

IS105 - Mappe

Gruppe 10 - Dieselgutta

Gruppemedlemmer:

Stian Blankenberg, Simen Fredriksen, Kristian Hagberg, Jone Manneråk, Rasmus Sørby, og Tarjei Taxerås.

Introduksjon	4
ICA01	5
1.2 Oppgaver	5
1.2.1 Binære tall	5
1.2.2 Informasjonsmengde	6
1.2.3 Arbeid med git	6
1.2.4 Samarbeid i Git og Introduksjon i Golang	7
ICA02	10
Oppgave 1	10
Oppgave 2	11
Oppgave 3	14
Oppgave 4	15
Oppgave 5	16
ICA03	21
Oppgave 1	21
Oppgave 2	23
Oppgave 3	24
Oppgave 4	27
ICA04	29
Oppgave 1	29
Oppgave 2	30
Oppgave 3	31
Oppgave 4	32
ICA05	34
Tankegang for kode:	34
Golang:	34
HTML:	34
ICA 06	36
Eksperiment 1	36
Eksperiment 2	37
Eksperiment 3	38
ICA07	40
Oppgave 1:	40
Oppgave 2:	42
Oppgave 3:	42

Kildereferanser**43**

ICA01

43

ICA02

43

ICA03

43

ICA04

43

ICA05

43

ICA06

43

ICA07

43

Introduksjon

Gjennom alle ICA-ene har gruppen samarbeidet tett. I enkelte oppgaver vil det favorisere enkelte personer gjennom “commits”, men vi ønsker å gjøre det klart at vi ofte har sittet to og to, der en har kommentert mens den andre har skrevet kode. Dette har ført til litt “urettferdig” fordeling av commits, noe som bør tas til betraktning. For å rense opp i repositoriene våre opprettet vi nye for hver ICA før vi lastet opp alle relevante filer på nytt, dette førte igjen til at fordelingen av commits ble redusert. I flere oppgaver har vi brukt andre repositories på Github som utgangspunkt og inspirasjon for å komme frem til svar på de forskjellige eksperimentene og oppgavene. En liste av disse referansene kan finnes i bunnen av dette dokumentet.

ICA01

Deltakere: Simen Fredriksen, Stian Blankenberg, Jone Manneråk, Kristian Hagberg, Tarjei Taxerås og Rasmus Sørby.

Nicolay Bråthen Leknes og Tommy Nilsson var med ved første gjennomgang.

Repository: <https://github.com/Dieselgutta/ICA01>

1.2 Oppgaver

1.2.1 Binære tall

Konverter følgende desimaltall til 2-tallssystemet (binært tallsystem):

(1) $1 = 1 - 1\text{bit}$

(2) $2 = 10 - 2\text{bit}$

(3) $5 = 101 - 3\text{bit}$

(4) $8 = 1000 - 4\text{bit}$

(5) $16 = 10000 - 5\text{bit}$

(6) $256 = 100000000 - 8\text{bit}$

For å konvertere tallene om til 2-tallssystemet brukte vi metoden hvor vi deler på 2 og setter opp 1 som rest der det ikke går. Eks:

$$5 : 2 = 1$$

$$2 : 2 = 0$$

$$1 : 2 = 1$$

Når man har regnet om til binære tall slik vi har vist så leser man svaret nedenfra og opp. Med dette mener vi at man ser på det nederste tallet i "tabellen" først, slik at du leser $1 : 2 = 1$ først, så det første binære tallet er 1.

$$5 = 101$$

Konverter følgende binære tall til desimaltall (mest signifikante bit-en er til venstre):

(1) $100 = 4$

(2) $1001 = 9$

(3) $1100110011 = 819$

- Vi gjorde det slik:

$$100 = 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 0 + 0 + 4 = 4$$

Vi har 0 på enerplassen, 0 på toerplassen og 1 på firerplassen.

1.2.2 Informasjonsmengde

Det binære tallet kan være mellom 000 og 111, altså et tall mellom 0 og 7.

Lise får vite at tallet er et oddetall må det da enten være 1, 3, 5 eller 7. Det er altså 4/8 muligheter her. Ved å bruke formelen $\log_2(1/(M/N))$ hvor M er antall oddetall, og N er totalt antall muligheter får vi.

$$1) \log_2(1/(4/8)) = 1 \text{ bit}$$

Man kan her tenke seg at tallet har to muligheter, enten er det et oddetall, eller så er det ikke det. (1 bit)

Videre får Per vite at tallet ikke er et multiplum av 3. (altså 0, 3 eller 6). Da er det altså $\frac{5}{8}$ muligheter her.

$$2) \log_2(1/(5/8)) = 0.6780719051126377 \text{ bits}$$

Her kan vi også tenke praktisk tenke oss at tallet har to muligheter, enten er det et multiplum av 3, eller så er det ikke det. (1bit)

Videre får Oskar vite at to av de binære tallene er 1, mulighetene blir da enten 011, 101 eller 110 (3, 5 og 6)

$$3) \log_2(1/(3/8)) = 1 \text{ bit}$$

Louise får vite alt de andre vet, hun vet altså at det er ett oddetall, at det ikke er et multiplum av 3, og at det enten er 011, 101 eller 110 (3, 5 eller 6).

Av disse alternativene er det bare 5 som stemmer overens med alle kriteriene. Det blir

$$4) \log_2(1/(1/8)) = 3 \text{ bits}$$

Louise har da 3 bits med informasjon, og kan dermed finne fram til at det tresifrede binære tallet er 101 (5).

1.2.3 Arbeid med git

Kommandoer:

```
$ git clone https://github.com/Dieselgutta/ICA01.git
```

```
$ cd ICA01 $ git add hello.go.txt
```

```
$ git status $ git commit -m "endringer"
```

```
$ git push origin master
```

1.2.4 Samarbeid i Git og Introduksjon i Golang

1) Ved å ha flere branches i et hovedrepository sikrer man seg selv for å endre i master. De som jobber sammen kan se endringene som andre man samarbeider med har gjort i prosjektet. Man sikkerhetskopierer filene, og man lar flere personer jobbe med flere ulike type kode. Eksempelvis kan noen jobbe med flere features, mens andre jobber med strukturering, opprydding og refactoring. Et eksempel på dette er dersom en ny programvare skal bli lansert så vil ikke informasjonen du har lagret gå tapt. Dette betyr også at debugging og maintenance blir enklere, siden man ikke gjør endringer i master-branchen. Man er da beskyttet mot å ødelegge prosjektet.

På en annen side er git-flow en vrien måte å bruke om man jobber i et agilt prosjekt, der man enten med laste opp eller ned flere ganger om dagen. Man må også legge til endringer, commite og pushe når man skal endre i prosjektet, dette kan være komplisert fordi man må gjøre det i riktig rekkefølge. På den andre siden kan dette også være en enkel og nyttig funksjon dersom man bruker det riktig, og ofte nok. Å dele opp prosjekter i forskjellige grener eller “branches” forsikrer brukerne mot å endre på feil ting.

Mange problemer med å bruk av github kan komme av dårlig kommunikasjon i gruppen. Hvis flere skal endre på en kode, blir det dumt om alle endrer på den samme tingen. Det er derfor viktig å kommunisere på en god måte hva, når og hvor du gjør endringer. Dette er et problem man ser i alle former for gruppearbeid, og det er ikke spesielt for git flow modeller, men det er et problem man ofte ser.

2) Windows: Portable Executables, for eksempel (.exe, .dll .acm), Mac: Mach-O, Linux: COFF (Common Object File Format), for eksempel .o eller .obj. På Linux er det ikke lenger COFF som er vanlig. Dette er fordi alle filer på Linux kan kjøres. Dette kommer fra filosofien “everything on Linux is a file”. Dette betyr at endingen, eller filformatet, kun beskriver hva filen gjør/hvordan den blir kjørt.

De har forskjellige filer for å optimalisere systemet. Filene lagres og hentes ut på forskjellige måter i de forskjellige systemene, og det er derfor naturlig at de opererer annerledes.

Når man prøver å kjøre en fil i Windows vil operativsystemet kun se på filnavnet for å bestemme hvordan den skal kjøre den. Dersom det ikke finnes noe filnavn, eller filnavnet er feil, vil ikke filen bli kjørt riktig. Dette skjer ikke i Mac OS X eller Unix/Linux. Her er informasjon om filen lagret i begynnelsen av selve filen, istedenfor på slutten i form av et filnavn. På grunn av dette kan en fil bli kjørt i Mac eller Linux uansett om den har et filnavn eller ikke. Dette kalles en MIME-type, og blir ikke brukt av Windows-systemer.

3) Konstruksjonen er ulik java. I java bruker man semikolon “;” i slutten av en linje, noe man ikke gjør i Golang. Det er også forskjeller som `fmt`. istedenfor `“System.out.”` for print-funksjoner osv. hvor `fmt` må importeres før bruk. Man har heller ikke felt og constructorer i golang. I Golang kommer også variabel-typen etter variabel-navnet, her trenger man også bare å skrive variabel-typen bak den siste variabelen i en rekke (så lenge det er samme variabel). Arbeid med variabler i Golang er mye simplere, de kan alle konstrueres på rekke og rad uten å erklære variabelen i begynnelsen. Variabler får også en nullverdi automatisk hvis man ikke definerer den til annet (0, false, “ ”). Man kan også returnere flere resultater med en return, i motsetning til java hvor man bare kan returnere ett resultat. Man kan også navngi “return”-verdier for å gjøre dokumentasjonen enklere.

4) Når man bruker et programmeringsmiljø kommuniserer det direkte med plattformen man er på. Den vil også illustrere at man kan hente ut informasjon fra filer som ligger i akkurat samme mappe. Dette gjør det mer strukturert og er da lettere å hente ut informasjon, uten å måtte lete gjennom hele koden.

5) Ettersom github gjør deling av arbeid enklere kan det være hensiktsmessig å legge denne filen til et repository. Det blir lettere for andre å gjøre endringer til filen, eller å bruke filen i kombinasjon med andre program. Siden `logcli` kan brukes som et utgangspunkt i `logbcli` kan det også være greit å ha den opplastet til github.

6) Pakken `log` inneholder for det meste funksjoner mens pakken `fmt` inneholder en rekke forkortelser og definisjoner. Pakken `log` må vi selv implementere og må vite hvor vi oppretter filene og hvor vil velger å legge dem på datamaskinen. Den inneholder også (type `Logger`). `fmt` inneholder print-funksjoner, og ikke noe særlig mer, den printer kun det som er definert.

ICA02

Deltakere: Simen Fredriksen, Stian Blankenberg, Jone Manneråk, Kristian Hagberg, Tarjei Taxerås og Rasmus Sørby.

Nicolay Bråthen Leknes og Tommy Nilsson var med ved første gjennomgang.

Repository: <https://github.com/Dieselgutta/ICA02>

Oppgave 1

Oppg/Navn	Simen	Stian	Jone	Tarjei	Kristian	Rasmus
Antall prosesser på datamaskin	209	212	284	78	98	146
Antall prosesser i virtuell server	137	134	137	138	130	139
Prosesser som “kjører”	103	207	205	75	81	80
Prosessortype	Intel ® Core (TM) i7-6500	Intel Core i5	Intel(R) Core(TM) i5-3210M	AMD A8	Intel(R) Pentium(R) B950	Intel core i7-4500
Arkitektur	x84_64	x84_64	x84_64	x84_64	x84_64	x84_64
Klokkefrekvens	2,60GHz	2,5GHz	2.50GHz	2,2GHz	2.10GHz	1.80GHz
Primært minne	8GB	4GB	10GB	8GB	4GB	8GB
Størrelse på cache	L1=128kb L2=512 L3=4mb	L2=1mb L3=3mb	L2=1mb L3=3mb	L1=256kb L2=2mb	L3=2mb	L1=128kb L2=512kb L3=4mb
Antall CPU-cores tilgjengelig på datamaskin	2	4	4	4	2	4

Antall CPU-cores tilgjengelig på server	4	4	4	4	4	4
Hvilken bruker mest minne?	Google Chrome	Safari	Safari	Google Chrome	Google Chrome	Google Chrome

Det er ikke mulig å finne nøyaktig antall siden det varierer hele tiden. Det meste som skjer på en pc krever en egen prosess, selv tastetrykk.

Prosessene som ikke kjører kommer opp som suspendert, dette betyr at de er lastet inn i hovedminnet, men ikke bruker noe prosessorkraft. De kan derfra enkelt endre status til kjørende. Er brukt for prosesser som "venter" på å bli startet.

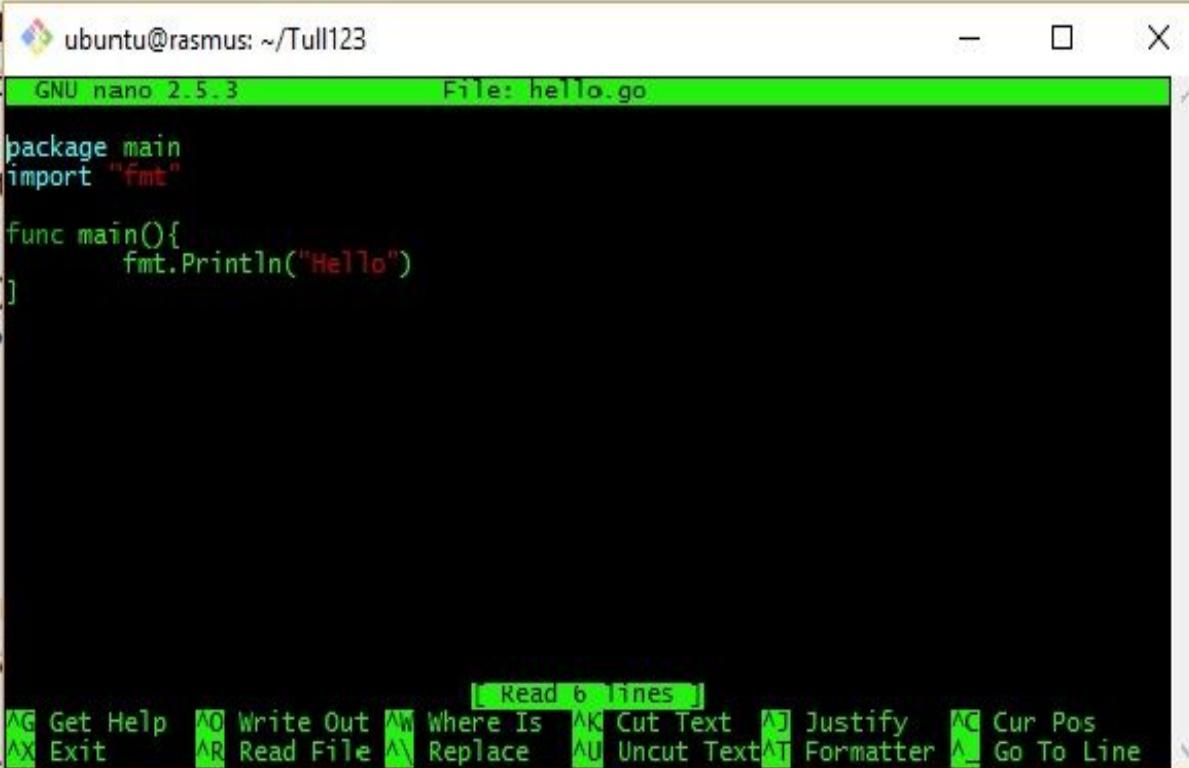
Hos oss var det nettlesere som stod for hovedbruket av minne. Google Chrome er her godt kjent for å kreve store mengder minne, en av grunnene til dette er at den splitter det meste nettleseren består av i mindre deler. Dette forhindrer at resten av nettleseren crasher hvis en plug-in gjør det, men fører igjen til duplikater av prosesser. Det er mye informasjon som blir midlertidig lagret i minnet når man har en nettleser oppe, spesielt ved bruk av flere faner, her skjer det igjen duplisering av prosesser.

For at en prosess skal kjøre kreves det en prosessor, hovedminne, input-enhet og output-enhet. Prosessoren utfører kalkulasjonene som kreves for at et program skal kjøre. Det prosessoren kalkulerer blir underveis lagret i hovedminnet. Prosessoren kan igjen hente ting som er lagret i hovedminnet for å utføre videre kalkulasjoner.

For å utføre prosesser kreves det en input-enhet som gir instruksjoner til prosessoren, dette er typisk fysiske ting som tastatur og datamus. Output-enheten er slik man mottar informasjonen som prosessoren produserer, dette kan være andre fysiske ting som en skjerm, en printer, eller høyttalere.

Oppgave 2

For å skrive koden fant vi først ut hvor i ubuntu vi ville lage koden. Etter å ha valgt en mappe, skrev vi inn “nano hello.go” for å lage et hello.go-program.



```
ubuntu@rasmus: ~/Tull123
GNU nano 2.5.3 File: hello.go

package main
import "fmt"

func main(){
    fmt.Println("Hello")
}

[ Read 6 lines ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^N Replace   ^U Uncut Text ^T Formatter ^_ Go To Line
```

Dette er et bilde av koden.

I dette eksempelet ville vi lage kode for Windows. Dette ble gjort ved å spesifisere OS=windows (GOOS=windows) og GOARCH=amd64. Dette er spesifikasjonene for en av PC'ene og kan variere for andre plattformer. Dette blir altså til en .exe-fil for windows.

```
ubuntu@rasmus: ~/Tull123
ubuntu@rasmus:~/Tull123$ nano hello.go
ubuntu@rasmus:~/Tull123$ GOOS=windows GOARCH=amd64 go build hello.go
ubuntu@rasmus:~/Tull123$ ls
hello  hello.exe  hello.go  LICENSE  README.md
ubuntu@rasmus:~/Tull123$ |
```

Her ser vi kommandoen for å gjøre om filen til en exe-fil og en "ls"-kommando for å bekrefte at filen ligger der. Filen vi bruker er hello.exe.

Etter dette lastet vi opp koden via github. Vi har ikke tatt bilder av dette ettersom vi regner med at folk kjenner til den vanlige måten å laste opp/ned kode eller prosjekter fra Github. Dette er kanskje ikke den optimale måten å gjøre det på, spesielt med tanke på at filen ble en del større enn vanlig kode, men det fungerte greit og det var få problemer med det. Etter at filen ble lastet ned, slettet vi den, fordi det generelt sett ikke er god praksis å ha slike filtyper på GitHub.

En bedre måte å overføre på er gjennom SCP. SCP står for Secure Copy Protocol, og er en god måte å overføre fra en remote maskin til en local. Dette kan gjøres ved å bruke terminalen slik: `scp -i ~downloads/Pem-filnavn`

`ubuntu@IP:/home/ubuntu/filnavn`. For meg vil det da se slik ut: "`scp -i ~downloads/RRS.pem ubuntu@IP:/home/ubuntu/Hello.exe .`" Da vil Hello.exe-filen bli lastet ned i den mappen jeg befinner meg i, i terminalen.

```
Tarjei@DESKTOP-EQSIGGA MINGW64 ~/ica02 (master)
$ scp -i ~/Downloads/keytax.pem ubuntu@[redacted]:/home/ubuntu/hello.exe .
The authenticity of host '[redacted]' can't be established.
ECDSA key fingerprint is [redacted].
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[redacted]' (ECDSA) to the list of known hosts.
hello.exe                                100% 2381KB  7.0MB/s  00:00

Tarjei@DESKTOP-EQSIGGA MINGW64 ~/ica02 (master)
$ ls
algorithms/  boring/  hello.exe*  keytax.pem  LICENSE  Oppgave2/  README.md  sum/
```

Oppgave 3

```
23
24 var sum_tests_int32 = []struct {
25     n1      int32
26     n2      int32
27     expected int32
28 }{
29     {1, 2, 3},
30     {-2, 5, 3},
31     {2147483640, 8, 2147483648},
32 }
33
34 func TestSumInt32(t *testing.T) {
35     for _, v := range sum_tests_int32 {
36         if val := SumInt32(v.n1, v.n2); val != v.expected {
37             t.Errorf("Sum(%d, %d) returned %d, expected %d", v.n1, v.n2, val, v.expected)
38         }
39     }
40 }
```

Her er et eksempel på hvordan man kan skrive det i ATOM. Dette er da int32.

```
PS C:\Go\Work\src\sumTommy\sum> go test
# sumTommy/sum
.\sum_test.go:13: constant 128 overflows int8
.\sum_test.go:31: constant 2147483648 overflows int32
.\sum_test.go:48: constant -2 overflows uint32
.\sum_test.go:85: constant 9223372036854775809 overflows int64
FAIL    sumTommy/sum [build failed]
PS C:\Go\Work\src\sumTommy\sum>
```

Det vi har gjort her er at vi har laget tester som produserer feil. Det vi får som feilmelding på int32 er "constant 2147483648 overflows". Det vil da si at den er ikke innenfor int32 sin rekkevidde.

Settet av alle signerte 32-bits heltall er kun innenfor dette rekkevidde (-2147483648 til 2147483647), men vi gikk utenfor denne rekkevidden, fordi da produserer den feil. Slik som i eksempelet over.

```

1  package main
2
3  import(
4      "os"
5      "strconv"
6      "fmt"
7  )
8
9  func main(){
10     ArgsA := os.Args[1]
11     ArgsB := os.Args[2]
12     i, _ := strconv.ParseFloat(ArgsA, 64)
13     j, _ := strconv.ParseFloat(ArgsB, 64)
14
15     fmt.Println (i + j)
16 }
17

```

Her har vi da tatt inn to parametere og skrevet de ut i Powershell. Da vil det bli slik som dette:

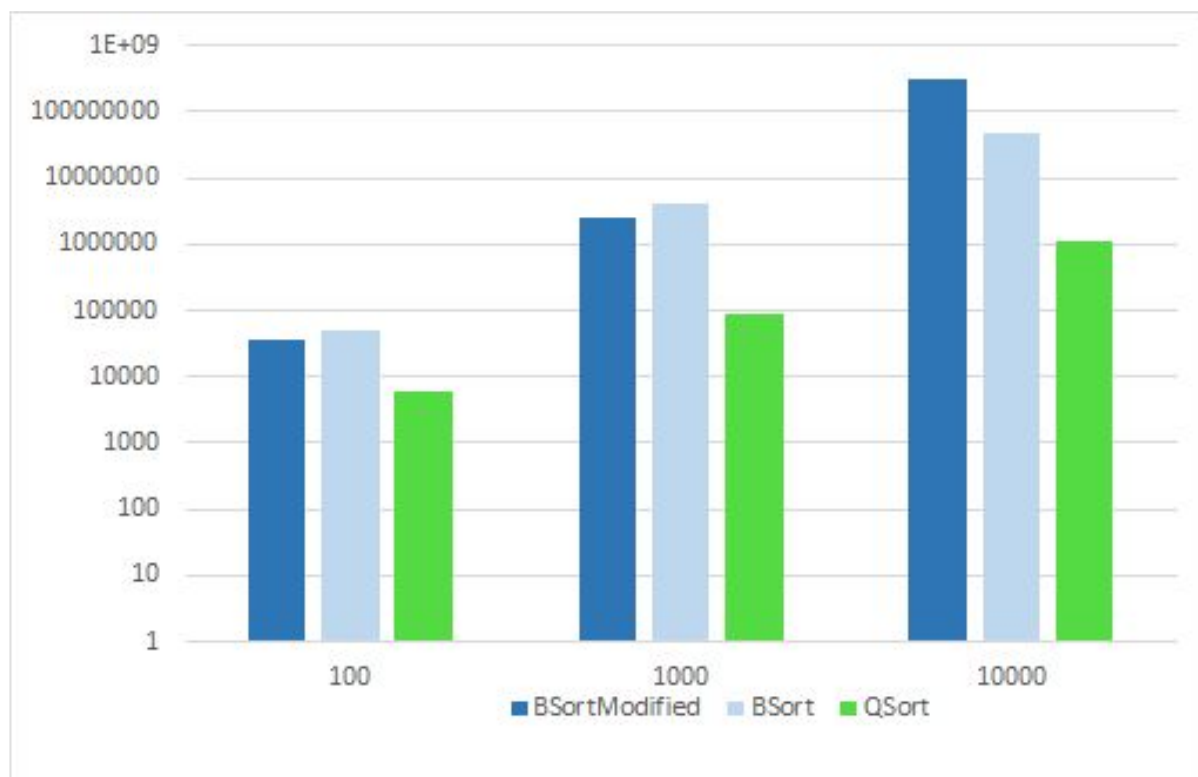
```

PS C:\Go\Work\src\sumTommy> go run main_sum.go 1 2
3

```

Det som skjer er at vi legger inn to argumenter (parametere) slik at den skal skrive det ut når vi prøver å kjøre denne i powershell. Det som blir gjort er at filen blir kjørt sammen med to variabler (tall) slik powershell kan regne ut "regnestykke" og dermed printer svaret rett under. 1+2=3.

Oppgave 4

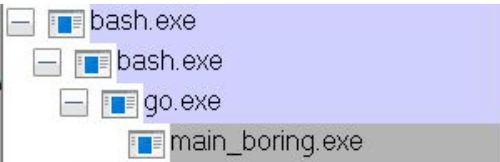


Logaritmisk graf som viser de tre forskjellige benchmark-testene opp mot hverandre. Benchmark-testene går gjennom en sortering av henholdsvis 100, 1000 og 10000 tall. Det kommer her tydelig fram effektiviteten til QuickSort sammenlignet med BubbleSort; ved 1000 tall ligger QuickSort på rundt 1 000 000 nanosekunder, mens begge BSort testene går forbi 50 000 000 ns (BSortModified ender her opp på langt over 300 000 000 ns).

Sammenlignet med Big-O regnearket får vi bekreftet det vi så ovenfor; til tross for at BSort har potensial til å være bedre enn QSort, er den vanligvis suboptimal. Det kommer også fram at BSortModified bare er optimal sammenlignet med BSort opp til et viss punkt. Dette er sammenlignbart med andre sorteringsalgoritmer hvor vanligvis optimal algoritme blir suboptimal sammenlignet med en annen etter et viss punkt. Selv om vi bruker relativt få verdier i testene, kan vi konkludere med at sorteringsfunksjonene er gjennomsnittlige i forhold til Big-O basert på resultatet.

Oppgave 5

Først kjørte vi `main_boring.go` via `bash`. Denne kjører kontinuerlig, og printer ut `index +1` hvert sekund. Vi lastet deretter ned Process Explorer. Man kan herfra også enkelt avslutte prosessen ved å bruke `ctrl-c` kommandoen.



bash.exe	6 648 K	9 400 K	4008
bash.exe	6 020 K	5 792 K	5580
go.exe	9 976 K	13 676 K	5592
main_boring.exe	< 0.01	3 152 K	6 376 K 2276

Her kan man se en liste over kjørende prosesser, sammen med utfyllende informasjon om hvor mye CPU de bruker, og hvor mange threads de er knyttet opp med. Man kan også enkelt avslutte prosessen herfra ved å høyreklikke på den og velge “kill process”.

Vi fant `bash` på denne listen, og valgte videre den kjørende boring prosessen.

main_boring.exe:2276 Properties

TCP/IP		Security		Environment	
Image	Performance	Performance Graph		GPU Graph	
Count: 5					
TID	CPU	Cycles	Delta	Start Address	
7380	< 0.01	308 669		main_boring.exe+0x53070	
6268	< 0.01	73 856		main_boring.exe+0x533c0	
6584	< 0.01	68 292		main_boring.exe+0x533c0	
3508				main_boring.exe+0x533c0	
124				main_boring.exe+0x533c0	

Her kan man se 5 forskjellige threads som er knyttet opp til boring prosessen.

Trykker man inn på disse står det at samtlige prosesser er sleeping, de venter altså på nye ordre.

```

27591 ubuntu 20 0 21388 5084 3148 S 0.0 0.1 0:00.08 bash
27628 ubuntu 20 0 205M 15124 8276 S 0.0 0.2 0:00.08 go run main_boring.go
27650 ubuntu 20 0 3500 1916 1500 S 0.0 0.0 0:00.00 /tmp/go-build604376607/command-line-arguments/_obj/exe/main_boring
27654 ubuntu 20 0 3500 1916 1500 S 0.0 0.0 0:00.00 /tmp/go-build604376607/command-line-arguments/_obj/exe/main_boring
27653 ubuntu 20 0 3500 1916 1500 S 0.0 0.0 0:00.00 /tmp/go-build604376607/command-line-arguments/_obj/exe/main_boring
27652 ubuntu 20 0 3500 1916 1500 S 0.0 0.0 0:00.00 /tmp/go-build604376607/command-line-arguments/_obj/exe/main_boring
27651 ubuntu 20 0 3500 1916 1500 S 0.0 0.0 0:00.00 /tmp/go-build604376607/command-line-arguments/_obj/exe/main_boring
27635 ubuntu 20 0 205M 15124 8276 S 0.0 0.2 0:00.00 go run main_boring.go
27634 ubuntu 20 0 205M 15124 8276 S 0.0 0.2 0:00.00 go run main_boring.go
27633 ubuntu 20 0 205M 15124 8276 S 0.0 0.2 0:00.00 go run main_boring.go
27632 ubuntu 20 0 205M 15124 8276 S 0.0 0.2 0:00.00 go run main_boring.go
27631 ubuntu 20 0 205M 15124 8276 S 0.0 0.2 0:00.00 go run main_boring.go
27630 ubuntu 20 0 205M 15124 8276 S 0.0 0.2 0:00.00 go run main_boring.go
27629 ubuntu 20 0 205M 15124 8276 S 0.0 0.2 0:00.00 go run main_boring.go

```

Mens vi kjørte main_boring.go på den virtuelle serveren gjennom et vindu, kunne vi se alle prosessene gjennom et nytt vindu. Her brukte vi et tillegg som het htop for lettere visualisering av prosesser.

```

ubuntu@tarjeitax:~$ ps aux | grep main_boring;
ubuntu 27628 0.0 0.1 210500 13352 pts/0 Sl+ 13:43 0:00 go run main_boring.go
ubuntu 27650 0.0 0.0 3500 1920 pts/0 Sl+ 13:43 0:00 /tmp/go-build604376607/command-line-arguments/_obj/exe/main_boring
ubuntu 27711 0.0 0.0 12944 1004 pts/1 S+ 13:57 0:00 grep --color=auto main_boring
ubuntu@tarjeitax:~$

```

Man kan også bruke kommandoen “ps aux | grep main_boring;” for å finne prosesser som inneholder main_boring i informasjonen. Her får vi opp 3 forskjellige prosesser.

```

top - 14:00:41 up 36 days, 23:22, 2 users, load average: 0.00, 0.00, 0.00
Threads: 8 total, 0 running, 8 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.0 sy, 0.0 ni, 99.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.1 st
KiB Mem : 8175056 total, 6912180 free, 111732 used, 1151144 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 7686672 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
 27628 ubuntu    20   0 210500 13352  8276 S   0.0   0.2   0:00.04 go
 27629 ubuntu    20   0 210500 13352  8276 S   0.0   0.2   0:00.00 go
 27630 ubuntu    20   0 210500 13352  8276 S   0.0   0.2   0:00.00 go
 27631 ubuntu    20   0 210500 13352  8276 S   0.0   0.2   0:00.00 go
 27632 ubuntu    20   0 210500 13352  8276 S   0.0   0.2   0:00.00 go
 27633 ubuntu    20   0 210500 13352  8276 S   0.0   0.2   0:00.00 go
 27634 ubuntu    20   0 210500 13352  8276 S   0.0   0.2   0:00.01 go
 27635 ubuntu    20   0 210500 13352  8276 S   0.0   0.2   0:00.00 go

```

Ved å bruke PID'en til den øverste prosessen (27628) kunne vi finne hvor mange tråder som er tilknyttet denne prosessen. Her kan vi også se hvilken status trådene har ved å se hvilke symbol de har under “S” (status). Vi kan da se at samtlige av trådene er sleeping.

Her er resultatet ved undersøkelse av main_boring_goroutine.go:

bash.exe	6 644 K	9 308 K	1080
bash.exe	6 008 K	5 732 K	388
go.exe	10 000 K	13 464 K	7808
main_boring_goroutine.exe	0.02	3 140 K	6 368 K 5224

CPU Usage: 16.47%	Commit Charge: 34.06%	Processes: 90	Physical Usage: 31.56%
-------------------	-----------------------	---------------	------------------------

main_boring_goroutine.exe:5224 Properties

TCP/IP		Security	Environment		Strings
Image	Performance	Performance Graph		GPU Graph	Threads

Count: 5

TID	CPU	Cycles Delta	Start Address
5220	0.01	879 834	main_boring_goroutine.exe+0x53170
7668	< 0.01	539 517	main_boring_goroutine.exe+0x534c0
7984	< 0.01	223 425	main_boring_goroutine.exe+0x534c0
5344	< 0.01	201 641	main_boring_goroutine.exe+0x534c0
544	< 0.01	42 787	main_boring_goroutine.exe+0x534c0

```
ubuntu@tarjeitax: ~/fis105-ica02
top - 12:30:41 up 36 days, 21:52,  2 users,  load average: 0.00, 0.02, 0.00
Threads:  8 total,   0 running,   8 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,   0.0 sy,   0.0 ni, 99.9 id,   0.0 wa,   0.0 hi,   0.0 si,   0.1 st
KiB Mem : 8175056 total, 6918744 free,  111336 used, 1144976 buff/cache
KiB Swap:   0 total,   0 free,   0 used. 7688744 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
 27070 ubuntu    20   0 210500 13152  8140 S   0.0   0.2   0:00.05 go
 27071 ubuntu    20   0 210500 13152  8140 S   0.0   0.2   0:00.00 go
 27072 ubuntu    20   0 210500 13152  8140 S   0.0   0.2   0:00.02 go
 27073 ubuntu    20   0 210500 13152  8140 S   0.0   0.2   0:00.00 go
 27074 ubuntu    20   0 210500 13152  8140 S   0.0   0.2   0:00.00 go
 27075 ubuntu    20   0 210500 13152  8140 S   0.0   0.2   0:00.00 go
 27076 ubuntu    20   0 210500 13152  8140 S   0.0   0.2   0:00.02 go
 27099 ubuntu    20   0 210500 13152  8140 S   0.0   0.2   0:00.01 go
```

Boring10 funksjonen vil nok være mer dominerende med å få designert prosessorkraft ettersom den skriver ut tall i et mye høyere tempo.

ICA03

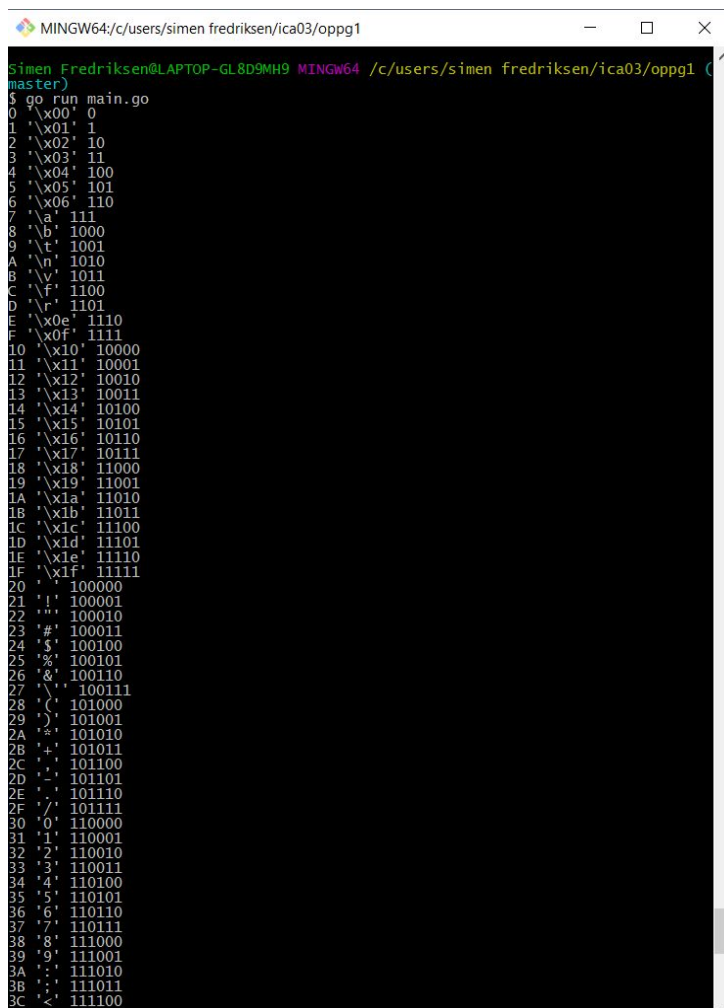
Deltakere: Simen Fredriksen, Stian Blankenberg, Jone Manneråk, Kristian Hagberg, Tarjei Taxerås og Rasmus Sørby.

Nicolay Bråthen Leknes og Tommy Nilsson var med ved første gjennomgang.

Repository: <https://github.com/Dieselgutta/ICA03>

Oppgave 1

a) Vi kjørte filen main.go i Git Bash. Da skriver vi ut en tabell med alle tegn i «string literal» ascii. Vi får ut alle ascii-kode heksadesimalt med store bokstaver, symbol for ascii-kode og ascii kode binært.



```
MINGW64:/c:/users/simen fredriksen/ica03/oppg1
Simen_Fredriksen@LAPTOP-GL8D9MH9 MINGW64 /c:/users/simen fredriksen/ica03/oppg1 (
master)
$ go run main.go
0 '\x00' 0
1 '\x01' 1
2 '\x02' 10
3 '\x03' 11
4 '\x04' 100
5 '\x05' 101
6 '\x06' 110
7 '\a' 111
8 '\b' 1000
9 '\t' 1001
A '\n' 1010
B '\v' 1011
C '\f' 1100
D '\r' 1101
E '\x0e' 1110
F '\x0f' 1111
10 '\x10' 10000
11 '\x11' 10001
12 '\x12' 10010
13 '\x13' 10011
14 '\x14' 10100
15 '\x15' 10101
16 '\x16' 10110
17 '\x17' 10111
18 '\x18' 11000
19 '\x19' 11001
1A '\x1a' 11010
1B '\x1b' 11011
1C '\x1c' 11100
1D '\x1d' 11101
1E '\x1e' 11110
1F '\x1f' 11111
20 ' ' 100000
21 '!' 100001
22 '"' 100010
23 '#' 100011
24 '$' 100100
25 '%' 100101
26 '&' 100110
27 '\'' 100111
28 '(' 101000
29 ')' 101001
2A '*' 101010
2B '+' 101011
2C ',' 101100
2D '-' 101101
2E '.' 101110
2F '/' 101111
30 '0' 110000
31 '1' 110001
32 '2' 110010
33 '3' 110011
34 '4' 110100
35 '5' 110101
36 '6' 110110
37 '7' 110111
38 '8' 111000
39 '9' 111001
3A ':' 111010
3B ';' 111011
3C '<' 111100
```

Vi fant ingen forskjeller når vi kjørte programmet på nettskyen (UH-laaS).

På instansen i nettskyen (UH-laaS) så utskriften slik ut:

```
ubuntu@simen: ~/ICA03/Oppg1
ubuntu@simen:~/ICA03/Oppg1$ go run main.go
0 '\x00' 0
1 '\x01' 1
2 '\x02' 10
3 '\x03' 11
4 '\x04' 100
5 '\x05' 101
6 '\x06' 110
7 '\a' 111
8 '\b' 1000
9 '\t' 1001
A '\n' 1010
B '\v' 1011
C '\f' 1100
D '\r' 1101
E '\x0e' 1110
F '\x0f' 1111
10 '\x10' 10000
11 '\x11' 10001
12 '\x12' 10010
13 '\x13' 10011
14 '\x14' 10100
15 '\x15' 10101
16 '\x16' 10110
17 '\x17' 10111
18 '\x18' 11000
19 '\x19' 11001
1A '\x1a' 11010
1B '\x1b' 11011
1C '\x1c' 11100
1D '\x1d' 11101
1E '\x1e' 11110
1F '\x1f' 11111
20 ' ' 100000
21 '!' 100001
22 '"' 100010
23 '#' 100011
24 '$' 100100
25 '%' 100101
26 '&' 100110
27 '\'' 100111
28 '(' 101000
29 ')' 101001
2A '*' 101010
2B '+' 101011
2C ',' 101100
2D '-' 101101
2E '.' 101110
2F '/' 101111
```

Osv...

b) Når vi lagde en funksjon `greetingASCII()` i samme filen `ascii.go`, som skriver ut "Hello :-)" og utførte programmet på instansen i nettskyen (UH-laaS) ble det seende

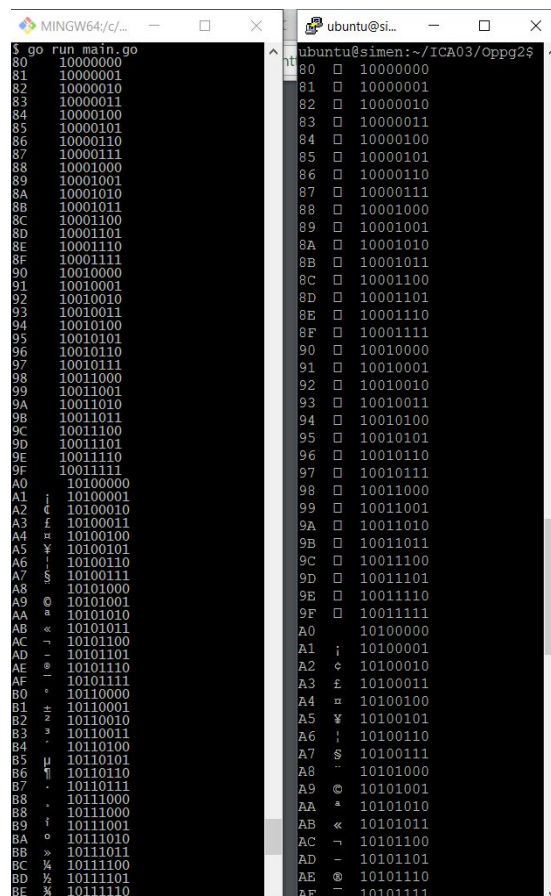
slik ut:

```
76 'v' 1110110
77 'w' 1110111
78 'x' 1111000
79 'y' 1111001
7A 'z' 1111010
7B '{' 1111011
7C '|' 1111100
7D '}' 1111101
7E '~' 1111110
7F '\u007f' 1111111
"Hello :~)"ubuntu@simen:~/ICA03/Oppg1$
```

c) Koden for testen ligger i Github-repositoriet.

Oppgave 2

a) Vi kjørte filen main.go til oppgave 2 i Git Bash og i nettskyen (UH-laaS) . Da genererte den alle tallene og symbolene fra byte-verdier, til Ascii. Det så slik ut(Git Bash til venstre og nettskyen til høyre):



```
80 10000000
81 10000001
82 10000010
83 10000011
84 10000100
85 10000101
86 10000110
87 10000111
88 10001000
89 10001001
8A 10001010
8B 10001011
8C 10001100
8D 10001101
8E 10001110
8F 10001111
90 10010000
91 10010001
92 10010010
93 10010011
94 10010100
95 10010101
96 10010110
97 10010111
98 10011000
99 10011001
9A 10011010
9B 10011011
9C 10011100
9D 10011101
9E 10011110
9F 10011111
A0 10100000
A1 10100001
A2 10100010
A3 10100011
A4 10100100
A5 10100101
A6 10100110
A7 10100111
A8 10101000
A9 10101001
AA 10101010
AB 10101011
AC 10101100
AD 10101101
AE 10101110
AF 10101111
```

Dette er bare en liten del av det som kom opp da vi kjørte programmet.

Tankegangen vår var litt lik som i oppgave 1, men vi gjorde endringer for at den

skulle iterere. Vi samarbeidet mye i gruppen for å komme fram til det riktige resultatet. Vi ser at vi får et problem når vi skal printe ut ukjente symboler.

b) Tankegangen var lik som i oppgave 1. Vi brukte samme main-funksjon som i oppgave a. Da vi kjørte programmet så det slik ut: Her ser vi at både oppgave a og b blir printet ut samtidig, dette er fordi vi har brukt samme main-fil til de to ettersom de hører til samme mappe (iso).

```
FD  ý  11111101
FE  þ  11111110
FF  ý  11111111
"Salut, ça va ☺) ☐50"ubuntu@simen:~/ICA03/Oppg2$
```

Vi fikk ikke ut € symbolet, i stedet viser nettskyen et rektangulært symbol foran "50".

c) Koden for testen ligger oppe på Github-repositoriet.

Oppgave 3

a) %s printer ut symboler for bytes av en string

%q printer ut symboler for en string

%+q printer ut symboler for Unicode som er utenfor ASCII kodesett.

%c printer ut symboler som den tolker innenfor utvidet ASCII kodesett.

I byte-sekvensen må man endre slik at det blir

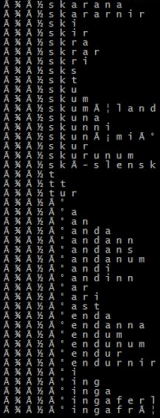
«\x22\xC2\xBD\x3F\x3D\x3F\x20\xE2\x8C\x98\x22»

Vi får ikke ut ☸ symbolet når vi printer, i stedet viser den et rektangulært symbol; dette tyder på at den ikke fant symbolet.

```
Simen Fredriksen@LAPTOP-
$ go run 3a.go
"½?=? ☐"
Simen Fredriksen@LAPTOP-
```

b) Vi har under her skrevet ut lang01.wl, lang02.wl og lang03.wl. Resultatene vi fikk ser ut til å være de samme som var i de originale filene, bare at øverst er de skrevet ut i heksadesimaler, og under er de skrevet ut i vanlig string. Men der ser vi også at noen av printene her ikke skrev de ut nøyaktig som de stod, dette er trolig fordi at de forskjellige språkene reagerer annerledes med den samme printen, som her ser ut til å ha en innstilling til å bruke det engelske kodesettet. Noe som resulterte i at alle tegnene som ikke er til felles med det engelske kodesettet ble sendes uleselig ut.

[illegible][illegible]



- “EF DA A3 D2 D3 CB 0A EF DA A3 D2 D3 CB C1 0A EF ...” i unicode blir “u00ef, u00da, u00a3, u00d2, u00d3, u00cb, \n, u00ef, u00da, u00a3, u00d2, u00d3, u00cb, u00c1, \n, u00ef..”
- “FE FD 73 6B 61 72 0A FE FD 73 6B 61 72 61 6E 61 ...” i unicode blir “u00fe, u00fd, s, k, a, r, \n, u00fe, u00fd, s, k, a, r, n, a”.
- “F8 79 65 73 70 65 73 69 61 6C 69 73 74 65 6E 0A ...” i unicode blir “ u00f8, y, e, s, p, e, s, i, a, l, i, s, t, e, n, \n”.

c)

```
Simen Fredriksen@LAPTOP-GL8D9MH9 MINGW64 /c/users/simen fredriksen/  
$ go run 3c_main.go  
Henrik Arnold Wergeland (født 17. juni 1808, død 12. juli 1845)  
Vi ere en nasjon vi med,  
vi små en alen lange,  
et fedreland vi frydes ved,  
og vi, vi ere mange.  
Vårt hjerte vet, vårt øye ser  
hvor godt og vakkert Norge er,  
vår tunge kan en sang blant fler  
av Norges æres-sange.  
  
Mer grønt er gresset ingensteds,  
mer fullt av blomster vevet  
enn i det land hvor jeg tilfreds  
med far og mor har levet.  
Jeg vil det elske til min død,  
ei bytte det hvor jeg er fødd,  
om man et paradís meg bød  
av palmer oversvevet.
```

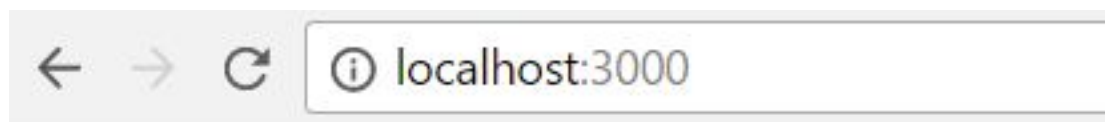
Oppgave 4

a) Ved å skrive jp bak filen får man Japansk.


Ved å skrive Is bak filen får man Islandsk.

```
Simen Fredriksen@LAPTOP-GL8D9MH9 MINGW64 /c/users/  
$ go run 4a.go jp  
"nord og sør" på japansk er "北と南"  
  
Simen Fredriksen@LAPTOP-GL8D9MH9 MINGW64 /c/users/  
$ go run 4a.go is  
"nord og sør" på islandsk er "norður og suður"  
  
Simen Fredriksen@LAPTOP-GL8D9MH9 MINGW64 /c/users/  
$
```

b) Ved å bruke U+23F0, får vi ut et klokke symbol, som vist i bildet.



Hvordan gjør det, P?

 Saturday, 13-May-17 15:13:27 CEST

ICA04

Deltakere: Simen Fredriksen, Stian Blankenberg, Jone Manneråk, Kristian Hagberg, Tarjei Taxerås og Rasmus Sørby.

Repository: <https://github.com/Dieselgutta/ICA04>

Oppgave 1

a) I windows brukes 0D 0A for å bryte linjen, mens Mac/Unix brukes kun 0A for å bryte linjen. Dette er fordi at Windows-systemet må ha en “Carriage Return” før line-breaker-en, mens Mac OS ikke trenger det. Dette er for at det ikke var behov for en driver når man skulle printe. Dette fører til 1 mer byte per linjeskift. Problemet kan være at filen kan inneholde en del kode som kan mistolkes når det blir overført fra et operativsystem til et annet. Eller andre veien, at det ikke inneholder nok kode for å kjøres ordentlig.

b)

```
~\Documents\GitHub\ICA04\Oppg1 [master = +1 ~1 -0 !]> go run main.go -f files\te
xt1.txt
22 54 65 73 74 65 72 20 6C 69 6E 6A 65 73 6B 69 66 74 2E 0D 0A 20 20 20 20 4F 67
20 65 6E 20 74 69 6C 20 2E 2E 2E 0D 0A 20 20 20 20 4F 67 20 65 6E 20 74 69 6C 2
0 2E 2E 2E 0D 0A 20 20 20 20 4F 67 20 65 6E 20 74 69 6C 20 2E 2E 2E 22 0D 0A ["
e s t e r   l i n j e s k i f t .
      0 g   e n   t i l   . . .
      0 g   e n   t i l   . . .
      0 g   e n   t i l   . . . "
122 54 65 73 74 65 72 20 6C 69 6E 6A 65 73 6B 69 66 74 2E 0D 0A 20 20 20 20 4F 6
7 20 65 6E 20 74 69 6C 20 2E 2E 2E 0D 0A 20 20 20 20 4F 67 20 65 6E 20 74 69 6C
20 2E 2E 2E 0D 0A 20 20 20 20 4F 67 20 65 6E 20 74 69 6C 20 2E 2E 2E 22 0D 0A ["
T e s t e r   l i n j e s k i f t .
      0 g   e n   t i l   . . .
      0 g   e n   t i l   . . .
      0 g   e n   t i l   . . . "
]
Det er 4 Carriage returns.
~\Documents\GitHub\ICA04\Oppg1 [master = +1 ~1 -0 !]>
```

Bilde 1: Vi ser at en tekstfil laget i Windows har “Carriage return”.

```
[krs-vg-087:Oppg1 jone.skribeland$ go run main.go -f /Users/jone.skribeland/docum
ents/ICA04/Oppg1/files/tekst.txt
68 65 6C 6C 6C 6F 0A C3 85 73 73 65 6E 20 73 74 C3 A5 72 20 72 65 20 74 65 3F 0A
3A 29 0A [h e l l o
  Å s s e n   s t Å v r   r e   t e ?
: )
]68 65 6C 6C 6C 6F 0A C3 85 73 73 65 6E 20 73 74 C3 A5 72 20 72 65 20 74 65 3F 0
A 3A 29 0A [h e l l o
  Å s s e n   s t Å v r   r e   t e ?
: )
]
Det er 0 Carriage returns.
krs-vg-087:Oppg1 jone.skribeland$
```

Bilde 2: Vi ser at en tekstfil laget i Mac ikke har Carriage return.

Vi skrev en kode som telte antall linjeskift, slik at vi skrev ut “Det er”, antall Carriage returns, “Carriage returns”.

Vi visste fra oppgave 1a at linjeskift representeres ved 0A og 0D 0A.

0A er for mac og unix, dette er lik LF som er \n.

0D 0A er for windows, dette er lik CRLF som er \r\n.

I kodene over kan du se

Oppgave 2

a) Som vi ser her gir programmet navnet på filen, størrelsen i bytes, permission for unix og den viser også andre ting som som det er en symbolsk link, om det er en Unix block eller character fil, om den er append only, om filen er en directory og om det er en regular file.

For å komme fram til dette brukte vi Lstat og FileInfo fra “Os” i Golang-biblioteket.

På denne måten kunne vi vise informasjonen enkelt. Koden i seg selv er relativt enkel og baseres på at vi sjekker om noe stemmer ved å bruke if og else-setninger og printer ut svaret via Println.

b)

```
ubuntu@magilou-worship:~/ICA04/Oppg2$ go run fileinfo.go -f /dev/stdin
Information about '/dev/stdin':
Size: 0B, 0.000000KiB, 0.000000MiB, 0.000000000GiB
Name : stdin
Size : 0
Mode/permission : Dcrw--w----
File is not a symbolic link
File is not append only
Is a device file
File is a char device file
File is not a block device file
Is a directory? : false
Is a regular file? : false
Unix permission bits? : -rw--w----
Permission in string : Dcrw--w----
```

Bilde 3 (stdin). Vi endret filnavnet her til /dev/stdin i koden for å finne stdin i ubuntu.

```

ubuntu@magilou-worship:~/ICA04/Oppg2$ go run fileinfo.go -f /dev/loop0
Information about '/dev/loop0':
Size: 0B, 0.000000KiB, 0.000000MiB, 0.000000000GiB
Name : loop0
Size : 0
Mode/permission : Drw-rw----
File is not a symbolic link
File is not append only
Is a device file
File is a not char device file
File is a block device file
Is a directory? : false
Is a regular file? : false
Unix permission bits? : -rw-rw----
Permission in string : Drw-rw----
ubuntu@magilou-worship:~/ICA04/Oppg2$

```

Bilde 4 (loop0). da det ikke var mulig å bruke ram0 så ble loop0 brukt i stedet..

Vi ser her at vi fikk veldig forskjellige resultater. Et av dem var at “loop0” er en block device file, mens “stdin” er en char device file. .

c) Eventuelle forskjeller vi ser her fra det forrige er at størrelsen er større (10 mot 6). Dette er pga. måten windows bruker linjeskift. Bortsett fra størrelsen er alt likt som i ubuntu.

Oppgave 3

a) De forskjellige kategoriene innen metoder for å arbeide med filer er:

Grunnleggende Operasjoner

Her er helt grunnleggende metoder som create/get file info/open/delete

Lesing og Skrivning

Her er kopiering av filer, flere nøyaktige metoder for å legge til tekst, samt flere nøyaktige metoder for å lese innholdet til en fil. Bruker ta typisk bytes for å komme fram til ønsket posisjon.

Arkivering

Zipper og unzipper filer. (Samler og “sprer” filer)

Komprimering

Komprimerer og dekomprimerer en fil.

Diverse

Diverse spesifikke metoder, som å hente en fil fra HTTP, eller opprette en midlertidig fil som blir slettet etter at dens funksjon er brukt opp.

b)

```
Simen Fredriksen@LAPTOP-GL8D9MH9 MINGW64 /c/users/simen fredriksen/ica04/oppg3 (
master)
$ go run scanner.go text1.txt ./Oppg3
Lines: 13
Runes: 220
Words: 33

5 mest brukte runer:
Antall: 50 Rune:
Antall: 11 Rune: t
Antall: 9 Rune: n
Antall: 8 Rune: "
Antall: 6 Rune: s
```

bilde 5: Her ser vi at programmet klarte å skrive ut antall runer og antall linjer i tekstfilen. Og de 5 mest brukte runene i tekstdokumentet.

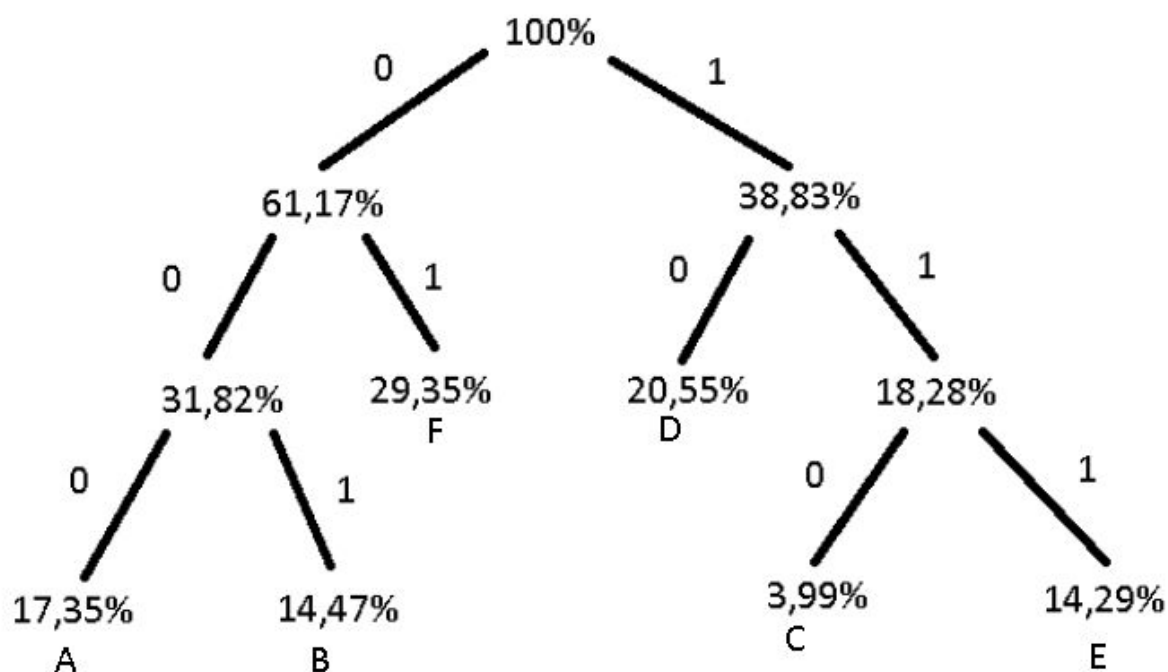
Oppgave 4

a) Beregning: Studenter i ett fakultet / Totalt antall studenter *100% = Sannsynlighet.

UiA's fakultet	Kode	Antall studenter (totalt 10539)	Sannsynlighet
Helse- og idrettsfag (A)	000	1829	17,35%
Humaniora og pedagogikk (B)	001	1525	14,47%
Kunstfag (C)	110	420	3,99%
Teknologi og realfag (D)	10	2166	20,55%
Lærerutdanning (E)	111	1506	14,29%
Økonomi og samfunnsvitenskap (F)	01	3093	29,35%

b) Du får minst informasjon fra fakultet F (Økonomi og Samfunnsvitenskap) og D (Teknologi og realfag) ettersom deres fakultets-kode er henholdsvis 01 og 10 i motsetning til resten av fakultetene som har 3 bits kode.

c)



d)

Kode	% * Bit-lengde	Gj. bits
000	17,35 * 3	52,05
001	14,47 * 3	43,41
110	3,99 * 3	11,97
10	20,55 * 2	41,10
111	14,29 * 3	42,87
01	29,35 * 2	58,70

Summert sammen blir dette 255,1 bits for en gjennomsnittlig lenke som inneholder fakultets-koder for 100 tilfeldige elever.

ICA05

Deltakere: Simen Fredriksen, Stian Blankenberg, Jone Manneråk, Kristian Hagberg, Tarjei Taxerås og Rasmus Sørby.

Repository: <https://github.com/Dieselgutta/ICA05>

Server på Ubuntu ved <http://158.39.77.237:8001/>

Tankegang for kode:

Golang:

Vi satte opp en enkel webserver ved å bruke `HandleFunc` og `ListenAndServe` på port 8001. Vi brukte en funksjon som laget klienten i func `basicHandler`. Denne sørger for at vi får en respons på siden og den knytter `template/index.html` til serveren for å få det opp som en nettside. Vi brukte en API fra `OpenWeatherMap` som vi dekodet i `Func decode`. Denne decoder dataen som blir hentet ut fra funksjonen `func getData`. Denne dataen er da tilgjengelig, og vi bruker den til å sette opp "structs". De forskjellige parameterne som er gruppert i `api-en` blir samlet i en struct som videre blir samlet i en struct kalt "Weather". Her blir også variablene som ikke har en gruppe lagt inn (eks. `Name string`). I `API-en` er temperaturen definert ved Kelvin-måleenheten. Dette blir gjort om til Celsius ved å ta temperaturen minus det absolutte nullpunktet (-273.15).

HTML:

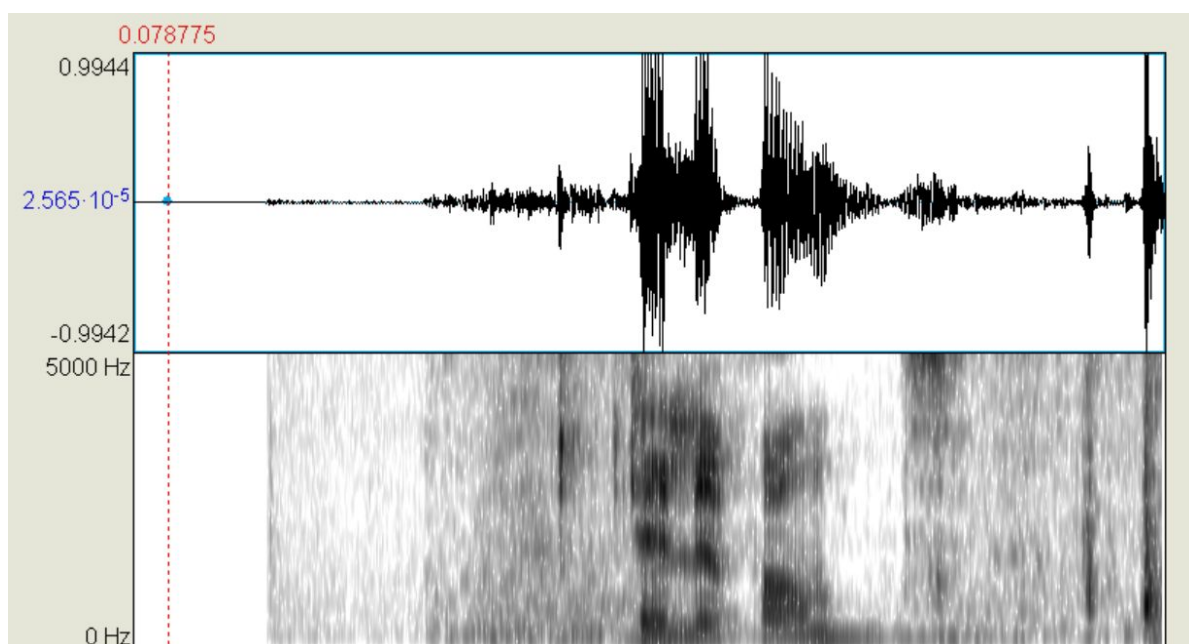
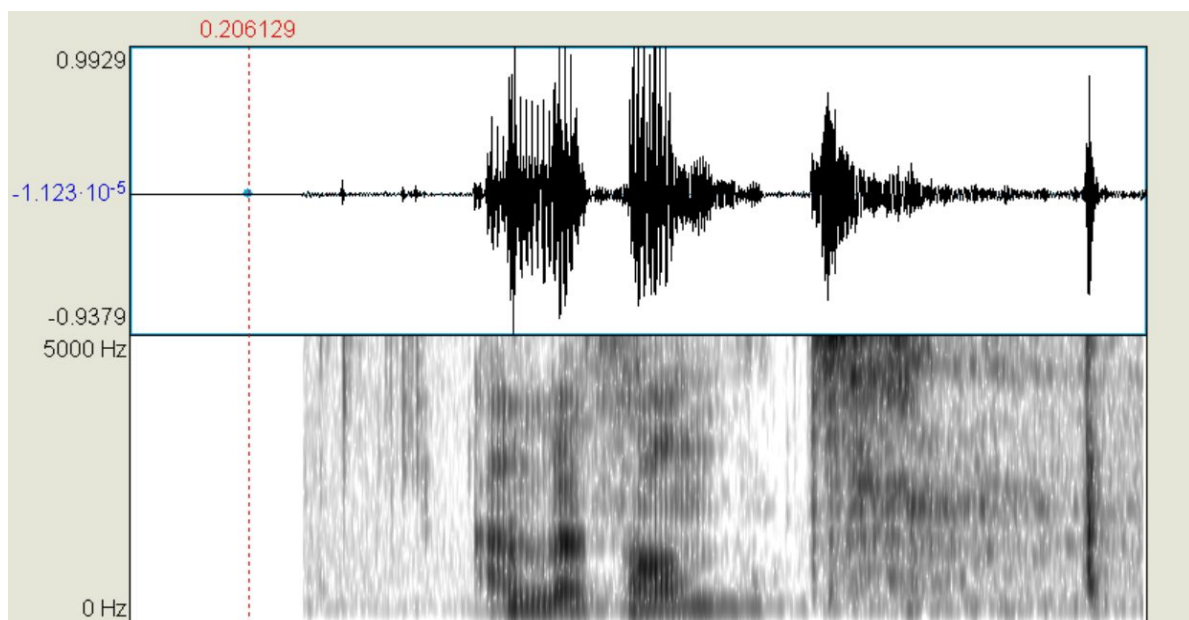
`Index.html` blir brukt som en mal (template) for nettsiden. Her bruker man diverse HTML-koder for å utforme sidens innhold. Vi bruker et stylesheet fra `w3schools.com` for å definere de forskjellige HTML-kodene som vi bruker. Dette innebærer endringer som farge og andre designmuligheter. HTML-siden kaller på Golang sine structs for å hente ut/vis fram informasjon hentet fra `API-en`. Vi har også embedded et kart fra google maps som bruker `Name` og `Country` fra Struct for å definere lokasjon. Denne ble enkelt generert ved hjelp av <http://www.embedgooglemap.net/>.

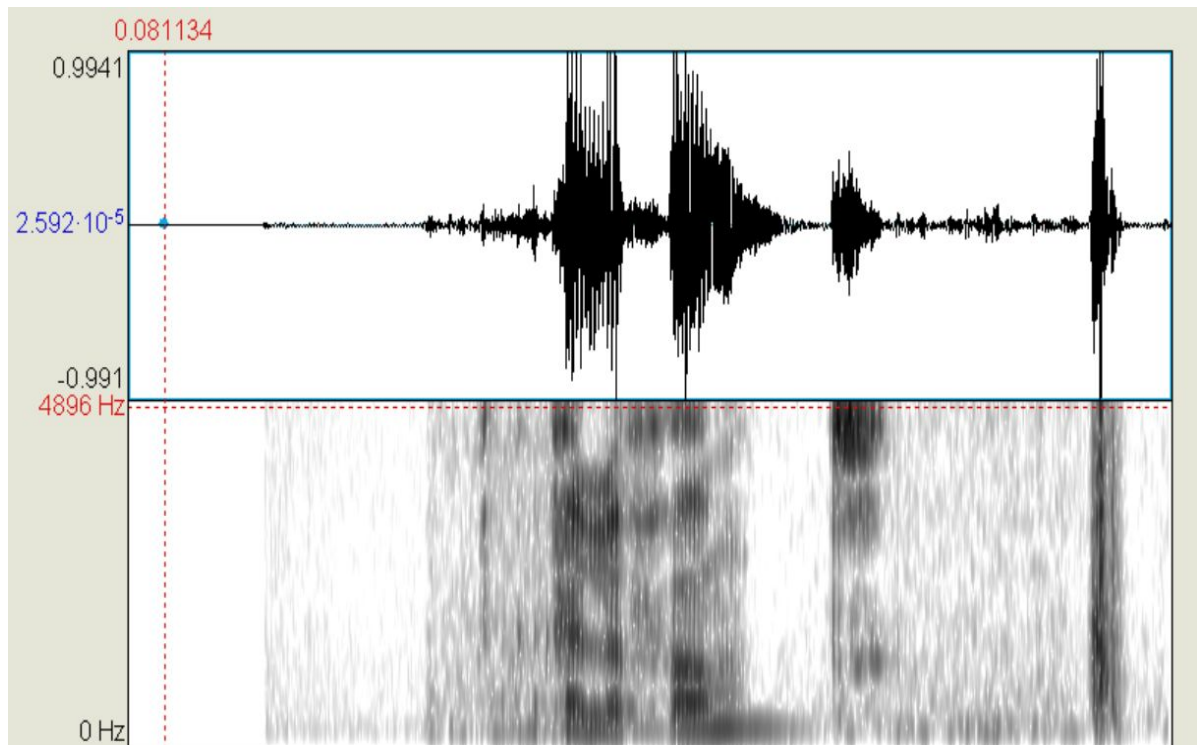
ICA 06

Deltakere: Simen Fredriksen, Stian Blankenberg, Jone Manneråk, Kristian Hagberg, Tarjei Taxerås og Rasmus Sørby.

Repository: <https://github.com/Dieselgutta/ICA06>

Eksperiment 1





Ordet vi valgte å undersøke for lydfrekvenser var “elefant”. Dette ordet ga oss mulighet til å sammenligne to e-vokaler i forskjellig sammenheng, samt en a-vokalen. Det mest tydelige som kom fram her var uttalelsen til den første e-vokalen. Ved å sammenligne formantenes frekvenser opp mot en tabell for ulike vokaler, kunne vi fort se en sammenheng mellom dialekt og uttalelse av e. Mens en sørlending kom nærmere æ på tabellen, endte en østlending med en noe finere ε. Ellers var det større likhet ved den andre e-vokalen, samt a-vokalen hvor formant-frekvensene stemte overens med en a.

Hvis vi skulle laget en språkgjennkjennelses-algoritme hadde hovedfokuset nok ligget rundt formantenes tilknytning til hverandre. Ved hjelp av en kort setning burde det være mulig å “luke vekk” alle de språk-alternativene som ikke inneholder en tilsvarende oppbygging av formant-forhold. Identifisering av det som på engelsk heter “hisses and pops” ville nok vært til stor nytte her; dette er naturlige lyder som oppstår ved tale, og som kan være til hjelp for å skille språk fra hverandre.

Eksperiment 2

I eksperiment 2 brukte vi eksempelet fra <https://github.com/parente/espeakbox>. Vi måtte laste ned docker på ubuntu, deretter kjørte vi programmet med denne kommandoen: `sudo docker run --name espeakbox -d -p 8080:8080 parente/espeakbox`. Man kan endre tekst, pitch, språk og hurtighet på teksten som

skal komme ut i taleform i adresselinjen. I det eksempelet vi ar linket under ser vi at text=hei, voice=no (no betyr at det skal uttales på norsk), pitch = 80 og speed = 170.

<http://158.39.77.30:8080/speech?text=hei&voice=no&pitch=80&speed=170>

Eksperiment 3

Vi prøvde å bruke google sin speech API, slik som det er i dette eksempelet: <https://www.google.com/intl/en/chrome/demos/speech.html>, men fikk det ikke til. Vi slet med å få API nøkkelen til å fungere. Vi prøvde også med wit-engine, men slet med å forstå hvordan denne fungerte.

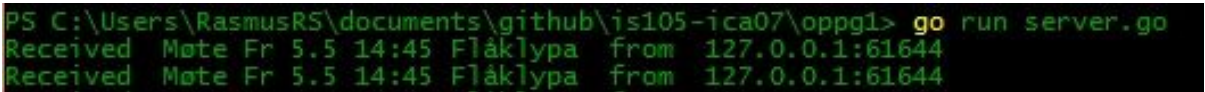
ICA07

Deltakere: Simen Fredriksen, Stian Blankenberg, Jone Manneråk, Kristian Hagberg, Tarjei Taxerås og Rasmus Sørby.

Repository: <https://github.com/Dieselgutta/ICA07>

Oppgave 1:

- a. Koden for UDP klient og tjener finnes i GitHub.

b. 

- c.

i)

1. $18/50 = 0,36 \times 100\% = 36\%$
2. En UDP-pakke kan maksimum teoretisk sett være 65507 bytes, men det er uvanlig å benytte seg av dette. Den vanligste størrelsen er 512 bytes. Det finnes også andre pakker som 508 eller 548. Jo større de er, jo større er sannsynligheten for tap av data eller vanskeligheter med sending.

ii)

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	UDP	50	56454 → 1234 Len...
2	0.000248	127.0.0.1	127.0.0.1	UDP	74	1234 → 56454 Len...
3	34.451403	127.0.0.1	127.0.0.1	UDP	282	60964 → 32376 Le...
4	34.451454	127.0.0.1	127.0.0.1	ICMP	60	Destination unre...
5	34.480236	127.0.0.1	127.0.0.1	UDP	289	60964 → 32376 Le...
6	34.480267	127.0.0.1	127.0.0.1	ICMP	60	Destination unre...
7	34.480587	127.0.0.1	127.0.0.1	UDP	129	60964 → 32376 Le...
8	34.480619	127.0.0.1	127.0.0.1	ICMP	60	Destination unre...
9	34.480918	127.0.0.1	127.0.0.1	UDP	219	60964 → 32376 Le...
10	34.480942	127.0.0.1	127.0.0.1	ICMP	60	Destination unre...

▶ Frame 2: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0

▶ Null/Loopback

▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

▶ User Datagram Protocol, Src Port: 1234, Dst Port: 56454

▶ Data (42 bytes)

```

0000 02 00 00 00 45 00 00 46 8c c1 00 00 40 11 00 00 ....E..F ....@...
0010 7f 00 00 01 7f 00 00 01 04 d2 dc 86 00 32 fe 45 .....2.E
0020 46 72 6f 6d 20 73 65 72 76 65 72 3a 20 4d c2 af From ser ver: M..
0030 74 65 20 46 72 20 35 2e 35 20 31 34 3a 34 35 20 te Fr 5. 5 14:45
0040 46 6c c3 82 6b 6c 79 70 61 20                               Fl..klyp a

```

Loopback: lo0: <live capture in progress> Packets: 10 - Displayed: 10 (100.0%) Profile: Default

1)

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	UDP	50	56454 → 1234 Len...
2	0.000248	127.0.0.1	127.0.0.1	UDP	74	1234 → 56454 Len...

▶ Frame 1: 50 bytes on wire (400 bits), 50 bytes captured (400 bits) on interface 0

▶ Null/Loopback

▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

▶ User Datagram Protocol, Src Port: 56454, Dst Port: 1234

▶ Data (18 bytes)

```

0000 02 00 00 00 45 00 00 2e c3 1a 00 00 40 11 00 00 ....E... ....@...
0010 7f 00 00 01 7f 00 00 01 dc 86 04 d2 00 1a fe 2d .....-
0020 46 72 6f 6d 20 63 6c 69 65 6e 74 3a 20 6e c3 82 From cli ent: n..
0030 72 3f                                     r?

```

Loopback: lo0: <live capture in progress> Packets: 2 - Displayed: 2 (100.0%) Profile: Default

Oppgave 2:

a. Koden for TCP klient og tjener finnes i GitHub.

b.

i) TCP oppretter en kobling mellom to hoster og sjekker trafikken. Slik at den vet om den kommer frem. UDP blir sendt uten noen garanti for at det kommer frem. UDP er også raskere.

ii) En TCP pakke kan være 1500 bytes ved sending over et nettverk. Den kan også være større, men man vil da støte på flere problemer som forkortelse av en pakke, eller tap av data.

iii) Fragmentering er når den fysiske plassen på en harddisk ikke blir brukt til optimal kapasitet. Det oppstår som regel ved at noen datalagringsalgoritmer er ineffektive. Fragmentering kan løses ved defragmentering. Da kjører man en omfattende scan på harddisken og filene på den. Deretter vil den bestemme den beste måten å strukturere og lagre dine data.

iv) Man kan bruke UDP ved streaming av spill eller samtaler over nettet som skype. Da vil man kunne miste pakker, og det vil kunne "lagge"/hakke litt i samtalen eller spillet.

TCP kan brukes ved sending av data hvor tap av data vil kunne ødelegge innholdet, slik at for eksempel en mail blir ufullstendig. UDP fungerer bedre for direktesendte data enn TCP

Oppgave 3:

a. Koden ligger under mappen UDP i Github. Vi brukte eksemplet i denne funksjonen `func ExampleNewGCM_encrypt`, som ligger her

https://golang.org/src/crypto/cipher/example_test.go.

Kildereferanser

ICA01

<https://github.com/chinatsu/is105-uke04/blob/master/logbcli/main.go>

ICA02

ICA03

ICA04

<https://github.com/Gruppe12IS105/ICA03/tree/master/ICA03%204>

ICA05

<https://github.com/Gruppe12IS105/ICA05/blob/master/Websrv/websrv.go>

<https://github.com/Gruppe12IS105/ICA05/blob/master/Websrv/templates/index.html>

<http://www.embedgooglemap.net/>

ICA06

<https://github.com/IS-105-GitGroup/IS-105-Gruppe1/blob/master/ICA04/src/oppgaver/oppgave3.go>

ICA07

<http://stackoverflow.com/questions/26028700/write-to-client-udp-socket-in-go/26032240#26032240>

<http://www.minaandrawos.com/2016/05/14/udp-vs-tcp-in-golang/>

https://golang.org/src/crypto/cipher/example_test.go