DD2458, Problemlösning och programmering under press

Föreläsning 8: Aritmetik och stora heltal

Datum: 2007-11-06
Skribent(er): Martin Tittenberger, Patrik Lilja
Föreläsare: Per Austrin

Denna föreläsning handlade om beräkningar med flyttal och stora heltal.

1 Flyttalsaritmetik

Låt oss betrakta två exempel där flyttal inte beter sig som man tror.

Exempel 1: Avrundning

```
double x = 0.29;
int y = (int) (100 * x);
printf("%d", y);
```

Utdata från programmet ovan blir 28 istället för 29.

Exempel 2: Jämförelse

```
for (double x = 0.0; x != 1.0; x += 0.1)
    printf("Hello World\n");
```

Loopen kommer att köras oändligt många gånger eftersom termineringsvillkoret aldrig uppfylls.

1.1 Flyttalsrepresentation

Orsaken till att de märkliga fenomenen uppstår i exemplen ovan är att man i allmänhet inte kan representera reella tal exakt med ändligt mycket minne. I exemplen används primitiva datatyper vars precision är begränsad. Vi skall titta närmare på hur flyttal representeras i minnet.

1.1.1 IEEE Floating Point Representation

Ett flyttal x representeras på följande sätt enligt IEEE standarden: $x = S \cdot M \cdot 2^E$, där:

$$S = \text{Teckenbit} \in \{0, 1\}$$
$$M = \text{Mantissa (siffrorna i talet)}$$
$$E = \text{Exponent (storleken på talet)}$$

Typ	Teckenbit	Mantissa	Exponent	Storlek
float	1	24	8	32
double	1	53	11	64
long double[1]	1	113	15	128[2]

Tabell 1: Antal bitar för S, M och E för olika flyttalstyper.

1.1.2 Flyttalstyper

Som man kan notera i tabell 1 är summan av antalet bitar för de respektive typerna ett större än storleken (totala antalet bitar). Detta beror på att flyttal lagras i en normaliserad form där den mest signifikanta biten i mantissan alltid är satt. Därför behöver inte denna bit lagras. Om den mest signifikanta biten inte skulle vara satt så kan man istället representera talet som $S \cdot (2 \cdot M) \cdot 2^{E-1}$.

1.1.3 En närmare titt på double

Ett flyttal x av typen `double` representeras på följande sätt:

$$x = (-1)^S \cdot (1 + M \cdot 2^{-52}) \cdot 2^{E-1023}$$

där $S \in \{0, 1\}$, $0 \leq M < 2^{52}$, $1 \leq E < 2^{11} - 1$.

Värdena $E = 0$ och $E = 2^{11} - 1$ är reserverade för speciella tal. En `double` där $E = 0$ representerar talet:

$$(-1)^s \cdot (M \cdot 2^{-52}) \cdot 2^{-1023}$$

Notera att om $M = 0$ får vi precis 0. En `double` där $E = 2^{11} - 1$ kan representera oändligheten eller NaN[3], se tabell 2.

Exponent	Mantissa	Betydelse
0	0	± 0 (beror på S)
0	$\neq 0$	Ej normaliserat tal
2047	0	$\pm\infty$ (beror på S)
2047	$\neq 0$	NaN (Not a Number)

Tabell 2: Reserverade tal.

För att erhålla ∞ samt NaN i C++ utan att få varken kompileringsvarningar eller fel så behöver man först inkludera `<numeric>` och sedan anropa funktionerna:

```
std::numeric_limits<double>::infinity()
std::numeric_limits<double>::quiet_NaN()
```

I Java är motsvarande värden definierade som konstanter i klassen `Double` och kan kommas åt på följande sätt:

```
Double.NEGATIVE_INFINITY
Double.POSITIVE_INFINITY
Double.NaN
```

[1] Datatypen finns ej i Java.

[2] Storleken varierar mellan 80 och 128 bitar beroende på arkitektur.

[3] Resultatet av att någon operation misslyckades. Ex. $\sqrt{-1}$, $0.0/0.0$, $\infty - \infty$

1.2 Varför blir det fel?

Vi skall nu redogöra mer ingående för varför de märkliga fenomenen uppstod i de två första exemplen samt presentera ett par olika lösningsförslag.

Exempel 1

Talet 0.29 kan inte representeras exakt med en `double`. Datorn lagrar då istället den `double` som ligger närmast 0.29, vilken råkar vara $0x128F5C28F5C28F \cdot 2^{-54}$ och som har det ungefärliga värdet 0.28999999999999998. När talet sedan multipliceras med 100 och trunkeras till ett heltal erhåller vi resultatet 28.

Lösning

- Läs in talet som ”<int>.<int>”.
  Kan dock bli knepigt att kunna skilja på tal som t.ex. 5.1 och 5.01.
- Avrunda till närmsta heltal.
  `y = (int) (x * 100 + 0.5)`
  Detta fungerar så länge `x` inte är för stort (men i sådant fall så behöver man ändå en större datatyp för `y`).

Exempel 2

Talet 0.1 kan inte heller representeras exakt med en `double`. Men det kan däremot 1. Detta betyder att efter 10 iterationer i `for`-loopen så kommer inte variabeln x att anta exakt värdet 1 (men nästan). Termineringskriteriet kommer alltså aldrig att uppfyllas.

Lösning

Använd ej strikt jämförelse (`==`) för att kontrollera likhet mellan två flyttal a och b, ty a och b är sällan exakt lika. I detta fall lämpar det sig bättre att undersöka om talen är väldigt nära varandra.

Två metoder man kan tillämpa för att jämföra flyttal på är:

- **Absolutfel:**
  Vi definierar a och b som lika i absolut bemärkelse:
  $$a \overset{\text{abs}}{=} b \text{ om } |a - b| < \epsilon$$
  och a är mindre än b:
  $$a \overset{\text{abs}}{<} b \text{ om } a \overset{\text{abs}}{\neq} b \text{ och } a < b$$
- **Relativfel:**
  Vi definierar a och b som lika i relativ bemärkelse:
  $$a \overset{\text{rel}}{=} b \text{ om } |a - b| < \epsilon \cdot (|a| + |b|)$$
  och a är mindre än b:
  $$a \overset{\text{rel}}{<} b \text{ om } a \overset{\text{rel}}{\neq} b \text{ och } a < b$$

Absolutfel fungerar dåligt för mycket stora tal eftersom differensen mellan talen kan vara stor trots att den relativt sett är obetydlig. Relativfel fungerar dåligt för små tal av motsvarande skäl. Vill man vara helt säker så kan man använda båda metoder. Då gäller:

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

Vilket värde man skall välja på ϵ varierar från fall till fall, men ett värde mellan 10^{-7} och 10^{-13} räcker för de flesta tillämpningar.

1.3 float vs. double

I de flesta tillämpningar där man behöver använda flyttal så bör man välja `double`. Det ger betydligt högre precision än `float` och har väsentligen samma prestanda. `float` kan dock vara användbar om man har ont om minne, eller om avrundningsfel inte har någon betydelse, som t.ex. i datorgrafiksammanhang.

Om man behöver godtycklig precision så kan man använda sig av biblioteket *GMP*[4] för C++ eller klassen *BigInteger* i Java.

1.4 Exempel på när flyttal är att föredra framför heltal.

Finn $y = \sqrt[4]{x}, 1 \leq x \leq 10^{40}$ givet att y är ett heltal. Observera att x är för stort för att kunna lagras i en heltalstyp.

Lösning

1. Läs in x som en `double` för erhålla approximationen $X \approx x \cdot (1 \pm \epsilon)$ där $\epsilon \approx 10^{-15}$.
2. Låt $Y = \text{pow}(X, 0.25)$ vara en approximation av y.
3. Låt svaret vara Y avrundat till närmaste heltal.

1.4.1 Analys av precision

Låt oss analysera hur stort fel Y kan ha.

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

[4] GNU Multiple Precision Arithmetic Library, http://gmplib.org/

I tredje sista steget utnyttjade vi likheten $\frac{\ln(x)}{4} = \ln(y)$, och i näst sista steget utnyttjade vi att $e^{\pm\delta} \approx 1 \pm \delta$, för små värden på δ.

Vi får slutligen ett fel i vår beräkning som är $|Y - y| \approx y \cdot \epsilon \cdot (1 + \ln(y)) \approx 2 \cdot 10^{-4} \ll 0.5$, vilket är vad som krävs för att få korrekt svar efter avrundning.

1.5 Mer information

Mer utförlig information om flyttalshantering finns att läsa i följande artikel:

`http://docs.sun.com/source/806-3568/ncg_goldberg.html`

2 Heltalsaritmetik

2.1 Floor och ceil

Givet $x \in \mathbb{R}$ definieras:
$\lfloor x \rfloor$ = 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	5	6
5.3	5	6
-0.7	-1	0

Tabell 3: Exempel på floor och ceil

2.1.1 Operationer

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

$$\begin{aligned}
\lfloor x \rfloor = n &\Leftrightarrow n \leq x < n+1 \\
\lceil x \rceil = n &\Leftrightarrow n-1 < x \leq n \\
-\lfloor x \rfloor &= \lceil -x \rceil \\
-\lceil x \rceil &= \lfloor -x \rfloor \\
\lfloor x \pm n \rfloor &= \lfloor x \rfloor \pm n \\
\lceil x \pm n \rceil &= \lceil x \rceil \pm n \\
\left\lceil \frac{n}{m} \right\rceil &= \left\lfloor \frac{n+m-1}{m} \right\rfloor \text{ för } m > 0
\end{aligned}$$

2.1.2 Att beräkna $\lfloor \frac{n}{m} \rfloor$ och $\lceil \frac{n}{m} \rceil$

Det enklaste sättet att beräkna $\lfloor \frac{n}{m} \rfloor$ och $\lceil \frac{n}{m} \rceil$ är att först beräkna $\frac{n}{m}$ som en `double` och sedan använda sig av floor och ceil. Detta kommer att fungera så länge inte talen blir väldigt stora då vi riskerar att få avrundningsfel.

Dessa beräkningar kan göras på ett mer precist sätt. Idéen är att använda identiteterna ovan till att reducera alla fall till $\lfloor \frac{n}{m} \rfloor$ med $n, m > 0$, vilket ju precis motsvaras av heltalsdivision.

Algoritm 1: Beräkning av $\lfloor \frac{n}{m} \rfloor$, där $n, m \in \mathbb{Z}$

```
FLOOR-FRACTION(n, 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 2: Beräkning av $\lceil \frac{n}{m} \rceil$, där $n, m \in \mathbb{Z}$

```
CEIL-FRACTION(n, m)
(1)  if m > 0
(2)     return FLOOR-FRACTION(n + m − 1, m)
(3)  else
(4)     return FLOOR-FRACTION(−n − m − 1, −m)
```

2.1.3 Exempel på tillämpning

Kattisproblemet *Rare Easy Problem*[5] kan lösas på ett enkelt sätt med hjälp av floor och ceil. Problemet går ut på att hitta N givet $K = N - M$, där M är N med sista siffran borttagen.

Notera att $M = \lfloor N/10 \rfloor$. Vi vill alltså bestämma N så att $N - \lfloor N/10 \rfloor = K$.

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

Där vi använder definitionen av ceil, från 2.1.1, vid övergången från likhet till olikhet. Om K är delbart med 9 så kommer vi att få lösningarna $10K/9$ och $10K/9-1$, annars kommer lösningen att vara $\lfloor 10K/9 \rfloor$.

2.2 Stora heltal

Ibland räcker inte de primitiva datatyperna till. Exempelvis vore kryptosystem som RSA enkla att knäcka om man bara använde sig av 64-bitars nycklar. I verkliga tillämpningar använder man sig typiskt av färdiga bibliotek som *GMP* eller

[5] http://kattis.csc.kth.se/problem?id=easy

BigInteger. Här ska vi dock titta på hur man själv kan skapa enkla motsvarigheter vilket man kan vara tvingad till i problemlösningssammanhang.

2.2.1 Representation

Vi kan skriva ett tal x i någon bas b som:

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

där $0 \leq x_i < b$. Koefficienterna lagras exempelvis i en vektor.

2.2.2 Hur väljer vi b?

- $b = 2^k$, t.ex. 2^{32} eller 2^{64}

  Effektivt eftersom man utnyttjar minnesutrymmet väl vilket innebär att det blir få x_i och därmed få beräkningar. Att det dessutom är datorns interna representation gör att division- och moduloberäkningarna i nedanstående algoritmer kan implementeras effektivt med `shift`- och `and`-operationer. En nackdel är att en inläsning och utskrift i bas 10 blir jobbigt.

- $b = 10$

  Enkelt med inläsning, men ineffektivt eftersom b är så litet, vilket innebär att det blir det många x_i och därmed görs fler beräkningar och det krävs mer minne.

- $b = 10^d$

  Vid tillfällen då man behöver skriva ihop en lösning snabbt så är detta en bra kompromiss. In- och utläsning blir enkelt och samtidigt utnyttjas minnesutrymmet ganska väl. För att undvika `overflow` vid multiplikation så bör d väljas så att $10^{2d} = b^2$ får plats i den primitiva datatyp man valt för vektorn $[x_i]$. Om man t.ex. använder en 64-bitars heltalstyp så är $d = 9$ ett bra val.

2.2.3 Operationer

Vi antar att talen x, y och z är representerade som ovan i basen b. Vi tänker oss att $x_i = 0$ om $i > |x|$, där $|x|$ betecknar storleken på x:s vektor, samma sak gäller för y och z. Notera att dessa algoritmer är samma som man lärde sig i grundskolan.

Algoritm 3: Addition, $x + y$

ADD(x, y)
(1) $carry \leftarrow 0$
(2) $n \leftarrow \max(|x|, |y|)$
(3) **for** $i \leftarrow 0$ **to** $n - 1$
(4) $\quad tmp \leftarrow x_i + y_i + carry$
(5) $\quad z_i \leftarrow tmp \bmod b$
(6) $\quad carry \leftarrow \lfloor tmp/b \rfloor$
(7) $z_n \leftarrow carry$
(8) **return** z

Algoritm 4: Subtraktion, $x - y$

```
SUB(x, y)
(1)   if x < y
(2)     return −SUB(y, x)
(3)   borrow ← 0
(4)   n ← max(|x|, |y|)
(5)   for i ← 0 to n − 1
(6)     tmp ← x_i − y_i − borrow
(7)     z_i ← tmp mod b
(8)     if tmp < 0
(9)       borrow ← 1
(10)    else
(11)      borrow ← 0
(12)  return z
```

Algoritm 5: Multiplikation, $x \cdot y$

```
MUL(x, y)
(1)   z ← 0
(2)   n ← |x|
(3)   m ← |y|
(4)   for i ← 0 to n − 1
(5)     for j ← 0 to m − 1
(6)       z_{i+j} ← z_{i+j} + x_i · y_j
(7)     carry ← 0
(8)     l ← |z|
(9)     for j ← 0 to l − 1
(10)      z_j ← z_j + carry
(11)      carry ← ⌊z_j/b⌋
(12)      z_j = z_j mod b
(13)    z_l ← carry
(14)  return z
```

För varje varv i loopen gäller att:

$$\begin{aligned} z_{i+j} &= z_{i+j} \text{ från föregående varv} + x_i \cdot y_j + \text{minnessiffror} \\ &\leq b - 1 + (b-1)^2 + b - 1 \\ &= b^2 - 1 \end{aligned}$$

Det betyder att om vi väljer b så att $b^2 - 1$ ryms i den datatyp som används så löper vi ingen risk för `overflow`. Algoritmen har tidskomplexitet $\Theta(n^2)$, det finns dock betydligt snabbare algoritmer för multiplikation av stora tal.[6]

[6] Karatsuba $\Theta(n^{\log_2 3})$, Fast-Fourier-Transform $O(n \log n \log \log n)$

Algoritm 6: Division (med små nämnare)
Input: Heltal x och a, där $a < b$
Output: Tupel $(\lfloor x/a \rfloor, x \bmod a)$
DIV(x, a)
(1) $r \leftarrow 0$
(2) $n \leftarrow |x|$
(3) **for** $i \leftarrow n - 1$ **to** 0
(4) $\quad tmp \leftarrow b \cdot r + x_i$
(5) $\quad z_i \leftarrow \lfloor tmp/a \rfloor$
(6) $\quad r \leftarrow tmp \bmod a$
(7) **return** (z, r)

För att utföra division med stora nämnare krävs en betydligt krångligare algoritm.

2.2.4 Exponentiering

Antag att vi vill beräkna x^e för några heltal x och e. Den naiva ansatsen är att göra e stycken multiplikationer, vilket kan ta lång tid om e är stort. Det går att göra betydligt snabbare. Vi gör observationen att $x^e = x^{e \bmod 2} \cdot x^{2\lfloor \frac{e}{2} \rfloor}$ och utnyttjar det till att skapa en algoritm.

Algoritm 7: Beräkning av x^e genom upprepad kvadrering
SQUARE-AND-MULTIPLY(x, e)
(1) $z \leftarrow 1$
(2) **while** $e \neq 0$
(3) $\quad$ **if** $e \bmod 2 = 1$
(4) $\quad\quad z \leftarrow z \cdot x$
(5) $\quad x \leftarrow x \cdot x$
(6) $\quad e \leftarrow \lfloor e/2 \rfloor$
(7) **return** z

Eftersom e halveras i varje steg så kommer vi att göra $O(\log e)$ multiplikationer.