1. zadaća

Općenito ne postoji efikasan algoritam za faktorizaciju proizvoljnog prirodnog broja n. Na toj činjenici počiva i sigurnost RSA kriptosustava. U ovoj zadaći bit će posvećena pažnja nekim algoritmima za faktorizaciju velikih prirodnih brojeva kroz niz manjih zadataka. Sve algoritme treba implementirati u programskom jeziku `Python` i pritom koristite neku od verzija `2.x`.

Zadatak 1.
Implementirajte Euklidov algoritam za traženje najveće zajedničke mjere dva prirodna broja. Nadalje, implementirajte algoritam (opisan u prezentaciji) za rješavanje linearne kongruencije $ax \equiv b \pmod{n}$. *Algoritam na ulazu dobiva brojeve* a, b *i* n, *a na izlazu vraća listu svih rješenja.*

Svaki složeni prirodni broj n ima barem jedan prosti faktor $\leqslant \sqrt{n}$. Na temelju toga imamo naivni algoritam za ispitivanje prostosti zadanog prirodnog broja. Za zadani prirodni broj n dovoljno je redom provjeriti sve prirodne brojeve $2 \leqslant m \leqslant \sqrt{n}$ i prvi takav m na kojeg naletimo je prosti faktor broja n. Ukoliko n nije djeljiv niti s jednim od spomenutih brojeva m, tada je n prosti broj. Ovaj algoritam nije efikasan na velikim prirodnim brojevima jer ima eksponencijalnu složenost s obzirom na broj znamenaka ulaznog prirodnog broja.

Zadatak 2.
Implementirajte naivni algoritam za ispitivanje prostosti zadanog prirodnog broja n. *Ako je* n *prosti broj, algoritam neka na izlazu vrati* 1, *a inače neka vrati neki prosti faktor* $\leqslant \sqrt{n}$.

U praksi se najčešće koriste razni vjerojatnosni testovi za ispitivanje prostosti zadanog neparnog prirodnog broja. Takvi testovi mogu u razumnom vremenu s velikom vjerojatnošću točno odgovoriti na pitanje: *Da li je* n *prosti broj?* Ovdje ćemo pažnju posvetiti Miller-Rabinovom testu. Vrijedi sljedeća tvrdnja.

Neka je n *neparan prosti broj. Neka je* $n - 1 = 2^s \cdot d$ *pri čemu je* d *neparni broj. Tada za svaki* $a \in \mathbb{Z}_n \setminus \{0\}$ *vrijedi jedna od sljedećih relacija:*

- $a^d \equiv 1 \pmod{n}$,
- $a^{2^r \cdot d} \equiv -1 \pmod{n}$ *za neki* $0 \leqslant r \leqslant s - 1$.

Miller-Rabinov test se bazira zapravo na kontrapoziciji gornje tvrdnje. U nastavku slijedi opis algoritma.

Neka je n zadani neparni broj kojeg treba testirati da li je prost. Najprije broj $n - 1$ napišemo u obliku $2^s \cdot d$ pri čemu je d neparni broj. Drugim riječima, iz parnog broja $n-1$ (jer je n neparan) izlučimo sav njegov parni dio, tj. sve moguće dvojke. Na slučajni način odaberemo neki cijeli broj a takav da je $1 < a < n$. Izračunamo $a^d \bmod n$. Ako dobijemo ± 1, zaključujemo da je n prošao test i biramo ponovo na slučajni način novi broj a. U protivnom, ako ne dobijemo ± 1, uzastopno kvadriramo a^d modulo n sve dok ne dobijemo rezultat -1. Ako dobijemo -1, tada je n prošao test. Ako nikad ne dobijemo -1, tj. ako dobijemo $a^{2^{r+1} \cdot d} \equiv 1 \pmod{n}$, ali $a^{2^r \cdot d} \not\equiv -1 \pmod{n}$, onda smo sigurni da je

n složen broj. Ako n prođe test za k odabranih brojeva a, tada je vjerojatnost da je n složen $\leqslant \frac{1}{4^k}$. U praksi se najčešće uzima $k = 20$.

Zadatak 3.
Implementirajte Miller-Rabinov test za ispitivanje prostosti bilo kojeg prirodnog broja n. U slučaju da je $n = 1$ ili pak je n parni broj strogo veći od 2, algoritam odmah neka vrati vrijednost `False` *bez primjene Miller-Rabinovog testa. Isto tako, napravite listu svih prostih brojeva manjih od* 200 *tako da algoritam ne primjenjuje Miller-Rabinov test na neparne prirodne brojeve manje od* 200 *(uključujući i broj* 2*), nego neka jednostavno ispita da li je takav broj u toj listi i na temelju toga vrati odgovarajuću vrijednost* `True` *ili* `False`*. U svim ostalim slučajevima algoritam neka primjenjuje Miller-Rabinov test.*

Fermatova faktorizacija. Ako se prirodni broj n može napisati u obliku $n = x^2 - y^2$, tj. kao razlika kvadrata dva prirodna broja, tada dobivamo da je $n = (x - y)(x + y)$. Na taj način dobivamo dva, ne nužno prosta, faktora broja n pod pretpostavkom da $x - y$ i $x + y$ nisu trivijalni djelitelji broja n. Slijedi opis algoritma.

Neka je n zadani prirodni broj kojemu želimo naći neki netrivijalni faktor. Na početku stavimo $x_1 = \lceil \sqrt{n} \rceil$ i koristimo rekurziju $x_{i+1} = x_i + 1$. U svakom koraku provjeravamo sljedeća dva uvjeta:

- da li je $y_i = \sqrt{x_i^2 - n}$ prirodni broj, tj. drugim riječima da li je y_i potpuni kvadrat,
- da li su $x_i - y_i$ i $x_i + y_i$ netrivijalni djelitelji broja n.

Ako su oba uvjeta istinita, algoritam završava i vraća netrivijalne faktore $x_i - y_i$ i $x_i + y_i$ broja n. Ako barem jedan od gornjih uvjeta nije istinit, prelazimo na sljedeći korak. Ukoliko se dogodi da je $x_i = n$, tada algoritam nije uspio pronaći neke netrivijalne faktore broja n. U tom slučaju na izlazu vraćamo `False`.

Jasno je da je ovo jedna specijalna metoda faktorizacije koja je jako efikasna ukoliko je prirodni broj n produkt dva bliska prirodna broja. Uočite da se svaki neparni prirodni broj n može napisati kao razlika kvadrata dva prirodna broja. Naime, ako je $n = ab$, tada je $n = \left(\frac{a+b}{2}\right)^2 - \left(\frac{a-b}{2}\right)^2$. Pritom su $\frac{a+b}{2}$ i $\frac{a-b}{2}$ zaista cijeli brojevi zbog toga što su a i b neparni brojevi jer je n neparan broj. Dakle, nije nikakvi problem neparni broj prikazati kao razliku kvadrata dva prirodna broja ukoliko znamo dva njegova faktora. Međutim, problem faktorizacije prirodnog broja je općenito težak problem. No, u slučaju da je neparni prirodni broj produkt dva bliska prirodna broja, tada će spomenuti algoritam uspijeti u razumnom vremenu pronaći dva njegova bliska netrivijalna faktora a i b.

Zadatak 4.
Implementirajte Fermatovu metodu faktorizacije. Na ulazu se zadaje neparni prirodni broj n, a na izlazu se vraćaju dva netrivijalna faktora broja n. Također, implementirajte svoju funkciju koja će za zadani prirodni broj provjeriti da li je on potpun kvadrat ili nije (ta funkcija vam treba za provjeravanje prvog uvjeta od gore spomenuta dva uvjeta). Pazite još na problem vađenja korijena iz jako velikog prirodnog broja. U tu svrhu možete koristiti fantastični modul `gmpy` *umjesto da smišljate svoj algoritam za taj problem.*

Pollardova ρ metoda. Ovo je jedna od najjednostavnijih metoda faktorizacije čija je složenost znatno bolja od običnog dijeljenja i ovisi o najmanjem faktoru broja n. Stoga

i za ovu metodu možemo reći da je specijalna metoda faktorizacije. Algoritam se bazira na sljedećoj ideji za nalaženje nekog netrivijalnog faktora m prirodnog broja n:

- Konstruirati niz (x_i) cijelih brojeva koji je periodičan modulo m.
- Naći i i j za koje je $x_i \equiv x_j \pmod{m}$.
- Konačno je $m = M(|x_i - x_j|, n)$.

Niz (x_i) se konstruira pomoću neke funkcije $f : \mathbb{Z}_n \to \mathbb{Z}_n$ od koje se traži određena "slučajnost". U tu svrhu su dobri kvadratni polinomi, npr. $f(x) = x^2 + 1$. Odabere se neka početna vrijednost x_0, npr. $x_0 = 2$, a ostale vrijednosti se dobivaju kao iterirane vrijednosti funkcije f, tj. $x_{i+1} = f(x_i)$. Jasno je da u beskonačnom nizu $x_0, x_1, x_2, \ldots$ koji poprima samo konačno mnogo vrijednosti, od nekog mjesta se taj niz mora početi periodično ponavljati zbog $x_{i+1} = f(x_i)$.

Kako pronaći neki netrivijalni faktor m prirodnog broja n kada imamo konstruiran niz (x_i)? Ako bismo računali $M(|x_i - x_j|, n)$ za sve i, j, to bi bilo jako neefikasno. Puno bolje je računati samo $M(|x_{2i} - x_i|, n)$. Ova ideja se temelji na Floydovom algoritmu za traženje ciklusa. U tom slučaju za računanje vrijednosti $y_i = x_{2i}$ nije potrebno računati međuvrijednosti $x_{i+1}, x_{i+2}, \ldots, x_{2i-1}$, već se računanje odvija na sljedeći način:

$$x_i = f(x_{i-1}) \bmod n,$$
$$y_i = f\big(f(y_{i-1})\big) \bmod n.$$

Traženje traje tako dugo dok $M(|x_{2i} - x_i|, n)$ ne postane različita od 1. Ako je u tom slučaju $M(|x_{2i} - x_i|, n) \neq n$, tada je $M(|x_{2i} - x_i|, n)$ netrivijalni faktor od n. Ako pak je $M(|x_{2i} - x_i|, n) = n$, tada algoritam nije uspio pronaći neki netrivijalni faktor prirodnog broja n i u tom slučaju treba promijeniti funkciju f i pokušati ponovo sve iz početka s novom funkcijom f. Standarno možemo birati funkcije oblika $f(x) = x^2 + c, c \in \mathbb{Z}\backslash\{0, -2\}$. Uočite da algoritam neće uspijeti ukoliko je na ulazu prosti broj p jer će u tom slučaju uvijek biti $M(|x_{2i} - x_i|, n)$ jednaka 1 ili p. Inače je algoritam vrlo efikasan na brojevima koji imaju male proste faktore. Npr., prosti broj s desetak znamenki u decimalnom zapisu je mali prosti broj, dok je za današnja računala prosti broj s 200 znamenki u decimalnom zapisu veliki prosti broj.

Zadatak 5.
Implementirajte Pollardovu ρ metodu. Algoritam počnite s funkcijom $f(x) = x^2 - 1$. Ako algoritam ne uspije pronaći netrivijalni faktor zadanog prirodnog broja n pomoću trenutne funkcije f, tada neka odabere neku drugu funkciju oblika $f(x) = x^2 + c$ za neki slučajno odabrani prirodni broj $0 < c < n - 2$.

Jedno moguće ubrzanje Pollardove ρ metode bazira se na sljedećoj činjenici:

Ako je $M(a, n) > 1$, tada je $M(ab, n) > 1$ za svaki $b \in \mathbb{N}$.

Stoga, umjesto da se u svakom koraku računa $M(|x_{2i} - x_i|, n)$, definira se varijabla z kao produkt 100 uzastopnih članova oblika $|x_{2i} - x_i|$ modulo n i nakon toga se izračuna $M(z, n)$. Na taj način se najveća zajednička mjera računa tek nakon svakih 100 koraka. Drugim riječima, svakih 100 računanja najvećih zajedničkih mjera, u ovom slučaju zapravo mijenjamo s 99 množenja modulo n i jednim računanjem najveće zajedničke mjere, što

pridonosi povećanju brzine izvođenja algoritma. S druge strane, u ovom slučaju je veća vjerojatnost da se dogodi da je $M(z,n) = n$ pa algoritam neće uspijeti naći netrivijalni faktor broja n. U tom slučaju možemo probati smanjiti broj 100, na primjer na 50, tj. da svakih 50 koraka računamo najveću zajedničku mjeru i u slučaju neuspjeha ponovo smanjiti broj 50 na pola i tako dalje redom. U najgorem slučaju ponovo ćemo doći do obične Pollardove ρ metode.

Zadatak 6.
Implementirajte modificiranu Pollardovu ρ metodu za traženje netrivijalnog faktora zadanog prirodnog broja n. Algoritam neka počne s funkcijom $f(x) = x^2 - 1$ i neka nakon svakih 100 koraka računa najveću zajedničku mjeru $M(z,n)$. Ukoliko algoritam ne uspije pronaći netrivijalni faktor broja n, tada se automatski kreće ispočetka s novim korakom računanja najvećih zajedničkih mjera $M(z,n)$ koji je upola manji od prethodnog koraka (u ovom slučaju svakih 50 koraka). U slučaju ponovnog neuspjeha, automatski se kreće ispočetka s upola manjim korakom računanja najvećih zajedničkih mjera $M(z,n)$ od prethodnog (u ovom slučaju svakih 25 koraka). Općenito, u slučaju neuspjeha, algoritam neka krene automatski ispočetka s upola manjim korakom računanja najvećih zajedničkih mjera $M(z,n)$ od prethodnog koraka. Ukoliko algoritam ne uspije pronaći netrivijalni faktor broja n niti nakon što dođe do obične Pollardove ρ metode (gdje se u svakom koraku računa najveća zajednička mjera), tada algoritam bira novu funkciju f oblika $f(x) = x^2 + c$ za neki slučajno odabrani prirodni broj $0 < c < n - 2$ i algoritam ponavlja ponovo istu priču s novom funkcijom f.

Pollardova $p-1$ metoda. Ovo je također jedna specijalna metoda faktorizacije koja se bazira na malom Fermatovom teoremu prema kojem je $a^{p-1} \equiv 1 \pmod{p}$ za svaki prosti broj p i prirodni broj a za koji je $M(a,p) = 1$. Tada je također $a^m \equiv 1 \pmod{p}$ za svaki višekratnik m broja $p-1$. Iz toga slijedi $a^m - 1 \equiv 0 \pmod{p}$ pa zaključujemo da je p faktor od $a^m - 1$. Na posljednjem zaključku se zapravo bazira ova metoda. Naime, ugrubo rečeno, ukoliko želimo pronaći neki netrivijalni faktor prirodnog broja n, tada gledamo brojeve oblika $a^t - 1$ i provjeravamo da li neki od njih ima zajednički faktor s n. Ukoliko ima, uspjeli smo pronaći neki netrivijalni faktor broja n.

Pitamo se kako u praktičnoj implementaciji birati brojeve oblika $a^t - 1$. Postoji više varijanti, a mi ćemo ovdje u opisu algoritma spomenuti jednu takvu varijantu koja ne zahtijeva puno dodatne teorije. Najčešće se uzima $a = 2$ (možemo pretpostaviti da je n neparan broj jer iz svakog parnog broja lagano izlučimo njegov parni dio i svedemo problem na neparni broj), a onda postoji više varijanti na koje načine možemo birati brojeve t.

U kojem slučaju je ova metoda uspješnija od ostalih metoda? Zapravo sve ovisi o broju $p-1$. Prosti faktor p je možda teško pronaći, međutim $p-1$ je potpuno drugačiji broj od broja p. Ukoliko su svi prosti faktori broja $p-1$ mali, tada će ova metoda uspjeti efikasno pronaći netrivijalni faktor zadanog broja n kojemu je p neki prosti faktor (koji ne mora biti mali broj). Ako odaberemo dobar t, tada će brojevi t i $p-1$ imati puno zajedničkih faktora i upravo na dobrom odabiru broja t počiva efikasnost algoritma (jasno, uz pretpostavku da su svi prosti faktori od $p-1$ mali).

U nastavku slijedi opis jedne varijante algoritma. Neka je n zadani neparni prirodni broj kojemu želimo naći neki netrivijalni faktor. Stavimo da je $a = 2$. Za $k \in \mathbb{N}$ gledamo brojeve oblika $x_k = \left(2^{k!} - 1\right) \bmod n$. U svakom koraku računamo $r_k = M(x_k, n)$. Ako je

$r_k \neq 1$ i $r_k \neq n$, tada je r_k netrivijalni faktor broja n i algoritam završava i vraća brojeve r_k i $\frac{n}{r_k}$. Ukoliko to nije slučaj, algoritam prelazi na sljedeći korak.

Zadatak 7.
Implementirajte opisanu Pollardovu $p-1$ metodu. Za uspješnu implementaciju potreban je algoritam za modularno potenciranje, tj. efikasno računanje izraza $2^{k!} \bmod n$. `Python` *već sadrži gotovu naredbu* `pow` *za ovaj problem koju slobodno koristite tako da ne morate implementirati algoritam za modularno potenciranje.*

Konačno, posljednja dva algoritma koje trebate implementirati se odnose na faktorizaciju prirodnog broja.

Zadatak 8.
Implementirajte algoritam za faktorizaciju malih prirodnih brojeva koji je baziran na implementiranom algoritmu u 2. zadatku. Ako $n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ faktorizacija prirodnog broja n na proste faktore, tada algoritam na izlazu mora vratiti rječnik

$$\{p_1 : \alpha_1,\, p_2 : \alpha_2,\, \ldots,\, p_k : \alpha_k\}.$$

Ako je n prosti broj, algoritam vraća rječnik $\{n : 1\}$. Ako je $n = 1$, algoritam vraća prazni rječnik $\{\}$.

Zadatak 9.
Implementirajte algoritam za faktorizaciju prirodnog broja n koji je baziran na već implementiranim algoritmima u zadacima 2, 3, 6 i 8. U nastavku slijedi opis načina rada algoritma.

- *Ako je $n \leqslant 10^{10}$, automatski se koristi implementirani algoritam u 8. zadatku.*
- *Ako je $n > 10^{10}$, tada se koriste algoritmi implementirani u zadacima 3 i 6. Modificirana Pollardova ρ metoda služi da pronađete neki prosti faktor broja n, a Miller-Rabinov test koristite za ispitivanje prostosti nekog broja. Pazite, Pollardova ρ metoda ne daje nužno prosti faktor promatranog broja n, već samo neki njegov faktor. Na vama je da osmislite kako pomoću (modificirane) Pollardove ρ metode, algoritma iz 2. zadatka i Miller-Rabinovog testa pronaći neki prosti faktor broja n. Nakon što algoritam pronađe neki prosti faktor p_1 zadanog broja n, tada ispituje koliko se puta taj faktor nalazi u broju n, tj. računa α_1. Dobivene podatke zapisuje u rječnik i nastavlja dalje faktorizirati broj $\frac{n}{p_1^{\alpha_1}}$ na gore već opisani način. Nakon što algoritam pronađe neki prosti faktor p_2 broja $\frac{n}{p_1^{\alpha_1}}$, izračuna α_2 i dobivene podatke spremi u rječnik. Dalje se nastavlja s faktorizacijom broja $\frac{n}{p_1^{\alpha_1} p_2^{\alpha_2}}$. I tako dalje redom. Ako se dođe do koraka u kojem dalje trebati nastaviti s faktorizacijom broj $m \leqslant 10^{10}$, tada se algoritam automatski prebacuje na implementirani algoritam u 8. zadatku pomoću kojeg se dovršava faktorizacija.*

Pazite da uvijek prije primjene Pollardove ρ metode provjerite da ju ne primjenjujete na prostom broju jer na prostim brojevima ta metoda sigurno ne funkcionira. U slučaju da algoritam ne uspijeva faktorizirati zadani prirodni broj n (dugo se izvodi), tada korisnik sam prekida izvođenje algoritma. Što znači "algoritam se dugo izvodi"? Riječ "dugo" može svašta značiti: deset minuta, dva sata, tri dana,... Na samom je korisniku da odluči

koliko će vremena dati algoritmu za rad. Ukoliko algoritam uspije faktorizirati zadani prirodni broj n, tada na izlazu vraća rječnik kako je to već opisano u 8. zadatku.

Na kraju još nekoliko praktičnih zadataka na kojima ćete testirati svoje implementirane algoritme.

Zadatak 10.
U svakoj od datoteka `broj1.txt`, `broj2.txt`, `broj3.txt`, `broj4.txt`, `broj5.txt`, `broj6.txt`, `broj7.txt`, `broj8.txt` *nalazi se spremljen po jedan prirodni broj. Na svakom od tih brojeva testirajte svoje implementirane algoritme iz zadataka 2.-9. Ukoliko se u nekom slučaju neki od algoritama dulje izvodi, sami procijenite da li će uspjeti i koliko mu vremena treba da obavi posao (dajte svakom barem desetak minuta šanse, ili više ili možda manje, ovisno već o vašim procjenama, teoretskom znanju o uspješnosti pojedinog algoritma, ali isto tako i o kvaliteti vaših implementacija).*

Zadatak 11.
U svakoj od datoteka `P1.txt` *i* `P2.txt` *nalazi se spremljen po jedan prirodni broj. Ispitajte da li je neki od tih brojeva prost. U slučaju da broj nije prost, pokušajte pronaći njegovu faktorizaciju na proste faktore pomoću svojih implementiranih algoritama.*

Zadatak 12.
Vaš poslodavac vas želi poslati na jedan tajni zadatak. Vi ste jedan od njegovih najboljih tajnih agenata. Međutim, za ovaj zadatak je potrebno imati i mnoge vještine dobrog matematičara. Naime, neprijatelj postaje sve opasniji i matematički obrazovaniji, vremena je sve manje, a sudbina svijeta ovisi o vašem znanju o RSA kriptosustavu. Neprijatelj šifrira važne poruke upravo RSA kriptosustavom. Stoga ste dobili probni zadatak da pokušate razbiti RSA šifrat čiji je javni ključ spremljen u datotekama `N0.txt` *i* `E.txt`*. Šifrat je spremljen u datoteci* `kod0.txt`*. Dajte sve od sebe jer ako Vi ne uspijete, poslodavac će poslati nekog drugog na tajni zadatak.*

Zadatak 13.
Opravdali ste status najboljeg tajnog agenta i uspješno riješili problem iz prethodnog zadatka. No, sada ste na ozbiljnom tajnom zadatku. Od vas se puno očekuje. Uspjeli ste neprijatelju ući u trag i dokopati se šifrirane poruke koja je spremljena u datoteci `kod1.txt`*. Također, saznali ste da je šifrat šifriran javnim ključem koji je spremljen u datotekama* `N1.txt` *i* `E.txt`*. U njemu je važna poruka o tome na koje mjesto trebate doći kako biste iznenadili neprijatelja i pokupili daljnje važne informacije. Kako je neprijatelj bio neoprezan prilikom šifriranja i potcijenio vaše matematičko znanje, to vama daje prostora da razbijete taj šifrat i ukradete mu važne informacije tako da odete na mjesto sastanka koje se spominje u šifriranoj poruci. Dajte sve od sebe.*

Zadatak 14.
Vaš poslodavac je prezadovoljan s vama. Uspješno ste prebrodili sve prepreke i do sada razbili sve šifrirane poruke. Međutim, to nije kraj. Neprijatelj ipak ne posustaje unatoč unutarnjim nesuglasicama zbog neuspjeha. Došli ste na loš glas u neprijateljskim krugovima i sada prate svaki vaš korak. Vi ste svjesni toga, ali svaka vaša pogreška može vas skupo koštati. Ovaj put oni vas planiraju iznenaditi na jednom od odredišta koja morate posjetiti. No, ne znate na kojem vas točno odredištu namjeravaju zaskočiti. Neprijatelj i

dalje hrabro komunicira preko šifriranih poruka ne znajući za vaše hakerske sposobnosti koje su na zavidnom nivou. Ponovo ste se dokopali njihove šifrirane poruke koju su slali preko nesigurnog komunikacijskog kanala. U toj poruci se između ostalog spominje mjesto na kojem vas namjeravaju napasti. Želite li se spasiti, ne preostaje vam ništa drugo, nego da razbijete šifriranu poruku koje ste se dočepali. Šifrirana poruka je spremljena u datoteci `kod2.txt`*. Također, saznali ste da je šifrirana javnim ključem koji je spremljen u datotekama* `N2.txt` *i* `E.txt`*. Što još reći? Vaša sigurnost je u pitanju. To je valjda dovoljna motivacija za razbijanje šifrata.*

Zadatak 15.
Na kraju, nakon svih vaših uzbuđenja u ulozi tajnog agenta, čeka vas samo još jedan zadatak. Posljednji šifrat kojeg trebate razbiti. Do sada ste stekli već puno iskustava pa ne bi to trebao biti neki problem. Šifrat je spremljen u datoteci `kod3.txt`*, a šifriran je javnim ključem koji je spremljen u datotekama* `N3.txt` *i* `E.txt`*. Pritom su datoteke* `kod3.txt` *i* `N3.txt` *spremljene u šifriranu datoteku* `kod3N3.zip`*. Šifru za* `zip` *datoteku ste saznali u prethodnim zadacima. Želite li saznati završetak priče, prionite na posao. Sigurno ste jako znatiželjni, što se na kraju dogodilo.*

Dodatne upute. Sve implementacije u prvih devet zadataka treba napraviti u programskom jeziku `Python 2.x` kako je već ranije rečeno. Pritom ne smijete koristiti nikakve eventualne gotove funkcije koje možda automatski rješavaju zadani problem kako biste pomoću njih izbjegnuli svoju implementaciju (osim onoga što je eksplicitno navedeno da se eventualno smije koristiti). No, kako bi se vidjelo da ste zaista svoje algoritme testirali na preostalim praktičnim zadacima, sve testove treba napraviti unutar `IPython` notebooka. Dakle, svoje implementacije funkcija spremite u jednu `py` datoteku (ta datoteka neka ne bude izvršna datoteka, nego neka samo sadrži implementirane funkcije). Tu datoteku učitajte u `IPython` notebook kako biste unutar njega mogli koristiti svoje implementirane funkcije. Isto tako, sve preostale priložene datoteke u kojima su spremljeni određeni brojevi također učitajte u svoj notebook pomoću `Python` funkcija za rad s datotekama. Dakle, nemojte copy-pejstati brojeve iz datoteka u notebook, nego ih preko gotovih `Python` funkcija za rad s datotekama učitajte u notebook, pretvorite u odgovarajući format i spremite u neke varijable preko kojih ćete pristupati tim brojevima. Pomoću gotove `IPython` funkcije `%time` mjerite koliko je vremena potrebno pojedinom algoritmu da se izvrši na pojedinom broju.

Što se tiče rješavanja zadnja četiri zadatka u kojima trebate razbiti RSA šifrate, princip rješavanja je sljedeći. Za razbijanje šifrata trebate koristiti svoje implementirane `Python` funkcije. Dakle, vaš postupak dobivanja tajnog dijela ključa mora biti jasno vidljiv u `IPython` notebooku. Samo dešifriranje obavite pomoću `Python` programa `guiRSA.py`. Način funkcioniranja tog programa opisan je u datoteci `rsa.pdf`. Nakon što razbijete šifrat, pomoću `Python` funkcija za rad s datotekama učitajte dešifrirani tekst iz datoteke u `IPython` notebook.

Napomena. Program `guiRSA.py` je zapravo prilagođen za rad na `ubuntu linuxu`. Ova implementacija neće ispravno raditi na `windowsima`.

Bodovanje. Zadaća nosi maksimalno 5 bodova, a ugrubo je princip bodovanja sljedeći:

riješeni zadaci	broj bodova
1., 2., 8.	1
3., 4., 7.	1
5., 6., 9.	1
12., 13., 14., 15.	1
`IPython` notebook	1

Pritom se pretpostavlja da ukoliko riješite samo neke od prvih devet zadataka da svakako svoje funkcije koje ste napravili testirate na 10. i 11. zadatku ukoliko se u tim dvama zadacima traži testiranje tih funkcija. Na primjer, ako riješite 1., 2. i 8. zadatak, tada je u taj 1 bod uključeno i testiranje tih funkcija na 10. i 11. zadatku, ukoliko to ti zadaci zahtijevaju. Slično je i za sve ostale kombinacije. Na primjer, ako ste riješili samo 3. i 5. zadatak, tada je u bodovanje uključeno i testiranje tih funkcija na 10. i 11. zadatku. Jedan bod dobivate za sam izgled `IPython` notebooka da sve bude uredno formatirano, jasno napisano, slično `IPython` materijalima na `moodlu`. Na primjer, ukoliko želite napisati neki svoj komentar, nemojte tekst direktno pisati u input ćeliju, već pretvorite ćeliju u tekst format. Radi lakše orijentacije, u datoteci `primjer.html` je pokazano testiranje funkcija na jednom broju.

Moodle. Na `moodle` treba predati sljedeće datoteke:

- `ipynb` datoteku – datoteka za `IPython` notebook
- `html` datoteku – `html` verzija `ipynb` datoteke
- `py` datoteku – datoteka u kojoj su vaši implementirani algoritmi
- sve ostale datoteke i direktorije koji su potrebni kod izvršavanja pojedinih naredbi u `IPython` notebooku ili kod čitanja `html` datoteke.

Najbolje je da sve spomenute datoteke i direktorije spremite u jednu `zip` datoteku i nju onda predate na `moodle`.

Napomena. Vaša zadaća se bude pregledavala na `linuxu` pa se preporuča korištenje tog operacijskog sustava prilikom izrade ove zadaće. Ukoliko ipak radite na `windowsima`, molimo vas da na kraju testirate svoju zadaću na `linuxu` prije nego što ju predate na `moodle`. Imajte na umu da se vaša zadaća boduje prema tome koliko dobro radi na `linuxu`.