TMA4140
Diskret matematikk

Plenumsregning 7

Høst 2024

1 Betrakt heltallene $R = \{0, 1, 2, 3\}$. Regn ut gangetabellen for tallene i R, mod 4. Mer presist, for alle $(a, b) \in R \times R$, skriv ned tallet $r \in R$ slik at

$$ab \equiv r \pmod 4$$

2 Vis at et heltall $k > 0$ kan deles på 3 hvis summen av sifrene, når k er skrevet i titallsystemet, også kan deles på 3.

(Hint: k kan deles på 3 hvis $k \equiv 0 \pmod 3$. Observer at $10 \equiv 1 \pmod 3$, og generelt $10^i \equiv 1 \pmod 3$ for alle $i \geq 0$. Dermed har vi for eksempel $321 \equiv 300 + 20 + 1 \equiv 3 + 2 + 1 \pmod 3$)

Teori:

Vi har sett at ligningssett på formen

$$x \equiv a_1 \pmod{b_1}$$
$$x \equiv a_2 \pmod{b_2}$$

har en unik løsning i intervallet $[0, b_1 b_2)$ hvis b_1 og b_2 er relativt primske. I neste oppgave skal vi drøfte hvordan vi kan takle ligningsett hvor vi ikke lenger har antagelsen om at modulusene må være relativt primske.

Et viktig hjelpemiddel vil være følgende implikasjon:

$$x \equiv a \pmod{mn} \implies x \equiv a \pmod m \wedge x \equiv a \pmod n$$

Kinesisk restteorem forteller oss at implikasjonspilen også gjelder andre veien i tilfellet hvor $\gcd(m, n) = 1$.

(Implikasjonspilen gjelder for øvrig generelt ikke andre veien. La for eksempel $n = m = 2$ og $a = 2$. Da er $x = 0$ en løsning til ligningene på høyresiden, men ikke på venstresiden.)

3 a) Skriv ligningen under om til et system av ligninger hvor modulusene er primtallspotenser (Hint: $18 = 2 \cdot 3 \cdot 3 = 2 \cdot 3^2$)

$$x \equiv 1 \pmod{18}$$

b) Forklar hvorfor ligningsettet

$$x \equiv 6 \pmod{25}$$
$$x \equiv 2 \pmod{20}$$

ikke har en løsning.

c) Finn alle løsninger til likningssettet

$$x \equiv 6 \pmod{26}$$
$$x \equiv 2 \pmod{20}$$

d) Beskriv generelt en algoritme for å redusere ligningsettet under til et ligningsett hvor en direkte kan lese av om den har en løsning, og hvor en så kan anvende Kinesisk restteorem

$$x \equiv a_1 \pmod{n_1}$$
$$x \equiv a_2 \pmod{n_2}$$
$$\vdots$$
$$x \equiv a_t \pmod{n_t}$$

Teori:

I neste del av plenumsregningen skal vi diskutere diverse strategier og algoritmer som lar oss effektivt regne med kongruenser på datamaskiner. For det aller første: I mange programmeringsspråk er det å finne rest etter heltallsdivisjon definert som en operasjon. I blant annet C++ og Python skrives denne operatoren % og kalles modulus. Helt presist har vi da at gitt naturlige tall[a] a, b hvor $b \neq 0$ har vi $a \,\%\, b = k$ hvis og bare hvis k er det minste naturlige tallet slik at $n \mid a - k$. Mmuntlig kaller vi gjerne denne operasjonen 'mod'. For eksempel har vi da

```
2 % 1 == 0
5 % 2 == 1
14 % 5 == 4
```

Modulus-operatoren er nyttig når vi vil arbeide med kongruensregning i praksis. La oss først tenke litt på hvordan vi i praksis kan regne med modulus-operatoren. Si at vi ønsker å beregne $(ab) \,\%\, k$, men uten å direkte regne ut ab. Kongruensregning gir oss at

$$ab \equiv ((a \,\%\, k) \cdot (b \,\%\, k)) \pmod{k}$$

Høyresiden er kongruent til $((a \,\%\, k) \cdot (b \,\%\, k)) \,\%\, k$. Vi har derfor

$$(ab) \,\%\, k = ((a \,\%\, k) \cdot (b \,\%\, k)) \,\%\, k$$

Erstatter vi multiplikasjon med addisjon får vi en tilsvarende regneregel:

$$(a + b) \,\%\, k = ((a \,\%\, k) + (b \,\%\, k)) \,\%\, k$$

[a] Vi kan også anvende modulus-operatoren på negative tall, men oppførselen av den er ikke veldefinert helt uten videre. For eksempel må vi bestemme oss om $3 \,\%\, -5$ skal være -2 eller 3. I C++ avhenger oppførselen av implementasjonen til kompilatoren man bruker. For å unngå å drøfte slike detaljer skal vi her kun anvende modulus-operatorene på ikke-negative heltall.

4 Velg $k \in \mathbb{N} \setminus \{0\}$. La oss si at vi ønsker å beregne $f_n \,\%\, k$ hvor f_n er et fibonnaccitall. Si at datasystemet du bruker kun støtter naturlige tall opp til en milliard, altså 10^9. Fibonaccitallene vokser raskt, og vi har allerede $f_{46} > 10^9$. La nå k være et relativt lite tall, si $k < 10^5$. Hvordan kan vi beregne

$$f_{100} \,\%\, k$$

uten å måtte regne med tall som er større enn en milliard?

Teori:

Vi skal nå drøfte hvordan vi kan regne ut *diskrete logaritmer*. Helt konkret skal vi drøfte noen algoritmer som kan brukes til å løse ligninger på formen

$$a^x \equiv b \pmod{p}$$

hvor $a, b, x \in \mathbb{N} \setminus 0$ og p er et primtall.

En enkel måte å finne x på er å teste alle muligheter. Det finnes selvsagt uendelig mange naturlige tall x vi kunne valgt, men vi trenger heldigvis ikke teste alle. Vi vet nemlig at $a^p \equiv a \pmod{p}$, så vi trenger bare lete blant $x \in \{1, 2, 3, 4, p-1\}$.

Naiv algoritme for diskret logaritme: For alle $x \in \{1, 2, 3, 4, \ldots, p-1\}$, beregn $y = a^x \,\%\, p$. Hvis $y = b$, returner x. Når vi implementerer dette kan vi observere at når vi skal regne ut $a^x \,\%\, p$ kan vi bruke at vi allerede har regnet ut $a^{x-1} \,\%\, p$. Vi kan enkelt gange dette siste utrykket med a og så ta modulus-operatoren.

```
def discrete_log_naive(a,b,p):
        y = 1
        for x in range(1,p):
                y *= a;
                y = y % p;

                if (y == b):
                        return x;
```

Denne algoritmen kan kalles en *brute force* algoritme, fordi vi med ren regnekraft sjekker alle mulige x og returnerer den riktige. Når p er stor kan denne algoritmen ta en del tid.

Når vi snakker om regnekraft er det vanlig å telle antallet operasjoner som blir gjort. I dette tilfellet har vi at antaller operasjoner vi gjør er omtrent proposjonalt til p. Vi sier derfor at denne naive algoritmen har kjøretid p.

Videre skal vi diskutere en raskere algoritme for diskrete logaritmer, nemlig *Shanks algoritme* (også kalt *Baby-step giant-step algoritmen*).

5

a) Vi skal her lete etter en løsning til ligningen $3^x \equiv 19 \pmod{59}$. Kan du finne en slik x gitt tallene regnet ut under?

$$3^0 \cdot 19 \equiv 19 \pmod{59}$$
$$3^1 \cdot 19 \equiv 57 \pmod{59}$$
$$3^2 \cdot 19 \equiv 53 \pmod{59}$$
$$3^3 \cdot 19 \equiv 41 \pmod{59}$$

og

$$3^4 \equiv 22 \pmod{59}$$
$$3^8 \equiv 12 \pmod{59}$$
$$3^{12} \equiv 28 \pmod{59}$$
$$3^{16} \equiv 26 \pmod{59}$$
$$3^{20} \equiv 41 \pmod{59}$$

b) Vi kan kopiere teknikken over til å regne ut diskrete logaritmer generelt: Si at vi ønsker å løse en ligning $a^x \equiv b \pmod{p}$. Om vi har $a^{ki} \equiv a^c \cdot b$ så har vi $a^{(ki-c)} \equiv b$, altså $x = ki - c$. Dette er observasjonen som lar oss løse forrige deloppgave.
Hvis vi velger oss en $k \in \mathbb{N}$ kan vi først regne ut $a^i \cdot b \% p$ for alle $i \in \{0, 1, 2 \ldots, k\}$ og så regne ut $a^{kj} \,\%\, p$ for $j > 0$ til vi finner en verdi vi har sett i den forrige utregningen vår. Om dette ikke skjer innen p/k steg finnes det ingen løsning. Denne metoden bruker rundt regnet opp til $k + p/k$ operasjoner.
Hva er den beste k vi generelt kan velge, slik at vi får færrest mulig nødvendige beregninger? Hvordan kan vi være sikker på at algoritmen finner en løsning, gitt at en eksisterer?

c) (Utfordring) Implementer Shanks algoritme (algoritmen diskutert over) i Python. (Hint: Se gjerne notat fra pensum. Valg av datastruktur til å lagre beregningene i første fase, nemlig $x^i \cdot b \,\%\, p$ er viktig. Det kan være lurt å bruke en *dictionary* d, hvor du under nøkkelen $x^i \cdot b \,\%\, p$ lagrer i, altså $d[x^i \cdot b \,\%\, p] = i$. Da kan du få kjøretid omtrent $(k + p/k) \cdot \log k$