

Cours 3

Structures de données dynamiques

Jean-Jacques.Levy@inria.fr

<http://jeanjacqueslevy.net>

tel: 01 39 63 56 89

secrétariat de l'enseignement:

Catherine Bensoussan

cb@lix.polytechnique.fr

Aile 00, LIX

tel: 01 69 33 34 67

<http://www.enseignement.polytechnique.fr/informatique/>

Plan

1. Listes – Arbres
2. Opérations de base
3. Polymorphisme?
4. Abstraction1: représentation des ensembles
5. Abstraction2: paquetage de grands nombres

Répertoire pour stocker ses TD

`/users/profs/info/TD/tc/g/n` sur `poly.polytechnique.fr` où $g \in \{1, 12\}$ est le numéro de groupe et $n \in \{1, 2, 3, \dots, 10\}$ le numéro du cours. Ne pas oublier de mettre son nom dans le nom du fichier.

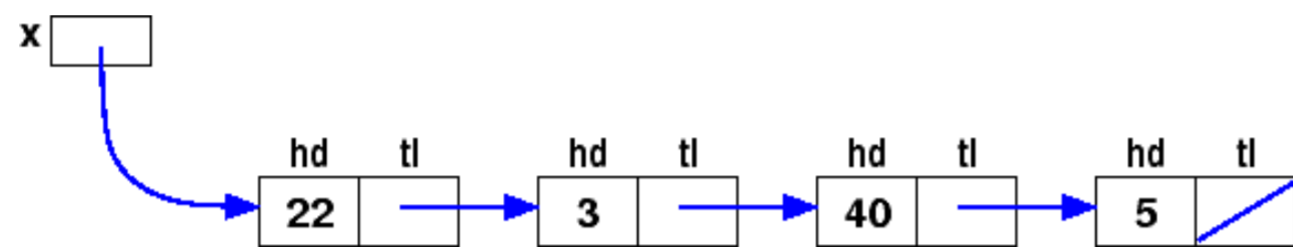
(Répertoire en accès en écriture pour tout le monde)

Listes

Comment représenter des **ensembles dynamiques**? Dont le nombre d'éléments peut varier dans le temps? Les tableaux ont leur dimension fixée à leur création. Il faut considérer le nombre maximal de leurs éléments \Rightarrow gachis (en place mémoire).

Symétriquement, comment représenter des **vecteurs** (ou matrices) **creux**? Un tableau prend l'espace mémoire nécessaire pour toutes les valeurs de l'indice, et donc pour les valeurs inutiles.

D'où la structure de données des listes dont les éléments sont les éléments non consécutifs de l'ensemble (ou des paires [indice, valeur] dans le cas des vecteurs ou matrices creux). On ne peut accéder que **séquentiellement** aux éléments d'une liste.



Représentation des listes

```
class Liste {
    int hd; Liste tl;

    Liste (int v, Liste a) {
        hd = v; tl = a;
    }
}
```

ou

```
class Liste {
    int indice; int valeur; Liste suivant;

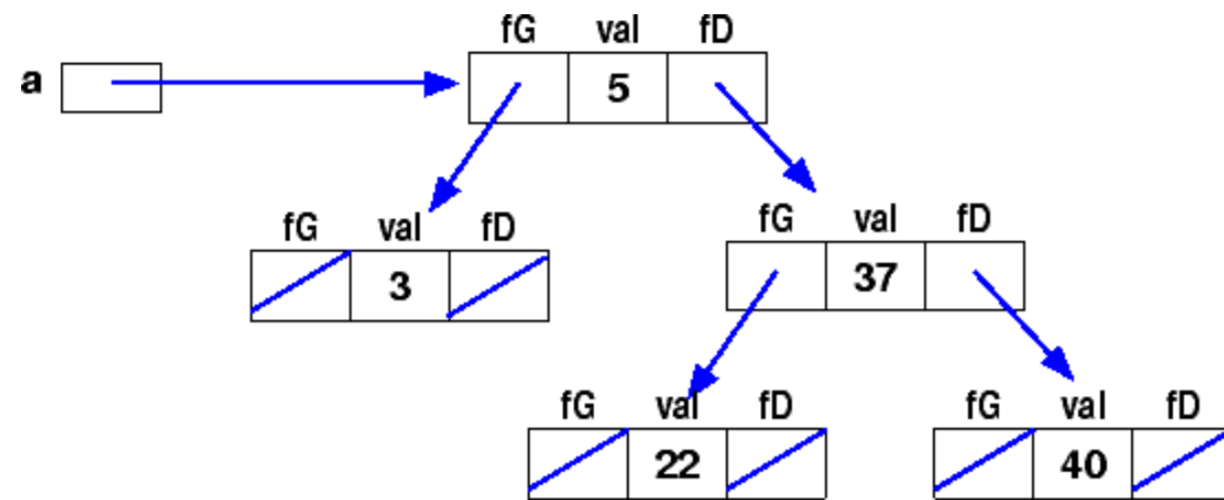
    Liste (int i, int v, Liste a) {
        indice = i; valeur = v; suivant = a;
    }
}
```

Exercice 1 Pour les fous de l'optimisation de l'espace-mémoire, il existe des représentations encore plus compactes des listes, e.g. *hash-consing*. Savez-vous en trouver une?

Arbres

Les arbres représentent des ensembles dynamiques (parfois ordonnés) avec **accès rapide** aux éléments (en $O(\log n)$ pour n éléments).

Les arbres représentent aussi les **termes** des structures algébriques (cf. plus tard), et sont donc la structure de base de tous les systèmes de calcul formel (**compilation**, **démonstration automatique**, **Maple**).



Opérations courantes sur les listes

- concaténation
- image miroir
- appartenance

Deux styles de programmation

- listes non modifiables \Rightarrow clarté
- listes modifiables \Rightarrow efficacité

En Java, les types des données ne permettent pas de distinguer entre ces deux styles de programmation (toute variable étant a priori modifiable).

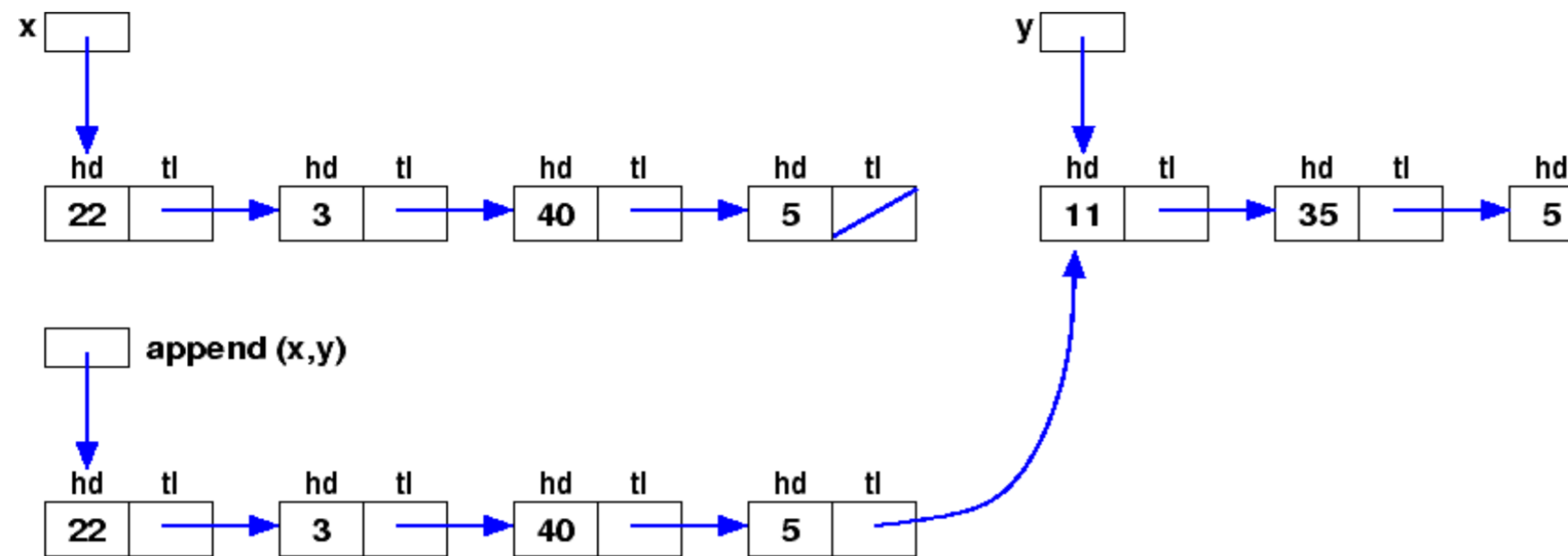
L'inefficacité des listes non modifiables est contre-balancée par l'existence d'un *garbage collector*, ie un *glâneur de cellules* qui récupère régulièrement les objets **non utilisés**. Cet effet est d'autant plus fort que le GC est efficace (en temps).

En majeure 1, les techniques de GC sont étudiées dans le cours Langages de programmation

Concaténation non destructive – Append

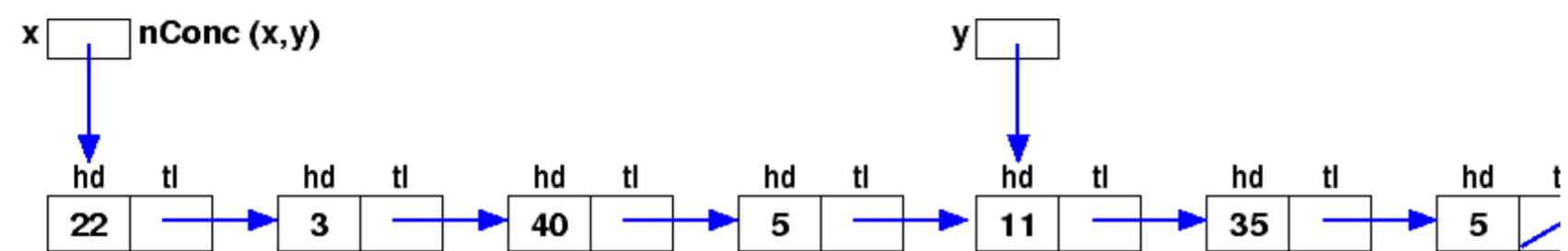
```
static Liste append (Liste a, Liste b) {  
    if (a == null) return b;  
    else return new Liste (a.hd, append (a.tl, b)) ;  
}
```

Récurrance structurelle sur le premier argument. En $O(n)$.



Concaténation destructive – *nConc*

```
static Liste nConc (Liste a, Liste b) {  
  if (a == null) return b;  
  else {  
    a.tl = nConc (a.tl, b);  
    return a;  
  }  
}
```



Récurrance structurelle sur le premier argument. En $O(n)$.

Modification de la valeur du premier argument de **nConc**.

Programmation dangereuse, car création implicite d'**alias**.

Exemples sur listes - suite

```
static Liste nConcI (Liste a, Liste b) {
    if (a == null) return b;
    else {
        Liste c = a;
        while (c.tl != null)
            c = c.tl;
        c.tl = b;
        return a;
    }
}
```

On combine les inconvénients: itératif et destructif! En $O(n)$.

Image miroir – version non destructive

```
static Liste reverse (Liste x) {
    if (x == null) return null;
    else return append (reverse (x.tl), new Liste (x.hd, null));
}

static Liste miroir (Liste x) { return miroir1 (x, null); }

static Liste miroir1 (Liste x, Liste r) { // r accumulateur
    if (x == null) return r;
    else return miroir1 (x.tl, new Liste(x.hd, r));
}
```

Récurivité terminale dans miroir1 \Rightarrow version itérative

```
static Liste miroirI (Liste x) {
    Liste r = null;
    for (; x != null; x = x.tl)
        r = new Liste (x.hd, r);
    return r;
}
```

Image miroir – version destructive

```
static Liste nReverse (Liste x) {
    if (x == null) return null;
    else {
        Liste y = nReverse (x.tl);
        x.tl = null;
        return nConc (y, x);
    }
}

static Liste nReverseI (Liste x) {
    Liste r = null;
    while (x != null) {
        Liste y = x.tl; x.tl = r; r = x; x = y;
    }
    return r;
}
```

Exercice 2 Calculer la complexité de ces différentes fonctions.

Exercice 3 Passer du récursif à l'itératif avec un accumulateur comme pour *miroir*.

Autres fonctions

```
static Liste listeAleatoire (int n) {
    Liste r = null;
    for (int i = 0; i < n; ++i)
        r = new Liste ((int) (Math.random() * 100), r);
    return r;
}

static void imprimer (Liste x) {
    for (Liste y = x; y != null; y = y.tl)
        System.out.print (y.hd + " ");
    System.out.println ();
}

static Liste lireDansTableau (String[] a) {
    Liste r = null;
    for (int i = a.length - 1; i >= 0; --i)
        r = new Liste (Integer.parseInt (a[i]), r);
    return r;
}
```

Autres fonctions – bis

```
static boolean member (int n, Liste x) {
    if (x == null) return false;
    else return n == x.hd || member (n, x.tl);
}

static boolean equal (Liste x, Liste y) {
    if (x == null) return y == null;
    else if (y == null) return x == null;
    else return x.hd == y.hd && equal (x.tl, y.tl);
}

static boolean eq (Liste x, Liste y) {
    return x == y;
}
```

Exercice 4 Décrire la différence entre `equal` et `eq`.

Arbres – représentation des arbres

```
class Arbre {
    int val; Arbre fG, fD;

    Arbre (int v, Arbre a, Arbre b) {
        val = v; fG = a; fD = b;
    }
}
```

Si l'arbre n'est pas binaire, plusieurs solutions avec des tableaux

```
class Arbre {
    int val; Arbre[] fils;
    Arbre (int v, Arbre[] a) { val = v; fils = a; }
}
```

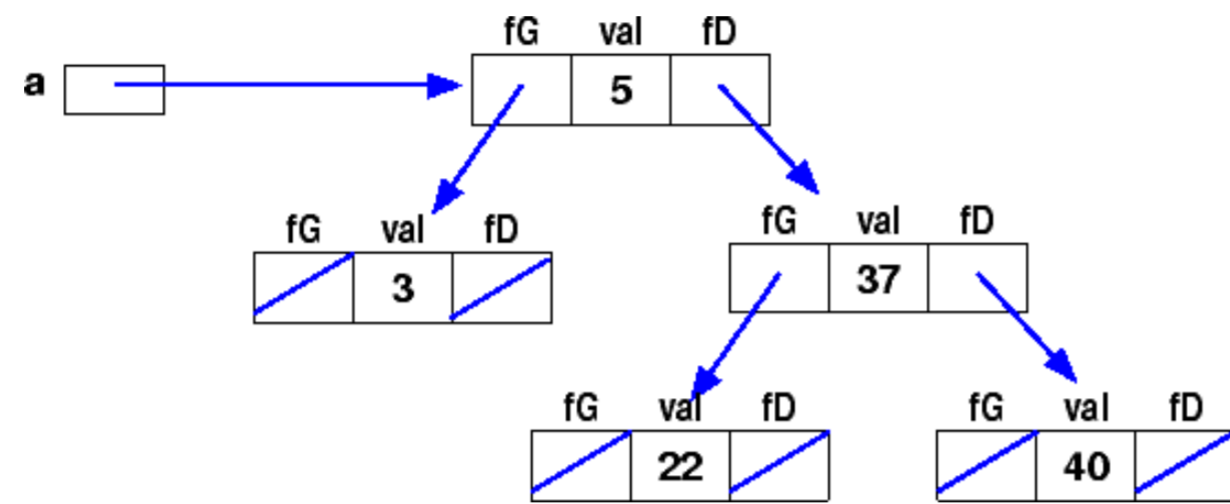
ou avec des listes

```
class Arbre {
    int val; ListeArbres fils;
    Arbre (int v, ListeArbres x) { val = v; fils = x; }
}

class ListeArbres {
    Arbre hd; ListeArbres tl;
    ListeArbres (Arbre a, ListeArbres x) { val = v; fils = x; }
}
```

Fonctions sur les arbres

```
static void imprimer (Arbre a) {  
    if (a != null) {  
        System.out.print ("["); imprimer (a.fG);  
        System.out.print (" " + a.val + " ");  
        imprimer (a.fD); System.out.print ("]");  
    }  
}
```



donne [[3] 5 [[22] 37 [40]]]

Fonctions sur les arbres

```
static int next;
static Arbre lireDansTableau (String[] s) {
    if (!s[next].equals ("["))
        return null;
    else {
        ++next; Arbre a = lireDansTableau (s);
        int v = Integer.parseInt (s[next]);
        ++next; Arbre b = lireDansTableau (s);
        if (!s[next++].equals ("]")) {
            System.err.println ("Mauvais arguments.");
            System.exit (1);
        }
        return new Arbre (v, a, b);
    }
}
```

Exercice 5 Ecrire un programme qui compte le nombre de feuilles n_f et de noeuds internes n_i .

Exercice 6 Vérifier que $n_f = n_i + 1$ si l'arbre est binaire (ie. tout noeud interne a deux fils).

Fonctions sur les arbres

```
static Liste listeDesNoeuds (Arbre a) {
    if (a == null) return null;
    else return append (listeDesNoeuds (a.fG),
                       append (new Liste (a.val, listeDesNoeuds (a.fD))));
}
```

Exercice 7 Faire une version plus efficace et esthétique.

Exercice 8 Ecrire une fonction `arbreDeLaListe` transformant une liste en un arbre (aussi équilibré que possible).

```
static boolean member (int n, Arbre x) {
    if (x == null) return false;
    else return n == x.val || member (n, x.fG) || member (n, x.fD);
}

static boolean equal (Arbre x, Arbre y) {
    if (x == null) return y == null;
    else if (y == null) return x == null;
    else return x.val == y.val && equal (x.fG, y.fG) && equal (x.fD, y.fD);
}
```

Des listes baroques

Autrefois, l'espace mémoire et la vitesse étaient très critiques. Toutes sortes d'optimisations avaient lieu.

- Listes **doublement chaînées**: pour aller rapidement au(x) prédécesseur(s) dans la liste.
- Listes **gardées**: *hack* pour ne pas avoir à singulariser le cas vide. Analogue des sentinelles pour éviter les débordements dans les tableaux. (Si utilisé pour faire des méthodes non statiques, c'est horrible.)
- Listes **circulaires**: autre technique pour accéder au(x) prédécesseur(s).

On peut combiner ces 3 optimisations souvent bien inutiles et aussi très laides.

Listes / Arbres polymorphes?

- Il y a une classe `Object` de tous les objets.
- Conversions implicites:
 - `C` → `Object`
 - `C[]` → `Object[]`
 - `t[]` → `Object`
- Conversions explicites: `(t) e` où `t` type quelconque
- Les variables de types **primitifs** (`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`) ne sont pas convertibles en `Object`.
- `Integer(n)`, `Float(x)`, etc construisent des objets contenant les valeurs `n`, `x`, etc, et dont les méthodes `intValue()`, `floatValue()`, etc donnent les valeurs `n`, `x`, etc.

Bref, **Java n'a pas de types de données polymorphes**. (cf. *Generic Java*, <http://www.cs.bell-labs.com/who/wadler/pizza/gj/>, par Odersky et Wadler)

Listes / Arbres polymorphes

```
class Liste {
    Object hd; Liste tl;
    Liste (Object v, Liste a) {
        hd = v; tl = a;
    }

    static Liste append (Liste a, Liste b) {
        if (a == null) return b;
        else return new Liste (a.hd, append (a.tl, b)) ;
    }

    public static void main (String[] args) {
        Liste x = new Liste(new Integer(2), new Liste( new Integer (3), null));
        Liste y = new Liste(new Integer(4), new Liste( new Integer (5), null));
        Liste z = append (x, y);
        int u = ((Integer) z.hd).intValue();
        System.out.println (u);
    }
}
```

Exercice 9 Faire mieux?

Abstraction – Encapsulation

Par exemple, comment représenter un ensemble dont le nombre d'éléments peut évoluer dynamiquement?

Par une liste, par un arbre?

Quelles sont les opérations à réaliser sur les ensembles? A fournir dans l'**interface** ou encore *API* (*application program interface*)

On veut donc construire un module avec certaines fonctions à exporter. Tout le reste est à cacher, car il s'agit de la représentation interne de notre abstraction.

Représentations d'ensembles

- **listes** $\langle 2, 10, 4, 2, 1, 6 \rangle$
- **listes sans répétitions** $\langle 2, 10, 4, 1, 6 \rangle$
- **listes ordonnées** $\langle 1, 2, 4, 6, 10 \rangle$
- **arbres** $\langle [[2] 10 [4]] 2 [1 [6]] \rangle$
- **arbres de recherche** $\langle [[[1] 2 [4]] 6 [10]] \rangle$

Listes ordonnées

```
class Ensemble {
    hd: int; tl: Ensemble;
    Ensemble (int x, Ensemble e) { hd = x; tl = e; }

    static boolean vide (Ensemble e) { return e == null; }

    static Ensemble ajouter (int x, Ensemble e) {
        if ( e == null || x < e.hd ) return new Ensemble (x, e);
        else if (x == e.hd) return e;
        else if (x > e.hd) return new Ensemble (e.hd, ajouter (x, e.tl));
    }

    static boolean appartient (int x, Ensemble e) {
        if ( e == null || x < e.hd ) return false;
        else return x == e.hd || appartient (x, e.tl);
    }

    static Ensemble enlever (int x, Ensemble e) {
        if ( e == null || x < e.hd ) return e;
        else if (x == e.hd) return e.tl;
        else if (x > e.hd) return new Ensemble (e.hd, enlever (x, e.tl));
    }
}
```

Listes et ensembles

Exercice 10 Calculer l'union, l'intersection, la différence symétrique.

Exercice 11 Donner des versions destructives de ces fonctions.

Exercice 12 Donner des versions itératives et comparer?

Exercice 13 Refaire ces programmes avec la représentation par des listes non ordonnées. Quels en sont les coûts?

Exercice 14 Comment passer d'une représentation à l'autre?

Exercice 15 Faire le tri par fusion sur les listes non ordonnées.

Entiers en précision arbitraire

Un cas classique d'utilisation des listes est de faire un paquetage d'opérations arithmétiques en précision arbitraire, ie au delà de $2^{31} - 1$.

Représentation d'un grand nombre. On suppose une base B donnée. Par exemple $B = 10$, mais plus vraisemblablement $B = 2^{30}$.

- *Little endian*: 1794 représenté par la liste $\langle 4, 9, 7, 1 \rangle$
Pentium
- *Big endian*: 1794 représenté par la liste $\langle 1, 7, 9, 4 \rangle$
IBM 370, PDP-10, Vax, Motorola, PPC.
- Petit et Grand: MIPS, Alpha

Nous prenons petit endien.

On fait facilement l'addition et la multiplication, la lecture et l'impression. Comment faire la division et le test de primalité, utiles pour faire de la cryptographie?

Grands nombres et division

On veut diviser u par v , ie $u = qv + r$ où $0 \leq r < v$. D'abord, on voit que toute la difficulté est de traiter le cas où $u/v < B$. On fera ensuite l'opération chiffre par chiffre du quotient.

Quand $u/v < B$, une bonne approximation du quotient s'obtient en regardant les 2 premiers chiffres de u et le premier chiffre de v , notamment quand le premier chiffre v_1 de v vérifie $v_1 \geq \lfloor B/2 \rfloor$.

En effet, posons $u = u_0u_1\dots u_n$ et $v = v_1\dots v_n$ en base B . Calculons $q' = \min(\lfloor (u_0 * B + u_1)/v_1 \rfloor, B - 1)$. Alors

Lemme 1: $q' \geq q$

Lemme 2: Si $v_1 \geq \lfloor B/2 \rfloor$, alors $q' - 2 \leq q \leq q'$

DIVISION DE u PAR v (Knuth vol2, p.272)

Soient $u = u_1u_2\dots u_{m+n}$ et $v = v_1\dots v_n$ Calculer $\lfloor u/v \rfloor = q_0q_1\dots q_m$ et $u \bmod v = r_1\dots r_n$

- 1- On normalise u et v en les multipliant par $d = \lfloor B/v_1 + 1 \rfloor$
- 2- Pour j variant de 0 à m , faire la séquence 3-4-5-6:
- 3- Si $u_j = v_1$, alors $q' = B - 1$,
sinon $q' = \lfloor (u_j * B + u_{j+1})/v_1 \rfloor$.
Tant que $q' * v_2 > (u_j * B + u_{j+1}) * B + u_{j+2}$, décrémenter q' .
- 4- Remplacer $u_j\dots u_{j+n}$ par $u_j\dots u_{j+n} - q' * v$.
- 5- $q_j \leftarrow q'$
Si résultat précédent négatif, faire 6:
- 6- Décrémenter q_j et additionner v à $u_j\dots u_{j+n}$
- 7- $q_0q_1\dots q_m$ est le résultat
 $u_{m+1}\dots u_{m+n}/d$ est le reste (car faut dénormaliser)

Cryptographie à clé publique

Eviter le partage des secrets. La méthode RSA utilise un système à 2 clés: une publique e et une autre secrète d . (Cf. le cours Morain en majeure Info ou le livre "Applied Cryptography" de Schneier.)

Soit $n = pq$ où p, q sont premiers. On encode avec une clé e première avec $(p-1)(q-1)$. On décode avec une clé d telle que

$$ed \equiv 1 \pmod{(p-1)(q-1)}$$

On suppose les messages m coupés en morceaux $m_i < n$, et on fait

$$c_i = m_i^e \pmod n$$

On décode par

$$m_i = c_i^d \pmod n$$

On vérifie que

$$c_i^d \equiv m_i^{ed} \equiv m_i \pmod n$$

On oublie donc p et q . La clé e est publique. Typiquement, pq a une centaine de chiffres décimaux.

Grands nombres premiers

Il faut donc trouver de grands nombres premiers p, q . (50 à 100 chiffres). Pour aller vite, on fait des tests probabilistes pour savoir si p est premier:

Méthode 1 (Lehmann)

1. Choisir un nombre aléatoire a tel que $a < p$
2. Calculer $a^{(p-1)/2} \bmod p$
3. Si $a^{(p-1)/2} \not\equiv 1$ ou -1 , répondre NON.
4. Sinon p est premier à 50%.

(On itère donc pour 10 valeurs de a).

Méthode 2 (Rabin-Miller, 1980)

Soit b la plus grande puissance de 2 telle que $p = 1 + 2^b m$

1. Choisir un nombre aléatoire a tel que $a < p$
2. Poser $j = 0$ et $z = a^m \bmod p$
3. Si $z = 1$ ou $z = p - 1$, alors répondre OUI.
4. Si $j > 0$ et $z = 1$, alors répondre NON.
5. Faire $j = j + 1$. Si $j < b$ et si $z = z^2 \bmod p \neq p - 1$, revenir en (4). Si $z = p - 1$, alors répondre OUI.
6. Si $j = b$ et $z = p - 1$ alors répondre NON.

Sur un nombre de 256 bits, se tromper après 6 tests est inférieur à $1/2^{51}$.