# Algorithmique des textes



*Source image : justgrims, https://www.flickr.com/photos/notbrucelee/8016192302*

■ **Note 1 Roadmap :**
- Des exercices.
- Les extensions à la fin.

■

Sources
- *Algorithms* Robert Sedgewick, Kevin Wayne
- *Éléments d'algorithmique* D. Beauquier, J. Berstel, Ph. Chrétienne
- *125 Problems in Text Algorithms with Solutions* Maxime Crochemore, Thierry Lecroq, Wojciech Rytter

# I Recherche dans un texte

## I.1 Principe de la recherche

On s'intéresse ici au problème suivant :

**Problème** - RECHERCHETEXTE
- Entrées :
  - ⋆ une chaîne de caractère $s$ sur l'alphabet $\Sigma$
  - ⋆ un autre chaîne de caractère $m$ sur ce même alphabet appelé *motif* et de longueur plus petite que $s$
- Sortie :
  un résultat partiel correspondant à l'indice de la première occurrence du motif dans la chaîne s'il est présent.

La différence fondamentale entre ce problème et celui de la recherche d'un sous-tableau dans un tableau est le fait qu'on considère un alphabet fini et dont le nombre d'éléments est le plus souvent négligeable par rapport à la taille des chaînes de caractères. Cela permet d'effectuer des optimisations qui ne sont pas sans rappeler les tris linéaires comme le tri par comptage.

On parle alors d'algorithmique du texte pour désigner des algorithmes tirant partie de cette contrainte sur les données. La plupart des algorithmes que l'on présente peuvent ainsi s'adapter aisément au cas de tableaux dont les éléments sont pris dans un ensemble fini de petit cardinal.

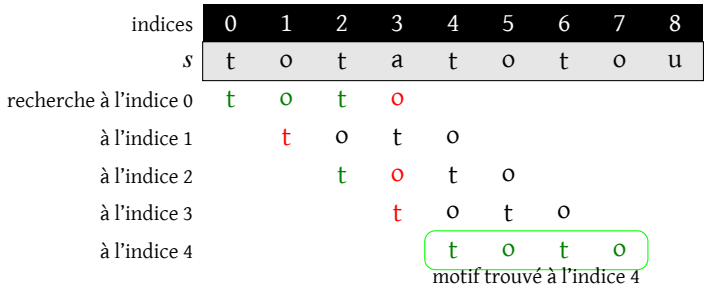
Avant d'entamer ce chapitre, remarquons qu'il existe, outre l'alphabet usuel, trois alphabets très importants :

- celui des caractères ASCII usuels
- celui contenant les deux éléments 0 et 1, ce qui permet de travailler sur des recherche en binaire.
- et enfin, très important pour la biologie, l'alphabet à quatre lettres A, T, G et C correspondant aux bases d'un brin d'ADN et qui ouvre la porte à beaucoup d'applications en bio-informatique.

■ **Note 2** Il y aura sûrement des applications bio-info dans la partie programmation dynamique, faire le lien ici. ■

## I.2 Algorithme naïf en force brute

Une solution naïve consiste à parcourir chaque position de $s$ afin de tester si le motif est présent à partir de cette position.




Cela donne l'implémentation assez directe suivante :

OCaml

```
exception Trouve of int
exception PasDeMotif

(* renvoie un booléen indiquant si m est présent
 * dans s à l'indice i *)
let cherche_motif (m :string) (s :string) (i :int) : bool =
    let p = String.length m in
    try
        for j = 0 to p-1 do
            if s.[i+j] <> m.[j]
            then raise PasDeMotif
        done;
        true
    with PasDeMotif -> false

let recherche_naive m s =
    let n = String.length s in
    let p = String.length m in
    try
        for i = 0 to n-p do
            if cherche_motif m s i
            then raise (Trouve i)
        done;
        None
    with Trouve i -> Some i
```

C

```
/* recherche_naive(m,s) recherche le motif m dans la chaine
 * s et renvoie l'indice de la première occurrence s'il est présent
 * ou -1 sinon */
int recherche_naive(const char *m, const char *s)
{
    int n = strlen(s);
    int p = strlen(m);

    for (int i = 0; i <= n-p; i++)
    {
        int j;
        for (j = 0; j < p; j++)
        {
            if (s[i+j] != m[j])
                break;
        }
        if (j == p)
            return i;
    }

    return -1;
}
```

La complexité temporelle en pire cas de cet algorithme correspond au maximum de comparaisons. On peut naturellement en déduire par majoration une borne en $O(np)$ mais on peut remarquer qu'il est assez difficile d'obtenir un exemple concret, ce qui fait penser que ce pire cas est *rare*.

**Remarque** Considérons la chaîne $s = aa...a = a^n$ qui contient $n$ fois la lettre $a$ et le motif $m = a^{p-1}b$ qui contient $p-1$ a et finit par un $b$. Dans l'algorithme, on va donc à chaque étape de la première boucle effectuer $p$ itérations dans la seconde avant de se rendre compte que le motif n'est pas présent en comparant $b$ et $a$. On a donc exactement $(n-p+1)p = \Theta(np)$ comparaisons et on retombe ainsi sur la complexité $O(np)$ pour ces exemples.

Ce qui va se passer dans une application usuelle de cet algorithme, c'est qu'au bout d'une ou deux comparaisons, on pourra invalider la position et passer à la suivante. On va alors avoir une complexité en $O(n+p)$ en considérant en plus la validation du motif dans le cas où il est présent. Ici $p \leq n$ donc $O(n+p) = O(n)$

mais c'est important de garder en tête cette complexité en $O(n+p)$ qu'on retrouvera car elle s'appliquera à des algorithmes où on effectue un prétraitement sur le motif pour l'appliquer ensuite sur plusieurs chaînes.

## I.3 Algorithme de Boyer-Moore

Dans un premier temps, on va présenter la variante usuelle de cet algorithme appelée algorithme de Boyer-Moore-Horspool. On présentera ensuite l'algorithme de Boyer-Moore en tant que tel.

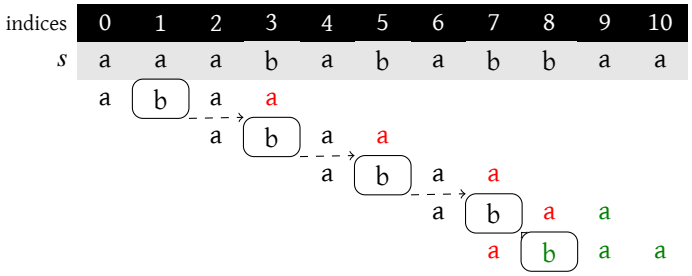
### I.3.i Principe de Boyer-Moore-Horspool

Le principe de l'algorithme de Boyer-Moore-Horspool est d'effectuer une recherche du motif comme précédemment mais en partant de la fin. On va alors tenter de trouver des suffixes de plus en plus grand du motif. Si on trouve ainsi le motif, on renvoie la position. Sinon, c'est qu'on a lu dans $s$ un mot de la forme $xm'$ où $m'$ est un suffixe strict de $m$ mais $xm'$ n'en est pas un. Si $x$ n'est pas présent dans $m$, alors on peut relancer la recherche juste après $x$ dans $s$. Si $x$ est présent dans $m$, on peut relancer la recherche en alignant ce caractère avec sa position la plus à droite dans $m$.

> **Remarque** Il faut tenir compte différemment du dernier caractère du motif, car il n'est pas utile de le réaligner. On considère alors, quand elle existe, l'occurrence précédente de ce caractère.

On obtient ainsi une stratégie de saut qui en cas d'échec relance la recherche plus loin.

Voici un premier exemple où on effectue une recherche de $abaa$ dans le mot $aababbbaa$. Cette stratégie a permis d'éviter une recherche inutile à partir de l'indice 1.




### I.3.ii Implémentation par table de saut

Pour réaliser ces sauts, on construit une table `droite` indexée par $\Sigma$ et telle que `droite[c]` indique l'indice de l'occurrence la plus à droite dans le motif $m$ du caractère $c$, en ignorant le dernier caractère du motif.

Ainsi, dans l'exemple précédent du motif `abaa`, on obtient la table suivante :

| `c` | 'a' | 'b' | 'c' | ... |
|---|---|---|---|---|
| `droite[c]` | 2 | 1 | $\emptyset$ | ... |

On a indiqué ici $\emptyset$ quand un caractère de $\Sigma$ n'est pas présent dans le motif, car il peut être présent dans $s$.

Cette table contient donc de l'ordre de $\Sigma$ éléments. On peut la réaliser par un tableau direct de taille $|\Sigma|$ étant donné un ordre d'énumération. On peut aussi la réaliser par un dictionnaire, ce qui est plus économe en espace si le motif contient peu de lettres différentes. On a choisit ici, pour des raisons pédagogiques, de considérer la numérotation ASCII naturelle associées au caractère de cette table.

OCaml
```ocaml
let taille_alphabet = 256

let calcule_droite (motif :string) : int option array =
    (* Calcule le tableau droite associé au motif *)
    let droite = Array.make taille_alphabet None in
    let p = String.length motif in
    for i = 0 to p-2 do
        let j = p-2-i in
        let c = motif.[j] in
        if droite.(Char.code c) = None
        then droite.(Char.code c) <- Some j
    done;
    droite
```

C
```c
int taille_alphabet = 256; // on pourrait passer par un define

/* Calcule le tableau droite associé au motif
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation
int *calcule_droite(char *motif)
{
    int *droite = malloc(sizeof(int) * taille_alphabet);
    int p = strlen(motif);

    memset(droite, -1, sizeof(int) * taille_alphabet);

    for (int i = 0; i < p-2; i++)
    {
        int j = p-2-i;
        char c = motif[j];
        if (droite[c] < 0)
            droite[c] = j;
    }

    return droite;
}
```

Afin d'implémenter l'algorithme lui-même, il est nécessaire de faire des calculs élémentaires mais précis pour déterminer le saut à effectuer. Si à la position $i + j$ on a un échec après avoir lu le caractère `c` où `droite[c]` contient la valeur $k$.

- Si $k = \emptyset$, c'est que le motif ne pourra jamais être trouvé tant que ce caractère `c` sera présent. On relance donc la recherche juste après à l'indice $i + j + 1$.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| a | b | b | a | a | d | a | c | a |
| d | a | c | | | | | | |
| | | | d | a | c | | | |

- Si $k \geq j$, cela signifie que `c` est présent plus à droite dans le motif, donc aligner cette occurrence ne permettrait pas d'avancer la recherche. Rien ne nous permet de savoir si `c` est présent ou non ailleurs dans le motif, on relance alors prudemment la recherche en $i + 1$.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| a | c | a | c | a | d | a | c | a |
| c | a | c | | | | | | |
| | c | a | c | | | | | |

- Sinon, on veut aligner ce `c` avec le caractère correspondant du motif, si on relance à l'indice $i'$, on souhaite ainsi avoir $i' + k = i + j$ donc $i' = i + j - k$.

OCaml

```
exception Difference
exception Trouve of int

let recherche_BMH (motif :string) (droite :int option array
    (chaine :string) : int option =
    let n = String.length chaine in
    let p = String.length motif in
    let i = ref 0 in
    try
        while !i <= n-p do
            try
                for j = p-1 downto 0 do
                    if chaine.[!i+j] <> motif.[j]
                    then begin
                        let dec =  match droite.(Char.code cha
                                | None -> j+1
                                | Some k when k < j -> j-k
                                | _ -> 1 in
                        i := !i + dec;
                        raise Difference
                    end
                done;
                raise (Trouve !i)
            with Difference -> ()
        done;
        None
    with Trouve k -> Some k
```

C

```
/* cherche motif dans chaine en utilisant la table de saut précalculée
 * droite. Renvoie l'indice de la première occurrence ou -1 s'il n'est p
 * présent */
int recherche_BMH(char *motif, int *droite, char *chaine)
{
    int n = strlen(chaine);
    int p = strlen(motif);

    for(int i = 0; i <= n-p; )
    {
        bool present = true;
        for (int j = p-1; j >= 0; j--)
        {
            if (chaine[i+j] != motif[j])
            {
                int k = droite[chaine[i+j]];
                present = false;
                if (k < 0)
                    i = i + j + 1;
                else if (k < j)
                    i = i + j - k;
                else
                    i = i + 1;
                break;
            }
        }
        if (present)
            return i;
    }

    return -1;
}
```

### I.3.ii.a Correction

Tout d'abord, remarquons que la terminaison ne pose pas de questions dans la mesure où on le nouvel indice auquel on relance la recherche est toujours strictement plus grand que le précédent.

Au sujet de la correction, il suffit de s'assurer que les indices écartés correspondent nécessairement à des recherches infructueuses. Sans perte de généralité, on peut supposer que la recherche s'effectue depuis le premier indice de $s$. Comme seul les sauts d'au moins deux indices sont ceux pour lesquels il est nécessaire de faire une preuve, cela correspond au cas où $m = m_1 c m_2 d m_3 x$ et $s = s_1 c m_3 s'$ avec $c, d$ et $x$ des caractères, $d \neq c$ et $c$ non présent dans $m_2 d m_3$.

Ainsi, toute recherche démarrant à des indices inférieurs échouera systématiquement, au plus tard, en comparant le caractère $c$ de $cm_3$ avec un caractère du motif dans $m_2 d m_3$ donc différent de $c$.

### I.3.ii.b Complexité

Tout d'abord, on remarque que la table de saut se construit en $O(\max(|m|, |\Sigma|))$ pour un motif $m$ sur un alphabet $\Sigma$.

Sans chercher à rentrer dans les détails, on peut raisonnablement penser **si l'alphabet contient assez de caractères** que les motifs auront peu de répétitions et qu'ainsi, les sauts seront presque toujours maximaux, ce qui permet d'obtenir de l'ordre de $\frac{n}{p}$ comparaisons où $n$ est la longueur de la chaîne et $p$ la longueur du motif.

Cependant, en pire cas, cet algorithme n'est pas meilleur que le précédent. Pour s'en convaincre, on va considérer un exemple proche de celui introduit poru l'algorithme naïf. Si on cherche $ba^{p-1}$ dans $a^n$ à l'indice $i$, il est nécessaire d'attendre de comparer au caractère $b$ pour constater un échec et devoir relancer l'algorithme à l'indice $i+1$. On va donc faire ici aussi $(n-p+1)p = \Theta(np)$ comparaisons.

La complexite temporelle en pire cas de Boyer-Moore-Horspool est donc de $O(np)$, même si, en pratique, elle est sous-linéaire.

> **Remarque** Si l'alphabet contient peu de caractères, ce qui est le cas en particulier du binaire, il y a de grandes chances qu'on soit dans ce cas pire cas. Ainsi, Boyer-Moore-Horspool n'est pas adapté pour ce type de texte.

## I.3.iii Principe de Boyer-Moore

Considérons le cas suivant de l'algorithme précédent : on cherche abbcabc dans cbacbbcabc.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| c | b | a | c | b | b | c | a | b | c |
| a | b | b | c | a | b | c | | | |
| | a | b | b | c | a | b | c | | |
| | | | a | b | b | c | a | b | c |

On remarque qu'en raison du fonctionnement de cet algorithme, on est forcé de faire de tous petits sauts et on est ramené à l'algorithme naïf. Cependant, après la première étape, on sait qu'on a lu un suffixe du motif bc qui est précédé d'un caractère a en sorte que bbc ne soit pas un suffixe du motif.

Il y a un autre endroit dans le motif où on peut trouver ⋆bc avec ⋆ un autre caractère que a. On pourrait donc relancer la recherche en alignant cette occurrence de bc avec celle qu'on vient de lire. Cela revient à sauter directement à la dernière étape dans cet exemple :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| c | b | a | c | b | b | c | a | b | c |
| a | b | b | c | a | b | c | | | |
| | | | a | b | b | c | a | b | c |

Pour pouvoir réaliser ce décalage, il est nécessaire de calculer une nouvelle table en parcourant le motif pour identifier de telles apparitions de suffixes.

On peut aller plus loin en considérant également le plus long préfixe du motif qui soit un suffixe du suffixe considéré. Par exemple, pour le motif bcabc on remarque que bc étant un préfixe, on peut effectuer un saut comme dans l'exemple suivant :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| c | a | a | b | c | c | b | b | c |
| b | c | a | b | c | | | | |
| | | | b | c | a | b | c | |

### I.3.iii.a Table des bons suffixes

■ **Note 3** Tout cela sera redéfini proprement plus tard dans le chapitre sur les langages. Je laisse cette partie en attendant pour que la présentation soit complète.

■

Il est nécessaire d'introduire des définitions précises pour formaliser la stratégie qu'on vient de présenter. Dans le contexte des langages, on parle plus souvent de mot que de chaîne de caractères, qui sont un type de données permettant de les représenter. Un mot sur l'alphabet $\Sigma$ est donc une suite finie $a_1 \dots a_n$ de lettres dans l'alphabet. On note $\mu$ l'unique mot vide, c'est-à-dire ne contenant aucune lettre. L'ensemble des mots sur $\Sigma$ est noté $\Sigma^*$. Si $u$ et $v$ sont des mots, $uv$ est le mot obtenu par concaténation.

**Remarque** $\Sigma^*$ muni de cette loi de composition a une structure proche de l'ensemble des entiers naturels $\mathbb{N}$ muni de l'addition :
- on a : $\forall u, v, w \in \Sigma^*, u(vw) = (uv)w = uvw$, on dit que la loi est associative ;
- elle possède un élément neutre $\mu$ : $\forall u \in \Sigma^*, \mu u = u\mu = u$.

On dit alors que $\Sigma^*$ est un **monoïde**. Cette structure très simple est cruciale en informatique.

**Définition I.1** Soit $u, v \in \Sigma^*$, on dit que $v$ est :
- un **suffixe** de $u$ s'il existe $w \in \Sigma^*$ tel que $u = wv$
- un **préfixe** de $u$ s'il existe $w \in \Sigma^*$ tel que $u = vw$

Lorsque $w \neq \mu$, on parle de suffixe ou de préfixe **propre**.
On dit que $v$ est un **bord** de $u$ lorsque $v$ est suffixe et préfixe propre de $u$.

**Exemple** Soit $u = abacaba$. $abac$ est un préfixe de $u$, $caba$ un suffixe et $aba$ un bord.

**Définition I.2** Soit $x = x_1 \dots x_n$ et $u, v$ deux suffixes **distincts** de $x$. On dit que $u$ et $v$ sont des suffixes **disjoints** quand on est dans l'un des cas suivants :

- $u = x$
- $v = x$
- $u \neq x, v \neq x$ et $x_{|x|-|u|} \neq x_{|x|-|v|}$.

Des suffixes disjoints sont donc des suffixes précédés par des lettres différentes dans $x$. On définit de même la notion de préfixes disjoints.

On considère un motif $x = x_0 \dots x_{n-1}$ et on va reprendre, en la précisant, la description précédente. Se faisant, on va construire une table `bonsuffixe` appelée la table des **bons suffixes** du motif $x$ et telle que, pour $i \in [\![0, n-1]\!]$, `bonsuffixe[i]` donne le nombre de positions dont on doit décaler le motif vers la droite pour relancer la recherche après la lecture du suffixe $x_{i+1} \dots x_{n-1}$.

Supposons qu'on vient de lire avec succès un suffixe propre $u$. Ainsi $x = x_0 \dots x_i u$ et on vient de lire dans la chaîne où on effectue la recherche $au$ avec $a \neq x_i$.

- Soit il existe un autre suffixe $buv$ de $x$ où $b \neq x_i$ et alors on appelle bon suffixe pour $u$ un tel suffixe de longueur minimale et on pose alors `bonsuffixe[i]`$= |v|$
- Sinon, on cherche $v$ de longueur minimale tel que $x$ soit un suffixe de $uv$ et on pose également `bonsuffixe[i]`$= |v|$.

On remarque que si $x$ est suffixe de $uv$ et qu'on a également $buv'$ suffixe de $x$, alors $|uv| = |u| + |v| \geq |x| \geq |buv'| \geq |u| + |v'|$ donc $|v| \geq |v'|$ ce qui permet de considérer le plus petit $v$ sur l'ensemble des cas.

### I.3.iii.b Table des suffixes

Afin de calculer efficacement `bonsuffixe` on va commencer par calculer la table des suffixes du motif, il s'agit de la table `suffixe` où `suffixe[i]` contient la longueur du plus long suffixe de $x$ de la forme $x_j \dots x_i$. Ainsi, si on note $S_i$ les suffixes de cette forme, on a :

$$\texttt{suffixe}[i] = \begin{cases} 0 & \text{si } S_i = \emptyset \\ \max\{\ |s| \mid s \in S_i\ \} & \text{sinon} \end{cases}$$

Nécessairement, `suffixe[n-1]=n` car $x$ convient.

**Exemple** Pour $x = bcabc$ on a :

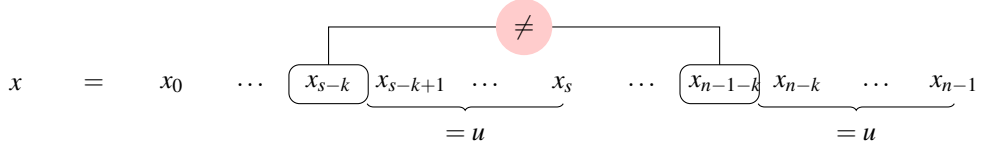
| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| `suffixe[i]` | 0 | 2 | 0 | 0 | 5 |

et pour $x = abbabba$ :

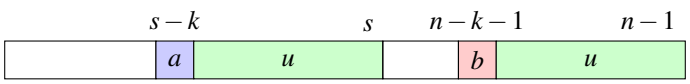
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| `suffixe[i]` | 1 | 0 | 0 | 4 | 0 | 0 | 7 |

Il est possible de construire `suffixe` avec un simple parcours linéaire en tirant partie de l'information déjà calculée. Pour cela, on va remplir `suffixe` de droite à gauche.

A tout moment, on va conserver le meilleur suffixe rencontré, c'est-à-dire celui pour lequel on est allé le plus loin à gauche avant d'avoir un échec de comparaison. On note $s$ la position la plus à droite de ce suffixe et $k$ sa longueur, il s'agit donc de $u = x_{s-k+1} \dots x_s$ et il y a eu un échec de comparaison en $x_{s-k}$. Par définition de `suffixe` on a `suffixe[s]`$= k$. Le mot $x$ s'écrit alors :

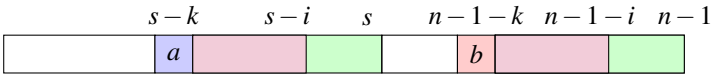



Ce qu'on peut représenter schématiquement ainsi :

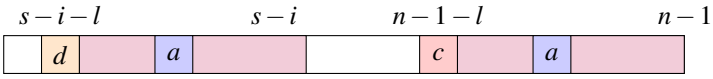


Maintenant, on considère la position $s-i$ où $s > s-i > s-k$, cela signifie qu'on cherche un suffixe depuis une position interne au mot $u$ de gauche. Le point clé permettant d'obtenir un algorithme linéaire est de remarquer que la situation est la même que dans le mot $u$ de droite. Or, comme on procède de gauche à droite, on a déjà calculé la valeur correspondante `suffixe[n-1-i]`. Là, on a deux cas :

- soit quand on a cherché le plus grand suffixe à partir de $n-i$, on s'est heurté à une erreur de comparaison à la position $n-k$. Dans ce cas, on a `suffixe[n-1-i]`$= k-i$ et on peut regarder, en partant de la position $s-k$, si on peut prolonger le suffixe finissant à la position $s-i$.




Pour effectuer ce prolongement, il suffit de comparer, caractère par caractère, vers la gauche en partant de la position $s-k$. On aboutira alors à une nouvelle position du suffixe finissant le plus à gauche qui finira en $s-i$.




Remarquons qu'il n'est pas nécessaire que $s-i-l \neq s-k$. C'est-à-dire que même si $a$ ne permet pas de prolonger le suffixe déduit de la position $n-i$, on considère tout de même que la nouvelle position de référence est $s-i$. On en déduit également la valeur `suffixe[s-i]`$= l$.

- soit `suffixe[n-1-i]`$= p \neq k-i$ et alors
  - ⋆ soit $p < k-i$, on a alors poru ce suffixe un échec dans $u$, ce qui limite de la même manière la valeur en $s-i$ : `suffixe[s-i]`$= p$.
  - ⋆ soit $p > k-i$, donc on doit avoir un $b$ après avoir le suffixe dans $u$ depuis $s-i$ pour le prolonger, or, c'est impossible car il y a un $a \neq b$. Ainsi, le suffixe est limité par $u$ : `suffixe[s-i]`$= k-i$.

Il reste à traiter le cas où $s-i \leq s-k$, ce qui revient à considérer qu'on a dépassé le précédent suffixe pouvant apporter une information. On procède donc naïvement pour trouver le plus grand suffixe depuis cette position.

On en déduit l'implémentation suivante :

OCaml

```
let calcule_suffixe (x :string) : int array =
    (* Prend en entrée un mot non vide x et renvoie son tab
    let n = String.length x in
    let suffixe = Array.make n (-1) in
    suffixe.(n-1) <- n ;
    let plus_a_gauche = ref (n-1) in
    let depart = ref (-1) in

    for j = n-2 downto 0 do
        if !plus_a_gauche < j
            && suffixe.(n-1- !depart+j) <> j - !plus_a_gauch
        then suffixe.(j) <- min suffixe.(n-1- !depart+j) (j - !p
        else begin
            plus_a_gauche := min j !plus_a_gauche ;
            depart := j ;
            while !plus_a_gauche >= 0
                && x.[!plus_a_gauche] = x.[n-1-j + !plus_a_
                plus_a_gauche := !plus_a_gauche - 1
            done ;
            suffixe.(j) <- !depart - !plus_a_gauche
        end
    done ;
    suffixe
```

```
/* Calcule le tableau suffixe associé au mot x
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation
int *calcule_suffixe(char *x)
{
    int n = strlen(x) ;
    int *suffixe = malloc(sizeof(int) * n) ;
    int plus_a_gauche = n-1 ;
    int depart = -1 ;

    memset(suffixe, -1, sizeof(int) * n) ;
    suffixe[n-1] = n ;

    for (int j = n-2 ; j >= 0 ; j--)
    {
        if (plus_a_gauche < j
            && suffixe[n-1-depart+j] != j-plus_a_gauche)
            suffixe[j] = MIN(suffixe[n-1-depart+j], j-plus_a_gauche) ;
        else {
            plus_a_gauche = MIN(plus_a_gauche, j) ;
            depart = j ;
            while (plus_a_gauche >=0
                    && x[plus_a_gauche] == x[n-1-j+plus_a_gauche]
                plus_a_gauche-- ;
            suffixe[j] = depart - plus_a_gauche ;
        }
    }

    return suffixe ;
}
```

**Remarque** Cette implémentation est optimisée par rapport à la description précédente en calculant directement sans introduire $i$ ou $k$, et en fusionnant deux cas qui reviennent à dupliquer du code.

On donne ici le code maladroit qui correspond à la traduction exacte de la description précédente :

OCaml

```
let calcule_suffixe_rep (x :string) : int array =
    (* Prend en entrée un mot non vide x et renvoie son tab
    let n = String.length x in
    let suffixe = Array.make n 0 in
    suffixe.(n-1) <- n ;
    let plus_a_gauche = ref (n-1) in
    let depart = ref (-1) in

    for j = n-2 downto 0 do
        if !plus_a_gauche < j
        then begin
            (* on a j = depart - i avec *)
            let i = !depart - j in
            (* et plus_a_gauche = depart - k avec *)
            let k = !depart - !plus_a_gauche in
            if suffixe.(n-1-i) <> k-i
            then suffixe.(j) <- min suffixe.(n-1-i) (k-i)
            else begin
                depart := j ;
                while !plus_a_gauche >= 0
                    && x.[!plus_a_gauche] = x.[n-1-j + !plus_
                    plus_a_gauche := !plus_a_gauche - 1
                done ;
                suffixe.(j) <- !depart - !plus_a_gauche
            end
        end else begin
            plus_a_gauche := j ;
            depart := j ;
            while !plus_a_gauche >= 0
                && x.[!plus_a_gauche] = x.[n-1-j + !plus_a_
                plus_a_gauche := !plus_a_gauche - 1
            done ;
            suffixe.(j) <- !depart - !plus_a_gauche
        end
    done ;
    suffixe
```

C

```
/* Calcule le tableau suffixe associé au mot x
 * le tableau renvoyé a été alloué, il devra être libéré après utilisati
int *calcule_suffixe_rep(char *x)
{
    int n = strlen(x) ;
    int *suffixe = malloc(sizeof(int) * n) ;
    int plus_a_gauche = n-1 ;
    int depart = -1 ;

    memset(suffixe, -1, sizeof(int) * n) ;
    suffixe[n-1] = n ;

    for (int j = n-2 ; j >= 0 ; j--)
    {
        if (plus_a_gauche < j)
        {
            // on a j = depart - i avec
            int i = depart - j ;
            // et plus_a_gauche = depart -k avec
            int k = depart - plus_a_gauche ;

            if (suffixe[n-1-i] != k-i)
                suffixe[j] = MIN(suffixe[n-1-i], k-i) ;
            else {
                depart = j ;
                while (plus_a_gauche >=0
                        && x[plus_a_gauche] == x[n-1-j+plus_a_gau
                    plus_a_gauche-- ;
                suffixe[j] = depart - plus_a_gauche ;
            }
        } else {
            plus_a_gauche = depart = j ;
            while (plus_a_gauche >=0
                    && x[plus_a_gauche] == x[n-1-j+plus_a_gauch
                plus_a_gauche-- ;
            suffixe[j] = depart - plus_a_gauche ;
        }
    }

    return suffixe ;
}
```

Python

```python
def calcule_suffixe_rep(x) :
    '''Prend en entrée un mot non vide x et renvoie son tableau de suffixe'''
    n = len(x)
    suffixe = [ None ] * n
    suffixe[n-1] = n
    plus_a_gauche = n-1
    depart = None

    for j in reversed(range(0,n-1)) :
        if plus_a_gauche < j :
            # on a j = depart - i avec
            i = depart - j
            # et plus_a_gauche = depart - k avec
            k = depart - plus_a_gauche
            if suffixe[n-1-i] != k-i :
                suffixe[j] = min(suffixe[n-1-i],k-i)
            else :
                depart = j
                while plus_a_gauche >= 0 \
                        and x[plus_a_gauche] == x[n-1-j+plus_a_gauche] :
                    plus_a_gauche = plus_a_gauche - 1
                suffixe[j] = depart - plus_a_gauche
        else :
            plus_a_gauche = depart = j
            while plus_a_gauche >= 0 \
                    and x[plus_a_gauche] == x[n-1-j+plus_a_gauche] :
                plus_a_gauche = plus_a_gauche - 1
            suffixe[j] = depart - plus_a_gauche

    return suffixe
```

On remarque que dans ce code, `plus_a_gauche` ne peut que diminuer, on effectue donc au plus $n$ itérations dans la boucle `while` pour tout l'algorithme. Donc, en considérant la boucle `for`, on effectue au plus $2n$ comparaisons de caractères : au plus une pour chaque itération de la boucle `for` pour voir si on entre dans le `while`, puis en tout au plus $n$ avant de sortir du `while`.

L'algorithme qu'on a obtenue est bien linéaire en $|x|$.

### I.3.iii.c Obtention de `bonsuffixe` à partir de `suffixe`

On reprend maintenant le calcul de `bonsuffixe[i]` dans le mot $x = x_0 \dots x_{n-1}$.

On cherche à obtenir des suffixe de la forme $buv$ de $x$ où $b \neq x_i$ et $u = x_{i+1} \dots x_{n-1}$ est un suffixe de $x$. Mais si `suffixe[k]`$= n-1-i$ cela signifie que ce suffixe est exactement $u$ et qu'il est soit préfixe, soit précédé d'une lettre différente de $x_i$, sinon $n-1-i$ ne serait pas maximal.

On a donc

$$\begin{aligned} \texttt{bonsuffixe}[n-1-i] &= \min\{\, n-1-k \mid \texttt{suffixe}[k] = n-1-i \,\} \\ &= n-1-\max\{\, k \mid \texttt{suffixe}[k] = n-1-i \,\} \end{aligned}$$

On remarque qu'on peut ainsi faire croitre $k$ et poser :

$$\texttt{bonsuffixe}[n-1-\texttt{suffixe}[k]] = n-1-k$$

On a aura alors naturellement, à la fin de la boucle, la valeur minimale placée en dernier.

Reste à considérer les valeurs non remplies ainsi dans le tableau `bonsuffixe`. Elles correspondent aux positions $i$ telles qu'il n'existe pas de suffixe de la forme $buv$. On doit donc chercher un mot $uv$ de longueur minimale dont $x$ est suffixe. Mais $u$ étant un suffixe de $x$, cela revient à considérer les bords de $x$. La table `suffixe` permet également de détecter les bords : si $x_0 \dots x_k$ est un bord c'est que `suffixe[k]`$= k+1$.

Soit $k < n-1$ maximal vérifiant cette condition. Pour tout $u = x_{i+1} \dots x_n$ suffixe de $x$, pour qu'il ait $x_0 \dots x_k$ comme suffixe, il faut qu'il soit strictement plus long (sinon on est dans le cas précédent), donc que $n-i > k+1 \iff i < n-1-k$. Dans ce cas, $x$ est alors suffixe de $uv$ où $v = x_{k+1} \dots x_{n-1}$ donc $|v| = n-1-k$. Les $k$ plus petits ne pourront alors que faire augmenter $|v|$, on peut ainsi poser `bonsuffixe[i]`$= n-1-k$.

On en déduit un remplissage en parcourant les $k$ dans l'ordre décroissant de $n-2$ à $0$, tout en maintenant l'indice $i$ de la prochaine valeur à remplir dans `bonsuffixe`. Dès qu'on détecte un bord, on place $n-1-k$ jusqu'à ce que $i \geq n-1-k$.

En sortie de boucle, il est possible que $i < n$ donc qu'il reste des valeurs à remplir. On remarque dans ce cas là que pour que $x$ soit un suffixe de $uv$ il faut que $v = x$. On a donc pour ces valeurs restantes `bonsuffixe[i]`$= n$.

Comme ce second cas est toujours plus long que le premier quand les deux se produisent en $i$, on implémente successivement les remplissages de sorte à obtenir la valeur minimum. On en déduit le programme suivant :

OCaml
```
let calcule_bonsuffixe (x :string) : int array =
    (* Prend en entrée un mot non vide x et renvoie son tab
    let n = String.length x in
    let suffixe = calcule_suffixe x in
    let bonsuffixe = Array.make n n in

    let suivant = ref 0 in
    for k = n-2 downto 0 do
        if suffixe.(k) = k+1 (* c'est un bord *)
        then begin
            for i = !suivant to n-2-k do
                bonsuffixe.(i) <- n-1-k
            done;
            suivant := n-1-k
        end
    done;

    for k = 0 to n-2 do
        bonsuffixe.(n-1-suffixe.(k)) <- n-1-k
    done;

    bonsuffixe
```

C
```
/* Calcule le tableau suffixe associé au mot x
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation
int *calcule_bonsuffixe(char *x)
{
    int n = strlen(x);
    int *suffixe = calcule_suffixe(x);
    int *bonsuffixe = malloc(sizeof(int) * n);
    int suivant = 0;

    memset(bonsuffixe, n, sizeof(int) * n);

    for (int k = n-2; k >= 0; k--)
    {
        if (suffixe[k] == k+1) // bord
        {
            for (int i = suivant; i < n-1-k; i++)
                bonsuffixe[i] = n-1-k;
            suivant = n-1-k;
        }
    }

    for (int k = 0; k < n-1; k++)
        bonsuffixe[n-1-suffixe[k]] = n-1-k;

    free(suffixe);

    return bonsuffixe;
}
```

Il est facile de constater que cet algorithme est de complexité temporelle linéaire en $|x|$.

### I.3.iii.d Algorithme de Boyer-Moore

On incorpore naturellement la table précédente à l'algorithme de Boyer-Moore en choisissant le meilleur décalage entre cette table et la stratégie précédente.

OCaml

```
let recherche_BM (motif :string)
    (droite :int option array) (bonsuffixe :int array)
    (chaine :string) : int option =
    let n = String.length chaine in
    let p = String.length motif in
    let i = ref 0 in
    try
        while !i <= n-p do
            try
                for j = p-1 downto 0 do
                    if chaine.[!i+j] <> motif.[j]
                    then begin
                        let dec =  match droite.(Char.code cha
                                | None -> j+1
                                | Some k when k < j -> j-k
                                | _ -> 1 in
                        i := !i + max dec bonsuffixe.(j) ;
                        raise Difference
                    end
                done ;
                raise (Trouve !i)
            with Difference -> ()
        done ;
        None
    with Trouve k -> Some k
```

```
/* cherche motif dans chaine en utilisant les tables de saut précalcul
 * droite et bonsuffixe. Renvoie l'indice de la première occurrence ou
 * présent */
int recherche_BM(char *motif, int *droite, int *bonsuffixe, char *ch
{
    int n = strlen(chaine) ;
    int p = strlen(motif) ;

    for(int i = 0 ; i <= n-p ; )
    {
        bool present = true ;
        for (int j = p-1 ; j >= 0 ; j--)
        {
            if (chaine[i+j]  != motif[j])
            {
                int k = droite[chaine[i+j]] ;
                int dec = 1 ;
                present = false ;
                if (k < 0)
                    dec = j + 1 ;
                else if (k < j)
                    dec = j - k ;
                i = i + MAX(dec, bonsuffixe[j]) ;
                break ;
            }
        }
        if (present)
            return i ;
    }

    return -1 ;
}
```

Supposons que le motif est de longueur $p$, que la chaîne dans laquelle on recherche est de longueur $n$ et que la taille de l'alphabet est une constante indépendante des entrées. La première partie de l'algorithme consiste à construire les tables de sauts, comme on l'a vu, elle est en complexité en temps et en espace en pire cas en $O(p)$.

On admet que l'algorithme Boyer-Moore complet, étant donné les deux tables de saut et d'autres modifications mineures non présentées ici, est en complexité temporelle en pire cas en $O(n)$.

Il est assez raisonnable de penser que soit $p \leq n$ quand on effectue une recherche, soit on compte chercher un même motif dans plusieurs textes et on réutilise ainsi les tables de sauts. Il n'est donc pas forcément très pertinent de parler de la complexité globale de l'algorithme, mais lorsqu'on le fait, on dit qu'elle est en $O(p+n)$. On rappelle ici le rôle de l'addition dans les complexités qui fait référence à la succession de deux traitements, un en $O(p)$ suivi d'un en $O(n)$.

## I.4 Algorithme de Rabin-Karp

### I.4.i Principe

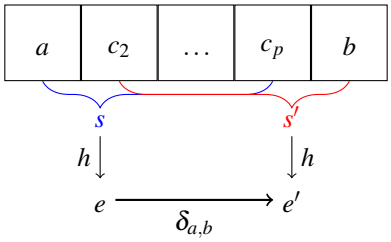
L'algorithme de Rabin-Karp est un algorithme de recherche d'un motif dans un texte qui utilise une notion d'empreinte pour déterminer, en temps constant, si il est probable que la position actuelle corresponde à une occurrence du motif.

Pour cela, si on cherche un motif de longueur $p$ sur l'alphabet $\Sigma$, on considère une **fonction de hachage** $h : \Sigma^p \to X$. Les éléments de l'ensemble $X$ sont appelés des empreintes et on suppose que l'égalité entre deux empreintes se vérifie en temps constant contrairement à l'égalité dans $\Sigma^p$ qui se vérifie en $O(p)$ dans le pire des cas. Le plus souvent, on choisit pour $X$ un type entier machine.

■ **Note 4** Sûrement mettre ici des renvois vers la partie portant le plus sur la notion de fonction de hachage pour la définition la plus complète.

■

Bien qu'il soit normalement aussi coûteux de calculer l'image par $h$ d'une sous-chaîne de longueur $p$ que de tester l'égalité entre cette sous-chaîne et le motif, le point essentiel de l'algorithme de Rabin-Karp est d'utiliser une fonction de hachage permettant un calcul incrémental en temps constant :




Ici, on considère donc, pour $a, b \in \Sigma$, une fonction de mise à jour $\delta_{a,b} : X \to X$ telle que pour tout $c_2, \ldots, c_p \in \Sigma$ on ait $\delta_{a,b}(h(ac_2 \ldots c_p)) = h(c_2 \ldots c_p b)$.

L'algorithme de Rabin-Karp procède alors ainsi pour chercher $m$ de longueur $p$ dans la chaîne $s = c_0 \ldots c_{n-1}$ où $n \geq p$ :

- calcul de $e_m = h(m)$ et $e = h(c_0..c_{p-1})$.
- Pour $i$ allant de 0 à $n - p$ :
  - ⋆ Si $e_m = e$, on renvoie un succès pour la recherche à la position $i$ si $m = c_i \ldots c_{i+p-1}$
  - ⋆ si $i < n - p$ on met à jour l'empreinte $e \leftarrow \delta_{c_i, c_{i+p}}(e)$.

La complexité temporelle liée à la gestion des empreintes est donc en $O(n + p) = O(n)$ car $n \geq p$. Par contre, pour calculer la complexité liée à la recherche $m = c_i \ldots c_{i+p-1}$, il est nécessaire d'estimer la proportion de faux positifs, c'est-à-dire de positions $i$ telles que $e_m = e$ mais $m \neq c_i \ldots c_{i+p-1}$. On va voir dans la partie suivante qu'on peut supposer qu'elle est négligeable, ce qui permet de considérer que l'algorithme de Rabin-Karp est linéaire.

### I.4.ii Choix d'une fonction de hachage

Réaliser une bonne fonction de hachage est une question très complexe qui dépasse le cadre du cours d'informatique de MPI. Cependant, il est possible de réaliser ici une fonction de hachage répondant aux contraintes de Rabin-Karp assez facilement.

Pour cela, on considère que les caractères sont des entiers compris entre 0 et 255, ce qui correspond au type des caractères non signés sur un octet. On peut alors identifier une chaîne de longueur $p$ avec un nombre entre 0 et $r^p - 1$ où $r = 2^8$, on note ainsi

$$P(c_0 \ldots c_{p-1}) = \sum_{i=0}^{p-1} c_i r^{p-1-i} = c_0 r^{p-1} + c_1 r^{p-2} + \cdots + c_{p-1}$$

On considère de plus un entier premier $q$ et on pose $h(s) = P(s) \mod q$ c'est-à-dire le reste de $P(s)$ dans la division euclidienne par $q$. On peut ainsi définir $\delta_{a,b}(e) = (r(e - ar^{p-1}) + b) \mod q$.

Si on précalcule $r^{p-1} \mod q$ il suffit d'un nombre d'opération constant, et indépendant de $p$, pour calculer la nouvelle empreinte à l'aide de $\delta_{a,b}$.

Le point essentiel est alors de déterminer un nombre premier $q$ tel qu'il soit peu probable d'obtenir des faux positifs. Une analyse mathématique permet d'affirmer que chaque élément de $[|0; q-1|]$ a de l'ordre de $\frac{r^p}{q}$ antécédents par $h$. Ainsi, si on choisit deux chaînes aléatoirement dans $\Sigma^p$, il y aura collision avec probabilité proche de $\frac{1}{q}$. En considérant $q$ proche de la taille maximale pour le type entier considéré, on minimise donc cette probabilité.

**Remarque** On peut également s'intéresser à des nombres $q$ pour lesquels le modulo soit rapide à calculer. Un exemple classique est $q = 2^{31} - 1$ car on peut déduire la division euclidienne de $a$ par $q$ de l'écriture de $a$ en base $2^{31}$. En effet, si $a = \sum_{k=0}^{n} a_k 2^{31k}$ comme $2^{31} - 1 | 2^{31k} - 1$ pour $k \geq 1$, on a $2^{31k} \equiv 1\ [q]$ et ainsi $a \equiv \sum_{k=0}^{n} a_k [q]$. On remarque que $a_k = (a >> 31k) \& 2^{31}$, on a alors soit $a_k < q$ et alors $a_k \mod q = a_k$, soit $a_k = q$ et $a_k \mod q = 0$. Il suffit donc de faire un masquage pour obtenir directement $a_k \mod q = (a >> 31k) \& q$.

On obtient alors le programme suivant :

OCaml
```
let rec fastmod a =
    let s = ref 0 in
    let x = ref a in
    let q = 0x7fffffff in
    while !x > 0 do
        s := !s + (!x land q);
        x := !x lsr 31
    done;
    if !s > q
    then fastmod !s
    else if !s = q
    then 0
    else !s
```

C
```
int64_t fastmod(int64
{
    int64_t s = 0;
    const int64_t q =

    while (a > 0)
    {
        s = s + a & q
        a = a » 31;
    }

    if (s > q)
        return fastmod
    if (s == q)
        return 0;
    return s;
}
```

Python
```
def fastmod(a) :
    s = 0
    q = 0x7fffffff
    while a > 0 :
        s = s + a & q
        a = a » 31
    if s > q :
        return fastmod(s)
    elif s == q :
        return 0

    return s
```

Le programme suivant implémente naïvement les calculs de $h$ et de $\delta_{a,b}$ :

OCaml
```
let hash r q s =
    let p = ref 1 in
    let e = ref 0 in
    for i = String.length s - 1 downto 0 do
        e := (!p * (Char.code s.[i]) + !e) mod q;
        p := (r * !p) mod q
    done;
    !e

let delta r q rp a b e = (* rp est r^(p-1) mod q *)
    (r * (e - rp * (Char.code a)) + Char.code b) mod q
```

C
```
int64_t hash(int64_t r, int64_t q, char *s, int n)
{
    int64_t p = 1;
    int64_t e = 0;

    for (int i = n-1; i >= 0; i--)
    {
        e = (p * s[i] + e) % q;
        p = (r * p) % q;
    }

    return e;
}


int64_t delta(int64_t r, int64_t q, int64_t rp,
        char a, char b, int64_t e)
{
    return (r * (e - rp * a) + b) % q;
}
```

### I.4.iii Implémentation

Une implémentation directe de l'algorithme de Rabin-Karp est donnée dans le programme qui suit. On se sert ici du caractère paresseux du && pour n'effecuter le test coûteux d'égalité des chaînes qu'en cas d'égalité des empreintes.

OCaml

```
exception Trouve of int

let rabin_karp m s =
    let n = String.length s in
    let p = String.length m in
    let r = 256 in
    let q = 0x7fffffff in (* 2^(31)-1 *)
    let rp = pow r (p-1) q in
    let me = hash r q m in
    let e = ref (hash r q (String.sub s 0 p)) in
    try
        for i = 0 to n-p+1 do
            if me =!e && m = String.sub s i p
            then raise (Trouve i);
            if i+p < n then e := delta r q rp s.[i] s.[i+p] !e
        done; None
    with Trouve k -> Some k
```

C

```
int rabin_karp(char *m, char *s)
{
    const int64_t r = 256;
    const int64_t q = 0x7fffffff;
    const int p = strlen(m);
    const int n = strlen(s);
    const int64_t rp = powmod(r,p-1,q);
    const int64_t me = hash(r,q,m,p);
    int64_t e = hash(r,q,s,p);
    for (int i=0; i <n-p+1; i++)
    {
        if (me == e && strncmp(m,(s+i),p) == 0)
            return i;
        if (i+p < n)
            e = delta(r,q,rp,s[i],s[i+p],e);
    }
    return -1;
}
```

Si on suppose qu'il est improbable d'obtenir un faux positif, il est possible de renvoyer un succès dès que les empreintes sont égales. L'avantage d'une telle version est alors d'être un algorithme sans retour sur les données. C'est-à-dire qu'il n'est pas nécessaire de garder en mémoire ou de réaccéder à un caractère.

### I.4.iv L'algorithme originel de Rabin et Karp

Si on regarde l'article originel de Rabin et Karp décrivant cette méthode, on peut être étonné du fait que la méthode précédemment décrite était considérée comme déjà connue dans la littérature par les auteurs. En fait, ce qu'ils décrivent et annoncent comme étant novateur est l'utilisation d'un algorithme probabiliste en choisissant aléatoirement une fonction de hachage à chaque lancement de l'algorithme. En pratique, il s'agit de choisir aléatoirement un nombre premier $q$ parmi un ensemble précalculé de nombres premiers.

L'algorithme que l'on vient de décrire a un pire cas qui est très improbable car on considère que la probabilité d'un faux positif est à peu près de $1/q$, donc moins de $5.10^{-10}$ pour $q = 2^{31} - 1$. Le problème ici est la notion de probabilité sur les entrées : est-on certain que l'algorithme recevra une entrée choisie uniformément ? Rabin et Karp parlent d'un *adversaire intelligent* qui aurait connaissance de la fonction de hachage choisie pour produire des entrées en pire cas. On pourrait ainsi imaginer une *attaque* sur serveur effectuant une recherche avec Rabin-Karp suite à l'entrée d'un utilisateur. Un adversaire pourrait construire une entrée en pire cas et tenter de surcharger le serveur en l'effectuant de manière répétée.

Pour bien mettre en lumière ce phénomène, nous allons ici construire, dans un cas très simple de fonction de hachage, une telle chaîne problématique. Pour cela, considérons la fonction de hachage précédemment décrite dans le cas de motif de taille 2, avec $\Sigma$ contenant les lettres de a à z, $r = 26$ et $q = 17$. On considère une recherche du motif `aa` dont l'empreinte est `0`, la même que celle des chaînes `ar` et `ra`. On peut donc considérer la chaîne `arar...ar` qui produira un faux positif à chaque étape.

**Remarque** Détail des calculs. Ici on associe à a la valeur 0, ..., à z la valeur 25. On a donc

$$h(aa) = (0 \times 26 + 0) \mod 17 = 0$$

$$h(ar) = (0 \times 26 + 17) \mod 17 = 0$$

$$h(ra) = (17 \times 26 + 0) \mod 17 = 0$$

L'empreinte reste ainsi nulle tout au long de l'algorithme de Rabin-Karp et on a un faux positif à chaque itération.

# II Compression

## II.1 Principe

On s'intéresse ici à la compression parfaite d'un texte, c'est-à-dire, étant donné un alphabet fixé $\Sigma$, qu'on cherche à réaliser un couple de fonctions $comp, dec : \Sigma^* \to \Sigma^*$ telles que :

- pour tout mot $m \in \Sigma^*$, $dec(comp(m)) = m$
- pour la plupart des mots $m$ qui correspondent aux données qu'on cherche à compresser, $|comp(m)| < |m|$.

**Remarque** Le fait que $dec \circ comp = id_{\Sigma^*}$ implique, comme on a pu le voir dans le cours de mathématique, que $comp$ est injective : deux mots différents ont nécessairement des images distinctes.

Si $A$ et $B$ sont deux ensembles finis tels que $|A| < |B|$, il n'existe pas de fonction injective de $A$ dans $B$. Ainsi, si on note $L_n$ les mots de $\Sigma^*$ de longueur au plus $n$, il ne peut exister de fonction injective de $L_n$ dans $L_m$ où $n < m$.

Autrement dit : il est impossible d'espérer pouvoir compresser toutes les données de $L_n$. Si certains mots vont diminuer de longueur après compression, d'autres vont nécessairement augmenter.

Tout l'enjeu des algorithmes de compression parfaites est alors de diminuer les longueurs des mots qui nous intéressent. Par exemple, si on s'intéresse à des mots issus de textes en français, il est plus important d'arriver à compresser une phrase comme `"ceci est un texte"` plutôt qu'une suite de caractères non signifiante comme `"c2#$%1ajdn //@#3d!fn"`.

## II.2 Algorithme d'Huffman

La définition et la construction de l'arbre de Huffman ont été présentées au paragraphe Algorithme d'Huffman - Compression. On va s'intéresser ici au processus complet permettant de compresser et décompresser des fichiers avec cet algorithme.

### II.2.i Calcul de la table d'occurences

Par souci d'efficacité, on calcule une table d'occurences pour l'ensemble des valeurs d'octets entre 0 et 255. Il suffit alors de parcourir le fichier pour incrémenter les valeurs correspondant aux octets lus.

OCaml
```
let table_occurrences nomdefichier =
    let fichier = open_in_bin nomdefichier in
    let occurrences = Array.make 256 0 in
    begin
     try
        while true do
            let c = input_byte fichier in
            occurrences.(c) <- occurrences.(c) + 1;
        done
      with End_of_file -> ()
    end;
    close_in fichier;
    occurrences
```

C
```
ERROR : src/algorithmique/../../snippets/algorithmique/huffman.c
```

### II.2.ii Sérialisation de l'arbre de Huffman

Afin de décompresser, il est nécessaire de connaître l'arbre de Huffman donnant le code préfixe. Pour cela, il faut stocker cet arbre dans le fichier comme une série d'octet, on parle de *sérialisation*. Cette notion sera prolongée dans le chapitre FIXME.

On choisit ici la représentation récursive `repr(a)` de l'arbre `a` définie ainsi :

- Si `a = Noeud(g,d)`, `repr(a) = 0 repr(g) repr(d)`
- Si `a = Feuille(c)`, `repr(a) = 1 c`.

**Exemple** Si `a = Noeud(Feuille 42, Noeud(Feuille 16, Feuille 64))`, on obtient la suite d'octets : `0 1 42 0 1 16 1 64`.

La lecture et l'écriture de la séralisation s'effectue alors simplement par récurrence :

```OCaml
let rec output_arbre f a =
    match a with
    | Noeud (x, y) -> begin
        output_byte f 0 ;
        output_arbre f x ;
        output_arbre f y
    end
    | Feuille c -> begin
        output_byte f 1 ;
        output_byte f c
    end

let rec input_arbre f =
    let code = input_byte f in
    match code with
    | 0 -> let g = input_arbre f in
        let d = input_arbre f in
        Noeud(g,d)
    | _ -> Feuille (input_byte f)
```

```C
ERROR : src/algorithmique/../../snippets/algorithmique/huffman.c
```

### II.2.iii Écriture dans un fichier un bit à la fois

■ **Note 5** À déplacer éventuellement dans une partie spécifique sur la gestion de fichiers. ■

Le propre de l'algorithme de Huffman est d'associer à chaque caractère un codage binaire de longueur variable. Afin de pouvoir écrire ce codage dans un fichier, il est nécessaire de grouper les bits par paquet de huit (octet en français, byte en anglais).

Ainsi, par exemple, si on a le codage suivant :

| c | `'a'` | `'b'` | `'c'` |
|---|---|---|---|
| code(c) | 0 | 100 | 101 |

et qu'on doit encoder `"abbaca"`, on obtient le mot binaire `010010001010` qu'on complète avec des 0 à la fin et qu'on sépare en octets : `01001000 10100000`. On obtient donc les deux octets, convertis en décimal, 72 et 160. Ce sont eux qu'on va écrire dans un fichier.

Une technique usuelle pour cela est de garder un accumulateur qui correspond à l'octet en train d'être construit ainsi que le nombre de bits qui ont été accumulé. Dès qu'on accumulé 8 bits, on peut construire l'octet, l'écrire dans le fichier, puis reinitialiser ces variables.

Quand on rajoute un bit $b$ à l'accumulateur, on veut passer de $acc = b_1 \dots b_k$ à $b_1 \dots b_k b = 2acc + b$.

On en déduit l'implémentation assez directe suivante :

```OCaml
type out_channel_bits = {
    o_fichier : out_channel;
    mutable o_accumulateur : int;
    mutable o_bits_accumules : int
}

let open_out_bits fn =
    { o_fichier = open_out_bin fn; o_accumulateur = 0; o_bits_accumules = 0 }

let output_bit f b =
    if f.o_bits_accumules = 8
    then begin
        output_byte f.o_fichier f.o_accumulateur;
        f.o_accumulateur <- 0;
        f.o_bits_accumules <- 0
    end;
    if b
    then f.o_accumulateur <- f.o_accumulateur + 1 lsl f.o_bits_accumules;
    f.o_bits_accumules <- 1 + f.o_bits_accumules
```

```C
ERROR : src/algorithmique/../../snippets/algorithmique/bitpacking
```

Il reste à traiter la question des zéros finaux, si l'accumulateur contient $k$ bits au moment de la fermeture du fichier, où $0 < k < 8$, il faut ajouter $8 - k$ zéros. On appelle cela du *padding* de l'anglais pour rembourrage. Ici, cela correspond à faire un décalage binaire vers la gauche d'autant (*shift left* en anglais). Comme il sera nécessaire de se souvenir que ces zéros ne sont pas signifiants à la lecture, on rajoute un octet final contenant cette valeur $k$.

On obtient alors la fonction de fermeture de fichier suivante :

```OCaml
let close_out_bits f =
    if f.o_bits_accumules = 0
    then output_byte f.o_fichier 0
    else begin
        let padding = 8 - f.o_bits_accumules in
        output_byte f.o_fichier (Int.shift_left f.o_accumulateur padding);
        output_byte f.o_fichier padding
    end;
    close_out f.o_fichier
```

```C
ERROR : src/algorithmique/../../snippets/algorithmique/bitpacking
```

**Remarque** Une autre possibilité consiste à ajouter un entier décrivant la taille des données non compressées. C'est d'ailleurs parfois un problème avec les formats car si la taille est stockée sur 4 octets cela limite la taille d'un fichier pouvant être compressé.

Pour la lecture, on procède de même en faisant attention à deux points :

- on va lire les bits dans l'octet de la gauche vers la droite, c'est-à-dire du bit de poids le plus fort au bit de poids le plus faible. Ainsi, si l'accumulateur contient $acc = b_1 \dots b_8$, il suffit de faire un *et* bit à bit avec `b10000000=0x80=128` pour obtenir $acc \& 0x80 = b_1 0 \dots 0$ donc un nombre qui vaut 0 si et seulement si $b_1 = 0$. Après avoir effectué cette lecture, il suffit de décaler vers la gauche en multipliant l'accumulateur par 2 : $2acc = b_2 \dots b_8 0$.
- on doit tenir compte des zéros finaux, pour ça, on a besoin de savoir qu'on est en train de lire le dernier caractère du fichier. On calcule donc la taille du fichier à son ouverture et on test si l'octet lu est l'avant-dernier, auquel cas on lit le dernier octet et on diminue d'autant le nombre de bits signifiants dans l'accumulateur.

On obtient alors le programme suivant pour la lecture :

```OCaml
type in_channel_bits = {
    i_fichier : in_channel;
    mutable i_accumulateur : int;
    mutable i_bits_accumules : int;
    i_taille : int
}

let open_in_bits fn =
    let fichier = open_in_bin fn in
    { i_fichier = fichier;
      i_accumulateur = 0; i_bits_accumules = 0; i_taille = in_channel_length fichier }

let input_bit f =
    if f.i_bits_accumules = 0
    then begin
        f.i_accumulateur <- input_byte f.i_fichier;
        f.i_bits_accumules <- 8;
        if pos_in f.i_fichier = f.i_taille - 1
        then begin
            let pad = input_byte f.i_fichier in
            f.i_bits_accumules <- f.i_bits_accumules - pad
        end
    end;
    let bit = f.i_accumulateur mod 2 = 1 in
    f.i_accumulateur <- f.i_accumulateur / 2;
    f.i_bits_accumules <- f.i_bits_accumules - 1;
    bit

let close_in_bits f = close_in f.i_fichier
```

```C
ERROR : src/algorithmique/../../snippets/algorithmique/bitpacking
```

### II.2.iv Compression d'un octet

Pour pouvoir compresser un octet, il est nécessaire d'obtenir le chemin qui mène jusqu'à la feuille dont il est l'étiquette dans l'arbre de Huffman. Pour cela, on commence par calculer l'ensemble des chemins de l'arbre de Huffman sous forme d'une table à 256 entrées qui contient le chemin associé à un octet s'il est présent dans l'arbre ou un chemin vide sinon. On parlera de représentation plate de l'arbre de Huffman.

Il suffit de faire un parcours exhaustif de l'arbre (FIXME référence aux parcours d'arbres) pour réaliser cette table :

```OCaml
let chemins a =
    let a_plat = Array.make 256 [] in
    let rec parcours a chemin =
        match a with
        | Noeud (g, d) ->
            parcours g (false::chemin);
            parcours d (true::chemin)
        | Feuille c -> a_plat.(c) <- List.rev chemin
    in parcours a []; a_plat
```

```C
ERROR : src/algorithmique/../../snippets/algorithmique/huffman.c
```

Afin de compresser un octet, on va donc aller lire le chemin dans cette table puis écrire le mot binaire correspond grâce aux fonctions d'écriture bit à bit :

```OCaml
let compresse_byte f a_plat c =
    let rec output_bool_list l =
        match l with
        | t::q -> Bitpacking.output_bit f t; output_bool_list q
        | [] -> ()
    in output_bool_list a_plat.(c)
```

```C
ERROR : src/algorithmique/../../snippets/algorithmique/huffman.c
```

**Remarque** A chaque fois qu'on va compresser un octet, on va parcourir la liste correspondant à son chemin. Comme les chemins les plus longs sont les moins fréquents, cela ne pose pas vraiment de problèmes.

Cependant, il est possible d'optimiser cela en ne stockant pas le chemin mais la fonction d'écriture elle-même. Ainsi, on ne va plus stocker des listes de booléens mais des fonctions du type `Bitpacking.out_channel_bit`

-> `unit` qui vont réaliser l'écriture compressé de l'octet correspondant.

Au cours du parcours de l'arbre, on maintient une fonction correspondant à l'écriture du préfixe du chemin `chemin`. Si on l'appel est effectué sur un noeud, on remplace `chemin` par la fonction qui appelle `chemin` puis écrit le bit correspondant au côté gauche ou droit.

Cela correspond au programme suivant :

OCaml
```
let chemins_continuation a =
    let a_plat = Array.make 256 (fun _ -> ()) in
    let rec parcours a chemin =
        match a with
        | Noeud (g, d) ->
            parcours g (fun f -> chemin f; Bitpacking.output
            parcours d (fun f -> chemin f; Bitpacking.output
        | Feuille c -> a_plat.(c) <- chemin
    in parcours a (fun _ -> ()); a_plat

let compresse_byte_continuation f a_plat c =
    a_plat.(c) f
```

C
```
ERROR : src/algorithmique/../../snippets/algorithmique/huffman
```

Python
```
ERROR : src/algorithmique/../../snippets/algorithmique/huffman.py does not exist
```

Cette représentation des chemins partiels par des fonctions est très classique dans le style de programmation fonctionnelle par passage de continuations.

### II.2.v Décompression d'un octet

Pour décompresser un octet, il suffit de parcourir l'arbre de Hufmann en lisant bit à bit le fichier compressé en descendant à gauche ou à droite selon que le bit lu soit 0 ou non. Dès qu'on arrive sur une feuille, on écrit dans le nouveau fichier le caractère correspondant.

OCaml
```
let rec decompresse_byte f a =
    match a with
    | Feuille c -> c
    | Noeud(g,d) ->
        decompresse_byte f (if Bitpacking.input_bit f then d else g)
```

C
```
ERROR : src/algorithmique/../../snippets/algorithmique/huffman.c
```

### II.2.vi Compression et décompression de fichiers

En mettant bout à bout l'ensemble des fonctions, on obtient la fonction suivante qui réalise la compression complète d'un fichier :

OCaml
```
let compresse_fichier nom_in nom_out =
    let occ = table_occurrences nom_in in
    let a = construit_arbre occ in
    let f_out = Bitpacking.open_out_bits nom_out in
    output_arbre f_out.o_fichier a;
    let a_plat = chemins a in
    let f_in = open_in_bin nom_in in
    begin
      try
        while true do
            let c = input_byte f_in in
            compresse_byte f_out a_plat c
        done
      with End_of_file -> ()
    end;
    close_in f_in;
    Bitpacking.close_out_bits f_out
```

C
```
ERROR : src/algorithmique/../../snippets/algorithmique/huffman.c
```

On obtient de même la fonction de décompression suivante :

```OCaml
let decompresse_fichier nom_in nom_out =
    let f_in = Bitpacking.open_in_bits nom_in in
    let a = input_arbre f_in.i_fichier in
    let f_out = open_out_bin nom_out in
    begin
      try
          while true do
              let c = decompresse_byte f_in a in
              output_byte f_out c
          done
      with End_of_file -> ()
    end;
    Bitpacking.close_in_bits f_in;
    close_out f_out
```

```C
ERROR : src/algorithmique/../../snippets/algorithmique/huffman.c
```

**Exemple** En compressant ainsi l'intégrale de Proust, on passe de 7543767 octets à 4249758 octets. A titre de comparaison, l'outil unix `zip` permet d'obtenir un fichier de 2724213 octets.

**Remarque** Le format de fichier présenté ici est rudimentaire. Les formats usuels sont en général plus complexes pour gérer

- l'identification : c'est-à-dire pouvoir déterminer qu'un fichier est un fichier compressé par un certain programme. Les fichiers `zip` commencent ainsi par les deux lettres PK.
- l'extensibilité : il est possible qu'on souhaite changer le format de sérialisation de l'arbre, ou même l'algorithme. En rajoutant un système de version sur les différentes parties, on peut permettre de faire évoluer un type de fichier en préservant la compatibilité avec les versions précédentes.

## II.3 Algorithme de Lempel-Ziv-Welch

L'algorithme d'Huffman est efficace, mais il présente un désavantage majeur : il nécessite de lire le contenu d'un fichier dans son intégralité pour pouvoir déterminer un code préfixe optimal. Il est toutefois possible de modifier l'algorithme pour lever cette limitation. Dans ce paragraphe, nous allons plutôt étudier une autre technique de compression qui, bien que moins efficace en pratique que Huffman, se programme assez facilement et permet de compresser des flux plutôt que des fichiers. C'est-à-dire qu'on peut compresser et décompresser des données au fur et à mesure qu'elles sont transmises.

Il s'agit de l'algorithme de Lempel-Ziv-Welch, appelé communément compression LZW, et qui est une modification faite en 1984 par Welch de l'algorithme de LZ78 de Lempel et Ziv.

### II.3.i Principe de la compression

L'idée de l'algorithme LZW est de faire avancer une fenetre sur le texte en maintenant une table des motifs déjà rencontrés. Quand on rencontre un motif déjà vu, on le code avec une référence vers la table et quand on rencontre un nouveau motif, on le code tel quel en rajoutant une entrée dans la table.

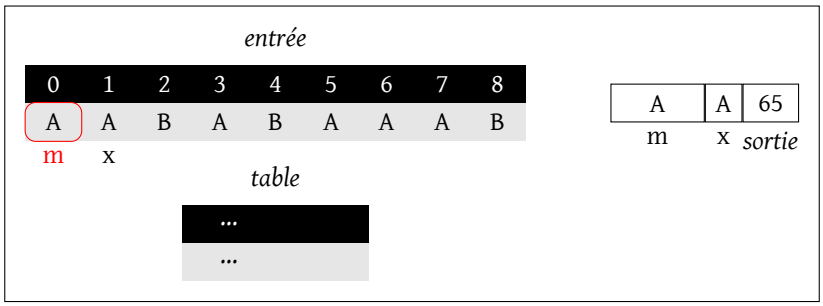
Pour la table, on peut utiliser un tableau dynamique de motifs (ref FIXME) dont la taille ne pourra pas dépasser $2^d$ éléments ou directement un tableau de $2^d$ valeurs optionnelles, dans la mesure où $d$ est en général petit. Ainsi, on pourra référencer chaque motif avec un mot de d bits. Afin de pouvoir retrouver efficacement l'indice associé à un motif, on utilise une table de hachage (ref FIXME) réalisant l'inverse de la table.

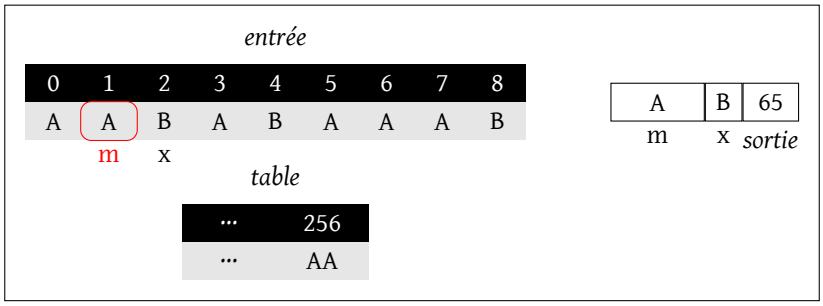
L'algorithme procède alors ainsi pour compresser :

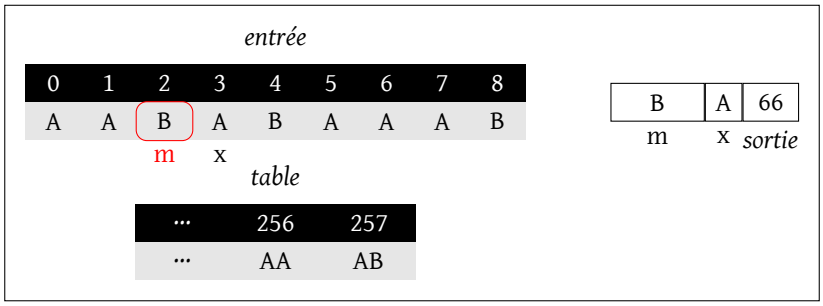
- on initialise la table avec une entrée pour chaque caractère, donc chaque octet, en considérant des caractères 8bit.
- on maintient une variable contenant le plus long suffixe $m$ du texte lu qui soit présent dans la table, il est initialisé avec la première lettre du texte.
- on lit alors chaque caractère $x$ :
  - ⋆ Soit $mx$ est dans la table, et alors on remplace le motif courant par $m \leftarrow mx$
  - ⋆ Soit $mx$ n'est pas dans la table, par construction $m$ y est nécessairement on produit alors le code correspondant à $m$, on rajoute une entrée dans la table pour $mx$ si elle contient moins de $2^d$ éléments et on repart de $m \leftarrow x$.
- quand tous les caractères ont été lus, on produit le code correspondant à $m$.

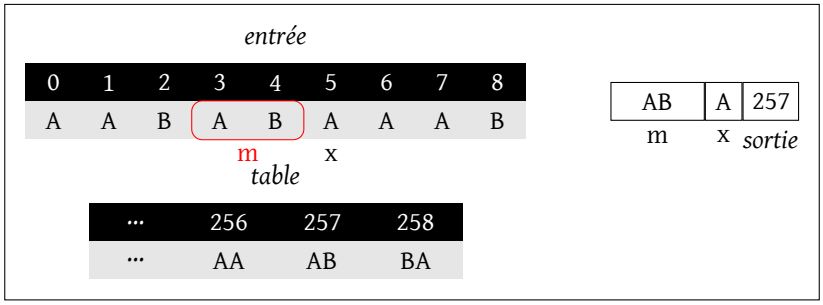
Voici les différentes étapes pour la compression de la chaîne AABABAAAB qui produit la suite d'entiers `65,256,66,65,25` qui seront alors codés dans un fichier sur d bits. Les 256 premières entrées de la table ont été volontairement

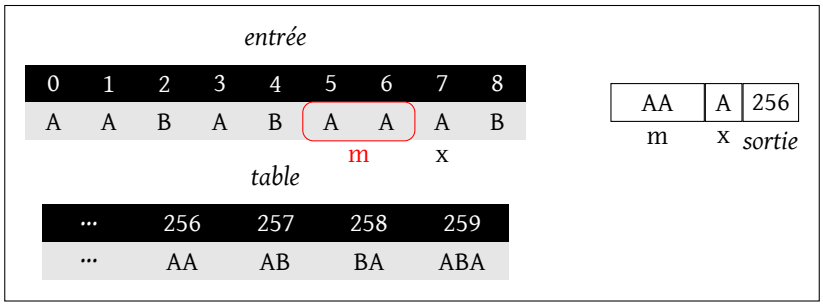
ignorées. On remarque juste que A correspond à l'index 65 et B à l'index 66.

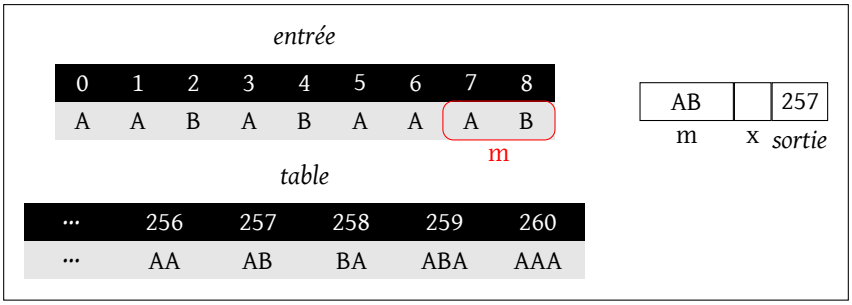


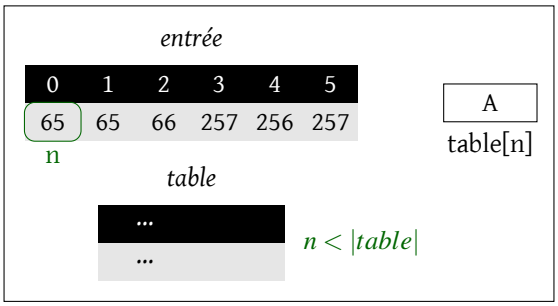
### II.3.ii Principe de la décompression

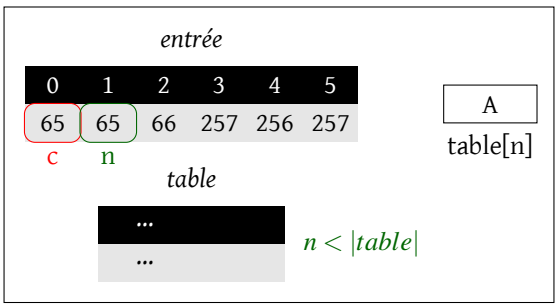
Pour décompresser, on effectue la procédure précédente en sens inverse. Cependant, il faut reconstruire la table en même temps qu'on lit le fichier compressé. Dans la majorité des cas, c'est assez immédiat. Pour le premier code lu, il s'agit forcément d'un réfèrence à un des 256 caractères, donc on le reprouduit. A partir du second code lu :

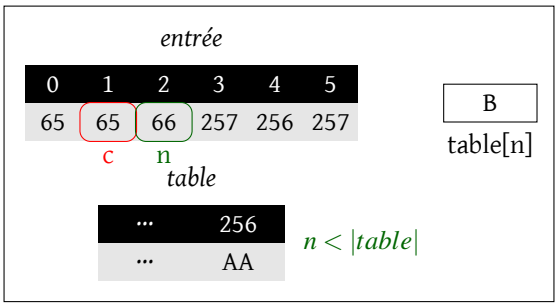
- on lit un code $n$ où $n < |\mathsf{table}|$ et $\mathsf{table}[n] = xm'$, $x$ est un caractère et $m'$ un mot.
- on écrit $xm'$ dans le fichier de sortie.
- on rajoute ensuite $mx$ dans la table où $c$ et le précédent code lu et $\mathsf{table}[c] = m$

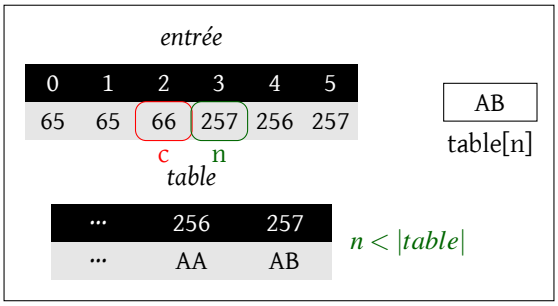
En faisant ainsi, on reproduit le processus de compression mais en remplissant la table avec un temps de retard. En effet, si on reprend le principe exposé plus haut, une entrée pour $mx$ est ajoutée dans la table quand on lit le caractère $x$ et que le motif lu précèdemment est $m$, on repart alors avec $x$ pour motif lu. C'est exactement ce qu'on fait ici en tenant compte du premier caractère de $\mathsf{table}[n]$. Contrairement à la compression, il est inutile ici de retrouver l'indice associé à un motif. On peut donc ignorer la table de hachage utilisée par la compression.

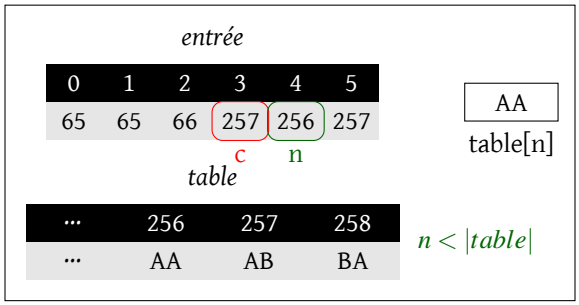
Voici les étapes de décompression de l'exemple précédent :

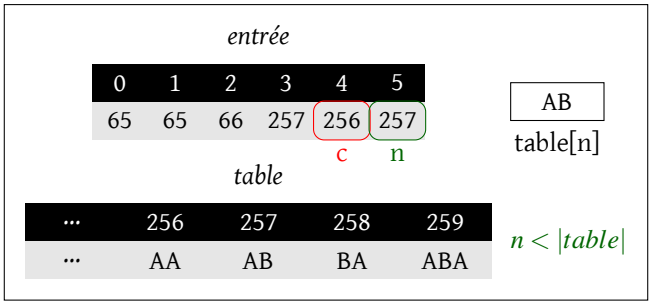




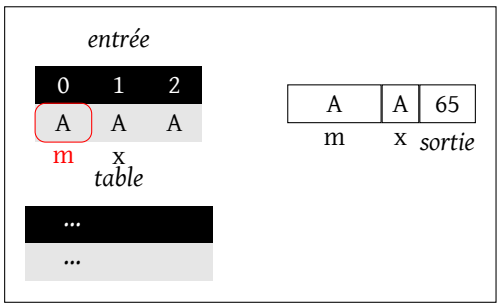



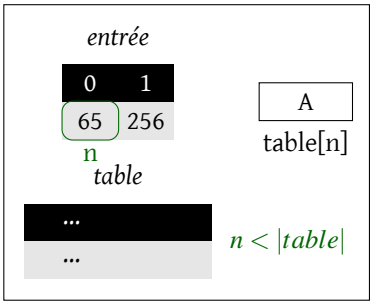

Il reste toutefois un cas à traiter, celui où $n = |\text{table}|$, c'est-à-dire quand on lit un code qui n'est pas encore présent dans la table. Pour comprendre ce cas, il est important d'identifier précisément quand il se produit dans le processus de compression. Comme on vient de le voir, on rajoute une entrée pour $mx$ après avoir produit le code $c$ correspondant à $m$. Pour que le code $n$ ne soit pas présent dans la table, il faut donc que $n$ corresponde à cette entrée $mx$. Or, quand on a compressé, on est reparti du motif $x$ à ce moment là, donc nécessairement $m$ commence par $x$. Cela signifie qu'on peut reconstruire $\text{table}[n]$ en décompressant avec $mx$ où $x$ est la première lettre de $\text{table}[c] = m$.

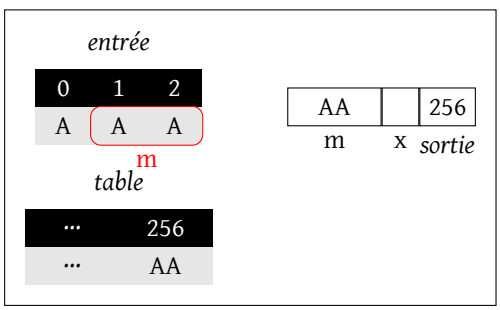
On en déduit alors la procédure complète suivante :

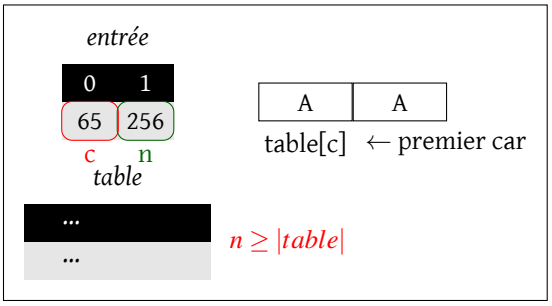
- on initialise la table avec une entrée pour chaque caractère, donc chaque octet, en considérant des caractères 8bit.
- on maintient une variable $c$ contenant le dernier code lu qu'on intialise avec le premier code en produisant le caractère correspondant.
- pour chaque code $n$ :
  - ⋆ Soit $n < |\text{table}|$ et $\text{table}[n] = xm'$ où $x$ est un caractère, alors on écrit $xm'$ en sortie
  - ⋆ Soit $n = |\text{table}|$ et alors on écrit en sortie $\text{table}[c]x$ où $x$ est le premier caractère de $\text{table}[c]$
  - ⋆ Dans tous les cas, on remplace $c \leftarrow n$ et on ajoute $\text{table}[c]x$ à la table.

Il y a un exemple classique où on a besoin de traiter ce cas : celui où le même caractère est présent plusieurs fois en début de fichier. Comme LZW est utilisé pour compresser des formats d'images où les pixels sont des indices dans une palette de 256 couleurs avec une couleur pour la transparence, le cas où la même couleur est présente au début du fichier est fréquent. Voici ici un exemple de compression et de décompression pour AAA :



**Remarque** On peut pousser un peu plus l'analyse précédente afin d'identifier précisément les situations menant à ces cas problèmatiques. Comme on vient de le voir, il est nécessaire que le texte à compresser contienne un motif de la forme $xwxwx$ où $w$ est un mot et $x$ une lettre. Mais pour que le motif lu contienne $xw$ au moment où on ajoute l'entrée $xwx$ il faut que $xw$ lui même soit dans la table, ce qui signifie que chaque préfixe de $xw$ est également dans la table au moment où on commence à le lire, mais que si le motif précédent est $w'$ alors $w'x$ n'était pas dans la table.

On considère un mot $w = a_1 \dots a_n a$ où toutes les lettres sont distinctes, et on va essayer de construire un mot $p$ tel que la compression de $p$ permettra avoir $a_1$, $a_1a_2$, $a_1a_2a_3$, ..., $a_1 \dots a_n$ dans la table.

Si on lit $a_1a_2$, on va ajouter $a_1$ puis $a_1a_2$ dans la table. On continue alors en partant de $a_2$ comme motif, pour pouvoir ajouter $a_1a_2a_3$ à la table, il faut lire $a_1a_2a_3$ ensuite car à la lecture de $a_1$, comme $a_2a_1$ n'est pas dans la table on l'ajoute et $m$ devient $a_1$, puis $a_1a_2$. Ainsi $a_1a_2a_1a_2a_3$ va ajouter $a_1, a_1a_2, a_1a_2a_3$ dans la table. En continuant ainsi avec $a_1a_2a_3a_4$, comme $a_3a_1$ n'est pas dans la table, on va procèder de même.

Le mot $p = a_1a_2a_1a_2a_3a_1a_2a_3a_4 \dots a_1a_2 \dots a_n$ va donc permettre d'ajouter tous les préfixes de $m$ dans la table.

Maintenant, pour obtenir le cas précèdent, il suffit de lire le texte $pwwa_1x = pa_1w_0a_1w_0a_1x$ où $w_0 = a_2 \dots a_n$ et $x$ n'est pas l'un des $a_i$. En effet, on lit d'abord $p$ ce qui permet d'ajouter tous les préfixes. Comme $a_na_1$ n'est pas dans la table, on continue avec $a_1$ comme motif, on l'augmente jusqu'à lire $a_1w_0$ puis on ajoute une entrée pour $a_1m_0a_1$ en ayant $a_1$ comme motif, et là on va lire $a_1w_0a_1x$ et à la lecture de $x$, produire le code de $a_1w_0a_1$ qui est le dernier saisi dans la table.

Remarquons qu'on peut aussi ne considèrer que $pwwa_1$ dans la mesure où la fin du fichier provoquera aussi l'écriture du code.

Si on considère $w =$`ABCD`, on a $p =$`ABABCABCD` et on pourra ainsi compresser le texte

$$pwwA = pAw_0Aw_0A = \texttt{ABABCABCDABCDABCDA}$$

Après lecture de $p$ on est dans la configuration suivante :

*entrée*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | A | B | C | A | B | C | D | A | B | C | D | A | B | C | D | A |
| | | | | | | | | m | x | | | | | | | | |

*table*

| ··· | 256 | 257 | 258 | 259 | 260 |
|---|---|---|---|---|---|
| ··· | AB | BA | ABC | CA | ABCD |

Puis après lecture de $wA = Aw_0A$ :

*entrée*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | A | B | C | A | B | C | D | A | B | C | D | A | B | C | D | A |
| | | | | | | | | | | | m | | x | | | | |

*table*

| ··· | 256 | 257 | 258 | 259 | 260 | 261 |
|---|---|---|---|---|---|---|
| ··· | AB | BA | ABC | CA | ABCD | DA |

Enfin, la lecture du reste du texte va déclencher l'ajout de `ABCDA` dans la table et l'écriture de son code.

*entrée*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | A | B | C | A | B | C | D | A | B | C | D | A | B | C | D | A |
| | | | | | | | | | | | | | | | m | | |

*table*

| ··· | 256 | 257 | 258 | 259 | 260 | 261 | 262 |
|---|---|---|---|---|---|---|---|
| ··· | AB | BA | ABC | CA | ABCD | DA | ABCDA |

A la décompression, on produira bien le texte correspondant avec le principe présenté plus haut :

*entrée*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 65 | 66 | 256 | 67 | 258 | 68 | 260 | 262 |
| | | | | | | c | n |

| ABCD | A |
|---|---|

table[c] ← premier car

*table*

| ··· | 256 | 257 | 258 | 259 | 260 | 261 |
|---|---|---|---|---|---|---|
| ··· | AB | BA | ABC | CA | ABCD | DA |

$n \geq |\text{table}|$

## II.3.iii Implémentation

Avant de commencer à implémenter l'algorithme, il est nécessaire de définir des fonctions de manipulation des entiers sur d bits et de lecture/écriture dans un fichier.

Tout d'abord, on définit des fonctions d'écriture d'entiers sous forme codée sur d bits à l'aide des fonctions vues précédemment :

OCaml
```
let input_code f longueur_code =
    let acc = ref 0 in
    for i = 0 to longueur_code - 1 do
        let b = input_bit f in
        if b
        then acc := !acc + (1 lsl i)
    done;
    !acc

let output_code f code longueur_code =
    assert (code < 1 lsl longueur_code);

    let acc = ref code in
    for i = 0 to longueur_code - 1 do
        output_bit f (!acc mod 2 = 1);
        acc := !acc / 2
    done
```

C
```
ERROR : src/algorithmique/../../snippets/algorithmique/bitpacking
```

On implèmente alors assez directement la compression :

OCaml

```
type table_bidir = {
    elements : string array;
    indices : (string, int) Hashtbl.t;
    mutable n_elements : int
}

let string_of_byte b = String.make 1 (Char.chr b)

let cree_table longueur_code =
    let taille_table = 1 lsl longueur_code in
    let elements = Array.make taille_table "" in
    let indices = Hashtbl.create taille_table in
    for i = 0 to 255 do
        let s = string_of_byte i in
        elements.(i) <- s;
        Hashtbl.add indices s i
    done;
    { elements = elements; indices =  indices; n_elements = 256 }

let ajoute_entree table s =
    table.elements.(table.n_elements) <- s;
    Hashtbl.add table.indices s table.n_elements;
    table.n_elements <- table.n_elements + 1

let compresse_fichier nom_in nom_out longueur_code =
    let table = cree_table longueur_code in
    let taille_table = 1 lsl longueur_code in
    let f_out = Bitpacking.open_out_bits nom_out in
    let f_in = open_in_bin nom_in in
    let m = ref (string_of_byte (input_byte f_in)) in
    begin
      try
        while true do
            let c = string_of_byte (input_byte f_in) in
            let mc = !m ^ c in
            match Hashtbl.find_opt table.indices mc with
            | Some _ -> m := mc
            | None -> begin
                let i = Hashtbl.find table.indices !m in
                Bitpacking.output_code f_out i longueur_code;
                if table.n_elements < taille_table
                then ajoute_entree table mc;
                m := c
            end
        done
      with End_of_file -> ()
    end;
    (* code final *)
    let i = Hashtbl.find table.indices !m in
    Printf.printf "%d " table.n_elements;
    Bitpacking.output_code f_out i longueur_code;
    close_in f_in;
    Bitpacking.close_out_bits f_out
```

C

```
ERROR : src/algorithmique/../../snippets/algorithmique/lzw.c does
```

Et on procède de même pour la décompression :

```OCaml
let decompresse_fichier nom_in nom_out longueur_code =
    let table = cree_table longueur_code in
    let taille_table = 1 lsl longueur_code in
    let f_in = Bitpacking.open_in_bits nom_in in
    let f_out = open_out_bin nom_out in
    let code = ref (Bitpacking.input_code f_in longueur_code) in
    output_string f_out table.elements.(!code);

    begin
      try
        while true do
            let nouveau = Bitpacking.input_code f_in longueur_code in
            let s_code = table.elements.(!code) in
            let s =
                if nouveau = table.n_elements
                then let x = String.sub s_code 0 1
                     in s_code ^ x
                else table.elements.(nouveau) in
            let x = String.sub s 0 1 in
            output_string f_out s;
            if table.n_elements < taille_table
            then ajoute_entree table (s_code ^ x);
            code := nouveau
        done
      with End_of_file -> ()
    end;
    Bitpacking.close_in_bits f_in;
    close_out f_out
```

```C
ERROR : src/algorithmique/../../snippets/algorithmique/lzw.c does
```

### II.3.iv Impact de la longueur du code

Afin d'étudier l'impact de la longueur du code sur la taille des fichiers compressés, on considère deux fichiers :

- `proust.txt` contenant, en 7543768 octets, l'intégrale de *à la recherche du temps perdu* de Marcel Proust
- `code.py` contenant 8566 octets de code source Python

On note np la taille en nombre d'octets après compression du fichier `proust.txt` et tp le nombre d'entrées dans la table à la fin du processus. De même, on note nc et tc les valeurs respectives pour le fichier `code.py`.

On obtient alors les valeurs suivantes en fonction du nombre d de bits du code :

| d | np | nc | tp | tc |
|---|---|---|---|---|
| 8 | 7543768 | 8566 | 256 | 256 |
| 9 | 5056398 | 5709 | 512 | 512 |
| 10 | 4195124 | 4431 | 1024 | 1024 |
| 11 | 3864174 | 3948 | 2048 | 2048 |
| 12 | 3612505 | 4039 | 4096 | 2947 |
| 13 | 3434424 | 4376 | 8192 | 2947 |
| 14 | 3262145 | 4712 | 16384 | 2947 |
| 15 | 3131790 | 5049 | 32768 | 2947 |
| 16 | 2998639 | 5385 | 65536 | 2947 |
| 17 | 2682264 | 5722 | 131072 | 2947 |
| 18 | 2554323 | 6058 | 262144 | 2947 |
| 19 | 2500314 | 6395 | 524288 | 2947 |
| 20 | 2557656 | 6731 | 1023317 | 2947 |

On constate qu'il est nécessaire d'avoir un texte riche pour bénéficier d'une grande longueur de code. Le fichier `code.py` ne contenant pas plus que 2947 motifs. Même si le fichier `proust.txt` en contient plus que les tailles considérées ici, il y a un compromis qui s'établit entre la richesse de la table et la taille du code. Ainsi, il semble que le fichier `proust.txt` soit compressé de manière optimale avec un code de longueur 19.

# III Problèmes supplémentaires