Informasjon

Du måtte klare 10 av 17 oppgaver for å få øvingen godkjent.

Finner du feil, mangler eller forbedringer, ta gjerne kontakt!

Oppgave 2

«*Optimal delstruktur*» og «*Grådighetsegenskapen*» er riktig.

Kommentar:
For optimal delstruktur og grådighetsegenskapen, se læreboka s. 424-425.

Det at det kun er en optimal løsning på problemet trenger ikke å være tilfelle. For eksempel vil HUFFMAN av og til måtte velge mellom to noder med samme verdi, men algoritmen er optimal uansett, selv om man får en ulik optimal løsning alt etter som hvilken node man velger.

Man trenger heller ikke overlappende delinstanser. Det kan man for eksempel se ved at man aldri trenger løsningen på samme delproblem mer enn én gang i verken aktivitetsutvelgelsesproblemet eller når vi lager huffmantrær.

Oppgave 3

«*At vi kan sette sammen en globalt optimal løsning på problemet ved å gjøre lokalt optimale valg.*» er riktig.

Kommentar:
Dette er definisjonen på grådighetsegenskapen hentet fra læreboka s. 424.

Oppgave 4

«*Aktivitet-utvelgelse (The activity-selection problem)*» og «*Det kontinuerlige ryggsekkproblemet (The fractional knapsack problem)*» er riktig.

Kommentar:
Alle problemene gås igjennom i kapittel 15 og 16 i læreboken hvor boken sier at kun de to riktige svarene kan løses ved grådighet.

Oppgave 5

7 er riktig.

Kommentar:
Hvis vi sorterer aktivitetene etter sluttid får vi som tabell over aktivitetsnummer: $\langle 1, 2, 6, 7, 3, 8, 4, 5\rangle$

I første rekursjonskall må vi velge en aktivitet med starttid etter 0. Den første i tabellen vår som oppfyller dette kravet er aktivitet 1.

Siden aktivitet 1 slutter ved tidspunkt 3 må vi finne neste i tabellen som starter etter tidspunkt 3. Den første slike aktiviteten er aktivitet 7, som dermed blir den andre aktiviteten som blir valgt av RECURSIVE-ACTIVITY-SELECTOR.

Oppgave 6

3 er riktig.

Kommentar:
Hvis vi fortsetter der løsningen på forrige oppgave slapp, får vi at det neste funksjonskallet skal finne den neste aktiviteten i tabellen ved å velge den av aktivtetene som starter etter tidspunkt 7 og slutter tidligst. Den første slike aktiviteten er aktivitet 4.

Nå må vi finne neste aktivitet i tabellen som starter etter aktivitet 4 slutter. Aktivitet 4 slutter ved tidspunkt 12 og ingen av aktivitetene starter etter dette, så totalt tre aktiviteter ble valgt. Siden denne algoritmen er optimal, så må tre aktiviteter være det maksimale antallet det er mulig å velge.

Oppgave 7

«*Nei*» er riktig.

Kommentar:
Et enkelt moteksempel er om man har to aktiviteter: $(\text{Starttid} = 1, \text{Sluttid} = 4, \text{Verdi} = 3)$ og $(\text{Starttid} = 2, \text{Sluttid} = 5, \text{Verdi} = 5)$. Aktivitetene er overlappende, så det er ikke mulig å velge begge. RECURSIVE-ACTIVITY-SELECTOR vil velge den første aktiviteten, siden den slutter tidligst, men det optimale valget er å velge den andre aktiviteten.

Oppgave 8

Se kommentaren til forrige oppgave.

Oppgave 9

«*Nei*» er riktig svar.

Kommentar:
Et moteksempel er en situasjon med følgende aktiviteter:

1. (Tidsfrist $= 8$, lengde $= 3$)
2. (Tidsfrist $= 11$, lengde $= 8$)
3. (Tidsfrist $= 7$, lengde $= 2$)
4. (Tidsfrist $= 4$, lengde $= 3$)

Her vil algoritmen først velge aktivitet 2 og sette den inn på tidspunkt 3. Deretter vil algoritmen se på aktivitet 1. Det seneste den kan settes inn er på tidspunkt 0, så den settes inn der. Dermed er det ikke tid til noen flere aktiviteter. Algoritmen klarte her kun å få plass til to aktiviteter, men en optimal løsning er aktivitene $\langle 4, 3, 1 \rangle$. Ergo er ikke algoritmen garantert optimal.

Oppgave 10

Se kommentaren til forrige oppgave.

Oppgave 11

```
def encode(data, encoding):
    encoded = ""
    for character in data:
        encoded += encoding[character]
    return encoded
```

Kommentar:
Vi kan forenkle koden over ved å anvende «list comprehension».

```
def encode(data, encoding):
    return "".join(encoding[i] for i in data)
```

Oppgave 12

```
# Tretraversering med en stakk
def encoding(node):
    # Stakken består av tuppler av noder og deres tilhørende kode
    stack = [(node, "")]
    codes = {}
```

```
    # Fortsett helt til vi har utforsket alle nodene
    while stack:
        node, code = stack.pop()
        # Hvis en node har et tegn er den en løvnode og vi har funnet koden
        # til det gitte tegnet.
        if node.character is not None:
            codes[node.character] = code
            continue

        # Ellers legger vi til det venstre og høyre barnet med kode som
        # tilsvarer koden til den nåværende noden pluss respektivt 0 eller 1
        stack.append((node.left_child, code + "0"))
        stack.append((node.right_child, code + "1"))

    return codes
```

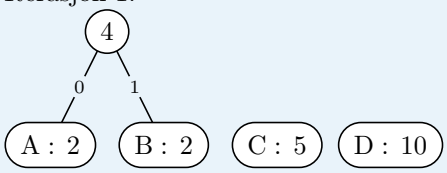
Oppgave 13

«*01*» er riktig.

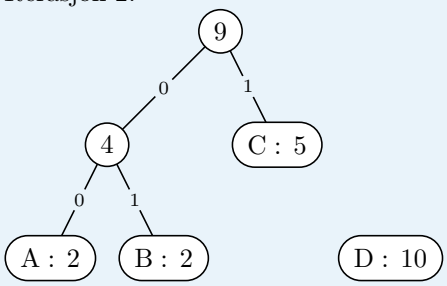
Kommentar:
Her er en visualisering av HUFFMAN. Merk at det er arbitrært om det er «A» eller «B» som velges først i første iterasjon av algoritmen. Her velger jeg at «A» velges først.

Iterasjon 0:

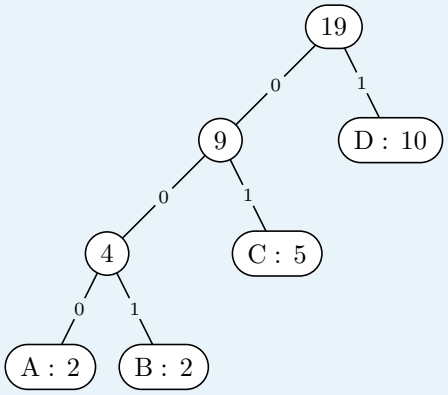
Iterasjon 1:



Iterasjon 2:



Iterasjon 3:



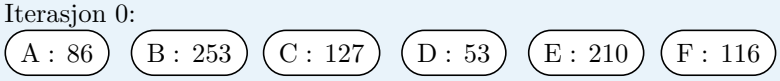
Ved å lese huffmankoden ut fra treet (strengen man får ved å traversere fra rota til noden med riktig tegn) ser vi at huffmankoden til «C» er 01.

Oppgave 14

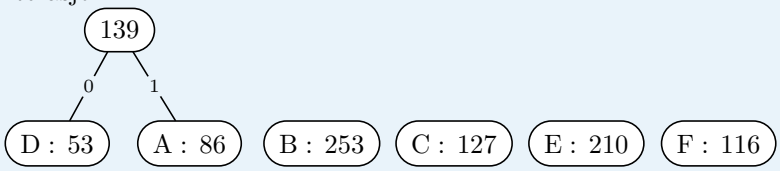
2072 er riktig.

Kommentar:
Her er en visualisering av HUFFMAN:

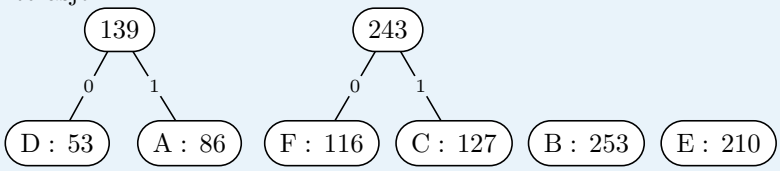
Iterasjon 0:



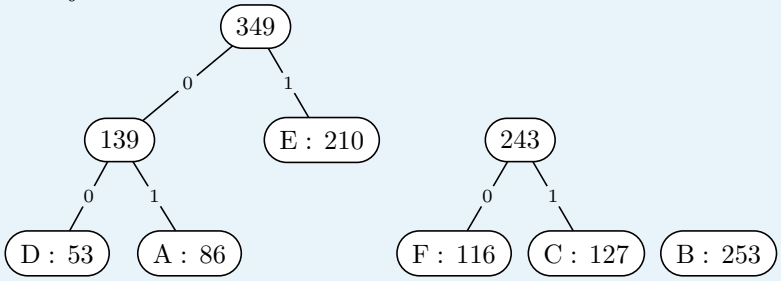
Iterasjon 1:



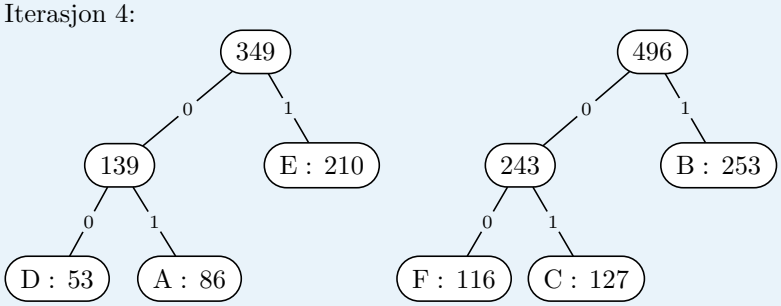
Iterasjon 2:



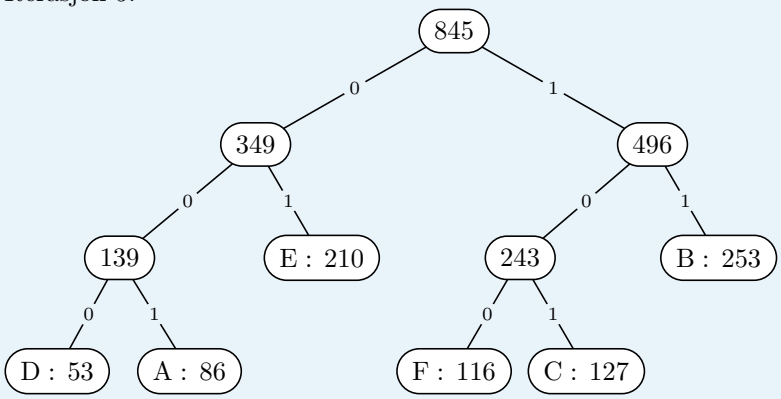
Iterasjon 3:



Iterasjon 4:



Iterasjon 5:



Her ser vi at «B» og «E» har en huffmankode med lengde 2, mens resten har lengde 3. Dermed blir lengden på den huffmankodede filen $2 \cdot (253 + 210) + 3 \cdot (86 + 127 + 53 + 116) = 2072$.

Oppgave 15

«*Et tegn vil aldri få en lengre kode enn et annet tegn med strengt lavere frekvens.*», «*Man er garantert at resultatet er en optimal prefikskode.*» og «*Hvis et tegn har strengt større frekvens enn de andre tegnene til sammen, vil dette tegnet få en kode med lengde* 1.» er riktig.

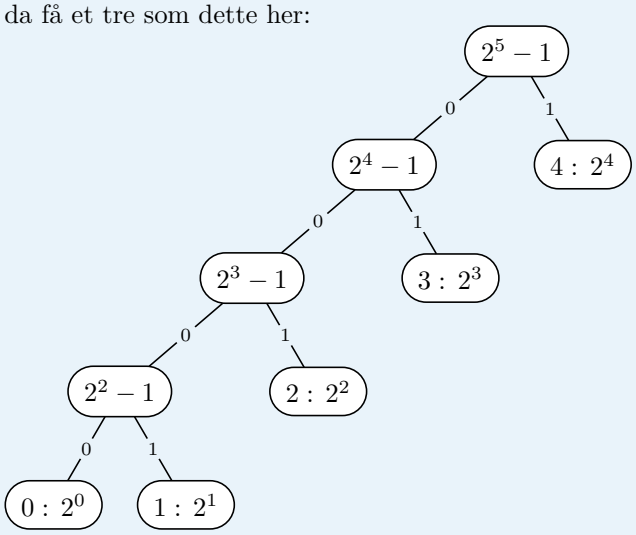
Kommentar:
Man er garantert at resultatet er en optimal prefikskode. Dette er teorem 16.4 i læreboka.

Et tegn vil aldri få en lengre kode enn et annet tegn med strengt lavere frekvens. Hvis ikke kunne tegnene bytte kode, og man ville dermed få en kode som gir en kortere koding, noe som bryter med at HUFFMAN gir en optimal prefikskode.

Hvis et tegn har strengt større frekvens enn de andre tegnene til sammen, vil dette tegnet få en kode med lengde 1. Si at HUFFMAN gir en prefikskode der et tegn med strengt større frekvens enn de andre tegnene til sammen lengde $k > 1$. Hvis du så bytter koden til dette tegnet med «1» og konkatinerer «0» foran koden til alle de andre tegnene, vil man få en koding som er kortere, siden lengden på koden til tegnet med størst frekvens minsker med minst én, mens koden til de andre tegnene øker med én, men disse har en samlet frekvens som er lavene enn tegnet med høyest frekvens.

Et huffmantre vil ikke alltid ha høyde $\Theta(\lg n)$ dersom det er n unike tegn i teksten. Et eksempel på at huffmantreet har høyde $\Theta(n)$ er om man har symboler $0 \leqslant i < n$ med frekvens 2^i. Man vil

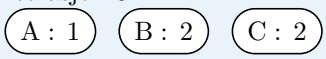
da få et tre som dette her:



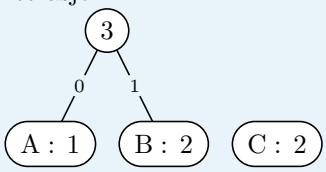
Et slikt tre vil ha høyde $n - 1 = \Theta(n)$.

To tegn med samme frekvens vil ikke nødvendigvis få like lang kode. Et eksempel på det er:

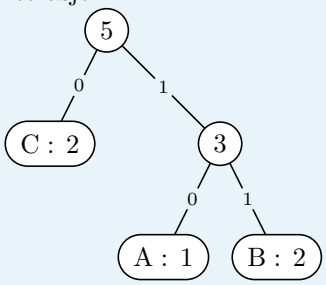
Iterasjon 0:



Iterasjon 1:



Iterasjon 2:



Her har «B» og «C» samme frekvens men ikke samme lengde på kode.

Oppgave 16

```python
from heapq import heappush, heappop


def build_decision_tree(decisions):
    # Trenger en liste til å representere en minheap med heapq-biblioteket
    heap = []
```

```
    # Bygger haugen og bruker en verdi 'c' til å skille mellom elementer
    # med lik sannsynlighet.
    c = 0
    for name, probability in decisions:
        heappush(heap, (probability, c, name))
        c += 1

    # Kombinerer de to minst sannsynlige tilfellene gjennom et ja-/nei-
    # spørsmål. Slik man bygger et Huffman tre.
    while len(heap) > 1:
        prob1, c, val1 = heappop(heap)
        prob2, _, val2 = heappop(heap)
        heappush(heap, (prob1 + prob2, c, (val1, val2)))

    # Her pakker vi ut treet som er laget i stakken.
    sol = {}
    stack = [("", heap.pop()[2])]
    while stack:
        code, val = stack.pop()
        # Hvis vi finner et tuppel så er dette det samme som et spørsmål
        # og vi legger til de to delene med en ekstra bit på slutten av
        # koden (f.eks. "101") deres.
        if isinstance(val, tuple):
            stack.append((code + "0", val[0]))
            stack.append((code + "1", val[1]))
        else:
            sol[val] = code

    return sol
```

Kommentar:
Grunnet antagelsen om at vi alltid kan konstruere spørsmål basert på de splittene vi gjør så blir dette problemet ekvivalent med å skulle lage et Huffman tre. Spesielt er del 2 med stakken her det samme som gjøres i oppgave 12.

Oppgave 17

«*Ja*» er riktig svar.

Kommentar:
Dette er fordi alle myntene har en verdi som er halvparten eller under av den neste mynten. Dette er et tilstrekkelig krav (men ikke et nødvendig krav). At kravet er tilstrekkelig kan vises ved at om den grådige algoritmen velger mynten med høyest verdi grådig. Om vi ikke hadde valgt denne mynten måtte den mynten erstattes med *minst* to mynter av lavere verdi. Dermed kan det ikke finnes løsninger som er bedre enn de den grådige algoritmen kommer med.

Oppgave 18

«*Nei*» er riktig svar.

Kommentar:
Et eksempel der den grådige algoritmen ikke er optimal, er 95, som har optimal løsning $50 + 45$, men den grådige algoritmen gir løsningen $60 + 30 + 1 + 1 + 1 + 1 + 1$.

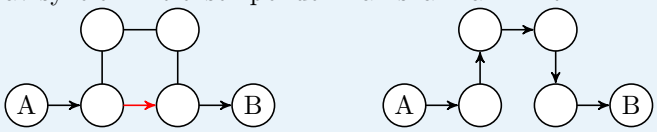
Oppgave 19

Her kan man sortere tabellen med kanter i stigende rekkefølge etter verdi. Man itererer så gjennom tabellen, og fjerner kanten man er på dersom den er en del av en sykel.

Kommentar:
For å vise at dette er korrekt er det først noen andre egenskaper det kan være lurt å vise.

Et sammenhengende komponent i en graf er en mengde med noder hvor det finnes en sti mellom alle nodene i mengden og det er ingen andre noder i grafen det er mulig å nå fra nodene i mengden.

Vi skal først vise at antallet sammenhengende komponenter ikke øker som følge av algoritmen. Dette er siden algoritmen kun fjerner kanter som er en del av sykler. Hvis det er en sti mellom alle noder før denne kanten fjernes, og denne stien bruker denne fjernede kanten, vil det fortsatt være en sti mellom nodene der man i stedet for å gå over denne fjernede kanten heller går over resten av sykelen. Et eksempel der man skal fra «A» til «B»:



Siden de samme nodene fortsatt kan nå hverandre etter algoritmen har fjernet en kant, vil algoritmen aldri dele en sammenhengende komponent i to. Dermed vil det være like mange slike komponenter når du starter algoritmen som det vil være når algoritmen er ferdig.

Vi skal nå se at antallet kanter som er igjen når algoritmen er ferdig, er likt antallet kanter som er i en optimal løsning. Hvor mange kanter er det i en optimal løsning?

En hvert sammenhengende komponent uten sykler er et tre, og alle trær med n noder har $n - 1$ kanter (mer om det i kapittel B.5 i læreboka). Dermed vil grafen ha $n - k$ kanter i algoritmens løsning, der n er antall noder og k er antallet sammenhengende komponenter i den opprinnelige grafen. Man kan se at dette vil være det antallet kanter algoritmen kommer til, siden den ikke endrer antallet sammenhengende komponenter, men samtidig sikrer at alle komponentene er asykliske. Man kan også se at dette er antallet kanter i optimal løsning, siden en optimal løsning vil også gjøre alle de sammenhengende komponentene asykliske, men vil heller ikke ha flere sammenhengende komponenter enn det vi startet med. Hvis en optimal løsning hadde delt et sammenhengende komponent i to, kunne vi lagt tilbake kanten som delte de i to og fortsatt hatt en asyklisk graf, men med høyere sum av kantverdier, siden alle kantverdier er positive.

Optimal løsning og løsningen algoritmen kommer med har like mange kanter, så om løsningen

algoritmen kommer med ikke er optimal betyr det at vi må bytte en eller flere kanter. La $A = \langle a_1, a_2, \ldots, a_\ell \rangle$ være kantene i algoritmens løsning som ikke er med i en optimal løsning. Ta en vilkårlig kant a_i og sett inn i optimal løsning. Etter drøftingen over, vil dette føre til en sykel (siden kanten ikke kan gå på tvers av to sammenhengende komponenter). a_i må være en kantene i sykelen med lavest verdi, for hvis det fantes en kant e i sykelen med lavere verdi ville vi oppnådd en bedre løsning enn den optimale løsningen ved å bytte e med a_i i den optimale løsningen, noe som gir en selvmotsigelse. a_i kan heller ikke være den eneste kanten i sykelen med lavest verdi, for da ville algoritmen ha fjernet den. Det må derfor funnes minst en kant b_i i sykelen med lik verdi som a_i, der b_i ble fjernet av algoritmen. Hvis vi bytter ut a_i med b_i i den optimale løsningen vil man derfor få en like god løsning, som dermed også er optimal. Ved å gjøre dette for alle kantene i A vil man kunne gå fra en optimal løsning til løsningen til algoritmen, uten at løsninen blir noe dårligere. Ergo er algoritmen optimal.

Denne algoritmen er en variant av Reverse-delete algorithm som igjen er en slags motsatt versjon av Kruskals algoritme, som er pensum i forelesning 9.

Oppgave 20

Man sorterer tabellen over landsbyer, og deretter setter ut brønnene grådig på *Første landsby som ikke er dekket av en brønn* $+ r$.

Kommentar:
Dette problemet kan minne litt om aktivitetsutvelgelsesproblemet. Den grådige egenskapen her er at hvis en optimal løsning har en brønn med en posisjon som er mindre enn *Avstand fra starten av veien til første landsby* $+ r$ kan man alltids sette denne på *Avstand fra starten av veien til første landsby* $+ r$, uten at det bryter med restriksjonen om at alle landsbyer skal ha en brønn i en avstand mindre eller lik r. Når denne brønnen er satt ut, kan man gjøre et tilsvarende argument for hvorfor det er minst like bra å sette en brønn på *Avstand fra starten av veien til første landsby som ikke er allerede dekket av en brønn* $+ r$. På den måten kan du grådig sette ut brønner og samtidig være sikker på at dette er en optimal løsning. Her domineres kjøretiden av sorteringen av landsbyene. Siden vi ikke vet noe om hvordan avstandene er fordelt, må vi bruke sammenligningsbasert sortering med kjøretid $\Theta(n \lg n)$ i verste tilfelle, noe som betyr at kjøretiden i verste tilfelle er $\Theta(n \lg n) = o(n^2)$.

Oppgave 21

Bruk algoritmen fra oppgaven over og binærsøk over d.

Kommentar:
Vi vet at d må være større eller lik 0, siden det er en avstand. Videre vet vi at dersom vi setter en brønn ved siste landsby vil alle landsbyene ha en avstand på maksimalt ℓ til nærmeste brønn, noe som vil si at $d \leqslant \ell$, siden vi har minst en brønn, noe som gir $0 \leqslant d \leqslant \ell$. I en optimal løsning må det finnes en brønn som ligger midt mellom to landsbyer og som har avstand d til begge. Hvis dette ikke var tilfelle, kunne vi flyttet brønnen enten tidligere eller senere og fått en bedre løsning. Siden landsbyene har en heltallig avstand til starten på veien, må et punkt midt mellom to landsbyer være på formen $t/2$ for en $t \in \mathbb{Z}$. Videre har vi:

$$0 \leqslant d = t/2 \leqslant \ell \implies 0 \leqslant t \leqslant 2\ell$$

Siden $t \in \mathbb{Z}$ finnes det $2\ell+1$ mulige verdier for t. Det å binærsøke over dette vil kreve $\Theta(\lg(2\ell+1)) = \Theta(\lg \ell)$ søk. Et søk vil si at vi kjører algoritmen fra oppgaven over, og sjekker om den verdien vi gjetter på for d tillater k brønner. Siden vi kun trenger å sortere tabellen med landsbyer én gang, blir kjøretiden $\Theta(n \lg n + n \lg \ell) = \Theta(n \lg(n\ell))$.

Oppgave 23

For eksamensoppgavene, se løsningsforslaget til den gitte eksamenen.

For oppgaver i CLRS, så finnes det mange ressurser på nettet som har fullstendige eller nesten fullstendige løsningsforslag på alle oppgavene i boken. Det er ikke garantert at disse er 100% korrekte, men de kan gi en god indikasjon på om du har fått til oppgaven. Det er selvfølgelig også mulig å spørre studassene om hjelp med disse oppgavene.

Et eksempel på et ganske greit løsningsforslag på CLRS, laget av en tidligere doktorgradsstudent ved Rutgers, finnes her.