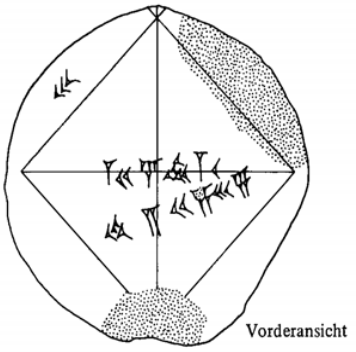
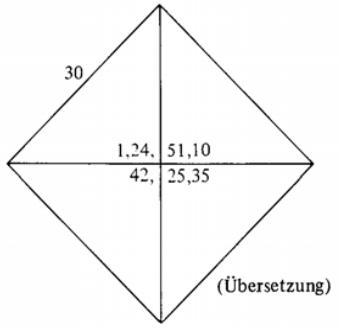
3 Historische Algorithmen

3.1 Heron - Verfahren

Im Zusammenhang mit Problemen der Flächenmessung, insbesondere der Landvermessung, entwickelten die Babylonier Verfahren zur Lösung quadratischer Gleichungen, die uns aus der Epoche von Hammurapi (etwa 1700 v. Chr.) auf Keilschriften überliefert sind (vgl. Abbildung unten). Eine Vorstufe zur Lösung allgemeiner quadratischer Gleichungen ist die Bestimmung von Quadratwurzeln. Das Verfahren wurde später etwa um 100 n.Chr. von den Griechen (Heron von Alexandria) aufgegriffen und beschrieben; es wird heute oft auch als Heron-Verfahren bezeichnet.





Heron von Alexandria lebte um 100 n.Chr. Seine Werke stellen eine Art Enzyklopädie in angewandter Geometrie, Optik und Mechanik dar. Sie haben oft den Charakter einer Formelsammlung. Viele der ihm zugeschriebenen Formeln und Verfahren waren schon vorher bekannt; so soll die Heronsche Formel für den Flächeninhalt von Dreiecken von Archimedes stammen; das Heron-Verfahren zum Wurzelziehen wurde schon viele Jahrhunderte vorher von den Babyloniern praktiziert.

Problemstellung, Interpretation und heuristische Grundidee

Problem: Gegeben sei eine Zahl **a**. Gesucht ist eine Zahl **b** mit der Eigenschaft **b * b = a**.

In unserer heutigen Terminologie ist **b** die (**Quadrat-**) **Wurzel** von **a**; im Zeichen: b = $\sqrt[2]{a}$

Geometrische Interpretation: Man ermittle die **Seitenlänge b** eines Quadrats vom Flächeninhalt **a**.

Oder: Man konstruiere ein Quadrat mit dem Flächeninhalt **a**.

Die Grundidee des babylonischen Verfahrens beruht auf dieser geometrischen Veranschaulichung und der folgenden Anwendung des "Prinzips der Bescheidenheit": Wenn man das zum Flächeninhalt a gehörende Quadrat nicht sofort bekommen kann, begnüge man sich mit etwas weniger, etwa mit einem Rechteck des Flächeninhalts a. Ein solches ist leicht zu konstruieren:

Man wähle z.B. eine Seitenlänge gleich a und die andere Seitenlänge gleich 1 (also eine Längeneinheit).

Das Störende daran ist, dass die so gewählten Seitenlängen im allgemeinen verschieden sind, dass das Rechteck also kein Quadrat ist. Man versucht nun schrittweise, ausgehend von dem Ausgangsrechteck immer "quadrat-ähnlichere" Rechtecke zu konstruieren.

Dazu geht man folgendermaßen vor: Man wählt die eine Seite des neuen Rechtecks als das arithmetische Mittel der Seiten des Ausgangsrechtecks und passt die andere Seite so an, dass sich der Flächeninhalt a nicht verändert. Sind **x0 (x0 = a)** und **y0 (y0 = 1)** die Seiten des Ausgangsrechtecks, so lauten die Seitenlängen des neuen Rechtecks:

$$x_1 := (x_0 + y_0) / 2 , \; y_1 := a / x_1$$

Entsprechend fährt man mit dem neuen Rechteck an Stelle des Ausgangsrechtecks fort und erhält so die allgemeine Iterationsvorschrift des **Heron-Verfahrens**:

$$x_{n+1} := (x_n + y_n) / 2 , \; y_{n+1} := a / x_n$$

Diese "gekoppelte" Iteration wird oft nach einer naheliegenden Umformung in der folgenden Form geschrieben, in der nur noch die (indizierte) Variable **x** vorkommt:

$$x_{n+1} := (x_n + a / x_n) / 2$$

Iterativer Algorithmus des Heron-Verfahrens in C

```
#include <stdio.h>                // heron_iterativ.c
#include <math.h>

void main(){
         double a = 0.0, epsilon = 0.00000001;
         double x = a, y = 1.0;
         unsigned long n = 0UL;

         printf("Heron-Verfahren zur Bestimmung der Quadratwurzel\n\n");
         do {
             printf("a (a > 0)                    = ");
             scanf("%lf", &a); fflush(stdin);
         } while(a<=0.0);

         do {
             printf("epsilon (epsilon > 0)        = ");
             scanf("%lf", &epsilon); while(getchar()!='\n');
         } while(epsilon <= 0.0);

         x = a;                           // Anfangswert x = a

         while(fabs(x*x - a) >= epsilon) // Abbruchbedingung
         {
           ++n;                           // Anzahl Iterationen
           x = (x + a/x)/2.0;             // alternativ kuerzer
         }

         printf("\niterativ sqrt( %10.2lf ) = %16.8lf\n", a, x);
         printf("math.h   sqrt( %10.2lf ) = %16.8lf\n", a, sqrt(a));
         printf("Iterationen   n             = %7d", n);
         getchar();
}

/*
Heron-Verfahren zur Bestimmung der Quadratwurzel

a (a > 0)                    = 2
epsilon (epsilon > 0)        = 0.00000001

iterativ sqrt(       2.00 ) =       1.41421356
math.h   sqrt(       2.00 ) =       1.41421356
Iterationen   n             =       4
*/
```

Rekursiver Algorithmus des Heron-Verfahrens in C

```
#include <stdio.h>                              // heron_rekursiv.c
#include <math.h>

double hr(double x, double a, double epsilon, unsigned long *n){
      (*n)++;                                   // Aufrufzaehler inkrementieren

      if(fabs(x*x-a)<= epsilon)                 // Abbruchbedingung
            return x;

      return hr((x+a/x)/2.0, a, epsilon, n);    // rekursiver Aufruf
}

void main(){
    double a = 0.0,                 // Argument a
           epsilon = 0.00000001;    // Genauigkeit

    unsigned long n = 0UL;          // Anzahl Iterationen

    printf("Heron-Verfahren zur Bestimmung der Quadratwurzel\n\n");

    do {
        printf("a (a > 0)                    = ");
        scanf("%lf", &a); while(getchar()!='\n');
    } while(a<=0.0);

    do {
        printf("epsilon (epsilon > 0)        = ");
        scanf("%lf", &epsilon); while(getchar()!='\n');
    } while(epsilon <= 0.0);

    printf("\nrekursiv sqrt( %10.2lf ) = %16.8lf\n",
                                          a, hr(a, a, epsilon, &n));
    printf("math.h   sqrt( %10.2lf ) = %16.8lf\n", a, sqrt(a));
    printf("Iterationen   n              = %7d", n);
    getchar();
}

/*
Heron-Verfahren zur Bestimmung der Quadratwurzel

a (a > 0)                    = 2
epsilon (epsilon > 0)        = 0.00000001

rekursiv sqrt(       2.00 ) =       1.41421356
math.h   sqrt(       2.00 ) =       1.41421356
Iterationen   n             =       5
*/
```

3.2 Euklidischer Algorithmus

Hintergrund ist stets der **Satz von der Division mit Rest**:

Zu je zwei natürlichen Zahlen **a** und **b** (mit **b > 0**) gibt es stets **eindeutig** bestimmte nichtnegative ganze Zahlen **q** und **r** mit der Eigenschaft **a = q * b + r** und **0 <= r < b**.
(Beweis mit vollständiger Induktion)

Beispiel: Aus **a = 17**, **b = 5** folgt **17 = 3 * 5 + 2** , d.h. **q = 5** , **r = 2**

Wenn **Div(a, b)** die **ganzzahlige Division a / b ohne Rest** und **Mod(a, b)** der **Divisionsrest** von **a / b** ist, dann gilt allgemein immer:

a = Div(a, b) *b + Mod(a, b) , d.h. **17 = Div(17, 5) * 5 + Mod(17, 5) = 3 * 5 + 2** ,

d.h. **q = Div(a, b)** und **Mod(a, b) = r**

Die entscheidende **Idee des Euklidischen Algorithmus** besteht nun darin, den **Satz von der Division mit Rest** nach dem Prinzip der "Wechselwegnahme" zu iterieren.

Dazu ersetzt man nach der Durchführung der Division mit Rest die ursprünglich größere Zahl **a durch die ursprünglich kleinere Zahl b** und **b durch den Rest r**.

Mit diesen **neuen Zahlen a und b** führt man wiederum das **Verfahren der Division mit Rest** durch und erhält ein **neues q** und ein **neues r**.
Mit diesen Zahlen verfährt man wiederum nach dem Prinzip der Wechselwegnahme und nimmt die kleinere so lange von der größeren weg wie es geht.

```
#include <stdio.h>              // EuklidAlg.c
#include <math.h>

// ggt(n,m) wird m, wenn n durch m teilbar, anderenfalls ist er gleich dem ggt(m, Rest(n/m))

long ggT(long n, long m){       // rekursiv
      if(abs(m)>abs(n)) return abs(ggT(m,n));
      if(!m) return abs(n);
      return abs(ggT(m,n%m));
}

// ggT(6,15) -> ggT(15,6) -> ggT(6,15%6) -> ggT(6,3) -> ggT(3,0) -> 3
//      n  m          n m         n    m         n m         n m   ggt

long Euklid(long a, long b){    // iterativ
      a = abs(a); b=abs(b);
      while(a*b){
          if(a >= b) a = a % b;
          else b = b% a;
      }
      return a + b;
}

void main(){
  int a = 0, b = 0;
  printf("ganze Zahl a     =     "); scanf("%5d", &a); while(getchar()!='\n'); // a  =   6
  printf("ganze Zahl b     =     "); scanf("%5d", &b); while(getchar()!='\n'); // b  =  15
  printf("ggT( %d , %d )    = %5d\n", a, b, ggT(a,b));    // rekursiv:   ggT( 6 , 15 ) =   3
  printf("Euklid( %d , %d ) = %5d\n", a, b, Euklid(a,b)); // iterativ: Euklid( 6, 15 ) =   3
  getchar();
}
```

3.3 Sieb des Eratosthenes

(Eratosthenes von Kyrene ca. 276-194 v.Chr.)

Jede natürliche Zahl **n** besitzt die „trivialen“ Teiler **1** und **n** selbst; jede von **1** verschiedene natürliche Zahl besitzt also **mindestens zwei Teiler**. Zahlen, die genau diese beiden trivialen Teiler besitzen, nennt man **Primzahlen**.

Nach dieser Definition wird die Zahl 1 also nicht zu den Primzahlen gerechnet. Ein Grund ist, daß sonst der **Fundamentalsatz der Zahlentheorie** nicht gelten würde.

Die Primzahlen sind einer der ältesten und interessantesten Untersuchungsgegenstände der Mathematik. Sie stellen u.a. die Bausteine dar, aus denen die natürlichen Zahlen (multiplikativ) aufgebaut sind.

Der **Fundamentalsatz der Zahlentheorie** besagt:

Jede natürliche Zahl n (n > 1) ist als Produkt von Primzahlen darstellbar: $\mathbf{n = p_1 * p_2 *... * p_s}$. Abgesehen von der Reihenfolge der Faktoren ist diese Darstellung **eindeutig**.

Auch für andere Zahlensysteme oder algebraische Systeme sind Primzahlen, Primelemente oder dem Primzahlbegriff nachgebildete Begriffe von zentraler Bedeutung.

Eine faszinierende Eigenschaft der Primzahlen ist die Unregelmäßigkeit, mit der sie in der Zahlenreihe auftreten.

Gesetzmäßigkeiten in der Primzahlreihe zu entdecken, war schon immer eine wichtige Forschungsrichtung in der Mathematik.

Schon im Altertum war man bestrebt, einen möglichst guten Überblick über die Primzahlen zu gewinnen. **Euklid** zeigte, daß es **unendlich viele Primzahlen** gibt.

Der griechische Mathematiker **Eratosthenes von Kyrene** gab das folgende Verfahren an, um **alle Primzahlen** bis zu einer bestimmten vorgegebenen Zahl **n** zu bestimmen.

Es sei hier am Beispiel **n = 20** erläutert.

1. Schreibe alle Zahlen von 1 bis 20 auf:
2. Streiche die Zahl 1 (sie wird aus guten Gründen nicht zu den Primzahlen gerechnet):
3. Unterstreiche die Zahl 2:
4. Streiche alle echten Vielfachen von 2; also die Zahlen 4, 6, 8, 10, 12, 14, 16, 18 und 20.
5. Unterstreiche die erste freie (d.h. noch nicht unterstrichene oder gestrichene) Zahl; in diesem Fall also die Zahl 3.
6. Streiche aus den verbleibenden Zahlen alle echten Vielfachen von 3; also die Zahlen 9 und 15.
7. Unterstreiche die kleinste freie Zahl; in diesem Fall also die Zahl 5.
8. Streiche aus den verbleibenden Zahlen alle echten Vielfachen der Zahl 5. Da die in Frage kommenden Zahlen 10, 15 und 20 bereits gestrichen sind, tritt in diesem Fall (Maximum=20) keine Veränderung auf.
9. Setze das Verfahren sinngemäß so lange fort, bis jede der Zahlen entweder unterstrichen oder gestrichen ist.
10. Ende des Verfahrens. Die unterstrichenen Zahlen sind die Primzahlen zwischen 1 und 20.

Durch dieses Verfahren werden also genau die Primzahlen „ausgesiebt“. Man nennt das Verfahren deshalb auch das **Sieb des Eratosthenes** bzw. kurz das Siebverfahren (englisch: sieve).

3.4 Verfahren zur Bestimmung der Primfaktoren

Den Algorithmus zur Bestimmung der **Primfaktoren** für ein ganzzahliges **n** ist folgender:
a.) Ist n gerade, dann ist ein Faktor 2 und das Verfahren ist auf n/2 anzuwenden.
b.) Ist n durch 3 teilbar, dann dividiere durch 3 und wende das Vefahren auf den Quotienten an.
c.) Ist n durch 5 teilbar, dann dividiere durch 5 und wende das Verfahren auf den Quotienten an.
d.) ....
Anscheinend muß man die Schritte wie b und c für alle Primzahlen durchführen. Man kann aber auch diesen Schritt für alle ungeraden Zahlen nacheinander durchführen, da z.B. ein Faktor 9 schon vorher als 3*3 eliminiert wurde. Das führt zu einem rekursiven Programm. Wir nehmen an, daß die Funktion PFR(n) als Ergebnis die Liste der Primfaktoren liefert, z.B.:
PFR(42) = (2,3,7) oder PFR(330) = (2, 3, 5, 11)

Jede natürliche Zahl **n>0** kann eindeutig in **Primfaktoren** zerlegt werden, z.B. 12 = 2*2*3 . Primzahlen selbst sind mit ihrem einzigen Primfaktor identisch, z.B. 17 = 17. Rekursiv läßt sich die **Liste *PF*** (*ArrayList*) der Primfaktoren ***PFR(decimal n)*** einer natürlichen Zahl n>0 wie folgt ermitteln (solange der lokale Stack nicht überläuft ...):

```
PFR(n) := WENN(Rest(n/2) == 0) DANN {Addiere 2 zu PF; PFR(n/2); }
                               SONST PF1(n,3);
```

mit

```
PF1(n, m) :=  WENN(n == 1) return;
              WENN(n<m²) DANN Addiere n zu PF;
                         SONST WENN(Rest(n/m) == 0)
                                   DANN { Addiere m zu PF; PF1(n/m,m); }
                                   SONST PF1(n, m+2);
```

Eine Implementation der rekursiven Berechnung in C:

```
#include <stdio.h>          // Primfaktoren rekursiv
#include <ctype.h>


void PF1(unsigned long long, unsigned long long);

void PFR(unsigned long long n)
{
    if (n % 2ULL == 0ULL)
    {
        printf(", 2");
        PFR(n / 2ULL);
    }
    else PF1(n, 3ULL);
}


void PF1(unsigned long long n, unsigned long long m)
{
    if (n == 1ULL) return;
    if (n < m * m) printf(", %llu",n);
    else if(n % m == 0ULL){printf(", %llu",m);PF1(n/m,m);}
         else PF1(n, m + 2ULL);
}
```

```
void CalcPrim(unsigned long long from, unsigned long long to)
{
    unsigned long long i=0ULL;
    for(i = from; i <= to; i++)
    {
        printf("%10llu : ",i);
        PFR(i);
        printf("\n");
        if(i==to) break;  // i++ > to bringt overflow !!
    }
}

void main()
{
  unsigned long long von = 1ULL, bis = 1ULL;
  do {
       printf("Berechnung der Primfaktoren:\n");
       printf("von = "); scanf("%lli", &von); while(getchar()!='\n');
       printf("bis = "); scanf("%lli", &bis); while(getchar()!='\n');
       CalcPrim(von, bis);
       printf("erneute Berechnung ? [y/n]: ");
  } while(tolower(getchar())=='y');
  getchar();
}

/*
Berechnung der Primfaktoren:
von = 18446744073709551615
bis = 18446744073709551615
18446744073709551615 : , 3, 5, 17, 257, 641, 65537, 6700417
erneute Berechnung ? [y/n]: n

Stack einstellen im VS2012:
Projekt | prim-Eigenschaften | Link | System |
Stapelreservierungsgroesse | 9999999999999999999

Berechnung der Primfaktoren:
von = 18446744073709551612
bis = 18446744073709551612
18446744073709551612 : , 2, 2, 3, 715827883, 2147483647
erneute Berechnung ? [y/n]:
*/
```

Alternativ können die Primfaktoren auch **iterativ** wie folgt ermittelt werden: Zuerst wird versucht, mit der **2** zu teilen, ist das nicht mehr möglich, wird mit der **3** fortgesetzt. Danach werden die Teiler um **2** erhöht, falls die Teilung ohne Rest nicht mehr möglich ist. Primfaktoren können nur bis <= sqrt(n) auftreten. Beispiel in C:

```
/* primes.c */
#include <stdio.h>
#include <math.h>

void primefactorization(unsigned long long x)
{
  unsigned long long i = 3ULL; /* counter */
  unsigned long long c = x;    /* remaining product */

  while ((c % 2ULL) == 0ULL) {
      printf("%llu, ",2ULL);
      c = c / 2ULL;
  }

  while (i <= (sqrt((double)c)+1.0)) {
     if ((c % i) == 0ULL) {
          printf("%llu, ",i);
          c = c / i;
     }
     else
          i = i + 2ULL;
  }

  if (c > 1ULL) printf("%llu, ",c);
}

main() {
        unsigned long long p = 0ULL;
        while(getchar()!='\n'); printf("Zahl = ");
        while (scanf("%lli",&p)!=EOF) {
               while(getchar()!='\n');
               printf("prime factorization of p=%llu \n", p);
               primefactorization(p);
               printf("\n\nZahl = ");
       }
}

/*
Zahl = 18446744073709551615
prime factorization of p=18446744073709551615
3, 5, 17, 257, 641, 65537, 6700417,

Zahl = 18446744073709551613
prime factorization of p=18446744073709551613
13, 3889, 364870227143809,
Zahl = ^Z
*/
```

3.5 Verbesserung des Ansatzes von Erathostenes zur Berechnung von Primzahlen

Gemäß der Definition von Primzahlen gilt fur jede Zahl **m**, die **keine Primzahl** ist:
Es gibt zwei Zahlen **i**, **k** mit den Eigenschaften **2 <= i, k <= m** und **i * k = m** .

Diese Gesetzmäßigkeit können wir benutzen, um einen sehr einfachen Algorithmus für eine Primzahltabelle zu formulieren:

. schreibe alle Zahlen von **2** bis **n** in eine Liste,
. bilde alle Produkte **i * k**, wobei **i** und **k** Zahlen zwischen **2** und **n** sind und
. streiche alle Ergebnisse, die vorkommen, aus der Liste.

Dass diese einfache Vorschrift das Gewünschte leistet, ist sofort zu sehen:

Alle Zahlen, die am Schluss noch in der Liste stehen, kamen nicht als Produktergebnis vor. Sie können also nicht als ein solches Produkt geschrieben werden und sind demnach Primzahlen.

Pseudocode: intuitiver Ansatz Primzahlentabelle (Rechenzeit 3 Ghz CPU für 10^6 ca. 2000 s)

```
procedure Primzahltabelle
begin
    schreibe alle Zahlen von 2 bis n in eine Liste
    for i := 2 to n do
         for k := 2 to n do
              streiche die Zahl i*k aus der Liste
         endfor
    endfor
end
```

Kann **i*k** nicht in der Liste gefunden werden, dann passiert nichts.

Wegen **i * k = k * i** kann **k >= i** festlegt werden, um diese Doppelungen bei der Berechnung zu vermeiden.

Wenn andererseits **i *k > n** ist, dann ist auch nichts zu streichen, da **Zahlen > n** nicht in der Liste stehen. Die **k-Schleife** braucht also nur für solche Werte ausgeführt werden, für die **i * k <= n** ist. Diese Bedingung selbst sagt uns, welche **k** das sind: **k <= n / i** .
Als Nebeneffekt können wir auch den Laufbereich für **i** begrenzen. Aus den beiden Einschränkungen **i <= k <= n / i** erhalten wir **i*i <= n**, also $i <= \sqrt[2]{n}$. Für höhere **i** ist der **k**-Bereich leer.

Pseudocode: verbesserter Algorithmus (Rechenzeit 3 Ghz CPU für 10^6 ca. 0.01 s, 10^9 ca 450 s)

```
procedure Primzahltabelle
begin
    schreibe alle Zahlen von 2 bis n in eine Liste
    for i := 2 to √n do
        for k := i to n/i do
            streiche die Zahl i*k aus der Liste
        endfor
    endfor
end
```

Beispiel **i = 9**: Das Produkt **i * k** durchläuft nur Vielfache von 9. Die sind aber als Vielfache von 3 schon durchlaufen und ebenfalls unnötig. So ist das mit allen Nichtprimzahlen, denn sie haben einen kleineren Primteiler, der vor ihnen schon ein Wert für **i** war.

Wir brauchen also die **k-Schleife nur für Primzahlen i** auszuführen, das hatte bereits **Eratosthenes** erkannt:

Verfahren von Eratosthenes (Rechenzeit 3 Ghz CPU für 10^6 ca. 0.02 s, 10^9 ca 70 s)

```
procedure Primzahltabelle
begin
   schreibe alle Zahlen von 2 bis n in eine Liste
   for i := 2 to √n do
       if i steht in der Liste then
          for k := i to n/i do
              streiche die Zahl i*k aus der Liste
          endfor
       endif
   endfor
end
```

Mit einem ähnlichen Argument wie für die i-Werte können wir auch die Werte der **k-Schleife** weiter einschränken: Wir brauchen nur diejenigen zu betrachten, die in der Liste gefunden werden. Steht **k** nämlich nicht mehr dort, ist **k als Nichtprimzahl** gestrichen worden und besitzt einen **Primteiler p < k**. Im Durchgang der **i-Schleife** mit **i = p** wurden alle Vielfachen von **p** gestrichen, insbesondere **k** und seine **Vielfachen**. Es ist nichts mehr zu tun.

Es wäre nun nahe liegend, den Algorithmus wie folgt zu ergänzen:

```
for k := i to n/i do
     if k steht in der Liste then
        streiche die Zahl i*k aus der Liste
     endif
```

Lassen wir den Algorithmus so laufen, erzeugt er die folgende Tabelle:

2 3 5 7 ***8*** 11 ***12*** 13 17 19 ***20*** 23 ***27*** ***28*** 29 31 ***32*** 37 . . .

Was läuft falsch? Sehen wir uns die ersten Schritte des Algorithmus genau an. Nach der Initialisierung der Liste mit allen Zahlen bis n steht darin:
2 3 4 5 6 7 8 9 10 11 ...

Zunächst wird i = 2 gesetzt und dann k = 2. Die 2 steht in der Liste, also wird i * k = 4 gestrichen:
2 3 – 5 6 7 8 9 10 11 ...

Im nächsten Schritt ist k = 3. Die 3 steht ebenfalls in der Liste und so wird i * k = 6 gestrichen:
2 3 – 5 – 7 8 9 10 11 ...

Jetzt passiert's: k = 4 steht nicht mehr in der Liste, denn sie wurde ja als erste gestrichen. Entsprechend wird für k = 4 wegen der neuen Bedingung nichts gemacht und der Algorithmus fährt mit k = 5 fort:
2 3 – 5 – 7 8 9 – 11 ...

So bleibt 2 *4 = 8 irrtümlich in der Liste stehen. Das Problem ist, dass k beim Schleifendurchlauf stets größere Zahlen i *k > k streicht und dann selbst erhöht wird. Irgendwann bekommt k den Wert eines vormaligen Produktes i * k und das Verfahren wirkt ungünstig auf sich selbst zurück:

Die **Lösung** besteht darin, **k** seinen Schleifenbereich **rückwärts** durchlaufen zu lassen. Dadurch wird diese Rückwirkung vermieden:

Primzahltabelle (Endversion) (Rechenzeit 3 Ghz CPU für 10^6 ca. 0.01 s, 10^9 ca 18 s)

```
procedure Primzahltabelle
begin
    schreibe alle Zahlen von 2 bis n in eine Liste
    for i := 2 to √n  do
        if i steht in der Liste then
           for k := n/i to i step -1 do
               if k steht in der Liste then
                  streiche die Zahl i*k aus der Liste
               endif
           endfor
        endif
    endfor
end
```

Um mit dem ersten intuitiven Algorithmus auf $\mathbf{n = 10^9}$ zu kommen, müssten wir die Zeit bei $\mathbf{n = 10^6}$ um den Faktor $\mathbf{(10^9/10^6)^2 = 10^6}$ erhöhen und würden dafür $\mathbf{1943*10^6}$ Sekunden = **61 Jahre** und **7 Monate** brauchen. Gegenüber dem sind die 18 Sekunden des letzten Algorithmus eine Reduzierung um den Faktor 254 500 000 .

Folgende Schlußfolgerungen können aus dem Beispiel gezogen werden:

1. Einfache Rechenverfahren sind **nicht automatisch effizient**.
2. Zu ihrer Beschleunigung muss man sie **gut verstehen**.
3. Es sind oft **viele Verbesserungen** möglich.
4. **Mathematische Ideen** können sehr **weitreichend** sein!