

Die LUG programmiert das $3m + 1$ -Problem

Ein ungelöstes Problem der Zahlentheorie ist, ob man bei mehrfacher Anwendung der unten aufgeführten Bildungsvorschrift (1) für jede natürliche Zahl 1 erreicht.

Die Annahme ist, dass es so ist – Vermutung von Collatz. Jedoch konnte bis zum heutigen Tage noch kein Beweis dafür erbracht werden. Man konnte auch keine natürliche Zahl finden, für die dies nicht gilt.

Hier soll es jedoch nicht um den Beweis dieser These gehen, sondern darum, für welche Zahl eines Intervalls man die meisten Schritte benötigt.

Aufgabenstellung: Gegeben sei die Bildungsvorschrift

$$x_{k+1} = \begin{cases} \frac{x_k}{2} & : x_k \text{ gerade,} \\ 3x_k + 1 & : \text{sonst} \end{cases} \quad (1)$$

Gesucht ist

$$\max_{x_1 \in [1, n]} \{k : x_k = 1\} \quad (2)$$

1. Versuch (Brute-Force)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #if 0
5 # define N 1000
6 # define PRINT_ALL
7 #else
8 # define N 1000000
9 #endif
10
11 int main(void)
12 {
13     int x1, kmax=0, x1max=0;
14
15     for (x1=1; x1 <= N; ++x1) {
16         register long long xk=x1;
17         register int k=1;
18         while (xk != 1) {
19             if (xk % 2 == 0)
20                 xk /= 2;
21             else
22                 xk = 3*xk+1;
23
24             ++k;
25         }
26 #ifdef PRINT_ALL
27         printf("%d %d\n", x1, k);
```

```

28 #endif
29
30     if (k > kmax) {
31         kmax = k;
32         x1max = x1;
33     }
34 }
35
36 #ifndef PRINT_ALL
37     printf("Max bei %d mit %d Schritten\n", x1max, kmax);
38 #endif
39
40     return EXIT_SUCCESS;
41 }

```

Für $n = 10^6$ ergibt sich $k=525$ für $x_1=837\,799$ in 1,93 Sekunden. Für $n = 10^7$ ergibt sich $k=686$ für $x_1=8\,400\,511$ in 22,57 Sekunden.

2. Versuch

Wenn man sich mehrere Folgen x_k für unterschiedliche x_1 ansieht, so fällt auf, dass man sehr viele Berechnungen mehrfach durchführt – für die Zahlen $x_k < x_1$ wurde die Anzahl der Schritte schon mindestens einmal vorher ermittelt.

Abbildung 1 zeigt, dass man im Großteil der Fälle nach weniger als 10 Schritten ein x_k erreicht, das kleiner ist als x_1 . Für gerade Zahlen – die Hälfte der Zahlen in einem Intervall – erreicht man sogar nach genau einem Schritt ein $x_k < x_1$.

Man könnte also in einen Array zu jedem x_1 das Ergebnis k vermerken. Kommt man bei einer späteren Berechnung auf ein $x_k < x_1$, so ist die Anzahl der Schritte von x_1 gleich der Summe von k und $array[x_k]$

Wir zählen also nur noch die Schritte (bei 0 beginnend) bis $x_k < x_1 \vee x_k = 1$ ($x_k = x_i$ sollte nach der math. Annahme nie auftreten – Zyklus). Jedoch führt dies zu Problemen bei der Berechnung für $x_1 = 1$, da hier die Schleife nie durchlaufen wird und somit $array[1]$ den Wert 0 bekäme. Dieses würde sich wiederum auf *alle* folgenden Berechnungen auswirken. Also setzen wir diesen einen Wert zu Beginn und starten mit $x_1 = 2$.

Da die Bedingung $x_k < x_1$ für alle $k > 1$ zum Abbruch führt, benötigen wir die Bedingung $x_k=1$ nicht mehr.

```

23     array[0] = 1;
24     for (i=1; i < N; ++i) {
25         register long long xk=i+1;    /* x1 = i+1 */
26         register int cnt=0;
27
28         while (xk > i) {
29             if (xk % 2 == 0)
30                 xk /= 2;
31             else
32                 xk = 3*xk+1;

```

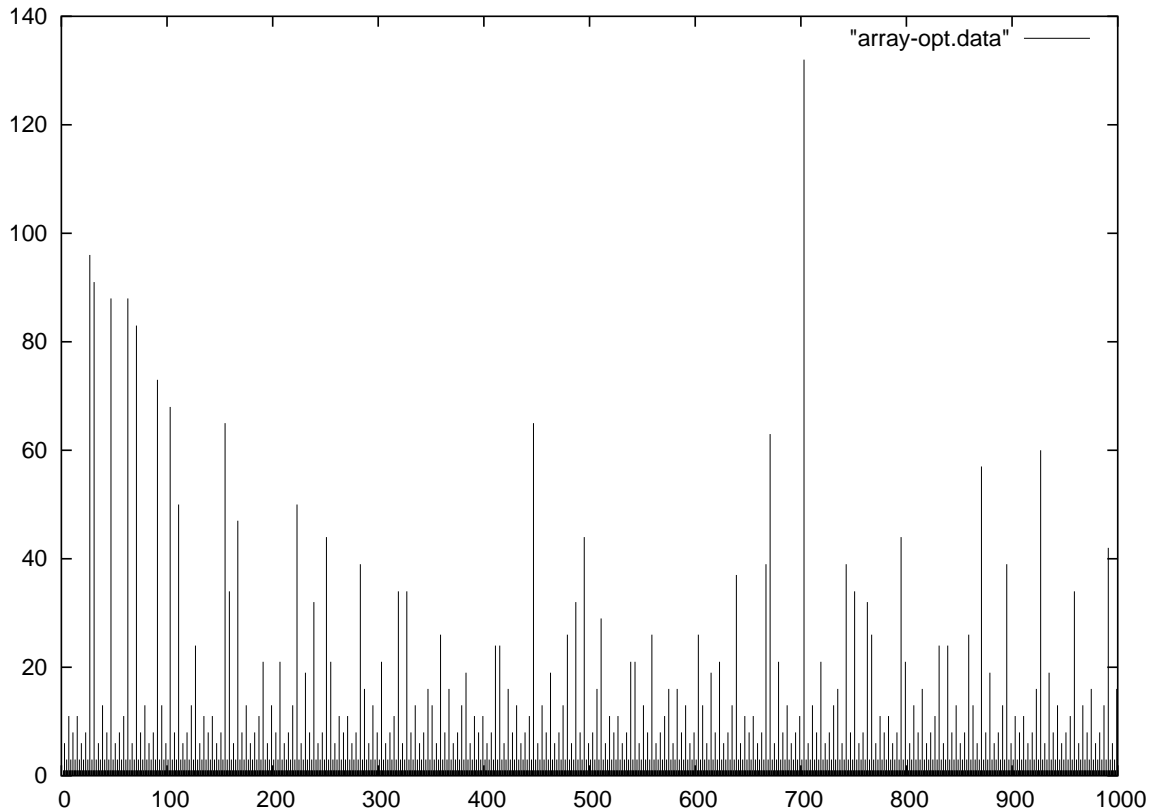


Abbildung 1: Anzahl der Schritte bis $x_k < x_1$ für $x_1 \in [1, 1000]$

```

33     ++cnt;
34   }
35   array[i] = cnt + array[xk-1];
36

```

Für 10^6 ergibt sich $k=525$ für $x_1=837\,799$ in 0,15 Sekunden. Also ein Laufzeitverbesserung um 92% gegenüber dem Brute-Force-Algorithmus. Für 10^7 ergibt sich $k=686$ für $x_1=8\,400\,511$ in 1,68 Sekunden.

3. Versuch

Ist z_n eine ungerade Zahl, d. h. $z_n = 2\alpha + 1$, ergibt der nächste Schritt

$$z_{n+1} = 3z_n + 1 = 3(2\alpha + 1) + 1 = 6\alpha + 4 = 2(3\alpha + 2) \quad (3)$$

immer eine gerade Zahl. Somit bringt der darauf folgende Schritt

$$z_{n+2} = \frac{z_{n+1}}{2} = 3\alpha + 2 = \frac{z_n + 1}{2} + z_n \quad (4)$$

Man kann also im Fall einer ungeraden Zahl zwei Schritte auf einmal machen.

```

29     if (xk % 2 == 0) {
30         xk /= 2;
31         ++cnt;
32     } else {
33         xk += (xk+1)/2;
34         cnt += 2;
35     }

```

Für 10^7 ergibt sich $k = 686$ für $x_1 = 8400511$ in 1,59 Sekunden.

4. Versuch

Für gerade Zahlen kann man die `while`-Schleife einsparen, da für alle geraden Zahlen nach genau einem Schritt die Berechnung abbricht und das Ergebnis feststeht: $array[x_i] = array[\frac{x_i}{2}] + 1$.

Um immer eine gerade Zahl zu bekommen, teilt man die Schleife so auf, dass in einem Durchlauf zwei aufeinander folgende Zahlen berechnet werden. Damit ist eine der beiden Zahlen gerade, die andere ungerade.

Für ungerade Zahlen kann man den ersten Schritt dann noch vor der `while`-Schleife berechnen und spart so die erste `if`-Prüfung.

Durch diese Aufteilung muss man jedoch `array[2]` ebenfalls setzen, da man sonst Probleme mit dem Schleifenanfang oder -ende bekommt.

```

23     array[0] = 1;
24     array[1] = 2;
25     for (i=2; i < N; ++i) {
26         register long long xk= i + i/2 +2;
27         register int cnt=2;
28
29         while (xk > i) {
30             if (xk % 2 == 0) {
31                 xk /= 2;
32                 ++cnt;
33             } else {
34                 xk += (xk+1)/2;
35                 cnt += 2;
36             }
37         }
38         array[i] = cnt + array[xk-1];
39
40         if (array[i] > kmax) {
41             kmax = array[i];
42             x1max = i+1;
43         }
44
45         ++i;
46         array[i] = array[ (i-1)/2 ]+1;

```

```

47         if (array[i] > kmax) {
48             kmax = array[i];
49             x1max = i+1;
50         }
51     }
52
53 #ifdef PRINT_ALL
54     printf("%d %d\n%d %d\n", i, array[i-1], i+1, array[i]);
55 #endif
56 }

```

Für 10^7 ergibt sich $k=686$ für $x_1=8\,400\,511$ in 1,38 Sekunden.

5. Versuch

Als nächstes sollte man auch einmal einen Blick auf die Datentypen und Kontrollstrukturen werden.

So fällt z. B. auf, dass zu Beginn der `while`-Schleife die Bedingung *immer* erfüllt ist. Es wäre also günstiger eine fußgesteuerte Schleife (`do-while`) der kopfgesteuerten Schleife vorzuziehen.

Betrachtet man einmal die Ergebnisse für 10^7 , so sieht man, dass diese bei weitem nicht an 2^{16} herankommen. Daher wäre ein `short` als Datentyp vollkommen ausreichend.

Für alle Variablen sind negative Werte ausgeschlossen, so dass man auch für alle einen `unsigned`-Datentyp wählen sollte.

Durch diese Umstellung erhalten wir nochmals eine Verbesserung von 25%. Für 10^7 ergibt sich $k=686$ für $x_1=8\,400\,511$ in 1,02 Sekunden.

6. Versuch

Die Umstellung des Datentyps für den Array hat uns schon einen Gewinn bei der Speichergröße gebracht. Man kann dies aber auch noch etwas ausbauen. Man sieht leicht, dass das kleinste Element, auf das man bei einer Berechnungsfolge zugreift, mindestens den Index $\frac{n-1}{2}$ hat. Es ist also nicht notwendig die Elemente $\leq \frac{n-1}{2}$ aufzuheben.

Damit ergibt sich also eine Größe von $\frac{n}{2}$ für das Array, was jedoch eine Umstellung beim Indexzugriff bedeutet.

Es gibt zwei Möglichkeiten: Man kann eine oder zwei `for`-Schleifen verwenden. Die erste Möglichkeit hat den Vorteil, dass sich die Code-Größe nicht wesentlich erhöht. Jedoch muss man Überprüfungen einbauen, damit man ggf. den Index umrechnet.

```

29     unsigned int ii=(i>=N/2)?i-N/2:i;
30
31     do {
32         if (xk % 2 == 0) {
33             xk /= 2;
34             ++cnt;

```

```

35     } else {
36         xk += (xk+1)/2;
37         cnt += 2;
38     }
39 } while (xk > i);
40
41 if (xk > N/2)
42     xk -= N/2;
43
44 array[ii] = cnt + array[ xk-1 ];

```

Für 10^7 ergibt sich $k=686$ für $x_1=8400511$ in 1,05 Sekunden.

Bei zwei `for`-Schleifen (eine von $1-\frac{n}{2}$, die andere von $\frac{n}{2}+1-n$) benötigt man keine Umrechnung der Indexvariable, da in beiden die Zuordnung fest ist.

Da da Zahl $i + \frac{n}{2}$ in der zweiten `for`-Schleife häufig benötigt wird, verwendet man besser eine Variable dafür.

```

61     unsigned int ii = i+N/2;
62     register unsigned long long xk = ii + ii/2 +2;
63     register unsigned short cnt=2;
64
65     do {
66         if (xk % 2 == 0) {
67             xk /= 2;
68             ++cnt;
69         } else {
70             xk += (xk+1)/2;
71             cnt += 2;
72         }
73     } while (xk > ii);
74
75     if (xk > N/2)
76         xk -= N/2;
77
78     array[i] = cnt + array[xk-1];

```

Für 10^7 ergibt sich $k=686$ für $x_1=8400511$ in 1,03 Sekunden.

Jedoch zeigt der Vergleich der Laufzeiten nur eine sehr kleine Verbesserung, die in sehr starkem Gegensatz zur Verschlechterung der Wartbarkeit des Codes steht!

7. Versuch

Aufbauend auf der Trennung des Bereichs in $1-\frac{n}{2}$ und $\frac{n}{2}+1-n$ kann man noch für den ersten Teilbereich die Überprüfung auf das Maximum weglassen, da in dem Intervall sich das Maximum nicht befinden kann.

Zu jeder Zahl x_k im Intervall von $[1, \frac{n}{2}]$ gibt es eine Zahl $2x_k$ im Intervall $[\frac{n}{2} + 1, n]$, die einen Schritt mehr benötigt als x_k .

Versuch	n	k_{max}	x_0	Zeit	Zeit rel. zum Vorgänger	... zum Anfang
1	10^6	525	837 799	1,93	–	–
1	10^7	686	8 400 511	22,57	–	–
2	10^6	525	837 799	0,15	7,8%	7,8%
2	10^7	686	8 400 511	1,68	7,24%	7,24%
3	10^7	686	8 400 511	1,59	94,6%	7,0%
4	10^7	686	8 400 511	1,38	86,7%	6,1%
5	10^7	686	8 400 511	1,02	73,9%	4,5%
6 1 Schl.	10^7	686	8 400 511	1,05	102,9%	4,6%
6 2 Schl.	10^7	686	8 400 511	1,03	101,0%	4,5%
7	10^7	686	8 400 511	0,99	96,1%	4,3%

Tabelle 1: Auflistung der Ergebnisse der gemachten Versuche

Ausblick

Für eine ungerade Zahl $m = 2k + 1$ liefert die Bildungsvorschrift (1) im ersten Schritt die Zahl

$$3m + 1 = 3(2k + 1) + 1 = 6k + 4 \quad (5)$$

und im nächsten Schritt

$$\frac{6k + 4}{2} = 3k + 2 \quad (6)$$

Die Anzahl der Schritte für die Zahlen $3m + 1$ und $\frac{3m+1}{2}$ sind weniger als die für die Zahl m . Somit können sie nicht das Maximum sein.

Ein anderer Algorithmus könnte also wie im 1. Versuch vorgehen, jedoch nicht alle Zahlen berechnen, sondern nur ausgewählte, die das Maximum sein könnten.

Vielleicht gibt es noch eine bessere Einschränkung als die eben gemachte und das Maximum kann z. B. nur von jeder 17. Zahl angenommen werden. Jedoch hatte ich keine Idee, dies weiter einzuschränken, und die Einschränkung sollte min. jede 14. Zahl bedeuten, damit man nur noch jeder 14. Berechnung durchführen muss, um in den Bereich der Optimierung von Versuch 2 zu kommen.

Auswertung der Ergebnisse

Die Auflistung aller Ergebnisse in Tabelle 1 zeigt recht gut, dass man große Verbesserungen i. A. nur mit einer neuen Idee (einem neuen Algorithmus) erreicht. In diesem Fall hier war es eine Verbesserung um 92%.

Auch zeigt sich an dieser Aufgabe, dass die Wahl der Datentypen und Kontrollstrukturen nicht unerheblich für die Laufzeit des Programms ist.

Anhand des Beispiels lässt sich auch gut die Optimierung des Compilers untersuchen. In Tabelle 2 sind die Zeiten der einzelnen Versuche für verschiedene Optimierungsstufen

Versuch	-00	-01	-02	-03	-0s
2	4,76	1,68	1,73	1,73	1,76
3	3,92	1,58	1,69	1,69	1,66
4	3,03	1,37	1,43	1,43	1,41
5	2,38	1,02	1,08	1,08	1,18
6 1 Schl.	2,48	1,04	1,08	1,09	1,28
6 2 Schl.	2,46	1,03	1,05	1,05	1,01
7	2,38	0,99	0,99	1,00	1,08

Tabelle 2: Gegenüberstellung der Laufzeiten der verschiedenen Optimierungsstufen des Compilers, $n = 10^7$

gegenüber gestellt. Dabei sieht man, dass der Compiler in der sonst überlichen Stufe -02 schlechteren Code produziert, als in der einfachen Stufe -01.

Anmerkung

Ich habe alle Versuche auf einem iBook G4 800 MHz mit dem gcc 3.3 und den Optionen -Wall -ansi gemacht. Für die Ergebnisse aus Tabelle 1 habe ich zusätzlich die Option -O1 verwendet.