à droite dans l'écriture décimale ;
— Dans le cas où n^2 a moins de quatre chiffres, on rajoute des zéros à gauche.

Ainsi, lors du calcul de $H(451)$, on calcule $451^2 = 203401$, puis on prend les quatre chiffres médians de $20\underline{3401}$, et on obtient $H(451) = 340$.

De la même manière, on a $H(12) = 144$ et $H(124) = 5376$.

Question 1. Pour chacun des n_i suivants, donner la valeur de $H(n_i)$

1. $n_0 = 720$
2. $n_1 = 31$
3. $n_2 = 155$

Correction :

1. $720^2 = 5\underline{18400}$ donc $H(720) = 1840$.
2. $31^2 = \underline{0961}$ donc $H(31) = 961$.
3. $155^2 = 2\underline{4025}$ donc $H(155) = 4025$.

Pour calculer H, on pourrait utiliser des considérations purement arithmétiques, mais on va passer par la représentation sous forme d'une chaîne de caractères.

Question 2. Proposer une fonction `entier_de_car` de signature `char -> int` qui renvoie la valeur du caractère d'un chiffre passé en argument (sous forme d'un caractère, et non d'une chaîne de caractères).

Correction :

```
let entier_de_car c = match c with
|'0' -> 0
|'1' -> 1
|'2' -> 2
|'3' -> 3
|'4' -> 4
|'5' -> 5
|'6' -> 6
|'7' -> 7
|'8' -> 8
|'9' -> 9
| _  -> failwith "Caractere non valide"
```

Question 3. Proposer une fonction `acceder` de signature `string -> int -> int` de sorte à ce que, pour s l'écriture décimale d'un entier $n = \sum_{k=0}^{l} a_k 10^k = \overline{a_l a_{l-1} \ldots a_1 a_0}^{10}$, `acceder s k` renvoie la valeur du chiffre a_k. Si $k > l$, on renverra 0.

```
acceder "73" 0 (*3*)
```

```
acceder "73" 1 (*7*)
```

```
acceder "73" 2 (*0*)
```

Correction :

```
let acceder s k =
    let l = String.length s in
    if l<=k then 0
    else entier_de_car (s.[l - 1 - k])
```

Question 4. Pour un nombre dont le carré utilise l chiffres en écriture décimale, quel est le nombre de chiffres droite(l) à droite du carré médian qui doivent être ignorés lors du calcul de H ?

Par exemple si lors du calcul d'une valeur de hachage, on obtient un nombre à 7 chiffres $a_6a_5\underline{a_4a_3a_2a_1}a_0$, *on doit ignorer un chiffre à droite, et on obtient* droite(7) = 1.

Correction :

— Si $l < 4$: on doit prendre tous les chiffres pour le carré médian, et on a besoin d'ignorer 0 chiffres.

— Si $l \geqslant 4$: si l est pair, on doit répartir $l-4$ nombres à gauche et à droite équitablement, et donc $\frac{l-4}{2}$; si l est impair on doit en répartir $l-4$ et donc $\frac{l-4}{2}$ arrondi à l'inférieur.

On obtient :

$$\text{droite}(l) = \begin{cases} 0 & \text{si l}< 4, \\ \left\lfloor \frac{l-4}{2} \right\rfloor & \text{sinon.} \end{cases}$$

Question 5. En déduire une fonction carre_median de signature int -> int qui renvoie la valeur du carré médian d'un entier.

On pourra utiliser la fonction string_of_int dont le comportement et la signature sont donnés dans le préambule.

Correction : On convertit le carré de l'entrée en chaîne de caractère, puis on sélectionne les chiffres donnés par les deux questions précédentes.

Une fonction auxilaire parcours les 4 éléments que l'on recherche. Elle a un argument supplémentaire, coefficient qui contient le facteur multiplicatif.

```
let carre_median n =
    let carre = string_of_int (n * n) in
    let l = String.length carre in
    let k = if l<4 then 0 else (l-4)/2 in
    let rec aux i coefficient = match i with
    | 4 -> 0
    | _ -> (acceder carre (k+i)) * coefficient + (aux (i+1) (coefficient * 10))
    in
    aux 0
```

La valeur de notre fonction de hachage est entre 0 et 9999, mais le nombre d'alvéoles n'est pas nécessaire 10000. En notant N le nombre d'alvéoles, on prend donc une fonction de hachage différentes :

$$H_N(n) = H(n) \mod N$$

Où $a \mod b$ est le reste de la division euclidienne de a par b.

Question 6. Calculer les valeurs suivantes :

1. $H_8(4)$
2. $H_8(6)$
3. $H_{10}(6)$
4. $H_8(3)$

Correction :

1. $H_8(4) = H(4) \mod 8 = 16 mod 8 = 0$ Donc $H_8(4) = 0$
2. $H_8(6) = 4$
3. $H_{10}(6) = 6$
4. $H_8(3) = 1$

Question 7. Proposer une fonction `hachage` de signature `int -> int -> int` de sorte à ce que `hachage n nb_alveoles` renvoie l'alvéole associé à un nombre en entrée n à l'aide du nombre d'alvéoles `nb_alveoles`.

Correction :

```
let hachage n nb_alveoles =  (carre_median n mod) nb_alveoles
```

On utilisera cette fonction dans la suite du sujet pour calculer l'alvéole dans laquelle un entier doit être.

Partie II : Manipulation de table de hachage

On peut désormais implémenter les différentes opérations de base que l'on veut pouvoir réaliser avec la table de hachage.

Question 8. Proposer une fonction `creer` de sorte à ce que `creer taille_table` crée une table de hachage vide avec `taille_table` alvéoles.

Correction :

```
let creer taille_table = Array.make taille_table []
```

Question 9. Proposer une fonction `chercher` de signature `int list array -> int -> bool` renvoie si un entier est dans une table de hachage passée en argument.

On prendra soin d'arrêter le parcours de liste dès qu'on voit l'élément qui nous intéresse.

Correction : On utilise la taille de la table pour caculer la fonction de hachage, et on cherche avec une fonction auxiliaire dans la liste.

```
let chercher table x =
    let rec aux l = match l with
    | [] -> false
    | p::q -> p=x || aux q
    in
    let nb = Array.length table in
    aux t.(hachage x nb)
```

Question 10. Représenter la table de hachage [|[8; 720]; [31; 155]; [1430]; []; [19564]; [14777; 27254]; [1345]; []|] après l'ajout de 3 dans celle-ci.

Correction : 3 doit être ajouté à la case $H_8(3) = 1$, on obtient :
[|[8; 720]; [3; 31; 155]; [1430]; []; [19564]; [14777; 27254]; [1345]; []|]

Question 11. Proposer une fonction ajouter de signature int list array -> int -> unit qui ajoute un entier dans une table de hachage passée en argument.

L'ordre des éléments au sein d'une liste n'est pas important : il est seulement nécessaire que l'élément soit dans la bonne alvéole indiquée par la fonction de hachage.

Correction :

```
let ajouter table x =
    let nb = Array.length x in
    let hach = hachage x nb in
    table.(hach) <- x::table.(hach)
```

Question 12. Représenter la table de hachage modifiée précédemment après le retrait de 31 dans celle-ci.

Correction : 31 doit être retiré dans la case $H_8(31) = 1$, on obtient :
[|[8; 720]; [3; 155]; [1430]; []; [19564]; [14777; 27254]; [1345]; []|]

Question 13. Proposer une fonction retirer de signature int list array -> int -> unit qui retire un entier dans une table de hachage passée en argument. On pourra supposer que l'entier est dans la table, et qu'il y est une seule fois exactement.

Correction : On utilise une fonction auxiliaire qui retire la première valeur de x.

```
let retirer table x =
    let n = Array.length table in
    let hach = hachage x n in
    let rec aux l = match l with
    | [] -> failwith "Erreur : element non present dans le tableau"
    | p::q -> if p = x then q else p::aux q
    in aux table.(hach)
```

Partie III : Étude de complexité

Dans cette partie et dans cette partie seulement, on considère pour complexité comme étant le nombre d'éléments parcourus lors de la modification ou la recherche dans la table de hachage.

Ainsi, lors de la recherche de 31 dans [|[8; 720]; [175; 1344; 31; 155]; [1350; 2230; 1430]; []; [19564]; [14777; 70; 27254]; [1345]; []|] , on doit parcourir les éléments 175, 1344, et enfin 31, nous donnant une complexité de 3.

Question 14. Quelle est la complexité en terme d'éléments parcourus de la fonction ajouter ?

Correction : On ne regarde pas d'élément lors de l'ajout : on ne fait qu'ajouter l'élément lui-même en tête de liste, la complexité est 0.

Question 15. Quelle est la complexité en terme d'éléments parcourus dans le meilleur des cas de la fonction chercher en fonction du nombre n d'éléments présents dans la table de hachage ? Quelle est sa complexité asymptotique en fonction de n et N ?

Correction : Dans le meilleur des cas, on trouve l'élément immédiatement : la complexité est constante égale à 1. La complexité asymptotique est donce en $O(1)$.

Question 16. Quelle est la complexité en terme d'éléments parcourus dans le pire des cas de la fonction chercher en fonction du nombre n d'éléments présents dans la table de hachage ? Quelle est sa complexité asymptotique ?

Correction : Dans le pire des cas, tous les éléments sont dans l'alvéole concernée, et l'élément est à la dernière position dans la liste. On doit donc voir les n éléments de la liste. On obtient une complexité en $O(n)$.

Question 17. On cherche à calculer la complexité moyenne. On suppose que les éléments sont aléatoirement répartis dans les N différentes alvéoles de manière équiprobables.

On admet que la moyenne du nombre d'éléments dans l'alvéoles dans laquelle on cherche l'élément est égale à :

$$N_{moyen}(n) = \sum_{k=0}^{n} k\binom{n}{k}\left(\frac{1}{N}\right)^k\left(\frac{N-1}{N}\right)^{n-k}$$

Montrer que $N_{moyen}(n) = \frac{n}{N}$.

Correction :

$$\begin{aligned}
N_{moyen}(n) &= \sum_{k=0}^{n} k\binom{n}{k}\left(\frac{1}{N}\right)^k\left(\frac{N-1}{N}\right)^{n-k} \\
&= \sum_{k=1}^{n} k\binom{n}{k}\left(\frac{1}{N}\right)^k\left(\frac{N-1}{N}\right)^{n-k} && \text{en retirant le cas } k=0 \text{ nul.} \\
&= \sum_{k=1}^{n} n\binom{n-1}{k-1}\left(\frac{1}{N}\right)^k\left(\frac{N-1}{N}\right)^{n-k} && \text{par la formule } n\binom{n-1}{k-1} = k\binom{n}{k}. \\
&= \sum_{k=0}^{n-1} n\binom{n-1}{k}\left(\frac{1}{N}\right)^{k+1}\left(\frac{N-1}{N}\right)^{n-1-k} && \text{par changement d'indice.} \\
&= \frac{n}{N}\sum_{k=0}^{n-1}\binom{n-1}{k}\left(\frac{1}{N}\right)^k\left(\frac{N-1}{N}\right)^{n-1-k} \\
&= \frac{n}{N}\left(\frac{1}{N}+\frac{N-1}{N}\right)^{n-1} && \text{par la formule du binôme de Newton.} \\
&= \frac{n}{N}
\end{aligned}$$

Question 18. En supposant que l'élément est dans la table de hachage et que les éléments dans chaque liste sont dans un ordre aléatoire, quel est la complexité moyenne $C_{moyen}(n)$? Quelle est sa valeur asymptotique ?

Correction : Pour trouver un élément dans une liste d'ordre aléatoire de longueur l, on a besoin d'en moyenne $\frac{1}{l}\sum_{k=1}^{l} k = \frac{l+1}{2}$

Par linéarité du calcul de la moyenne, on obtient une complexité moyenne de $C_{moyen}(n) = \frac{1}{2}(\frac{n}{N} + 1)$.

La complexité asymptotique est en $O(\frac{n}{N})$.

Partie IV : Redimensionnement

Il est donc intéressant de changer de taille pour accélérer l'accès à la table de hachage. Pour ce faire, on crée un nouveau tableau dans lequel on ajoute les éléments qui étaient déjà dans la table. Comme le nombre d'alvéoles changent, les entiers changent de place dans le tableau, et on doit donc recalculer leur position.

Question 19. Représenter la table `[|[3]; [2; 1; 8]; []|]` de hachage après avoir changé sa taille, c'est-à-dire le nombre d'alvéoles, pour être égale à 6.

Correction : On calcule les différentes valeurs de `[|[]; [1]; []; [3]; [8; 2]; []|]`

Question 20. Proposer une fonction `changer_taille` de signature `int list array -> int -> int list array` qui prend en argument une table de hachage et un entier `nouvelle_taille`, et qui renvoie une nouvelle table de hachage de taille `nouvelle_taille` qui contient les entiers contenus dans la première table de hachage.

L'ordre des entiers dans les listes de la table de hachage n'est pas important : il suffit que les entiers soient dans la bonne case de la table.

Correction : On utilise une fonction auxiliaire qui itère sur une liste pour en ajouter les éléments dans le nouveau tableau, ainsi qu'une boucle `for` qui itère sur les éléments du tableau :

```
let changer_taille table nb_nouveau =
    let nb_ancien = Array.length table in
    let resultat = Array.make nb_nouveau [] in
    let rec aux l = match l with
    | [] -> ()
    | p::q -> ajouter resultat p ; aux q in
    for i = 0 to nb_ancien - 1 do
        aux table.(i)
    done ; resultat
```

Question 21. Dans cette question, on utilise les opérations de base comme étant de complexité constante. En particulier un appel à `hachage` est considéré comme étant de coût constant. On supposera que la construction d'un tableau vide de taille N est en $O(N)$ dans le pire des cas.

Quelle est complexité asymptotique dans le pire des cas de `changer_taille` en fonction de n le nombre d'éléments présents dans la première table de hachage, et N le nombre d'alvéoles dans la nouvelle table de hachage ?

Correction : On fait n appels à la fonction auxiliaire, et on fait un appel de coût $O(N)$ lors de la construction du tableau de taille N. Le coût total est en $O(n+N)$.

Étant donné que la complexité moyenne de recherche dépend du nombre d'alvéoles et du nombre d'éléments présents dans la table, on cherche à garder le rapport $\frac{n}{N}$ strictement inférieur à une valeur seuille $\alpha \geqslant 0$.

Ainsi, à tout moment, on a que $n < \alpha N$. Si on rajoute un élément qui fait dépasser le seuil, on modifie la table pour que la nouvelle table soit de taille βN pour un $\beta > 1$ fixe.

On prend dans la suite $\alpha = \beta = 2$.

Question 22.

1. En partant d'une table de hachage vide à une seule alvéole, combien de changement de taille de table doit-on réaliser lors de l'ajout de 2^k éléments pour $k \in \mathbb{N}$?
2. Montrer que la complexité asymptotique totale pour l'ajout de 2^k éléments est en $O(2^k)$.

Correction :

1. On doit faire le changement de taille pour le nombre d'éléments ajoutés suivant : 2^1, 2^2, ..., 2^k. On a donc k redimensionnements.
2. La complexité des ajouts est linéaire en 2^k, et la complexité des redimensionnements est linéaire en $\sum_{i=1}^{k} = 2^{k+1} - 2$. Donc la complexité totale est en $O(2^k)$.

Question 23. Proposer une fonction `taille` qui calcule le nombre d'éléments dans une table de hachage passée en argument.

Correction : On utilise une fonction auxiliaire qui calcule la taille d'une liste dont ont somme les résultats sur les éléments de la table de hachage.

```
let taille table =
    let nb = Array.legnth table in
    let resultat = ref 0 in
    let rec aux l = match l with
    | [] -> 0
    | _::q -> 1 + aux q
    in
    for i = 0 to nb-1 do
        resulat:= !resultat + aux table.(i)
    done ;
    !resultat
```

Question 24. Quelle est la complexité asymptotique de la fonction `taille` en fonction de n le nombre d'éléments dans la table et N le nombre d'alvéoles dans la table ?

Correction : La fonction auxiliaire est exécutée n fois au total, et la boucle est exécutée N fois au total, car il faut parcourir toutes les alvéoles et tous les les éléments de la table.

La complexité asymptotique totale est en $O(n + N)$.

Pour l'instant, nous n'avons pas de solution efficace pour trouver le nombre d'éléments dans la table de hachage, il faut faire un parcours exhaustif, ce que nous ne pouvons pas nous permettre à chaque ajout.

Question 25. Proposer une manière de pouvoir efficacement trouver le nombre d'éléments dans une table de hachage. On pourra modifier le type de la table de hachage pour y rajouter des informations supplémentaires.

Comment doivent être modifiés les fonctions précédentes pour prendre en compte cette méthode ?

On n'implémentera pas nécessairement les fonctions, mais on pourra décrire de manière générale comment doivent avoir lieu les modifications.

Correction : On utilise une variable supplémentaire qui contient le nombre d'éléments dans la table de hachage que l'on mettra à jour à chaque ajout ou retrait. La table et ce compteur seront stockés dans un couple.

Par ailleurs, étant donné que la table peut être changée par la fonction d'ajout, il faut un moyen de pouvoir prendre cette modification en compte dans le type de la table de hachage.

Question 26. Proposer une manière de pouvoir prendre en compte les modifications éventuelles de la table lors de l'ajout. On pourra changer le type d'ajouter pour qu'elle renvoie la table de hachage, ou bien utiliser une variable mutable qu'on puisse modifier avec ajout.

Proposer une implémentation de la fonction `ajouter` qui prenne en compte les différentes modifications.

Correction : De sorte à pouvoir prendre en compte la modification du compteur d'élément et de la table de hachage, on utilise un couple de référence dont le premier élément est une référence d'entier, et le second est une référence de tableau de listes d'entiers. On obtient donc le type `(int ref * int list array ref)` pour une table de hachage.

Cela nous donne la fonction `ajouter` suivante :

```
let ajouter t x =
    let table = !(snd t) in
    let compteur = fst t in
    let taille0 = Array.length table in
    let h = hachage x taille0 in
    table.(h) <- x::table.(h) ;
    compteur:= !compteur + 1 ;
    if !compteur >= taille0 * 2 then
        (snd t) := changer_taille table (taille0 *2)
```