TP RSA OCaml (corrigé OCaml, Python)
Cryptographie, authentification par la méthode RSA

Les listings OCaml ou Python sont présentés en utilisant les environnements verb, lstlisting, verbatim, alltt de LaTeX.

- Les espaces superflus et les indentations, visibles dans le pdf, disparaissent lors d'un copier-coller. Ils on été remplacés par des "faux" espaces ␣ invisibles dans le pdf, et après un copier-coller depuis le pdf, il suffit de remplacer tous ces "faux" espaces par de "vrais" espaces pour obtenir le code source initial bien indenté.
- Dans le passage LaTeX vers pdf, les ' (quote simple) et les " (double quote) sont transformés en ' (apostrophe) et " (guillemets), ce qui pose un problème pour récupérer le code source des programmes par un "copier-coller" depuis le pdf. Pour éviter cette transformation dans les environnements verb, lstlisting, verbatim, alltt, il suffit de déclarer le package LaTeX upquote (\usepackage{upquote}).

La méthode de chiffrement **RSA** (R.**R**ivest, A.**S**hamir, L.**A**dleman, 1978) est une des méthodes utilisées pour crypter des informations confidentielles. Il s'agit d'une méthode dite "**à clé publique**", **asymétrique** :

1. Une personne ("**Alice**") met au point un chiffre (méthode de chiffrement) RSA constitué de deux clés :
   - une clé publique **P**, connue des autres personnes ("**Bob(s)**")
   - une clé secrète **S**, connue d'elle seule.
2. Lorsque **Bob** veut envoyer un message confidentiel **M** à **Alice**, il le chiffre avec la clé publique **P** de **Alice** et envoie le résultat **P(M)** à **Alice** qui le déchiffre avec sa clé secrète **S** pour obtenir **M = S(P(M))**.
3. Lorsque **Alice** veut envoyer un message (public) **M** à **tous ses correspondants**, en certifiant qu'elle en est bien l'auteur, elle le chiffre avec sa clé secrète **S** et envoie le résultat **S(M)** à **ses correspondants** qui le déchiffrent avec la clé publique **P** pour obtenir **M = P(S(M))**.

Remarque. **Les algorithmes de chiffrement et de déchiffrement sont publics.**

Un tel système ne fonctionne que si

- **S(P(M)) = P(S(M)) = M**
- Découvrir la clé secrète **S** est au moins aussi difficile que de décrypter un message chiffré avec la clé publique (le vilain "**Oscar**" ne doit pas être capable de décrypter un message chiffré destiné à **Alice**, ni de se faire passer pour **Alice**).

1 Description de la méthode de chiffrement RSA

L'indicateur d'Euler de $n \in \mathbb{N}^*$, $\varphi(n)$, est le nombre d'entiers compris entre 1 et n (inclus) premiers avec n.

- $\varphi(1) = 1$ et si $n > 1$ est premier, $\varphi(n) = n-1$,
- si p et q sont deux entiers strictement positifs premiers entre eux, alors $\varphi(pq) = \varphi(p)\,\varphi(q)$.

Petit théorème de Fermat : Si p est premier et si a n'est pas un multiple de p, alors $a^{p-1} \equiv 1 \mod p$.

1.1 Construction par Alice d'un couple de clés (S, P)

Soient $p > 1$ et $q > 1$ des nombres premiers distincts, $n = pq$ leur produit,
e un nombre premier avec $\varphi(n) = (p-1)(q-1)$ et $d = e^{-1} \mod (p-1)(q-1)$.

Pour tout entier positif $u < n$, on a $(u^e)^d = u^{ed} \equiv u \mod n$.

- la clé publique d'**Alice** est le couple **P** = (n, e), qu'**Alice** diffuse à tous vents,
- la clé secrète d'**Alice** est le couple **S** = (n, d).

n est le module de chiffrement, e est l'exposant de chiffrement, d est l'exposant de déchiffrement.

Justificatif :

$ed \equiv 1 \mod (p-1)(q-1)$, donc il existe k tel que $ed = 1 + k(p-1)(q-1)$.

Si u n'est pas multiple de p ni de q, d'après le petit théorème de Fermat, $\begin{cases} u^{ed} = u^{1+k(p-1)(q-1)} = u\left(u^{p-1}\right)^{k(q-1)} \equiv u \mod p \\ u^{ed} = u^{1+k(p-1)(q-1)} = u\left(u^{q-1}\right)^{k(p-1)} \equiv u \mod q \end{cases}$

et si u est un multiple de p, $u \equiv 0 \mod p$ et $u^{ed} \equiv 0 \mod p$ (de même avec q).

Donc, pour tout entier u, $(u^e)^d = u^{ed} \equiv u \mod n$.

L'entier $u^{ed} - u$ est un multiple de p et de q, qui sont premiers distincts, donc un multiple de leur produit $pq = n$.

1.2 Chiffrement d'un message envoyé par Bob, à l'aide de la clé publique de Alice

- Le message initial M est transformé en une séquence (u) d'entiers strictement inférieurs à n (simple changement de représentation, par une méthode ordinaire, publique, connue et immuable).
- Chaque entier u de cette séquence est élevé à la puissance e modulo n
- La séquence obtenue, $(v) = (u^e \mod n) = P(M)$, est envoyée à **Alice**.

1.3 Déchiffrement du message reçu par Alice, à l'aide de sa clé secrète

- Chaque entier v de la séquence reçue est élevé à la puissance d modulo n : $v^d = (u^e)^d = u^{ed} \equiv u \mod n$, ce qui donne la séquence (u) de représentation numérique du message initial M.
- Il ne reste plus qu'à transformer la séquence (u) pour obtenir le message initial M (à l'aide du changement de représentation réciproque de celui utilisé par **Bob**).

1.4 Sécurité

1. Le vilain **Oscar** ne doit pas être capable de déchiffrer un message $c = m^e$ à partir de la clef publique (n,e) : l'extraction de la racine e-ième de c dans $\mathbb{Z}/n\mathbb{Z}$ doit être très difficile, voire impossible.
2. Le vilain **Oscar** ne doit pas pouvoir construire la clé secrète (n,d) à partir de la seule information publique (n,e), ce qui suppose que la recherche de d à partir de la seule connaissance de (n,e) doit être très difficile, voire impossible.

   Si **Oscar** arrive à "casser" n comme le produit pq, il trouvera facilement d tel que $d = e^{-1} \mod (p-1)(q-1)$ et sera alors en mesure de déchiffrer tous les messages destinés à **Alice** et aussi de se faire passer pour **Alice**.

Alice doit donc choisir les nombres premiers p et q très grands et tels qu'il soit (quasiment) impossible, même en mobilisant toutes les ressources informatiques de la planète terre et de la lune réunies, de retrouver ces deux nombres p et q uniquement à partir de l'information "$n = pq$ avec p et q premiers". **Alice** doit aussi veiller à choisir e tel qu'on ne puisse pas déchiffrer un message à partir de la clef publique (n,e).

Remarque. La connaissance des algorithmes de factorisation permet d'éviter les situations où la décomposition en facteurs premiers de n est aisée ...

2 Usage du chiffrement RSA

Chiffrer l'ensemble d'un message long avec l'algorithme RSA est coûteux en temps de calcul et il est préférable d'utiliser un algorithme de chiffrement symétrique, plus rapide.

Le chiffrement RSA est utilisé pour communiquer une clé de chiffrement symétrique, qui permet alors de poursuivre l'échange de façon confidentielle : **Bob** envoie à **Alice** une clé de chiffrement symétrique, chiffrée avec la clé RSA publique d'**Alice**, puis. **Alice** et **Bob** échangent des données par l'algorithme de chiffrement symétrique.

2.1 La méthode RSA comme méthode d'authentification

Supposons que **Alice** veuille envoyer un message à **Bob(s)** de telle sorte que **Bob(s)** soit convaincue que c'est bien **Alice** qui est l'envoyeur (et non un vilain **Oscar** qui voudrait se faire passer pour **Alice**) :

- **Alice** chiffre le message avec sa clé secrète (n,d) (elle est la seule à pouvoir le faire)
- et l'envoie à **Bob(s)** qui le déchiffre avec la clé publique (n,e) de **Alice**.

Remarque. Le message ne sera pas confidentiel, n'importe quel vilain **Oscar** pourra en prendre connaissance.

Pour échanger des messages confidentiels, il suffirait que **Bob** possède aussi son propre chiffre :

- **Alice** chiffre son message avec sa clé secrète, puis le résultat avec la clé publique de Bob
- **Bob** déchiffre avec sa clé secrète, puis déchiffre le résultat avec la clé publique de **Alice**

Ainsi **Bob** et **Alice** peuvent converser en toute sécurité et confidentialité !

2.2 La méthode RSA comme méthode de signature

Lorsque la méthode d'authentification est réservée aux messages qui sont des signatures, elle se transforme en une méthode d'authentification (ou de certification) de signature.

Les cartes à puces utilisent une méthode d'identification basée sur le RSA : chaque carte comporte un nombre entier chiffré par une clé secrète (n,d) connue de la seule autorité bancaire. Le déchiffrement de ce nombre par la clé publique (n,e) doit donner une valeur qui correspond à des informations rédigées en clair sur la carte.

Remarque. On peut aussi introduire, à l'aide d'un codage RSA, une preuve de date, ce qui est utile pour la certification d'antériorité des dépots (brevets, etc ...).

En France, la signature électronique d'un document a la même valeur légale qu'une signature manuscrite (loi de mars 2000).

3 Mise en œuvre pratique

3.1 Numérisation du message, transformation réciproque

Il ne s'agit pas ici d'un codage secret, mais d'un codage pratique.

Le message est (transformé en) une séquence d'octets, entiers compris entre 0 et 255.

Chiffrer, à l'aide de la clé publique (n,e), chaque octet u par $u^e \mod n$, conduit à associer un très grand nombre à chaque octet et ainsi à acheminer un énorme volume de données. On préfère grouper les octets en paquets de k octets consécutifs, formant un entier grand (mais plus petit que n), avant le chiffrement.

3.1.1 Compactage numérique par paquets d'au plus k caractères

Un paquet de k octets consécutifs $a_0, a_1, a_2, \cdots, a_{k-1}$, sera écrit sous la forme numérique

$$u = a_{k-1} + a_{k-2}\times 256^1 + a_{k-3}\times 256^2 + \cdots + a_1 \times 256^{k-2} + a_0 \times 256^{k-1} = ((\cdots(a_0\times 256 + a_1)\times 256 + \cdots) + a_{k-2})\times 256 + a_{k-1}$$

prenant au pire la valeur : $255\left(1+256+256^2+\cdots+256^{k-1}\right) = 255\,\dfrac{256^k-1}{256-1} = 256^k - 1$

Pour ne pas perdre d'information en calculant les puissances de u modulo n (pour pouvoir retrouver $u = (u^e)^d \mod n$), il est nécessaire que u soit strictement inférieur à n : **le facteur de compactage k doit être inférieur à $\lfloor \log_{256} n \rfloor$**.

3.1.2 Décompactage numérique des paquets de k (au plus) caractères

On divise $u = a_{k-1} + a_{k-2}\times 256^1 + a_{k-3}\times 256^2 + \cdots + a_1 \times 256^{k-2} + a_0 \times 256^{k-1}$, par 256 en retenant les restes jusqu'à ce que $u = 0$, ce qui donne dans l'ordre chronologique, les coefficients $a_{k-1}, a_{k-2}, \cdots a_1, a_0$.

Remarque. On n'a pas besoin de la connaissance du facteur de compactage k : le compactage, seul fait de l'envoyeur, n'est pas un obstacle pour le destinataire.

3.2 Outils

Dans le cas des `int`, positifs, limités à l'intervalle $[0, max\_int = 2^{62}-1]$, pour que les produits $x\times y, x\times y \mod n$ restent dans cet intervalle, on impose $0 \leqslant x,y < 2^{30}$ et $0 < n < 2^{30}$.

1. **Exponentiation rapide modulo n.**
   Principe de l'exponentiation rapide, calcul de x^e :

   avec $e \in \mathbb{N}$, $0 \leqslant e = 2q + r$ où $r \in \{0,1\}$, on obtient $x^e = x^r \times (x^q) \times (x^q)$ en ayant calculé de même x^q.

   Pour l'exponentiation rapide d'entiers modulo n, on fait les produits modulo n : $x^e = \left(x^r * \left((x^q * x^q) \mod n\right)\right) \mod n$.
2. **Plus grand diviseur commun :** si $y = 0$, $\text{pgcd}(x,y) = x$, sinon $\text{pgcd}(x,y) = \text{pgcd}(y, (x \mod y))$.
3. **Décomposition de Bezout :** **théorème de Bezout :** pour $x, y \in \mathbb{N}$, il existe u et v dans $\mathbb{Z}$ tels que $ux + vy = \text{pgcd}(x,y)$.
   Calcul de (u,v,w) tels que $ux + vy = w \quad (= \text{pgcd}(x,y))$:
   - si $y = 0$, $\text{pgcd}(x,y) = x$ et $1x + 0y = x$, d'où $(u,v,w) = (1,0,x)$ convient ;
   - sinon, on a $x = qy + r$ avec $0 \leqslant r < y$, et $\text{pgcd}(x,y) = \text{pgcd}(y,r)$,
     puis, avec u', v', w tels que $u'y + v'r = w' \quad$ (où w' = $\text{pgcd}(y,r) = \text{pgcd}(x,y)$),
     on a : $w' = u'y + v'r = u'y + v'(x - qy) = v'x + (u' - v'q) = ux + vy = w$ où $\begin{cases} u = v' \\ v = u' - v'q \\ w = w' \end{cases}$
4. **Inverse modulo n, division modulo n** (application du théorème de Bezout) :

   Pour $n \in \mathbb{N}^*$ et $x \in \mathbb{N}^*$, premiers entre eux, $x \mod n$ est inversible dans $\left(\dfrac{\mathbb{Z}}{n\mathbb{Z}}, \times\right)$ et
   - l'inverse modulo n de x est l'entier k tel que $0 < k < n$ et $k\,x \equiv 1 \mod n$.
   - le quotient modulo n de $y \in \mathbb{N}$ par x est l'entier q tel que $0 \leqslant q < n$ et $y = x\,q \mod n$.

   Avec $x \wedge n = 1$, en prenant u et v dans $\mathbb{Z}$ tels que $ux + vn = 1$, on a $u\,x \equiv 1 \mod n$.
   On pose $a = u \mod n$ et on prend $k = a$ si $a > 0$ et $k = a + n$ sinon, ce qui donne :
   - $k\,x \equiv 1 \mod n$, avec $0 < k < n$
   - avec $q = k\,y \mod n$, on a $x\,q \equiv (x\,k)\,y \equiv y \mod n$, avec $0 \leqslant q < n$

   Si x et n ne sont pas premiers entre eux, c'est une erreur : `failwith "non inversible"`.

3.3 Chiffrement et déchiffrement RSA (méthodes publiques)

Le chiffrement RSA, de clé publique (n,e), de clé secrète (n,d), peut utiliser un facteur de compactage k compris entre 1 et $\lfloor \log_{256} n \rfloor$ (en principe $k = \lfloor \log_{256} n \rfloor$ est un meilleur choix).

Ici, on effectue le chiffrement sur un message représenté par une chaîne de caractères (string), avant d'envoyer la totalité du message chiffré sous forme de liste de chaînes-nombres-entiers (`string_of_int // string_of_big_int`),[a] que l'on déchiffre en totalité au moment de la réception, afin de récupérer le message initial :

chiffrer_RSA n e k mess chiffre la chaîne de caractères mess (`string`), selon la clé (n,e), avec un facteur de compactage k, en une liste de chaînes de caractères (`string_of_int // string_of_big_int`).

dechiffrer_RSA n d lc déchiffre la liste de chaînes de caractères (`string_of_int // string_of_big_int`) lc, selon la clé (n,d), sans connaître le facteur de compactage, en une chaîne de caractères (`string`).

a. Le contenu du message chiffré doit être indépendant du langage de programmation utilisé pour le chiffrement et pour le déchiffrement : on doit pouvoir chiffrer en Caml ou Python, déchiffrer en Python ou Caml. Voir détails sur les types String, Bytes de Ocaml et les classes str, bytes et bytearray de Python ci-dessous

3.4 Conception d'un chiffre RSA. Craquage d'un chiffre RSA

On utilise des test de primalité et tous les algorithmes connus, pour créer un chiffre et pour vérifier l'invulnérabilité (temporaire . . .) de ce chiffre (module de chiffrement résistant à tous les algorithmes connus de factorisation, . . .) sur des ordinateurs les plus performants possibles. Ici, pour simplifier la recherche de grands nombres premiers, on utilise des générateurs de nombres pseudo-aléatoires et pour les tests, on se limite à quelques algorithmes simples classiques.

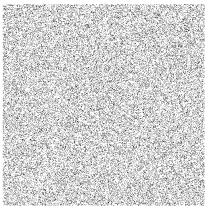
Warning (information documentation Python) : The pseudo-random generators should not be used for security purposes. For security or cryptographic uses, see the secrets module (Python $\geqslant$ 3.6).

3.4.1 Génération de nombres pseudo-aléatoires

1. **Module Random de OCaml** :

```
Random.int : int -> int = <fun>                (* 0 < arg < 2^30 *)
```

Random.int bound returns a random integer between 0 (inclusive) and bound (exclusive). bound must be greater than 0 and **less than** $2^{30} = 1073741824 =$ `2 lsl 29`.



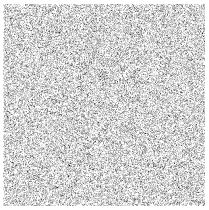
Analyse visuelle : représentation dans un plan où deux nombres pseudo-aléatoires consécutifs fournissent l'abscisse et l'ordonnée d'un point (ramené à un cadre 500×500).
Ici, avec $n = 2^{28} = 536870912$, 50000 fois `x = Random.int n , y = Random.int n`.

Remarque. Cette analyse visuelle n'est pas suffisante pour juger de la qualité du générateur.

2. **Transformation, pour l'intervalle** $]0, max\_int]$:

```
random_int : int -> int = <fun>                (* 0 < arg < 2^62 *)
```

random_int bound returns a random integer between 0 (inclusive) and bound (exclusive). bound must be greater than 0 and **less than $2^{62} - 1$ = max_int = 4611686018427387903** (bidouille utilisant Randon.int sur un découpage de n en tranches).



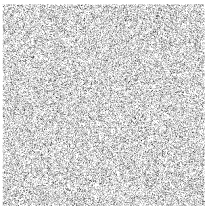
Analyse visuelle : représentation dans un plan où deux nombres pseudo-aléatoires consécutifs fournissent l'abscisse et l'ordonnée d'un point (ramené à un cadre 500×500).
Ici, avec `n = 50000000000000000`, 50000 fois `x = random_int n , y = random_int n`.

Remarque. Cette analyse visuelle n'est pas suffisante pour juger de la qualité du générateur.

3. **Générateur pseudo-aléatoire de big_int** :

```
random_big_int : big_int -> big_int = <fun>
```

random_big_int n renvoie un big_int choisi de façon pseudo-aléatoire entre 0 (inclus) et n (exclus), n devant être plus grand que 0.
(bidouille utilisant randon_int sur un découpage de n en tranches).



Analyse visuelle : représentation dans un plan où deux nombres pseudo-aléatoires consécutifs fournissent l'abscisse et l'ordonnée d'un point (ramené à un cadre 500×500).
Ici, avec `n = big_int_of_string "100000000000000000000000000000000000000"`
50000 fois `x = random_big_int n , y = random_big_int n`.

Remarque. Cette analyse visuelle n'est pas suffisante pour juger de la qualité du générateur.

3.4.2 Tests de primalité

1. **Test de primalité probabiliste de Miller-Rabin (1976-1980).**

   Soit $n \in \mathbb{N}$, impair, et d un entier impair tel que $n-1 = 2^s d$.
   Soit a un entier tel que $1 < a < n$.

   - Si $a^d \equiv 1 \mod n$
     alors n est *fortement probablement premier en base a* ;
   - sinon, s'il existe $r \in [\![0 \ldots s-1]\!]$ tel que $a^{2^r d} \equiv -1 \equiv n-1 \mod n$,
     alors n est *fortement probablement premier en base a* ;
   - sinon n est composé (non premier, factorisable) et a est un *témoin de Miller* de n.

   - D'après le théorème des nombres premiers, la probabilité pour qu'un nombre composé soit fortement probablement premier en base a est inférieure (largement) à $\frac{1}{4}$.
   - Si n est composé, la proportion des k-uples $(a_1, \ldots, a_k) \in \{2, \ldots, n-1\}^k$, tirés au hasard, tels que n soit fortement probablement premier pour chaque a_i est plus petite que $\frac{1}{4^k}$.

   Si pour $(a_1, \ldots, a_k) \in \{2, \ldots, n-1\}^k$, tirés au hasard, n est fortement probablement premier pour chaque a_i, la probabilité pour que n soit premier est d'autant plus forte que k est grand.

   (a) Wikipédia :
   Le théorème de Rabin permet de montrer que la probabilité qu'un nombre composé impair de p bits (compris pris entre 2^{p-1} et $m = 2^p$), soit déclaré probablement premier par l'algorithme de Miller-Rabin pour k nombres a tirés aléatoirement est inférieure à 4^{-k}. $\setminus \ldots \setminus$
   La probabilité pour un nombre impair quelconque de p bits d'être premier est de l'ordre de $2/\ln(m)$ quand $m = 2^p$ est suffisamment grand (par le théorème des nombres premiers). Ceci donne, pour k suffisamment grand pour que 4^{-k} soit négligeable devant $2/\log(m)$, une probabilité qu'un nombre (impair) de p bits soit composé, sachant que l'algorithme le déclare premier, de l'ordre de $\log(m)/(2 \times 4^k)$. Cette probabilité, celle que l'algorithme déclare faussement un nombre premier, devient rapidement très faible quand k augmente.
   En donnant $\varepsilon \ll 1$, on en déduit le nombre minimal k de tests à faire par la formule : $k = \dfrac{\log(\log(m)/(2\varepsilon))}{\log(4)}$.
   Pour $m = 2^{2048}$, $\varepsilon = 10^{-10}$, on trouve $k = 21$.
   Le temps de calcul du test de Miller-Rabin est en $O(k(\log n)^3)$

   (b) Autre source (référence perdue, oubliée ?) :
   Lorsque l'on "tire" k fois un nombre a, de façon aléatoire, $1 < a < n$, la probabilité pour que "n soit factorisable", sachant que l'algorithme a répondu k fois "n est (peut-être) premier", est majorée par $\dfrac{\log n - 2}{\log n - 2 + 4^{k+1}}$.
   En donnant $\varepsilon \ll 1$, on en déduit le nombre minimal k de tests à faire par la formule :
   $k = \dfrac{\log((\log(n)-2)(1-\varepsilon)/\varepsilon)}{\log(4)} - 1$. Pour $n = 2^{2048}$, $\varepsilon = 10^{-10}$, on trouve $k = 21$.

   (c) http://cas.ensmp.fr/~rouchon/MinesCrypto/SlidePrime.pdf
   Soit un grand entier N, et k assez grand pour que $4^k \gg \log(N)$: on tire au hasard $n \leqslant N$ pour lequel on fait k tests de Miller-Rabin.

   - La probabilité pour que n, choisi au hasard $\leqslant N$ et passant le test k fois, soit effectivement premier vérifie : $\text{Prob}(n \text{ premier} \mid \text{test passé } k \text{ fois}) \geqslant 1 - \log(N)/4^k$
   - En donnant $\varepsilon \ll 1$ et un ordre de grandeur pour n, N, on en déduit le nombre minimal k de tests à faire par la formule : $k = \dfrac{\log(\log(N)/\varepsilon)}{\log(4)}$. Pour $N = 2^{2048}$, $\varepsilon = 10^{-10}$, on trouve $k = 22$.

2. Autres tests ...
   - ...
   - ...
   - ...

3.4.3 Algorithmes de factorisation

1. **Méthode de factorisation de Fermat (1640).**

   Soit N impair, se décomposant sous la forme $N = xy$ avec $1 < x \leqslant y$, (x et y inconnus, impairs).

   $$\text{Si on pose } \begin{cases} u = \dfrac{x+y}{2} \\ v = \dfrac{y-x}{2} \end{cases} \text{, on a } \begin{cases} x = u - v \\ y = u + v \end{cases} \quad \text{et } N = u^2 - v^2.$$

   On va donc chercher u et v entiers tels que $N = u^2 - v^2$ et on retournera le couple $(u-v, u+v)$:

   Algorithme :

   - Initialisation : $u \leftarrow \lfloor \sqrt{N} \rfloor$, $v \leftarrow 0$, $r \leftarrow u^2 - N$ (c.a.d. $r \leftarrow u^2 - v^2 - N$)
   - Tant que $r <> 0$ faire
     - si $r < 0$ alors $u \leftarrow u + 1$, sinon $v \leftarrow v + 1$
     - $r \leftarrow u^2 - v^2 - N$ (calcul un peu lourd)
   - Retourner $(u - v, u + v)$

   Algorithme amélioré : en utilisant $u' = 2u + 1$ et $v' = 2v + 1$,

   - Initialisation : $u' \leftarrow 2\lfloor \sqrt{N} \rfloor + 1$, $v' \leftarrow 1$, $r \leftarrow \lfloor \sqrt{N} \rfloor^2 - N$
   - Tant que $r <> 0$ faire
     - si $r < 0$ alors $r \leftarrow r + u'$ puis $u' \leftarrow u' + 2$
     - sinon $r \leftarrow r - v'$ puis $v' \leftarrow v' + 2$
   - Retourner $\left(\dfrac{u' - v'}{2}, \dfrac{u' + v' - 2}{2} \right)$

   En pratique, l'algorithme de Fermat n'est utilisable que pour des nombres pas trop grands, de la forme $N = xy$ avec x et y de même ordre de grandeur (c'est à dire proches de $\sqrt{N}$).

2. **Méthode "p-1" de Pollard – 1974**

   Soit un entier n admettant un facteur premier p tel que $p - 1$ ne possède que de petits facteurs premiers.
   Soit B une borne telle que pour tout facteur premier u de $p - 1$ on ait $u \leqslant B$.
   Soit g un nombre compris entre 2 et $p - 2$, (par exemple $g = 2$ ou $g = 3$).

   Algorithme :

   - $a \leftarrow g$
   - Pour $k = 2$ jusqu'à B, $a \leftarrow a^k \mod n$ (exponentielle rapide modulo n)
   - Avec $d = pgcd(a - 1, n)$, si $1 < d < n$ alors (d est un facteur de n) sinon (échec ...)

3. **Méthode "rho" de Pollard – 1975**

   Soit $n = pq$ un entier, de facteur p inconnu.
   Soit g une fonction pseudo-aléatoire simple à calculer (on prend généralement g telle que $g(x) = (x^2 + 1) \mod n$).
   Soit $a \geqslant 1$ un entier (on prend généralement $a = 1$).
   Soient les suites (x_k) et $(y_k$, telles que $\begin{cases} x_0 = a & ; \quad x_{k+1} = g(x_k) \\ y_0 = g(x_0) & ; \quad y_{k+1} = g(g(y_k)) \quad (= x_{2k+2}) \end{cases}$

   - On calcule $pgcd(|x_k - y_k|, n)$, pour $k \geqslant 0$, tant que ce pgcd est égal à 1.
   - Si $pgcd(|x_k - y_k|, n) = n$ alors c'est un échec, sinon $pgcd(|x_k - y_k|, n)$ est un facteur de n.

   Remarque. En cas d'échec, on peut recommencer avec une autre fonction g, une autre valeur de a ....

4. **Autres ...**

   - courbes elliptiques
   - algorithme du crible quadratique
   - ...

   Remarque. Le crible d'eratosthène n'est pas une méthode performante ... .

Voir aussi le site de factorisation en ligne : https://www.dcode.fr/decomposition-nombres-premiers

3.4.4 Conception d'un chiffre

On cherche deux très grands nombres premiers p et q, tels que $n = pq$ résiste aux algorithmes de factorisation connus :

- p et q très grands (résistance à l'algorithme "rho" de Pollard),
- p et q très éloignés l'un de l'autre (résistance à l'algorithme de Fermat) ,
- $p-1$ et $q-1$ n'ont pas que des petits facteurs premiers (résistance à l'algorithme "p-1" de Pollard),,
- les voisins de p et de q n'ont pas que des petits facteurs premiers,
- ...

A titre d'exemple (**ce n'est pas une méthode souhaitable**), pour rechercher un grand nombre premier, on pourra utiliser `random_big_int` pour tirer au hasard un big_int entre a et b et "boucler" tant que ce nombre n'est pas premier (test de `miller_rabin`) .... Il n'y aura pas forcément une réponse ni forcément une réponse en un temps fini ....

`get_prime_big_int : big_int -> big_int -> big_int = <fun>`

get_prime_big_int a b, où a < b, renvoie un nombre (présumé) premier (tests de `miller_rabin`), compris entre a et b,

Si, en plus, on veut un nombre premier p tel que $p-1$ n'ait pas que des petits facteurs premiers (et d'autres propriétés ...), c'est plus délicat ....

Remarque. On pourra écrire au préalable la version réduite aux int, avec `Random.int` :
`get_prime : int -> int -> int = <fun>` Pour $0 < a < b < 2^{30}$,

get_prime_big_int a b, où a < b, renvoie un nombre (présumé) premier (tests de `miller_rabin`), compris entre a et b.

3.4.5 Attaques d'un chiffre RSA

1. **Attaque par "force brute"** : factorisation du module de chiffrement.

   Si un chiffre a été bien conçu, la seule manière de le casser par **"force brute"** est d'inventer un nouvel algorithme de factorisation, plus efficace que tous ceux connus jusqu'à présent ... et d'avoir de la chance ... et des ordinateurs plus performants que ceux du concepteur du chiffre ... et du temps ....

   Extrait Wikipédia (date ?) :

   "Le 12 décembre 2009, le plus grand nombre factorisé par ce moyen, en utilisant une méthode de calculs distribués, était long de 768 bits. Les clés RSA sont habituellement de longueur comprise entre 1 024 et 2 048 bits. Quelques experts croient possible que des clés de 1 024 bits seront cassées dans un proche avenir (bien que ce soit controversé), mais peu voient un moyen de casser de cette manière des clés de 4 096 bits dans un avenir prévisible. On peut néanmoins présumer que RSA reste sûr si la taille de la clé est suffisamment grande. On peut trouver la factorisation d'une clé de taille inférieure à 256 bits en quelques minutes sur un ordinateur individuel, en utilisant des logiciels librement disponibles. Pour une taille allant jusqu'à 512 bits, et depuis 1999, il faut faire travailler conjointement plusieurs centaines d'ordinateurs. Par sûreté, il est couramment recommandé que la taille des clés RSA soit au moins de 2 048 bits. "

2. **Attaque par itération** : déchiffrement avec une clef publique "faible", exemples.

   D'après https://agreg.org/Textes/public2018-C2.pdf
   Un exemple particulier de clés faibles sont les e tels que l'ordre δ de e dans $(\mathbb{Z}/\varphi(n)\mathbb{Z})^*$ soit petit.
   On a $e^\delta \equiv 1 \mod \varphi(n)$ et il suffit à **Oscar** observant $c = \mu^e \mod n$, d'itérer $\delta - 1$ fois l'opération de chiffrement RSA sur c pour retrouver μ.
   Oscar pourra prendre connaissance des messages destinés à **Alice**, mais ne pourra pas se faire passer pour elle.
   La probabilité pour qu'un élément tiré au hasard dans $(\mathbb{Z}/\varphi(n)\mathbb{Z})^*$ soit d'ordre $\leqslant \delta$ est majorée par $\dfrac{2\delta^3}{\varphi(\varphi(n))}$.

   - Avec $p = 34511$; $q = 13441$; $n = p \times q = 463862351$; $\varphi(n) = (p-1)(q-1)$
     $p-1 = 2 \times 5 \times 7 \times 17 \times 29$; $q-1 = 2^7 \times 3 \times 5 \times 7$;
     $\varphi((p-1)(q-1)) = (p-1)(q-1)\dfrac{(2-1)(3-1)(5-1)(7-1)(17-1)(29-1)}{2 \times 3 \times 5 \times 7 \times 17 \times 29} = 96337920.$
     Pour $\delta = 100$, $\dfrac{2\delta^3}{\varphi(\varphi(n))} = 0.020760257227891158.$
     et on trouve facilement (e, δ) tels que $e^\delta \equiv 1 \mod \varphi(n)$, par exemple $(e = 73793, \delta = 4)$; $(e = 25793, \delta = 12)$.
   - Avec $p = 255949039601$; $q = 2590689529$; $n = p \times q = 663084496851917037929$; $\varphi(n) = (p-1)(q-1)$
     $p-1 = 2^4 \times 5^2 \times 5987 \times 106877$; $q-1 = 2^3 \times 3^3 \times 7 \times 59 \times 113 \times 257$;
     $\varphi((p-1)(q-1)) = (p-1)(q-1)\dfrac{(2-1)(3-1)(5-1)(7-1)(59-1)...(106877-1)}{2 \times 3 \times 5 \times 7 \times 59 \times \cdots \times 106877} = 7728929989252303520$
     Pour $\delta = 100$, $\dfrac{2\delta^3}{\varphi(\varphi(n))} < 2.6e-13$ et je n'ai pas réussi à trouver d'exemple tels que $e^\delta \equiv 1 \mod \varphi(n)$.

3. **Autres attaques ...**

4 Réalisation du TP

Le fichier TP-RSA-S.ml, squelette de programme à compléter (voir listing ci-dessous) contient des éléments OCaml pré-écrits nécessaires au déroulement du TP ainsi que des déclarations de fonctions incomplètes dont il faut écrire le code. L'indication "à écrire en x lignes" sous-entend : "avec au plus une instruction par ligne" (à l'exception d'instructions élémentaires constituant un groupe simple cohérent). Sauf indication contraire, les fonctions ne doivent pas utiliser de variables référence.

Remarque. Les versions `int` ne servent qu'à écrire le plus simplement possible les algorithmes, qui, une fois au point, seront transformées en versions `big_int`. Par exemple :

```
let p = 13999 and q = 14243;;
let n = p q;;                          devient
let p1q1 = (p-1) * (q-1);;
```

```
let p = big_int_of_int 13999 and q = big_int_of_string "14243";;
let n = mult_big_int p q;;
let p1q1 = mult_big_int (pred_big_int p) (pred_big_int q);;
```

Dans le cas des `int`, positifs, limités l'intervalle $[0; max\_int = 2^{62}-1]$, il faut les produits $x \times y$, $x \times y \mod n$ restent dans cet intervalle. Le calcul de $x^k \mod n$ n'est valide que si n et x sont dans l'intervalle $]0, 2^{30}[$ et la valeur n des clés (n,e) et (n,d) est limitée à 2^{30}.

On pourra écrire et tester d'abord les versions "`int`" .

4.1 Définition de ressources

4.1.1 Ressources 1 :

`print_big_int : big_int -> unit`
Print an big integer, in decimal, on standard output.

4.1.2 Ressources 2 : Fonctions de calcul à écrire

1. **Version int (positif) :**

`exprap_mod : int -> int -> int -> int = <fun>` (en 4 lignes)
Pour n et a inférieurs à 2^{30}, `exprap_mod n a p` renvoie $a^p \mod n$

`gcd : int -> int -> int`
`gcd p q` calcule le pgcd de p et q.

`bezout : int -> int -> int vect = <fun>` (en 3 lignes)
Pour x et y inférieurs à 2^{30}, `bezout x y` renvoie un vecteur $[|u,v,w|]$, tel que $u\,x + v\,y = w$ où w est le pgcd de x et y.

`inverse_mod : int -> int -> int = <fun>` (en 3 lignes)
Pour n et x inférieurs à 2^{30}, `inverse_mod n x` renvoie l'inverse de x modulo n, dans l'intervalle $1 \ldots n$,
et provoque l'erreur "non inversible" lorsque cet inverse n'existe pas.

2. **Version big_int (positif) :**

`exprap_mod_big_int : big_int -> big_int -> big_int -> big_int = <fun>` (en 4 lignes)
`exprap_mod_big_int n a p` renvoie $a^p \mod n$

`gcd : big_int -> big_int -> big_int`
`gcd p q` calcule le pgcd de p et q.

`bezout_big_int : big_int -> big_int -> big_int vect = <fun>` (en 3 lignes)
`bezout_big_int x y` renvoie un vecteur $[|u,v,w|]$, tel que $u\,x + v\,y = w$ où w est le pgcd de x et y

`inverse_mod_big_int : big_int -> big_int -> big_int = <fun>` (en 3 lignes)
`inverse_mod_big_int n x` renvoie l'inverse de x modulo n, dans l'intervalle $1 \ldots n$,
et provoque l'erreur "non inversible" lorsque cet inverse n'existe pas.

4.1.3 Ressources 3 : Génération de grands nombres, test de nombres premiers, factorisation

1. **Version int** (avec limites à l'intervalle $]0, 2^{30}[$) :

   `miller_rabin: int -> int -> bool = <fun>`
   Pour 0 < n < 2^{30}, miller_rabin n e renvoie la valeur false si n est sûrement factorisable,
   et la valeur true si n n'a pas été décelé comme factorisable au bout de e essais.

   `isprime : int -> bool = <fun>`
   Pour 0 < n < 2^{30}, isprime n renvoi true si n est (peut-être) premier et false sinon, d'après le test de `miller_rabin`,
   exécuté 30 fois (le résultat devrait tenir compte de réponses concordantes de plusieurs algorithmes).

   A ECRIRE :

   `fermat : int -> float -> int * int = <fun>`
   `fermat n d` renvoie (x,y) tel que n = x y et l'exception "Time out" si le délai d (sec.) est dépassé.

   `pollard_pmu : int -> int -> int -> int = <fun> (* Pollard p-1 *)`
   Pour 0 < n < 2^{30}, pollard_pmu n b g, où n admet un facteur premier p, tel que p-1 ne possède que des petits facteurs premiers, b est une borne des facteurs premiers de $p-1$ et g est un entier compris entre 2 et p-1, renvoie un facteur de p (qui peut être 1, en cas d'échec).

   `pollard_rho : int -> float -> int = <fun> (* Pollard rho *)`
   Pour 0 < n < 2^{30}, pollard_rho n d renvoie un facteur de n (qui peut être 1, en cas d'échec)
   et l'exception "Time out" si le délai d (sec.) est dépassé.

2. **Version big_int :**

   `miller_rabin_big_int : big_int -> int -> bool = <fun>`
   miller_rabin_big_int n e renvoie la valeur false si n est sûrement factorisable
   et la valeur true si n n'a pas été décelé comme factorisable au bout de e essais.

   `isprime_big_int : big_int -> bool = <fun>`
   isprime_big_int n renvoi true si n est (peut-être) premier et false sinon, d'après le test de `miller_rabin`
   exécuté 30 fois (le résultat devrait tenir compte de réponses concordantes de plusieurs algorithmes).

   A ECRIRE :

   `fermat_big_int : big_int -> float -> big_int * big_int = <fun>`
   `fermat_big_int n d` renvoie (x,y) tel que n = x y et l'exception "Time out" si le délai d (sec.) est dépassé.

   `pollard_pmu_big_int : big_int -> int -> int -> big_int = <fun> (* Pollard p-1 *)`
   pollard_pmu_big_int p b g , où n admet un facteur premier p, tel que p-1 ne possède que des petits facteurs premiers, b est une borne des facteurs premiers de $p-1$ et g est un entier compris entre 2 et p-1, renvoie un facteur de p (qui peut être 1, en cas d'échec).

   `pollard_rho_big_int : big_int -> float -> big_int = <fun> (* Pollard rho *)`
   pollard_rho_big_int n d renvoie un facteur de n (qui peut être unit_big_int, en cas d'échec)
   et l'exception "Time out" si le délai d (sec.) est dépassé.

4.2 Fonctions de chiffrement et déchiffrement RSA

Le contenu du message chiffré doit être indépendant du langage de programmation utilisé pour le chiffrement et pour le déchiffrement : on doit pouvoir chiffrer en Caml ou Python, déchiffrer en Python ou Caml. Pour cela, on utilise la représentation unicode (code UTF-8) des caractères composant le message à chiffrer ou à déchiffrer.

string -> bytes (utf-8) -> chiffrer -> int list -> **string (0..9) list** -> int list -> déchiffrer -> bytes (utf-8) -> **string**

- En OCaml, le type String est une séquence d'octets (bytes), codes UTF-8 sur 1, 2, 3 ou 4 octets, quasi-identique au type Bytes.
- En Python, la classe str n'étant pas une séquence d'octets, on utilise comme intermédiaire un objet de la classe bytes ou de la classe bytearray, séquence d'octets (bytes), codes UTF-8 sur 1, 2, 3 ou 4 octets.

Voir détails, exemples pour les types String, Bytes de Ocaml et les classes str, bytes et bytearray de Python, ci-dessous.

1. **Version int :**

`chiffrer_RSA : int -> int -> int -> string -> string list = <fun>`

Pour $0 < n < 2^{30}$, chiffrer_RSA n e k mess chiffre la chaîne de caractères mess (`string`), selon la clé (n,e), avec un facteur de compactage k, en une liste de chaînes de caractères (`string_of_int`).

`dechiffrer_RSA : int -> int -> string list -> string = <fun>`

Pour $0 < n < 2^{30}$, dechiffrer_RSA n d ls déchiffre la liste de chaînes de caractères (`string_of_int`) ls, selon la clé (n,d), sans connaître le facteur de compactage, en une chaîne de caractères (`string`).

- Pour `chiffrer_RSA`, on avance dans la séquence à chiffrer, avec un buffer numérique qui, lorsqu'il est plein (facteur de compactage atteint ou fin de lecture), est chiffré puis ajouté en tête de la liste à émettre,

```
let chiffrer_RSA n e k mess =          (* 0 < n < 2^30 *)
  let rec chiffrer i m j =             (* i = 0 .. k,
    if j = String.length mess
    then if m = 0 then [] else [exprap_mod n m e]
    else let m1 = (int_of_char mess.[j]) + (256 * m) in
         if i = k then ... else ...
  in
  List.map string_of_int (chiffrer 1 0 0);;
(*  val chiffrer_RSA : int -> int -> int -> string -> string list = <fun> *)
```

- Pour `dechiffrer_RSA`, il faut marquer le pas dans la lecture de la liste numérique reçue, pendant que l'on déchiffre puis décompacte la valeur numérique lue en octets destinés à aller un à un en tête du résultat,

```
let dechiffrer_RSA n d ls =    (* 0 < n < 2^30 *)
   let rec degroupe accu = function
      | y when y = 0 -> accu
      | y -> let q = y / 256 and r = y mod 256 in
             degroupe ((String.make 1 (char_of_int r)) ^ accu) q
   and     dechiffrer = function
      | []   -> ""
      | m::q -> ...
   in
   dechiffrer (List.map int_of_string ls);;
(* val dechiffrer_RSA : int -> int -> string list -> string = <fun> *)
```

2. **Version big_int :** Adaptation des fonctions écrites dans le cas int.

`chiffrer_RSA_big_int : big_int -> big_int -> int -> string -> string list = <fun>`

chiffrer_RSA_big_int n e k mess chiffre la chaîne de caractères mess, selon la clé (n,e), avec un facteur de compactage k, en une liste de chaînes de caractères (`string_of_big_int`).

`dechiffrer_RSA_big_int : big_int -> big_int -> string list -> string = <fun>`

dechiffrer_RSA_big_int n d ls déchiffre la liste de chaînes de caractères (`string_of_big_int`) ls, selon la clé (n,d), sans connaître le facteur de compactage, en une chaînes de caractères.

4.3 Tests élémentaires (exemples de chiffrement et déchiffrement)

4.3.1 Chiffre à module de chiffrement < 2^{30}

Vous êtes **Alice**, et votre chiffre a comme clés : $\begin{cases} \text{publique} & (n,e) = (199387757, 16481) \\ \text{secrète} & (n,d) = (199387757, 194170193) \end{cases}$

Ici, $n < 2^{30}$,

1. on utilisera d'abord la version int : fonctions `chiffrer_RSA` et `dechiffrer_RSA`
2. puis la version big_int : fonctions `chiffrer_RSA_big_int` et `dechiffrer_RSA_big_int`

1. **Décodage :**

   Que signifie le message reçu suivant, chiffré avec votre clé publique (n,e) :

   ```
   [ "118662545";  "99068013"; "132200431";  "89291238";  "68393044";  "22108232"; "185504288";
     "113582398"; "159540846"; "185844420"; "153659394";  "90374658";  "67338774"; "114267357";
      "58719682"; "126317319";  "14898677"; "177687227"; "161993329"; "184753009"; "193210174";
      "67752315";   "2790210"; "187934739"; "179014827";  "56565905"; "117787729" ]
   ```

2. **Codage :**

   Votre associé, qui est parti en vacances en oubliant (?) de vous donner le nouveau code d'accès à la salle des coffres, chiffre le message "Le code secret d'accès à la salle des coffres est ZpX01Wzz2", à l'aide de votre clé publique.

   - Quelle est la teneur du message chiffré (avec un facteur de compactage k = 3) que vous recevez ?
   - Vérifiez que vous pouvez le déchiffrer.

3. **Vérification de la résistance du chiffre à une attaque :**

   Pour vérifier la résistance du chiffre, vous essayez de factoriser n par la méthode de Fermat.
   Combien de temps faudra-t'il pour "casser" n

   (a) 10 secondes ?

   (b) plus ? beaucoup plus ?

   (c) moins ? beaucoup moins ?

   Vérifier ensuite que vous pouvez en déduire immédiatement la clé secrète (n,d), avec $d = e^{-1} \mod (p-1)(q-1)$.

4.3.2 Chiffre à module de chiffrement compris entre 2^{30} et max_int

Vous êtes **Alice**, et vous voulez prendre comme chiffre (n,e), (n,d), avec :

$$\begin{cases} p = 276464707,\ q = 114626177 & \text{premiers} \\ n = p \times q = 31690092438835139 & \text{tel que } 2^{30} < n < \text{max\_int, avec } \log_{256}(n) \approx 6.83 \\ e = 152423 & \\ d \text{ inverse de } e \text{ modulo } n & \end{cases}$$

1. Vérifier que cela ne marche pas dans le domaine des int.
2. Vérifier que cela marche dans le domaine des big_int.

4.3.3 Facteur de compactage trop grand

Vous êtes **Alice**, et vous avez gardé le chiffre précédent, dans le domaine des big_int.
Vérifiez que si l'on vous envoie un message chiffré avec votre clé publique et un facteur de compactage trop grand (par exemple $k = 8$), vous ne pouvez pas le déchiffrer.

4.4 Attaques du chiffre

4.4.1 Déchiffrement par itération sur une clef publique faible

Alice a choisi un chiffre dont la partie publique est $(n,e) = (463862351, 25793)$

Vous êtes le vilain **Oscar** et vous interceptez le message chiffré, de **Bob** à **Alice**, suivant :

```
[ "122295419"; "383353354"; "411357142"; "115761993"; "200776304"; "253827617"; "387846596";  "86485943";
  "304292861";  "69907154"; "162698440"; "431972996"; "378369222"; "368814328"; "311566305"; "112084975";
  "282193578"; "364978913";  "33681360"; "185026348"; "181764853" ]
```

- Déterminer δ (petit) tel que $e^\delta \equiv 1 \mod \varphi(n)$.
  Indication : pour $\delta = 1, 2, ...$, calculer $100^{e^\delta} \mod n$, soit $\underbrace{(..((100^e \mod n)^e \mod n)...)^e \mod n}_{\delta \text{ fois}}$, jusqu'à obtenir 100.
- Avec une altération de la fonction `dechiffrerRSA`, déchiffrez le message envoyé par **Bob** à **Alice**.

4.4.2 Attaque par force brute (Fermat) sur un chiffre simple

Alice a choisi un chiffre dont la partie publique est $(n,e) = (65509912209240151035623, 27329)$

Vous êtes le vilain **Oscar** et vous interceptez le message chiffré, de **Bob** à **Alice**, suivant :

```
[ "58708423107954916923760"; "3143689942199918046615 5"; "44401475955424185740 3"; "20107848284631489486860";
  "58312015784159115214176"; "13978316466576225911582"; "28937690863344110299079"; "52763225684222112646755";
  "27937192474109965023930"; "22057576210221896807178"; "39521329825987178664787"; "51726656168921412590539" ]
```

- Trouvez la clé secrète de **Alice** (factorisez $n = pq$ par la méthode de Fermat et trouvez $d = e^{-1} \mod (p-1)(q-1)$).
- Déchiffrez le message envoyé par **Bob** à **Alice**.

4.4.3 Attaque par force brute (Pollard) sur un chiffre plus élaboré

Vous êtes toujours le vilain **Oscar** mais il ne fallait pas vous vanter d'avoir craqué la clé secrète d'**Alice** ! **Alice** a changé son chiffre, et sa nouvelle clé publique est $(n,e) = (663084496823009987142848811 19, 206977)$.
Ce chiffre semble résister à la méthode de Fermat, mais **Alice** a peut-être construit $n = p \times q$ avec p et q grands et premiers, simplement éloignés l'un de l'autre, sans plus de précautions, et il est possible que $p-1$ (ou $q-1$) ne possède que des "petits" facteurs premiers.

- Trouvez la clé secrète de **Alice**.
- Déchiffrez le message suivant, envoyé par **Bob** à **Alice** :

```
[ "1146228123974337830308698241 2";  "706248904889505872219814698 5"; "272612860349718267199569968 72";
  "213817576264180784817612887 18"; "566926195250498242683701442 03"; "408866159142534891892778739 39";
    "133148660649654723673568439";   "465321848979391965308630523" ]
```

4.5 Construction d'un chiffre résistant

Vous êtes **Alice** et **Oscar** commençant à sérieusement vous ennuyer ..., vous décidez de construire un chiffre "sérieux" !

- Trouvez p et q premiers très grands, très éloignés l'un de l'autre, réputés premiers et tels que $p-1$ et $q-1$ possèdent au moins un grand facteur premier ... (par exemple $p-1 = 2 \times p'$ et $q-1 = 2 \times q'$, avec p' et q' premiers),

  avec 2^{64} = 18446744073709551616 < p < 2^{96} = 79228162514264337593543950336
  et 2^{128} = 340282366920938463463374607431768211456 < q < 2^{160} = 1461501637330902918203684832716283019655932542976

  (d'où 2^{192} = 6277101735386680763835789423207666416102355444464034512896 < $n = pq < 2^{256} = \cdots$).
- Choisissez votre e.
- Calculez votre nouvelle clé privée (n,d).
- Vérifiez que votre chiffre résiste bien aux attaques précédentes
  - **apprendre au préalable à utiliser le menu "Interrupt OCaml" ...**
  - ou introduire un temps maximum d'exécution dans les fonctions ...

```
let test n xx delai =            (* delai : float, secondes *)
   let time0 = Sys.time () in
   ...
   if Sys.time () -. time0 > delai then failwith "Time out!"
   ...
```

- Publiez votre nouvelle clé publique (n,e).

Vous pouvez maintenant recevoir vos premiers messages ... "confidentiels".

4.6 Oscar vient d'acheter un nouvel ordinateur superpuissant ...

5 String, Bytes, str, bytes, Unicode, UTF-8, ASCCI 128, 256, 512

5.1 OCaml (String, Bytes)

Before the introduction of type bytes in OCaml 4.02, strings were mutable. Now, strings are intended to be immutable, and bytes is the type to be used for "mutable strings". In order not to break too much existing code, this distinction is not yet enabled by default. In default mode (-unsafe-string command-line option), bytes and string are synonymous.
All new code should avoid this feature and be compiled with the -safe-string command-line option to enforce the separation between the types string and bytes.

5.1.1 String

A string is an immutable data structure that contains a fixed-length sequence of (single-byte) characters. Each (single-byte) character can be accessed in constant time through its index.
Chaque caractère usuel peut être représenté par 1, 2, 3 ou 4 bytes (octets) :

- ascii 000-127 (UTF-8, 1 octet), comme expression littérale e, hexadécimale \x65 ou décimale \101 (~~octale~~)
- ascii 128-255 (UTF-8, 2 octets), comme expression littérale é, hexadécimale \xc3\xa9, ou décimale \195\169 (~~octale~~)
- plus généralement, UTF-8 avec 1, 2, 3 ou 4 octets, comme expression hexadécimale ou décimal (~~octale~~)

Remarque. Pour un code UTF-8 invalide ou non reconnu, `print_string` donne une réponse avec une forme octale :

```
print_string "ab\193\129";; (* ab\301\201 *) (* octal: code invalide ou inconnu *)
```

```
(*======= code ascii < 128  (litéral, hexa, décimal) ====*)
let s = "abcdef+\x61\x62\x63\x64\x65\x66+\097\098\099\100\101\102";;   (* string = "abcdef+abcdef+abcdef" *)
printf "%3d - %10s" (String.length s) s;;                             (* 20 - abcdef+abcdef+abcdef *)
s.[0] <- 'x';; (* Warning 3: deprecated: String.set Use Bytes.set instead. *)
s.[0];;        (* char = 'x' *)
(*======= code ascii [128,255] --> UTF-8 (2 octets) =========*)
let s = "éè+\xc3\xa9\xc3\xa8+\195\169\195\168";;
      (* string = "\195\169\195\168+\195\169\195\168+\195\169\195\168" *)
printf "%3d - %10s" (String.length s) s;;  (* 14 - éè+éè+éè *)
(*============= caractères spéciaux ============*)
let s = "\"";;                              (* s : string = "\"" *)
printf "%3d - %10s" (String.length s) s;;   (* 1 - " *)
(*======== codes UTF-8 avec 1 ou 2 ou 3 ou 4 octets ==================*)
let w = "z\x7A\122";;                       (* string = "zzz" *)
printf "%3d - %10s" (String.length w) w;;   (* 3*1 = 3 - zzz *)
let w = "\xC3\x86\195\134";;                (* string = "\195\134\195\134" *)
printf "%3d - %10s" (String.length w) w;;   (* 2*2 = 4 - ÆÆ *)
let w = "\xe2\x82\xac\226\130\172";;        (* string = "\226\130\172\226\130\172" *)
printf "%3d - %10s" (String.length w) w;;   (* 3*2 = 6 - €€ *)    (* non directement représenté en pdf *)
let w = "\xF0\x9D\x84\x9E\240\157\132\158";; (* string clé de sol UTF-8 F0 9D 84 9E *)
printf "%3d - %10s" (String.length w) w;;   (* 4*2 = 8 - 𝄞 *)     (* non directement représenté en pdf *)
(*======== code UTF-8 invalide: print --> réponse en base 8 --*)
let s = "\195";;                            (* s : string = "\195" *)
printf "%3d - %10s" (String.length s) s;;   (* 1 - \303 *)                  (* octal *)
let s = "\193\129";;                        (* string = "\193\129"*)
printf "%3d - %10s" (String.length s) s;;   (* 2 - \301\201 *)              (* octal *)
let w = "abc\xED\xaF\xFF";;                 (* string = "abc\237\175\255" *)
printf "%3d - %10s" (String.length w) w;;   (* 6 - abc\355\257\377 *)        (* octal *)

printf "%4d%4d%4d%4d%4d%4d" 195 0Xc3 0xC3 0O303 0o303 0b11000011;; (* 195 195 195 195 195 195 *)
```

5.1.2 Bytes (depuis OCaml 4.02)

A byte sequence is a mutable data structure that contains a fixed-length sequence of bytes. Each byte can be indexed in constant time for reading or writing.
Remarque. Il y a peu de différences entre les types String et Bytes de OCaml . . .

```
let s = Bytes.of_string "\195\131\194\169\195\131\195\169xyz";; (*bytes="\195\131\194\169\195\131\195\169xyz"*)
s.[0], s.[9];;           (* - : char * char = ('\195', 'y') *)
printf "%3d - %10s" (String.length s) s;;  (* 11 - ÃÃ©éxyz *)    (* non directement représenté en pdf *)
s.[0] <- 'x';;           (* Warning 3: deprecated: String.set Use Bytes.set instead. *)
Bytes.set s 4 'a';;
s;;                      (* bytes = "x\131\194\169a\131\195\169xyz" *)
printf "%3d - %10s" (String.length s) s;;  (* 11 - x\203©a\203éxyz *)    (* octal *)
```

5.2 Python (str, bytes, bytearray)

5.2.1 str

String (class str) objects are immutable sequences of unicode characters. La classe str contient des caractères

- ascii étendu littéral, par exemple é
- ascii étendu sous forme hexa 2 chiffres, par exemple \xE9 (é)
- ~~ascci sous forme décimale, binaire~~
- ascci sous forme octale c, \cc, \ccc, (1 à 3 chiffres octaux, de \0 à \777 = 511), par exemple \351 (é)
- unicode sous forme hexa 4 chiffres, par exemple \u00E9 (é)
- unicode sous forme hexa 8 chiffres, par exemple \U000000E9 (é)

chacun comptant pour 1 dans la longueur.

```
s = "abcdef+\x61\x62\x63\x64\x65\x66+\141\142\143\144\145\146"+\
    "+\u0061\u0062\u0063\u0064\u0065\u0066+\U00000061\U00000062\U00000063\U00000064\U00000065\U00000066"
print(type(s), len(s), s)                  # str 34 abcdef+abcdef+abcdef+abcdef+abcdef
# s[0] = 65 # TypeError: 'str' object does not support item assignment
s = chr(511) + chr(0o777) + chr(0x1ff) + "\u01ff\U000001ff" +\
    chr(8364) + chr(0o20254) + chr(0x20ac) + "\u20AC\U000020ac"
print(type(s), len(s), s)        # str 10 □□□□□□□□□□ non représenté en pdf, soit ₰₰₰₰€€€€€
#----------------------- Accents
s = "é\xe9\351\u00e9\U000000e9"  # Accent: littéral hexa octal unicode(4) Unicode(8)
print(type(s), len(s), s)        # str 5 ééééé
#----------------------- Non supportés par IDLE
s = "\U0001d120"                 # clé de sol unicode
try: print(type(s), len(s), s)
except: print('ERROR ... IDLE 3: Non-BMP (Basic Multilingual Plane) character not supported in Tk')

print( 195, 0Xc3, 0xC3, 0O303, 0o303, 0b11000011)   # 195 195 195 195 195 195
```

5.2.2 bytes

Bytes (class bytes) objects are immutable objects of single bytes.

```
s = "éèpqr9"
print( s.encode( 'utf-8' ) )        # b'\xc3\xa9\xc3\xa8pqr9'           ( 'utf-8' = default )
s = b'\xc3\xa9\xc3\xa8\x70qr\71'    # hexa, litéral et octal (\71 octal = '9')
print( list(s) )                    # [195,169, 195,168, 112, 113, 114, 57], décimal
# s[0] = 97  # TypeError: 'bytes' object does not support item assignment
print( s[0] )                       # 195 décimal
print(type(s), len(s), s)           # bytes 8 b'\xc3\xa9\xc3\xa8pqr9'
print( s.decode( 'utf-8' ) )        # éèpqr9                           ( 'utf-8' = default )

s = b'\u0070\xc3\xa9\xc3\xaa\x70qr' # \u0070 non reconnu comme unicode, -> \ u 0 0 7 0
print( list(s) )                    # [92, 117, 48, 48, 55, 48, 195,169, 195,170, 112, 113, 114] (décimal)
print(type(s), len(s), s)           # bytes 13 b'\\u0070\xc3\xa9\xc3\xaapqr'
print( s.decode() )                 # \u0070êêpqr

s = "é\u0070\xc3\xa9\xc3\xaa\x70\161r\U0000D11E".encode('utf8')         # \161 = 'q' (octal)
print( list(s) )            # [195,169, 112, 195,131,194,169,195,131,194,170, 112,113,114, 237,132,158] (décimal)
print(type(s), len(s), s)   # bytes 17 b'\xc3\xa9p\xc3\x83\xc2\xa9\xc3\x83\xc2\xaapqr\xed\x84\x9e'
print( s.decode() )         # é   ppqr       soit ép□□□□pqr□ non représenté en pdf

s = bytes([97,98,99,100,101,102,103,104,105,106,107,108,109,  195,169,  195,168])    # UTF-8, entiers 0..255
print(type(s), len(s), s)       # bytes 17 b'abcdefghijklm\xc3\xa9\xc3\xa8'
print( s.decode() )             # abcdefghijklméè
```

5.2.3 bytearray

Bytearray (class bytearray) objects are mutable objects of single bytes.

```
s = bytearray([97,98,99,100,101,102,103,104,105,106,107,108,109,  195,169,  195,168]) # UTF-8, entiers 0..255
print(type(s), len(s), s)           # bytearray 17 bytearray(b'abcdefghijklm\xc3\xa9\xc3\xa8')
print( s.decode())                  # abcdefghijklméè
s[0] = 65
# s[1] = 'B'           # TypeError: an integer is required
print( s.decode())   # Abcdefghijklméè
```

6 OCaml : module big_int (operations on arbitrary-precision integers)

Big integers (type `big_int`) are signed integers of arbitrary size.

`type big_int` — The type of big integers.

`val val zero_big_int : big_int` — The big integer 0.

`val val unit_big_int : big_int` — The big integer 1

Arithmetic operations

`val minus_big_int : big_int -> big_int` — Unary negation.

`val abs_big_int : big_int -> big_int` — Absolute value.

`val add_big_int : big_int -> big_int -> big_int` — Addition.

`val succ_big_int : big_int -> big_int` — Successor (add 1).

`val add_int_big_int : int -> big_int -> big_int` — Addition of a small integer to a big integer.

`val sub_big_int : big_int -> big_int -> big_int` — Subtraction.

`val pred_big_int : big_int -> big_int` — Predecessor (subtract 1).

`val mult_big_int : big_int -> big_int -> big_int` — Multiplication of two big integers.

`val mult_int_big_int : int -> big_int -> big_int` — Multiplication of a big integer by a small integer.

`val square_big_int : big_int -> big_int` — Return the square of the given big integer.

`val sqrt_big_int : big_int -> big_int`

Return the positive integer square root of the given big integer. Raise Invalid_argument if argument is negative.

`val quomod_big_int : big_int -> big_int -> big_int * big_int`

Euclidean division of two big integers.The first part of the result is the quotient, the second part is the remainder. Writing `(q,r) = quomod_big_int a b`, we have a = q * b + r and 0 <= r < abs(b). Raise Division_by_zero if the divisor is zero.

`val div_big_int : big_int -> big_int -> big_int`

Euclidean quotient of two big integers. This is the first result q of `quomod_big_int` (see above).

`val mod_big_int : big_int -> big_int -> big_int`

Euclidean modulus of two big integers. This is the second result r of `quomod_big_int` (see above).

`val gcd_big_int : big_int -> big_int -> big_int` — Greatest common divisor of two big integers.

`val power_int_positive_int : int -> int -> big_int`

`val power_big_int_positive_int : big_int -> int -> big_int`

`val power_int_positive_big_int : int -> big_int -> big_int`

`val power_big_int_positive_big_int : big_int -> big_int -> big_int`

Exponentiation functions. Return the big integer representing the first argument a raised to the power b (the second argument). Depending on the function, a and b can be either small integers or big integers. Raise Invalid_argument if b is negative.

Comparisons and tests

`val sign_big_int : big_int -> int` — Return 0 if the given big integer is zero, 1 if it is positive, and -1 if it is negative.

`val compare_big_int : big_int -> big_int -> int`

`compare_big_int a b` returns 0 if a and b are equal, 1 if a is greater than b, and -1 if a is smaller than b.

`val eq_big_int : big_int -> big_int -> bool`

`val le_big_int : big_int -> big_int -> bool`

`val ge_big_int : big_int -> big_int -> bool`

`val lt_big_int : big_int -> big_int -> bool`

`val gt_big_int : big_int -> big_int -> bool`

Usual boolean comparisons between two big integers.

`val max_big_int : big_int -> big_int -> big_int` — Return the greater of its two arguments.

`val min_big_int : big_int -> big_int -> big_int` — Return the smaller of its two arguments.

`val num_digits_big_int : big_int -> int` — Return the number of machine words used to store the given big integer.

Conversions to and from strings

`val string_of_big_int : big_int -> string`

Return the string representation of the given big integer, in decimal (base 10).

`val big_int_of_string : string -> big_int` — Convert a string to a big integer, in decimal.

The string consists of an optional - or + sign, followed by one or several decimal digits.

Conversions to and from other numerical types

`val big_int_of_int : int -> big_int` — Convert a small integer to a big integer.

`val is_int_big_int : big_int -> bool`

Test whether the given big integer is small enough to be representable as a small integer (type int) without loss of precision.
On a 32-bit platform, returns true if and only if argument is between -2^{30} and $2^{30}-1$.
On a 64-bit platform, returns true if and only if argument is between -2^{62} and $2^{62}-1$.

`val int_of_big_int : big_int -> int`

Convert a big integer to a small integer (type int).
Raises Failure "int_of_big_int" if the big integer is not representable as a small integer.

`val float_of_big_int : big_int -> float` — Returns a floating-point number approximating the given big integer.

$< \mathscr{FJN} >$ (énoncé TP RSA)

7 OCaml : squelette de programme, non entièrement exécutable

```
(* OCaml >= 4.02                *)
(* Squelette TP RSA. A COMPLETER *)
open Printf;;
#load "graphics.cma";;
open Graphics;;
#load "nums.cma"
open Big_int;;
let deux_big_int = big_int_of_int 2;;
#load "unix.cma";;
(*============================================== Utilitaires   *)
let print_big_int (b : Big_int.big_int) : unit = print_string (string_of_big_int b);;
(*====================== Calculs version int *)
let rec exprap_mod n a = function  (* a^e mod n, récursif; 0 < n, a < 2^30 < sqrt(max_int) = 2147483648 *)
  | 0 -> ...
  | e -> ... ... ... ;;
(* val exprap_mod : int -> int -> int -> int = <fun> *)
let rec gcd (x : int) (y : int) : int = ...
;;
let rec bezout x = function                        (* ax+by = gcd x y; 0 < x, y < 2^30 *)
    0 -> ...
  | y -> ... ... ;;
(* val bezout : int -> int -> int array = <fun> *)
let inverse_mod (n : int) (x : int) : int =       (* x^(-1) mod n; 0 < n, x < 2^30 *)
  ....
;;
let quotient_mod (n : int) (x : int) (y : int) : int =    (* x/y mod n; 0 < n, x , y < 2^30 *)
    ....
;;
(*======================== Calculs version big_int *)
let exprap_mod_big_int n a =                          (* a^e mod n; version récursive *)
  ... ... ... ;;
(* val exprap_mod_big_int : big_int -> big_int -> big_int -> big_int = <fun> *)
let rec bezout_big_int x = function
  ...  ...;;
(* val bezout_big_int : big_int -> big_int -> .big_int array = <fun> *)
let inverse_mod_big_int (n : big_int) (x : big_int) : big_int =   (* x^(-1) mod n *)
  ... ... ... ;;
let quotient_mod_big_int (n : big_int) (x : big_int) (y : big_int ) : big_int =    (* x/y mod n *)
  ...
;;
(*========================================================= Nombres pseudo-aléatoires  *)
(*========================== Version int *)
let random_int (n : int) : int =          (* 0 < n <= max_int *)
  let d = 2 lsl 25 in
  let k = ref 0 and q = ref n and r = ref 0 in
  while !q <> 0 do r := !q; q := !q / d; k := !k+1 done;
  let u = ref (Random.int d) in
  for i = 0 to !k
  do u := ( (!u * (Random.int d)) + (Random.int d) ) mod max_int;
     if !u < 0 then u := !u + max_int
  done;
  !u mod n
;;
open_graph " 800x800+10+10"; set_color black;  (* test graphique de random_int *)
let n = 100000 in
for i = 1 to 25000
do let x = random_int n and y = random_int n in
   let x1 =  ((float_of_int x) /. (float_of_int (n))) and y1 =  ((float_of_int y) /. (float_of_int (n))) in
   let x2 = 100 + int_of_float (500. *. x1) and y2 = 100 + int_of_float (500. *. y1) in
   plot x2 y2
done;;
```

```
(*======================== Version big_int *)
let random_big_int n =
  let k = num_digits_big_int n in
  let u = ref (big_int_of_int (random_int max_int)) in
  for i = 0 to k
  do let z = random_int max_int in
     let zz = (random_int max_int) in
     u := add_big_int (mult_int_big_int zz !u) (big_int_of_int z)
  done;
  mod_big_int !u n;;
(* val random_big_int : big_int -> big_int = <fun> *)

open_graph " 800x800+10+10"; set_color black; (* test graphique de random_big_int *)
let n = big_int_of_string "10000000000000000000000000000000000000" in
for i = 1 to 25000
do let x = random_big_int n and y = random_big_int n in
   let x1 = (float_of_big_int x) /. (float_of_big_int n)
   and y1 = (float_of_big_int y) /. (float_of_big_int n) in
   let x2 = 100 + int_of_float (500. *. x1)
   and y2 = 100 + int_of_float (500. *. y1) in
   plot x2 y2
done;;

(*============================================ Test de primalité, factorisation
(*==================== Version int  *)
let miller_rabin (n : int) (k : int) : bool =                (* n < 2 ** 30 *)
  if n < 4 then true
  else
  begin
    let z = n-1 in
    let s = ref 0 and d = ref z in
    while !d mod 2 = 0 do s := !s+1; d := !d / 2 done;

    if !s = 0 then false else
    try
      for _ = 1 to k    (* k essais *)
      do let a = 1 + random_int (z-1) in
         let b = ref (exprap_mod n a !d) in
         try if !b = 1
             then failwith "premier"
             else begin for r = 0 to !s-1
                        do if !b mod n = z then failwith "premier"
                           else b := (!b * !b) mod n
                        done;
                        failwith "factorisable"
                  end;
         with Failure "premier" -> ();      (* passe *)
      done;
      true                                  (* "premier" *)
    with Failure "factorisable" -> false    (* "factorisable" *)
  end
;;
let isprime (n : int) : bool = (miller_rabin n 30)                    (* n < 2 ** 30 *)
;;
let get_prime (a : int) (b : int) : int =
  let c = b - a and x = ref 4 in
  while not (isprime !x) do x := a + (random_int c) done;
  !x
;;
```

```
let eratosthene (n : int) : string =
  let nn = ref n and h = ref 0 and k = ref 0 and ss = ref "" in

  while !nn mod 2 = 0 do h := !h+1; nn := !nn / 2 done;
  if !h > 0 then ss := !ss ^ "2";
  if !h > 1 then ss := !ss ^ "^" ^ (string_of_int !h);

  for i = 1 to (int_of_float (sqrt (float_of_int n))) /2
  do let u = 2*i+1 in
     k := 0; while !nn mod u = 0 do k := !k+1; nn := !nn / u done;
     if !k > 0 then if String.length !ss > 0
                    then ss := !ss ^ "*" ^ (string_of_int u)
                    else ss := !ss ^ (string_of_int u);
     if !k > 1 then ss := !ss ^ "^" ^ (string_of_int !k)
  done;

  if !nn > 1
  then if String.length !ss > 0
       then ss := !ss ^"*" ^ (string_of_int !nn) else ss := !ss ^ (string_of_int !nn);
  !ss
;;
let fermat (n : int) (delay : float) : int * int =                    (* delay (sec.) *)
  let time0 = Sys.time () in
  ...
  while !r <> 0
  do if !r > 0
     then ....
     else ....
     let time1 = Sys.time () in if time1 -. time0 > delay then failwith "Time out !"
  done;
  ...
;;
let pollard_pmu (n : int) (b : int) (g : int) : int =  (* Pollard p-1 *)  (* n < 2 ** 30 *)
  ...
;;
let pollard_rho (n : int) (delay : float) : int =      (* Pollard rho *)  (* n < 2 ** 30 ; delay (sec.) *)
  let time0 = Sys.time () in
  let g = (function x -> (x*x+1) mod n) in
  let x = ref 1 and y = ref 1 and d = ref 1 in
  while !d = 1
  do ... ...
     let time1 = Sys.time () in if time1 -. time0 > delay then failwith "Time out !"
  done;
  if !d = n then 1 else !d
;;
(*============================== Version big_int *)
let miller_rabin_big_int (n : big_int) (e : int) : bool =
  ...
;;
let isprime_big_int (n : big_int) : bool =
   ...
;;
let get_prime_big_int (a : big_int) (b : big_int) : big_int =
   ...
;;
let fermat_big_int (n : big_int) (delay : float) : big_int * big_int =   (* delay (sec.) *)
   ...
;;
let pollard_pmu_big_int (n : big_int) (b : int) (g : int) : big_int =    (* Pollard p-1 *)
   ...
;;
let pollard_rho_big_int (n : big_int) (delay : float) : big_int =        (* Pollard rho *)  (* delay (sec.) *)
   ...
;;
```

```
(*============================================== Chiffrage et déchiffrage *)
(*============================ Version int *)
let chiffrer_RSA n e k mess =          (* 0 < n < 2^30 *)
  let rec chiffrer i m j =
    if j = String.length mess
    then if m = 0 then [] else [exprap_mod n m e]
    else let m1 = (int_of_char mess.[j]) + (256 * m) in
         if i = k then (exprap_mod n m1 e)::(chiffrer 1 0 (j+1)) else chiffrer (i+1) m1 (j+1)
  in List.map string_of_int (chiffrer 1 0 0);;
(*  val chiffrer_RSA : int -> int -> int -> string -> string list = <fun> *)

let dechiffrer_RSA n d ls =    (* 0 < n < 2^30 *)
   let rec degroupe accu = function
      | y when y = 0 -> accu
      | y ->  let q = y / 256 and r = y mod 256 in
              degroupe ((String.make 1 (char_of_int r)) ^ accu) q
   and     dechiffrer = function
      | []   -> ""
      | m::q -> degroupe (dechiffrer q) (exprap_mod n m d)
   in dechiffrer (List.map int_of_string ls);;
(* val dechiffrer_RSA : int -> int -> string list -> string = <fun> *)

(*========================= Version big_big_int *)
let chiffrer_RSA_big_int n e k mess =
  ...
(* val chiffrer_RSA_big_int : big_int -> big_int -> int -> string -> string list = <fun> *)

let dechiffrer_RSA_big_int n d ls =
  ...
(* val dechiffrer_RSA_big_int : big_int -> big_int -> string list -> string *)

(*============================================================ Tests élémentaires *)
(*============================ version big_int *)
(* ------ chiffre ----- *)
let n = big_int_of_string "199387757";;
let e = big_int_of_int 16481;;
let d = big_int_of_int 194170193;;
(* ------ chiffrage ----- *)
(* ------ message chiffré ----- *)
let lc = [ "118662545";  "99068013"; "132200431";  "89291238";  "68393044";  "22108232"; "185504288";
           "113582398"; "159540846"; "185844420"; "153659394";  "90374658";  "67338774"; "114267357";
            "58719682"; "126317319";  "14898677"; "177687227"; "161993329"; "184753009"; "193210174";
            "67752315";   "2790210"; "187934739"; "179014827";  "56565905"; "117787729" ];;
(* ------ craquage ----- *)
let x, y = fermat_big_int n 10.;;
print_big_int x;;
print_big_int y;;
let x1y1 = mult_big_int (pred_big_int x) (pred_big_int y);;
let s = inverse_mod_big_int x1y1 e;;
print_big_int s;;
(* ------ déchiffrage ------ *)
print_string (dechiffrer_RSA_big_int n s lc);;
(*======================  La clef du coffre       *)
(* ----- chiffrage (chiffre identique au précédent) *)
let ms = "Le code secret d'accès à la salle des coffres est ZpX01Wzz2";;
let lc = chiffrer_RSA_big_int n e 3 ms;;  (* k = 3 < kmax *)
(* ------ message chiffré ----- *)
(* ------ déchiffrage ------ *)
print_string (dechiffrer_RSA_big_int n d lc);;
```

```
(*======================================= module de chiffrement > 2^30 *)
(*============== Version int = Probleme ---*)
(* ------ chiffre ----- *)
let p = 276464707 and q = 114626177;;
let n = p * q;;
print_int n;;            (*   31690092438835139 *)
print_int max_int;;      (* 4611686018427387903 *)
print_int (2 lsl 30);;           (* 2147483648 *)

let e = 152423;;
let d = inverse_mod ((p-1) * (q-1)) e;;
print_int e;;                (*          152423 *)
print_int d;;                (* 1553494996035671 *)
print_int (( e * d) mod ((p-1) * (q-1)));;              (* différent de 1 : PROBLEME ! Stop! *)
(*============================== Version Big_int *)
(* ------ chiffre ---- *)
let p = big_int_of_int 276464707 and q = big_int_of_int 114626177;;
let n = mult_big_int p q;;
print_big_int n;;                   (*   31690092438835139  *)
let p1q1 = mult_big_int (pred_big_int p) (pred_big_int q);;
let e = big_int_of_int 152423;;
let d = inverse_mod_big_int p1q1 e;;
print_big_int e;;                   (*          152423 *)
print_big_int d;;                   (* 1553494996035671 *)
print_big_int (mod_big_int (mult_big_int e d) p1q1);;       (* = 1 ok *)
(* ------ chiffrage ----- *)
let ms = "Le sujet du prochain DS d'info comportera trois parties. Bob.";;
printf "k < %12.2f" ((log (float_of_big_int n)) /. log 256.);;
let lc = chiffrer_RSA_big_int n e 6 ms;;                                (* k = 6 <= kmax *)
(* ------ message chiffré ------ *)
(* ------ déchiffrage ------ *)
print_string (dechiffrer_RSA_big_int n d lc);;
(*================= Facteur de compactage trop grand ---------*)
(* ------ chiffrage  (chiffre identique au précédent 5.3.2.2. ) *)
let lc = chiffrer_RSA_big_int n e 8 ms;;                                (* k = 8 trop grand *)
(* ------ déchiffrage ------ *)
print_string (dechiffrer_RSA_big_int n d lc);;

(*======================================== Attaques *)
(*================= Déchiffrement par itération *)
let repdechiffrer_RSA n e o ls =  (* 0 < n < 2^30 *)(* e ^ o = 1 mod phi(n) *)
   let rec dechiffrer = function
      | [] -> ""
      | m::q -> let w = ref m in for i = 1 to o-1 do w := exprap_mod n !w e done;
                degroupe (dechiffrer q) !w
   and     degroupe accu = function
      | y when y = 0 -> accu
      | y ->  let q = y / 256 and r = y mod 256 in
              degroupe ((String.make 1 (char_of_int r)) ^ accu) q
   in dechiffrer (List.map int_of_string ls);;
(* val repdechiffrer_RSA : int -> int -> int -> string list -> string = <fun> *)

let n = 463862351 and e = 25793;;
let mu = 100 in
let w = ref (exprap_mod n mu e) and i = ref 1 in
while !w <> mu && !i < 50
do w := exprap_mod n !w e ;
   i := !i + 1;
   if !w = mu then printf "\nn = %10d w = %10d i = %2d" n !w !i
done;;  (* n =  463862351 w =        100 i = 12 *)

let ms = "J'ai des informations sur le sujet du prochain DS d'info. Bob.";;
print_string (repdechiffrer_RSA n e 12 (chiffrer_RSA n e 3 ms));;
```

```
(*======================== Attaque Fermat *)
(* ------ chiffre: ------ *)
let n = big_int_of_string "65509912209240151035623" and e = big_int_of_int 27329;;
(*--------- craquage ---------*)
let x, y = fermat_big_int n 60.;;
let s = inverse_mod_big_int ((x-1)*(y-1)) e;;
(* ------ message chiffré ------ *)
let lc = [ "58708423107954916923760"; "31436899421999180466155";   "444014759554241857403";
           "20107848284631489486860"; "58312015784159115214176"; "13978316466576225911582";
           "28937690863344110299079"; "52763225684222112646755"; "27937192474109965023930";
           "22057576210221896807178"; "39521329825987178664787"; "51726656168921412590539" ];;
(* ------ déchiffrage ------ *)
print_string (dechiffrer_RSA_big_int n s lc);;

(*========================== Attaques Pollard *)
(* ------ chiffre: ------ *)
let n = big_int_of_string "663084496823009987142848811l9" and e = big_int_of_string "206977";;
(*------ craquage ------------*)
let x = pollard_pmu_big_int n 270 2;;
print_big_int x;;
let x = pollard_rho_big_int n 10.;;
print_big_int x;;
...
let s = inverse_mod_big_int ((x-1)*(y-1)) e;;
(* ------ message chiffré ------ *)
let lc = ["1146228123974337830308698241l";  "706248904889505872219814698l"; "272612860349718267199569968l2";
          "213817576264180784817612887l8"; "56692619525049824268370144203"; "40886615914253489189277873939";
            "133148660649654723673568439";   "4653218489793919653086305l3" ];;
(* ------ déchiffrage ------ *)
print_string (dechiffrer_RSA_big_int n s lc);;
(*======================================================= Chiffre plus résistant *)
(* recherche de u, a <= u < b, u premier, u-1 = 2*v, avec v premier *)
let get_primeB_big_int a b =
  let c = sub_big_int b a and x = ref (big_int_of_int 4) and fini = ref false in
  while not !fini
  do x := add_big_int a (random_big_int c);
     if isprime_big_int !x then
     begin let x1 = pred_big_int !x in
           let x2 = div_big_int x1 deux_big_int in
           if isprime_big_int x2 then fini := true
     end
  done;
  !x;;
(* get_primeB_big_int : big_int -> big_int -> big_int = <fun> *)

let p = get_primeB_big_int (big_int_of_string          "10000000000000000000")
                           (big_int_of_string "100000000000000000000000000000");;
print_big_int p;;                              (* 80547583779756205444531102l9 *)

let q = get_primeB_big_int (big_int_of_string "1000000000000000000000000000000000")
                           (big_int_of_string "100000000000000000000000000000000000000000000000000000000");;
print_big_int q;;                              (* 662929336042400956680059163332327024942680309088189159 *)
(* ------ chiffre ---- *)
let p = big_int_of_string "80547583779756205444531102l9"
and q = big_int_of_string "662929336042400956680059163332327024942680309088189159";;
let n = mult_big_int p q;;
print_big_int n;;   (* 53397356234933446127751467795404491472541804305555374601775600281372381214392653l1 *)
let e = ...;;
...
(* ------ craquage ------ *)
(* let x, y = fermat_big_int n (3600. *. 24.);;            *)
(* let x = pollard_pmu_big_int n 270 2;;                   *)
(* let x = pollard_rho_big_int n (3600. *. 24.);;          *)
...
(*==== FIN ===*)
```

$< \mathscr{FIN} >$ (énoncé TP RSA + squelette) . . .

8 OCaml : corrigé TP RSA

```
(* OCaml >= 4.02   *)
(* Corrigé TP RSA. *)
open Printf;;
#load "graphics.cma";;
open Graphics;;
#load "nums.cma"
open Big_int;;
let deux_big_int = big_int_of_int 2;;
#load "unix.cma";;
(*============================ Utilitaires ===========*)
let print_big_int (b : Big_int.big_int) : unit =
   print_string (string_of_big_int b)
;;
(*========= Calculs version int *)
let rec exprap_mod n a = function  (* a^e mod n, récursif; 0 < n, a < 2^30 < sqrt(max_int) = 2147483648 *)
  | 0 -> 1
  | e -> let u = exprap_mod n a (e/2) in
         let v = (u*u) mod n in if e mod 2 = 0 then v else (v*a) mod n
;;
let exprap_mod n a e  =            (* a^e mod n, itératif; 0 < n, a < 2^30 < sqrt(max_int) = 2147483648 *)
  let r = ref 1 and b = ref a and exp = ref e in
  while !exp > 0
  do if !exp land 1 = 1 then r := (!r * !b) mod n;
     exp := !exp lsr 1;
     b := (!b * !b) mod n
  done;
  !r;;
(* exprap_mod : int -> int -> int -> int = <fun> *)

let rec gcd (x : int) (y : int) : int = if y = 0 then x else gcd y (x mod y)
;;
let rec bezout x = function                        (* ax+by = gcd x y; 0 < x, y < 2^30 *)
    0 -> [|1; 0; x|]
  | y -> let u = bezout y (x mod y) in [| u.(1); u.(0) - u.(1) * (x / y); u.(2) |]
;;
let bezout x y =     (* version itérative *)      (* ax+by = gcd x y; 0 < x, y < 2^30 *)
   let u = [|1; 0; x|] and v = [|0; 1; y|] and t = [|0; 0; 0|] in
   while v.(2) <> 0
   do let q = u.(2) / v.(2) in
      for i = 0 to 2 do t.(i) <- u.(i) - q * v.(i) done;
      for i = 0 to 2 do u.(i) <- v.(i); v.(i) <- t.(i) done;
   done;
   u;;
(* val bezout : int -> int -> int array = <fun> *)

let inverse_mod (n : int) (x : int) : int =       (* x^(-1) mod n; 0 < n, x < 2^30 *)
  let u = bezout x n in
  if u.(2) = 1 then let z = u.(0) in if z >= 0 then z else z+n
               else failwith "non inversible"
;;
let quotient_mod (n : int) (x : int) (y : int) : int =    (* x/y mod n; 0 < n, x , y < 2^30 *)
    (x * (inverse_mod n y)) mod n
;;
exprap_mod 500 7 4;;      (* 401 *)
gcd 48 18;;               (* 6 *)
bezout 132 246;;          (* [|-13; 7; 6|] *)
inverse_mod 10 7;;        (* 3 *)
inverse_mod 10 6;;        (* non inversible *)
quotient_mod 10 2 7;;     (* 7 * 3 = 1 (10); 2 * 3 = 6 (10) *)
```

```
(*========== Calculs version big_int *)
let exprap_mod_big_int n a =                          (* a^e mod n; version récursive *)
  let rec exprm = function
    | e when eq_big_int e zero_big_int -> unit_big_int
    | e -> let (q, r) = quomod_big_int e deux_big_int in
           let u = exprm q in
           let v = mod_big_int (square_big_int u) n in
           if eq_big_int r zero_big_int then v else mod_big_int (mult_big_int v a) n
  in exprm
;;
let exprap_mod_big_int n a e =                        (* a^e mod n; version itérative *)
  let r = ref unit_big_int and b = ref a and exp = ref e in
  while gt_big_int !exp zero_big_int
  do if not (eq_big_int (mod_big_int !exp deux_big_int) zero_big_int)
     then r := mod_big_int (mult_big_int !r !b) n;
     exp := div_big_int !exp deux_big_int;
     b := mod_big_int (mult_big_int !b !b) n;
  done;
  !r;;
(* val exprap_mod_big_int : big_int -> big_int -> big_int -> big_int = <fun> *)

let rec bezout_big_int x = function
  | y when eq_big_int y zero_big_int -> [| unit_big_int; zero_big_int; x|]
  | y -> let u = bezout_big_int y (mod_big_int x y) in
         [| u.(1); sub_big_int u.(0) (mult_big_int u.(1) (div_big_int x y)); u.(2) |];;
(* val bezout_big_int : big_int -> big_int -> .big_int array = <fun> *)

let inverse_mod_big_int (n : big_int) (x : big_int) : big_int =   (* x^(-1) mod n *)
  let u = bezout_big_int x n in
  if eq_big_int u.(2) unit_big_int
  then let z = mod_big_int u.(0) n in if ge_big_int z zero_big_int then z else add_big_int z n
  else failwith "non inversible"
;;
let quotient_mod_big_int (n : big_int) (x : big_int) (y : big_int ) : big_int =    (* x/y mod n *)
   mod_big_int (mult_big_int x (inverse_mod_big_int n y)) n
;;
print_big_int (exprap_mod_big_int (big_int_of_int 100) (big_int_of_int 7) (big_int_of_int 4));;(*1=2401(100)*)
print_big_int (inverse_mod_big_int (big_int_of_int 100) (big_int_of_int 7));; (* 43: 7 * 43 = 301 = 1 (100) *)
print_big_int (quotient_mod_big_int (big_int_of_int 100) (big_int_of_int 2) (big_int_of_int 7));; (*86=2*43 *)
(*====================================*)
(* Nombres pseudo-aléatoires         *)
(*====================================*)
(*========== Version int *)
let random_int n =          (* 0 < n <= max_int *)
  let d = 2 lsl 25 in
  let k = ref 0 and q = ref n and r = ref 0 in
  while !q <> 0 do r := !q; q := !q / d; k := !k+1 done;
  let u = ref (Random.int d) in
  for i = 0 to !k
  do let z = Random.int d in
     let zz = Random.int d in
     u := ( (!u * zz) + z ) mod max_int;
     if !u < 0 then u := !u + max_int
  done;
  !u mod n;;
(* val random_int : int -> int = <fun> *)

open_graph " 800x800+10+10"; set_color black;  (* test graphique de random_int *)
let n = 100000 in
for i = 1 to 25000
do let x = random_int n and y = random_int n in
   let x1 =  ((float_of_int x) /. (float_of_int (n))) and y1 =  ((float_of_int y) /. (float_of_int (n))) in
   let x2 = 100 + int_of_float (500. *. x1) and y2 = 100 + int_of_float (500. *. y1) in
   plot x2 y2
done;;
```

```
(*========== Version big_int *)
let random_big_int n =
  let k = num_digits_big_int n in
  let u = ref (big_int_of_int (random_int max_int)) in
  for i = 0 to k
  do let z = random_int max_int in
     let zz = (random_int max_int) in
     u := add_big_int (mult_int_big_int zz !u) (big_int_of_int z)
  done;
  mod_big_int !u n;;
(* val random_big_int : big_int -> big_int = <fun> *)

open_graph " 800x800+10+10"; set_color black; (* test graphique de random_big_int *)
let n = big_int_of_string "100000000000000000000000000000000000000" in
for i = 1 to 25000
do let x = random_big_int n and y = random_big_int n in
   let x1 = (float_of_big_int x) /. (float_of_big_int n)
   and y1 = (float_of_big_int y) /. (float_of_big_int n) in
   let x2 = 100 + int_of_float (500. *. x1)
   and y2 = 100 + int_of_float (500. *. y1) in
   plot x2 y2
done;;

(*============================== Test de primalité, factorisation *)
(*========== Version int  *)
let miller_rabin (n : int) (k : int) : bool =                (* n < 2 ** 30 *)
  if n < 4 then true
  else
  begin
    let z = n-1 in
    let s = ref 0 and d = ref z in
    while !d mod 2 = 0 do s := !s+1; d := !d / 2 done;

    if !s = 0 then false else
    try
      for _ = 1 to k    (* k essais *)
      do let a = 1 + random_int (z-1) in
         let b = ref (exprap_mod n a !d) in
         try if !b = 1
             then failwith "premier"
             else begin for r = 0 to !s-1
                        do if !b mod n = z then failwith "premier"
                           else b := (!b * !b) mod n
                        done;
                        failwith "factorisable"
                  end;
         with Failure "premier" -> ();      (* passe *)
      done;
      true                                  (* "premier" *)
    with Failure "factorisable" -> false    (* "factorisable" *)
  end
;;
let isprime (n : int) : bool = (miller_rabin n 30)                    (* n < 2 ** 30 *)
;;
let get_prime (a : int) (b : int) : int =
  let c = b - a and x = ref 4 in
  while not (isprime !x) do x := a + (random_int c) done;
  !x
;;
isprime ((2 lsl 16) -1);;               (* 2^17 -1 *)  (* true *)
isprime ((2 lsl 17) -1);;               (* 2^18 -1 *)  (* false *)
isprime ((2 lsl 18) -1);;               (* 2^19 -1 *)  (* true *)
isprime ((2 lsl 22) -1);;               (* 2^23 -1 *)  (* false *)
isprime ((2 lsl 30) -1);;               (* 2^31 -1 *)  (* true *)
```

```
let eratosthene n =
  let nn = ref n and h = ref 0 and k = ref 0 and ss = ref "" in

  while !nn mod 2 = 0 do h := !h+1; nn := !nn / 2 done;
  if !h > 0 then ss := !ss ^ "2";
  if !h > 1 then ss := !ss ^ "^" ^ (string_of_int !h);

  for i = 1 to (int_of_float (sqrt (float_of_int n))) /2
  do let u = 2*i+1 in
     k := 0; while !nn mod u = 0 do k := !k+1; nn := !nn / u done;
     if !k > 0 then if String.length !ss > 0
                    then ss := !ss ^ "*" ^ (string_of_int u)
                    else ss := !ss ^ (string_of_int u);
     if !k > 1 then ss := !ss ^ "^" ^ (string_of_int !k)
  done;

  if !nn > 1
  then if String.length !ss > 0
       then ss := !ss ^ "*" ^ (string_of_int !nn) else ss := !ss ^ (string_of_int !nn);
  !ss;;
(* val eratosthene : int -> string = <fun> *)

let n = 2*3*4*5*6*7*8*9*10*11*12*13*14*15*16 in (* 20922789888000 *)
eratosthene n;;                  (* "2^15*3^6*5^3*7^2*11*13" *)
eratosthene (20922789888000);;   (* "2^15*3^6*5^3*7^2*11*13" *)
eratosthene (20922789888000+1);; (* "17*61*137*139*1059511" *)

let fermat n delay =                    (* delay (sec.) *)
  let time0 = Sys.time () in
  let s2n = int_of_float (sqrt(float_of_int n)) in
  let x' = ref (2*s2n+1) and y' = ref 1
  and r = ref (s2n*s2n - n) in
  while !r <> 0
  do if !r > 0
     then begin r := !r - !y'; y' := !y' + 2 end
     else begin r := !r + !x'; x' := !x' + 2 end;
     let time1 = Sys.time () in if time1 -. time0 > delay then failwith "Time out !"
  done;
  (!x' - !y')/2, (!x' + !y'-2)/2;;
(* val fermat : int -> float -> int * int = <fun> *)

let pollard_pmu n b g =  (* Pollard p-1 *)  (* n < 2 ** 30 *)
  let a = ref g in
  for k = 2 to b do a := exprap_mod n !a k done;  (* -> a^(b!) modulo n *)
  let d = gcd (!a-1) n in
  if (1 < d) && (d < n) then d else 1;;
(* val pollard_pmu : int -> int -> int -> int = <fun> *)

let pollard_rho n delay =      (* Pollard rho *)  (* n < 2 ** 30 ; delay (sec.) *)
  let time0 = Sys.time () in
  let g = (function x -> (x*x+1) mod n) in
  let x = ref 1 and y = ref 1 and d = ref 1 in
  while !d = 1
  do x := g !x; y := g (g !y);
     d := gcd (abs (!x - !y)) n;
     let time1 = Sys.time () in if time1 -. time0 > delay then failwith "Time out !"
  done;
  if !d = n then 1 else !d;;
(* val pollard_rho : int -> float -> int = <fun> *)

let x, y = fermat (151237 * 153509) 10. in printf "%6d * %6d\n" x y;;  (* 151237 * 153509 *)

let p = 322009 and q = 5147 in printf "\n%20s" (eratosthene (p-1)); printf "%20s\n" (eratosthene (q-1));
printf "%20d" (pollard_pmu (p*q) 200 2);  (* 5147 *) printf "%20d" (pollard_rho (p*q) 10.);;     (* 5147 *)
let p = 2590689529 and q = 5100097 in printf "\n%20s" (eratosthene (p-1)); printf "%20s\n" (eratosthene (q-1));
printf "%20d" (pollard_pmu (p*q) 300 2);  (* 1 *) printf "%20d" (pollard_rho (p*q) 10.);;     (* 5100097 *)
```

```
(*============== Version big_int *)
let miller_rabin_big_int (n : big_int) (k : int) : bool =
  if le_big_int n (big_int_of_int 3) then true
  else
  begin
    let z = pred_big_int n in
    let s = ref 0 and d = ref z in
    while eq_big_int (mod_big_int !d deux_big_int) zero_big_int
    do s := !s +1; d := div_big_int !d deux_big_int done;

    if !s = 0 then false else
    try
      for _ = 1 to k     (* k essais *)
      do let a = succ_big_int ( random_big_int (pred_big_int z) ) in
         let b = ref (exprap_mod_big_int n a !d) in
         try if eq_big_int !b unit_big_int
             then failwith "premier"
             else begin for r = 0 to !s-1
                        do if eq_big_int (mod_big_int !b n) z
                           then failwith "premier"
                           else b := mod_big_int (square_big_int !b) n
                        done;
                        failwith "factorisable"
                  end
         with Failure "premier" -> ();
      done;
      true                                    (* "premier"      *)
    with Failure "factorisable" -> false      (* "factorisable" *)
  end
;;
let isprime_big_int (n : big_int) : bool =
   (miller_rabin_big_int n 30)
;;
let get_prime_big_int (a : big_int) (b : big_int) : big_int =
  let c = sub_big_int b a in
  let x = ref (big_int_of_int 4) in
  while not (isprime_big_int !x) do x := add_big_int a (random_big_int c) done;
  !x
;;

isprime_big_int (pred_big_int (power_int_positive_int 2  31));;  (* true *)
isprime_big_int (pred_big_int (power_int_positive_int 2  43));;  (* false *)
isprime_big_int (pred_big_int (power_int_positive_int 2  61));;  (* true *)
isprime_big_int (pred_big_int (power_int_positive_int 2  73));;  (* false *)
isprime_big_int (pred_big_int (power_int_positive_int 2  89));;  (* true *)
isprime_big_int (pred_big_int (power_int_positive_int 2  97));;  (* false *)
isprime_big_int (pred_big_int (power_int_positive_int 2 107));;  (* true *)
isprime_big_int (pred_big_int (power_int_positive_int 2 117));;  (* false *)

let fermat_big_int (n : big_int) (delay : float) : big_int * big_int =    (* delay (sec.) *)
  let time0 = Sys.time () in
  let s2n = sqrt_big_int n in
  let x1 = ref (succ_big_int (mult_int_big_int 2 s2n))
  and y1 = ref unit_big_int
  and r = ref (sub_big_int (square_big_int s2n) n) in
  while not (eq_big_int !r zero_big_int)
  do if ge_big_int !r zero_big_int
     then begin r := sub_big_int !r !y1; y1 := add_int_big_int 2 !y1 end
     else begin r := add_big_int !r !x1; x1 := add_int_big_int 2 !x1 end;
     let time1 = Sys.time () in if time1 -. time0 > delay then failwith "Time out !"
  done;
  div_big_int (sub_big_int !x1 !y1) deux_big_int,
  div_big_int (add_int_big_int (-2) (add_big_int !x1 !y1)) deux_big_int
;;
```

```
let pollard_pmu_big_int n b g =                  (* Pollard p-1 *)
  let a = ref (big_int_of_int g) in
  for k = 2 to b do a:= exprap_mod_big_int n !a (big_int_of_int k) done;

  let d = gcd_big_int (pred_big_int !a) n
  in if (lt_big_int unit_big_int d) && (lt_big_int d n) then d else unit_big_int
;;
(* val pollard_pmu_big_int : big_int -> int -> int -> big_int = <fun> *)

let pollard_rho_big_int n delay =                (* Pollard rho *)  (* delay (sec.) *)
  let time0 = Sys.time () in
  let g = function x -> succ_big_int (mod_big_int (square_big_int x) n) in    (* x^2+1 *)
  let x = ref unit_big_int and y = ref unit_big_int and d = ref unit_big_int in
  while eq_big_int !d unit_big_int
  do x := g !x; y := g (g !y);
     d := gcd_big_int (abs_big_int (sub_big_int !x !y)) n;
     let time1 = Sys.time () in if time1 -. time0 > delay then failwith "Time out !"
  done;
  if eq_big_int !d n then unit_big_int else !d
;;
(* val pollard_rho_big_int : big_int -> float -> big_int =  <fun> *)

let n = mult_big_int (big_int_of_int 26215815793911) (big_int_of_int 26215815793973) in
let x, y = fermat_big_int n 60. in
print_big_int x; print_newline ();
print_big_int y; print_newline ();;
(*-----------------------------------------*)
let u = big_int_of_string "238130989262374727";;       (* premier *)
let v = big_int_of_string "1245671712577924737133361";; (* premier *)
let p = big_int_of_int (2*7*13*53*53*53 + 1);;
print_big_int p ;;                               (* 27095615 *)
let n = mult_big_int p (mult_big_int u v);;
print_big_int n;; (* 803745457258617645630804802052238990423382344905 *)

print_big_int (pollard_pmu_big_int n 180 2);;                    (* 27095615 *)
print_big_int (pollard_rho_big_int n 10.);;                      (* 5 *)
(*---------------------------------*)
let n = big_int_of_string "184467440737095516l7" in
print_big_int (pollard_rho_big_int n 10.); print_newline ();      (* 274177 *)
print_big_int (pollard_pmu_big_int n 1000 2); print_newline ();   (* 1 *)
print_big_int (pollard_pmu_big_int n 20 3);print_newline ();      (* 274177 *)
printf "%20s" (eratosthene (274177-1));; (* 2^8*3^2*7*17 *)

(*======================================== Chiffrage et déchiffrage *)
(*=========== Version int *)
let chiffrer_RSA n e k mess =          (* 0 < n < 2^30 *)
  let rec chiffrer i m j =
    if j = String.length mess
    then if m = 0 then [] else [exprap_mod n m e]
    else let m1 = (int_of_char mess.[j]) + (256 * m) in
         if i = k then (exprap_mod n m1 e)::(chiffrer 1 0 (j+1)) else chiffrer (i+1) m1 (j+1)
  in List.map string_of_int (chiffrer 1 0 0);;
(*  val chiffrer_RSA : int -> int -> int -> string -> string list = <fun> *)

let dechiffrer_RSA n d ls =    (* 0 < n < 2^30 *)
   let rec degroupe accu = function
      | y when y = 0 -> accu
      | y ->  let q = y / 256 and r = y mod 256 in
              degroupe ((String.make 1 (char_of_int r)) ^ accu) q
   and     dechiffrer = function
      | []   -> ""
      | m::q -> degroupe (dechiffrer q) (exprap_mod n m d)
   in dechiffrer (List.map int_of_string ls);;
(* val dechiffrer_RSA : int -> int -> string list -> string = <fun> *)
```

```
(*=========== Version big_int *)
let chiffrer_RSA_big_int n e k mess =
  let rec chiffrer i m j =
    if j = String.length mess
    then if eq_big_int m zero_big_int then [] else [exprap_mod_big_int n m e]
    else let m1 = add_int_big_int (int_of_char mess.[j]) (mult_int_big_int 256 m) in
         if i = k then (exprap_mod_big_int n m1 e)::(chiffrer 1 zero_big_int (j+1))
                  else chiffrer (i+1) m1 (j+1)
  in List.map string_of_big_int (chiffrer 1 zero_big_int 0);;
(* val chiffrer_RSA_big_int : big_int -> big_int -> int -> string -> string list = <fun> *)

let dechiffrer_RSA_big_int n d ls =
   let rec degroupe accu = function
      | y when eq_big_int y zero_big_int -> accu
      | y ->  let (q,r) = quomod_big_int y (big_int_of_int 256) in
              degroupe ((String.make 1 (char_of_int (int_of_big_int r))) ^ accu) q;
   and     dechiffrer = function
      | []   -> ""
      | m::q -> degroupe (dechiffrer q) (exprap_mod_big_int n m d)
   in dechiffrer (List.map big_int_of_string ls);;
(* val dechiffrer_RSA_big_int : big_int -> big_int -> string list -> string *)
(*============================ Tests élémentaires *)
(*========== Version big_int -------*)
(* ------ chiffre ----- *)
let p = big_int_of_int 13999 and q = big_int_of_int 14243;;
let n = mult_big_int p q;;
print_big_int n;;           (*           199387757 *)
print_int max_int;;         (* 4611686018427387903 *)
print_int (2 lsl 30);;      (*          2147483648 *)

let p1q1 = mult_big_int (pred_big_int p) (pred_big_int q);;
let e = big_int_of_int 16481;;
let d = inverse_mod_big_int p1q1 e;;
print_big_int e;;                     (*     16481 *)
print_big_int d;;                     (* 194170193 *)
print_big_int (mod_big_int (mult_big_int e d) p1q1);; (* verif -> 1 *)
(* ------ chiffrage ----- *)
let ms = "Le prochain DS d'info ne comportera qu'une petite partie de logique. Signé Bob.";;
printf "k < %12.2f" ((log (float_of_big_int n)) /. log 256.);;  (* 3.45 *)
let lc = chiffrer_RSA_big_int n e 3 ms;;   (* k = 3 < 3.45 *)
(* ------ message chiffré ----- *)
let lc = [ "118662545";  "99068013"; "132200431";  "89291238";  "68393044";  "22108232"; "185504288";
           "113582398"; "159540846"; "185844420"; "153659394";  "90374658";  "67338774"; "114267357";
            "58719682"; "126317319";  "14898677"; "177687227"; "161993329"; "184753009"; "193210174";
            "67752315";   "2790210"; "187934739"; "179014827";  "56565905"; "117787729" ];;
(* ------ craquage ----- *)
let x, y = fermat_big_int n 10.;;
print_big_int x;;
print_big_int y;;
let x1 = pred_big_int x and y1 = pred_big_int y;;
let x1y1 = mult_big_int x1 y1;;
let s = inverse_mod_big_int x1y1 e;;
print_big_int s;;
(* ------ déchiffrage ------ *)
let mdc = dechiffrer_RSA_big_int n d lc in print_string mdc;;
(*=========== La clef du coffre       *)
(* ----- chiffrage (chiffre identique à celui de l'exemple précédent 5.3.1. ) *)
let ms = "Le code secret d'accès à la salle des coffres est ZpX01Wzz2";;
let lc = chiffrer_RSA_big_int n e 3 ms;;  (* k = 3 < kmax *)
(* ------ message chiffré --- *)
(* ------ déchiffrage ------- *)
let mdc = dechiffrer_RSA_big_int n d lc in print_string mdc;;
```

```
(*==================================== module de chiffrement > 2^30 *)
(*============== Version int = Problème ---*)
(* ------ chiffre ----- *)
let p = 276464707 and q = 114626177;;
let n = p * q;;
print_int n;;            (*   31690092438835139 *)
print_int max_int;;      (* 4611686018427387903 *)
print_int (2 lsl 30);;           (* 2147483648 *)
let e = 152423;;
let d = inverse_mod ((p-1)*(q-1)) e;;
printf "e = %20d  d = %20d" e d;;       (* e = 152423  d = 1553494996035671 *)
print_int (( e * d) mod ((p-1)*(q-1)));;          (* différent de 1 : PROBLEME ! Stop! *)
(*=============== Version Big_int ----------------*)
(* ------ chiffre ---- *)
let p = big_int_of_int 276464707 and q = big_int_of_int 114626177;;
let n = mult_big_int p q;;
print_big_int n;;                   (*   31690092438835139  *)

let p1q1 = mult_big_int (pred_big_int p) (pred_big_int q);;
let e = big_int_of_int 152423;;
let d = inverse_mod_big_int p1q1 e;;
print_big_int e;;                   (*           152423 *)
print_big_int d;;                   (* 1553494996035671 *)
print_big_int (mod_big_int (mult_big_int e d) p1q1);;                    (* = 1 ok *)
(* ------ chiffrage ----- *)
let ms = "Le sujet du prochain DS d'info comportera trois parties. Signé Bob.";;
printf "k < %12.2f" ((log (float_of_big_int n)) /. log 256.);;           (* k < 6.85 *)
let lc = chiffrer_RSA_big_int n e 6 ms;;                                 (* k = 6 <= kmax *)
(* ------ déchiffrage -------- *)
let mdc = dechiffrer_RSA_big_int n d lc in print_string mdc;;

(*================= Facteur de compactage trop grand *)
(* ------ chiffrage  (chiffre identique à celui de l'exemple précédent) *)
let lc = chiffrer_RSA_big_int n e 8 ms;;                                 (* k = 8 trop grand *)
(* ------ déchiffrage ------ *)
let mdc = dechiffrer_RSA_big_int n d lc in print_string mdc;;

(*======================================== Attaques  *)
(*================= Déchiffrement par itération *)
let repdechiffrer_RSA n e o ls =  (* 0 < n < 2^30 *)(* e ^ o = 1 mod phi(n) *)
   let rec dechiffrer = function
      | [] -> ""
      | m::q -> let w = ref m in for i = 1 to o-1 do w := exprap_mod n !w e done;
                degroupe (dechiffrer q) !w
   and     degroupe accu = function
      | y when y = 0 -> accu
      | y ->  let q = y / 256 and r = y mod 256 in
              degroupe ((String.make 1 (char_of_int r)) ^ accu) q
   in dechiffrer (List.map int_of_string ls);;
(* val repdechiffrer_RSA : int -> int -> int -> string list -> string = <fun> *)

let n = 463862351 and e = 25793;;
let mu = 100 in
let w = ref (exprap_mod n mu e) and i = ref 1 in
while !w <> mu && !i < 50
do w := exprap_mod n !w e ;
   i := !i + 1;
   if !w = mu then printf "\nn = %10d w = %10d i = %2d" n !w !i
done;;  (* n =  463862351 w =        100 i = 12 *)

let ms = "J'ai des informations sur le sujet du prochain DS d'info. Bob.";;
print_string (repdechiffrer_RSA n e 12 (chiffrer_RSA n e 3 ms));;
```

```
(*================= Attaque Fermat *)
(* ------ chiffre: module de chiffrement------ *)
let p = big_int_of_int 255949039601 and q = big_int_of_int 255949044823;;
isprime_big_int p;;  (* true *)
isprime_big_int q;;  (* true *)
let n = mult_big_int p q;;
print_big_int n;;           (* 65509912209240151035623 *)
(*..... *)
let e = big_int_of_int 27329;;
(*..... *)
(*--------- craquage ---------*)
let x, y = fermat_big_int n 60.;;
print_big_int x;;                  (* 255949039601 *)
print_big_int y;;                  (* 255949044823 *)
let x1y1 = mult_big_int (pred_big_int x) (pred_big_int y);;
let s = inverse_mod_big_int x1y1 e;;
print_big_int s;;
(* ------ message chiffré ------ *)
let ms = "La deuxième partie du prochain DS d'info portera sur les arbres binaires équilibrés. Bob.";;
print_string ms;;
let lc = chiffrer_RSA_big_int n e 8 ms;; (* k = 8 *)

let lc = [ "58708423107954916923760"; "31436899421999180466155";   "444014759554241857403";
           "20107848284631489486860"; "58312015784159115214176"; "13978316466576225911582";
           "28937690863344110299079"; "52763225684222112646755"; "27937192474109965023930";
           "22057576210221896807178"; "39521329825987178664787"; "51726656168921412590539" ];;
(* ------ déchiffrage ------ *)
let mdc = dechiffrer_RSA_big_int n s lc in print_string mdc;;     (* retrouve les accents *)
(* "La deuxième partie du prochain DS d'info portera sur les arbres binaires équilibrés. Bob." *)

(*==================== Attaques Pollard *)
(* ------ chiffre: module de chiffrement------ *)
let p = big_int_of_string "2590689529" and q = big_int_of_string "25594903958984194711";;
eratosthene(2590689529 - 1);;             (* 2^3*3^3*7*59*113*257 *)
(* q-1 = 25594903958984194711 - 1 = 2*3*5*59*14460397716940223    *)
let n = mult_big_int p q;;
print_big_int n;;                         (* 66308449682300998714284881119 *)
(*..... *)
let e = big_int_of_string "206977";;
(*..... *)
(*------ craquage ------------*)
let x = pollard_pmu_big_int n 270 2;;
print_big_int x;;                       (* 2590689529 *)
let x = pollard_rho_big_int n 10.;;
print_big_int x;;                       (* 2590689529 *)
let y = div_big_int n x;;
print_big_int x;;           (* 2590689529 *)
print_big_int y;;           (* 25594903958984194711 *)
let x1y1 = mult_big_int (pred_big_int x) (pred_big_int y);;
let s = inverse_mod_big_int x1y1 e;;
print_big_int s;;
(* ------ message chiffré ------ *)
let ms = "La troisième partie du prochain DS d'info portera sur les automates. Bob.";;
print_string ms;;
let lc = chiffrer_RSA_big_int n e 10 ms;;

let lc = ["1146228123974337830308698241 2";  "706248904889505872219814698 5"; "272612860349718267199569968 72";
          "213817576264180784817612887 18"; "566926195250498242683701442 03"; "408866159142534891892778739 39";
            "133148660649654723673568439";   "465321848979391965308630523" ];;
(* ------ déchiffrage ------ *)
let mdc = dechiffrer_RSA_big_int n s lc in print_string mdc;;     (* retrouve les accents *)
(* "La troisième partie du prochain DS d'info portera sur les automates. Bob." *)
```

```
(*====================================== Chiffre plus résistant *)
(* recherche de u, a <= u < b, u premier, u-1 = 2*v avec v premier *)
let get_primeB_big_int a b =
  let c = sub_big_int b a and x = ref (big_int_of_int 4) and fini = ref false in
  while not !fini
  do x := add_big_int a (random_big_int c);
     if isprime_big_int !x then
     begin let x1 = pred_big_int !x in
           let x2 = div_big_int x1 deux_big_int in
           if isprime_big_int x2 then fini := true
     end
  done;
  !x;;
(* get_primeB_big_int : big_int -> big_int -> big_int = <fun> *)

let p = get_primeB_big_int (big_int_of_string "10000000000000000000")
                           (big_int_of_string "100000000000000000000000000000");;
print_big_int p;;                              (* 80547583779756205444531102 79 *)

let q = get_primeB_big_int (big_int_of_string "1000000000000000000000000000000000000")
                           (big_int_of_string "10000000000000000000000000000000000000000000000000000000000");;
print_big_int q;;                              (* 662929336042400956680059163332327024942680309088189159 *)

(* On reprend ces valeurs dans la suite ... *)
(* ------ chiffre ---- *)
let p = big_int_of_string "80547583779756205444531102 79"
and q = big_int_of_string "662929336042400956680059163332327024942680309088189159";;
let n = mult_big_int p q;;
print_big_int n;;   (* 53397356234933446127751467795404491472541804305555374601775600281372381214392653 61 *)
let w = (power_int_positive_int 2 271) in
print_big_int w;;   (* 37942751801283770916395740367646853645359508575237100024449461127712974320414228 48 *)
let p1 = pred_big_int p and q1 = pred_big_int q;;
let p1q1 = mult_big_int p1 q1;;
let e = big_int_of_string "3554386219";;
isprime_big_int e;;
let d = inverse_mod_big_int p1q1 e;;
print_big_int e;; (* 3554386219 *)
print_big_int d;; (* 23502918482598587343755590343857594898988127783729617494495789405821065408717529 95 *)
print_big_int (mod_big_int (mult_big_int d e) p1q1);; (* 1 ok *)
(* ------ chiffrage ------ *)
let ms = "Bravo Alice! Ton nouveau code est super résistant. Oscar est dans les choux ...
.Je t'envoie le sujet et le corrigé dans le prochain courrier. Bob.";;

let z = float_of_big_int n; in (log z) /. (log 256.);; (* 33.93 = borne sup du facteur k de groupement *)
let lc = chiffrer_RSA_big_int n e 20 ms;;  (* k = 20 < 33 *)
(* ------ message chiffré ----- *)
(* ------ craquage ------ *)
(* let x, y = fermat_big_int n (3600. *. 24.);; *)
(* let x = pollard_pmu_big_int n 270 2;; *)
(* let x = pollard_rho_big_int n (3600. *. 24.);; *)
(* ------ déchiffrage ------ *)
let mdc = dechiffrer_RSA_big_int n d lc in print_string mdc;;

(*=============================================================*)
(* Oscar vient d'acheter un nouvel ordinateur superpuissant ...  *)
(*=============================================================*)
(* === The End === *)
```

9 Python 3 : corrigé TP RSA

```python
#!/usr/bin/env python3
""" Corrigé TP RSA. """
import matplotlib.pyplot as plt, math, random, time
import sys
try: shell = sys.stdout.shell
except AttributeError: raise RuntimeError("you must run this program in IDLE")
#=================== Simulation du module graph de OCaml
def open_graph(xmin,xmax,ymin,ymax):
    plt.figure(figsize=(10,10))
    plt.box("off"); plt.axis("off")  # "off": supp axis, grid. Sinon: auto-tracés
    plt.tight_layout(0.1)
    plt.xlim(xmin, xmax)
    plt.ylim(ymin, ymax)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.gcf().set_facecolor("white")

#============ Test Random ================================
print("randint(0,b) =", random.randint(0,10000000000000000000000000000000000))

open_graph(100,700,100,700)
n = 100000000000000000000
lx = []; ly = []; ls = []
for i in range(0,100000):
    x = random.randint(0,n); y = random.randint(0,n)
    x1 = x/n; y1 = y/n
    x2 = 100 + 500 * x1; y2 = 100 + 500 * y1
    lx.append(x2); ly.append(y2); ls.append(0.5)
plt.scatter(lx,ly, s=ls)
ms = "Close the graph window to continue..."
print(ms); plt.text(100,650,ms, ha="left",va="center", color="black", zorder=3, fontsize=30)
plt.show() #block=True)

#=========================================================
def exprapmod(x,y,n):
    r = 1
    while y > 0:
        if y & 1 > 0: r = (r * x) % n
        y >>= 1
        x = (x * x) % n
    return r

def bezout(x,y):
    if y == 0: return [1, 0, x]
    u = bezout(y, x % y)
    return [u[1], u[0]-u[1]*(x//y), u[2]]

def inversemod(x, n):
    u = bezout(x, n)
    if u[2] == 1:
        z = u[0]
        return z if z >= 0 else z+n
    else: return 0                    # 0 <=> "non inversible"

def quotientmod(x,y,n):
    return ((x * (inversemod(y,n))) % n)

print("7^4 % 500 =",  exprapmod(7,4,500) )    #  401
print("gcd(48,18) =", math.gcd(48,18) )       #  6
print("bezout(132,246) =",  bezout(132,246) ) #  [|-13; 7; 6|]
print("7^(-1) % 10 =", inversemod(7, 10) )    #  3
print("6^(-1) % 10 =", inversemod(6,10))      #  0 non inversible
print("2/7 % 10 =", quotientmod(2,7,10))      #  7 * 3 = 1 (10); 2 * 3 = 6 (10)
```

```
#=========================================================
class Premier(Exception): pass
class Compose(Exception): pass

def miller_rabin(n,k):
    if n < 4 : return True
    z = n-1; s = 0; d = z
    while d % 2 == 0: s = s+1; d = d//2

    if s == 0 : return False
    try:
        for s in range(0,k):
            a = 1 + random.randint(0, z-1)
            b = exprapmod(a, d, n)
            try:
                if b == 1: raise Premier
                else:
                    for i in range(0, k):
                        if b % n == z: raise Premier
                        else: b = (b * b) % n
                    raise Compose
            except Premier: pass
        return True
    except Compose: return False

def isprime(n): return miller_rabin(n,30)

def getprime(a,b):
    x = 4
    while not isprime(x): x = random.randint(a,b)
    return x

shell.write("----- Tests primalité (nombres de Mersenne)----\n", "COMMENT")
print("2^17 -1 :", isprime ((2 << 16) -1)) # 2^17 -1  true
print("2^23 -1 :", isprime ((2 << 22) -1)) # 2^23 -1  false
print("2^31 -1 :", isprime ((2 << 30) -1)) # 2^31 -1  true
print("2^43 -1 :", isprime (2 ** 43 -1))   #    false
print("2^61 -1 :", isprime (2 ** 61 -1))   #    true
print("2^73 -1 :", isprime (2 ** 73 -1))   #    false
print("2^89 -1 :", isprime (2 ** 89 -1))   #    true
print("2^117 -1 :", isprime (2 ** 117 -1)) #    false

print("getprime(a,b) =", getprime( 10000000000000000, 100000000000000000000000000))
#=========================================================
def eratosthene(n):
    nn = n; h = 0; k = 0; ss = ""

    while nn % 2 == 0: h = h+1; nn = nn // 2
    if h > 0: ss = ss + "2";
    if h > 1: ss = ss + "^" + str(h)

    for i in range(1, math.floor(math.sqrt(n)/2)):
        u = 2*i+1; k = 0
        while nn % u == 0: k = k+1; nn = nn // u
        if k > 0: ss = ss + "*" + str(u) if len(ss) > 0 else ss + str(u)
        if k > 1: ss = ss + "^" + str(k)

    if nn > 1: ss = ss + "*" + str(nn) if len(ss) > 0 else ss + str(nn)
    return ss

shell.write("----- Tests eratosthene\n", "COMMENT")
n = 2*3*4*5*6*7*8*9*10*11*12*13*14*15*16
print("n = 2*3*4*5*6*7*8*9*10*11*12*13*14*15*16 =", n)  # 20922789888000
print("n =", eratosthene(n))                             # "2^15*3^6*5^3*7^2*11*13"
```

```
def fermat(n, delay):
    time0 = time.clock()
    s2n = math.floor(math.sqrt(n))
    x1 = 2*s2n+1; y1 = 1; r = s2n*s2n - n
    while r != 0:
        if r > 0: r = r - y1; y1 = y1 + 2
        else: r = r + x1; x1 = x1 + 2
        if (time.clock() - time0) > delay: return (0,0)
    return ((x1 - y1)//2, (x1 + y1 - 2)//2)

def pollardpmu(n, b, g):
    a = g
    for k in range(2,b+1): a = exprapmod(a,k,n)     # -> a^(b!) mod n
    d = math.gcd(a-1, n)
    return d if (1 < d) and (d < n) else 1

def pollardrho(n, delay):
    time0 = time.clock()
    f = lambda z: z*z+1
    x, y, d = 2, 2, 1
    while d == 1:
        x = f(x) % n
        y = f(f(y)) % n
        d = math.gcd(x-y, n)
        if (time.clock() - time0) > delay: return (0,0)
    return d

shell.write("----- Tests Fermat\n", "COMMENT")
print("fermat(151237 * 153509) =", fermat(151237 * 153509, 10.))
print()
p = 1512374445; q = 153509; n = p*q
x,y = fermat(n, 10.)
print("p =", p, "; q =",q, "; n = p*q =", n)
print("(x,y) = fermat(n) =", (x,y) )    # 14583355, 15919731
print("x*y =", x*y)

shell.write("----- Tests Pollard\n", "COMMENT")
u = 238130989262374727 ; v = 124567171257792473713361   # premiers
p = 2*7*13*53*53*53 + 1
n = p * u * v               # 80374545725861764563080480205223899042338234490 5
print("u =", u, "; v =", v)
print("p =", p, "=", eratosthene(p))
print("p-1 =", p-1, "=", eratosthene(p-1))
print("n = u*v*p =", n)

print("pollardpmu(n,180,2) =", pollardpmu(n, 180, 2))   # 27095615
print("pollardrho(n) =", pollardrho(n,60))              # 95
print()
p= 2590689529; q = 5100097
print("p =", p, "; q =",q)
print("pollardrho (p*q) =", pollardrho(p*q, 60) )
#=============================================================================
def chiffrerRSA(n: int, e: int, k: int, message: str) -> list: # -> str list
    mess = message.encode('utf8')    # <class 'bytes'>
    def chiffrer(i, m, j):
        if j == len(mess): return [] if m == 0 else [exprapmod(m,e,n)]
        else:
            m1 = mess[j] + 256 * m
            if i == k:
                w = chiffrer(1, 0, j+1)
                w.insert(0, exprapmod(m1,e,n))
                return w
            else: return chiffrer(i+1, m1, j+1)
    return [str(i) for i in chiffrer(1, 0, 0)]
```

```
def dechiffrerRSA(n: int, d: int, ls0: list) -> str:  # ls0: str list
    ls = [int(s) for s in ls0]
    def degroupe(accu, y):
        if y == 0: return accu
        else:
            q = y // 256; r = y % 256
            accu = [r] + accu
            return degroupe(accu, q)
    def dechiffrer(j):
        return [] if j == len(ls) else degroupe(dechiffrer(j+1), exprapmod(ls[j], d, n))
    r = bytes(dechiffrer(0))
    return r.decode('utf8')

#======= 5.3.1/2.1. ========== Premier message
shell.write("----- Premier message\n", "COMMENT")
# ------ chiffre: module de chiffrement------
p = 13999; q = 14243; n = p * q
print("p =", p, " ; q =", q)
print("n =", n)               # 199387757

p1 = p-1; q1 = q-1; p1q1 = p1 * q1
e = 16481
d = inversemod(e,p1q1)
print("e =", e)
print("d =", d)               # 194170193
print( "(e*d)% p1q1 =", (e * d) % p1q1) # vérif -> 1
# ------ chiffrage (k=3) -----
ms = "Le prochain DS d'info ne comportera qu'une petite partie de logique. Signé Bob."
print("kmax =", math.log(n)/math.log(256))
lc = chiffrerRSA(n, e, 3, ms)  # k = 3 < kmax
print(lc)
#------ message chiffré reçu -----
lc = ['118662545',  '99068013', '132200431',   '89291238',  '68393044',  '22108232', '185504288',
      '113582398', '159540846', '185844420',  '153659394',  '90374658',  '67338774', '114267357',
       '58719682', '126317319',  '14898677',  '177687227', '161993329', '184753009', '193210174',
       '67752315',   '2790210', '187934739',  '179014827',  '56565905', '117787729' ]
#------ craquage -----
(x,y) = fermat(n, 10.)
print("Fermat(n): x =", x, " ; y =", y)
x1 = x-1; y1 = y-1; x1y1 = x1 * y1
s = inversemod(e, x1y1)
print("s =", s)
#------ déchiffrage ------
mdc = dechiffrerRSA(n, d, lc)
shell.write(mdc+"\n","STRING")

#===== 5.3.1/2.2. La clé du coffre
shell.write("----- La clé du coffre\n", "COMMENT")
#------ chiffre inchangé; chiffrage + déchiffrage -----
ms = "Le code secret d'accès à la salle des coffres est ZpX01Wzz2"
lc = chiffrerRSA(n, e, 3, ms)  # k = 3 < kmax
mdc = dechiffrerRSA(n, d, lc)
shell.write(mdc+"\n","STRING")

#============== Facteur de compactage trop grand
shell.write("----- Facteur de compactage trop grand\n", "COMMENT")
#------ chiffre inchangé; chiffrage  + déchiffrage -----
ms = "Le code secret d'accès à la salle des coffres est ZpX01Wzz2"
print("kmax =", math.log(n)/math.log(256))
print("k =", 8)  # > 7
lc = chiffrerRSA(n, e, 8, ms)     # k = 8 trop grand
try:
    mdc = dechiffrerRSA(n, d, lc)
    shell.write(mdc+"\n","STRING")
except: shell.write("Erreur: Facteur de compactage trop grand\n", "COMMENT")
```

```
#=======================================
# Attaques
#=======================================
#====== Déchiffrement par itération
def repdechiffrerRSA(n: int, e: int, o:int, ls0: list) -> str:  # ls0: str list
    ls = [int(s) for s in ls0]
    def degroupe(accu, y):
        if y == 0: return accu
        else:
            q = y // 256; r = y % 256
            accu = [r] + accu
            return degroupe(accu, q)
    def dechiffre(j):
        if j == len(ls): return []
        m = ls[j]
        for i in range(1,o): m = exprapmod(m, e, n)
        return degroupe(dechiffre(j+1), m)
    r = bytes(dechiffre(0))
    return r.decode('utf8')

# recherche d'un exemple
p = 34511; q = 13441; n = p*q
print( "p = %10d %5r" %(p, (isprime(p))));
print( "q = %10d %5r" %(q, (isprime(q))));
print( "n = %10d" %n)                          # n < sqrt(max_int)
p1q1 = (p-1)*(q-1)
for e in range(3, 90000):
    w = 1
    for i in range(1, 20):
        w = (w * e) % p1q1
        if w == 1: print( "OK : n = %10d e = %4d i = %2d %5r" %(n, e, i, (isprime(e))))
#... OK : n =  463862351 e = 25793 i = 12 true  ...

# test de l'exemple
n = 463862351; e = 25793
mu = 100
w = exprapmod(mu, e, n)
i = 1
while w != mu and i < 200:
    w = exprapmod(w, e, n)
    i = i+1
    if w == mu: print( "n = %10d e = %10d i = %2d" %(n, e, i))

# déchiffrement
ms = "J'ai des informations sur le sujet du prochain DS d'info. Bob."
lc = chiffrerRSA(n, e, 3, ms)

mdc = repdechiffrerRSA(n, e, 12, lc)
shell.write(mdc+"\n","STRING")
```

```
#============= Attaque Fermat
shell.write("----- Attaque Fermat\n", "COMMENT")
# ------ chiffre: module de chiffrement------
p = 255949039601; q = 255949044823; n = p * q
print("p = ", p, "\nq = ", q, "\nn = p*q = ", n)    # n = 65509912209240151035623
#.....
e = 27329
#.....
#--------- craquage ---------*)
x, y = fermat(n, 60.)
print("Fermat(n): x =", x, "; y =", y)
x1 = x-1; y1 = y-1; x1y1 = x1 * y1
s = inversemod(e, x1y1)
print("s =", s)
#------ message crypté reçu
ms = "La deuxième partie du prochain DS d'info portera sur les arbres binaires équilibrés. Bob."
lc = chiffrerRSA(n, e, 8, ms)   # k = 8
print("lc = ", lc)
lc =  ['58708423107954916923760', '31436899421999180466155', '444014759554241857403',
       '20107848284631489486860', '58312015784159115214176', '13978316466576225911582',
       '28937690863344110299079', '52763225684222112646755', '27937192474109965023930',
       '22057576210221896807178', '39521329825987178664787', '51726656168921412590539']
#------ déchiffrage ------
mdc = dechiffrerRSA(n, s, lc)
print(mdc)
shell.write(mdc+"\n","STRING")

#================ Attaque Pollard
shell.write("----- AttaquesPollard\n", "COMMENT")
# ------ chiffre: module de chiffrement------
p = 2590689529; q = 25594903958984194711; n = p * q
print("p = ", p, "\nq = ", q, "\nn = p*q = ", n)          # n = 663084496823009987142848811 19
#.....
e = 206977
#.....
#------ craquage ---------
x = pollardpmu(n, 270,2)
print("Pollar p-1: x =", x)   # 2590689529
x = pollardrho(n, 60)
print("Polard rho: x =", x)   # 2590689529
y = n // x
print("x =", x)               # 2590689529
print("y =", y)               # 25594903958984194711

print("x-1 =", x-1, "=", eratosthene(x-1))  # 2^3*3^3*7*59*113*257
# print("y-1 =", y-1, "=", eratosthene(y-1))  # 2*3*5*59*14460397716940223  TROP LONG !!
x1 = x-1; y1 = y-1; x1y1 = x1 * y1
s = inversemod(e, x1y1)
print("s =", s)
#------ message crypté reçu
ms = "La troisième partie du prochain DS d'info portera sur les automates. Bob."
lc = chiffrerRSA(n, e, 10, ms)   # k = 10
print("lc = ", lc)

lc = [ '1146228123974337830308698241 2', '706248904889505872219814698 5',
       '272612860349718267199569968 72', '213817576264180784817612887 18',
       '566926195250498242683701442 03', '408866159142534891892778739 39',
       '133148660649654723673568439', '465321848979391965308630523']
#------ déchiffrage ------
mdc = dechiffrerRSA(n, s, lc)
shell.write(mdc+"\n","STRING")
```

```
#=======================================
# Chiffre plus résistant
#=======================================
# recherche de u, a <= u < b, u premier, u-1 = 2*v où v est premier
def getprimeB(a,b):
    x = 4
    fini = False
    while not fini:
        x = random.randint(a,b)
        if isprime(x):
            x1 = x-1
            x2 = x1 // 2
            if isprime(x2): fini = True
    return x

shell.write("----- Tirage p et q premiers très grands, éloignés, p*q résistant à pollard p-1\n", "COMMENT")
p = getprimeB(                                          10000000000000000000,
                                                  100000000000000000000000000000)
print("p =", p)                                 #   79224892836852166358173375 07
q = getprimeB(                              1000000000000000000000000000000000000,
                      100000000000000000000000000000000000000000000000000000000)
print("q =", q)   #    191778011118097192080954935570995131417434260764509603

# On reprend ces valeurs dans la suite ...
shell.write("----- Chiffre plus résistant\n", "COMMENT")
# ------ chiffre: module de chiffrement------
p = 792248928368521663581733750 7
q = 191778011118097192080954935570995131417434260764509603
n = p * q
print("p =", p)
print("q =", q)
print("n      =", n)      # 151935923792958933521584045253250884439754646703403458323471754162249809799357972 1
print("2^270-1=",2**270-1)# 189713759006418854581978701838234268226797542876185500122247305638564871602071142 3

e = 3554386219
d = inversemod(e, (p-1)*(q-1))
print("e =", e)           # 3554386219
print("d =", d)           # 47121305243802179816001861205722973949125794426712436803962420933682898930898787 5
print("(e*d) % p1q1 =", (e * d) % p1q1)    # vérif -> 1
# ------ craquage ------
# (x,y) = fermat(n, 3600*24)  # abandon
# x = pollardpmu(n, 1000,2)   # abandon
# x = pollardrho(n, 3600*24)  # abandon
# ------ chiffrage -----
ms = "Bravo Alice! Ton nouveau code est super résistant. Oscar est dans les choux ... ."+\
     "Je t'envoie le sujet et le corrigé dans le prochain courrier. Bob."
print("kmax = ", math.log(n)/math.log(256))
print("k = ", 20)  # < 33
lc = chiffrerRSA(n, e, 20, ms)
#------ message chiffré reçu -----
print(lc)
#------ déchiffrage ------
mdc = dechiffrerRSA(n, d, lc)
shell.write(mdc+"\n","STRING")

#=====================================================================
# 5.6. Oscar vient d'acheter un nouvel ordinateur superpuissant ...
#=====================================================================

#=== FIN ===
```
