Verifica di Primalità - Progetto per l'esame di Algoritmi e Strutture Dati II

Luca Cinti Lorenzo Masetti

30 agosto 2000

Indice

1 Obiettivi del progetto

Lo scopo di questo progetto è quello di testare l'efficienza di alcuni algoritmi probabilistici per la ricerca di numeri primi molto grandi. La ricerca di numeri primi molto grandi è un problema molto studiato e che trova la sua applicazione più importante nella crittografia con il metodo *RSA*. Un problema collegato alla ricerca di numeri primi molto grandi ma molto più difficile da risolvere è quello della ricerca dei fattori primi di un numero composto molto grande. Proprio sul differente ordine di complessità di questi due problemi si basa infatti il metodo crittografico *RSA*.

Gli algoritmi di verifica della primalità che abbiamo implementato sono la *verifica di pseudoprimalità di base a* e la *verifica di primalità randomizzata di Miller-Rabin*. Questi due algoritmi sono probabilistici quindi quando rispondono positivamente non danno la sicurezza assoluta che il numero trovato sia primo, ma la percentuale di errore, specialmente nel caso del secondo metodo, è talmente bassa da rendere queste procedure ottime per tutti gli scopi pratici.

Per illustrare il secondo problema a cui abbiamo fatto cenno, la ricerca della scomposizione di interi in fattori primi, abbiamo implementato l'*Euristica rho di Pollard*.

I numeri primi sono, fortunatamente, abbastanza frequenti quindi la ricerca di un numero primo di un numero di cifre fissato richiede un numero di prove abbastanza basso. Infatti se consideriamo la *funzione di distribuzione dei numeri primi*, $\pi(n)$, definita per ogni naturale n come il numero di primi minori uguali a n, un importante teorema afferma che:

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

Da questa relazione otteniamo l'approssimazione $\pi(n) = n/\ln n$ che ci dice che la probabilità che un numero n scelto casualmente sia primo è $1/\ln n$; quindi si dovranno controllare approssimativamente $\ln n$ numeri vicini a n per trovarne uno primo. Ad esempio trovare un numero primo di 100 cifre dovrebbe richiedere la verifica di circa $\ln 10^{100} \approx 230$ numeri.

2 Rappresentazione di interi molto grandi

2.1 La rappresentazione decimale

Il primo problema che si presenta nel trattare numeri interi molto grandi è quello della loro rappresentazione. Infatti rappresentando gli interi su 32 bit come nel caso dei *longint* del *Pascal*, il massimo intero rappresentabile è $2^{31} - 1 = 2147483647$[1], un numero molto piccolo rispetto alle 100 cifre che ci eravamo proposti di trattare. Si è reso quindi necessario adottare una rappresentazione diversa degli interi ed implementare tutte le operazioni aritmetiche via software.

La soluzione che abbiamo scelto prevede di rappresentare gli interi con un array di 100 byte, ognuno dei quali può assumere i valori 0..9. Questa rappresentazione comporta un evidente spreco di spazio dal momento che per rappresentare un intero in questo modo si occupano 100 byte (800 bit) (mentre una rappresentazione binaria di numeri dello stesso ordine di grandezza occuperebbe $\log_2 10^{100} \approx 322$ bit) ma presenta il vantaggio di poter applicare facilmente gli algoritmi che si usano comunemente quando si eseguono le operazioni con carta e penna.

2.2 La unit Mathe

La unit *Mathe* contiene le procedure per la gestione dei numeri rappresentati come array di cifre decimali. Contiene la dichiarazione del tipo *RecNo* che verrà utilizzato al posto del normale tipo intero in tutte le procedure che implementano gli algoritmi per la ricerca di numeri primi.

```
const NoCfr = 100;
type RecNo = array [1..NoCfr] of byte;
```

Per alcuni algoritmi si è resa necessaria anche una rappresentazione binaria dei numeri. La rappresentazione binaria è simile a quella decimale: naturalmente le cifre possono assumere solo i valori 0 o 1 ed il numero di cifre è dato da $\text{NoCfr} * (\log_2(10)) \approx 3.322\text{NoCfr}$

```
const BinCfr = (Trunc(NoCfr*3.322)) + 1;
BinNo = array [1..BinCfr] of byte;
```

[1]Utilizzando il tipo *cardinal* del *Free Pascal* (interi senza segno a 32 bit) è possibile arrivare a $2^{32} - 1 = 4294967295$

Alcune procedure elementari

InitNo provvede a inizializzare un numero a zero, ponendo tutte le sue cifre uguali a zero.

Procedure WriteNo Visualizza un numero a video.

Procedure ReadNo Legge un numero da tastiera. La procedura legge una stringa di caratteri da tastiera e li pone nella maniera opportuna nella variabile passata come parametro, controllando anche che la stringa letta contenga effettivamente solo valori numerici.

Procedure AssegnaNo è simile a *ReadNo* con la differenza che la stringa da convertire non è letta da tastiera ma viene passata come parametro.

Procedure Let Pone il primo numero passato come parametro uguale al secondo parametro.

Function EqNo Restituisce vero se i numeri passati come parametro sono uguali, attraverso un confronto dei valori di ogni elemento dei due numeri.

Procedure RandNo Restituisce un numero casuale assegnando un numero casuale compreso tra 0 e 9 ad ogni cifra del numero.

Procedure RandomNo Restituisce un numero casuale minore o uguale del primo parametro passato alla procedura.

Calcolo del numero di cifre

In questo tipo di rappresentazione è importante sapere quante sono le cifre significative di un numero. Questa informazione è fornita dalla function *CalcCfr* che scandisce il numero da sinistra a destra fermandosi quando trova la prima cifra diversa da 0. Se il numero è 0 *CalcCfr* restituisce 0, quindi la condizione $CalcCfr(N) = 0$ equivale a $N = 0$.

Addizione

L'addizione viene effettuata dalla procedura *Sum* che prende i due numeri da sommare come primi due argomenti e restituisce il risultato nel terzo argomento.

L'algoritmo utilizzato è semplice: si sommano le cifre dei due numeri da destra a sinistra; se il risultato è maggiore di 10 viene decrementato di 10 e viene impostata una variabile booleana che indica che alla prossima iterazione bisognerà aggiungere un'unità alla somma delle due cifre a causa del riporto.

Sottrazione

La sottrazione viene eseguita dalla function *Dif* che ha questa intestazione

```
Function Dif(A, B: RecNo; var Ris: RecNo): boolean;
```

La funzione viene normalmente chiamata come procedura senza salvare il valore booleano restituito (è una possibilità che il *Turbo Pascal* mette a disposizione a partire dalla versione 7.0). In questo caso dev'essere $A >= B$ e nel terzo parametro viene restituita la differenza tra i due interi. Il valore booleano restituito dalla funzione viene utilizzato per i confronti infatti *Dif* restituisce `TRUE` se $A < B$, `FALSE` altrimenti.

La procedura funziona in modo simile all'addizione, scandendo i due numeri da destra a sinistra e calcolando la differenza cifra per cifra. Naturalmente in questo caso si avrà un riporto negativo nel caso che il risultato della differenza tra le due cifre sia negativo. Se il riporto si propaga fino all'ultima cifra vuol dire che $A < B$ quindi il valore che la funzione deve restituire è uguale al valore della variabile booleana *Rip* alla fine del ciclo.

La differenza viene richiamata dalle due funzioni *Greater(A,B)* e *Less(A,B)* che corrispondono rispettivamente a $A > B$ e $A < B$.

Incremento e Decremento di uno

Le due procedure *IncrUnoNo* e *DecrUnoNo* lavorano in modo simile ad addizione e a sottrazione con la differenza che il secondo termine è dato da 1 e quindi ci si può fermare non appena non si ha il riporto.

Divisione per 2

Per ottenere la metà di un numero si scandisce a partire dalla cifra più significativa verso destra, dividendo tutte le cifre a metà e segnalando un riporto se la cifra era dispari. Un riporto provocato dall'ultima cifra sta ad indicare che il numero è dispari. La funzione restituisce 0 o 1 per indicare che il numero è pari o dispari.

Moltiplicazione

La funzione di moltiplicazione è fornita dalla procedura *MulNo*, che sfrutta l'algoritmo elementare per la moltiplicazione, moltiplicando attraverso somme successive ogni cifra del moltiplicatore per il moltiplicando. I risultati di questa operazione vengono poi sommati dopo un opportuno shift a sinistra di un numero di volte pari alla posizione della cifra corrente del moltiplicando. La funzione di shift a sinistra è messa a disposizione dalla procedura ShiftNo,

che stabilisce anche se tale shift porta ad un overflow nella moltiplicazione. Questo algoritmo risulta essere naturalmente più veloce di quello derivante dalla definizione di prodotto come somma successiva del moltiplicando per un numero di volte uguale al moltiplicatore, che renderebbe il prodotto tra numeri talmente lento da impedire nella pratica gli algoritmi di verifica della primalità e di scomposizione in fattori primi di un numero.

Divisione

La funzione di divisione è implementata dalla procedure *DivNo*, la quale sfrutta l'algoritmo di divisione elementare per restituire il quoziente ed il resto tra due numeri A e B. La procedura assume che sia $B \neq 0$. Questa convenzione è comunque rispettata in tutti i programmi del nostro progetto, che verificano questa condizione prima di effettuare una chiamata a DivNo. Inizialmente si controlla che $B < A$ ed in caso contrario si assegna semplicemente 0 al risultato e il valore di A al resto. L'algoritmo inizialmente si preoccupa di "abbassare" il numero opportuno di cifre del dividendo, ponendo nella variabile *Temp* un numero di cifre di A pari al numero di cifre di B. A questo punto se Temp è ancora minore di B, si esegue uno shift a sinistra del numero Temp con la funzione *ShiftNo* e si abbassa un'altra cifra di A ponendola nella prima cifra di Temp. Dopo queste operazioni la prima cifra del quoziente è uguale alla divisione tra Temp e B, restituita dalla procedura *Div10*. Il resto di tale divisione va shiftato di una posizione e sostituito con la prossima cifra di A e richiamando su tale numero la funzione *Div10* si otterrà una nuova cifra del quoziente e un nuovo resto. Questo procedimento viene iterato attraverso un `for` per tutte le rimanenti cifre di A fino a quando non si esauriscono le cifre del dividendo in modo tale che Q sarà il quoziente e l'ultimo resto sarà il resto della divisione tra A e B.

Massimo Comun Divisore

La procedura di divisione viene sfruttata nella procedura *MCDNo* per calcolare il *Massimo Comun Divisore* tra due numeri. L'algoritmo utilizzato è quello di Euclide.

Trasformazione di un numero nella sua rappresentazione binaria

La trasformazione binaria di un numero, implementata dalla procedura *TrasfBin* è una funzionalità molto importante per il nostro progetto perché è sfruttata dagli algoritmi per la verifica della primalità e per la scomposizione in fattori primi. L'algoritmo seguito è semplice. Consiste nel dividere il numero A da trasformare in binario per due fino a farlo diventare 0. I resti

di tali rappresentazioni presi in ordine contrario rappresentano proprio il numero A in binario. I resti sono forniti dalla funzione *DivDue(A,B)* che riceve un numero A, pone la metà in B e restituisce il resto della divisione. Questa funzione, chiamata con i parametri DivDue(A,A), finché A=0 fornisce la successione di resti che costituiscono la rappresentazione binaria di A.

```
Procedure TrasfBin(A: RecNo; var B: BinNo);
var i, j: integer;
begin
   for i:= 1 to BinCfr do B[i]:= 0;
   i:= 1;
   repeat
      B[i]:= DivDue(A,A);
      i:=i+1;
   until CalcCfr(A)=0;
end;
```

Calcolo della radice quadrata

L'euristica rho di Pollard richiede il calcolo della radice quadrata approssimata di un intero (in realtà il valore che serve è $\sqrt[4]{n}$ che si può facilmente ottenere applicando due volte la radice quadrata). Per trovare la radice quadrata abbiamo usato l'algoritmo d'Erone, che consiste nel calcolarsi i termini di una successione che converge alla radice quadrata di x.

Detto x il numero di cui bisogna calcolare la radice, la successione è definita per ricorrenza da

$$a_1 = x$$
$$a_{n+1} = \frac{(a_n + x/a_n)}{2}$$

In realtà, perché la successione converga a $\sqrt{x}$, non dev'essere necessariamente $a_1 = x$, è sufficiente che sia $a_1 > \sqrt{x}$. Per essere sicuri che questa condizione sia soddisfatta, la scelta più semplice è quella di porre $a_1 = x$.

Serve però anche una stima dell'errore per sapere quando a_n è un'approssimazione accettabile di $\sqrt{x}$.

Si dimostra che l'errore sul termine n-esimo è maggiorato da

$$a_{n-1} - a_n$$

cioè vale la diseguaglianza

$$a_n - \sqrt{x} < a_{n-1} - a_n$$

L'algoritmo per il calcolo della radice può essere descritto dal seguente frammento di programma *Pascal*: prima si controlla se $x = 0$ (e in quel caso $\sqrt{x} = 0$), altrimenti si entra in un ciclo `repeat` che assegna a r (che è inizializzata a x) il valore del termine successivo della successione e mantiene in p il valore precedente di x. Si esce dal ciclo quando (p - r) (cioè $a_{n-1} - a_n$, l'errore) è minore di 1, che è la massima precisione possibile lavorando nell'insieme dei naturali.

```
if x=0 then RadQuadr:=0
else begin
  r:=x;
  repeat
       p:=r;
       r:=(r+x/r)/2;
  until (p-r)<1;
  RadQuadr:= r
end;
```

L'adattamento di questa procedura ai numeri rappresentati in base 10 utilizza le procedure di moltiplicazione e divisione descritte precedentemente. Ecco il codice per la procedura *RadQuadr*.

```
Procedure RadQuadr(x:RecNo; var R:RecNo);
var p,t,inut, Uno: RecNo;

begin
    InitNo(R);
    AssegnaNo(Uno,'1');
    if CalcCfr(x)<>0 then
    begin
     Let(R,X);
     repeat
           Let(p,r);
           {r:=(r+x/r)/2 }
           DivNo(x,r,t,inut);
           Sum(t,r,t);
           DivDue(t,r,t);
           Dif(p,r,t);
     until not greater(t,uno);
     end;
end;
```

3 Algoritmi per la verifica di primalità

3.1 Pseudoprimalità

Questo algoritmo si basa sul *Teorema di Fermat*, che afferma che, se n è primo si ha:

$$a^{n-1} \equiv 1 \text{ (mod n)} \; \forall a \in \mathbb{Z}_n^+$$

Se un numero n soddisfa il teorema di Fermat solo per un particolare $a \in \mathbb{Z}_n^+$ si dice che n è uno *pseudoprimo di base a*.

Un numero che non soddisfa la relazione $2^{n-1} \equiv 1$ (mod n) è sicuramente composto; un numero che soddisfa la relazione (e quindi è primo o pseudoprimo di base 2) viene dichiarato primo dalla procedura *pseudoprimo*. La convenzione è la stessa utilizzata dalla procedura *MillerRa*:

$$pseudoprimo(n) = \begin{cases} \texttt{TRUE} & \text{se } n \text{ è composto (sicuramente)} \\ \texttt{FALSE} & \text{se } n \text{ è primo (o pseudoprimo di base 2)} \end{cases}$$

```
function pseudoprimo(n: longint): boolean;
begin
   pseudoprimo:= modulexp(2,n-1,n) <> 1;
end;
```

La function *modulexp(i,n,m)* calcola i^n (mod m) in modo da evitare l'overflow che si verificherebbe calcolando prima i^n e poi prendendo il resto modulo m. Un ciclo scandisce la rappresentazione binaria di n, dal bit meno significativo a quello più significativo, salvando in d il valore delle potenze crescenti di 2 modulo m e, se l'i-esimo bit della rappresentazione binaria di n è 1, moltiplicando il risultato parziale per d cioè per 2^i (mod m).

Per chiarezza riportiamo qui la versione della procedura *modulexp* che opera sui longint. La versione che opera sui numeri in base 10 di cento cifre è del tutto simile, anche se presuppone una doppia rappresentazione del numero n, che deve essere precedentemente convertito in binario. La function *bit(n,i)* restituisce l'i-esimo bit della rappresentazione binaria di n.

```
function modulexp(i, n, m: longint): longint;
{Calcola (i^n) mod m}
var d : longint;
    j : integer;
    ris : longint;
begin
        d:=i mod m;
```

```
        ris:=1;
        for j:=0 to 31 do begin
                if bit(n,j)=1 then
                    ris:=ris*d mod m;
                d:=d*d mod m;
                end;
        modulexp:=ris;
end;
```

La function *pseudoprimo* sbaglia abbastanza raramente. Vi sono solo 22 valori minori di 10000 (e 78 valori minori di 100000 come abbiamo verificato nelle simulazioni) per i quali l'algoritmo fornisce una risposta sbagliata. Il più piccolo di tali numeri è 341, che è composto ($11 * 31 = 341$), ma soddisfa la relazione ($2^{340} \equiv 1 \pmod{341}$). Inoltre la probabilità che un numero n sia uno pseudoprimo di base a (nel nostro caso $a = 2$) tende a zero se $n \to \infty$; in altre parole più grande è n minore è la probabilità che sia uno pseudoprimo di base a.

3.2 Verifica di primalità randomizzata di Miller-Rabin

Un primo modo di migliorare la procedura *pseudoprimo*, è quello di controllare se il numero è pseudoprimo per più valori della base a, e non solo per la base 2. Sfortunatamente però esistono alcuni interi composti, detti *numeri di Carmichael* che soddisfano l'equazione $a^{n-1} \equiv 1 \pmod{n})\ \forall a \in \mathbb{Z}_n^+$. Questi numeri sono estremamente rari; i primi tre sono 561 ($= 11 * 3 * 17$), 1105 e 1729.

La verifica di primalità di Miller-Rabin risolve questi problemi con due modifiche della funzione *pseudoprimo*.

- Prova con più valori a di base invece che con il solo valore 2
- Mentre calcola a^{n-1} controlla se ha scoperto una radice quadrata non banale di 1 modulo n.

Come prima presentiamo per semplicità il codice della procedura che lavora con i longint. La function MillerRa restituisce `TRUE` se il numero è composto, `FALSE` se è (quasi sicuramente) primo.

```
function MillerRa(n: longint; s: integer): boolean;
var j: integer;
    a: longint;
```

```
function Witness(a,n: longint): boolean;
var d, i, x: longint;
begin
     Witness:= false; d:=1;
     for i:= 31 downto 0 do
     begin
          x:= d;
          d:= (d*d) mod n;
          if (d=1) and (x<>1) and (x <> n-1) then
              Witness:= true;
          if bit(n-1, i) = 1 then d:= (d*a) mod n;
     end;
     if (d<>1) then Witness:= true;
end;

begin
     MillerRa:= false;
     for j:= 1 to s do
     begin
          a:= random(n-2) + 1; {  1 <= a <= n-1 }
          if witness(a, n)
             then MillerRa:= True;
     end;
end;
```

La routine *witness* esegue un contollo simile a $a^{n-1} \equiv 1$ (mod n) ma più raffinato. Viene chiamata per s valori diversi e casuali della base a e restituisce `TRUE` se a è testimone del fatto che n sia composto (basta infatti che n non soddisfi la relazione per un solo a per dire che n è composto).

Witness è un miglioramento della procedura *modulexp* descritta precedentemente. Calcola a^{n-1} (mod n) con lo stesso metodo di *modulexp* ma controlla anche, ad ogni passo, se è stata trovata una radice quadrata non banale di 1 (la condizione è $d = 1 \wedge x \neq 1 \wedge x \neq n-1$) e in quel caso restituisce `TRUE` perché si può dimostrare che solo se n è composto può esistere una radice quadrata non banale di 1 modulo n.

La procedura funziona correttamente fino a che d è minore della radice quadrata del massimo intero rappresentabile, per valori più grandi di d si verifica un overflow durante il calcolo di $d * d$ che porta a risultati errati.

MillerRa effettua una ricerca probabilistica, infatti se n è primo la procedura restituirà comunque `FALSE`, ma se n è composto la probabilità che

MillerRa restituisca `TRUE` cresce all'aumentare del numero di prove s. In particolare si può dimostrare che

> Per un qualunque intero dispari $n > 2$ e intero s positivo, la probabilità che *MillerRa(n,s)* dia un errore è al più 2^{-s}

Questo errore è così piccolo da rendere la procedura *MillerRa* utilizzabile ai fini pratici anche con valori piccoli di s.

4 Scomposizione di interi in fattori primi

Le verifiche di primalità descritte precedentemente permettono di stabilire che un intero n è composto ma non forniscono i fattori primi di n. Questo problema è molto più difficile e a tutt'oggi, anche con l'utilizzo di calcolatori estremamente potenti, non si è andati oltre la scomposizione di un numero arbitrario di 200 cifre decimali.

Un algoritmo banale per trovare i fattori primi di un numero dispari n consiste nel provare a dividerlo per tutti gli interi dispari fino a $\sqrt{n}$ (in realtà basterebbe controllare se è divisibile per tutti i numeri *primi* $\leq \sqrt{n}$, ma questo metodo presuppone di aver determinato tutti i numeri primi $\leq \sqrt{n}$ e quindi non è praticabile).

4.1 Euristica rho di Pollard

La prova di divisione richiede, per scomporre un numero n, un lavoro proporzionale a $\sqrt{n}$. La procedura *PollRho* ha invece una complessità dell'ordine di $\sqrt[4]{n}$ e, anche e il suo successo non è garantito (per questo è detta euristica), essa è in pratica molto efficace.

Ecco l'implementazione di *PollRho* per i longint. La function restituisce `TRUE` se ha trovato almeno un fattore di n. I fattori che vengono trovati sono salvati in un array (definito globale nella unit) nel quale vengono inseriti dalla procedura *InsFatt*. MCD è una procedura che calcola il massimo comun divisore di due interi.

```
function PollRho(n: longint): boolean;
var x, y, k, i, d: longint;
    trovato: boolean;

procedure InsFatt(x: longint);
begin
     j:= 1;
```

```
     while (fatt[j] <> x) and (j < ind) do j:= j+1;
     if j = ind then
        begin fatt[j]:= x; ind:= ind+1;
                    trovato:= true end;
end;

begin
   Trovato:= false;
   ind:= 1;
   i:= 1;
   x:= random(n-1);
   y:= x;
   k:= 2;
   while i < trunc(sqrt(sqrt(n))) do
   begin
        i:= i+1;
        x:= (sqr(x) - 1) mod n;
        d:= MCD(y - x, n);
        if (d<>1) and (d<>n) then InsFatt(d);
        if i=k then begin y:= x; k:= 2*k end;
   end;
   PollRho:= trovato;
end;
```

La procedura calcola nella variabile `x` l'iesimo termine (dove `i` è il contatore incrementato ad ogni iterazione del ciclo `while`) della ricorrenza

$$x_i = (x_{i-1}^2 - 1) \bmod n$$

(il valore di base x_1 è scelto casualmente in $\mathbb{Z}_n$: in pratica la ricorrenza genera una successione di numeri pseudocasuali in $\mathbb{Z}_n$).

Nella variabile `y` vengono salvati i valori di x_i per i quali i è una potenza di 2. Per far questo si utilizza la variabile `k` che è inizializzata a 2 e viene raddoppiata ogni volta che $i = k$ per trovare la prossima potenza di 2. I fattori per cui si prova a dividere n sono dati da $MCD(y, x)$

Poiché, nella ricorrenza utilizzata, x_i dipende solo da x_{i-1} può darsi che la procedura *PollardRho* entri in un loop e controlli sempre gli stessi divisori (da questo comportamento deriva il nome dell'euristica perché se si rappresentano graficamente i valori x_i si ottiene un disegno simile alla lettera greca ρ dove il "corpo" di ρ è il ciclo).

Il ciclo viene ripetuto per $\sqrt[4]{n}$ iterazioni, perché si dimostra che è probabile che la procedura trovi almeno un fattore di un numero composto dopo $\sqrt[4]{n}$

passi. Questa condizione di terminazione del ciclo porta ad avere una complessità inferiore di quella dell'algoritmo banale che si traduce nella pratica in tempi di esecuzione accettabili, almeno per numeri non troppo grandi.

5 Simulazioni

5.1 Simulazione per gli algoritmi di verifica della primalità

5.1.1 Attendibilità degli algoritmi di verifica della primalità, prima versione (per i numeri fino a 2^{16})

Nel file *simulazione.pas*[2] contenuto nella cartella *cardinal* si trova il codice del programma Pascal per la verifica dell'attendibilità degli algoritmi di pseudoprimalità e di Miller-Rabin.

Per testare queste due procedure si rende necessario determinare tutti i numeri primi fino ad un valore fissato con una procedura "sicura", che non abbia cioè i margini di incertezza propri degli algoritmi probabilistici.

Un possibile algoritmo per trovare i numeri primi è il *crivello di Eratostene*, che esegue i seguenti passi:

1. Si inizializza un array di booleani, `primo[2..Max]` a `true` per i numeri dispari, a `false` per i numeri pari maggiori di 2.

2. Si esegue un ciclo: alla sua prima iterazione si pone `i=3`. Si settano a `false` i multipli di i a partire da i^2. Quindi si ricerca il prossimo elemento per cui $primo[i] = true$ (che sarà il numero primo successivo) e si ritorna ad eseguire il ciclo con questo nuovo valore di i. Il ciclo continua cancellando tutti i multipli dei numeri primi "superstiti" fino a quando $i^2 > Max$.

```
Procedure InitPrimo;
var i,k : cardinal;
    x,y : integer;
begin
     Writeln('Ricerca dei numeri primi da 2 a ',Max);
     primo[2]:=true;
```

[2]Questo programma non può essere compilato con il *Turbo Pascal* che non permette la definizione di array indicizzati dai longint. Altri compilatori, come ad esempio il *Free Pascal* (disponibile gratuitamente per Linux e per Windows) permettono questo tipo di dichiarazione. Il tipo *cardinal* utilizzato nel programma è un intero a 32 bit senza segno; questo tipo di intero non è disponibile nel Turbo Pascal 7.0 ma solo nel Free Pascal.

```
     i:=3;
     while i<=Max do begin
        primo[i]:=true;
        primo[i+1]:=false;
        i:=i+2;
        end;
     i:=3;  k:=9;
     repeat
           Write('Escludo i multipli di ',i,':' );
           GetCoords(x,y);
           while k<=Max do begin
(*non sono primi i multipli di i (primo) a partire da k=i^2*)
                 gotoxy(x,y);Write(k:9);
                 primo[k]:=false;
                 k:=k+i;
           end;
           writeln;
           repeat
                 i:=i+1  (*trova il prossimo numero primo*)
           until primo[i];
           k:=i*i;
     until (k>Max);
end;
```

La procedura *Riempi* scrive in altri due array di booleani di lunghezza Max il risultato delle procedure Pseudoprimo e MillerRa per i numeri dispari da 3 a `Max`.

```
i:=3;
while (i<=Max) do begin
      gotoxy(x,y); Write((i div 2):6,'/',(Max div 2));
      pseudo[i]:= not PseudoPrimo(i);
      miller[i]:= not MillerRa(i,s);
      i:=i+2;
      end;
```

La procedura *Confronto* confronta il contenuto di questi due array con quello dell'array *primo*, riempito da $InitPrimo$. I numeri per cui la procedura *PseudoPrimo* sbaglia sono inseriti nell'array *errori*.

Descrivendo la procedura *MilllerRa* (sez. 3.2, pag. 9) abbiamo detto che l'overflow si verifica per valori di n maggiori della radice quadrata del massimo numero rappresentabile. Su 32 bit in aritmetica naturale (cioè utilizzando

il tipo `cardinal`) il massimo intero rappresentabile è $2^{32} - 1$, la sua radice quadrata è circa $2^{16} = 65536$. Scegliendo $Max = 65000$ ci mettiamo al sicuro da qualsiasi possibilità di overflow.

```
countPseudo:=0; countMiller:=0; totprimi:=1;
i:=3; ie:=1;
while i<=Max do begin
        if primo[i] then inc(TotPrimi);
        if pseudo[i] and not primo[i] then  begin
                inc(countPseudo);
                errori[ie]:=i;
                inc(ie);
                end;
        if miller[i] and not primo[i] then inc(countMiller);
        i:=i+2;
end;
```

L'output di un'esecuzione della programma *simulazione* è il seguente:

```
- i numeri primi minori di 65000 sono 6493
  (con l'approssimazione n/ln(n) si ottiene 5865)
- la procedura pseudoprimo sbaglia 62 volte (rapporto 0.010)
- la procedura MillerRa sbaglia 0 volte (rapporto 0.000)

I numeri per i quali la procedura PseudoPrimo sbaglia sono
       341       561       645      1105      1387      1729
      1905      2047      2465      2701      2821      3277
      4033      4369      4371      4681      5461      6601
      7957      8321      8481      8911     10261     10585
     11305     12801     13741     13747     13981     14491
     15709     15841     16705     18705     18721     19951
     23001     23377     25761     29341     30121     30889
     31417     31609     31621     33153     34945     35333
     39865     41041     41665     42799     46657     49141
     49981     52633     55245     57421     60701     60787
     62745     63973
```

Si può notare che la procedura MillerRa non sbaglia mai per i numeri da 2 a 65000. Esistono invece 62 pseudoprimi di base 2 minori di 65000, per i quali la procedura *pseudoprimo* dà un risultato errato.

5.1.2 Confronto delle procedure di verifica della primalità (prima versione)

La simulazione precedente ci assicura che, per numeri abbastanza piccoli, la procedura MillerRa non sbaglia mai. D'altra parte anche l'analisi teorica ci assicura che la probabilità che l'algoritmo di Miller-Rabin sbagli è circa 2^{-s} (nel nostro caso $s = 10$ quindi il risultato è sicuro al 99,9%).

Quindi possiamo testare l'attendibilità della prova di pseudoprimalità assumendo che la procedura *MillerRa* non sbagli mai e contando quante volte la procedura *Pseudoprimo* restituisce un risultato che contrasta con la procedura *MillerRa*. Questo metodo permette di evitare la ricerca dei numeri primi con un metodo non probabilistico.

L'implementazione di questo metodo è contenuta nel programma *confronto.pas*. Il codice, che qui riportiamo, è estremamente semplice: per ogni numero dispari minore del valore `High` (letto da tastiera) si confrontano i risultati delle due procedure probabilistiche. Se un numero è dichiarato primo da *Pseudoprimo* e composto da *MillerRa* si incrementa la variabile *err*.

```
err:=0; tot:=1;
i:=3;
while i<=High do begin
         gotoxy(x,y); Write(i:9);
         if not Pseudoprimo(i) then begin
                if MillerRa(i,s) then
                        err:=err+1
                 else
                        tot:=tot+1;
                 end;
                 i:=i+2;
                 end;
```

Anche questo programma viene implementato su interi senza segno su 32 bit, quindi per valori maggiori di 65536 si presenta il rischio di overflow nella procedura MillerRa.

5.1.3 Attendibilità degli algoritmi di verifica della primalità, seconda versione (utilizzando la rappresentazione degli interi come array di cifre decimali)

Per superare il limite di 65536 senza incorrere nell'overflow abbiamo scritto una versione del programma di simulazione che opera anche con i numeri rappresentati come array di cifre decimali.

Le modifiche da apportare al programma originale sono abbastanza semplici. Visto che i numeri considerati devono essere rappresentabili sui *cardinal* e quindi minori di $2^{32}-1$, si può usare una versione della unit *mathe* che opera su 18 cifre decimali (quelle necessarie perchè non si verifichi l'overflow moltiplicando numeri di 9 cifre) invece di 100 in modo da rendere l'esecuzione delle operazioni più veloce. È stato anche necessario utilizzare una versione della *unit* `uprimi` che lavora sui *cardinal* (e con i nomi delle procedure modificati aggiungendo una C in fondo per evitare conflitti con le analoghe procedure della unit *umprimi*) e una *unit* `conversioni` che fornisce la procedura per convertire un *cardinal* in *RecNo*. In questo modo se il numero di cui si deve controllare la primalità è minore di 65536 si richiama la procedura che utilizza i *cardinal*, altrimenti si converte il numero e si richiama la procedura che lavora sui *RecNo*.

Naturalmente questo modo di procedere ha un prezzo: la maggiore lentezza delle procedure di verifica della primalità che utilizzano i numeri rappresentati in base 10. Per questo motivo non è comunque possibile affrontare numeri significativamente grandi in tempi di esecuzione accettabili. Riportiamo comunque i risultati ottenuti per i numeri dispari da 3 a 100000. Con il metodo descritto nella prossima sezione siamo riusciti invece a considerare i numeri fino a 1000000.

```
- i numeri primi minori di 100000 sono 9592
  (con l'approssimazione n/ln(n) si ottiene 8686)
- la procedura pseudoprimo sbaglia 78 volte (rapporto 0.008)
- la procedura MillerRa sbaglia 0 volte (rapporto 0.000)

I numeri per i quali la procedura Pseudoprimo sbaglia sono
       341       561       645       1105      1387      1729
       1905      2047      2465      2701      2821      3277
       4033      4369      4371      4681      5461      6601
       7957      8321      8481      8911      10261     10585
       11305     12801     13741     13747     13981     14491
       15709     15841     16705     18705     18721     19951
       23001     23377     25761     29341     30121     30889
       31417     31609     31621     33153     34945     35333
       39865     41041     41665     42799     46657     49141
       49981     52633     55245     57421     60701     60787
       62745     63973     65077     65281     68101     72885
       74665     75361     80581     83333     83665     85489
       87249     88357     88561     90751     91001     93961
```

5.1.4 Confronto delle procedure di verifica della primalità (seconda versione)

Anche il metodo del confronto tra i due algoritmi di verifica di pseudoprimalità può essere modificato in modo simile per trattare i numeri più grandi di 65536. Questo programma risulta più veloce del precedente perché se un numero viene dichiarato composto dalla procedura Pseudoprimo non si richiama la verifica di Miller-Rabin.

Il programma scrive anche i risultati parziali a intervalli di 10000 fino a 100000 e di 50000 da 100000 in poi. In questo modo è possibile verificare che la percentuale di pseudoprimi diminuisce all'aumentare dei numeri considerati.

In questa tabella si possono leggere i risultati della simulazione per i numeri dispari fino a 1000000: nella seconda colonna è riportato il numero di primi trovati, nella terza il numero di pseudoprimi di base 2 (i numeri per i quali la procedura *pseudoprimo* sbaglia) e nella quarta la percentuale di pseudoprimi. Si può osservare che la percentuale passa dall' 1,790% per i numeri fino a 10000 allo 0,312% per i numeri fino a 1000000.

n	$\pi(n)$	pseudo	% d'errore
10000	1229	22	1.790
20000	2262	36	1.592
30000	3245	40	1.233
40000	4203	49	1.166
50000	5133	55	1.071
60000	6057	58	0.958
70000	6935	65	0.937
80000	7837	68	0.868
90000	8713	75	0.861
100000	9592	78	0.813
150000	13848	91	0.657
200000	17984	106	0.589
250000	22044	120	0.544
300000	25997	138	0.531
350000	29977	146	0.487
400000	33860	152	0.449
450000	37706	161	0.427
500000	41538	172	0.414
550000	45322	178	0.393
600000	49098	187	0.381
650000	52831	195	0.369
700000	56543	208	0.368
750000	60238	220	0.365
800000	63951	224	0.350
850000	67617	232	0.343
900000	71274	238	0.334
950000	74907	241	0.322
1000000	78498	245	0.312

5.1.5 Analisi dei risultati

- Le simulazioni portano a concludere che la prova di pseudoprimalità dà un risultato corretto in una percentuale di casi che, partendo dal 98,2% per i numeri minori di 10000, arriva al 99,688% per i numeri dell'ordine di grandezza di 10^5. Le prove confermano infatti che la percentuale di pseudoprimi di base due diminuisce considerando numeri più grandi.

- L'algoritmo di Miller-Rabin non dà mai risultati sbagliati nei limiti

dell'ordine di grandezza sul quale è stato possibile effettuare la simulazione (10^4). Questi risultati confermano la previsione teorica secondo la quale la possibilità di un errore è minore di 2^{-10} (0,097%).

- La percentuale di numeri primi diminuisce: dal 12,29% per i primi 10000 numeri al 7,89% per i numeri fino a 1000000
- Possiamo controllare la bontà dell'approssimazione $n/\ln(n)$: per $n = 10^6$ si ha $n/\ln(n) \approx 72348$, mentre $\pi(n) = 78498$ (errore del 7,83%).
- Bisogna notare che i valori di $\pi(n)$ riportati nella tabella non sono sicuri al 100%, essendo stati ricavati dall'algoritmo di Miller-Rabin. Tuttavia l'analisi teorica ci dice che l'errore *massimo* su $\pi(n)$ è $10^6 * 2^{-10} = 976$. Un errore di questo tipo non cambierebbe le conclusioni esposte in questa sezione.
- In realtà però il valore $\pi(n) = 78498$ è esatto, come risulta dalla tabella riportata nell'appendice B. Questo vuol dire che il comportamento dell'algoritmo di Miller-Rabin è molto buono: non sbaglia mai per i numeri minori di 10^6! Infatti la stima di 2^{-s} per la probabilità che l'algoritmo di Miller-Rabin dia un errore è un limite superiore che, almeno nelle prove da noi effettuate, non viene raggiunto.

5.2 Simulazione dell'euristica rho di Pollard

5.2.1 Descrizione del programma per la simulazione

La simulazione dell'euristica rho di Pollard si trova nel file *verpoll.pas*. L'idea di questo programma è quella di verificare se la funzione *PollardRho* riesce a trovare almeno un fattore primo di un numero sicuramente composto perché precedentemente analizzato dalla funzione *MillerRa*.

```
procedure SimPollard(Num: longint; var count,
                     fattori: integer);
begin
     Inc(count);
     if PollRho(Num) then
     begin
          Inc(fattori);
     end;
end;
```

```
...

 for j:= 1 to Num do
    begin
       c:= 0; f:= 0;
       Write('Prova Num ', j);
       i:= 0;
       repeat
           pr:= Random(MaxInt-1)+1;
           if (MillerRa(pr, 10)) and ((pr mod 2) = 0)
                                 and (pr <> 0) then
           begin
                SimPollard(pr, c, f);
                i:= i+1;
           end;
        gotoxy(20, WhereY); write('Prova ', i, '/', NProve*j);
       until i=j*NProve;
       gotoxy(20,WhereY);
       Write(c:4, '         ', f:4);
       Writeln('                ', ((f / c)*100):5:1,' %');
       CTot:= Ctot + c;
       FTot:= FTot + f;
    end;
```

A questo scopo il programma esegue un numero di prove pari al valore della costante *Num*. Alla k-esima prova cerca un numero di valori composti uguale al valore del prodotto di Num * *NProve*, altra costante definita dalla procedura. Per ognuno di questi numeri casuali valuta, attraverso la chiamata alla procedura PollardRho, se l'algoritmo di rho di Pollard riesce a trovare almeno un fattore non banale. A termine della k-esima prova la variabile `c`, conterrà il numero di valori provati, mentre la variabile `f` conterrà il numero di valori per cui la funzione emphPollardRho ha dato il risultato atteso. Il loro rapporto percentuale, stampato a video, sarà chiamato *Tasso di Efficienza*, intendendo con questo termine la percentuale dei casi in cui la procedura PollardRho riesce a trovare un fattore non banale di un numero composto. Al termine delle iterazioni il rapporto percentuale tra `FTot` e `CTot`, darà il tasso di efficienza globale dell'euristica rho di Pollard rilevata con la simulazione.

5.2.2 Analisi dei risultati

L'analisi dei risultati della simulazione effettuata sull'algoritmo Rho di Pollard porta a concludere che tale algoritmo riesce con una ottima probabilità a trovare almeno un fattore primo non banale di un numero composto. La simulazione infatti testa un buon numero di non primi casuali e sembra portare nella maggior parte dei casi a risultati abbastanza confortanti. È importante notare che i numeri provati sono sicuramente composti, visto che sono stati dichiarati tali dalla procedura *MillerRa* (infatti questo algoritmo può sbagliare solo nell'affermare che è primo un numero che in realtà è composto, ma un numero dichiarato composto da *MillerRa* lo è sicuramente!).

Riportiamo l'output dell'esecuzione di una simulazione della procedura di Pollard

```
Simulazione della tecnica rho di Pollard per la ricerca di
un fattore di un numero

                Provati      Trovati       Tasso d'Efficienza

Prova Num 1         100           98                  98.0 %
Prova Num 2         200          198                  99.0 %
Prova Num 3         300          294                  98.0 %
Prova Num 4         400          396                  99.0 %
Prova Num 5         500          490                  98.0 %
Prova Num 6         600          587                  97.8 %
Prova Num 7         700          690                  98.6 %
Prova Num 8         800          789                  98.6 %
Prova Num 9         900          888                  98.7 %
Prova Num 10       1000          984                  98.4 %

   Totali :
Provati: 5500
Trovati: 5414
Tasso di efficienza globale: 98.4 %
```

Ripetute esecuzioni del programma indicano che la procedura di Pollard riesce a trovare un fattore primo non banale di un numero composto in una percentuale di casi che oscilla tra il 97% e il 99% con un algoritmo di complessità $O(\sqrt[4]{n})$ e questo risultato conferma la validità pratica dell'euristica rho di Pollard.

A Descrizione dei vari file che compongono il progetto

Per comodità riportiamo l'elenco dei file che compongono il progetto con la relativa descrizione.

I file sono divisi in cartelle che indicano il tipo di dati su cui operano i vari programmi: nelle directory *longint* e *cardinal* si trovano i programmi che operano su longint e cardinal mentre nella directory *mathe* si trovano i programmi che operano sui numeri rappresentati come array di cifre decimali. La cartella *simulazione* contiene i programmi per la simulazione degli algoritmi di verifica della primalità.

Cartella	File	Descrizione
longint	uprimi.pas	Unit che contiene il codice delle procedure per la verifica di primalità e per l'euristica rho di Pollard sui longint
	verif.pas	Programma che permette di testare rapidamente le varie procedure sui longint. Con un menù si può scegliere uno dei tre algoritmi e testarlo su un numero inserito da tastiera
	simpoll.pas	La simulazione per l'euristica rho di Pollard (cfr. sez. 5.2)
cardinal	uprimi.pas	Unit del tutto simile a quella con lo stesso nome contenuta nella cartella *longint* ma che opera sui cardinal.
	simulazione.pas	Prima versione della simulazione per gli algoritmi di verifica di primalità (cfr. sez. 5.1.1)
	confronto.pas	Confronto tra i due algoritmi per la verifica della primalità (cfr. sez. 5.1.2)

Cartella	File	Descrizione
mathe	mathe.pas	Unit per la gestione delle operazioni sugli interi rappresentati come array di cifre decimali (cfr. sez. 2.2)
	umprimi.pas	Unit che contiene l'implementazione dei tre algoritmi sugli interi in rappresentazione decimale.
	verif.pas	Programma analogo a quello con lo stesso nome contenuto nella cartella *longint*, ma che lavora sulla rappresentazione decimale.
simulazione	mathe.pas, umprimi.pas	Unit analoghe a quelle con lo stesso nome contenute nella cartella *mathe*, ma che lavorano con 18 cifre decimali invece di 100.
	uprimi2.pas	Unit quasi identica a *uprimi.pas* della directory cardinal ma con i nomi delle procedure cambiati per evitare conflitti con la unit *umprimi*.
	conversioni.pas	Unit che contiene le procedure per la conversione da *cardinal* a rappresentazione decimale e viceversa
	simulazione.pas	Seconda versione della simulazione per gli algoritmi di verifica di primalità (cfr. sez. 5.1.3)
	confronto.pas	Programma simile a quello con lo stesso nome contenuto nella cartella *cardinal* ma che opera anche sulla rappresentazione decimale degli interi (cfr. sez. 5.1.4)
	confronto2.pas	Versione del programma *confronto* che offre la possibilità di continuare la simulazione da un certo numero inserendo i risultati parziali.

B Tabella dei valori della funzione di distribuzione dei numeri primi per le potenze di 10 fino a 10^{20}

Questa tabella dei valori della funzione di distribuzione dei numeri primi $\pi(n)$, che si può trovare su internet all'indirizzo
`http://www.utm.edu/research/primes/howmany.shtml#table`
conferma che il valore ricavato per $\pi(10^6)$ con l'algoritmo di Miller-Rabin è corretto.

n	$\pi(n)$
10	4
100	25
1,000	168
10,000	1,229
100,000	9,592
1,000,000	78,498
10,000,000	664,579
100,000,000	5,761,455
1,000,000,000	50,847,534
10,000,000,000	455,052,511
100,000,000,000	4,118,054,813
1,000,000,000,000	37,607,912,018
10,000,000,000,000	346,065,536,839
100,000,000,000,000	3,204,941,750,802
1,000,000,000,000,000	29,844,570,422,669
10,000,000,000,000,000	279,238,341,033,925
100,000,000,000,000,000	2,623,557,157,654,233
1,000,000,000,000,000,000	24,739,954,287,740,860
10,000,000,000,000,000,000	234,057,667,276,344,607
100,000,000,000,000,000,000	2,220,819,602,560,918,840