Modular Arithmetik mit mehreren Exponenten

Allgemeines Verfahren

$$\forall\, Variabeln \in \mathbb{N}$$

$$b_n^{b_{n+1}^{\cdots^{b_{max}}}} = s_n$$

Die komplette Zahl sei s_0, also ist die «tiefste» Basis b_0.

Hier ist das Verfahren für eine Stufe gezeigt. Das gesamte Verfahren ist rekursiv und beginnt immer bei Index 0, resp. $s_0 \equiv r_0 \; mod \; y_0$. Alle Variablen welche ohne Index geschrieben sind haben Index n (gewisse Variablen haben einen Index, welcher aber kein n enthält, diese Indices beziehen sich nicht auf die Stufe).

Wir möchten $s \; mod \; y$ berechnen, es gilt: $s = b^{s_{n+1}} \equiv r \; mod \; y$.

Wenn: $y = 1$, diese Bedingung ist nicht nötig für die Korrektheit des Verfahrens

$$s \equiv 0 \; mod \; 1$$

Wenn: $s = s_{max}$

$$b \equiv r \; mod \; y$$

Wenn: $s_{n+1} \leq y$

$$b^{s_{n+1}} \equiv r \; mod \; y$$

Wenn: $s_{n+1} > y$

Wir definieren: $0 \leq o \leq w < d \leq y$

$$b^0 \equiv t_0, \cdots, b^o \equiv t_o, \cdots, b^w \equiv t_w, \cdots, b^d \equiv t_d \; mod \; y$$

Wichtig sind die Exponenten o und d, denn für diese gilt: $t_o = t_d$, o und d sind die kleinsten Zahlen für welche dies gilt. Es gibt immer ein o und d weil es nur y Werte für t gibt. Wir definieren: $l = d - o$

$$o \equiv m_o, \cdots, w \equiv m_w, \cdots, d \equiv m_d \; mod \; l$$

Im Folgenden werde ich zeigen, dass wenn $s_{n+1} \equiv m_w \; mod \; l$ gilt, auch $b^{s_{n+1}} \equiv t_w \; mod \; y$ gilt.

$$w = m_w + q * l$$

$$b^{s_{n+1}} = b^{k*l + m_w} = b^{(k+q)*l + w}$$

Es gilt: $b^o * b^l = b^{o+d-o} \equiv t_d = t_o \; mod \; y$, wir können diese Identität so oft wie wir wollen anwenden, deshalb gilt auch: $b^o * (b^l)^{(k+q)} \equiv t_o \; mod \; y$

$$b^w = b^o * b^{w-o} \equiv t_o * t_{w-o} \equiv t_w \; mod \; y$$

$$b^{s_{n+1}} = b^{(k+q)*l + w} = b^o * (b^l)^{(k+q)} * b^{w-o} \equiv t_o * t_{w-o} \equiv t_w \; mod \; y$$

Wie schon erwähnt wird für die Berechnung von s_n der $mod \; l$ Wert von s_{n+1} (r_{n+1}) benötigt, daher die Rekursion. Deshalb sind die Folgenden Definitionen nötig: $l = y_{n+1}, r_{n+1} = m_w, t_w = r$

Implementierung

In diesem Abschnitt geht es um eine mögliche Implementierung, des zuvor gezeigten Verfahren, in C++. Der Fokus liegt auf linearer Laufzeit für eine Rechnung und einfacher Nutzung. Es wird bei langem nicht alles im Detail besprochen, ich habe jedoch versucht die wichtigsten Aspekte zu erläutern.

Im Quellcode werden Sie solche «/*c42*/» und im Skript solche «(c42, h13, cpp78)» Kommentare finden. So können Sie das im Skript besprochene gleich im Quellcode nachvollziehen. (comment 42, in mame.h line13, in mame.cpp line78)

1. Einfache Nutzung

In C++ ist Objekt Orientierung möglich, dies ist einer der Gründe weshalb ich C++ und nicht C gewählt habe. Offensichtlich geht es in diesem Projekt um Integer mit mehreren Exponenten (Abkürzung: «ime»), deshalb habe ich solche Integer als Klasse resp. Datentyp definiert (c0, h5). Im Wesentlichen enthält die Klasse einen «std::vector» oder einfach gesagt ein dynamisches Array (c1, h8), welches alle b enthält. Im «public:» Teil ist der «constructor» (c2, h24, cpp151), eine «mod» (c3, h26, cpp154) Methode und eine Operator Überladung für «%» (c4, h28, cpp159) deklariert. Jedoch rufen «mod» und «%» nur die echte «m_mod» (c5, h20, cpp70) Methode auf, welche im «private:» Teil deklariert ist. Die restlichen «private:» Methoden werden im Abschnitt über Lineare Laufzeit besprochen.

2. Lineare Laufzeit und Algorithmus

2.1 Abbruchs Kriterien

$y = 1$ (c6, cpp73) und $s = s_{max}$ (c7, cpp79) sind trivial und werden hier nicht besprochen. Bei $s_{n+1} \leq y$ (c8, cpp85) müssen wir eine Zahl geschrieben in Exponenten mit einer ohne Vergleichen. Dafür verwenden wir Logarithmen. Wichtig zu wissen ist: Nicht grösser (not <) entspricht kleiner gleich (≥). Dies ist wichtig weil, in der Implementierten Version eigentlich $s_{n+1} > y$ überprüft wird. Die «m_log» Methode (c9, cpp26) verwendet den Logarithmus Naturalis aus der <cmath> library um einen Logarithmus mit wählbarer Basis zu berechen. Nun zur Methode mit den komischen Namen «m_sn_se_int» (c10, cpp33), dies ist eine Abkürzung für «s_n smaler or equal int». Es gilt: $s_n \geq b_n^{\cdots b_k}$ wenn: $n\_max \geq k$. In den folgenden Vergleichen kann das Verfahren nachvolzogen werden.

$$s_n \geq b_n > int$$

$$\log_{b_n}(s_n) \geq b_{n+1} * \log_{b_n}(b_n) = b_{n+1} * 1 > \log_{b_n}(int)$$

$$\dots$$

$$\log_{b_{max-1}}(\dots \log_{b_n}(s_n)\,) = b_{max} > \log_{b_{max-1}}(\dots \log_{b_n}(int)\,)$$

Wenn keine rote Bedingung korrekt ist, gilt $s_n \leq int$.

«m_pow» (c11, cpp49) berechnet eine Base mit Exponenten. «m_sn» (c12, cpp59) berechnet s_n mithilfe der «m_pow» Methode. Wieso habe ich nicht die «pow» Funktion aus <cmath> verwendet? Datentypen, bei der Berechnung von Gleitkomazahlen kann es Rundungsfehler geben. Deshalb habe ich «pow» spezifisch für Integer implementiert. Auch «log» ist betroffen. Ich weis nicht viel über Rundungsfehler, ausgenomen dass sie existieren. Wenn ich $s_{n+1} \leq y$ überprüfe habe ich $y + 100$ anstelle von y gewählt (c13, cpp87), um sicher zu gehen. Dieser Teil ist ein schwaches Glied im Programm, dass gebe ich offen zu.

2.2 o, d in Linearer Laufzeit (o, d, l werden im Abschnitt Allgemeines Verfahren eingeführt)

Wir sprechen hier von Linearität bezogen auf den Teiler resp. y_0. Die gesamte Laufzeit ist Linear wenn alle Teile linear sind, $O(n)$ ist in allen Teilen, ausgenommen der Bestimmung von d, unproblematisch. Kurz vorab, wenn t_d gefunden ist beginnen durch alle Potenzen zu iterieren bis wir t_o finden, dies ist linear (c14, cpp116). Auch bei der Bestimmung von d iterieren wir durch die Potenzen. Bei d suchen wir nicht nach einem bestimmten t, wir müssen also speichern welche t bereits vorkamen. Ganz wichtig: das Überprüfen ob ein t bereits bei einer kleineren Potenz existierte, muss konstante Laufzeit aufweisen, wenn die gesamte Laufzeit Linear sein soll. Eine Lösung ist ein boolean Array mit länge y (c15, cpp91). Der Wert an der Speicheradresse t speichert den momentanen Status von t (c16, cpp98).

2.3 von m_w zu t_w

In der «m_mod» Methode ist ein Stück Code für die Bestimmung von t_w gegeben m_w vorgeshen (c17, cpp135).

$$0, \dots, m_d \; oder \; m_o, \dots, l-1 \; mod \; l$$

In der Abbildung oben sehen wir die Folge aller Natürlichen Zahlen von 0 bis $l-1$, m_w liegt irgendwo in dieser Folge. Wir können zwei weitere Folgen aus Natürlichen Zahlen an die bereits existierende anheften. Von m_d gegen 0 werden Zahlen von d abwärts angeheftet. Von m_o gegen $l-1$ werden Zahlen von o aufwärts angeheftet. In dieser Konfiguration ist eine Zahl aus einer der angehefteten Folgen kongruent zu der dazugehörenden in der ursprünglichen Folge. Wir

berechnen den Abstand von m_w auf m_d *oder* m_o. Wenn der Abstand grösser gleich 0 ist addieren wir ihn zu o, weil m_w in diesem Fall grösser gleich m_o ist und w somit in der Angehefteten Folge mit o am Anfang liegt (c18, cpp137). Anderwärtig wird die Diverenz auf d addiert, die Argumentation «weshalb» ist Analog zur vorherigen (c19, cpp141).

2.4 «powmod»

Vorab, die Version welche ich im Skript verwende stammt nicht von mir, die Credits stehen im Code. Die Methode «m_powmod» (c20, cpp4) kann eine Base mit Exponenten sehr effizient Modulo rechen. Wir iterieren durch jedes Bit des Exponenten, wenn es gesetzt ist multplizieren $b^{2^j}\ mod$ (c21, cpp11). Beim ersten Bit währe dies $b^{2^0}\ mod$. Wir erhöhen j durch quadrieren, $(b^{2^j})^2 = b^{2*2^j} = b^{2^{j+1}}\ mod$ (c22, cpp15).

3. Range

In dieser Implementierung wird ein «unsigned long long int» als Datentyp verwendet, dieser Datentyp hat 64 Bit, also ist die höchste Zahl welche gespeichert werden kann $2^{64} - 1 = 18446744073709551615$. Im Programm multiplizieren wir soche Datentype miteinader, damit der wir keinen Speicher Overflow riskiren könne wir nur mit Zahlen kleiner als die Wurzel rechen. $\sqrt{2^{64} - 1} > 4294967294$. Ein guter Richtwert währe für alle b und für y_0 Zahlen unter einer Miliarde zu nehmen.