Test Di Primalità

Esperienze di Programmazione

Alessio Bardelli, 544270

A.A. 2019/2020

Indice

1 Introduzione al Problema

1.1 Cenni ai Numeri Primi

Si inizia dando la definizione di numero primo.

Definizione 1 (Numero Primo)**.** *Un numero $n \in \mathbb{N}$, è chiamato numero primo, o semplicemente primo, se $n > 1$ e $\forall\, a, b \in \mathbb{N}.\ 1 < a, b < n \Rightarrow a \cdot b \neq n$. Ovvero un numero è primo se è maggiore di 1 e non può essere scritto come il prodotto di due numeri naturali più piccoli.*

I numeri naturali maggiori di 1 che non sono primi sono chiamati numeri composti.

Ogni numero naturale ha sia 1 che se stesso come divisore. Conseguentemente se un numero naturale ha un altro divisore, questo non può essere primo. Questa idea porta ad una definizione diversa ma equivalente dei numeri primi.

Definizione 2 (Numero Primo)**.** *Sono i numeri con esattamente due divisori positivi, 1 e il numero stesso.*

1.2 Test di Primalità

Per molte applicazioni, come ad esempio la crittografia, c'è spesso la necessità di trovare numeri primi grandi e "casuali". Fortunatamente però i numeri primi grandi non sono troppo rari, così che non sia richiesto troppo tempo per trovare un primo verificando via via interi casuali della dimensione appropriata. A tal proposito si da la seguente:

Definizione 3 (Funzione di distribuzione dei primi)**.** *Si definisce con $\pi(n)$ la funzione che specifica il numero di primi $\leq n$.*

$$\pi(n) = \#\{p \mid prime(p) \wedge 2 \leq p \leq n\}$$

Per esempio $\pi(10) = 4$, poiché ci sono 4 numeri primi minori o uguali a 10, cioè 2, 3, 5 e 7. Il teorema dei numeri primi fornisce un'importante approssimazione di $\pi(n)$, che descrive la distribuzione asintotica dei primi tra gli interi positivi.

Teorema dei Numeri Primi [1]. *Sia $n \in \mathbb{N}$ e sia dato $\pi(n)$, $n/\log(n)$ è una buona approssimazione della funzione $\pi(n)$, nel senso che il limite del quoziente delle due funzioni $\pi(n)$ e $n/\log(n)$ quando n tende a infinito è 1.*

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\log_e n} = 1$$

Il precedente teorema è conosciuto anche come la *legge asintotica della distribuzione dei numeri primi*. Usando la notazione asintotica il precedente risultato può essere riformulato come

$$\pi(n) \sim \frac{n}{\log_e n}$$

Utilizzando il teorema dei numeri primi, si ottiene che la probabilità che un numero scelto casualmente sia primo è pari a $\frac{1}{\log_e n}$. Pertanto, si dovrebbero esaminare approssimativamente $\log_e n$ interi scelti casualmente vicino ad n per trovare un primo che sia della stessa lunghezza di n.

Al fine di stabilire la primalità del numero selezionato casualmente, si esegue il **test di primalità**; tale test è una procedura algoritmica che, dato un numero naturale n in input, restituisce *true* se n è un numero primo, *false* se n è un numero composto.

E' stato dimostrato che il **test di primalità** appartiene alla classe di complessità $\mathcal{P}$[2].

2 Algoritmi

2.1 Prova di Divisione

Un semplice approccio al problema di verificare la primalità è la *prova di divisione*. Questo algoritmo si basa sulla definizione (2) di numero primo e sul seguente teorema:

Teorema 1. *Sia $n \in \mathbb{N}$. Se n è un numero composto allora avrà sicuramente un fattore $q \in \mathbb{N}$ tale che $2 \leq q \leq \sqrt{n}$.*

Dimostrazione. Se per assurdo esistessero due fattori $q, p \in \mathbb{N}$ tali che $q, p > \sqrt{n}$, risulterebbe che $n = q \cdot p > \sqrt{n} \cdot \sqrt{n} = n$, il che è assurdo. Di conseguenza almeno uno dei due fattori, q o p, è $\leq \sqrt{n}$. □

Si cerca quindi di dividere n per ogni intero compreso tra 2 e $\sqrt{n}$. Di conseguenza n è primo se e solo se nessuno dei divisori provati divide n.

2.2 Test di Fermat

Questo algoritmo si basa sul *piccolo teorema di Fermat* che, nella sua forma più generale dice:

Piccolo Teorema di Fermat. *Se n è un numero primo, allora per ogni intero $a \in \mathbb{Z}$ vale che*

$$a^n \equiv a \ (mod\ n).$$

Da questo teorema segue il corollario:

Corollario 1. *Se n è un numero primo e $a \in \mathbb{Z}$ è tale che $MCD(n, a) = 1$, allora*

$$a^{n-1} \equiv 1 \ (mod\ n).$$

Dunque, se si vuole verificare la primalità di un certo numero n, è sufficiente scegliere un numero casuale e coprimo con n e, vedere se la congruenza espressa dal corollario (1) vale. Se per qualche valore di a, la congruenza non vale, allora n è sicuramente composto e a prende il nome di testimone di Fermat per la composizione di n.

E' poco probabile che la precedente congruenza valga per un a casuale quando n è composto, per cui, se tale congruenza vale, si dice che n è probabilmente primo. Può accadere difatti che, la congruenza venga soddisfatta anche da numeri non primi. Ad esempio $2^{340} \equiv 1 \ (mod\ 341)$ anche se $341 = 11 \times 31$; si dice che 341 è uno pseudo-primo di Fermat di base 2. Questo fa si che il Test di Fermat non sia una procedura deterministica.

2.3 Test di Miller–Rabin

Questo algoritmo si basa su proprietà algebriche differenti da quelle utilizzate nelle precedenti versioni del test di primalità.

Poiché non ha senso scegliere un numero pari, visto che questo sarà certamente composto, ammettiamo che n sia un numero dispari; poniamo $n - 1 = 2^w \cdot z$, con w e z interi positivi, in particolare z dispari e $w \geq 1$ il più grande esponente che si può dare a 2; sia a un intero arbitrario tale che $2 \leq a < n$; se n è primo allora valgono i due predicati seguenti:

i. $MCD(n, a) = 1$

ii. $(a^z \equiv 1 \ (mod\ n))$ **or** $(\exists i \ t.c. \ 0 \leq i \leq w - 1 \wedge a^{2^i z} \equiv -1 \ (mod\ n))$.

Il predicato *(i)* deriva dalla definizione di numero primo, mentre il predicato *(ii)* deriva da note proprietà dell'algebra modulare [3].

Si ha che se n è un intero composto, il numero di interi a compresi tra 2 e $n - 1$ che soddisfano entrambi i predicati *(i)* e *(ii)* è minore di $\frac{1}{4}$. Dunque se per un intero a scelto a caso tra 2 e $n - 1$, almeno uno dei predicati *(i)* o *(ii)* è falso, allora n è certamente un numero composto; se entrambi i predicati sono veri, possiamo solo concludere che n è composto con probabilità minore di $\frac{1}{4}$ e, è primo con probabilità maggiore di $\frac{3}{4}$ [3]. Conseguentemente l'errore fatto da questo test è misurato dalla probabilità che un numero composto sia dichiarato come probabilmente primo. Si noti che anche questo test, come il test di Fermat, è un test non deterministico.

Si dice che a è un testimone forte del fatto che n è composto; invece, nel caso in cui n è composto e, i predicati precedenti valgono lo stesso, a prende il nome di bugiardo forte.

3 Implementazione Degli Algoritmi

3.1 Prova di Divisione

Di seguito si riporta lo pseudo-codice dell'algoritmo.

Algoritmo 1: Trival-Division Test

```
Input: n ≥ 2
for i = 2 to √n do
    if i divide n then
        return false;
return true;
```

Assumendo che ogni prova di divisione richieda tempo costante, la complessità di questo algoritmo è $\mathcal{O}(\sqrt{n})$, che è esponenziale rispetto alla lunghezza dell'input n.

3.2 Test di Fermat

Di seguito si riporta lo pseudo-codice dell'algoritmo.

Algoritmo 2: Fermat Test

```
Input: n ≥ 2
Prendi a nell'intervallo [2, n) tale che MCD(a, n) = 1;
if a^{n-1} mod n = 1 then
    return true;
else
    return false;
```

Si noti che per quanto a e n siamo dei valori grandi, utilizzando il metodo delle quadrature successive, si riesce a calcolare $a^{n-1} \ mod \ n$ in $\log^3 n$ operazioni. Dunque l'algoritmo impiega in media $\mathcal{O}(\log^3 n)$ operazioni per terminare.

Come discusso precedentemente questo algoritmo non è deterministico. Conseguentemente il testimone a che viene scelto casualmente, potrebbe mentire. Alla luce di questo, per migliorare l'algoritmo, si possono scegliere k testimoni e ripetere per ognuno di essi la procedura. Si noti che questa

modifica non comporta alcun degrado delle prestazioni dell'algoritmo, poiché questo impiega in media $\mathcal{O}(k \cdot \log^3 n)$ operazioni per terminare.

3.3 Test di Miller–Rabin

Il parametro k determina il numero di testimoni da provare. Più testimoni vengono provati, migliore sarà l'accuratezza del test. Se n è composto, questo test dichiara n come probabilmente primo con una probabilità al massimo di 4^{-k} [3]. Si noti dunque che la probabilità di errore dell'algoritmo è $< 10^{-18}$ se si provano 30 testimoni. L'algoritmo può essere scritto in pseudo-codice come segue:

Algoritmo 3: Miller-Rabin-Test

```
   Input: n ≥ 2, k ≥ 1
 1 Trovare w, z tali che n − 1 = 2^w · z e z dispari;
 2 for j = 1 to k do
 3 |   Scegliere un intero casuale a nell'intervallo [2, n);
 4 |   if MCD(n, a) ≠ 1 then
 5 |   |   return false;
 6 |   x = a^z mod n;
 7 |   if x = 1 or x = n − 1 then
 8 |   |   goto line 2;
 9 |   for i = 1 to w − 1 do
10 |   |   x = x^2 mod n;
11 |   |   if x = n − 1 then
12 |   |   |   goto line 2;
13 |   return false;
14 return true;
```

Si noti che a differenza del predicato *(ii)* che, imporrebbe di testare se $a^{2^i z} \ mod \ n = -1$, si testa se $a^{2^i z} \ mod \ n = n - 1$, poiché vale che $n - 1 \equiv -1 \ (mod \ n)$.

Anche in questo caso, come per il test di Fermat, se si utilizza il metodo delle quadrature successive, per eseguire i calcoli delle linee 6 e 10, l'algoritmo impiega in media $\mathcal{O}(k \cdot \log^3 n)$ operazioni per terminare.

3.4 Classe *Libreria*

L'implementazione degli algoritmi proposti precedentemente è stata fatta utilizzando il linguaggio Java. In particolare, per rappresentare e operare su

interi di grandi dimensioni, si è fatto uso della classe *BigInteger* offerta dal linguaggio stesso. La classe *Libreria* offre tra gli altri i seguenti metodi statici:

- `static boolean prime_1(BigInteger n)`

  Tale metodo implementa lo pseudo-codice proposto per l'algoritmo di *prova di divisione*.

- `static boolean prime_2(BigInteger n)`

  Tale metodo implementa lo pseudo-codice proposto per il *test di Fermat*.

- `static boolean prime_3(BigInteger n, int k)`

  Tale metodo implementa lo pseudo-codice proposto per il *test di Miller-Rabin*.

- `static BigInteger getRandom(BigInteger n, BigInteger m)`

  Tale metodo è utilizzato per ottenere un intero nell'intervallo $[min, max]$ in modo pseudo-casuale.

- `static BigInteger getPrime(BigInteger n, BigInteger m, int k)`

  Tale funzione, facendo uso dei metodi *getRandom* e *prime_3* sopra descritti, restituisce un intero "probabilmente" primo e pseudo-casuale tramite l'approccio descritto nel primo capitolo.

4 Alcuni Risultati

Per testare il metodo statico *prime_1* si è utilizzato 4 numeri primi, rispettivamente di 5, 10, 15 e 20 cifre decimali, ottenendo così i seguenti risultati:

Numero di cifre decimali	Numero testato	Risultato	Tempo
5	10007	true	0.005 sec
10	1000000007	true	0.024 sec
15	$10^{14} + 31$	true	0.720 sec
20	$10^{19} + 51$	true	7 min 32.5 sec

Tali interi, sono stati generati tramite il metodo *nextProbablePrime* della classe *BigInteger*. Si noti come all'aumentare del valore dell'intero testato, aumenti esponenzialmente il tempo impiegato per stabilire la primalità dell'intero stesso. Si evince dunque che, utilizzando l'algoritmo implementato dal metodo *prime_1*, i risultati, quando l'intero è di dimensioni molto grandi, sono prodotti in tempo non utile.

Per quanto riguarda il test degli algoritmi implementati rispettivamente dai metodi statici *prime_2* e *prime_3*, si utilizzano 2 numeri primi, rispettivamente di 91 e 121 cifre decimali, ottenendo così i seguenti risultati:

prime_2			
Numero di cifre decimali	Numero testato	Risultato	Tempo
91	$2^{300} - 153$	true	0.004 sec
121	$2^{400} - 593$	true	0.005 sec

prime_3, $k = 30$			
Numero di cifre decimali	Numero testato	Risultato	Tempo
91	$2^{300} - 153$	true	0.018 sec
121	$2^{400} - 593$	true	0.026 sec

Osserviamo che il metodo *prime_2* è leggermente più veloce del metodo *prime_3*. Questi ultimi due metodi sono molto efficienti ma, presentano delle criticità dovute, come discusso nei capitoli precedenti, al loro non determinismo. Difatti se si sceglie l'intero 1590231231043178376951698401 che, è composto [4], si ottiene il seguente risultato:

prime_2		
Numero testato	Risultato	Tempo
1590231231043178376951698401	true	0.0002 sec

Tale numero prende il nome di *numero di Carmichael*. Sfortunatamente, il test di primalità di Fermat che, è implementato dal metodo *prime_2*, produce un risultato scorretto per tutti i numeri di Carmichael, qualsiasi testimone sia scelto. In aggiunta si ha che, se pur molto rari, i numeri di Carmichael sono infiniti [5].

Anche il test di Miller-Rabin, implementato dal metodo *prime_3*, non è deterministico, come il test di Fermat, ma a differenza di quest'ultimo il test di Miller-Rabin ha la possibilità di aumentare e diminuire a piacimento la precisione del test tramite il parametro k.

In oltre come si evince dal seguente risultato, il test di Miller-Rabin supera anche il problema dei numeri di Carmichael:

prime_3, $k = 30$		
Numero testato	Risultato	Tempo
159023123104317837695169840 1	false	0.0004 sec

Riferimenti bibliografici

[1] Goldfeld, Dorian (2004). "The elementary proof of the prime number theorem: an historical perspective".

[2] Agrawal Manindra; Kayal Neeraj; Saxena Nitin (2004). "PRIMES is in P" Annals of Mathematics. 160 (2): 781–793.

[3] Rabin Michael O. (1980), "Probabilistic algorithm for testing primality", Journal of Number Theory, 12 (1): 128–138.

[4] "Least Carmichael number with n prime factors.", The Online Encyclopedia Of Integer Sequences, A006931

[5] W. R. Alford; Andrew Granville; Carl Pomerance (1994). "There are Infinitely Many Carmichael Numbers", Annals of Mathematics. 139 (3): 703–722.

Appendice

```
1  import java.util.Random;
2  import java.math.BigInteger;
3  import static java.math.BigInteger.ZERO;
4  import static java.math.BigInteger.ONE;
5  import static java.math.BigInteger.TWO;
6
7  class libreria {
8
9      static BigInteger[] test_prime_1 = new BigInteger[] {
10         new BigInteger("10000").nextProbablePrime(),
11         new BigInteger("1000000000").nextProbablePrime(),
12         new BigInteger("100000000000000").nextProbablePrime(),
13         new BigInteger("10000000000000000000").nextProbablePrime()
14     };
15
16     static BigInteger[] test_prime_2_3 = new BigInteger[] {
```

```
17
18          /* n = 2^300 - 153. */
19          TWO.pow(300).subtract(BigInteger.valueOf(153)),
20
21          /* n = 2^400 - 593. */
22          TWO.pow(400).subtract(BigInteger.valueOf(593)),
23
24          /* The largest Carmichael number known. */
25          new BigInteger("1590231231043178376951698401")
26      };
27
28      static BigInteger getRandom(BigInteger min, BigInteger max) {
29
30          Random rand = new Random(System.nanoTime());
31          int len = max.bitLength();
32          BigInteger res = new BigInteger(len, rand);
33
34          // return (res + min) % max
35          return res.add(min).mod(max);
36      }
37
38      static BigInteger getPrime(BigInteger min, BigInteger max, int k) {
39
40          min = min.divide(TWO); max = max.divide(TWO);
41
42          // res = (getRandom(min, max) * 2) + 1
43          BigInteger res = getRandom(min, max).multiply(TWO).add(ONE);
44
45          while (!prime_3(res, k))
46              res = res.add(TWO);
47
48          return res;
49      }
50
51      static String convTime(long a, long b) {
52
53          double c = (double)(b - a) / (double)1000000000;
54          return (int)(c / 60) + " min " + (c % 60) + " sec";
55      }
56
57      static boolean prime_1(BigInteger n) {
58
59          // n < 2
```

```
60          if (n.compareTo(TWO) < 0)
61              return false;
62
63          // n % 2 == 0
64          else if (n.mod(TWO).equals(ZERO))
65              return false;
66
67          BigInteger THREE = ONE.add(TWO), i;
68
69          // i = 3; i <= sqrt(n); i += 2
70          for (i = THREE; i.compareTo(n.sqrt()) <= 0; i = i.add(TWO))
71
72              // n % i == 0
73              if (n.mod(i).equals(ZERO))
74                  return false;
75
76          return true;
77      }
78
79      static boolean prime_2(BigInteger n) {
80
81          // n < 2
82          if (n.compareTo(TWO) < 0)
83              return false;
84
85          // n % 2 == 0
86          else if (n.mod(TWO).equals(ZERO))
87              return false;
88
89          BigInteger n_1 = n.subtract(ONE), a = getRandom(TWO, n_1);
90
91          // gcd(a, n) != 1
92          while (!a.gcd(n).equals(ONE))
93              a = getRandom(TWO, n_1);
94
95          if (a.modPow(n_1, n).equals(ONE))
96              return true;
97          else
98              return false;
99      }
100
101     static boolean prime_3(BigInteger n, int k) {
102
```

```
103         // n < 2
104         if (n.compareTo(TWO) < 0)
105             return false;
106 
107         // n % 2 == 0
108         else if (n.mod(TWO).equals(ZERO))
109             return false;
110 
111         BigInteger z = n.subtract(ONE), w = ZERO;
112 
113         while (z.mod(TWO).equals(ZERO)) {
114 
115             z = z.divide(TWO);
116             w = w.add(ONE);
117         }
118 
119         for (int i = 0; i < k; i++) {
120 
121             BigInteger a = getRandom(TWO, n.subtract(ONE));
122 
123             // gcd(a, n) != 1
124             if (!a.gcd(n).equals(ONE))
125                 return false;
126 
127             BigInteger x = a.modPow(z, n), j = ONE;
128             boolean stop = false;
129 
130             // x == 1 || x == n - 1
131             if (x.equals(ONE) || x.equals(n.subtract(ONE)))
132                 continue;
133 
134             // j <= w - 1 && !stop
135             while (j.compareTo(w.subtract(ONE)) <= 0 && !stop) {
136 
137                 // x = (x * x) % n
138                 x = x.multiply(x).mod(n);
139 
140                 if (x.equals(n.subtract(ONE)))
141                     stop = true;
142                 else
143                     j = j.add(ONE);
144             }
145 
```

```
146             if (!stop)
147                 return false;
148         }
149 
150         return true;
151     }
152 
153     public static void main(String[] args) {
154 
155         for (BigInteger i : test_prime_1) {
156 
157             System.out.println("Numero scelto: " + i);
158 
159             long a = System.nanoTime();
160             System.out.print("Trial Division Test: Risultato: " + prime_1(i));
161             long b = System.nanoTime();
162             System.out.println(" Tempo: " + convTime(a, b));
163             System.out.println("--------------------");
164         }
165 
166         for (BigInteger i : test_prime_2_3) {
167 
168             System.out.println("Numero scelto: " + i);
169 
170             long a = System.nanoTime();
171             System.out.print("Fermat Test: Risultato: " + prime_2(i));
172             long b = System.nanoTime();
173             System.out.println(" Tempo: " + convTime(a, b));
174             System.out.println("----------------------------------------");
175         }
176 
177         for (BigInteger i : test_prime_2_3) {
178 
179             System.out.println("Numero scelto: " + i);
180 
181             long a = System.nanoTime();
182             System.out.print("Miller-Rabin Test: Risultato: " + prime_3(i, 30));
183             long b = System.nanoTime();
184             System.out.println(" Tempo: " + convTime(a, b));
185             System.out.println("----------------------------------------");
186         }
187     }
188 }
```