DD2458, Problemlösning och programmering under press

Föreläsning 9: Talteori

Datum: 2007-11-13
Skribent(er): Niklas Lindbom och Daniel Walldin
Föreläsare: Per Austrin

Den här föreläsningen behandlar modulär aritmetik, kinesiska restsatsen, primalitet och faktorisering.

1 Modulär aritmetik

Modulär aritmetik innebär beräkningar innehållande mod n.

$a \equiv b (\text{mod } n) \Leftrightarrow a - b = k \cdot n$, för något $k \in \mathbb{Z}$

Vid implementation har vi $a = b \cdot \lfloor \frac{a}{b} \rfloor + a \bmod b$ eller $a = b \cdot (a \text{ div } b) + a \ \% \ b$ där $(a \text{ div } b)$ har olika betydelser för:

om $a \geq 0 : (a \text{ div } b) = \lfloor \frac{a}{b} \rfloor$
om $a < 0 : (a \text{ div } b) = \lceil \frac{a}{b} \rceil$

Där vi använder div för att beteckna heltalsdivision i programspråket.

1.1 Aritmetiska operatorer

När man gör beräknigar med addition, subtraktion, multiplikation och division med mod n får man svar tillhörande mängden $\mathbb{Z}_n = \{0, 1, ...n-1\}$. Exempelvis addition och subtraktion:

$(x + y) \bmod n$
$(x - y) \bmod n$

$0 \leq x, y < n$

Det är frestande att implementera mod genom att använda %-operatorn rakt av, men det fungerar inte för subtraktion eftersom:

$-12 \% 7 = -5$
$-12 \bmod 7 = 2$

Det kan enkelt avhjälpas genom att stega upp %-operatorn:

Algoritm 1: Implementation of modulo-operatorn.
Input: Heltal a, b.
Output: x, där $a \equiv x \bmod b$ och $0 \leq x < b$.
MOD(a, b)
(1) **if** $(a\%b) < 0$
(2) **return** $(a\%b + b)$
(3) **else**
(4) **return** $(a\%b)$

1.1.1 Addition

Vid addition enligt $(x + y)$ % n är användningen av %-operatorn onödigt långsam. En aningen snabbare metod av addition kan implementeras som:

Algoritm 2: Addition.
Input: Heltal x, y, n.
Output: $(x + y) \bmod n$
ADDITION(x, y, n)
(1) $z \leftarrow x + y$
(2) **if** $(z \geq n)$
(3) $z \leftarrow z - n$
(4) **return** z

1.1.2 Subtraktion

Subtraktion enligt $(x-y) \bmod n = (n+x-y)$ % n behöver inte heller göras onödigt krånglig då man vet att talen $x, y \in \mathbb{Z}_n$. Detta kan implementeras som:

Algoritm 3: Subtraktion.
Input: Heltal x, y, n.
Output: $(x - y) \bmod n$
SUBTRAKTION(x, y, n)
(1) **if** $(y > x)$
(2) $z \leftarrow n + x - y$
(3) **else**
(4) $z \leftarrow x - y$
(5) **return** z

1.1.3 Multiplikation

Multiplikation av två tal

$$z \leftarrow (x \cdot y) \ \% \ n$$

fungerar bra eftersom inget av talen är negativa men det kan förekomma vissa komplikationer vid beräkning av stora tal.

Problem:

Om n är stort så finns det risk för overflow. Det vill säga om man representerar alla tal med 32-bitars representation kan man bara representera 16 bitars modulär multiplikation.

Möjliga lösningar:

- Använd stora heltal - BigInteger finns färdigt i java.
- Använd "Försiktig multiplikation", se nedan.
- Om man räknar mod med ett n som ryms i 32 bitar kan man stoppa in alla tal i 64-bitars tal.

Försiktig multiplikation

Vid implementation: Kom ihåg att multiplikation förhåller sig till addition, som exponent förhåller sig till multiplikation.

Vid multiplikation kan man utnyttja

$$x \cdot y = x \cdot (y \bmod 2) + x \cdot (2 \cdot \lfloor \tfrac{y}{2} \rfloor)$$

för att utföra en så kallad försiktig multiplikation. DoubleAndAdd är en iterativ implementation av ovanstående, som kör $\log(y)$ iterationer.

Algoritm 4: Double and Add.
Input: Heltal x, y, n.
Output: $x \cdot y$
DoubleAndAdd(x, y, n)
(1) $z \leftarrow 0$
(2) **while** $y \neq 0$
(3) **if** $y \bmod 2 \neq 0$
(4) $z \leftarrow (z + x) \bmod n$
(5) $x \leftarrow (x + x) \bmod n$
(6) $y \leftarrow \lfloor \frac{y}{2} \rfloor$
(7) **return** z

1.1.4 Division

Det är viktigt att förstå vad som menas med division i det här fallet. Vid räkning av division tillsammans med modulo-operatorn är det inte helt självklart. Först och främst, vad betyder division? Jo, någon form av multiplikationsinvers.

$$z \equiv \frac{x}{y} \bmod n = x \cdot y^{-1} \bmod n$$

Vi behöver hitta en invers y^{-1} så att:

$$y \cdot y^{-1} = 1 \bmod n$$

För att hitta en sådan invers kan man använda Euklides algoritm som baseras på följande:

$$\exists\, y^{-1} : y \cdot y^{-1} = 1 \bmod n$$

$$\Leftrightarrow$$

$$\exists\, k, y^{-1} : y \cdot y^{-1} + k \cdot n = 1 \Leftrightarrow \gcd(y, n) = 1$$

Euklides algoritm

Givet $a, b \in \mathbb{Z}$ hittar Euklides algoritm $x, y \in \mathbb{Z}$ så att $x \cdot a + y \cdot b = \gcd(a, b)$ vilket är precis det verktyg vi behöver för att kunna dividera.

Algoritm 5: Euklides algoritm.
Input: Heltal a, b.
Output: Ger x, y som uppfyller $x \cdot a + y \cdot b = \gcd(a, b)$.

```
EUCLIDEAN(a, b)
(1)  if a = 0
(2)      return (0, 1)
(3)  else if b = 0
(4)      return (1, 0)
(5)  else
(6)      (y', x') ← EUCLIDEAN(b, a % b)
(7)      return (x', y' − (a div b) · x')
```

Korrekthet

Låt $d = \gcd(a, b) = \gcd(b, a \bmod b)$ så

$$b \cdot y' + (a \bmod b) \cdot x' = d$$

Vi vet att $a \bmod b = a - b \cdot \lfloor \frac{a}{b} \rfloor$ så

$$b \cdot y' + \left(a - b \cdot \lfloor \tfrac{a}{b} \rfloor\right) \cdot x' = d$$

$$a \cdot x' + b\left(y' - x' \lfloor \tfrac{a}{b} \rfloor\right) = d$$

Körtid

Man kan visa att efter två rekursiva anrop kommer någon av a och b att ha halverats. Det maximala antalet rekursiva anrop, och därmed körtiden, är därför $\mathcal{O}(\log a + \log b)$.

2 Kinesiska restsatsen

Problem:

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \end{cases}$$

Givet att $\gcd(n_1, n_2) = 1$

Lös ekvationen m.a.p. x

Kinesiska restsatsen (el. CRT - Chinese Remainder Theorem) säger att: Det existerar ett unikt $x \bmod M$, $M = n_1 n_2$, som uppfyller ekvationen.

Låt:

$$m_1 = n_1^{-1} \bmod n_2$$
$$m_2 = n_2^{-1} \bmod n_1$$

Och bilda $x = (a_1 m_2 n_2 + a_2 m_1 n_1) \bmod M$. Vi visar nu att detta x löser ekvationssystemet: $x \bmod n_1 = a_1 n_2^{-1} n_2 + a_2 m_1 n_1 \pmod{n_1} = a_1$, eftersom $n_2^{-1} n_2 = 1$ och $a_2 m_1 n_1 \pmod{n_1} = 0$. Pss för den andra ekvationen.

Vad gör man om n_1, n_2 ej relativt prima (dvs $\gcd(n_1, n_2) \neq 1$) då? Inför variabelbytet $x' = x - a_1, a' = a_2 - a_1$ som ger:

$$x' = 0 \pmod{n_1}$$
$$x' = a' \pmod{n_2}$$

Låt $d = \gcd(n_1, n_2)$, $n_1 = d \cdot n_1'$ och $n_2 = d \cdot n_2'$

$$x' = k_1 n_1 = k_1 \cdot d n_1'$$
$$x' = k_2 n_2 + a' = k_2 \cdot d n_2' + a'$$

Ur ekvationerna ovan får vi $a' = d \cdot (k_1 n_1' - k_2 n_2')$. Man måste alltså ha ett a'

som är jämnt delbart med d för att få en lösning. Vi kan då skriva:

$$a' = d \cdot a''$$
$$x' = d \cdot x''$$

Så att:

$$x'' = k_1 n_1'$$
$$x'' = k_2 n_2' + a''$$

Och med andra ord:

$$x'' = 0 \pmod{n_1'}$$
$$x'' = a'' \pmod{n_2'}$$

$\gcd(n_1', n_2') = 1 \Rightarrow$ de är relativt prima - kan lösas på samma sätt som tidigare.

Lösning:

$x = d \cdot x'' + a$ som är unik mod $(n_1' n_2' \cdot d)$

3 Primalitet

Givet tal $N \in \mathbb{Z}^+$, avgör om N är ett primtal.
($n = \log_2 N = \#$ bitar i N)

Naivt: Prova alla tal upp till $\sqrt{N}$ som möjliga faktorer. Detta är dock långsamt, $\mathcal{O}(\sqrt{N}) = \mathcal{O}(2^{\frac{n}{2}})$.

3.1 Primtalsåll - Eratosthenes såll

Ide: Skapa en array isprime[1...N] som talar om för varje tal om det är ett primtal eller inte.
Konstruktionen börjar med antagandet att alla tal är primtal varefter algoritmen sållar bort alla tal som inte är primtal.

Algoritm 6: Implementation av Eratosthenes såll.
Input: Storlek N av primtals-sållet.
Output: bool-array för uppslagning av primtal.
ERATOSTHENES(N)

```
(1)   for i = 2 to N
(2)       isprime[i]← true
(3)   for p = 2 to N
(4)       if isprime[p]
(5)           for r = 2p, 3p, ... to N
(6)               isprime[r]← false
```

Komplexiteten för algoritmen är:

$$\mathcal{O}\left(N + \sum_{p \leq N} \frac{N}{p}\right) = \mathcal{O}\left(N \cdot \sum_{p \leq N} \frac{1}{p}\right) = \mathcal{O}\left(N \log(\log(N))\right) = \mathcal{O}\left(2^n \log(n)\right).$$

Där vi i mittensteget utnyttjade att $\sum_{p \leq N} \frac{1}{p} \approx \log(\log(N)) + B_1$. Där $B_1 \approx 0.2615$ kallas Merten's konstant.

Vid implementering finns det många sätt att göra algortimen snabbare och mer minnessnål.

Implementationstrick

De trick som är listade nedan kommer inte att förbättra den asymptotiska tidskomplexiteten, men programmet kommer att gå betydligt snabbare.

Spara bara udda tal i sållet

- Specialbehandla tal 1 och 2, returnera "icke primtal" respektive "primtal".
- Lagra endast udda tal från 3 och uppåt.

Då vi inte lagrar jämna tal kommer minnesanvändningen att halveras. Genom att bara undersöka udda tal blir det färre p att gå igenom i den yttre loopen. Dessutom blir det hälften så många r att undersöka i den innersta loopen $(3p, 5p, 7p, ...)$, för varje p. Sammantaget minskar det körtiden för algoritmen ungefär en faktor 4.

Lagra 8 bools per byte

- En listig implementation av arrayen minskar minnesanvändningen avsevärt.

Det innbär att vi reducerar minnesanvändningen med en faktor 8 och kommer att göra implementationen snabbare tack vare förbättrad cache-prestanda.

Gå inte igenom bool-arrayen onödigt antal gånger

- För att ytterliggare optimera algoritmen vill vi undvika att titta på tidigare uppslagningar i arrayen.

Om till exempel $p = 7$, så har vi redan slagit upp $3 \cdot 7$ och $5 \cdot 7$, när p var 3 och 5. För att undvika titta att på dessa ännu en gång kan den inre loopen bytas från $r = 3p, 5p, 7p, ...$ till $r = p^2, p^2 + 2p, p^2 + 4p, ...$ Vi kommer dock inte undvika alla tidigare uppslagningar, som tex $3 \cdot 5 \cdot 7$.

3.2 Miller-Rabin

Fermats lilla sats säger att om N primtal gäller $a^{N-1} \equiv 1 \pmod{N}$ för alla $a \in \mathbb{Z}_N$. Vilket leder oss till en idé:

Algoritm 7: Primtalstest.
Input: Tal N att testa för primalitet.
Output: Sant eller falskt.
PRIMTALSTEST(N)
(1) $a \leftarrow random(1, N-1)$
(2) $b = a^{N-1} \bmod N$
(3) **if** $b \neq 1$
(4) **return** N är ej ett primtal
(5) **else if** $b = 1$
(6) **return** Gissa att N är ett primtal

Där $random(1, N-1)$ slumpar ett värde mellan 1 och $N-1$.

...Tyvärr visar det sig att detta inte riktigt fungerar. Så kallade Carmichael-tal är ej primtal och uppfyller: $a^{N-1} \equiv 1 \pmod{N}$ för alla $a \in \mathbb{Z}_N$.

Miller-Rabin primalitetstest utnyttjar att:

- Om N är ett primtal finns bara två kvadratrötter till 1 (mod N) : ± 1
- Om N inte är ett primtal finns minst fyra kvadratrötter.

Metod:

- Skriv $N - 1 = s \cdot 2^r$ (s udda).
- Tag slumpvis $a \in \mathbb{Z}_N$ och låt $u_0 = a^s, u_i = u_{i-1}^2$ (observera $u_r = a^{N-1}$).
- Om nu $u_0 \neq 1$ och alla $u_i \neq -1$ $(0 \leq i < r)$ så $\Rightarrow$ N ej primtal.
- Annars $\Rightarrow$ gissa att det är ett primtal.

Genom att upprepa denna procedur flera gånger kan vi med stor sannolikhet bestämma om N är ett primtal eller inte. Om man kör testet k gånger är sannolikheten att få fel 4^{-k}.

Algoritm 8: Miller-Rabin primalitetstest.
Input: Tal N att testa för primalitet och k antal tester.
Output: Sant eller falskt.

```
MillerRabin(N, k)
(1)   for i = 1 to k
(2)     Låt N − 1 = 2^t · u, där t ≥ 1 och s udda.
(3)     a ← random(1, N − 1)
(4)     u_0 ← a^s mod N
(5)     for j = 1 to t
(6)       u_j ← u_{j−1}^2 mod N
(7)       if u_j = 1 och u_{j−1} ≠ 1 och u_{i−1} ≠ N − 1
(8)         return N är ej ett Primtal
(9)     if u_t ≠ 1
(10)      return N är ej ett Primtal
(11)  return N är primtal
```

4 Faktorisering

För faktorisering av tal finns en uppsjö metoder, bland annat:

Naivt $\mathcal{O}(\sqrt{N})$

Pollard-ρ $\mathcal{O}(\sqrt[4]{N})$

Den naiva lösningen är att testa talet genom att dela talet ifråga med alla heltal $\leq \sqrt{N}$ som då givetvis får komplexiteten $\mathcal{O}(\sqrt{N})$.

En bättre och snabbare lösning fås genom att använda Pollards ρ-algoritm som har komplexitet $\sqrt[4]{N}$. Den bygger på observationen att x och y är kongruenta mod p med sannolikhet 0.5 efter att $1.177\sqrt{p}$ tal har valts slumpvisa. Om p är en av faktorerna i N, talet vi vill faktorisera, så har vi att $1 < \gcd(|x-y|, N) \leq N$ eftersom p delar både $|x-y|$ och N. [1]

Algoritm 9: Pollards ρ-algoritm.
Input: N, talet som ska faktoriseras. $f(x)$ är en slumpfunktion modulo N.
Output: en icke-trivial faktor från N, annars förkasta.

```
ALGORITM PollardRho(N)
(1)   x ← 2
(2)   y ← 2
(3)   d ← 1
(4)   while d = 1
(5)       x ← f(x)
(6)       y ← f(f(y))
(7)       d ← gcd(|x − y|, N)
(8)   if d = N
(9)       return failure
(10)  else
(11)      return d
```

[1] `http://en.wikipedia.org/wiki/Pollard_rho, 2007-11-15`