DD2458, Problemlösning och programmering under press

Föreläsning 9: Talteori

Datum: 2009-11-11
Skribent(er): Ting-Hey Chau, Gustav Larsson, Åke Rosén
Föreläsare: Fredrik Niemelä

Den här föreläsningen handlar om talteori och behandlar främst modulär aritmetik, största gemensamma delare och primalitet.

1 Fortsättning från föregående föreläsning

För att beräkna x^k för heltalen $x, k \geq 0$ betraktas först den naiva lösningen

$$x^k = \underbrace{x \cdot x \cdot x \cdot \cdots \cdot x}_{k},$$

som har komplexiteten $\mathcal{O}(k)$ om inte talens längd tas med i betraktelsen. Om man istället räknar $x \cdot x$ och tar det värdet och multiplicerar det med sig självt krävs som exempel endast två räkneoperationer istället för tre för att räkna ut x^4. Detta kan ses som

$$x^k = x^{k \bmod 2} \cdot x^{\lfloor k/2 \rfloor^2}.$$

Varje steg i operationen kan ses som att man stegar igenom ettorna och nollorna i den binära representationen av k:

$$k = b_0 + 2b_1 + 2^2 b_2 + \ldots.$$

För varje steg undersöker man om $x^{k \bmod 2}$ är 1 och multiplicerar i så fall på det x^n-värdet. Detta sätt kräver endast $\log_2 k$ iterationer och kan ses i algoritm 1.

Algoritm 1: Snabb potensräkning för heltal.
Input: Två heltal $x, k \geq 0$
Output: x^k

```
POW(x, k)
(1)   r ← 1
(2)   while k ≠ 0
(3)     if k mod 2 = 1
(4)       r ← r · x
(5)     x ← x^2
(6)     k ← ⌊k/2⌋
(7)   return r
```

2 Modulär aritmetik

Det finns en skillnad mellan det man i matematiken menar med modulo och %-operatorn i programspråk som C++:

Definition 2.1. $a \bmod b = a - b\lfloor \frac{a}{b} \rfloor$

Definition 2.2. `a%b = a - b*(a/b)`, *där* `(a/b)` *i exempelvis C++ trunkerar tal och fungerar annorlunda än* $\lfloor a/b \rfloor$ *för negativa tal, därför behöver man vara varsam när man använder* `%`*-operatorn för negativa tal.*

2.1 Räkneoperationer i modulo n

Räkneoperationer för heltalen $0 \leq a, b < n$:

Addition `(a + b)%n`

Subtraktion `(n + a - b)%n`
n läggs till för undvika att % matas med ett negativt tal för att åstadkomma $(a - b) \bmod n$.

Multiplikation `(a * b)%n`
Det här sättet att multiplicera a och b kan vara opraktiskt då det kräver att man kan lagra väldigt stora tal. Ett alternativt sätt, som inte kräver att ab är inom datatypens lagringsförmåga, är att använda upprepade additioner modulo n. Pseudo-kod kan ses i algoritm 2.

Division Division definieras enligt $b/a = ba^{-1}$, där a^{-1} är den multiplikativa inversen så att $aa^{-1} \equiv 1 \pmod{n}$. Eftersom det här kräver att a och n är relativt prima är division inte definierad om det inte uppfylls. Inversen kan beräknas med Euklides utökade algoritm (se 2.4).

Algoritm 2: Modulär multiplikation av heltal som inte kräver att $a \cdot b$ är inom datatypens lagringsförmåga.
Input: $\mathbf{Z} \ni a, b, n \geq 0$
Output: $a \cdot b \pmod{n}$
ModMult(a, b, n)
(1) $r \leftarrow 0$
(2) **while** $b \neq 0$
(3) **if** $b \bmod 2 = 1$
(4) $r \leftarrow (r + a) \bmod n$
(5) $a \leftarrow$ModPlus(a, a) enligt definitionen i 2.1
(6) $e \leftarrow \lfloor e/2 \rfloor$
(7) **return** r

2.2 Största gemensamma delare

Definition 2.3. *Låt a, b vara två heltal,* $a, b \neq 0$. $GCD(a, b)$ *är det största tal som delar både a och b utan rest.*

Definition 2.4. *Om $GCD(a, b) = 1$ så säger man att a och b är relativt prima.*

Den naiva metoden för att räkna ut $GCD(a, b)$ är att primtalsfaktorisera a och b, lagra faktorerna i $\{A_i\}$ respektive $\{B_j\}$ och sedan ta produkten av alla element de har gemensamt. Denna metod är dock väldigt ineffektiv.

Lemma 2.5. *$GCD(a, b)$ delar $a - b$.*

Bevis. Eftersom $GCD(a, b)$ delar både a och b kan vi skriva ($x_a, x_b \in \mathbf{Z}$)

$$\begin{aligned} a &= x_a \cdot GCD(a, b) \\ b &= x_b \cdot GCD(a, b) \\ a - b &= GCD(a, b) \cdot (x_a - x_b), \end{aligned}$$

eftersom $(a - b)$ är uttryckt som en heltalsmultipel av $GCD(a, b)$ är beviset klart. □

Detta ligger till grunden för Euklides algoritm för att hitta $GCD(a, b)$. Låt oss anta att $a > b$ (eftersom GCD är kommutativ kan vi byta plats på a och b). Enligt lemma 2.5 gäller $GCD(a, b) = GCD(a, a - b) = GCD(b, a - b)$ men då gäller också $GCD(a, b) = GCD(b, a \bmod b)$ eftersom modulo kan ses som upprepade minusoperationer. Detta implementeras rekursivt enligt algoritm 3.

För fibonaccitalen F_{N+1}, F_{N+2} kan man visa att det krävs N steg för att beräkna $GCD(F_{N+1}, F_{N+2})$ och att det inte finns några mindre tal som kräver fler steg. Detta ger ett värsta fall för algoritmen. Om algoritmen hittar $GCD(a, b)$, $a > b$, i N steg så betyder det att det minsta talet $b \geq F_{N+1} \geq \phi^N$, där ϕ är det gyllene snittet. Då gäller

$$\begin{aligned} \phi^N &< b \\ N &< \log_\phi(b) = \log_2(b)/\log_2(\phi) \approx 1{,}44 \log_2 b, \end{aligned}$$

och $1{,}44 \log_2 b$ ger då en övre gräns för antal modulo-operationer som behöver utföras för Euklides algoritm.

Algoritm 3: Euklides algoritm.
Input: $\mathbf{Z} \ni a, b \geq 0$
Output: Största gemensamma delare av a och b

```
GCD(a, b)
(1)   if b = 0
(2)      return a
(3)   else
(4)      return GCD(b, a mod b)
```

2.3 Steins algoritm

En ofta ännu effektivare metod för att beräkna GCD(a, b) är Steins algoritm som inte använder några multiplikationer eller divisioner utan istället endast binära shift-operationer. Denna metod används fördelaktigt på stora tal, då shift-operationer är mycket billigare än aritmetiska funktioner för stora tal.

Algoritm 4: Steins algoritm.
Input: $\mathbf{Z} \ni a, b \geq 0$
Output: Största gemensamma delare av a och b
SteinGCD(a, b)
(1) **if** $a = 0$ **or** $b = 0$
(2) **return** max(a, b)
(3) $s \leftarrow 0$
(4) **while** a, b båda jämna
(5) $a \leftarrow \lfloor a/2 \rfloor$ (binärshifta ett steg åt höger)
(6) $b \leftarrow \lfloor b/2 \rfloor$
(7) $s \leftarrow s + 1$
(8) **while** b jämn
(9) $b \leftarrow \lfloor b/2 \rfloor$
(10) **while** $a \neq 0$
(11) **while** a jämn
(12) $a \leftarrow \lfloor a/2 \rfloor$
(13) **if** $a < b$
(14) $a, b \leftarrow b, a$
(15) $a \leftarrow a - b$ (jämn efter tilldelningen)
(16) $a \leftarrow \lfloor a/2 \rfloor$
(17) **return** $b \cdot 2^s$

2.4 Euklides utökade algoritm

Euklides utökade algoritm returnerar den största gemensamma delaren i form av multipler av parametrarna a och b. Metoden kan användas för att hitta den multiplikativa inversen till a mod b. Algoritmen hittar talen x och y i Bézouts identitet:

$$ax + by = GCD(a, b)$$

Den utökade algoritmen kan även användas för att hitta den multiplikativa inversen till ett tal a mod n, genom att räkna ut $ax + bn = 1$ (notera att $GCD(a, n)$ måste vara 1 för att division ska vara definierad), x identifieras sedan som inversen a^{-1}.

Algoritm 5: Euklides utökade algoritm.
Input: Två heltal $a, b \geq 0$
Output: Returnerar $(x, y) \in \mathbf{Z}^2$ så att $ax + by = GCD(a, b)$.
ExtGCD(a, b)
(1) **if** $b = 0$
(2) **return** $(1, 0)$
(3) **else if** $a = 0$
(4) **return** $(0, 1)$
(5) **else**
(6) $(y, x) \leftarrow$ ExtGCD(b, $a \bmod b$)
(7) **return** $(x, y - \lfloor a/b \rfloor x)$

3 Kinesiska restsatsen

Sats 3.1 (Kinesiska restsatsen)**.** *Låt $n_1, n_2, \ldots, n_k$ vara relativt prima positiva tal, då har följande system av kongruenser*

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \\ x \equiv a_3 \pmod{n_3} \\ \quad \vdots \\ x \equiv a_k \pmod{n_k} \end{cases}$$

en lösning i modulo $n = n_1 n_2 \ldots n_k$.

Bevis. Eftersom alla tal i $\{n_i\}$ är relativt prima (enligt definition 2.4), existerar ett tal b_i för alla $i = 1, 2, \ldots, k$ sådant att

$$b_i N_i \equiv 1 \pmod{n_i}, \text{ där } N_i = \frac{n}{n_i}.$$

Eftersom N_i innehåller all faktorer i n förutom n_i betyder det att $b_i N_i \equiv 0 \pmod{n_j}$ för alla $j \neq i$. Låt sedan

$$x = a_1 b_1 N_1 + a_2 b_2 N_2 + \cdots + a_k b_k N_k,$$

eftersom alla termer i x är 0 i modulo i förutom $a_i b_i N_i$ som är a_i, löser x systemet. För att hitta en alternativ lösning antas ett alternativt värde till b_i (för godtyckligt i) som vi kallar b_i'. Det nya värdet kan uttryckas $b_i' = b_i + c_i$ där

$$c_i \equiv 0 \pmod{n_i},$$

vilket försäkrar att b_i' fortfarande ger en lösning eftersom

$$b_i' N_i = \underbrace{b_i N_i}_{\equiv 1} + \underbrace{c_i N_i}_{\equiv 0} \equiv 1 \pmod{n_i}.$$

c_i måste således vara en multipel av n_i, vilket gör $c_i N_i$ till en multipel av n. Ändringen av b_i' för alla i ändrar alltså alltid lösningen med en multipel av n, vilket bevisar att lösningen alltid är i modulo n. □

Kinesiska restsatsen kan implementeras enligt lösningsmetoden för x i ovanstående bevis. För att bestämma b_i används Euklides utökade algoritm (algoritm 5) genom följande tilldelning

$$(b_i, y) \leftarrow \text{ExtGCD}(N_i, n_i),$$

och eftersom y representerar en multipel av n_i påverkar den inte kongruensen och kan slängas bort.

3.1 Inte relativt prima

Om $n_1, n_2, \ldots, n_k$ inte är relativt prima finns en metod för att skriva om problemet så att de är det. Metoden går ut på att två kongruensrelationer som inte är relativt prima skrivs om till en relation. Det här görs med par tills alla är relativt prima och systemet kan lösas med kinesiska restsatsen.

Säg att $d = GCD(n_1, n_2) \neq 1$ och inför till att börja med variabelbytena $x' = x - a_1$ och $a' = a_2 - a_1$; det följer nu att

$$\begin{cases} x' & \equiv & 0 & \pmod{n_1} \\ x' & \equiv & a' & \pmod{n_2}. \end{cases}$$

n_1 och n_2 kan skrivas som multipler av d enligt $n_i = dn_i'$ för $i = 1$ och 2. Ovanstående kongruenser kan då skrivas om till (där $k_i \in \mathbf{Z}$)

$$\begin{cases} x' & = & k_1 \cdot dn_1' \\ x' & = & k_2 \cdot dn_2' + a' \end{cases}$$

Bryts nu a' ut ser man att det måste var en multipel av d, vilket gör att både a' och x' kan skrivas som

$$\begin{cases} a' & = & da'' \\ x' & = & dx''. \end{cases}$$

x'' bryts ut och det är enkelt att se att $x'' = k_1 n_1'$ är 0 i modulo n_1' och $x'' = k_2 n_2' + a''$ är a'' i modulo n_2'. Eftersom n_1' och n_2' är relativt prima (eftersom deras gemensamma delare d har tagits bort) kan följande system lösas med vanliga metoden:

$$\begin{cases} x'' & \equiv & 0 & \pmod{n_1'} \\ x'' & \equiv & a'' & \pmod{n_2'}. \end{cases}$$

Lösningen ges sedan av $x = dx'' + a_1$.

4 Primalitet

Definition 4.1. *Ett heltal $n \geq 2$ är ett primtal om och endast om det är delbart med sig självt, talet 1 och inget annat.*

Den naiva metoden för att avgöra om ett tal p är ett primtal är att dividera p med alla tal som ligger mellan 1 och p, om inget tal delar p är det ett primtal.

4.1 Erastothenes primtalssåll

Ett primtalssåll avgör inte bara primaliteten hos talet p, utan alla tal upp till p.

1. Skapa en lista med alla tal från 1 till p.

2. Börja med talet $i = 2$ (eftersom det är det första primtalet)
3. Stryk alla dess multipler upp till p.
4. Fortsätt med nästa primtal (talet som ännu inte är struket).
5. Repetera steg 3 och 4 tills i är större än p.
6. När i har itererat klart har alla tal antingen markerats som primtal eller strukits.

Tidskomplexiteten för Erastothenes såll är

$$\mathcal{O}(N + \sum \frac{N}{p}) = \mathcal{O}(N + \sum_{p \leq N} \frac{1}{p}),$$

där

$$\sum_{p \leq N} \frac{1}{p} \approx \log\log N + \underbrace{0{,}2615}_{\text{Mertens konstant}}.$$

En schematisk bild av sållet kan ses i figur 4.1 och pseudo-kod i algoritm 6.

1	②	③	~~4~~	5	~~6~~	7	~~8~~	9
~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	...	
⋮								

1	②	③	~~4~~	⑤	~~6~~	⑦	~~8~~	~~9~~
~~10~~	⑪	~~12~~	⑬	~~14~~	~~15~~	~~16~~	...	
⋮								

Figur 1: Visar hur sållet ser ut först efter att alla tal delbara med 2 har strukits och sedan när det är färdigt. Talen som är markerade med en cirkel är primtal.

Algoritm 6: Erastothenes primtalssåll.
Input: $N \in \mathbf{N}$
Output: En lista $\{P_i\}$ för $i = 1, 2, 3 \ldots N$ där P_n är **true** om n är ett primtal, annars **false** .

```
PrimeSieve(N)
(1)   for i = 2 to N
(2)     P_i ← true
(3)   for p = 2 to N
(4)     if P_p = true
(5)       for i = 2p, 3p, ... to N
(6)         P_i ← false
```

4.1.1 Optimering

Antag att talet p är en produkt av två tal, där båda talen samtidigt inte kan vara större än $\sqrt{p}$, för om så är fallet kommer produkten av talen vara större än talet p,

$$p = a \cdot b \implies a \leq \sqrt{p} \text{ eller } b \leq \sqrt{p}.$$

På så vis vet vi att det räcker med att gå igenom tal upp till $\sqrt{p}$. Detta är dock fortfarande långsamt och kräver $\sqrt{p}$ iterationer.

Ytterligare optimering kan göras genom att ändra den inre loopen på rad 5 till en iteration över $i^2, i^2 + 2i, i^2 + 4i, \ldots, p$. Detta minskar antalet uppslagningar eftersom vi slipper att iterera exempelvis både 7×11 och 11×7.

Vidare kan man optimera genom att enbart iterera över udda tal i den yttre loopen eftersom 2 är det enda jämna primtalet. Om vi enbart lagrar de udda talen lyckas vi även halvera minnesanvändningen; $p = 1$ och $p = 2$ hårdkodas in i algoritmen.

Genom att lagra åtta element av $\{P_i\}$ i varje byte istället för att använda en `bool`-vektor så kan vi minska minnesanvändningen med en faktor 8. Detta ger också en bättre cache-lokalitet.

4.2 Miller–Rabin

Miller–Rabin är ett primalitetstest som bygger på Fermats lila sats:

Definition 4.2 (Fermats lilla sats)**.** *Om p är ett primtal, så gäller för varje heltal a att*

$$a^{p-1} \equiv 1 \pmod{p}.$$

Algoritmen bygger på att testa för ett stort antal a i ovannämnda sats så att primaliteten hos p kan fastställas med stor sannolikhet. Sannolikheten att svaret är sant för ett sammansatt tal är inte mer än 4^{-k}, där k är antal test). Om algoritmen returnerar falskt, betyder det att det garanterat inte är ett primtal.

Algoritmen är mycket effektivare för att bestämma primalitet än Erastothenes primtalssåll, eftersom den inte kräver kunskap om alla primtal upp till talet p vi testar. En nackdel är att finns en hel klass med tal som uppfyller Fermats lilla sats för alla $a \in \mathbf{Z}$, men ändå inte är primtal. Dessa tal kallas Carmichael-tal och det första hittas vid talet 561, som är produkten av $3 \times 11 \times 17$.

Pseudo-kod kan ses under algoritm 7.

5 Faktorisering av stora heltal

Det finns flera olika metoder för att faktorisera heltal, bland annat kvadratiskt såll och elliptiska kurvor. En enkel metod är Pollard–ρ som beskrivs nedan.

5.1 Pollard–ρ

Pollard–ρ är en algoritm för faktorisering av ett tal, n, som använder en funktion modulo n, $f(x)$, för att generera en pseudo-slumpmässig talföljd. Två tal x, y genereras genom $x = f(x)$ och $y = f(f(y))$, det vill säga y går igenom följden dubbelt så snabbt och i varje iteration beräknas $d = GCD(|x - y|, n)$. Så länge $d = 1$ upprepas

Algoritm 7: Miller–Rabin
Input: N, k
Output: Returnerar **true** om det antagligen är ett primtal och **false** om det definitivt inte är det.

```
MillerRabin(N, k)
(1)    repeat k
(2)       Sätt s och x så att N − 1 = 2^s · x
(3)       a ← Random(2, N − 1)
(4)       u ← a^x mod N
(5)       if n = −1
(6)          next k
(7)       for i = 1 to s − 1
(8)          n ← u^2 mod N
(9)          if u = 1
(10)            return false
(11)         if u = −1
(12)            next k
(13)         return false
(14)      return true
```

detta, om $d = n$ så returnerar algoritmen **fail** och om $1 < d < n$ så är d en faktor till n.

Om $d = n$ betyder det att en cykel i talföljden har hittats och att forsätta blir då en upprepning av tidigare arbete. I detta fall kunde ingen faktor hittas. Lösningen på detta är att helt enkelt byta $f(x)$ mot en annan funktion.

Om n är ett primtal så kommer $GCD(|x - y|, n)$ alltid att vara 1 oberoende av vilket $f(x)$ som används.

Algoritm 8: Pollard–ρ.
Input: n, talet som ska bli faktoriserat.
Output: en icke-trivial faktor av n, eller **fail** om den misslyckas.

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