Modular Arithmetik mit mehreren Exponenten

Allgemeines Verfahren

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

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

Wir möchten berechnen, es gilt: .

Wenn: , diese Bedingung ist nicht nötig für die Korrektheit des Verfahrens

Wenn:

Wenn:

Wenn:

Wir definieren:

Wichtig sind die Exponenten und , denn für diese gilt: , und sind die kleinsten Zahlen für welche dies gilt. Es gibt immer ein und weil es nur Werte für gibt. Wir definieren:

Im Folgenden werde ich zeigen, dass wenn gilt, auch gilt.

Es gilt: , wir können diese Identität so oft wie wir wollen anwenden, deshalb gilt auch:

Wie schon erwähnt wird für die Berechnung von der Wert von () benötigt, daher die Rekursion. Deshalb sind die Folgenden Definitionen nötig: , ,

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 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

(c6, cpp73) und (c7, cpp79) sind trivial und werden hier nicht besprochen. Bei (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 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 « smaler or equal int». Es gilt: wenn: . In den folgenden Vergleichen kann das Verfahren nachvolzogen werden.

Wenn keine rote Bedingung korrekt ist, gilt .

«m\_pow» (c11, cpp49) berechnet eine Base mit Exponenten. «m\_sn» (c12, cpp59) berechnet 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 überprüfe habe ich anstelle von gewählt (c13, cpp87), um sicher zu gehen. Dieser Teil ist ein schwaches Glied im Programm, dass gebe ich offen zu.

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

Wir sprechen hier von Linearität bezogen auf den Teiler resp. . Die gesamte Laufzeit ist Linear wenn alle Teile linear sind, ist in allen Teilen, ausgenommen der Bestimmung von , unproblematisch. Kurz vorab, wenn gefunden ist beginnen durch alle Potenzen zu iterieren bis wir finden, dies ist linear (c14, cpp116). Auch bei der Bestimmung von iterieren wir durch die Potenzen. Bei suchen wir nicht nach einem bestimmten , wir müssen also speichern welche bereits vorkamen. Ganz wichtig: das Überprüfen ob ein 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 (c15, cpp91). Der Wert an der Speicheradresse speichert den momentanen Status von (c16, cpp98).

2.3 von zu

In der «m\_mod» Methode ist ein Stück Code für die Bestimmung von gegeben vorgeshen (c17, cpp135).

In der Abbildung oben sehen wir die Folge aller Natürlichen Zahlen von bis , liegt irgendwo in dieser Folge. Wir können zwei weitere Folgen aus Natürlichen Zahlen an die bereits existierende anheften. Von gegen werden Zahlen von abwärts angeheftet. Von gegen werden Zahlen von 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 auf . Wenn der Abstand grösser gleich ist addieren wir ihn zu , weil in diesem Fall grösser gleich ist und somit in der Angehefteten Folge mit am Anfang liegt (c18, cpp137). Anderwärtig wird die Diverenz auf 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 (c21, cpp11). Beim ersten Bit währe dies . Wir erhöhen durch quadrieren, (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 . Im Programm multiplizieren wir soche Datentype miteinader, damit der wir keinen Speicher Overflow riskiren könne wir nur mit Zahlen kleiner als die Wurzel rechen. . Ein guter Richtwert währe für alle und für Zahlen unter einer Miliarde zu nehmen.