# Devoir Non Surveillé 4

À rendre le 22 avril 2024

## INFORMATIQUE MP2I

**Tournez la page S.V.P.**

# 1 Expressions arithmétiques

On définit en OCaml les expressions littérales comme des arbres binaires, où les nœuds internes sont étiquetés par des opérateurs binaires (ici + et ×), et les feuilles sont étiquettées soit par un nombre entier soit par la variable $x$ :

Expressions littérales

```
type operator = Add | Mul
type expr =
  | Int of int
  | Var
  | Op of expr * operator * expr

let exemple = Op (Int 2, Add, Op (Op (Int 3, Add, Int 7), Mul, Int 4))
let exemple_bis = Op (Op (Int 3, Mul, Var), Add, Int (-126))
```

☐ 1 – Écrire une fonction nombres : expr -> int qui compte le nombre de feuilles (nombres ou variables) dans une expression littérale.

**Réponse 1.**

```
let rec nombres = function
  | Int _ | Var -> 1
  | Op (g, r, d) -> nombres g + nombres d
```

☐ 2 – Écrire une fonction operateurs : expr -> int qui compte le nombre d'opérateurs binaires dans une expression littérale.

**Réponse 2.**

```
let rec operateurs = function
  | Int _ | Var -> 0
  | Op (g, r, d) -> 1 + operateurs g + operateurs d
```

☐ 3 – Que peut-on dire des résultats des deux fonctions précédentes sur une même expression littérale ? le démontrer.

**Réponse 3.** On peut remarquer que la différence entre le nombre de feuilles et le nombre d'opérateurs est toujours 1.
Prouvons le par induction structurelle :

$\langle \emptyset, r, \emptyset \rangle$ **:** Dans le cas d'une feuille, il y a une feuille et aucun opérateur, c'est OK.

$\langle g, r, d \rangle$ **:** Le nombre de feuilles est $f = f_g + f_d$ et le nombre d'opérateurs $o = o_g + o_d + 1$, avec $f_g = o_g + 1$ et $f_d = o_d + 1$, ce qui donne $f = f_g + f_d = o_g + 1 + o_d + 1 = o + 1$. C'est OK.

☐ 4 – Écrire une fonction eval : int -> expr -> int qui prend en entrée un entier $x$ et une expression littérale et qui renvoie le résultat de l'expression si la variable vaut $x$.
*On pourra vérifier les valeurs de eval 0 exemple et eval 42 exemple_bis.*

**Réponse 4.**

```
let rec eval x = function
  | Int n -> n
  | Var -> x
  | Op (g, Add, d) -> eval x g + eval x d
  | Op (g, Mul, d) -> eval x g * eval x d
```

☐ 5 – Écrire des fonctions prefixe : expr -> string,
postfixe : expr -> string et infixe : expr -> string qui permettent de transformer une expression littérale en une chaîne de caractères qui la représente en notation préfixe, postfixe ou infixe.

**Réponse 5.**

```
let string_of_op = function
  | Add -> "+"
  | Mul -> "*"

let rec parcours pprint = function
  | Int n -> string_of_int n
  | Var -> "x"
  | Op (g, r, d) -> pprint (string_of_op r) (parcours pprint g) (parcours pprint d)

let prefixe = parcours (fun op g r -> op ^ "␣" ^ g ^ "␣" ^ d)
let postfixe = parcours (fun op g r -> g ^ "␣" ^ d ^ "␣" ^ op)
let infixe = parcours (fun op g r -> "(" ^ g ^ op ^ d ^ ")")
```

☐ 6 – En réalité, les expressions que l'on manipule ici sont des polynômes (il n'y a pas de division). Écrire une fonction degree : expr -> int qui calcule une borne sur le degré d'un polynôme représenté par une expression littérale.
Donner la complexité de votre fonction.
*On pourra simplifier les disjonctions de cas en utilisant* 0 *comme borne sur le degré du polynôme identiquement nul* $P(X) = 0$.

**Réponse 6.**

```
let rec degree = function
  | Int n -> 0
  | Var -> 1
  | Op (g, Add, d) -> max (degree g) (degree d)
  | Op (g, Mul, d) -> degree g + degree d
```

Cette fonction parcours tous les nœuds de l'arbre et effectue uniquement des opérations élémentaires sur chacun donc sa complexité est linéaire en la taille de l'arbre.

☐ 7 – Écrire une fonction egales : expr -> expr -> bool qui teste l'égalité entre deux expressions littérales basée sur l'estimation du degré implémentée à la fonction précédente.
Estimer la complexité de votre fonction.

**Réponse 7.**

```
let rec egales e1 e2 =
  let d = max (degree e1) (degree e2) in
  let rec test_valeurs i =
```

```
    (* Si e1 - e2 possede (d+1) racines il est identiquement nul *)
    if i = 0 then true
    else eval i e1 = eval i e2 && test_valeurs (i-1)
  in
  test_valeurs (d+1)
```

eval et degree ont une complexité linéaire en la taille de l'arbre. On en déduit que pour deux arbres de taille $\leq n$ représentant des polynômes de degré $\leq d$, la complexité de cette fonction est en $O(dn)$.

Nous souhaitons à présent développer et réduire nos expressions pour obtenir une forme canonique. Nous allons donc représenter à présent les polynômes par une suite finie de monômes. Remarquons que le polynôme identiquement nul sera représenté par une liste vide, les monômes étant donc de degré positif ou nul. Pour que la représentation canonique soit unique, nous trierons les monômes par degré strictement décroissant.

Polynômes

```
type monomial = {deg : int; coeff : int}
type polynomial = monomial list
```

☐ 8 – Écrire une fonction add_monomial : monomial -> polynomial -> polynomial qui ajoute un monôme à un polynôme.
En déduire une fonction sum : polynomial -> polynomial -> polynomial qui somme deux polynômes.

**Réponse 8.**

```
let rec add_monomial m : polynomial -> polynomial = function
  | [] -> [m]
  | t :: q ->
     if t.deg = m.deg then
       let new_coeff = t.coeff + m.coeff in
       if new_coeff = 0 then q
       else {deg = t.deg; coeff = new_coeff} :: q
     else if t.deg < m.deg then m :: q
     else t :: add_monomial m q

let rec sum p : polynomial -> polynomial = function
  | [] -> p
  | t :: q -> sum (add_monomial t p) q
```

☐ 9 – Écrire une fonction distribute : monomial -> polynomial -> polynomial qui multiplie un polynôme par un monôme donné en utilisant la distributivité.
En déduire une fonction prod : polynomial -> polynomial -> polynomial qui multiplie deux polynômes.

**Réponse 9.**

```
let rec distribute m : polynomial -> polynomial = function
  | [] -> []
  | t :: q -> let t' = {deg = t.deg + m.deg; coeff = t.coeff * m.coeff} in
              t' :: distribute m q

let rec prod p : polynomial -> polynomial = function
  | [] -> []
  | t :: q -> sum (distribute t p) (prod p q)
```

☐ 10 – Écrire une fonction expand_simplify : expr -> polynomial qui développe et réduit une expression littérale pour en déduire sa forme polynomiale.

**Réponse 10.**

```
let choose_op = function
  | Add -> sum
  | Mul -> prod

let rec expand_simplify : expr -> polynomial = function
  | Int n -> if n = 0 then [] else [{deg = 0; coeff = n}]
  | Var -> [{deg = 1; coeff = 1}]
  | Op (g, r, d) -> (choose_op r) (expand_simplify g) (expand_simplify d)
```

☐ 11 – Expliquer comment calculer le degré d'un poylnôme donné sous forme de liste de monôme, ainsi que comment tester l'égalité de deux polynômes.
Estimer les complexités de ces deux opérations.

**Réponse 11.** On peut obtenir le degré d'un polynôme en $O(1)$ en regardant le degré du premier monôme comme ils sont triés dans l'ordre décroissant.
On peut tester l'égalité de deux polynômes en vérifiant que leurs listes de monômes sont égales, la complexité est donc $O(d)$ dans le pire des cas où $d$ est le plus grand degré des deux.

## 2 À cheval sur deux tableaux triés

Le but de cette partie est de déterminer le $k+1$-ième plus petit élément, dans deux tableaux triés.
Par exemple, si on considère les tableaux , 2, 4, 6, 80, 2, 4, 6, 8 et , 3, 5, 7, 91, 3, 5, 7, 9, le 4-ième plus petit élément est 3. C'est-à-dire que si on regroupait les deux tableau en un tableau trié, il aurait la position 3. On ne vérifiera pas que les tableaux donnés en entrée sont effectivement triés. On suppose dans toute la suite que $k < n_1 + n_2$, sinon le problème n'a pas de solution.

☐ 12 – Écrire une fonction **int** *fusion(**int** tab1[], **int** tab2[], **int** n1, **int** n2) qui renvoie un tableau trié de taille $n_1+n_2$ contenant les éléments de $t_1$ et $t_2$, où $n_i$ est la longueur de $t_i$. On prendra soin d'allouer la mémoire dynamiquement (malloc).

**Réponse 12.**

**C**

```
int *fusion(int tab1[], int tab2[], int n1, int n2){
  int *tab = malloc((n1 + n2) * sizeof(int));
  int i = 0, j = 0;
  while (i + j < n1 + n2){
    if (j == n2 || tab1[i] < tab2[j]){
      tab[i + j] = tab1[i];
      i = i + 1;
    }
    else {
      tab[i + j] = tab2[j];
      j = j + 1;
    }
  }
  return tab;
}
```

☐ 13 – En déduire une fonction **int** select_naif(**int** k, **int** tab1[], **int** tab2[], **int** n1, **int** n2) qui détermine le $k+1$-ième plus petit élément de $t_1 \cup t_2$ en utilisant fusion. On prendra soin de libérer la mémoire allouée par fusion.

**Réponse 13.**

**C**

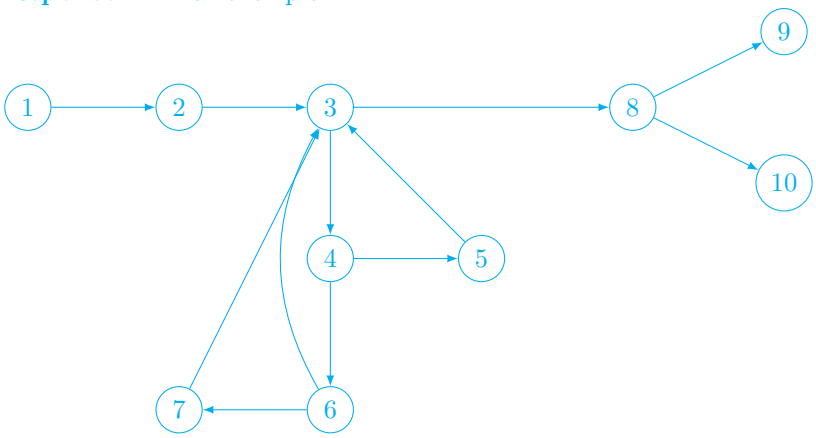
```c
int select_naif(int k, int tab1[], int tab2[], int n1, int n2){
  int *tab = fusion(tab1, tab2, n1, n2);
  int res = tab[k];
  free(tab);
  return res;
}
```

Pour éviter l'allocation et l'initialisation de ce tableau, on se propose d'utiliser le code suivant, manipulant le même tableau, mais sans le créer.

```c
int select_lin(int k, int tab1[], int tab2[], int n1, int n2){
  assert(k < n1 + n2);                          // 1
  int i = 0, j = 0;                             // 2
  while (i + j < k) {                           // 3
    if (j >= n2 || tab1[i] < tab2[j])           // 4
      { i = i + 1; }                            // 5
    else if (i >= n1 || tab1[i] >= tab2[j])     // 6
      { j = j + 1; }                            // 7
  }
  if (j == n2 || tab1[i] < tab2[j])             // 8
    { return tab1[i]; }                         // 9
  else { return tab2[j]; }                      // 10
}
```

☐ 14 – Dessiner le graphe de flot de contrôle de la fonction select_lin et proposer un jeu de test couvrant l'ensemble de ses branches.

**Réponse 14.** Par exemple :




Remarque : on a pas dessiné de nœud pour la fin de la fonction, ni pour l'erreur potentielle du assert.

☐ 15 – Donner un variant permettant de montrer la terminaison de la boucle **while** et donc de la fonction select_lin. En déduire la complexité temporelle dans le pire des cas en utilisant la notation $O(\cdot)$.

**Réponse 15.** $k - i - j$ est un variant.
En effet, il est toujours positif grâce à la condition d'arrêt, et de plus, si la première condition n'est pas satisfaite, la deuxième l'est nécessairement (si $j \geq n_2$ et $i + j < k < n_1 + n_2$, alors $i < n_1$). Donc on a toujours soit $i$ soit $j$ qui augmente strictement alors que l'autre stagne. $k - i - j$ décroît de 1 à chaque étape.
De plus, chaque itération s'exécute en $O(1)$, donc la complexité temporelle dans le pire des cas de cette fonction est $O(k) = O(n_1 + n_2)$.

☐ 16 – Démontrer à l'aide d'un invariant de la boucle **while** la correction partielle de la fonction select_lin. On rappelle que les tableaux $t_1$ et $t_2$ sont supposés triés.

**Réponse 16.** On a l'invariant suivant :
$i+j$ éléments sont plus petits que le minimum de $t_1[i]$ et $t_2[j]$.

**Initialisation :** Au début, $i=j=0$ et le minimum du tableau est soit $t_1[0]$ soit $t_2[0]$ car les tableaux sont triés.

**Hérédité :** Supposons la propriété vraie au début d'une itération.
Alors $i+j$ éléments sont plus petits que le minimum de $t_1[i]$ et $t_2[j]$.
Si $t_1[i] < t_2[j]$, alors au moins $i+j+1$ éléments sont plus petits que $t_2[j]$ et $t_1[i+1]$. De plus, si $t_1[i+1] < t_2[j]$ aucun autre élément ne peut être plus petit que $t_1[i+1]$ (on suppose que les tableaux sont sans doublons, quitte à travailler avec l'ordre lexicographique sur le triplet (valeur, tableau, indice) qui nous donnera une relation d'ordre totale). En effet, les éléments $t_1[k]$ pour $k > i+1$ sont plus grands car $t_1$ est trié, et les éléments $t_2[k]$ pour $k > j$ sont plus grands que $t_2[k]$ car $t_2$ est trié, donc plus grands que $t_1[i+1]$ par transitivité.
On a donc $i+j+1 = i'+j$ éléments plus petits que le minimum de $t_1[i+1] = t_1[i']$ et $t_2[j]$.
L'autre cas est similaire en échangeant les rôles de $t_1$ et $t_2$.

À la fin de la boucle, l'invariant reste vrai donc exactement $i+j$ éléments sont plus petits que le minimum de $t_1[i]$ et $t_2[j]$, et $i+j=k$, d'où la correction de l'algorithme.
Remarque : pour simplifier l'argument, on a ignoré le cas où un des deux tableaux est parcouru entièrement. On peut facilement étendre ce raisonnement en ajoutant virtuellement $+\infty$ à la fin de chaque tableau et conclure la correction partielle dans tous les cas.

On souhaite désormais trouver un algorithme plus efficace, utilisant mieux le fait que les tableaux soient triés. Soit $i$, tel que $0 \leq i$, $i < n_1$, $i \leq k$. Posons $j = k - i$ (remarquons que $j < n_2$ car $k < n_1 + n_2$).

☐ 17 – Montrer que si $t_1[i-1] \leq t_2[j]$ et $t_2[j-1] \leq t_1[i]$, alors le $(k+1)$-ième plus petit élément est $\min(t_1[i], t_2[j])$.

**Réponse 17.** Il y a $i$ éléments de $t_1$ plus petits que $t_1[i]$ et que $t_2[j]$ car ils sont tous deux plus grands que $t_1[i-1]$, et il y a de même $j$ éléments de $t_2$ plus petits que $t_1[i]$ et que $t_2[j]$ car ils sont tous deux plus grands que $t_2[j-1]$. Tous les autres éléments de $t_1$ sont plus grands que $t_1[i]$ et les éléments de $t_2$ sont plus grands que $t_2[j]$. Donc il y a exactement $i+j=k$ éléments plus petits que le minimum des deux.

☐ 18 – Montrer que si $t_1[i-1] > t_2[j]$, alors le $(k+1)$-ième plus petit élément est plus petit que $t_1[i]$.

**Réponse 18.** Il y a $i$ éléments dans $t_1$ plus petits que $t_1[i-1]$ et $j+1$ éléments de $t_2$ plus petits que $t_1[i-1]$, donc l'élément que l'on recherche est plus petit que $t_1[i]$.

☐ 19 – Déduire des deux questions précédentes une fonction **int** select_log(**int** k, **int** tab1 [], **int** tab2 [], **int** n1, **int** n2) qui détermine le $(k+1)$-ième plus petit élément en temps $O(\log(n_1))$.

**Réponse 19.**

**C**

```
int min(int a, int b){
  if (a < b) return a;
  return b;
}

int select_log(int k, int tab1[], int tab2[], int n1, int n2){
  int d = k - min(n2, k);
  int f = min(n1-1, k);
  /* On cherche par dichotomie le nombre i-1 d'éléments de tab1 plus petit que l'élément c
  /* L'indice i recherché est au moins 0 et au moins k-n2 et ne peut dépasser k ni n1-1 */
  while (f > d){
    int i = (d + f) / 2;
    int j = k - i;
    if ((j == n2 || i == 0 || tab1[i-1] <= tab2[j])
        && (i == n1 || j == 0 || tab2[j-1] <= tab1[i]))
```

```
      return min(tab1[i], tab2[j]);
    if (j == k || tab1[i-1] > tab2[j]) f = i - 1;
    else d = i + 1;
  }
  assert(false);
  return -1;
}
```

☐ 20 – Expliquer comment on pourrait résoudre le problème dans le pire des cas en $O(\log(\min(n_1, n_2)))$.

**Réponse 20.** Il suffit d'inverser les rôles de $t_1$ et $t_2$ si $n_2 < n_1$, par exemple avec un appel récursif.

# 3 Scrabble – OCaml (20 pts)

Dans cette section, nous allons nous intéresser à un programme qui joue au Scrabble.
Chaque lettre de l'alphabet latin ('A','B',...,'Z') est associé à un nombre de points allant de 1 à 10.
Étant donné un ensemble de 7 lettres, on cherche à faire le mot valide valant le plus de points. Si on arrive à placer ses 7 lettres, on gagne un bonus de 50 points.
Nous allons ici ignorer le placement du mot sur le plateau.
Pour savoir si un mot est valide, nous disposons d'un fichier `dictionnaire.txt` qui contient la liste de tous les mots valides, séparés par des espaces (on suppose la présence d'un espace supplémentaire en première et dernière position, de telle manière que chaque mot est entouré par deux espaces). Nous affectons son contenu à une variable dict : string, dont on peut accéder au $i$-ième caractère par dict.[i].

```
let f = open_in "dictionnaire.txt"
let dict = input_line f
let () = close_in f
```

Dans un premier temps, intéressons nous à la possibilité de faire un Scrabble, c'est-à-dire de placer les 7 lettres. Il faut donc chercher dans le dictionnaire tous les mots contenant nos 7 lettres.

☐ 21 – Dans le pire des cas, combien peut-on former de mots (pas nécessairement valides) différents que l'on doit tester avec 7 lettres ?

**Réponse 21.** $7! = 5040$.

Nous allons adapter l'algorithme de Rabin-Karp pour trouver aussi les anagrammes d'un motif, c'est-à-dire un facteur du mot qui contient les mêmes lettres que le motif, mais pas nécessairement dans le même ordre. Pour ceci, on propose de donner un code à chaque caractère ainsi : '␣' -> 0, 'A' -> 1, 'B' -> 2... 'Z' -> 26.

```
let code c =
  if c = ' ' then 0 else
    begin
      assert ('A' <= c && c <= 'Z');
      int_of_char c - int_of_char 'A' + 1
    end
```

On propose d'utiliser la fonction de hachage suivante sur les mots de taille arbitraire :

$$h : c_0, \dots, c_n \mapsto \prod_{i=0}^{n} p_{\bar{c}_i},$$

où $p_k$ désigne le $k$-ième nombre premier, et $\bar{c}$ désigne le code du caractère $c$.

☐ 22 – Montrer que deux mots **b** et **c** sont des anagrammes si et seulement si $h(\mathbf{b}) = h(\mathbf{c})$.

**Réponse 22.**
=> commutativité de la multiplication
<= unicité de la décomposition en produit de facteurs premiers

On suppose à présent qu'on dispose d'un tableau **p** contenant les 27 premiers nombres premiers :

```
let p = [|
    2; 3; 5; 7;11;
    13;17;19;23;29;
    31;37;41;43;47;
    53;59;61;67;71;
    73;79;83;89;97;
    101;103|]
```

☐ 23 – Écrire une fonction hash : string -> int -> int qui prend en entrée une chaîne de caractère **c** et un entier $n$ et qui calcule $h(c_0, \dots, c_{n-1})$.
Cette fonction vérifiera que la taille de **c** est au moins $n$.

**Réponse 23.**

```
let rec hash str n =
  assert (String.length str >= n);
  if (n = 0) then 1 else
    p.(code str.[n-1]) * hash str (n-1)
```

☐ 24 – Si on trouve un facteur $\mathbf{b} = b_0 \dots b_6$ du dictionnaire tel que $h(\mathbf{b}) = h(\mathbf{c})$, peut-on être sur de l'existence d'un anagramme de $\mathbf{c}$ dans le dictionnaire ? Justifier.

**Réponse 24.** Non. Par exemple, le facteur de 7 lettres "AUSSETT" apparaît dans le mot "CHAUSSETTE" de dictionnaire mais n'est pas un mot valide.

☐ 25 – Paul pense que vérifier l'égalité des images par $h$ n'est pas suffisant et propose de tester que le motif contienne bien deux espaces en considérant les anagrammes de $\mathbf{c}' = {}_\sqcup c_0 \dots c_{6\sqcup}$. Si on trouve un facteur $\mathbf{b} = b_0 \dots b_8$ du dictionnaire tel que $h(\mathbf{b}) = h(\mathbf{c}')$, peut-on être sur de l'existence d'un anagramme de $\mathbf{c}$ dans le dictionnaire ? Justifier.

**Réponse 25.** Toujours insuffisant. Un des deux espaces peut se trouver au milieu du facteur : "TE CHAUX " par exemple si on imagine que le mot suivant "CHAUSSETTE" est "CHAUX".

☐ 26 – Écrire une fonction verifie : 'a −> 'a −> string −> int −> int −> bool qui prend en argument la valeur $h(\mathbf{c}')$ pour un certain motif $\mathbf{c}' = {}_\sqcup\mathbf{c}_\sqcup$, avec $\mathbf{c}$ de taille $n$, la valeur $h(b_i, b_{i+n+1})$, le dictionnaire $\mathbf{b}$, la taille du motif $n$ et la position $i$ et qui vérifie si on a bien un anagramme de $\mathbf{c}$ présent en position $i+1$ dans le dictionnaire $\mathbf{b}$.
*remarque : la fonction ne prend pas le motif* $\mathbf{c}$ *en argument.*

**Réponse 26.**

```
let verifie h1 h2 str n i =
  assert (String.length str >= i+n+2);
  h1 = h2 && str.[i] = ' ' && str.[i+n+1] = ' '
```

☐ 27 – Peut-on avoir un dépassement de capacité lorsque l'on calcule une image par $h$ d'un facteur de taille 9 ? Pour la suite, on supposera que l'on a jamais de dépassement de capacité.

**Réponse 27.** La valeur maximale que peut prendre l'image par $h$ est $103^9 < 2^{62} - 1$ donc on ne peut pas avoir de dépassement de capacité. Cependant, il faut faire attention à l'ordre des opérations par la suite et privilégier la division avant la multiplication, car pour un facteur de taille 10, on a $103^10 > 2^{62} - 1$, donc on risque le dépassement de capacité

On propose l'implémentation suivante à compléter d'une variante de l'algorithme de Rabin-Karp :

variante de Rabin-Karp

```
let rabin_karp motif texte =
  let n = String.length motif in
  let h = hash motif n in
  let emp = hash texte n in
  let rec parcours emp i =
    if i > String.length texte - n then
      false
    else if verifie h emp texte n i then
      true
    else
      let nouv_emp = (* A COMPLETER *) in
      parcours nouv_emp (i+1)
  in
  parcours emp 0
```

☐ 28 – Compléter l'algorithme en remplaçant le commentaire.

**Réponse 28.**

```
let rabin_karp motif texte =
  let n = String.length motif in
  let h = hash motif n in
  let emp = hash texte n in
  let rec parcours emp i =
    if i > String.length texte - n then
```

```
      false
    else if verifie h emp texte n i then
      true
    else
      let nouv_emp = (emp/p.(code texte.[i]))*p.(code texte.[i+n-1]) in
      parcours nouv_emp (i+1)
  in
  parcours emp 0
```

☐ 29 – Quelle est la complexité de cet algorithme ?

**Réponse 29.** $O(n+m)$ où $n$ est la taille du motif et $m$ la taille du dictionnaire. En effet, la verification et le calcul de la nouvelle empreinte se font en temps constant.

☐ 30 – Peut-on adapter cette technique pour chercher aussi les anagrammes de sous-ensembles de 6 lettres de notre tirage ? Est-ce avantageux ?

**Réponse 30.** On peut adapter l'algorithme de Rabin-Karp pour chercher plusieurs motifs de même taille. On ne factorisera pas le calcul de l'empreinte entre les facteurs de 6 lettres et les facteurs de 7 lettres, mais on pourra tester les 7 empreintes des sous-ensembles de 6 lettres pour chaque position en ne calculant qu'une seule fois l'empreinte de facteur de commençant à cette position.

# 4 Complexité en moyenne des tables de hachages

Dans cette section, nous estimons la complexité moyenne dans le pire des cas pour les opérations de recherche et d'insertion dans une table de hachage. Nous supposerons que la fonction de hachage s'évalue en $O(1)$.
Nous notons $n$ le nombre d'éléments dans la table de hachage et $m$ le nombre de cases du tableau. Nous noterons $\alpha = \frac{n}{m}$ le facteur de remplissage de la table.
Nous supposons que pour toute entrée $x$, notre fonction de hachage renvoie un nombre aléatoire entre 0 inclu et $m$ exclu, et que les résultats pour deux entrées distinctes $x$ et $y$ sont indépendants.
Nous avons donc que la probabilité de collision pour deux éléments donnés est $\frac{1}{m}$.

☐ 31 – Quelle est la probabilité qu'il y ait au moins une collision parmi les $n$ premiers éléments qui ont été insérés en fonction de $n$ et $m$ ?

**Réponse 31.** S'il n'y en a pas encore eu, la $i$-ième insertion donne une collision avec probabilité $\frac{i-1}{m}$.
On en déduit que la probabilité d'avoir eu au moins une collision est de 1 si $n > m$ et $\prod_{i=1}^{n}\left(1-\frac{i-1}{m}\right)$ sinon.

## 4.1 Gestion des collisions par chaînage

Dans cette section, on décide de stocker dans chaque case du tableau une liste chaînée d'associations pour gérer les collisions. Ainsi, la complexité de la recherche est proportionnelle à la taille de la liste contenue dans la case explorée. La complexité de l'addition d'un élément est $O(1)$ si on fait les ajouts en tête de liste.

☐ 32 – Après $n$ insertions dans des cases aléatoires parmi $m$ cases, quelle est l'espérance du nombre d'éléments dans une case donnée ? En déduire la complexité en espérance dans le pire des cas d'une recherche.

**Réponse 32.** Les tirages sont indépendants, chaque tirage ajoute en espérance $\frac{1}{m}$ éléments dans une case donnée. Par linéarité de l'espérance, il y a en espérance $\alpha = \frac{n}{m}$ éléments dans chaque case. On en déduit que la complexité en espérance dans le pire des cas est $O(1+\alpha)$ car dans le pire des cas, on parcourt la liste en entier avant de trouver l'élément ou de ne pas le trouver. (on note toutefois que l'analyse probabiliste devrait être plus poussée car le fait d'étudier les probabilités conditionnelles dans les cas où la clé recherchée appartient ou non à l'ensemble des clés peut faire varier légèrement les probabilités)

## 4.2 Gestion des collisions par adressage ouvert avec sondage linéaire

Dans cette section, on décide pour gérer les collisions que si la case est déjà prise, on va chercher dans la case suivante, jusqu'à trouver une case vide.

☐ 33 – Expliquer pourquoi la distribution des $n$ cases occupées parmi les $m$ cases disponible ne correspond pas à choisir une des $\binom{m}{n}$ configurations aléatoirement de manière uniforme.

**Réponse 33.** Si on choisit $n = 2$ par exemple, on a 1.5 fois plus de chances de tomber sur 2 cases occupées côte à côte que deux cases séparés par une case. En effet, si on note $x$ la première case choisie, $x-1$, $x$ et $x+1$ mènent à la configuration de deux cases contigües occupées, alors que seuls $x-2$ et $x+2$ mènent à la seconde configuration.

Fin de l'épreuve