Institutt for datateknologi og informatikk

Eksamensoppgave i TDT4109 - Informasjonsteknologi, grunnkurs

Løsningsforslag

Eksamensdato: 1. desember 2021

Eksamenstid (fra-til): 09:00 – 13:00

Hjelpemiddelkode/Tillatte hjelpemidler: A / Alle hjelpemidler tillatt

Oppgave 1 (25%)

Denne oppgaven teller 25% av karakteren.

Svar på disse teorioppgavene på en kort, komplett og konsis måte.

1a - komprimering (5%)

Denne oppgaven har to deler:

1. Forklar forskjellen på Lossy og Lossless komprimering.
2. Gi et eksempel på hvordan "run-length encoding" kan brukes til å komprimere noe.

1a) I lossless (tapsfri) komprimering er det fortsatt mulig å gjenskape den originale strukturen fullt og helt med å reversere komprimeringen, og ingen data har blitt tapt. I lossy komprimering utnytter en det faktum at mennesker ikke klarer å oppfatte alle detaljer like bra, og fjerner de detaljene som vi ikke er så gode på å legge merke til. Det gjør at vi for alltid mister detaljer og gjør kvaliteten dårligere siden vi ikke kan gjenskape det. Lossy komprimering egner seg godt på f.eks video, bilder og lyd, men ikke på tekstdokumenter hvor en må bruke lossless komprimering

b) Run-length enkoding benytter seg av at data ofte har mange repeterende strukturer etter hverandre, og istedenfor å kode hver eneste verdi av disse, koder en heller hvor mange av de som kommer etter hverandre. Et eksempel er bilder, som kan ha mange like pixler. Si et bilde består av 200 røde pixler etter hverandre, så komprimerer en det med å si at her kommer det 200 røde pixler. Dette funker bra når det er mye repeterende strukturer i dataen, men kan ende opp med å ta mer plass enn original-dataen dersom dette ikke stemmer.

1b - minne (5%)

Forklar kort de ulike minnene vi benytter oss av i en datamaskin, og når vi benytter oss av de forskjellige minnetypene.

Vi deler minner opp i fire deler, hvor det for hver del kan lagres mer data, det blir billigere å lage, men det blir og treigere å hente data fra.

- Registre - Små, og veldig raskere minner som ligger på CPUen som er veldig dedikerte til en spesifikk oppgave. Brukes for å lagre instruksjoner for oppgaver CPUen skal gjøre, og som mellomlagre for verdier som CPUen skal regne med
- Cache - Brukes for å effektivisere bruken av RAM, og lagrer ting en regner med en skal hente fra RAM snart for å hindre at RAM blir flaskehalsen i systemet.
- RAM - Datamaskinens arbeidsminne. Ligger ikke i CPUen. Det er her all informasjon om programmet datamaskinen kjører ligger, og det er her CPUen henter data og instruksjoner den skal kjøre for å kjøre et program.
- Sekundær lagring - Brukes for alt som datamaskinen skal lagre når strømmen har skrudd seg av, og for persistent lagring av all data.

1c - TCP/UDP (5%)

Forklar forskjellen på TCP og UDP, og når vi bør benytte oss av den ene eller andre.

Både TCP og UDP fungerer i transport-laget i nettverket og har som mål å transportere datapakker til applikasjonen hvor det skal brukes. I UDP gis det ingen garanti for at dataen kommer frem, den blir sendt fra server til destinasjon kun en gang. I TCP gis det en garanti for at dataen skal komme frem, ved at det konstant sendes bekreftelser frem og tilbake mellom sender og mottaker for å sørge for at alle pakker har kommet frem. Dette gjør at UDP egner seg veldig godt dersom hastighet er viktig siden det går mye raskere og sende data over UDP, samtidig som det går fint om noen detaljer mangler, som f.eks ved streaming av video. Dersom alle detaljer bør være der, eller en har god tid på å sende egner TCP seg derimot mye bedre, f.eks ved sending av mail.

1d - sikkerhet (5%)

Forklar de ulike begrepene Konfidensialitet, Integritet og Tilgjengelighet i et sikkerhets-perspektiv, og hva en ondsinnet angriper kan gjøre for å bryte med disse prinsippene.

- **Konfidensialitet** (hemmelighold): Kun autoriserte personer får tilgang til spesifikk data og system
- **Integritet**: Kun autoriserte personer kan bruke og endre data og system
- **Tilgjengelighet:** Autoriserte personer skal alltid ha tilgang til deres data og systemer

Eksempler på brudd

- **Konfidensialitet:** Du logger inn på en nettside, men blir logget inn som en annen bruker og har mulighet til å se all data om dem.
- **Integritet**: Ondsinnede angripere klarer å utnytte en sårbarhet til å trekke bank-kontoen din for penger.
- **Tilgjengelighet:** Ondsinnede angripere overbelaster en server med falske forespørsler for å gjøre at tjenester ikke responderer til faktiske brukere. (*Denial of service*)

1e - IP (5%)

Forklar hvorfor overgangen fra IPV4 til IPV6 er så vanskelig, og hvorfor vi må gå over til ny versjon.

IPv4 benytter seg av 32 bit for IP-adresser, mens IPv6 benytter seg 128 bit. Det gjør at IPv6 kan ha veldig mange fler adresser enn IPv4. På grunn av at veldig mange flere mennesker har koblet seg til internett, og pga "Internet of Things" er veldig mange fler enheter og koblet til internett har vi brukt opp tilgjengelige IPv4 adresser, og har måttet gjøre flere triks for å klare og fortsatt bruke det i internett. Derfor er vi nødt til å gå over til IPv6 for å få flere IP-adresser tilgjengelig.

Overgangen er vanskelig fordi milliarder av servere og infrastruktur må oppgraderes for å kunne kommunisere med denne nye protokollen, og før disse er oppgradert kan de ikke benytte seg av IPv6, men kun den gamle IPv4 protokollen.

Oppgave 2 (40%)

Oppgave 2 består av 9 ulike oppgaver (2a - 2i) med programmering og dra-og-slipp / innfylling i programkode. Til sammen utgjør disse oppgavene 40% av karakteren. På oppgaver hvor du kan teste om koden din virker, vil du normalt score mer poeng ved å ha et program som passerer iallfall noen av testene, enn ved å ha et program som prøver å få til alt, men får kjørefeil og ikke passerer noen av testene. Det er likevel bedre å levere noe kode selv om den ikke virker, enn å levere blankt, da sensor med et levert svar kan ha mulighet til å se over koden manuelt og vurdere om det kan gis litt poeng for delvis riktig tankegang.

2a - Max Manus (3%)

Pythons standardfunksjon max() returnerer den største av verdiene i en sekvens. For eksempel vil max("manus") gi verdien "u" fordi u er lengst ut i alfabetet av bokstavene i manus. Tilsvarende vil min("manus") gi verdien "a". Skriv en funksjon alfa(streng, tall) som får inn en tekststreng pluss et tall. Hvis tallet er 0 skal funksjonen returnere det minste tegnet i strengen (dvs. fremst i alfabetet), hvis tallet er noe annet enn 0, skal den returnere det største tegnet i strengen.

```
def alfa(streng, tall):
    if tall == 0:
        return min(streng)
    else:
        return max(streng)
```

2b - Byer i Belgia (3%)

Gitt ei ordbok kalt geografi som har land som nøkkelverdi, og for hver nøkkel ei liste av byer i dette landet. Et eksempel på mulig innhold er: geografi = {'Irland': ['Dublin', 'Cork'], 'Polen': ['Lodz', 'Krakow', 'Gdansk'], 'Belgia': ['Bryssel', 'Gent', 'Liege', 'Namur'] } Funksjonen ant_land(geografi) skal returnere antallet land som fins i ordboka. Med eksemplet over skal dette kallet returnere tallet 3 fordi det fins info om tre land i ordboka. Hint: funksjonen len() gir antall element i f.eks. strenger, lister, tupler, mengder, ordbøker. NB: Funksjonen din skal også virke for ordbøker med annet antall land enn eksemplet gitt her, men strukturen kan alltid antas å være den samme (land som nøkkelverdi, så liste med byer for hvert land).

```
def ant_land(geografi):
    return len(geografi)
```

2c - Sjekk tall (4%)

Skriv en funksjon sjekk(tall) som får inn et tall. Hvis tallet er et oddetall, skal funksjonen returnere strengen "odde", hvis partall returnere strengen "par", og hvis tallet har en desimaldel > 0, skal funksjonen returnere strengen "des". (Dvs. f.eks., 3 -> "odde", 3.0 -> "odde", 3.1 -> "des", 4.0 -> "par", 4.2 -> "des") NB: Funksjonen skal returnere tekststrengen, IKKE printe den.

```
def sjekk(tall):
    if tall % 2 == 0:
```

```
        return "par"
    elif tall % 2 == 1:
        return "odde"
    else: # desimaldel vil komme med i rest ved modulo
        return "des"
```

2d - Sum av lengder (5%)

Funksjonen sum_len(L) får inn ei liste L som inneholder heltall, og skal returnere summen av lengdene til tallene i lista. F.eks. sum_len ([34, 502, 4]) skal returnere 6 fordi det første tallet er 2 siffer langt, det andre 3 siffer langt, og det tredje 1 siffer, og 3+2+1 er 6. Trekk kodefragmenter til riktig sted så funksjonen virker som den skal. (NB: Det fins flere kodefragment enn hull i koden, dvs. en del av fragmentene skal ikke brukes)

```
def sum_len(liste):
    summen = 0
    for tall in liste:
        summen += len ( str (tall) )
    return summen
```

2e - Største tverrsum (5%)

Tverrsummen til et tall er summen av sifrene, f.eks. tverrsummen til 125 er 1+2+5, altså 8. Anta at det allerede fins en funksjon tverrsum(tall), du trenger ikke lage denne. Vi skal nå lage funksjonen max_tverrsum(L), som får inn som parameter ei liste med heltall. Funksjonen skal returnere den største tverrsummen blant tallene i lista, samt indeksposisjonen denne ble funnet på. For eksempel, max_tverrsum ([8, 12, 77, 1111]) skal returnere tuppelet (14, 2) fordi den største tverrsummen er 14, og denne får vi for tallet 77 som står på indeks 2 i lista. Lista som kommer inn som parameter antas å alltid inneholde minst ett tall, dvs. er aldri tom. Oppgave: Velg riktige alternativ nedenfor, slik at koden for max_tverrsum() vil funke. Det blir IKKE gitt minuspoeng for feilsvar, så du BØR fylle inn alle hull heller enn å sette noe blankt.

```
# Antar ved starten at første element er det største
# ved å sette mpos = 0 og msum = tverrsum(L[0]), dette
# er en vanlig løsningsteknikk for å finne største element i
liste.
# Dermed kan løkka deretter være range(1,...) siden første element
# allerede er tatt med i beregningen.
def max_tverrsum(L):
    mpos = 0
    msum = tverrsum(L[0])
    for i in range(1, len(L)):
        if tverrsum(L[i]) > msum:
            mpos = i
            msum = tverrsum(L[i])
    return (msum, mpos)
```

2f - Ordbok fra ordbok (5%)

Gitt ei ordbok tallbok hvor nøklene er strenger som er forkortelser for språk (norsk, svensk, ...), og hvor verdien på hver nøkkel er ei liste med ord for tallene 0-6 på dette språket. tallbok

= {'NO': ['null', 'en', 'to', 'tre', 'fire', 'fem', 'seks'], 'SE': ['noll', 'ett', 'två', 'tre', 'fyra', 'fem', 'sex'], 'DK': ['null', 'en', 'to', 'tre', 'fire', 'fem', 'seks'], 'IT': ['zero', 'uno', 'due', 'tre', 'cuattro', 'cinque', 'sei'], 'ES': ['cero', 'uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis'], 'PT': ['zero' 'um', 'dois', 'tres', 'cuatro', 'cinco', 'seis'] } Vi skal lage en funksjon same_word(num, tallbok) som får inn som parameter et heltall (0-6) og ei ordbok med struktur som vist over. Den skal returnere ei ny ordbok hvor nøklene er ord for dette tallet, og hvor verdifeltet er lista over alle språk som har dette ordet for tallet - såframt mer enn ett språk har samme ord. Eksempel på resultat ved kjøring: same_word(3, tallbok) returnerer {'tre': ['NO', 'SE', 'DK', 'IT'], 'tres': ['ES', 'PT']} same_word(6, tallbok) returnerer {'seks': ['NO', 'DK'], 'seis': ['ES', 'PT']} - merk at her er nøklene SE og IT ikke med i returnert ordbok fordi hver av disse var alene om sin skrivemåte for tallet 6.

Trekk kodefragmentene til riktig sted så funksjonen virker som den skal. (3 av fragmentene skal ikke brukes). Det blir ikke gitt minuspoeng for feilplasserte kodefragment, så du bør fylle inn alle hull heller enn å sette noe blankt.

```
# Funksjonen bruker to ordbøker, temp og result, for å få til
# at vi unngår å få med i sluttresultatet result de språkene
# som IKKE har det aktuelle tallordet felles med noen andre.
# Dvs., ordbok-innslagene kopieres over fra temp til result
# først når det er minst 2 innslag i temp for det aktulle ordet
def same_word(num, tallbok):
    temp, result = {}, {} # starter med å lage to tomme ordbøker
    for landkode in tallbok: # går i løkke gjennom alle språkene
        tallord = tallbok[landkode][num] # tallordet som språket
har for tallet num
        if tallord in temp:
            temp[tallord].append(landkode) # legger til nytt språk
for tallord
            result[tallord] = temp[tallord] # kopierer over til
result siden det nå er minst 2
        else:
            temp[tallord] = [landkode] # tallord ikke funnet før,
legger inn i temp
    return result
```

2g - Fjerne serieduplikat (5%)

Vi ønsker en funksjon fjern_dup(liste) som får inn ei liste med tall som parameter, og som skal fjerne serieduplikater i lista, dvs. tilfeller hvor samme tall kommer flere ganger rett etter hverandre. Som man kan se av gitte test cases: 1. Her kommer ingen tall flere ganger på rad. Dermed endres lista ikke. 2. Tallet 4 kommer to ganger på rad, derfor fjernes en av disse firerne 3. Lista har flere på rad både av 2'ere, 3'ere, 1'ere, i alle disse tilfellene fjernes tall fra lista slik at det ikke kommer identiske tall på rad NB: Funksjonen skal gjøre muterende endring i lista som kommer inn som parameter, den trenger derfor ikke returnere noe. Den skal heller ikke printe noe.

```
# Løsning hvor man lager ny liste, deretter muterer
# denne inn i opprinnelig liste
# NB: tilordningen her MÅ være liste[:] = ny_liste,
# hvis vi gjør liste = ny_liste, blir det en ikke-muterende
endring,
```

```
# da blir den nye lista kun puttet i en lokal variabel i
funksjonen,
# og endrer ikke lista som ble gitt inn som argument.
def fjern_dup(liste):
    ny_liste = [liste[0]]
    for tall in liste:
        if tall != ny_liste[-1]:
            ny_liste.append(tall)
    liste[:] = ny_liste

# Løsning hvor man fjerner dupliserte element direkte
# i den opprinnelige lista. Merk at while-løkke her er
# mer hensiktsmessig enn for-løkke, da vi ikke vet ved starten
# av løkka hvor mange runder den skal gå, siden det kan
# forsvinne elementer fra lista underveis.
# liste.pop(i) fjerner elementet på indeks i.
# i += 1 skal bare gjøres hvis vi ikke popper, siden
# hvis vi popper vil neste element komme til indeks i
# selv om i er uendret. Ved at vi starter med i=1, ikke i=0,
# vil testen liste[i] == liste[i-1] alltid gå bra, og det nullte
elementet
# skal jo uansett aldri fjernes...
def fjern_dup(liste):
    i = 1
    while i < len(liste):
        if liste[i] == liste[i-1]:
            liste.pop(i) # eller: del liste[i]
        else:
            i += 1
```

2h - Liste fra streng (5%)

Funksjonen list_str(s) skal få inn en streng s som parameter. Den skal returnere ei liste med bokstaver, nærmere bestemt de bokstavene i strengen s som har tegnet 'f' 1 tegn foran seg, og tegnet 'f' 6 tegn bak seg. Eksempel: list_str('abcdefX') skal returnere ei tom liste [ ]. Bokstaven X har riktig nok 'f' 1 tegn foran seg, men har ikke 'f' 6 tegn bak seg. list_str('fedcabYabcdef') skal returnere ei tom liste [ ]. Bokstaven Y har riktig nok 'f' 6 tegn bak seg, men bokstaven 'f' står ikke 1 tegn foran Y. list_str('xyzzy') skal returnere ei tom liste [ ]. Ingen av bokstavene i strengen har påkrevd tegn foran eller bak. list_str('abcdefSabcdefOabcdefSabcdef') skal returnere ['S', 'O', 'S'] fordi akkurat disse tre tegnene i strengen har 'f' 1 tegn foran seg og 'f' 6 tegn bak seg.

```
# En utfordring her er å unngå at for-løkka går utenfor indeks når
# tegnet vi vurderer skal sammenlignes med tegn lenger framme og
bak i strengen.
# Enkleste måte er å starte løkka det antall inn i strengen som er
avstand til
# tegnet foran, og slutte løkka det antall som er avstand til
tegnet bak,
# dvs. i stedet for range(0, len(s)) ha range(5, len(s)-1)
# når vi her skal ha 'b' 5 tegn foran og 'a' 1 tegn bak,
# slik at vi kun vurderer de tegnene som kan være aktuelle
kandidater for lista.
```

```
# Hvis man ikke kom på denne ideen men i stedet bruker ei løkke
(0, len(s)) må man
# teste at i-5 < 0, og at i+1 < len(s), FØR man putter disse
uttrykkene som indeks for s.
def list_str(s):
    liste = []
    for i in range(5,len(s)-1):
        if s[i-5] == 'b' and s[i+1] == 'a':
            liste.append(s[i])
    return liste
```

2i - Sjekk to og to (5%)

Funksjonen sjekk_to(streng) får inn en streng som alltid er 6 tegn lang, og hvor hvert tegn er et siffer, f.eks. '051033'. Følgende regler gjelder for disse sifrene: Tallet som dannes av de to første sifrene, kan maks være 27. Tallet som dannes av de to midterste sifrene, kan maks være 16. Tallet som dannes av de to bakerste sifrene, kan maks være 52. Funksjonen skal returnere: 1 hvis strengen er ok, jfr. test #1 - alle verdier er innenfor grensene. 2 hvis to første siffer gir for høyt tall, jfr. test #2 med 42 mens maks tillatt var 27. 3 hvis to midterste siffer gir for høyt tall, jfr. test #3 med 20 mens maks tillatt var 16. 5 hvis to bakerste siffer gir for høyt tall, jfr. test #4 med 99 mens maks tillatt var 52. 6 hvis både fremste to og midterste to er for høye, jfr. test #5. 10 hvis både fremste to og bakerste to er for høye, jfr. test #6. 30 hvis både fremste, midterste, bakerste er for høye, jfr. test #7. Merk: Funksjonen må fungere generelt for strenger med 6 siffer, ikke bare for de strengene som er gitt som test case her.

```
# Løsning med for-løkke, oppnådd ved å sette opp
# tuppel (2,3,5) for faktorer det skal ganges med for ulike feil
# og et tilsvarende tuppel for grenseverdiene, hvorpå løkka
# går tre runder og slicer ut de tre delene av strengen.
# I stedet for tupler kan man selvsagt også bruke lister.
def sjekk_to(streng):
    resultat = 1
    factors = (2, 3, 5)
    limits = (27, 12, 52)
    for i in range(3):
        if int(streng[i*2:i*2+2]) > limits[i]:
            resultat *= factors[i]
    return resultat

# Løsning hvor man i stedet bruker en serie if-setninger,
# en for overskridelse av hver grenseverdi.
# Merk at man må ha tre uavhengige if-setninger her, ikke elif,
# siden man kan ha brutt en, to eller tre grenser uavhengig av
# hverandre.
def sjekk_to(streng):
    resultat = 1
    if int(streng[:2]) > 27:
        resultat *= 2
    if int(streng[2:4]) > 12:
        resultat *= 3
    if int(streng[4:]) > 52:
        resultat *= 5
    return resultat
```

```
# Løsning med enda flere if-setninger (hvis man ikke kom på
# at 6 var 2 x 3, 10 var 2 x 5, og 30 var 2 x 3 x 5, som
muliggjorde
# å få riktige resultat ved enkel multiplikasjon i løsningene
over)
# elif-setningen som returnerer 15 trenger ikke være med, siden
dette
# caset ikke var nevnt i oppgaveteksten og det heller ikke var
noen
# test cases verken for student eller sensor som prøvde dette, så
den
# vil overhodet ikke påvirke resultatet, men er
# tatt med av estetiske grunner siden det er en rimelig antagelse
at
# det skal være slik, og dermed gir en mer komplett løsning.
def sjekk_to(streng):
    resultat = 1
    if int(streng[:2]) > 27 and int(streng[2:4]) > 12 and
int(streng[4:]) > 52:
        return 30
    elif int(streng[2:4]) > 12 and int(streng[4:]) > 52:
        return 15
    elif int(streng[:2]) > 27 and int(streng[4:]) > 52:
        return 10
    elif int(streng[:2]) > 27 and int(streng[2:4]) > 12:
        return 6
    elif int(streng[4:]) > 52:
        return 5
    elif int(streng[2:4]) > 12:
        return 3
    elif int(streng[:2]) > 27:
        return 2
    else:
        return 1
    return resultat
```

Oppgave 3 (35%)

Oppgave 3 består av et sett med deloppgaver innenfor samme tema. I enkelte oppgaver vil man kunne bruke funksjoner skrevet i tidligere oppgaver for å hjelpe seg. Man kan velge å kalle disse funksjonene selv om man ikke har klart å løse dem - det er bare å 'late som' i besvarelsen. Du skal nå lage et program for en dagligvarebutikk. Programmet holder styr på lager, håndterer oppdatering av lager og salg. Dataen lagres i fil når butikken stenger, og leses fra fil når butikken åpnes. Butikken kan også generere middagsforslag til en gitt pris. Oppgaven baserer seg på strukturene matvarer og beholdning.

'matvarer' er en todimensjonal liste med følgende struktur: [[navn, pris, rett],[navn2, pris2, rett2], ...]

Hver innerliste inneholder en string 'navn', et heltall 'pris' og en string 'rett' for en matvare.

matvarer = [['laks', 59, 'middag'], ['kjøttdeig', 25, 'middag'], ['ris', 15, 'middag'], ['ost', 99, 'frokost'], ['bønner', 7, 'middag'], ['soyasaus', 33, 'middag'],['banan', 4, 'mellommåltid']]

'beholdning' er en dictionary med følgende struktur: {navn:antall, navn2:antall2, ...}

Hver nøkkel er en string 'navn', og hver verdi er et heltall 'antall' som beskriver hvor mange av den spesifikke varen som er på lager. Du kan anta at alle varer som finnes i beholdning også finnes som en vare i matvarer. Andre eventuelle antagelser må skrives i din besvarelse.

beholdning = {'laks': 6, 'kjøttdeig': 14, 'ris': 15, 'ost': 9, 'bønner': 6, 'soyasaus': 0, 'banan': 1}

Tallene i eksemplene til oppgavene forutsetter at man har fått til tidligere oppgaver.

3.1 - finn_pris(matvarer, let_etter) (4%)

Vi trenger å finne prisen på matvarene i butikken.

Skriv funksjonen finn_pris(matvarer, let_etter): Du skal finne prisen til en bestemt matvare, gitt av parameteret 'let_etter' i den todimensjonale listen 'matvarer'. Hvis let_etter ikke finnes i matvarer skal funksjonen returnere 0, ellers skal prisen på matvaren returneres.

Parametre: 'matvarer' er den todimensjonale listen som beskrevet i starten av oppgave 3. 'let_etter' er en streng som inneholder matvaren som det letes etter.

Eksempel: >>> print(finn_pris(matvarer, 'ost')) 99 >>> print(finn_pris(matvarer, 'finnesikke')) 0

```
def finn_pris(matvarer, let_etter):
    for vare in matvarer:
        if let_etter == vare[0]:
            return vare[1]
    return 0
```

3.2 - oppdater_matvare(beholdning, matvare, antall) (4%)

Butikken har behov for å kunne oppdatere en og en vare i beholdningen.

Skriv funksjonen oppdater_matvare(beholdning, matvare, antall): hvis 'antall' har en negativ verdi skal beholdningen gå med med tilsvarende mengde. du kan anta at alle matvarer man får inn som parametre allerede eksisterer. du kan anta at du ikke får en oppdatering som gir en negativ beholdning av en matvare. returner den oppdaterte beholdningen.

Parametre: dictionaryen 'beholdning' som beskrevet i starten av oppgave 3. en streng 'matvare'. heltallet 'antall'.

Eksempel på kjøring: >>> print(beholdning['kjøttdeig']) 14 >>> oppdater_matvare(beholdning, 'kjøttdeig', -1) >>> print(beholdning['kjøttdeig']) 13

```
def oppdater_matvare(beholdning, matvare, antall):
    beholdning[matvare] += antall
    return beholdning
```

3.3 - oppdater_beholdning(beholdning, endringer) (4%)

Butikken får påfyll av ulike matvarer når det skjer en leveranse. Da må beholdningen oppdateres. På samme måte må beholdningen reduseres når noen kjøper noe i butikken.

Skriv funksjonen oppdater_beholdning(beholdning, endringer): legg til/trekk fra endringer til beholdningen. returner den oppdaterte beholdningen.

Parametre: 'beholdning' er som tidligere. 'endringer' er en todimensjonal liste med matvarer og antall på format: [[navn,antall], [navn2,antall2], ...]

Eksempel: >>> print(beholdning) {'laks': 6, 'kjøttdeig': 13, 'ris': 15, 'ost': 9, 'bønner': 6, 'soyasaus': 0, 'banan': 1} >>> beholdning = oppdater_beholdning(beholdning, [['ost', 2], ['kjøttdeig', -4], ['bønner', -1]]) >>> print(beholdning) {'laks': 6, 'kjøttdeig': 9, 'ris': 15, 'ost': 11, 'bønner': 5, 'soyasaus': 0, 'banan': 1}

```
def oppdater_beholdning(beholdning, endringer):
    for endring in endringer:
        oppdater_matvare(beholdning, endring[0], endring[1])
    return beholdning
```

3.4 - vis_priser(beholdning, matvarer, tekst) (5%)

Butikken trenger å vite hvilke varer som er i sortimentet!

Skriv funksjonen vis_priser(beholdning, matvarer, tekst): finn hvilke matvarer som er i 'tekst', fjern unødvendige tegn og ord som ikke er matvarer. returner en liste av tupler der hvert tuppel inneholder navn på vare og pris på den varen. du kan anta at 'tekst' er i lower-case.

Parametre: 'beholdning' og 'matvarer' er lik tidligere. 'tekst' er en string med matvarer og eventuelle andre ord, der ordene er skilt ved mellomrom. Teksten kan inneholde '.' og ','. F.eks: 'bønner og kjøttdeig, med banan.'

Eksempel: >>> print(vis_priser(beholdning, matvarer, 'bønner med soyasaus, og ris')) [('bønner', 7), ('soyasaus', 33), ('ris', 15)]

```
def vis_priser(beholdning, matvarer, tekst):
    varer = tekst.replace(',','').replace('.','').split(' ')
    liste = []
    for elem in varer:
```

```
        if elem in beholdning:
            liste.append((elem, finn_pris(matvarer, elem)))
    return liste
```

3.5 - salg(matvarer, beholdning, handleliste) (5%)

Butikken er opptatt av glade kunder! Derfor ønsker butikken å gi kundene mulighet til å levere en handleliste. Dette er en liste med alle matvarene kunden ønsker. Hvis kunden ønsker å handle flere av samme vare så vil matvaren gjentas (som ['ost', 'ost', 'banan']). Butikken skal selge matvarer helt til det er tomt for dem. Hvis kunden ønsker to ost, men det finnes kun en, så skal én ost selges.

Funksjonen salg(matvarer, beholdning, handleliste) skal gjøre følgende ting: Skrive ut alle matvarer som selges, og hvilken pris de selges for. Oppdatere beholdning for å reflektere salget som er gjennomført. Skrive ut sluttsummen for alle matvarer som ble solgt. Returner et tuppel bestående av alle matvarer som ble solgt.

Parametre: 'matvarer' og 'beholdning' er som tidligere. 'handleliste' er en liste med matvarer. Du kan anta at alle matvarene finnes i både 'matvarer' og 'beholdning'.

Eksempel: >>> print(salg(matvarer, beholdning,['ost','banan','banan', 'kjøttdeig', 'kjøttdeig'])) ost 99 banan 4 kjøttdeig 25 kjøttdeig 25 Totalsum: 153 ('ost', 'banan', 'kjøttdeig', 'kjøttdeig')

```
def tilgjengelig_vare(beholdning, vare):
    if beholdning[vare] >= 1:
        return True
    return False

def salg(matvarer, beholdning, handleliste):
    total, tup = 0, ()
    for vare in handleliste:
        if tilgjengelig_vare(beholdning, vare):
            pris = finn_pris(matvarer, vare)
            print(vare, pris)
            oppdater_matvare(beholdning, vare, -1)
            total += pris
            tup = tup + (vare,)
    print('Totalsum:', total)
    return tup
```

3.6 - skriv_beholdning(beholdning), les_beholdning() (5%)

Når butikken stenger må varebeholdningen lagres til fil, og så må varebeholdningen leses inn fra fil igjen når butikken åpner. Skriv funksjonene skriv_beholdning(beholdning) og les_beholdning().

skriv_beholdning(beholdning) skal: lagre innholdet i dictionaryen 'beholdning' på et passende format. hvis filen finnes fra før, skal denne skrives over.

Parametre skriv_beholdning(beholdning): 'beholdning' er som tidligere. les_beholdning() skal: lese den samme filen, opprette den samme dictionaryen, og returnere den.

Du velger selv format og struktur på filen, men det kan jo være enkelt å ta utgangspunkt i den samme folderen som programmet kjøres fra.

NB! Husk at eventuelle moduler må importeres. Filen trenger ikke være lesbar for mennesker, det du måles på er at det fungerer.

```
# med binærfil
import pickle

def skriv_beholdning(beholdning):
    with open('varebeholdning.dat', 'wb') as fil:
        pickle.dump(beholdning, fil)

def les_beholdning():
    with open('varebeholdning.dat', 'rb') as fil:
        beholdning = pickle.load(fil)
    return beholdning

# med tekstfil
# def skriv_beholdning(beholdning):
#     skriveliste = []
#     for nokkel in beholdning:
#         skriveliste.append(nokkel +
';'+str(beholdning[nokkel])+'\n')
#     with open('beholdning.txt', 'w') as fil:
#         fil.writelines(skriveliste)

# def les_beholdning():
#     beholdning = {}
#     with open('beholdning.txt', 'r') as fil:
#         linjer = fil.readlines()
#     for linje in linjer:
#         linje = linje.strip(' ').strip('\n')
#         delt = linje.split(';')
#         beholdning[delt[0]] = int(delt[1])
#     return beholdning
```

3.7 - tilfeldig_middag(matvarer, budsjett) (8%)

Noen ganger trenger kunder litt inspirasjon for å lage seg en middag. Lag en funksjon tilfeldig_middag(matvarer, budsjett) som skal generere en tilfeldig middag med forskjellige matvarer av retten 'middag' innenfor gitt budsjett. Funksjonen trenger ikke ta hensyn til om matvaren er på lager, den skal kun gi inspirasjon til middagsretter.

Parametre: 'matvarer' er som tidligere. 'budsjett' er et heltall.

tilfeldig_middag(matvarer, budsjett) skal: velge tilfeldige matvarer i kategorien 'middag' som man har nok penger til å kjøpe, duplikat av matvarer er ikke lov. holde styr på hvor mye de tilfeldige varene koster. returnere en liste med tilfeldige middagsvarer som koster tilsammen maks 'budsjett' når det ikke lenger finnes varer som kan legges til. løsningen skal implementeres ved hjelp av en REKURSIV FUNKSJON på en vesentlig måte, enten funksjonen tilfeldig_middag selv, eller en hjelpefunksjon som gjør hoveddelen av jobben. Hvis ikke, så vil det maks gi halv poengsum for deloppgaven.

Eksempel på kjøring: >>> tilfeldig_middag(matvarer, 70) ['laks', 'bønner'] >>> tilfeldig_middag(matvarer, 50) ['ris', 'bønner', 'kjøttdeig'] >>> tilfeldig_middag(matvarer, 20) ['ris']

```python
# ALTERNATIV 1, UTEN REKURSJON:
from random import randint

def tilfeldig_middag(matvarer, budsjett):
    tilfeldig_middag = []
    middagsvarer = []
    for vare, pris, rett in matvarer:
        if rett == 'middag' and pris<=budsjett:
            middagsvarer.append([vare,pris])
    while middagsvarer:
        if budsjett<=0:
            return tilfeldig_middag
        valg = randint(0,len(middagsvarer)-1)
        tilfeldig_middag.append(middagsvarer[valg][0])
        budsjett -= middagsvarer[valg][1]
        del middagsvarer[valg]
        middagsvarer = tilgjengelige_varer(middagsvarer, budsjett)
    return tilfeldig_middag

def tilgjengelige_varer(varer, budsjett):
    vareliste = []
    for vare, pris in varer:
        if pris <= budsjett:
            vareliste.append([vare,pris])
    return vareliste

# ALTERNATIV 2, MED REKURSJON:
from random import randint

def tilfeldig_middag_rekursjon(matvarer, budsjett):
    middagsvarer = [[vare, pris] for vare, pris, måltid in
matvarer if måltid == 'middag']
    return velge_varer(middagsvarer, budsjett)

def velge_varer(varer, budsjett):
    varer = [[vare, pris] for vare, pris in varer if pris <=
budsjett]
    if len(varer) == 0:
        return []
    valgt = varer.pop(randint(0,len(varer)-1))
    matvare = valgt[0]
    budsjett -= valgt[1]
    return [matvare] + velge_varer(varer, budsjett)
```