# Algorithmique du texte #1

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

# Table des matières

# Alphabet etc.

## Alphabet

Un *alphabet* est un ensemble fini de symboles, appelés *lettres* (ou *caractères*, ou…)

- Exemple : $\{a, \alpha, \nu, 1\}$

## Mot

Un *mot* sur un alphabet $\Sigma$ est une suite finie (éventuellement vide) de lettres de $\Sigma$

- Exemple : *martre*
- Exemple : le mot vide (généralement noté $\varepsilon$)

On note $\Sigma^*$ l'ensemble (infini) des mots finis sur $\Sigma$, et $|u|$ la *longueur* d'un mot (son nombre de termes)

## Langage

Un *langage* est un ensemble (possiblement vide, possiblement **infini**) de mots sur un même alphabet

- Exemple : l'ensemble infini des mots écrits uniquement avec un *a*

# Représentation dans un langage de programmation

## En toute généralité

Il suffit de savoir représenter des lettres et ensuite d'utiliser une représentation d'une suite (par ex. un type `list` en OCaml, un tableau en C…)

## Cas particulier de l'alphabet « usuel »

Les langages de programmation fournissent (généralement) des types dédiés pour manipuler lettres & mots

- En OCaml : `char` et `string`
- En C : `char` et `char *`

On parle généralement de *caractères* et de *chaînes de caractères* dans ce contexte

**Attention** : dans le contexte des chaînes de caractère, le type `char *` n'est pas « juste » celui des pointeurs vers `char` (*cf.* ci-dessous)

# Représentation *bis*

## Taille d'un `char`

La taille (en octets) d'un type `char` n'est pas forcément fixe, notamment quand il permet de représenter des caractères *unicode* variés

Pour simplifier, on se limitera en TP à la manipulation de caractères latins non accentués :(

# `char` et `string` en OCaml

## `char`

- Écriture des littéraux entre « quotes » simples, par ex. `'a'`
- Pas de fonctions associées à connaître officiellement
- Dotés d'un ordre total

## `string`

- Écriture des littéraux entre « quotes » doubles, par ex. `"a"`, `"hewwo"`
- Type **immuable**
- Indexation des éléments *de type* `char` par la syntaxe `s.[i]`
- Dotées d'un ordre total hérité des `char` (ordre lexicographique)
- Fonctions de bibliothèque à connaître officiellement :
  - `String.length`
  - `^` (`String.cat`)
  - `print_string`

# `char` et `char *` en C

## `char`

- Écriture des littéraux entre « quotes » simples, par ex. `'a'`
- Formateur pour affichage : `%c`
- Pas de fonctions associées à connaître officiellement

On suppose des `char` de taille un octet, qui peuvent être convertis vers des `uint8_t`

## `char *` : null-terminated strings

Une chaîne de caractères (un mot) $a_0 a_1 \cdots a_{n-1}$ est représentée par un pointeur `s` vers une zone mémoire de taille **n+1** octets tel que pour $i \in [\![n]\!]$, `s[i]` contient le `char` représentant $a_i$, et `s[n]` **vaut le caractère zéro** (ou nul) `0` (ou `'\0'`)

Cet encodage spécifique est supposé pour tous les littéraux de type `char *` et pour toutes les fonctions de la bibliothèque standard opérant sur de tels types

# `char *` *bis*

- Écriture des littéraux entre « quotes » doubles, par ex. `"a"`, `"OHAI"`
- Formateur pour affichage : `%s`
- Les variables de type `char *` **initialisées** sont **immuables**
  - Utilisez un tableau quand la mutabilité est nécessaire, mais attention à la taille !
- Fonctions de bibliothèque de `<string.h>` à connaître :
  - `size_t strlen(const char *s)` : renvoie la longueur de l'argument **sans compter son caractère nul terminal**
  - `char *strcpy(char *dst, const char *src)` : copie `src` dans `dst`, qui doit pointer vers une zone mémoire suffisamment grande **qui n'intersecte pas** `src`, et renvoie `dst`
  - `char *strcat(char *dst, const char *src)` : copie `src` **après** `dst`, qui doit pointer vers une zone mémoire suffisamment grande **qui n'intersecte pas** `src`, et renvoie `dst`
- Fonction de bibliothèque de `<stdlib.h>` à connaître :
  - `int atoi(const char *s)` : convertit de façon non fiable une chaîne de caractères en `int` (par ex. `"1234"` en `1234`)

# Exemple 1

```c
int main(void)
{
    char *g1 = "OHAI";
    char g2[6] = "hello"; // 6 == 5 + 1; g2[5] == 0

    printf("%s %s %lu %lu\n", g1, g2, strlen(g1), strlen(g2));
    // OHAI hello 4 5
    // g1[0] = 'o'; typiquement erreur de segmentation
    g2[2] = 'w';
    g2[3] = 'w';
    printf("%s\n", g2); // hewwo

    return EXIT_SUCCESS;
}
```

# Exemple 2

```c
int main(void)
{
    char *s1 = "il fait ";
    char *s2 = "beau aujourd'hui";
    char *s12 = malloc(strlen(s1) + strlen(s2) + 1);
    strcpy(s12, s1);
    strcat(s12, s2);
    printf("%s\n", s12); // il fait beau aujourd'hui

    return EXIT_SUCCESS;
}
```

# Table des matières

# Facteurs ; sous-mots

On fixe un alphabet $\Sigma$

## Préfixe

Soit $u, p, v$ trois mots t.q. $u = pv$, $p$ est un *préfixe* (ou *facteur gauche*) de $u$

## Suffixe

Soit $u, s, v$ trois mots t.q. $u = vs$, $s$ est un *suffixe* (ou *facteur droit*) de $u$

## Facteur (propre)

Soit $u, p \neq \varepsilon, s \neq \varepsilon, f$ quatre mots t.q. $u = pfs$, $f$ est un *facteur* (ou facteur propre) de $u$

## Sous-mot

Un sous-mot $u'$ d'un mot $u$ est une sous-suite de $u$ : $\forall i \in [\![i, |u'|]\!]$, $u'_i = u_{\varphi(i)}$ pour une certaine fonction strictement croissante $\varphi : [\![1, |u|]\!] \to [\![1, |u|]\!]$

- Exemple : $ab$ ($ba$) est (n'est pas) un sous-mot de *arbre*

# Problème de la recherche de facteur

Soit deux mots $u$, $v$ sur un même alphabet, on veut savoir si $u$ est un facteur de $v$ (et si oui, éventuellement trouver l'une ou plusieurs de ses *occurrences* dans $v$)

- On note $k = |u|$ et $n = |v|$ les tailles des deux entrées

## Algorithme naïf (détection seule)

Pour tous les *décalages* $d \in [\![0, n-k]\!]$ possibles, on teste $\mu(u, v, d) := \bigwedge_{0 \leq i < k} u_i = v_{i+d}$
Dans le pire cas :

- Un test coûte $O(k)$
- Il y a $O(n-k) = O(n)$ décalages à tester

Coût $O(k \times n)$ (« quadratique »)

## Améliorations asymptotiques ?

- En utilisant des automates finis (*cf.* l'année prochaine)
- Algorithme « de Rabin-Karp »
- Algorithme « de Boyer-Moore »

# Table des matières

# Algorithme « de Rabin-Karp »

## Idée

On applique l'algorithme naïf, mais en essayant d'accélérer les tests (le calcul de $\mu(u, v, d)$)

## Une approche à base de fonction de hachage

On peut utiliser une fonction de hachage $H$ et :

- Précalculer $h_u := H(u)$
- Pour $d \in [\![0, n-k]\!]$, calculer $h_{vd} := H(v_d v_{d+1} \cdots v_{d+k-1})$
- Si $h_u \neq h_{vd}$ on a nécessairement $\neg\mu(u, v, d)$
- Sinon on ne peut pas conclure, et l'on teste comme dans l'algorithme naïf

## Analyse de coût préliminaire

- Toujours $O(n)$ décalages à tester dans le pire cas
- Coût total $O(k + t \times n)$, où $t$ est le coût *amorti* (éventuellement *espéré*) d'un test infructueux
- On gagne quelque chose si $t = o(k)$

# RK *bis*

## Conditions informelles sur $H$

- $H(s)$ doit dépendre de *toutes les lettres* de $s$
  - Sinon il faut systématiquement faire un test complet quand certaines lettres ne correspondent pas
- Le coût (amorti) d'un calcul de $H(s)$ doit être un $o(|s|)$ !

## Fonction de hachage *cyclique* (ou *incrémentale*)

On dit informellement d'une fonction de hachage $H : \Sigma^n \to I$ qu'elle est *cyclique* si étant donnés $H(s_0 \cdots s_{n-1})$ et $s_n$, le calcul de $H(s_1 \cdots s_n)$ **peut se faire en temps** $O(1)$

## Utilité

Si $H$ est une fonction de hachage cyclique (et t.q. calculer $H(s)$ a un coût linéaire en $|s|$), le coût total du calcul de $H(v_0 \cdots)$, $H(v_1 \cdots)$, …, $H(v_{n-k-1} \cdots)$ est $\leq C \times k + (n-k) \times D$ pour certaines constantes $C$, $D$

- On paye $O(k)$ une fois, puis $O(1)$ marginalement pour chaque calcul supplémentaire

# Une famille de fonctions de hachage cyclique

## Préliminaires

On fixe un corps fini $\mathbb{F}$ (par ex. $\mathbb{Z}/p\mathbb{Z}$ pour un certain nombre premier $p$), et un encodage $\varphi$ de $\Sigma$ dans $\mathbb{F}$

- Concrètement, on traite les `char` comme des entiers (par ex. `uint8_t`)

Cet encodage définit un encodage $\Phi : \Sigma^* \to \mathbb{F}[X]$ des mots sur $\Sigma$ dans les polynômes sur $\mathbb{F}$ :

$$\Phi(s_0 \cdots s_{n-1}) = \sum_{i=0}^{n-1} \varphi(s_i) X^{n-i-1}$$

- On « inverse le sens de lecture » de $s$ pour faciliter l'implémentation (*cf.* ci-dessous)

On note $E(P, x)$ l'évaluation d'un polynôme $P$ en $x$

- Rappel : $E$ est linéaire : $E(P + Q, x) = E(P, x) + E(Q, x)$

# Une famille de fonctions de hachage cyclique *bis*

## Hachage polynomial

On définit la *famille* de fonction de hachage $H_x$ paramétrée par $x \in \mathbb{F}^\times$ comme :

$$H_x(s) = E(\Phi(s), x)$$

- On interprète $s$ comme un polynôme, que l'on évalue en le paramètre (non nul) $x$

## $H_x$ est cyclique

Soit $n$ une longueur de mot fixée :

$$H_x(s_1 \cdots s_n) = x \times (H(s_0 \cdots s_{n-1}) - \varphi(s_0) \times x^{n-1}) + \varphi(s_n)$$

ce qui peut se calculer en temps $O(1)$ si l'on suppose que l'on a précalculé $x^{n-1}$

# Une famille de fonctions de hachage cyclique *ter*

## $H_x(s)$ est calculable en temps $O(|s|)$

On peut évaluer un polynôme de façon cyclique (méthode « de Horner ») :

$$E(P, x) = (\cdots((P_n) \times x + P_{n-1}) \times x + \cdots) \times x + P_0$$

## $H$ a de bonnes propriétés statistiques 

Soit $u$, $v \neq u$ deux mots de même longueur $n$, la probabilité sur un choix uniforme de $x$ que $H_x(u) = H_x(v)$ est « faible »

$$\Pr_{x \sim \mathbb{F}^\times} H_x(u) = H_x(v) = \Pr_{x \sim \mathbb{F}^\times} H_x(u - v) = 0 \leq (n-1)/(\#\mathbb{F} - 1)$$

- Par linéarité de $H_x$ et la majoration du nombre de racines d'un polynôme non nul de degré $n - 1$

# RK *ter* : bilan avec $H_x$

Soit $u$, $v$ mots de longueurs $k$ et $n \geq k$, et $H_x : \Sigma^k \to \mathbb{F}$ comme précédemment, en prenant $\mathbb{F} = \mathbb{Z}/p\mathbb{Z}$ pour un certain nombre premier $p$

- Précalculer $H_x(u)$, $H_x(v_0 \cdots v_{k-1})$ et $x^{k-1}$ coûte un $O(k)$
- Calculer $H_x(v_1 \cdots v_k)$, …, $H_x(v_{n-k} \cdots v_{n-k-1})$ coûte un $O(n-k)$
- Calculer $\mu(u, v, d)$ coûte $O(k)$ si $h_u = h_{vd}$ (c'est une **collision** quand les mots sont en fait différents) et $O(1)$ sinon
- La probabilité (sur $x$) que $h_u = h_{vd} \leq (k-1)/(p-1)$ (pour des mots différents) ; l'espérance du nombre de collisions *quand u n'est pas facteur de v* (ce qui correspond au pire cas) est $(n-k) \times (k-1)/(p-1)$

L'espérance du coût est un $O(k + (n-k) \times (1 + k^2/p))$

RK est un algorithme probabiliste de type *Las Vegas*

# RK *quater* : en pratique

- On peut facilement prendre $p$ relativement grand (par ex. $2^{31} - 1$ en TP) : on peut approcher le coût en pratique par un $O(k + (n - k)) = O(n)$ pour les facteurs de taille « raisonnable »
  - Si vous recherchez un mot dans un texte, il est peut probable que celui-ci fasse plus d'un million de caractères…
- Les opérations modulo $p$ nécessaires pour le calcul de $H_x$ peuvent cacher **une grosse constante** si elles sont implémentées avec des « vraies » divisions
  - L'équivalent de plusieurs dizaines de comparaisons de caractères, ce qui rend l'approche tout à fait inintéressante si $k < 50$
  - Heureusement les compilateurs modernes optimisent tout ça si $p$ est une constante connue à la compilation
  - On peut aussi choisir $p$ exprès pour garantir un bon coût à la main sans dépendre du compilateur, *cf.* TP (bonus)

# RK *quinquies* : bonus : recherche de facteurs multiples

L'algorithme s'adapte facilement à la recherche **en une seule passe** de facteurs multiples

- On précalcule les empreintes par $H_x$ de tous les facteurs, on les stocke dans une structure de recherche efficace, et l'on teste chaque décalage pour chaque motif de même empreinte
- (Juste un peu pénible si les facteurs n'ont pas tous la même taille)
- Espérance du coût $O(f \times k + (n-k) \times (1 + f \times k^2/p))$ pour $f$ facteurs (version « simple recherche »)

# Table des matières

# Algorithme « de Boyer-Moore »

- L'algorithme « de Rabin-Karp » résout le problème de recherche de facteur de façon élégante et efficace avec un coût (espéré) optimal (sans supposition sur les entrées) quand $k = O(\sqrt{p})$
- L'algorithme « de Boyer-Moore » propose une solution alternative déterministe qui dans certains cas s'exécute en coût $O(n/k)$ (« sous-linéaire » !), mais qui n'améliore pas l'algorithme naïf dans le pire cas

## Principe à haut niveau

On précalcule une table qui dépend uniquement de $u$, et l'on utilise celle-ci pendant la recherche « naïve » pour ne pas tester certains décalages que l'on sait ne pas pouvoir fonctionner

# Table de décalage

## Définition

Soit $u$ le mot à rechercher, on définit une fonction partielle $T_u : \Sigma \to \mathbb{N}$ qui associe à chaque lettre présente dans $u$ l'indice de sa **dernière** *occurrence* dans $u$

- En pratique il est confortable d'étendre cette fonction à $\{-1\} \cup \mathbb{N}$ pour la rendre totale en définissant $T_u(x) = -1$ pour tout $x \notin u$

## Utilité

On note $v^d = v_d v_{d+1} \cdots v_{d+k-1}$ le sous-mot de longueur $k$ de $v$ décalé de $d$ positions « vers la droite » ; alors pour tout $i$ et $j = T_u(v_i)$, si l'on souhaite « aligner » $u$ avec $v$ en « incluant » $v_i$ en supposant une absence de débordement :

- On considère *a priori* les $v^d$ pour $d \in [\![i - k + 1, i]\!]$
- On peut exclure les décalages qui « commencent trop loin » pour que $v_i$ apparaisse dans $u$
- Ce sont les $d$ tels que $d + j < i$
  - Aussi vrai quand $j = -1$
- On peut restreindre la recherche à $d \in [\![i - j, i]\!]$
  - Si $j = -1$ il n'y a rien à tester ; si $j = k - 1$ on ne gagne rien

# Exemples

# BM *bis* : description

L'algorithme « de Boyer-Moore » utilise une table de décalage pour accélérer la recherche naïve :

- On précalcule une représentation de $T_u$ t.q. l'on peut ensuite calculer $T_u(c)$ en temps (éventuellement espéré) $O(1)$
  - Faisable en temps $O(|u|)$ avec une table de hachage, ou en temps $O(\#\Sigma)$ avec un tableau (à supposer que l'on dispose d'une bijection $\Sigma \to [\![\#\Sigma]\!]$ efficacement calculable)
- On initialise ensuite un décalage $d$ à 0, et tant que l'on n'a pas décidé si $u$ est facteur de $v$ :
  - On teste si $v^d = u$ en les lisant **de droite à gauche** (en les comparant caractère par caractère par indice décroissant)
  - S'il $v^d = u$ on termine — $u$ est un facteur de $v$
  - Sinon soit $i$ le plus grand indice tel que $v_i^d \neq u_i$, — trouvé en temps $O(i)$
    on augmente $d$ de $\max(1, i - T_u(v_i^d))$ et l'on recommence

    $i - T_u(v_i^d)$ peut être négatif par ex. si $T_u(v_i^d) > i$ et que l'on avait $v^d_{T_u(v_i^d)} = v_i^d$

# BM *ter* : exemple d'exécution

# BM *quater*

## Analyse de coût

- Dans le pire cas, pour une majoration brutale le premier désaccord entre $v^d$ et $u$ se trouve systématiquement à $v^d_j$ pour $j = O(1)$, et chaque itération coûte un $O(k)$ et fait avancer d'un $O(1)$ : pas mieux que l'algorithme naïf :(
- Dans le meilleur cas, on a systématiquement pour $j = O(1)$ que $v^d_{k-j}$ n'est pas une lettre de $u$, et chaque itération coûte un $O(1)$ et fait avancer de $O(k)$ : coût « sous-linéaire » en $O(n/k)$ !
- En pratique : ???

## Remarque

Il existe plusieurs variantes de l'algorithme « de Boyer-Moore », dont certaines sont plus sophistiquées que celle-ci. On a juste présenté la version *a priori* au programme