2D1458, Problemlösning och programmering under press

Föreläsning 8: Aritmetik och stora heltal

Datum: 2006-11-06
Skribent(er): Elias Freider och Ulf Lundström
Föreläsare: Per Austrin

Den här föreläsningen handlar om beräkningar med flyttal och behandling av heltal som inte ryms i grundläggande datatyper.

1 Flyttalsaritmetik

Flyttal beter sig inte alltid som man tror.

Exempel 1 - Avrundning

```
double x = 0.29;
int y = (int)(100*x);
print(y); //Skriver ut 28
```

Exempel 2 - Jämförelse

```
for(double x = 0.0; x!=1.0; x+=0.1)
  print("Hello"); //fortsätter i all oändlighet
```

1.1 Flyttalsrepresentation

Anledningen till beteendena vi stötte på ovan är att alla reella tal inte kan representeras exakt med grundläggande datatyper. Detta beror på hur flyttalen representeras i minnet.

I C/C++/Java representeras ett flyttal x enligt följande:

$x = S \cdot M \cdot 2^E$

S – teckenbit $\in \{1, -1\}$
M – mantissan (heltal)
E – exponenten (heltal) Storleken på mantissan och exponenten varierar med datatypen enligt tabell 1.

flyttalstyp	storlek	S	E	M
float	32	1	8	24
double	64	1	11	53
long double	128	1	15	113

Tabell 1: Antal bitar för S, E, M och totalt för olika flyttalstyper. Siffrorna för long double gäller sun-datorerna på Nada och kan variera mellan plattformar.

Som man ser i tabell 1 så är summan av antal bitar för de olika delarna ett mer än totala antalet bitar. Detta beror på att man använder normaliserad representation där den mest signifikanta biten i mantissan alltid är satt. Detta fungerar eftersom man alltid kan justera exponenten för att ”kompensera” för en större mantissa. Om den mest signifikanta biten inte skulle vara satt kan man i stället representera talet med $S(2M)2^{E-1}$. Därför behöver inte den högsta biten lagras.

1.1.1 En närmare titt på double

En double x representeras alltså på följande sätt:

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

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

Vissa värden på E är reserverade för speciella tal som inte kan representeras enligt denna formel.

En double med $E = 0$ representerar i stället

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

där ettan från föregående formel tagits bort för att kunna representera ± 0.

$E = 2047, M = 0$ betyder $\pm\infty$ beroende på s.

I C++ ger med många kompilatorer en varning om man skriver t.ex. 1.0/0.0 för att få doublevärdet för ∞. För att slippa varningen kan man istället skriva

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

$E = 2047, M \neq 0$ representerar ”Not a number”, NaN. Detta är t.ex. resultatet av 0.0/0.0 och ∞/∞. NaN har den speciella egenskapen att $NaN == NaN$ är falskt. NaN kan man få utan kompileringsfel genom

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

1.2 float vs. double

I de allra flesta fall där man behöver flyttal bör man använda double. Det ger en betydligt högre precision är float, men går ändå nästan lika fort att räkna med. Float kan dock vara bra om man har ont om minne, eller om avrundningsfel inte har någon betydelse, som t.ex. i datorgrafiksammanhang.

1.3 Varför blir det fel?

Exemplen i början har enkla förklaringar om man känner till datorns flyttalsrepresentation.

Exempel 1

0.29 kan inte representeras exakt med en double. Datorn lagrar då istället det doublevärde som ligger närmast 0.29 vilket råkar vara ungefär $0.28999999999999998 = 0.29 - 2 \cdot 10^{-18}$. När detta multipliceras med 100 och trunkeras till ett heltal får vi då 28.

Exempel 2

0.1 kan inte heller representeras exakt med en double. Det kan däremot 1. När vi har adderat 0.1 tio gånger kommer vi därför få ett tal som inte är exakt 1 och loopens slutkriterium kommer därför inte att uppfyllas.

1.4 Hur löser vi problemen?

Det finns ett antal lösningar på problemen i båda exemplen i början.

Exempel 1

Lösning 1: Avrunda till närmsta istället för nedåt vid omvandling till heltal. Detta görs enklast genom `y=(int)(x*100+0.5)`. Detta funkar om x inte är ohyggligt stort.
Lösning 2: Skriv ut talet som en double med 0 decimaler.
Lösning 3: Problemet uppstår normalt vid inläsning av flyttal. Då kan man istället läsa in talet som heltal, punkt, heltal. Tänk dock på att $5.01 \neq 5.1 = 5.10$.

Exempel 2

Lösning: Använd ej $a == b$ för att kontrollera likhet mellan a och b. Använd istället $|a - b| < \epsilon$ för något litet ϵ (absolut jämförelse) eller $|a - b| < \epsilon(|a| + |b|)$ (relativ jämförelse). Absolut jämförelse fungerar dåligt för mycket stora tal a och b eftersom differensen mellan talen då kan vara stor trots att den är relativt obetydlig. Relativ jämförelse fungerar dåligt för mycket små tal a och b eftersom . Vill man vara helt säker kan man använda båda. Då ska man anse det som likhet om någon av relativ eller absolut jämförelse ger likhet. Välj lämpligen $10^{-13} \leq \epsilon \leq 10^{-7}$. $\leq, <, \geq, >$ kan i vissa fall behöva implementeras på motsvarande sätt.

1.5 Flyttal vs. heltal

I de flesta fall där man kan välja bör man använda heltal istället för flyttal, eftersom man då slipper avrundningsfel. Det finns dock vissa fall då flyttal har sina fördelar.

1.5.1 Exempel: Finn $\sqrt[4]{x}$ givet att detta är ett heltal.

Lösning: Läs in x som en double, låt $y = \text{pow}(x, 0.25)$ och skriv ut y avrundat till 0 decimaler.

Motivering: Låt $X \approx x \cdot (1 \pm \epsilon)$ vara vår approximation av (d.v.s. det double-värde som ligger närmast) x. Låt

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

I näst sista steget används att $e^w = 1 + w + \frac{w^2}{2} + ...$ och $w \approx 0 \Rightarrow e^w \approx 1 + w$. Vi får då ett fel i vår beräkning som är $|Y - y| \approx y \cdot \epsilon \cdot \ln(y) \ll 0.5$ vilket är vad som krävs för att få korrekt svar efter avrundning.

1.6 Mer information

Mer utförlig information om flyttalshantering i datorer finns i följande artikel:

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

2 Floor och Ceil

Givet $x \in \mathbb{R}$ defineras
$\lfloor x \rfloor = \text{floor}(x)$ som x avrundat till närmsta heltal nedåt samt
$\lceil x \rceil = \text{ceil}(x)$ som x avrundat till närmsta helatal uppåt.

Tabell 2: Exempel på ceil och floor

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

2.1 Operationer

$$\lfloor x \rfloor = n \iff n \leq x < n + 1$$

$$\lceil x \rceil = n \iff n - 1 < x \leq n$$

2.2 Identiteter

$$-\lfloor x \rfloor = \lceil -x \rceil$$
$$-\lceil x \rceil = \lfloor -x \rfloor$$
$$\lfloor x \pm n \rfloor = \lfloor x \rfloor \pm n$$
$$\lceil x \pm n \rceil = \lceil x \rceil \pm n$$
$$\left\lceil \frac{n}{m} \right\rceil = \left\lfloor \frac{n + m + 1}{m} \right\rfloor \text{ för } m > 0$$

2.3 Beräkning av $\left\lfloor \frac{n}{m} \right\rfloor$ och $\left\lceil \frac{n}{m} \right\rceil$

Om m, n inte är för stora är det enklast att konvertera talen till doubles och använda floor()- eller ceil()-funktionerna.

Genom att använda identiteterna ovan kan man skriva mer robusta funktioner för floor och ceil:

```
floor(int n, int m)
1: if m<0 then
2:    return floor(-n, -m)
3: else if n<0 then
4:    return -ceil(-n, m)
5: else
6:    return n/m
7: end if
```

```
ceil(int n, int m)
1: if m>0 then
2:    return floor(n+m-1, m)
3: else
4:    return floor(-n-m-1,-m)
5: end if
```

2.4 Rare easy problem

Kattisproblemet rare easy problem (http://kattis.csc.kth.se/problem?id=easy) har en enkel explicit lösning som kan härledas med ceil och floor. Problemet går ut på att finna N givet $K = M - N$ där M är N med sista siffran borttagen.

Vi börjar med att observera $M = \lfloor N/10 \rfloor$.
Vi vill alltså lösa $N - \lfloor N/10 \rfloor = K$

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

Om K är delbart med 9 är lösningarna $10K/9$ och $10K/9 - 1$ annars $\lfloor 10K/9 \rfloor$.

3 Stora heltal

Stora heltal är heltal som inte får plats i standard-datatyper som long i Java och long long i C/C++. I Java finns BigInteger för att hantera dessa och i C/C++ finns GMP (GNU Multiprecision Library), men den senare finns oftast ej tillgänglig på Kattis.

3.1 Representation

Ett stort heltal x kan representeras i basen b som

$$x = x_{n-1}b^{n-1} + ... + x_1 b + x_0$$

där $0 \leq x_i < b$. Talen $x = (x_i)$ kan lagras som en vektor av tal.

Olika val av b kan vara fördelaktiga i olika sammanhang.

• **Potens av 2 t.ex.** $2^{32}, 2^{64}$ **eller** 2^{16}**.**

Detta är mest effektivt eftersom man utnyttjar minnesutrymmet väl och det då blir få x_i. Dessutom kan division- och modulo-operationerna som krävs i nedanstående algoritmer utföras effektivare med dessa baser eftersom det är den interna representationen i datorn. En nackdel är att det blir jobbigt med inläsning och utskrift i basen 10.

• $b = 10$

Detta är enkelt men ineffektivt då det kräver många x_i och därmed mycket minnesutrymme och fler beräkningar än med större b.

• $b = 10^d$

Bra kompromiss för enkel kod då det lätt går att skriva ut och mata in tal i basen 10 samtidigt som minnet utnyttjas ganska väl. T.ex. d = 9 och vi har 2^{64} long long. Vill ha att $10^{2d} = b^2$ får plats i datatypen för att slippa overflow vid t.ex. multiplikation.

3.2 Operationer

Variablerna x, y och z är stora positiva tal representerade i basen b som ovan. Vi tänker oss att $x_i = 0$ om $i > n$ där n är antalet tal i vektorn x. Samma sak gäller för y och z. Endast små logiska tillägg krävs för att kunna hantera även negativa tal. Algoritmerna är samma eller snarlika de som man använder vid beräkning på papper.

3.2.1 Addition

$z \leftarrow x + y$

1: $carry \leftarrow 0$
2: $n \leftarrow$ maximala storleken av x och y
3: **for** $i = 0$ to $n - 1$ **do**
4: $\quad tmp \leftarrow x_i + y_i + carry$
5: $\quad z_i \leftarrow tmp \bmod b$
6: $\quad carry \leftarrow [tmp \geq b]$
7: **end for**
8: $z_n \leftarrow carry$

3.2.2 Subtraktion

$z \leftarrow x - y$ där x antas vara större än y.

1: $borrow \leftarrow 0$
2: $n \leftarrow$ maximala storleken av x och y
3: **for** $i = 0$ to $n - 1$ **do**
4: $\quad tmp \leftarrow x_i - y_i - borrow$
5: $\quad z_i \leftarrow tmp \bmod b$
6: $\quad borrow \leftarrow [tmp < 0]$
7: **end for**

3.2.3 Multiplikation

$z \leftarrow x \cdot y$

1: $z \leftarrow 0$
2: $m \leftarrow$ storleken av x
3: $n \leftarrow$ storleken av y
4: **for** $i = 0$ to $m - 1$ **do**
5: **for** $j = 0$ to $n - 1$ **do**
6: $z_{i+j} \leftarrow z_{i+j} + x_i \cdot y_j$
7: **end for**
8: propagera minnessiffror i z
9: **end for**

Algoritmen kräver att den primitiva datatypen rymmer $b^2 - 1$. Det finns betydligt snabbare algoritmer för multiplikation av stora tal.

3.2.4 Division

$z \leftarrow \lfloor x/y \rfloor$ och $r \leftarrow x \bmod y$ där $y < b$

1: $r \leftarrow 0$
2: $n \leftarrow$ storleken av x
3: **for** $i = n - 1$ to 0 **do**
4: $tmp \leftarrow b \cdot r + x_i$
5: $z_i \leftarrow \lfloor tmp/y \rfloor$
6: $r \leftarrow tmp \bmod y$
7: **end for**

För att kunna hantera division med stora heltal krävs en betydligt krångligare algoritm.