I detta projekt ska varje grupp skicka till övningsledaren ett RSA-krypterat meddelande tillsammans med en digital signatur.

1. Meddelandet borde bekräfta gruppens identitet och bestå av 11-15 symboler text (inga siffror !) inklusiva mellanslag. Fyll i med mellanslag i slutet av meddelandet, om det behövs, för att få 15 symboler totalt.

2. Dela upp ditt meddelande i tre blockar av 5 symboler var. Låt oss kalla blockarna för A,B och C. Kodera varje block som ett tal mellan 0 och 24299999, med samma metod som i Exempel 3.42 i boken 'Aritmetik och Algebra'. I detta läge har du alltså tre sådana tal, som vi ska kalla för A^*, B^* och C^*.

3. Den första delen av det ni skickar till oss ska nu bestå av tre tal A^{**}, B^{**} och C^{**} som är RSA-krypterade versioner av talen A^*, B^* och C^* resp.

Kryptera med vår publika nyckel

$$(n_A, s_A) = (590678437223029, 1000951).$$

4. Den andra delen av ert meddelande ska utgöra en digital signatur.

(a) Först måste ni hitta och skicka till oss er publika nyckel (n_B, s_B). Följ följande procedur för att hitta er nyckel :

(i) Välj nummerlappan av en person i gruppen. På lappan står ett par av tal $(k, 121-k)$, där $1 \leq k \leq 59$. Skicka också dessa tal till oss, förresten.

(ii) Hitta nu, med hjälp troligtvis av någon form av matematisk mjukvara (se nedan) primtalen p_B, q_B där

$$p_B = k\text{:te primtal efter } 24300000,$$
$$q_B = (121-k)\text{:te primtal efter } 24300000.$$

(iii) Beräkna då ert

$$n_B = p_B q_B.$$

(iv) Välj som ert s_B något 7-siffrigt tal som är relativt prim med n_B.

(b) För att kunna skriva en digital signatur, måste ni också beräkna talet $t_B \in \{0, ..., n_B - 1\}$ så att

$$t_B s_B \equiv 1 \pmod{n_B}.$$

När ni har gjort detta, tillämpa metoden på sidan 56 i boken till triplet (A^*, B^*, C^*) för att skapa en digital signatur $(A^{***}, B^{***}, C^{***})$. Skicka dessa tre tal till oss.

Algoritmer

För att kunna klara av uppgiften, måste ni ha algoritmer för att genomföra följande typer av procedurer :

(I) primalitetsprovning (behövs i Steg 4(a,ii)).

(II) beräkningen av SGD av två tal (behövs i Steg 4(a,iv)).

(III) beräkningen av $a^{-1} \pmod{n}$, där $\text{SGD}(a, n) = 1$. Detta behövs i Steg 4(b).

(IV) beräkningen av $a^b \pmod{n}$, där a, b, n kommer att vara tre stora tal. Detta behövs i Steg 3 och Steg 4(b).

Angående (I), finns det massor av olika algoritmer i den matematiska litteraturen för att testa om ett givet naturligt tal n är prim eller inte. Den enklaste algoritmen är att för varje tal $a \leq \sqrt{n}$ testa med division algoritmen om $a|n$. Då är n ett primtal omm inga delare hittas.

Denna algoritm är hopplöst ineffektiv för stora tal n (i realisktiska kryperings situationer, är $n \approx 10^{200}$). Tyvärr har vi inte tiden i denna kurs för att beskriva någon bättre algoritm. Därför uppmuntras ni att använda här någon matematisk mjukvara (se nedan) som implementerar en sådan mer effektiv algoritm som en enda funktion.

Algoritmen för (II) är Euklides algoritm. Det är lätt att skriva ett program som implemanterar detta och alla matematiska mjukvaror har den som en enda funktion (se nedan).

Algoritmen för (III) är Euklides algoritm baklänges. Lite svårare att implementera än Euklid själv, men inte med mycket. Flera matematiska paket

har en funktion som gör det (se nedan).

Algoritmen för (IV) är den s.k. 'square and multiply' algoritmen. Finns paket som implementerar den som en enda funktion (se nedan). För ni som är intresserad av att skriva era egna program, låt mig beskriva algoritmen :

Steg 1 : Skriv b i bas 2. Den lätta algoritmen för att göra detta har vi sett i avsnitt 2.6 av boken 'Algebra och Geometri'. Om $2^k \leq b < 2^{k+1}$, då får vi en representation

$$b = \sum_{i=0}^{k} \epsilon_i 2^i,$$

där varje ϵ_i är 0 eller 1.

Steg 2 : Beräkna successivt

$$\begin{aligned} a_0 &:= a \pmod{n}, \\ a_1 &:= a^2 \pmod{n}, \\ a_2 &:= a^{2^2} \pmod{n}, \\ a_3 &:= a^{2^3} \pmod{n}, \\ &\cdot \quad \cdot \quad \cdot \\ &\cdot \quad \cdot \quad \cdot \\ &\cdot \quad \cdot \quad \cdot \\ a_k &:= a^{2^k} \pmod{n}. \end{aligned}$$

För att undvika jättestora ohanterbara tal, genomför beräkningen av talen a_i med rekursion formeln

$$a_i \equiv a_{i-1}^2 \pmod{n}.$$

På detta sätt blir alla tal som uppstår under beräkningen mindre än n^2.

Steg 3 : Vi har nu att

$$a^b \equiv \prod_{i=0}^{k} a_i^{\epsilon_i} \pmod{n}.$$

Beräkna då successivt $A_0, A_1, ..., A_k$ där

$$A_j := \prod_{i=0}^{j} a_i^{\epsilon_i} \pmod{n}.$$

För denna beräkning, använd rekursion formeln

$$A_j \equiv a_j^{\epsilon_j} A_{j-1} \pmod{n}.$$

En gång till, på detta sätt blir alla tal som uppstår under beräkningen mindre än n^2.

Mjukvaror

Uppgiften kan genomföras i antingen *mathematica* eller *maple*. Dessa program kan aktiveras i din mdstud konto med kommandet *rcopt*. Både två har funktioner som implementerar de flesta av de fyra algoritmerna ovan.

MATHEMATICA :

(I) PrimeQ[n];

svarar 'true' om n är ett primtal, annars 'false'.

(II) GCD[a, b];

beräknar SGD av talen a och b.

(III) Kan inte hitta något, tyvärr.

(IV) PowerMod[a, b, n];

implementerar 'square and multiply' algoritmen för att beräkna $a^b \pmod{n}$.

MAPLE har följande motsvarande funktioner :

(I) isprime(n);

(II) gcd(a, b);

(III) msolve($b * a = 1, n$);

(IV) $a\ b \bmod n$;