Institut for Matematik og Datalogi
Syddansk Universitet, Odense

18. april 2020

DM507 — Algoritmer og Datastrukturer

Eksaminatorie-timer uge 16, Forår 2020

Instruktorerne for DM507

Indhold

1 Eksaminatorie-timer I

1.1 Opgave 1 - Cormen et al. opgave 2.3

I følgende opgave er vi interesserede i at undersøge korrektheden af Horner's rule, som kan anvendes til at evaluere et n'te grads polynomium, givet koefficienter $a_0, \ldots, a_n$ og en værdi x:

$$P(x) = \sum_{k=0}^{n} a_k x^k \tag{1}$$
$$= a_0 + \ldots + a_{n-1}x^{n-1} + a_n x^n \tag{2}$$
$$= a_0 + x \cdot (a_1 + x \cdot (a_2 + \ldots + x \cdot (a_{n-1} + x \cdot a_n) \ldots)) \tag{3}$$

I Eqn. 2 evalueres polynomiet $P(x)$ direkte ved at regne rækken, mens en omskrivning giver Horner's rule som anvendes i Eqn. 3. Korresponderende pseudokode som evaluerer udtrykket i Eqn. 3 er givet nedenfor:

```
1  # Lad x være den værdi polynomiet evalueres ved
2  # Lad n være den graden af polynomiet der evalueres
3  # Lad A være en liste af koefficienter
4  HornersRule(x, n, A):
5      y = 0
6      for i = n downto 0:
7          y = A[i] + x * y
8      return y
```

1.1.1 Delopgave a

I denne delopgave angiver vi køretiden i Θ-notation for ovenstående pseudokode som implementerer Horner's rule. Ved en inspektion af for-løkken ser vi at køretiden klart er $\Theta(n)$.

1.1.2 Delopgave b

I denne delopgave sammenligner vi Horner's rule i Eqn. 3 med en *naiv algoritme* der evaluerer polynomiet direkte ved at regne rækken i Eqn. 2.

Den naive algoritme der bruger Eqn. 2 til at evaluere polynomiet $P(x)$ er givet i form af nedenstående pseudokode:

```
1  # Lad x være den værdi polynomiet evalueres ved
2  # Lad n være den graden af polynomiet der evalueres
3  # Lad A være en liste af koefficienter
4  NaiveMethod(x, n, A):
5      y = 0
6      for i = 0 to n:
7          z = 1
8          for j = 1 to i: # Regn potens af x
9              z = z * x
10         y = y + A[i] * z   # Gang på korresponderende koefficient
11     return y
```

Vi ser at denne algoritme har en køretid på $\Theta(n^2)$. Grunden til at dette er tilfældet er at hver potens af x skal regnes, hvilket samlet nødvendigvis tager $1 + 2 + \ldots + n = \Theta(n^2)$ tid, da vi skal beregne $x,\ x \cdot x = x^2,\ x \cdot x \cdot x = x^3, \ldots$ osv.

1.1.3 Delopgave c & d

I denne delopgave beviser vi korrektheden af invarianten for for-løkken i pseudokoden, der implementerer Horner's rule:

Invarianten: I *starten* af hvert gennemløb er

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k \tag{4}$$

Initialization (basistilfælde): Før første gennemløb er $i = n$, hvilket medfører at $y = 0$ og invarianten holder trivielt. Vi ser dette ved at indsætte $i = n$ i Eqn. 4, således at rækken kommer til at gå fra $k = 0$ til $k = n - (n + 1) = -1$. Rækken er tom og derfor har vi at højresiden af 4 er nul. Da y er sat lig nul før første gennemløb, er 4 opfyldt.

Maintenance (induktionsskridt): Antag at invarianten i Eqn. 4 for y holder ved starten af det i'te gennemløb. Efter det i'te gennemløb af for-løkken, med $n > i \geq 0$, lader vi i' og y' betegne de nye værdier (efter det i'te gennemløb) af i og y. Vi ser i koden at i for-løkken er $i' = i - 1$ og at y' er givet ved følgende:

$$\begin{aligned}
y' &= a_i + x \cdot y && && (5)\\
&= a_i + x \cdot \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k && \text{(udtryk for } y \text{ substitueres ind)} && (6)\\
&= a_i + \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k+1} && (x \text{ ganges ind i rækken}) && (7)\\
&= a_i x^0 + \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k+1} && \text{(introducer } x^0 = 1 \text{ i første term)} && (8)\\
&= \sum_{k=-1}^{n-(i+1)} a_{k+i+1} x^{k+1} && (a_i x^0 \text{ absorberes i rækken}) && (9)\\
&= \sum_{k=0}^{n-(i+1)+1} a_{k+i+1-1} x^{k+1-1} && \text{(juster indeks } k \text{ med + 1)} && (10)\\
&= \sum_{k=0}^{n-(i+1)+1} a_{k+i} x^k && \text{(reducer udtrykket ift. indeks)} && (11)\\
&= \sum_{k=0}^{n-(i'+1)} a_{k+i'+1} x^k && \text{(introducer } i' = i - 1 \Leftrightarrow i = i' + 1) && (12)
\end{aligned}$$

Vi ser i denne sammenhæng at invarianten også holder for starten af næste gennemløb af for-løkken, dvs. for $i' = i - 1$.

Termination: Algoritmen terminerer når $i = -1$ vi har derfor, ved brug af invarianten, følgende:

$$y = \sum_{k=0}^{n-(-1+1)} a_{k-1+1}x^k \tag{13}$$

$$= \sum_{k=0}^{n} a_k x^k \tag{14}$$

Hvilket præcist er formlen for et n'te grads polynomium $P(x)$ og udtrykket i Eqn. 2.

Det følger herefter fra beviset af invarianten, at algoritmen der implementerer Horner's rule er korrekt og evaluerer et n'te grads polynomium $P(x)$ i overensstemmelse med formlen for et n'te grads polynomium.

1.2 Opgave 2 - Cormen et al. opgave 4.2-7

Vis hvordan man kan gange to komplekse tal $a + bi$ og $c + di$ ved kun tre multiplikationer af reelle tal. Algoritmen skal tage a, b, c og d som input og skabe den reelle del $ac - bd$ og den imaginære del $ad + bc$.

Opgaven går ud på at skabe

1. $ac - bd$
2. $ad + bc$

fra a, b, c, d ved brug af 3 multiplikationer. Hvis man udregner 1 og 2, som de er, så giver det os 4 multiplikationer ac, bd, ad og bc.

Hvis vi derimod lægger tal sammen, før vi ganger dem, så kan vi få "lavet"flere multiplikationer med kun en multiplikation.

$$(x + y) \cdot z = xz + yz$$

Her har vi via addition og en multiplikation skabt to multiplikationer og dermed "sparer"vi en multiplikation. Hvis vi prøver os lidt frem:

$$(a + b) \cdot c = ac + bc$$

Den giver dog kun ac fra 1 og bc fra 2. Hvis vi prøver at lægge d sammen med c:

$$(a + b) \cdot (c + d) = ac + ad + bc + bd$$

Så kan vi se, at vi nu har alle ledene ac, bd, ad og bc. Herfra kan vi prøve at få 1 eller 2 til at stå alene.

Hvis der kigges på 2, så er den inkluderet i $(a + b) \cdot (c + d)$. Dog skal ac og bd trækkes fra for at få $ad + bc$ til at stå alene.

$$(a + b) \cdot (c + d) - ac - bd = ac + ad + bc + bd - ac - bd = ad + bc$$

Men ac og bd er ledene i 1. Altså hvis vi udregner $a \cdot c$, $b \cdot d$ og $(a + b) \cdot (c + d)$, så kan 1 og 2 udregnes.

Hvis det opsættes ligesom Strassen's algoritme:

$$
\begin{aligned}
S_1 &= a + b \\
S_2 &= c + d \\
P_1 &= S_1 \cdot S_2 \\
P_2 &= a \cdot c \\
P_3 &= b \cdot d \\
ac - bd &= P_2 - P_3 \\
ad + bc &= P_1 - P_2 - P_3
\end{aligned}
$$

Dermed har vi udregnet $ac - bd$ og $ad + bc$ via 3 multiplikationer P_1, P_2 og P_3. Ethvert produkt af to komplekse tal kan nu udregnes således:

$$(a + bi) \cdot (c + di) = (ac - bd) + (ad + bc) \cdot i = (P_2 - P_3) + (P_1 - P_2 - P_3) \cdot i$$

1.3 Opgave 3 - Cormen et al. opgave 15.1-3

I denne opgave skal vi lave en algoritme for en modificeret version af rod-cutting problemet. I den modificerede version koster det et konstant beløb c at skære vores stang (engelsk: rod) over. Lad os tage udgangspunkt i algoritmen `Bottom-Up-Cut-Rod` på side 366 i [1] og ændre den til at tage højde for at det koster c hver gang man skærer stangen over.

`Bottom-Up-Cut-Rod` virker overordnet set ved at man bygger et array `r` op fra bunden sådan at `r[j]` indeholder den optimale pris for en stang af længde `j`. Altså starter man fra `j` $= 1$ og slutter når `j` $= n$, hvor n er længden på vores stang. Koden i for-løkken på linje 5-7 er en direkte implementering af ligning 15.2 på side 362 i [1]. Det vigtige ved denne ligning er at man kun skal tage højde for et nyt cut, hvis man antager at der for r_{n-i} (hvor $i = 1, 2, ..., j - 1$) allerede er taget højde for det antal gange stangen blev skåret over. Det vil sige at linje 6 skal laves om fra

```
q = max(q, p[i] + r[j-i])
```

til

```
q = max(q, p[i] - c + r[j-i])
```

Dette holder dog ikke for en stang som ikke skal skæres over, da man her ikke skal trække c fra. For at løse det kan vi lade den inderste for-løkke stoppe ved `j-1` og lade `r[j] = max(q, p[j])` efterfølgende, hvilket gør det samme som koden ville have gjort originalt ved $i = j$. Alternativt kan man også starte med at sætte `q = p[j]` og så se om en opdeling af stangen resulterer i en bedre pris. Den alternative version resulterer i følgende kode:

```
1  Modified-Bottom-Up-Cut-Rod(p,n):
2      Let r[0..n] be a new array
3      r[0] = 0
4      for j = 1 to n:
5          q = p[j]
6          for i = 1 to j-1:
7              q = max(q, p[i] - c + r[j-i])
8          r[j] = q
9      return r[n]
```

1.4 Opgave 4 - Cormen et al. opgave 15.4-1

Bestem en *længste fælles delsekvens* (LCS) af $\langle 1,0,0,1,0,1,0,1\rangle$ og $\langle 0,1,0,1,1,0,1,1,0\rangle$.

Hertil kan den rekursive formel

$$\text{lcs}(i,j) = \begin{cases} 0 & \text{hvis } i = 0 \vee j = 0 \\ \text{lcs}(i-1,j-1)+1 & \text{hvis } i,j > 0 \wedge x_i = y_j \\ \max\{\text{lcs}(i-1,j),\ \text{lcs}(i,j-1)\} & \text{hvis } i,j > 0 \wedge x_i \neq y_j \end{cases} \tag{15}$$

fra [2, s. 9] benyttes.

I [2] gøres der opmærksom på, at man kan udfylde tallen over $\text{lcs}(i,j)$ bottom-up på flere måde - horisontalt, vertikalt eller diagonalt. Udfyldningen af tabellen er en ensartet proces. Derfor er hele processen ikke gennemgået nedenunder, men i stedet er der gengivet et par eksempler fra et tilfældigt sted under udfyldningen. Betragt tabellen i følgende stadie af udfyldningen:

	j	0	1	2	3	4	5	6	7	8	9
i		y_j	0	1	0	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	1	0	0↑	1↖	1←	1↖	1↖	1←	1↖	1↖	1←
2	0	0	1↖	1↑	2↖	2←	2←	2↖	2←	2←	2↖
3	0	0	1↖	1↑	2↖	2↑	2↑	3↖	3←	3←	3↖
4	1	0	1↑	2↖	2↑	3↖	3↖	3↑	4↖	4↖	4←
5	0	0	1↖	2↑	3↖	3↑	3↑	4↖	4↑	4↑	5↖
6	1	0	1↑	2↖	3↑	4↖	4↖				
7	0	0									
8	1	0									

Vi udfylder horisontalt, og nu er $i = 6$ og $j = 6$. Eftersom $x_6 = 1$ og $y_6 = 0$, så følger det af (15) at $\text{lcs}(6,6) = \max\{\text{lcs}(5,6),\ \text{lcs}(6,5)\} = \max\{4,4\} = 4$. I dette tilfælde er $\text{lcs}(5,6) = \text{lcs}(6,5)$, så den gemte pil kan enten være en venstre-pil eller en op-pil. Dermed er tabellen nu:

	j	0	1	2	3	4	5	6	7	8	9
i		y_j	0	1	0	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	1	0	0↑	1↖	1←	1↖	1↖	1←	1↖	1↖	1←
2	0	0	1↖	1↑	2↖	2←	2←	2↖	2←	2←	2↖
3	0	0	1↖	1↑	2↖	2↑	2↑	3↖	3←	3←	3↖
4	1	0	1↑	2↖	2↑	3↖	3↖	3↑	4↖	4↖	4←
5	0	0	1↖	2↑	3↖	3↑	3↑	4↖	4↑	4↑	5↖
6	1	0	1↑	2↖	3↑	4↖	4↖	4↑			
7	0	0									
8	1	0									

Nu er $i = 6$ og $j = 7$. Eftersom $x_6 = 1$ og $y_7 = 1$, så følger det af (15) at $\text{lcs}(6,7) = \text{lcs}(5,6) + 1 = 5$, og at vi skal gemme en skrå pil. Dermed er tabellen nu:

	j	0	1	2	3	4	5	6	7	8	9
i		y_j	0	1	0	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	1	0	0↑	1↖	1←	1↖	1↖	1←	1↖	1↖	1←
2	0	0	1↖	1↑	2↖	2←	2←	2↖	2←	2←	2↖
3	0	0	1↖	1↑	2↖	2↑	2↑	3↖	3←	3←	3↖
4	1	0	1↑	2↖	2↑	3↖	3↖	3↑	4↖	4↖	4←
5	0	0	1↖	2↑	3↖	3↑	3↑	4↖	4↑	4↑	5↖
6	1	0	1↑	2↖	3↑	4↖	4↖	4↑	5↖		
7	0	0									
8	1	0									

Dette er proceduren hele vejen igennem. Observer at vi i ovenstående fra starten har udfyldt base-case alle de steder, hvor den har effekt - dette kunne man have udeladt og istedet brugt (15) slavisk hele vejen igennem.

Den endelige udfyldte tabel er

	j	0	1	2	3	4	5	6	7	8	9
i		y_j	0	1	0	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	1	0	0↑	1↖	1←	1↖	1↖	1←	1↖	1↖	1←
2	0	0	1↖	1↑	2↖	2←	2←	2↖	2←	2←	2↖
3	0	0	1↖	1↑	2↖	2↑	2↑	3↖	3←	3←	3↖
4	1	0	1↑	2↖	2↑	3↖	3↖	3↑	4↖	4↖	4←
5	0	0	1↖	2↑	3↖	3↑	3↑	4↖	4↑	4↑	5↖
6	1	0	1↑	2↖	3↑	4↖	4↖	4↑	5↖	5↖	5↑
7	0	0	1↖	2↑	3↖	4↑	4↑	5↖	5↑	5↑	6↖
8	1	0	1↑	2↖	3↑	4↖	5↖	5↑	6↖	6↖	6↑

hvor $\text{lcs}(8,9)$ angiver, at den længste fælles delsekvens har længde 6.

Nu kan vi følge de gemte pile baglæns fra $\text{lcs}(8,9)$, og når en "skrå pil"følges udskrives x_i $(= y_j)$ ellers udskrives intet. Herved følger

	j	0	1	2	3	4	5	6	7	8	9
i		y_j	0	1	0	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	1	0	0↑	$\underline{1}$↖	1←	1↖	1↖	1←	1↖	1↖	1←
2	0	0	1↖	1↑	$\underline{2}$↖	2←	2←	2↖	2←	2←	2↖
3	0	0	1↖	1↑	2↖	2↑	2↑	$\underline{3}$↖	3←	3←	3↖
4	1	0	1↑	2↖	2↑	3↖	3↖	3↑	$\underline{4}$↖	4↖	4←
5	0	0	1↖	2↑	3↖	3↑	3↑	4↖	4↑	4↑	5↖
6	1	0	1↑	2↖	3↑	4↖	4↖	4↑	5↖	$\underline{5}$↖	5↑
7	0	0	1↖	2↑	3↖	4↑	4↑	5↖	5↑	5↑	$\underline{6}$↖
8	1	0	1↑	2↖	3↑	4↖	5↖	5↑	6↖	6↖	6↑

hvor de blå markeringer angiver den sti man følger og en understregning angiver steder hvor x_i $(= y_j)$ udskrives. I slutningen har man en LCS for $\langle 1,0,0,1,0,1,0,1\rangle$ og $\langle 0,1,0,1,1,0,1,1,0\rangle$ udskrevet i omvendt rækkefølge. Denne kan nemt vendes rundt, og dermed vil en LCS for de to sekvenser være: $\langle 1,0,0,1,1,0\rangle$

1.5 Opgave 5 - Cormen et al. opgave 15.4-2

Vi skal lave en algoritme der genskaber en LCS (længste fælles delsekvens) ved kun at bruge den udfyldte c-tabel, $X = \langle x_1, x_2, ..., x_m \rangle$ og $Y = \langle y_1, y_2, ..., y_n \rangle$. Den bruger altså ikke de pile som vi ser i f.eks. forrige opgave, men genskaber dem undervejs, dvs. vi behøver ikke at gemme pilene for at kunne finde en LCS efterfølgende.

Idé: hvis vi står i et felt (i, j) med værdi k, så ved vi, at der findes en LCS som har længde k. Hvis bogstaverne x_i og y_j er ens, så ved vi, at der findes en LCS, hvor det sidste bogstav er x_i $(= y_j)$ og hvor resten er en LCS for X_{i-1} og Y_{j-1}. Vi ved også, at der så findes en LCS med længden $k-1$, og at denne længde står i feltet $(i-1, j-1)$ (det svarer til at gå skråt). Vi kan derfor rekursivt udskrive denne LCS med længde $k-1$, og så bagefter udskrive bogstavet x_i, og på den måde få udskrevet hele strengen. Hvis bogstaverne x_i og y_j ikke er ens, så skal vi ikke udskrive et bogstav, men blot lave et rekursivt kald. For at finde ud af, om vi skal gå til venstre eller op, så ser vi på værdierne i de felter. Vi skal blot sørge for at gå hen til det felt, som har samme værdi som os (k), da vi her kan få printet en LCS som også har længde k. Baseret på den måde c-tabellen er bygget på, ved vi at mindst et af felterne har værdien k. Hvis både det felt til venstre og det over os har værdi k, så er det lige meget hvilket vi vælger. Nogle gange vil forskellige valg føre til forskellige strenge der bliver printet ud, men vi er garanteret at alle er længste fælles delsekvenser.

En algoritme der gør ovenstående er beskrevet nedenunder. Vi bliver også nødt til at holde styr på hvilken celle (i, j) vi er i, og vi skal huske at stoppe når der ikke længere er mere at printe. Det sker f.eks. når feltet vi er nået til har værdi 0. Dem der bruger algoritmen skal starte med at sætte $i = m$ og $j = n$ (længderne på X og Y).

```
1  Print-LCS(c,X,Y,i,j):
2      if c[i,j] == 0:
3          return
4
5      if X[i] == Y[j]:
6          Print-LCS(c,X,Y,i-1,j-1)
7          print(X[i])
8      else if c[i-1,j] == c[i,j]:
9          Print-LCS(c,X,Y,i-1,j)
10     else:
11         Print-LCS(c,X,Y,i,j-1)
```

Mindst en af i og j falder med 1 ved hvert rekursive kald, og vi er garanteret at hvis $i = 0$ eller $j = 0$ så er $c[i, j] = 0$. Dvs. en øvre grænse for antal rekursive kald er $m + n = O(m + n)$. Da vi kun bruger et konstant stykke arbejde i hvert rekursive kald (vi ser højst på to nabofelter), er det totale arbejde også $O(m + n)$.

1.6 Opgave 6 - Eksamen juni 2010, opgave 4

I denne opgave er vi interesserede i at finde den største *vægtede* delsekvens af to sekvenser X og Y, som består af tegn. Med andre ord, så er vægten af en fælles delsekvens summen af vægtene af tegnene i delsekvensen af X og Y. Vi er endvidere interesserede i at finde den delsekvens af X og Y som har den største vægt.

I opgavebeskrivelsen er vi givet et konkret eksempel på en tabel (se også Tabel 1) med tegn og deres korresponderende vægt, sammen med et eksempel på en delsekvens Z = acd af

sekvenserne X = acbd og Y = bacda. Ved at anvende nedenstående Tabel 1, så kan vi beregne vægten af denne delsekvens til at være $2 + 1 + 3 = 6$.

Tegn c	a	b	c	d
Vægt $w(c)$	2	4	1	3

Tabel 1: Tabel med tegn og deres korresponderende vægt

1.6.1 Delopgave a

I denne opgave er vi givet to sekvenser X = acbd og Y = bacda, som vi er interesserede i at finde den største vægtede delsekvens af dvs. vi er interesserede i at beregne $W(n, m)$.

I denne forbindelse anvender vi vægtene givet i Tabel 1 og udfylder en tabel ved, at anvende den rekursive formel for $W(i, j)$. Ved at udfylde tabellen kan vi efterfølgende finde den største vægt en fælles delsekvens af X og Y kan have.

Da foregående værdier anvendes i rekursionen for $W(i, j)$ påvirker dette hvordan tabellen kan udfyldes. Mere præcist bemærker vi at, for at kunne beregne værdien $W(n, m)$ skal der tages højde for mulige tilbagerettede afhængigheder, som det også ses i den rekursive formel hvor bl.a. foregående værdier $W(i-1, j)$, $W(i, j-1)$, $W(i-1, j-1)$ kan indgå i beregningerne for værdien $W(i, j)$.

Da dette er tilfældet nævner vi, at den nedenstående tabel udfyldes struktureret fra $j = 0$ til $j = 5$ for hvert $i = 0, \ldots, 4$. Ved at anvende rekursionen for $W(i, j)$ og den tidligere bemærkning omkring hvordan tabellen udfyldes, så resulterer dette i tabellen:

	j	0	1	2	3	4	5
i		y_j	b	a	c	d	a
0	x_i	0	0	0	0	0	0
1	a	0	0	2	2	2	2
2	c	0	0	2	3	3	3
3	b	0	4	4	4	4	4
4	d	0	4	4	4	7	7

Ved at se på indgangen $W(n = 4, m = 5) = 7$ i tabellen, ser vi at dette er den største vægt en delsekvens af X og Y kan have.

1.6.2 Delopgave b

I denne delopgave angiver vi pseudokoden for en algoritme der er baseret på dynamisk programmering og beregner den største vægt en delsekvens af X og Y kan have. Følgende pseudokode er baseret på den rekursive formel for $W(i, j)$:

```
# Lad X og Y være sekvenser af tegn
# Lad w( * ) være en funktion der returnerer en vægt givet et tegn
LongestCommonWeightedSubsequence(X, Y, w):
    n = X.length
    m = Y.length
    W # Lad W være en tom tabel af størrelse (n + 1) * (m + 1)
    for i = 0 to n:
        W[i][0] = 0
    for j = 0 to m:
        W[0][j] = 0
    for i = 1 to n:
        for j = 1 to m:
            if X[i] != Y[j]:
                W[i][j] = max(W[i - 1][j], W[i][j - 1])
            else:
                W[i][j] = max(W[i - 1][j], W[i][j - 1], W[i - 1][j - 1] + w(X[i]))
    return W[n][m]
```

I forhold til køretiden af denne algoritme ser vi, at vi indsætter i tabellen `W` i $O(1)$ tid (vi besøger desuden kun et konstant antal nabofelter når $W(i,j)$ evalueres og tabellen udfyldes). Vi gør dette først $n+1$ gange, så $m+1$ gange og til sidst $n \cdot m$ gange (angivet i samme rækkefølge som for-løkkerne forekommer). Køretiden vil derfor endeligt være $O(n)+O(m)+O(n \cdot m) = O(n \cdot m)$.

1.6.3 Delopgave c

Vi bruger notationen $X_i = x_1, \ldots, x_i$ og $Y_j = y_1, \ldots, x_j$. Se på en optimal løsning L for strengparret X_i og Y_j, dvs. en fælles delsekvens for disse med vægt $W(i,j)$. Som enhver anden fælles delsekvens kan den ses som en parring af ens tegn fra X_i og Y_j. Hvis $x_i \neq y_j$, kan parret (x_i, y_j) ikke være en del af L, og L ligger derfor helt inden for enten X_{i-1} og Y_j eller X_i og Y_{j-1}. Den må nødvendigvis være en optimal fælles delsekvens i disse (ellers er der en bedre sekvens end L i et af disse strengepar, og derfor også i strengparret X_i og Y_j, hvilket er i modstrid med at L er optimal). Dette viser $W(i,j) = \max\{W(i-1,j), W(i,j-1)\}$, og dermed anden linie i rekursionsformlen.

Hvis $x_i = y_j$, kan parret (x_i, y_j) godt være en del af løsningen L (og er så det sidste par i L). Hvis det er, ligger resten af L helt inden for X_{i-1} og Y_{j-1}, og må være en optimal løsning der (findes en bedre løsning der, kan en bedre løsning (end L) findes for X_i og Y_j ved at tilføje parret (x_i, y_j), hvilket er i modstrid med at L er optimal). Værdien af L er derfor $W(i-1,j-1)+w(x_i)$. Hvis derimod parret (x_i, y_j) ikke er en del af L, kan der argumenteres som ovenfor for at $W(i,j) = \max\{W(i-1,j), W(i,j-1)\}$. Vi ved ikke om parret (x_i, y_j) er med i L eller ej, men den er én af delene, så udtrykket $\max\{W(i-1,j), W(i,j-1), W(i-1,j-1)+w(x_i)\}$ må være en overgrænse for $W(i,j)$.

Omvendt findes der fælles delsekvenser af X_i og Y_j hvis vægt er hver af de tre værdier, der tages maksimum over (disse fælles delsekvenser er henholdsvis den optimale fælles delsekvens for X_{i-1} og Y_j, den optimale fælles delsekvens for X_i og Y_{j-1}, og den optimale fælles delsekvens for X_{i-1} og Y_{j-1} med parret (x_i, y_j) tilføjet). Udtrykket er derfor også en undergrænse for $W(i,j)$, som jo er den største vægt en fælles delsekvens af X_i og Y_j har. Derfor er udtrykket præcist lig med $W(i,j)$. Dette viser den sidste linie i rekursionsformlen. Hvis én af strengene er tomme, er den optimale delsekvens nødvendigvis tom, og har derfor vægten nul, hvilket viser første linie i rekursionsformlen.

2 Eksaminatorie-timer II

2.1 Opgave 1 - Opvarmning til projekt del III

2.1.1 Python

Herunder er Python kode for et program, der læser en fil én byte ad gangen og tæller hvor mange af hver af de 256 mulige bytes er repræsenteret i filen.

```
1  import sys
2  #start array off with 256 zeroes in a list
3  byteCounter = [0] * 256
4
5  #open the file in 'rb' read binary mode
6  with open(file=str(sys.argv[1]), mode='rb') as in_file:
7      #read first byte
8      currentByte = in_file.read(1)
9
10     #we continue reading until we reach end of file
11     while currentByte != b'':
12         #convert byte to int
13         byteAsInt = currentByte[0]
14         #use new int as index in array and increment
15         byteCounter[byteAsInt]+=1
16         #read next byte
17         currentByte = in_file.read(1)
18
19 #print result
20 for index in range(len(byteCounter)):
21     print('Byte ' + str(index) + ':', byteCounter[index])
```

Når man læser fra filer i Python kan det være en god ide at bruge

```
1  with open("path.txt") as file:
2      line = file.readline()
```

dette sørger for at filen kun er åben så længe man bruger den. Man kan dog ikke tilgå filen når man ikke længere er indenteret under `with` udtrykket.

På linje 6 hentes et argument systemet, dvs. programmet skal kaldes på følgende måde f.eks fra commandline "python programNavn.py sti/til/fil.type". I python er `sys.argv[0]` navnet på det program der bliver kørt, fra index 1 og opefter er så de argumenter programmet er kaldt med.

På linje 10 når der skrives `b''` betyder det et tomt byte object. Præfikset `b` indikerer et byte objekt. Når `.read(1)` funktionen læser EOF (end of file) så returnerer den et tomt byte objekt.

2.1.2 Java

Herunder er Java kode for en metode, der læser en fil én byte ad gangen og tæller, hvor mange af hver af de 256 mulige bytes er repræsenteret i filen.

```
import java.io.FileInputStream;

public static void main(String[] args) throws Exception{

    FileInputStream stream = new FileInputStream(args[0]);
    int[] byteCounter = new int[256]; //Tracks the number of bytes encountered.

    int currentByte = stream.read(); //Reads the first byte.
    while(currentByte != -1){    //stream.read() returns the byte read, or -1 if end of file.
        byteCounter[currentByte]++; //Increase the index corresponding to the last read byte.
        currentByte = stream.read(); //Reads the next byte.
    }

    for(int i=0; i<256; i++){    //Prints the result of byteCounter.
        System.out.println("Byte "+i+": "+byteCounter[i]);
    }
}
```

Først så åbnes en FileInputStream til filen, som skal læses. Filens navn kan gives som en String, så længe at filen ligger i samme mappe som koden. Der skabes også et int array, som holder styr på antallet af bytes i filen.

Derefter læses hver byte en efter en og noteres i byteCounter. Dette fortsættes til filen er læst. Når filen er læst, så returnerer stream.read() -1.

Til sidst printes antallet af forekomster af hver byte.

2.2 Opgave 2 - Opvarmning til projekt del III

2.2.1 Python

Herunder er Python kode for et program, der læser en givet fil én bit ad gangen ved at bruge det udleverede bitIO bibliotek.

```
import sys
import bitIO as bio #import library

#open the file in 'rb' read binary mode
with open(file=str(sys.argv[1]), mode='rb') as in_file:
    #give input file to bitreader
    bitReader = bio.BitReader(in_file)

    #read first bit
    currentBit = bitReader.readbit()

    #continue reading until end of file
    while bitReader.readsucces() != 0:
        #print bit and read next
        print(currentBit)
        currentBit = bitReader.readbit()
```

Den simpleste måde at importere et brugerlavet bibliotek er at ligge det på samme mappeniveau som resten af ens kilde-kode filer og så kalde import.

På linje 13 bruges en funktion fra bitIO biblioteket til at finde status for forrige kald til `.readbit()` hvis EOF er nået vil dette returnere 0.

Et eksempel på en tekst fil, et enkelt tegn i:

```
E
```

Outputtet når programmet køres med denne fil bliver:

```
0
1
0
0
0
1
0
1
```

Hvis vi oversætter dette til et heltal får vi: 69, ascii tegn nr. 69 er 'E'.

Programmet kan læse alle slags filformater, men når vi læser ascii er det nemt at verificere at der er blevet læst korrekt.

2.2.2 Java

Herunder er Java kode for en metode, der læser en givet fil én bit ad gangen ved at bruge readBit() fra det udleverede BitInputStream bibliotek.

```java
import java.io.FileInputStream;

public static void main(String[] args) throws Exception{
    BitInputStream stream = new BitInputStream(new FileInputStream(args[0]));

    int currentBit = stream.readBit(); //Reads the first bit
    while(currentBit!=-1){ //stream.readBit() returns the bit read, or -1 if end of file.
        System.out.print(currentBit); //Prints the currentbit without creating a new line (ln)
        currentBit = stream.readBit(); //Reads the next bit
    }
}
```

Hvor den sidste metode læste og talte bytes i et array, så printer denne metode bits, lige efter de læses.

2.3 Opgave 3 - Opvarmning til projekt del III

2.3.1 Python

Herunder er Python kode for et program, der læser 4 gange 32 bit og konverterer hver af dem som et heltal.

```python
import sys
import bitIO as bio #import library

#open the file in 'rb' read binary mode
with open(file=str(sys.argv[1]), mode='rb') as in_file:
    #give input file to bitreader
    bitReader = bio.BitReader(in_file)

    #read 4 * 32 bits
    firstInt = bitReader.readint32bits()
    secondInt = bitReader.readint32bits()
    thirdInt = bitReader.readint32bits()
    fourthInt = bitReader.readint32bits()

    #print result
    print(firstInt)
    print(secondInt)
    print(thirdInt)
    print(fourthInt)
```

Et eksempel på en tekst fil, én linje med 16 tegn:

```
aaaabcdexyzzklmn
```

Outputtet når programmet køres med denne fil bliver:

```
1633771873
1650680933
2021227130
1802268014
```

Hvis vi oversætter første linje i ouputtet til two's complement får vi: 01100001 01100001 01100001 01100001. Vi kan konvertere 01100001 til decimal-tal og får 97, ascii tegn nr. 97 er 'a'.

Linje to bliver til: 01100010 01100011 01100100 01100101. Dette kan oversættes til decimal-tallene 98, 99, 100 og 101 som er ascii værdierne for 'b', 'c', 'd' og 'e'.

2.3.2 Java

Herunder er Java kode for en metode, der læser 4 gange 4 bytes fra filen og tolker dem som ints, ved at bruge readInt() fra det udleverede BitInputStream bibliotek.

```java
import java.io.FileInputStream;

public static void main(String[] args) throws Exception{
    BitInputStream stream = new BitInputStream(new FileInputStream(args[0]));
    int firstInt = stream.readInt(); //Reads the first int
    int secondInt = stream.readInt();  //Reads the second int
    int thirdInt = stream.readInt(); //Reads the third int
    int fourthInt = stream.readInt();  //Reads the fourth int
    System.out.println("First int is :"+firstInt); //Prints the first int
    System.out.println("Second int is :"+secondInt);//Prints the second int
    System.out.println("Third int is :"+thirdInt); //Prints the third int
    System.out.println("Fourth int is :"+fourthInt);//Prints the fourth int
}
```

Samme test, som bliver kørt i Python versionen, køres her med filen:

```
aaaabcdexyzzklmn
```

Outputtet bliver det samme som i Python versionen:

```
First int is :1633771873
Second int is :1650680933
Third int is :2021227130
Fourth int is :1802268014
```

Da samme udregning gælder, så kan vi se, at programmet virker.

2.4 Opgave 4 - Eksamen juni 2013, opgave 7

2.4.1 Spørgsmål a

For at udfylde tabellen for $C(i, j)$ skal vi bruge den rekursive formel samt tabellen over værdierne for $P(k, l)$. Den rekursive formel kigger kortsagt på alle måder man kan opdele en plade og den tilhørende pris, hvor den største af disse bliver valgt.

I stedet for at gå igennem processen for hvert felt i tabellen for $C(i, j)$, gennemgås kun nogle af udregningerne for at vise den generelle metode. Man kan finde en tabel over alle resultater lige før spørgsmål b.

Først kigger vi på $C(1, 1)$, hvilket er case 1 i den rekursive formel, hvor vi bare skal aflæse prisen $P(1, 1) = 1$. Herefter kigger vi på $C(1, 2)$ som lander os i case 3 og vi ender med at skulle finde maksimum af prisen for et uddelt stykke $P(1, 2) = 2$ og opdelingen $C(1, 1) + C(1, 1) = 2$ (t skal være mellem $1 \leq t < 2$ hvilket må betyde $t = 1$ i det her tilfælde). $C(1, 3)$ bliver til max af den hele plade $P(1, 3) = 3$, og opdelingerne $C(1, 1) + C(1, 2) = 1 + 2$ og $C(1, 2) + C(1, 1) = 1 + 2 = 3$, her leder $t = 1$ og $t = 2$ til at de samme delproblemer skal løses. På samme måde kan man løse $C(1, 4)$ og $C(j, 1)$.

Nu kan vi kigge på $C(2, 2)$, hvor er vi i case 4 da både $i > 1$ og $j > 1$. Vi skal her finde det største af den hele plade $P(2, 2) = 5$, den ene måde at splitte pladen vandret på: $C(1, 2) + C(1, 2) = 4$ og den ene måde at splitte pladen lodret på: $C(2, 1) + C(2, 1) = 4$. Det er altså stadig bedst at sælge pladen hel her. Læg også mærke til at man kan bruge tidligere resultater til at udregne de nye, f.eks. har vi allerede fundet værdien for $C(2, 1)$, så vi skal ikke til at udregne den igen.

Til sidst ser vi på at udfylde $C(4, 4)$, her er vi igen i case 4. Prisen for den hele plade er $P(4, 4) = 21$. Der er følgende måder at splitte pladen vandret på: $C(1, 4) + C(3, 4) = 4 + 16$, $C(2, 4) + C(2, 4) = 10 + 10$ og $C(3, 4) + C(1, 4) = 16 + 4$. Der er det samme antal måder at splitte pladen lodret på: $C(4, 1) + C(4, 3) = 5 + 17$, $C(4, 2) + C(4, 2) = 10 + 10$ og $C(4, 3) + C(4, 1) = 5 + 17$. Det vil altså sige at $C(4, 4) = 5 + 17 = 22$ og at man skal dele en 4×4 plade i to stykker med størrelserne 4×1 og 4×3 for at kunne sælge den for mest muligt.

i/j	1	2	3	4
1	1	2	3	4
2	2	5	7	10
3	3	8	11	16
4	5	10	17	22

2.4.2 Spørgsmål b

Til at finde en algoritme som beregner $C(i, j)$ baseret på dynamisk programmering kan vi bruge samme metode som vi brugte for at udfylde tabellen ovenover. Mere præcist kan vi bruge bottom-up metoden, hvor vi først løser de mindste instanser af problemet inden vi fortsætter til de større. I forhold til en konkret algoritme kan vi først løse $C(1,1), C(1,2), ..., C(1,j)$, så $C(2,1), C(2,2), ..., C(2,j)$ og så videre indtil vi når $C(i,1), C(i,2), ..., C(i,j)$, hvor vi kan returnere $C(i,j)$. Det kan vi skrive i pseudokode sådan her:

```
1  Choco-Best-Cut(P, n, m):
2      Lad C være et array med størrelsen n*m
3      for i = 1 to n:
4          for j = 1 to m:
5              Fill-Table-C(i, j, C, P)
6      return C[n][m]
```

Så skal vi bare lave en metode `Fill-Table-C` som implementerer den rekursive formel. Det kan vi gøre ved at lave en direkte implementation af de 4 cases sådan her:

```
1  Fill-Table-C(i, j, C, P):
2      x = P[i][j]
3      if i > 1 and j == 1:
4          for s = 1 to i - 1:
5              x = max(x, C[s][1] + C[i - s][1])
6      else if i == 1 and j > 1:
7          for t = 1 to j - 1:
8              x = max(x, C[1][t] + C[1][j - t])
9      else if i > 1 and j > 1:
10         for s = 1 to i - 1:
11             x = max(x, C[s][j] + C[i - s][j])
12         for t = 1 to j - 1:
13             x = max(x, C[i][t] + C[i][j - t])
14     C[i][j] = x
```

Det er her vigtigt at vi aldrig tilgår en plads i `C`arrayet som ikke er blevet udfyldt af vores algoritme. Men heldigvis sikrer vores systematiske gennemgang at det ikke kan ske.

Pladsforbruget af vores algoritme bliver domineret af størrelsen på vores `C`-array som også er det eneste plads vi bruger udover lokale variabler. Derfor er vores pladsforbrug $\Theta(nm)$ (størrelsen på arrayet `C`).

Køretiden af algoritmen kan vi finde ved først at kigge på `Fill-Table-C`. Her vil vi i worst-case skulle finde maximum af 2 tal $O(i+j)$ gange (sidste case), hvilket er $O(\max(i,j))$. Dette kan man se hvis f.eks. i er større eller lig med j så vil $i+j \leq 2i$. Selve metoden `Fill-Table-C` bliver kaldt $n \cdot m$ gange, en gang for hvert felt i C-arrayet. Sætter vi de to ting sammen får vi en asymptotisk køretid på $O(n \cdot m \cdot \max(n,m))$

2.4.3 Spørgsmål c

Her skal vi argumentere for at den rekursive formel $C(i, j)$ er korrekt, det gør vi ved først at argumentere for det generelle tilfælde hvor $i > 1$ og $j > 1$.

Lad os kigge på en optimal opdeling OPT af en chokolade plade. Enten er den optimale opdeling den uopdelte plade, eller også er pladen blevet skåret over mindst 1 gang.

Er pladen skåret over er den blevet delt i 2, og eventuelle yderligere klip i OPT vil bryde dem ned i endnu mindre dele (hvis nødvendigt). Denne nedbrydning må være optimal for hver af disse to delplader, for ellers kunne OPTs handlinger erstattes af optimale opdelinger af disse to delplader, og derved selv blive bedre.

For alle muligheder for placering af første deling af pladen ser vi på summen af priserne på de optimale opdelinger af de to opståede delplader. Blandt disse tal møder vi (pga. ovenstående) værdien af OPT. Vi ser også på P[i,j], hvilket er prisen på OPT, hvis den uopdelte plade er optimal.

Uanset om optimale opdeling af pladen er den uopdelte plade eller en opdelt plade gælder følgende:

- Når vi ser på priserne for mulige opdelinger, så vi ser kun på tal som er mindre end eller lig med OPTs værdi.
- Blandt de tal, vi ser på, findes værdien for OPT.

Derfor er OPTs værdi lig maximum af de tal, vi ser på. Dette maksimum er netop hvad fjerde linie i den rekursive formel beregner.

De andre cases i formlen kan argumenteres for på samme måde, i disse cases er der blot færre måder man kan dele dem op på til at starte med. I case 1 i formlen kan man ikke dele pladen op, og den optimale pris må derfor være prisen $P(1, 1)$ for en 1×1 plade.

2.5 Opgave 5 - Cormen et al. opgave 14.4-5

Giv en $O(n^2)$ algoritme for at finde den *længste monotont stigende delsekvens* (LMSD) af en given sekvens af n tal.

Det er altid en god idé at kigge på et konkret eksempel. Dels så man forstår problemet, man vil løse, men også fordi dynamisk programmering kræver man finder en rekursiv beskrivelse af løsningen vha. optimale løsninger til delproblemer, og ens tankegang på et konkret eksempel kan måske koges ned til netop denne. Betragt f.eks. følgende sekvens af 5 tal

$$\langle 4, 20, 42, 0, 66 \rangle.$$

I dette tilfælde er den LMSD følgende:

Input sekvens	4	20	42	0	66
	↓	↓	↓		↓
LMSD	4	20	42		66

Lad os anskue ovenstående på følgende måde: vi har en LMSD $\langle 4, 20, 42, 66 \rangle$ for hele sekvensen. Her er 66 det sidste tal i delsekvensen, som har en længde på 4. Kigger vi på 42, som er det næstsidste tal i $\langle 4, 20, 42, 66 \rangle$, så har den LMSD, hvori 42 er det sidste element, længden

3. Vi kigger altså på et lidt andet problem - nemlig hvad er *længden af den længste monotont stigende delsekvens, hvor det sidste element er det i'te tal i inputsekvensen.* Løsningen til dette problem kan bruges til at løse det oprindelige problem.

Lad $l(i)$ angive længden af den længste monotont stigende delsekvens, hvor det sidste element er det i'te tal fra input sekvensen A. Der er to cases at betragte:

- Base-case er klart, hvor $A[j] > A[i]$ for alle $j < i$. Her er $l(i) = 1$, da den ikke vil kunne være den sidste i en længere monotont stigende delsekvens. Dette case opstår f.eks. med 0 og 4 i eksemplet ovenover.
- Hvis vi ikke er i base-case, så må $l(i)$ være det maksimale af optimale løsninger for tal der kommer før i og er højst det i'te tal (dvs. $l(j)$ hvor $j < i$ og $A[j] \leq A[i]$) plus én (for det i'te tal selv). Hvis der findes en bedre løsning til $l(j)$, så kunne vi benytte denne til den optimale løsning $l(i)$ og opnå en ægte bedre løsning. Denne case opstår ved resten af tallene i eksemplet ovenfor (dvs. for 20, 42 og 66).

Løsningen kan altså beskrives rekursivt ved

$$\mathrm{l}(i) = \begin{cases} 1 & \text{hvis } S = \emptyset \\ \max_{s \in S}\{l(s)\} + 1 & \text{hvis } S \neq \emptyset \end{cases} \tag{16}$$

hvor $S = \{j | j < i \wedge A[j] \leq A[i]\}$.

Benyttes en tabel med (16), så behøver vi kun løse hvert delproblem én gang. Vi har at gøre med en 1-dimensionel tabel (et array), hvor man for at udfylde $l(i)$ for det i'te element skal lave $O(1)$ arbejde for hvert element der kommer før det i'te - man skal altid bestemme mængden S og evt. udføre en max-operationen. Den samlede køretiden er da

$$1 + 2 + \cdots + (n-1) = \Theta(n^2)$$

Ovenstående var ikke det oprindelig problem som var ønsket løst, men man kan nemt løse det, da det bare er at bruge max-operationen over tabellen l, da den længste monotont stigende delsekvens af input jo ender ved et eller andet tal $A[i]$. Eftersom den er længst, må den må have længde $l[i]$, og værdierne $l[j]$ for $j \neq i$ kan ikke være større. Derfor vil max-operationen finde den. Dette ændrer ikke køretiden, da max-operationen tager $\Theta(n)$ tid.

2.5.1 Udfordringen

Opgavebeskrivelsen påstår man kan løse ovenstående problem ved at kombinere to algoritmer, som vi har set tidligere - nemlig en sorteringsalgoritme og algoritmen til LCS-problemet. Udfordringen består i at bevise de løser det sammen problem. Dette bevis består af to dele: (1) hvis en delsekvens er en monotont stigende delsekvens af X, så er det en fælles delsekvens mellem X og X' (som er X sorteret) (2) hvis en delsekvens er en fælles delsekvens mellem X og X', så er det en monotont stigende delsekvens af X.

(1) Lad D være en monotont stigende delsekvens af X. Observer der ikke eksisterer nogle inversioner mellem tallene i hverken D eller X' (da X' er sorteret). Da D er en delsekvens af X, så må ethvert tal i D også findes i X' (da den er en permutation af X). Da der ikke er nogle inversioner i D, så står tallene i D relativt til hinanden på samme måde, som de samme tal står relativt til hinanden i X'. Dermed må D være en fælles delsekvens mellem X og X'.

(2) Lad F være en fælles delsekvens mellem X og X'. Eftersom der ikke er nogen inversioner i X', og F er en delsekvens af X' (ellers kunne den ikke være en fælles delsekvens), så er der ingen inversioner i F. Tallene i F er altså i monotont stigende orden.

Eftersom en delsekvens af X er monotont stigende hvis og kun hvis den er en fælles delsekvens mellem X og X', så må den længste monotont stigende delsekvens af X være den længste fælles delsekvens mellem X og X'.

Køretiden er asymptotisk set uændret, da denne metode f.eks. kræver $\Theta(n \lg n)$ tid til sorteringen samt $\Theta(n^2)$ for at finde en LCS [2, s. 11].

2.6 Opgave 6 - Cormen et al. opgave 15-2

Observer, at et palindrom $p_1, p_2, ..., p_{k-1}, p_k$ består af en række par med samme bogstaver ((p_1, p_k) er et par, (p_2, p_{k-1}) er et par osv.). Der er $\lfloor \frac{k}{2} \rfloor$ par, og hvis k er ulige, er det midterste bogstav $p_{\lceil \frac{k}{2} \rceil}$ ikke parret med nogen.

Vi er givet strengen $X = \langle x_1, x_2, ..., x_n \rangle$. Lad LPS$[i, j]$ angive længden af den længste palindrom delsekvens (longest palindrome subsequence) af $X_{i,j} = \langle x_i, x_{i+1}, ..., x_j \rangle$, hvor $i \leq j+1$. Vi forestiller os, at vi har en tabel med n rækker og n kolonner. Vi vil gerne finde værdien LPS$[1, n]$ (og senere hvilke bogstaver der er med i palindromet af denne længde). Dette felt er markeret med X i vores tabel:

i \ j	1	2	3	4	5	6
1						X
2						
3						
4						
5						
6						

Vi ved allerede værdien af LPS$[i, j]$ for nogle værdier af i og j (base cases):

- Hvis $i = j + 1$, så er strengen $X_{j+1,j}$ tom, og den længste palindrome delsekvens må have størrelse 0.
- Hvis $i = j$, så har strengen $X_{j,j}$ længde 1, og dette ene bogstav er et palindrom i sig selv, så her har den længste palindrome delsekvens størrelse 1.

Vores base cases viser sig i vores tabel ved, at vi på hoveddiagonalen ($i = j$) kun har 1, og vi på diagonalen under hoveddiagonalen ($i = j + 1$) kun har 0:

i \ j	1	2	3	4	5	6
1	1					
2	0	1				
3		0	1			
4			0	1		
5				0	1	
6					0	1

Vi ser nu på den rekursive formel uden base cases inkluderet, dvs. når $i < j$. Her er vi givet $X_{i,j} = \langle x_i, x_{i+1}, ..., x_j \rangle$. Vi har to cases:

1. Hvis $x_i = x_j$, så ved vi at en optimal løsning, OPT, mindst har længde 2 og indeholder mindst ét par, da vi kan lave et palindrom med x_i og x_j. Vi ved, at OPT må indeholde mindst ét af bogstaverne x_i eller x_j fra det yderste par, fordi hvis det ikke havde nogen fra dette par, kunne vi udvide OPT og få en bedre løsning ved at tilføje parret til OPT, men dette er en modstrid da OPT ikke kan udvides (fordi den er optimal). Hvis OPT kun indeholder x_i og ikke x_j, så må OPT have parret x_i med et andet bogstav, men vi kan i stedet parre x_i med x_j og stadig have en palindromisk delsekvens af samme størrelse. Det samme kan vi gøre hvis OPT indeholder x_j og ikke x_i. Vi ved altså, at der findes en optimal løsning med det yderste par (x_i, x_j). Vi ved også, at en optimal løsning må indeholde en optimal løsning til delstrengen $X_{i+1,j-1}$, fordi hvis den ikke gjorde det, så kunne vi blot udskifte den ikke-optimale løsning for $X_{i+1,j-1}$ med en optimal, og få en bedre løsning for $X_{i,j}$, hvilket ville være en modstrid.

   Vi kan konkludere, at i case 1 er $\text{LPS}[i, j] = 2 + \text{LPS}[i + 1, j - 1]$

2. Hvis $x_i \neq x_j$, så kan OPT ikke indeholde parret (x_i, x_j), dvs. OPT må enten ligge i $X_{i,j-1}$ eller i $X_{i+1,j}$. Ingen palindromisk delsekvens af disse to delstrenge kan være større end OPT (da de også er palindromiske delsekvenser i $X_{i,j}$).

   Dvs. i case 2 er $\text{LPS}[i, j] = \max\{\text{LPS}[i, j - 1], \text{LPS}[i + 1, j]\}$

Vores endelige rekursive formel for LPS er dermed:

$$\text{LPS}[i, j] = \begin{cases} 0 & i = j + 1 \\ 1 & i = j \\ 2 + \text{LPS}[i + 1, j - 1] & i < j \text{ og } x_i = x_j \\ \max\{\text{LPS}[i, j - 1], \text{LPS}[i + 1, j]\} & i < j \text{ og } x_i \neq x_j \end{cases}$$

Det er nu også klart, at vi har brug for base case $i = j + 1$, da vi kan risikere at dette sker når vi lægger en til i og trækker en fra j i den tredje case.

I starten af opgaven konkluderede vi, at det var feltet $\text{LPS}[1, n]$ vi gerne ville have værdien for. Men ud fra vores rekursive beskrivelse kan vi se, at vi ikke bare kan starte med at udregne i dette felt, da de felter vi skal kigge på fra dette felt ikke er udfyldt endnu. Vi skal altså udfylde tabellen på en smartere måde. Da vi muligvis har behov for at se på feltet til venstre for os, nedenunder os, og skråt ned til venstre for os, så er en mulig rækkefølge vi kan udfylde tabellen på givet ved (her er tallene i fed den rækkefølge vi udfylder felterne i, så 1 er udfyldt først, så 2 osv.):

i \ j	1	2	3	4	5	6
1		**1**	**6**	**10**	**13**	**15**
2			**2**	**7**	**11**	**14**
3				**3**	**8**	**12**
4					**4**	**9**
5						**5**
6						

På denne måde er vi sikre på, at de felter vi har behov for at se på, allerede er blevet udfyldt.

Vi kan også se, at når vi står i et felt ser vi højst på 2 andre felter (det sker når $i < j$ og $x_i \neq x_j$), og da tabellen har størrelse n^2 (vi ser endda kun på halvdelen af felterne), så er køretiden $\Theta(n^2)$.

Algoritmen er kun beskrevet i tekst, men en implementering vil minde meget om longest common subsequence. Vi kan også på samme måde som i longest common subsequence holde styr på, hvor et feldt fik dens værdi fra, og bruge dette til en udskrivning af en længste palindromisk delsekvens.

Som beskrevet i opgavebeskrivelsen kan man også løse problemet ved at finde longest common subsequence mellem en streng, og strengen vendt om. Da LCS også har køretid $\Theta(n^2)$ og vi kan vende en streng om i lineær tid, så er køretiderne for de to metoder ens.

Litteratur

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[2] Rolf Fagerberg. Dynamisk programmering - Flere eksempler. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/dynamicProgrammingSlides2.pdf`, 2020.