

# LØSNINGSFORSLAG

## Kontinuasjonseksamen i TDT4109 IT Grunnkurs

12 august 2022

### 1a)

Transistorer hadde en rekke fordeler fremfor releer og vakuumrør. De hadde ingen bevegelige deler og ble dermed ikke like lett ødelagt. De var mye mindre i størrelse og billigere i produksjon. De var også billigere i drift pga lavere strømforbruk og mindre varmeutvikling, og kunne utføre regneoperasjoner mye raskere enn de eldre teknologiene.

### 1b)

22 vil binært være representert som 10110, som er

$$1 * 2^{**4} + 0 * 2^{**3} + 1 * 2^{**2} + 1 * 2^{**1} + 0 * 2^{**0}, \text{ altså}$$

$$16 + 0 + 4 + 2 + 0$$

Vi trenger å representere både positive og negative heltall. Hvis man f.eks. bruker 8 bits (eksempel gitt i forelesning) kan man bruke den fremste biten til fortegn (1 for negativt, 0 for positivt). Da vil vi ha (uten bruk av toerkomplement):

$$+22 : 00010110$$

$$-22: 10010110$$

Denne representasjonen vil gjøre det mulig å representere tall fra og med -127 til og med +127. Dette vil sløse litt med plass, fordi tallet 0 blir representert på to måter (+0 og -0). Ideen med toerkomplement er å unngå denne sløsingene med plass, ved å la første bit indikere tallet -128 heller enn bare negativt fortegn, vi kan da representere tall fra og med -128 til og med +127.

Med toerkomplement blir da

$$+22: 00010110 \text{ (som før)}$$

$$-22: 11101010$$

Representasjonen for -22 fremkommer her ved at første bit er -128, som gjør at vi deretter trenger +106 for å ende opp med -22. 106 representeres binært som 1101010, dvs.

$$64 + 32 + 0 + 8 + 0 + 2 + 0$$

## 1c)

UNICODE var en forbedring fremfor ASCII og Extended ASCII ved at man kunne representere mange flere forskjellige tegn, og på en måte som fungerte på tvers av ulike land.

ASCII kunne bare representere 128 forskjellige tegn, og av bokstaver var bare de i det engelske alfabetet med.

Extended ASCII kunne støtte 256 forskjellige tegn, hvor de 128 fra ASCII var felles standard mens de øvrige kunne brukes til andre formål, og ble brukt til dels ulikt fra land til land. Det ble nå mulig å støtte norske bokstaver (i Norge) men overført til andre land ville slik tekst vises med andre symboler.

UNICODE løste dette ved å støtte mange flere tegn samtidig, så man kunne dekke alle bokstaver i alle språk med korrekt utveksling av tekstlige data mellom landene.

Feilene ved sammenligning av æ, ø, å i Python skyldes at sammenligningen med > eller < ikke skjer etter norske alfabetregler, men basert på tegnenes tallverdi. Både for de store og små variantene av disse bokstavene er det slik at Å, å ha lavere tallverdi enn Æ, æ og Ø, ø, f.eks. å: 229, æ: 230, ø: 248, og dette blir feil siden å kommer sist i det norske alfabetet.

## 1d)

(a)

Overklokking betyr at man kjører kode raskere enn det som prosessorens klokkefrekvens tilsier, dvs. utfører flere operasjoner per tidsenhet. Fordelen med dette er raskere responstid på beregningene, mens en ulempe er at det kan øke strømforbruket.

Underklokking er det motsatte, man kjører langsommere enn klokkefrekvensen tilsier. Dette sparer energi, men responstiden blir tregere.

(b)

I tilfellet med firmaet Raske Beregninger er det IKKE sikkert at overklokking vil gi noen særlig nytteverdi. Dette fordi oppgaveteksten sier at det er tregheter i nettverket knyttet til opplasting av data fra kundene som gjør at det går for langsomt, noe som ikke vil løses ved at beregningene i prosessoren går raskere. Det ville være mer fornuftig av firmaet Raske Beregninger å først se på muligheter for å øke kapasiteten i nettverket.

## 1e)

Den typen angrep som Raske Beregninger er utsatt for her kalles generelt Tjenestenektangrep (engelsk: Denial of Service), hvor angripere prøver å overbelaste et system med meningsløse forespørsler slik at det ikke rekker å gjøre nyttig arbeid for legitime brukere.

Mer spesifikt kalles den typen tjenestenektangrep som er beskrevet her "SYN flooding" (på lysark fra forelesning Nettverk del 4), hvor angriperen sender en mengde synkroniseringsforespørsler til firmaets tjenermaskin, men uten å kvittere for svarene på disse forespørslene. Firmaets tjenermaskin blir dermed stående og vente på slike kvitteringer og blir overbelastet med dette så den ikke klarer å betjene legitime kunder på en tilfredsstillende måte. Formålet med et slikt angrep er typisk å ødelegge for offeret (her: firmaet Raske Beregninger og deres kunder). Akkurat hva som skal være angriperens motivasjon er imidlertid umulig å si ut fra oppgaveteksten. Det kan være ren vandalisme uten annet mål eller mening enn bare å ødelegge, men det kan også være økonomiske eller politiske motiver.

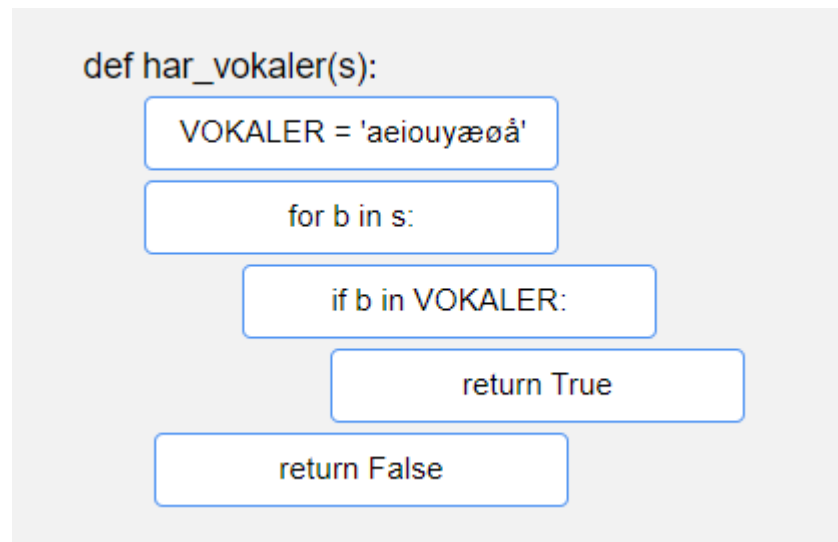
**2a)**

Riktig plassering av kodefragment:

```
def speeding_fine(f):  
    if fart > 130:  
        bot = 10000  
    elif fart > 110:  
        bot = 5000  
    elif fart > 100:  
        bot = 3000  
    else:  
        bot = 0  
    return bot
```

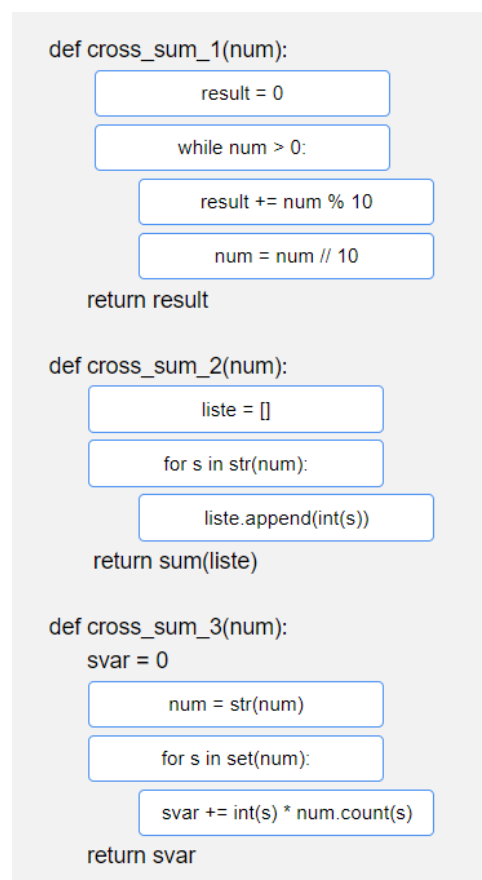
2b)

Riktig plassering av kodefragment:



2c)

Riktig plassering av kodefragment:



## 2d)

Riktig innfylling av tekst i feltene:

```
def count_same(num, i):  
    res = 0  
    n = num[i]  
    while i < len(num) and num[i] == n :  
        res += 1  
        i += 1  
    return res
```

I nest siste felt vil det også være korrekt å fylle inn `= res +`

I siste felt vil det også være korrekt å fylle inn `= i +`

## 2e)

Riktige valg av alternativer:

```
def remove_same_digits(num):  
    num = str(num)  
    i = 0  
    while i < len(num) :  
        if num[i] in '0123456789':  
            ct = count_same( num, i )  
            if ct > int(num[i]) :  
                num = num[:i] + num[i+ct:]  
            else:  
                i += ct  
        else:  
            i += 1  
    return int(num)
```

I det nest siste feltet, vil også alternativet `i += 1` fungere og gi uttelling som korrekt.

2f)

Riktige valg av alternativer:

```
def clean_file(filnavn):  
    with open (filnavn) as infile:  
        info = infile.read()  
        info = info.replace(',', " ").replace('%', "  
    liste = info.split('\n')  
    for i in range(len(liste)) :  
        liste[i] = liste[i].split()  
        liste[i][0:-2] = [' '.join(liste[i][0:-2])]  
        liste[i] = ';' .join(liste[i])  
    clean_info = '\n' .join(liste)  
    with open('clean_' + filnavn, 'w') as outfile:  
        outfile.write(clean_info)
```

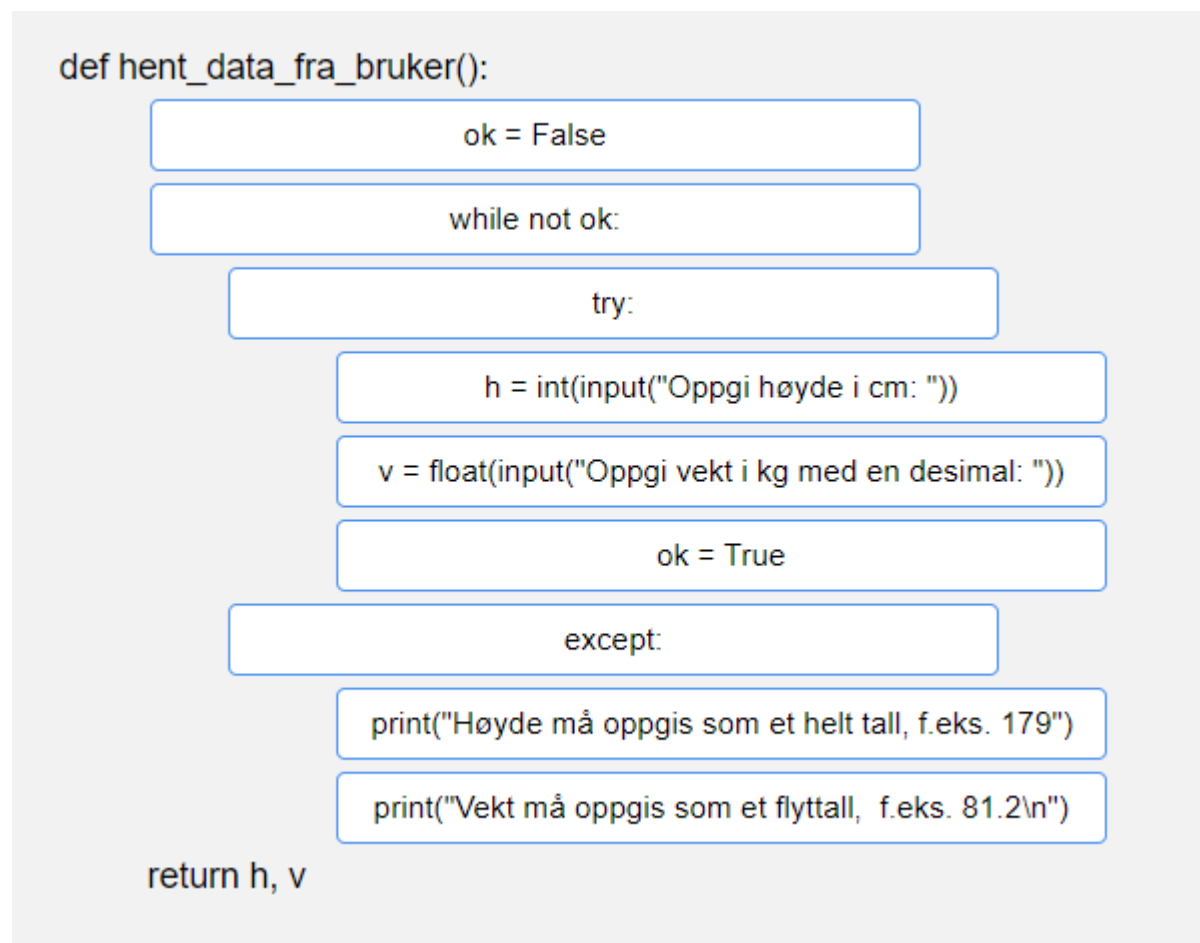
2g)

Riktige valg av alternativer:

```
def file_2_dictionary(filnavn):  
    fil = open (filnavn)  
    areal = {}  
    for line in fil :  
        data = line.split('; ')  
        areal[ data[0] ] = [ int(data[1]) , float(data[2]) ]  
    fil.close()  
    return areal
```

2h)

Riktige plassering av kodefragment:



3a)

En plausibel, rett fram løsning av denne funksjonen (andre løsninger kan også funke):

```
def sum_pos_odd(liste):
```

```
    summen = 0
```

```
    for num in liste:
```

```
        if num > 0 and num % 2:
```

```
            summen += num
```

```
    return summen
```

```
def sum_pos_odd(liste):  
    summen = 0  
    for num in liste:  
        if num > 0 and num % 2:  
            summen += num  
    return summen
```

### 3b)

En plausibel, rett fram løsning av denne funksjonen (andre løsninger kan også funke):

# LØSNING

```
def longest_streak(streng, resultat):
```

```
    streak = 0
```

```
    longest = 0
```

```
    for siffer in streng:
```

```
        if int(siffer) == resultat:
```

```
            streak += 1
```

```
            if streak > longest:
```

```
                longest = streak
```

```
        else:
```

```
            streak = 0
```

```
    return longest
```

```
def longest_streak(streng, resultat):
    streak = 0
    longest = 0
    for siffer in streng:
        if int(siffer) == resultat:
            streak += 1
            if streak > longest:
                longest = streak
        else:
            streak = 0
    return longest
```

# KODE FOR Å TESTKJØRE FUNKSJONEN

```
s = '000111101333313003311111'
```

```
winning_streak = longest_streak(s, 3)
```

```
drawing_streak = longest_streak(s, 1)
```

```
losing_streak = longest_streak(s, 0)
```

```
print(winning_streak, drawing_streak, losing_streak)
```



### 3c)

En plausibel, rett fram løsning av denne funksjonen (andre løsninger kan også funke):

# LØSNINGSFORSLAG

def co\_location(L):

    D = {}

    for element in L:

        pos = (element[2], element[3])

        if pos in D:

            D[pos].append(element[:2])

        else:

            D[pos] = [element[:2]]

    return D

```
def co_location(L):
    D = {}
    for element in L:
        pos = (element[2], element[3])
        if pos in D:
            D[pos].append(element[:2])
        else:
            D[pos] = [element[:2]]
    return D
```

# KODE FOR TESTKJØRING:

L = [ [ 'Anna', 12131415, 41.40309, 2.15203 ],

      [ 'Nils', 21222222, 63.40309, 5.19909 ],

      [ 'Don', 21558888, 63.40309, 5.19909 ],

      [ 'Lea', 77131415, 41.40309, 2.15203 ],

      [ 'Joe', 99131415, 25.22232, 19.33213 ] ]

print(co\_location(L))

### 3d)

En mulig løsning av denne oppgaven – delt opp i en main-funksjon pluss to andre funksjoner, i tillegg til at den kaller `co_location()` fra forrige deloppgave. Andre løsninger også mulig, inkl bare skript.

```
def find_close_coords(coord, D):
    result = []
    for elem in D.keys():
        if abs(elem[0]-coord[0]) <= 0.00003 and abs(elem[1]-coord[1]) <= 0.00003:
            result.append(elem)
    return result

def make_output(coords, D):
    if coords == []:
        txt = "No nearby persons"
    else:
        txt = 'Nearby persons:\n'
        for elem in coords:
            txt += f'{elem}: '
            for p in D[elem]:
                txt += f'{p[0]} ({p[1]}) ; '
            txt += '\n'
        return txt

def main():
    name = input('Name of missing person: ')
    latitude = float(input('Latitude of last observation: '))
    longitude = float(input('Longitude of last observation: '))
    D = co_location(L)
    coord = (latitude, longitude)
    c_list = find_close_coords(coord, D)
    message = make_output(c_list, D)
    print(message)

main()
```