TP Caml 3 & 4: Cryptographie par RSA

lionel.rieg@ens-lyon.fr

14 & 21 octobre 2008 – 28 octobre & 18 novembre 2008

1 Principe de RSA

Le chiffrement [1] RSA est le plus connu des crypto-systèmes dit *asymétriques* ou *à clé publique*. Cette appellation vient de l'inégale répartition des informations entre le chiffreur et le déchiffreur : la clé publique est connue de tous tandis que la clé privée n'est connue que d'une personne. Ainsi, si tout le monde peut chiffrer des messages avec la clé publique, seul le possesseur de la clé privée est en mesure de les déchiffrer.

RSA repose en substance sur la difficulté de calculer une racine d-ème dans $(\mathbb{Z}/n\mathbb{Z})^*$. L'étape préliminaire au chiffrement est la création des clés de la façon suivante :

1. trouver deux grands entiers premiers aléatoires p et q
2. calculer $n = pq$
3. calculer $\varphi(n) = (p-1)(q-1)$
4. trouver e (***e**ncryption exponent*) premier avec $\varphi(n)$
5. calculer d (***d**ecryption exponent*) l'inverse de e modulo $\varphi(n)$

Les clés sont alors :
— (e, n) la clé publique
— (d, n) la clé privée

Comme $(\mathbb{Z}/n\mathbb{Z})^*$ est de cardinal $\varphi(n)$, si m est premier avec n, $(m^e)^d = m^{ed} = m^{k\varphi(n)+1} = m \mod n$. Le chiffrement se fait donc en élevant l'entier m qu'on souhaite envoyer à la puissance e modulo n. Le déchiffrement se fait quant à lui en élevant le cryptogramme à la puissance d toujours modulo n. Puisque (e, n) est public, n'importe qui peut chiffrer mais seul le possesseur de la clé privée (d, n) sait déchiffrer. La sûreté du système vient de l'impossibilité actuelle de retrouver m à partir de $m^e \mod n$ de façon efficace.

2 Implémentation de RSA

2.1 Outils arithmétiques

Comme on l'aura remarqué, cette technique utilise de nombreuses opérations arithmétiques, à savoir :
— un générateur aléatoire de nombres
— le calcul du pgcd
— un test de primalité
— l'algorithme d'Euclide étendu pour le calcul de l'inverse modulaire
— l'exponentiation modulaire

Et pour que ce ne soit pas trop simple, on ne peut pas utiliser le type `int` car les entiers que l'on va utiliser seront beaucoup trop grands. On va donc utiliser le type `num` défini dans le module de calcul en précision arbitraire. Je rappelle qu'il faut pour cela ouvrir le module avec un

```
#open "num";;
```

Ainsi, sauf précision contraire, les entiers seront dans tout ce TP de type `num`.

1. Remarquez que *cryptage* est un anglicisme banni par l'Académie Française

Comme on n'utilise pas l'arithmétique usuelle de Caml Light, les opérations élémentaires +, −, ∗, /, mod, quo sont redéfinis pour le type num de manière agréable à utiliser (*i.e.* infixe) : +/, −/, ∗/, //, modN et quoN. Les autres primitives auxquelles vous aurez droit sont square_num, sqrt_num, string_of_num, num_of_string, num_of_int et int_of_num. Cette dernière fonction retourne un résultat modulo 2^{31} car on perd de la précision (selon la machine, le modulo peut varier).

On ne se préoccupera pas ici du générateur aléatoire qui est un problème hautement non trivial et on utilisera simplement la fonction random_num que je vous fournis. Pour ceux que cela intéresse, ce générateur très basique utilise la méthode du carré médian de von Neumann.

► **Question 0** Définir les entiers 0, 1 et 2 de type num.

► **Question 1** Écrire une fonction qui calcule le pgcd de deux entiers. On peut la faire au choix de façon récursive ou itérative mais elle devra être curryfiée et aura donc le type

```
pgcd : num → num → num
```

► **Question 2** Écrire un test de primalité naïf qui, sur l'entrée n, calcule le pgcd de n et k pour $1 \leqslant k \leqslant \lfloor \sqrt{n} \rfloor$ et vérifie qu'il vaut bien à chaque fois 1.

► **Question 3** Écrire une fonction qui génère un nombre premier aléatoire en prenant un nombre aléatoire et en testant sa primalité. Afin de permettre un contrôle plus fin, la fonction prendra en arguments deux entiers inf et range qui seront respectivement une borne inférieure sur la valeur du nombre et la longueur de l'intervalle dans lequel on autorise la recherche.

► **Question 4** Écrire une fonction prime_with telle que prime_with n rendra un nombre e choisi aléatoirement entre 2 et n et qui soit premier avec n.

► **Question 5** Écrire l'algorithme d'Euclide étendu qui servira pour l'inversion modulaire. Il aura pour type

```
extended_pgcd : num → num → num * num
```

de sorte que extended_pgcd a b renvoie un couple (u, v) de coefficients de Bezout pour a et b, *i.e.* u et v vérifient $au + bv = \text{pgcd}(a, b)$.

► **Question 6** Écrire enfin la dernière opération arithmétique qui manque, l'exponentiation modulaire rapide. Elle sera de type

```
modular_exp : num → num → num → num
```

et on veillera à calculer les résidus modulo n après chaque multiplication, sans quoi les résultats peuvent très vite devenir énormes.

2.2 Implémentation du protocole

On va commencer par générer une paire de clés. Pour faire la distinction entre clés privées et publiques, on utilise le type suivant :

```
type key = Private of num * num | Public of num * num;;
```

► **Question 7** Écrire une fonction qui prend en paramètre deux entiers inf et range de même signification qu'à la question 3 et fabrique une paire de clés à l'aide de la méthode décrite dans la première partie.

► **Question 8** Écrire enfin les fonctions de chiffrement et de déchiffrement d'un entier. On veillera à retourner une erreur dans le cas où le mauvais type de clé est donné. Elles seront de type :

```
encode : key → num → num
decode : key → num → num
```

► **Question 9** On souhaite plutôt chiffrer des messages que des nombres, aussi il faut une fonction de conversion de l'un vers l'autre. Écrire donc une fonction qui transforme une chaîne de caractères en un entier. Pour cela, on verra un caractère comme un chiffre dans une base de numération à 256 chiffres dont le numéro peut être obtenu à l'aide de la fonction int_of_char. Le type de la fonction de conversion sera :

```
encode_string : num → string → num
```

Pour être sûr que le résultat soit premier avec n, on pourra rajouter un caractère arbitraire à la fin du message dont on ajustera la valeur. Écrire également la fonction réciproque.

► **Question 10** Tester tout ce que qui a été écrit jusque là (même si cela devrait déjà avoir été fait par petits bouts au fur et à mesure) en chiffrant un message. On peut même envisager d'en échanger entre les postes par l'intermédiaire du répertoire de partage.

Les deux parties suivantes présentent deux directions dans lesquelles on peut prolonger ce TD et elles sont totalement indépendantes. Choisissez celle qui vous plaît le plus.

3 Un meilleur test de primalité

Le test de primalité qu'on utilise ici est évidemment extrêmement coûteux (sa complexité est exponentielle en $\log(n)$) et donc inutilisable en pratique. On lui préfère donc des tests probabilistes dont le taux d'erreur peut être rendu si faible qu'il est illusoire de vouloir faire mieux, l'ordinateur ayant davantage de chances d'imploser durant le calcul (ou plus probablement de faire une erreur matérielle), voire tout autre événement extravagant hautement improbable.

3.1 Le test de Fermat

La première amélioration est d'utiliser le *petit théorème de Fermat* :

$$p \text{ premier } \implies \forall a \in [\![1,\ p-1]\!],\ a^{p-1} \equiv 1 \mod n$$

Si un entier n satisfait cette propriété pour de nombreux a, il y a de bonnes chances que n soit premier. En effet, à supposer que si n est composé alors il existe un nombre a pour lequel $a^{n-1} \not\equiv 1 \mod n$, la probabilité d'erreur est majorée par 1/2 car l'ensemble des nombres qui passent le test forme un sous-groupe strict de $(\mathbb{Z}/n\mathbb{Z})^*$. Pour tester cela, on choisit a au hasard entre 1 et $n-1$, on l'élève à la puissance $n-1$ et on vérifie que le résultat est bien congru à 1 modulo n. On répète ce test autant de fois que nécessaire pour atteindre la précision voulue.

► **Question 11** Écrire un test de primalité à l'aide du test de Fermat en utilisant l'exponentiation modulaire rapide. L'algorithme prendra en paramètre le nombre maximum de tests à réaliser et l'entier à tester.

L'inconvénient majeur de cette méthode est qu'il existe des nombres composés, dit de Carmichael, pour lesquels tous les entiers strictement plus petits passent le test de Fermat. Sur ces entiers, l'algorithme précédent se trompe nécessairement.

3.2 Test de Rabin-Miller

On veut modifier le test de Fermat pour ne pas se laisser abuser par les nombres de Carmichael. Pour cela, on rajoute une technique de détermination de composition : on va chercher les racines carrées de 1. En effet, si n est premier, $\mathbb{Z}/n\mathbb{Z}$ est un corps donc il y a exactement deux solutions à l'équation $X^2 = \overline{1}$: $\overline{1}$ et $\overline{-1}$. À l'inverse, si n n'est pas premier, il peut y en avoir davantage : par exemple, dans $\mathbb{Z}/15\mathbb{Z}$, les solutions sont $\overline{-1}$, $\overline{1}$, $\overline{-4}$ et $\overline{4}$. Voici comment on va utiliser les racines carrées de $\overline{1}$ pour déterminer si n est premier :

1. on élimine le cas où n est pair
2. on écrit $n - 1 = 2^S \cdot t$ avec t impair
3. on choisit aléatoirement a dans $[\![2,\ n-2]\!]$
4. si $a^t \equiv \overline{1} \mod n$, on s'arrête (en renvoyant n premier)
5. on calcule $(a^t)^{2^k}$ pour $k \in [\![1,\ s-1]\!]$
6. si aucun des $(a^t)^{2^k}$ ne vaut -1, alors n est composé

► **Question 12** Écrire une fonction qui décompose l'entier n donné en argument en un couple (s, t) tel que $n = 2^s \cdot t$ avec t impair.

► **Question 13** Implémenter à présent le test de primalité de Rabin-Miller.

► **Question 14** Tester le gain de temps que procure cet algorithme comparé à l'algorithme naïf.

On peut remarquer que ce test est l'un des deux utilisés en pratique, le second étant celui de Soloway et Strassen, basé sur les courbes elliptiques. Il existe aussi depuis 2002 un algorithme polynomial déterministe pour décider la primalité d'un nombre mais son exposant est trop important (supérieur à 10) pour qu'il soit utilisable en pratique.

4 Attaques de RSA

Casser un crypto-système revient à être capable de retrouver n'importe quel message à partir de son cryptogramme. Il existe de nombreuses techniques d'attaque de RSA mais aucune ne représente une menace sérieuse lorsqu'il est bien utilisé. En effet, elles se basent la plupart du temps sur des valeurs particulières des exposants e et d ou des défauts d'implémentation. Pour votre culture, les plus performants actuellement utilisent à fond l'algorithme LLL de réduction de réseaux.

On ne va ici étudier que deux algorithmes simples pour la plus élémentaire des attaques, la force brute. Elle est très coûteuse en terme de ressources mais ne repose sur aucun cas particulier : l'idée est simplement de factoriser n. En effet, une fois que l'on possède p et q, on peut calculer $\varphi(n)$ et déduire d de e par inversion modulaire.

4.1 Algorithme naïf

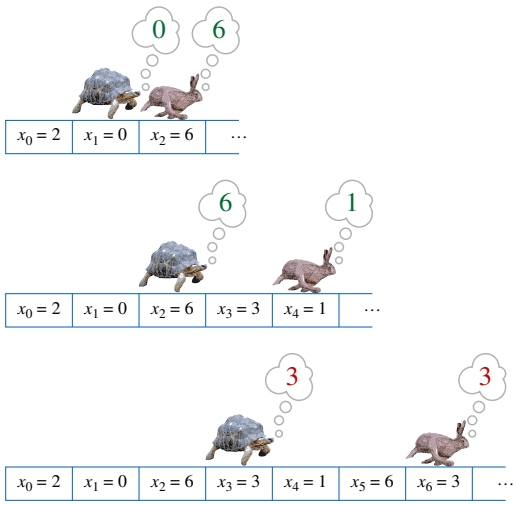
► **Question 15** Écrire un algorithme de factorisation naïf basé sur le même principe que le test de primalité de la question 2.

Cet algorithme trouve à coup sûr un facteur s'il en existe un mais il requiert $\lfloor\sqrt{n}\rfloor$ calculs de pgcd. Il a donc un coût exponentiel en la taille $\log n$ de n et ne peut pas être utilisé en pratique car n est bien trop grand.

4.2 Algorithme rho de Pollard

On peut faire mieux avec l'*algorithme rho* de Pollard, algorithme probabiliste basé sur deux faits : le paradoxe des anniversaires et l'algorithme du lièvre et de la tortue. Commençons par examiner l'algorithme du lièvre et de la tortue.

Il est dû à Floyd et permet de détecter les cycles dans une suite du type $u_{n+1} = f(u_n)$ en utilisant le fait que si une même valeur apparaît deux fois dans la suite, alors il y a un cycle. Le fonctionnement de l'algorithme est très simple comme nous allons le voir.
Le lièvre et la tortue partent initialement d'un même point. À chaque étape, la tortue avance de 1 dans la suite et le lièvre de 2. Si le lièvre croise la tortue (*i.e.* ils ont la même valeur), alors il y a un cycle.
Réciproquement, s'il y a un cycle dans la suite, alors la tortue va y entrer au bout d'un certain temps. Une fois les deux dans le cycle, le lièvre rattrape la tortue de 1 pas à chaque étape donc il vont finir par se rencontrer.
Sur l'exemple ci-contre avec la suite $2, 0, 6, 3, 1, 6, 3, 1, 6, \ldots$, on voit que le lièvre rattrape la tortue au bout de 3 étapes.



crédit image : David Eppstein, Wikimedia

► **Question 16** Implémenter l'algorithme du lièvre et de la tortue. Il sera de type

```
cycle_detect : (num → num) → num → num → bool
```

et, en plus de la fonction génératrice de la suite, il prendra en argument une graine pour amorcer la suite et le nombre maximum d'étapes à réaliser.

Le paradoxe des anniversaires nous dit que parmi 23 personnes, la probabilité d'en avoir deux nées le même jour est supérieure à 1/2. En considérant les dates de naissance (supposées aléatoires) en jours, cela revient à trouver deux entiers congruents modulo 365. De même, trouver deux entiers aléatoires dont la différence soit multiple d'un entier p quelconque avec une probabilité supérieure à $1/2$ ne nécessite qu'environ $\sqrt{2p\ln 2} \approx 1,177\sqrt{p}$ tentatives.

L'idée de l'algorithme rho est alors d'utiliser une fonction de $[\![1, n]\!]$ dans $[\![1, n]\!]$ comme générateur d'une suite (u_n) pseudo-aléatoire de nombres dont on espère que les différences seront multiples d'un facteur premier de n. On calcule ensuite le pgcd de n et de la différence des deux entiers u_i et u_{2i}. L'algorithme du lièvre et de la tortue permet de détecter les cycles dans la suite des nombres générés afin de ne pas répéter les mêmes tests. Le nom même d'*algorithme rho* rappelle que la trajectoire du générateur va revenir sur elle-même pour former un cycle, tel un ρ.

► **Question 17** Implémenter l'algorithme rho de factorisation de Pollard.

► **Question 18** Tenter de décrypter[2] un cryptogramme. Prendre n pas trop grand !

2. *i.e.* déchiffrer sans avoir la clé de déchiffrement

5 Solutions

▸ **Question 1** Une version récursive :

```
let rec pgcd a b =
  if a =/ zero then b else pgcd (b modN a) a;;
```

et une autre impérative :

```
let pgcd a b =
  let r = ref a in
  let s = ref b in
    while !s <>/ zero do
      let swap = !s in
        s := !r modN !s;
        r := swap
    done;
    !r ;;
```

▸ **Question 2** Voici une version impérative :

```
let is_prime n =
  let i = ref deux in
  let pgcd_res = ref un in
    while !pgcd_res =/ un && !i <=/ sqrt_num n do
      pgcd_res := pgcd !i n;
      i := !i +/ un
    done;
    !pgcd_res =/ un;;
```

et une version récursive, plus courte comme souvent :

```
let is_prime n =
  let rec is_prime_aux n bound k =
    k >/ bound || ((pgcd n k =/ un) && (is_prime_aux n bound (k +/ un)))
  in
    is_prime_aux n (sqrt_num n) deux;;
```

▸ **Question 3** Je ne donne ici qu'une version impérative mais une version récursive existe évidemment.

```
let generate_prime inf range =
  let guess = ref (inf +/ (random_num range)) in
    while not is_prime !guess do
      guess := inf +/ (random_num range)
    done;
    !guess ;;
```

Cette question et la suivante utilisent des algorithmes dont la terminaison n'est pas assurée. En effet, le générateur aléatoire pourrait ne générer que des nombres non premiers mais cela n'arrive qu'avec une probabilité très faible. En pratique, ce n'est pas un problème car il y a suffisamment de nombres premiers.

▸ **Question 4** Cette question (tout comme la précédente) illustre à merveille la concision de la récursivité par rapport au style impératif :

```
let rec prime_with n =
  if n =/ deux then un
  else
    let r = deux +/ random_num (n -/ deux) in
      if pgcd r n =/ un then r else prime_with n;;
```

vs.

```
let prime_with n =
  if n =/ deux then un
  else
    let i = ref un in
    let pgcd_res = ref deux in
      while !pgcd_res <>/ un do
        i := deux +/ random_num (n -/ deux);
        pgcd_res := pgcd !i n
      done;
      !i ;;
```

▸ **Question 5**

```
let rec extended_pgcd a b =
  if a modN b =/ zero then (zero, un)
  else
    let x, y = extended_pgcd b (a modN b) in
      (y, x -/ y */ (a quoN b));;
```

Cette question et la suivante sont de bons exemples d'améliorations de l'implémentation lorsqu'on accepte de se fatiguer un peu : on peut écrire une version récursive terminale qui a l'avantage de n'utiliser qu'un espace mémoire constant (linéaire en fait puisqu'il faut stocker les entiers qui ne sont pas bornés).

```
let extended_pgcd a b =
  let rec term_rec_extended_pgcd r_i (s_i, t_i) r_j (s_j, t_j) =
    if r_i modN r_j =/ zero then (s_j, t_j)
    else
      let q = r_i quoN r_j in
        term_rec_extended_pgcd
          r_j (s_j, t_j) (r_i modN r_j) (s_i -/ s_j */ q, t_i -/ t_j */ q)
  in
    if b =/ zero then (un, zero)
    else term_rec_extended_pgcd a (un, zero) b (zero, un);;
```

Pour justifier la correction de cet algorithme (de term_rec_extended_pgcd en fait), on utilise l'invariant suivant : $r_i = s_i \cdot a + t_i \cdot b$ et idem pour $r_j = r_{i+1}$. Dans ce cas, le cas d'arrêt et correct car $r_i \mod r_j = 0$ indique que r_j est le pcgd donc les coefficients de Bezout sont (s_j, t_j). Pour l'appel récursif, on dispose des relations

$$r_i = s_i a + t_i b \qquad r_{i+1} = s_{i+1} a + t_{i+1} b$$

et on écrit la division euclidienne de r_i par r_{i+1} :

$$\begin{aligned} r_{i+2} &= r_i - q_{i+1} r_{i+1} \\ &= (s_i a + t_i b) - q_{i+1}(s_{i+1} a + t_{i+1} b) \\ &= (s_i - q_{i+1} s_{i+1}) a + (t_i - q_{i+1} t_{i+1}) b \end{aligned}$$

Pour trouver un tel algorithme, une autre solution consiste à l'écrire de façon impérative puis à transformer le code. Cette méthode est néanmoins moins générale.

▸ **Question 6** Cette fois-ci, une version récursive :

```
let rec modular_exp m e n =
  if e =/ zero then un
  else
    let mm = modular_exp m (e quoN deux) n in
      if e modN deux =/ zero then (square_num mm) modN n
      else (((square_num mm) modN n) */ m) modN n;;
```

et une récursive terminale :

```
let modular_exp m e n =
  let rec rec_term_mod_exp m e n r =
  if e =/ zero then r
  else
    let m_2 = (square_num m) modN n in
      if e modN deux =/ zero
      then rec_term_mod_exp m_2 (e quoN deux) n r
      else rec_term_mod_exp m_2 (e quoN deux) n ((r */ m) modN n)
  in
    rec_term_mod_exp m e n un;;
```

▸ **Question 7** Dans cette question, il ne faut pas oublier que l'inverse de e modulo $\varphi(n)$ peut être calculé (par l'algorithme d'Euclide étendu car $ud + v\varphi(n) = 1$ donne $ud \equiv 1 \mod \varphi(n)$) comme un nombre négatif, ce qui pose un problème pour l'exponentiation modulaire. De plus, p et q doivent être distincts sans quoi $\varphi(n) = p(p-1)$.

```
let generate_keys inf range =
  let p = generate_prime inf range in
  let q =
    let guess = ref (generate_prime inf range) in
      while p =/ !guess do guess := generate_prime inf range done;
      !guess in
  let n = p */ q in
  let phi_n = (p -/ un) */ (q -/ un) in
  let e = prime_with phi_n in
  let d = (phi_n +/ fst (extended_pgcd e phi_n)) modN phi_n in
    (Private (d, n), Public (e, n));;
```

▸ **Question 8**

```
let encode priv message =
  match priv with
    | Private _ → failwith "encode: expect public key for encoding"
    | Public (e, n) → modular_exp message e n;;

let decode pub message =
  match pub with
    | Public _ → failwith "decode: expect private key for decoding"
    | Private (d, n) → modular_exp message d n;;
```

▸ **Question 9** Il n'est pas besoin de calculer les puissances de 256 car le calcul peut se faire de façon incrémentale :

$$\overline{a_n \dots a_1 a_0}^{256} = a_n \cdot 256^n + \cdots + a_1 \cdot 256^1 + a_0 = (a_n \cdot 256^{n-1} + \cdots + a_1) \cdot 256 + a_0 = (\overline{a_n \dots a_1}^{256}) \cdot 256 + a_0$$

```
let encode_string pub s =
  match pub with
    | Private _ → failwith "encode_string: expect public key"
    | Public (_, n) →
        let base = num_of_int 256 in
        let res = ref zero in
          for i = 0 to (string_length s) - 1 do
            res := base */ !res +/ num_of_int (int_of_char s.[i])
          done;
          (* !res, !res +1 et !res +2 ne peuvent être tous multiples de p ou q *)
          res := !res */ base;
          if pgcd !res n =/ un then !res
          else if pgcd (!res +/ un) n =/ un then !res +/ un else !res +/ deux;;
```

Pour le décodage, on ne peut connaître la taille de la chaîne à décoder car il n'existe pas de logarithme pour les entiers de taille arbitraire. On peut donc ou bien augmenter la taille de la chaîne au fur et à mesure (à la rigueur passer par une liste de caractères pour ne pas faire un nombre linéaire de concaténation), ou bien faire une première lecture de la chaîne qui ne sert qu'évaluer sa taille. C'est cette seconde solution que j'ai choisie ici.

```
let decode_string n =
  let base = num_of_int 256 in
  let nb = ref (n quoN base) in (* on supprime le dernier caractère *)
  let taille = ref 1 in
    while !nb >=/ base do (* calcul de la taille *)
      incr taille ;
      nb := !nb quoN base
    done;
    nb := n quoN base;
    let s = make_string ! taille '?' in
      for i = ! taille - 1 downto 0 do (* écriture de la chaîne *)
        s.[i] <- char_of_int (int_of_num (!nb modN base));
        nb := !nb quoN base
      done;
      s ;;
```

▸ **Question 10** Différents test rapides qui testent séparément puis combinent tout ce qui a été fait jusqu'à présent :

```
let test_code_string m =
  let n = encode_string (snd k) m in
    m, n, decode_string n ;;

let test_encode nb =
  let n = num_of_string nb in
  let sqrt_n = sqrt_num n in
  let priv, pub =
    new_generate_keys (sqrt_n */ (sqrt_num sqrt_n)) sqrt_n in
  let enc = encode pub n in
    n, priv, pub, enc, n =/ decode priv enc ;;

let test message =
  let log_size = int_of_float (1.205 *. float_of_int (string_length message)) in
  let size = num_of_string ("1e" ^ (string_of_int (2 + log_size))) in
  let priv, pub = new_generate_keys size (sqrt_num size) in
  let nb = encode_string pub message in
  let enc = encode pub nb in
  let dec = decode priv enc in
    message, nb, priv, pub, enc, dec, decode_string dec, nb =/ dec ;;
```

▸ **Question 11** Le nombre de tests à réaliser est un entier de type int et non num car cela est amplement suffisant : on peut atteindre un taux d'erreur d'à peine $1/2^{2^{31}-1} \approx 10^{-646456992}$! Voici une version impérative :

```
let fermat precision n =
  let k = ref 0 in
  let a = ref un in
  let res = ref un in
    while !res =/ un && !k < precision + 1 do
      a := deux +/ random_num (n -/ quatre);
      res := modular_exp !a (n -/ un) n;
      incr k
    done;
    !k = precision + 1;;
```

et une version récursive (terminale) :

```
let rec fermat precision n =
  precision = 0 || (* pareil que « if precision = 0 then true else » *)
  let a = deux +/ random_num (n -/ quatre) in
    (modular_exp a (n -/ un) n =/ un && fermat (precision - 1) n);;
```

▸ **Question 12**

```
let decompose n =
  if n =/ zero then zero, zero
  else
    let s = ref zero and t = ref n in
      while !t modN deux =/ zero do
        t := !t quoN deux;
        s := !s +/ un
      done;
      !s, !t;;
```

▸ **Question 13** Comme une itération du test de Rabin-Miller est assez compliquée, il ne faut pas hésiter à découper le code en fonctions très simples et suffisamment courtes pour limiter les risques d'erreurs. Notamment, il vaut mieux séparer le test et les répétitions pour atteindre la précision souhaitée. De plus, les **if** a x y **then** true **else** b z t qui apparaissent sont avantageusement remplacés par des (a x y) || (b z t).

```
let rec search_square_root n a s =
  a =/ n -/ un || (* et non -1 car les résidus sont positifs *)
  (s >/ un && search_square_root n ((square_num a) modN n) (s -/ un));;

let rabin_miller_step n s t =
  let a = deux +/ random_num (n -/ quatre) in
  let a_t = modular_exp a t n in
    a_t =/ un || (* si a^t = 1, inutile de chercher des racines carrées de 1 *)
      search_square_root n a_t s;;
```

À nouveau, il vaut mieux séparer la répétition du test de Rabin-Miller et sa préparation qui consiste d'une part en la gestion du cas particulier n pair et d'autre part en la décomposition de n en s et t. Décomposer n par avance évite de le refaire à chaque itération du test de Rabin-Miller.

```
let rec iter_rabin_miller precision n s t =
  precision = 0 ||
  (rabin_miller_step n s t && iter_rabin_miller (precision - 1) n s t);;

let rabin_miller precision n =
  if n modN deux =/ zero then n =/ deux
  else let s, t = decompose (n -/ un) in iter_rabin_miller precision n s t;;
```

▸ **Question 14** J'ai choisi ici arbitrairement de limiter à 50 le nombre d'itérations. Comme la probabilité d'erreur d'un test de Rabin-Miller est inférieure à $\frac{1}{4}$, cela nous donne une probabilité d'erreur inférieure à $2^{-100} \leqslant 10^{-30}$, précision amplement suffisante.

```
let new_generate_prime inf range =
  let guess = ref (inf +/ (random_num range)) in
    while not rabin_miller 50 !guess do
      guess := inf +/ (random_num range)
    done;
    !guess;;

let new_generate_keys inf range =
  let p = new_generate_prime inf range in
  let q =
```

```
    let guess = ref (new_generate_prime inf range) in
      while p =/ !guess do guess := new_generate_prime inf range done;
      !guess in
  let n = p */ q in
  let phi_n = (p -/ un) */ (q -/ un) in
  let e = prime_with phi_n in
  let d = (phi_n +/ fst (extended_pgcd e phi_n)) modN phi_n in
   (Private (d, n), Public (e, n));;
```

Un petit test montre que l'amélioration est significative. Par exemple

```
let k1 =
  generate_keys (num_of_string "10000000000000000") (num_of_string "1000");;
let k2 =
  new_generate_keys (num_of_string "10000000000000000") (num_of_string "1000");;
```

k1 demande plus de 20 minutes pour être calculé et k2 moins de 2 secondes.

► **Question 15**

```
let rec factorize_aux factors_list i n =
  if i >/ sqrt_num n then n :: factors_list
  else
    if n modN i =/ zero
    then factorize_aux (i :: factors_list ) i (n quoN i)
    else factorize_aux factors_list (i +/ un) n;;

let factorize n = rev ( factorize_aux [ ] deux n );;
```

Afin de donner une forme canonique au résultat, j'ordonne la liste des facteurs par ordre croissant.

► **Question 16**

```
let cycle_detect f u_0 max_iter =
  let tortoise = ref (f u_0) in
  let hase = ref (f (f u_0)) in
  let nb_step = ref un in
    while ! tortoise <>/ !hase && !nb_step < max_iter do
       tortoise := f ! tortoise ;
      hase := f (f !hase)
    done;
    !nb_step =/ max_iter;;
```

► **Question 17**

```
let rec rho_aux nb_iter acc f n =
  if nb_iter =/ zero || n =/ un then acc
  else
    let a = deux +/ (random_num (n -/ quatre)) in
    let tortoise = ref (f a) in
    let hase = ref (f (f a)) in
    let pgcd_res = ref un in
      while !pgcd_res =/ un && !tortoise <>/ !hase do
        pgcd_res := pgcd n (abs_num (!hase -/ ! tortoise ));
         tortoise := f ! tortoise ;
        hase := f (f !hase)
      done;
      if !hase =/ ! tortoise
      then rho_aux ( nb_iter -/ un) acc f n
      else rho_aux nb_iter (!pgcd_res :: acc) f (n quoN !pgcd_res );;
```

```
let rho f n =
  let cst = ceiling_num ((num_of_int 1178) // (num_of_int 1000)) in
    rev (rho_aux (cst */ sqrt_num (sqrt_num n)) [] f n);;
```

Cet algorithme n'effectue pas la factorisation en facteurs premiers, il ne fait que trouver des facteurs mais ceux-ci ne sont pas nécessairement premiers. Cela n'est pas un problème ici car n est le produit de deux nombres premiers.

► **Question 18**

```
let decrypt factorize key message =
  match key with
    | Private _ → failwith "decrypt: should not have the private key"
    | Public (e, n) →
        match factorize n with
          | [] → failwith "decrypt: unable to factorize n"
          | [p; q] →
              let phi_n = (p -/ un) */ (q -/ un) in
              let d, _ = extended_pgcd e phi_n in
                modular_exp message d n
          | _ → failwith "decrypt: n should be composite";;

let naive_decrypt = decrypt factorize ;;
```