ALGORITMI DI MEMORIZZAZIONE NUMERI PRIMI – RAINBOW PRIME TABLE

Napoli 3-12-2007

Aggiornato il 5-12-2007

BIOGRAFIA

Rosario Turco è un ingegnere elettronico, laureato all'Università Federico II di Napoli, che lavora dal 1990 in società del gruppo Telecom Italia.

Le sue competenze professionali sono in ambito delle architettture hw/sw Object Oriented (OOA/OOP, AOP, SOA, Virtualization) e in generale "Java 2 Enterprise Edition". Ha lavorato molti anni nella progettazione e sviluppo di sistemi informatici su piattaforme Windows/Unix/Linux e con linguaggi Java, C, C++.

Negli ultimi anni si sta particolarmente interessando alla crittografia e alla Teoria dei Numeri.

Sommario

Nel seguito vengono presentati delle modalità con cui si possono implementare semplici algoritmi per memorizzare numeri primi o calcolarli e il cui scopo è semplicemente di individuare l'i-esimo numero primo in tempi ragionevoli.

PRIMI SUL TRAGUARDO

La determinazione dell'i-esimo numero è primo è un problema secolare, specie se il numero di cifre in gioco è notevole.

Si devono memorizzare o si possono calcolare? Quali sono i metodi efficienti ed efficaci sia nel primo caso che nel secondo?

Il calcolo è solitamente la strada preferibile rispetto a quella della memorizzazione, ma si cozza con le seguenti domande: siamo in grado con una formula di sapere esattamente l'i-esimo numero primo, senza far uso di statistica e frequenza dei numeri primi e soprattutto in un tempo ragionevole?

In realtà esistono molti algoritmi a tale riguardo, ma è possibile crearne uno semplice ed efficace? E se dovessimo proporne uno ad un sito che fornisce un servizio sull'i-esimo numero primo?

Nel seguito le strade disponibili.

MEMORIZZAZIONE DEI NUMERI PRIMI

Un tempo si realizzavano delle tabelle cartacee di numeri primi grandi fino a 10 milioni di numeri dispari. Oggi si può usare il computer. Supponiamo di voler memorizzare, in qualche modo (quale?), almeno 10 miliardi di numeri dispari: come si potrebbe fare?

Analizziamo il problema. La sofferenza evidente è lo spazio: si può ridurre lo spazio di memorizzazione senza perdere le informazioni? In pratica sì: memorizzando ad esempio i numeri dispari (il 2 già sappiamo che è un numero primo) in esadecimale (2^4=16bit).

Per cui rappresentando in un esadecimale almeno 16 dispari, il numero non primo lo indico col valore 0 (false) mentre con 1 (true) se il numero è primo; inoltre la posizione del bit mi indica il numero dispari che sto considerando (vedi tabella successiva).

Dispari	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
Numero Primo?	0	1	1	1	0	1	1	0	1	1	0	1	0	0	1	1

Il numero di sopra di 0 e 1 in esadecimale è 76D3.

In questo modo si comprimono di 16 volte i numeri interi da rappresentare. Quindi devo memorizzare solo 625 milioni di numeri interi, valore ottenuto da

$$1x10^{10}/2^4=2^{10}x5^{10}x2^{-4}=2^6x5^6x5^4=5^4x10^6.$$

Qui non ci sono problemi nell'usare in linguaggio C il tipo unsigned long int. Ma come si memorizzano: in memoria RAM o su un DB?

In pratica 625 milioni di interi a 32 bit (4byte) richiedono più di 2,5Gbyte per una memoria RAM. Se disponiamo solo di un PC non è la strada giusta.

Per questo tipo di problema è meglio memorizzarli su un DB e quindi su filesystem. E' sufficiente creare una tabella con i campi:

- PROGRES progressivo dell'esadecimale di tipo sequence
- VAL_ESA il valore dell'esadecimale di tipo NUMBER(11)

Come si trova la posizione POS dove è il numero che ci interessa nell'esadecimale per vedere se è primo o no? Ad esempio 31, 63 o 91?

Sappiamo che i dispari seguono la formula 2n+1. Se 2n+1=31 -> n=15. Ora la parte intera di m = n/15=1 (m rappresenta PROGRES; in questo caso è il primo esadecimale). Nella divisione usiamo 15 perché immaginiamo che i numeri siano memorizzati come in un array logico da 0 a 15.

Ma all'interno dell'esadecimale, qual è la posizione del numero? Pos=15 mod 16 = 15. Infatti 31 è in posizione 15. Accedendo a tale bit, con una maschera di bit (linguaggio C) si determina il valore 0 o 1 e quindi se il numero richiesto è primo.

Se 2n+1=91, n=45, m=45/15=3, pos=45mod16=13

Accedendo al 13 bit si vede se il bit è 0 o 1 e si scopre se è un numero primo o meno.

Ovviamente se dovessimo riempire il DB con tali dati è sufficiente creare un programma C/ProC che li genera. Ma realmente serve la memorizzazione di tutti i numeri primi?

CALCOLO DELL'I-ESIMO NUMERO PRIMO: LE RAINBOW PRIME TABLE

Osservando i risultati ottenibili da un programma, si possono apprendere (poi vediamo cosa intendiamo per "apprendere") i valori di indice da cui iniziare per il conteggio di primalità ed il valore del numero primo corrispondente. In tal modo una terna di valori (i-esimo primo, valore del primo, indice di calcolo) può essere registrata in una *Rainbow Prime Table*. Questo permette, a questo punto, una buona accelerazione nella ricerca dell'i-esimo numero primo in determinati intervalli memorizzati (ad esempio memorizzando solo i valori ogni milione). Il programma, non disponendo degli infiniti numeri primi (cosa impossibile) è così una via di mezzo tra:

- calcolo per i valori non disponibili, estraendo il valore più prossimo di riferimento da cui partire (e quindi accelerando: "effetto turbo-primi")
- estrazione del valore di riferimento, invece, disponibile.

E' evidente che il programma è veloce se il numero primo i-esimo non disponibile è al massimo maggiore di un milione dell'ultimo disponibile; ulteriormente aumenta il tempo di calcolo.

Inizialmente si può produrre un generatore che utilizza il tipo unsigned long int riesce a memorizzare una buona quantità di milioni di numeri primi fin quando il valore del numero primo e dell'indice rimane al di sotto di 2^{32}-1 se siamo su una architettura a 32 bit o 2^{64}-1 se siamo su una architettura a 64 bit. Per superare il limite architetturale si possono sviluppare algoritmi ad hoc per trattare numeri come stringhe di qualunque lunghezza di cifre.

L'idea dell'algoritmo nasce dal fatto che se generassimo l'i-esimo numero solo con una tecnica a "forza bruta" per raggiungere l'i-esimo cercato ci vorrebbe troppo tempo.

L'algoritmo prende a prestito qualche idea dal mondo hacker, cioè l'idea di sfruttare delle tabelle analoghe alle RAINBOW TABLE, che per l'occasione vengono da me definite RAINBOW PRIME TABLE.

L'algoritmo permette sia di individuare dati contenuti nel suo database header (rainbow.h), sia di ricavarseli basando il suo calcolo a partire dall'ultimo dato disponibile per velocizzare. Il suo database è una RAINBOW PRIME TABLE dove sono memorizzati solo i-esimi primi a salto di un milione. Se il dato è contenuto negli intervalli di valori previsti dal suo database o ad un salto di un milione del suo ultimo dato disponibile, il tempo di individuazione dell'i-esimo numero primo è ottimizzato da pochi secondi ad un massimo di 15 minuti per un Pentium IV. Se il salto diventa maggiore del milione il tempo di ricerca si allunga.

Se si modifica il programma per lavorare su un DB (quindi apprendere i numeri primi a passo di 1 milione) e lo si "pilota" attraverso un main che ciclicamente gli chiede salti di un milione alla volta, il programma stesso genera il database, fino a che si arriva al problema dell'unsigned long int di cui prima, che se si vuole andare oltre occorre aggiungere software ad hoc non difficile da implementare.

L'algoritmo nel caso migliore fa solo m passi, dove m è il numero di elementi nella RAINBOW PRIME TABLE. Nel caso peggiore fa m+1+N-m passi.

L'algoritmo fornito, secondo un'analisi computazionale, è comunque circa O(N) per l'inserimento su DB, ma fa un solo accesso al DB se deve cercare il primo più vicino ed è O(N) nel calcolo. In assenza del numero primo, una volta calcolato può registrarlo su DB.

Tale tipologia di algoritmo è applicabile, ad esempio, su siti che possono fornire il servizio di "trovare l'i-esimo numero primo", con i-esimo anche molto grande.

APPENDICE

Nel seguito vediamo un algoritmo in C, che per semplicità sfrutta una memorizzazione statica in un file header (rainbow.h) e che utilizza l'unsigned long int dell'architettura. I due limiti sono eliminabili facilmente utilizzando un Db nel primo caso e non il file

header, e librerie sw che trattano i numeri come stringhe. Il secondo punto è descritto nell'articolo "Analisi computazionale - Metodo di superamento dei limiti hardware a 32 e 64 bit".

Nel seguito l'esempio.

Sorgente main.c

```
#include <stdlib.h>
#include "prototype.h"


int main(int argc, char *argv[])
{
 long int Pos=0;


 printf("\nAlgoritmo di generazione numero primo i-esimo con 2n+1\n");

 do{

     Pos = 0;
     printf("\nInserire l'i-esimo del numero primo da ottenere (0 per uscire): ");
     scanf("%d",&Pos);

     if(Pos!=0){
       printf("\nElaborazione in corso ...\n");


       (void) genprime(Pos);
       system("PAUSE");
     }

 }while(Pos != 0);

 return 0;
}
```

Sorgente prototype.h

```
#if !defined prototype_prime
#define prototype_prime


static unsigned long int NumItem=0;

typedef struct{
        unsigned long int PiInit1;
        unsigned long int PiEnd1;
        unsigned long int PiInit2;
        unsigned long int PiEnd2;
       }SUBSET;

int prime(unsigned long int iVal, int iDeb);
unsigned long int CountPidiN(unsigned long int iValue);
int SxDuePrimiOpt(unsigned long int iValue, int smp);

void SxDuePrimiSubset(SUBSET *p, unsigned long int iValue);
unsigned long int gen6n(unsigned long int Value);
unsigned long int gen2n(unsigned long int Value);
void genprime(unsigned long int Pos);

#endif
```

Sorgente rainbow.h

```
#if !defined RAINBOW_DEFINE
#define RAINBOW_DEFINE

#define RAINBOW_LENGHT  22

typedef struct{
      unsigned long int CountPrime;
      unsigned long int ValuePrime;
      unsigned long int Index;
      }RAINBOWPRIMETABLE;

RAINBOWPRIMETABLE rainbow[RAINBOW_LENGHT]={
                  100,541,271,
                  1000,7919,3960,
                  10000,104729,52365,
                  100000,1299709,649855,
                  1000000,15485863,7742932,
                  2000000,32449829,16224915,
                  2200000,35922457,17961229,
                  2500000,41157019,20578510,
                  3000000,49973837,24986919,
                  4000000,67859573,33929787,
                  5000000,86017553,43008777,
                  6000000,104382461,52191231,
                  7000000,122935271,61467636,
                  8000000,141634967,70817484,
                  9000000,160464901,80232451,
                  10000000,179406803,89703402,
                  11000000,198473557,992336779,
                  12000000,2006081257,1003040629,
                  13000000,2027502493,1013751247,
                  14000000,2048941799,1024470900,
                  15000000,2070395441,1035197721,
                  16000000,2091849247,1045924624
                  };
#endif
```

Sorgente gen2n.c

```
#include <stdio.h>
#include <stdlib.h>
#include "prototype.h"

unsigned long int gen2n(unsigned long int Value){

 return( (2*Value) + 1 );
}
```

Sorgente genprime.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "prototype.h"
#include "rainbow.h"

void genprime(unsigned long int Pos){

  unsigned long int iNew=0, i=0, j=0;
  unsigned long int Init=0, End=0, flag_coincidenza = 0;


  if( Pos != 0 ){

    /* Prendo il tempo per una valutazione prestazionale */

    /* L'acceleratore */
      if(Pos < 100 ){
          Init = 1;
          End=Pos;
```

```
    }
    else{

  /* Faccio uso della RainbowPrimeTable */
       /* Cerco uno che coincide prima */
       for(j=0;j<RAINBOW_LENGHT;j++){
          if(Pos==rainbow[j].CountPrime){

            /* Ho trovato l'elemento in tabella */
            Init=rainbow[j].Index;
            End=Init;
            i=End;          /* devo saltare il ciclo successivo di generazione primi */
            iNew =rainbow[j].ValuePrime;
            flag_coincidenza = 1;
            break;          /* interrompo subito il ciclo attuale */
           }

       }
       if( !flag_coincidenza ){/* Se non ho trovato il valore esatto */

       /* Adesso visto che non coincide con nulla, cerco gli intervalli */
         for(j=0;j<RAINBOW_LENGHT;j++){

            /* Se + maggiore anche dell'ultimo disponibile in tabella */
            if((Pos>rainbow[j].CountPrime)&& (j == RAINBOW_LENGHT-1)){

               Init=rainbow[j].Index;
               End=Init+Pos-rainbow[j].CountPrime;
               break;
            }
            else
            {

              if((Pos>rainbow[j].CountPrime) &&(Pos<rainbow[j+1].CountPrime)){
                 Init=rainbow[j].Index;
                 End=Init+Pos-rainbow[j].CountPrime;

              }

             }
          }
       }
    }


    for(i=Init; i<End; i++){/* cilco di generazione prim dispari */

      iNew = gen2n(i);


      if( prime(iNew, 0 ) == 1){  /* proseguo al sucsessivo per trovate l'i-esimo */
        /* stampaOne(iNew,8); */
      }
      else{
        End++; /* mi amplio l'intervallo perchè ho scartato un numero dispari non primo e devo sempre trovare l'iesimo */
      }
    }

   /* L'ho trovato e lo stampo */
   printf("\n\nPosizione richiesta    : %d\n", Pos);
   printf("\nNumero primo richiesto : ");
   stampaOne(iNew,8);
   printf("\nValore Indice          : %d", i);
   printf("\n\n");
  }
  return;
}
```

Sorgente prime.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

unsigned long int prime(unsigned long int iVal, int iDeb)
{
  int i=0, iCount=0;
  unsigned long int iSq=0, iPri=0;

  /* iCount conta i Divisori */

  float fVal = iVal;

  if ( (iVal == 0)||(iVal ==1) ) return iPri;  /* si esclude lo zero e l'1*/

  iSq = sqrt(fVal);

  if( iDeb > 0 )
    printf("\n sqrt : %d\n", iSq);

  if( iVal%2 == 0){

     iCount++;
  }
  else{
    /* Cerco i divisori con il modulo escludendo l'1 e il 2*/

  for(i=3;i<=iSq;i=i+2){ /* Cerco i divisori dispari */
       if( iVal%i == 0) {
         iCount++; /* Esiste un divisore, allora non è primo */
         break;    /* Interrompo al primo Divisore trovato */
       }
   }
  }
  if (iCount == 0 ) iPri=1;  /* Se iCount = 0 allora è primo */

  return iPri;
}
```

Sorgente stampaone.c

```
#include "prototype.h"

void stampaOne(unsigned long int Val1, int num){

     if( NumItem < num ){
       printf("%d\t", Val1);
       NumItem++;
     }
     else{
       NumItem = 1;
       printf("\n");
       printf("%d\t", Val1);
     }
     return;
}
```