PORADNIKI

Algorytm Szyfrowania RSA

CZĘŚĆ I

NOTACJA

Z : Zbiór liczb całkowitych
Z^+ : Zbiór dodatnich liczb całkowitych
a|b : a dzielone przez b
gcd(a,b) : Największy wspólny dzielnik a i b
O : Notacja Big-O
[x] : Funkcja Entier (największej liczby całkowitej)
a==b : a przystaje do b
a^b=a^b.

Definicje

Podzielność : Biorąc pod uwagę liczby całkowite a i b, a jest podzielne przez b, lub b jest podzielne przez a jeśli istnieje liczba całkowita d , taka ,że b = ad, i może być zapisane jako a|b.
Np 3|6 ponieważ 6/3 = 2 lub 6 = 2*3

Podstawowe twierdzenie arytmetyki: Dowolna liczba całkowita n, może być zapisana jednoznacznie (z wyjątkiem kolejności współczynników) jako iloczyn liczb pierwszych.

$n = p_1^{a1} * p_2^{a2} * \ldots * p_n^{an}$, n ma (a1+1)*(a2+1)*...*(an+1) różnych dzielników. Np $18 = 2^1 * 3^2$. Całkowita liczba dzielników dla 18 to (1+1)(2+1)=6,mianowicie 3,9,6,18,1,2

gcd(a,b):Dane są dwie niezerowe liczby całkowite a i b, ich gcd jest to największa liczba całkowita d taka ,ze d|a i d|b. Uwaga: d jest również podzielna przez dowolną liczbę całkowitą, która dzieli się zarówno przez a i b. Np gcd(30,15) = 15, 15|30 i 15|15, 15 jest podzielne przez dowolną liczbę całkowitą, która dzieli się przez (30,15). Widzimy ,że 5|30 i 5|15, co oznacza ,że 15 powinno być podzielne przez 5, co jest prawdą.

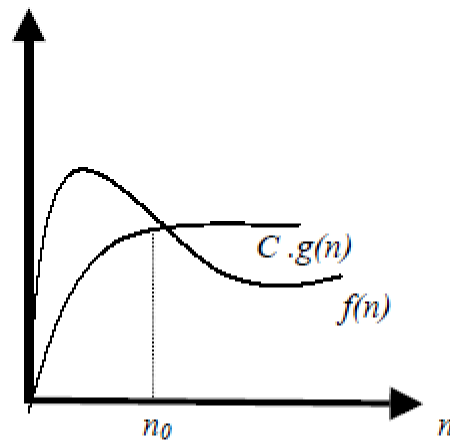
CZĘŚĆ II

Matematyczne podstawy

Notacja Big-O

Funkcja f(n) = *O*(g(n)) lub f=*O*(g), jeśli istnieje stałe c, n_0, takie ,że f(n) <= C.g(n) dla wszystkich n >= n_0.
Poniższy rysunek pokazuje wzrost funkcji f(n) i g(n). Dla n>=n_0, widzimy ,że f(n) <=C.g(n) tj. f(n) jest ograniczana przez C.g(n) od góry. Obserwujemy również ,że wykres jest w pierwszej ćwiartce,

a zatem wszystkie wartości n są dodatnie



Spójrzmy na prosty przykład ilustrujący to pojęcie

Np. $f(n) = (n+)^2$
$= n^2+2n+1$ ->(1)
$\le n^2+2n^2$
$\le 3n^2 = C.g(n)$ gdzie C=3 a $n_0 = 2$
Dlatego górną granicą jest $O(n^2)$
Spójrzmy ponownie na (1)

n^2+2n+1
$\le n^3+2n^2$
$\le 3n^2 = C.g(n)$ gdzie C=3 a $n_0 = 1$

Tu górną granicą jest $O(n^3)$
Która jest poprawną granicą $O(n^2)$ czy $O(n^3)$? Obydwie granice są poprawne. Jednak $O(n^2)$jest bliższe rzeczywistej górnej granicy, zatem wybieramy $O(n^2)$jako górną granicę dla powyższego przykładu. Złożoność jest najważniejszym czynnikiem, który decyduje o czasie pracy algorytmu.

Zasady dla dodwania binarnego
Niech a oznacza pierwszy bit, b drugi bit , c oznacza przeniesienie, s oznacza rozwiązanie, wtedy-

Jeśli

a=0, b=0, c=0 ; s=0, c=0.

a=0, b=0, c=1 ; s=1, c=0.

a=1, b=0, c=0 ; s=1, c=0.

a=0, b=1, c=0 ; s=1, c=0.

a=1, b=0, c=1 ; s=0, c=1.

a=0, b=1, c=1 ; s=0, c=1.

a=1, b=1, c=0 ; s=0, c=1.

a=1, b=1, c=1 ; s=1, c=1.

Wykonanie tej procedury raz jest nazywane operacją bitową. Dodani dwóch k-bitowych liczb wymaga k bitowych operacji. Exlusive OR jest tym samym co bitowe dodawanie modulo 2 lub dodawania bez przeniesienia. Np Dodajemy m =1010 z k = 101

```
1010 +
101
----
1111
```

Każde dodanie bitu wykonywane jest według jednej z wymienionych zasad.Zatem aby dodać k bitową liczbę przez inną k bitową potrzebujemy k bitowych operacji. Aby dodać 'm' bitową liczbę z 'k' bitową liczbą, m > k, pobierane jest k bitowych operacji. Zauważmy ,że Najbardziej Znaczący Bit (MSB) z 1010 nie ma odpowiedniego bitu ze 101 do dodania. Tu zapiszemy bit MSB z 1010 w rozwiązaniu bez wykonywania innych operacji binarnych

Zasady mnożenia binarnego

Zasady odejmowania binarnego są takie same jak w przypadku logicznej bramki AND

0.0=0
0.1=0
1.0=0
1.1=1

Zilustrujemy to przykładem. Niech m będzie kk bitową liczbą całkowitą a n liczbą l bitową
Np. Mnożymy m = 11101 z n = 1101

```
     11101 * (k)
      1101   (l)
----------
```

```
    11101 (wiersz 1)
  11101   (wiersz 2)
 11101    (wiersz 3)
---------
101111001
```

Drui wiersz dodatkowo nie oblicza 0*1101, ponieważ nie spowodowało by to żadnej różnicy w sumie całkowitej. Zatem po prostu przesuwamy o kolejną pozycję i przenosimy do kolejnego mnożenia. Obserwujemy ,że są końcowe dodatkowe wiersze l. Aby wykonać dodawanie, dodajemy wiersz 1 do wiersza 2. Potem dodajemy tą sumę częściową z kolejnym wierszem itd. Obserwujemy ,że przy każdym kroku dodawania jest końcowe k bitowych operacji, kiedy k>l. Zatem końcowa granica w czasie mnożenia liczby k bitowej prze l bitową = k * l. Jeśli obie są k bitowymi liczbami, wtedy górna granica w czasie mnożenia k z k = k2 bitowych operacji, gdzie k = $[\log_2 m]+1$. [x] jest funkcją największej liczby całkowitej <= x gdzie x należy do zbioru liczb rzeczywistych
Na przykład:
[15/2]=[7.5] = 7
[-7.5] = -8
Zatem $k^2 = O(([\log_2 m]+1)*([\log_2 m]+1)) = O([\log_2 m]+1)^2$

Zasady binarnego odejmowania

0-0=0
1-0=1
1-1=0
0-1=1 z przeniesieniem z kolejnego znaczącego bitu

Np. 10101-10011

10101 –

10011

00010

Jeśli spojrzymy na odejmowanie, zobaczymy ,że binarne odejmowanie ma takie same górne ograniczenie czasowe jak binarne dodawanie, którym jest O(k), gdzie k jesy liczbą bitów w danych wyjściowych.

Zasady binarnego dzielenia
Zilustrujemy zasadę binarnego dzielenia przykładem
Podzielimy m = $(1010)_2$ przez n=$(11111)_2$. Niech q oznacza iloraz a r resztę.

1010| 11111 | q=11

1010

01011 –

1010

0001 $= r$

Niech n będzie k bitową liczbą całkowitą. Każdy krok obejmuje jedno mnożenie i jedno odejmowanie. Mnożenie na każdym kroku zabiera krańcowe k bitowych operacji. $(1010)_2$ zajmuje 4 bity przestrzeni. Zatem, każdy krok odejmowania zajmuje 4 operacje binarne. Są końcowe k kroki odejmowania i pobiejest całkowita ilość operacji bitowych to(4*k)*k

$$= O\left(([\log_2 n]+1) * ([\log_2 n]+1) \right)$$

$$= O\left([\log_2 n]+1\right\}^2$$

$$= O(k^2).$$

jest górną granicą czasu dla binarnego dzielenia

Relacje i klasy równoważności

Jeśli A i B nie są pustymi zbiorami, relacja do A i b jest podzbiorem A*B, iloczynem kartrezjańskim. Jeśli R jest właściwym podzbirem A*B i uporządkowanej pary (a,b) € R, mówimy ,że a jest powiązane z b, reprezentowane przez aRb. Zbiór A nazywa się podzbiorem B jeśli istnieje jeden element w zbiorze B, którego nie ma w zbiorze A. Rozważmy zbiory A = {0,1,2} i B = {3,4,5}. Niech R = {(1,3), (2,4), (2,5)} tj.

1R3
2R4
2R5

Widzimy ,że relacja R'jest mniejsza niż" przechowywana ponieważ
1<3
2<4
2<5

W związku z tym kolejność występowania jest tu ważna. Relacja równoważności jest zwrotna, symetryczna i przechodnia z definicji. Rozkład zbioru jest rozkładem zbioru na podzbiory, tak ,że każdy element danego zbioru staje się elementem pewnego podzbioru a część wspólna jest zbiorem pustym. Oznacza to ,że element nie może pojawić się ponownie w więcej niż jednym podzbiorze. Podzbiory w podziale są nazywane komórkami lub blokami.

Cały rozkład dla zbioru A={1,2} to

{1,2}
{1}, {2}
{2},{1}

Klasa równoważności elementu a € A jest zbiorem elementów A z ktöymi a jest powiązany. Jest to oznaczone przez [a]. Ta notacja nie jest myląca z funkcją największej liczby całkowitej.
Niech R będzie relacją równoważności na zbiorze A ={6,7,8,9,10} definowanym przez R={(6,6)(7,7)(8,8)(9,9)(10,10)(6,7)(7,6)(8,9)(9,8)(9,10)(10,9)(8,10)(10,8). Klasy równoważności to

[6]=[7]={6,7}
[8]=[9]=[10]={8,9,10}
Rozkład {(6,7)(8,9,10)}
Zbiór klas równoważności nazywa się kalsą pozostałości i oznaczone jest przez **Z**/m**Z**. Dowolny zbiór elementów dla klasy pozostałości jest wyliczany modulo m
Na przykład Klasa równoważności dla **Z**/5**Z** to [0],[1],[2],[3],[4], takie ,że

[0]={. . .,-10,-5,0,5,10, . . . }

[1]={. . .,-9,-4,-1,1,6,11, . . . }

[2]={ . . .,-8,-3,2,7, . . . }

[3]={ . . .,-7,-2,3,8, . . .}

[4]={ . . .,-6,-1,4,9, . . .}

Jasne jest ,że dowolny element [0]modulo 5 =0 .Dowolny element [1] modulo 5 = 1 itd

Algortym Euklidesa
Jeśli znamy czynniki pierwszw liczby, łatwo jest znaleźć jej gcd
Np gcd(20,10)
20 = 2*2*5 -> (3)
10 = 2*5 -> (4)

W (3) i (4),wspólnymi czynnikami są {2,5} a ich gcd to 2*5 = 10. Dla dużych liczb 'trudno' jest znaleźć ich czynniki pierwsze. Algorytm Euklidesa jest środkiem dla do znalezienia gcd (a,b) nawet jeśli ich czynniki pierwsze nie są znane. Aby znaleźć gcd (a,b), a>b dzielimy b przez a i zapisujemy iloraz i resztę jak poniżnej:

$a = q_1 * b + r_1$

$b = q_2 * r_1 + r_2$

$r_1 = q_3 * r_2 + r_3$

$r_2 = q_4 * r_3 + r_4$

.....

.....

$r_j = q_{j+2} * r_{j+1} + r_{j+2}$

$r_{j+1} = q_{j+3} * r_{j+2} + r_{j+3}$

$r_{j+2} = q_{j+4} * r_{j+3} + r_{j+4}$

$a = q_1 * b + r_1$

$b = q_2 * r_1 + r_2$

$r_1 = q_3 * r_2 + r_3$

$r_2 = q_4 * r_3 + r_4$

.....

.....

$r_j = q_{j+2} * r_{j+1} + r_{j+2}$

$r_{j+1} = q_{j+3} * r_{j+2} + r_{j+3}$

$r_{j+2} = q_{j+4} * r_{j+3} + r_{j+4}$

Na przykład znajdźmy gcd(2107,896)

$2107=2.896+315$

$896=2.315+266$

$315=1.266+49$

$266=5.49+21$

$49=2.21+7$

Ostatania niezerowa reszta jest gcd. Jeśli popracujemy w górę, zobaczymy ,że ostatani niezerowa resza dzieli się przez wszystkie reszty a i b. Oczywiste jest ,że algorytm Euklidesa daje gcd w skończonej liczbie kroków ponieważ reszty są ściśle malejące z kroku na krok.

Złożoność czasowa algorytmu Euklidesa

Najpierw pokażemy ,że $r_{j+2} < ½\, r_j$.

Przypadek 1 : Jeśli $r_{j+1} < 1/2 r_j$, wyraźnie $r_{j+2} < r_{j+1} <= 1/2 r_j$.

Przypadek 2 : Jeśli $r_{j+1} > 1/2r_j$, wiemy ,że $r_j=1.r_{j+1}+r_{j+2}$. Więc $r_{j+2} = r_j-r_{j+2}$. Więc, mamy $r_{j+2}<1/2r_j$, ponieważ $r_{j+2}>1/2r_j$.

Oznacza to ,że reszta będzie przynajmniej rozpoołowiać się w każdnych dwóch krokach. Zatem, całkowita liczba podzielników końcowych to $2.[\log_2 a]$ gdzie [] jest notacją dla funkcji największej liczby całkowitej Jest to $O(\log a)$ Każe dzielenie nie ma liczby większej niż a. Widzimy ,że dzielenie zabiera $O(\log^2 a)$ operacji bitowych. Zatem całkowity wymagany czas to $O(\log a)*O(\log^2 a) = O(\log^3 a)$ dla znalezienia gcd przy użyciu algorytmu Euklidesa.

Złożony algortym Euklidesa
Jeśli d=gcd(a,b) i a>b, wtedy istnieją liczby całkowite u i v takie ,że d=ua+vb. Znalezienie u i v może być wykonane przy $O(\log 3 a)$operacji bitowych. Widzimy ,że kongruencja au = d(mod b) ma rozwiązanie, ponieważ d=gcd(a,b). Dlategoe też, au/d=1 (mod b/d). Tak więc (au)/d = 1+v.(b/d), gdzie u i v są dowolnymi liczbami całkowitymi z odpowiednim znakiem. Zatem , (au)/d + (bv)/d =1, a zatem d=ua+bv

7=49-2.21

=49-2(266-5.49)

=-2.266+11.49

=-2.266+11(315-266)

=11.315-13.266

=11.315-13(896-2.315)

=-13.896+37.315

=-13.896+37(2107-2.986)

=(37).2107+(-87).896

Złożoność czasowa rozszerzonego algorytmu Euklidesa
Pozostała część będzie połówką w każdym z dwóch etapów. W związku z tym, całkowita liczba dzieleń to przede wszystkim $2.[\log_2 a]$, gdzie [] jest notacją dla funkcji największej liczby całkowitej. Jest to $O(\log a)$.Każde dzielenie ma liczbę większą niż a. Widzieliśmy ,że dzielenie pobiera $O(\log^2 a)$operacji bitowych. Teraz, dla odwrócenia w każdym kroku wymagane jest dodawanie lub odejmowanie, które zabiera $O(n)$ czasu. Dlatego też całkowity czas $= O(\log^3 a) + O(\log a)$, co ponownie daje $O(\log^3 a)$

Zbieżność liniowa
Definicja : Dane są liczby całkowiyte a,b,m i m>0, mówimy ,że a jest zbieżne z b modulo m, zapisane jako a == b mod m, jeśli m dzieli się a-b
Np. 7 == 2 mod 5 ponieważ 5|(7-2)
Również a == 0 mod m , jeśli m | a
Dwie kongruencje z tymi samymi modułami mogą być dodawane,

odejmowane lub mnożone, element przez element jak gdyby były równaniami.

Prawo skracania dla kongruencji stanowi ,że jeśli ac==bc (mod m), wtedy a == b(mod m|d) ,gdzie d =gcd(m,c) >1
Np

Jeśli 1.5 == 3.5 (mod 10), używając prawa skracania możemy zapisać

1==3 (mod 2) ponieważ 5 =gcd(5,10)

Np Jeśli 3.5 == 8.5 (mod 3), nie możemy zastosować prawa skracania ponieważ gcd (5,3) =1 Widzimy ,że 3 (niekongruentne do) 8 (mod 3)

Wzglęndnie pierwsze : Dwie liczby a i b są względnie pierwsze, jeśli gcd (a,b) -1. Np 5,2 są relatywnie pierwsze ponieważ gcd(5,2) =1.

Istnienie odwrotności mnożenia : Elementy Z/mZ, które mają odwrotność mnożenia są tymi, które są relatywnie pierwsze dla m, tj kongruencja ax == 1 mod m ,ma unikalne rozwiązanie (mod m) jeśli gcd(a,m)=1. Dodatkowo, jeśli istnieje odwrotność, może być znaleziona $O(\log^3 m)$ operacjami bitowymi, używając rozszerzonego algorytmu Euklidesa.
Np aby znaleźć x = 52^{-1} (mod 7), tf 52x == 1 (mod 7)
Określamyu gcd (52,7)
52=7.7+3
7=2.3+1 (przez algorytm Euklidesa)

1=7-2.3
= 7-2.[53-7.7]
=(-2).52 + (15).7 (rozszerzony algorytm Euklidesa)
= (u)a + (v)b

Dlatego też , u=-2 jest rozwiązaniem dla kongruencji, mamy u=x=-2(mod 7) = 5(mod 7)
Możemy zweryfikować to sprawdzając
Czy 52*(5) == 1 (mod 7)?
Tak ponieważ 7|(260-1) a 52^{-1}(mod 7) to 5

Czasami wymagane jest rozwiązanie w postaci ax == b mod m. Jeśli gcd(a,m), wtedy równanie będzie miało d różnych rozwiązań. Jeśli d = 1, wtedy mamy unikalne rozwiązanie. Najpierw znajdziemy x_0 takie ,że ax_0 == 1(mod m) jak omówiono powyżej. Potem znajdziemy $x=x_0$*b(mod m), które jest wymagane jako rozwiąanie kongruencji. Np znajdziemy rozwiązanie dla kongruencji 3x == 2(mod 5). Widzimy ,że gcd(3,5) =1. Zatem jest unikalne rozwiązanie dla kongruencji między 0 a 4. Najpierw, znajdziemy rozwiązanie dla 3x0 == 1 (mod 5), odkryjemy ,że x_0 = 2. Dlatego , rozwiąznie dla kongruencji to x=2*2(mod 5) = 4. Zweryfikujemy wynik aby sprawdzić czy 3.4 == 2(mod 5) jest prawdą? Ponieważ 5|10, nasze rozwiązanie jest poprawne.
Oto jeszcze jeden przykład odwrotności

Np. Dla kongruencji 3.x == a (mod 12), ma 3 unikalne rozwiązania między 0 a 11, ponieważ gcd(3,12)=3. Rozważmy przypadek kiedy a=0,3,6 i 9

3x==0 (mod 12)

3x==3 (mod 12)

3x==6 (mod 12)

3x==9 (mod 12),

każda kongruencja ma dokładnie 3 rozwiązania

Index	0	1	2	3	4	5	6	7	8	9	10	11
3x	**0**	3	6	9	**12**	15	18	21	**24**	27	30	33
3x-3		**0**	3	6	9	**12**	15	18	21	**24**	27	30
3x-6			**0**	3	6	9	**12**	15	18	21	**24**	27
3x-9				**0**	3	6	9	**12**	15	18	21	**24**

3x == 0 (mod 12), ma rozwiązanie w indeksie 0,4,8
3x == 3 (mod 12), ma rozwiązania w indeksie 1,5,9
3x == 6 (mod 12) ma rozwiązania w indeksie 2,6,10
3x == 9 (mod 12) ma rozwiązania w indeksie 3,7,11

Z tej tabeli można zaobserwować ,że wyjątkowość rozwiązania wynika ze wzgledu na naturalny sposób ułożenia liczb.

Funkcja totient Eulera (phi(n))
Jeśli n > 1, funkcja totient jest definiowana jako liczby naturalne nie przekraczające n, które są względnie pierwsze dla n
Np

n:	1	2	3	4	5	6	7	8	9	10
Phi(*n*):	1	2	2	2	4	2	6	4	6	4

Niektóre włąściwości phi(n) to:

1. Jeśli n jest liczbą pierwszą, wtedy phi(n) = n-1. Jest tak ponieważ żadna z liczb od 1 do n-1 nie dzieli się przez n
2. phi(mn) = phi(m)*phi(n) , jeśli gcd(m,n) =1
   Np. Wiemy,że 35 = 7*5 a gcd(7,5)=1. Wiemy również ,ze phi(7) =6 a phi(5) =4

   01 02 03 04 **05** 06 ***07*** 08 09 **10**

   11 12 13 ***14*** **15** 16 17 18 19 **20**

   21 22 23 24 **25** 26 27 ***28*** 29 **30**

   31 32 33 34

Oczywiście wszystkie mnożenia 5 mają gcd(35,5) > 1 i są **pogrubione.** Wszystkie mnożenie 7 mają gcd(35,7) > 1 i są ***pogrubioną kursywą.*** Żadna z tych liczb nie jest względnie pierwsza dla 35. Zatem mamay całkowitą 6+4=10 liczb które nie są względnie pierwsze dla 35. Więc jest 34-10= 24 liczb ,które są względnie pierwsze dla 35. Zweryfikujmy, phi(7*5) = phi(7)* phi(5) = 6 * 4 = 24, co pasuje do naszej obserwacji.

Algorytm dla binarnego potęgowania modulo m

W algorytmie szyfrowania RSA, jednym z czasochłonnych kroków jets wyliczenie b^n modulo m. Teraz spójrzmy na wydajny algorytm , który wykonuje to działanie wydajniej

Niech $n_0, n_1, ... n_{k-1}$ oznaczają cyfry binarne n, tj $n=n_0+2n_1+...+2^{k-1}n_{k-1}$. $\{n_j=0$ lub $1;\ 0<=j<=k-1\}$

Krok 1: Ustawiay a =1

Krok 2: Obliczamy $b_1=b^2$ mod m Jeśli $n_0=1$ (a<-b) a pozostaje niezmienione

Krok 3: Obliczamy $b_2=b_1^2$ mod m. Jeśli $n_1=1$ (mnożymy a przez b_1 mod m) utrzymujemy jeszcze a niezmienione

Krok 4: Obliczamy $b_3=b_2^2$ mod m .Jeśli $n_2=1$ (mnożymy a przez b_2 mod m) utrzymujemy jeszcze a niezmienione

.

.

.

Krok n : W kroku j^{tym} mamy obliczyć $b_j==b^{(2\^j)}$ mod m .Jeśli $n_j=1$ (mnożymy a przez b_j mod m)i utrzymujemy jeszcze a niezmienione. Po kroku (k-1) mamy pożądany rezultat a == b^n mod m. Na przykład obliczmy 5^6 mod 7

Wiemy ,że $n=6=(110)_2$.

$b_1 = 5^2$ mod 7=4; $n_0=0$, a=1

$b_2 =4^2$ mod 7=2; $n_0=1$, a=1*2=2

$b_2=2^2$ mod 7=4 n2=1,a=2*4=8 mod 7 =1

Tak więc mamy a=1, co implikuje 5^6 mod 7=1

Złożoność czasowa dla binarnego potęgowania modulo m:

Niech $k=\log_2 m$, $l=\log_2 n$

Wartość b jest zawsze mniejsza niż m ponieważ wartość ta jest zawsze redukowanym modulo m. Dlatego też, obliczenie b_2 zabiera $O(k^2)$operacji bitowych. Aby znaleźć wynik kwadratu modulo m. Weźmiey inne dzielenie ,które obejmuje przede wszystkim $O(2k-1)^2$ operacji bitowych. Jest tak ponieważ jeśli pomnożymy k bitową liczbę całkowitą przez inna k bitową liczbę całkowitą, ich iloczyn k*k ma przede wszystkim k+k-1=2k-1 bitów w wyniku. Ponownie mama operację mnożenia jeśli $n_i=1$, które zabierają $O(k^2)$ operacji bitowych. Operacja ta jest powtarzana l razy. Więc całkowity czas to

$$O(l)*[O(k^2) + O(2k-1)^2 + O(k^2)]$$
$$= O(l)* O(k^2)$$

Czas (b^n modulo m)=$O(\log n)$ * $O(\log^2 m)$

Wprowadzenie do teorii pól skończonych

Pole skończone jest to zbiór F z operacjami multiplikatywnymi i addytywnymi, które spełaniają poniższe zasady – łączności i przemienności zarówno dla dodawania i mnożenia, istnienia addytywnje tożsamości 0 i multiplikatywnej tożsamości 1, addytywnej odwrotności i multiplikatywnej odwrotności dla wszystkich liczb z wyjątkiem 0. Pole **Z**/p**Z** liczb całkowitych modulo liczba pierwsza p. Przez odniesienie do "Porządku" elementów, mamy na myśli conajmniej dodatnią liczbę całkowitą modulo p, które daje 1

Multilikatywne generatory pola skończonego w F_p^* są tymi elementami w F_p^* ,które mają maksymalny porządek. Wydaje się,że porządek dowolnego a (element) F_p^* dzielonego przez q-1. Każde skończone pole ma generator. Jeśli g jest multiplikatywnym generatorem F_p, wtedy g^j również jest generatorem jeśli i tylko jeśli gcd(j,q-1) =1. W szczególności, jest phi(q-1) różnych generatorów w generatorach multiplikatywnych F_p^*. Jako przykład, prześledźy generator F_{19}^*. Sprawdzimy czy jeśli 2 jest generatorem w danym polu pierwszym

2^1= =2 mod 19

2^2= =4 mod 19

2^3= =8 mod 19

2^4= =16 mod 19

2^5= =13 mod 19

2^6= =7 mod 19

2^7= =14 mod 19

2^8= =9 mod 19

2^9= =18 mod 19

2^{10}= =17 mod 19

2^{11}= =15 mod 19

2^{12}= =11 mod 19

2^{13}= =3 mod 19

2^{14}= =6 mod 19

2^{15}= =12 mod 19

2^{16}= =5 mod 19

2^{17}= = 10 mod 19

2^{18}= =1 mod 19

Widzimy ,że daje to sekwencję

$\{2,4,8,16,13,7,14,9,18,17,15,11,3,6,12,5,10,1\}$

Obserwujemy ,że zbiór zawiera wszystkie elementy pola pierwszego. Widać również ,że 2 ma maksymalny porządek i jest generatorem w danym polu pierwszym. Jeśli uzyskamy jednen generator w polu pierwszym, łatwo jest znaleźć pozostałe generatory. Zauważmy

$$\gcd(3,9\text{-}1)=1$$

$$\gcd(5,9\text{-}1)=1$$

$$\gcd(7,9\text{-}1)=1$$

Zatem pozostałe genratory w F_9^* to

$$2^3 \bmod 9 = 8$$

$$2^5 \bmod 9 = 5$$

$$2^7 \bmod 9 = 2$$

Jeśli weźmiemy 3 i przestujemy go jako generator w F_9^*.

$$4^1 \bmod 9 = 4$$

$$4^2 \bmod 9 = 7$$

$$4^3 \bmod 9 = 1$$

$$4^4 \bmod 9 = 4$$

$$4^5 \bmod 9 = 7$$

$$4^6 \bmod 9 = 1$$

$$4^7 \bmod 9 = 4$$

$$4^8 \bmod 9 = 7$$

$$4^9 \bmod 9 = 1$$

Widzimy ,że 4 ma porządek 3, ponieważ generuje tylko trzy elementy tego zbioru mianowicie {4,7,1}

Małe twierdzenie Fermata

Niech p będzie liczbą pierwszą. Liczba całkowita a spełnia ^{ap} == a mod p, a liczba całkowita nie jest podzielna przez p spełania a^{p-1} == 1 mod p

Spójrzmy na klasę resztkową **Z/**5**Z [0],[1],[2],[3],[4]** taką ,że

[0]={. . .,-10,-5,0,5,10, . . . }

[1]={. . .,-9,-4,-1,1,6,11, . . . }

[2]={ . . .,-8,-3,2,7, . . . }

[3]={ . . .,-7,-2,3,8, . . .}

[4]={ . . .,-6,-1,4,9, . . .}

Mamy

$$(0*a)*(1*a)*(2*a)*(3*a)*(4*a) == 0*1*2*3*4 (\mathrm{mod}\ 5)$$

gdzie 0,1,2,3,4 są klasami reszt a a jest liczbą całkowitą. Jest tak ponieważ (0*a)*(1*a)*(2*a)*(3*a)*(4*a) jest prostą zmianą 0*1*2*3*4 (modulo 5). Tak więc mamy
a^4*4! == 4!(modulo 5)
Dlatego też, 5|(a^4*4!)-4!
Zatem ,5|4 * (a^4-1)

5 nie może być podzielone przez 4! ponieważ p=5 jest liczbą pierwszą. Więc 5|(a^4-1), co oznacza ,że a^4 == 1 (modulo 5).Pomnożymy obie strony przez 5 i mamy a^5 == a (modulo 5). Dla powiedzmy a=2 wtedy
2^4 == 1 (modulo 5) powinno być prawdą. Mamy 16 == 1 (modulo 5) ponieważ 5|(16-1) jest prawdą. Nasze obserwacje są zgodne z małym twierdzeniem Fermata. Również 2^5 == 2 (modulo 5) jest prawdą po weryfikacji

Twierdzenie Eulera
Jest to uogólnienie małego twierdzenia Fermata. Stanowi ,że dla dwóch liczb całkowitych a i n takich ,że gcd(a,n)=1, $a^{(phi(n)}$ == 1 mod n.
Niech R={x_1,x_2,...,$x_{phi(n)}$}będzie zbiorem liczb całkowitych, które są względnie pierwsze dla n. Pomnożenie każdego elementu R przez a(modulo n), mamy inny zbiór S={ax_1(mod n), ax_2(mod n), ... ,a$x_{phi(n)}$ (mod n)}. Ponieważ a jest względnie pierwsze do n a x_i jest wazględnie pierwsza dla n, Iloczyn z $_{(i=1\ do\ i=phi(n)}$(axi mod n) = Iloczyn z $_{(i=1\ do\ i=phi(n)}$ (x_i mod n)
Dlatego $a^{phi(n)}$*Iloczyn $_{(i=1\ do\ i=phi(n)}$$x_i$ = Iloczyn z $_{(i=1\ do\ i=phi(n)}$(x_i mod n)
Więc mamy $a^{phi(n)}$ == 1 mod n. Pomnożenie obu stron przez a, $a^{phi(n)+1}$ == a mod n

Następstwa twierdzenia Eulera
Jeśli gcd(a,n)=1 i jeśli k jest najmniejszą nie ujemną resztą l modulo phi(n), wtedy a^l == a^k mod m. Mamy l == k mod phi(n) lub l=c*phi(n)+k, dla dowolnej liczby całkowitej c
Znamy $a^{phi(n)}$ == 1 mod n (z twierdzenia Eulera) i

$a^{phi(n)}$ * $a^{phi(n)}$ * ... * $a^{phi(n)}$ ==1*1*...*1 (mod n)
(c razy) (c razy)

$a^{c.phi(n)}$ == 1 mod n. Pomnożenie obu stron przez a^k daje nam

$a^{c.phi(n)+k}$ == a^k mod n
Dlatego, a^1 == a^k mod m
Użyjemy tekj właściwości w algorytmie RSA podczas deszyfrowania tj jeśli e i d będą dwoma liczbami całkowitymi takimi ,że e*d == 1 mod phi(n) a gcd (e, phi(n) = 1 wtedy M^{e*d} ==M^1 mod n gdzie M jest inną dowolną liczbą całkowitą

CZĘŚĆ III

Algorytm szyfrowania RSA

RSA jest algorytmem szyfrowania kluczem publicznym zaprojektowanym przez Rivesta, Shamira i Adlemana. Jego siła leży w ogromanej trudności w faktoryzacji dużych liczb. RSA jest szyfrem blokowym a tekst jawny jest szyfrowany w blokach. Tekst jawny i tekst zaszyfrowany są liczbami całkowitymi między 0 a n-1, dla pewnego n.Niech blok tekstu jawnego będzie przedstawiany przy użyciu k bitowej liczby całkowitej i niech 2^k będzie największą liczbą całkowitą, jaką ten blok może przechować, wtedy $2^k < n$ powinno być prawdą Wartość całkowita jaką może przedstawić tekst jawny musi być mniejsza niż n, w przeciwnym razie wykonywane jest (modulo n), które powoduje ,że proces szyfrowania / deszyfrowania staje się unikatowy.

Krok 1: Znajdujemy dwie liczby pierwsze p i q losowo,kilka cyfr po przecinku (każda o rozmiarze conajmniej 150 cyfr dziesiętnych). Losowo, to znaczy przez generator liczb pseudo losowych. Jeśli daną wyjściową liczby pseudo losowej z jest liczba parzysta, sprawdzamy czy z+1 jest liczbą pierwszą a czy z+3 nie itd. Może to być zrobione przez odpowiedni test pierwszości. Według twierdzenia liczb pierwszych, częstotliwość liczb bliskich z to (1/log z), więc z *O*(z) testami możemy liczbę pierwszą >= z

Krok 2: Obliczamy n=p*q
Krok 3: Teraz wybieramy liczbę losową e (0<e<phi(n)), taką, że e*d == 1 mod phi(n) gcd(e, phi(n))=1. Znajdziemy $d=e^{-1}$ mod phi(n) używając rozszerzonego algorytmu Euklidesa. Ponieważ odwrotność jest unikalna, tj gcd(e, phi(n) =1, jesteśmy pewni ,że istnieje dokładnie jedno rozwiązanie między 0 a phi(n), które spełnia powyższe równanie

Klucz publiczny to :(e,n)
Klucz prywatny to : (d,n)

Krok 4: Niech M będzie tekste jawnym a C tekstem zaszyfrowanym

Szyfrowanie

$$f(M) = C = M^e \bmod n$$

Deszyforowanie

$$f^{-1}(C) = M = M^{ed} \; mod \; n = M^{k*\mathrm{phi}(n)+1} = M.$$

Teraz mamy dwa przypadki

Przypadek 1: Jeśli gcd(M,n) =1 ,wtedy przez następstwa twierdzenia Eulera, M^{e*d} = M mod n, ponieważ e*d =1 mod phi(n)

Przypadek 2: Jeśli gcd(M,n) > 1 a M <n=pq, wtedy M=hp lub M=lq (dla dowolnych liczb całkowitych h i l). Mamy $M^{phi(q)}$ == 1 mod q (twierdzenie Eulera)

Dlatego $M^{k*phi(p)*phi(q)}$ == 1 mod q

lub $M^{k*\mathrm{phi}(n)} == 1 \bmod q$.

lub $M^{k*\mathrm{phi}(n)} = 1 + cq$, (dla dowolnego całkowitego c)

lub $M^{k*\mathrm{phi}(n)+1} = M(1+cq)$ (Pomnożenie obu stron przez M)

lub $M^{k*\mathrm{phi}(n)+1} = M + mcq$

lub $M^{k*\mathrm{phi}(n)+1} = M + (hp)cq$

lub $M^{k*\mathrm{phi}(n)+1} = M + hc(pq)$

lub $M^{k*\mathrm{phi}(n)+1} = M + hc(n)$

lub $M^{k*\mathrm{phi}(n)+1} = M \bmod n$, jak wymagano

Zatem w obu przypadkach mamy poprawne deszyfrowanie

Uwaga: RSA jest podatny na blokowy atak metodą powtórzenia i odpowiedni tryb łączenia taki jak Cipher Block Chain (CBC), które mogą zostać wykorzystane .Wszystkie klasyczne szyfry są podatne na ataki man in the middle.

Przykład algorytmu RSA

Teraza spojrzymy na uproszczony przykład ilustrujący ten algorytm

Niech p=3 a q=11, będą dwoma losowo wybranymi liczbami pierwszymi

n=3*11=33

phi(n) = (3-1)*(11-1) =20

Wybieramy losowo e, takie ,że gcd(e,20)=1. Niech e=7, gcd(20,7)=1.

Zatem istnieje całkowite d takie ,że 7*d=1 mod 20 lub d=7^{-1}20

gcd(20,7)

20=2.7+6

7=1.6+1

Dlatego

1=7-6

=7-(20-2.7)

= -(1).20 +(3).7

Więc d=3
Niech tekst jawny M=2
Wtedy C=2^7 mod 33 =29
a M=29^3 mod 33 =2 jak żądamy

Test pierwszości Millera – Rabina

Według małego twierdzenia Fermata, jeśli b jest względnie pierwsza do n
wtedy bn-1 == 1 mod n -> (5)
gdzie b i n są dodatnimi liczbami całkowitymi a n>0 Jeśli n będzie nieparzystą złożoną liczbą całkowitą a gcd(n,b) =1 a (5) jest prawdą, wtedy jest to nazywane *pseudo liczbą pierwszą* . *Liczba Carmichaela* jest złożoną liczbą całkowitą n spełniając (5) dla każdego b €$(Z/nZ)^*$
Matematycznie zilustrujemy to prostym przykładem
Zbadamy wszystkie generatory w F_7^*.

$2^1=2$	$3^1=3$	$4^1=4$	$5^1=5$	$6^1=6$
$2^2=4$	$3^2=2$	$4^2=2$	$5^2=4$	$6^2=1$
$2^3=1$	$3^3=6$	$4^3=1$	$5^3=6$	$6^3=6$
$2^4=2$	$3^4=4$	$4^4=4$	$5^4=2$	$6^4=1$
$2^5=4$	$3^5=5$	$4^5=2$	$5^5=3$	$6^5=6$
$2^6=1$	$3^6=1$	$4^6=1$	$5^6=1$	$6^6=1$

Patrząc na tą tabelkę, widzimy ,że 2 nie jest generatorem ponieważ generuje tylko połowę liczby elementów danego pola. Podobnie 4 i 6 nie są generatorami. Jedyne generatory to 3 i 6. Jeśli spojrzymy na ostatnie wiersz tablicy, reszta elementów do potęgi n-1^{tej} (n=7-1 tu) jest 1 dla wszystkich przypadków.Właśnie ze względu na małe twierdzenie Fermata. Łatwo zobaczyć ,że jeśli b2 == 1 mod n, wtedy b=(+lub-) 1
Np 2^6=1 implikuje ,że pierwiastek kwadratowy 2^6 będzie (+lub-) 1. Również, 3^6=1 implikuje ,że pierwiastek kwadratowy 3^6 będzie (+lub-) 1. Zobaczymy też ,że jest to prawda ponieważ 3^6=6=-1 mod 7. Podobnie, zobaczymy ,że to jest prawda dla wszystkich innych elementów w tablicy i jest podstawą dla testu Millera – Rabina. Łatwo zobaczyć

Algorytm dla testu Millera-Rabina

Test Millera-Rabina jest to algorytm probabilistyczny.
Krok 1: Wybieramu liczbę całkowitą nieparzystą n>=3 i rozpatrujemy parzystą liczbę całkowitą n-1. Liczba ta może być wyrażona w postaci potęgi 2 razy liczba nieparzysta

$n-1 -2^{k}*q$

tj dzielimy n-1 przez 2 dopóki nie uzyskay liczby nieparzystej q.

Krok 2: Wybieramy liczbę losową a, taką ,że a<n
Krok 3: Jeśli a^{q} mod n=1 (wyświetla"Prawdopodobnie liczba pierwsza")
Krok 4: for j=0 to k-1
if ($a^{(2^j)*q}$ mod n=n-1) return(Prawdopodobna liczba pierwsza)
Krok 5: return(Liczba złożona)

Jeśli test zwraca 'Prawdopodbnie liczba pierwsza' dla t prób, wtedy szansa ,że jest to prawdziwa liczba pierwsza to $1-4^{-t}$. Jeśli t=10, prawdopdobieństwo ,że n jest liczbą pierwszą większą niż 0.99999

CZĘŚĆ IV

Kod w języku Python

```
from math import *
from types import *
from random import random
from sys import stdout as out
from time import time,gmtime,strftime
from base64 import encodestring as b64, decodestring as unb64
# Wszystkie poniższe funkcje są używane do wizualizacji procesu generowania klucza
# kiedy używają interpretera python
_rsa_dsp_sequence = ("|/-\\", '>')
_rsa_dsp_i = 0
_rsa_dsp_t = 0
def rsadsp(d):
global rsa_dsp
rsa_dsp = d

def _rsa_dsp_init():
global _rsa_dsp_t
_rsa_dsp_t = time()
def _rsa_dsp_end():
out.write(strftime(" # keys created in %H:%M:%S\n", gmtime(time()-
_rsa_dsp_t)))
def _rsa_dsp_iter(b=False):
if (b):
out.write(_rsa_dsp_sequence[1])
else:
global _rsa_dsp_i
_rsa_dsp_i += 1
_rsa_dsp_i %= len(_rsa_dsp_sequence[0])
out.write(_rsa_dsp_sequence[0][_rsa_dsp_i]+'\b')
# randrange()nie pracuje dla zbyt dużych zakresów, np. 2048 bitów
def rand(start):
fl = random()
ll = long(fl * (10**17)) # to jest maksymalna precyzja
ll *= start
ll /= (10**17)
```

```
return ll
# zwraca liczbę bajtów w pamięci, które są wymagane do przechowywania danej liczby
# całkowitej long i

def bytelen(i):
blen = 0
while (i != 0):
blen += 1 # one more byte
i >>= 8 # and shift.
return blen
# hexunpack zmienia liczbę całkowita long i na łańcuch pythona, który zawiera tą
# samą liczbę w formacie little endian
def hexunpack(i,l=0):
sval = ""
if not l: l = bytelen(i)
for j in range (l):
ival = i & 0xFF
i = i >> 8
sval += chr(ival)
return sval
# hexpack odczytuje łańuch i interpretuje go jako liczbę long przechowywaną bajt po
# bajcie w formacie little endian i zwraca tą liczbę całkowitą
def hexpack(s,l=0):
hret = 0L
if not l: l = long(len(s))
for i in range(l):
val = long(ord(s[i]))
val = val << long(i*8)
hret += val
return long(hret)
# raw encryption algorithm for RSA keys and
# python strings.
def raw_Encrypt(s, key):
if (type(s) != StringType):
return None
# długośćbajtu modulo części klucza
blen = long(bytelen(key[1]))
# thh pierwsze dwa bajty przechowują długość szyfru blokowego określonego przez
# samą długość klucza
rev = hexunpack(blen,2)
# prosty podpis na końcu naszego łańcucha,
# potem jest dopełniany zerami do długości bloku
# Należy uważać aby nie pominąć żadnych danych,
# będziemy szyfrować bloki (blen-1) bajtów.
s += "\x01"
while len(s) % (blen-1):
s += '\x00'
# wykonujemy rzeczywiste szyfrowanie bloku
for i in xrange(0,len(s),blen-1):
rev += hexunpack(ModExp(hexpack(s[i:i+blen],blen-1), key[0],
key[1]),blen)
return rev
# to jest podprogram deszyfrowania. Działa bardzo podobnie do podprogramu
# szyfrowania.
def raw_Decrypt(s, key):
if (type(s) != StringType):
return None
rev = ""
# wyciągamy długość bloku z pierwszych dwóch bajtów i sprawdzamy czy pozostały
# łańcuch ma poprawną długość
blen, s = hexpack(s[0:2],2), s[2:]
if len(s) % blen: return None
# teraz przechodzimy pętlą przez pozostały łańcuch i deszyfrujemy każdy blok. P
# Pamiętaj ,że szyfrowaliśmy bloki aktualną długością bloku (blen-1) bajtów

for i in xrange(0,len(s),blen):
rev += hexunpack(ModExp(hexpack(s[i:i+blen],blen), key[0],
```

```
key[1]),blen-1)
# znajdujemy podpisna końcu. Wszystkie zera, które wystąpiły w tym podpisie będą
# obcięte. Jednak jeśli nie ma podpisu,  nie jest to łańcuch szyfrowany naszym
# podprogramem szyfrującym i dlatego nasze wyniki są fałszywe
sig = rev.rfind("\x01")
if (sig == (-1)): return None
else: return rev[0:sig]
# Jest to główna klasa modułu rsa.Obiekty rskey są zwracane przez funkcję keypair()
# która generuje dwa pasujące klucze. Obiekt rskey dostarcza mechanizmu do
# szyfrowania i deszyfrowania danych i może być przedstawiony jaki zakodowany
# łańcuch Base64
class rsakey:
# Konstruktor przyjmuje jako pierwszy i jedyny argument już działający klucz
# Klucz ten może być przekazany jako nazwa pliku, łańcuch zakodowany base64
# lub dwuelementowa sekwencja przechowująca decydujące liczby
def __init__(self,keys=None):
self.__key = 0 # najpierw inicjujemy wartości źródłowe zerem
self.__mod = 0
# Jeśli argument klucza jest łańcuchem, najpierw zinterpretujemy ten łańcuch
# jako nazwa pliku i spróbujemy zładować ten klucz z pliku. Jeśli jest to
# niepoprawna nazwa pliku pojawi się wyjątek i możemy założyć ,że łańcuch nie jest
# nazwą pliku ale reprezentacją łańcucha klucza zakodowaną w base64
if type(keys) is StringType:
try: self.load(keys)
except: self.read(keys)
# Jeśli argument jednak nie jest łańcuchem ale sekwencją, możemy bezpośrednio
# spróbować zainicjować nasze wartości źródłowe
elif type(keys) in [ListType,TupleType]:
if (len(keys)!=2): raise ValueError("a valid key consists of 2
integer numbers")
else: self.set(keys[0],keys[1])
# Wszystko inne ,z wyjątkiem wartości None nie jest poprawnym argumentem.

elif type(keys) is not NoneType:
raise ValueError("argument must be a string representation of
the keys or a tuple/list")
# To jest źródłowy podprogram szyfrowania i deszyfrowania. Powinien być rzadko
# wywoływany bezpośrednio, chyba ,że chcesz zaimplementować swój własny mechanizm
# szyfrowani i deszyfrowania
def crypt(self, x):
return ModExp(x,self.__key,self.__mod)
# len(rsakey) will return the length of the key
# in bits. This also equals the block length that
# will be used when encrrpting arbitrary data.
def __len__(self):
return bytelen(self.__mod)*8
# Reprezentacja łańcucha klucza jest tylko zrzutem raw wartości źródłowych
# kodowanych w base64.
def __repr__(self):
return str(self)
def __str__(self):
b = max(bytelen(self.__key),bytelen(self.__mod))
v = hexunpack(self.__key,b) + hexunpack(self.__mod)
return b64(v)
# rsakey.read()będzie odczytywąc reprezentację łańcucha wygenerowanego przez tą
# klasę (see __str__()) i ustawi właściwie wartości źródłowe
def read(self,s):
try: s = unb64(s)
except: raise ValueError("key must be base64 encoded.")
if len(s)%2: raise ValueError("invalid key")
k = s[0:len(s)/2]
m = s[len(s)/2:]
self.set(hexpack(k),hexpack(m))
# Podprogram set może być użyty do ustawienia wartości źródłowej bezpośrednio
def set(self,k,m):
self.__key, self.__mod = k, m
# Podprogramy szyfrowania/deszyfrowania tylko zawijają podprogramy raw, omówione
```

```
# na początku pliku źródłowego
def encrypt(self,s):
return raw_Encrypt(s,[self.__key,self.__mod])
def decrypt(self,s):
return raw_Decrypt(s,[self.__key,self.__mod])

# Funkcja dump() function zrzuca klczu do pliku ASCII przez zapisanie
# reprezentacji łańcucha z  self.__str__() do pliku
#
# Powiązana funkcja load() będzie odczytywała taką reprezentację łańcucha z pliku
# i przekaże ten łańcuch do funkcji read ()dla zainicjowania wartości
def dump(self,filename):
t = open(filename,"w")
t.write(str(self))
t.truncate()
t.close()
def load(self,filename):
return self.read(open(filename,"r").read())
# Dla bardzo dużych kluczy, szyfrowanie i deszyfrowanie danych może być bardzo
# wolne. Dlatego, małe łańcuchy, takie jak hasł lub klucze dla innych
# mechanizmów szyfrowania powinny być szyfrowane przy użyciu funkcji pencrypt i
# mechanisms should be encrypted by using the
# pdecrypt które tylko wywołują działanie ModExp () raz
#
# Z tego powodu, dana która ma być zaszyfrowana jest interprtowana jako jedna
# duża liczba całkowita (bajt po bajcie) a ta pojedyuncza liczba jest
# szyfrowan / deszyfrowana.
def pencrypt(self, s):
i = self.crypt(hexpack(s))
return b64(hexunpack(i))
def pdecrypt(self, s):
i = self.crypt(hexpack(unb64(s)))
return hexunpack(i)
# Funkcja ModExp jest szybszym sposobem wykonania następujących arytmetycznych
# zadań:
#
# (a ** b) % n
def ModExp(a,b,n):
d = 0L
t = 0L
i = 0
n = long(n)
if (b == 0): return (1%n) # easy.
elif (b < 0): return (-1) # error.
else:
d = 1L
i = int(log(b)/log(2))
while (i >= 0):

d = (d*d)%n; t = long(2**i)
if (b&t): d = long(d*a)%n
i -= 1
return d
# Algorytm Millera-Rabina jest używany dla weryfikacji czy
# liczba jest liczbą pierwszą.
def MRabin(number,attempts):
rndNum = 0L
retVal = False
i = 0
if (number < 10):
return Fermat(number);
else:
retVal = True;
for i in xrange(attempts):
rndNum = rand(number-2)
if (rndNum < 2): rndNum = rndNum + 2
if (Witness(rndNum, number)):
```

```
retVal = False
break
return retVal
# funkcja witness jest używana przez algorytm Millera-rabina dla udowodnienia
# ,że liczba NIE jest pierwsza
def Witness(witness,number):
f = 1; x = 0;
t = 0; i = 0;
retVal = False;
i = int(log(number-1)/log(2))
while (i >= 0):
x = f
f = x * x % number
t = 2 ** i
if ((f==1) and (x!=1) and (x!=(number-1))):
retVal = True
break
if (((number-1) & t) != 0):
f = f * witness % number;
i -= 1
if (retVal):
return True
else:
if (f != 1): return True
else: return False

# Fermat jest dużo prostszą i mniej wiarygodną funkcją dla sprawdzania czy liczba
# jest pierwsza czy nie. Czasami daje falszywe rezultaty ale jest dużo szybsza
# not. It sometimes gives false results but is
# niż algorytm Millera-Rabina.
def Fermat(number):
return bool((number==2)or(ModExp(2,(number-1),number)==1))
# Ta funkcja oblicza gcd dwóch liczb
def GCD(a,b):
if (b!=0):
if ((a%b)!=0): return GCD(b,(a%b))
else: return b
else: return a
# Rozszezony algorytm Euklidesa.
def exeu(a, b):
q=0L; r=0L;
x = [0L,0L,0L]
y = [0L,0L,0L]
if not b: return [1,0]
else:
x[2] = 1; x[1] = 0
y[2] = 0; y[1] = 1
while (b>0):
q=a/b
r=a-q*b
x[0]=x[2]-q*x[1];
y[0]=y[2]-q*y[1]
a,b=b,r
x[2]=x[1];x[1]=x[0];
y[2]=y[1];y[1]=y[0];
return [x[2],y[2]]
# Ta funkcja generuje losową liczbę pierwszą przy użyciu powyższego algorytmu
def prime(bytes, init=0L):
i = init
# jeśli już znamy dużą liczbę pierwszą,czasami szybciej znajdziemy "kolejną"
# liczbę pierwszą przez odgadnięcie gdzie zacząć wyszukiwani
if i: i+= long(log(i)/2)
else: i = rand(2**bytes)
if not i%2: i+=1 # chose the first uneven number
# p wymaganą precyzją dla algorytmu Millera-Rabina. Dla dużych liczb, wyższa
# wartość p zapepwnia ,że algortym ten zwróci wiarygodny wynik
p = int(ceil(sqrt(bytes)))*2
```

```
if (p > 40): p = 40
f = False # f is true if i is prime
while not f:
while not Fermat(i): # find a number that might be prime
i += 2
if (rsa_dsp): _rsa_dsp_iter()
if (rsa_dsp): out.write("!\b");
f = MRabin(i,p) # verify that it is prime
if (rsa_dsp): _rsa_dsp_iter(True)
return i # return the prime number
# funkcja keypair zwraca krotkę 2 obiektów rsakey ,które mogą być użyte dla
# szyfrowania kluczem publicznym via RSA. Prametr bitmax paramter określa  w bitach
# długość generowanych kluczy
def keypair(bitmax):
p = 0L; q = 0L;
e = 0L; d = 0L;
n = 0L
bWorks = False;
if (bitmax % 2): bitmax += 1
maxB = 2L ** long(bitmax/2)
if (rsa_dsp): _rsa_dsp_init()
# znajdowanie dwóch dużych liczb pierwszych
p = prime(bitmax/2)
q = prime(bitmax/2, p)
# obliczenie n=p*q i p=phi(n)=phi(p*q)=(q-1)*(p-1)
# ponadto należy usunąć z pamięci liczby pierwsze ponieważ nie są dłużej
# wymagane
n,p = (q*p), (q-1)*(p-1)
del q
while not bWorks:

bWorks = True
# znajdowanie liczby losowej e z gcd(phi(n),e)!=1
# będzie to klucz szyfrujący(klucz publiczny)
e = rand(maxB)*rand(maxB)
while (p/e > 5): e=rand(maxB)*rand(maxB)
while (GCD(p,e)!=1): e+=1
# obliczenie multiplikatywnej odwrotności e  phi(n),to będzie klucz deszyfrujący
# (klucz prywatny)
sum = exeu(p,e)
if ((e * sum[1] % p) == 1): d = sum[1]
else: d = sum[2]
# testowanie kluczy dla sprawdzenia które są poprawne i działające

if ((d>1) and (e>1) and (n<>0) and (e<>d)):
for a in range(4):
ascNum = rand(255)
if rsa_dsp: _rsa_dsp_iter()
cipher = ModExp(ascNum,e,n)
if rsa_dsp: _rsa_dsp_iter()
if (ModExp(cipher,d,n)!=ascNum):
bWorks = False
break
else:
bWorks = False
if rsa_dsp:
_rsa_dsp_iter(True)
_rsa_dsp_end()
e = long(e)
n = long(n)
d = long(d)
return rsakey((e,n)),rsakey((d,n))
rsadsp(True)
if __name__ == "__main__":
e,d = keypair(1024)
print "\nPublic Key:"
print e
```

```
print "\nPrivate Key:"
print d
raw_input()
```