Distinguer nombres premiers et nombres composés,
et décomposer ces derniers en facteurs premiers,
est un des problèmes les plus importants
et les plus utiles en arithmétique.
C.F. Gauss, *Disquisitiones Arithmeticae*, 1801

CHAPITRE XI

Primalité et factorisation d'entiers

Objectifs

Le théorème fondamental de l'arithmétique garantit que tout entier positif n s'écrit de manière unique comme produit $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ d'un certain nombre $k \geq 0$ de facteurs premiers $p_1 < p_2 < \cdots < p_k$ de multiplicités $e_1, e_2, \ldots, e_k \geq 1$. Étant donné un entier n, trois problèmes pratiques se posent :

(1) Déterminer rapidement si n est premier ou composé.

(2) Si n est premier, en trouver une preuve concise et facilement vérifiable.

(3) Si n est composé, trouver rapidement sa décomposition en facteurs premiers.

Pour les petits entiers ces problèmes sont faciles à résoudre. Par contre pour les grands entiers, déjà à partir de 30 décimales, les méthodes naïves échouent de manière catastrophique. Ce chapitre discutera quelques méthodes plus efficaces. Contrairement à ce que l'on pourrait penser, les trois problèmes sont bien distincts. Il se trouve que le premier admet de solutions efficaces, le deuxième aussi pourvu que l'on sache factoriser $n-1$, tandis que le troisième est en général très difficile.

Sommaire

Voici trois questions préliminaires qui nous serviront de fil conducteur :

Question 0.1. Comment prouver qu'un entier donné n est premier ? On pourrait tester tous les diviseurs possibles, … mais c'est hors de question si n est grand, comme 1 299 808 706 099 639 584 492 326 223 873. Existe-t-il une preuve de primalité qui soit concise et facile à vérifier ? Vous trouverez une réponse au §2.10. Le §4 esquisse une méthode récente et assez spectaculaire, qui résout le problème *théoriquement*.

Question 0.2. Comment prouver qu'un entier donné est composé ? C'est facile, dites-vous, il suffit d'en exhiber un facteur. Certes, de petits facteurs sont faciles à trouver par des essais successifs … mais ce n'est plus praticable pour des facteurs grands. Ainsi pour des cas comme $n = 24! - 1$ on constate que la recherche exhaustive est trop coûteuse. Nous regarderons des critères plus efficaces au §2.

Question 0.3. Après s'être convaincu qu'un entier donné, disons $n = 24! - 1$, est composé de grands facteurs, on revient à la question de factorisation : comment trouver efficacement la décomposition en facteurs premiers ? Ceci peut être une tâche très dure. Au §3 nous présentons une idée simple et amusante, qui permet de trouver des facteurs de taille moyenne, disons entre 10^6 et 10^{12}.

1. Méthodes exhaustives

1.1. Primalité. Commençons par la méthode évidente pour tester la primalité d'un entier :

Algorithme XI.1 Test de primalité par essais exhaustifs (non optimisé)

Entrée: un nombre naturel $n \geq 2$
Sortie: le plus petit facteur premier de n

$r \leftarrow \lfloor \sqrt{n} \rfloor$
pour d **de** 2 **à** r **faire si** $d \mid n$ **alors retourner** d
retourner n

Exercice/M 1.1. Justifier cet algorithme : bien que d parcoure nombres premiers et composés, pourquoi peut-on assurer que le facteur trouvé soit premier ? Que se passe-t-il pour n premier ?

Exercice/P 1.2. Vous pouvez déjà écrire une fonction `bool estPremier( const Integer& n )` qui teste si `n` est un nombre premier par des essais successifs. Veillez tout particulièrement aux cas $n = 0, \pm 1, \pm 2$.

Optimisation. — On peut économiser 50% du temps, en procédant, à partir de $d = 3$, par pas de $+2$. On peut encore économiser 33% du temps en procédant, à partir de $d = 5$, par pas de $+2, +4, +2, +4, \ldots$. D'autres optimisations analogues sont possibles mais de plus en plus complexes et de moins en moins efficaces (le détailler). On développera une optimisation plus systématique avec le crible d'Ératosthène plus bas (§1.4).

Exercice/M 1.3. Quel est le nombre d'itérations dans le meilleur des cas ? dans le pire des cas ? Jusqu'à quel n environ ce test est-il raisonnable ? Peut-on ainsi déterminer la nature de 1 219 326 331 002 895 961 ou de 1 219 326 331 002 895 901 ? (On les analysera dans l'exercice 2.32 et 3.17 plus bas.)

1.2. Factorisation. Regardons ensuite la méthode évidente de factorisation :

Algorithme XI.2 Factorisation par essais exhaustifs (non optimisé)

Entrée: un nombre naturel $n \geq 2$
Sortie: l'unique décomposition $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ en facteurs premiers
$p_1 < p_2 < \cdots < p_k$ avec multiplicités $e_1, e_2, \ldots, e_k \geq 1$

$r \leftarrow \lfloor \sqrt{n} \rfloor$, $p \leftarrow 2$, $e \leftarrow 0$
tant que $p \leq r$ **faire**
  tant que $p \mid n$ **faire** $n \leftarrow n/p$, $e \leftarrow e+1$
  si $e > 0$ **alors** rajouter p^e à la factorisation, puis recalculer $r \leftarrow \lfloor \sqrt{n} \rfloor$, $e \leftarrow 0$
  $p \leftarrow p+1$
fin tant que
si $n > 1$ **alors** rajouter le facteur premier n à la factorisation

Exercice/M 1.4. Justifier cet algorithme : pourquoi ne produit-il que des facteurs premiers ? Pourquoi un éventuel facteur restant $n > 1$ est-il premier ?

Exercice/P 1.5. Vous pouvez déjà écrire une fonction `vector<Integer> factorisation( Integer n )` qui calcule la factorisation $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ et renvoie le vecteur $(p_1, e_1; p_2, e_2; \ldots; p_k, e_k)$. Pour faire joli, vous pouvez ajouter une fonction `void affiche( vector<Integer>& dec )` qui permet d'afficher une décomposition comme `2^3·3^2·5·17`.

Optimisation. — Vous pouvez implémenter les optimisations indiquées en exercice 1.2. Y a-t-il une possibilité de n'effectuer qu'une seule division euclidienne pour tester $p \mid n$ et déduire le quotient n/p le cas échéant ? (Est-ce une optimisation importante ?) Plus bas on optimisera par le crible d'Ératosthène (§1.4), et plus tard on ajoutera le test de primalité de Miller-Rabin (§2.5).

Exercice/M 1.6. Quelle est la complexité en nombre d'itérations de la factorisation par divisions successives : dans le meilleur des cas ? dans le pire des cas ? Jusqu'à quel n environ cette méthode est-elle raisonnable ? Peut-on ainsi factoriser 24! – 1 par exemple ? (On factorisera cet exemple dans l'exercice 3.16.)

Exercice/P 1.7. On appelle *nombre de Mersenne* $M_k = 2^k - 1$ avec $k \geq 2$. Décomposer les nombres $M_2, \ldots, M_{60}$ en facteurs premiers. Que se passe-t-il pour M_{61} ? Expliquer le ralentissement observé. (On reprendra cet exemple dans l'exercice 2.34 plus bas).

1.3. Complexité. Essayons de préciser la complexité asymptotique des deux méthodes exhaustives ci-dessus. Soulignons d'abord qu'il est facile de *poser* un problème de primalité ou de factorisation : pour cela il suffit de présenter le nombre n à analyser, disons en système binaire. Son écriture est ainsi de longueur $\ell = \text{len}(n)$ avec $2^{\ell-1} \leq n < 2^\ell$. Cette longueur est une mesure adéquate pour la complexité du nombre n : c'est la mémoire nécessaire pour le stocker et aussi le temps nécessaire pour le transmettre.

Proposition 1.8. *Soit $c(\ell)$ la complexité en nombre d'itérations dans le pire cas quand on applique les méthodes exhaustives décrites ci-dessus aux entiers de longueur ℓ. Alors $c(\ell)$ est exponentielle en ℓ.*

Exercice/M 1.9. Détailler cette proposition. Après réflexion, il reste tout de même une question sur la répartition des nombres premiers : est-ce que le pire des cas se produit réellement pour une longueur ℓ donnée ? C'est effectivement le cas : le « postulat de Bertrand », démontré par Tchebycheff en 1850, affirme qu'il existe au moins un nombre premier dans l'intervalle $[\![a, 2a]\!]$ quel que soit $a \geq 1$. (Il en existe même beaucoup plus, comme vous pouvez déduire du théorème 1.17 ci-dessous.)

1.4. Le crible d'Ératosthène. Rappelons un des plus anciens théorèmes des mathématiques :

Théorème 1.10. *Il existe une infinité de nombres premiers.*

Vous êtes cordialement invités à redémontrer ce joli résultat. Après ce constat fondamental, considérons la question pratique : comment construire la liste de *tous* les nombres premiers $\leq m$? Ératosthène de Kyrène (environ 275–194 avant notre ère) développa une méthode qui est connue sous le nom de *crible d'Ératosthène*. Dans la version originale on écrit $2, 3, 4, 5, \ldots, n$ puis on raye les multiples de $2, 3, 5, \ldots$. Sur ordinateur ceci occupe trop d'espace. L'algorithme XI.3 ci-dessous en est une variante plus économe :

Algorithme XI.3 Une variante du crible d'Ératosthène

Entrée: la liste $(p_0 = 2, p_1 = 3, \ldots, p_{k-1})$ des k plus petits nombres premiers, avec $k \geq 2$
Sortie: la liste $(p_0 = 2, p_1 = 3, \ldots, p_{k-1}, p_k)$ des $k+1$ plus petits nombres premiers

$p \leftarrow p_{k-1} + 2$, $i \leftarrow 1$
tant que $p_i^2 \leq p$ **faire**
  si $p_i \nmid p$ **alors** $i \leftarrow i+1$ **sinon** $p \leftarrow p+2$, $i \leftarrow 1$
fin tant que
retourner la liste prolongée $(p_0 = 2, p_1 = 3, \ldots, p_{k-1}, p)$

Exercice/P 1.11. Prouver la correction de l'algorithme XI.3 puis l'implémenter en une fonction `void ajoutePremier( vector<int>& liste )`. Peut-on ainsi construire, dans un temps raisonnable, la liste des nombres premiers jusqu'à 10^6 ? jusqu'à 10^7 ? jusqu'à 10^8 ? jusqu'à 10^9 ?

Exercice 1.12. Pour $n \in \mathbb{N}$ on note $\pi(n)$ le nombre des entiers premiers dans l'intervalle $[\![1, n]\!]$. Écrire un programme qui lit n au clavier et qui affiche $\pi(n)$. Dans la mesure du possible vérifier les valeurs du tableau suivant (dans lequel au moins une erreur s'est glissée).

k	$\pi(10^k)$	k	$\pi(10^k)$	k	$\pi(10^k)$	k	$\pi(10^k)$
1	4	6	78 498	11	4 118 054 813	16	279 238 341 033 925
2	25	7	664 579	12	37 607 912 018	17	2 623 557 157 654 233
3	168	8	5 761 455	13	346 065 536 839	18	24 739 954 287 740 860
4	1 129	9	50 847 534	14	3 204 941 750 802	19	234 057 667 276 344 607
5	9 592	10	455 052 511	15	29 844 570 422 669	20	2 220 819 602 560 918 840

1.5. Factorisation limitée. Dans un programme on procède typiquement comme suit : on fixe une borne m et construit, une fois pour toute, la liste des petits nombres premiers $p_0, \ldots, p_k \leq m$. Selon le tableau ci-dessus, un choix entre 10^7 et 10^8 semble raisonnable, donc le type `int` suffira.

Exercice/M 1.13. En reprenant les algorithmes XI.1 et XI.2, expliquer pourquoi il est avantageux de parcourir seulement la liste $p_0, \ldots, p_k$. Vérifier que le test de primalité reste correct pour tout $n \leq m^2$. De même, la factorisation est complète si le facteur restant satisfait $n \leq m^2$. Si $n > m^2$ on peut au moins garantir que n n'a pas de petits facteurs, mais on ne peut pas conclure que n soit premier.

Exercice/P 1.14. On se contente, dans un premier temps, d'une factorisation limitée $n = p_{i_1}^{e_1} p_{i_2}^{e_2} \cdots p_{i_\ell}^{e_\ell} \cdot \hat{n}$ de sorte que le facteur restant $\hat{n}$ n'ait pas de diviseurs $\leq m$. L'implémenter en une fonction

```
vector<Integer> factorisation( Integer& n, const vector<int>& premiers )
```

qui extrait les petits premiers stockés dans la liste `premiers`. Ici la fonction remplace n par le facteur restant $\hat{n}$: celui-ci vaut 1 si la factorisation est complète, et > 1 si la factorisation laisse un facteur de nature indéterminée. (Justifier le mode de passage des deux paramètres.)

Optimisation. — Plus bas, dans l'exercice 2.33, on ajoutera le test de primalité selon Miller-Rabin. Ceci sert à compléter la factorisation dans les cas ou le facteur restant $\hat{n}$ est premier.

Exercice/P 1.15. Donner les factorisations limitées de $n!+1$, pour $n \in [\![1,100]\!]$ disons, en indiquant les cas qui laissent un facteur de nature indéterminée. Si vous voulez vous pouvez regarder d'autres familles d'exemples comme $b \cdot n! + c$ ou $b^n + c$ avec $b \geq 2$ et $c \in \mathbb{Z}$ fixés. La deuxième famille inclut en particulier les nombres de Mersenne $2^n - 1$ et les nombres de Fermat $2^{2^n} + 1$.

1.6. Le théorème des nombres premiers. Pour trouver la valeur exacte de $\pi(n)$ il n'y a que le comptage fastidieux, plus ou moins optimisé. Deux questions évidentes se posent : peut-on approcher $\pi(n)$ par une fonction simple ? Quel est son comportement asymptotique ?

En regardant les tableaux des nombres premiers jusqu'à 10^6, qui venaient d'être publiés entre 1770 et 1811, Gauss conjectura que $\pi(n) \sim \frac{n}{\ln n}$. À la même époque Legendre proposa l'approximation $\frac{n}{\ln n - 1}$, ce qui correspond mieux aux valeurs $\pi(n)$ pour $10^4 \leq n \leq 10^{20}$ (les comparer). Bien entendu, le comportement asymptotique de ces deux approximations est le même. Basé sur les idées fondamentales de Riemann (1859), la conjecture de Gauss et Legendre fut finalement démontrée, indépendamment, par Hadamard et de la Vallée Poussin (1896) :

Théorème 1.16 (théorème des nombres premiers, version qualitative)**.** *On a l'équivalence asymptotique* $\pi(n) \sim \frac{n}{\ln n}$*, c'est-à-dire le quotient* $\frac{\pi(n)}{n/\ln n}$ *converge vers* 1 *pour* $n \to \infty$*. Autrement dit, la proportion des nombres premiers parmi les nombres dans l'intervalle* $[\![1,n]\!]$ *est à peu près* $\frac{1}{\ln n}$*.* □

La version suivante donne un encadrement étonnamment précis, dû à J. Rosser et L. Schoenfeld, *Approximate formulas for some functions of prime numbers*, Illinois Journal of Mathematics 6 (1962) 64–94.

Théorème 1.17 (théorème des nombres premiers, version quantitative)**.** *Pour* $n \geq 59$ *on a l'encadrement*

$$\frac{n}{\ln n}\left(1+\frac{1}{2\ln n}\right) \quad < \quad \pi(n) \quad < \quad \frac{n}{\ln n}\left(1+\frac{3}{2\ln n}\right). \tag{1}$$

Pour mieux apprécier ce résultat, on comparera avec profit le comptage exact avec $\frac{n}{\ln n}$ et l'encadrement. Tandis que tout comptage s'arrête forcément assez tôt, l'encadrement est valable éternellement !

Afin d'illustrer la fascination que provoque le théorème des nombres premiers, citons le résumé donné par Don Zagier dans son article "The first 50 million primes", Mathematical Intelligencer (1977) 1-19 :

> There are two facts about the distribution of prime numbers of which I hope to convince you so overwhelmingly that they will be permanently engraved in your hearts. The first is that, despite their simple definition and role as the building blocks of the natural numbers, the prime numbers (...) grow like weeds among the natural numbers, seeming to obey no other law than that of chance, and nobody can predict where the next one will sprout. The second fact is even more astonishing, for it states just the opposite : that the prime numbers exhibit stunning regularity, that there are laws governing their behaviour, and that they obey these laws with almost military precision.

Exercice/M 1.18. Expliquer pourquoi l'intervalle $[\![n, n+k[\![$ contient à peu près $k/\ln n$ nombres premiers. Quand on choisit un grand entier de manière aléatoire, quelle est la probabilité que l'on tombe sur un premier ? La formulation est volontairement provocatrice ; lui donner un sens mathématique précis.

Exercice/M 1.19. L'encadrement (1) du théorème 1.17 est-il suffisamment précis pour détecter la faute de frappe dans le tableau de l'exercice 1.12 ?

Exercice/M 1.20. L'encadrement (1) permet de déduire le postulat de Bertrand. Voyez-vous comment ? Peut-on obtenir ce résultat à partir de la version qualitative $\pi(n) \sim \frac{n}{\ln n}$ seulement ?

2. Le critère probabiliste selon Miller-Rabin

La situation. — Étant donné un nombre naturel n, la factorisation limitée (§1.5) permet d'extraire efficacement les petits facteurs premiers (s'il y en a). Dans certains cas favorables, on arrive ainsi à une factorisation complète. Sinon, il faut tôt ou tard abandonner la recherche exhaustive. Au moins on assure ainsi que le facteur restant n'a plus de petits facteurs.

Le problème. — Il nous reste à déterminer si un entier n donné, sans petits facteurs, est premier ou composé. À première vue la situation semble désespérée. Fort heureusement le critère de Miller-Rabin, développé dans la suite, permet un test efficace, couramment utilisé dans la pratique. Il s'agit d'ailleurs d'une belle application de la structure du groupe $\mathbb{Z}_n^\times$ (chapitre IX, §1).

Comment s'y prendre ? — En principe on pourrait tout de suite énoncer le critère de Miller-Rabin (le lemme 2.16 et le théorème 2.20 plus bas) et passer directement à l'implémentation (§2.8). Mais une telle démarche ne serait ni motivée ni motivante, et l'origine de l'énoncé resterait obscur. Il est plus naturel de développer ce critère en cinq petites étapes, amplement illustrées d'exemples. Cette démarche retrace le développement historique et logique de ce critère fort utile.

2.1. Éléments non inversibles dans $\mathbb{Z}_n^*$. Rappelons qu'un entier $n \geq 2$ est premier si et seulement si $\mathbb{Z}_n^* = \mathbb{Z}_n \smallsetminus \{0\}$ est un groupe (d'ordre $n-1$) pour la multiplication. Par contraposé : étant donné n, il suffit d'exhiber un élément nun nul $x \in \mathbb{Z}_n^*$ mais non inversible pour prouver que n est composé. Est-il possible d'en trouver un assez rapidement ? Malheureusement cette idée se révèle peu praticable : il y a simplement trop peu de tels éléments !

Exercice/M 2.1. Supposons que n est composé, avec factorisation $n = p_1^{e_1} \cdots p_k^{e_k}$. Vérifier que la probabilité qu'un élément $x \in \mathbb{Z}_n$ choisi au hasard soit inversible vaut $\frac{\varphi(n)}{n} = \prod_{i=1}^{i=k}(1 - \frac{1}{p_i})$. Si n n'a pas de petits facteurs, alors la probabilité de tomber sur un élément non inversible est proche de 0. Expliquer, en choisissant n, pourquoi elle peut même être arbitrairement petite.

Remarque 2.2. Trouver un élément $x \in \mathbb{Z}_n^*$ non inversible revient à trouver un facteur de n : en représentant x par $\tilde{x} \in \mathbb{Z}$, on peut rapidement calculer $d = \operatorname{pgcd}(n, \tilde{x})$, qui est un facteur de n vérifiant $1 < d < n$. Ainsi tomber sur un élément non inversible de $\mathbb{Z}_n^*$ est aussi probable que deviner un facteur de n.

2.2. Le test de Fermat. Le petit théorème de Fermat (voir le chapitre X, §1) affirme que pour n premier tout élément $x \in \mathbb{Z}_n^\times$ vérifie $x^{n-1} = 1$. Ceci peut servir à tester la primalité d'un nombre n donné :

Proposition 2.3. *Si $x \in \mathbb{Z}_n^*$ vérifie $x^{n-1} \neq 1$, alors n est composé. Dans ce cas on dit que x est un* témoin de décomposabilité *de n, ou aussi que x* témoigne contre la primalité *de n. À noter qu'un tel témoin prouve que n est composé sans pour autant donner une quelconque information sur les facteurs de n.* □

Remarque 2.4. Le critère que $x^{n-1} = 1$ est nécessaire pour la primalité de n mais non suffisante. Soulignons donc que $x^{n-1} = 1$ ne prouve en rien que n soit premier. Par exemple, pour $n = 15$ on trouve que $2^{14} \equiv 4 \pmod{15}$, donc 2 est un témoin. Par contre $4^{14} \equiv 1 \pmod{15}$, donc 4 n'est pas un témoin. Il s'agit d'une asymétrie fondamentale : ce genre de test peut prouver que n est composé, mais il ne peut pas prouver que n est premier.

La question cruciale pour toute application pratique est la suivante : étant donné un nombre composé n, peut-on rapidement trouver un témoin $x \in \mathbb{Z}_n^*$?

Remarque 2.5 (le principe probabiliste)**.** On ne sait pas prédire quels éléments vont témoigner. On choisit donc $x_1, x_2, \ldots, x_k \in \mathbb{Z}_n^*$ au hasard et les teste un par un. Si l'on trouve un témoin, ceci prouve que n est composé. Sinon, n est possiblement premier mais on ne peut être sûr.

Pour que cette approche soit intéressante, il faut assurer un taux de réussite assez grand. Notre question devient donc : si l'on choisit a au hasard, quelle est la probabilité que a témoigne contre la primalité de n ? Déjà les éléments non inversibles ne satisfont pas $x^{n-1} = 1$, mais ce cas est trop peu probable si n n'a pas de petit facteur (voir plus haut). Heureusement, le test de Fermat augmente considérablement nos chances ! Expérimentez avec le programme `temoins.cc` pour vous convaincre que ce test peut être assez efficace dans certains cas.

2.3. Nombres de Carmichael. Après la première euphorie, voici la mauvaise nouvelle :

Remarque 2.6 (Carmichael)**.** Il existe des nombres composés n tel que $a^n \equiv a \pmod{n}$ pour tout $a \in \mathbb{Z}$. En particulier, tout élément inversible $x \in \mathbb{Z}_n^\times$ vérifie $x^{n-1} = 1$. On appelle un tel n un *nombre de Carmichael*. Les plus petits exemples sont 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341, 41041, … On sait depuis 1994 qu'il en existe une infinité.

Exercice/M 2.7. Établissons que les nombres de Carmichael sont la seule obstruction pour le test de Fermat. Si n est composé mais non un nombre de Carmichael, alors l'ensemble $\{x \in \mathbb{Z}_n^\times \mid x^{n-1} = 1\}$ forme un sous-groupe d'indice ≥ 2 dans $\mathbb{Z}_n^\times$. Par conséquent, le test de Fermat trouve un témoin avec probabilité d'échec $< \frac{1}{2}$. En itérant ce test k fois, la probabilité d'échec devient $< 2^{-k}$, donc arbitrairement petite.

Exercice/M 2.8. Vérifier que $n = 561 = 3 \cdot 11 \cdot 17$ est effectivement un nombre de Carmichael : montrer que le groupe $\mathbb{Z}_{561}^\times \cong \mathbb{Z}_3^\times \times \mathbb{Z}_{11}^\times \times \mathbb{Z}_{17}^\times$ est d'ordre 320, et que tout $x \in \mathbb{Z}_n^\times$ vérifie $x^{80} = 1$. Par conséquent $x^{560} = 1$ et $x^{561} = x$ pour tout $x \in \mathbb{Z}_{561}^\times$, et cette dernière égalité tient même pour tout $x \in \mathbb{Z}_{561}$. Même calcul pour les nombres suivants : $1105 = 5 \cdot 13 \cdot 17$, puis $1729 = 7 \cdot 13 \cdot 19$, puis $2821 = 7 \cdot 13 \cdot 31$, …

Exercice/M 2.9. Soit $n = pqr$ avec $p = 6k+1$ et $q = 12k+1$ et $r = 18k+1$ premiers. (Pour $k = 1$ par exemple on obtient $n = 1729$.) Montrer que n est un nombre de Carmichael : tout $x \in \mathbb{Z}_n^\times$ vérifie $x^{36k} = 1$ et $36k$ divise $n-1$. En supposant qu'il existe une infinité de tels nombres, déduire que la probabilité pour trouver un témoin avec le test de Fermat peut être arbitrairement petite.

Exercice/M 2.10. Supposons que $n = p_1 \cdots p_k$ avec des facteurs premiers $p_1 < \cdots < p_k$ de sorte que $p_i - 1$ divise $n-1$ pour tout i. Vérifier que n est un nombre de Carmichael. Réciproquement, tout nombre de Carmichael est forcément de cette forme-ci, impair, et composé d'au moins trois facteurs premiers. *Indication.* — On appliquera le théorème chinois et le fait que le groupe $\mathbb{Z}_{p^e}^\times$ est cyclique d'ordre $(p-1)p^{e-1}$.

2.4. L'astuce de la racine carrée. Le test de Fermat a ses défauts, mais il est trop beau pour être abandonné. Essayons donc de l'optimiser. Le lemme suivant explicite l'observation clé :

Lemme 2.11. *Soit p un nombre premier impair. Alors -1 est le seul élément d'ordre 2 dans $\mathbb{Z}_p^\times$.*

DÉMONSTRATION. Effectivement, -1 est d'ordre 2. Réciproquement, si x est d'ordre 2, alors $x^2 = 1$. Ceci implique $0 = x^2 - 1 = (x-1)(x+1)$, ce qui n'est possible que pour $x - 1 = 0$ ou $x + 1 = 0$, donc $x = \pm 1$. Comme $+1$ est d'ordre 1, on conclut que -1 est le seul élément d'ordre 2. □

Exercice/M 2.12. Si n est composé le groupe $\mathbb{Z}_n^\times$ peut contenir plusieurs éléments d'ordre 2. Dans $\mathbb{Z}_8$, par exemple, les éléments $3, 5, 7$ sont tous d'ordre 2.

Regardons $n = pq$ avec deux premiers $p > q > 2$. Par le théorème chinois nous avons un isomorphisme d'anneaux $\psi\colon \mathbb{Z}_p \times \mathbb{Z}_q \xrightarrow{\sim} \mathbb{Z}_n$. Dans $\mathbb{Z}_p^\times \times \mathbb{Z}_q^\times$ les éléments d'ordre 2 sont $(-1,-1)$, $(+1,-1)$ et $(-1,+1)$. Dans $\mathbb{Z}_n^\times$ nous avons donc également trois éléments d'ordre 2 : outre la solution standard $\psi(-1,-1) = -1$ nous trouvons deux autres éléments, $\psi(+1,-1)$ et $\psi(-1,+1)$. Les expliciter pour $n = 15$.

Exercice/M 2.13. D'après le théorème 1.13 énoncé au chapitre IX, le groupe $\mathbb{Z}_{p^e}^\times$ est cyclique d'ordre $(p-1)p^{e-1}$ pour tout premier $p \geq 3$ et exposant $e \geq 1$. En déduire que -1 est le seul élément d'ordre 2 dans $\mathbb{Z}_{p^e}^\times$. Pour $n = p_1^{e_1} \cdots p_k^{e_k}$ montrer qu'il existe exactement $2^k - 1$ éléments d'ordre 2 dans $\mathbb{Z}_n^\times$. Comment les expliciter ?

Le lemme précédent mène directement à un test de primalité un peu plus raffiné. Afin de l'illustrer vous pouvez analyser des exemples avec le programme `temoins.cc`.

Proposition 2.14. *Si n est un nombre premier impair, alors tout $x \in \mathbb{Z}_n^*$ vérifie $x^{\frac{n-1}{2}} = \pm 1$.* □

Exemple 2.15 (Carmichael, suite). Comme remarqué plus haut, $n = 561$ est un nombre de Carmichael : les 320 éléments inversibles $x \in \mathbb{Z}_n^\times$ satisfont tous au test de Fermat $x^{n-1} = 1$. Par contre, seuls 160 satisfont au critère $x^{\frac{n-1}{2}} = \pm 1$. Le test $x^{\frac{n-1}{2}} = \pm 1$ est donc plus fin que le test $x^{n-1} = 1$. (Voir le programme `temoins.cc`.)

Hélas, ce test raffiné n'est pas infaillible : dans le cas $n = 1729$ tous les éléments inversibles $x \in \mathbb{Z}_n^\times$ satisfont non seulement au test $x^{n-1} = 1$, mais aussi au test $x^{\frac{n-1}{2}} = \pm 1$. Comment sauver la situation ? En l'occurrence, remarquons que l'exposant $\frac{n-1}{2} = 864$ est un nombre pair. Si $x^{864} = +1$, on peut donc tester de plus si $x^{432} = \pm 1$: ce n'est rien d'autre que notre lemme, appliqué une deuxième fois. Si l'on trouve

$x^{432} = +1$, on peut tester si $x^{216} = \pm 1$, et ainsi de suite. On vient de découvrir un test encore plus fin : c'est la méthode de Miller-Rabin !

2.5. Le test de Miller-Rabin. Vers la fin des années 1970, G. Miller puis M. Rabin ont proposé un test de primalité très puissant. Il découle directement de notre développement précédent :

Lemme 2.16. *Soit $n \geq 3$ un entier impair. On décompose $n-1 = 2^e q$ avec $e \geq 1$ et q impair. Si n est premier, alors tout $x \in \mathbb{Z}_n^*$ satisfait ou $x^q = 1$ ou bien il existe $k \in [\![0, e[\![$ tel que $x^{2^k q} = -1$.*

Définition 2.17. On dit que $x \in \mathbb{Z}_n^*$ passe le test de Miller-Rabin si $x^q = 1$ ou $x^{2^k q} = -1$ pour un $k \in [\![0, e[\![$. Dans ce cas on dit aussi que n est *pseudo-premier* à base x. (À noter que ceci ne veut pas dire que n soit premier.) Réciproquement, si $x \in \mathbb{Z}_n^*$ ne passe pas le test de Miller-Rabin, alors on dit que x est un *témoin de décomposabilité* de n au sens de Miller-Rabin, ou aussi que *x témoigne contre la primalité* de n.

Exercice/M 2.18. Montrer le lemme et expliquer en quoi il est un raffinement du test de Fermat. Vérifier que l'algorithme suivant est une traduction fidèle du critère de Miller-Rabin :

Algorithme XI.4 Test de pseudo-primalité selon Miller-Rabin

Entrée: un entier impair $n \geq 5$ et un entier $x \in [\![2, n-2]\!]$.
Sortie: le message « pseudo-premier » ou « composé »

décomposer $n-1 = 2^e q$ avec $e \geq 1$ et q impair	// divisions itérées
calculer $y \leftarrow x^q \bmod n$	// puissance dichotomique
si $y = 1$ **alors retourner** « pseudo-premier »	// x passe tous les tests car $x^q \equiv 1$.
pour k **de** 1 **à** e **faire**	
si $y = n-1$ **alors retourner** « pseudo-premier »	// premier cas : $x^{2^{k-1}q} \equiv -1$, ça passe
$y \leftarrow y^2 \bmod n$	// après ce calcul on a $y = x^{2^k q} \bmod n$
si $y = 1$ **alors retourner** « composé »	// second cas : $x^{2^{k-1}q} \not\equiv \pm 1$ mais $x^{2^k q} \equiv 1$
fin pour	
retourner « composé »	// x ne passe même pas le test de Fermat.

Exercice/M 2.19. Rassurons-nous que la complexité algorithmique du critère de Miller-Rabin est tout à fait raisonnable. Soit n un entier de longueur $\ell = \mathrm{len}(n)$ en base 2. Rappelons d'abord qu'une multiplication modulo n est de complexité $O(\ell^2)$ avec la méthode scolaire, voire $O^+(\ell)$ avec une méthode plus sophistiquée. Vérifier que le test de Miller-Rabin nécessite moins de 2ℓ multiplications pour calculer x^q et ses carrés successifs. Au total, tester si $x \in [\![2, n-2]\!]$ satisfait au critère de Miller-Rabin est de complexité $O(\ell^3)$ voire $O^+(\ell^2)$.

2.6. Probabilité d'erreur. A priori le critère de Miller-Rabin est plus fin mais aussi un peu plus complexe que les critères précédents. Son intérêt réside dans le beau résultat suivant, montré en 1980 indépendamment par M. Rabin et L. Monier. Il garantit une grande probabilité de réussite :

Théorème 2.20. *Soit $n \geq 3$ impair. Si n est premier, alors tout $x \in \mathbb{Z}_n^*$ satisfait au critère de Miller-Rabin. Si n est composé, alors il y a au moins $\frac{3}{4}(n-1)$ témoins $x \in \mathbb{Z}_n^*$ au sens de Miller-Rabin.* □

Exemple 2.21 (Carmichael, suite et fin). Pour $n = 1729 = 7 \cdot 13 \cdot 19$ le groupe $\mathbb{Z}_n^\times$ est d'ordre $6 \cdot 12 \cdot 18 = 1296$. Tous les éléments $x \in \mathbb{Z}_n^\times$ satisfont aux tests $x^{n-1} = 1$ et $x^{\frac{n-1}{2}} = \pm 1$. Par contre, seuls 162 passent le test de Miller-Rabin. Autrement dit, il existe 1566 témoins, soit plus de 90%, conformément au théorème. (Voir `temoins.cc`.)

Corollaire 2.22. *Pour tester si un nombre n est composé on choisit $x_1, x_2, \ldots, x_k \in \mathbb{Z}_n^*$ au hasard et effectue le test de Miller-Rabin pour chacune de ces bases. Si n est premier, l'algorithme répondra toujours que n est pseudo-premier à base $x_1, x_2, \ldots, x_k$. Si par contre n est composé, alors l'algorithme répondra pseudo-premier à base $x_1, x_2, \ldots, x_k$ avec une probabilité $\leq 4^{-k}$; autrement dit, avec une probabilité $\geq 1 - 4^{-k}$ on trouve un témoin x_i qui prouve que n est composé.* □

Remarque 2.23. Supposons que nous soumettons un nombre n donné à k tests aléatoires de Miller-Rabin, qui répondent k fois « pseudo-premier ». Que peut-on en déduire ? On lit souvent la formulation suivante : « Le nombre n est premier avec probabilité $\geq 1 - 4^{-k}$. » Strictement parlant, cette phrase n'a aucun sens :

le nombre n est ou premier ou composé, c'est une propriété de n et non une question de probabilité. L'affirmation prend un sens quand on la reformule plus précisément, comme dans le corollaire précédent : c'est l'algorithme qui est probabiliste, et il reste une certaine probabilité d'erreur.

Exercice/M 2.24. Dans le test de Miller-Rabin la seule erreur possible est une conclusion erronée « probablement premier » alors que n est composé. Cette erreur se produit avec une probabilité inférieure à 4^{-k}. Feriez-vous confiance en un résultat qui est correct avec une probabilité d'erreur $\leq 4^{-k}$ pour $k = 10$? pour $k = 30$? pour $k = 100$? Comparer la probabilité 4^{-100} avec la probabilité de gagner plusieurs fois consécutives au loto. Contempler la conclusion d'Émile Borel (*Les probabilités et la vie*, 1943) : « Un phénomène dont la probabilité est 10^{-50} ne se produira donc jamais, ou de moins ne sera jamais observé. » Qu'en pensez-vous ?

Exercice/M 2.25. Tous nos calculs sont effectués sur des machines réelles, donc imparfaites : la probabilité ε que les circuits électriques produisent une erreur est très petite mais elle est certainement non nulle (ne mentionnons que mouvement thermique, radiation extérieure, effets quantiques). Par conséquent, même le résultat d'un algorithme parfaitement déterministe peut être erroné avec une probabilité d'erreur $\varepsilon > 0$ quand on l'exécute sur une machine imparfaite. Qu'estimez-vous, pour quel k a-t-on $4^{-k} \approx \varepsilon$? Sous cet angle rediscuter la sûreté du test de Miller-Rabin.

2.7. Un test optimisé. Le test de Miller-Rabin remédie à tous les défauts du test de Fermat, en particulier il reconnaît aisément les nombres de Carmichael comme composés. On verra dans ce paragraphe qu'il arrive même à les factoriser !

Comme remarqué au §2.1, tout élément non inversible $\bar{x} \in \mathbb{Z}_n^*$ nous fait cadeau d'un facteur de n, simplement en calculant $\operatorname{pgcd}(n,x)$. Dans ce sens l'observation suivante peut être utile :

Lemme 2.26. *Si $y \in \mathbb{Z}_n$ vérifie $y^2 = 1$ mais $y \neq \pm 1$, alors les éléments $y \pm 1 \in \mathbb{Z}_n^*$ sont non inversibles.*

DÉMONSTRATION. Comme on a vu dans le lemme 2.11, trouver un tel y prouve que n est composé. Supposons $n = pq$ pour simplifier. D'après le théorème chinois nous disposons d'un isomorphisme d'anneaux $\phi\colon \mathbb{Z}_n \xrightarrow{\sim} \mathbb{Z}_p \times \mathbb{Z}_q$. Notre élément y s'envoie donc sur $\phi(y) = (+1,-1)$ ou $\phi(y) = (-1,+1)$. C'est une information précieuse : $y+1$ correspond à $(2,0)$ ou $(0,2)$, et $y-1$ correspond à $(0,-2)$ ou $(-2,0)$. Ce sont donc des éléments non nuls mais non inversibles. □

Exercice/M 2.27. Compléter la démonstration précédente et en déduire l'algorithme XI.5 ci-dessous. Le principe est le même que le test de Miller-Rabin (algorithme XI.4). L'aspect nouveau est qu'on essaie d'exploiter d'éventuels éléments non inversibles qui peuvent apparaître au cours des calculs.

Algorithme XI.5 Test de primalité selon Miller-Rabin, version étendue

Entrée: un entier impair $n = 2^e q + 1$ et un entier $x \in [\![2, n-2]\!]$.

Sortie: le message « pseudo-premier » ou « composé » ou un facteur non trivial de n

$d \leftarrow \operatorname{pgcd}(n,x)$	// rapide avec l'algorithme d'Euclide.
si $d > 1$ **alors retourner** d	// x non inversible donne un facteur de n.
$y \leftarrow x^q \bmod n$	// rapide avec la puissance dichotomique.
si $y = 1$ **alors retourner** « pseudo-premier »	// x passe tous les tests car $x^q \equiv 1$.
pour k **de** 1 **à** e **faire**	
si $y = n-1$ **alors retourner** « pseudo-premier »	// Premier cas : x passe le test de Miller-Rabin.
$z \leftarrow y$, $y \leftarrow y^2 \bmod n$	// On remplace y par y^2 mais on stocke la racine.
si $y = 1$ **alors retourner** $\operatorname{pgcd}(n, z+1)$	// Second cas : ici $z \not\equiv \pm 1$ mais $z^2 \equiv 1$.
fin pour	
retourner « composé »	// x ne passe même pas le test de Fermat.

L'algorithme ci-dessus trouve un facteur de n chaque fois que x passe le test de Fermat modulo n mais non le test de Miller-Rabin. Ainsi les nombres de Carmichael, considérés plus haut comme le pire cas pour le test de Fermat, sont particulièrement faciles à factoriser. (Expliquer pourquoi.)

Deux autres cas sont possibles : Si x ne passe pas le test de Fermat, alors n est composé, mais le témoin x ne nous indique malheureusement aucun facteur de n. (C'est le cas le plus fréquent.) Si x passe le test de Miller-Rabin (y compris Fermat) on soupçonne que x est premier. On peut itérer le test pour être plus sûr.

2.8. Implémentation du test. La « longue marche » précédente avait pour but de motiver et d'élucider le critère de Miller-Rabin. Nous pouvons enfin passer à son implémentation.

Exercice/P 2.28. Écrire une fonction `bool estPseudoPremier( Integer n, Integer x )` qui teste si n est pseudo-premier à base x au sens de Miller-Rabin. Adapter le mode de passage de n et x aux besoins de votre fonction.

Exercice/P 2.29. Implémenter la version étendue du test de Miller-Rabin en une fonction
`bool estPseudoPremier( Integer n, Integer x, Integer& facteur )`
qui garde un éventuel facteur de n trouvé lors des calculs. Adapter le mode de passage de n et x aux besoins de votre fonction. Comme premier test et application, factoriser les nombres de Carmichael donnés dans la remarque 2.6. Ces nombres se révèlent plutôt sympathiques, après tout...

Exercice/P 2.30. Implémenter une fonction `bool estProbPremier( Integer n, int k=50 )` qui choisit successivement $x_1, x_2, \ldots, x_k \in [\![2, n-2]\!]$ de manière aléatoire et effectue le test de Miller-Rabin pour chacune de ces bases. *Indication.* — Quant à la primalité attraper d'abord le cas $n < 0$, puis $n = 0, 1, 2, 3$, puis les nombres pairs. Adapter le passage du paramètre `n`, le cas échéant, et expliquez l'utilisation du paramètre `k`, initialisé par défaut.

Remarque 2.31. On ne peut pas implémenter littéralement un algorithme probabiliste, car on ne dispose pas de source de nombres vraiment aléatoires. Dans la pratique tous les générateurs produisent de nombres « pseudo-aléatoires », c'est-à-dire une suite déterministe qui a l'aire d'être « suffisamment aléatoire ». Pour les sceptiques, l'usage des nombres aléatoires est un sujet délicat, voir Knuth [**8**], tome 3. Dans la pratique vous pouvez utiliser la fonction suivante, qui fait appel à un générateur de nombres pseudo-aléatoires, fourni par la bibliothèque `GMP`, auquel on fera confiance :

```
Integer random( const Integer& min, const Integer& max )
{ // construire un nombre aléatoire entre min et max inclus
  static gmp_randclass generateur(gmp_randinit_default);
  return generateur.get_z_range(max-min+1) + min; }
```

Exercice/P 2.32. Reprenez les deux entiers de l'exercice 1.3 et déterminez leur nature. Expliquer l'intérêt de la méthode de Miller-Rabin vis-à-vis la méthode naïve de l'exercice 1.3.

Exercice/P 2.33. Dans votre fonction qui réalise la factorisation limitée (exercice 1.14), ajoutez le test de primalité dans le cas d'un facteur restant $\hat{n} > 1$. Ceci permet de compléter la factorisation si $\hat{n}$ est (probablement) premier. Sinon, le facteur restant est laissé comme tel.

Exercice 2.34. Montrer que le nombre de Mersenne $M_{61} = 2^{61} - 1$ est premier, au moins très probablement. Factorisez quelques nombres M_k plus grands en indiquant lesquels laissent un facteur composé sans petits facteurs.
Remarque. — Pour les nombres de Mersenne $M_n = 2^n - 1$ il existe des tests plus spécifiques, analogues au test de Pépin pour les nombres de Fermat. Par exemple, si $2^n - 1$ est premier, alors n est premier. La réciproque est fausse : déjà $2^{11} - 1$ n'est pas premier. Si p est premier, alors tout facteur premier $q \mid M_p$ est de la forme $q = kp + 1$. Vous pouvez le vérifier empiriquement ou essayez de le montrer.

2.9. Comment trouver un nombre premier ? On se propose ici de produire de très grands nombres premiers, on dirait de manière industrielle. Voici une méthode possible :

Exercice/P 2.35. Écrire une fonction `Integer prochainPremier( Integer n )` qui trouve le plus petit premier $p > n$. Trouver ainsi le prochain nombre premier après 10^k pour $k = 2, \ldots, 200$.

Exercice/M 2.36. On pourrait également choisir un nombre premier de manière aléatoire dans l'intervalle $[\![a, b]\!]$. Évidemment il faut éviter que l'intervalle soit trop petit : s'il ne contient pas de nombre premier, inutile d'insister. Pour cela vous pouvez déduire une minoration de $\pi(b) - \pi(a-1)$ du théorème 1.17. En particulier le postulat de Bertrand affirme que l'intervalle $[\![a, 2a]\!]$ contient toujours de nombres premiers.

Exercice/M 2.37. Expliquer pourquoi l'algorithme XI.6 finira par trouver un nombre pseudo-premier comme promis. En s'inspirant de l'exercice 1.18, donner une estimation du nombre d'essais nécessaire.

Exercice/M 2.38. Supposons que pour chaque p on effectue des tests itérés de Miller-Rabin, de sorte que la probabilité d'erreur Prob(p pseudo-premier | p composé) soit ε. Supposons que l'algorithme XI.6 renvoie un pseudo-premier p. Quelle est la probabilité d'erreur de cet algorithme ? *Indication.* — La réponse n'est pas ε mais $\varepsilon \ln p$ environ ! Si vous

Algorithme XI.6 Engendrer un nombre pseudo-premier aléatoire entre a et b

Entrée: deux nombres naturels $a < b$ tels que $\pi(a-1) < \pi(b)$
Sortie: un nombre aléatoire $p \in [\![a,b]\!]$ qui soit pseudo-premier

$a' \leftarrow \lceil \frac{a-1}{2} \rceil$, $b' \leftarrow \lfloor \frac{b-1}{2} \rfloor$
boucle
  choisir $p' \in [\![a',b']\!]$ de manière aléatoire et poser $p \leftarrow 2p'+1$
  tester la pseudo-primalité de p par des tests itérés de Miller-Rabin
  si p est pseudo-premier **alors retourner** p
fin boucle

vous y connaissez en théorie des probabilités, on peut calculer Prob(p composé | p pseudo-premier) par les probabilités conditionnelles et la formule de Bayes. C'est une jolie application de la théorie élémentaire des probabilités. Comme toujours l'interprétation soigneuse des probabilités fait partie de l'honnêteté scientifique.

2.10. Comment prouver un nombre premier ? Rappelons que les deux observations clé de la méthode de Miller-Rabin sont les suivantes :

(1) Il est facile de déterminer si $x \in \mathbb{Z}_n^*$ est un témoin de décomposabilité.

(2) Si n est composé il est très probable de trouver un témoin $x \in \mathbb{Z}_n^*$.

Ceci correspond à deux types de problèmes bien distincts : *vérifier* une preuve et *trouver* une preuve. (Vous savez de votre propre expérience mathématique que trouver est en général plus difficile que vérifier.) Dans l'approche de Miller-Rabin le premier problème est résolu de manière déterministe, le deuxième de manière probabiliste.

Soulignons à nouveau que cette méthode permet de *prouver* qu'un nombre n est composé, mais en cas d'échec seulement de *soupçonner* que n est premier. L'asymétrie provient du fait qu'un seul témoin suffit pour montrer que n est composé, mais sans aucun témoin on ne peut pas conclure avec certitude.

Pour être sûr que n est premier nous voudrions également disposer d'une preuve. C'est tout à fait possible et l'idée est même très simple : il suffit de produire un élément d'ordre $n-1$ dans $\mathbb{Z}_n^\times$. Cette question a été résolue au §2.3 du chapitre IX. En voici le critère efficace :

Lemme 2.39. *On considère un entier n pour lequel on suppose connue la décomposition en facteurs premiers $n-1 = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$. Si un élément $g \in \mathbb{Z}_n^\times$ vérifie $g^{n-1} = 1$ alors son ordre divise $n-1$. Si de plus $g^{(n-1)/p_i} \neq 1$ pour tout $i = 1,2,\ldots,k$ alors l'ordre de g est exactement $n-1$. Dans ce cas $\mathbb{Z}_n^\times = \mathbb{Z}_n \smallsetminus \{0\}$, donc $\mathbb{Z}_n$ est un corps et n est un nombre premier.* □

La connaissance de la décomposition $n-1 = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ est une hypothèse délicate, car la factorisation peut être difficile en général. Heureusement elle est faisable dans beaucoup d'exemples.

Exercice/P 2.40. Comme avant on stocke la décomposition $n-1 = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ sous forme d'un vecteur $(p_1,e_1;p_2,e_2;\ldots;p_k,e_k)$. Écrire une fonction `bool estRacinePrimitive(g,n,decomp)` qui teste si g est d'ordre $n-1$ dans $\mathbb{Z}_n^\times$. Écrire une fonction `Integer racinePrimitive(n,decomp)` qui trouve la plus petite racine primitive modulo n, supposé premier, par des essais successifs. Choisir les modes de passage adéquats. Expliquer pourquoi on passe la décomposition de $n-1$ comme un paramètre. Comme alternative, serait-il une bonne idée de factoriser $n-1$ dans la fonction `estRacinePrimitive` ?

Exercice/P 2.41. Montrer que $n = 27!+1$ est premier en exhibant une racine primitive. Même exercice avec $37!+1$ et $73!+1$ puis $77!+1$. Plus généralement vous pouvez analyser $n!+1$ à condition d'effectuer au préalable un test de primalité.

Définition 2.42. En guise de résumé, précisons ce qu'il faut pour prouver la primalité d'un entier n. On appelle $C = (n,g;p_1,e_1;\ldots;p_k,e_k) \in \mathbb{N}^{2k+2}$ un *certificat de primalité* si

– on a $n-1 = p_1^{e_1} \cdots p_k^{e_k}$ avec $p_1 < \cdots < p_k$ premiers et $e_1,\ldots,e_k \geq 1$,

– le nombre g vérifie $g^{n-1} \equiv 1$ ainsi que $g^{(n-1)/p_i} \not\equiv 1$ modulo n pour $i = 1,\ldots,k$.

Dans ce cas on dit aussi que C *certifie* la primalité de n.

Exemple 2.43. Le plus petit certificat est $(2,1)$: la factorisation de $2-1=1$ est le produit vide, et $g=1$ est un élément d'ordre 1, comme exigé. Voici quelques certificats plus intéressants :

– $(3,2;2,1)$ certifie la primalité de 3. (Le vérifier. Y a-t-il d'autres certificats ?)
– $(17,3;2,4)$ certifie la primalité de 17. (Le vérifier. Y a-t-il d'autres certificats ?)
– $(103,5;2,1;3,1;17,1)$ certifie la primalité de 103 (le vérifier).
– $(n,5;2,21;3,14;17,9;103,3)$ certifie la primalité de
$n = 1\,299\,808\,706\,099\,639\,584\,492\,326\,223\,873$ (le vérifier).

Le dernier exemple soulève la question pratique : comment vérifier efficacement un certificat donné $(n,x;p_1,e_1;\ldots;p_k,e_k)$? D'abord il est facile à vérifier que $p_1^{e_1}p_2^{e_2}\cdots p_k^{e_k} = n-1$. Avec la puissance dichotomique on peut ensuite vérifier que $g^{n-1} \equiv 1$ et $g^{(n-1)/p_i} \not\equiv 1$ modulo n. Pour terminer la preuve de la primalité de n il ne reste qu'à prouver la primalité de $p_1,\ldots,p_k$. C'est simple : on exige des certificats !

Définition 2.44. Une *preuve de primalité* pour $n \in \mathbb{N}$ est une liste de certificats $C_1, C_2, \ldots, C_l$ pour des nombres premiers $n_1 < n_2 < \cdots < n_l = n$ de sorte que tout nombre premier utilisé dans un certificat C_j soit lui-même certifié par un certificat antérieur C_i ($i < j$).

Exemple 2.45. Les certificats de l'exemple précédent fournissent une preuve de primalité pour l'entier $n = 1\,299\,808\,706\,099\,639\,584\,492\,326\,223\,873$. Vérifier puis contempler ce bel exploit.

Exercice 2.46. Expliquer comment implémenter une fonction qui *vérifie* une preuve de primalité, puis une fonction qui *construit* une preuve de primalité. Expliquer pourquoi la vérification est facile, alors que la construction d'une preuve peut être difficile. À noter en particulier que la construction d'une preuve de primalité de n nécessite la factorisation de $n-1$, tâche qui peut être très coûteuse. Dans la pratique on ne lancera une telle recherche qu'après s'être convaincu par un test de Miller-Rabin. Si vous ne craignez pas de longs projets de programmation, vous pouvez implémenter toutes ces fonctions en C++.

3. Factorisation par la méthode ρ de Pollard

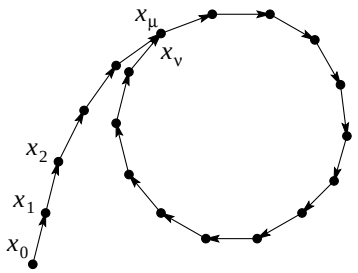
Dans ce qui précède nous avons discuté deux méthodes de grande importance pratique : la factorisation limitée (§1.5) et le critère de primalité selon Miller-Rabin (§2.5). Leur combinaison permet de factoriser tout nombre entier qui soit composé de petits facteurs premiers et au plus un grand facteur premier. (Rappeler pourquoi.) Reste à traiter le cas où n n'a pas de petits facteurs mais le test de Miller-Rabin prouve que n est composé. Avouons qu'en général cette tâche peut être extrêmement difficile !

La méthode ρ de Pollard décrite dans la suite est un algorithme probabiliste pour trouver des facteurs de taille moyenne, disons entre 10^6 et 10^{12}. Il existe des méthodes plus performantes, mais la méthode ρ a le mérite d'être très facile à implémenter. Si vous êtes impatients, vous pouvez commencer par l'implémentation et tester sa performance pratique (§3). Pour comprendre son succès, par contre, il nous faut quelques préparatifs (§3.1-3.3).

3.1. Détection de cycles selon Floyd. L'idée de base est d'analyser des suites récurrentes. Soit E un ensemble fini et $f\colon E \to E$ une application quelconque. À partir d'une valeur initiale $x_0 \in E$ on définit la suite $(x_k)_{k\in\mathbb{N}}$ par $x_{k+1} = f(x_k)$ pour tout $k \in \mathbb{N}$.

Exercice/M 3.1. Montrer qu'il existe $\mu < \nu$ tels que $x_\mu = x_\nu$. On pose $\lambda = \nu - \mu$. En déduire que la suite (x_k) devient ultimement périodique, c'est-à-dire $x_k = x_{k+\lambda}$ pour tout $k \geq \mu$.

Si l'on choisit μ et ν minimaux on dit que μ est *l'indice d'entrée* dans la période et λ est la *longueur* de la période. Cette situation est représentée par le dessin à droite. Il explique pourquoi la méthode de Pollard est aussi nommée la méthode ρ.



Comment déterminer μ et ν algorithmiquement ? La manière évidente est de stocker toute la suite $x_0,\ldots,x_{\nu-1}$ puis de chercher x_ν dans cette liste. Si x_ν n'y appartient pas encore, on le rajoute et continue avec $\nu \leftarrow \nu+1$. L'inconvénient de cette approche est, évidemment, qu'il faut stocker puis parcourir une liste éventuellement très longue. Afin de faciliter la recherche d'indices $k < k'$ avec $x_k = x_{k'}$, on se contentera d'une solution non minimale :

Exercice/M 3.2. Il existe $\ell > 0$ tel que $x_\ell = x_{2\ell}$. Si l'on choisit ℓ minimal alors $\mu \leq \ell \leq \nu \leq 2\ell$. En particulier ℓ et ν sont de même ordre de grandeur, à un facteur 2 près.

Il est facile de déterminer la valeur minimale ℓ : en commençant par $x_0 = y_0$ on parcourt la suite $x_{k+1} = f(x_k)$ et en même temps la la suite « à double vitesse » $y_{k+1} = f(f(y_k))$, et on teste successivement pour $\ell = 1,2,3,\dots$ si $x_\ell = y_\ell$. Ainsi on n'a plus besoin de stocker toute la liste, les deux valeurs courantes x_ℓ et y_ℓ suffisent !

3.2. Le paradoxe des anniversaires. En général il est difficile de prédire la longueur ℓ en fonction de f et de la valeur initiale x_0, mais elle est assez facile à mesurer sur des exemples donnés. Quand la suite passera-t-elle « en boucle » ? Il est clair que plus E est grand, plus ν et ℓ ont tendance à être grands. Essayons de donner un sens quantitatif à cette heuristique.

Exercice/M 3.3. Combien y a-t-il de fonctions $f : E \to E$? Quelle est la proportion des fonctions f avec $\nu(f) \geq 1$? avec $\nu(f) \geq 2$? avec $\nu(f) \geq 3$? En général, déterminer la proportion $p_{n,k} = \frac{\#\{f \mid \nu(f) \geq k\}}{\#\{f : E \to E\}}$.

Exercice/P 3.4 (le paradoxe des anniversaires)**.** En supposant que les anniversaires sont équidistribués sur les 365 jours de l'année, combien de personnes faut-il pour avoir deux de même jour d'anniversaire avec une probabilité $\geq \frac{1}{2}$? Écrire un programme qui calcule ce nombre, étonnamment petit. (Sans effort supplémentaire vous pouvez généraliser les paramètres $n = 365$ et $q = \frac{1}{2}$ dans ce calcul.)

Voici la version asymptotique de ce phénomène :

Proposition 3.5. *Soit E un ensemble fini de cardinal n. Quand on choisit $f : E \to E$ au hasard, la valeur moyenne de $\nu(f)$ est donnée par $\overline{\nu}_n = \sum_{k=1}^{n} p_{n,k}$. Pour $n \to \infty$ on a $\overline{\nu}_n \sim \sqrt{\frac{\pi}{2} n}$.* □

Autrement dit, pour un choix aléatoire de f on s'attend à ce que la suite récurrente $x_{k+1} = f(x_k)$ boucle à un indice ν d'ordre $\sqrt{n}$ environ. Il en est de même pour l'indice ℓ, de même ordre que ν mais plus facile à déterminer dans la pratique. Pour une preuve simple que $\overline{\nu}_n \in O(\sqrt{n})$ voir Gathen&Gerhard [**11**], théorème 19.5. Pour un développement détaillé, voir Knuth [**8**], vol. 2, §3.1, exercices 11 et 12 avec leur solution à la fin du tome, qui renvoient au vol. 1, §1.2.11.3 pour le calcul asymptotique.

3.3. Polynômes et fonctions polynomiales. Pour l'ensemble E on prendra dans la suite l'anneau $\mathbb{Z}_n$ et pour applications $f : \mathbb{Z}_n \to \mathbb{Z}_n$ nous regardons des fonctions polynomiales, comme $f(x) = x^2 + 1$.

Afin d'éviter toute confusion, il sera utile de distinguer *polynômes* $P \in \mathbb{Z}_n[X]$ et *fonctions polynomiales* $f : \mathbb{Z}_n \to \mathbb{Z}_n$. Rappelons que tout élément de $\mathbb{Z}_n[X]$ s'écrit comme $\sum_k a_k X^k$ avec coefficients $a_k \in \mathbb{Z}_n$ et qu'on a l'égalité $\sum_k a_k X^k = \sum_k b_k X^k$ si et seulement si $a_k = b_k$ pour tout k. Tout polynôme $P = \sum_k a_k X^k$ définit une fonction $\Phi_P : \mathbb{Z}_n \to \mathbb{Z}_n$ par $\Phi_P(x) = P(x) = \sum_k a_k x^k$, appelée la *fonction polynomiale* associée à P. Évidemment $P = Q$ entraîne $\Phi_P = \Phi_Q$, mais la réciproque est fausse : on peut avoir $\sum_k a_k x^k = \sum_k b_k x^k$ pour tout $x \in \mathbb{Z}_n$ sans que $a_k = b_k$ pour tout k.

Proposition 3.6. *Si p est premier, alors toute fonction $f : \mathbb{Z}_p \to \mathbb{Z}_p$ est polynomiale : il existe un polynôme $P \in \mathbb{Z}_p[X]$ et un seul de degré $< p$ tel que $f = \Phi_P$, à savoir l'interpolant de Lagrange $P = \sum_{a \in \mathbb{Z}_p} f(a) \prod_{b \in \mathbb{Z}_p \smallsetminus \{a\}} \frac{X-b}{a-b}$. L'application $\Phi : \mathbb{Z}_p[X] \to \mathrm{Fonc}(\mathbb{Z}_p, \mathbb{Z}_n)$, $P \mapsto \Phi_P$ est donc surjective. Il s'agit d'un homomorphisme d'anneaux qui a pour noyau l'idéal $(X^p - X)$.*

Exercice/M 3.7. Prouver cette proposition.

Supposons désormais que $n = p_1 p_2 \cdots p_m$ avec des nombres premiers $p_1, p_2, \dots, p_m$. Pour simplifier nous supposons que $p_1 < p_2 < \cdots < p_m$. (Nous négligeons ici le cas de facteurs multiples.) D'après le théorème des restes chinois, l'anneau $\mathbb{Z}_n$ se décompose en $\mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_m}$. Étant donné des fonctions $f_i : \mathbb{Z}_{p_i} \to \mathbb{Z}_{p_i}$ on peut les assembler en une fonction $f : \mathbb{Z}_n \to \mathbb{Z}_n$ d'après le schéma suivant :

$$\begin{array}{ccc} \mathbb{Z}_n & \xrightarrow{\ f\ } & \mathbb{Z}_n \\ \cong\downarrow & & \downarrow\cong \\ \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_m} & \xrightarrow{f_1 \times \cdots \times f_m} & \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_m} \end{array}$$

Proposition 3.8. *On dit qu'une fonction $f : \mathbb{Z}_n \to \mathbb{Z}_n$ est* décomposable *(en $f_1, \dots, f_m$) si elle peut être construite comme dans le diagramme précédent. Toute fonction polynomiale $f = \Phi_P$ est décomposable. Réciproquement toute fonction décomposable $f : \mathbb{Z}_n \to \mathbb{Z}_n$ est polynomiale.*

Exercice/M 3.9. Prouver cette proposition. Étant donné une fonction $f = \Phi_P$ comment la décomposer ? Réciproquement, étant donné f décomposable, comment construire un polynôme $P \in \mathbb{Z}_n[X]$ tel que $f = \Phi_P$? Expliquer pourquoi les fonctions décomposables ne forment en général qu'une petite fraction des applications $f : \mathbb{Z}_n \to \mathbb{Z}_n$.

3.4. L'heuristique de Pollard. Après ces préparations, nous allons les appliquer à la factorisation. Les arguments précédents se résument dans l'interprétation probabiliste suivante :

Corollaire 3.10. *Pour p premier fixons un degré $d \geq p$ et choisissons un polynôme $P \in \mathbb{Z}_p[X]$ de degré $< d$ de manière équiprobable, chacun avec probabilité p^{-d}. Alors la fonction polynomiale associée Φ_P tombe sur n'importe quelle fonction $\mathbb{Z}_p \to \mathbb{Z}_p$ avec la même probabilité p^{-p}. On s'attend donc au paradoxe des anniversaires : une suite récurrente $x_{k+1} = P(x_k)$ va boucler après $\sqrt{p}$ itérations environ.* □

Corollaire 3.11. *Supposons que $n = p_1 p_2 \cdots p_m$ est composé de premiers $p_1 < p_2 < \cdots < p_m$ et que l'on choisit aléatoirement un polynôme $P \in \mathbb{Z}_n[X]$ de degré $< n$. Ensuite on décompose la fonction polynomiale associée $f = \Phi_P$ en $f_i : \mathbb{Z}_{p_i} \to \mathbb{Z}_{p_i}$. Alors f_i est équidistribuée sur les $p_i^{p_i}$ fonctions possibles. On s'attend donc à nouveau au paradoxe des anniversaires : une suite récurrente $x_{k+1} = P(x_k)$ va boucler modulo p_1 après $\sqrt{p_1}$ itérations environ, modulo p_2 après $\sqrt{p_2}$ itérations environ, … et finalement modulo p_m après $\sqrt{p_m}$ itérations environ.* □

Exemple 3.12. Voici un exemple qui est trop petit pour être réaliste, mais suffisamment grand pour illustrer le principe. Regardons l'itération de $f : \mathbb{Z}_{221} \to \mathbb{Z}_{221}$ avec $f(x) = x^2 + 1$ et $x_0 = 1$:

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
x_k	1	2	5	26	14	197	135	104	209	145	31	78	118	2	…

Ici $\mu = 1$ et $\lambda = 12$. On constate que $\ell = 12$ est le plus petit indice positif tel que $x_\ell = x_{2\ell}$. Comme $221 = 13 \cdot 17$, regardons les suites dans les quotients. Modulo 17 la suite correspond à l'itération de $\mathbb{Z}_{17} \to \mathbb{Z}_{17}$, $x \mapsto x^2 + 1$ avec valeur initiale $x_0 = 1$:

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
x_k	1	2	5	9	14	10	16	2	5	9	14	10	16	2	…

Ici $\mu = 1$ et $\lambda = 6$ ainsi que $\ell = 6$. Modulo 13 nous obtenons :

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
x_k	1	2	5	0	1	2	5	0	1	2	5	0	1	2	…

Ici $\mu = 0$ et $\lambda = 4$ ainsi que $\ell = 4$. (Est-ce un hasard que $12 = \mathrm{ppcm}(4,6)$?)

Remarque 3.13. Avec beaucoup de bienveillance, on peut considérer l'exemple comme une illustration de la proposition 3.5. Bien sûr l'application $x \mapsto x^2 + 1$ n'a rien d'aléatoire, et les coïncidences $12 \approx \sqrt{221}$ et $6 \approx \sqrt{17}$ et $4 \approx \sqrt{13}$ peuvent sembler peu convaincantes. Soulignons que l'argument statistique ne décrit que le comportement *en moyenne*, et c'est ce comportement qui est souvent observé dans les exemples. (Des exceptions seront inévitables.) Pour vous en convaincre, vous pouvez tester d'autres fonctions polynomiales et d'autres entiers composés n avec votre implémentation plus bas.

3.5. L'algorithme de Pollard. Après l'heuristique, formulons l'algorithme probabiliste qui en découle :

Algorithme XI.7 Factorisation selon la méthode ρ de Pollard

Entrée: un entier $n > 1$, une valeur initiale $x \in \mathbb{Z}_n$ et un polynôme $P \in \mathbb{Z}_n[X]$

Sortie: un diviseur $d > 1$ de n (possiblement $d = n$)

```
y ← x                          // x et y représentent x_k et x_{2k} de la suite récurrente
répéter
   x ← P(x) mod n              // On passe de x_k à x_{k+1}
   y ← P(P(y)) mod n           // On passe de x_{2k} à x_{2k+2}
   d ← pgcd(x − y, n)          // On espère que x ≡ y modulo p_1 mais non modulo n
jusqu'à d > 1
retourner d
```

Exercice/M 3.14. Montrer que l'algorithme XI.7 s'arrête puis qu'il est correct. Expliquer son comportement en moyenne : quel résultat obtient-on et avec combien d'itérations ? En quoi cette méthode est-elle intéressante vis-à-vis la factorisation par essais successifs ?

3.6. Implémentation et tests empiriques. Jusqu'ici notre développement repose sur un solide argument statistique qui suppose que nous choisissons la fonction $f = \Phi_P$ aléatoirement. Or, il est très coûteux, voire impossible, de stocker puis d'évaluer des polynômes de très grand degré (d'ordre 10^9 disons). On choisira donc des polynômes de petit degré, typiquement quadratique, en espérant que cet échantillon sera suffisamment représentatif pour reproduire le paradoxe des anniversaires et son asymptotique décrit dans la proposition 3.5. (D'ailleurs, les fonctions linéaires ne conviennent pas. Pourquoi ?)

Exercice/P 3.15. Implémenter l'algorithme de Pollard pour $f(x) = x^2 + a$ en une fonction

```
Integer pollard( const Integer& n, Integer x, int a, int max=500000 )
```

Justifier l'usage d'un paramètre `max` pour majorer le nombre d'itérations dans le pire cas.

- Tester votre fonction sur $n = 221$, $x = 1$, $a = 1$ pour vérifier qu'elle produit bien les résultats de l'exemple 3.12 plus haut : on trouve le facteur $\text{pgcd}(x_4 - x_8, 221) = 13$.
- Tester votre fonction sur l'exemple $n = 143$ et $x = 1$. Vérifier que $a = 1$ donne le diviseur trivial $d = 143$. Que donnent les paramètres $a = 2$ et $a = 3$?

Ces expériences indiquent que l'on ne peut pas prédire le bon choix du polynôme f. Si un premier essai échoue, alors on recommence avec d'autres paramètres.

Exercice/P 3.16. Pour résumer le développement de ce chapitre, écrire un programme qui permet d'entrer un nombre entier puis de le factoriser (au moins partiellement) :

(1) Mettez au point une factorisation limitée pour trouver les petits facteurs (§1.5).

(2) Déterminer la nature d'un éventuel facteur restant par le test de Miller-Rabin (§2.8).

(3) Le cas échéant, appliquer la méthode ρ de Pollard. En cas d'échec il faut éventuellement réessayer. En cas de succès assurer que votre factorisation est complète, et réitérer si nécessaire.

Essayez ainsi de décomposer $2\cdot 19!+1$, $4\cdot 19!-1$, $6\cdot 19!+1$, $20!+1$, $24!-1$ en facteurs premiers. Indiquez les méthodes utilisées et les facteurs ainsi trouvés. Si la méthode en question dépend de paramètres à choisir, comme la factorisation limitée ou la méthode de Pollard, explicitez-les.

Exercice 3.17. Décomposer les deux exemples de l'exercice 1.3, donnés au début du chapitre.

Exercice 3.18. Pour $k = 20, \ldots, 40$ essayer de décomposer $k! + 1$ en facteurs premiers.

3.7. Conclusion. Le principal argument en faveur de l'approche heuristique de Pollard est, bien sûr, le succès pratique amplement illustré. Notre heuristique explique aussi pourquoi la méthode de Pollard échouera probablement dans le cas où n est composé de grands facteurs premiers : le nombre d'itérations nécessaire devient trop grand. C'est le cas par exemple pour

$$38! + 1 = 14\,029\,308\,060\,317\,546\,154\,181 \cdot 37\,280\,713\,718\,589\,679\,646\,221.$$

Pour savoir d'avantage sur des méthodes plus performantes, qui sont capables d'exhiber, par exemple, la factorisation précédente, on consultera avec profit l'excellent livre de R. Crandall et C. Pomerance [**15**]. Il existe plusieurs sites web consacrés à la chasse aux factorisations. La page de R. Brent est un bon endroit pour commencer, `web.comlab.ox.ac.uk/oucl/work/richard.brent/factors.html`

4. Le critère déterministe selon Agrawal-Kayal-Saxena

Le problème de primalité. — On veut analyser un entier n, donné sous forme d'un développement binaire de longueur $\ell = \text{len}(n)$. Il s'agit de déterminer si n est premier ou composé. On a déjà vu que le test exhaustif est d'une complexité exponentielle (cf. la proposition 1.8). Il est prohibitif pour des grands entiers, disons à partir d'une longueur $\ell \approx 60$, soit 20 décimales environ.

Approche probabiliste. — Le test de Miller-Rabin est d'une complexité polynomiale $O(\ell^3)$ voire $O^+(\ell^2)$, comme esquissé en exercice 2.19. De nombreux exemples témoignent de son efficacité, et d'un point de vue pratique on pourrait considérer le problème comme résolu. Cependant, sa nature probabiliste

laisse une certaine probabilité d'erreur (arbitrairement petite, mais non nulle). Il en est de même d'ailleurs pour le test de Solovay-Strassen, que nous avons discuté dans le projet X.

Approche déterministe. — Existe-t-il une méthode qui résolve le problème de primalité de manière *universelle*, c'est-à-dire pour n'importe quel $n \in \mathbb{N}$, et qui soit à la fois *déterministe* et de complexité *polynomiale* ? Cette question est restée longtemps ouverte et hantait les théoriciens. Assez récemment, en juillet 2002, la réponse spectaculaire fut établie par M. Agrawal, N. Kayal et N. Saxena :

Théorème 4.1. *Il existe un algorithme qui, pour tout entier n donné, détermine si n est premier ou composé, et dont le temps d'exécution est polynomial en la longueur* $\mathrm{len}(n)$ *du nombre n.*

Comme l'algorithme AKS est assez élémentaire, nous nous proposons ici de l'implémenter. Si ce sujet vous donne envie d'expérimenter, les considérations pratiques qui suivent vous permettront d'écrire votre propre programme AKS et d'entreprendre une exploration empirique. Quant aux mathématiques sous-jacentes, pour la plupart également élémentaires, nous nous contentons de présenter les idées essentielles. L'algorithme n'a sans doute pas encore trouvé sa forme optimale, et de nombreuses améliorations sont actuellement développées. Vous trouverez de plus amples explications dans la littérature :

(1) L'article de Agrawal-Kayal-Saxena est paru dans *Annals of Mathematics*, 160 (2004) 781–793, `www.math.princeton.edu/~annals/issues/2004/Sept2004/Agrawal.pdf`

(2) Un développement élémentaire a été rédigé par Michiel Smid, détaillant toutes les preuves que nous admettons, `www.scs.carleton.ca/~michiel/primes.ps.gz`.

4.1. Une belle caractérisation des nombres premiers. Nous reprenons à nouveau le petit théorème de Fermat : si n est premier, alors $a^n = a$ modulo (n) pour tout $a \in \mathbb{Z}$. Cette observation se généralise à un anneau quelconque, pourvu que l'on adapte convenablement l'énoncé :

Lemme 4.2. *Soient x, y deux éléments d'un anneau commutatif A. Alors pour tout $n \in \mathbb{N}$ on a la formule binomiale $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$. Si de plus n est premier, alors n divise le coefficient binomial $\binom{n}{k}$ pour tout $k = 1, \dots, n-1$, et ainsi $(x+y)^n \equiv x^n + y^n$ modulo (n).* □

Exercice/M 4.3. Montrer ce lemme et en déduire le petit théorème de Fermat $a^p \equiv a \pmod{p}$ par récurrence sur a.

On peut appliquer ce lemme comme un critère de primalité. Le test AKS est basé sur la jolie observation qu'en prenant l'anneau $\mathbb{Z}[X]$ le critère nécessaire est aussi suffisant :

Proposition 4.4. *Soit $n \geq 2$ et $a \in \mathbb{Z}$. Si n est premier alors $(X+a)^n \equiv X^n + a$ dans $\mathbb{Z}_n[X]$. Réciproquement, si $\mathrm{pgcd}(a, n) = 1$ et $(X+a)^n \equiv X^n + a$ dans $\mathbb{Z}_n[X]$, alors n est premier.* □

Exercice/M 4.5. Essayez de montrer cette proposition. *Indication.* — La première partie est claire. Quant à la deuxième, supposons que $n = p^e q$ avec p premier et $p \nmid q$. Déduire de $(X+a)^n \equiv X^n + a$ modulo (p) que $q = 1$. Déduire de $(X+a)^{p^e} \equiv X^{p^e} + a$ modulo (p^e) que $e = 1$. (Pour une solution voir Smid.)

C'est une très belle caractérisation de la primalité de n. Quant au calcul pratique, ce critère reste malheureusement impraticable pour n grand, à cause de l'immense longueur du polynôme $(X+a)^n$. Déjà pour $n \approx 10^{20}$, ce polynôme dépasse la capacité de tout ordinateur.

4.2. Polynômes cycliques. Afin de rendre le critère précédent calculable, on réduit modulo $(X^r - 1)$, ce qui veut dire que nous posons $X^r = 1$ dans tous nos calculs. Grâce à cette relation on peut immédiatement réduire tout monôme X^s au monôme $X^{s \bmod r}$. On calcule ainsi avec des exposants modulo r, d'où le nom polynômes « cycliques ». L'avantage de cette approche est que les calculs deviennent faisables, même assez efficaces (ce qui sera à discuter après implémentation). L'inconvénient est que la caractérisation précédente se transforme en un critère nécessaire qui n'est plus forcément suffisant :

Corollaire 4.6. *Si n est premier alors $(X+a)^n = X^n + a$ modulo $(n, X^r - 1)$. Par contraposé : si $(X+a)^n \not\equiv X^n + a$ modulo $(n, X^r - 1)$ alors n est composé.*

Exemple 4.7. Comme remarqué plus haut, $n = 1729$ est un nombre de Carmichael, c'est-à-dire tout $a \in \mathbb{Z}_n$ vérifie $a^n = a$. Ici le test AKS se révèle plus fin : on a $(X+2)^n \equiv 819X^2 + X + 912$ modulo $(n, X^4 - 1)$, ce qui ne coïncide pas avec $X^n + 2 \equiv X + 2$. À noter aussi que pour $X \mapsto 1$ on obtient exactement le test

de Fermat ; ici $(1+2)^n \equiv 819+1+912 \equiv 3$, comme il faut. (Cet exemple sera facile à calculer avec l'implémentation développée dans la suite.)

Convention. — On supposera dans la suite que $\operatorname{pgcd}(n,r)=1$. Ceci est tout à fait légitime : on envisage de tester la primalité de n, et un $\operatorname{pgcd}(n,r)>1$ donnerait immédiatement un facteur de n.

4.3. Une implémentation basique. Commençons par fixer les types des données. Pour les calculs dans $\mathbb{Z}$ nous auront besoin, comme d'habitude, d'un type `Integer` modélisant les grands entiers. Pour les exposants des monômes, les indices et les longueurs des vecteurs, par contre, les petits entiers suffiront. (Pourquoi ?) De toute façon, notre compilateur exige que les vecteurs soient indexés par un type primitif, comme `int` ou `unsigned int`. Pour fixer notre choix, nous appelons ce type `SmInt` pour *small integer* :

```
#include <vector>                // définir la classe générique vector
#include <gmpxx.h>               // déclarer les grands entiers de GMP
typedef mpz_class       Integer; // grands entiers (coefficients)
typedef unsigned int    SmInt;   // petits entiers (exposants)
typedef vector<Integer> Poly;    // polynôme stocké comme vecteur
```

Convention. — Tout polynôme P modulo (n,X^r-1) peut être représenté, de manière unique, par $a_0+a_1X+\cdots+a_{r-1}X^{r-1}$ avec coefficients $a_i \in [\![0,n[\![$. Sous cette forme-là il est stocké comme un vecteur $(a_0,a_1,\ldots,a_{r-1})$ de taille r exactement.

Exercice/P 4.8. Écrire une fonction qui effectue la multiplication « cyclique » :

```
Poly mult_cyclique( const Poly& p, const Poly& q, const Integer& n )
```

Indication. — La multiplication peut se réaliser, comme d'habitude, par deux boucles imbriquées. Veillez toutefois à ne calculer qu'avec des polynômes modulo (X^r-1) et de ne jamais dépasser la longueur r. Pensez aussi à réduire les coefficients modulo n à la fin.

Exercice/P 4.9. Comme toujours, la puissance dichotomique s'impose pour effectuer le calcul de $(X+a)^n$ modulo (n,X^r-1). Écrire une telle fonction

```
Poly puissance_cyclique( Poly base, Integer exp, const Integer& n )
```

qui emploie la multiplication cyclique implémentée dans l'exercice précédent.

Exercice/M 4.10. Soulignons à quel point les choix faits pour cette implémentation sont judicieux. Que se passerait-il si l'on ne réduisait pas systématiquement modulo n ? Que se passerait-il si l'on ne réduisait pas systématiquement modulo X^r-1 ? Expliquer heuristiquement pourquoi les réductions modulo n dans chaque coefficient *et* modulo X^r-1 pour tout monôme garantissent des données de taille bornée. (On analysera la complexité du calcul plus bas.)

Exercice/P 4.11. Écrire finalement une fonction qui teste si $P(X)^n = P(X^n)$ modulo (n,X^r-1) :

```
bool testFermat( const Poly& p, const Integer& n )
```

Indication. — Pour calculer $Q(X) := P(X)^n$ modulo (n,X^r-1) on se servira de la puissance dichotomique ci-dessus. Pour $P(X^n)$ il y a une astuce : comme r et n sont premiers entre eux, $P(X^n)$ mod (X^r-1) n'est qu'une permutation des coefficients (l'expliciter). On compare $Q(X)$ et $P(X^n)$ via cette permutation d'indices. Déterminer r et $t = n \bmod r$ peut se réaliser ainsi :

```
SmInt r= p.size();               // la taille du polynôme en question
SmInt t= Integer(n%r).get_ui();  // transformer Integer en unsigned int
```

Exercice/P 4.12. Implémenter une fonction qui teste plus spécifiquement si $(X-a)^n \equiv X^n-a$ modulo (n,X^r-1). C'est le test d'AKS proprement dit, avec paramètres $r \in \mathbb{N}$ et $a \in \mathbb{Z}$:

```
bool temoinAKS( const Integer& n, SmInt r, Integer a )
```

Tester vos fonctions sur beaucoup d'exemples. Si n est premier, votre fonction répond-elle correctement ? Si n est composé, trouve-t-on rapidement un témoin ? Que se passe-t-il pour les nombres de Carmichael ? Donner plus d'exemples illustrant que le test AKS est plus fin que le test de Fermat. Statistiquement, est-ce un critère intéressant dans la pratique ?

Remarque 4.13. Pour r grand, le facteur limitant est la multiplication cyclique des polynômes modulo (n,X^r-1). Notre implémentation nécessite r^2 opérations dans $\mathbb{Z}_n$. Afin de profiter de la multiplication rapide d'entiers, on peut transformer les polynômes $P(X)$ et $Q(X)$ en deux entiers $\hat{P}=P(B)$ et $\hat{Q}=Q(B)$ avec $B=2^k$ suffisamment grand. On

calcule $\hat{R} = \hat{P}\hat{Q}$ avec une multiplication rapide, et en retrouve le polynôme $R(X)$ tel que $R(B) = \hat{R}$. Ce procédé permet de calculer le produit $R = PQ$ avec une complexité presque linéaire en r.

4.4. Analyse de complexité. Essayons de déterminer la complexité du test AKS, c'est-à-dire du test de Fermat étendu à l'anneau $\mathbb{Z}_n[X]/(X^r-1)$, implémenté comme ci-dessus :

Proposition 4.14. *Soit n un entier de longueur $\ell = \mathrm{len}(n)$ en base 2. Le coût pour effectuer un test de Fermat dans $\mathbb{Z}_n[X]/(X^r-1)$ avec l'implémentation ci-dessus est d'ordre $r^2\ell^3$. Avec une multiplication optimisée on peut descendre jusqu'à une complexité presque d'ordre $r\ell^2$.*

ESQUISSE DE PREUVE. La multiplication de deux éléments dans $\mathbb{Z}_n$ est de complexité $O(\ell^2)$ avec la méthode scolaire. Optimal serait $O^+(\ell)$ avec des méthodes plus sophistiquées.

La multiplication de deux polynômes cycliques dans $\mathbb{Z}_n[X]/(X^r-1)$ nécessite r^2 multiplications dans $\mathbb{Z}_n$ avec la méthode scolaire, utilisée ci-dessus. Optimal serait $O^+(r)$. On arrive donc a une complexité de $O(r^2)O(\ell^2)$, voire $O^+(r)O^+(\ell)$ avec une implémentation optimisée.

La puissance dichotomique nécessite au plus 2ℓ multiplications pour calculer $P(X)^n$ modulo (n, X^r-1). Au total on arrive donc à une complexité de $O(r^2)O(\ell^3)$, voire $O^+(r)O^+(\ell^2)$. □

4.5. La découverte d'Agrawal-Kayal-Saxena. Avec notre implémentation on obtient un coût d'ordre $sr^2\ell^3$ pour effectuer s tests d'AKS dans $\mathbb{Z}_n[X]/(X^r-1)$. À première vue cela semble raisonnable ; le problème est de majorer r et s. La découverte surprenante d'Agrawal-Kayal-Saxena est le résultat suivant :

Théorème 4.15 (Agrawal-Kayal-Saxena, 2002)**.** *Il existe une constante $C > 0$ telle que pour tout nombre naturel n on puisse trouver un nombre premier $r \leq C\,\mathrm{len}(n)^6$ tel que le critère suivant soit vrai :*

Si $\mathrm{pgcd}(n,a) = 1$ et $(X+a)^n = X^n + a$ modulo (n, X^r-1) pour tout $a = 1, \ldots, s$ avec $s = 2\lfloor\sqrt{r}\rfloor \mathrm{len}\, n$, alors n est premier ou une puissance d'un nombre premier. □

On admettra ce théorème. Il peut être démontré par des méthodes élémentaires (voir Smid). Ce résultat implique, comme promis, la conclusion énoncée dans l'introduction :

Corollaire 4.16. *Étant donné un nombre naturel n on peut déterminer s'il est premier ou composé par un algorithme de complexité $O(\mathrm{len}(n)^{19})$. En utilisant une multiplication optimisée on peut descendre jusqu'à une complexité $O^+(\mathrm{len}(n)^{12})$. En particulier, le problème de primalité admet une solution algorithmique de complexité polynomiale dans la longueur* $\mathrm{len}(n)$.

Exercice/M 4.17. Déduire le corollaire du théorème, en (ré)analysant l'implémentation ci-dessus : traduire d'abord la phrase « un nombre premier r d'ordre $O(\mathrm{len}(n)^6)$ » en une formulation précise, puis appliquer la proposition 4.14. Expliquer ensuite pourquoi il est peu coûteux de déterminer si n est une puissance parfaite : rappeler que l'on peut rapidement calculer $a = \lfloor\sqrt[k]{n}\rfloor$ puis comparer a^k avec n. Il suffit de ce faire pour $k = 2, 3, \ldots, \mathrm{len}(n)$. (Pourquoi ?) Quelle en est la complexité totale ?

4.6. Vers un test pratique ? Pour rendre le test AKS praticable, il faut expliciter *comment* choisir r et s. En voici une version simplifiée, qui se prête bien à l'implémentation :

Théorème 4.18 (Agrawal-Kayal-Saxena, 2002)**.** *Soit n un nombre naturel. Soit $q \geq 32\,\mathrm{len}(n)^2$ un premier tel que $r := 2q+1$ soit également premier et $n \not\equiv 0, \pm 1$ modulo r. Si $\mathrm{pgcd}(n,a) = 1$ et $(X+a)^n = X^n + a$ modulo (n, X^r-1) pour tout $a = 1, \ldots, s$ avec $s = 2\lfloor\sqrt{r}\rfloor \mathrm{len}(n)$, alors n est premier ou une puissance d'un nombre premier.* □

On en déduit l'algorithme XI.8 pour déterminer si un entier donné est premier ou composé.

Remarque 4.19. La correction puis l'efficacité de l'algorithme précédent repose sur l'existence de certains nombres premiers : on appelle q un *nombre premier de Sophie Germain* si q et $r = 2q+1$ sont tous les deux premiers. Ces nombres ne sont pas si rares que cela. Voici les exemples jusqu'à 1000 :

$$2, 3, 5, 11, 23, 29, 41, 53, 83, 89, 113, 131, 173, 179, 191, 233, 239, 251, 281, 293,$$
$$359, 419, 431, 443, 491, 509, 593, 641, 653, 659, 683, 719, 743, 761, 809, 911, 953$$

Le théorème des nombres premiers nous dit que la probabilité qu'un nombre q soit premier est environ $\frac{1}{\ln q}$. Si l'on est prêt à croire que la primalité de q et la primalité de $2q+1$ sont en quelque sorte

Algorithme XI.8 Une variante du test de primalité selon Agrawal-Kayal-Saxena

Entrée: un nombre naturel n

Sortie: le message « premier » ou « composé »

pour k **de** 2 **à** $\text{len}(n)$ **faire** $a \leftarrow \lfloor \sqrt[k]{n} \rfloor$: **si** $a^k = n$ **alors retourner** « composé »
initialiser $q \leftarrow 32\,\text{len}(n)^2$, $r \leftarrow 2q+1$
tant que q ou r est composé ou $n \equiv 0, \pm 1$ modulo r **faire** $q \leftarrow q+1$, $r \leftarrow r+2$
$s \leftarrow 2\lfloor \sqrt{r} \rfloor \text{len}(n)$
pour a **de** 1 **à** s **faire**
  si $\text{pgcd}(n,a) > 1$ **alors retourner** « composé »
  si $(X-a)^n \not\equiv X^n - a$ modulo (n, X^r-1) **alors retourner** « composé »
fin pour
retourner « premier »

indépendantes, alors il semble plausible que la probabilité de tomber sur un nombre premier de Sophie Germain soit environ $\frac{1}{\ln(q)^2}$.

Conjecture 4.20. *La quantité des nombres premiers de Sophie Germain suit l'asymptotique*

$$\#\{q \leq x \mid q \text{ et } 2q+1 \text{ sont premiers } \} \sim \text{const} \frac{x}{\ln(x)^2}.$$

Contrairement au théorème des nombres premiers, cette conjecture reste toujours ouverte. Elle correspond assez bien aux données empiriques : quand on cherche un nombre premier de Sophie Germain $q \geq 32\,\text{len}(n)^2$, il semble toujours en exister un de cet ordre de grandeur-là.

Proposition 4.21. *Supposons qu'il existe une constante C de sorte qu'il soit toujours possible de trouver un nombre premier de Sophie Germain $q \in [\![32\,\text{len}(n)^2, C\,\text{len}(n)^2]\!]$. Dans ce cas r et s sont de l'ordre de grandeur $O(\text{len}(n)^2)$, et l'algorithme ci-dessous est de complexité $O(\text{len}(n)^9)$, voire $O^+(\text{len}(n)^6)$ en utilisant des multiplications sophistiquées.*

Même sous cette hypothèse optimiste, l'efficacité du test AKS laisse encore à désirer :

Exemple 4.22. Si l'on prend $q = 809$ et $r = 1619$, on peut tester la primalité des nombres n jusqu'à 5 bits, car $32 \cdot 5^2 = 800$. Ceci toujours sous réserve, bien entendu, que $n \not\equiv 0, \pm 1$ modulo r ; si jamais cette condition pose problème, il faut passer à $q = 911$, $r = 1823$ ou $q = 953$, $r = 1907$ etc … En tout état de cause, pour les petits nombres n cette démarche n'est pas efficace.

Exemple 4.23. Prenons un exemple plus réaliste : disons que l'on veut tester des entiers à 100 bits, soit 30 décimales environ. Ici le test AKS est moins ridicule. Pour $\text{len}(n) = 100$ les nombres premiers $q = 320009$, $r = 640019$ conviennent, ainsi que $q = 320063$, $r = 640127$, puis $q = 320081$, $r = 640163$ etc. On voit, d'une part, qu'il existe beaucoup de nombres premiers de Sophie Germain, et d'autre part que les valeurs de r dans l'algorithme sont déjà assez élevées.

Exemple 4.24. Regardons finalement les entiers à 1000 bits, soit 300 décimales environ. Pour $\text{len}(n) = 1000$ les nombres premiers $q = 32000651$, $r = 64001303$ conviennent, ainsi que $q = 32000669$, $r = 64001339$, puis $q = 32000753$, $r = 64001507$, etc. Bien que théoriquement intéressant, le test AKS commence à occuper trop de mémoire et de consommer trop de temps pour être praticable dans cet ordre de grandeur.

Ces exemples indiquent que le beau résultat d'AKS ne se prête pas encore à une implémentation efficace. En particulier la version actuelle ne peut pas concurrencer avec le test probabiliste selon Miller-Rabin, qui traite des entiers à 1000 bits assez rapidement (le tester). Pour l'instant c'est donc l'aspect théorique qui fait le succès du résultat AKS. Toutefois, après cette innovation inattendue, on peut espérer que d'autres bonnes surprises nous attendent encore. À suivre …

Exercice/P 4.25. Expérimenter avec votre implémentation du test AKS pour étudier son comportement dans des exemples concrets. Par mesure de prudence, vérifier la correction des réponses fournies par votre programme, disons en recoupant avec un test de Miller-Rabin. Pour un nombre composé donné n peut-on trouver r et a petits de sorte que $(X+a)^n \not\equiv X^n + a$ modulo (n, X^r-1) ? Est-ce que la performance

empirique est meilleure que la prévision théorique ? Chercher dans la littérature une variante optimisée qui utilise des paramètres r et s plus petits, puis l'implémenter. Déterminer empiriquement le temps et la mémoire nécessaires ; jusqu'où ce test vous semble-t-il praticable ?

5. Résumé et perspectives

En guise de résumé, voici l'approche aux problèmes évoqués au début de ce chapitre.

(1) Tout d'abord on établit une fois pour toute un tableau des petits nombres premiers (jusqu'à 10^7 disons) en utilisant le crible d'Ératosthène.

(2) Pour tester si un nombre n est premier on teste d'abord la divisibilité par des petits nombres premiers. Si l'on trouve un facteur, alors n est composé.

(3) Si pour un grand entier n on n'a pas trouvé de petits facteurs, on passe au test de Miller-Rabin. Si l'on trouve un témoin, alors n est composé, sinon n est probablement premier.

(4) Si n est probablement premier et on exige une preuve, alors on essaiera de factoriser $n-1$ pour trouver un élément d'ordre $n-1$ dans $\mathbb{Z}_n^\times$. Celui-ci prouve que n est premier.

(5) Si n n'est pas premier, alors on essaiera de le factoriser. On extrait d'abord les petits facteurs premiers. Si le facteur restant est trivial ou premier, on a terminé.

(6) Dans le pire des cas, n est composé de grands facteurs. Pour le factoriser on applique la méthode de Pollard ou une méthode plus puissante.

Le principal problème réside dans l'étape 6 : la factorisation d'un nombre composé de grands facteurs. Plusieurs méthodes de factorisation ont été développées, dont chacune a poussé la limite du possible un peu plus loin. Pour en savoir plus on consultera avec profit l'excellent livre de R. Crandall et C. Pomerance [**15**]. Malgré tout progrès dans ce domaine, la factorisation de grands entiers reste un problème difficile, dont on ne connaît pas à ce jour de solution efficace.

Suppose moreover that the numbers p and q are thrown away by mistake,
but that their product pq is saved. How to recover p and q ?
It must be felt as a defeat for mathematics that, in these circumstances, the most promising
approaches are searching the waste paper basket and applying mnemo-hypnotic techniques.
H.W. Lenstra, *Elliptic curves and number-theoretic algorithms*, 1986

PROJET XI

Application à la cryptographie

Objectifs

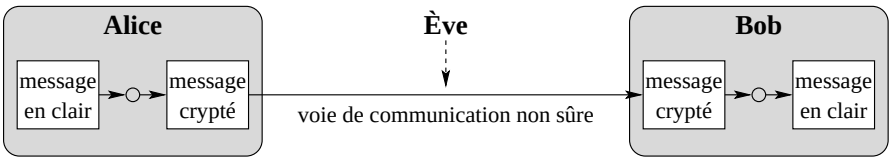
- Comprendre les idées essentielles de la cryptographie à clef publique.
- Implémenter une méthode répandue, la cryptographie selon RSA.

Sommaire

1. Qu'est-ce que la cryptographie ?

Des codes secrets ont été utilisés depuis fort longtemps dans le monde militaire et diplomatique pour assurer la confidentialité des messages. À nos jours l'usage civil s'est répandu avec l'avènement des ordinateurs et surtout de l'internet. Pour une introduction à cette fascinante histoire, je recommande vivement le livre de Simon Singh, *Histoire des codes secrets*, LGF, 2001.

1.1. Le problème de base. Le problème de base peut se résumer comme suit : Alice et Bob souhaitent établir une communication sécurisée afin d'échanger des informations confidentielles. Malheureusement le support (lettre, téléphone, internet) n'est pas sûr et peut être intercepté par la méchante Ève.



L'idée est de crypter des messages avant de les envoyer, de sorte que seule la personne autorisée puisse les décrypter. Alors que l'idée est évidente, la mise en œuvre l'est beaucoup moins !

1.2. Cryptage selon César. D'après la légende, Jules César utilisa un cryptage « alphabétique » suivant le schéma suivant. Les messages sont écrits dans un certain alphabet $\mathscr{A}$, disons l'alphabet latin usuel. Pour le cryptage on fixe une permutation $c\colon \mathscr{A} \to \mathscr{A}$ que l'on l'applique lettre par lettre. Le décryptage est ainsi l'application de la permutation inverse $d = c^{-1}$. Par exemple, avec

$$c = \begin{bmatrix} \texttt{ABCDEFGHIJKLMNOPQRSTUVWXYZ} \\ \texttt{XYZABCDEFGHIJKLMNOPQRSTUVW} \end{bmatrix} \quad \text{et} \quad d = \begin{bmatrix} \texttt{ABCDEFGHIJKLMNOPQRSTUVWXYZ} \\ \texttt{DEFGHIJKLMNOPQRSTUVWXYZABC} \end{bmatrix}$$

le message $m = \texttt{CECI EST UN MESSAGE}$ est crypté en $\bar{m} = m^c = \texttt{ZBZF BPQ RK JBPPXDB}$, puis décrypté en $m = \bar{m}^d = \texttt{CECI EST UN MESSAGE}$. Le programme `cesar.cc` vous fournira d'autres exemples.

Remarque 1.1. Le cryptage alphabétique n'est pas du tout sûr ! Le message crypté dévoile beaucoup trop d'information, et celle-ci peut être utilisée pour casser le code. Dans notre exemple on retrouve la longueur des mots, ce qui laisse deviner au moins les mots très courts. De la même veine, la *fréquence* des lettres (des paires, des triplets, ...) donne des indications assez précises. À noter que dans un texte français typique la lettre 'E' est la plus fréquente. Ainsi une simple analyse statistique du texte crypté dévoilera l'image de 'E', au moins très probablement. (Contempler aussi Georges Perrec, *La disparition*.)

1.3. Attaques statistiques. Pour juger de la sûreté d'un système cryptographique, il faut surtout connaître ses faiblesses : Quelles sont les attaques possibles ? Peut-on décrypter par « force brute » ? Ou par des analyses statistiques ? Ou par d'autres astuces, jusqu'ici inaperçues ? La principale faiblesse du cryptage selon César est qu'il succombe à des analyses statistiques :

Exercice 1.2. Décryptez le message « `VX VHWX XLM MKHI YTVBEX T VTLLXK` ». Explicitez votre démarche, notamment vos hypothèses tacites, puis le système de cryptage/décryptage dévoilé. Un logiciel adapté pourrait-il faire le même travail ? Peut-on être sûr que vous avez retrouvé le message initial ?

Exercice 1.3. Si ce genre de casse-tête vous amuse et que vous cherchez un défi, vous pouvez essayer de décrypter le message suivant : « `ERL BLCNEOGOJQR GFLGOQJCL LTO ER BLE BFET TECL NGJT OQEHQECT BGT OCLT IECL G ILACXBOLC. PQET PLRLV IL DQECRJC FG BCLEPL. GPLA ER BLE IL TOGOJTOJUEL LO ER IJAOJQRRGJCL DCGRAGJT ER BCQSCGNNL BLEO FL DGJCL JRTOGROGRLNLRO.` »

Remarque 1.4 (Bruit aléatoire). Afin de rendre l'analyse statistique plus difficile, on peut ajouter un « bruit aléatoire ». Plus explicitement, on peut identifier l'alphabet $\mathscr{A} = \{\mathtt{A}, \ldots, \mathtt{Z}\}$ avec le groupe $\mathbb{Z}_{26}$. À chaque lettre $a \in \mathbb{Z}_{26}$ on peut ainsi ajouter un nombre aléatoire $b \in \mathbb{Z}_{26}$ afin d'obtenir la lettre bruitée $c = a + b$. À la place de la lettre initiale a on note ensuite les deux lettres c et b, afin de retrouver la lettre initiale $a = c - b$. Par exemple, la lettre `E` est maintenant codé par une des paires `EA` ou `FB` ou `GC` etc... Le programme `cesar.cc` implémente cette idée comme option. Qu'en pensez-vous ? Cette astuce rend-elle le code plus sûr ?

Remarque 1.5 (Camouflage). Alice et Bob ont tout intérêt à dévoiler le moins d'information possible :

Éviter des répétitions: Envoyer deux fois le même message m est potentiellement dangereux, car Ève voit passer deux fois le même message crypté $\overline{m}$. Si par exemple le message `JSRYSD-QZMC YSBUVR` déclenche toujours la même action, observable par Ève, ceci permet une interprétation sans explicitement décrypter le message.

Éviter des provocations: Réciproquement, Ève pourrait tenter de provoquer un certain message, par exemple `EVE VIENT DE ME TELEPHONER`. Bien sûr Ève s'attend à ce que le mot `TELEPHONE` apparaisse quelque part dans le message, ce qui facilitera le décryptage.

Éviter le silence: Le fait d'envoyer un message ou d'en envoyer aucun contient de l'information, facilement observable pour Ève. Il vaut mieux crypter puis envoyer des messages comme `CECI EST UN MESSAGE VIDE POUR RAISON DE CAMOUFLAGE`, sans pour autant livrer des informations statistiques gratuites à Ève.

Dans tous ces cas, un bruit aléatoire judicieux permet à Alice et Bob de diminuer des corrélations trop évidentes : entre les parties d'un message, entre messages successifs, ainsi qu'entre messages et informations extérieures. Réciproquement, Ève a intérêt a collectionner toute information accessible, sur une longue durée, afin d'effectuer des analyses statistiques les plus fines possibles.

1.4. Cryptage et décryptage. Le cryptage selon César n'est pas sûr, certes, mais il nous indique une démarche plus générale qui mérite d'être analysée. Notons M l'ensemble des messages, disons des textes utilisant l'alphabet usuel $\mathscr{A} = \{\_, \mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$. Par commodité nous supposerons, comme avant, que les messages cryptés sont exprimés dans le même alphabet.

Choix du cryptage et du décryptage: Alice et Bob conviennent au préalable d'utiliser un cryptage $\mathrm{crypt}\colon M \to M$ et un décryptage $\mathrm{décrypt}\colon M \to M$ qu'ils estiment sûr.

Quel niveau de sécurité ? Ici « sûr » veut dire qu'il est difficile pour Ève de retrouver le message en clair m à partir du message crypté $\overline{m}$ dont elle dispose seulement. Par exemple, si l'on crypte lettre par lettre on retombe sur le cryptage à la César. Pour arriver à un bon niveau de sécurité il faut donc un système plus élaboré... Heureusement pour Alice et Bob, il y a beaucoup d'applications $\mathrm{crypt}\colon M \to M$ intéressantes, et possiblement plus « sûres ».

Quels que soient les détails, l'approche naïve souffre de deux problèmes fondamentaux :

Il faut absolument que la méthode reste secrète ! Dès que les fonctions crypt et décrypt sont dévoilées, elles n'offrent plus aucune sécurité. Pour ne pas se fier à une sécurité illusoire, il est donc très important de changer régulièrement le système cryptographique pour assurer un bon niveau de confidentialité. (Il faut prévoir un stock assez large...)

Comment se mettre d'accord sur une méthode de cryptage/décryptage ? Évidemment Alice ne peut pas la détailler à Bob en utilisant le support non sécurisé. L'accord préalable nécessite donc une deuxième voie de communication plus sûre (rendez-vous, lettre recommandée, ligne téléphonique sécurisée, etc.) Ainsi le vrai problème n'est que transféré et reste ouvert.

1.5. Cryptographie à clef. À cause du premier problème ci-dessus il s'avère beaucoup plus utile d'effectuer le cryptage/décryptage en fonction d'une clef, c'est-à-dire d'un paramètre supplémentaire.

Définition 1.6. Un *système cryptographique* à clef consiste en une méthode de cryptage crypt : $C \times M \to M$ et une méthode de décryptage décrypt : $D \times M \to M$, ainsi qu'une méthode pour produire des paires de clefs $(c,d) \in C \times D$ mutuellement inverses. Ceci veut dire que la clef $c \in C$ permet de crypter le message m avec pour résultat le message crypté $\overline{m} = \text{crypt}_c(m)$, tandis que la clef correspondante $d \in D$ permet de décrypter le message $\overline{m}$ avec pour résultat le message initial $m = \text{décrypt}_d(\overline{m})$.

La cryptographie à clef offre au moins deux grands avantages :

(1) On peut désormais publier le système cryptographique (les fonctions crypt et décrypt), par exemple pour l'analyser et le discuter publiquement, ou bien pour le breveter. Pour les utilisateurs Alice et Bob il suffira de garder secrète leur clef respective, c et d.

(2) Comme bénéfice supplémentaire, les utilisateurs peuvent régulièrement changer de clefs, sans pour autant changer entièrement de système cryptographique.

Exemple 1.7. Dans le cryptage alphabétique la clef pour le cryptage est la permutation $c : \mathscr{A} \to \mathscr{A}$, et la clef pour le décryptage est la permutation inverse $d = c^{-1} : \mathscr{A} \to \mathscr{A}$. Ici $C = D = \text{Sym}(\mathscr{A})$ est le groupe symétrique, et le cryptage/décryptage est l'opération lettre par lettre, comme discuté ci-dessus.

À noter qu'a priori les ensembles C et D n'ont rien à voir : le cryptage et le décryptage sont assez indépendants, on exige seulement qu'une paire de clefs (c,d) fournisse deux fonctions $\text{crypt}_c : M \to M$ et $\text{décrypt}_d : M \to M$ mutuellement inverses. Plus précisément, pour le décryptage il suffit d'exiger $\text{décrypt}_d \circ \text{crypt}_c = \text{id}_M$. Si l'ensemble M des messages est fini, ceci entraîne la bijectivité, donc automatiquement $\text{crypt}_c \circ \text{décrypt}_d = \text{id}_M$. Ce n'est en général plus vrai pour M infini.

Exemple 1.8 (Camouflage)**.** Pour camoufler les messages on peut ajouter un bruit aléatoire, comme expliqué dans la remarque 1.4, puis crypter le texte ainsi obtenu. Inversement, après le décryptage on supprime la partie aléatoire afin d'obtenir le texte initial. Ce procédé garantit toujours que $\text{décrypt}_d \circ \text{crypt}_c = \text{id}_M$, mais on n'a plus $\text{crypt}_c \circ \text{décrypt}_d = \text{id}_M$. (Le détailler.) Pourquoi est-ce important que M soit infini ?

2. Cryptographie à clef publique

Un système cryptographique à clef permet d'utiliser une méthode publique et de changer fréquemment les clefs. Ceci résout le premier des deux problèmes fondamentaux, mais non le deuxième : comment échanger les clefs via une ligne non sûre ?

Vers 1975 Diffie et Hellman introduisirent le concept de cryptographie à clef publique, qui est l'objet de ce paragraphe. Ce principe permet de résoudre le problème d'échange de clefs d'une manière aussi élégante que radicale : on publie la clef du cryptage ! Il restait à implémenter ce concept…

Soulignons d'abord que le cryptage à la César est incompatible avec le concept de clef publique : la clef pour le cryptage est la permutation $c : \mathscr{A} \to \mathscr{A}$. Quand on la publie, il est très facile d'en déduire la clef pour le décryptage, $d = c^{-1} : \mathscr{A} \to \mathscr{A}$. Cette symétrie rend impossible une publication *partielle* de clefs : si l'on en connaît une, on en connaît l'autre.

☞ *Pour la cryptographie à clef publique il est primordial que la clef c pour le cryptage ne dévoile pas la clef d pour le décryptage.* ✍

Une telle asymétrie est-elle possible ? En avril 1977 R. Rivest, A. Shamir et L. Adleman proposèrent une implémentation basée sur la difficulté de factoriser des grands nombres entiers. Ce système cryptographique à clef publique est maintenant connu sous le nom de RSA et très largement répandu.

2.1. Le protocole RSA. Rappelons qu'il est relativement facile de trouver deux grands nombres premiers « aléatoires » p et q, puis de calculer leur produit $n = pq$. Par contre, sans aucune connaissance de sa genèse il est en général très difficile de factoriser n. Cette difficulté peut sembler étonnante, elle pourrait même être considérée comme un grave défaut de notre théorie dans l'état actuel. Ironiquement, *l'absence* d'une solution efficace peut aussi avoir des applications intéressantes !

Une telle application fut proposée par Rivest, Shamir et Adleman. Il s'agit d'établir une communication sécurisée entre Bob et Alice par une voie de communication publique. C'est un problème très classique, qui à nos jours s'applique notamment à l'internet.

Choix des clefs: Alice choisit deux grands nombres premiers distincts p et q et calcule $n = pq$ ainsi que $\varphi(n) = (p-1)(q-1)$. Elle choisit également un nombre c premier avec $\varphi(n)$ et calcule un inverse d tel que $cd \equiv 1$ modulo $\varphi(n)$. Tous ces calculs sont faciles à effectuer (le rappeler).

Clef sécrète: Alice connaît le triplet (n,c,d) vérifiant $cd \equiv 1$ modulo $\varphi(n)$. La clef sécrète est la valeur de d, qu'elle garde précieusement pour elle-même.

Clef publique: La clef publique d'Alice est la paire (n,c), qu'elle affiche publiquement. Pour recevoir des messages secrets elle a intérêt que tout le monde connaisse (n,c).

Cryptage: Pour envoyer un texte à Alice, Bob représente son message comme un nombre $m \in [\![0,n[\![$, disons en codant les lettres par $1,\dots,26$, l'espace par 0, et un texte par un développement en base 27. À l'aide de la clef publique d'Alice, il calcule alors le message crypté $\overline{m} = m^c \bmod n$.

Décryptage: Alice reçoit le message crypté $\overline{m} \in [\![0,n[\![$. À l'aide de sa clef sécrète d elle en déduit le message en clair $m = \overline{m}^d \bmod n$. Ceci est possible parce que $cd \equiv 1$ modulo $\varphi(n)$, ce qui assure $\overline{m}^d = m^{cd} = m$ dans $\mathbb{Z}_n^\times$.

Exercice/M 2.1 (Inversibilité)**.** Vérifier que $\mathrm{crypt}_c\colon \mathbb{Z}_n \to \mathbb{Z}_n$, $m \mapsto m^c$ et $\mathrm{décrypt}_d\colon \mathbb{Z}_n \to \mathbb{Z}_n$, $\overline{m} \mapsto \overline{m}^d$ définissent deux applications mutuellement inverses.

Remarque 2.2. La sécurité du système RSA se base sur les hypothèses suivantes :
- Sans connaître p et q, il est difficile de trouver $\varphi(n)$ à partir de n.
- Sans connaître $\varphi(n)$, il est difficile de trouver d à partir de c.
- Sans connaître d, il est difficile de trouver m à partir de $\overline{m}$.

D'un point de vue pratique ces hypothèses sont vérifiées dans le sens qu'à l'heure actuelle il n'existe pas d'algorithme efficace publiquement connu pour factoriser des grands entiers n, ni pour calculer $\varphi(n)$ à partir de n, ni pour calculer d à partir de c, ni pour calculer m à partir de $\overline{m}$. Toutefois il est important à souligner que la sécurité du système RSA se base sur l'expérience et non sur une preuve mathématique.

Remarque 2.3. Soit n composé de deux nombres premiers distincts. Alors trouver $\varphi(n)$ équivaut à factoriser n. Effectivement, la factorisation $n = pq$ permet de calculer $\varphi(n) = (p-1)(q-1)$. Réciproquement les racines du polynôme $X^2 + (\varphi(n) - n - 1)X + n$ sont les facteurs p et q cherchés.

2.2. Implémentation du protocole RSA.

Exercice 2.4 (Puissance modulaire)**.** Implémenter une fonction

```
Integer puissance( Integer base, Integer exp, const Integer& mod )
```

qui calcule efficacement $b^e \bmod n$ pour des entiers $b \in \mathbb{Z}$, $e \geq 0$, $n \geq 1$.

Exercice 2.5 (Inverse modulaire)**.** Implémenter une fonction

```
Integer inverse( Integer a, const Integer& mod )
```

qui calcule efficacement l'inverse de a modulo n, si a est inversible, et qui renvoie 0 sinon. S'il existe, l'inverse de a est représenté par l'unique entier $u \in \{1,\dots,n-1\}$ tel que $au \equiv 1 \pmod{n}$.

Exercice 2.6 (Développement d'un entier en une base donnée)**.** Rappeler la méthode de Horner et l'utiliser pour écrire deux fonctions

```
Integer convert( const vector<Integer>& chiffres, const Integer& b )
vector<Integer> convert( Integer n, const Integer& b )
```

qui effectuent le développement en base `b` d'un entier de type `Integer`. Pour une spécification détaillée voir le fichier `rsa.cc`. Vous y trouvez également la fonction `renverser(vec)` qui pourra vous être utile.

On se propose de coder un *texte* selon la méthode RSA. Pour cela il faut d'abord transformer le texte en entier, puis retransformer un entier en texte. Vous trouverez dans `rsa.cc` une petite fonction `majuscules( string& texte )` qui transforme un texte en lettres majuscules et underscore '`_`'.

Exercice 2.7 (Transformations entre textes et entiers)**.** On interprète les lettres `A,...,Z` comme nombres $1,\dots,26$ et '`_`' comme 0. Ce sont donc les « chiffres » en base $b = 27$. On peut ainsi interpréter un texte

$t = (c_k, c_{k-1}, \ldots, c_1, c_0)$ comme l'entier $m = c_k b^k + c_{k-1} b^{k-1} + \cdots + c_1 b^1 + c_0$, et réciproquement tout entier m comme un texte t. Écrire ainsi deux fonctions

```
Integer convert( const string& texte )
string convert( Integer nombre )
```

qui effectuent la transformation entre texte et entier. Testez vos fonctions sur quelques exemples (à joindre en commentaire au fichier source). Les transformations devraient être inverses l'une à l'autre.
Indication. — Après avoir calculé `Integer c= n%27` la conversion `int(c)` n'est malheureusement pas acceptée ; il faut appeler la fonction `c.get_ui()`, notation un peu lourde pour *get unsigned integer.*

Exercice 2.8 (Cryptage RSA)**.** Écrire une fonction qui crypte un texte selon la méthode RSA :

```
cryptageRSA( const Integer& mod, const Integer& exp, string& texte )
```

On se servira des fonctions précédentes, avec les notations $n =$ `mod`, $c =$ `exp`, $t =$ `texte`. Le texte t est transformé en un entier m, puis m est développé en $m_k, m_{k-1}, \ldots, m_1, m_0 \in \{0, \ldots, n-1\}$ en base n. Ensuite on peut calculer $m'_i \leftarrow m_i^c \bmod n$ et en déduire l'entier crypté m', puis le texte crypté t'.

Test de correction. — La fonction `cryptage( istream& in, ostream& out )` lit les données n, c, t du flot d'entrée et écrit n, c, t' dans le flot de sortie. Si votre programme compilé s'appelle `cryptage`, alors la commande `./cryptage < test1.rsa` produira le résultat `15 3 FELICITATIONS`. Vérifier aussi le (dé)cryptage dans `test2.rsa`. Quel message se cache dans `test3.rsa` ?

Exercice 2.9 (Question bonus)**.** Si cela vous inspire, essayez de décrypter le message dans `test4.rsa`. Pour la décomposition en facteurs premiers vous pouvez employer tout logiciel à votre disposition. Quels consignes formuleriez-vous pour le choix des nombres premiers p et q afin d'assurer la sûreté du système RSA ? Sur quelles informations ou hypothèses vos consignes se basent-elles ?

Exercice 2.10 (Question bonus)**.** En quoi se ressemblent les cryptages selon César et selon RSA ? Quelle est la principale différence ? Pourquoi pour César la publication de la clef c serait catastrophique tandis que pour RSA elle n'est pas considérée comme inquiétante ?

2.3. Production de clefs. Jusqu'ici on sait appliquer le cryptage/décryptage à une clef (n, c) et un texte donnés. Ce dernier paragraphe implémente finalement la production de clefs (n, c) convenables.

Exercice 2.11 (Test de primalité)**.** Implémenter une fonction

```
bool estPseudoPremier( const Integer& n, const Integer& x )
```

qui teste si n est pseudo-premier par rapport à la base x, puis

```
bool estProbablementPremier( Integer n, int essais=100 )
```

qui effectue le test de Miller-Rabin (jusqu'à 100 fois). En déduire une fonction

```
Integer produirePremierAleatoire( Integer pmin, Integer pmax )
```

qui produit un nombre premier aléatoire entre $p_{\min}$ et $p_{\max}$.

Exercice 2.12 (Production de clefs)**.** Écrire une fonction

```
produireClefRSA( int bits, Integer& n, Integer& c, Integer& d )
```

qui produit une clef aléatoire, de la taille souhaitée en bits, pour le cryptage RSA. Avec ces clefs aléatoires, tester le cryptage/décryptage sur quelques exemples (à joindre en commentaire au fichier source).

2.4. Signature et authentification.

Exercice 2.13. Expliquer comment Alice peut *signer* un document électronique, de sorte que tout le monde puisse *vérifier* la signature, mais personne ne puisse la *falsifier*. *Indication.* — La signature d'un document D en clair peut être un message m qui répète le document D et ajoute « lu et approuvé, Alice ». Évidemment ce message est trivial à falsifier. Pour cette raison Alice donne comme signature $\bar{m} = m^d \bmod n$. Discuter la question à savoir si cette méthode résout le problème de signature.

Exercice 2.14. Produire une clef publique (n, c) avec clef secrète d (que vous ne publiez pas, évidemment). Écrire un petit message en guise de conclusion de ce projet, puis signez-le avec votre clef secrète d. Je vérifierai votre signature avec votre clef publique (à joindre avec le message signé).