AlCo5

Bart van den Aardweg, Robert Jan Schlimbach, Steven Raaijmakers

October 2015

1. Zie priem.py

   (a) $n = 97873562348712697823 = 9,7873562 \cdot 10^{19}$

   (b) $\phi(n) = 97873562328925539528 = 9,7873562 \cdot 10^{19}$

   (c) $e = 7$. Dit is het eerste priemgetal waarvoor geld dat $ggd(\phi, e) = 1$

   (d) $d = 83891624853364748167 = 8,38916248534 \cdot 10^{19}$.

   (e) We kiezen $|M_i| = 1$ waardoor we elke letter gemakkelijk kunnen omzetten naar Ascii. Dit versleutelen we vervolgens via $M_{\text{versleuteld}} = {M_{\text{ascii}}}^e \bmod n$. Dit kunnen we later weer ontsleutelen door: $M_{\text{ontsleuteld}} = {M_{\text{versleuteld}}}^d \bmod n$.

M	M_{ascii}	$M_{\text{versleuteld}}$	$M_{\text{Ontsleuteld}}$
H	72	10030613004288	72
e	101	107213535210701	101
l	108	171382426877952	108
l	108	171382426877952	108
o	111	207616015289871	111
	32	34359738368	32
W	87	37725479487783	87
o	111	207616015289871	111
r	114	250226879128704	114
l	108	171382426877952	108
d	100	100000000000000	100

   Bij het ontsleutelen hebben we berekeningen in vormen van: $10030613004288^{83891624853364748167} \bmod 97873562348712697823$. Dit konden we uitrekenen door gebruik te maken van een algoritme die Modular Exponentiation kon oplossen. Hiermee konden we controleren dat het antwoord bij "H" inderdaad 72 was.

   (f) Wanneer dat het geval is zal er te snel een herhaling op te merken zijn tussen letters.

   (g) Wanneer we voor $g = 2$ nemen, ontstaat via Diffie-Hellman de sleutel $x = 7809436859$.

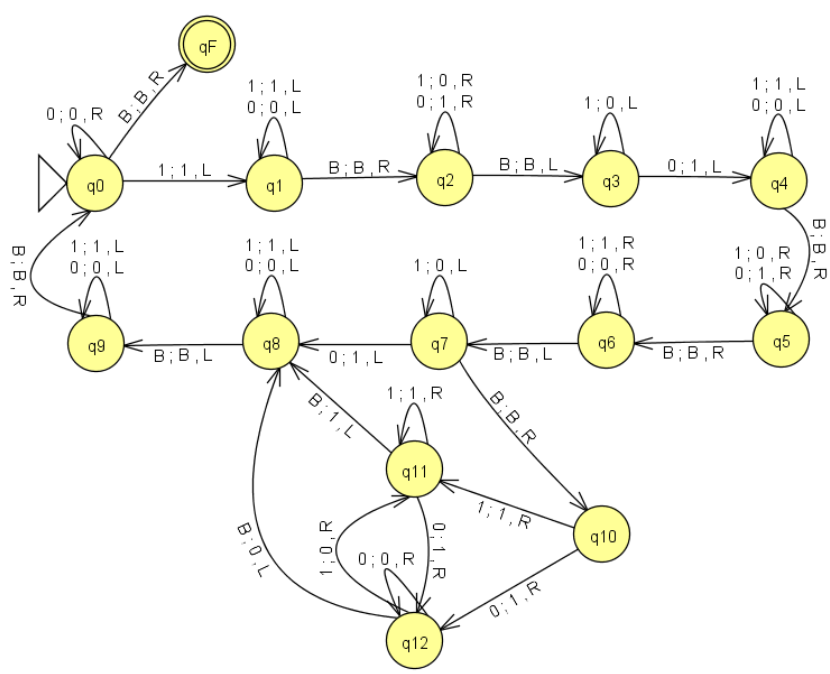
2. • Turing machine: (test de turing machine) voor de turing machine moet de tape als volgt geïnitialiseerd worden: ...BxByB..., met B blank en x en y de binaire getallen om op te tellen, dus bijvoorbeeld: ...B11B11B... Alles hiervoor en hierna zijn ook B's. De Turing machine is in 5 stappen op te delen:

   – Check of x gelijk is aan 0 (q_0 en q_1): loop van links naar rechts door alle getallen van x, als B bereikt is zonder 1 te vinden dan is x gelijk aan 0 en moet de machine stoppen, het antwoord staat dan op de plaats van y. Zodra er een 1 gevonden wordt, ga dan naar de volgende stap.

   – Trek 1 af van x (q_2 t/m q_5): doe dit door one's complement te nemen van x, dan 1 erbij op te tellen en vervolgens weer one's complement te nemen. Beweeg de kop naar het begin van y.

   – Tel 1 op bij y (q_6, q_7 en q_{10} t/m q_{12}): ga naar het meest rechter getal, loop dan van rechts naar links en verander telkens elke 1 in een 0. Zodra er een 0 gevonden is, verander deze in een 1 en ga naar de volgende stap. Als B bereikt is, dan is er geen 0 gevonden en moet het hele getal y 1 plaats opgeschoven worden naar rechts. Het eerste (meest linker getal) wordt een 1.

   – Beweeg helemaal van ergens in y naar het meest linker getal van x, dus tot het bereiken van een tweede B (q_8 en q_9). Ga weer naar de eerste stap.



- Random Access Machine: de registers moeten als volgt geïnitialiseerd worden (om het makkelijker te schrijven is in het vervolg $k = |x| + |y| + 1$):
  - Register 0 is leeg
  - Registers 1 t/m $|x| + |y| + 1$: eerst de bits van x (meest linker bit in register 1), dan # of wat dan ook, dan de bits van y.
  - Register k + 1: waarde 0
  - Register k + 2: waarde 1
  - Register k + 3: waarde 2
  - Register k + 4: waarde $|x|$
  - Register k + 5: waarde $|y|$

  De machine werkt door eerst alle bits van x en daarna van y van binair naar decimaal te converteren en bij elkaar op te tellen. De getallen $|x|$ en $|y|$ moeten ingevuld worden in de registers, anders kan niet bepaald worden waar y eindigt. Dit kan anders niet, omdat de RAM start met alle ongebruikte registers op 0 en het einde van y kan een 0 of een 1 zijn. Met de gegeven instructies JGTZ en JZERO kan de machine niet bepalen of y gestopt is en hij zou eindeloos doorgaan.

```
1:  LOAD    k + 4   // Waarde |x|
2:  STORE   0       // Laad |x| in Register 0

// start: wordt steeds aangeroepen bij een nieuw bit,
// k + 6 telt de macht van 2,
// als een bit 0 is ga dan naar de volgende,
// register 0 wijst naar het register van het bit.
3:  LOAD    k + 2   // Waarde 1
4:  ADD     k + 6
5:  STORE   k + 6   // Register k + 6 += 1
6:  LOAD    #0
7:  JZERO   25      // Jump naar next als Register(inhoud Register 0) == 0
8:  LOAD    k + 6
9:  STORE   #0      // Register(inhoud Register 0) = Register k + 6

// loop_1: als we bij het eerste bit zijn,
// tel dan 1 bij het tussenresultaat (k + 8) op
// en ga naar het volgende bit.
10: LOAD    k + 7
11: JGTZ    16      // Jump naar loop_2 als Register k + 7 > 0
12: LOAD    k + 2   // Waarde 1
13: ADD     k + 8
14: STORE   k + 8   // Register k + 8 += 1
15: JUMP    25      // Jump naar next

// loop_2: we zijn niet bij het eerste bit,
```

```
// dus er moet nu k + 6 keer een 2 opgeteld
// worden bij het tussenresultaat.
// Als dat gebeurt is ga dan naar het volgende bit.
16: LOAD    #0
17: JZERO   25      // Jump naar next als Register(inhoud Register 0) == 0
18: LOAD    k + 3   // Waarde 2
19: ADD     k + 8
20: STORE   k + 8   // Register k + 8 += 2
21: LOAD    #0
22: SUB     k + 2   // Waarde 1
23: STORE   #0      // Register(inhoud Register 0) -= 1
24: JUMP    16      // Jump naar loop_2

// next: als het eerste bit net is opgeteld,
// geef dat dan aan in de registers
// (k + 7 = 1 en k + 6 -= 1).
// Als we bij het tweede getal (y) zijn,
// gebruik dan een offset voor het volgende bit.
// Als er geen volgende bits meer zijn,
// ga dan naar het volgende getal als dat er is (next_number).
25: LOAD    0
26: SUB     k + 2   // Waarde 1
27: STORE   0       // Register 0 -= 1
28: LOAD    k + 7
29: JGTZ    34      // Dit stukje hoeft alleen voor het eerste bit
30: LOAD    k + 2   // Waarde 1
31: STORE   k + 7   // Register k + 7 = 1
32: LOAD    k + 1   // Waarde 0
33: STORE   k + 6   // Register k + 6 = 0
34: LOAD    k + 9
35: JGTZ    39      // Jump naar next_y als Register k + 9 > 0
36: LOAD    0
37: JGTZ    3       // Jump naar start als Register 0 > 0
38: JUMP    43      // Jump naar next_number

// next_y: tel een offset |x| + 1 op om de juiste registers
// voor y te krijgen, ga dan verder met bits optellen.
// Als de bits op zijn, ga dan naar het volgende getal,
// als dat er is (next_number)
39: LOAD    0
40: SUB     k + 4   // Waarde |x|
41: SUB     k + 2   // Waarde 1
42: JGTZ    3       // Jump naar start als inhoud Register 0 - |x| - 1 > 0

// next_number: als y is opgeteld, dan zijn we klaar.
// Laat anders register 0 naar het eerste bit van y wijzen,
```

```
// zet de registers k + 6 en k + 7 weer op 0,
// en begin met het optellen van de bits van y.
43: LOAD    k + 9
44: JGTZ    55       // Jump naar end als Register k + 9 > 0
45: LOAD    k + 2    // Waarde 1
46: STORE   k + 9    // Register k + 9 = 1
47: LOAD    k + 4    // Waarde |x|
48: ADD     k + 2    // Waarde 1
49: ADD     k + 5    // Waarde |y|
50: STORE   0        // Register 0 = |x| + 1 + |y|
51: LOAD    k + 1    // Waarde 0
52: STORE   k + 6    // Register k + 6 = 0
53: STORE   k + 7    // Register k + 7 = 0
54: JUMP    3        // Jump naar start

// end: zet het tussenresultaat k + 8 in register 0 en stop.
55: LOAD    k + 8
56: STORE   0        // Eindresultaat in register 0
57: HALT
```

3. Stel dat er een Turing Machine M_P is die voor een gegeven Turing Machine M kan beslissen of deze halt wanneer er een priemgetal wordt ingevoerd. M_H is een Turing Machine die een Turing Machine als input neemt die op zijn beurt input x heeft:

$$M_H(TM(x)) = \begin{cases} 1 & \text{als TM halt op x} \\ 0 & \text{Anders...} \end{cases}$$

M_P wordt omschreven op de volgende manier:

$$M_P(TM(x)) = \begin{cases} 1 & \text{als TM halt alleen op priemgetallen} \\ 0 & \text{Anders...} \end{cases}$$

Vervolgens kan M_H geschreven worden in de termen van M_P. M_H construeert een Turing machine TM_P door TM en x te gebruiken. TM_P zal halten wanneer de input een priemgetal is en zal anders loopen. De acceptende staat van TM_P verbinden we vervolgens met de initiële staat van een Turing Machine die halt op de priemgetallen. TM_P zal loopen als TM niet op x zal halten (of de invoer geen priemgetal is). De uitvoer van deze Turing Machine is dus gelijk aan de uitvoer van TM_P. Aan TM_P stoppen we vervolgens in M_P. De uitvoer van $M_P(TM_P)$ is gelijk aan die van $M_H(TM(X))$. Het construeren van TM_P kan met een Turing

Machine, waardoor we het HP hebben opgelost. Omdat het HP echter geen oplossing heeft kan deze Turing Machine dus ook niet bestaan:

$M_P \rightarrow M_H$

$\neg M_H$

$\therefore \neg M_P$

4. (a) De maximale hoeveelheid stappen die een turingmachine kan hebben op een enkelzijdige tape van 35 regels lang met een alphabet met daarin {0,1} is 2^{35}, namelijk het tellen van 0 tot en met 2^{35}.
   Dit getal is een ongelofelijke 34359738368. Doordat er 10 stappen zijn, minus q_f, kan de turingmachine bij elke keer dat hij tot 34359738368 heeft geteld overgaan naar een niewe stap. Dit betekent dat de turingmachine hoogstens bij de laatste stap kan ontdekken dat hij de serie niet zal volbrengen. Dit betekent dus dat de turingmachine pas bij de een na laatste stap kan ontdekken dat er een fout in de serie zit en dat hij deze dus niet zal volbrengen. Dit betekent dat de machine pas bij 9 * 34359738368 ontdekt dat het niet lukt. Dit is 309237645312.

   (b) We nemen als invoer langs de horizontale as, de bitrepresentatie s_i van alle Turingmachines die in n^2 ruimte voor alle mogelijke invoer in die ruimte een taal L accepteren. Langs de horizontale as de Turingmachines M_i met ruimte $\log n$. Op de diagonaal staat een 0 als machine M_i invoer s_i accepteert, en een 1 als machine M_i invoer s_i verwerpt. De machine die niet in de rij kan voorkomen is de machine die niet niet op invoer si verwerpt als er een 0 op plaats (i, i) staat en wel accepteert als er een 1 op plaats (i, i) staat. Er is dan geen Turing machine in ruimte $\log n$ die de taal L accepteert, terwijl er wel Turingmachines zijn in ruimte n^2 die de taal L accepteren.

5. Voor elke taal L is er dus een algoritme dat L in polynomiale tijd $p(|x|)$ accepteert. Het algoritme kan in 1 tijdsstap 1 stukje geheugen aanroepen, dus het kan maximaal $p(|x|)$ geheugen gebruiken. Het bekijkt dus ook alleen certificaten van $p(|x|)$. Het checkt alle certificaten en dat kan in $p(|x|)$ dus in polynomiaal veel geheugen. Dus is er voor elke taal in NP een machine die de taal accepteerd in polynomiaal veel geheugen.

   (a) Er zijn n in- en uitgangen, dus de lengte van de oplossing is $O(n)$. Om de oplossing te controleren moeten de n uitgangen van de mogelijke oplossing berekend worden, dit kan in het aantal poorten (een of andere constante) maal n, dus $O(n)$. De oplossing is van polynomiale lengte en de oplossing checken kan in polynomiale tijd, dus het probleem is NP.

   (b) De oplossing is van de lengte $O(V)$. Om een mogelijke oplossing te controleren kan een algoritme gebruikt worden om de maximum flow te vinden, bijvoorbeeld Dinic's blocking flow algoritme, dat vind de max flow in $O(V^2E)$ met V het aantal knopen en E het aantal kanten. Verder moet de max flow dan nog vergeleken worden met k. De oplossing controleren lukt dus in $O(V^2E)$. De oplossing is van polynomiale lengte en de oplossing checken kan in polynomiale tijd, dus het probleem is NP.

(c) De oplossing heeft als lengte $O(k)$. De oplossing kan gecontroleerd worden door voor elke persoon die in gesprek is (dit zijn er $2k$), te kijken of er een persoon in de buurt is die ook in gesprek is (behalve de gesprekspartner). Dus per $2k$ personen moeten maximaal $N - 1$ personen gecheckt worden, min 1 keer het aantal personen dat al gecheckt is. Dit kan dus in $O(N!)$. De oplossing is van polynomiale lengte en de oplossing checken kan in polynomiale tijd, dus het probleem is NP.