1. Recherche de motifs dans un texte

1.1. Présentation du problème

On se place sur un alphabet Σ (un ensemble de lettres/symboles).

1.1.1. Entrées du problème

On se donne deux chaînes de caractères t (un texte) et m (un motif).

On peut préciser l'aphabet Σ

1.1.2. Sortie du problème

Si m apparaît dans t, on veut récupérer l'indice de sa première occurence (en gros).

1.1.3. Taille de l'entrée

On notera:

- n la taille (le nombre de caractères) de t
- p la taille de m
- k la taille de Σ

On aura toujours $p < n$ et k petit devant n

Exemple

- $\Sigma = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$
- $t = abbcbbaacbba$
- $m = bba$

On a $t = abbc \boxed{bba}_4 \, ac \boxed{bba}_9$

On renvoit 4.

1.2. Approche naive

1.2.1. Implémentation

```
Pour i allant de 0 à n - p - 1 faire
    j := 0

    Tant que j < p et t[i + j] = m[j], j <- j + 1

    Si j = p, Renvoyer Vrai
Fin
```

1.2.2. Complexité

Dans le pire cas: $O(p(n-p))$

Le pire cas est rare et s'amorti en $O(n)$

1.3. Algorithme de Boyer-Moore

1.3.0. Deux principes

Si une comparaison entre t_i et m_i échoue:

(i) Si $t_i \notin m$, on peut directement décaler m pour qu'il ne soit plus en face de t_i

(ii) Si $t_i \in m$, on peut directement décaler m pour qu'il soit en face de deux lettres qui sont les mêmes.

1.3.1. Table de saut

On veut savoir de combien décaler. La table de saut est une table qui à chaque lettre associe l'indice de la dernière occurence de la lettre dans le motif.

Exemple

- $m =$ mots

Lettre	Indice
m	0
o	1
t	2

- $m = ababaac$ sur $\Sigma = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}$

Lettre	Indice
a	5
b	3

Implémentation Avec une table de hachage (ou un arbre binaire de recherche) qui a comme clés les lettres et comme valeurs les indices dans le motif.

Si on n'a pas de table de hachage, on peut le faire dans un tableau en numérotant les lettres et juste utiliser un tableau. Les lettres ASCII sont déjà numérotées.

1.3.2.

On compare les lettres de t et m en allant de $i + p - 1$ jusqu'à i.

On suppose qu'à l'indice j on rencontre un problème $t[i + j] \neq m[j]$.

- Si $t[i + j] \notin m$, alors on reprend à $i' = i + j$
- Si $t[i + j] \in m$ à un indice $l > j$, décaler pour mettre en face $t[i + j]$ et $m[l]$ revient à pousser m vers la gauche et le principe de l'algorithme est qu'on a déjà vérifié cette possibilité. On a pas d'information utile, on décale de 1. Donc $i' = i + 1$
- Si $t[i + j] \in m$ à un indice $l < j$, alors on décale pour mettre $t[i + j]$ en face de $m[l]$. On aura $i' + l = i + j \iff i' = i + j - l$

1.3.3. Algorithme

Pseudo-Code

```
Fonction Boyer-Moore(t, m):
    Calculer la table de sauts
    i := 0

    Tant que i < n faire
        j := p - 1

        Tant que t[i + j] = m[j] faire
            j <- j - 1

            Si j = - 1 faire
                Retourner i
            Fin
        Fin

        Trouver le nouveau i selon les trois cas
    Fin

    Renvoyer -1
Fin
```

1.3.4. Implémentation

En OCaml

```
let calcule_table_saut m p =
    let table = Hashtbl.create 256 in
    for i = p - 2 downto 0 do
        if not (Hashtbl.mem table m.[i]) then (
            Hashtbl.add table m.[i] i
        )
    done;

    table

let calcule_table_saut_bis m p =
    let tab = String.make 256 (-1) in
    for i = 0 to p - 2 do
        tab.(int_of_char m.[i]) <- i
    done;

and nouveau_i table t m i j p =
    let res = ref i in
    if not (Hashtbl.mem table t.[i+j] m) then
        res := i + j + 1
    else (
        let l = Hashtbl.find t m.[i+j] in
        res := if l > j then i + 1 else i + j - l
    )
    !res
in

let boyer_moore t m =
    let p = String.lenght t and n = String.length m
    and res = ref (-1) and i = ref 0
    and table = calcule_table_saut m p in

    while !res = (-1) && !i <= n do
        let j = ref (p - 1) in
        while t.[!i + !j] = m.[!j] do
            j := !j - 1;
        done;
        if !j = -1 then res := !i
        else i := nouveau_i table t m i j p
    done;
    !res
;;
```

1.3.5. Terminaison

`j` est un variant de la boucle `Tant que` interne.

`n - i` est un variant de la boucle externe car `nouveau_i` renvoit toujours `r > i`

1.3.6. Correction

On ne renvoie un indice $i \neq -1$ que si on a testé que

$$\forall j \in [\![0, p-1]\!],\ t[i+j] = m[j]$$

De plus les sauts réalisés ne sautent jamais des indices qui auraient pu être solution.

1.3.7. Complexité

Coût de la creation de la table de saut:

- Une création de table de taille $O(k) = O(|\Sigma|)$
- Une boucle `Pour` avec juste des tests de présence dans la table de hachage et des ajouts à celle-ci donc $O(p)$

  Au total on a un $O(\max(k, p))$

`nouveau_i` est en $O(1)$

On peut espérer en géneral ne faire que quelques comparaisons avant de se rendre compte le motif n'est pas à cet indice.

On peut aussi espérer faire des sauts grands (donc de taille proche de p) à chaque tour de la boucl eexterne et donc avoir un nombre de tours de boucle proche de $\frac{n}{p}$

Donc au mieux on peut avoir une complexité en $O\Big(\max(k, p) + \frac{n}{p}\Big)$ et $O(np)$ dans le pire cas.

Remarque

L'algorithme présenté s'appelle en réalité Boyer-Moore-Horspool. Le vrai algorithme de Boyer-Moore utilise une deuxième table de hachage.

1.4. Algorithme de Rabin - Karp

1.4.1. Principe

Résoudre la recherche de motifs dans un texte en utilisant une fonction de hachage.

On considère:

$$h_n : E' \to E$$

où E' désigne l'ensemble des mots de longueur p.

E s'appelle l'ensemble des empreintes, on suppose qu'on peut tester l'égalité de deux élements de E en temps constant.

On considère un des facteurs de taille p de t, $t[i, i+p-1]$ on calcule son haché $h\Big(t[i, i+p-1]\Big)$

et on peut déterminer si ce n'est pas m en testant $h\Big(t[i, i+p-1]) = h(m)\Big)$.

Si on a pas égalité, on a pas trouvé le motif.

Si on a égalité, on a peut être trouvé le motif mais ça peut être une collision. Il faut alors tester lettre par lettre.

1.4.2. Une bonne fonction de hachage

Les fonctions de hachage injectives, c'est en géneral illusoir.

On va utiliser une fonction de hachage qui s'incrémente en temps constant, c'est à dire qu'on peut calculer $h\Big(t[i, i+p-1]\Big)$ à partir de $h\Big(t[i-1, (i-1)+p-1]\Big)$ en temps constant.

On se place sur l'alphabet ASCII et on considère $c = c_0, ..., c_{p-1}$. On récupère les codes ASCII des $c_i \in \Sigma$ on les note $x_0, ..., x_{p-1} \in \mathbb{N}$.

On calcule pour $r = 2^8$

$$P(c) := \sum_{i=0}^{p-1} x_i \cdot r^{p-i-1}$$

On voit donc c comme un nombre en base r dont les chiffres sont les x_i.

Pour le moment, P est une bijection (parce que la représentation en base r en est une) mais son ensemble d'arrivée est trop grand, on est obligés de le restreindre en prenant la réduction modulo q un grand nombre premier:

$$h(c) := P(c) \mod q$$

Cette fonction de hachage peut être incrémentée grâce à la fonction suivante:

$$\delta_{a,b}(e) := r \cdot \Big(e - ar^{p-1}\Big) + b \mod q$$

- e est l'ancien haché
- a est la lettre retirée
- b est la lettre ajoutée

Propriété

$$h\Big(t[i, i+p-1]\Big) = \delta_{t[i-1],t[i+p-1]}\left(h\Big(t[i-1,(i-1)+p-1]\Big)\right)$$

Preuve

On a:

$$P\Big(t[i-1],...,t[(i-1)+p-1]\Big) = t[i-1]\cdot r^{p-1} + \sum_{j=1}^{p-1} t[i+j-1]\cdot r^{p-j-1}$$

et

$$P\Big(t[i],...,t[(i-1)+p-1]\Big) = t[i-1]\cdot r^{0} + \sum_{j=1}^{p-1} t[i+j-1]\cdot r^{p-j-1}$$

$$\begin{aligned}
&r\cdot\left(P\Big(t[i-1],...,t[(i-1)+p-1]-ar^{p-1}\Big)+b\right)\\
&= r\cdot\sum_{j=1}^{p-1} t[i+j-1]r^{p-j-1} + t[i-1]r^{p-1} - t[i-1]r^{p-1} + b\\
&= \sum_{j=1}^{p-1} t[i+j-1]r^{p-j} + t[i+p-1]\\
&= P\Big(t[i],...,t[i+p-1]\Big)
\end{aligned}$$

et les égalités marchent également $\mod q$.
En précalculant r^{p-1} alors le calcul de δ se fera en $O(1)$. Si on a peu de collisions, on peut estimer que l'algorithme de Rebin-Karp sera en temps linéaire.

1.4.3. Bien choisir q

q doit être premier, pour bien repartir les images de h.

Alors, une chaine de caractère a de l'ordre de $\frac{256^p}{q}$ collisions. Ainsi, pour minimiser le nombre de collisions, il faut maximiser q.

Le choix $q = 2^{31} - 1$ est géneralement employé si les empreintes sont les entiers (de taille normale). C'est un nombre premier proche du plus grand entier.

1.4.4. Implémentation

```
int premier_h(char* t, int q, int p) {
    int res = (int)t[0];
    for (int i = 1; i < p; i += 1)
        res = 256 * res + (int)t[i];
    return res % q;
}

int delta(int h, int p, int q, int rpmoins1, int a, int b)
    return ((256 * (h - a*rpmoins1)) + b) % q;

bool test_egal(char* t, char* m, int i, int p) {
    for (int j = 0; j < p; j += 1)
        if (m[i] != t[i+j]) return false;
    return true;
}

int rabin_carp(char* t, char* m, int q) {
    int n = strlen(t);
    int p = strlen(m);
    int rpmoins1 = 1;
    for (int i = 1; i < p; i += 1)
        rpmoins1 = 256 * rpmoins1;
    int h = premier_h(t, q, p);
    int hm = premier_h(m, q, p);

    for (int i = 0; i < n-p+1; i += 1) {
        if (h == hm && test_egal(t, m, i p)) return i;
        h = delta(h, p, q, rpmoins1, t[i], t[i+p]);
    }

    if (h == hm && test_egal(t, m, n-p+1, p)) return n-p+1;
    return -1;
}
```

2. Compression du texte

On veut compresser des textes pour qu'ils prennent moins de place en mémoire mais sans perdre d'information.

On cherche une fonction de compression:

$$\mathcal{C} : \Sigma^p \to \mathbb{B}^q$$

et une fonction de décompression:

$$\mathcal{D} : \mathbb{B}^q \to \Sigma^p$$

qui vérifient:

(i) $\mathcal{D} \circ \mathcal{C} = \mathrm{id}$
(ii) Pour la majorité des textes t: $|\mathcal{C}(t)| < |\mathcal{D}(t)|$

Dans (ii), on ne peut pas écrire "pour tous textes". En effet, si on considére des textes de taille n écrits en ASCII, alors le cardinal de l'ensemble de départ est 2^{8n}. Or supposons qu'on ait $|\mathcal{C}(t)|_{\mathbb{B}} < |t|_{\mathbb{B}}$. Alors l'ensemble d'arrivée est de taille maximale $2^{8(n-1)}$ (au moins un octet de moins). On aurait alors $\mathcal{C}\left(\mathbb{B}^{8n}\right) \subseteq \mathbb{B}^{8(n-1)}$. Or (i) indique que c doit être injective. Or une application injective ne peut pas avoir un ensemble d'arrivée strictement plus petit que son ensemble de départ.

2.1. Un exemple simple: le codage RLE

On remplace une suite de plusieurs occurences du même caractère par un nombre puis le caractère.

2.2. Compression de Huffmann

L'idée est de coder chaque caractère sur un nombre de bits qui peut être différent en donnant des codes courts aux caractère communs et des codes longs aux caractères rares.

Définition

Un code est dit *préfixe* lorsque le code associé à un caractère n'est jamais dans le préfixe du code associé à un caractère y

L'idée est de se placer sur un alphabet Σ, de considérer un texte de Σ, et de supposer que l'on a une fonction

$$f_t : \Sigma \to \mathbb{N}$$

renvoyant le nombre d'occurences de chaque lettre.

On a: $|t| = \sum_{t\in\Sigma} f_t(x)$. On cherche un code préfixe $c : \Sigma \to \mathbb{B}^p$ qui minimise:

$$\sum_{x\in\Sigma} \left(f(x) \cdot \Big|c(x)\Big|_{\mathbb{B}} \right)$$

On suppose qu'on a accès dans la suite aux quantités $f_t(x)$ quelque soit $x \in \Sigma$.

Un tableau contenant les $f_t(x)$ peut être calculé en $O\Big(|\Sigma| + |t|\Big)$.

On peut aussi utiliser des tables de fréquences pour des domaines dédiés (par exemple pour la langue française)

Définition

Un *arbre de Huffmann* d'un code $c : \Sigma \to \mathbb{B}^p$ est une représentation arboréscente (arbre binaire) où les lettres Σ sont stockées au feuilles et le codage est stocké au noeuds avant une feuille. Aller à gauche donne un 0 et aller à droite donne un 1.

2.2.1. Encodage

Soit t un texte, et on suppose que l'on a construit un arbre de Huffmann optimal $\mathcal{A}$.

Si on doit parcourir l'arbre à chaque fois qu'on veut un code pour une lettre. On fait plein de fois (jusqu'à $|t|$ fois) un $O(|\Sigma|)$: ce n'est pas assez efficace. Pour

pouvoir trouver le code d'une lettre en $O(1)$, on aplatit l'arbre. On fait une fois le parcours et on stocke le résultat dans un tableau associatif indexé par les lettres.

2.2.2. Implémentation

```
type arbre_h =
    | Vide
    | Feuille of char
    | Noeud of arbre_h * arbre_h
```

```
let aplatir (a: arbre_h): string array =
    let rs = Array.make 256 "" in

    let rec aux (r: string): arbre_h -> unit = function
        | Vide -> ()
        | Feuille c -> rs.(int_of_char c) <- radical
        | Noeud (g, d) ->
            aux (radical ^ "0") g;
            aux (radical ^ "1") d;
    in
        aux "" a;
    rs
```

```
let encoder (t: string): string =
    let codes = calcul_arbre t |> aplatir and res = ref "" in
    String.iter
        (fun c -> res := !res ^ codes.(int_of_char c) t;
    !res
```

```
let decoder (a: arbre_h): string =
    let n = String.length code and res = ref "" in
    let rec aux (i: int): arbre_h -> string = function
        | Vide -> failwith "ce code n'existe pas"
        | Feuille chr ->
            res := !res ^ String.make 1 chr;
            if i <> n then aux i a;
        | _ when i >= n -> failwith "mauvais dernier code"
        | Noeud (g, d) when codes.[i] = '0' -> aux (i+1) g
        | Noeud (g, d) -> aux (i+1) d
    in
        aux 0 a;
    !res
```

2.2.3. Algorithme de Huffmann

On veut calculer un arbre optimal pour le texte donné en entrée. On suppose qu'on dispose des lettres et de la fréquence (ou nombre d'apparations dans le texte) de chacune. Si x est une lettre, on note $f(x)$ sa fréquence.

L'algorithme de Huffmann utilise une file de priorité min qui contient des couples (fréquence, arbre de Huffmann réduit à certains caractères).

On initialise la file en y mettant autant d'arbres qu'il y a de lettres. Si x est une lettre, on met le couple $\Big(f(x), -(x) - \Big)$ dans la file.

Tant qu'il reste strictement plus d'un arbre, prendre $\mathcal{A}_1$ et $\mathcal{A}_2$ les deux arbres de fréquences les plus faibles, on crée alors:

$$\mathcal{A}_3 := \mathcal{A}_2 - (*) - \mathcal{A}_3$$

La fréquence $f(\mathcal{A}_3)$ est définie comme la résultante de celle de $\mathcal{A}_2$ et celle de $\mathcal{A}_3$.

On ajoute alors $\Big(f(\mathcal{A}_3), \mathcal{A}_3\Big)$ à la file de priorité.

L'arbre restant à la fin est l'arbre de Huffmann.

2.2.4. Correction

Propriété

Pour tout texte t sur tout alphabet Σ, cet algorithme renvoit un arbre de Huffmann $\mathcal{A}$ qui minimise

$$S(\mathcal{A}) := \sum_{x \in \Sigma} f_t(x) \cdot p_{\mathcal{A}}(x)$$

où f_t est la fonction occurence et $p_{\mathcal{A}}$ la fonction profondeur

Preuve (Récurrence sur $|\Sigma|$)

I. Si $|\Sigma| \in \{1, 2\}$: l'arbre est optimal.

H. On suppose la propriété vraie pour un certain $n := |\Sigma| \geq 2$. On considère alors l'alphabet Σ' de taille $n + 1$. On considère un texte quelconque t et on note $\mathcal{A}$ l'arbre envoyé par l'algorithme de Huffmann. Supposons qu'il existe $\mathcal{A}'$ tel que:

$$S(\mathcal{A}') < S(\mathcal{A})$$

Soient c_0 et c_1 les caractères les moins fréquents (les plus profonds dans $\mathcal{A}$)
On peut supposer que c_0 et c_1 sont à profondeur maximale dans $\mathcal{A}$. Si ils ne le sont pas, les échanger avec des lettres c_2, c_3 qui y sont diminue $S(\mathcal{A})$. On a en effet: $p_{\mathcal{A}}(c_0) \leq p_{\mathcal{A}}(c_2)$ et $f_t(c_0) \leq f_t(c_2)$ (et de même avec c_1 et c_3)
Alors:

$$\sum_{x \in \Sigma} f_t(x) p_{\mathcal{A}}(x) \geq \sum_{x \in \Sigma \setminus \{c_0, c_2\}} \Big[f_t(x) \cdot p_{\mathcal{A}}(x) + f_t(c_2) \cdot p_{\mathcal{A}}(c_0) + f_t(c_0) \cdot p_{\mathcal{A}}(c_2) \Big]$$

On peut maintenant supposer sans perte de géneralité que c_0 et c_1 sont enfants des mêmes noeuds car s'ils ne sont pas on peut les échanger avec d'autres noeuds de profondeur maximale.
On crée nu nouvel alphabet Σ'' qui est Σ' sauf que c_0 et c_1 sont fusionnés en une lettre c dont le nombre d'occurences est $f_t(c) = f_t(c_0) + f_t(c_1)$.
On a $|\Sigma''| = n$ et $\mathcal{B}'$ l'arbre. L'algorithme de Huffmann sur le nouvel alphabet fait les mêmes étapes sauf la première. On en déduit:

$$S(\mathcal{A}) = S(\mathcal{B}) \text{ et } S(\mathcal{A}') = S(\mathcal{B}')$$

et comme on avait $S(\mathcal{A}') < S(\mathcal{A})$ on a $S(\mathcal{B}') < S(\mathcal{A})$: absurde par HR.

2.2.5. Complexité

- Taille des occurences: $O(|\Sigma| + |t|)$
- Construction de l'arbre: $O(\log(|\Sigma|)) \cdot O(|\Sigma|) = O(|\Sigma| \log(|\Sigma|))$
- Mise à plat: $O(|\Sigma|)$
- Compression: $O(|t|)$
- Décompression: $O(|code|)$

On a supposé que la concaténation de chaînes de caractères se fait en $O(1)$

2.2.6. Construction de l'arbre optimal

```
let calcule_arbre (t: string): arbre_h =
    let arbre_restants = creer_vide (0, Vide)
    and fs = nb_apparitions t
    and continuer = ref true in

    for i = 0 to 255 do
        ajoute
            arbre_restants
            (fs.(i), Feuille (char_of_int i))
    done;

    while !continuer do
        let f1, a1 = retire arbre_restants
        and f2, a2 = retire arbre_restants

        if est_vide arbre_restants then (continuer := false)

        ajoute arbre_restants (f1 + f2, Noeud (a1, a2))
    done;

    let (_, r) retire arbre_restants in r
```

2.3. Algorithme LZU (Lempel-Ziv-Welch)

Huffmann a des inconvénients:

- il faut l'arbre pour décoder
- ne marche que si on a accès à tout le texte, par exemple dans un fichier.

On peut avoir envie d'encoder sans avoir tout le texte d'un seul tenant, par exemple si on a un algorithme qui produit des données et qu'on veut les encoder à la voler.

2.3.1. Encodage

On se donne en entrée le texte (ou une fonction qui donne le morceau suivant du texte) et l'alphabet.

On crée un dictionnaire vide et on lui ajoute un code pour chaque lettre

L'algorithme utilise une variable m qui sert de mémoire pour stocker les caractères non encore exploités.

Initialement $m =$ "". Ensuite pour chaque lettre x jusqu'à la fin du texte:

Cas C-1 Si m_x est un motif de la table, alors $m := mx$

Cas C-2

Si m_x n'est pas dans la table, on a un nouveau motif. m est dans la table par hypothèse, donc on écrit le code de m. On ajoute le motif mx à la table avec le premier code disponible. On repart de $m := x$. A la fin de la boucle on écrit le code de m.

x/m/m'	ajouts table	codes
x = 'p', m = " ", m' ="p"	$\emptyset$	$\emptyset$
x = 'a', m="p", m'="a"	$+\frac{\text{pa}}{5}$	2
x = 't', m="a", m'="t"	$+\frac{\text{at}}{6}$	2, 0
x = 'a', m="t", m'="a"	$+\frac{\text{pa}}{5}$	2, 0, 4
x = 't', m="a", m'="at"	$\emptyset$	$\emptyset$
x = 'e', m="at", m'="ate"	$+\frac{\text{ate}}{8}$	2, 0, 4, 6
x = 'a', m="e", m'="a"	$+\frac{\text{ate}}{8}$	2, 0, 4, 6
x = 'e', m="at", m'="ate"	$+\frac{\text{ea}}{9}$	2, 0, 4, 6, 1
x = 't', m="a", m'="at"	$\emptyset$	$\emptyset$
x = 'e', m="at", m'="ate"	$\emptyset$	$\emptyset$
x = 'r', m="ate", m'="r"	$+\frac{\text{ater}}{10}$	2, 0, 4, 6, 1, 8
x = 'r', m="r", m'="r"	$+\frac{\text{rr}}{11}$	2, 0, 4, 6, 1, 8, 3
x = 'e', m="r", m'="e"	$+\frac{\text{re}}{12}$	2, 0, 4, 6, 1, 8, 3

Au final: $2, 0, 4, 6, 1, 8, 3, 3, 1$

2.3.2. Décodage

Si on a la table: facile. Mais on en a pas besoin, on peut la reconstruire. Il nous faut cependant un dictionnaire dont les clés sont des entiers et les valeurs sont les caractères associés.

2.3.3. Algorithme

Initialiser le tableau à l'envers

Lire le premier code, qui correspond à une lettre en sortie et initialiser m à cette lettre.

Pour chaque autre code c:

Cas D-1

Si $c < |\tau|$, alors le code est dans la table. On note $xs = \tau[c]$.

Ecrire le motif de sortie.

Ajouter mx à la table avec le premier code disponible. Enfin $m := xs$

Cas D-2

Si $c = |\tau|$, alors le code n'est pas encore dans la table. On connait la majorité du motif: c'est m. Donc on écrit $mm[0]$ en sortie, on ajoute $mm[0]$ à la table et on repart de $m := mm[0]$