

Fattorizzazione con algoritmo generalizzato con quadrati perfetti in ambito delle forme $6k\pm 1$

ing. Rosario Turco, dott. Michele Nardelli, prof. Giovanni Di Maria, Francesco Di Noto, prof. Annarita Tulumello, prof. Maria Colonnese.

Sommario

L'articolo è un ulteriore approfondimento del precedente lavoro "Test di primalità, Fattorizzazione e $\pi(N)$ con forme $6k\pm 1$ " (vedi riferimento [1]). Il presente lavoro mostra come, in ambito fattorizzazione con forme $6k\pm 1$, si possa introdurre un algoritmo generalizzato con cui affrontare tutti i tipi di quadrati perfetti: terne pitagoriche, quadrati perfetti per numeri a distanza 2 (sia numeri gemelli che non) o quadrati perfetti per numeri a distanza maggiore di 2. Viene, per l'occasione, introdotto un Teorema inedito sull'argomento, che permette di risolvere, tramite un algoritmo efficiente, la problematica

Premessa

In [1] sono state presentate le forme $6k\pm 1$ e le loro proprietà, sono stati definiti un Teorema di primalità, dei corollari ed un Teorema di fattorizzazione.

Nel presente lavoro, fermo restando la tecnica di fattorizzazione già spiegata in [1], affronteremo come è possibile, invece, generalizzare, nell'ambito dell'algoritmo di fattorizzazione, la tecnica dei quadrati perfetti; in modo che essa vada bene non solo, come in [1], per numeri gemelli e non (numeri a distanza 2 es: 5 e 7), ma anche per terne pitagoriche e numeri (primi e non) a distanza maggiore di 2 (es: 11 e 17).

Definiamo un nostro Teorema.

Teorema Fattorizzazione QPG

"Sia dato un intero positivo P. Se esiste una coppia di valori (Q', Q) , entrambi quadrati perfetti o anche costituenti insieme a P una terna pitagorica, tale che $P + Q' = Q$ con $Q > P$, allora P è scomponibile in un prodotto di fattori nel seguente modo:

$$P = (\sqrt{Q} - \sqrt{Q'}) * (\sqrt{Q} + \sqrt{Q'}) \quad (1)$$

Se nella (1) il termine $\sqrt{Q} - \sqrt{Q'} = 1$ è inutile scomporre in fattori P in tale modo; mentre non è conveniente, in termini di velocità di fattorizzazione, cercare una coppia (Q, Q') se nell'ambito della ricerca della coppia il termine Q' è diventato maggiore di P".

Il metodo consiste nel ricercare, quindi, un Q' quadrato perfetto che, sommato a P, fornisce un altro quadrato perfetto Q. Vediamo degli esempi esplicativi.

Esempio 1 (P pari composto con fattori a distanza 4)

$P=60$ $P < Q$: $Q=64$, $Q'=4 < P$

$$\sqrt{Q} = 8, \sqrt{Q'} = 2$$

$$P = (8 - 2) * (8 + 2) = 6 * 10$$

Esempio 2 (P dispari composto con primi a distanza 4)

$P=77$ $P < Q$: $Q=81$, $Q'=4 < P$

$$\sqrt{Q} = 9, \sqrt{Q'} = 2$$

$$P = (9 - 2) * (9 + 2) = 7 * 11$$

Esempio 3 (P dispari composto con primi a distanza 2 o gemelli)

$P=35$ $P < Q$: $Q=36$, $Q'=1 < P$

$$\sqrt{Q} = 6, \sqrt{Q'} = 1$$

$$P = (6 - 1) * (6 + 1) = 5 * 7$$

Esempio 4 (P dispari composto con una terna pitagorica)

$$P=9 \quad P<Q: Q=25, Q' =16<P$$

$$\sqrt{Q} = 5, \sqrt{Q'} = 4$$

$$P = (5 - 4) * (5 + 4) = 1 * 9$$

Inutile usare questo metodo perché la differenza delle radici restituisce 1 in questo caso.

Esempio 5 (P numero primo)

Se si proseguisse senza fermarsi quando $Q'>P$ si otterrebbe:

$$P=17 \quad P<Q: Q=81, Q' =64>P$$

$$\sqrt{Q} = 9, \sqrt{Q'} = 8$$

$$P = (9 - 8) * (9 + 8) = 1 * 17$$

Ma ci si arresta prima perché $Q'>P$. D'altra parte per i numeri primi se si continuasse il metodo non è applicabile comunque, come nel caso precedente (differenza radici). In altri termini il metodo, in generale, si arresta perché la ricerca della coppia (Q', Q) non è più considerata conveniente, in due casi possibili:

- differenza delle radici = 1
- $Q' > P$

In entrambi i casi il numero non viene considerato un numero da scomporre col metodo del quadrato perfetto: potrebbe essere un composto o un primo ma per la scomposizione in fattori e per la primalità si preferisce applicare la tecnica più veloce, il “secondo passo” dopo il controllo del quadrato perfetto, descritta in [1]. Si può facilmente osservare che tutti i primi hanno la differenza delle radici pari a 1 con $Q' > P$. Questo è il segnale di “numero primo”, ma si preferisce uscire prima, invece, di provare molti valori Q' , questo nell’ottica di velocizzare la scomposizione. Esempi: $23+121=144$, $29+196=225$ etc. I composti presentano, spesso, anch’essi una differenza delle radici pari a 1 ma già a $Q'<P$ (quindi ci si arresta al 1° che si trova) e sono caratterizzati, in quanto composti, di avere più soluzioni all’equazione $P + Q' = Q$. Anche qui la differenza 1 e $Q'<P$ potrebbe essere considerato come segnale di composto. Esempi $21+4=25$ ma anche $21+100=121$.

Condizione aggiuntiva

Ovviamente nel ciclo di ricerca della soluzione, nessuno dei termini della (1) deve essere maggiore o uguali a P. Si esce dal ciclo se si viola tale condizione.

Algoritmo

Il Teorema fornisce un criterio di arresto conveniente o un criterio di non convenienza nell’uso della “tecnica del quadrato perfetto generalizzata”. In ogni caso il Teorema permette un algoritmo più veloce rispetto a [1], perché a parità di passi, due passi di cui uno di controllo quadrato perfetto e l’altro di scomposizione senza quadrato perfetto, ora nel “passo del controllo quadrato perfetto” si provano tutte le distanze tra i numeri (terne pitagoriche, quadrati perfetti per numeri a distanza 2 e a distanza maggiore di 2). Nell’APPENDICE INFORMATICA viene presentato l’algoritmo sopra discusso, presentato nel **sorgente quadrpGen.c.**, ma integrato nell’ambito della fattorizzazione al posto del sorgente quadrp.c presentato in [1]

Riferimenti

- [1] Test di primalità, Fattorizzazione e $\pi(N)$ con forme $6k\pm 1$ - ing. Rosario Turco, dott. Michele Nardelli, prof. Giovanni Di Maria, Francesco Di Noto, prof. Annarita Tulumello. CNR Solar
- <http://www.gruppoeratostene.netandgo.eu/> Gruppo Eratostene
- <http://xoomer.alice.it/stringtheory/Home.html> Michele Nardelli
- <http://www.geocities.com/SiliconValley/Port/3264> R. Turco - Aladdin’s Lamp (sezione MISC)

APPENDICE INFORMATICA

```
//
// R. Turco (C) 2008
// Fattorizzazioni con forme  $6n+1$ 
// stackprime.h
//
#ifdef PRIME_STACK
#define PRIME_STACK
static unsigned long int stack[10000];
// prototype
void fattnp1(unsigned long int Num);
int vnm1(unsigned long int nt,unsigned long int *V1, unsigned long int *V2);
int vnp1(unsigned long int nt,unsigned long int *V1, unsigned long int *V2);
int quadrp(unsigned long int Num, unsigned long int *V1, unsigned long int *V2);
int quadrpGen(unsigned long int Num, unsigned long int *V1, unsigned long int *V2);
#endif

#include <stdio.h>
#include <stdlib.h>
#include "stackprime.h"

//
// R. Turco (C) 2008 v 1.1
// Fattorizzazioni con forme  $6n+1$ 
// Main
//

int main(int argc, char *argv[])
{
    unsigned long int Num=0;
    int i = 0;
    printf("\n\n*** Fattorizzazione numeri naturali - R.Turco (C) 2008 v. 1.1 ***");
    printf("\n\n Algoritmo basato su Generalizzaz. Quadrati perfetti");

    do{

        printf("\n\nInserisci numero (0 per uscire): \n");

        scanf("%u",&Num);
        if( Num != 0 ){
            for(i=0; i<=10000; i++){
                stack[i]=0;
            }
            (void) fattnp1(Num);
            system("PAUSE");
        }

    }while(Num != 0 );

    return 0;
}

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

// fattnp1: gestisce situazioni mod 5
// fattnm1: gestisce situazioni mod 1
//
// Abstract:
// Indipendentemente dalla formula per mod 5 o mod 1
// se nt è noto, fissato K1 si calcola K2
// se il valore T temporaneo calcolato è minore di nt si aumenta K1
// e si ricalcola T (ciclo)
// se il valore T temporaneo calcolato è maggiore di nt è inutile
// proseguire
// altrimenti T=nt quindi sono stati trovati K1 e K2
//
// return value: VA e VB come puntaori
// return status: 1 se primo, 0 se non primo
```

```

//
//
int vnm1(unsigned long int nt, unsigned long int *VA, unsigned long int *VB){
// Restituisce status = 1 se primo, 0 se non lo è
// Caso modulo 5
unsigned long int K1=0;
unsigned long int K2=0;
unsigned long int T1=0;
unsigned long int V1=0;
unsigned long int V2=0;
unsigned long int A1=0;

int fex=0;
int status=0;

K1=1;
K2=1;

if( nt==1 ) { /* Non va bene la formula: è primo */
*VA=0;
*VB=nt;
return 1;
}

A1 = (6*K1-1)*(6*K2+1);

if( A1 > nt ) { /* Non va bene la formula: è primo */
*VA=0;
*VB=nt;
return 1;
}

printf("\nMod 5 - Ciclo (6n-1)(6n+1)\n");

do{

K2 = ((nt+1)/6 - K1)/(6*K1-1);

if( K2 == 0 ){
fex=1; // E' primo
}

T1 = (6*K1-1)*(6*K2+1);

if(T1 == nt){
fex=1; // L' ho trovato
}

if( T1 < nt ){
K1++;
}

if( T1 > nt ){
fex=1;
}

}while((nt != T1) && (fex==0));

if( nt == T1 ){
/* E' un composto */
V1=6*K1-1;
V2=6*K2+1;

if( K2 == 0 ){
V2=0;
}
status=0;
}
else{
/* E' un primo */
V1=0;
V2=0;
status=1;
}
}

```

```

}
*VA=V1;
*VB=V2;
return status;
}

int vnp1(unsigned long int nt,unsigned long int *VA, unsigned long int *VB){
// Restituisce status = 1 se primo, 0 se non lo è
// Mod 1
unsigned long int K1=0;
unsigned long int K2=0;

unsigned long int T1=0;
unsigned long int T2=0;
unsigned long int A1=0;
unsigned long int A2=0;

unsigned long int V1=0;
unsigned long int V2=0;
int fex=0;
int status=0;

K1=1;
K2=1;

if(nt==1){ /* Non vanno bene le formule: è primo */
*VA=0;
*VB=nt;
return 1;
}

A1 = (6*K1-1)*(6*K2-1);
A2 = (6*K1+1)*(6*K2+1);

if((A1>nt) && (A2>nt)){ /* Non vanno bene le formule: è primo */
*VA=0;
*VB=nt;
return 1;
}

if( A1 <= nt ){ /* Va provata la formula */
printf("\nMod 1 - Ciclo (6n-1)(6n-1)\n");
do{
K2 = ((nt-1)/6 + K1)/(6*K1-1);

if( K2 == 0 ){
fex=1; // E' primo
}

T1 = (6*K1-1)*(6*K2-1);

if(T1 == nt){
fex=1; // L' ho trovato
}

if( T1 < nt ){
K1++; // Devo cercare il valore di K1
}

if( T1 > nt ){
fex=1; // E' inutile proseguire
}
}while((nt != T1) && (fex==0));
}

if( A2 <= nt ) && (nt != T1){ /* Va provata la formula */

K1=1;
K2=1;

```

```

fex=0;

printf("\nMod 1 - Ciclo (6n+1)(6n+1)\n");

do{

    K2 = ((nt-1)/6 - K1)/(6*K1+1);

    if( K2 == 0 ){
        fex=1; // E' primo
    }

    T2 = (6*K1+1)*(6*K2+1);

    if(T2 == nt){
        fex=1; // L' ho trovato
    }

    if( T2 < nt ){
        K1++;
    }

    if( T2 > nt ){
        fex=1;
    }

}while((nt != T2) &&(fex==0));
}

if( (nt==T1) || (nt==T2)){

    /* E' un composto */
    if( nt==T1 ){
        V1=(6*K1-1);
        V2=(6*K2-1);

        if( K2 == 0 ){
            V2=0;
        }
    }
    if( nt==T2 ){

        V1=(6*K1+1);
        V2=(6*K2+1);

        if( K2 == 0 ){
            V2=0;
        }
    }
    status=0;
}
else{ /* E' un primo */
    V1=0;
    V2=0;
    status = 1;
}

*VA=V1;
*VB=V2;
return status;
}

#include <stdio.h>
#include <stdlib.h>
#include "stackprime.h"

//
// R. Turco (C) 2008 v 1.1
// Fattorizzazione con forme 6n+1
// Abstract: L' algoritmo gestisce uno stack di 10mila fattori primi o composti
// e cerca anche se esistono quadrati perfetti (numeri gemelli)
//

```

```

void fattnp1(unsigned long int Num){
    unsigned long int nt = Num;

    unsigned long int V1 = 1;
    unsigned long int V2 = 1;
    unsigned long int *V3 = 0;
    unsigned long int *V4 = 0;

    int prime=0; /* Non è primo */

    int i=0; // indice dello stack
    long int T = -1; // Valore di confronto
    int quadrPerf=0; // Indicatore del quadrato perfetto

    printf("\n");

    if( nt < 12 ){
        printf("\nNumero < 12");
        printf("\nfatt-> %u \n", nt);
        printf("\n");
        system("PAUSE");
        exit(0);
    }

    V3=(unsigned long int*)malloc(sizeof(V1));
    V4=(unsigned long int*)malloc(sizeof(V2));

    do{

        prime=0;
        quadrPerf=0;

        T=nt%6;

        printf("Fattorizzo #: %u \n", nt);

        // Prova del quadrato perfetto

        quadrPerf=quadrpGen(nt, V3, V4);
        if( quadrPerf == 1 ){
            // Quadrato perfetto
            printf("\nQuadrato perfetto: %u %u \n", *V3, *V4);
            T=9; // passo per la zona di comodo dei q p
            nt = *V3;
            system("PAUSE");
        }

        // Prova del quadrato perfetto

        switch(T){
            case 0:
            case 2:
            case 4:
            case 8:
                T=2; // devo dividere per 2
                nt = nt/T;
                printf("\nfatt-> %u *\n", T);
                break;

            case 1:
            case 5:
            case 9: // il caso 9 è di comodo per i quadrati perfetti
                if( T == 1 ){
                    // Occorrerà verificare fattorizzazione di due numeri
                    // V1 e V2;
                    // Ricerca dei fattori k1 e k2
                    // su due equivalenze se serve  $(6n-1)(6n-1)$  o  $(6n+1)(6n+1)$ 
                    prime = vnp1(nt,V3,V4);
                }

                if( T == 5){
                    // Occorrerà verificare fattorizzazione di due numeri
                    // V1 e V2;

```

```

// Ricerca dei fattori k1 e k2
// su una sola equivalenza (6n-1)(6n+1)
prime = vnm1(nt,V3,V4);
}

V1=*V3;
V2=*V4;

if(prime){ // è primo
    printf("\nfatt-> %u *\n", nt);
    nt=0;
    i--;
    if( i < 0 ) i=0;
    if( stack[i] != 0 ){
        // Controllo lo stack
        nt = stack[i];
        stack[i] = 0;
        V2=0;
    }
}

// Se ho V1 e V2, allora V2 va nello stack
if((V2>1) && (!prime)){
    // Inserisco nello stack
    stack[i]=V2;
    i++;
}

// Se V1 già lo posso escludere dai successivi
// controlli, allora prendo un valore dallo stack
//
if((V1 < 12) && (V1>0) && (!prime)){
    printf("\nfatt-> %u *\n", V1);
    V1=1; /* per uscire se lo stack fosse vuoto*/
    i--; // ritorno alla posizione precedente dello stack
    if( i < 0 ) i=0;
    if( stack[i] != 0 ){
        nt=stack[i]; /* Prendo V2 e svuoto lo stack*/
        stack[i]=0;
        V2=0;
        if( nt < 12) && quadrPerf ) {
            printf("\nfatt-> %u *\n", nt);
            nt=1;
        }
    }
}

// adesso mi chiedo se il valore dello stack è da processare
// oppure ho finito già (i=0 e nt=0), visto che V1=1
//

if((i==0) && (V1==1) && (nt==0)){ // Ho finito V1 e lo stack nt==0 o < 12
    printf("\nfatt-> %u *\n", nt);
    nt=0;
    i--;
    if( i < 0 ) i=0;
    if( stack[i] != 0 ){
        nt=stack[i]; /* Prendo V2 e svuoto lo stack*/
        stack[i]=0;
        V2=0;
    }
}

}
else{
    if( (V1 > 1) && (!prime)){
        nt=V1;
    }
}

if( (V2 == 0) && (V1 == 1) && i<2){
    i--;
    if( i < 0 ) i=0;
    if( stack[i] != 0 ){
        nt=stack[i]; /* Prendo V2 e svuoto lo stack*/
    }
}

```

```

        stack[i]=0;
        V2=0;
    }
}
break;
case 3:
    nt = nt/T;
    printf("\nfatt-> %u *\n", T);
    break;

default:
    nt = nt/T;
    printf("\nfatt-> %u *\n", T);
    break;
}
    T = -1; // Lo setto ad un valore impossibile per il contesto
}while(nt >= 12); // Se c' è qualcosa nello stack devo ciclare

if( nt < 12) && (nt>1) {
    printf("\nfatt-> %u *\n", nt);
}
if( V3!= NULL)
    free(V3);
if( V4!= NULL)
    free(V4);
return;
}

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

int quadrpGen(unsigned long int Num, unsigned long int *VA, unsigned long int *VB){
    // R.Turco (C) 2008 versione 1.2
    // restituisce 0 se non è un quadrato perfetto
    // restituisce 1 se lo è con i valori V1 e V2
    //
    int status=0; // per default non è un quadrato perfetto
    unsigned long int q=1, Q=0, Q1=0;// per default non è un quadrato perfetto

    double y=0.0;
    double y1=0.0;
    double z=0.0;
    double *w;

    w = (double*)malloc(sizeof(double));

    //
    // Cerchiamo la coppia Q' e Q quadrato perfetti
    // P + Q1 = Q
    // Ci arrestiamo se Q1 > P
    //

    *VA=0;
    *VB=0;

    do
    {
        Q1 = q*q;
        Q = Num + Q1;
        y=sqrt(Q);
        y1=sqrt(Q1);
        z=modf(y,w);
        q++;

        if( z == 0.0 ) { // controllo la parte dopo la virgola quella frazionaria
            *VA= (int) (y - y1);
            *VB= (int) (y + y1);
            /* printf("\nQ P\n"); */
            status = 1; // Quadrato perfetto
        }
    }
}

```

```

    }
}while( (Q1 <= Num) && (Q1 < Q) && (status==0) && (*VA < Q) && (*VB < Q));

if( (*VA == 1) || (*VB ==1) || (*VA>=Num) || (*VB >=Num))
{
    *VA=0;
    *VB=0;
    status = 0; /* non fattorizzabile con questo algoritmo */
}
/* printf("\nVA=%f VB=%f\n", *VA, *VB); */
if( w!= NULL) {
    free(w);
}

return status;
}

```