# INF122: Oblig 0

2023

Universitetet i Bergen

## Substitusjonschiffer

I kapittel 5 av Huttons "Programming in Haskell" finnes det et eksempel på hvordan en enkel form for kryptering, kalt Cæsarchiffer, kan knekkes. I denne obligen skal vi bruke samme teknikk, men på en litt større klasse av chiffer som heter substitusjonschiffer. Derfor anbefales det at du går tilbake til eksempelet i læreboken og leser det før du begynner på obligen.

En enkel måte å endre en tekst på, slik at den blir uleselig, er ved å bytte ut hvert tegn med et annet tegn. I Cæsarchiffer byttes tegnene ved at man flytter hver bokstav et visst antall plasser frem i alfabetet, men man kan tenke seg at man kan stokke om tegnene på andre måter, og man kan inkludere spesialtegn, slik som mellomrom, punktum og komma i tabellen med tegn for å gjøre enda større endringer. Denne teknikken, hvor man velger en arbitrær omstokking av alle bokstaver og tegn, kalles *substitusjonschiffer*, fordi hvert tegn "substitueres" av et annet. Valget av omstokking er den *hemmelige nøkkelen* som kan brukes til å kryptere og dekryptere. Fore eksempel kan følgende tabell være en hemmelig nøkkel:

| Klartekst | A | B | C | D | E | F | G | H | I | J | K | L | M | . | , |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Chiffer | Q | W | E | R | T | Y | U | I | O | P | A | _ | D | S | B |

| Klartekst | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | _ | ! |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Chiffer | F | G | H | J | K | L | Z | X | C | V | ! | N | M | . | , |

Dersom vi bruker nøkkelen ovenfor til å kryptere strengen "HASKELL RULES!" får vi "IQLAT .KX TL,". For å dekryptere må vi bruke tabellen baklengs.

Hvis teksten bruker $n$ ulike tegn, så finnes det $n!$ mulige valg av omstokkninger. For eksempel, det engelske alphabetet, pluss mellomrom, punktum, komma, bindestrek, utropstegn og spørsmåltegn, blir 57 tegn, hvis vi regner store og små bokstaver for seg. Da finnes det $57! = 405269195048772167556806019054323221349803847962266021451844812800000000000000$ ulike hemmelige nøkler, altså omstokkinger, vi kunne valgt. I motsetning til Cæsarchiffer så er det ikke mulig å teste alle mulige nøkler, hvis man vil knekke koden. Det er rett og slett for mange av dem. Allikevel er dette en form for kryptering som ansés veldig enkel å knekke, og helt ubrukelig i moderne kryptografi. Vi skal se nærmere på hvorfor i denne obligen. Først skal vi implementere

selve krypteringen, og så gjøre noen forsøk på å knekke den.

Fra et programmeringsperspektiv er poenget med denne obligen at dere skal jobbe med funkjonell programmering på lister. Vi skal bruke lister og tupler til å modellere konseptene fra kryptograpfien ovenfor, slik at typene til funksjonene vi skriver blir forståelige og i stor grad forklarer hva funksjonene gjør. Her er typene vi skal jobbe med:

```
import qualified Data.Set as Set

type Key = [(Char,Char)]
type FrequencyTable = [(Char,Double)]
type Alphabet = String
type Dictionary = Set.Set String
```

**`Key`** Typen `Key` representerer lister av par der hvert par inneholder to tegn. Det første tegnet er det som skal byttes ut, og det andre tegnet er det det skal byttes med. Listen representerer en bijektiv funksjon fra tegnsettet til seg selv for kryptering og dekryptering. For eksempel, nøkkelen fra tabellen ovenfor ville representeres av listen nedenfor.

```
key :: Key
key = [ ('A', 'Q'), ('B', 'W'), ('C', 'E'), ('D', 'R'), ('E', 'T')
      , ('F', 'Y'), ('G', 'U'), ('H', 'I'), ('I', 'O'), ('J', 'P')
      , ('K', 'A'), ('L', ' '), ('M', 'D'), ('.', 'S'), (',', 'B')
      , ('N', 'F'), ('O', 'G'), ('P', 'H'), ('Q', 'J'), ('R', 'K')
      , ('S', 'L'), ('T', 'Z'), ('U', 'X'), ('V', 'C'), ('W', 'V')
      , ('X', '!'), ('Y', 'N'), ('Z', 'M'), (' ', '.'), ('!', ',')]
```

**`FrequencyTable`** Representerer lister av par som inneholder et tegn og en flyttallsverdi. Tegnet representerer et symbol fra tekstkorpuset, mens den tilhørende flyttallsverdien representerer den relative frekvensen av det symbolet i korpuset.

**`Alphabet`** En streng som inneholder alle de unike tegnene som kan forekomme i teksten. Brukes i Cæsarchiffer for å definere rekkefølgen når symbolene byttes ut.

**`Dictionary`** En samling unike strenger i et `Set`. Disse strengene er ord fra korpuset. I denne konteksten brukes det for å validere om en dekryptert streng inneholder forståelige ord.

## Begrepsliste

Disse begrepene vil bli introdusert i løpet av teksten, men de er også samlet her, slik at du kan refere til denne listen når du leser oppgavene.

**Hemmelig nøkkel** En bit informasjon som fungerer som input til en krypteringsalgoritme. Nøkkelen holdes hemmelig og er kun kjent for avsender og mottaker. I disse øvelsene er nøkkelen en liste med substitusjoner.

**Klartekst** Den opprinnelige, ukrypterte meldingen som skal sendes fra avsender til mottaker.

**Chiffertekst** Den krypterte versjonen av klarteksten, produsert ved å anvende en krypteringsalgoritme på den.

**Substitusjonchiffer** En enkel form for kryptering hvor hvert symbol i klarteksten erstattes med et annet bestemt symbol.

**Korpus** En stor samling av tekst som brukes for statistiske analyser. I dette tilfellet brukes det for å lage en frekvenstabell og ordliste. Du finner korpuset du skal bruke i filen "corpus.txt".

**Frekvenstabell** En tabell som viser hvor ofte hvert symbol forekommer i et gitt korpus.

**$\chi^2$-test (Chi-kvadrat test)** En statistisk test som måler hvor godt observerte data passer med en forventet distribusjon. I denne oppgaven bruker vi $\chi^2$ for å sammenligne frekvensen av symboler.

**Grådig algoritme** En algoritme som tar lokale optimaliseringsbeslutninger i hvert steg i håp om å finne en global optimal løsning.

**Utgjevningsfaktor** En liten verdi som legges frekvenser i frekvenstabellen for å unngå problemer med nullverdier i statistiske beregninger.

## Oppgaver

Oppgavene skal leveres individuelt, og du skal kunne gjøre rede for løsningen du har levert. Dersom du sammarbeider med noen andre under løsningen av oppgavene, spesifiser det i en kommentar øverst i filen du leverer inn.

Filen du leverer på CodeGrade skal hete `Oblig0.hs` og inneholde modulen `Oblig0`. Du finner en mal med funksjonssignaturer på MittUiB.

### Oppgave 1: Substitusjonschiffer

I denne oppgaven skal du implementere kryptering og dekryptering for substitusjonschiffer. Som nevnt skal vi bruke typen `Key`, som er et assosiasjonsliste for `Char` til `Char` substitusjoner. Elementene i listen forteller hvilke tegn som skal erstattes. For eksempel, dersom listen inneholder paret `('a','x')` skal alle forekomster av `'a'` i input erstates med `'x'`.

*OBS: Dersom et tegn ikke finnes i nøkkelen skal det tegnet beholdes.*

For både klartekst og chiffertekst skal vi bruke typen `String`. Husk at `String = [Char]`.

a) Skriv en funksjon `encode :: Key -> String -> String` som gitt en hemmelig nøkkel krypterer en klartekst. *Hint:* Hvis du bruker `lookup` kan du bruke en `case`-split eller `fromMaybe` for å beholde tegn som ikke finnes i nøkkelen.
b) Skriv en funksjon `decode :: Key -> String -> String` som gitt en hemmelig nøkkel dekrypterer en chiffertekst tilbake til klartekst. *Hint:* Selve koden til `decode` vil ligne veldig på koden til `encode`. Hvis du definerer en hjelpefunksjon `invert :: Key -> Key` som snur på nøkkelen, kan du definere `decode` ved hjelp av `encode`.
c) Skriv en funksjon `caesar :: Alphabet -> Integer -> Key` som lager en omstokkingsnøkkel som roterer alfabetet et gitt antall steg. Du kan bruke denne til å teste `encode` og `decode`.

Som en enkel test av funksjonene dine kan du laste inn programmet i GHCi og kjøre følgende:

```
> caesar "abcde" 2
[('a','c'),('b','d'),('c','e'),('d','a'),('e','b')]
```

```
> alphabet = ['a'..'z'] ++ ['A'..'Z'] ++ " .-?!"
> key = caesar ['a'..'z'] 5 ++ caesar ['A'..'Z'] 5 ++ caesar " .,-?!" 5
> plaintext0 = "Hello, world!"
> ciphertext0 = encode key plaintext0
> putStrLn ciphertext0
Mjqqt.!btwqi?
> decode key ciphertext0 == plaintext0
True
> plaintext1 = "Symbols: # $ %"
> ciphertext1 = encode key plaintext1
> putStrLn ciphertext1
Xdrgtqx:!#!$!%
> decode key ciphertext1 == plaintext1
True
```

### Oppgave 2: Frekvenstabeller

Grunnen til at substitusjonchiffer er så sårbare er at hver forekomst av et tegn i klarteksten krypteres til det samme tegnet. I språket vårt forekommer ulike tegn med ulik frekvens. For eksempel vil mellomrom dukke opp veldig ofte, ettersom det brukes mellom hvert ord i hver setning. Andre tegn, slik som bokstaven "x" bruker ikke like ofte. Man kan derfor se på chifferteksten hvilke tegn som forkommer ofte og begynne å gjette på hvilke tegn de motsvarer i den klarteksten.

Det første vi behøver er en funksjon som regner ut frekvensen av hvert ord i en tekst. Frekvensen er definert som antallet forekomster i en tekst, delt på lengden av teksten. Vi bruker dobbelpresisjonsflyttall for å representere frekvensen, og typen`FrequencyTable` som assosierer hvert tegn med sin frekvens.

En slik frekvenstabell er en enkel *modell av språket*. Vi skal bruke modellen til å forsøke å gjenkjenne når vi nærmer oss riktig nøkkel ved å sammenligne tegnfrekvensen med modellen. Denne sammenligningen heter $\chi^2$ ("chi kvadrat" på norsk eller "chi squared" på engelsk).

a) Skriv en funksjon `count :: String -> FrequencyTable` som regner ut frekvensen for hvert tegn i en streng. *Hint:* Forsøk å unngå å regne ut lengden på listen flere ganger. Den naive algoritmen for å telle antall forekomster vil ha kvadratisk tidskompleksitet, da den iterer igjennom hele resten av listen etter forekomster. Det finnes enkle løsninger som er mer effektive, for eksempel ved sortering, eller å gå via `Data.Map`. Disse vil istedet ha `n·log n` tidskompleksitet.

For å enkelt kunne lage en frekvenstabell fra et korpus behøver vi en funksjon som leser inn en fil og bruker `count` på inneholdet i filen.

b) Skriv en funksjon `loadFrequencyTable :: FilePath -> IO FrequencyTable` som leser inn en fil og lager en frekvenstabell for inneholdet. *Hint:* Funksjonen `readFile :: FilePath -> IO String` leser inn en hel fil.

Neste steg for å knekke koden er å gjøre en enkel gjetning på hva den hemmelige nøkkelen kan være, basert på modellen og den observerte frekvensen. Denne gjetningen er ikke nødvendigvis riktig, men vil tjene som startpunkt for videre forbedring.

c) Skriv en funksjon `initialGuess :: FrequencyTable -> FrequencyTable-> Key` som matcher to frekvenstabeller ved å assosiere tegnet med høyest frekvens i den ene tabellen med den som har høyest frekvens i den andre, og så videre nedover. Dersom én av tabellene er kortere enn den andre stopper assosieringen når den korteste tabellen er slutt. *Hint:* Bruk funksjonen `sortBy` til å sortere tabellene på riktig måte. Etter sorteringen kan du enten projisere bort frekvensene og bruke `zip`, eller kombinere med `zipWith`.

Nå skal vi regne ut $\chi^2$ statistikken. Den måler hvor nært en observert frekvenstabell ligger til modellen.

Den matematiske formelen for $\chi^2$ er $\chi^2 = \sum_{i \in S} \frac{(O_i - E_i)^2}{E_i}$, hvor $S$ er mengden over alle mulige utfall (i vårt tilfelle er $S$ alle tegn som forekommer i modellen eller strengen). I denne formelen står $O_i$ for den observerte frekvensen og $E_i$ den forventende frekvensen i følge modellen.

En ekstra komplikasjon er at det kan hende vi observerer et tegn som ikke forekommer i modellen. Da vil formelen ovenfor inneholde divisjon med null. En standard teknikk for å håndtere dette er å tildele en lav, men ikke-null sannsynlighet til slike tegn. Dette kalles en utjevningsfaktor (Engelsk: smoothing factor). Du kan bruke utjevningsfaktoren `1/10000`.

d) Skriv funksjonen `chiSquared :: FrequencyTable -> FrequencyTable -> Double` som regner ut $\chi^2$ statistikken. Første argument er modellen, altså den forventende frekvenstabellen. Andre argument er den observerte frekvensen. *Hint:* Læreboken har en eksempel implementasjon av $\chi^2$ for et litt enklere tilfelle. Du kan bruke den som et utgangspunkt, men du må ta høyde for at vi her jobber med assosiasjonslister. En fremgangsmåte vil være å lage en liste over alle unike tegn, og så slå hvert tegn opp i begge frekvenstabellene (og bruke utjevningsfaktoren hvis det trengs) for å regne ut hver summand.

Som en enkel test av funksjonene dine kan du laste inn programmet i GHCi og kjøre følgende:

```
> count "Hello"
[('H',0.2),('e',0.2),('l',0.4),('o',0.2)]
> import Data.Tuple (swap)
> import Data.List (sortBy)
> model <- loadFrequencyTable "corpus.txt"
> -- Find the number of unique symbols in the corpus
> length model
87
> -- List the symbols of the corpus (order might vary)
> map fst model
"\n\r !\"#$%&'()*+,-./0123456789:;<=>?
ABCDEFGHIJKLMNOPQRSTUVWXYZ]abcdefghijklmnopqrstuvwxyz"
> -- Define a natrual sorting for frequency tables
> oplex x y = compare (swap y) (swap x)
> -- List of symbols in the corpus, sorted by frequency
> map fst $ sortBy oplex $ model
" etaoinsrhldcumpfgywb.,vkIT-A'S1\"0Cx
)(2M9OPEBWNRHDLFjG3z;q5:8476U?J#YK/V&*!%=<>$\r\nQZX+]"
> plaintext2 = "Values are becoming more and more
                recognized as issues in psychotherapeutic discourse"
> -- Unique characters in the plaintext, sorted by frequency
```

```
> map fst $ sortBy oplex $ count plaintext2
"e sroicaunmdtphgzylbV"
> key = caesar ['a'..'z'] 5 ++ caesar ['A'..'Z'] 5 ++ caesar " .,-?!" 5
> ciphertext2 = encode key plaintext2
> map fst $ sortBy oplex $ count ciphertext2
"j!xwtnhfzsriyumlqgedA"
> guessedKey2 = initialGuess (count plaintext2) (count ciphertext2)
> decode guessedKey2 (encode key "home") == "home"
True
> -- Test that the plaintext is a better fit for the model than the ciphertext.
> chiSquared model (count plaintext2) < chiSquared model (count ciphertext2)
True -- In fact, chiSquared model (count plaintext2) should be a bit less than 1
     -- while chiSquared model (count ciphertext2) should be more than 100
```

### Oppgave 3: Grådig optimering

I denne oppgaven skal vi bruke en grådig algoritme til å forbedre gjetningen på hva den hemmelige nøkkelen var, gitt kun cipherteksten som input. Idéen er enkel: en korrekt dekryptert tekst vil ha frekvenser som ligner på frekvensene i naturlig språk. Gitt en frekvenstabell som modellerer naturlig språk kan vi bruke $\chi^2$ til å måle hvor nærme den dekrypterte teksten kommer til modellen: Lavere $\chi^2$ er en bedre fit til modellen. I hvert steg av algoritmen forsøker vi alle mulige bytter av nøkkelverdier og, på grådig vis, tar den som minimerer $\chi^2$. Slik fortsetter vi til ingen bytter gir en lavere $\chi^2$.

*Forbehold:* Algoritmen ovenfor er ikke særlig rask og gir ikke nødvendigvis nøkkelen med lavest $\chi^2$. Grådige algoritmer kan sette seg fast i et lokalt minimum. Det er også langt fra sikkert at den hemmelige nøkkelen er den som minimerer $\chi^2$. Vi kan derfor ikke forvente en perfekt dekryptering fra denne algoritmen. I neste oppgave skal vi gjøre et ytterligere steg for å forbedre gjetningen på hva den hemmelige nøkkelen kan være.

Vi implementer algoritmen i et par steg.

a) Skriv en funksjon, `swapEntries :: (Char,Char) -> (Char, Char) -> Key -> Key`, som bytter om to av substitsjonene i en nøkkel. *Hint:* Du kan bruke `map` med en hjelpefunksjon til å gå igennom alle nøkkelens verdier og oppdaterer hvis det matcher et av parene som skal byttes om.

Funksjonen `swapEntries` kan skrives slik at den fungerer på alle assosiasjonslister.

b) Endre typen til `swapEntries` slik at den er så generell som mulig.

I den grådige algoritmen behøver vi en liste med umiddelbare endringer som kan gjøres til nøkkelen, slik at vi kan finne ut hvilken som er best.

c) Lag en funksjon, `neighbourKeys :: Key -> [Key]` som gitt en nøkkel generer alle nøkler man kan få fra denne ved å bytte om to ulike elementer. *Hint:* Du kan bruke listekomprehensjon sammen med `swapEntries` for å løse denne oppgaven.

Nå skal vi sette sammen funksjonene vi har laget så lang for å implementere en grådig algoritme som forsøker å finne en nøkkel som minimerer $\chi^2$ av den dekrypterte teksten i forhold til en gitt modell.

d) Skriv en funksjon `greedy :: FrequencyTable -> String -> Key -> Key` som gitt en modell og en chiffertekst, og en initiell gjetning på nøkkel, på en grådig måte gjetter en nøkkel som dekrypterer chifferteksten slik at $\chi^2$ i forhold til modellen minimeres. *Hint:* Du kan bruke rekursjon for å implementere den grådige algoritmen. Du kan bruke map eller listekomprehensjon for å gå igjennom `neighbourKeys` og bruke `decode` og regne ut `chiSquared` for hver av de nye nøklene. Deretter kan du bruke `minimum` for å finne den nye nøkkelen som har minst $\chi^2$ og sammenligne med den gitte nøkkelen og dens $\chi^2$.

Som en enkel test av funksjonene dine kan du laste inn programmet i GHCi og kjøre følgende:

```
> swapEntries ('a','b') ('b','c') (caesar "abcde" 1)
[('a','c'),('b','b'),('c','d'),('d','e'),('e','a')]
> -- Test the generalisation of swapEntries by
> -- applying it to integers and functions:
> swapEntries (1,2) (2,1) [(1,2),(2,1)]
[(1,1),(2,2)]
> ($ 0) <$> lookup 1 (swapEntries (1,cos) (2,sin) (zip [1..3] [cos,sin,tan]))
Just 0.0
> -- Test the neighbourKeys function (order of elements may vary)
> neighbourKeys (caesar "abc" 1)
[[('a','c'),('b','b'),('c','a')]
,[('a','a'),('b','c'),('c','b')]
,[('a','b'),('b','a'),('c','c')]]
> model <- loadFrequencyTable "corpus.txt"
> plainText3 <- take 2000 <$> readFile "corpus.txt"
> key = caesar ['a'..'z'] 3 ++ caesar ['A'..'Z'] 3 ++ caesar " .-?!" 3
> cipherText3 = encode key plaintext3
> k0 = initialGuess model (count ciphertext3)
> kg = greedy model ciphertext3 k0
> -- Test that χ^2 is small:
> chiSquared model (count (decode kg ciphertext3)) < 0.1
True
```

### Oppgave 4: Ordbokoptimiering

Hvis man forsøker å dekryptere ved hjelp av nøkkelen som den grådige algoritmen i forrige oppgave vil man mest sannsynlig ikke få noe fornuftig ut. Det beste vi kan håpe på er at de vanligste symbolene, slik som mellomrom og bokstaven "e" er på plass. Sammenlign for eksempel begynnelsen av korpuset med den forsøkt dekrypterte strengen:

```
"It is safe to say that ours is the only dining room in West Los Angeles"
"ka is stpe ao stu ahta ogrs is ahe onlu dininw room in Hesa 0os Snweles"
```

Ikke akkurat en perfekt match, men noen viktige symboler er på plass. Kankje noen få endringer vil gi en perfekt match? Neste idé er å gjøre nok en grådig algoritme, men denne gangen forsøker vi å gjøre slik at teksten består mest mulig av ord fra språket vi dekrypterer til. I vårt tilfelle blir det engelsk.

Vi begynner med å lage en ordbok selv, fra et korpus lokalisert i en fil. Typen `Dictionary` representerer ordbøker som et `Data.Set` av strenger.

a) Skriv en funksjon, `loadDictionary :: FilePath -> IO Dictionary`, som leser et korpus og finner alle unike ord som forekommer. *Hint:* Når filen er lest inn kan du bruke funksjonene `words` og `Set.fromList` for å lage ordboken.

Det neste steget er å bruke en ordbok til å telle hvor mange gyldige ord som forekommer i en streng.

b) Skriv en funksjon, `countValidWords :: Dictionary -> String -> Integer` som teller hvor mange av ordene i en streng som forekommer i ordboken. *Hint:* Du kan bruke `filter` sammen med `Set.member` til å filtrere de ordene som forekommer i ordboken.

Til slutt skal vi implementere den nye grådige algoritmen. Den skal forsøke å maksimere antall gyldige ord i den dekrypterte teksten. Eller er oppsettet likt som i forrige oppgave. Vi begynner med en initiell gjetning på nøkkelen, og en ordbok. Så forsøker vi alle enkle bytter av verdier i nøkkelen og ser om de forbedrer antallet dekrypterte ord som ligger i ordboka. Slik forsetter vi til ingen bytter gir flere ordbok-ord.

c) Skriv en funskjon, `greedyDict :: Dictionary -> String -> Key -> Key`, som implementer den grådige algoritmen for å finne den nøkkelen som maksimerer antallet ordbokord i den dekrypterte teksten. *Hint:* Du kan bruke samme oppsett som i `greedy`, med rekursjon og bruk av funksjonene `neighbourKeys` men istedet for `chiSquared` og `minimum`, må du bruke `countValidWords` og `maximum`.

Som en enkel test av funksjonene dine kan du laste inn programmet i GHCi og kjøre følgende:

```
> dictionary <- loadDictionary "corpus.txt"
> Set.member "the" dictionary
True
> countValidWords dictionary "hello world"
2
> countValidWords dictionary "not quite carrect"
2
> plainText4 <- take 2000 <$> readFile "corpus.txt"
> model <- loadFrequencyTable "corpus.txt"
> key = caesar ['a'..'z'] 3 ++ caesar ['A'..'Z'] 3 ++ caesar " .-?!" 3
> cipherText4 = encode key plainText4
> k0 = initialGuess model (count cipherText4)
> kg = greedy model cipherText4 k0
> kd = greedyDict dictionary cipherText4 kg
> -- A simple function to test the percentage two strings match
> match a b p = 100*(length $ filter id $ zipWith (==) a b) > p*(length a)
> -- Check that decrypted cipher text matches plain text at least 95%
> match (decode kd cipherText4) plainText4 95
True
```