Nombres premiers

Test de primalité

Pour déterminer si un entier n est premier, le plus simple est de tester successivement tous les diviseurs d possibles de n. Lorsque n n'est pas trop grand, cette méthode est efficace, à condition de remarquer qu'on peut cesser la recherche d'un diviseur dès que $d^2 > n$. En effet, si n est *composite*, c'est-à-dire produit de deux facteurs (différents de 1), le plus petit d'entre eux est inférieur ou égal à la racine carrée de n.

Voici une fonction qui associe à n la valeur 1 si n est premier, 0 sinon (après avoir testé le diviseur 2, on ne teste plus que les diviseurs impairs, pour diviser par deux le temps de calcul) :

```
int premier (long n) {
  long d;

  if (n % 2 == 0)
    return (n == 2);
  for (d = 3; d * d <= n; d = d + 2)
    if (n % d == 0)
      return 0;
  return 1;
}
```

Commentaires de programmation

- L'instruction `return (n == 2)` retourne 1 si n vaut 2, 0 sinon, ce qui exprime que 2 est le seul nombre premier pair. Il faut éviter d'écrire, comme le font la plupart des débutants :
  ```
  if (n == 2) return 1; else return 0;
  ```
  qui dit la même chose, mais de façon inutilement compliquée.

- Une instruction `return` termine l'exécution de la fonction, quelle que soit sa place. L'instruction finale `return 1` est exécutée seulement lorsque l'instruction `return 0` de la boucle ne l'a jamais été, c'est-à-dire lorsque la recherche d'un diviseur a échoué.

Complexité

Ce test de primalité exécute, si n est premier, $sqrt(n)/2$ divisions. Pour des entiers représentables sur 32 chiffres binaires, la complexité ne dépasse donc pas 2^{15} divisions, qui peuvent être effectuées en quelques millisecondes sur une machine moderne.

Mais en général, si n s'écrit avec k chiffres en base b, la complexité de cet algorithme est proportionnelle à $b^{k/2}$, et est donc exponentielle en fonction de k. Si l'on dispose de fonctions pour effectuer des opérations exactes sur des entiers de longueur quelconque (représentés par des tableaux), comme c'est le cas avec le langage Maple, il est hors de

question d'appliquer cet algorithme à des nombres ayant beaucoup plus de 20 chiffres décimaux : 10^{10} divisions (de grands nombres) exigent 10000 secondes, à raison d'une par microseconde, soit 3 heures ; si n s'écrit avec 50 chiffres décimaux, le temps d'exécution dépasse l'âge de l'univers ($15 * 10^9$ années valent approximativement $5 * 10^{17}$ secondes).

On peut accélérer l'algorithme en ne testant pas les diviseurs multiples de 3 (à part 3 lui-même) ; il reste à tester les diviseurs égaux à 1 ou 5 modulo 6, soit un tiers des nombres inférieurs à la racine de n (au lieu de la moitié dans la fonction ci-dessus). Mais le développement de cette idée ne mène à aucun progrès décisif, car il faut tester au moins les diviseurs premiers, et la densité de ceux-ci ne décroît que très lentement : le *théorème fondamental de l'arithmétique* énonce que la densité des nombres premiers inférieurs à N vaut asymptotiquement $1 / ln(N)$.

Voir en annexe la table des nombres premiers inférieurs à 10000, avec leur répartition comparée à celle donnée par la densité asymptotique ci-dessus.

Décomposition en facteurs premiers

Le test de primalité précédent, par recherche systématique d'un diviseur, peut être facilement modifié pour fournir la décomposition en facteurs premiers d'un entier n. Lorsqu'on trouve un diviseur d, celui-ci est nécessairement premier, sinon on aurait déjà trouvé un diviseur plus petit ; il suffit donc d'enregistrer d dans un tableau, et de poursuivre la recherche des facteurs de n/d. Voici une fonction qui construit le tableau des facteurs premiers de n, et retourne la taille de ce tableau, c'est-à-dire le nombre de facteurs premiers :

```
int decompose (long facteur[], long n) {
  long d = 3;
  int i = 0;

  while (n % 2 == 0) {
    facteur[i++] = 2;
    n = n / 2;
  }
  while (d * d <= n)
    if (n % d == 0) {
      facteur[i++] = d;
      n = n / d;
    }
    else
      d = d + 2;
  if (n > 1)
    facteur[i++] = n;
  return i;
}
```

Commentaires de programmation

- L'instruction `facteur[i++] = d;` est une abréviation très courante de :
  `facteur[i] = d; i++;`
  C possède une abréviation symétrique (moins courante) `facteur[++i] = d;` pour :
  `i++; facteur[i] = d;`

- Pour accélérer les calculs, on a à nouveau traité à part le cas du diviseur 2, seul nombre premier pair.

- Lorsque d est un facteur premier, il ne faut pas lui ajouter 2 après avoir divisé n par d, car un facteur peut apparaître avec une multiplicité supérieure à 1. On ne peut donc

pas écrire la boucle sous la forme :

```
for (d = 3; d * d <= n; d = d + 2)
```

Inversement, il serait maladroit de réinitialiser *d* avec la valeur 3 après avoir divisé *n* par *d*, car l'algorithme fournit les facteurs dans l'ordre croissant.

- Ne pas oublier le dernier facteur premier, qui est *n*, lorsque $d^2 > n$. Si le nombre à décomposer est une puissance de 2, la fonction (telle qu'elle est écrite ci-dessus) se termine avec $n = 1$, d'où le test pour éliminer ce cas particulier.

Complexité

L'algorithme est rapide si *n* ne possède que de petits facteurs premiers ; on prouve hélas que ce n'est pas le cas statistiquement, et si par exemple $n = pq$ est le produit de deux nombres premiers voisins, l'algorithme a une complexité exponentielle en fonction du nombre de chiffres nécessaires pour écrire *n*.

Test de Fermat

Le petit théorème de Fermat énonce que, si *p* est premier, et si *a* n'est pas un multiple de *p* :

$$a^{p-1} = 1 \ (\mathbf{mod}\ p)$$

Pour tester si *n* est premier, on peut donc choisir $a < n$ (*a* est appelé *base* du test), par exemple $a = 2$, et calculer a^{n-1} (**mod** *n*) ; si le résultat est différent de 1, *n* est certainement *composite* ; par contre, on ne peut rien conclure de façon sûre si le résultat vaut 1, car la réciproque du petit théorème de Fermat est fausse : il y a seulement une assez forte probabilité pour que *n* soit premier. En recommençant le test pour d'autres valeurs de la base *a* on améliore en général fortement la *probabilité* d'obtenir une réponse correcte, lorsque *n* semble premier.

Lorsque $a^{n-1} = 1$ (**mod** *n*), avec *n composite*, on dit que *a* est un *menteur* pour *n*, ou que *n* est *pseudo-premier* pour la base *a*. Voir en annexe le comportement des nombres composites impairs inférieurs à 2^{15} et pseudo-premiers pour la base 2.

L'efficacité de ce test repose sur l'algorithme d'exponentiation rapide. Il en existe une variante un peu plus sophistiquée du point de vue mathématique, appelée test de *Miller-Rabin*, qui reste très simple à programmer et augmente considérablement la probabilité d'obtenir une réponse correcte lorsque *n* semble premier.

Test de Miller-Rabin

On décompose l'exposant $n - 1$ sous la forme $2^k e$, avec *e* impair, et on considère la suite :

$$p_0 = a^e, p_1 = a^{2e}, p_2 = a^{4e}, p_3 = a^{8e}, \ldots p_k = a^{n-1}$$

(tous les calculs sont effectués **mod** *n*). Chaque p_i ($i > 0$) est le *carré* du précédent, or si *n* est premier :

- le dernier terme p_k de la suite vaut 1 (petit théorème de Fermat) ;
- l'équation $x^2 = 1$ (**mod** *n*) a exactement deux solutions, 1 et -1 (autrement dit $n - 1$).

On en déduit facilement que deux cas, et deux seulement, sont possibles pour la suite des p_i lorsque *n* est premier :

1. $p_0 = 1$, et donc pour tout i, $p_i = 1$;
2. il existe $i < k$ tel que $p_i = -1$, et donc pour tout $j > i$, $p_j = 1$.

Si la suite des p_i ne vérifie aucune de ces deux conditions, n est sûrement composite ; si par contre elle vérifie l'une de ces deux conditions, n est vraisemblablement premier. Lorsque le test de Miller-Rabin est positif, alors que n est composite, on dit que a est un *menteur fort* pour n, ou que n est *fortement pseudo-premier* pour la base a.

Voici une fonction qui réalise le test de Miller-Rabin, en associant au couple (a, n) la valeur 1 si le test de base a est réussi (n semble premier) et 0 sinon (n est sûrement composite). La fonction `puissance` est celle de la leçon sur l'exponentiation rapide.

```
int rabin (long a, long n) {
  long p, e, m;
  int i, k;

  e = m = n - 1;
  for (k = 0; e % 2 == 0; k++)
    e = e / 2;

  p = puissance (a, e, n);
  if (p == 1) return 1;

  for (i = 0; i < k; i++) {
    if (p == m) return 1;
    if (p == 1) return 0;
    p = (p * p) % n;
  }

  return 0;
}
```

Commentaires de programmation

- L'instruction `if (p == m) return 1;` compare p à -1 (**mod** n) : comme le reste de la division de deux entiers positifs est toujours positif, on remplace -1 par $m = n - 1$.
- Dans la boucle l'instruction `if (p == 1) return 0;` est exécutée si et seulement si $p_i = 1$ avec $i > 0$ (sinon on n'entre pas dans la boucle) et p_{i-1} différent de -1 (sinon l'exécution de la boucle est déjà terminée). C'est le cas étudié dans la remarque ci-dessous.
- La dernière instruction `return 0;` est exécutée si et seulement si p_{k-1} est différent de 1 et -1. Il est alors inutile de tester si $p_k = 1$ (test de Fermat) car dans ce cas 1 admet p_{k-1} pour racine carrée, et donc n est composite.

L'expérience montre que le test de Miller-Rabin est beaucoup plus efficace que le test de Fermat, car il existe beaucoup moins de menteurs forts que de menteurs simples : voir en annexe le comportement du test de Miller-Rabin pour tous les nombres composites impairs inférieurs à 2^{15} et pseudo-premiers pour la base 2 (selon le test de Fermat).

Remarque : lorsque a est un menteur pour n, mais pas un menteur fort, la suite des p_i fournit un entier x différent de 1 et -1, et tel que $x^2 = 1$; $x - 1$ et $x + 1$ sont alors des diviseurs de 0 (**mod** n), donc ont des facteurs communs avec n : il suffit d'appliquer l'algorithme d'Euclide pour obtenir rapidement des facteurs explicites de n. Par exemple

pour $n = 341$, on trouve que $32^2 = 1$ (**mod** 341) : 341 est divisible par 31 et possède le facteur 11 en commun avec 33.

Primalité et factorisation : complexités comparées

Le test de Fermat permet de prouver très rapidement qu'un entier n est *composite* ; par contre ce test ne fournit pas de facteur de n, et il n'existe aujourd'hui **aucun algorithme efficace de factorisation**. L'échec de la recherche, pourtant très active, d'un algorithme efficace (c'est-à-dire de complexité polynomiale en fonction du nombre de chiffres de l'entier à factoriser) conduit tous les spécialistes de la question à conjecturer qu'il n'en existe pas.

Le meilleur (et très sophistiqué) algorithme connu a permis de factoriser en 1994 un nombre n écrit avec 129 chiffres décimaux (produit de deux facteurs premiers de même ordre de grandeur que la racine de n) en utilisant 1600 machines pendant plusieurs mois !

La situation est radicalement différente pour les tests de primalité : le test de *Miller-Rabin* est un test probabiliste très efficace, et il existe des algorithmes sûrs (c'est-à-dire non probabilistes) sophistiqués et efficaces pour prouver qu'un nombre est premier.

Il est ainsi possible aujourd'hui de produire très rapidement des nombres premiers comportant 150 chiffres décimaux ; une fois deux de ces nombres multipliés entre eux, il est rigoureusement impossible, en pratique, de retrouver les facteurs du produit (pour qui, bien sûr, ne connaît que ce produit), et on est presque certain que cette impossibilité est définitive.

Voir aussi

Exponentiation rapide, cryptographie.