kleiner kommunikation
max@kleiner.com

//Exklusiv für Entwickler

1 XXL Rechnen mit Grossen Zahlen

Dieser Bericht soll Magiern, Mathematiker und Informatiker zugleich dienen. Auch vor der Code Welt macht das Rechnen mit großen Zahlen, nebst der Finanzwelt, nicht halt. Mit externen Bibliotheken verschafft man sich Zugang zur Kalkulation großer Zahlen und viele System wie Kryptologie, Astrologie oder Umweltsimulation nutzen vermehrt große und in der Regel auch ganze Zahlen. Lassen Sie sich im Folgenden vom Umfang der schieren Unendlichkeit überzeugen und die hohe Flexibilität der Algorithmen zeigen.

1.1 Sesam öffne dich

In der Physik und Astronomie lassen sich ja zur Darstellung sehr großer und sehr kleiner Einheiten exponentielle Zahlen verwenden. Nehmen wir mal die Zahl 10^24, also eine 1 mit 24 Nullen, die schreibt man wie folgt: 1 000 000 000 000 000 000 000 000.

Die Schreibweise exponentieller Zahlen, also 10^24, erscheint uns jedoch wesentlich kompakter und flexibler. Eine solche Zahl die noch einen Namen hat ist bspw. die Dezilliarde, d.h. 10^63 mit dem Kurzzeichen L für Luma. So nebenbei, wie spricht man unsere Zahl aus? Ja 10^24 ist eine Quadrillion mit dem Kürzel Y für Yotta also nach Giga^9, Tera^12, Beta^15, Exa^18 und Zetta^21 folgt Yotta mit 10^24. Wenn die Inflation der Superlative so weitergeht wird man in 50 Jahren mal den Ausdruck Yottageil hören.;)

Aufgepasst im Englischen ist 1 Billion aber 10^9 und nicht eine Milliarde und so fort. Diese Tatsache wird gelegentlich bei Übersetzungen betreffend Staatsschulden oder einer Entfernung in Lichtjahren vernachlässigt.

Im Folgenden lässt sich jeder Schritt mit einem freien Tool (es muss nicht immer MATLAB oder maple sein) nachvollziehen, maXbox genannt, die man unter:

http://sourceforge.net/projects/maxbox/ kostenlos beziehen kann. Wir arbeiten gleich mit dem Skript 161_bigint_class_tester.txt und der 200: 200_big_numbers.txt, die sich im Unterverzeichnis /examples befinden. So lässt sich jeder Schritt gleich nachvollziehen und ausprobieren. Das Tool spricht die Sprache Object Pascal oder C und lässt sich nach dem Auspacken des Archivs ohne Installation oder einer Administration gleich starten. Das Skript kann man mit laden, kopieren oder reinziehen mit F9 einfach kompilieren und schon hat uns die große Zahl im Bann der trauten Unendlichkeit.

Bevor wir kurz zur Typenlehre kommen, werde ich gleich die Frage der Fragen beantworten, was den nun die größte Zahl im System ist. Da hat man sich in der Fachwelt gütig darauf geeinigt, eine Konstante zu definieren:

```
Infinity= 1.0 / 0.0; // Unit Math
(*$EXTERNALSYM Infinity*)
```

Unendlich, in bestimmten Rechensystemen der Kehrwert von 0, ist größer als alle Zahlen dieser Liste und ist selbst keine Zahl. Mit ∞ oder eben Infinity lässt sich zwar in beschränktem Umfang rechnen, jedoch sind viele Ausdrücke, die ∞ enthalten, entweder selbst ∞ oder nicht definiert.Die Idee das 1 durch 0 praktisch unendlich ist, erscheint genial. Mit dieser Konstante kann man tatsächlich rechnen wie oben behauptet, etwa um zu beweisen, dass eine sehr grosse Zahl minus eine kleine Zahl immer noch faktisch unendlich ist! Als Ergebnis erscheint schlicht und ergreifend `+Inf` also eine positive Unendlichkeit, das ging ja nochmals gut.

```
  float1:= Infinity;
  float2:= 23;
  Writeln('float1 - float2 = '+FloatToStr(float1 - float2));
```

Nun das Speichern von gigantischen Zahlen lässt sich selten über diese definierte Konstante lösen. Beispielsweise will man die Fakultät von 70 oder 100 wissen. Der Taschenrechner zeigt mit n! die Zahl

Fact(70) = 1.1978571669969891796072783721689e+100 an, also eine Zahl mit 100 Stellen.

Diese Funktion selbst ist in Delphi auf den seit der Version 4 vorhandenen Int64 Typ beschränkt. Int64 hat einen Wertebereich von `-2^63..2^63 - 1` eben 64 Bit mit Vorzeichen. Da ist etwa bei `FactInt(25)` Schluss mit der Darstellung, wobei man klar die interne Berechung gegenüber der externen Darstellung (Repräsentation) unterscheiden muss.

Generell gilt, dass arithmetische Operationen mit Integer-Werten einen Wert des Typs Integer zurückliefern, der in der aktuellen Implementation mit dem 32-Bit-Longint identisch ist. Operationen können nur dann einen Wert vom Typ Int64 verarbeiten, wenn sie für einen oder mehrere Int64-Operanden vorgesehen sind. Aus diesem Grund erfordert es einen `IntToStr64` mit einer entsprechende Parametrisierung vorzunehmen:

```
Function IntToStr64(i: Int64): String;
```

Es gibt auch Standardroutinen mit Integer-Argumenten, die verkürzen Int64-Werte auf 32 Bit. Nun mit dem 64-Bit Delphi sieht die Technik anders aus. In Java gibt es den BigDecimal der 2.0 *einen Long.MAX_VALUE darstellt, d.h. ein vorzeichenloser Int64 im Gleitkommaformat ist.

Weiter kommt man in einer 32- oder 64-bit Umgebung mit dem Typ `Extended`, wobei die Genauigkeit ja umgekehrt proportional mit der Zahlengrösse im Verhältnis Mantisse (Nachkommastelle) und Exponent abnimmt. Also entweder gross oder genau! Wie sagte schone der Buchhalter: Lieber ungefähr richtig als präzise falsch. Bei Fact(70) erhalten wir nur noch 1.2E+0100

```
  Writeln(FloatToStr(Fact(70)))
```

Stimmt nicht ganz, denn mit der Funktion `Format` erhalten wir mit 1.19785716699698918E100 eine genauere Darstellung, die bei der einfachen Konvertierung auf der Strecke bleibt:

```
  Writeln(Format('with Format %f',[Fact(70)]));
```

Mit `Extended` hat man aber eine Bruchzahl die sich z.B. mit `Mod` als Ganzzahloperator nicht mehr verarbeiten lässt. Zudem ist der Typ nicht so einfach portierbar wie die anderen reellen Typen. Verwenden Sie `Extended` mit Bedacht, wenn Sie Datendateien anlegen, die gemeinsam und plattformübergreifend genutzt werden sollen. Mit der nun folgenden Big Integer Bibliothek erhält man den vollen Bereich von 70! als genaue Ganzzahl:

11978571669969891796072783721689098736458938142546425857555362864628009582789845319680000000000000000

Wie aber erreichen wir diese Größenordnung? Nun kann ich z.B. 1000*2000 rechnen, indem ich 1000 mal 2000 addiere. Dabei werd ich aber bestimmt ineffizient bleiben (oder zumindest der Algorithmus...). In diesem Fall wäre 10 mal 10*2000 zu rechnen, wobei eine Faktorisierung bekannte Zahlen voraussetzt. Man überlegt sich auch andere Zahlensysteme (z.B. ist eine Binäre Zahl ja durch eine einfache Verschiebung um eine Stelle nach links blitzschnell mit 2 multipliziert. Oder eine Hex-Zahl mal 16 oder die Zahl ganz als String zu speichern, etc...)

Unser erster Ansatz sieht bei einer Addition so aus:

```
Type
  TMyDatentyp = array of byte;
var
  Zahl1, Zahl2, Ergebnis: TMyDatentyp;

function Addition(aZahl1, aZahl2: TMyDatentyp): TMyDatentyp;
var
```

```
  i: Integer;
begin
  SetLength(result, 1000);
  for i:= 0 to high(aZahl1) do
    result[i]:= aZahl1[i] + aZahl2[i];
end;

begin
  SetLength(Zahl1, 1000;
  SetLength(Zahl2, 1000);
  Ergebnis:= Addition(Zahl1, Zahl2);
end;
```

Wir operieren mit einem dynamischen Array mit `SetLength` und addieren wie die Schulmethode Zahl für Zahl, jedoch hier noch ohne Übertrag und Formatierung, so dass ein Ergebnis dann einzeln wieder ausgelesen wird. Auch das Einlesen erfolgt atomar, will heißen, Zahl um Zahl:

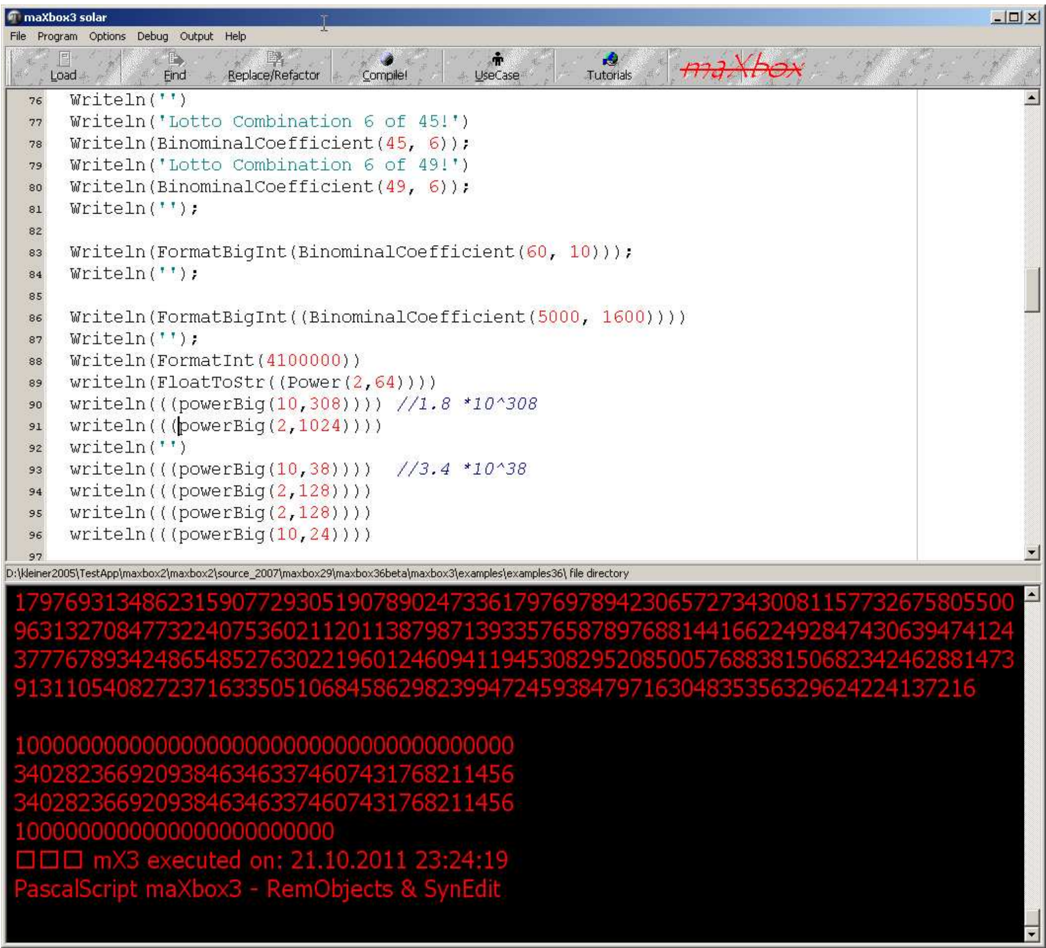
```
  zahl1[0]:=9; //952
  zahl1[1]:=5;
  zahl1[2]:=2;
  zahl2[0]:=9; //943
  zahl2[1]:=4;
  zahl2[2]:=3;
  //without carry bit
  Ergebnis:= Addition(Zahl1, Zahl2);
  for i:= 0 to 100 do
     Write(intToStr(Ergebnis[i]))
```

Konkret addieren wir 952 plus 943. Das ist sicher noch kein XXL Ergebnis soll aber das Kalkulationsschema zeigen, welches mit der Dimensionierung von `SetLength` eine 1000stellige Zahl verarbeiten kann. Um ein offenes, dynamisches Array im Speicher anzulegen, ruft man `SetLength` auf. Umständlich, jedoch als Prinzip verständlich, ist das Abfüllen der Zahl, das mit einem Zahlstring in unserem Objekt als nächstes vermieden wird. Betrachten wir mal die Anforderungen.

Unser Objekt Prototyp basiert ja auf der Idee, nach der Schulmethode zu rechnen, d.h. Zahl für Zahl in einem Array jeweils mit dem Übertrag als Verschiebung zu verarbeiten.

Die Zahlen werden jedoch schnell recht groß, sodass die Integer Datentypen eh schnell hinfällig sind. In Java gibt es bspw. die BIGINT-Bibliothek, die große Zahlen (200-300 Stellen) verarbeitet, und dabei die Operation Potenzieren bzgl. Modulo Operator beherrscht (RSA). In einem Cryptosystem wie RSA ist a^b `mod` n die zentrale Funktion neben der Verwaltung von 200-300 Stellen. `BigInt` ist eine durch die Bibliothek vordefinierte Klasse (dahinter verbirgt sich `java.math.BigInteger`), deren Elemente sich wie (beliebig genaue) ganze Zahlen verhalten. Also meistens gibt es zwei Ansätze, erstens die Zahl als String zu betrachten und dann mit der Ord-Funktion zu wandeln oder eben mit einem Array of Byte direkt oder binär zu operieren.

Die folgende Klasse wurde mit dem Gedanken entwickelt, dass ein laufendes System oder Skript schnell und leicht anzupassen sei und in der Ausbildung wie der Forschung nützlich ist. Die Klasse hat ein durchgängiges try finally Konstrukt, das sich immer wieder als Template benützen lässt und soll verständlich und leicht erweiterbar sein. Kleinere logische oder optische Anpassungen lassen sich zum Teil sogar während des laufenden Skriptes ändern, d.h. es ist nicht nötig, dass man neu kompiliert. Nun das ist der Vorteil eines Stack basierten Interpreters wie die maXbox eben ist.



// bigint_maxbox.png

Abb. 1: In der Box ersichtlich ist die Berechnung von 2^1024 beim Cursor

Meistens braucht man für ein Programm nur einen extrem großen Datentypen für Ganzzahlen. Hintergrund kann sein: Ich will große Primzahlen generieren können. Dazu brauch ich aber einen größeren Datentyp, als `Integer`, `LongInt` oder `Extended` ist. Genauer gesagt handelt es sich bei meiner Anforderung um eine Zahl, die so ca. 10000 - 1 Mio. Stellen hat. Zudem sollte der Datentyp irgendwie komprimiert im Arbeitsspeicher vorliegen (also nicht einfach als String, das währe eine Verschwendung). Nun in unserem Fall reicht die Zahlengröße so gegen 30000 Stellen auch ohne Komprimierung aus. Die Prinzip-Klasse hat die folgende Struktur:

```
TMyBigInt = class
   private
     Len: Integer;
     Value: AnsiString;
     procedure Trim;
     procedure Shift(k: Integer);
     procedure MultiplyAtom(Multiplier1: TMyBigInt; Multiplier2: Integer);
   public
     constructor Create(iValue: Integer = 0);
     procedure Add(Addend1, Addend2: TMyBigInt);
     procedure Multiply(Multiplier1, Multiplier2: TMyBigInt); overload;
     procedure Multiply(Multiplier1: TMyBigInt; Multiplier2: Integer); overload;
     function ToString: string;
     procedure CopyFrom(mbCopy: TMyBigInt);
   end;
```

Wie arbeiten mit einem Zahlstring als `AnsiString`, der für den Nutzer gegenüber einem Array of Zahl einfacher zu handhaben ist, zudem liefert die Klasse den zugehörigen Konverter `ToString` gleich mit. `Shift` hat die Aufgabe den Zahlstring von hinten mit k Nullen aufzufüllen, also mit Base^k zu multiplizieren. Base ist als Konstante mit 10 definiert. Mit `Trim` lassen sich die führenden Nullen entfernen. Die Länge der Wert-Strings wird angeglichen und die Berechnung von Übertrag und Summe sind natürlich mit von der Partie. Wie berechne ich nun 70!? (Witzig: Ausrufe- und Fragezeichen friedlich vereint)

```
function GetBigIntFact: string;
var mbResult: TMyBigInt;
     i: integer;
begin
  mbResult:= TMyBigInt.Create(1);
  try
    for i:= 1 to 70 do
      mbResult.Multiply1(mbResult, i);
    Result:= mbResult.ToString;
  finally
    mbResult.Free;
  end;
end;
```

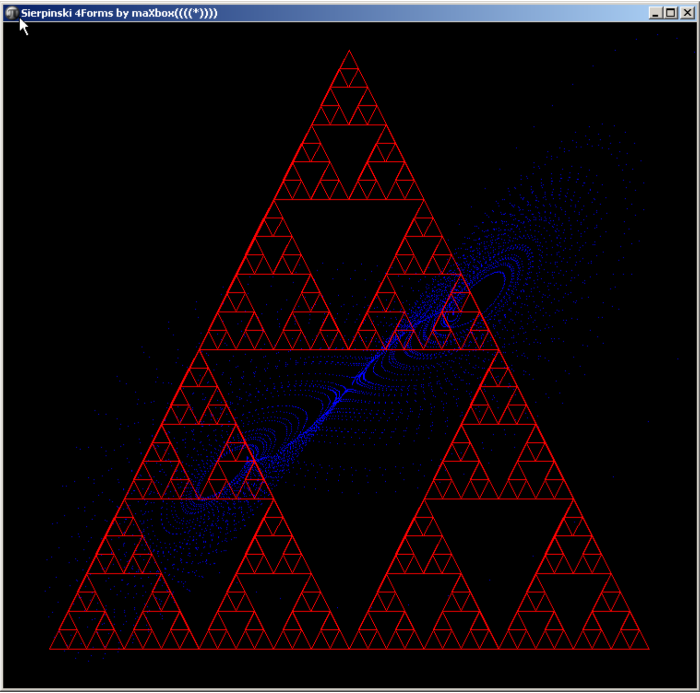
Nach dem obligaten und lokalen Konstruktor lassen sich die Faktoren aufmultiplizieren und als String zurückwandeln. Eigentlich wandelt die Methode `ToString` das Zahlstring-Array ein einen qualifizierten String zurück. So vermeiden wir, nie einen beschränkten Typen gebrauchen zu müssen. Die Klasse ist noch nicht vollständig, da z.B. die Division fehlt.

Die Multiplikation ist immer eine Multiplikation mit einer Ziffer, die aber bei einer Division mit möglichen Gleitkommazahlen nicht mehr (oder mit Aufwand) garantiert ist. Wichtig auch mit `Free` den Speicher sofort wieder freizugeben, da die Zahlen beträchtlich und mächtig aufsummieren. Anbei noch die Hilfsfunktion `ToString`, die als Kernmethode das Prinzip verdeutlicht:

```
function TMyBigInt.ToString: string;
var i: Integer;
begin
  Trim;
  Result:= '';
  for i:= Len downto 1 do Result:= Result + IntToStr(Ord(Value[i]));
end;
```

Anhand dieser Funktion läßt sich kurz exemplarisch zeigen, wie `MyBigInt` funktioniert:

1. Die „Zahlen“ als Ansi Wert speichern (Value).
2. Von hinten den Wert in der Länge nach iterieren.
3. Mit Ord bspw. die 0 auf ASCII Wert 48 setzen usw.
4. Den Wert kumulieren und als String zurückgeben.
5. Evtl. noch Formatieren oder Trimen (als `FormatBigInt`)



// pascaltriangle.png

Abb. 2: Sierpinski als Grosse Dimension

Eine weitere Möglichkeit bietet die Welt der Unendlichen Reihen, wie Abb. 2 bei einer Sierpinski Figur zeigt. Das eigentliche Sierpinski-Dreieck im streng mathematischen Sinn ist das Grenzobjekt, das nach unendlich vielen Iterationsschritten übrig bleibt. Jeder Laufindex ist ja letzten Endes wieder auf einen Typ beschränkt. Mit dem Sierpinski-Dreieck verwandt ist das Pascalsche Dreieck. Dabei entsprechen die geraden Zahlen im Pascal-Dreieck den Lücken im Sierpinski-Dreieck! Diese Skripte sind unter 239_pas_sierpinski.txt und 172_pascal_triangleform2.txt verfügbar.

1.2 Think Big in der Bibliothek

In der realen Welt der großen Zahlen gibt es einige Theorien wobei ich eine kurz erwähnen möchte.

Die Stirling-Formel ist eine mathematische Formel, mit der man für große Fakultäten Näherungswerte berechnen kann. Sie ist benannt nach dem Mathematiker James Stirling.

Die Stirling-Formel findet überall dort Verwendung, wo die exakten Werte einer Fakultät nicht von Bedeutung sind. Insbesondere bei der Berechnung der Information einer Nachricht und bei der Berechnung der Entropie eines statistischen Ensembles von Subsystemen ergeben sich mit der Stirling-Formel starke Vereinfachungen.

Auf dem freien Markt gibt es einige Bibliotheken die auf der einen oder anderen Theorie beruhen. Wichtig erscheint mir vor allem, dass man versteht was das Ding tut. Denn bei diesen Grössen-ordungen hat man wenig Erfahrung und noch weniger Referenzmöglichkeiten um die Resultate zu verifizieren. Diese variieren in Umfang von Anpassung und Erweiterung von optischen Komponenten oder Umbenennen von Feldern bis zur Neuentwicklung von Modulen.

Auch Zusatzfunktionen wie ein Compare, StrToBigNum oder ein Log müssen jeweils ausgebaut werden wenn die Methode nicht schon im Modul vorhanden ist. Man ist auch nicht vor Überraschungen gefeit, da ein ganzzahliger 64 Bit-Wert auf einmal nun als BIGINT-Typ und nicht mehr als BCD Typ daherkommt.

Auch haben große Zahlen ihre natürlichen Grenzen in der Speicherfähigkeit. Man kann zwar theoretisch mit fast beliebig großen Zahlen rechnen, aber z.B. 2357569^15847657 ist dann doch eine Nummer zu groß! Die Zahl hat gegen 10^14 Nullen. Wenn die Zahl als String in einer Datei vorliegt, wären die mehrere hundert Terabyte groß, die auf keine derzeit erhältliche Festplatte passt. Und der Rechenaufwand ging wohl in die Lichtjahre.

So eine 10000! benötigt in der maXbox rund eine halbe Minute und hat genau 35660 Stellen, ergibt etwa 10 A4 Seiten und lässt sich auch als Exponent formatieren: 2.846259680917054518906413212119e+35659. Interessant ist auch mal auszurechnen, wie lang denn eine simple Int64 als Zeit in Sekunden ausreicht. 9223372036854775807 / 1000 = 9223372036854775,807 Sekunden = fast 292.471.208 Jahre. Oder 292471208677.53 /1000.

Jedenfalls unsere Anfangszahl 10^24 alias eine Quadrillion ist schlussendlich so realisiert:

```
function PowerBig(aval,n: integer): string;
var mbResult: TMyBigInt;
     i,z: integer;
begin
  mbResult:= TMyBigInt.Create(1);
  try
    for i:= 1 to n do
      mbResult.Multiply(mbresult, aval);
    Result:= mbResult.ToString;
  finally
    mbResult.Free;
  end;
end;
```

Für Delphi Entwickler interessant sind die Bibliotheken VLI, MPArith und BigInt. Gute Erfahrung haben wir auch mit der LockBox 3 von Turbo Power gemacht, die eigentlich eine Crypto Lib ist (AES, DES, 3DES, Blowfish, Twofish, SHA, MD5, RSA Digitale Signatur...) aber deshalb auch eine integrierte BigInt Klasse hat! Für den Entwickler und den weiteren Ausbau, hat die Open Source Bibliothek einiges zu bieten, im Anhang ist der Link dazu ersichtlich. Viel Spaß noch mit Think Big.

Max Kleiner

Literatur:

H.T. Lau, “A Numerical Library in C for Scientists and Engineers", CRC Press, 1994, Ch. 6.

Earl F. Glynn (www.efg2.com) and Ray Lischner (www.tempest-sw.com) for math unit

Links:

http://sourceforge.net/projects/maxbox/

http://sourceforge.net/projects/tplockbox/

http://sourceforge.net/projects/bigint-dl/files/