Informasjon

Du måtte klare 8 av 16 oppgaver for å få øvingen godkjent.

Finner du feil eller mangler eller har forslag til forbedringer, ta gjerne kontakt!

Oppgave 2

«*Det må finnes overlappende delproblemer*» og «*Problemet må ha optimal delstruktur*» er riktig.

Kommentar:
Dette er de to kravene man normalt stiller for at det skal gi mening å anvende dynamisk programmering. Hvis problemet ikke har optimal delstruktur, så gir det ikke mening å finne optimale løsninger på delproblemene da disse ikke kan brukes i løsningen til problemet. På samme måte gir det ikke mening å anvende dynamisk programmering hvis det ikke finnes overlappende delproblemer. Uten overlappende delproblemer har man ikke bruk for å ta vare på løsningene på delproblemene etter man har regnet de ut, da disse ikke vil brukes videre.

Svaret «*Problemet må kun ha én riktig løsning*» kan vi ganske enkelt se at er galt, da stavkuttingsproblemet ofte kan ha flere mulige løsninger som alle gir samme høyeste profitt. «*Input må kun bestå av heltall*» stemmer heller ikke, da hvordan input ser ut ikke har noe å si så lenge det finnes en naturlig oppdeling av problemet i delproblemer. Dette kan for eksempel ses i korteste vei eksempelet som brukes i læreboken, her er input en graf og ikke heltall.

Oppgave 3

«*En optimal løsning på problemet kan konstrueres fra optimale løsninger på delinstansene.*» er riktig.

Kommentar:
Se side 379 i læreboken (overskriften «Optimal substructure») for en definisjon.

Oppgave 4

«*Flere oppdelinger av problemet inneholder noen av de samme delproblemene*» er riktig.

Kommentar:
Se side 384 i læreboken (overskriften «Overlapping subproblems») for en definisjon.

Oppgave 5

13 er riktig.

Kommentar:
Den beste oppdelingen får man når man deler staven i deler med lengde $2, 2, 2$ og 1, noe som gir en pris på $4 + 4 + 4 + 1 = 13$. Den nest beste oppdelingen får man når man deler staven i deler med lengde 5 og 2, noe som gir en pris på $8 + 4 = 12$.

Denne instansen av problemet er så liten at man ganske greit kan lete seg frem til beste løsning. Hadde man hatt en mye større n hadde den letteste strategien vært å for hver av de mulig lengdene, $i \leqslant n$ regne ut hvor mye den beste oppdelingen hadde gitt. Da kan man bruke svarene fra de lengdene som er mindre enn den man for øyeblikket ser på. Egentlig akkurat slik som man ville gjort i et iterativt dynamisk program. Da hadde man fått at

$$
\begin{aligned}
r_1 &= \max(p_1) = 1 \\
r_2 &= \max(r_1 + r_1, p_2) = 4 \\
r_3 &= \max(r_1 + r_2, p_3) = 5 \\
r_4 &= \max(r_1 + r_3, r_2 + r_2, p_4) = 8 \\
r_5 &= \max(r_1 + r_4, r_2 + r_3, p_5) = 9, \\
r_6 &= \max(r_1 + r_5, r_2 + r_4, r_3 + r_3, p_6) = 12 \\
r_7 &= \max(r_1 + r_6, r_2 + r_5, r_3 + r_4, p_7) = 13
\end{aligned}
$$

Oppgave 6

«8 staver av lengde 1», *«2 staver av lengde 2 og 1 stav av lengde 4»*, *«2 staver av lengde 3 og 1 stav av lengde 2»* og *«4 staver av lengde 2»* er riktig.

Kommentar:
Her vet vi at oppdelingen av en stav på lengde 6 i en som har lengde 2 og en som har lengde 4 er en unik løsning. Det vil si at det ikke finnes noen andre oppdelinger av stangen på lengde 6 som har samme eller høyere verdi. Dette betyr blant annet at følgende ulikheter gjelder (ikke en uttømmende liste):

$$
\begin{aligned}
p_2 + p_4 &> 6 \cdot p_1 \\
p_2 + p_4 &> 2 \cdot p_3 \\
p_4 &> 2 \cdot p_2 \quad \textit{siden } p_2 + p_4 > 3 \cdot p_2
\end{aligned}
$$

Dette betyr at alle alternativ som inneholder minst 6 staver av lengde 1, 2 av lengde 3 eller 2 av lengde 2 ikke kan være gyldige løsninger for $n = 8$. I alle slike situasjoner kan man erstatte disse med en stav av lengde 2 og en av lengde 4 og få en større profitt.

For oppdelingen i en stav på lengde 3 og en stav på lengde 5 er det derimot ikke så lett å si noe om dette er mulig. Ved å sette opp formler kan man se at dette bør kunne være en mulig optimal

løsning i noen situasjoner. For eksempel hvis $p = \langle 1, 6, 9, 13, 17, 1, 1, 1 \rangle$ er denne oppdelingen en av flere løsninger med verdi 26, og det finnes her ingen løsninger med verdi over 26.

Oppgave 7

```
def sheet_cutting(w, h, p):
    # Løser problemet med hjelp av iterasjon.
    for i in range(1, w + 1):
        for j in range(1, h + 1):
            # For hver størrelse prøver vi først å dele horisontalt,
            # for så å dele vertikalt. Den deling som gir to
            # subinstanser med høyest samlet verdi blir verdien til
            # denne størrelsen.
            for k in range(1, j // 2 + 1):
                p[i, j] = max(p[i, j - k] + p[i, k], p[i, j])
            for k in range(1, i // 2 + 1):
                p[i, j] = max(p[i - k, j] + p[k, j], p[i, j])
    return p[w, h]
```

Oppgave 8

«*Man kan potensielt ende opp med å løse færre delinstanser.*» er riktig.

Kommentar:
Når man løser et problem med memoisering løser man kun de delinstansene man trenger for den spesifikke instansen av problemet, så man kan slippe å måtte løse alle delinstansene slik man ender opp med å gjøre ved iterasjon. Man ender derimot opp med mer overhead grunnet de rekursive kallene. Man løser alle delinstanser maksimalt en gang ved både iterasjon og memoisering, så dette kan ikke være et riktig alternativ. Det er heller ikke garantert at man får en bedre kjøretid ved memoisering. For eksempel vil man få nøyaktig samme tidskompleksitet ved memoisering og iterasjon for stavkuttingsproblemet.

Oppgave 9

«*Man har generelt mindre overhead.*» er riktig. Se kommentaren til oppgave 8 for en forklaring.

Oppgave 10

«*Nei, fordi dekomponeringen ikke har overlappende delproblemer.*» er riktig.

Kommentar:
Delproblemene i dekomponeringen i quicksort vil være å sortere elementene til venstre for og til høyre for pivoten hver for seg. På grunn av dette kan man ikke ende opp overlapp mellom delproblemene.

Oppgave 11

3 er riktig.

Kommentar:
Dette kan man verifisere for hånd. Man kan komme seg til $(2, 3)$ på følgende måter:

1. *ned, ned, høyre*
2. *ned, høyre, ned*
3. *høyre, ned, ned*

Oppgave 12

210 er riktig.

Kommentar:
Det tar altfor lang tid å telle seg frem til hver løsning, så her må man finne en formel for å regne dette ut. Vi kan se at det er to muligheter for å komme seg til rute (x, y). Enten må man gå igjennom rute $(x-1, y)$ eller rute $(x, y-1)$, men man kan ikke gå igjennom begge på samme sti. Fra begge av disse er det kun en mulig måte å komme seg til (x, y), det vil si for $(x-1, y)$ må man hoppe en kolonne mot høyre og for $(x, y-1)$ må man hoppe en rad nedover. Siden alle stier som går igjennom $(x-1, y)$ dermed vil være forskjellige fra alle stier som går igjennom $(x, y-1)$ er antall stier som går til (x, y) lik summen av antall stier som går til $(x-1, y)$ og antall stier som går til $(x, y-1)$. Vi har derfor at $f(x, y) = f(x-1, y) + f(x, y-1)$ når $x > 1$ og $y > 1$. Når $x = 1$ eller $y = 1$ så finnes det kun en sti til ruten og dermed har man $f(x, y) = 1$.

Med formelen over kan man iterativt bevege seg bortover rutenettet og regne ut hver $f(x, y)$ ved hjelp av de allerede utregnete verdiene for $f(x', y')$. Dette blir da en iterativ dynamisk programmeringsløsning. Man kunne selvfølgelig startet med $f(x, y)$ og regnet denne ut ved rekursjon i stedet for, altså en løsning ved memoisering.

Hvis man fyller ut rutenettet med verdiene for $f(x, y)$, og roterer 45 grader med klokken ender man interessant nok opp med Pascals trekant. Dette viser at $f(x, y)$ er lik bionomial koeffisienten.

Oppgave 13

```
def longest_decreasing_subsequence(s):
    # Liste til å ta vare på løsninger på delinstanser.
    subproblems = []

    # Løs først problemet gradvis for listen, fra start til slutt.
    for index, value in enumerate(s):
        # Lagrer løsningene som tuppelet
        # (lengde, forrige indeks, verdi)
        optimal = (1, index, value)
        for i in range(index):
            if s[i] > value and subproblems[i][0] + 1 > optimal[0]:
                optimal = (subproblems[i][0] + 1, i, value)
        subproblems.append(optimal)

    # Finn den løsningen med størst lengde og lag en liste bestående
    # av elementene som finnes i denne løsningen ved å følge
    # referansene til forrige element.
    solution = max(subproblems, key=lambda subproblem: subproblem[0])
    sequence = [solution[2]]
    while solution[0] > 1:
        solution = subproblems[solution[1]]
        sequence.insert(0, solution[2])
    return sequence
```

Oppgave 14

«*Ja*» er riktig.

Kommentar:
Dette problemet kan fint løses ved å lage en ny tabell T av samme lengde som A. Deretter, for hvert tall i A, summerer man tallet og alle tall som kommer før det i A inntil man enten kommer helt til starten av A eller har en sum som finnes i B og T er satt til TRUE for den indeksen man er kommet til. I det første tilfellet setter man $T[i]$ = FALSE ellers setter man $T[i]$ = TRUE. Hvis man har $T[n]$ == TRUE finnes det en eller flere slik oppdelinger, ellers finnes det ingen. Denne taktikken går ut på at T indikerer om det er mulig å dele opp de i første tallene i A på en slik måte at man løser problemet. Så $T[n]$ indikerer da at det er mulig å dele opp hele A på en slik måte.

Denne oppgaven er inspirert av oppgave o) på eksamen høsten 2012, hvor man skulle finne en løsning på dette problemet. I løsningsforslaget til eksamenen finnes det pseudokode for algoritmen som er forklart over.

Oppgave 15

«*Nei, denne oppdelingen i delinstanser har ikke optimal delstruktur.*» er riktig.

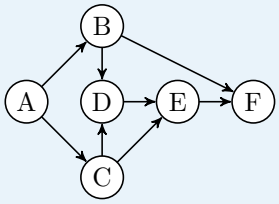
Kommentar:
Dette er grunnet at man kun kan bruke hver veistrekning en gang. Dermed vil det ikke nødvendigvis være mulig å anvende lengste vei mellom A og C til å komme seg til B, fordi man kan trenger minst en av veistrekningene i denne lengste veien mellom A og C for å komme seg fra C til B. Det står også en del om dette på side 382 i læreboken.

Oppgave 16

«*Ja*» er riktig.

Kommentar:
Siden alle veiene er enveiskjørte og det ikke er mulig komme seg tilbake til der man startet uten å kjøre feil vei på en veistrekning har man optimal delstruktur. Dette fordi hvis den lengste veien fra C til B overlapper med den lengste veien fra A til C, så betyr det det er mulig å gå en runde hvor man starter i C og slutter i C. Dette da, man for en posisjon D i den lengste veien fra A til C kan komme seg til C, men man kan da også komme seg fra C til D. Dette strider i mot problemspesifikasjonen og dermed kan ikke disse to veiene overlapper. Det vil si man har optimal delstruktur.

Når det kommer til overlappende delproblemer, så er dette illustrert i figuren under. Denne grafen representerer et veinettverk, hvor vi ønsker å finne lengste vei mellom A og F. Der kan vi se at både når vi utforsker A $\rightarrow$ B $\rightarrow$ D og A $\rightarrow$ C $\rightarrow$ D trenger vi å vite hva som er lengste vei fra D til F. Altså det finnes overlappende delinstanser i denne instansen av problemet. For større, mer innviklete instanser av problemet vil antall overlappende delinstanser eksplodere.



Utfordring: Design en algoritme som ved dynamisk programmering løser problemet diskutert i denne oppgaven.

Oppgave 17

Det kontinuerlige ryggsekkproblemet kan løses ved å første sortere alle gjenstandene basert på v_i/w_i. Altså man sorterer gjenstandene basert på verdi per vektenhet. Deretter plukker man den gjenstående gjenstanden med høyest verdi per vektenhet helt til man ikke lengre har plass til hele denne gjenstanden. Da tar man med seg så mye man har plass til av denne gjenstanden.

Løsningen beskrevet over er optimal fordi den fyller ryggsekken helt, samtidig som den har med seg mest mulig av gjenstandene som gir mest verdi for vekten deres.

Oppgave 18

Det ubegrensede ryggsekkproblemet kan løses på lignende måte som det binære ryggsekkproblemet. Faktisk kan nesten samme pseudokode anvendes. Pseudokoden nedenfor løser det ubegrensede ryggsekkproblemet, og er nesten lik den brukt i forelesningen for det binære ryggsekkproblemet. Den eneste forskjellen er definisjonen av y. I det binære ryggsekkproblemet så man på $\mathrm{K}[i-1, j-w_i]$, fordi man kun kunne ha en av hver gjenstand. Derfor måtte man se på verdien ved vekten $j - w_i$ etter å ha bestemt seg for om man vil ha med gjenstand $i - 1$. Altså, man kunne ikke se på noen av resultatene i rad i, der man allerede kunne ha valgt å plukke med seg gjenstand i. Nå derimot, må man også se på $\mathrm{K}[i, j - w_i]$ fordi det er lov å ha med samme gjenstand flere ganger. Merk at vi ikke lenger ser på $\mathrm{K}[i - 1, j - w_i]$ fordi vi er garantert at $\mathrm{K}[i, j - w_i] \geqslant \mathrm{K}[i - 1, j - w_i]$.

```
KNAPSACK(n, W)
 1  let K[0..n, 0..W] be a new array
 2  for j = 0 to W
 3      K[0, j] = 0
 4  for i = 1 to n
 5      for j = 0 to W
 6          x = K[i − 1, j]
 7          if j < w_i
 8              K[i, j] = x
 9          else y = K[i, j − w_i] + v_i
10              K[i, j] = max(x, y)
```

Oppgave 19

Dette er ikke mulig, og kan vises ved å anta at det er mulig. Det vil si anta at det finnes en algoritme $\mathrm{A}(n, \mathrm{W}, k)$ som løser det begrensede ryggsekkproblemet og som har bedre kjøretid enn noen av de mulige algoritmene for det binære ryggsekkproblemet. Men, da kan vi designe en algoritme B for det binære ryggsekkproblemet på følgende måte:

```
B(n, W)
1  return A(n, W, 1)
```

Denne algoritmen må ha samme kjøretid som A, siden det eneste som gjøres i B, men ikke i A er å kalle A noe som har tidskompleksitet $\mathrm{O}(1)$. Dermed har vi en motsigelse, og det er derfor ikke mulig at en slik algoritmen A eksisterer.

Det kan fremstå som om dette er en litt arbitrær ting å skulle vise. Hvorfor bryr vi oss egentlig om at det ikke kan finnes en slik algoritme? Poenget her er at vi har vist at det begrensede ryggsekkproblemet er minst like vanskelig som det binære ryggsekkproblemet. Dette er i seg selv nyttig, da alle nedre grenser vi kan vise for det binære ryggsekkproblemet gjelder for det begrensede ryggsekkproblemet. Spesielt er konseptet med slike *reduksjoner* mellom algoritmer ofte nødvendig når man snakker om klassifikasjoner av algoritmer. For eksempel så er man avhengig av slike reduksjoner når man viser NP-kompletthet, noe vi vil se i forelesning 13.

Oppgave 20

```
def find_path(weights):
    # Lagrer løsningene til delinstansene i en 2D-liste, og populerer
    # denne med løsningen på rad 1. Løsningene er på formatet
    # (total vekt, x-koordinat til piksel i forrige rad)
    subproblems = [[(weight, -1) for weight in weights[0]]]
    for row in range(1, len(weights)):
        row_results = []
        for column, value in enumerate(weights[row]):
            # Starter med uendelig vekt, slik at alle senere
            # alternativer vil garantert være bedre.
            best = (float("inf"), -1)
            # Ser kun på elementene med y-koordinat i området
            # [column - 1, column + 1], men må avgrense ved endene til
            # bildet.
            start = max(0, column - 1)
            end = min(column + 2, len(weights[row]))
            for index in range(start, end):
                element = subproblems[-1][index]
                if element[0] + value < best[0]:
                    best = (element[0] + value, index)
            row_results.append(best)
        subproblems.append(row_results)

    # Finner den beste løsningen fra den siste raden i bildet.
    # optimal er på formatet ((løsning på delinstansen), indeks)
    optimal = (float("inf"), -1), -1
    for index, element in enumerate(subproblems[-1]):
        if element[0] < optimal[0][0]:
            optimal = element, index

    # Bygger stien ved å følge referansene til x-koordinatet i
    # løsningene på delinstansene.
    path = []
    for y in range(len(weights) - 1, -1, -1):
        path.insert(0, (optimal[1], y))
        if y > 0:
            optimal = subproblems[y - 1][optimal[0][1]], optimal[0][1]

    return path
```

Oppgave 22

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.