

# Chapitre 22

Chaînes de caractères.

## Sommaire.

<b>1</b>	<b>Caractères.</b>	<b>1</b>
<b>2</b>	<b>Chaînes de caractères.</b>	<b>1</b>
2.1	En C.	1
2.2	En OCaml.	1
<b>3</b>	<b>Entrées et Sorties.</b>	<b>1</b>
3.1	En C.	1
3.2	En OCaml.	1
<b>4</b>	<b>Fichiers textuels.</b>	<b>1</b>
4.1	En C.	1
4.2	En OCaml.	1
<b>5</b>	<b>Recherche de motifs.</b>	<b>1</b>
5.1	Notations.	1
5.2	Algorithme naïf.	1
5.3	Algorithme de Boyer-Moore-Horspool ★.	1
5.3.1	Principe.	1
5.3.2	Table de saut.	1
5.3.3	Algorithme.	1
5.4	Algorithme de Rabin-Karp.	1
<b>6</b>	<b>Compression.</b>	<b>1</b>
6.1	Lempel-Ziv-Welch.	1
6.2	Codage de Huffman.	1

Les propositions marquées de ★ sont au programme de colles.

## 1 Caractères.

### Définition 1: Caractère.

Ils sont codés sur un octet en OCaml, les normes garantissent le code ASCII.  
Leur type est `char` en C et en OCaml, le lien entre le code ASCII et le caractère peut être obtenu par conversion explicite en C, et avec `char_of_int` ou `int_of_char` en OCaml.

Il existe des opérations entre caractères : `<`, `<=`, `>`, `>=`, `==`, `!=`, `=`, `<>`.

On a l'assurance que les minuscules, majuscules et chiffres soient dans l'ordre.

Affichage : `"%c"` en C, `print_char` en OCaml.

## 2 Chaînes de caractères.

Les constantes littérales seront entre guillemets.

### 2.1 En C.

#### Définition 2

Il n'existe pas de chaîne de caractère en C, on appelle chaîne de caractères une adresse où sont stockés des caractères.

Affichage : `printf("%s", chaine);` ou bien `printf("chaine");`

### 2.2 En OCaml.

#### Définition 3

Le type est `string`.

Pour accéder au *i*ème caratère, on utilise `chaine.[i]`.

Les indices commencent à 0, les accès sont en temps constant. Les chaînes de caractère sont immuables.

Affichage : `print_string chaine;` ou bien `print_endline chaine;`.

Concaténation : `chaine1 ^ chaine2`.

Longueur : `String.length chaine`.

## 3 Entrées et Sorties.

### Définition 4: Flux de données.

Un **flux de données** est une transmissions ininterrompue d'une suite de données.

Un processus a trois flux standards :

— `<` : entrée standard.

— `>` : sortie standard.

— `2>` : sortie d'erreur.

### 3.1 En C.

#### Définition 5

Codes usuels : entiers : `%d`, flottants : `%f`, caractères : `%c`, pointeurs : `%p`.

Sortie standard : `printf(const char*, ...)`

Elle bufferise par lignes.

Sortie erreur : `fprintf(FILE*, const char*, ...);` avec `stderr` pour `FILE*`.

Elle ne bufferise pas.

Lire sur l'entrée standard : `scanf(const char*, ...);`

### 3.2 En OCaml.

#### Définition 6

Sortie standard : `print_string`, `print_int`, `print_float`, ...

Sortie erreur : `output_string : out_channel -> string -> unit` avec `stderr` pour `out_channel`.

Lire sur l'entrée standard : `read_line`.

Ces fonctions sont définies dans le module `Stdlib` ouvert par défaut.

## 4 Fichiers textuels.

### 4.1 En C.

#### Définition 7

On utilise `fopen(const char*, const char*)`; en spécifiant le chemin du fichier et le mode d'ouverture.

Lecture : `fscanf(FILE*, const char*, ...)`;

Écriture : `fprintf(FILE*, const char*, ...)`;

Fermeture : `fclose(FILE*)`;

En cas d'oubli de fermeture, on épuise les ressources du système et les écritures peuvent ne pas se faire.

### 4.2 En OCaml.

#### Définition 8

Même principe :

Ouverture : `open_in` en lecture, `open_out` en écriture.

Lecture : `input_line` ou bien `input_char`.

Écriture : `output_string` ou bien `output_char`.

Fermeture : `close_in` ou bien `close_out`.

En fin de fichier, on obtient l'exception `End_of_file`.

## 5 Recherche de motifs.

### 5.1 Notations.

#### Notation

Soit *T* un texte de longueur *n*, *M* un motif de longueur *m* ≤ *n*, sur un même alphabet *ℳ*.

On utilise les notations de python sur les chaînes de caractère.

On cherche *S* = {*s* ∈ ℕ | ∀*i* ≤ *m* − 1, *T*[*s* + *i*] = *M*[*i*]}.

### 5.2 Algorithme naïf.

#### Définition 9

##### Algorithme 1 : Algorithme Naïf

**Entrées** : Un texte *T*, un motif *M*

**Sorties** : L'ensemble *S*

*S* ← ∅

**pour** *s* allant de 0 à *n* − *m* **faire**

**si** *M* = *T*[*s* : *s* + *m*] **alors**

*S* ← *S* ∪ {*s*}

**fin**

**fin**

**retourner** *S*

Dans le pire des cas, cet algorithme a une complexité en  $\Theta(m \cdot n)$ .

### 5.3 Algorithme de Boyer-Moore-Horspool ★.

#### 5.3.1 Principe.

#### Définition 10

En comparant de droite à gauche, on peut déplacer le motif plus rapidement.

On applique un prétraitement sur le motif, pour connaître la dernière occurence de chaque caractère.

#### 5.3.2 Table de saut.

#### Définition 11

Indexée par l'alphabet, la case *c* donne l'indice de l'occurence la plus à droite dans *M* privé de sa dernière lettre du caractère *c*, sinon −1.

Ce prétraitement s'effectue avec une complexité en  $O(\max(n, m))$ .

#### 5.3.3 Algorithme.

#### Définition 12

##### Algorithme 2 : Construction de la table de saut

**Entrées** : Un motif *M*

**Sorties** : Table de saut sur *M*

*T* ← table de saut vide.

**pour** *c* dans *ℳ* **faire**

    | *T*[*c*] ← −1.

**fin**

**pour** *i* allant de 0 à *m* − 2 **faire**

    | *T*[*M*[*i*]] ← *i*.

**fin**

**retourner** *T*.

##### Algorithme 3 : Algorithme de Boyer-Moore-Horspool

**Entrées** : Un texte *T*, un motif *M*

**Sorties** : L'ensemble *S*

*S* ← ∅.

*T* ← table de saut sur *M*.

*s* ← 0.

**tant que** *s* ≤ *n* − *m* **faire**

    | *i* ← *m* − 1.

**tant que** *i* ≥ 0 et *M*[*i*] = *T*[*s* + *i*] **faire**

        | *i* ← *i* − 1.

**fin**

**si** *i* = −1 **alors**

        | *S* ← *S* ∪ {*s*}.

**fin**

    | *s* ← *s* + *m* − max(1, *i* − *T*[*s* + *m* − 1]).

**fin**

**retourner** *S*.

### 5.4 Algorithme de Rabin-Karp.

Sera ajouté plus tard (?).

## 6 Compression.

### 6.1 Lempel-Ziv-Welch.

#### Définition 13: Algorithme Lempel-Ziv-Welch

**Principe:** parcours de gauche à droite du texte à compresser en maintenant un dictionnaire qui à un facteur du texte associe un code, qu'on suppose être le code ASCII.

##### Algorithme 4 : LZW Compression

**Entrées** : Un texte

**Sorties** : Code compressé du texte

Créer un dictionnaire *D*.

Mettre les codes ASCII des caractères dans le dictionnaire.

*m* ← mot vide.

**tant que** il y a un caractère à lire **faire**

    | Soit *x* ce caractère.

**si** *mx* est une clé de *D* **alors**

        | *m* ← *mx*.

**fin**

**sinon**

        | Ajouter *mx* à *D* avec la première valeur non utilisée.

        | Écrire le code de *m* sur la sortie.

        | *m* ← *x*.

**fin**

**fin**

    | écrire le code de *m*.

##### Algorithme 5 : LZW Décompression

**Entrées** : Un code

**Sorties** : Texte décompressé

Créer un dictionnaire *D* double associant un code à une lettre.

Remplir *D* avec le code ASCII.

*m* ← mot vide.

**tant que** il y a un code à lire **faire**

    | Soit *x* ce code.

**si** *x* est une clé de *D* **alors**

        | *x*<sub>0</sub> ← le premier caractère du mot associé à *x*.

**si** *mx*<sub>0</sub> ∉ *D* **alors**

            | Ajouter *mx*<sub>0</sub> à *D* avec la première valeur non attribuée.

            | Écrire *m* sur la sortie.

**fin**

**fin**

**sinon**

        | *m*<sub>0</sub> ← le premier caractère de *m*.

        | Ajouter *mm*<sub>0</sub> à *D* avec la première valeur non attribuée.

        | Écrire *mm* sur la sortie.

**fin**

    | *m* ← le mot associé à *x*.

**fin**

    | écrire *m* sur la sortie.

### 6.2 Codage de Huffman.

#### Définition 14: ★

##### Algorithme 6 : Codage de Huffman

**Entrées** : Un texte

**Sorties** : Son encodage

Mettre les couples (caractère, fréquence) dans une chaîne de priorité.

**tant que** file contient plusieurs éléments **faire**

    | Extraire les 2 éléments de plus basses fréquences.

    | Créer un noeud binaire dont ces éléments sont les fils.

    | Insérer ce noeud dans la file, avec pour fréquence la somme des fréquences de ses fils.

**fin**

**retourner** l'élément de la file.