Algorithmes sur les Textes

1 Données texte

1.1 Encodage des données

On distingue classiquement deux grandes catégories de données :

- données **quantitatives**:
  - numérique entier ou réel, discrètes ou continues, bornées ou non. ex: poids, taille, âge, taux d'alcoolémie,...
  - temporel : date, heure
  - numéraire
  - etc...
- données **qualitatives**:
  - de type vrai/faux (données booléennes). ex: marié/non marié, majeur/non majeur
  - de type appartenance à une classe. ex: célibataire/marié/divorcé/veuf, salarié/chômeur/retraité etc...
  - de type texte (autrement dit "chaine de caractères"). ex: nom, prénom, ville,...

D'un point de vue informatique, il n'existe pas de distinction entre le quantitatif et le qualitatif. **Tout est valeur numérique**.

Une valeur numérique (digitale) consiste en:

- un champ de bits (dont la longueur est exprimée en octets)
- un processus de décodage : le **format** (ou type)

- Les données manipulées par un programme:
  - sont codées sous un format binaire,
  - correspondant à un des types informatiques que vous connaissez déjà :
    - entier,
    - chaîne de caractères,
    - réel simple ou double précision etc...
  - ce qui implique en particulier :
    - une précision finie,
    - éventuellement des contraintes de place (le nom ou le prénom ne pouvant pas dépasser n caractères, les entiers pouvant être choisis sur un intervalle fini etc...).

1.2 Codage des caractères

```
char x;
x = 'x';
```

- x une variable de type caractère
- 'x' la valeur numérique (encodé):

- Codage ASCII :
  - codage de symboles du clavier numérique.
  - Le nombre de symboles étant inférieur à 256, on le code sur un octet : 2^8 (= 256) valeurs différentes
- Codage UTF-8 :
  - codage universel qui permet entre autre de coder les accents
- Codage latin-1 (caractères latins étendus)
- etc...

La table ASCII

\	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Il existe différents encodages binaires possibles pour les caractères :

- le code ASCII code les caractères du clavier anglais sur 7 bits, ce qui permet d'interpréter chaque caractère comme un entier entre 0 et 127
  - ainsi :

```
char c = '/';
int code = (int) c;
System.out.println(code);
```

  - affiche la valeur 47 (le code ASCII du caractère `'/'`)

voir aussi : java_type_character.wp

1.3 Codage des mots et du texte

Chaîne de caractère :

```
String s = "bonjour";
```

- s est une variable
- `"bonjour"` est une séquence de caractères
  - "chaîne" de caractères : type *string* en anglais
  - assimilable à un tableau de caractère :

```
for (char c : s.toCharArray()) {
            System.out.println(c);
        }
```

Affiche :

```
b
o
n
j
o
u
r
```

Le programme affiche les caractère 1 par 1, c'est une "chaîne" de plusieurs caractères individuels.

- La norme UTF-8 encode les caractères sur un nombre d'octets variant entre 1 et 4. Il permet ainsi de coder un nombre de caractères considérablement plus élevé.
  - Exemple : le smiley '□' appartient à la norme utf-8. Pour obtenir la valeur entière correspondante :

```
String s = "□";
int code = s.codePointAt(0);
System.out.println(code);
```

- On peut inversement afficher le caractère à partir de son code entier :

```
int code1 = 233;
int code2 = 119070;

char char1 = (char) code1;
String char2 = new String(Character.toChars(code2));

System.out.println(char1);
System.out.println(char2);
```

Donnée texte

Un texte, au même titre qu'un mot, est une chaîne de caractères (dont la longueur est définie par la longueur de la séquence de caractères qui définissent le texte, ponctuations, espaces et caractères de retour à la ligne compris).

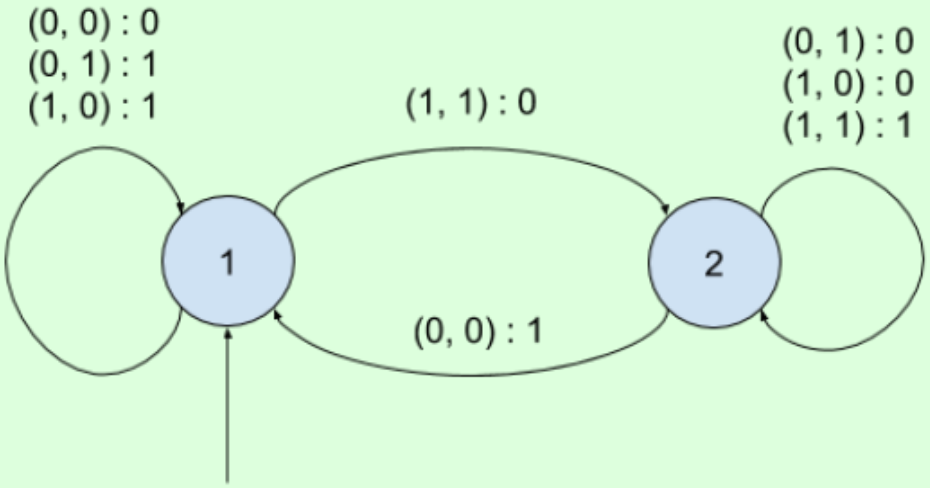
Les caractères séparateurs

Par définition les caractères séparateurs définissent la taille des espaces entre les mots, ainsi que les passages à la ligne lors de l'affichage du texte.

- : caractère nul * '': un espace simple * '\t' : tabulation * '\n' : passage à la ligne ("retour chariot") * '\b' : retour arrière ("*backspace*") * etc. </note> **Codage du texte** L'ensemble des messages possibles peut être réduit à l'ensemble des entiers naturels. En effet, chaque caractère d'un texte peut être traduit en entier en interprétant le code binaire correspondant comme un entier. Pour traduire une *chaîne de caractères* en entier, il faut "construire un nombre" à partir de chaque caractère de la chaîne en prenant en compte sa position. * ainsi, dans le système décimal, la position du chiffre dans le nombre définit à quelle puissance de 10 il appartient (unité, dizaines, centaines, etc…) Le chiffre le plus à gauche a la puissance la plus élevée et celui le plus à droite la puissance la plus faible. * Si on suppose, pour simplifier, que chaque caractère est codé par un entier entre 0 et 255 (soit le code ASCII "étendu"), alors toute séquence de caractères (de claviers européens) exprime un nombre en base 256. * Un tel nombre s'appelle un "bytestring" ("chaine d'octets"). * Il existe une fonction encode qui effectue une telle traduction * Exemple : <code java> String s = "paul.poitevin@centrale-marseille.fr"; *Encodage de la chaîne en bytes en utilisant UTF-8 byte[] b = s.getBytes();* Affichage des bytes encodés for (byte byteValue : b) { System.out.print(byteValue + " "); } </code> ce qui affiche: <code> 112 97 117 108 46 112 111 105 116 101 118 105 110 64 99 101 110 116 114 97 108 101 45 109 97 114 115 101 105 108 108 101 46 102 114 </code> Un nombre en base 256 est difficile à lire et interpréter. On le traduit en base 10 : <code java> *Conversion des bytes en BigInteger BigInteger bigInteger = new BigInteger(b);* Affichage du résultat System.out.println("i = " + bigInteger); </code> Ce qui donne : <code> i = 852806586114976881510056778471769180697471859150728801241505293627173168229624211058 </code> === 1.4 Textes et bases de textes === * Bases de textes : ensemble constitué de plusieurs textes * Bases de documents, * Dossiers contenant des documents * Collections de livres (électroniques) * → 10 - 10^4 * Contenus en ligne (descriptifs de films, articles de journaux, descriptifs de produits.) * → 10^5 - 10^6 * Ensemble du web (les pages web, en tant que telles, peuvent être considérées comme du texte mis en forme par du html). * → 10^9 * Messageries, Blogs, Forums : * 10^3 - 10^6 ==Problématiques de la recherche de texte :== * temps de réponse des algorithmes : * l'utilisateur classique veut attendre moins de 2 à 3 secondes. * Sur des bases extrêmement grandes (type recherche web), il faut donc être très performant pour atteindre ces temps de réponses. * Stockage des données : * intégralité des textes ou simple descriptif/résumé? * Les moteurs de recherche par exemple ne stockent aucune page web, ils ne stockent que des index et des références. ==Remarque :== Un document texte pourra être

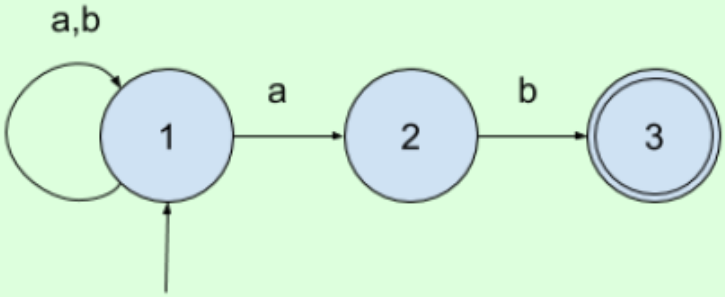
décrit soit comme : * une séquence de caractères (lettres) * une séquence de termes (mots) ==== 2 Recherche dans les textes ==== ==Exemples :== * → Recherche d'un terme : "artichaud" * → Recherche d'un motif (expression régulière) : une adresse email, une URL, un expéditeur, un numéro tel === 2.1 Rechercher un terme === **Recherche simple** d : document de taille m On cherche un algorithme qui retourne toutes les occurrences (position dans le doc) d'un certain terme t (de taille k<m) t = "ami" d : "Les amis de mes amis sont mes amis." ^ ^ ^ 4 16 30 * → La position de la première occurrence du terme t : * → Les positions de toutes les occurrences du terme t : * Complexité : $O(m \times k)$ on note d le texte et t le motif recherché dans le texte <code> Algo : recherche_simple Données : d, t : chaînes de caractères Début n = len (d) m = len (t) i ←- 0 tant que i < n - m: j ←- 0 tant que j < m et d[i+j] = t[j] : j += 1 si j == m : retourner i sinon : i ←- i + 1 Fin </code> **Remarque :** Il peut être nécessaire de vérifier que le terme est précédé et suivi par les caractères d'espacement pour éviter de détecter les mots dont le mot recherché est préfixe ou suffixe. Cette approche a un inconvénient : après une comparaison infructueuse, la comparaison suivante débutera à la position i + 1, sans tenir aucun compte de celles qui ont déjà eu lieu à l'itération précédente, à la position i. **Algorithme de Boyer-Moore** L'algorithme de Boyer-Moore examine d'abord la chaîne t et en déduit des informations permettant de ne pas comparer chaque caractère plus d'une fois. * On suppose qu'on peut tester si un caractère c appartient au motif t en temps constant * Le but est de calculer un décalage permettant de ne pas inspecter les positions où il n'y a aucune chance de trouver le motif t. * On commence par chercher la position i = m - 1 * Soit c = d[i] le dernier caractère * Si c n'est pas dans t, le décalage vaut m * Sinon on note k la position de la dernière occurrence de c dans t * si k vaut m-1 (dernier caractère), le décalage vaut m * Sinon le décalage est égal à m - 1 - k <note tip> Exemple RECHERCHE DE CHAINES CODEES EN MACHINE CHINE CHINE CHINE CHINE CHINE CHINE CHINE CHINE CHINE RECHERCHE DE CHAINES CODEES EN MACHINE </note> === 2.2 Compter les mots === Lecture séquentielle des caractères : "Les amis de mes amis sont mes amis." ^ position initiale de la tête de lecture * Il ne faut compter que les debuts de mots * Un début de mot est un caractère *alphanumérique* : <code> {a,à,ä,b,c,ç,d,e,é,è,ê,ë,…,z,A,B,C,…,Z,1,2,3,…,0} </code> * Tout ce qui n'est pas alphanumérique est un caractère *séparateur* <code> {!,#,$,%,&,",',…} </code> * Autrement dit on compte les couples (caractère séparateur, caractère alphanumérique) * remarque : le début de chaîne compte comme un caractère séparateur **remarque** : pour extraire la liste des mots présents dans le texte, on doit identifier les débuts et les fins de mots : * un début de mot est un couple

(sep., alphanum.) * une fin de mot est un couple (alphanum., sep.) <code> Algo : compte-mots Données : - d : chaîne de caractères Début: nb_mots ←- 0 sep ←- Vrai pour tout c dans d: si sep est Vrai et c est alphanumérique: sep ←- Faux nb_mots ←- nb_mots + 1 sinon si sep est Faux et c est séparateur: sep ←- Vrai retourner nb_mots Fin </code> Code équivalent : compter les mots à l'aide d'un automate fini: <code python> def compte_mots(d): state = 0 cpt = 0 for i in range(len(d)): if state == 0 and is_alpha(d[i]): state = 1 cpt += 1 if state == 1 and is_sep(d[i]): state = 0 return cpt </code> **Présentation**: un automate fini décrit les différentes étapes d'un calcul sous la forme d'un graphe orienté. * les sommets du graphe sont les états. Un état identifie une étape de calcul. L'ensemble des états représente la mémoire (finie) de l'automate. Il existe un état initial, qui est celui dans lequel l'automate démarre au début du calcul. * les arêtes représentent les transitions d'état. Un transition correspond à l'exécution d'un calcul élémentaire. Pour réaliser des calculs, on a besoin d'opérandes. Les opérandes sont lus séquentiellement en entrée de l'automate. Ils obéissent à un certain alphabet (ou ensemble de symboles, …) Σ. Enfin, des résultats de calcul sont produits en sortie de l'automate. Plus concrètement, à chaque symbole lu en entrée, l'automate consulte une table des symboles acceptés à partir de l'état courant. Si le symbole est accepté, l'automate effectue la transition d'état, et produit une sortie (par exemple incrémenter le compteur de mots). Dans le cas contraire, il s'arrête. **Définition**: Un automate fini est défini par : * un alphabet d'entrée Σ (symboles acceptés) * un alphabet de sortie Σ' * un ensemble (fini) de sommets : S * un ensemble fini d'arêtes : $A : S \times \Sigma \rightarrow S \times \Sigma'$ qui a tout couple (état, symbole d'entrée) associe un couple (état, symbole de sortie) * un état initial $s_0 \in S$ L'état initial et la séquence des symboles lus définit la séquence de symboles produits en sortie (le résultat du calcul) <note tip> Exemple : un automate qui effectue l'addition binaire: * $\Sigma = \{0, 1\}^2$ * $\Sigma' = \{0, 1\}$ * $S = \{1, 2\}$ * $s_0 = 1$

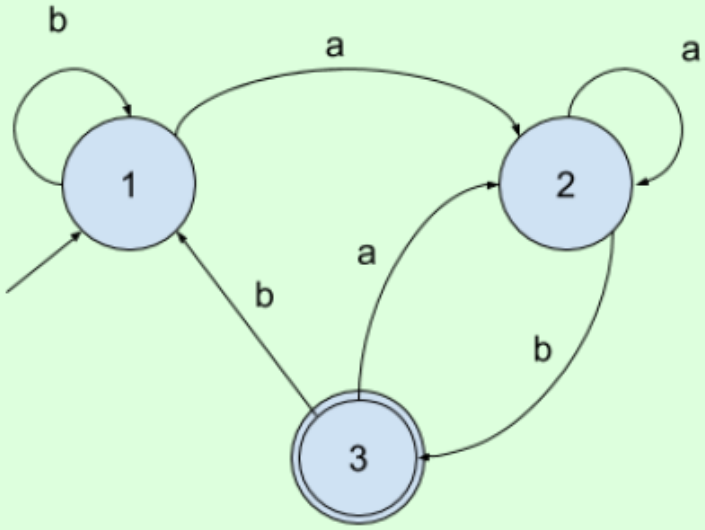


</note> La famille des automates finis définit une classe de calculs (l'ensemble des calculs réalisables par des automates finis): * séquences * boucles et branchements conditionnels Mais pas : * appels récursifs remarque: il existe des automates de calcul plus puissants : * automates à piles (calcul récursif) * machines de Turing et machines de Turing universelles (calcul sur des automates) permettant d'implémenter des calculs plus complexes === 2.3 Recherche de motifs et expressions régulières === De manière plus générale recherche d’expressions peut être effectuée à l’aide d’automates finis **non déterministes**. * Parmi les états on distingue les états initiaux et terminaux. * Il existe (parfois) plusieurs transitions possibles pour un même symbole lu * Lorsque l'automate s'arrête, on regarde si l'état est terminal. S'il est terminal, le mot est accepté (autrement dit le motif est "reconnu") Remarque : pour tout automate fini non déterministe A, il existe un automate fini déterministe A′ qui reconnait le même langage (plus compliqué à écrire). Représentation graphique : * les états dans des cercles, * l’unique état initial par une flèche pointant sur un état, * les états terminaux par un double cercle concentrique sur l’état. * Les transitions d’état quant à elles sont représentées par une flèche allant de l’état de départ jusqu’à l’état d’arrivée indexée par le caractère (ou le groupe de caractères) autorisant la transition. Lecture séquentielle des caractères : "Les amis de mes amis sont mes amis." ^ position initiale de la tête de lecture <note tip> Exemple : mots se terminant par ab * $\Sigma = \{a, b\}$ L'automate doit reconnaître les mots suivants : ab bab abab bbab aabab abbab babab bbbab aaabab etc..

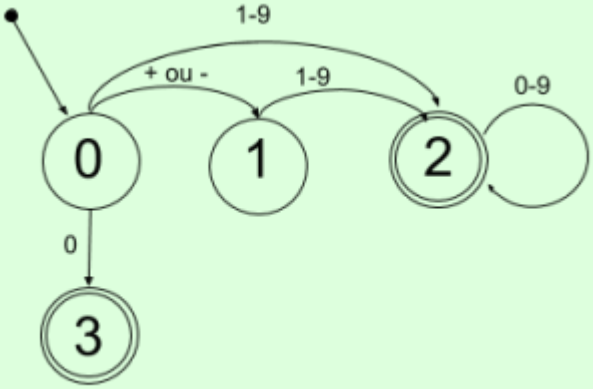
Non déterministe:



Déterministe:



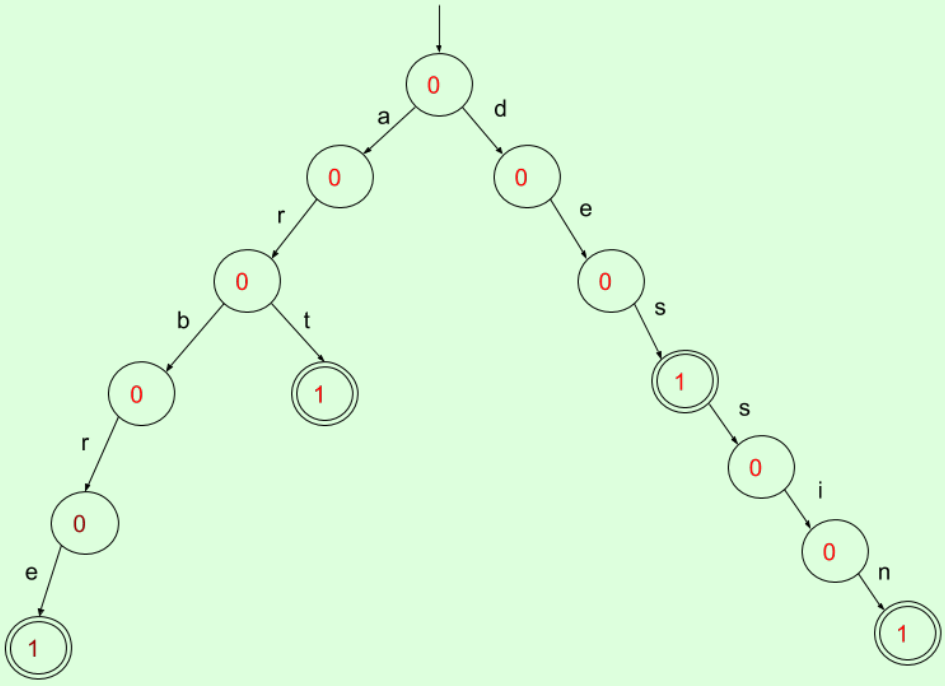
Pour aller plus loin : Cours5.html </note> **Exemples**: * Reconnaître un nombre entier : +4, -455, 1024, 0



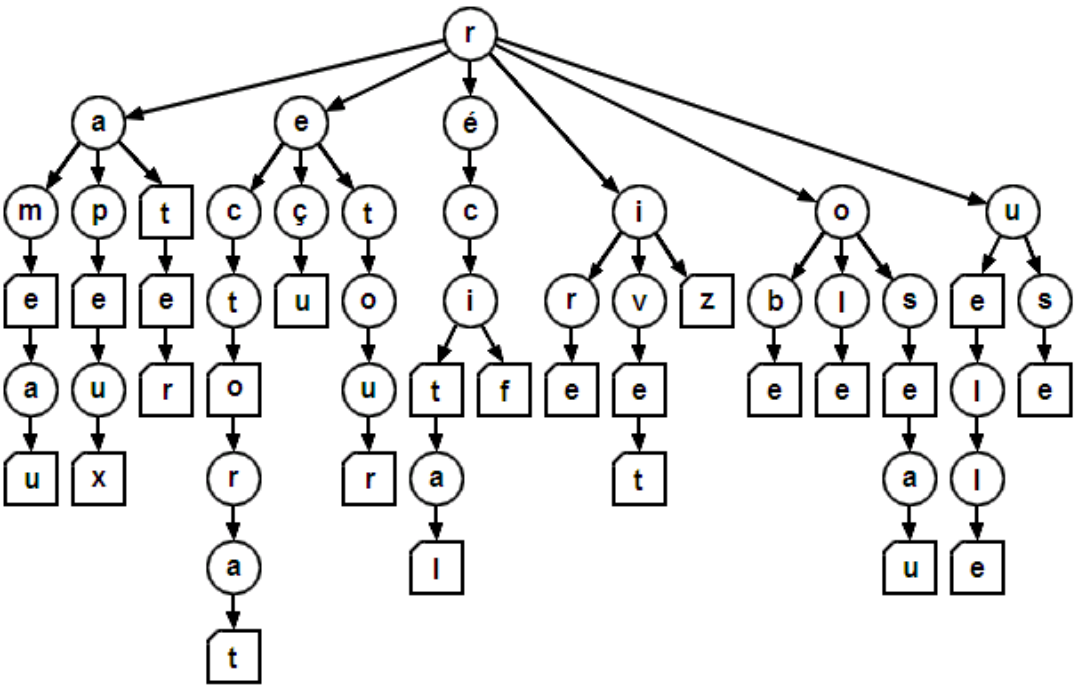
* Reconnaître un nombre réel : * TODO **Remarques** : * Les classes d'expressions qui peuvent être reconnues par un automate

fini sont appelées des *expressions régulières* * En python, les expressions régulières s'expriment à l'aide d'une syntaxe spécifique à l'aide de la librairie re * Les expressions régulières (regex) servent à décrire des motifs complexes à chercher ("marcher") dans les chaînes de caractères. Traduction de l'automate non déterministe précédent sous forme d'expression régulière : [ab]*ab Remarque : les langages qui peuvent être reconnus par des expressions régulières sont appelés les *langages réguliers*. Exemples de langages non réguliers: * reconnaître les palindromes * reconnaître une expression arithmétique * vérifier la syntaxe d'un code informatique === Syntaxe des expressions régulières Python === Définition : Il s'agit d'une syntaxe "condensée" de description d'automates finis permettant de reconnaître des motifs dans un texte. En Python, les expressions régulières sont implémentées dans la librairie re <code python> import re d = "Les astronautes de la mission Gemini 8 sont désignés le 20 septembre 1965 : Armstrong est le commandant et David Scott le pilote. Ce dernier est le premier membre du groupe d'astronautes à recevoir une place dans l'équipage titulaire d'une mission spatiale. La mission est lancée le 16 mars 1966. Celle-ci est la plus complexe réalisée jusque-là, avec un rendez-vous et un amarrage du vaisseau Gemini avec l'étage de fusée Agena et une activité extravéhiculaire (EVA) qui constitue la deuxième sortie américaine et la troisième en tout, réalisée par Scott. La mission doit durer 75 heures et le vaisseau doit effectuer 55 orbites. Après le lancement de l'étage-cible Agena à 15 h 00 UTC, la fusée Titan II GLV transportant Armstrong et Scott décolle à 16 h 41 UTC. Une fois en orbite, la poursuite de l'étage Agena par le vaisseau Gemini 8 s'engage." liste_termes = re.findall(r"([1-9]\d*|0)", d) </code> == Transitions : caractères et groupes de caractères== * a : le caractère a * [ab] : le caractère a ou le caractère b * [a-z] : n'importe quel caractère minuscule entre a et z * [^a] : n'importe quel caractère sauf le caractère a * : le caractère espace * \n : le caractère "passage à la ligne" * . : n'importe quel caractère * \. : le caractère "." uniquement * [1-9] : un chiffre entre 1 et 9 * \w : n'importe quel caractère alphanumérique * \s : n'importe quel caractère d'espacement * \d : n'importe quel chiffre Def : une expression est définie comme une suite de transition. Le mot est accepté lorsque la suite de transitions est respectée Exemple : r"chal[eu]t" accepte chalet et chalut r"\w\w\w" accepte tous les mots de 3 lettres **Branchements et itération** * Les parenthèses permettent : * de factoriser une expression (qui peut alors être traitée comme une transition) * (artichaud) * de définir des branchements (Union): * (chien|chat) * * : le caractère ou l'expression précédente répété entre 0 et n fois * + : le caractère ou l'expression précédente répété entre 1 et n

fois * ? : le caractère ou l'expression précédente répété entre 0 et 1 fois <note tip> **Récapitulatif** 1. **Caractères littéraux :** * Les caractères alphabétiques et numériques sont traités littéralement. Par exemple, le motif "abc" correspond à la chaîne "abc". 2. **Caractères spéciaux :** * Certains caractères ont une signification spéciale dans une expression régulière et doivent être échappés s'ils doivent être traités littéralement. Ces caractères spéciaux incluent . ^ $ * + ? { } [ ] \ | ( ). 3. **Classes de caractères :** * [abc] : Correspond à un caractère qui est soit a, b ou c. * [^abc] : Correspond à un caractère qui n'est pas a, b ou c. * [a-z] : Correspond à un caractère alphabétique en minuscules. * [A-Z] : Correspond à un caractère alphabétique en majuscules. * [0-9] : Correspond à un chiffre. 4. **Caractères génériques :** * . : Correspond à n'importe quel caractère sauf une nouvelle ligne. * \d : Correspond à un chiffre (équivalent à [0-9]). * \D : Correspond à un caractère qui n'est pas un chiffre. * \w : Correspond à un caractère alphanumérique (équivalent à [a-zA-Z0-9_]). * \W : Correspond à un caractère qui n'est pas alphanumérique. 5. **Quantificateurs :** * * : Correspond à zéro ou plusieurs occurrences du caractère précédent. * + : Correspond à une ou plusieurs occurrences du caractère précédent. * ? : Correspond à zéro ou une occurrence du caractère précédent. * {n} : Correspond exactement à n occurrences du caractère précédent. * {n,} : Correspond à au moins n occurrences du caractère précédent. * {n,m} : Correspond à entre n et m occurrences du caractère précédent. 6. **Ancrages :** * ^ : Correspond au début de la chaîne. * $: Correspond à la fin de la chaîne. 7. **Groupes et Alternatives :** * () : Crée un groupe. Par exemple, (abc)+ correspond à une ou plusieurs occurrences de "abc". * | : Représente une alternative (ou). Par exemple, a|b correspond à "a" ou "b". 8. **Échappement :** * \ : Permet d'échapper un caractère spécial pour le traiter littéralement. Par exemple, \\ correspond à un seul backslash. </note> ==exemples== * reconnaître une adresse mail : <code> r'\w[\.\w\-]*\w@\w[\.\w\-]*\.\w\w\w?' </code> ==== 3 Complétion / Correction ==== Un algorithme de complétion est un mécanisme logique permettant d'anticiper la saisie et de proposer des mots automatiquement pour faciliter les recherches dans un formulaire sur une page web par exemple. On utilise pour cela une structure de données arborescente, où chaque nœud de l'arbre est une étape de lecture et chaque arête correspond à la lecture d'une lettre. Les nœuds sont indexés par les lettres suivantes possibles du mot, avec un compteur par nœud pour savoir si celui-ci est final ou non (le nœud est final si le compteur est >0). On commence par construire un arbre de complétion à partir de mots de vocabulaire, <note tip> V = {art, arbre, des, dessin}



</note> L'arbre construit de cette manière est très large mais peu profond : * Pour chaque nœud, le nombre de fils est de l'ordre de la taille de l'alphabet utilisé * La hauteur maximale est celle de la taille maximale des mots de vocabulaire n_{max} * Par construction, le nombre d'arêtes est borné par $|V|\times n_{max}$ Une fois l'arbre construit, on l'utilise pour compléter un début de mot proposé par l'utilisateur (souvent plusieurs complétions possibles). Exemple : * début de mot : ar * complétions possibles : {art, arbre} **Autre exemple:**



==== 4 Comparaison/appariement de textes ==== On cherche à exprimer une distance entre deux chaînes de caractères. Une

distance entre 2 textes d1 et d2 est telle que : * dist(d1,d2) = dist(d2,d1) * dist(d1,d2) ≥ 0 * dist(d1,d2) + dist(d2,d3) ≥ dist(d1,d3) === Distance de Hamming === La distance de Hamming entre deux chaînes de même taille est définie comme le nombre de caractères non appariés. Ainsi la distance de Hamming entre "passoire" et "pastèque" est égale à 4. **Exemple** : p a s s o i r e | | | x x x x | p a s t e q u e distance = 4 Peut-on généraliser cette distance à des chaînes de taille différente? === Distance d'édition === La distance d'édition est définie, pour deux chaînes de longueur quelconque, comme le nombre minimal d'opérations permettent de transformer d1 en d2, avec les opérations suivantes : * **ins**(a) ⇢ insertion du caractère a * **perm** (a, b) ⇢ remplacement de a par b * **del** (a) ⇢ suppression du caractère a Il existe différentes manières de transformer la chaîne d1 en d2. On peut par exemple supprimer tous les caractères de d1 et insérer tous les caractères de d2, mais c'est rarement le nombre d'opérations optimal (|d1|+|d2|). - Exemples: * distance entre "robe", "arbre", et "porte". <code> - r o b - e v | ^ | v | a r - b r e </code> dist = 3 <code> r o b - e x | x v | p o r t e </code> dist = 3 <code> a - r b r e x v | x ^ | p o r t - e </code> dist = 4 ===Calcul complet cloche/hochet=== La résolution de ce problème repose sur les principes de la programmation dynamique. Un problème d'**optimisation combinatoire** se caractérise par : * un problème * un ensemble de solutions à ce problème * une fonction de coût (ou une fonction objectif) qui attribue un coût (resp un gain) à chacune des solutions possibles apportées au problème Le nombre de solutions possibles à un problème d'optimisation combinatoire croît exponentielleemnt avec la taille du problème. Trouver une solution par énumération à un tel problème devient rapidement impossible pour des problèmes de taille modérée. Certains problèmes d'OC peuvent être résolus selon le principe de la **programmation dynamique** qui consiste à décomposer le problème en sous-problèmes (et en sous-solutions): * soit un problème P présentant une solution S de coût C * Étant donné un sous problème $p \subset P$, * on liste l'ensemble des sous-solutions s, s', s applicables à p
    - et on sélectionne celle dont le coût est le plus faible
- on modifie S selon la sous-solution sélectionnée
- on met à jour le coût global
- on met à jour le sous-problème
- et on recommence jusqu'à la convergence (p = P)

Dans le cas de l'appariement de chaîne:

- une solution est un ensemble de transformations acceptables pour passer de la chaîne A à la chaîne B
- le coût est la somme des coûts des transformations appliquées
- un sous-problème consiste à apparier un morceau de A avec un morceau de B

En pratique:

- on représente l'ensemble des transformations de d1 vers d2 sous la forme d'un tableau de (m + 1) lignes et (n + 1) colonnes, avec m = |d1| et n = |d2|
- pour chaque case (i,j) du tableau,
  - le passage vers la case (i, j+1) correspond à ins(d2[j])
  - le passage vers la case (i+1, j) correspond à del(d1[i])
  - le passage vers la case (i+1, j+1) correspond à perm(d1[i],d2[j]) ou id(d1[i],d2[j]) si d1[i]=d2[j]
- la valeur de la distance au niveau de la case (i,j) est égale au minimum de :
  - 1 + dist(i, j+1)
  - 1 + dist(i+1, j)
  - dist(i+1, j+1) si d1[i]=d2[j], ou 1 + dist(i+1, j+1) sinon
- la distance au niveau de la case (m,n) vaut 0
- la distance d'édition est donnée par la valeur dans la case (0,0)

=== Algorithme ===

Préparation

```
variables globales : d1, d2 : chaînes de caractères
m = |d1|
n = |d2|
```

Récursif!!

```
algo : distance
données :
  i, j : etape de calcul
début
  si i = m et j = n :
     retourner 0
  sinon si i = m :
     retourner dist(i, j+1) + 1
  sinon si j = n :
     retourner dist(i + 1, j) + 1
  sinon si d1[i] = d2[j] :
     retourner min(dist(i, j+1) + 1, dist(i + 1, j) + 1, dist(i
+ 1, j + 1))
  sinon
     retourner min(dist(i, j+1) + 1, dist(i + 1, j) + 1, dist(i
+ 1, j + 1) + 1)
fin
```

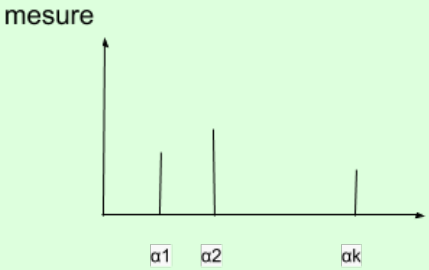
=== Alignement glouton ===

==== 5. Modèles génératifs (Hors programme) ====

Soit un document d :

- constitué de T symboles $d[1]$, ..., $d[i]$, ...
- appartenant à l'alphabet $A = \{\alpha_1,...,\alpha_K\}$ constitué de K symboles.

Une description statistique d'un texte correspond à un histogramme qui porte sur un ensemble de symboles :



Modèles probabiliste : la suite de symbole observés (le message) est générée par un processus aléatoire: $d = (d_1, d_2, ..., d_T)$

- chaque d_i est la réalisation d'un tirage aléatoire
- obéissant à une distribution de probabilité p

Les symboles sont au choix :

- des caractères appartenant à un alphabet
- des termes appartenant à un vocabulaire

=== 5.1 Modèles probabilistes ===

Les modèles probabilistes interprètent les données de type texte comme étant générées par une distribution de probabilité P inconnue.

La distribution P définit le langage utilisé dans le texte. On ne s'intéresse pas au sens du message, on regarde seulement comment les symboles se répartissent dans les documents, leurs fréquences d'apparition, les régularités, ...

=== Fréquence d'un symbole ===

Soit $\alpha \in A$ un symbole de l'alphabet. On note $P(X=\alpha)$ la fréquence d'apparition de ce symbole *dans le langage $\mathcal{L}$ considéré*.

On a par définition~: $$\sum_{\alpha \in V} P(X=\alpha) = 1$$

Exemple: $$\boldsymbol{p}_\text{Français} = (0.0942, 0.0102, 0.0264, 0.0339, 0.01587, 0.095, 0.0104, 0.0077, 0.0841, 0.0089, ...)$$ où

- $p_1 = 0.0942$ est la fréquence de la lettre 'A',
- $p_2 = 0.0102$ est la fréquence d'apparition de la lettre 'B'
- etc.

=== Probabilité jointe ===

On s'intéresse maintenant aux fréquence d'apparition de couples de lettre successives.

Soient α et β deux symboles de l'alphabet.

- Les séquences de deux caractères sont classiquement appelées des *bigrammes*.
- On définit de même les *trigrammes* comme les séquences de trois caractères
- etc.

On notera $\boldsymbol{P}_\mathcal{L}$ la matrice des fréquences des bigrammes dans un langage $\mathcal{L}$ donné, où P_{ij} donne la fréquence du bigramme (α_i,α_j).

Exemple: $$\boldsymbol{P}_\text{Français} = 10^{-5} \times \left(\begin{array}{cccc} 1.5 & 116.8 & 199.1 & ...\\ 62.8 & 1.6 & 0.14 & ...\\ 184.8 & 0 & 52.4 & ...\\ &...&&& \end{array} \right)$$

où

- $P_{11} = 1.5 \times 10^{-5}$ est la fréquence du bigramme 'AA',
- $P_{12} = 116.8 \times 10^{-5}$ est la fréquence d'apparition du bigramme 'AB'

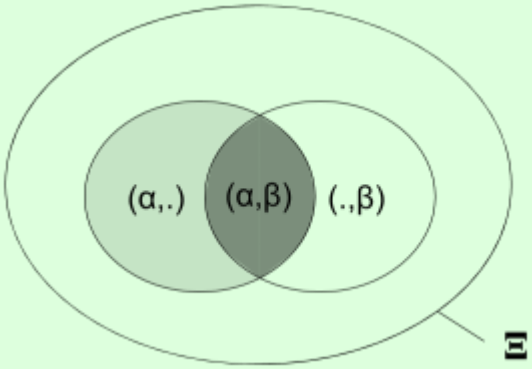
- etc.

avec bien sûr : $$\sum_{(i,j) \in \{1,...,K\}^2} P_{ij}=1$$

voir comptage des bigrammes en français

La **probabilité conditionnelle** du caractère β étant donné le caractère précédent α est définie comme :

$$P(Y = \beta | X=\alpha) = \frac{|\xi \in \Xi : (X,Y)=(\alpha,\beta)|}{|\xi \in \Xi : X = \alpha|}$$



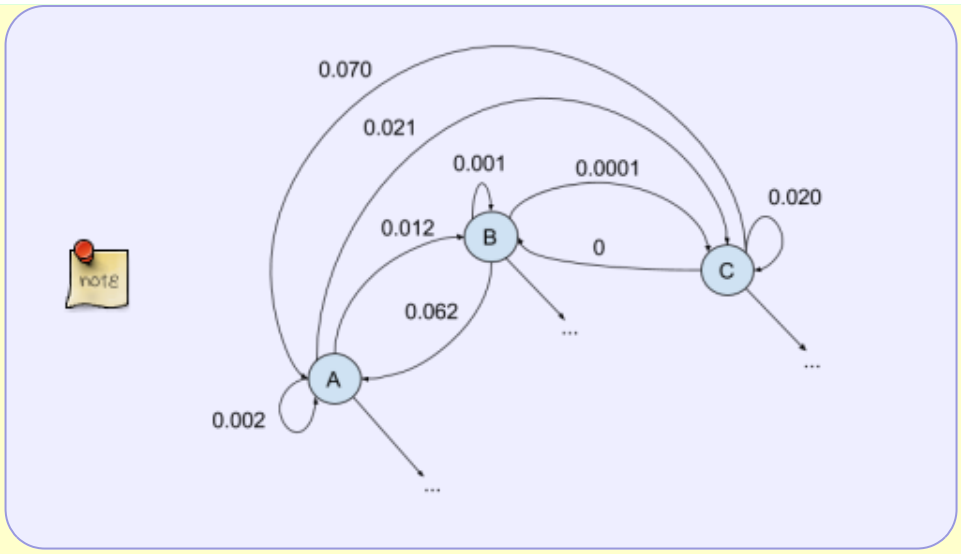
Soit en français :

$$ M_\text{Français} = \left(\begin{array}{cccc} 0.0016 & 0.0124 & 0.0211 & ...\\ 0.0615 & 0.0016 & 0.0001 & ...\\ 0.0700 & 0.0000 & 0.0198 & ...\\ & ... &&& \end{array} \right) $$ où :

- M_{11} est la probabilité de voir un 'A' suivre un 'A'
- M_{12} est la probabilité de voir un 'B' suivre un 'A'
- etc.

La matrice des probabilités conditionnelles M permet de définir un **modèle génératif** de langage inspiré des **processus aléatoires de Markov**:

- La production d'un mot ou d'un texte est modélisée comme un parcours aléatoire sur une chaîne de Markov définie par la matrice de transitions M.
- La fréquence d'apparition des lettres est modélisée comme la mesure stationnaire de la chaîne de Markov, autrement dit le vecteur de probabilité vérifiant : $$ \boldsymbol{p} = \boldsymbol{p} M$$



On peutétendre ce principe à la distribution des mots dans les textes, ce qui permet de produire des *modèles génératifs de langage*.

- Exemple : le pseudo latin ("Lorem Ipsum") : www.lipsum.com
- Exemple de pseudo-français (Utilisant une trace (mémoire) de 1 mot):

j'ai vu parfois des yeux, remonter vers toi bien fatiguée! n'est pas un appel de la terre– je hume à coups mutins les voiles la blafarde lumière puisée au delà les vieux flacon débouché rien ne puis la pourriture les forêts ou bien que vénus par des choses dans les forts des senteurs confondues de ma chère, prêtre orgueilleux danse amoureusement l'éphémère ébloui par ses couleurs du haut qu'avec effroi dans sa beauté où je puis, sans remord un fleuve invisible d'un rayon frais n'éclaira vos banquiers un parfait d'une girouette ou décor suborneur ce temps! n'est plus ma carcasse superbe pyrrhus auprès d'un ange enivré du souvenir pour moi même dans le tortu, il fée, dévotes et mange retrouve jamais enfanté au poète– cependant de minéraux charmants, horreur, plus t'enfourcher! folle, si bien loin des laves les amants nous lançant son sein palpitant les blessés ou sirène qu'importé le coin du vin des jongleurs sacrés au loin de ton bétail, embusqué, et ton juge que ce globe entier dans les temps et d'un mouvement qui m'accable sur moi hurlait longue misère toi sans pitié de pleurs aboutit dans l'or et ne vibre que le soleil d'un chemin bourbeux croyant par votre corps brûlé par mille labyrinthes c'est un etre maudit soit actif ou de l'antre taciturne je le regard m'a déjà flaire peut être n'importe où les vrais rois pour le frais n'éclaira vos riches cités dans son coeur racorni,

=== 5.2 Espaces de plongement (Word embedding) === Le plongement des mots (word embedding)

- est une technique en traitement automatique du langage naturel (TALN)

- qui consiste à représenter les mots **sous forme de vecteurs de nombres réels dans un espace vectoriel**.
- L'idée est :
  - de projeter les mots dans cet espace vectoriel
  - où la proximité spatiale entre les vecteurs reflète la sémantique des mots.

Largement utilisés dans diverses tâches de traitement du langage naturel:

1. classification de texte,
2. la traduction automatique,
3. l'analyse des sentiments,
4. la recherche d'information, etc

Word2Vec est un algorithme d'apprentissage de représentations de mots (embeddings) développé par Tomas Mikolov et son équipe chez Google en 2013.

- L'idée fondamentale est que les mots ayant des contextes similaires ont tendance à avoir des significations similaires.
- Word2Vec utilise des modèles de **prédictions** pour apprendre des représentations vectorielles en analysant les contextes d'occurrence des mots dans un corpus de texte.

Il existe deux architectures principales de Word2Vec : Skip-Gram et CBOW

=== 1. Skip-Gram === Dans l'approche Skip-Gram, le modèle tente de prédire les mots environnants (contexte) à partir d'un mot donné (mot central). Le processus d'apprentissage consiste à maximiser la probabilité d'observer les contextes donnés un mot central :

$$\max \sum_{t=1}^{T} \sum_{-c \leq j \leq c, j \neq 0} \log P(w_{t+j} \mid w_t)$$

où T est la taille du corpus, w_t est le mot central, w_{t+j} est le mot contexte, et c est la taille de la fenêtre contextuelle.

=== 2. CBOW (Continuous Bag of Words) === Dans l'approche CBOW, le modèle tente de prédire le mot central à partir des mots contextuels (contexte). Le processus d'apprentissage consiste à maximiser la probabilité d'observer le mot central étant donnés les contextes:

$$\max \sum_{t=1}^{T} \log P(w_t \mid w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c})$$

où T est la taille du corpus, w_t est le mot central, et w_{t-i} sont les mots contextuels dans la fenêtre de contexte.

=== Fonctionnement Général === Le processus d'apprentissage dans Word2Vec implique la création d'une matrice de co-occurrence, où chaque entrée représente la fréquence ou la probabilité d'occurrence conjointe de deux mots. À partir de cette matrice, le modèle ajuste les vecteurs de mots de manière itérative pour maximiser la

probabilité d'observation du contexte étant donné le mot central.

Une fois l'apprentissage terminé, les vecteurs de mots obtenus (les embeddings) capturent les relations sémantiques entre les mots dans l'espace vectoriel. Des mots similaires seront représentés par des vecteurs similaires, ce qui permet d'effectuer des opérations algébriques intéressantes telles que $\text{"roi"} - \text{"homme"} + \text{"femme"} \approx \text{"reine"}$.

Word2Vec a été révolutionnaire en raison de sa capacité à apprendre des représentations de mots utiles à partir de grands volumes de texte non annoté, et ses embeddings sont souvent utilisés comme points de départ pour de nombreuses tâches de traitement du langage naturel (NLP) et d'apprentissage automatique.