SIN5U1 : Algorithmique avancée

Cours 7 -- Table de hachage, dictionnaire et le devoir2

Michel Van Caneghem

Octobre 2012

Le problème

Une compagnie de téléphone veut fournir un service d'identifiant de l'appelant. C'est à dire à partir du numéro de téléphone retrouver le nom de l'appelant. Le nombre de numéro de téléphone est $R = 10^{10} - 1$ et il y a n abonnés. Comment faire cela de la manière la plus efficace possible.

- ✔ Un arbre balancé(AVL, red-black) avec le numéro de téléphone comme clé : $O(n)$ espace mémoire et $O(\log n)$ accès.
- ✔ Un tableau avec R entrées : temps d'accès optimal $O(1)$ par contre utilisation d'une mémoire importante $O(R)$ (de l'ordre de 40 Go).

La solution

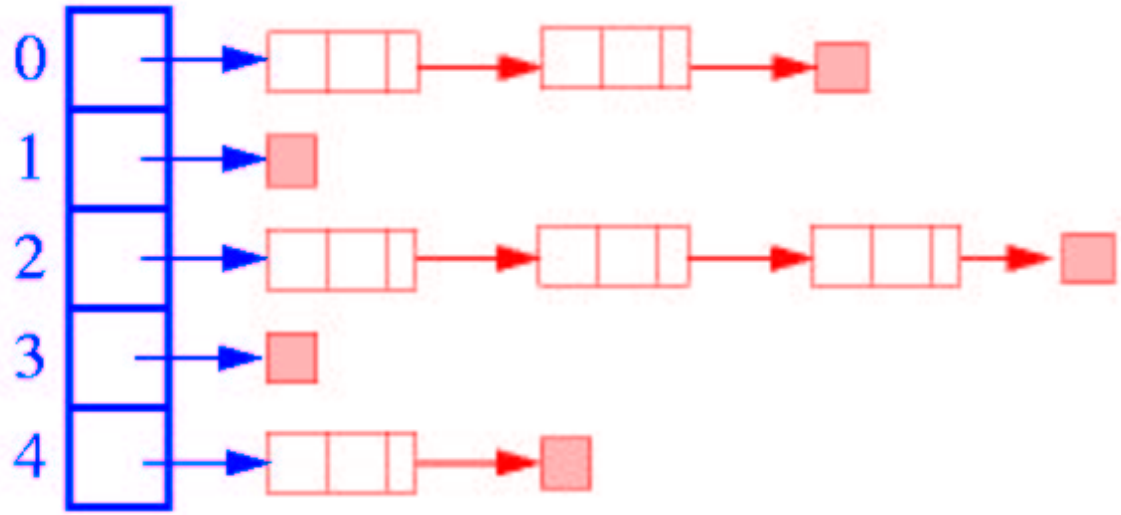
Une table de hachage ("hash-code") est la solution avec un temps d'accès de $O(1)$ et une utilisation de $O(n+N)$ en mémoire, ou N est la taille de la table.
La fonction de hachage transforme l'espace des clés en un espace plus petit. Supposons que l'on prenne le numéro de téléphone modulo 5. Si Michel a comme numéro de téléphone 04 91 26 00 03, alors il sera rangé dans la case 3.

0	1	2	3	4
		Gilles	Michel	

Gilles (04 91 26 00 02) dans la case 2, mais ou ranger Julien (04 91 26 00 63) ?

Comment résoudre les conflits ?

Il suffit de construire une liste chaînée qui contient toutes les entrées ayant la même valeur de hachage.



Le temps d'accès moyen sera $O(n/N)$ et donc réglable en fonction de N (la taille de la table).

Comment construire une bonne fonction de hachage ?

Dans ce qui suit nous allons supposer que la taille de la table est N et que le nombre d'entrées est M. Il faut alors que chaque entrée soit répartie le plus uniformément possible.
Cela dépend de la nature de la clé. Si la clé est numérique alors.

- Si c'est un entier k alors on choisit $h(k) = k \pmod N$. Pourquoi a-t-on intérêt à choisir N premier ? (Je n'en sais rien !)
- Si k est un nombre flottant $0 \leq k < 1$ alors on choisit $h(k) = \lfloor N * k \rfloor$.

Une bonne fonction de hachage (2)

Si k est une chaîne $k = c_0 c_1 \ldots c_{n-1}$ (c_i code numérique du $i$ème caractère de k). Il suffit de choisir un nombre a et de calculer avec le schéma de Horner :

$$h(k) = c_0 + a c_1 + a^2 c_2 + \cdots + a^{n-1} c_{n-1} \pmod{N}$$

```
k = 0;
for (int i = n - 1; i >= 0; i--)
    k = k*a + c[i] % N;
```

Java a choisit $a = 31$, vous pouvez prendre pour a n'importe quelle valeur, sauf une puissance de 2, si le codage d'un caractère est basé sur une puissance de 2.

Une bonne fonction de hachage (3)

Mieux ? (les devoirs des années précédentes ne l'ont pas montré !) : fonction de hachage universelle.

```
a = 31415; b = 27183;
k = 0;
for (int i = 0; i < n; i++) {
    k = k*a + c[i] % N;
    a = a*b %(N - 1);
}
```

On peut espérer que la probabilité de conflit entre deux chaînes est $1/N$, si les clés sont réparties uniformément. *Qu'est-ce que l'on peut en déduire ?*

Le problème du remplissage

En anglais : "Occupancy problem". On veut placer au hasard M balles[les mots] dans N boites[les cases]. On cherche à savoir comment les différentes boites vont être remplies. De manière plus précise : soit X_i la variable aléatoire qui compte le nombre de balles dans la boite i. On souhaite connaître :

- La moyenne et la variance de cette loi
- La probabilité qu'il y ait k balles dans la boite i :
  $Pr[X_1 = k]$ (y compris pour $k = 0$)
- La probabilité pour qu'une boite contienne plus de k balles :
  $Pr[X_1 > k]$
- Le nombre moyen de tirages pour avoir une boite avec deux balles.
- Le nombre moyen de tirages pour remplir toutes les boites "coupon collector"

Moyenne et variance

Comme toutes les boites sont équivalentes, on peut raisonner sur la boite 1. Grace à la linéarité des espérances, on a :

$$E[\sum_{i=1}^{N} X_i] = M = \sum_{i=1}^{N} E[X_i] = NE[X_1] \text{donc } E[X_1] = M/N = \alpha$$

Ceci n'est pas suffisant pour connaître la répartition, car toutes les balles pourraient être dans la même boite. Il faut calculer la variance : $\sigma^2 = E[(X_1 - E[X_1])^2 = E[X_1^2] - E[X_1]^2$.

Mais dans notre cas il est plus intéressant de calculer directement la loi de Probabilité.

La loi de probabilité

Cherchons maintenant la répartition dans chaque boite. Pour cela nous allons calculer quelle est la probabilité que la boite 1 contienne exactement k boules.
Si on considère la boite 1, pour chaque tirage d'une boule, on a une probabilité de $\frac{1}{N}$ que la boule soit dans la boite 1 et $1 - 1/N$ que la boule soit ailleurs que dans la boite 1. Il s'agit d'un schéma de Bernouilli (loi binomiale).

$$Pr[X_1 = k] = C_M^k \left(\frac{1}{N}\right)^k \left(1 - \frac{1}{N}\right)^{M-k}$$

On pose toujours $\alpha = M/N$.

La loi de probabilité(2)

Si $k = 0$ alors : $Pr[X_1 = 0] = \left(1 - \frac{1}{N}\right)^M$ si M et N sont grand alors : $\left(1 - \frac{1}{N}\right)^M = \left(\left(1 - \frac{1}{N}\right)^N\right)^{M/N} = e^{-\alpha}$. Donc premier résultat :

$$Pr[X_1 = 0] = e^{-\alpha}$$

De manière générale on trouve, si k est petit et M et N grand :

$$Pr[X_1 = k] = \frac{\alpha^k}{k!} e^{-\alpha}$$

Ce qui n'est rien d'autre qu'une loi de Poisson. (Problème des files d'attente, ...)

Vérification expérimentale

J'ai pris comme exemple un dictionnaire du français, avec la fonction de hachage de Java. Il s'agit de voir comment les clés se répartissent dans les différentes entrées de la table. On a $M = 342538$. J'ai choisit $N = 600000$. On a donc $\alpha = 0.57$. Voici les résultats expérimentaux et théoriques :

k	nb de boites mesurée	loi de poisson
0	338982	339011
1	193485	193540
2	55447	55246
3	10400	10513
4	1492	1500
5	175	171
6	17	16
7	2	1

L'occupation maximale d'une boite

On peut se demander quel est le nombre maximal de balles que l'on peut trouver dans une boite. Ce n'est pas la bonne question !! Il faut se fixer une probabilité à priori, par exemple $p_0 = \frac{1}{1000}$. Et se demander quel est le plus petit k tel que $Pr[X_1 > k] < p_0$:

$$Pr[X_1 > k] = 1 - \sum_{i=1}^{k} C_M^i \left(\frac{1}{N}\right)^i \left(1 - \frac{1}{N}\right)^{M-i}$$

On va utiliser la loi de Poisson. Remarquons tout d'abord que (formule de Taylor) :

$$e^{\alpha} \leq 1 + \frac{\alpha}{1!} + \cdots + \frac{\alpha^k}{k!} + \frac{\alpha^{k+1}}{(k+1)!} \times e^{\alpha}$$

L'occupation maximale d'une boite (2)

$$Pr[X_1 > k] = 1 - e^{-\alpha} \sum_{i=0}^{k} \frac{\alpha^k}{k!} < 1 + e^{-\alpha} \left(\frac{\alpha^{k+1}}{(k+1)!} e^{\alpha} - e^{\alpha} \right)$$

$$= \frac{\alpha^{k+1}}{(k+1)!} < \frac{\alpha^k}{k!}$$

On sait que : $n! > (\frac{n}{e})^n$.

$$Pr[X_1 > k] < \alpha^k \times (\frac{e}{k})^k = \left(\frac{e\alpha}{k} \right)^k$$

$$Pr[X_1 > t\alpha] = \left(\frac{e}{t} \right)^{t\alpha}$$

L'occupation maximale d'une boite (3)

Prenons deux exemples : $\alpha = 0.5$ et $\alpha = 2$.

k	$\alpha = 0.5$ $Pr[X > k]$	$\alpha = 2$ $Pr[X > k]$
2	0.46	--
4	0.01	--
8	$< 10^{-6}$	0.045
16	$< 10^{-17}$	$< 10^{-6}$

On en déduit que si $\alpha = 0.5$ alors on a une probabilité de $\frac{1}{1000000}$ d'avoir une liste de longueur supérieure à 8, et si $\alpha = 2$, une liste supérieure à 16.

Méthode A : Liste chaînées

Toutes les clés ayant le même hachage (conflit) sont chaînées entre-elles. Cherchons le nombre moyen d'éléments à comparer en cas de succès (le mot est dans le dictionnaire) et en cas d'échec.
En cas d'échec, c'est plus simple : il y a M mots dans le dictionnaire et N entrées. Soit le hachage n'est pas dans la table, soit il faut parcourir toute la liste chaînée jusqu'au bout. On a vu que la longueur moyenne d'une liste était de $\alpha = M/N$ donc le nombre moyen de comparaison est α
Remarque : On ne s'intéresse qu'au nombre de comparaisons, il ne faudrait cependant pas négliger le temps du au hachage.

Méthode A : Liste chaînées (2)

En cas de succès : On peut admettre que l'on parcourt en moyenne la moitié de la liste. Comme cette longueur moyenne est α et qu'il faut toujours au moins une comparaison on trouve :
Le nombre moyen de comparaisons est donc $1 + \alpha/2$

Méthode A : Liste chaînées (3)

Occupation mémoire : N mots pour la table + M mots pour les clés + M mots pour les liens. Ce qui donne en tout :

$$N + M + M = N + 2N\alpha = N(1 + 2\alpha)$$

Exemple : si $M = 300\,000, N = 100\,000, \alpha = 3$: occupation mémoire = 700 000 et nombre moyen de comparaisons par consultation : 2,5
Exemple : si $M = 300\,000, N = 600\,000, \alpha = 0.5$: occupation mémoire = 1 200 000 et nombre moyen de comparaisons par consultation : 1,25

Ceci est un exemple d'algorithme ou on échange la taille mémoire avec la performance -- Remarquez que maintenant une taille mémoire de 4Go est courante

Méthode A : Liste chainées (4)

Voici un test avec le dictionnaire du français et le texte de Proust d'un ancien devoir.

```
M = 342538, nbMots = 79019, inDico = 77784,
outDico = 1235, nbEchecs = 4756
---------------------------------------
  N         alpha   lgSu    lgEc
100000      3.425   2.671   3.797
200000      1.713   1.822   1.904
300000      1.142   1.625   1.33
400000      0.856   1.473   0.989
500000      0.685   1.241   0.667
600000      0.571   1.342   0.635
700000      0.489   1.273   0.567
800000      0.428   1.214   0.453
```

Méthode B : Linear Probing

Dans ce cas on met les entrées directement dans la table. Quand il y a un conflit, alors on parcours la table circulairement et on met la nouvelle entrée dans la première case libre. Quand on recherche un élément on fait la même chose, si on tombe sur une case vide, alors l'élément n'est pas dans la table.

```
public boolean containsKey(String s) {
    int i = hash(s);
    while (table[i] != null) {
        if (s.equals(table[i])) return true;
        i = (i + 1) % table.length;
    }
    return false;
}
```

Méthode B : Linear Probing (2)

Il se crée des "clusters" et l'analyse est beaucoup plus complexe. Il faudrait connaître la répartition statistique des longueurs des cluster.

```
alpha = 0.57, N = 600 000, M = 342538
Nb de clusters = 104311, lg moy. d'un cluster = 3.284,
carré moyen de la taille d'un cluster = 4.706
taille  Nombre   taille  Nombre
  1     43258      8     1901
  2     20741      9     1485
  3     11989     10     1041
  4      7328     ....
  5      5026     73        3
  6      3732     74        0
  7      2608     75        1
```

Méthode B : Linear Probing (3)

Je ne sais pas démontrer ces formules. Mais voici les nombres de comparaisons moyens (dans ce cas $\alpha < 1$) :

En cas de succès :

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

En cas d'échec :

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Occupation mémoire : N et c'est tout !!

Méthode B : Linear Probing (4)

Un essai :

```
M = 342538, nbMots = 79019, inDico = 77784, outDico = 1235
-----------------------------------------------------------
   N      alpha   lgSu    lgSuT   lgEc    lgEcT   nbCmpT
 400000   0.856   4.684   3.972  26.761  24.613   491618
 500000   0.685   2.35    2.087   4.188   5.539   202704
 600000   0.571   1.834   1.666   2.24    3.217   153287
 700000   0.489   1.594   1.478   1.323   2.415   130296
 800000   0.428   1.33    1.374   0.945   2.028   107937
 900000   0.381   1.326   1.308   1.109   1.805   108411
1000000   0.343   1.259   1.261   0.595   1.658   100729
1100000   0.311   1.247   1.226   0.867   1.553   101141
1200000   0.285   1.2     1.199   0.453   1.478    95482
```

Méthode C : Double Hashing

Comme la méthode précédente sauf que quand il y a un conflit, on ne cherche pas la prochaine case libre, mais on fait un second hachage et on cherche ainsi la prochaine case libre. Le but est de diminuer la taille des clusters.

```
public boolean containsKey(String s) {
    int i = hash(s);
    while (table[i] != null) {
        if (s.equals(table[i])) return true;
        i = (i + hash2(s)) % table.length;
    }
    return false;
}
```

Méthode C : Double Hashing (2)

Voici les formules :

En cas de succès :

$$\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

En cas d'échec :

$$\frac{1}{1-\alpha}$$

Occupation mémoire : N et c'est tout !!

Méthode C : Double Hashing (3)

Mes résultats

```
M = 342538, nbMots = 79019, inDico = 77784, outDico = 1235
-----------------------------------------------------------
    N     alpha  lgSu    lgSuT   lgEc    lgEcT   nbCmpT
 400000   0.856  2.329   2.264   8.238   6.944   220311
 500000   0.685  1.766   1.686   2.177   3.175   147722
 600000   0.571  1.537   1.482   1.574   2.331   127064
 700000   0.489  1.356   1.373   1.083   1.957   110607
 800000   0.428  1.244   1.305   0.816   1.748   100682
 900000   0.381  1.203   1.259   0.712   1.616    96994
1000000   0.343  1.207   1.225   0.569   1.522    96583
1100000   0.311  1.183   1.198   0.525   1.451    94527
1200000   0.285  1.144   1.177   0.397   1.399    90903
```

Qui est le meilleur ?

Si on veut faire des comparaisons équitables, il faut se mettre dans les mêmes conditions au niveau de l'occupation mémoire. Par exemple si l'on choisit dans le cas du chaînage d'avoir une mémoire totale de 900 000 mots on trouve alors avec M = 342538 : N = 214924 et $\alpha = 1.59$.

Méthode	α	lgSu	lgEch
SeparateChaining	1.59	1.708	1.592
LinearProbing	0.381	1.326	1.109
DoubleHashing	0.381	1.203	0.712

Donc double hashing est la meilleure méthode sauf si le temps du double hachage est trop long !!

Malgré tout je vous demanderait d'implanter la première méthode pour le devoir 2 -- c'est plus simple !!

La classe HashMap de Java

La taille de l'espace est une puissance de 2. Quand il n'y a plus assez de place (mémoire disponible inférieure a un certain seuil), la taille de la table est doublée et toute la table est reconstruite. Utile quand on ne connait pas à priori la taille de la table. *Il y a 14518 triplets dans le devoir2*

Il y a une double fonction de hachage : la première fonction est la méthode hash définie pour toutes les classes. La seconde fait un mélange subtil des bits et on prend les k derniers bits. Attention parfois cela marche très mal.

Les fonctions en Java

```
HashMap<String, ArrayList<Integer>> dico3 = new HashMap<String, ArrayList<Integer>>()
dico3.get(tri)
dico3.put(tri, l)
dico3.containsKey(tri)
```

Hashcode ou arbre binaire ?

Quand on teste les performances, si la fonction de hashcode est correcte alors l'utilisation des tables de hashcode est beaucoup plus performante que celles des arbres binaires (AVL) : 2 a 3 fois plus rapide. Et son implantation est bien plus simple !!!

Mais... Si on s'intéresse a un ordre des clé, par exemple toutes les clé entre 2 valeurs, alors la table hashcode ne sait pas faire. De même si l'on souhaite avoir des résultats ordonnés.

Donc tout va dépendre de l'utilisation recherchée -- Dans le cadre du correcteur orthographique, la table de hashcode est préférable.

Le Devoir 2 - Correcteur orthographique

Il ne s'agit que de corriger les fautes d'orthographes. C'est à dire que si le mot n'appartient pas au dictionnaire, il y a une faute. Le but est donc : étant donné un mot qui n'est pas dans un dictionnaire, de trouver les mots les plus *proches* dans ce dictionnaire.

Il y a beaucoup de méthodes. La méthode que je vous propose utilise les trigrammes. Il y a 2 étapes :

① la construction du dictionnaire des tri-grammes

② La correction orthographique d'un mot

Le Devoir 2 - La construction du dictionnaire

On lit une liste de mots corrects (le dictionnaire). Pour chaque mot

1. on rajoute un caractère « $ » au début et à la fin du mot. « accueil » devient « $accueil$ »
2. Puis on construit la liste des tri-grammes apparaissant dans ce mot. « $ac », « acc », « ccu », « cue », « uei », « eil », « il$ » .
3. On construit alors au fur et à mesure le dictionnaire des tri-grammes. Chaque tri-gramme pointant vers la liste des mots contenant ce tri-gramme. Par exemple « acc » pointe vers la liste « accueil », « accent », « accélération », « raccourci », ….

Le Devoir 2- La correction d'un mot

1. on rajoute un caractère « $ » au début et à la fin du mot. « acceuil » devient « $acceuil$ »
2. Puis on construit la liste des tri-grammes apparaissant dans ce mot. « $ac », « acc », « cce », « ceu », « eui », « uil », « il$ » .
3. En consultant notre liste de tri-grammes, on recherche les mots (corrects) du dictionnaire contenant le plus possible de ces tri-grammes. Par exemple « accueil » contient 3 tri-grammes en commun avec « acceuil », mais « fauteuil » contient aussi 3 tri-grammes commun avec « acceuil ».
4. On va trier cette liste de candidats à la correction en fonction du coefficient de Jaccard. On ne va garder que les candidats ayant un coefficient > 0,2.
5. Parmi ces candidats potentiels, on va les trier en utilisant la distance de Levenshtein. On va proposer comme liste de correction, les 5 premiers de cette liste par exemple.
6. On peut améliorer ce classement en utilisant la fréquence des mots du français.

Le Devoir 2 - Le coefficient de Jaccard

Quand on cherche les mots contenant le plus de triplets en commun avec un mot faux, on trouve souvent des mots beaucoup plus grands -- qui ne sont donc pas bon.

Ce coefficient est un moyen simple de mesurer la similarité de deux ensembles A et B.

$$c = \frac{|A \cap B|}{|A \cup B|}$$

ce qui se traduit dans notre cas par (si M est le mot faux et si D est un mot du dictionnaire) :

$$c = \frac{nbTrigrammesCommuns(D)}{|M| + |D| - nbTrigrammesCommuns(D)}$$

Le nombre de trigramme d'un mot est $|mot| + 2 - 2$

Le Devoir 2 - La distance de Levenshtein

C'est la distance naturelle de comparaison de 2 mots. On va considérer les opérations de remplacement d'un caractère par un autre, de suppression d'un caractère et d'insertion d'un caractère. Cette distance entre 2 mots est le nombre minimal d'opération pour passer d'un mot à un autre. Par exemple la distance entre omnibulé et obnubilé (le bon) est de 3 :

1. omnibulé
2. obnibulé - remplacement m par b
3. obnubulé - remplacement i par u
4. obnubilé - remplacement de u par i

On va voir cet algorithme qui se programme avec la programmation dynamique dans le prochain td.

Remarque : ces deux mots n'ont aucun tri-grammes en commun -- sauf le dernier « lé$ ». Donc.....

Le codage des caractères

- Le code ASCII/ISO 646
- Le code ISO 8859-1/ISO 8859-15 code actuel
- Le code Unicode
- Le code UTF-8/UTF-16 mieux ?

ASCII/ISO 646

- ASCII (American Standard Code for Information Interchange), Bob Bemer en 1961. C'est un code sur 7 bits, le premier bit étant 0 (Dans le temps on utilisait ce bit comme bit de parité).
- Les caractères de 0 à 31 ainsi que le 127 ne sont pas affichables, et correspondent à des directives de terminal. Le caractère 32 est l'espace blanc. Les autres correspondent aux chiffres, aux lettres majuscules et minuscules et à quelques symboles de ponctuation. Problème LF/CR.
- ASCII : norme américaine, la norme officielle en France ISO 646 (pas de variantes nationales).
- Problèmes : Pas d'accents et de nombreuses variantes pour utiliser le 8ème bit (Mac, EBDIC,...)

ISO 8859-1

- L'ISO 8859-1 [Ou Latin-1] (1992) recouvre les caractères utilisés par les langues européennes suivantes : albanais, allemand, anglais, basque, catalan, danois, gaélique écossais, espagnol, féringien, finnois, français. Code sur 8 bits.
- Code 191 caractères. Mais les caractères suivants ont été oubliés : œ, Œ et Ÿ. Et bien sûr il manquait le caractère € de l'euro !!. Pour corriger cela il y a maintenant le code ISO 8859-15 : *c'est celui qui est utilisé de manière courante dans la plupart des ordinateurs actuels.*
- Mais il fallait tout une collection de codes pour chaque groupe de pays, par exemple ISO 8859-2 (latin-2 ou européen central), ISO 8859-7 (grec), ... Ne parlons pas du Japon et de la Chine !!
- En 2004, le groupe de normalisation a décidé d'abandonner ce code au profit de l'Unicode et de l'UTF-8.

ISO 8859-15

ISO 8859-15																
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
Ax	NBSP	¡	¢	£	€	¥	Š	§	š	©	ª	«	¬		®	¯
Bx	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Unicode

- Unicode, dont la première publication remonte à 1991, a été développé dans le but de remplacer l'utilisation de pages de code nationales.
- Jeu de caractères abstraits : La couche la plus élevée est la définition du jeu de caractères. Par exemple, Latin-1 a un jeu de 256 caractères et Unicode normalise actuellement près de 100 000 caractères.
- Jeu de caractères codés : on ajoute à la table précédente un index numérique. Notons bien qu'il ne s'agit pas d'une représentation en mémoire, juste d'un nombre (de 4 à 6 caractères hexadécimaux).
- Exemples :
  - é : *latin small letter e with acute* (hexa = E9) &#233 ; or &eacute ; [ISO 8859-15 : E9]
  - œ : *latin small ligature oe* -- (hexa = 153) &#339 ; or &oelig ; [ISO 8859-15 : BD]

UTF-8

Codage des caractères Unicode sur plusieurs octets. (Il existe d'autres formats comme UTF-16).

0vvvvvvv	1 octet codant 1 à 7 bits
110vvvvv 10vvvvvv	2 octets codant 8 à 11 bits
1110vvvv 10vvvvvv 10vvvvvv	3 octets codant 12 à 16 bits

Quelques exemples :

é	E9	1110 1001	11000011 10101001
œ	153	1 0101 0011	11000101 10010011
€	20AC	10 0000 1010 1100	11100010 10000010 10101100

- On ne peut plus déduire la longueur d'une chaîne, en comptant le nombre d'octets.
- Il faut connaître le type de codage d'un texte.