Informatique

Vincent Pilaud

Avril 2004

1 Quelques remarques sur l'informatique

Contrairement à ce que pensent nombre de personnes, l'informatique peut être tout autre que l'organisation de réseaux et le debugage de W... Présentons tout de suite deux aspects complètement différents : l'informatique théorique et l'algorithmique.

Le but de **l'informatique théorique** est de définir proprement les outils que nous utiliserons demain (par exemple, Turing a définit autour de 1950 l'ordinateur que nous connaissons aujourd'hui, et il a fallut quelques années avant qu'il puisse être techniquement réalisable). Il est indispensable de définir les notions de :

- **Langage formel** : la machine doit pouvoir recevoir des instructions de l'exterieur (entrée/lecture), se donner ses propres instructions (calcul), et nous renvoyer son résultat (sortie/écriture),
- **Machine** : automate, machine de Turing,... il faut définir la notion de machine, qui est indépendante de la technologie,
- **Calculabilité** : les problèmes calculables sont ceux qui peuvent être résolus par notre modèle de machine,
- **Complexité** : le temps (complexité temporelle) et l'espace (complexité spatiale) qu'un problème calculable donné requiert pour sa résolution par notre modèle de machine.

Nous nous intéresseront ici à un aspect pratique de l'informatique : **l'algorithmique**, ou programmation. On trouvera un algorithme effectuant une suite de commandes autorisées permettant de résoudre un problème correctement (on doit renvoyer la bonne réponse : on parle de **correction**) et efficacement (on doit travailler le moins de temps et avec le moins d'espace possible : on parle de **complexité**).

On s'autorise ici les commandes suivantes :

- l'**assignation** : $x \leftarrow y$; (signifie mettre la valeur de y dans la case x),
- toutes les **opérations usuelles** réalisées par une calculatrice simple : $+, -, \times, \div, sqrt, mod, E[\quad], \ldots$ et les **tests** $=, <, >$ que l'on peut appliquer sur tous les **réels**,
- les **opérations logiques** : NON (unaire), ET, OU (binaires) que l'on peut appliquer sur les **booléens** : VRAI et FAUX,
- la **structure de condition** : SI (condition) ALORS (commande1) SINON (commande2) FSI ; ,
- l'**itération** : les deux structures de boucle POUR (compteur=début) JUSQU'À (fin) FAIRE (commande) FPOUR ; TANTQUE (condition) FAIRE (commande) FTANTQUE ; , et l'initialisation de tableau : $T \leftarrow$ TABLEAU (n, m) x; (signifie initialiser un tableau de taille (n, m) à la valeur x),
- la **récursivité** : REC, et l'initialisation de liste $L \leftarrow \emptyset$; ,
- la **sortie** : $x \rightarrow$; (signifie afficher x).

2 Exponentiation rapide

On se donne deux entiers x et n. On veut calculer x^n.

2.1 Algorithme naif

La première idée qui vient à l'esprit est de multiplier x par lui même n fois à la suite :

Puissance(x, n)=
  $res \leftarrow 1$;
  TANTQUE $(n > 0)$ FAIRE $(res \leftarrow res \times x; n \leftarrow n - 1;)$ FTANTQUE ;

$res \rightarrow ;;$

La complexité en temps est linéaire en n : on effectue n multiplications pour le calcul de x^n.

2.2 Algorithme d'exponentiation rapide

On peut en fait aller beaucoup plus vite pour effectuer cette opération : on considère la décomposition en base 2 de l'entier n et on va effectuer les multiplications suivant ces puissances.

La méthode repose sur le résultat simple suivant :

$$\forall p \in \mathbb{N}, x^{2p} = (x^2)^p \text{ et } x^{2p+1} = x.(x^2)^p$$

On écrit alors l'algorithme :

```
REC PuissanceRapide(x, n)=
    SI (n = 0) ALORS (1 →)
    SINON (SI (n mod 2 = 0) ALORS (PuissanceRapide(x × x, n/2) →;)
            SINON (x×PuissanceRapide(x × x, (n − 1)/2) →;)
            FSI ;)
    FSI ; ;
```

L'algorithme est beaucoup plus rapide puisqu'il est maintenant de l'ordre de $\ln n$: si $n = 2^l + 2^{l-1}n_{l-1} + \ldots + 2n_1 + n_0$ (avec $l \in \mathbb{N}$ et $n_0, \ldots, n_{l-1} \in \{0:1\}^l$), on effectue $l + n_0 + \ldots + n_{l-1}$ multiplications pour le calcul de x^n, soit dans le pire des cas $2l = 2E[\frac{\ln n}{\ln 2}]$ multiplications.

3 Problème du voyageur de commerce

3.1 Description problème

On considère le plan euclidien muni d'un repère orthonormal. Soit $N \in \mathbb{N}$ et $(a_1, b_1), (a_2, b_2), \ldots, (a_N, b_N)$ des points du plan (localisant des villes $V_1, V_2, \ldots, V_N$ sur la carte). Le but est de trouver le plus court chemin pour relier toutes les villes (le voyageur de commerce veut gagner du temps de trajet).

Exemple 1. *On se donne les points suivants :*

points	*A*	*B*	*C*	*D*	*E*	*F*
abscisses	*-2*	*0*	*1*	*0*	*2*	*0*
ordonnées	*0*	*0*	*1*	*-1*	*0*	*1*

On construit le tableau des distances entre les points (arrondies au deuxième chiffre significatif) :

	A	*B*	*C*	*D*	*E*	*F*
A	*0*	*2*	*3,16*	*2,24*	*4*	*2,24*
B	*2*	*0*	*1,41*	*1*	*2*	*1*
C	*3,16*	*1,41*	*0*	*2,24*	*1,41*	*1*
D	*2,24*	*1*	*2,24*	*0*	*2,24*	*2*
E	*4*	*2*	*1,41*	*2,24*	*0*	*2,24*
F	*2,24*	*1*	*1*	*2*	*2,24*	*0*

On a plusieurs trajets possibles : A-B-C-D-E-F de longueur $l_{ABCDEF} = 2 + 1,41 + 2,24 + 2,24 + 2,24 = 10,13$, *E-C-F-B-D-A de longueur* $l_{ECFBDA} = 1,41 + 1 + 1 + 1 + 2,24 = 6,65$.

3.2 Algorithme naif

On pourrait penser que le meilleur chemin est de partir du point le plus au nord, puis de relier à chaque fois la ville la plus proche. Ce premier algorithme est rapide mais clairement incorrect : généralement, ce n'est pas le plus court chemin pour relier toutes les villes.

Exemple 2. *Dans notre exemple, la ville la plus au nord est C. On va de proche en proche et on obtient le trajet : C-F-B-D-A-E de longueur* $l_{CFBADE} = 1 + 1 + 1 + 2,24 + 4 = 9,24$. *Ce n'est pas le meilleur chemin.*

3.3 Algorithme violent

Pour être sûr de la correction de l'algorithme, une méthode envisageable est de tester tous les chemins possibles. On sait en effet qu'il n'y a qu'un nombre fini de chemins possibles, $N! = N(N-1)(N-2)\dots 2$ exactement, donc il est possible de tous les essayer. Cet algorithme donne nécessairement le bon résultat, mais demande énormément d'opérations : en supposant donné le tableau des distances, il faut effectuer $N!(N-1)$ additions pour déterminer les longueurs de tous les chemins, puis $N!$ comparaisons pour déterminer le meilleur chemin.

3.4 Algorithme génétique

Un concept récent a été développé en informatique : l'algorithmie génétique. Elle se base sur l'idée simple d'utiliser les théories de l'évolution en biologie pour les appliquer à des problèmes algoritmiquement difficiles (ie dont les solutions actuelles donnent des complexités importantes).

Appliquons ici cette méthode au problème du voyageur de commerce. L'idée est la suivante : on part d'un chemin arbitraire, par exemple le chemin $V_1 - V_2 - \dots - V_N$ de longueur l_1, et on effectue deux calculs :

1. **Mutation** : on choisit au hasard $i < j \in \{1; \dots; N\}$ et on permute les deux villes V_i et V_j dans notre chemin. On obtient alors le chemin $V_1 - V_2 - \dots - V_{i-1} - V_j - V_{i+1} - \dots - V_{j-1} - V_i - V_{j+1} - \dots - V_N$, dont on calcule la longueur l_2.
2. **Croisement** : On choisit au hasard $i \in \{1; \dots; N\}$ et $j_1, \dots, j_i \in \{1; \dots; N\}^i$ et on met au début de notre chemin les villes $V_{j_1}, \dots, V_{j_i}$. On obtient alors le chemin $V_{j_1} - \dots - V_{j_i} - V_1 - V_2 - \dots - V_{j_1-1} - V_{j_1+1} - \dots - V_{j_2-1} - V_{j_2+1} - \dots - V_{j_i-1} - V_{j_i+1} - \dots - V_N$, dont on calcule la longueur l_3.

On garde le chemin qui a la plus petite longueur, puis on réitère l'opération un grand nombre de fois.

Rien ne nous dit que l'on obtiendra le meilleur chemin, mais les résultats obtenus sont néanmoins très convainquants : avec un nombre d'opérations restreint, on arrive à déterminer un chemin assez intéressant.

Ecrivons l'algorithme avec nos commandes autorisées : (on utilise la commande RAND qui choisit au hasard un nombre réel dans $[0; 1[$).

1. Tableau des distances à partir des coordonnées des villes :

   Dist(N, abs, ord)=
   $d \leftarrow$ TABLEAU(N, N) 0;
   POUR $(i = 1)$ JUSQU'À (N) FAIRE
   (POUR $(j = 1)$ JUSQU'À (N) FAIRE $(d[i, j] \leftarrow \sqrt{(abs[i] - abs[j])^2 + (ord[i] - ord[j])^2};)$ FPOUR ;)
   FPOUR ;
   $d \rightarrow$; ;

2. Longueur d'un chemin à partir du tableau des distances :

   Longueur(N, L, d)=
   $res \leftarrow 0$;
   POUR $(i = 1)$ JUSQU'À $(N - 1)$ FAIRE $(res \leftarrow res + d[L[i], L[i + 1]];)$ FPOUR ;
   $res \rightarrow$; ;

3. Minimum de deux et trois nombres :

   Min(a, b)=
   SI $(a < b)$ ALORS $(a \rightarrow;)$ SINON $(b \rightarrow;)$ FSI ; ;

   Minimum(a, b, c)=
   Min(Min$(a, b), c) \rightarrow$; ;

4. Mutation dans un chemin :

Mutation(N, L)=
$P \leftarrow$ TABLEAU$(1, N)$ 0;
$i \leftarrow E[N.\text{RAND}] + 1$;$j \leftarrow E[N.\text{RAND}] + 1$;
POUR $(k = 1)$ JUSQU'À (N) FAIRE $(P[k] \leftarrow L[k];)$ FPOUR ;
$P[i] \leftarrow L[j]; P[j] \leftarrow L[i];$
$P \rightarrow ; ;$

5. Croisement :

Contient(L, p, e)=
bool ←FAUX ;
POUR $(k = 1)$ JUSQU'À (p) FAIRE
(SI $(L[k] = e)$ ALORS (*bool* ←VRAI ;) SINON () FSI ;)
FPOUR ;
bool $\rightarrow ; ;$

Croisement(N, L)=
$Q \leftarrow$ TABLEAU$(1, N)$ 0;
$i \leftarrow E[N.\text{RAND}] + 1$;
POUR $(k = 1)$ JUSQU'À (i) FAIRE $(j_k \leftarrow E[N.\text{RAND}] + 1$;$Q[k] \leftarrow L[j_k];)$ FPOUR ;
$l \leftarrow 1$
POUR $(k = 1)$ JUSQU'À (N) FAIRE
(SI (Contient$(Q, i, L[k])$) ALORS () SINON $(Q[i + l] \leftarrow L[k]; l \leftarrow l + 1;)$ FSI ;)
FPOUR ;
$Q \rightarrow ; ;$

6. Programme complet :

Voyageur(N, T, abs, ord)=
$L \leftarrow$ TABLEAU$(1, N)$ 0;
$d \leftarrow$ Dist(N, abs, ord) ;
POUR $(i = 1)$ JUSQU'À (N) FAIRE $(L[i] \leftarrow i;)$ FPOUR ;
POUR $(t = 1)$ JUSQU'À (T) FAIRE
$(P \leftarrow$Mutation$(N, L); Q \leftarrow$ Croisement(N, L);
SI (Minimum(Longueur(N, L, d),Longueur(N, P, d),Longueur(N, Q, d))=Longueur(N, L, d)) ALORS ()
SINON (SI (Min(Longueur(N, P, d),Longueur(N, Q, d))=Longueur(N, P, d)) ALORS $(L \leftarrow P;)$
SINON $(L \leftarrow Q;)$
FSI ;)
FSI ;)
FPOUR ;
$L \rightarrow ; ;$

4 Tri de listes

4.1 Description du problème

Il y a une différence entre un tableau trivial (avec une seule ligne) et une liste :

1. dans un tableau T de taille (n, m), la taille est fixée, mais on peut accéder instantanément à n'importe quel élément par la commande $T[i, j]$. En particulier dans un tableau T trivial, le i-ième élément s'obtient par $T[i]$ (c'est ce qu'on a utilisé comme structure pour *abs* et *ord*). Inversement, pour construire un tableau trivial à partir d'éléments $t_1, \ldots, t_n$, il suffit de faire la boucle : POUR $(i = 1)$ JUSQU'À (n) FAIRE $(T[i] \leftarrow t_i)$ FPOUR ;
2. dans une liste $L = [l_1; l_2; \ldots; l_n]$, la taille n'est pas fixée, mais on n'a accès qu'à deux choses : le premier élément l_1, appelé tête, par la commande TÊTE(L) et le reste de la liste $[l_2; l_3; \ldots; l_n]$, appelé queue, par la commande QUEUE(L). En particulier, pour obtenir le i-ième élément, il faut appeler i fois QUEUE, puis une fois TÊTE. Inversement, pour construire une liste à partir de la tête t et de la queue Q, on écrira : $L \leftarrow t :: Q$;

On peut transformer un tableau trivial de longueur n en liste et inversement : écrivons ces algorithmes.

Tableau-liste(T, n)=
  $L \leftarrow \emptyset$;
  POUR $(i = 0)$ JUSQU'À $(n - 1)$ FAIRE $(L \leftarrow T[n - i] :: L\,;)$ FPOUR ;
  $L \rightarrow ;$;

Liste-tableau(L)=
  $T \leftarrow$ TABLEAU(1,Long(L)) 0;
  $i \leftarrow 1$;
  TANTQUE (NON$(L = \emptyset)$) FAIRE $(T[i] \leftarrow$ TÊTE$(L); i \leftarrow i + 1; L \leftarrow$ QUEUE$(L)\,;)$ FPOUR ;
  $T \rightarrow ;$;

Quelques algorithmes classiques sur les listes :

REC Longueur(L)=
  SI $(L = \emptyset)$ ALORS $(0 \rightarrow ;)$ SINON $(1+$Longueur(QUEUE$(L)) \rightarrow ;)$ FSI ; ;

REC Concatenation(L, M)=
  SI $(L = \emptyset)$ ALORS $(M \rightarrow ;)$
    SINON (TÊTE(L) :: Concatenation(QUEUE$(L), M) \rightarrow ;$)
  FSI ; ;

REC Supprime(L, e)=
  SI (TÊTE$(L) = e$) ALORS (QUEUE$(L) \rightarrow ;$) SINON (TÊTE(L) :: Supprime(QUEUE$(L), e) \rightarrow ;$) FSI ; ;

REC Minimum(L)=
  SI (Longueur$(L) = 1$) ALORS (TÊTE$(L) \rightarrow ;$) SINON (Min(TÊTE(L),Minimum(QUEUE(L))) $\rightarrow ;$) FSI ; ;

On se donne $n \in \mathbb{N}$ et on se donne une liste $L = [l_1; l_2; \ldots; l_n]$ ou un tableau $T = [T[1]; T[2]; \ldots; T[n]]$ indifféremment (puisqu'on peut changer l'un en l'autre en peu d'étapes (c'est à dire beaucoup moins d'étapes qu'il nous faut pour le trier)). Le but est de trier la liste ou le tableau par ordre croissant.

On s'intéressera à la complexité en nombre de comparaisons (les comparaisons sont la base du tri) dans le pire des cas (on notera cette complexité $CP(n)$) et en moyenne (on notera cette complexité $C(n)$).

4.2 Algorithme naif

La première idée est de trier récursivement la liste : on trouve le minimum de la liste, on le met au début, puis on trie la liste restante. Ecrivons l'algorithme :

REC TriMin(L)=
SI (Longueur(L) < 2) ALORS ($L \rightarrow$;) SINON ($x \leftarrow$ Minimum(L) ; x :: TriMin(Supprime(L, x)) $\rightarrow$;) FSI ; ;

La recherche du maximum de la liste demande de comparer tous ses termes, donc $n-1$ comparaisons. Ensuite, il faut retrouver le maximum dans la liste : dans le pire des cas, il est à la fin et on fait à nouveau $n-1$ comparaisons ; en moyenne, il est au milieu et on fait $\frac{n}{2}$ comparaisons.

Donc $CP(n) = (n-1) + (n-1) + CP(n-1) \Rightarrow CP(n) = n(n-1)$ et de même $C(n) = \frac{3n(n-1)}{4}$.

4.3 Tri bulle

L'idée du tri bulle est une idée simple : faire remonter petit à petit les plus petits éléments vers le début du tableau. Ecrivons l'algorithme :

TriBulle(T,n)=
POUR ($i = 2$) JUSQU'À (n) FAIRE
($j \leftarrow i$;
TANTQUE (($j > 1$)ET($T[j] < T[j-1]$)) FAIRE ($x \leftarrow T[j]; T[j] \leftarrow T[j-1]; T[j-1] \leftarrow x; j \leftarrow j-1$;)
FTANTQUE ;)
FPOUR ;

Cet algorithme présente un invariant de boucle : à l'étape i, les i premiers éléments du tableau sont triés. Ainsi on est sûr de la terminaison et de la correction (à la fin, tous les éléments du tableau sont triés).

Dans le pire des cas, le tableau est trié à l'envers, et il faudra donc faire $1+2+\ldots+n-1 = \frac{n(n-1)}{2}$ comparaisons. La complexité dans le pire des cas est donc comparable à celle de l'algorithme précédent. Il en est de mème de $C(n)$.

4.4 Tri fusion

L'idée du tri fusion est de "diviser pour régner". Pour trier une liste de n éléments, on la sépare en deux liste d'environ $\frac{n}{2}$ éléments que l'on trie séparément avant de les recoler ensemble. Il est évident que pour que cet méthode soit efficace, il faut que la séparation et le recolement soient efficaces. Ecrivons l'algorithme :

REC Separe(L)=
SI (Longueur(L) $= 0$) ALORS (($\emptyset, \emptyset$) $\rightarrow$;)
SINON (SI (Longueur(L) $= 1$) ALORS ($L, \emptyset$) $\rightarrow$;)
SINON ((M, N) $\leftarrow$ Separe(QUEUE(QUEUE(L))) ; (TÊTE(L) :: M,TÊTE(QUEUE(L)) :: N) $\rightarrow$;)
FSI ;)
FSI ; ;

REC Fusionne(M, N)=
SI (Longueur(N) $= 0$) ALORS ($M \rightarrow$;)
SINON SI (Longueur(M) $= 0$) ALORS ($N \rightarrow$;)
SINON (SI (TÊTE(N)<TÊTE(M)) ALORS (TÊTE(N) :: Fusionne(M,QUEUE(N)) $\rightarrow$;)
SINON (TÊTE(M) :: Fusionne(QUEUE(M), N) $\rightarrow$;)
FSI ; ;
FSI ;)
FSI ; ;

REC TriFusion(L)=
(M, N) $\leftarrow$ Separe(L) ; Fusionne(TriFusion(M),TriFusion(N)) $\rightarrow$; ;

Ici, $C(n) = C_{séparation} + 2C(\frac{n}{2}) + C_{fusion} = n + 2C(\frac{n}{2}) + n$. Posons $X(p) = \frac{C(2^p)}{2^p}$.

$$X(p) = \frac{C(2^p)}{2^p} = \frac{2C(2^{p-1}) + 2.2^p}{2^p} = X(p-1) + 2 \Rightarrow X(p) = 2p \Rightarrow C(n) = 2n.\frac{\ln n}{\ln 2}$$

5 Codes

Pour transmettre un message, on envoie une suite de 0 et de 1, appelés bits. Ils sont émis sous forme d'impulsions électriques et on reconstitue le message d'origine à la réception. On est confronté à plusieurs problèmes :

1. **Codage** du signal (c'est le rôle de la cryptographie) : il faut pouvoir cacher les informations envoyées pour qu'elles ne soient accessibles qu'au destinataire.
2. **Correction** du signal (c'est le rôle de la théorie des codes) : lors de la transmission du signal, il y a souvent un certain nombre d'erreurs, que l'on doit pouvoir corriger pour accéder au signal de départ.

5.1 Codage du signal

5.1.1 Codes mono- et poly-alphabetiques

Il existe de très nombreuses façons de coder un message. La plus simple est de faire correspondre à l'alphabet un nouvel alphabet qui servira à écrire le message. On parle de code mono-alphabetique. Cependant, il est très facile de casser ce genre de code par une simple analyse de fréquence.

Exemple 3. *Le but est de déchiffrer le message suivant :*
VZ DVQDZFK XWY MQBZRPY VZRYYWPK VWQF RXWZV WK VWQFY FWLWY DZF VZBHWZQN, KSQK ZQ VSPA XW VWQF EMWBRP. RVY PW VW LWQVWPK DZY, RVY WYYZJWPK XW VQKKWF... VW LRWRVVZFX IQR FWAZFXW XWFFRWFW VQR P'ZDDWFESRK DVQY IQ'QP AFZPX ERBWKRWFW YWBW XWY WDZLWY XW YWY FWLWY ; XW XWYWYDSRFY XW KSQK EW IQ'RV ZQFZRK DQ SQ LSQVQ WKFW, WK Z IQSR, YZPY YZLSRF DSQFIQSR, RV Z FWPSPEW. TSVVW WYK EWKKW YWPKWPEW IQR XRK IQW V'MSBBW EMSRYRK YZ FSQKW. RV WP EMSRYRK XWY EWPKZRPWY IQR KSQKWY PW YSPK IQW XWY RBDZYYWY.

On fait un tableau des fréquences (sur 384 lettres) que l'on compare au tableau du scrabble :

lettre	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*
fréquence	*3*	*8*	*0*	*12*	*10*	*24*	*0*	*1*	*9*	*1*	*28*	*7*	*5*
pourcentage	*0.78*	*2.08*	*0*	*3.13*	*2.6*	*6.25*	*0*	*0.26*	*2.34*	*0.26*	*7.29*	*1.82*	*1.30*
lettre	*N*	*O*	*P*	*Q*	*R*	*S*	*T*	*U*	*V*	*W*	*X*	*Y*	*Z*
fréquence	*1*	*0*	*20*	*31*	*31*	*19*	*1*	*0*	*26*	*69*	*16*	*37*	*25*
pourcentage	*0.26*	*0*	*5.21*	*8.07*	*8.07*	*4.95*	*0.26*	*0*	*6.77*	*17.97*	*4.17*	*9.64*	*6.51*

lettre	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*
pourcentage	*9*	*2*	*2*	*3*	*15*	*2*	*2*	*2*	*8*	*1*	*1*	*5*	*3*
lettre	*N*	*O*	*P*	*Q*	*R*	*S*	*T*	*U*	*V*	*W*	*X*	*Y*	*Z*
pourcentage	*6*	*6*	*2*	*1*	*6*	*6*	*6*	*6*	*2*	*1*	*1*	*1*	*1*

on en déduit immédiatement que W=e puisque W apparaît nettement plus souvent que toutes les autres lettres. D'autre part on sait que F, K, P, Q, R, S, V, Y et Z représentent l'une des lettres a, i, l, n, o, r, s, t, u. Une étude plus poussée sur le contexte nous permet de retrouver ces lettres et on complète ce qui nous manque. On obtient le texte de départ :
LA PLUPART DES HUMAINS LAISSENT LEUR IDEAL ET LEURS REVES PAR LAMBEAUX, TOUT AU LONG DE LEUR CHEMIN. ILS NE LE VEULENT PAS, ILS ESSAYENT DE LUTTER... LE VIEILLARD QUI REGARDE DERRIERE LUI N'APPERCOIT PLUS QU'UN GRAND CIMETIERE SEME DES EPAVES DE SES REVES ; DE DESESPOIRS DE TOUT CE QU'IL AURAIT PU OU VOULU ETRE, ET A QUOI, SANS SAVOIR POURQUOI, IL A RENONCE. FOLLE EST CETTE SENTENCE QUI DIT QUE L'HOMME CHOISIT SA ROUTE. IL EN CHOISIT DES CENTAINES QUI TOUTES NE SONT QUE DES IMPASSES.

La deuxième solution est d'utiliser plusieurs codes mono-alphabetiques en boucle. On parle de code poly-alphabetique. Mais là encore, une étude un peu poussée de la fréquence permet de retrouver la taille de la boucle, puis les codes mono-alphabetiques utilisés.

Durant la seconde guerre mondiale, l'armée allemande utilisait un code basé sur ces codes poly-alphabetiques, mais la boucle était très grande et changeait tous les jours : c'est la machine Enigma. Il était donc très difficile de déchiffrer les messages envoyés par les Allemands. Cette recherche s'est cependant révélée efficace, puisque le service

de renseignement anglais (contrairement à ce que l'on pense, composé en partie de beaucoup de mathématiciens inconnus qui ont joué un rôle crucial dans l'obtention de renseignements) parvenait parfois à trouver la boucle du jour et pouvait ainsi intercepter tous les messages de la fin de la journée. D'autre part, cette recherche a motivé l'élaboration de l'ancètre de notre ordinateur actuel, pour accélérer le temps de calcul.

5.1.2 Codage RSA

Actuellement, le codage utilisé est le codage RSA (les initiales des noms de leurs inventeurs), qui est basé sur la difficulté rencontrée pour la factorisation en nombres premiers. Présentons ce codage rapidement :

1. Le destinataire choisit :
   - p et q deux nombres premiers, et calcule $n = p.q$ et $\phi(n) = (p-1)(q-1)$,
   - c et d dans $\{1; \ldots; \phi(n)-1\}$ tels que $c.d = 1 \ (mod\ \phi(n))$,

   puis rend publique (n'importe qui a accès à l'information) la clef (n, c).
2. L'expéditeur écrit son message et le transforme en une suite d'entiers, qu'il regroupe par paquets $k_1, \ldots, k_s$ avec $\forall i \in \{1; \ldots; s\}, k_i < \min(p, q)$, et calcule $l_1 = k_1^c \ (mod\ n), \ldots, l_s = k_s^c \ (mod\ n)$ qu'il rend publique.
3. Le destinataire calcule $l_1^d \ (mod\ n), \ldots, l_s^d \ (mod\ n)$ et retombe sur le message initial.

Ce code est basé sur deux choses :

- le théorème d'Euler :
$$\forall n \in \mathbb{N}^*, \forall a \in \mathbb{Z}, pgcd(n, a) = 1 \Rightarrow a^{\phi(n)} = 1 \ (mod\ n)$$
Donc si on écrit $c.d = z.\phi(n) + 1$, on a bien : $\forall i \in \{1; \ldots; s\}$,
$l_i^d \ (mod\ n) = (k_i^c \ (mod\ n))^d \ (mod\ n) = k_i^{c.d} \ (mod\ n) = k_i^{z.\phi(n)}.k_i \ (mod\ n) = (k_i^{\phi(n)})^z.k_i \ (mod\ n) = k_i \ (mod\ n)$.
On retrouve donc bien le message de départ.

- Seules la clef (n, c) et la suite $l_1, \ldots, l_s$ sont publiques. Ce que l'on veut cacher est la suite $k_1, \ldots, k_s$, que l'on ne peut récupérer qu'en connaissant d, que l'on ne peut récupérer qu'en connaissant p et q. Il faut donc pouvoir factoriser n et retrouver p et q ce qui est impensable si p et q sont initialement choisis suffisament grands, puisqu'il est très difficile de factoriser en nombres premiers.

Exemple 4. *On utilise ici de petits entiers pour simplifier le calcul.*
Le destinataire prend : $p = 29$ *et* $q = 31$ *(*$n = 29.31 = 899$ *et* $\phi(n) = 28.30 = 840$*) et* $c = 41$ *et* $d = 41$ *(*$c.d = 41^2 = 1681 = 840.2 + 1 = 1 \ (mod\ 840)$*).*
Le public ne connaît que $n = 899$ *et* $c = 41$.
L'expéditeur veut envoyer le message "code rsa". Il a l'alphabet suivant : A=1, B=2, ..., Z=26, espace=27, .=28. Il veut donc transmettre la suite : 3, 15, 4, 5, 27, 18, 19, 1. Il calcule les puissances 41*-ièmes modulo* 899 *de cette suite : 106, 201, 283, 180, 740, 617, 351, 1 et les envoie.*
Le destinataire calcule les puissances 41*-ième modulo* 899 *de cette suite : 3, 15, 4, 5, 27, 18, 19, 1 et retrouve le message initial "code rsa".*

Algorithme de factorisation en nombres premiers :

```
Prem(n)=
    m ← E[sqrt(n)] + 1; bool ←VRAI ; i ← 2;
    TANTQUE ((bool =VRAI)ET(i < m)) FAIRE
    (SI (n mod i = 0) ALORS (bool ←FAUX ;) SINON (i ← i + 1 ;) FSI ;)
    FTANTQUE ;
    bool → ; ;

REC Factorisation(n)=
    m ← E[sqrt(n)] + 1; bool ←VRAI ; i ← 2;
    TANTQUE ((bool =VRAI)ET(i < m)) FAIRE
    (SI (n mod i = 0) ALORS (bool ←FAUX ; i :: Factorisation(n ÷ i) → ;) SINON (i ← i + 1 ;) FSI ;)
    FTANTQUE ;
    SI (bool =VRAI) ALORS ([n] →;) SINON () FSI ; ;
```

5.2 Correction du signal

Lors de la transmission d'un signal, il peut y avoir deux types erreurs :
- **inversion** : un 1 devient un 0, ou inversement. Le message paraît correct, mais il est faussé,
- **effacement** : un bit est effacé. Cette fois, on voit qu'il nous manque une donnée.

Le but d'un code correcteur est de permettre de repérer les inversions et de corriger les inversions et les effacements, en ajoutant le moins possible de bits de correction. Nous allons juste donner un exemple de code correcteur sans prétendre expliquer la théorie.

Exemple 5. *L'expéditeur veut transmettre un mot m de 4 bits, donc un nombre entre 0 et 15. Il va transmettre les 7 bits $p_1, \dots, p_7$ suivants :*

1. $p_1 = 1$ *si* $m \geq 8$,
2. $p_2 = 1$ *si* $m \in \{4, 5, 6, 7, 12, 13, 14, 15\}$,
3. $p_3 = 1$ *si* $m \in \{2, 3, 6, 7, 10, 11, 14, 15\}$,
4. $p_4 = 1$ *si* $m \in \{1, 3, 5, 7, 9, 11, 13, 15\}$,
5. $p_5 = 1$ *si* $m \in \{1, 2, 4, 7, 9, 10, 12, 15\}$,
6. $p_6 = 1$ *si* $m \in \{1, 2, 5, 6, 8, 11, 12, 15\}$,
7. $p_7 = 1$ *si* $m \in \{1, 3, 4, 6, 8, 10, 13, 15\}$.

Le destinataire reçoit un mot $q_1, \dots, q_7$ de 7 bits qui peut être différent du mot envoyé. On suppose ici qu'il y a au plus une inversion. Le destinataire calcule les trois nombres suivants :

1. $X_1 = q_4 + q_5 + q_6 + q_7 mod 2$,
2. $X_2 = q_2 + q_3 + q_6 + q_7 mod 2$,
3. $X_3 = q_1 + q_3 + q_5 + q_7 mod 2$.

Si $X_1 = X_2 = X_3 = 0$, il n'y a aucune erreur. Sinon, l'erreur se trouve au bit $\overline{X_1 X_2 X_3}^2$.

Pour prouver ce résultat, il suffit de faire deux tableaux (le premier sans erreur, le second avec une erreur) :

mot	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
p_1	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
p_2	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*1*
p_3	*0*	*0*	*1*	*1*	*0*	*0*	*1*	*1*	*0*	*0*	*1*	*1*	*0*	*0*	*1*	*1*
p_4	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*1*
p_5	*0*	*1*	*1*	*0*	*1*	*0*	*0*	*1*	*0*	*1*	*1*	*0*	*1*	*0*	*0*	*1*
p_6	*0*	*1*	*1*	*0*	*0*	*1*	*1*	*0*	*1*	*0*	*0*	*1*	*1*	*0*	*0*	*1*
p_7	*0*	*1*	*0*	*1*	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*0*	*1*	*0*	*1*
X_1	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
X_2	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
X_3	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

place de l'erreur	*1*	*2*	*3*	*4*	*5*	*6*	*7*
X_1	*0*	*0*	*0*	*1*	*1*	*1*	*1*
X_2	*0*	*1*	*1*	*0*	*0*	*1*	*1*
X_3	*1*	*0*	*1*	*0*	*1*	*0*	*1*