Szyfrowanie RSA

Liczby pierwsze

Na początek przypomnijmy sobie parę użytecznych wiadomości o liczbach pierwszych. Są one znane od starożytności a ich znaczenie jest ogromne – w matematyce i tym bardziej w kryptografii. Dlaczego, o tym przekonamy się już niedługo.
Liczba pierwsza jaka jest każdy widzi. Jednak dla pewności przypomnimy definicje:

Liczba pierwsza jest liczbą naturalną posiadającą dokładnie dwa różne podzielniki - 1 oraz samą siebie.

Zatem nie jest liczbą pierwszą liczba 0 ani liczba 1. Najmniejszą liczbą pierwszą jest liczba 2, a jak dowiódł Euklides, liczb pierwszych jest nieskończenie dużo. Powstaje pytanie, jak sprawdzić czy dana liczba jest liczbą pierwszą, bądź też jak wygenerować liczby pierwsze. Jeśli chodzi o sprawdzenie czy dana liczba jest pierwsza to można posłużyć się bezpośrednio definicją liczby pierwszej, to znaczy sprawdzić czy jakakolwiek liczba w przedziale <2, p-1> (p jest sprawdzaną liczbą) dzieli bez reszty liczbę p. Jeśli tak, to liczba p nie jest liczbą pierwszą. Metoda ta oczywiście działać będzie, natomiast łatwo zauważyć, że dla dużych liczb wykonywać się będzie wiele niepotrzebnych operacji dzielenia. Wynika to z faktu, że biorąc pod uwagę dowolną liczbę naturalną, mogą zaistnieć 4 przypadki:

1. Liczba p jest pierwsza i w całym przedziale od 2 do p-1 nie posiada dzielnika.
2. Liczba p posiada pierwiastek będący jej podwójnym dzielnikiem, np. 25 = 5 x 5
3. Pierwiastek z p nie jest liczbą pierwszą, a jeden z dzielników jest większy od pierwiastka z p. Wtedy wszystkie inne dzielniki musza być mniejsze od pierwiastka z p. np. 22 = 2 x 11
4. Pierwiastek z p nie jest liczbą pierwszą i wszystkie jego dzielniki są mniejsze od pierwiastka z p. np. 36 = 2 x 2 x 3 x 3

Z powyższych punktów wynika prosty wniosek – jeśli p jest liczbą złożoną to posiada dzielniki w przedziale <2, $\sqrt{p}$ >. Wystarczy wiec sprawdzić podzielność tylko w tym przedziale. Jeśli żadna liczba z tego przedziału nie dzieli p to znaczy, że p jest pierwsza. Zysk jest duży, ale na tym nie koniec.

Spójrzmy na kilka początkowych liczb pierwszych:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 ...

Jeśli nie weźmiemy pod uwagę 2 oraz 3, każdą z nich można zapisać w postaci 6n – 1 lub 6n +1 dla pewnej liczby naturalnej n. Nie jest to przypadek. Spójrzmy:

1. Liczby postaci 6n nie mogą być pierwsze gdyż są podzielne przez 2 oraz 3
2. Liczby 6n-2, 6n+2, 6n-4, 6n+4 są podzielne przez 2, wiec również nie są pierwsze.
3. Liczby 6n-3 oraz 6n+3 nie są pierwsze gdyż dzieli je 3.
4. Liczby 6n-5 oraz 6n+5 również wypadają, a dlaczego - pomyślcie sami.

Wynika z tego, że każda liczba pierwsza będzie postaci 6n-1 lub 6n+1. Nie znaczy to wcale ze każda liczba takiej postaci jest pierwsza. Nie są to wzory na kolejne liczby pierwsze (taki wzór nie istnieje). Wzory te ograniczają nam jedynie zbiór kandydatów do sprawdzenia poprzednią metoda. Wystarczy, że będziemy sprawdzać liczby postaci 6n-1 oraz 6n+1 dla kolejnych liczb naturalnych.
Oczywiste jest, ż żadna liczba pierwsza nie jest parzysta. Dodatkowo zauważmy, że nie trzeba sprawdzać podzielności przez 4, 6, 8 ,10... jeśli sprawdziło się podzielność przez 2, nie trzeba sprawdzać podzielności przez 6,9,12,15... jeśli się sprawdziło podzielność przez 3 itd. Wynika z tego, że dla danej liczby p wystarczy sprawdzić jej podzielność przez liczby pierwsze w przedziale <2, $\sqrt{p}$ >. Jest to wygodne podejście jeśli generujemy liczby pierwsze w przedziale np. <2,100000>, gdy generujemy liczby pierwsze po kolei.

Sito Eratostenesa

Powyższe metody nie są najlepsze, jeśli chodzi o wygenerowanie wszystkich liczb pierwszych mniejszych od zadanej np. 1000000. Dużo szybszą metodę podał starożytny uczony Eratostenes. Zamiast sprawdzać podzielność kolejnych liczb naturalnych przez znalezione liczby pierwsze, możemy wyrzucać ze zbioru liczb naturalnych wielokrotności kolejnych liczb naturalnych, które nie zostały wcześniej wyrzucone. Oto przykład:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Bierzemy 2 i wyrzucamy wszystkie jej wielokrotności. Zrealizować to można za pomocą dodawania tej samej liczby, co jest wydajniejsze niż użyte wcześniej dzielenie. Otrzymujemy:

2 3 5 7 9 11 13 15 17 19

Następnie bierzemy 3 i usuwamy jej wielokrotności. Wynik:

2 3 5 7 11 13 17 19

W realizacji komputerowej najłatwiej jest reprezentować liczby za pomocą indeksów tablicy wartości logicznych. Wartość false na miejscu o indeksie i oznacza ze liczba i została wyrzucona ze zbioru.
Oczywiście podczas realizacji algorytmu trzeba uważać by nie wykonywać zbędnych operacji. Biorąc kolejną liczbę i chcąc usuwać jej wielokrotności, należy sprawdzić czy ta liczba nie została sama wcześniej usunięta, jeśli tak to usunięte zostały także wszystkie jej wielokrotności, nie ma wiec sensu powtarzać tej czynności.
Są możliwe i inne ulepszenia. Zauważmy, że w pierwszym kroku usuwamy wszystkie wielokrotności liczby 2. Kiedy w następnym kroku usuwamy wielokrotności 3, widzimy ze 6 = 3x2 już zostało usunięte, gdyż 6 dzieli się przez 2. Pierwszą usuniętą wielokrotnością liczby 3 jest jej kwadrat, czyli 9 = 3x3. Dla kolejnej liczby, czyli 5, usuwamy 10, ale ona już została usunięta (dzieli się przez 2), kolejno 15, ale ona również została wcześniej usunięta (dzieli się przez 3) Podobnie dla 20=5x4. Pierwszą usuniętą wielokrotnością liczby 5 jest znów jej kwadrat 25=5x5. Jak łatwo zauważyć jest tak zawsze dla kolejnych liczb i ich wielokrotności. Dla kolejnych liczb, ich wielokrotności mniejsze od ich kwadratu posiadają dzielniki równe wcześniej znalezionym liczbom pierwszym, zatem zostały usunięte już ze zbioru wcześniej. Mamy zatem bardzo użyteczny wniosek z powyższych rozważań. Jeśli p jest górnym ograniczeniem naszego zbioru, to co się stanie jeśli dojdziemy do $\sqrt{p}$? Wtedy pierwszą

liczbą, którą wypada nam wyrzucić jest dopiero p. Zatem wielokrotności liczb większych od $\sqrt{p}$ zostały usunięte w trakcie usuwania wielokrotności ich czynników pierwszych. Cały proces można zatem przerwać po przekroczeniu granicy równej $\sqrt{p}$. Zysk jest ogromny. Przykładowo, dla p =1000000 wystarczy, że dojdziemy do p'=1000. Tysiąc razy liczb mniej do przejrzenia to już coś.

Oczywiście korzystne jest odpowiednie wyznaczenie pierwszej wielokrotności, która ma być usunięta. Z powyższych rozważań wynika, że nie trzeba usuwać wielokrotności mniejszych niż kwadrat kolejnej liczby, gdyż takie wielokrotności zostały usunięte wcześniej.

W praktyce liczby pierwsze używane w kryptografii są dużo większe niż zakres liczb całkowitych w tradycyjnych językach programowania, takich jak C, Pascal. Często potrzebne są liczby pierwsze o długości 2000 – 4000 bitów lub nawet większe. Jednakże małe liczby pierwsze, jak zobaczymy dalej, również są przydatne. Powstaje jednak pytanie jak generować duże liczby pierwsze. Przedstawione powyżej metody są za mało wydajne by stosować je do generacji bardzo dużych liczb pierwszych. W praktyce zatem będziemy generować dużą liczbę i następnie poddawać ją testowi na „pierwszość". Test pozwoli określić czy dana liczba jest pierwsza z pewną dokładnością, prawdopodobieństwem. Test powtórzony kilkakrotnie pozwoli nam zmniejszyć prawdopodobieństwo pomyłki. Nie uzyskamy zatem pewności, lecz dowolnie małe prawdopodobieństwo. Poznamy zatem metodę probabilistyczną generowania dużych liczb pierwszych, jednakże w praktyce obliczenia nasze przeprowadzać będziemy na liczbach tak dużych, na jakie pozwoli nam zakres języka w którym będziemy implementować algorytm. W zastosowaniach praktycznych skorzystalibyśmy z bibliotek implementujących liczby całkowite dowolnej precyzji. Bibliotekę taką można napisać samemu, jednakże jako że zadanie takie nie jest rzeczą łatwą, jeśli chcemy stworzyć rzeczywiście niezawodne klasy i funkcje, lepiej jest skorzystać z gotowych bibliotek dostępnych w Internecie lub skorzystać z języka programowania obsługującego liczby całkowite dowolnej wielkości Zanim jednak zajmiemy się przedstawieniem metody generowania dużych liczb pierwszych potrzebne nam będą pewne wiadomości.

Największy wspólny dzielnik NWD

Mając dwie liczby całkowite a i b ich największym wspólnym dzielnikiem nazywamy największą liczbę d=NWD(a,b) która dzieli bez reszty obie liczby a i b. Dla przypomnienia – najmniejsza wspólna wielokrotność dwóch liczb a i b NWW(a,b) to najmniejsza liczba całkowita, która jest podzielna przez a oraz b. Związek miedzy tymi dwoma wielkościami jest następujący: NWW(a,b)=ab/NWD(a,b).

Jak zatem znaleźć NWD(a,b)? Rozwiązanie tego problemu znane jest od dawna. Wymyślił je starożytny matematyk Euklides.

Algorytm Euklidesa
DANE: dwie liczby całkowite a i b, a>b
While b!=0 do
(a, b)← (b, a mod b)
end
return a

Jak widać, w każdym kroku należy zastępować odpowiednio liczby a i b obliczając resztę z dzielenia aktualnego a przez aktualne b. Reszty te w każdym kolejnym kroku są coraz mniejsze, co łatwo sprawdzić, zatem algorytm zawsze będzie miał rozwiązanie. Dla przykładu znajdźmy NWD dla liczb 48 i 36

NWD(64, 36) = NWD(36, 64 mod 36) = NWD(36, 28)= NWD(28, 36 mod 28)= NWD(28,8)= NWD(8, 28 mod 8)= NWD(8, 4)= NWD(4, 8 mod 4)= NWD(4, 0)

Zatem NWD(64, 36) = 4. Żadna inna liczba większa od 4 nie dzieli jednocześnie 64 i 36.

Zamiast dokonywać dzielenia modulo można wykonywać odejmowania:

Algorytm Euklidesa 2
DANE: dwie liczby całkowite a i b, a>b
While b!=0 do
(a, b)← (b, a - b)
end
return a

Przykład:

NWD(64, 36)=NWD(36,64-36)= NWD(36,28)= NWD(28,36-28)= NWD(28,8)= NWD(8,28-8)= NWD(8,20) =NWD(20,8)= NWD(12,8)= NWD(8,4)= NWD(4,4)= NWD(4,0)

Jak widać wynik jest ten sam jednak potrzeba było (w tym przypadku) więcej kroków. Jednak trzeba wziąć pod uwagę, że operacja dodawania (zatem również odejmowania) jest szybciej wykonywana niż dzielenie modulo.

Często można przyśpieszyć działanie algorytmu Euklidesa dopuszczając dzielenie z ujemnymi resztami. Przykładowo:

64 = 1 x 36 + 28
ale również
64 = 2 x 36 – 8

Jeśli wybierać będziemy mniejsze reszty zmniejszymy ilość potrzebnych kroków.
Przykład:

NWD(64,36)= NWD(36,8)
36=4 x 8 + 4 oraz 36 = 5 x 8 – 4 (tu akurat bez różnicy)
NWD(36,8)= NWD(8,4)= NWD(4,0)=4

Algorytm Euklidesa działa, ponieważ kolejne pary mają ten sam zbiór wspólnych dzielników, zatem w szczególności maja ten sam największy wspólny dzielnik. Nie będziemy jednak przytaczać dokładnego dowodu.

Jeszcze inna odmiana algorytmu Euklidesa korzysta z następujących kroków:

Algorytm Euklidesa 3
DANE: dwie liczby całkowite a i b, a>b

WYJSCIE: d = NWD(a,b)
While a!=b do
Jeśli a i b są parzyste to d=2*d', gdzie d'=NWD(a/2, b/2)
Jeśli jedna jest parzysta a druga nieparzysta (np.b jest parzysta) to d=d', gdzie d'=NWD(a,b/2)
Jeśli obie są nieparzyste i np. a>b , to d=d', gdzie d'=NWD(a-b,b)
Jeśli a jest równe b to d=a
end
return a

Przykład:
NWD(64,36)=2*NWD(32,18)=4*NWD(16,9)=4*NWD(8,9)=4*NWD(4,9)=4*
NWD(2,9)=4* NWD(1,9)= 4* NWD(1,8)= 4* NWD(1,7)=...= 4* NWD(1,1)=4

W przykładzie powyżej ilość kroków jest zdecydowanie większa od ilości kroków w sposobie pierwszym, jednak metoda ta ma swoje zalety. Jak widać mamy w tej metodzie tylko dzielenie przez 2 i odejmowanie. Jest to szczególnie użyteczne gdy operujemy na liczbach w zapisie dwójkowym, gdzie dzielenie przez 2 sprowadza się do przesunięcia bitów w prawo.
Przykładowo:

12 / 2 = 6
$12_{(10)} = 1100_{(2)}$
→→ (przesuniecie bitów – usuniecie bitu najmniej znaczącego)
$6_{(10)} = 110_{(2)}$

Mnożenie przed dwa sprowadza się natomiast do przesunięcia bitów w lewo i dopisaniu na miejscu powstałego najmniej znaczącego bitu zera.
$3_{(10)} = 11_{(2)}$
←
$2*3 = 6_{(10)} = 110_{(2)}$

Operacje przesunięcia bitów mogą być zaimplementowane bardzo wydajnie.

Do czego się nam przyda algorytm Euklidesa, o tym za chwile. Teraz jeszcze zdefiniujmy pojecie względnej pierwszości dwóch liczb.

Def. Dwie liczby a i b są względnie pierwsze, jeśli NWD(a,b)=1

Działania modulo

Dodawanie, odejmowanie i mnożenie modulo nie różnią się zbytnio od swoich zwykłych odpowiedników. Polegają one na wykonaniu działania „normalnie" a następnie skróceniu wyniku modulo dana liczba. Przykładowo załóżmy, że będziemy wykonywać działania modulo p = 5.
Zatem jeśli a = 7 oraz b=4 mamy:

(a+b) mod p = (7+4) mod 5 =11 mod 5 = 1
(a-b) mod p = (7-4) mod 5 =3 mod 5 = 3
(a*b) mod p = (7*4) mod 5 =28 mod 5 = 3

Tutaj mała uwaga. Resztę przyjmować będziemy zawsze jako liczbę dodatnią większą lub równą zero i mniejszą od dzielnika. Jaki jest zatem wynik działania

-1 mod 5 ?

Mamy dwie możliwości zapisu –1 za pomocą 5:
-1 = 0*5 –1
lub
-1 = -1*5 + 4

Przyjmować będziemy tę drugą odpowiedź, czyli –1 mod 5 = 4.

Co jednak z dzieleniem modulo? Okazuje się ze nie jest to takie proste jak z pozostałymi działaniami modulo. Szczególnie interesuje nas przypadek znajdowania dla danej liczby b jej odwrotności modulo p czyli 1/b mod p , gdzie 1<=b<p. Znając odwrotność danej liczby modulo p możemy wykonywać dzielenie przez tą liczbę modulo p czyli obliczyć a/b mod p. Jednak nie zawsze możemy obliczyć odwrotność danej liczby modulo p.

Twierdzenie: Element odwrotny do b modulo p definiujemy jako taką liczbę a, taką ze ab=1(mod p)
Liczba a istnieje tylko gdy b oraz p są względnie pierwsze, czyli NWD(p,b) = 1.

Zauważmy, że warunek ten jest spełniony zawsze gdy p jest liczbą pierwszą.

Ponownie pominiemy dowód tego twierdzenia i zapytamy od razu co ono nam daje. Otóż mając NWD(p,b) = d można zawsze jednoznacznie zapisać go jako:

d = ub + vp

w naszym wypadku d=1 wiec

1=ub + vp

Szukanym elementem odwrotnym a jest u. Jest tak dlatego iż z ab=1(mod p) wynika ze
p | 1-ab (p dzieli ab-1) z kolei 1-ub=vp wiec p | 1-ub . Zatem a=u.

Zatem, reasumując, by obliczyć liczbę odwrotną do b modulo p, należy najpierw sprawdzić czy są one względnie pierwsze, czyli czy NWD(p,a)=1. Jeśli tak, należy znaleźć takie u oraz v, że 1=ub + vp. Wtedy szukanym elementem odwrotnym jest u (UWAGA: jeśli u wyjdzie ujemne, trzeba zredukować u modulo p zanim użyje się go jako odwrotności b). Jak znaleźć u oraz v? Posłuży nam do tego rozszerzony algorytm Euklidesa:

Rozszerzony algorytm Euklidesa:

DANE: m>0, n>0
WYJSCIE: k =NWD(n,m)
u, v – liczby całkowite, takie ze um+vn=k

assert n>=0 , m>=0, n>=m
(a, a')←(n,m)

(u, u’, v, v’)←(0,1,1,0)

while a’!=0 do
    q=a/a’ //dzielenie bez reszty, całkowite
    (a,a’)←(a’, a-qa’)
    (u, u’, v, v’)←(u’,u-qu’,v’,v-qv’)
end
return NWD(n,m)=a, oraz u, v

Przykład: Obliczmy 160^{-1} mod 841.d

Liczby 841 oraz 160 musza być względnie pierwsze by szukany element odwrotny istniał. Obliczmy zatem rozszerzonym algorytmem Euklidesa NWD(841, 160)
(a,a’)←(841,160)
(u, u’, v, v’)←(0,1,1,0)

PETLA:
Krok 1;
q=841/160=5
(a, a’)←(160, 841-5*160)=(160, 41)
(u, u’, v, v’)←(1,0-5*1,0,1-5*0)=(1,-5,0,1)

Krok 2:
q=160/41=3
(a, a’)←(41, 160-3*41)=(41, 37)
(u, u’, v, v’)←(-5,1-3*(-5),1,0-3*1)=(-5,16,1,-3)

Krok 3:
q=41/37=1
(a, a’)←(37, 41-1*37)=(37, 4)
(u, u’, v, v’)←(16,-5-1*16,-3,1-1*(-3))=(16,-21,-3,4)

Krok 4:
q=37/4=9
(a, a’)←(4, 37-9*4)=(4, 1)
(u, u’, v, v’)←(-21,16-9*(-21),4,-3-9*4)=(-21,205,4,-39)

Krok 5:
q=4/1=4
(a, a’)←(1, 4-4*1)=(1,0)
(u, u’, v, v’)←(205,-21-4*205,-39,4-4*(-39))=(205,-841,-39,160)

a’==0 wiec STOP

NWD(841,160) = a = 1 = 205*160 – 39*841

Zatem element odwrotny do 160 modulo 841 istnieje gdyż NWD(841,160)=1 i wynosi on 205. Można sprawdzi poprawność obliczeń:

160*205 (mod 841) = 32800 (mod 841) = 1

Zatem zgadza się. Umiemy obliczać element odwrotny danej liczby modulo p, jeśli te liczby są względnie pierwsze.

Algorytm Iteracyjnego Podnoszenia do Kwadratu

Potrzebna nam będzie jeszcze umiejętność obliczania wyrażeń typu b^n mod m, gdzie liczby n oraz m są dużymi liczbami, przykładowo mamy obliczyć 123^{35}mod 133. Zakładamy, że b<m. Zamiast podnosić b do tak wysokiej potęgi możemy iteracyjnie mnożyć b przez samego siebie i za każdym razem skracać wynik modulo m. Zatem w wyniku zawsze będziemy mieć do czynienia z liczbami nie przekraczającymi m.
Częściowe iloczyny oznaczamy jako a. Na końcu algorytmu szukany wynik będzie końcową wartością a.

Algorytm IPdK:

1. Zapisz wykładnik n w postaci binarnej, $n_{(2)}=n_o + 2n_1 + 4n_2 + ...+ 2^{k-1}n_{k-1}$, gdzie k jest ilością bitów w zapisie binarnym liczby n.
2. Najmniej znaczący bit n_0 służy nam jedynie do ustalenia początkowej wartości a. Jeśli $n_0=1$ to ustaw a=b, jeśli $n_0=0$ ustaw a=1.
3. Następnie podnosimy b do kwadratu i skracamy modulo m, czyli $b_1=b^2$mod m. Jeśli $n_1=1$ to wykonujemy a=a*b mod m, w przeciwnym wypadku, gdy $n_0=0$, nie zmieniamy wartości a.
4. W każdym kroku i obliczamy wartość $b_i=b_{i-1}^2$ mod m, i jeśli $n_i=1$ to obliczamy $a_i=a_{i-1}*b_i$ mod m.
5. Wynikiem jest końcowa wartość a.

Przykład: Obliczmy 123^{35}mod 133

Najpierw zapisujemy $35_{(10)}=100011_{(2)}$

1. $n_0=1$ więc a = b = 123
2. $b_1=b^2$ mod m = 123*123 mod 133=100 oraz $n_1=1$ więc $a_1=a*b_1$ mod m = 123*100 mod 133 = 64
3. $b_2=b_1^2$ mod m = 100*100mod 133=25 oraz $n_2=0$
4. $b_3=b_2^2$ mod m = 25*25mod 133=93 oraz $n_3=0$
5. $b_4=b_3^2$ mod m = 93*93mod 133=4 oraz $n_4=0$
6. $b_5=b_4^2$ mod m = 4*4mod 133=16 oraz $n_5=1$ więc $a_5=64*16$ mod 133=93

Zatem 123^{35}mod 133 = 93

Umiejętność ta przyda nam się w następnym punkcie.

Generowanie dużych liczb pierwszych

Jak już mówiliśmy wcześniej, algorytmy przedstawione na początku tego opracowania, nie są zbyt dobrym rozwiązaniem, jeśli chcemy wygenerować naprawdę dużą liczbę pierwszą, przykładowo 4000-bitową. Dopiero takiej długości liczby, lub nawet dłuższe, gwarantują bezpieczeństwo.

Metoda jaką przyjmiemy jest bardzo prosta – wybierzemy dowolną liczbę i sprawdzimy czy jest pierwsza. Liczb pierwszych jest dość dużo, w okolicy danej liczby n średnio jedna liczba

na ln(n) jest liczbą pierwszą. Logarytm naturalny nie jest funkcją szybko rosnącą, przykładowo wartość logarytmu naturalnego z liczby 2^k nieznacznie przekracza 0,7*k. Dla liczb 2000-bitowych, mieszczących się w zakresie $2^{1999} – 2^{2000}$, mamy mniej więcej 1386 liczb pierwszych. Dodatkowo z rozwiązań od razu możemy wyrzucić liczby w sposób oczywisty złożone, na przykład parzyste.

Algorytm GenerujLiczbePierwszą:

1. Ustal zakres, z jakiego będzie generowana liczba pierwsza
2. Ustal ilość prób generowania liczby pierwszej
3. Dopóki nie przekroczono maksymalnej ilości prób, wygeneruj losowa liczbę z zadanego przedziału i sprawdź czy jest pierwsza, jeśli nie, wygeneruj następną losową liczbę.

Należy zabezpieczyć się przed przypadkiem gdy dany przedział nie zawiera żadnej liczby pierwszej – program nie powinien się wtedy zawieszać. W praktyce należy uważać by dany przedział nie był również zbyt mały, gdyż, ktoś, znając przedział może spróbować wygenerować wszystkie liczby pierwsze z tego przedziału – jeśli jest on zbyt mały, może się to udać, co może prowadzić do naruszenia przyjętych zasad bezpieczeństwa. Ilość prób można określić wykorzystując wcześniej podane informacje. Przykładowo r (czyli ilość prób) może być równa

$r = 100*(\log_2 u + 1)$

gdzie u to górne ograniczenie naszego zakresu poszukiwań. Dla liczby 2000-bitowej $\log_2 u + 1$ wynosi 2000. Czynnik 100 daje dodatkową gwarancję, ze prawdopodobieństwo nie znalezienia liczby pierwszej w danej ilości prób jest znikome.
Dodatkowo należy pamiętać o korzystaniu z generatora losowego liczb, który będzie generował liczby z równomiernym rozkładem.. Można również w celu uniknięcia sprawdzania liczb parzystych (które wiadomo, ze nie są pierwsze) ustawić najmniej znaczący bit wygenerowanej liczby.
Sprawdzenie pierwszości danej losowej liczby przebiega następująco:

Algorytm SprawdzCzyPierwsza:

1. **assert** n>=3
2. Sprawdź podzielność liczby n przez znane, małe liczby pierwsze np. <1000. Pozwala to szybko wykryć większość liczb złożonych podzielnych przez małe liczby. Jeśli badana liczba n jest podzielna przez którąś ze znanych, małych liczb pierwszych, zwróć false.
3. Jeśli badana liczba n przeszła zwycięsko krok 2, wykonaj na niej test Rabina-Millera. Jeśli przejdzie go zwycięsko, zwróć true, inaczej zwróć false.

Pozostaje do omówienia test Rabina-Millera.

Test Rabina-Millera

Test ten ma charakter probabilistyczny, to znaczy, ze nie uzyskamy pewności czy dana liczba jest pierwsza. Możemy jednak dowolnie zmniejszać prawdopodobieństwo pomyłki. Test ten ma na celu sprawdzenie czy dana, **nieparzysta** liczba n jest pierwsza. Dojście do testu

Rabina-Millera w algorytmie SprawdzCzyPierwsza daje pewność ze n jest nieparzyste (inaczej byłaby podzielna przez 2 co byłoby wykryte wcześniej).
W teście Rabina-Millera dla danej liczby n wybieramy losową liczbę a, taką że a<n. Liczbę a nazywamy bazą i sprawdzamy pewną właściwość a modulo n. Właściwość ta jest spełniona jeśli n jest pierwsza. Niestety, jeśli n nie jest pierwsza, to w przypadku co najwyżej 25% rożnych liczb a (wartości bazy) właściwość ta również zachodzi. Powtarzając wiec test dla rożnych wartości bazy a, będziemy zmniejszać prawdopodobieństwo pomyłki. Im więcej różnych wartości bazy a, tym mniejsze prawdopodobieństwo pomyłki. Jeśli dla danej wartości a, liczba n nie przejdzie testu, to znaczy, że nie jest pierwsza, jeśli natomiast przejdzie, nie wiemy na pewno czy jest pierwsza, ale po coraz większej ilości różnych a, prawdopodobieństwo tego, że się mylimy, maleje. Jeśli n jest pierwsza, przejdzie przez każdą edycję testu dla jakiejkolwiek wartości a. Jeśli nie jest pierwsza, wykaże to przynajmniej 75% możliwych wartości a. Można zatem ustalić zadany poziom bezpieczeństwa, przykładowo ustalając dopuszczalne prawdopodobieństwo pomyłki na 2^{-128}.

Oto omawiany test:

Funkcja Rabin-Miller
Dane: n – liczba nieparzysta>=3 (nie dopuszczamy wartości 2)
Wynik: wartość logiczna wskazująca czy dana liczba n jest pierwsza, czy też nie

assert n>=3 oraz (n mod 2) =1
Wyznaczamy takie (s, t), że s jest nieparzyste i $2^t s= n-1$
(s, t) ←(n-1,0)
while s mod 2 = 0 **do**
    (s, t)←(s/2, t+1)
end

W zmiennej k generujemy dane o prawdopodobieństwie uzyskania fałszywego wyniku. Prawdopodobieństwo nie przekracza 2^{-k}. Każemy pracować pętli tak długo, aż fałszywy wynik stanie się mało prawdopodobny.

k←0
while k<128 **do** //ustalony poziom bezpieczeństwa na 128
    Wybieramy losową liczbę a spełniającą 2<=a<=n-1
    a←random({2..n-1})
    v←a^s mod n
    Jeśli v=1, liczba n przeszła test dla bazy a, w przeciwnym wypadku nadal może go przejść, dlatego sprawdzamy dalej
    if v!=1 **then**
        Ciąg v, v^2, .., $v^{pow(2,t)}$ musi kończyć się wartością 1 a jeśli n jest liczbą pierwszą, ostatnią wartością różną od 1 musi być n-1. Zatem
        i←0
        while v!=n-1 **do**
            if i=t-1 **then**
                return false *//test zakończony niepowodzeniem*
            else
                (v, i)←(v^2 mod n, i+1)
        end while
    end if

Dojście do tego miejsca oznacza, ze n przeszła test względem bazy a. Wobec tego prawdopodobieństwo uzyskania fałszywej odpowiedzi zostało zredukowane o czynnik 2^2, zatem zwiększamy k o 2.
k←k+2
end while // pętla po różnych wartościach bazy a
return true //n przeszła testy dla wszystkich wylosowanych baz a, wiec można zwrócić true

Dlaczego ten test działa? Podstawą jego działania jest małe twierdzenie Fermata. Mówi ono, że dla dowolnej liczby pierwszej n i dla wszystkich liczb a, takich że 1<=a<n, zachodzi $a^{n-1} \bmod n = 1$. Więc jeśli n jest pierwsza , własność ta zachodzi, jednak istnieją pewne liczby złożone, zwane liczbami Carmichaela, które mimo iż są złożone, przechodzą test dla wielu baz a.

Jeszcze kilka wyjaśnień. Przedstawienie liczby n-1 w postaci $2^t s$, gdzie s jest nieparzysta, pozwala nam zamiast obliczania a^{n-1} obliczyć a^s a następnie ten wynik podnosić do kwadratu t razy i otrzymać $a^{s*pow(2,t)} = a^{n-1}$. I teraz, jeśli $a^s = 1 \pmod n$, powtórne podniesienie do kwadratu nie zmieni wyniku, wiec będziemy mieć $a^{n-1}=1 \pmod n$. W przypadku $a^s \neq 1 \pmod n$ sprawdzamy kolejne kwadraty tej liczby czyli ciąg a^s, $a^{s*pow(2,1)}$, $a^{s*pow(2,2)}$,..., $a^{s*pow(2,t)}$ (wszystkie modulo n). Gdyby n była liczbą pierwszą, to ostatnia liczba musiałaby być równa 1. Kiedy n jest liczbą pierwszą, jedynymi liczbami x spełniającymi warunek $x^2=1 \pmod n$ są 1 i n-1. Gdyby zatem n była liczbą pierwszą, w podanym ciągu musiałaby wystąpić liczba n-1 (stad warunek pętli while v!=n-1), gdyż w przeciwnym wypadku nigdy ostatni wyraz nie równałby się 1. Jeśli powyższe nie zachodzi, to zwracamy false, przerywamy test i wychodzimy z funkcji. Sprawdzamy liczbę n dla rożnych wartości a dopóki prawdopodobieństwo błędu nie spadnie poniżej (w tym przykładzie) 2^{-128}.

Poznaliśmy zatem wszystkie potrzebne nam narzędzia matematyczne, które będą nam potrzebne przy właściwym szyfrowaniu RSA.

Szyfrowanie RSA

W poprzednim laboratorium mieliśmy do czynienia z szyfrowaniem z kluczem symetrycznym, to znaczy taki sam klucz musiał być znany zarówno nadawcy jak i odbiorcy, aby, odpowiednio, zaszyfrować i odszyfrować wiadomość. Z tego wniosek, że musiał on być znany im obu przed wysłaniem pierwszej zaszyfrowanej wiadomości. W przeszłości zatem, przed podjęciem szyfrowanej korespondencji, musieli spotkać się i osobiście uzgodnić hasło, lub sposób szyfrowania, który w obu przypadkach (szyfrowanie-odszyfrowywanie) przebiega według tego samego mechanizmu i z wykorzystaniem tego samego hasła. Co jednak robić w sytuacji, gdy dane osoby nie mogą się spotkać ani w żaden bezpieczny sposób uzgodnić wspólnego hasła? Rozwiązaniem są systemy z kluczami niesymetrycznymi, w których podczas szyfrowania i odszyfrowywania używa się innych klucz. Zazwyczaj każdy uczestnik takiego systemu posiada dwa rodzaje kluczy: publiczny, który udostępnia wszystkim chętnym, na przykład na swojej stornie internetowej, a drugi prywatny, który zachowuje w tajemnicy. Za pomocą klucza publicznego chętni wysyłają do osoby, która udostępniła ten klucz zaszyfrowane tym kluczem wiadomości. Posiadacz klucza prywatnego jest w stanie za jego pomocą odszyfrować wiadomość. Cały sekret tkwi w tym, iż klucz publiczny nie może odszyfrować zaszyfrowanej nim wiadomości, można tego dokonać jedynie znając klucz prywatny.

Przykładem takiego systemu jest RSA. RSA przedstawiony został oficjalnie w roku 1979 i jest dziełem trzech ludzi: Ronalda Rivesta, Adi Shamira oraz Leonarda Adlemana - jego nazwa pochodzi o pierwszych liter nazwisk swych twórców.

Jak działa RSA? Zaczniemy od dwóch dużych, losowo wybranych, rożnych liczb pierwszych p i q. Obliczamy ich iloczyn n=pq. Liczbę n będziemy wykorzystywać do wykonywania operacji modulo. Jednak widzimy, ze nie jest ona liczbą pierwszą, wiec nie zachodzi równość, $x^{n-1} =1 \pmod n$ dla $0<x<p$. Aby użyć RSA musimy znaleźć taki wykładnik t by $x^t=1 \pmod n$ dla (prawie) wszystkich x. Spójrzmy: jeśli zachodzi $x^t=1 \pmod n$ to zachodzi również $x^t=1 \pmod p$ oraz $x^t=1 \pmod q$. Liczby p i q są pierwsze więc równanie $x^t=1 \pmod n$ będzie zachodzić tylko wtedy gdy p-1 będzie dzielnikiem t oraz q-1 będzie dzielnikiem t. Zatem najmniejsza liczba o takiej właściwości to NWW(p-1, q-1)=(p-1)(q-1)/NWD(p-1, q-1). Przyjmijmy zatem w dalszych rozważaniach, że t= NWW(p-1,q-1). Czasami używa się funkcji Eulera. Funkcja ta dla liczby n=pq gdzie p i q są pierwsze wynosi
$\phi(n) = (p-1)(q-1)$
Wartość tej funkcji jest wielokrotnością naszego t. Użycie jej zamiast t również da dobre wyniki.

Wracajmy do RSA. Mając p , q, n oraz t. Potrzebujemy teraz dwóch różnych wykładników e oraz d. Muszą one spełniać warunek $ed=1 \pmod t$. Jako publicznie znany wykładnik e wybieramy niewielką liczbę nieparzystą. Posługując się rozszerzonym algorytmem Euklidesa obliczamy d jako odwrotność e modulo t.
Aby zaszyfrować wiadomość m, nadawca oblicza tekst zaszyfrowany

$c=m^e \pmod n$

Aby odszyfrować tekst c, odbiorca oblicza

$c^d \pmod n$

co jest równe oryginalnej wiadomości m. Klucz publiczny stanowi para (n, e). Kluczem prywatnym jest zestaw (p, q, t, d). Powinien on pozostać ukryty. Ze względu na zależność $x^t=1 \pmod n$, wykładniki obliczeń modulo n należy brać modulo t, gdyż wielokrotności t w wykładniku operacji modulo n nie wpływają na wynik.

Jak widać umiemy przeprowadzić wszystkie potrzebne operacje matematyczne. Musimy tylko jeszcze zwrócić uwagę na parę szczegółów. Otóż jeśli e będzie miało wspólny czynnik z t = NWW(p-1,q-1), to nie będzie istniał element odwrotny do e modulo t, czyli nie obliczymy potrzebnego d. Aby można go było obliczyć, e i t muszą być względnie pierwsze. Zatem po wygenerowaniu liczb pierwszych p i q, należy sprawdzić czy dla wybranej wartości e liczby p-1 oraz q-1 nie mają wspólnych czynników z wybranym e. Wybór malej wartości e przyspiesza obliczenia i upraszcza konstrukcję systemu. Zatem jako e możemy przyjąć e=3. Po wygenerowaniu p i q sprawdzamy czy nie są one podzielne przez 3. Innymi słowy wylosowana liczba k musi spełniać k mod 3 != 1. Innym wyborem na e może być np. 5, 17 itd. Liczba e nie musi być duża, duże muszą być liczby p i q.
Jak widać na podstawie jedynie klucza publicznego nie da się łatwo odgadnąć liczb w kluczu prywatnym. Można by tego dokonać rozkładając n na czynniki by poznać p i q, jednak nie jest znany wydajny algorytm rozkładu liczb na czynniki czyli faktoryzacji liczb złożonych. Dlatego mówi się, że problem rozkładu na czynniki pierwsze jest tak ważny w kryptografii.

Liczba n powinna mieć kilka tysięcy bitów długości by zapewnić bezpieczeństwo na odpowiednim poziomie.

Przykład: Wybieramy e=3. Dobieramy takie liczby pierwsze p i q aby p-1 oraz q-1 nie dzieliły się przez 3. Warunek taki spełniają p=53 oraz q=71. Zatem n = 53*71=3763. Za t przyjmijmy t = (53-1)*(71-1)=3640. Obliczamy teraz element odwrotny do e=3 modulo t=3640. Wynosi on d=2427. Sprawdzamy:

3*2427 mod 3640 = 7281 mod 3640 = 1 – zgadza się

Załóżmy, że chcemy zaszyfrować wiadomość zapisaną jako m=1655. Klucz publiczny stanowi para (n, e) = (3763, 3). Szyfrujemy m:

$c = m^e \bmod n = 1655^3 \bmod 3763 = 3477$

Klucz prywatny to zestaw liczb (p, q, t, d) = (53, 71, 3640, 2427). Odszyfrowujemy c:

$m = c^d \bmod n = 3477^{2427} \bmod 3763 = 1655$

A wiec odzyskaliśmy oryginalną wiadomość. Należy zwrócić jednak uwagę na fakt, iż jeśli oryginalna wiadomość podniesiona do potęgi e (e jest małe) nie przekroczy n, to nie zostanie ona zaszyfrowana, gdyż nie nastąpi skrócenie modulo.
Przykład: chcemy zaszyfrować m = 12

$c = m^e \bmod n = 12^3 \bmod 3763 = 1728 \bmod 3763 = 1728$

Jako że e jest znane, gdyż jest częścią klucza publicznego, łatwo obliczyć po prostu pierwiastek trzeciego stopnia z 1728 i odzyskać m. Należy o tym pamiętać.

Powyższe podejście użycia RSA do szyfrowania wiadomości ma swoją wadę, mianowicie długość wiadomości do zaszyfrowania jest ograniczona wielkością n (musi być od niej mniejsza). Oprócz tego trzeba najpierw uzgodnić, opracować sposób przedstawiania wiadomości np. tekstowych za pomocą liczb. Można przyjąć wartości kodów ASCII, ale jest to podejście niezbyt dobre. Dłuższa wiadomość można podzielić na mniejsze części, nie przekraczające n, i kodować każdą z tych części osobno. Jednak jest to w praktyce niewydajne, gdyż szyfrowanie RSA wymaga przeprowadzenia wielu skomplikowanych i czasochłonnych operacji matematycznych, jak potęgowanie modulo. Alternatywnym i dużo lepszym w praktyce rozwiązaniem jest kodowanie za pomocą RSA nie samej wiadomości, tylko klucza, który służy do szyfrowania i odszyfrowywania wiadomości. Sama wiadomość nie ma wtedy ograniczeń co do długości. Sam klucz jest wykorzystywany do szyfrowania blokowego lub strumieniowego. Oczywiście klucz ten, zapisany jako liczba, musi spełniać odpowiednie warunki by mógł by zaszyfrowany za pomocą kluczy RSA, tzn. nie może przekraczać n.

Szyfrowanie za pomocą klucza można zrealizować przy pomocy operacji logicznej XOR. Załóżmy, że nasza wiadomość to ciąg bitów

m = 1001100011

a klucz to również ciąg bitów

k = 1000111010
Wynik operacji XOR jest równy 1 jeśli oba bity przekazane jako argumenty są różne, a zero gdy są identyczne. Zatem:

1 XOR 1 = 0
1 XOR 0 = 1
0 XOR 0 = 0
0 XOR 1 = 1

Stosując te operacje na ciągach bitów wiadomości i klucza otrzymujemy:

m = 1001100011
k = 1000111010
---
c = 0001011001

Co jest ważne, to to, że odszyfrowanie przebiega w identyczny sposób – bierzemy ciąg bitów szyfrogramu i ciąg bitów klucza i przeprowadzamy ponownie operacje XOR:

c = 0001011001
k = 1000111010
---
m = 1001100011

Jeśli wiadomość, którą chcemy zaszyfrować jest dłuższa niż długość klucza, to dzielimy ją na mniejsze bloki równe długości klucza i każdy blok szyfrujemy osobno. Jest to przykład szyfrowania blokowego (szyfrowaniem strumieniowym zajmiemy się przy okazji jednego z kolejnych ćwiczeń). Oczywiście stosując ten sposób maksymalna długość klucza jest ograniczona przez n. Jeśli zależy nam na dłuższym kluczu, możemy zwiększyć n lub też zastosować jeszcze inne podejście wykorzystujące funkcję mieszającą. Funkcja mieszająca to funkcja, która pobiera jako dane wejściowe dowolnej długości ciąg bitów i zwraca jako wynik ciąg bitów o **stałym** rozmiarze. Pomysł polega zatem na zaszyfrowaniu kluczami RSA danych wejściowych do funkcji mieszającej, która wygeneruje nam ciąg bitów klucza o danej, pożądanej przez nas długości. Dopiero ten klucz będzie użyty do zaszyfrowania wiadomości. Mimo iż dane wejściowe do funkcji mieszającej są ograniczone przez n (gdyż je właśnie chcemy szyfrować) to długość klucza nie zależy od tych danych wejściowych ani od parametru n RSA, możemy wiec uzyskać klucz o długości, która zależy od zaimplementowanej przez nas funkcji mieszającej. Do odbiorcy wysyła się zaszyfrowaną wiadomość i zaszyfrowane dane wejściowe dla funkcji mieszającej. Odbiorca odszyfrowywuje swym kluczem prywatnym RSA dane wejściowe dla funkcji mieszającej, generuje za pomocą tej funkcji klucz i używa go do rozkodowania wiadomości za pomocą operacji XOR. Aby jednak nie komplikować sobie tymczasowo życia, nie będziemy używać funkcji mieszającej i pozostaniemy przy szyfrowaniu RSA samego klucza, którym będziemy szyfrować wiadomość za pomocą operacji XOR.

Zadania do zrealizowania:

W naszych ćwiczeniach nie będziemy posługiwać się na razie żadną specjalną biblioteką umożliwiającą użycie liczb całkowitych dowolnej wielkości. Wykorzystując zakres liczb całkowitych dostępnych w danym języku, zaimplementujemy same mechanizmy i algorytmy szyfrowania RSA. Uzyskany przez nas program będzie bezpieczny na tyle na ile pozwala na to wielkość zastosowanych liczb (i rzetelność implementacji).

1. Napisz funkcję testującą czy dana liczba jest liczba pierwszą.
2. Napisz funkcję generującą liczby pierwsze nie większe od zadanego parametru. Zastosuj sito Eratostenesa. Wygenerowane liczby zapisz do pliku.
3. Napisz funkcję obliczającą NWD dla dwóch podanych liczb wykorzystując algorytm Euklidesa.
4. Napisz funkcję obliczającą NWD dla dwóch podanych liczb wykorzystując rozszerzony algorytm Euklidesa.
5. Wykorzystując funkcję z poprzedniego punktu napisz funkcję obliczającą element odwrotny do danej liczby modulo inna liczba.
6. Napisz funkcję obsługującą potęgowanie modulo. Wykorzystaj algorytm iteracyjnego podnoszenia do kwadratu.
7. Napisz funkcję generującą liczbę pierwszą z danego przedziału wykorzystującą test Rabina-Millera do sprawdzenia, czy wylosowana liczba jest pierwsza.
8. Zaimplementuj w programie szyfrowanie pliku za pomocą metody RSA wykorzystując wcześniej napisane funkcje. Szyfrowanie wiadomości odbywa się wykorzystując operacje XOR na bitach klucza danej długości i bitach pobranych ze strumienia szyfrowanego pliku. Szyfrowanie RSA wykorzystane jest do szyfrowania klucza. Wyeksportuj do plików klucze publiczny i prywatny, tak byś mógł udostępnić swój klucz publiczny wraz ze swoim oprogramowaniem by inni mogli wysyłać do Ciebie zaszyfrowane wiadomości. W zaszyfrowanej wiadomości powinien być również oczywiście zapisany zaszyfrowany klucz.