TDT4113 - Datateknologi, programmeringsprosjekt

Oppgave 3: Koder

Dette dokumentet beskriver den tredje oppgaven i ProgLab 2, som vi kaller "Koder". For denne oppgaven gjelder at:

- Oppgaven skal løses individuelt
- Oppgaven skal løses med objektorientert kode skrevet i Python
- Fristen for oppgaven er *2 uker*, dvs. implementasjonen din leveres på its learning senest 4. oktober kl 12:00 og demonstreres senest 4. oktober kl 14:00.

NB! Les hele oppgaveteksten før du begynner på implementasjonen. Funksjonaliteten i systemet beskrives skrittvis, men det kan være lurt å se "hele pakka" før du velger designet. Husk at du må bruke et OO-design for å få prosjektet godkjent.

1 Bakgrunn for oppgaven

I denne oppgaven skal du implementere kryptografi-algoritmer for å *kryptere* en tekst, og for å *dekryptere* teksten igjen. I tillegg skal du implementere en "brute force" metode for å forsøke å bryte/hacke krypteringen.

Kryptering på et skjematisk nivå er vist i Figure 1. En sender og en mottaker ønsker å kommunisere over en ikke sikker kommunikasjonslinje (for eksempel på mail). De to er enige om noen *krypteringsnøkler* samt en algoritme for å bruke nøklene til å kryptere meldingen. Senderen tar den originale meldingen i klartekst, sender den gjennom krypteringsalgoritmen med sin nøkkel og får som resultat en kryptert tekst. Mottager får den krypterte teksten, og bruker sin nøkkel for å dekryptere. Resultatet er at mottaker får dokumentet tilbake i klartekst, uten at den direkte lesbare meldingen er eksponert på kommunikasjonslinjen.

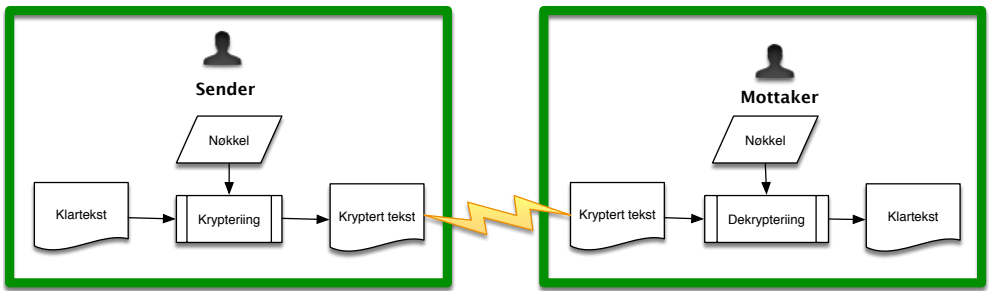

Figure 1: Forenklet skjematisk beskrivelse av kryptografi

Vi skal se på et antall forskjellige krypterings-algoritmer av forskjellig vanskelighetsgrad. Algoritmene baserer seg på et sett av lovlige tegn teksten kan inneholde (for eksempel alfabetet). I denne beskrivelsen vil jeg tenke på alfabetet som de 26 bokstavene som brukes i engelsk, alle skrevet med stor bokstav. Dette er gjort for at vi skal kunne se på tegnene og enkelt gjenkjenne dem (og så passer de inn på en linje, så det gjør det enklere å lage dette dokumentet). I implementasjonen skal du bruke et større alfabet, nemlig alle tegn med ASCII-verdier fra og med 32 (mellomrom) til og med 126 (tegnet "~").

Krypteringsnøklene er det som gjør sender og mottaker i stand til å kommunisere. Som vi skal se senere vil sender og mottaker ikke nødvendigvis ha samme nøkkel, men de to nøklene står i samsvar til hverandre. Hvordan dette gjøres avhenger av hvilken krypteringsmetode som er i bruk. Hvis en tredjepart får tilgang til krypteringsnøklene og vet hvilken algoritme som er i bruk kan han selvsagt bryte krypteringen, noe vi kommer tilbake til senere.

2 Foreslått kode-struktur

Dere skal implementere flere ciphere, så det er nyttig å definere en superklasse Cipher som har implementasjon av de forskjellige krypterings-algoritmene som sub-klasser. Klassen bør holde informasjon som deles av alle cipher, for eksempel det lovlige alfabetet. Det er også nyttig å representere eksplisitt at alfabetet totalt har 95 tegn, slik at alle mod-operasjonene skal gjøres med mod 95 i implementasjonen deres (mens jeg altså bruker mod 26 i dette dokumentet). I tillegg er det nyttig å legge dummy-versjoner av de nødvendige metodene her, for eksempel encode og decode. Lag også metoden verify, som får klar-tekst inn, koder klar-teksten, dekoder cipher-teksten, og verifiserer at den dekodede teksten er identisk med den initielle klar-teksten. Det er også en mulighet å lage en metode

generate_keys som genererer krypteringsnøkler som skal distribueres til sender og mottaker.

Vi har to eller tre "personer" involvert i denne oppgaven: En som sender meldingen, en som mottar den, og evt. en som forsøker å hacke meldingen. Implementer derfor klassen Person. En instans av denne klassen har en nøkkel (tilgang til verdien av self.key) og en cipher-algoritme (instans av en av sub-klassene til Cipher) som han kan jobbe med. Klassen trenger derfor metodene set_key, get_key, og operate_cipher. Lag sub-klassene Sender og Receiver, og tilpass sub-klassenes bruk av self.operate_cipher() slik at senderen genererer cipher-tekst, og mottakeren genererer klar-tekst. En Hacker-klasse er også nyttig å lage; denne detaljeres mer senere. For bedre å forstå hvordan du kan fylle disse klassene med innhold er det lurt å lese hele dette dokumentet først, slik at du får et fullstendig bilde av funksjonaliteten som skal implementeres. For din egen del kan det være nyttig med et UML-diagram også, men dette kreves ikke for å bestå oppgaven.

I denne oppgaven kan du fritt benytte hjelperutinene i filen `cypto-util.py`. Før du evt. bruker noe derfra er det viktig at du ser på koden og forstår hvordan den virker slik at den ikke gir deg feil i programmet ditt.

Implementasjon – Del 1:
Implementer klassene Cipher, Person, Sender, og Receiver.

3 Forskjellige cipher

3.1 Caesar

Den første koden som skal implementeres er det såkalte Caesar-chipheret. Dette cipheret benytter en hemmelig nøkkel gitt som et heltall, for eksempel 2. Ut fra denne nøkkelen lages en tabell som denne (som er spesiell for nøkkelen vi valgte):

Caesar-cipher med nøkkel 2																									
Klar-tekst:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Tall-verdi:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Kodet verdi:	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	0	1
Kodet tekst:	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B

I den øvesrte linjen står alle bokstavene i alfabetet, fulgt av en linje med tall (starter på null, øker med en for hver bokstav). Disse tallverdiene representerer bokstavene. For å kode en bokstav, for eksempel "P" slår vi opp hvilken tallverdi denne har. Her er svaret at "P" gir verdien 15. Så *legger vi til* verdien av nøkkelen, og får kodet verdi, gitt i den tredje linjen. Siden vi valgte nøkkel 2 blir

svaret 15+2=17. Endelig slår vi opp hvilken bokstav dette tallet tilsvarer i den fjerde linjen, og finner at for verdien 17 tilsvarer bokstaven "R".

Om vi har lyst til å kode ordet "PYTHON" blir svaret med nøkkel 2 "RAVJQP".

For å dekode en melding kan vi tenke oss at vi går vi fra nederste linje og oppover i samme tabell. Si at vi har fått den kodede teksten "MQFG". For å de-kryptere denne strengen starter vi med bokstaven "M" i linjen for kodet tekst. Denne bokstaven tilsvarer kodet verdi 12, som gir tall-verdi for klartekst 10, og skal derfor "oversettes" til bokstaven "K". Videre blir "Q" til "O", "F" blir til "D", og "G" blir til "E" slik at "MQFG" dekrypteres enkelt til "KODE".

Hvis vi ser på krypteringstabellen ser vi at kodet verdi er beregnet som original tallverdi pluss verdien for nøkkel for bokstavene fra "A" til og med "X". Når vi kommer til "Y", som har tallverdi 24 er det vanskeligere, fordi tall-verdien $24 + 2 = 26$ ikke tilsvarer noen bokstav i alfabetet vårt. I stedet for å bare legge til nøkkelverdien når vi går fra original tall-verdi til kodet tallverdi er operasjonen derfor å ta summen *modulo 26* – ettersom det er 26 bokstaver i alfabetet. Når du implementerer med det lengre alfabetet skal du selvsagt bruke modulo 95 i stedet.

Det å *dekryptere* dette cipheret er egentlig det samme som å kryptere en melding, bare vi velger riktig nøkkel. Dette er krypteringstabellen for nøkkel 24, som viser seg å virke:

Caesar cipher med nøkkel 24																									
Klar-tekst:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Tall-verdi:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Kodet verdi:	24	25	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Kodet tekst:	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X

La oss late som om vi ønsker å kryptere teksten "MQFG', altså kryptere på nytt den samme teksten som vi fikk som kodet versjon av ordet "KODE". Vi får nå at "M" $\rightarrow$ "K", "Q" $\rightarrow$ "O", "F" $\rightarrow$ "D", og "G" $\rightarrow$ "E", så vi er tilbake til det samme ordet som vi startet med, og vi ser at kryptering først med nøkkel 2, deretter med nøkkel 24 gir oss klarteksten tilbake. Med andre ord er dekryptering av koden det samme som å kryptere med nøkkel 24. Grunnen til at dette virker er at hvis man har en bokstav med tall-verdien x, legger til 2 modulo 26 (første kryptering) og deretter legger til 24 modulo 26 kommer vi tilbake der vi startet:

$$(x + 2 \bmod 26) + 24 \bmod 26 = x + 26 \bmod 26 = x.$$

Generelt gjelder for et alfabet med 26 tegn at hvis en tekst er kodet med nøkkel n, $1 \leq n \leq 25$ kan vi dekryptere den med å gjøre samme operasjon, men med nøkkel $26 - n$. Og, joda, man kan selvsagt også bruke -2 som nøkkel i dekrypteringen, fordi $x + 2 + (-2) \bmod 26 = x$, og $-2 \bmod 26 = 24$ holder jo også. Det er vanlig å bruke positive tall som nøkler, derfor går vi for 24 og ikke -2 i denne beskrivelsen.

Implementasjon – Del 2:
Lag klassen Caesar som en sub-klasse av Cipher. Dummy-metodene fra superklassen (encode, decode, generate_keys, etc.) skal implementeres for Caesar-cipheret. Verifiser at det virker ved å bruke verify(). Test hele infrastrukturen ved å la en sender og en receiver dele nøkler, la sender kode meldingen og gi cipher-tekst til receiver, og la receiver dekode meldingen.

3.2 Multiplikasjons-cipher

Caesar-cipheret er enkelt og greit en operasjon der vi tar hver bokstavs tall-verdi og legger til et tall. Multiplikasjons-cipheret, som blir beskrevet nå, er mye godt det samm e, men som det ligger i navnet er addisjonen nå erstattet med en multiplikasjon. Tabellen for multiplikasjons-cipher med nøkkel 3 ser dermed slik ut; merk at vi også her gjør operasjonene modulo 26, slik at for eksempel "K", som har verdien 10 ikke blir oversatt til verdien $10 \cdot 3 = 30$, men til $10 \cdot 3 \mod 26 = 4$, altså bokstaven "E":

Multiplikasjons-cipher med nøkkel 3																									
Klar-tekst:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Tall-verdi:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Kodet verdi:	0	3	6	9	12	15	18	21	24	1	4	7	10	13	16	19	22	25	2	5	8	11	14	17	20	23
Kodet tekst:	A	D	G	J	M	P	S	V	Y	B	E	H	K	N	Q	T	W	Z	C	F	I	L	O	R	U	X

Dersom vi bruker denne tabellen til å kryptere meldingen "KODE" ender vi med "EQJM".

For å dekryptere en melding er det naturlig å tenke at vi tar tall-verdien av en bokstav og *deler* med nøkkel-verdien, ettersom vi ganget med denne under krypteringen. Det virker dessverre ikke. For eksempel har "E" verdien 4, og om vi deler med nøkkelverdien 3 får vi ikke et heltall. I stedet skal vi lete etter en ny nøkkelverdi som kan brukes til dekryptering, og som passer med den nøkkelen som ble brukt for å kryptere. Dersom vi kaller den nye nøkkelen m, krever vi at det å først ganger et tall x med den originale nøkkelen n og deretter med den nye nøkkelen m, så skal det gi oss tilbake x, dvs. at

$$(x \cdot n \bmod 26) \cdot m \bmod 26 = x$$

for alle $x = 0, 1, \ldots, 25$. Dette kan også skrives som at $n \cdot m = 1 \bmod 26$, og verdien av m kalles modulo-invers til n. Det er litt komplekst å finne hvilken verdi som er modulo-invers til en nøkkel, men du kan fritt benytte `modular_inverse` i filen `cypto-util.py`.

Det viser seg at om man krypterer med nøkkel $n = 3$ kan man dekryptere med å multiplisere med nøkkelen $m = 9$. Dette paret passer med definisjonen fordi $n \cdot$

$m \bmod 26 = 9 \cdot 3 \bmod 26 = 27 \bmod 26 = 1$. Den tilsvarende krypteringstabellen ser slik ut:

Multiplikasjons-cipher med nøkkel 9																									
Klar-tekst:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Tall-verdi:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Kodet verdi:	0	9	18	1	10	19	2	11	20	3	12	21	4	13	22	5	14	23	6	15	24	7	16	25	8	17
Kodet tekst:	A	J	S	B	K	T	C	L	U	D	M	V	E	N	W	F	O	X	G	P	Y	H	Q	Z	I	R

Dette fungerer, fordi vi får at "E" $\rightarrow$ "K", "Q" $\rightarrow$ "O", "J" $\rightarrow$ "D", og "M" $\rightarrow$ "E". De-kryptering når krypteringsnøkkelen var 3 er altså det samme som å kryptere igjen, men med nøkkel 9.

Multiplikasjons-cipher fungerer bare for noen nøkler. Kravet for at en nøkkel n skal virker er at når for to tall x_1 og x_2 så er $n \cdot x_1$ modulo $26 = n \cdot x_2$ modulo 26 bare mulig dersom x_1 modulo $26 = x_2$ modulo 26. Her ser du hva som skjer når vi benytter nøkkel 2:

Multiplikasjons-cipher med nøkkel 2																									
Klar-tekst:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Tall-verdi:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Kodet verdi:	0	2	4	6	8	10	12	14	16	18	20	22	24	0	2	4	6	8	10	12	14	16	18	20	22	24
Kodet tekst:	A	C	E	G	I	K	M	O	Q	S	U	W	Y	A	C	E	G	I	K	M	O	Q	S	U	W	Y

Ettersom både "A" og "N" blir kodet med bokstaven "A" i kodet tekst er kodingen ikke unik, og det er ikke mulig å entydig dekryptere meldingen. Problemet er at ettersom "A" har tallverdi 0 og "N" har tallverdi 13 gir de begge samme kodete tallverdi ($2 \cdot 0$ modulo $26 = 0$ for bokstaven "A", mens bokstaven "N" også gir $2 \cdot 13$ modulo $26 = 26$ modulo $26 = 0$). Når cipher-teksten inneholder bokstaven "A" vet vi dermed ikke om vi skal "oversette" denne med "A" eller "N", og dekodingen er ikke unik. En nøkkel n virker bare dersom den har en modulo invers, dvs. det finnes en m slik at $n \cdot m = 1 \mod 26$.

Implementasjon – Del 3:
Implementer cipheret i klassen Multiplicative, som en sub-klasse av Cipher. Pass spesielt på at metoden som genererer nøkler oppfører seg riktig.

3.3 Affine cipher

Et problem med muliplikasjons-cipheret er at den alltid koder en "A" som en "A" (dersom tall-verdien til "A" er 0, vil den kodede verdien være $0 \cdot n = 0$ når vi enkoder med nøkkel n). Det er derfor greit å lage en kombinasjon av multiplikasjons-cipher og Caesar-cipher, kalt *affine cipher*. Denne bruker et tuple

av to heltall som nøkkel, $n = (n_1, n_2)$, der n_1 brukes til multiplikasjons-cipheret og n_2 til Caesar-cipheret.

Her ser du resultatet av enkoding med nøkkel $n = (3, 2)$. Vi bruker først multiplikajson med nøkkel 3, deretter Caesar med nøkkel 2.

Affine cipher med nøkkel $(3, 2)$																									
Klar-tekst:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Tall-verdi:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Kodet verdi:	2	5	8	11	14	17	20	23	0	3	6	9	12	15	18	21	24	1	4	7	10	13	16	19	22	25
Kodet tekst:	C	F	I	L	O	R	U	X	A	D	G	J	M	P	S	V	Y	B	E	H	K	N	Q	T	W	Z

Når man dekoder er det viktig å gjøre operasjonene i motsatt rekkefølge, altså først Caesar med nøkkel 24 (som dekoder bruken av nøkkel $n_2 = 2$), deretter multiplikasjons-cipher med nøkkel 9 (som dekoder bruken av nøkkel $n_1 = 3$).

Implementasjon – Del 4:
Implementer cipheret i klassen Affine, som en sub-klasse av Cipher.

3.4 "Ubrytelige" cipher

Det er rimelig greit å hacke disse cipherne for eksempel ved å se på hvor ofte hver bokstav forekommer i cipherteksten, og sammenligne av med hva som er vanlig i naturlig språk.[1] For å gjøre koden vanskeligere å bryte fant man på den såkalt "ubrytelige" koden på 1700-tallet, der man i stedet for å ha en enkelt nøkkel-verdi har et *nøkkel-ord*. La oss bruke ordet "PIZZA", og la oss se hvordan vi kan kode meldingen "HEMMELIGHET".

Det første vi gjør er å skrive ut klarteksten, og oversette den til tallverdier. Deretter repeterer vi nøkkelordet så mange ganger vi trenger for å komme opp på lengden til meldingen (i eksempelet trenger vi litt over 2 ganger PIZZA), og oversetter bokstavene til *sine* tallverdier. Tallverdiene legges sammen (mod 26), som gir raden med kodede tall-verdier. Disse tallene gjøres om til bokstaver, og gir den kodede teksten "WMLLEAQFGEI". Legg merke til at de to E'ene i klartekst-ordet blir kodet med hhv. "M" og "E", så det er ikke lenger sånn at en bokstav oversettes til en fast kode.

"Ubrytelig" cipher med nøkkelord PIZZA											
Klar-tekst:	H	E	M	M	E	L	I	G	H	E	T
Tall-verdi:	7	4	12	12	4	11	8	6	7	4	19
Nøkkelord:	P	I	Z	Z	A	P	I	Z	Z	A	P
Nøkkel-verdi:	15	8	25	25	0	15	8	25	25	0	15
Kodet verdi:	22	12	11	11	4	0	16	5	6	4	8
Kodet tekst:	W	M	L	L	E	A	Q	F	G	E	I

[1] Se for eksempel her: `https://en.wikipedia.org/wiki/Letter_frequency`.

For å dekryptere koden trenger vi et annet kode-ord som "matcher" det vi brukte for å lage ciphertekst. Kodeordet genereres ved at vi posisjon for posisjon bytter symbolet som står der med et annet. Vi finner riktig symbol ved følgende lille øvelse: Finn tall-verdien til symbolet i det originale kode-ordet, la oss kalle denne n. Da skal dekrypteringsnøkkelen på denne plassen ha symbolet som har tallverdi $26 - n \mod 26$. Det betyr at dekrypteringsnøkkelen for "PIZZA" blir "LSBBA".

Implementasjon – Del 5:
Implementer cipheret i klassen Unbreakable, som en sub-klasse av Cipher.

3.5 RSA

Alle cipherne vi har implementert så langt har den svakheten at sender og mottaker må "matche" nøklene sine, og dersom en av nøklene blir kjent for en tredjepart kan denne enkelt hacke den kodede meldingen. Dette er ikke tilfellet for RSA-cipheret. For RSA-kryptering offentliggjør mottaker hvilken nøkkel han ønsker at skal brukes for å kryptere meldinger han skal motta. Dette er "ufarlig" fordi det er vanskelig (komputasjonelt sett en NP-hard oppgave) å finne dekrypteringsnøkkelen som passer til denne. De matematiske detaljene for å se at RSA cipheret virker er litt langhåret, så vi skal kun se på hvordan den implementeres her.

3.5.1 Krypteringsnøkler

Det første skrittet er å finne krypterings-nøkler. Mottaker gjør følgende:

1. Generer to tilfeldige primtall p og q,. For å implementere denne jobben kan du benytte funksjonen `generate_random_prime` i filen `cypto-util.py` om ønskelig. Metoden tar b (antall bit) som input, og genererer et primtall som er mellom 2^b og 2^{b+1}.
2. Definer $n = p \cdot q$ og $\phi = (p-1)(q-1)$.
3. Velg e som et tilfeldig heltall som er større enn 2 og mindre enn ϕ. Her kan `random.randint(3, phi-1)` benyttes.
4. Sett d som modulo-invers til e med hensyn på ϕ, dvs. d velges slik at $d \cdot e = 1 \mod \phi$. Husk at `modular_inverse` finnes i filen `cypto-util.py`.
5. Nå er nøklene ferdig generert:

- (n, e) er nøkkelen som sender skal bruke for å kryptere meldinger som skal sendes til denne mottakeren. Mottaker kan fritt offentliggjøre nøkkelen uten at cipheret (enkelt) kan brytes.
- (n, d) er hemmelig og beholdes av mottaker for å kunne dekryptere meldingene.

3.5.2 Kryptering av en melding bestående av heltall

For å se på hva som skjer når sender skal kryptere en melding, la oss først anta at han skal sende et positivt heltall t, slik at $0 \leq t < n$. Dette krypteres med $c = t^e \mod n$. Denne beregningen gjøres effektivt med Python-kommandoen `pow(t, e, n)`.

Mottaker dekrypterer meldingen ved å beregne $t' \leftarrow c^d \mod n$. Det kan nå vises at dekodingen er unik og riktig, dvs. at $t' = t$, men vi forbigår detaljene her i stillhet.

3.5.3 Kryptering av tekstmeldinger

Neste utfordring er å "oversette" en tekstmelding til heltall t, slik at vi kan bruke krypteringsmetoden for heltall beskrevet over. Dette gjør vi ved å ta teksten, oversette den til en byte-string, for så å re-kode byte-stringen til en (unsigned) int. Anta at vi har tekst-strengen "KODE". ASCII-verdiene for bokstavene er 75 ("K"), 79 ("O"), 68 ("D") og 69 ("E"). Binær-verdiene for disse fire tallene er (med bitstreng-lengde 8, ettersom ASCII-alfabetet er 256 tegn langt, og $\log_2(256) = 8$) er henholdsvis "`01001011`", "`01001111`", "`01000100`" og "`01000101`". Bit-steng-representasjonen for "KODE" finner vi ved å sette disse strengene sammen, dvs. "`01001011010011110100010001000101`". Dette er binær-representasjonen for tallet 1263486021, som dermed er tallet vi bruker for "KODE". Dette tallet kan vi sende gjennom krypteringen som angitt over.

Hvis meldingen er veldig lang kan dette heltallet bli for stort (husk at $t < n = p \cdot q$ kreves for at enkodingen av heltallet t skal være gyldig). I stedet for å ta hele strengen som et enkelt tall er det derfor greit å ta blokker av strengen med fast lengde l, og la hver blokk definere heltall. Hvis vi vil spesifisere "KODE" med blokk-lengde $l = 2$ vil det gi oss blokkene "KO" og "DE". Generelt får vi en liste av (del-)meldinger som hver for seg kan kodes om til heltall som vi deretter kan kjøre gjennom krypteringen. Typisk vil man velge $l \leq b/4$, der b var antallet bits brukt for å generere p og q. Et vanlig valg er $b = 1024$ og $l = 256$, men for raskere kjøring av koden kan for eksempel $b = 8, l = 1$ brukes.

Funksjonen `blocks_from_text` i `cypto-util.py` gjør mye av denne jobben for deg: Den tar en tekst inn, og returnerer listen med heltall klar for kryptering. På

samme vis vil mottaker ha behov for å sette sammen de (dekrypterte) heltallene til en tekstmelding igjen. Her kan du bruke `text_from_blocks`, som også ligger i `cypto-util.py`.

Implementasjon – Del 6:
Implementer cipheret i klassen RSA, som en sub-klasse av Cipher.

4 Hacking av cipher

Alle cipherne vi har sett på bortsaett fra RSA kan enkelt hackes. Hvis vi, for eksempel, vet at senderen og mottakeren bruker Caesar-cipheret kan vi bare prøve oss fram med nøkkel $n = 0, 1, 2, \ldots$ og sjekke hva som gir best mening. Hver kandidat-nøkkel genererer en foreslått klartekst som en hacker kan validere, for eksempel ved å se på hvert ord i kandidat-teksten og se om det er et "godkjent" ord. Dette er brute-force-hacking, og virker overraskende godt for de cipherne vi har gått gjennom her. Implementer en hacker-klasse som kan brute-force cipher-tekst. Den skal evaluere de forskjellige alternative nøklene ved å se på om ordene som blir generert som klartekst er engelske ord (liste over engelske ord er i fila `english_words.txt` som ligger på its learning). Alle cipherne som gjennomgås her unntatt RSA skal kunne hackes av hackeren din. For enkelhets skyld rent implementasjonsmessig er det smart å utvide implementasjonen av hvert cipher slik at det selv kan fortelle hva som er mulige nøkler. For Caesar er for eksempel dette definert som range(0, ⟨lengden til alfabetet⟩).

Som kandidatnøkler for den ubrytelige cipheren kan du benytte alle ordene i fila `english_words.txt`.

Implementasjon – Del 7:
Implementer klassen Hacker, som enten kan være en subklasse av Person eller av Receiver; velg det du synes er mest naturlig for din kode. Klassen skal ha en metode som tar en enkodingen av en (engelsk) tekst og et cipher som input, og returnerer den best mulige dekodingen av teksten etter å ha testet kandidat-nøkler helt til den blir fornøyd. Sjekk at dette virker for alle relevante cipher.

5 Hva kreves for å bestå oppgaven

For å bestå denne oppgaven må du:

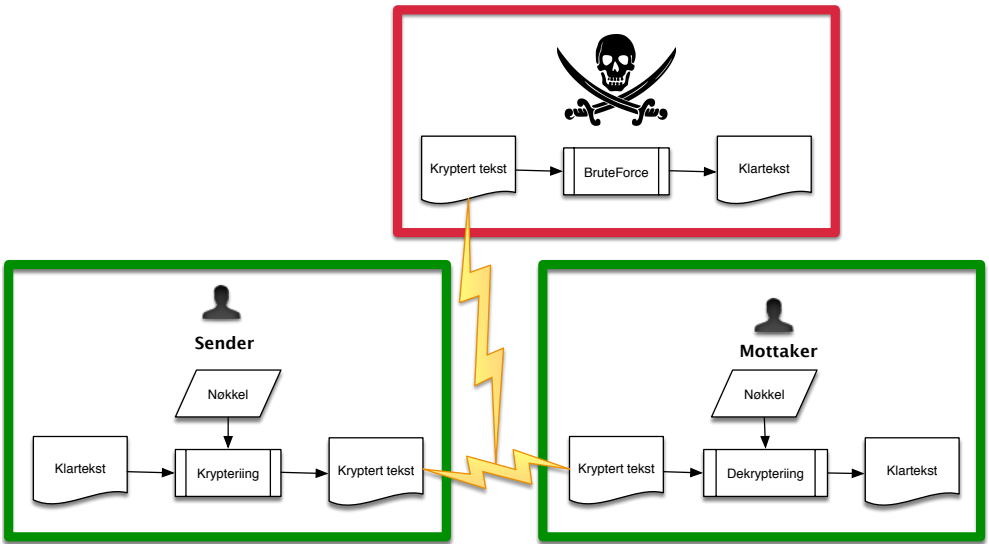

Figure 2: Hackeren får fatt på ciphertekst, og kan brute-force meldingen ved å forsøke alle keys.

- Løse alle de obligatoriske del-oppgavene
- Du skal gjøre arbeidet alene, og få det godkjent innen fristen.
- Systemet skal implementeres med objekt-orientert Python.