INF 4130 Svarforslag til «Midterm», 01/11-2011

Oppgave 1

1.a

Den generelle reglen blir:

Dersom S[i] = T[j]: *true* dersom B[i-1, j-1] = *true* eller om B[i-1, j-1] = *true*
ellers: *false*

Dersom S[i] ≠ T[j]: *true* dersom B[i, j-1] = *true*
ellers: *false*

Initialsiseringen blir som følger:

B[0, j] for alle j settes til *true*, siden den tomme strengen alltid kan oppnås ved fjerninger

B[i,0] for i>0 settes til *false*, siden S-strengen her er lenger enn T-strengen, og da er det ikke håp.

I eksempelet under ser vi på S = ”abc” og T = ”aacbbcc”. Vi ser at det går bra.

		T						
		a	a	c	b	b	a	c
	t	t	t	t	t	t	t	t
a	.	t	t	t	t	t	t	t
S b	.	.	.	.	t	t	t	t
c	.	.	.	.	.	.	.	t

Et program som gjør dette kan være:

```
for j = 0 to T.length do { B[0, j] = true; }
for i = 1 to S.length do { B[i, 0] = false; }

for i = 1 to S.length do {                  // Rekkefølgen av løkkene er vilkårlig
   for j = 1 to T.length do {
      if S[i] == T[j] then {
         if B[i , j -1] == true or B[i-1, j -1] == true then { B[i, j] = true; }
                                                            else { B[i, j] = false;}
         } else {
            if B[i , j -1] == true then { B[i, j] = true; }
                                   else { B[i, j] = false; }
      }
   }
}
return B[S.length, T.length];
```

1.b

Vi innfører to varianter av true, nemlig: B[i, j] = *tu*, som betyr at man kan få likhet mellom S[1:i] og T[1:j], *uten* at man behøver å fjerne T[j]. Ellers B[i, j] = *tm*, som betyr at man kan få likhet mellom S[1:i] og T[1:j], men da er man nødt til å fjerne T[j].

Den generelle reglen blir da:

Dersom S[i] = T[j]: *tu* dersom B[i-1, j-1] = *tu* eller B[i, j] = *tm*
ellers: *tm* dersom B[i, j-1] = *tu*
ellers: *false*

Dersom S[i] ≠ T[j]: *tm* dersom B[i, j-1] = *tu*
elles: *false*

Initialsiseringen blir som følger:

B[0, 0] settes til *tu*, siden de tomme strenger er like (uten noe fjerning)

B[0, 1] settes til *tm*, siden vi kan oppnå den tomme streng ved å fjerne det ene symbolet i T[1].

B[0, j] for j>1 settes til *false*, siden vi måtte fjerne alle (og dermed flere ved siden av hverandre) om vi skulle oppnå den tomme streng.

B[i,0] for i>0 settes til *false*, siden S-strengen her er lenger enn T-strengen, og da er det ikke håp.

I eksempelet under ser vi på S = ”abc” og T = ”aacbcc”. Vi ser at vi her kan få likhet.

```
                          T
     |      a    a    c    b    c    c
  ---+-----------------------------
     |  tu   tm   .    .    .    .    .
     a |  .    tu   tu   tm   .    .    .
S    b |  .    .    .    .    tu   tm   .
     c |  .    .    .    .    .    tu   tu
```

Oppgave 2

For $0 \leq i \leq j \leq n$, lar vi c_{ij} være kostnaden av å kutte (del)stokken med endepunkter i og j. Vi kan bruke følgende rekurrensrelasjon, som vi altså fyller ut i en to-dimensjonal tabell $c[0,i][0,j]$:

$$c_{ij} = \min_k \{c_{ik} + c_{kj} + (p_j - p_i) : 0 \leq i < k < j \leq n\}.$$

Initialbetingelsene vil være $k_{ii+1}=0$.

Cluet er jo å se at dette er det samme som matrisemultiplikasjon, men at det her vil være en litt annen måte å beregne kostnaden på nå vi setter sammen to deler. Her vil det være lengden av den aktuelle stokken, avstanden mellom endepunktene, som vi finner ved $p_j - p_i$.

Om man heller vil skrive kode, kan det se slik ut:

```
for i = 0 to n-1 do
  c[i, i+1] = 0                          // initialbetingelsene (diag=1)
od
for diag = 2 to n do
  for i = 0 to n-diag do                 // beregner ihht
    j = i+diag                           // rekurrensrelasjonen
    min = ∞                              // angitt over (finner beste k)
    tempcut = i                          // på samme måte som på side
    for k = i+1 to j-1 do                // 271 i læreboka
      temp = c[i,k] + c[k,j] + (p[j]-p[i])
      if temp < min then
        min = temp
        tempcut = k
      fi
    od
    c[i,j] = min
    firstcut[i,j] = tempcut
  od
od
```

Oppgave 3

3.a

$l.m = 1$,
$l.u = 0$.

De uavhengige mengdene vil være $\{l\}$ og $\varnothing$, hhv.

3.b

```
v.m = 1     // node v er selv med
v.u = 0     // node v er ikke med
FOR i in v.children DO
{
   v.m = v.m + i.u                 // Er v med, kan vi ikke ta med barna.
   v.u = v.u + max(i.u, i.m)       // Er v ikke med, kan vi ta med de barna
}                                  // vi ønsker, men behøver ikke ta alle.
```

Det lille vanskelige her blir vel her nå (?) å se at man ikke *må* ta med alle barn av en node *v* som ikke selv er tatt med, men at man *kan* gjøre det, og at man altså tar med/utelater på den måten som gir høyest verdi; tilsvarende er det ikke om man har tatt med node *v*, da kan man *ikke* ta med noen av barna.

3.c

Svaret finner vi ved å ta max($r.u$, $r.m$).

Oppgave 4

4.a

1	2	3	4	5	6	6	6	Omregning av vinduet modulo 3	Match?
1	2	3						123/3 = 41 / 0(mod 3)	spuriøs match
	2	3	4					234/3 = 78 / 0(mod 3)	spuriøs match
		3	4	5				345/3 = 115 / 0(mod 3)	spuriøs match
			4	5	6			456/3 = 152 / 0(mod 3)	spuriøs match
				5	6	6		566/3 = 188,66 / 2(mod 3)	ikke match
					6	6	6	666/3 = 222 / 0(mod 3)	ekte match stopp

4.b

Det forventede antall spuriøse matcher er $n/3$ om vi gjør n forsøk. Vi gjør seks forsøk, og forventer derfor 6/3=2. De fire spuriøse matchene vi fikk er altså noe i overkant av det forventede antallet.

Oppgave 5

5.a

For at en heuristikk-funksjon h skal være monoton, må følgende holde:

i. For enhver måltilstand X skal vi ha $h(X) = 0$
ii. For enhver lovlig overgang fra M til N skal vi ha $h(M) \leq h(N) + \text{cost}(M, N)$.

Denne er ikke monoton, for eksempel vil tilstanden **HHHHSS_SS**, som er en lovlig slutt-tilstand, ha heuristikk-verdi 2. Et brudd på monotonitetskravet (punkt i).

Denne er monoton. La oss kalle heuristikken h, og se på kravene hver for seg:

i. Dette kravet er trivielt oppfylt, ettersom det i en måltilstand, uansett hvor den åpne ruten befinner seg, ikke vil være noen hvite brikker til høyre for noen svart brikke.
ii. Dette kravet er også oppfylt ettersom et enkeltflytt av kostnad 1 aldri vil endre på heuristikk-verdien, slik at vi alltid vil ha $h(N) = h(M)$ for en overgang fra M til N, og altså $h(M) \leq h(N) + 1$. For et hopp vil vi aldri endre heuristikken med mer enn 1, ettersom vi bare kan hoppe over 1 rute. Vi vil altså ha $h(N) \geq h(M) - 1$. Og altså $h(M) \leq h(M) - 1 + 2 \leq h(N) - 1$.

5.b

Ja, professoren kan benytte datterens foreslåtte funksjon. Denne vil også være monoton om delfunksjonene er det.

i-kravet er trivielt oppfylt når delfunksjonene er monotone. Vi må vise at ii-kravet også er oppfylt for h når h_i-ene er monotone. Vi kan uten tap av generalitet se på bare to av delfunksjonene, h_1 og h_2. Anta motsatt, altså at

$h_1(M) \leq h_1(N) + \text{cost}(M, N)$,
$h_2(M) \leq h_2(N) + \text{cost}(M, N)$, (delfunksjonene monotone)

men at

$h(M) > h(N) + \text{cost}(M, N)$. (sammensatt funksjon ikke monoton)

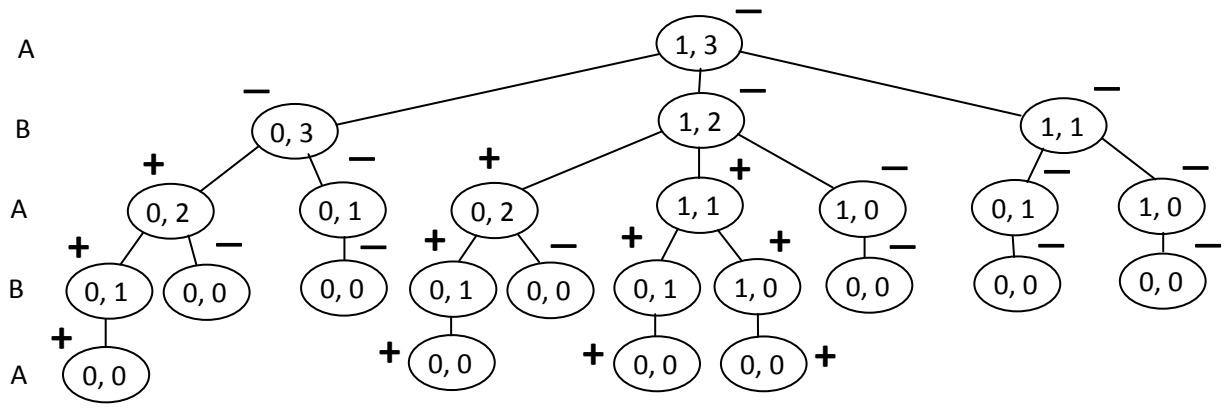
La så $h(N)$ være "dominert" av h_1 og $h(M)$ være "dominert" av h_2, slik at $h(N) = h_1(N) > h_2(N)$ og $h(M) = h_2(M) > h_1(M)$, men da vil vi ha

$h_2(M) > h_1(N) + \text{cost}(M, N) > h_2(N) + \text{cost}(M, N)$,

en selvmotsigelse når $h2$ er monoton i utgangspunktet.

Oppgave 6

6.a



6.b

Merkene er angitt på treet over.

6.c

Nei, for rotnoden er merket med –.

6.d

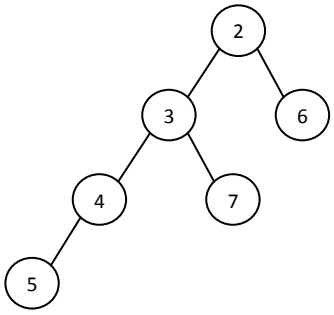
Ja. Her er det bare å se på det midterste subtreet i treet fra 1.a. Roten av det er riktignok merket med –, men man må kompensere for at det her er **B** som er i trekket, og for den finnes en vinnende strategi.

6.e

Ola har rett. For selv om man ser på det dårligste trekket først, så kan det neste man ser på være det beste trekket fra denne situasjonen, og det kan gjøre at man får avskjæring på neste nivå under senere noder.

Oppgave 7

7.a



Høyrestiene 3 og 2-7 spleises, vi får 2-3-7. Vi må snu i 2 fordi nullstikravet er brutt.

7.b

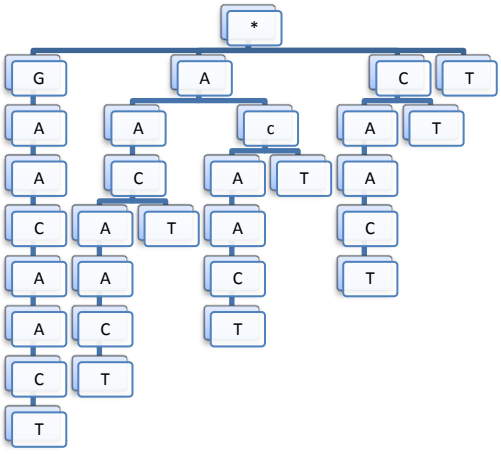
Vi er altså gitt følgende streng:

```
GAACAACT
```

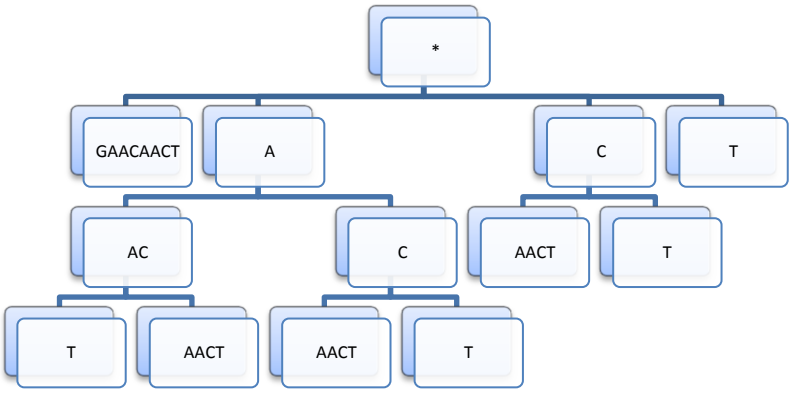
Strengen har følgende suffixer:

```
GAACAACT
 AACAACT
  ACAACT
   CAACT
    AACT
     ACT
      CT
       T
```

Som vel skulle gi oss følgende ukomprimerte suffix-tre:



Og altså i komprimert form, som suffix-tre:



7.c

Vi skal her altså finne shift-avstandene som brukes i Horspool-algoritmen. Det dreier seg bare om å finne (korteste) avstand en bokstav har fra strengens (patterents) ende.

`G`	7	(Nærmeste (eneste) `G` i avstand 7 fra enden)
`A`	2	(Nærmenste `A` i avstand 2 fra enden)
`C`	1	(Nærmeste `C` i avstand 1 fra enden)
`T`	8	(Endetegnet får ingen lavere verdi fra strengen)
`{A, …, Å}\{G,A,C,T}`	8	(Resten får strengens lengde)

[slutt]