Définitions et notations

On s'intéresse au problème de la recherche d'une chaîne de caractères appelée motif dans une autre chaîne de caractères appelée texte. Plus précisement, on veut lister toutes les occurences (par leur position) du motif dans le texte.
On notera :

Définitions et notations

On s'intéresse au problème de la recherche d'une chaîne de caractères appelée motif dans une autre chaîne de caractères appelée texte. Plus précisement, on veut lister toutes les occurences (par leur position) du motif dans le texte.
On notera :

- m le motif et l_m sa longueur

Définitions et notations

On s'intéresse au problème de la recherche d'une chaîne de caractères appelée motif dans une autre chaîne de caractères appelée texte. Plus précisement, on veut lister toutes les occurences (par leur position) du motif dans le texte.
On notera :

- m le motif et l_m sa longueur
- t le texte et l_t sa longueur

Définitions et notations

On s'intéresse au problème de la recherche d'une chaîne de caractères appelée motif dans une autre chaîne de caractères appelée texte. Plus précisement, on veut lister toutes les occurences (par leur position) du motif dans le texte.
On notera :

- m le motif et l_m sa longueur
- t le texte et l_t sa longueur

D'autre part, parfois le problème se réduira à déterminer si m est présent ou non dans t, ou encore on cherchera uniquement la première occurence.

Définitions et notations

On s'intéresse au problème de la recherche d'une chaîne de caractères appelée motif dans une autre chaîne de caractères appelée texte. Plus précisement, on veut lister toutes les occurences (par leur position) du motif dans le texte.
On notera :

- m le motif et l_m sa longueur
- t le texte et l_t sa longueur

D'autre part, parfois le problème se réduira à déterminer si m est présent ou non dans t, ou encore on cherchera uniquement la première occurence.

Exemple

La recherche du motif m=`exe` ($l_m = 3$) dans le texte t =`un excellent exemple et un exercice extraordinaire` ($l_t = 50$) doit produire la liste d'occurrences : [13 ; 27].

Définitions et notations

On s'intéresse au problème de la recherche d'une chaîne de caractères appelée motif dans une autre chaîne de caractères appelée texte. Plus précisement, on veut lister toutes les occurences (par leur position) du motif dans le texte.
On notera :

- m le motif et l_m sa longueur
- t le texte et l_t sa longueur

D'autre part, parfois le problème se réduira à déterminer si m est présent ou non dans t, ou encore on cherchera uniquement la première occurence.

Exemple

La recherche du motif m=`exe` ($l_m = 3$) dans le texte t =`un excellent exemple et un exercice extraordinaire` ($l_t = 50$) doit produire la liste d'occurrences : [13 ; 27].

```
un␣excellent␣exemple␣et␣un␣exercice␣extraordinaire
0              13             27                  49
```

Recherche naïve

Pour recherche si un motif m se trouve dans un texte t, on peut :

1. parcourir chaque caractère de t jusqu'à celui d'indice ? inclus (indice de la dernière occurrence possible) :

indice dans le texte	0	...	?	$l_t - 1$
indice dans le motif			0	$l_m - 1$

Recherche naïve

Pour recherche si un motif m se trouve dans un texte t, on peut :

1. parcourir chaque caractère de t jusqu'à celui d'indice $l_t - l_m$ inclus (indice de la dernière occurrence possible) :

indice dans le texte	0	...	$l_t - l_m$	$l_t - 1$
indice dans le motif			0	$l_m - 1$

Recherche naïve

Pour recherche si un motif m se trouve dans un texte t, on peut :

1. parcourir chaque caractère de t jusqu'à celui d'indice $l_t - l_m$ inclus (indice de la dernière occurrence possible) :

indice dans le texte	0	...	$l_t - l_m$	$l_t - 1$
indice dans le motif			0	$l_m - 1$

2. si le caractère correspond au premier caractère du motif m, alors on avance dans le motif tant que les caractères coïncident.

Recherche naïve

Pour recherche si un motif m se trouve dans un texte t, on peut :

1. parcourir chaque caractère de t jusqu'à celui d'indice $l_t - l_m$ inclus (indice de la dernière occurrence possible) :

indice dans le texte	0	...	$l_t - l_m$	$l_t - 1$
indice dans le motif			0	$l_m - 1$

2. si le caractère correspond au premier caractère du motif m, alors on avance dans le motif tant que les caractères coïncident.
3. si on atteint la fin du motif, alors m se trouve dans t. Sinon on passe au caractère suivant de t.

Recherche naïve

Pour recherche si un motif m se trouve dans un texte t, on peut :

1. parcourir chaque caractère de t jusqu'à celui d'indice $l_t - l_m$ inclus (indice de la dernière occurrence possible) :

indice dans le texte	0	...	$l_t - l_m$	$l_t - 1$
indice dans le motif			0	$l_m - 1$

2. si le caractère correspond au premier caractère du motif m, alors on avance dans le motif tant que les caractères coïncident.
3. si on atteint la fin du motif, alors m se trouve dans t. Sinon on passe au caractère suivant de t.

Exemple

Visualisation en ligne du fonctionnement de l'algorithme

Implémentation

- En C, on rappelle que les chaines de caractères sont des tableaux équipés d'une sentinelle `'\0'` qui en indique la fin.

.

Implémentation

- En C, on rappelle que les chaines de caractères sont des tableaux équipés d'une sentinelle `'\0'` qui en indique la fin. En supposant qu'on recherche simplement l'indice de la première occurence et qu'on renvoie -1 si le motif ne se trouve pas dans la chaine, la signature de la fonction est `int naive(char* motif, char *texte)`.

Implémentation

- En C, on rappelle que les chaines de caractères sont des tableaux équipés d'une sentinelle `'\0'` qui en indique la fin. En supposant qu'on recherche simplement l'indice de la première occurence et qu'on renvoie -1 si le motif ne se trouve pas dans la chaine, la signature de la fonction est `int naive(char* motif, char *texte)`.
- En OCaml, les chaines de caractères sont des tableaux non mutables de caractères, et on rappelle qu'on accède au caractère d'indice `i` de la chaine `s` avec la syntaxe `s.[i]`.

Implémentation

- En C, on rappelle que les chaines de caractères sont des tableaux équipés d'une sentinelle `'\0'` qui en indique la fin. En supposant qu'on recherche simplement l'indice de la première occurence et qu'on renvoie -1 si le motif ne se trouve pas dans la chaine, la signature de la fonction est `int naive(char* motif, char *texte)`.
- En OCaml, les chaines de caractères sont des tableaux non mutables de caractères, et on rappelle qu'on accède au caractère d'indice `i` de la chaine `s` avec la syntaxe `s.[i]`. En utilisant les aspects impératifs du langage, on peut travailler directement sur les chaines à l'aide de boucles et d'indices de parcours. .

Implémentation

- En C, on rappelle que les chaines de caractères sont des tableaux équipés d'une sentinelle `'\0'` qui en indique la fin. En supposant qu'on recherche simplement l'indice de la première occurence et qu'on renvoie -1 si le motif ne se trouve pas dans la chaine, la signature de la fonction est `int naive(char* motif, char *texte)`.
- En OCaml, les chaines de caractères sont des tableaux non mutables de caractères, et on rappelle qu'on accède au caractère d'indice `i` de la chaine `s` avec la syntaxe `s.[i]`. En utilisant les aspects impératifs du langage, on peut travailler directement sur les chaines à l'aide de boucles et d'indices de parcours. Sinon on pourra les transformer en liste de caractères.

Exercices

1. En C, donner une implémentation de la fonction `int naive(char* motif, char *texte)` qui renvoie l'indice de la première occurence de `motif` dans `texte` (-1 si absent).
2. En OCaml
   1. Même question avec une fonction `naive : string -> string -> int`, on pourra éventuellement gérer la sortie anticipée de boucle à l'aide d'une exception.
   2. Ecrire une fonction `str_to_list : string -> char list` qui transforme une chaine de caractères en la liste de caractères correspondante.
   3. Ecrire une version récursive de la recherche naive sans utiliser les aspects impératifs du langage. On pourra éventuellement écrire une fonction annexe `est_prefixe 'a list -> 'a list -> bool` qui renvoie `true` si la première liste est le début de la seconde.

Coût de la recherche simple

En notant l_m la longueur du motif et l_t celle de la chaine :

Coût de la recherche simple

En notant l_m la longueur du motif et l_t celle de la chaine :

- La boucle `for` est parcourue au plus $l_t - l_m + 1$ fois

Coût de la recherche simple

En notant l_m la longueur du motif et l_t celle de la chaine :

- La boucle `for` est parcourue au plus $l_t - l_m + 1$ fois
- Pour chacun de ces parcours, la boucle `while` interne est parcourue au plus l_m fois

Coût de la recherche simple

En notant l_m la longueur du motif et l_t celle de la chaine :

- La boucle `for` est parcourue au plus $l_t - l_m + 1$ fois
- Pour chacun de ces parcours, la boucle `while` interne est parcourue au plus l_m fois

Au plus, l'algorithme effectue donc $l_m(l_t - l_m + 1)$ comparaisons.

Coût de la recherche simple

En notant l_m la longueur du motif et l_t celle de la chaine :

- La boucle `for` est parcourue au plus $l_t - l_m + 1$ fois
- Pour chacun de ces parcours, la boucle `while` interne est parcourue au plus l_m fois

Au plus, l'algorithme effectue donc $l_m(l_t - l_m + 1)$ comparaisons.

Exemple

Combien de comparaisons seront nécessaires si on recherche le motif `bbbbbbbbba` (neuf fois le caractère `b` suivi d'un `a`) dans une chaine contenant un million de `b` ?

Accélération de la recherche

Supposons qu'on recherche le motif `extra` dans la chaine `un excellent exemple et un exercice extraordinaire`. La comparaison naïve ci-dessus commence par :

Accélération de la recherche

Supposons qu'on recherche le motif `extra` dans la chaine `un excellent exemple et un exercice extraordinaire`. La comparaison naïve ci-dessus commence par :

u	n		e	x	c	e	l	l	e	n	t
↕											
e	x	t	r	a							

Accélération de la recherche

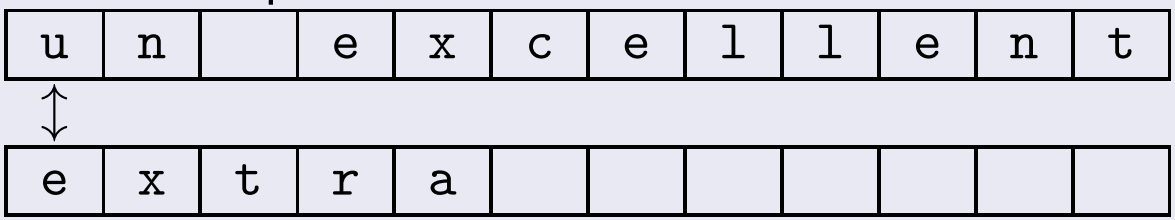
Supposons qu'on recherche le motif `extra` dans la chaine `un excellent exemple et un exercice extraordinaire`. La comparaison naïve ci-dessus commence par :

u	n		e	x	c	e	l	l	e	n	t
↕											
e	x	t	r	a							

Deux idées vont permettre d'accélérer la recherche :

Accélération de la recherche

Supposons qu'on recherche le motif `extra` dans la chaine `un excellent exemple et un exercice extraordinaire`. La comparaison naïve ci-dessus commence par :



Deux idées vont permettre d'accélérer la recherche :

- Commencer par la fin du motif.

Accélération de la recherche

Supposons qu'on recherche le motif `extra` dans la chaine `un excellent exemple et un exercice extraordinaire`. La comparaison naïve ci-dessus commence par :

u	n		e	x	c	e	l	l	e	n	t
↕											
e	x	t	r	a							

Deux idées vont permettre d'accélérer la recherche :

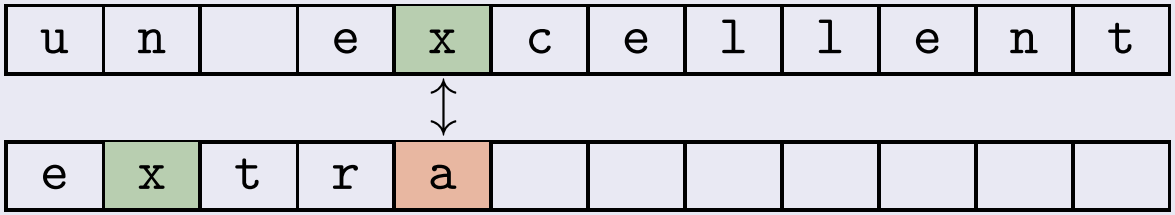
- Commencer par la fin du motif.
- Prétraiter le motif de façon à éviter des comparaisons inutiles.

Accélération de la recherche

Dans l'exemple ci-dessus cela donne :

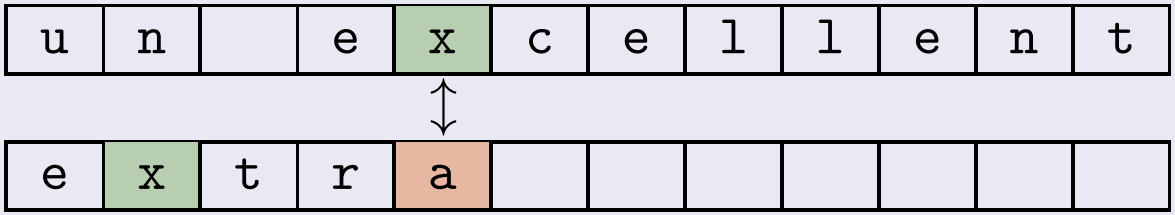
Accélération de la recherche

Dans l'exemple ci-dessus cela donne :



Accélération de la recherche

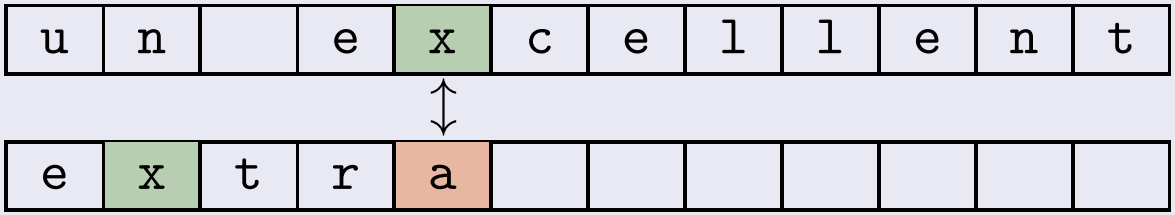
Dans l'exemple ci-dessus cela donne :



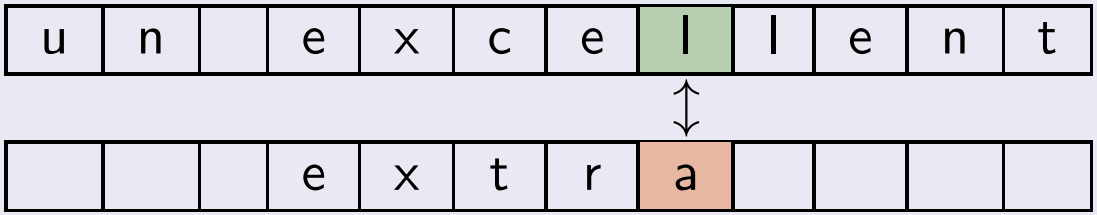
On peut directement décaler le motif de 3 emplacements car le dernier `x` du motif se trouve à 3 emplacements de la fin du motif.

Accélération de la recherche

Dans l'exemple ci-dessus cela donne :

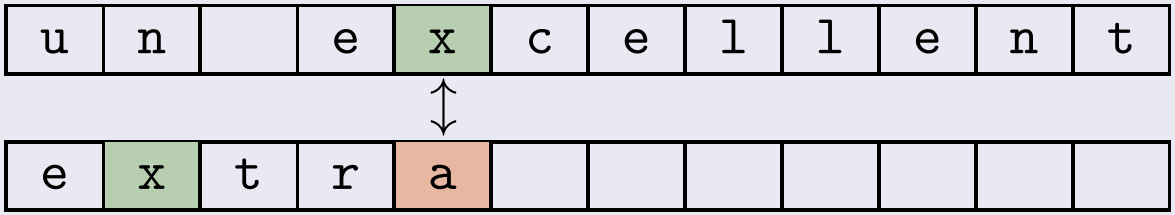


On peut directement décaler le motif de 3 emplacements car le dernier `x` du motif se trouve à 3 emplacements de la fin du motif.

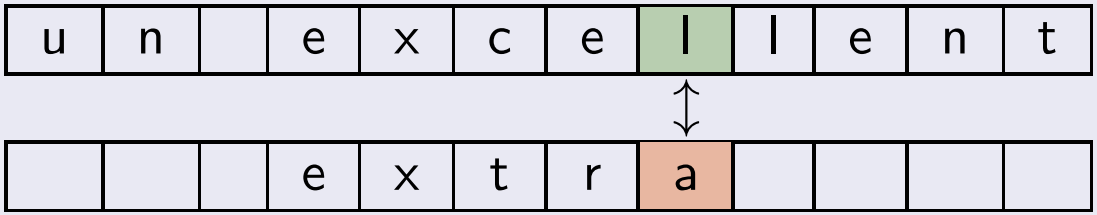


Accélération de la recherche

Dans l'exemple ci-dessus cela donne :



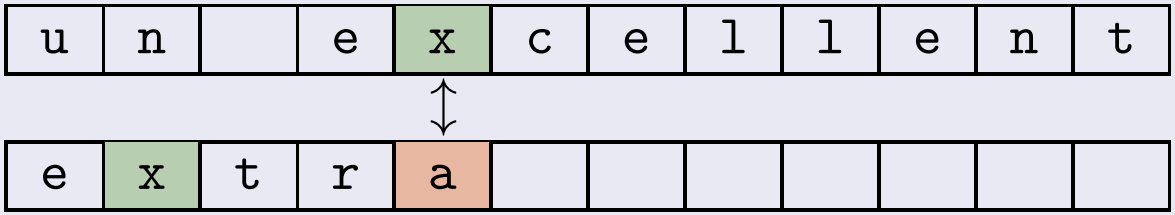
On peut directement décaler le motif de 3 emplacements car le dernier `x` du motif se trouve à 3 emplacements de la fin du motif.



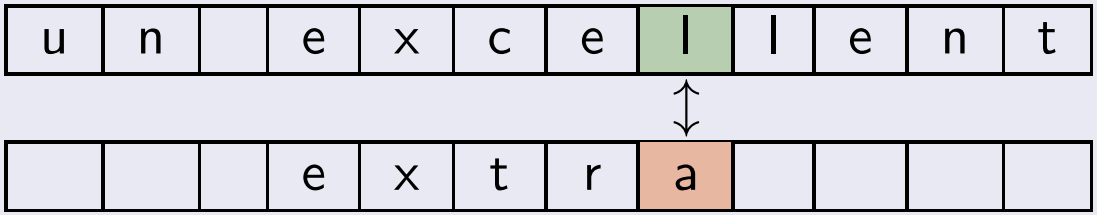
Cette fois, le `l` ne se trouve pas dans le motif, on peut donc décaler de la longueur du motif. Et la recherche s'arrête en ayant effectué seulement deux comparaisons.

Accélération de la recherche

Dans l'exemple ci-dessus cela donne :



On peut directement décaler le motif de 3 emplacements car le dernier `x` du motif se trouve à 3 emplacements de la fin du motif.



Cette fois, le `l` ne se trouve pas dans le motif, on peut donc décaler de la longueur du motif. Et la recherche s'arrête en ayant effectué seulement deux comparaisons.

Visualisation en ligne

Visualisation en ligne du fonctionnement de l'algorithme accéléré

Algorithme de Boyer-Moore-Hoorspool

- La première phase consiste en un prétraitement du motif, afin de construire une fonction de décalage qui indique pour chaque caractère c :

Algorithme de Boyer-Moore-Hoorspool

- La première phase consiste en un prétraitement du motif, afin de construire une fonction de décalage qui indique pour chaque caractère c :
  - Si c est dans le motif, Le nombre de caractères entre la *dernière occurrence* de c et la fin du motif (l'avant dernière si c est le dernier caractère.)

Algorithme de Boyer-Moore-Hoorspool

- La première phase consiste en un prétraitement du motif, afin de construire une fonction de décalage qui indique pour chaque caractère c :
  - Si c est dans le motif, Le nombre de caractères entre la *dernière occurrence* de c et la fin du motif (l'avant dernière si c est le dernier caractère.)
  - Sinon la longueur du motif

Algorithme de Boyer-Moore-Hoorspool

- La première phase consiste en un prétraitement du motif, afin de construire une fonction de décalage qui indique pour chaque caractère c :
  - Si c est dans le motif, Le nombre de caractères entre la *dernière occurrence* de c et la fin du motif (l'avant dernière si c est le dernier caractère.)
  - Sinon la longueur du motif
- Ensuite on effectue une recherche en partant de la fin du motif en cas de non correspondance, on décale de la valeur fournie par la fonction de décalage.

Algorithme de Boyer-Moore-Hoorspool

- La première phase consiste en un prétraitement du motif, afin de construire une fonction de décalage qui indique pour chaque caractère c :
  - Si c est dans le motif, Le nombre de caractères entre la *dernière occurrence* de c et la fin du motif (l'avant dernière si c est le dernier caractère.)
  - Sinon la longueur du motif
- Ensuite on effectue une recherche en partant de la fin du motif en cas de non correspondance, on décale de la valeur fournie par la fonction de décalage.

Exemple

Algorithme de Boyer-Moore-Hoorspool

- La première phase consiste en un prétraitement du motif, afin de construire une fonction de décalage qui indique pour chaque caractère c :
  - Si c est dans le motif, Le nombre de caractères entre la *dernière occurrence* de c et la fin du motif (l'avant dernière si c est le dernier caractère.)
  - Sinon la longueur du motif
- Ensuite on effectue une recherche en partant de la fin du motif en cas de non correspondance, on décale de la valeur fournie par la fonction de décalage.

Exemple

1. Construire la table de décalage du motif "`toto`"

Algorithme de Boyer-Moore-Hoorspool

- La première phase consiste en un prétraitement du motif, afin de construire une fonction de décalage qui indique pour chaque caractère c :
  - Si c est dans le motif, Le nombre de caractères entre la *dernière occurrence* de c et la fin du motif (l'avant dernière si c est le dernier caractère.)
  - Sinon la longueur du motif
- Ensuite on effectue une recherche en partant de la fin du motif en cas de non correspondance, on décale de la valeur fournie par la fonction de décalage.

Exemple

1. Construire la table de décalage du motif "`toto`"
2. Simuler le fonction de l'algorithme de Boyer-Moore-Hoorspool pour recherche ce motif dans le chaine "`zéro plus zéro = la tête à toto`"

Exercices

1. Ecrire en C, une fonction de signature `int *cree_decalage(char *motif)` qui renvoie la table de décalage d'un motif. On représentera un caractère par son code (donc un entier) et on suppose qu'on utilise la table ASCII standard qui contient 127 caractères.
2. Simuler la recherche de `abb` dans le texte `b...b` (t fois la lettre b) avec l'algorithme de Boyer-Mooore. Donner le nombre de comparaisons effectué et comparer avec la recherche naïve.

Implémentation en C

Fonction qui renvoie `true` si `motif` est présent dans `texte` et `false` sinon.

```
1  bool appartient_bmh(char *motif, char *texte){
2      int *dec = cree_decalage(motif);
3      int lt = strlen(texte);
4      int lm = strlen(motif);
5      int idx = 0;
6      int im;
7      while (idx < lt - lm + 1){
8          im = lm - 1;
9          while (im >= 0 && texte[idx + im] == motif[im]){
10             im = im - 1;}
11         if (im < 0){
12             return true;}
13         idx += dec[(int)texte[idx + lm - 1]];}
14     return false;}
```

Principe

- L'idée de l'algorithme est d'utiliser une fonction de hachage h sur les chaines de caractères, et de comparer pour chaque indice `i` où une correspondance peut exister (de 0 à $l_t - l_m$) les hash du motif $h(m)$ avec le hash du texte commençant à l'indice `i` et de longueur l_m c'est-à-dire $h((t_i, t_{i+1}, \ldots, t_{i+l_m-1}))$.

Principe

- L'idée de l'algorithme est d'utiliser une fonction de hachage h sur les chaines de caractères, et de comparer pour chaque indice `i` où une correspondance peut exister (de 0 à $l_t - l_m$) les hash du motif $h(m)$ avec le hash du texte commençant à l'indice `i` et de longueur l_m c'est-à-dire $h((t_i, t_{i+1}, \ldots, t_{i+l_m-1}))$.
- On effectue la comparaison caractère par caractère seulement dans le cas où les deux hash sont égaux.

Principe

- L'idée de l'algorithme est d'utiliser une fonction de hachage h sur les chaines de caractères, et de comparer pour chaque indice `i` où une correspondance peut exister (de 0 à $l_t - l_m$) les hash du motif $h(m)$ avec le hash du texte commençant à l'indice `i` et de longueur l_m c'est-à-dire $h((t_i, t_{i+1}, \ldots, t_{i+l_m-1}))$.
- On effectue la comparaison caractère par caractère seulement dans le cas où les deux hash sont égaux.

Exemple

On suppose qu'on recherche « `de` » dans le texte « `bahcdef` » et que la fonction de hachage fait simplement la somme des codes ASCII des caractères. Calculer le hash du motif et détailler les étapes de l'algorithme.

Principe (amélioré)

Principe (amélioré)

Principe (amélioré)

- Si le calcul du hash de la partie du texte $(t_i, t_{i+1}, \ldots, t_{i+l_m-1})$ s'effectue en $O(m)$ alors la complexité est la même que celle de la recherche naïve.

Principe (amélioré)

- Si le calcul du hash de la partie du texte $(t_i, t_{i+1}, \ldots, t_{i+l_m-1})$ s'effectue en $O(m)$ alors la complexité est la même que celle de la recherche naïve.
- On tire parti du fait que deux hash consécutifs à calculer ne diffère que de deux caractères :
  $(t_i, t_{i+1}, \ldots, t_{i+l_m-1})$
  $(t_{i+1}, \ldots, t_{i+l_m-1}, t_i + l_m)$
  et on utilise une fonction de hachage dite déroulante (*rolling hash*) pour calculer les hash de proche en proche en $O(1)$.

Exemple

1. Ecrire en C, une fonction de hachage h qui effectue simplement la somme des codes des caractères.
2. Montrer qu'il s'agit d'une fonction de hachage déroulante et donner l'expression de $h(t_{i+1}, \ldots, t_{i+l_m-1}, t_i + l_m)$ en fonction de $h(t_i, t_{i+1}, \ldots, t_{i+l_m-1})$
3. Ecrire une l'implémentation de l'algorithme de Rabin-Karp qui utilise cette fonction de hachage.

   lors que les deux hash sont égaux (celui du motif) et celui de la partie de texte, on pouura utiliser directement la fonction strncmp de `<string.h>` qui prend en argument deux chaines de caractères et un entier n et renvoie 0 lorsque les n premiers caractères des deux chaines sont égaux.

Hypothèses

On considère dans toute la suite, qu'on travaille sur un fichier texte au format ASCII et on rappelle que ce format permet de représenter 128 caractères (de code 0 à 127) tous codés sur 8 bits. Par conséquent la taille d'un tel fichier (en bits) est simplement 8 fois son nombre de caractères. On cherche à compresser la taille d'un tel fichier *sans perdre d'informations*.

Hypothèses

On considère dans toute la suite, qu'on travaille sur un fichier texte au format ASCII et on rappelle que ce format permet de représenter 128 caractères (de code 0 à 127) tous codés sur 8 bits. Par conséquent la taille d'un tel fichier (en bits) est simplement 8 fois son nombre de caractères. On cherche à compresser la taille d'un tel fichier *sans perdre d'informations*.

Principe

Le principe de base de l'algorithme d'Huffman est de représenter par un code plus court les caractères les plus fréquents :

- on commence par calculer le nombre d'occurrences de chaque caractère
- on construit un arbre dont les feuilles sont les caractères et où la profondeur d'un caractère est fonction directe de sa fréquence (plus un caractère est haut dans l'arbre et plus il apparaît souvent)
- on génère un code prefixe unique pour chaque caractère.

Illustration sur un exemple

On souhaite compresser le texte « `les petits tests` ».

Illustration sur un exemple

On souhaite compresser le texte « `les petits tests` ».

1.

Caractère	␣	e	i	l	p	s	t
Occurrences	2	3	1	1	1	4	4

Illustration sur un exemple

On souhaite compresser le texte « `les petits tests` ».

1.

Caractère	␣	e	i	l	p	s	t
Occurrences	2	3	1	1	1	4	4

Illustration sur un exemple

On souhaite compresser le texte « `les petits tests` ».

1. | Caractère | ␣ | e | i | l | p | s | t |
   |---|---|---|---|---|---|---|---|
   | Occurrences | 2 | 3 | 1 | 1 | 1 | 4 | 4 |
2. Au départ chaque caractère représente un arbre comportant un noeud unique :
   (i)^1 (l)^1 (p)^1 (␣)^2 (e)^3 (s)^4 (t)^4

Illustration sur un exemple

On souhaite compresser le texte « `les petits tests` ».

1.

Caractère	␣	e	i	l	p	s	t
Occurrences	2	3	1	1	1	4	4

2. Au départ chaque caractère représente un arbre comportant un noeud unique :
   (i)^1 (l)^1 (p)^1 (␣)^2 (e)^3 (s)^4 (t)^4
3. On extrait les deux arbres portant les plus petits nombres d'occurrences et on les assemblent en un nouvel arbre ayant la somme de leur nombre d'occurrence :
   (i)^1 (l)^1 forment

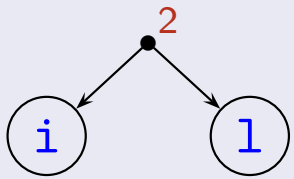
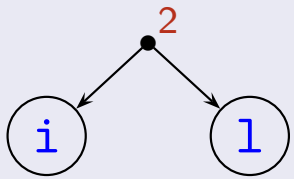


Illustration sur un exemple

On souhaite compresser le texte « `les petits tests` ».

1.

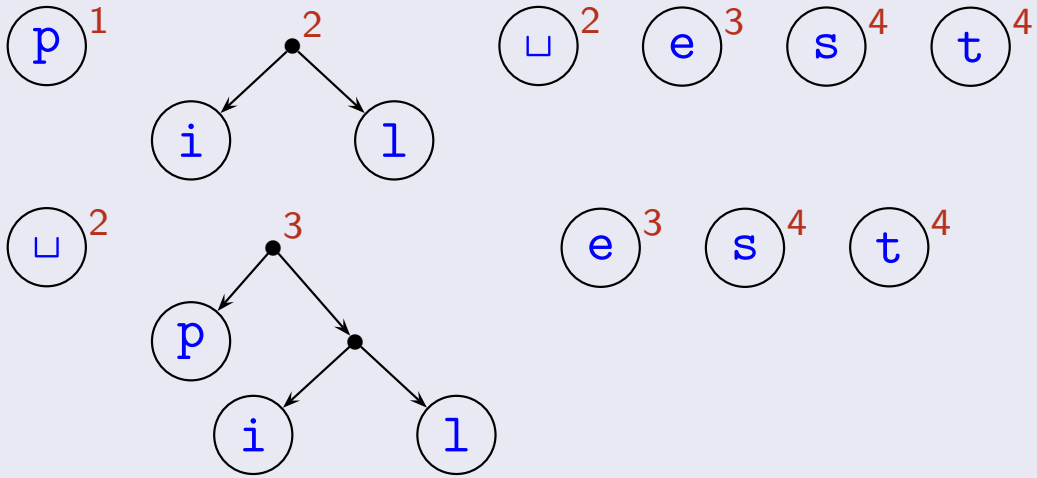
Caractère	␣	e	i	l	p	s	t
Occurrences	2	3	1	1	1	4	4

2. Au départ chaque caractère représente un arbre comportant un noeud unique :
   (i)^1 (l)^1 (p)^1 (␣)^2 (e)^3 (s)^4 (t)^4
3. On extrait les deux arbres portant les plus petits nombres d'occurrences et on les assemblent en un nouvel arbre ayant la somme de leur nombre d'occurrence :
   (i)^1 (l)^1 forment
   

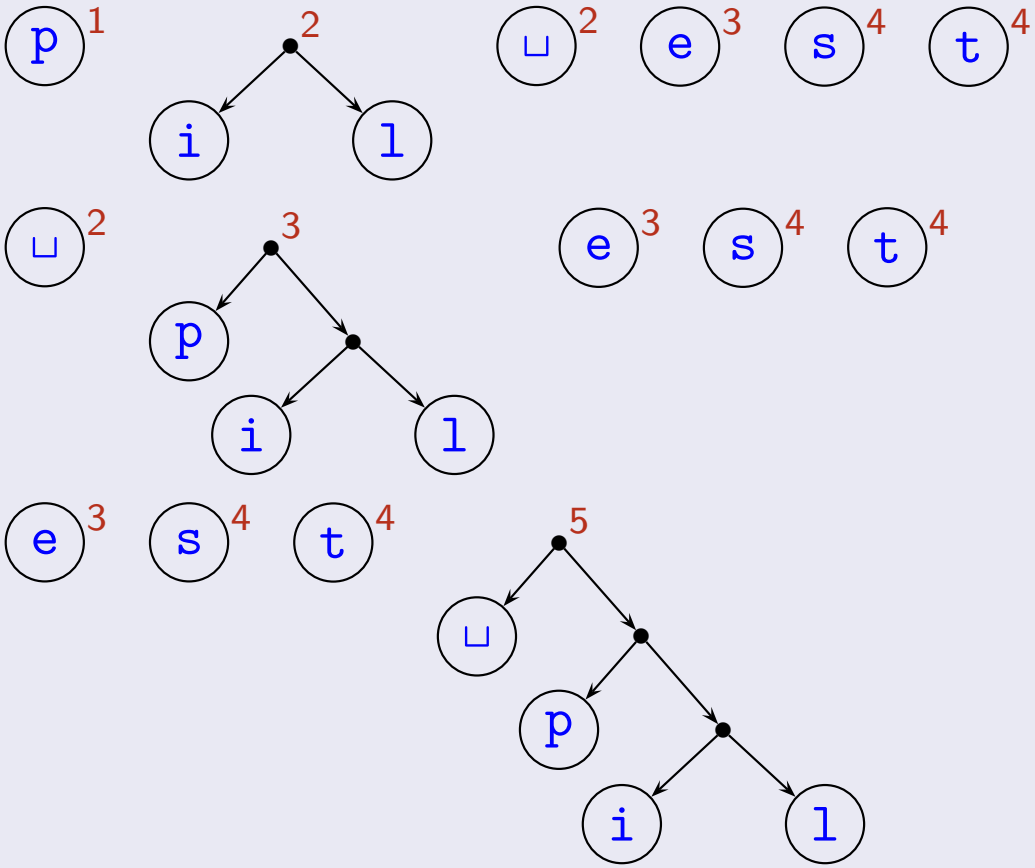
4. Ce nouvel arbre est inséré dans la structure de données et on réitère le processus jusqu'à ce qu'il reste un seul arbre

Encore quelques étapes pour illustrer . . .

Encore quelques étapes pour illustrer . . .



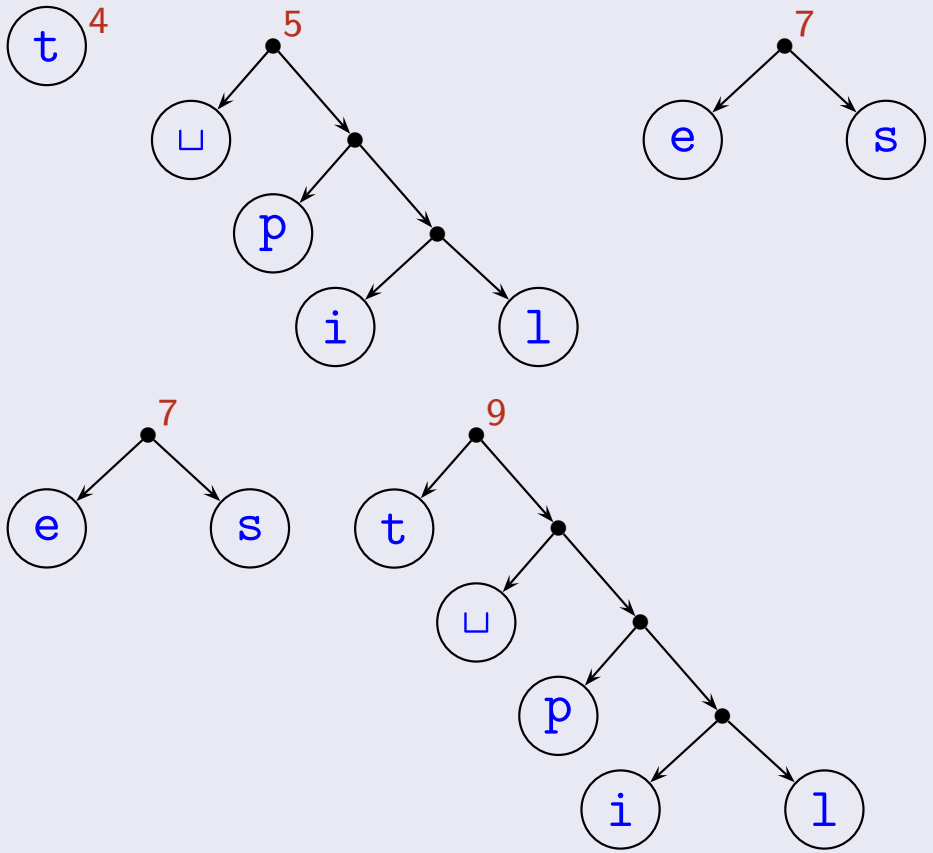
Encore quelques étapes pour illustrer . . .



Et pour finir . . .

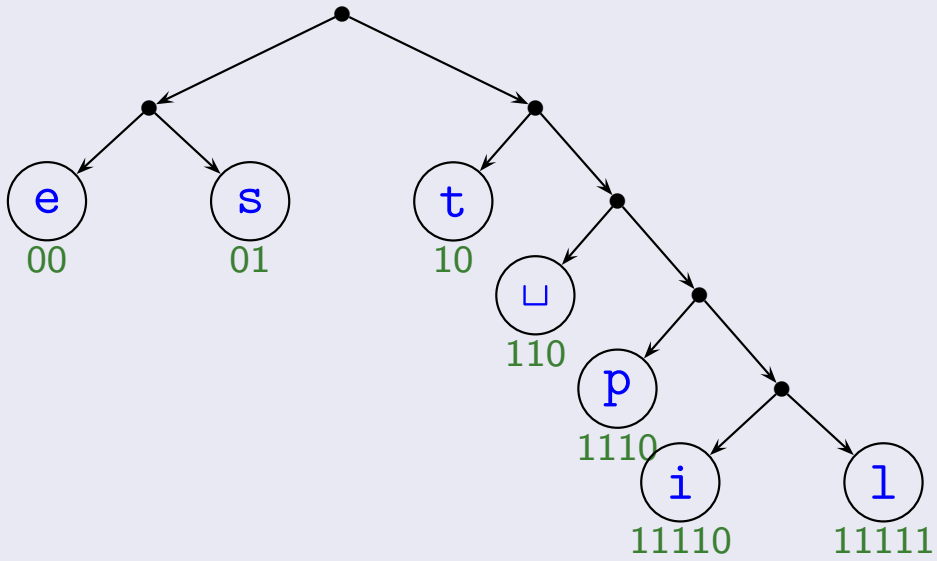
Et pour finir . . .

Et pour finir . . .



Attribution des codes

Sur l'arbre final on attribut un code à chaque caractère en fonction de sa position dans l'arbre (on ajoute 0 en allant à gauche et 1 en allant à droite)



On obtient un code préfixe, c'est-à-dire que le code d'un caractère n'est jamais le début du code d'un autre (ce qui évite toute ambigüité lors de la décompression). Le texte initial contenait 4 caractères `s` pour un total de 32 bits, maintenat les `s` sont codés sur 2 bits et donc occupent simplement `8 bits`

Exercices

1. Calculer le taux de compression de l'algorithme sur l'exemple précédent.

Exercices

1. Calculer le taux de compression de l'algorithme sur l'exemple précédent.
2. Quelle structure de données est la plus adaptée pour ranger les arbres au fur et à mesure de leur construction, pourquoi ?

Exercices

1. Calculer le taux de compression de l'algorithme sur l'exemple précédent.
2. Quelle structure de données est la plus adaptée pour ranger les arbres au fur et à mesure de leur construction, pourquoi ?
3. Faire fonctionner l'algorithme à la main pour compresser le texte « trop top »

Exercices

1. Calculer le taux de compression de l'algorithme sur l'exemple précédent.
2. Quelle structure de données est la plus adaptée pour ranger les arbres au fur et à mesure de leur construction, pourquoi ?
3. Faire fonctionner l'algorithme à la main pour compresser le texte « trop top »

Remarques

- On montre que le codage de Huffman est optimal pour un code préfixe.
- Pour des textes « normaux » Le taux de compression de l'algorithme est d'environ 50 % mais les variations peuvent être importantes.

Introduction

- Le nom de cet algorithme provient de celui de ses trois inventeurs : Lemp, Ziv et Welch.

Introduction

- Le nom de cet algorithme provient de celui de ses trois inventeurs : Lemp, Ziv et Welch.
- L'algorithme a l'avantage de pouvoir compresser un texte au fur et à mesure de sa lecture et est donc adapté à la compression d'un flux de données

Introduction

- Le nom de cet algorithme provient de celui de ses trois inventeurs : Lemp, Ziv et Welch.
- L'algorithme a l'avantage de pouvoir compresser un texte au fur et à mesure de sa lecture et est donc adapté à la compression d'un flux de données
- C'est l'algorithme utilisé dans l'utilitaire de compression `compress` et intervient dans le format d'image GIF.

Introduction

- Le nom de cet algorithme provient de celui de ses trois inventeurs : Lemp, Ziv et Welch.
- L'algorithme a l'avantage de pouvoir compresser un texte au fur et à mesure de sa lecture et est donc adapté à la compression d'un flux de données
- C'est l'algorithme utilisé dans l'utilitaire de compression `compress` et intervient dans le format d'image GIF.

Principe

Le principe est d'attribuer un code aux sous chaines (prefixes) rencontrées lors du parcours du texte de façon à disposer d'un code compact si elles se présentent à nouveau.

Illustration sur un exemple

On veut compresser le texte `saisissais` qui ne contient que les 3 lettres `a` `i` et `s`. On commence par attribuer à chaque lettre un code : `a` $\rightarrow 0$, `i` $\rightarrow 1$ et `s` $\rightarrow 2$.

Illustration sur un exemple

On veut compresser le texte `saisissais` qui ne contient que les 3 lettres `a` `i` et `s`. On commence par attribuer à chaque lettre un code : `a` $\rightarrow 0$, `i` $\rightarrow 1$ et `s` $\rightarrow 2$. Puis on parcourt le texte en émettant le code du plus long préfixe rencontré. On crée un nouveau code pour ce prefixe augmenté du caractère le suivant dans le texte (de façon à disposer d'un code lorsqu'on le rencontrera à nouveau.)

Illustration sur un exemple

On veut compresser le texte `saisissais` qui ne contient que les 3 lettres `a` `i` et `s`. On commence par attribuer à chaque lettre un code : `a` $\rightarrow 0$, `i` $\rightarrow 1$ et `s` $\rightarrow 2$. Puis on parcourt le texte en émettant le code du plus long préfixe rencontré. On crée un nouveau code pour ce prefixe augmenté du caractère le suivant dans le texte (de façon à disposer d'un code lorsqu'on le rencontrera à nouveau.)

Position	Emis	Nouveau préfixe
saisissais	2	`sa` $\rightarrow 3$

Illustration sur un exemple

On veut compresser le texte `saisissais` qui ne contient que les 3 lettres `a` `i` et `s`. On commence par attribuer à chaque lettre un code : `a` $\rightarrow 0$, `i` $\rightarrow 1$ et `s` $\rightarrow 2$. Puis on parcourt le texte en émettant le code du plus long préfixe rencontré. On crée un nouveau code pour ce prefixe augmenté du caractère le suivant dans le texte (de façon à disposer d'un code lorsqu'on le rencontrera à nouveau.)

Position	Emis	Nouveau préfixe
saisissais	2	`sa` $\rightarrow 3$
saisissais	0	`ai` $\rightarrow 4$

Illustration sur un exemple

On veut compresser le texte `saisissais` qui ne contient que les 3 lettres `a` `i` et `s`. On commence par attribuer à chaque lettre un code : `a` $\rightarrow 0$, `i` $\rightarrow 1$ et `s` $\rightarrow 2$. Puis on parcourt le texte en émettant le code du plus long préfixe rencontré. On crée un nouveau code pour ce prefixe augmenté du caractère le suivant dans le texte (de façon à disposer d'un code lorsqu'on le rencontrera à nouveau.)

Position	Emis	Nouveau préfixe
saisissais	2	`sa` $\rightarrow 3$
saisissais	0	`ai` $\rightarrow 4$
saisissais	1	`is` $\rightarrow 5$

Illustration sur un exemple

On veut compresser le texte `saisissais` qui ne contient que les 3 lettres `a` `i` et `s`. On commence par attribuer à chaque lettre un code : `a` $\rightarrow 0$, `i` $\rightarrow 1$ et `s` $\rightarrow 2$. Puis on parcourt le texte en émettant le code du plus long préfixe rencontré. On crée un nouveau code pour ce prefixe augmenté du caractère le suivant dans le texte (de façon à disposer d'un code lorsqu'on le rencontrera à nouveau.)

Position	Emis	Nouveau préfixe
`saisissais`	2	`sa` $\rightarrow 3$
`saisissais`	0	`ai` $\rightarrow 4$
`saisissais`	1	`is` $\rightarrow 5$
`saisissais`	2	`si` $\rightarrow 6$

Illustration sur un exemple

On veut compresser le texte `saisissais` qui ne contient que les 3 lettres `a` `i` et `s`. On commence par attribuer à chaque lettre un code : `a` $\to 0$, `i` $\to 1$ et `s` $\to 2$. Puis on parcourt le texte en émettant le code du plus long préfixe rencontré. On crée un nouveau code pour ce prefixe augmenté du caractère le suivant dans le texte (de façon à disposer d'un code lorsqu'on le rencontrera à nouveau.)

Position	Emis	Nouveau préfixe
saisissais	2	`sa` $\to 3$
saisissais	0	`ai` $\to 4$
saisissais	1	`is` $\to 5$
saisissais	2	`si` $\to 6$
saisissais	5	`iss` $\to 7$

Illustration sur un exemple

On veut compresser le texte `saisissais` qui ne contient que les 3 lettres `a` `i` et `s`. On commence par attribuer à chaque lettre un code : `a` $\to 0$, `i` $\to 1$ et `s` $\to 2$. Puis on parcourt le texte en émettant le code du plus long préfixe rencontré. On crée un nouveau code pour ce prefixe augmenté du caractère le suivant dans le texte (de façon à disposer d'un code lorsqu'on le rencontrera à nouveau.)

Position	Emis	Nouveau préfixe
saisissais	2	`sa` $\to 3$
saisissais	0	`ai` $\to 4$
saisissais	1	`is` $\to 5$
saisissais	2	`si` $\to 6$
saisissais	5	`iss` $\to 7$
saisissais	3	`sai` $\to 8$
saisissais	5	

Illustration sur un exemple

On veut compresser le texte `saisissais` qui ne contient que les 3 lettres `a` `i` et `s`. On commence par attribuer à chaque lettre un code : `a` $\rightarrow 0$, `i` $\rightarrow 1$ et `s` $\rightarrow 2$. Puis on parcourt le texte en émettant le code du plus long préfixe rencontré. On crée un nouveau code pour ce prefixe augmenté du caractère le suivant dans le texte (de façon à disposer d'un code lorsqu'on le rencontrera à nouveau.)

Position	Emis	Nouveau préfixe
saisissais	2	`sa` $\rightarrow 3$
saisissais	0	`ai` $\rightarrow 4$
saisissais	1	`is` $\rightarrow 5$
saisissais	2	`si` $\rightarrow 6$
saisissais	5	`iss` $\rightarrow 7$
saisissais	3	`sai` $\rightarrow 8$
saisissais	5	

Codage initial : 2012122012

Illustration sur un exemple

On veut compresser le texte `saisissais` qui ne contient que les 3 lettres `a` `i` et `s`. On commence par attribuer à chaque lettre un code : `a` $\to 0$, `i` $\to 1$ et `s` $\to 2$. Puis on parcourt le texte en émettant le code du plus long préfixe rencontré. On crée un nouveau code pour ce prefixe augmenté du caractère le suivant dans le texte (de façon à disposer d'un code lorsqu'on le rencontrera à nouveau.)

Position	Emis	Nouveau préfixe
saisissais	2	`sa` $\to 3$
saisissais	0	`ai` $\to 4$
saisissais	1	`is` $\to 5$
saisissais	2	`si` $\to 6$
saisissais	5	`iss` $\to 7$
saisissais	3	`sai` $\to 8$
saisissais	5	

Codage initial : 2012122012

Codage compressé : 2012535

Exercice

Compresser le texte `blablabla` en attribuant les codes de départ suivant :
`a` $\rightarrow 0$, `b` $\rightarrow 1$ et `l` $\rightarrow 2$.

Exercice

Compresser le texte `blablabla` en attribuant les codes de départ suivant :
`a` $\rightarrow 0$, `b` $\rightarrow 1$ et `l` $\rightarrow 2$.
On obtient la liste de codes : `[1; 2; 0; 3; 5; 4;]`

Exercice

Compresser le texte `blablabla` en attribuant les codes de départ suivant : `a` $\rightarrow 0$, `b` $\rightarrow 1$ et `l` $\rightarrow 2$.
On obtient la liste de codes : `[1; 2; 0; 3; 5; 4;]`

Principe de la décompression

On peut reconstruire la table de codage au fur et à mesure de la décompression. On n'a donc pas besoin de joindre la table de codage des prefixes au fichier compressé. Ce dernier contient déjà tous les éléments qui permettront sa décompression.

Exercice

Compresser le texte `blablabla` en attribuant les codes de départ suivant : `a` $\rightarrow 0$, `b` $\rightarrow 1$ et `l` $\rightarrow 2$.
On obtient la liste de codes : `[1; 2; 0; 3; 5; 4;]`

Principe de la décompression

On peut reconstruire la table de codage au fur et à mesure de la décompression. On n'a donc pas besoin de joindre la table de codage des prefixes au fichier compressé. Ce dernier contient déjà tous les éléments qui permettront sa décompression.

- On démarre avec la table initiale de codage ne contenant que les caractères

Exercice

Compresser le texte `blablabla` en attribuant les codes de départ suivant : `a` $\rightarrow 0$, `b` $\rightarrow 1$ et `l` $\rightarrow 2$.
On obtient la liste de codes : `[1; 2; 0; 3; 5; 4;]`

Principe de la décompression

On peut reconstruire la table de codage au fur et à mesure de la décompression. On n'a donc pas besoin de joindre la table de codage des prefixes au fichier compressé. Ce dernier contient déjà tous les éléments qui permettront sa décompression.

- On démarre avec la table initiale de codage ne contenant que les caractères
- A chaque émission d'un code, on ajoute dans le table de codage la chaine émise suivie du premier caractère du code suivant.

Exercice

Compresser le texte `blablabla` en attribuant les codes de départ suivant :
`a` $\rightarrow 0$, `b` $\rightarrow 1$ et `l` $\rightarrow 2$.
On obtient la liste de codes : `[1; 2; 0; 3; 5; 4;]`

Principe de la décompression

On peut reconstruire la table de codage au fur et à mesure de la décompression. On n'a donc pas besoin de joindre la table de codage des prefixes au fichier compressé. Ce dernier contient déjà tous les éléments qui permettront sa décompression.

- On démarre avec la table initiale de codage ne contenant que les caractères
- A chaque émission d'un code, on ajoute dans le table de codage la chaine émise suivie du premier caractère du code suivant.
  ⚠ *Sauf dans un cas . . . (voir plus loin)*

Illustration sur un exemple

Table initial de codage : `a` $\rightarrow 0$, `i` $\rightarrow 1$ et `s` $\rightarrow 2$

⚠ Les nouveaux prefixes se contruisent comme étant la chaine émise augmentée du premier caractère du code suivant.

Codage	Décompression	Nouveau préfixe

Illustration sur un exemple

Table initial de codage : `a` $\rightarrow 0$, `i` $\rightarrow 1$ et `s` $\rightarrow 2$

⚠ Les nouveaux prefixes se contruisent comme étant la chaine émise augmentée du premier caractère du code suivant.

Codage	Décompression	Nouveau préfixe
2012535	`s`	`sa` $\rightarrow 3$

Illustration sur un exemple

Table initial de codage : a $\to 0$, i $\to 1$ et s $\to 2$

⚠ Les nouveaux prefixes se contruisent comme étant la chaine émise augmentée du premier caractère du code suivant.

Codage	Décompression	Nouveau préfixe
2012535	s	sa $\to 3$
2012535	sa	ai $\to 4$

Illustration sur un exemple

Table initial de codage : a $\rightarrow 0$, i $\rightarrow 1$ et s $\rightarrow 2$

⚠ Les nouveaux prefixes se contruisent comme étant la chaine émise augmentée du premier caractère du code suivant.

Codage	Décompression	Nouveau préfixe
2012535	s	sa $\rightarrow 3$
2012535	sa	ai $\rightarrow 4$
2012535	sai	is $\rightarrow 5$

Illustration sur un exemple

Table initial de codage : a $\to 0$, i $\to 1$ et s $\to 2$

⚠ Les nouveaux prefixes se contruisent comme étant la chaine émise augmentée du premier caractère du code suivant.

Codage	Décompression	Nouveau préfixe
2012535	s	sa $\to 3$
2012535	sa	ai $\to 4$
2012535	sai	is $\to 5$
2012535	sais	si $\to 6$

Illustration sur un exemple

Table initial de codage : `a` $\rightarrow 0$, `i` $\rightarrow 1$ et `s` $\rightarrow 2$

⚠ Les nouveaux prefixes se contruisent comme étant la chaine émise augmentée du premier caractère du code suivant.

Codage	Décompression	Nouveau préfixe
2012535	s	sa $\rightarrow 3$
2012535	sa	ai $\rightarrow 4$
2012535	sai	is $\rightarrow 5$
2012535	sais	si $\rightarrow 6$
2012535	saisis	sis $\rightarrow 7$

Illustration sur un exemple

Table initial de codage : a $\to 0$, i $\to 1$ et s $\to 2$

⚠ Les nouveaux prefixes se contruisent comme étant la chaine émise augmentée du premier caractère du code suivant.

Codage	Décompression	Nouveau préfixe
2012535	s	sa $\to 3$
2012535	sa	ai $\to 4$
2012535	sai	is $\to 5$
2012535	sais	si $\to 6$
2012535	saisis	sis $\to 7$
2012535	saisissa	sai $\to 8$

Illustration sur un exemple

Table initial de codage : a $\to 0$, i $\to 1$ et s $\to 2$

⚠ Les nouveaux prefixes se contruisent comme étant la chaine émise augmentée du premier caractère du code suivant.

Codage	Décompression	Nouveau préfixe
2012535	s	sa $\to 3$
2012535	sa	ai $\to 4$
2012535	sai	is $\to 5$
2012535	sais	si $\to 6$
2012535	saisis	sis $\to 7$
2012535	saisissa	sai $\to 8$
2012535	saisissais	

Exercice

On rappelle que la compression de `blablabla` en attribuant les codes de départ : `a` $\to 0$, `b` $\to 1$ et `l` $\to 2$ donne la liste de codes : `[1; 2; 0; 3; 5; 4;]`.
Montrer qu'un décompressant cette liste de codes en suivant l'algorithme décrit plus haut, on retrouve bien le texte de départ.

Exercice

On rappelle que la compression de `blablabla` en attribuant les codes de départ : `a` $\to 0$, `b` $\to 1$ et `l` $\to 2$ donne la liste de codes : `[1; 2; 0; 3; 5; 4;]`.
Montrer qu'un décompressant cette liste de codes en suivant l'algorithme décrit plus haut, on retrouve bien le texte de départ.

Reste une subtilité . . .

Exercice

On rappelle que la compression de `blablabla` en attribuant les codes de départ : `a` $\to 0$, `b` $\to 1$ et `l` $\to 2$ donne la liste de codes : `[1; 2; 0; 3; 5; 4;]`.
Montrer qu'un décompressant cette liste de codes en suivant l'algorithme décrit plus haut, on retrouve bien le texte de départ.

Reste une subtilité . . .

- Compresser le texte `taratatata` en attribuant les codes de départ suivant : `a` $\to 0$, `r` $\to 1$ et `t` $\to 2$.

Exercice

On rappelle que la compression de `blablabla` en attribuant les codes de départ : `a` $\rightarrow 0$, `b` $\rightarrow 1$ et `l` $\rightarrow 2$ donne la liste de codes : `[1; 2; 0; 3; 5; 4;]`.
Montrer qu'un décompressant cette liste de codes en suivant l'algorithme décrit plus haut, on retrouve bien le texte de départ.

Reste une subtilité . . .

- Compresser le texte `taratatata` en attribuant les codes de départ suivant : `a` $\rightarrow 0$, `r` $\rightarrow 1$ et `t` $\rightarrow 2$.
  On obtient la liste de codes : `[2; 0; 1; 0; 3; 7; 0]`

Exercice

On rappelle que la compression de `blablabla` en attribuant les codes de départ : `a` $\to 0$, `b` $\to 1$ et `l` $\to 2$ donne la liste de codes : `[1; 2; 0; 3; 5; 4;]`.
Montrer qu'un décompressant cette liste de codes en suivant l'algorithme décrit plus haut, on retrouve bien le texte de départ.

Reste une subtilité . . .

- Compresser le texte `taratatata` en attribuant les codes de départ suivant : `a` $\to 0$, `r` $\to 1$ et `t` $\to 2$.
  On obtient la liste de codes : `[2; 0; 1; 0; 3; 7; 0]`
- Décompresser la liste obtenue, quel situation (qui n'était pas apparue dans les exemples précédents) se produit ?

Exercice

On rappelle que la compression de `blablabla` en attribuant les codes de départ : `a` $\to 0$, `b` $\to 1$ et `l` $\to 2$ donne la liste de codes : `[1; 2; 0; 3; 5; 4;]`.
Montrer qu'un décompressant cette liste de codes en suivant l'algorithme décrit plus haut, on retrouve bien le texte de départ.

Reste une subtilité . . .

- Compresser le texte `taratatata` en attribuant les codes de départ suivant : `a` $\to 0$, `r` $\to 1$ et `t` $\to 2$.
  On obtient la liste de codes : `[2; 0; 1; 0; 3; 7; 0]`
- Décompresser la liste obtenue, quel situation (qui n'était pas apparue dans les exemples précédents) se produit ?
- Modifier l'algorithme de décompression initial afin de traiter ce cas.

Implémentation

L'implémentation détaillée en OCaml, sera vue en TP, on utilise les tables de hachage du module `Hashtbl` afin de stocker les codes des prefixes, c'est-à-dire que les clés de la table sont les préfixes (donc des chaines de caractères) et les valeurs les codes associées.