DD2458, Problemlösning och programmering under press

Föreläsning 8: Aritmetik I

Datum: 2009-11-03
Skribent(er): Andreas Sehr, Carl Bring, Per Almquist
Föreläsare: Fredrik Niemelä

1 Flyttal

Att representera heltal med basen två har inte varit något problem. Precisionen är god och uträkningar exakta i för plus, minus och gånger. Men så fort datorn ska beräkna division eller representera ett reellt tal med ett flyttal så blir det problem med precisionen. Detta på grund av flyttalens uppbyggnad. Ett flyttal går att representera på andra sätt, exempelvis två heltal i ett bråk, två heltal som anger heltalsdel samt decimaldel, eller ett tal som har en fixed-point som då har uppdelat i sig hur många bitar som ska representera heltal respektive decimaler.
Det moderna sättet att bygga upp flyttal i persondatorer följer en standard som första gången togs fram i mitten på 80-talet, IEEE 754. Denna standard använder sig av floating-point format på flyttalet. Detta innebär att beroende på heltalsdelens storlek i bitar blir det olika många bitar över för att representera decimaldelen. Det man gör egentligen är att ta ett tal skrivet på sättet 1,234567 och sedan flytta decimalen till den punkt i talen som är korrekt, ex 1234,567.

1.1 IEEE 754 standarden

IEEE 754 standarden som beskriver hur flyttal ska sparas består i huvudsak av tre delar, teckenbit, exponent och en mantissa. Standarden kom ut först 1985 men har nyligen genomgått en uppdatering som blev klar 2008. Exponentens bas är i datorn som standard 2 och behöver därför inte sparas.
Följande tabell beskriver hur flyttal är uppbyggda med enkel samt dubbel precision:

	Teckenbit (S)	Exponent (E)	Mantissa (M)	Totalt (bitar)
`float`	1	8	24	33
`double`	1	11	53	65
`long double`	1	15	113	129

Uträkningen av det reella talet i en `double` sker enligt formeln $S * (1 + M * 2^{-52}) * 2^{E-1023}$. Teckenbiten indikerar om värdet är positivt eller negativt, 0 betyder positivt och 1 betyder negativt. Exponent bitarna ska kunna lagra både positiva och negtiva tal och därför har man ett värde som dras bort från exponentvärdet, för enkel precision är detta värde 127 och för dubbel precision är det 1023. Formeln för att få fram detta normaliseringsvärde är $(2^{\text{bitar i exponent}-1} - 1)$. Exponentvärdet är det värde som flyttar decimaltecknet. Mantissan, även känd som signifikant, representerar precisions bitarna i talet. Mantissan är egentligen en bit längre, den mest signifikanta biten kan man få genom att kolla på exponenten och behöver därför

inte sparas. Om det normaliserade exponentvärdet ligger mellan 0 och $2^e - 1$ så är den mest signifikanta biten 1 och värdet är normaliserat, annars är den 0 och värdet antas vara denormaliserat.
Beroende på hur bitarna står i exponenten och mantissan kan flyttal även lagra Inf(oändlighet, ∞) och NaN (Not A Number).

Exponent	Mantissa	Typ
0	0	0
0	!= 0	De-normaliserad
$1 till 2^e - 2$	oberoende	Normaliserad
$2^e - 1$	0	∞
$2^e - 1$	Skilt från noll	NaN

Dessa värden har speciella egenskaper som kommer fram när man räknar med dem. Det går att räkna med oändligheten i exempelvis min och maxjämförelser. I den senaste standarden(754-2008) så har man lagt till NaN i max och min-jämförelser så att det alltid returnerar det tal som är giltigt.
Det går även att räkna med dessa värden men de kommer bete sig enligt följande tabell:

Operation	Resultat
∞ + x	∞
∞ * -x	-∞
∞ - Inf	NaN
∞ * 0	NaN

1.2 Fallgropar med flyttal

Att försöka räkna med flyttal som om det vore heltal är inte att rekommendera, detta eftersom koden då inte kommer bete sig som den intuitivt säger.
Betänk följande kodsnuttar i C++:

```
double x = 0.29;
int y = (int)(100*x);
cout << y;
```

Genom att bara titta på koden så misstänker vi att 29 kommer skrivas ut men på grund av avrundningsfel som uppstår mellan `double` och `int` så kommer 28 att skrivas ut.
En enkel loop som ökar värdet med 0.1 varje iteration istället skulle inte vara så konstigt. Ex vill vi ha alla tiondelsprocent av ett värde.

```
for(double x = 0.0; x != 1.0; x += 0.1)
cout << x << endl;
```

Denna kod kommer inte att terminera efter 10 iterationer som man först kan tro. Programmet kommer att fastna i loopen eftersom 0.1 inte går att representera i flyttal exakt. Det kommer hela tiden bli ett avrundningsfel och när jämförelsen med 1.0 väl sker kommer x vara extremt lite mindre än 1.0 och därför blir jämförelsen fel. Detta gör att även följande kod kommer bli fel:

```
for(double x = 0.0; x < 1.0; x += 0.1)
cout << x << endl;
```

Loopen kommer köra 11 gånger eftersom den vid det 10 varvet fortfarande är mindre än 1.0.
För att korrigera detta kan man lägga till en avrundningsneutraliserare kallad epsilon:

```
for(double x = 0.0; x < 1.0 - e; x += 0.1)
cout << x << endl;
```

1.3 Epsilon ϵ

Att jämföra flyttal kan vara knepigt på grund av avrundningsfel. För att motverka detta kan man använda Epsilon vilket motsvarar den övre felgränsen för en flyttalstyp. Det finns två metoder man kan använda sig av för jämförelse av flyttal:

Absolutfel

Vi definerar a och b som lika i absolut bemärkelse om:

$$a \overset{abs}{=} b \text{ om } |a - b| < \epsilon$$

och a är mindre än b i absolut bemärkelse om:

$$a \overset{abs}{<} b \text{ om } a \overset{abs}{\neq} b \text{ och } a < b$$

Relativfel

Vi definerar a och b lika i relativ bemärkelse om:

$$a \overset{rel}{=} b \text{ om } |a - b| < \epsilon * (|a| + |b|)$$

och a är mindre än b i relativ bemärkelse om:

$$a \overset{rel}{<} b \text{ om } a \overset{rel}{\neq} b \text{ och } a < b$$

Absolutfel fungerar dåligt vid stora flyttal och motsvarande så fungerar relativfel dåligt vid små flyttal. Säkrast är då att kombinera de olika metoderna:

$$a \overset{komb}{=} b \text{ om } a \overset{abs}{=} b \text{ eller } a \overset{rel}{=} b$$

$$a \overset{komb}{<} b \text{ om } a \overset{komb}{\neq} b \text{ och } a < b$$

Algoritm 1: epsilon-approx
Input: An floating point type
Output: An approximation of epsilon
EPSILON-APPROX(float)
(1) $float = 1.0$
(2) **do**
(3) $float = float/2.0$
(4) **while**$((1.0 + (float/2.0)) \neq 1.0)$
(5) **return** $float$

Att bestämma ϵ

Det går att uppskatta ϵ för en flyttalstyp enkelt med hjälp av linjärsökning enligt algoritmen epsilon-approx [1].

Det finns även förberäknade värden för ϵ i C++ om man tittar i biblioteket <limits> [2]. Exempel:
$numeric\_limits < float >:: epsilon() = 1.19209 * 10^{-7}$
$numeric\_limits < double >:: epsilon() = 2.22045 * 10^{-16}$

1.4 Exempel: analysera precisionen för $\sqrt[4]{x}$ givet att detta är ett heltal

Låt x vara en `double` och låt $y = pow(X, 0.25)$ avrundat till 0 decimaler. Låt även $X \approx x * (1 \pm \epsilon)$ vara vår approximation av x.

$$\begin{aligned} Y &= pow(X, 0.25) \\ &= (1 \pm \epsilon) * X^{\frac{1}{4}} \\ &= (1 \pm \epsilon) * e^{\frac{ln(X)}{4}} \\ &\approx (1 \pm \epsilon) * e^{\frac{ln(x)}{4} * (1 \pm \epsilon)} \\ &= (1 \pm \epsilon) * y * e^{\pm \epsilon * ln(y)} \\ &\approx (1 \pm \epsilon) * (1 \pm \epsilon * ln(y)) * y \\ &\approx y \pm y * \epsilon * (1 \pm ln(y)) \end{aligned}$$

I tredje sista steget så utnyttjade vi likheten $\frac{ln(x)}{4} = ln(y)$ och i det följande steget utnyttjade vi att $e^{\pm\delta} \approx 1 \pm \delta$ då δ är litet. Vi får slutligen ett fel i vår beräkning $|Y - y| \approx y * \epsilon * (1 + ln(y)) \approx 2 * 10^{-4} \ll 0.5$, vilket kommer ge korrekt svar efter avrundning.

[1] http://en.wikipedia.org/wiki/Machine_epsilon
[2] http://www.cplusplus.com/reference/std/limits/numeric_limits/

2 Floor och Ceil

Givet $x \in \mathbb{R}$ defineras följande: $\lfloor x \rfloor = floor(x)$ som det största heltal mindre än eller lika med x. $\lceil x \rceil = ceil(x)$ som det minsta heltal större än eller lika med x.

x	$\lfloor x \rfloor$	$\lceil x \rceil$
5.3	5	6
5.0	5	5
0.7	0	1
0.0	0	0
-0.7	-1	0
-5.0	-5	-5
-5.3	-6	-5

Exempel på $floor(x)$ och $ceil(x)$.

2.1 Operationer

Givet $x \in \mathbb{R}$ och $n, m \in \mathbb{Z}$ gäller följande:

$$
\begin{array}{ccc}
\lfloor x \rfloor = n & \iff & n \leq x < n+1 \\
\lceil x \rceil = n & \iff & n-1 < x \leq n \\
-\lfloor x \rfloor & = & \lceil -x \rceil \\
-\lceil x \rceil & = & \lfloor -x \rfloor \\
\lfloor x \pm n \rfloor & = & \lfloor x \rfloor \pm n \\
\lceil x \pm n \rceil & = & \lceil x \rceil \pm n \\
\lceil \frac{n}{m} \rceil & = & \lfloor \frac{n+m-1}{m} \rfloor \text{ för } m > 0
\end{array}
$$

2.2 Beräkning av $\lfloor \frac{n}{m} \rfloor$ och $\lceil \frac{n}{m} \rceil$

Den naiva lösningen vore att konvertera till `double` och sedan använda $floor(x)$ och $ceil(x)$. Detta fungerar dåligt då n och m är stora eftersom man då riskerar att få avrundningsfel. Det går att göra dessa beräkningar på ett mer precist sätt. Tanken är att omvandla alla fall till $\lfloor \frac{n}{m} \rfloor$ där $n, m > 0$, vilket motsvaras av heltalsdivision.

Algoritm 2: floor-fraction
Input: A fraction between $n, m \in \mathbb{Z}$
Output: An integer
FLOOR-FRACTION(numerator n, denominator m)
(1) **if** $m < 0$
(2) **return** FLOOR-FRACTION$(-n, -m)$
(3) **else if** $n < 0$
(4) **return** $-$CEIL-FRACTION$(-n, m)$
(5) **else**
(6) **return** n/m

Algoritm 3: ceil-fraction
Input: A fraction between $n, m \in \mathbb{Z}$
Output: An integer
CEIL-FRACTION(numerator n, denominator m)
(1) **if** $m > 0$
(2) **return** FLOOR-FRACTION$(n + m - 1, m)$
(3) **else**
(4) FLOOR-FRACTION$(-n - m - 1, -m)$

2.3 Exempel: Rare Easy Problem

Kattisproblemet Rare Easy Problem[3] går ut på att hitta N, givet att $K = N - M$ och att M är N med sista siffran avhuggen. Vi observerar att $M = \lfloor \frac{N}{10} \rfloor$. Så vad vi vill lösa är $N - \lfloor \frac{N}{10} \rfloor = K$.

$$\begin{array}{rcl} K & = & N - \lfloor \frac{N}{10} \rfloor \\ & = & N + \lceil -\frac{N}{10} \rceil \\ & = & \lceil \frac{9N}{10} \rceil \\ K - 1 & < & \frac{9N}{10} \leq K \\ 10(K-1) & < & 9N \leq 10K \\ 10(K-1) + 1 & \leq & 9N \leq 10K \\ \frac{10K}{9} - 1 & \leq & N \leq \frac{10K}{9} \\ \lceil \frac{10K}{9} \rceil - 1 & \leq & N \leq \lfloor \frac{10K}{9} \rfloor \end{array}$$

Om K är delbart med 9 så får vi två lösningar $\frac{10K}{9}$ och $\frac{10K}{9} - 1$.
Om K inte är delbart med 9 får vi endast en lösning $\lfloor \frac{10K}{9} \rfloor$.

3 Godtyckligt stora heltal

De flesta programmeringsspråk erbjuder stöd för heltal genom primitiva datatyper som `short`, `int` och `long` men dessa har ofta en fix övre gräns för hur stora heltal de kan representera. Begreppet godtyckligt stora heltal(Bignum)[4] innebär heltal som är såpass stora att de inte får plats i dessa primitiva datatyper. Bignums är ofta långsamma att använda eftersom operationer på dem utförs via högnivå mjukvara istället för lågnivå logik som kan utföras direkt på processorns ALU. Användning av Bignums bör således undvikas i görligaste mån. Om applikationen kräver representation av stora heltal men exakt precision inte är behövlig så räcker ofta flyttalen till som representation. Men ibland krävs Bignum och då är det bra att veta vad de innebär.

3.1 Historia

Stöd för Bignum i diverse programmeringsspråk har erbjudits sedan 1980-talet och det programmeringsspråk som sägs ha varit först heter Maclisp[5]. Numera erbjuds

[3] http://kattis.csc.kth.se/problem?id=easy
[4] http://en.wikipedia.org/wiki/Bignum
[5] http://maclisp.info/

stöd för Bignums i de flesta programmeringsspråk, antingen direkt i språket eller via tilläggsbibliotek, så uppfinn inte hjulet på nytt, använd det som finns att tillgå. I Java är stöd för Bignum inbyggt och åtkomst sker via klassen java.math.BigInteger[6]. I C/C++ finns en mängd tredjeparts bibliotek där GNU Multiple Precision Arithmetic Library (GMP)[7] är det mest använda. GMP är förövrigt skapat av Torbjörn Granlund[8] som forskar här på KTH.

3.2 Implementation

Om ni vill skapa en egen implementation för representation av Bignum så behöver ni lösa lagringsproblematiken. Talen som ska representeras är större än vad som ryms i de primitiva datatyperna så någon typ av uppdelning av talen behöver ske. Bryt upp talen i delar givet någon specifik bas och lagra dem sedan i de primitiva datatyperna via någon dynamisk datastruktur, eller mer formellt:

Definition 3.1 *Låt* $\mathbf{Z}$ *beteckna heltalen och* X *vara ett Bignum sådan att* $X \in \mathbf{Z}$. *Inför en bas* $b \in \mathbf{Z}$ *och låt* X *representeras i basen* b *på följande sätt.*

$$X = x_{n-1}b^{n-1} + x_{n-2}b^{n-2} + \cdots + x_2b^2 + x_1b + x_0$$

Varje x_i lagras då sedan i någon primitiv datatyp.
Att de bör lagras via någon dynamisk datastruktur (så som en string, eller en vector<primitiv datatyp>) har att göra med att antalet x_i kommer att förändras i och med att operationer utförs på X.

3.2.1 Vilken bas bör väljas?

Binär Att bryta upp X givet en binär bas av typ $b = 2^k$ medför mycket effektiv beräkningar. Datorn representerar redan tal i basen 2 så minnet utnyttjas väl och väljes k smart kan optimeringar i beräkningsalgoritmerna utföras. Då binär bas inte används särskilt mycket i verkliga livet innebär det däremot ofta att, ganska krångliga, konverteringar till och från bas 2 behöver göras ihop med in- och utmatning av data. Om tyngd ligger på beräkningar, och inte på in- utmatning av data, så är en binär bas bra.

Decimal Om X istället bryts upp givet en decimal bas av typ $b = 10$ så blir in- och utmatning av data enkel att hantera. Däremot utnyttjas datorns minne väldigt dåligt då det blir väldigt många x_i och detta gör även att beräkningar går långsamt. Ett sätt att minska på mängden minne som behövs för att lagra ett tal i decimal bas är att välja en bas av typ $b = 10^d$. Om d väljs så att $10^{2 \cdot d} = 2^k$ (där 2^k betecknar storleken på den primitivt underliggande datatypen) uppfylls så underlättas implementationen av vissa operationer på talet eftersom man i så fall inte behöver oroa sig för `overflow`.

[6] http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigInteger.html
[7] http://gmplib.org/
[8] http://www.nada.kth.se/ tege

3.2.2 Operationer

Att utföra operationer på Bignum sker som sagt till stor del genom mjukvara. De algoritmer som presenteras här är naiva och liknar de räknesätt som lärdes ut under matematik undervisningen på lågstadiet.
Att tänka på när man implementerar algoritmer för Bignum är det viktigt att hålla reda på hur storleken av det aktuella talet kan variera under algoritmens gång. Ibland kan mer minne behöva allokeras och ibland kan det frigöras (ex. vid division). Det finns mer sofistikerade algoritmer[9] (särskilt för multiplikation och division). Vilken bas som valts spelar in på hur effektiva algoritmerna är och vissa optimeringar går endast att genomföras givet en viss bas. Endast en algoritm för inmatning av data presenteras, utmatning utförs analogt.

Operation	Komplexitet
Addition	$\Theta(n)$
Division	$O(n^2)$
Exponentiering	$O(nM(n))$ där $M(n)$ betecknar komplexiteten för multiplikation
Jämförelse	$O(n)$
Inmatning	$\Theta(n)$
Multiplikation	$O(n^2)$
Subtraktion	$\Theta(n)$

Algoritm 4: Addition
Input: Two Bignums and their base
Output: A Bignum
ADD(Bignum a, Bignum b, base $base$)

```
(1)  n = max(number of digits in a, number of digits in b)
(2)  carry = 0
(3)  result = 0
(4)  for i = 0 to n − 1
(5)      tmp = a_i + b_i + carry
(6)      result_i = tmp mod base
(7)      carry = tmp/base
(8)  result_n = carry
(9)  return result
```

[9] http://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations#Arithmetic_functions

Algoritm 5: Division
Input: Two Bignums
Output: Two Bignums representing quotient and rest of the divison
DIV(Bignum *numerator*, Bignum *denominator*)

```
(1)   if denominator == 0
(2)     return error
(3)   counter = 0
(4)   while not numerator < denominator
(5)     denominator *= 2
(6)     ++counter
(7)   denominator /= 2
(8)   quotient = 0
(9)   rest = numerator
(10)  while counter not 0
(11)    quotient *= 2
(12)    if not rest < denominator
(13)      rest -= denominator
(14)      ++quotient
(15)    denominator /= 2
(16)    − − counter
(17)  return quotient, rest
```

Algoritm 6: Exponentiering
Input: A Bignum and an integer
Output: A Bignum
POW(Bignum x, Integer n)

```
(1)   if n < 0
(2)     x = 1/x
(3)     n = −n
(4)   i = n
(5)   y = 1
(6)   z = x
(7)   while not i == 0
(8)     if not i mod 2 == 0
(9)       y *= z
(10)    z *= z
(11)    i = FLOOR-FRACTION(i, 2)
(12)  return y
```

Algoritm 7: Jämförelse
Input: Two Bignums
Output:

$$\begin{cases} \text{-1} & \text{if } a < b \\ 0 & \text{if } a = b \\ 1 & \text{if } b < a \end{cases}$$

CMP(Bignum a, Bignum b)
(1) check sign of a and b
(2) **if** signs differ and a has minus sign
(3) **return** -1
(4) **else**
(5) **return** 1
(6) **if** both have plus sign
(7) **if** number of digits in $a <$ number of digits in b
(8) **return** -1
(9) **else**
(10) **return** 1
(11) **foreach** a_i and b_i in a and b
(12) **if** $a_i < b_i$
(13) **return** -1
(14) **else**
(15) **return** 1
(16) **else**
(17) the opposite of lines 7-15
(18) **return** 0

Algoritm 8: Inmatning av data
Input: A stream or string of characters and and a base
Output: A Bignum
IN(Stream or string X, base b)
(1) **while** X has characters left
(2) extract x_i
(3) store x_i in datatype representing b^i
(4) **if** X is signed with minus **then** negate datatype representing b^n

Algoritm 9: Multiplikation
Input: Two Bignums and their base
Output: A Bignum
MUL(Bignum a, Bignum b, base $base$)
(1) $|a|$ = number of digits in a
(2) $|b|$ = number of digits in b
(3) $|result| = |a| * |b|$
(4) **for** $i = 0$ **to** $|a|$
(5) **for** $j = 0$ **to** $|b|$
(6) $result_{i+j}$ += $a_i + b_j$
(7) $carry = 0$
(8) **for** $j = 0$ **to** $|result|$
(9) $result_j$ += $carry$
(10) $carry = result_j / base$
(11) $result_j = result_j$ mod $base$
(12) **return** $result$

Algoritm 10: Subtraktion
Input: Two Bignums and their base
Output: A Bignum
SUB(Bignum a, Bignum b, base $base$)
(1) **if** $a < b$
(2) **return** $-$SUB(b, a)
(3) $n = max$(number of digits in a, number of digits in b)
(4) $borrow = 0$
(5) $result = 0$
(6) **for** $i = 0$ **to** $n - 1$
(7) $tmp = a_i - b_i - borrow$
(8) $result_i = tmp$ mod $base$
(9) **if** $tmp < 0$
(10) $borrow = 1$
(11) **else**
(12) $borrow = 0$
(13) **return** $result$