Eksamen IN3130 og INF4130 høsten 2019

25. november kl. 14:30-18:30

Til alle oppgavene kan du supplere svaret med en tegning. Du skal tegne på spesialark som du får utdelt. Les informasjonen om håndtegning i lenken under oppgavelinjen. Fyll ut informasjonen på arket med én gang -- du vil ikke få ekstratid til dette når eksamen er over.

Tillatte hjelpemidler: Alle trykte og skrevne hjelpemidler er tillatt.

Med svarforslag i rødt

Oppgave 1 Flyt i nettverk

Spørsmål 1(a)

Vi har en rettet graf *G*, og *A* og *B* er to (forskjellige) noder i *G*. Vi sier at et sett (en mengde) av rettede veier fra *A* til *B* i *G* er *kant-uavhengig* dersom to veier i settet aldri bruker samme kant (men de kan godt gå gjennom samme node).

Du skal beskrive hvordan man kan finne det maksimale antall veier man kan ha i et kant-uavhengig sett av veier fra A til B. Du skal altså ikke finne selve veiene, men bare antallet. Når du beskriver din løsningsmetode kan du henvise til algoritmer som er kjent fra pensum, og bare angi hvilke data algoritmen(e) skal arbeide på. Forklar.

Hint: Overskriften på oppgaven er *Flyt i nettverk*

Problemet her er veldig nær til Mengers Teorem, som behandles i Kap 14.2.6 (side 442) i læreboka, og løsningen ligger i Korollar 14.2.10, og i teksten under det. Hele 14.2.6 er pensum, men vi snakket på forelesningen lite direkte om Mengers Teorem, men la derimot vekt på det som står under, nemlig at om vi har heltallige kapasiteter, så vil vi kunne oppnå maks flyt med en heltallig flyt, og at Ford-Fulkerson-algoritmen vil gi oss en slik flyt.

Antakeligvis vil mange svare noe i retning av at «man skal lage et flytnettverk av grafen ved at man setter kapasiteten av hver kant til 1. Så finner man maks flyt fra A til B i dette nettverket på standard måte (Ford-Fulkerson). Denne flyten blir altså enten 1 eller 0 på alle kanter, og kantene med flyt 1 vil dermed angi det maksimale antall veier i et kantuavhengig sett av veier fra A til B».

Spørsmål 1(b)

Vi sier nå at et sett med rettede veier fra *A* til *B* er *node-uavhengig* dersom to veier i settet aldri går gjennom samme node (bortsett fra at alle starter i *A* og ender i *B*)

Du skal beskrive hvordan man kan finne det maksimale antall veier man kan ha i et node-uavhengig sett av veier fra *A* til *B*. Du skal altså ikke finne selve veiene, men bare antallet. Når du beskriver din løsningsmetode kan du henvise til algoritmer som er kjent fra pensum, og bare angi hvilke data algoritmen(e) skal arbeide på. Forklar kort.

Trikset her er å dele hver node N i to noder N1 og N2, med en kant med kapasitet 1 fra N1 til N2. Videre skal alle kanter inn til N gå til N1, og alle kanter ut fra N gå fra N2. Alle kanter skal ha kapasitet 1. Om vi så finner en maksimal 0/1-flyt F i dette nettverket fra A til B er det opplagt at vi også har et node-uavhengig sett av F veier fra A til B, og at det ikke kan finnes noe større slikt sett. Dette er mer opplagt enn i 1(a) siden kantene med flyt 1 selv må utgjøre et antall node-uavhengige veier.

Oppgave 2

Spørsmål 2(a)

Er følgende språk uavgjørbart? Gi et bevis.
L = { M | M er koden til en Turingmaskin som avgjør HAMILTONICITY }

The language is undecidable. The proof is by a straightforward modification of the standard reduction from the Halting problem.

The reduction algorithm R reads M and x (the instance of Halting), and outputs the code of the machine M'. M' decides HAMILTONICITY if and only if M halts on x.

M': Simulate M on input x
Run an algorithm that decides Hamiltonicity

Since Hamiltonicity is decidable, the algorithm exists.

For complete credit the students should address the issue that R is indeed an algorithm, by describing it briefly.

R contains in its code, as a constant, a simple modification of the universal Turing machine, which simulates a (any given) specific M on a specific x, then runs a Hamiltonicity algorithm. R reads M and x from its input, inserts them into the slots ('variables') for M and x, and outputs the result as M'.

It is straightforward to show that this indeed is a reduction (that M' is in L if and only if M halts on x). For full credit, the students should at least mention this.

Oppgave 2.b

På forelesningene så vi at problemet PARTITION (gitt en mengde A og et heltall "størrelse" s(a) for hvert element a i A; avgjør om A kan deles i to delmengder slik at summen av størrelsene for hver delmengde er lik) er NP-komplett. Det er imidlertid ikke noe spesielt med tallet 2 her.

(1) Vis at 3-PARTITION (oppdeling i tre delmengder med lik sum) er NP-komplett.

(2) Generaliser beviset til å gjelde for k-PARTITION for enhver konstant k.

Vi kan imidlertid ikke generalisere uendelig. Det er opplagt at n-PARTITION, hvor n er antall elementer i A, er avgjørbart i polynomisk tid (alle elementene må opplagt ha samme størrelse).
(3) Hva med (n-1)-PARTITION? Eller (n-k)-PARTITION, hvor k er en konstant. Bevis svarene dine.

3-PARTITION is in NP: The certificate of membership is the solution. An algorithm can easily (in polynomial time) add up three sets of numbers and verify that the sums are the same.

(1) We prove that 3-PARTITION is complete for NP by the following reduction from PARTITION. (The idea is obvious – we add to A an element b, whose size is the required half of the sum.) Given an instance (A, s()) of PARTITION, we create an instance (B, q()) of 3-PARTITION as follows. B is A U {b}; q(a) = s(a) for all a in A; and q(b) is one half of the sum of s(a) for all a in A. It is easy to show that this can be done in polynomial time; and that the resulting instance of 3-PARTITION has a solution if and only if the original instance of PARTITION does – but the students should sketch an argument to receive full credit.

(2) A straight-forward modification of the above proof will do the job.

(3) (n-k)-PARTITION can be solved in polynomial time. The key is to observe that if a solution should exist, at least n-2k element sizes will have to be equal (because they will have to be partitioned as single elements). The remaining 2k (or less) elements can then be partitioned into the k (or less) subsets by exhaustive search. Since k is constant, the search will take only constant time.

Oppgave 2.c

For hver påstand, indiker om den er korrekt med «JA» eller «NEI», og gi en kort forklaring.

1. En Turingmaskin kan beregne enhver funksjon som kan beregnes på en moderne PC.
2. Noen uavgjørbare problemer kan løses ved å bruke en parallelldatamaskin.
3. Vi modellerer problemer som formelle språk for å kunne dele dem inn i klasser.
4. Hvert problem, eller formelle språk, har et "komplement". For eksempel er NON-HAMILTONICITY problemet å avgjøre om en gitt graf ikke er hamiltonsk. Hvis et problem er NP-komplett, så er komplementet også NP-komplett.
5. Noen NP-komplette problemer har polynomisk gjennomsnittstidskompleksitet.

1. TRUE according to Church-Turing thesis; but not provable.
2. FALSE. A sequential machine, including the Turing machine, can simulate a parallel machine. A RAM can simulate a single step or cycle of a p-processor PRAM by executing one step or cycle of each of the p processors in sequence.
3. TRUE. When a problem is represented by a formal languages, it 'becomes' a set of strings, and hence an element of a set of sets of strings. The subsets of the latter are classes.
4. FALSE (as implication). The complement is still NP-hard; but it is (for all we know…) not in NP, because a certificate of membership is missing.
5. TRUE. Hamiltonicity and 3-colorability are examples, as shown in class.

Oppgave 3 Dynamisk programmering

Spørsmål 3(a)

Vi har gitt et alfabet *A* (f.eks. alle norske bokstaver), samt en mengde *M* med ikketomme "ord" over dette alfabetet. Vi får så gitt en "lang" streng *S* over det samme alfabetet, og problemstillingen er å avgjøre om S kan deles opp i mindre biter slik at hver bit er et ord i mengden *M*. Symbolene (eller bokstavene) i S er nummerert fra 1 til n.

Vi skal løse dette med dynamisk programmering, og må derfor tenke på hva slags tabell som er egnet til å lagre svar på delproblemer. Foreslå en tabell for dette, og angi hva betydningen av dataene i denne skal være i den delen som er fylt ut.

Hint: Dette er nokså rett fram. Les resten av oppgaven før du svarer.

Tabellen kan naturlig være en boolsk array indeksert fra 1 til n, og etter hvert som den fylles ut skal T[i] == true bety at T[1:i] kan deles opp i biter som hver er i M, mens T[i] == false betyr at dette ikke er mulig.

Spørsmål 3(b)

Angi med en programskisse hvordan du vil fylle ut tabellen, og hva du vil gjøre i hvert skritt under utfyllingen. Bruk navnet *T* på tabellen i programskissen. Vi antar at mengden av ord i *M* er vesentlig større enn lengden av *S*, og at vi har et tabellverk over ordene i mengden *M*. For oppslag i denne finnes en metode

```
boolean isInM(String R) { ... }
```

som besvarer om *R* er med i mengden *M* eller ikke.

En sammenhengende substreng (en "bit") av *S* fra og med indeks *i* til og med indeks *j* kan du angi som S[i:j]. Husk å angi hvordan man finner svaret på problemet etter at tabellen er ferdig utfylt! Forklar.

Vurder også om man kan initialisere tabellen på passelig måte for å få færrest mulig tester inne i løkker i programmet.

Det kan lønne seg å utvide tabellen T til å være indeksert fra 0 til n, og at man initialiserer T[0] til å være true. Det tilsvarer at vi kan dele opp det tomme initial-segmentet av S i «null biter» som «hver» er i M (altså et krav som alltid er oppfylt). Resten av T kan greiest initialiseres til false. Programmet kan da bli noe slikt (men husk at det bare er spurt etter en skisse):

```
// T er en boolsk array indekset fra 0 til n. Antar at hele T er initialiert til false
T[0] = true;
for (integer i = 1, i <= n, i++) {
   for (integer j = 1, j <= i, j++) {
      if ( T[j-1]   and  isInM( T[j : i] ) )  { T[i] = true; <Leave the j-loop>;}    // Virker nå også for j == 0!
   }
}
```

Her tester vi altså for hver *i* om noen av de kortere intialsegmentene S[1:j] som har den ønskede egenskap kan forlenges til S[1:i] med et ord fra M. Om vi finner en slik behøver vi ikke teste flere *j*-verdier, og kan med en gang øke *i*.

Svaret ligger etter utførelse i T[n]. Dette er opplagt ut fra «invarianten» angitt i 3(a)

Spørsmål 3(c)

Vi antar at et oppslag i *M*-tabellen tar tid *O*(|*R*|), der |*R*| er antall bokstaver i strengen *R*. Av hvilken orden vil tidsbruken til algoritmen fra (b) være, uttrykt i *n*? Forklar.

Programmet i 3(b) har to løkker inne i hverandre.

- Den ytterste går med *i* fra 1 til *n*
- Den innerste går for hver *i* med *j* fra 1 til *i* (og vil i verste fall kalles for alle disse)
- Inne i den innerste gjøres et kall på isInM som tar tid fra *j* til 1 (og som i verste fall kalles for alle slike verdier)

Dette gir etter kjent resonnement tid *O*(*n*^3). Dette kan som «worst case» også oppnåes (med en faktor 1/6 pga. stadig «halvering»).

Spørsmål 3(d)

Vi stiller nå det ekstra kravet at oppdelingen av *S* i biter skal ha så få biter som mulig. Svaret skal rett og slett være det minste antall biter man kan klare seg med når hver bit skal være i *M*. Om *S* ikke kan deles opp i biter der hver bit er i *M,* så skal svaret være 0. Angi hva slags tabell du nå vil bruke, og angi med en programskisse hvordan du vil fylle den ut for å finne dette antallet. Forklar.

Man kan denne gangen ha en integer array T indeksert fra 0 til *n*, der T[*i*] angir om S[1: *i*] *kan* deles opp i biter fra M (angitt som T[*i*] >= 0) eller ikke (angitt som T[*i*] < 0). For første tilfellet angir T[*i*] det minste antall biter som er nødvendig.
Grunnen til at vi vil bruke negative tall (gjerne -1) til å angi «går ikke» er at det passer å initialisere T[0] til 0. Programmet kan da bli omtrent slik (men husk at det altså bare er spørsmål etter en skisse):

```
// T er en integer array indekset fra 0 til n. Initialiserer slik:
T[0] = 0;
for (integer i = 1, i <= n, i++) {
  integer minsteTilNaa = <STOR>;    // F.eks. lengden av S pluss 1
   for (integer j = 1, j <= i, j++) {
      if ( T[j-1] >= 0   and  isInM( T[j : i]) )  {
          if ( T[j-1] + 1 < minsteTilNaa )) {      // Her får vi en ny beste-notering
              minsteTilNaa = T[j-1] + 1;
              // Nå kan vi IKKE hoppe ut av j-løkka her. Alle j-muligheter må undersøkes!
          }
      }
   }  // Slutt j-løkka
   if (minsteTilNaa = <STOR>) { T[i] = -1; } else { T[i] = minsteTilNaa ;}
}  // Slutt i-løkka
```

Svaret står til slutt i T[*n*].

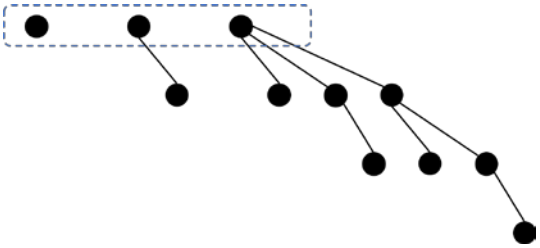
Oppgave 4 Prioritetskøer

I deloppgavene under skal du kun angi (eller skal vi også si begrunne kort?) hvilke av skogene som er lovlige og hvilke som er ulovlige binomial-heaper og Fibonacci-heaper. (Flere valg kan være lovlige i hver deloppgave.)

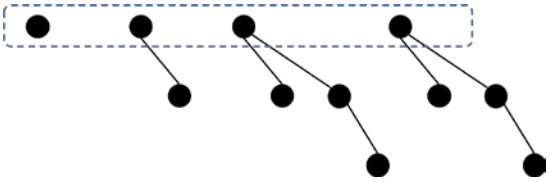
Spørsmål 4(a)

Hvilke av skogene under er en lovlig binomial-heap?

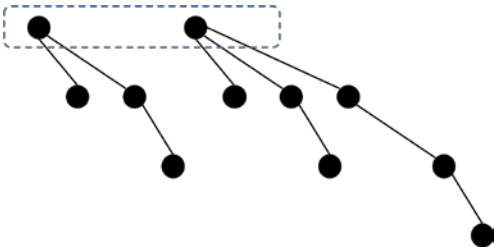
i.



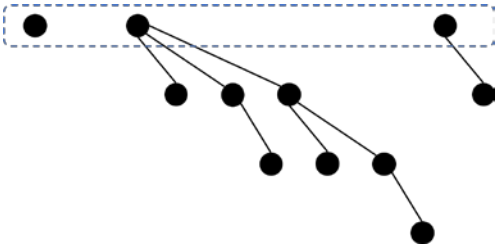
ii.



iii.



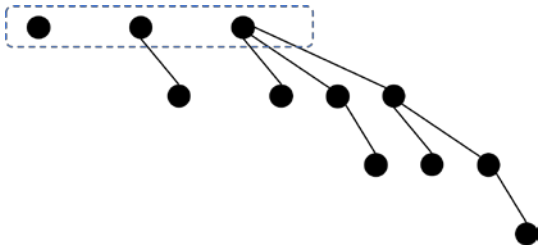
iv.



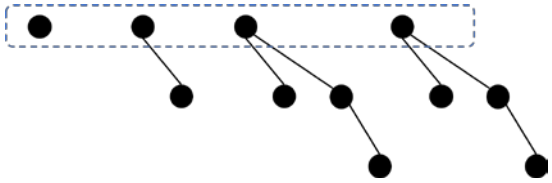
1. **OK**, alle trærne er korrekte binomialtrær, vi har maksimalt en av hver type, og de er rett sortert.
2. **Ikke OK**, to like B2 må evt. slås sammen til B3 (1101 [eller vanligst 1011] er binær representasjon av 11 - vi har 11 elementer i binomialheapen vår).
3. **Ikke OK**, det siste treet er ikke et korrekt binomialtre, det er ikke fullstendig.
4. **Ikke OK**, i en binomialheap forutsetter vi at binomialtrærne skal holdes sortert etter størrelse. (Trærne er imidlertid korrekte og vi har maks ett av hver størrelse.)

Spørsmål 4(b)
Hvilke av skogene under er en lovlig Fibonacci-heap?

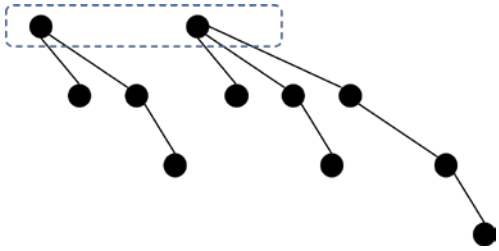
i.



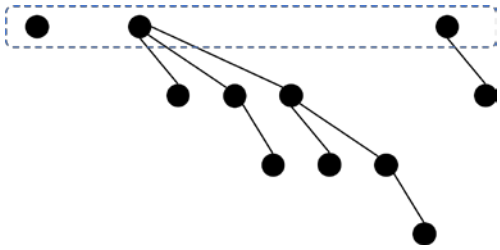
ii.



iii.



iv.



1. **Dette er OK**. Her er det faktisk også en korrekt binomialheap.
2. **Dette er OK**. I en Fibonacci-heap, feks. etter en lazy merge, kan vi ha flere like binomialtrær.
3. **Dette er OK**. I en Fibonacci heap, etter noen delete-operasjoner, kan vi ha partielle binomialtrær.
4. **Dette er OK**. I en Fibonacci-heap, f.eks etter en lazy-merge (eller noen delete-operasjoner), kan vi ha trær som ikke ligger i sortert rekkefølge.

Oppgave 5: A* og heuristikker
Vi ser på heuristikker for A*-algoritmen.

Spørsmål 5(a)
Besvar følgende, kort:

i. Hva vil det si at en heuristikk er monoton?

Definisjonen på en monoton heuristikk krever at vi skal ha:

1. $h(g) = 0$ for alle målnoder g,
2. $h(u) \leq c(u,v) + h(v)$ for alle u og v langs en sti mot en målnode hvor $c(u,v)$ er kostnaden/vekten av kanten uv.

Fra en målnode g og «bakover» kan altså ikke heuristikken vokse brattere enn $c(u,v)$.

ii. Hva er det vi oppnår/unngår med en monoton heuristikk i A*-algoritmen?

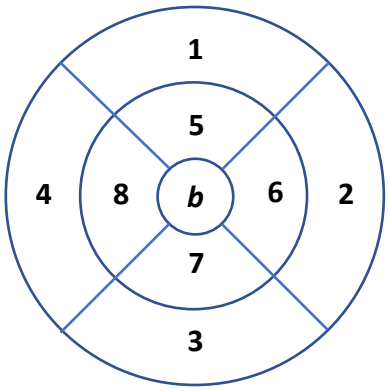
Det vi direkte oppnår med en monoton heuristikk er at nodene tas ut av prioritetskøen (PQ) i rett rekkefølge. Vi finner aldri en ny kortere vei til en node vi har tatt ut av prioritetskøen og over i treet av noder som vi er ferdige med.

Det vil altså aldri være noen re-beregning av noder som har fått sin korsteste-sti-verdi satt.

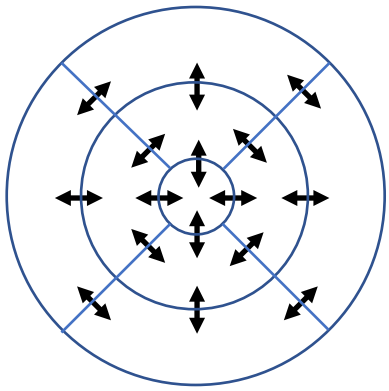
Den monotone heuristikken gir oss tidseffektiv algoritme.

Spørsmål 5(b)

Vi er gitt en rund variant av 8-spillet. Figur 1 viser den ønskede slutt-tilstanden for spillet, med den blanke brikken *b* («hullet») i midten.



Figur 1 Slutt-tilstanden for det runde 8-spillet.



Figur 2 Mulige flytt av den blanke brikken («hullet») i det runde 8-spillet.

Som i den obligatoriske oppgaven, er det enklest å modellere de mulige trekkene vi kan gjøre ved å se på hvordan den blanke brikken («hullet») flyttes. Figur 2 viser de mulige måtene å flytte den blanke brikken på for alle rutene på brettet. (Legg merke til at vi kun flytter en og en brikke, vi vrir ikke hele ringer i ett flytt.)

Besvar følgende: Er heuristikken *h* nedenfor monoton? Begrunn svaret kort.

h = antall feilplasserte brikker

Ja, *h* er monoton. Ett flytt (en brikke flyttes «inn i hullet») vil ha kostnad 1 og flyttet vil maksimalt kunne bedre (senke) heuristikkverdien med 1.

Står vi i tilstand X og gjør et flytt slik at vi kommer over i tilstand Y, vil vi altså ha:

h(Y) >= H(X) - 1

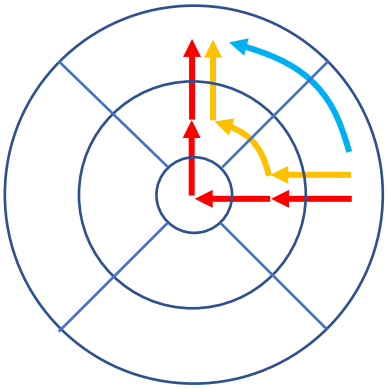
Vi vil med andre ord ha:

h(X) <= 1 + h(Y)

Spørsmål 5(c)

I den runde varianten av 8-spillet er ikke avstanden mellom to ruter helt entydig. Figur 3 viser tre mulige stier man kan flytte hullet langs, alle med forskjellig lengde. (Den røde stien tilsvarer å flytte langs en Manhattan-sti. Den blå stien vil på mange måter tilsvare et diagonalt flytt i det tradisjonelle, firkantede 8-spillet.)

I det tradisjonelle, firkantede 8-spillet kan vi bruke summen av Manhattan-avstandene mellom brikkenes nåværende posisjon og riktige posisjon som heuristikk.



Figur 3 Tre mulige mål på avstanden mellom to ruter på brettet.

Besvar følgende:

i. Beskriv (med ord) et egnet mål på et avstandsmål mellom to ruter på det runde brettet, som lar deg bruke summen av disse avstandene mellom brikkenes nåværende posisjon og riktige posisjon som heuristikk for A*-algoritmen på det runde 8-spillet.

Vi kan (les:må) bruke den korteste stien (indikert med blå pil) mellom to aktuelle ruter som avstandsmål for å få en monoton heuristikk. Vi kan ikke bruke Manhattan-avstanden (røde piler), eller mellomtingen (gule piler).

ii. Begrunn hvorfor ditt valgte avstandsmål vil gi en monoton heuristikk.

Ved å bruke summen av korteste stier mellom brikkers plass og riktige plass som heuristikk, vil vi maksimalt forbedre (senke) heuristikken med 1 i hvert flytt. Vi flytter kun en brikke, og den kan kun flytte seg ett steg langs den korteste stien mot sin rette plass.

Hadde vi isteden benyttet et annet (lengre) avstandsmål, ville vi med et flytt (f.eks langs den blå pilen) kunne forbedret heuristikken med mer enn 1. (Noen av flyttene på det runde brettet vil på den måten tilsvare diagonale flytt på et firkantet brett.)

Hadde vi valgt Mahattan-avstanden, og gått fra en tilstand X med et flytt langs den blå pilen til en tilstand Y kunne vi ha forbedret heuristikken med 4, slik at vi ville hatt

$h(Y) >= h(X) - 4$

Med h(Y) i intervallet [h(X)-4, h(X)-2] vil vi ikke ha

$h(X) <= h(Y) +1$

Oppgave 6: Strengsøk

Vi skal søke etter patterent ananas med to ulike strengsøke-algoritmer. Anta alfabetet vårt består av de vanlige norske bokstavene.

Spørsmål 6(a)

For hver mulige indeks for mismatch i patterent, vis hvor lange shift vi får med KnuthMorrisPratt-algoritmen. Indiker overlappende prefikser og suffikser.

0	1	2	3	4	5								
a	n	a	n	a	s								

Ingen overlapp er mulig.
Flytt: 1 steg.

0	1	2	3	4	5								
a	n	a	n	a	s								

Fortsatt ikke mulig med overlapp, strengen til venstre for mismatch-ruten er ikke lang nok til å få noe overlappende suffiks og prefiks, den er bare en lang.
Flytt: 1 steg.

0	1	2	3	4	5								
a	n	a	n	a	s								

Her har vi både et suffiks til venstre for mismatch-ruten, n, og et prefiks a, men det er ingen overlapp. Vi kan altså flytte 2 steg, og begynne igjen i ruten med index 2 i T.
Flytt: 2 steg.

0	1	2	3	4	5								
a	n	a	n	a	s								
		a	n	a	n	a	s						

Her har vi et prefix, a, som overlapper med et suffix, a. Vi kan altså flytte to steg, slik at de overlappende prefiksene og suffiksene står overfor hverandre.
Flytt: 2 steg.

0	1	2	3	4	5								
a	n	a	n	a	s								
		a	n	a	n	a	s						

Her har vi et prefiks, an, som overlapper med et suffix, an. Vi kan altså fortsatt bare flytte to steg, de overlappende prefiksene og suffiksene må stå overfor hverandre.
Flytt: 2 steg.

0	1	2	3	4	5								
a	n	a	n	a	s								
		a	n	a	n	a	s						

Her har vi et prefiks, ana, som overlapper med et suffix, ana. Vi kan altså fortsatt bare flytte to steg, de overlappende prefiksene og suffiksene må stå overfor hverandre.
Flytt: 2 steg.

Spørsmål 6(b)

Beregn shiftene-verdiene vi får med patternet i den forenklede Hoorspool-algoritmen som vi så på i kurset (den forenklede BoyerMoore-algoritmen).

ananas

a: 1
n: 2
resten av bokstavene i alfabetet: 6