KTH Datavetenskap
och kommunikation

Övningsuppgifter i

Algoritmer, datastrukturer och komplexitet

hösten 2015

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 1

Algoritmanalys

Ordo Jämför följande par av funktioner med avseende på hur dom växer då n växer. Tala i varje fall om ifall $f(n) \in \Theta(g(n))$, $f(n) \in O(g(n))$ eller $f(n) \in \Omega(g(n))$.

	$f(n)$	$g(n)$
a)	$100n + \log n$	$n + (\log n)^2$
b)	$\log n$	$\log n^2$
c)	$\frac{n^2}{\log n}$	$n(\log n)^2$
d)	$(\log n)^{\log n}$	$\frac{n}{\log n}$
e)	$\sqrt{n}$	$(\log n)^5$
f)	$n2^n$	3^n
g)	$2^{\sqrt{\log n}}$	$\sqrt{n}$

Division I denna uppgift ska en divisionsalgoritm kallad trappdivision analyseras. Så här ser trappdivision ut när 721 delas med 64. (Den som är van vid liggande stolen skriver istället nämnaren på högra sidan av täljaren.)

```
   0 1 1
 ---------
64 | 7 2 1
   -0
   ---
   7 2
   -6 4
   ----
     8 1
     -6 4
     ----
     1 7
```

Algoritmen börjar med att kolla hur många gånger 64 går i 7, den mest signifikanta siffran i 721. Eftersom $64 > 7$ är det 0 gånger. Vi har $0 \cdot 64 = 0$ så vi subtraherar 0 från 7 och får 7. Vi multiplicerar 7 med 10 (basen), flyttar ner nästa siffra i 721, nämligen 2, och fortsätter att dividera 72 med 64 för att få nästa siffra i kvoten. Vi fortsätter på detta sätt, subtraherar antalet gånger 64 går i varje tvåsiffrigt tal, flyttar ner nästa siffra och slutar när alla heltalssiffror i kvoten har beräknats. Talet 17 längst ner är divisionens rest.

Vi kan formulera detta i följande algoritm där vi beräknar $q = b/a$ där a, b är n-bitstal ($a = a_{n-1}a_{n-2}\cdots a_0$, $b = b_{n-1}b_{n-2}\cdots b_0$) i basen B. Låt $x \ll y$ vara x vänsterskiftat y steg. För att det senare ska vara lättare att bevisa att algoritmen är korrekt har ingångsvillkor, utgångsvillkor och invarianter satts ut (gås igenom på föreläsning 6).

$Div(a, b, n) =$
PRE: $a > 0, b \geq 0$, a och b lagras med n bitar.
POST: $qa + r = b, 0 \leq r < a$
$r \leftarrow 0$
for $i \leftarrow n - 1$ **to** 0 **do**
INV: $(q_{n-1}...q_{i+1}) \cdot a + r = (b_{n-1}...b_{i+1}), 0 \leq r < a$
$r \leftarrow (r \ll 1) + b_i$ /* Byt till nästa siffra */
$q' \leftarrow 0$
$a' \leftarrow 0$
while $a' + a \leq r$ **do** /* Hitta max q' så att $q'a \leq r$ */
INV: $a' = q'a \leq r$
$a' \leftarrow a' + a$
$q' \leftarrow q' + 1$
$q_i \leftarrow q'$
$r \leftarrow r - a'$
return $\langle q, r \rangle$ /* kvot och rest */

Vad är tidskomplexiteten? Bör bitkostnad eller enhetskostnad användas?

Euklides algoritm Analysera Euklides algoritm som hittar största gemensamma delaren mellan två heltal. Analysera både med avseende på enhetskostnad och bitkostnad och analysera skillnaden. Euklides algoritm lyder på följande sätt, där vi förutsätter att $a \geq b$.

$gcd(a, b)=$
if $b|a$ **then**
$gcd \leftarrow b$
else
$gcd \leftarrow gcd(b, a \bmod b)$

Potenser med upprepad kvadrering Följande algoritm beräknar tvåpotenser när exponenten själv är en tvåpotens.

Indata: $m = 2^n$
Utdata: 2^m
$power(m) =$
$pow \leftarrow 2$
for $i \leftarrow 1$ **to** $\log m$ **do**
$pow \leftarrow pow \cdot pow$
return pow

Analysera denna algoritm både med avseende på enhetskostnad och bitkostnad och förklara vilken kostnadsmodell som är rimligast att använda.

Lösningar

Lösning till Ordo

a) Vi beräknar gränsvärdet för kvoten mellan funktionerna.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{100n+\log n}{n+(\log n)^2}=\lim_{n\to\infty}\frac{100+(\log n)/n}{1+(\log n)^2/n}=100.$$

Eftersom gränsvärdet är en konstant drar vi slutsatsen att $f(n)\in\Theta(g(n))$.

b) Vi noterar att $\log n^2=2\log n$, så $\log n\in\Theta(\log n^2)$.

c)

$$\frac{g(n)}{f(n)}=\frac{n(\log n)^2}{n^2/\log n}=\frac{(\log n)^3}{n}\to 0 \text{ då } n\to\infty.$$

Därför är $g(n)\in O(f(n))$ och $f(n)\in\Omega(g(n))$.

d) Substituera $m=\log n$ och jämför sedan $f_2(m)=m^m$ och $g_2(m)=2^m/m$:

$$\frac{g_2(m)}{f_2(m)}=\frac{2^m}{m\cdot m^m}=\frac{1}{m}\left(\frac{2}{m}\right)^m\to 0$$

då $n\to\infty$. Därför är $f(n)\in\Omega(g(n))$ och $g(n)\in O(f(n))$.

e) Polynomiska funktioner vinner alltid över polylogaritmiska. Därför är $f(n)\in\Omega(g(n))$ och $g(n)\in O(f(n))$.

Om man vill ha ett mer formellt bevis kan man använda lemmat som säger att för alla konstanter $c>0$, $a>1$ och alla monotont växande funktioner $h(n)$ är $(h(n))^c\in O(a^{h(n)})$. Om vi väljer $h(n)=\log n$, $c=5$ och $a=\sqrt{2}$ så får vi $(\log n)^5\in O(\sqrt{2}^{\log n})=O(\sqrt{n})$ eftersom $\left(2^{1/2}\right)^{\log n}=\left(2^{\log n}\right)^{1/2}=n^{1/2}$.

f) $\frac{f(n)}{g(n)}=\frac{n2^n}{3^n}=n\left(\frac{2}{3}\right)^n$. En exponentialfunktion som $(2/3)^n$ vinner alltid över en polynomisk funktion (som n), men låt oss bevisa det med l'Hôpitals regel. Formulera först om uttrycket: $n\left(\frac{2}{3}\right)^n=n/\left(\frac{2}{3}\right)^{-n}$. Inför beteckningar för nämnare och täljare, $r(n)=n$ och $s(n)=\left(\frac{2}{3}\right)^{-n}$, och derivera.

$$r'(n)=1,\qquad s'(n)=\left(\frac{2}{3}\right)^{-2n}\left(\frac{2}{3}\right)^n\ln\left(\frac{3}{2}\right)=\frac{\ln(3/2)}{\left(\frac{2}{3}\right)^n}$$

Vi kan använda l'Hôpital eftersom $r(n)\to\infty$, $s(n)\to\infty$ och $s'(n)\neq 0$.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{r(n)}{s(n)}=\lim_{n\to\infty}\frac{r'(n)}{s'(n)}=\frac{\left(\frac{2}{3}\right)^n}{\ln\left(\frac{3}{2}\right)}\to 0$$

och vi har visat att $f(n)\in O(g(n))$.

g) $\sqrt{n}\in\Omega(2^{\sqrt{\log n}})$. Notera bara att $\sqrt{n}=2^{\frac{1}{2}\log n}$ och att $\lim_{n\to\infty}2^{\sqrt{\log n}-\frac{1}{2}\log n}=0$.

□

Lösning till Division
Eftersom indata alltid består av två tal men talens storlek kan variera, och eftersom tiden bara beror på talstorleken, så måste rimligen bitkostnad användas.

For-slingan går n varv. I varje varv har vi ett konstant antal tilldelningar, jämförelser, additioner och subtraktioner samt en while-slinga som går högst B varv (eftersom B är basen och talet $B \cdot a \geq r$). Eftersom basen kan anses vara en konstant så innehåller varje varv i for-slingan ett konstant antal operationer. Och eftersom all aritmetik görs med n-bitstal så tar varje jämförelse, addition och subtraktion tiden $O(n)$. Totalt får vi $n \cdot c \cdot O(n) = O(n^2)$. □

Lösning till Euklides algoritm
Eftersom algoritmen är rekursiv är det inte uppenbart att den alltid tar slut (terminerar), så vi börjar med att bevisa det. I termineringsbevis använder man oftast en potentialvariabel som har en undre gräns (till exempel noll) och vid varje anrop minskar med ett heltalssteg. I denna algoritm kan vi använda a som potentialvariabel. Eftersom a alltid är minst lika stor som b och algoritmen terminerar så fort $a = b$ eller $b = 1$ så har vi en undre gräns för a och vi kan se att a minskar i varje anrop. Antalet anrop beror tydligen på parametrarnas storlek, så låt oss beräkna hur den största parametern, a, minskar. Låt a_i vara as värde i anrop nummer i i algoritmen.

Lemma 1 $a_{i+2} \leq a_i/2$.

Bevis. Vi vet att $a_{i+2} = b_{i+1}$ och $b_{i+1} = a_i \bmod b_i$, varför $a_{i+2} = a_i \bmod b_i$. Anta nu att $a_{i+2} > a_i/2$. Detta innebär att $b_i \geq a_i/2$, vilket ger en motsägelse, eftersom $a_i = a_{i+2} + cb_i > a_i/2 + a_i/2 = a_i$. □

Med hjälp av lemmat kan vi visa att $\lceil \log a_{i+2} \rceil \leq \lceil \log \frac{a_i}{2} \rceil = \lceil \log a_i - \log 2 \rceil = \lceil \log a_i \rceil - 1$. Det betyder att i vartannat anrop av funktionen minskas parametrarnas storlek med (minst) en bit. Därför kan antalet anrop vara högst $2\lceil \log a \rceil$.

I varje rekursivt anrop görs bara en modulooperation och med enhetskostnad tar det konstant tid. Därmed har vi kommit fram till att tidskomplexiteten är $2\lceil \log a \rceil = 2n \in O(n)$.

När vi analyserar algoritmen med avseende på bitkostnad måste vi vara noggrannare. En division mellan två n-bits heltal tar (som vi har sett) $O(n^2)$, och modulooperationen tar därför $O(n^2)$. Så om vi gör $O(n)$ anrop och varje anrop tar $O(n^2)$ så blir den totala bitkomplexiteten $O(n^3)$. Vilken kostnadsmodell som är bäst beror på om talen lagras i variabler av fix längd eller har dynamisk längd. □

Lösning till Potenser med upprepad kvadrering
Slingan går $\log m = n$ varv. Varje varv tar konstant tid med enhetskostnad. Komplexiteten blir därför $O(n)$.

Om vi använder bitkomplexitet måste vi undersöka vad varje multiplikation i slingan kostar. Vi antar att kostnaden för att multiplicera ett l-bitstal med sig själv är $O(l^2)$ (vilket är en överdrift – det finns snabbare sätt). I varv i var variabeln $pow = pow_i$ värdet 2^{2^i} så varje multiplikation kostar $O((\log pow_i)^2) = O((\log 2^{2^i})^2) = O(2^{2i})$. Om vi summerar över alla varv får vi

$$\sum_{i=1}^{\log m} c2^{2i} = c \sum_{i=1}^{\log m} 4^i = 4c \sum_{i=0}^{\log m - 1} 4^i = 4c\frac{4^{\log m} - 1}{4 - 1} \in O(4^{\log m}) = O(4^n).$$

Algoritmen har alltså linjär komplexitet med avseende på enhetskostnad men exponentiell komplexitet med avseende på bitkostnad! Om talen lagras i variabler av fix längd kommer man snart att slå i taket. I praktiken behövs variabler av dynamisk storlek och bitkostnad passar därför bäst. □

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 2

Datastrukturer och grafer

På denna övning är det också **inlämning av skriftliga lösningar av teoriuppgifterna till labb 1** och muntlig redovisning av teoriuppgifterna. Teoriuppgifterna redovisas individuellt, till skillnad från labben som görs i par.

Samsortering Beskriv en algoritm som samsorterar k stycken var för sig sorterade listor i tid $O(n \log k)$ där n är det totala antalet element.

Datastruktur med max och min Konstruera en datastruktur som har följande operationer och komplexitet:

- `insert(x)` (insättning av ett element) tar tid $O(\log n)$,
- `deletemin()` (borttagning av det minsta elementet) tar tid $O(\log n)$,
- `deletemax()` (borttagning av det största elementet) tar tid $O(\log n)$,
- `findmin()` (returnerar minsta elementet) tar tid $O(1)$,
- `findmax()` (returnerar största elementet) tar tid $O(1)$.

n är antalet element som för närvarande finns lagrade. Du får anta att man kan jämföra två element och ta reda på vilket som är störst i tid $O(1)$.

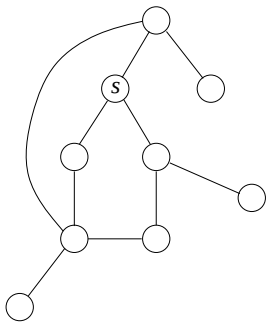
Bloomfilter med borttagning Det finns bara två tillåtna operationer på ett vanligt Bloomfilter: Insert och IsIn. Hur kan man modifiera datastrukturen så att den också tillåter operationen Remove, som inte får ta längre tid än Insert?

Suffixregeluppslagning i Stava Stava har cirka 1000 suffixregler av typen

-orna ← *-a, -an, -or*

När ett ord som *dockorna* ska kontrolleras kommer Stava att i tur och ordning gå igenom alla suffix efter den första stavelsen, det vill säga *-ϵ, -a, -na, -rna, -orna, -korna, -ckorna*. Hur ska suffixreglerna lagras för att det ska gå snabbt att slå upp suffixen?

Grafsökning Gör en DFS- och en BFS-genomgång av följande graf. Starta i hörnet s och numrera hörnen i den ordning dom besöks.



Topologisk sortering av DAG En DAG är en riktad acyklisk graf. En topologisk sortering av en sån graf är en numrering av hörnen så att alla kanter går från ett hörn med lägre nummer till ett hörn med högre nummer.

Modifiera den vanliga djupetförstsökningsalgoritmen så att den konstruerar en topologisk sortering av grafen i linjär tid.

Klick En *klick* (eng. clique) i en oriktad graf är en mängd hörn där varje par av hörn har en kant mellan sig. Ett viktigt problem i datalogin är problemet att hitta den största klicken i en graf.

Ett annat mycket omtalat problem är problemet att hitta den största *oberoende mängden* i en graf. En oberoende mängd (eng. independent set) är en mängd hörn som inte har någon kant alls mellan sig.

Dessa båda problem är så lika varandra att om man har en algoritm som givet en graf hittar den största klicken så kan man utnyttja den algoritmen i en för övrigt mycket enkel algoritm som givet en graf hittar den största oberoende mängden. Visa detta!

Kantmatrisprodukt På föreläsningen beskrevs hur en riktad graf $G = \langle V, E \rangle$ representeras som en kantmatris B (eng. incidence matrix) av storlek $|V| \times |E|$. Beskriv vad elementen i matrisen BB^T representerar (där B^T är B transponerad).

Bipartithet Beskriv och analysera en algoritm som avgör om en graf är bipartit. Tidskomplexiteten för algoritmen ska vara linjär i antalet kanter och antalet hörn i grafen.

Lösningar och ledningar

Ledning till Samsortering
Metod 1: Använd en heap där ett element från varje lista finns med och där det minsta elementet ligger i roten. Varje heapoperation tar $O(\log k)$ och det görs n insättningar i och n borttagningar ur heapen.
Metod 2: Samsortera listorna parvis. Nu finns det bara $k/2$ sorterade listor. Samsortera dessa listor parvis, och så vidare. Efter $\log k$ samsorteringsomgångar finns det bara en lista kvar. Varje samsorteringsomgång tar tid $O(n)$ eftersom det är totalt n element som ska samsorteras i varje omgång.

Lösning till Datastruktur med max och min
Antingen kan man lösa uppgiften med hjälp av två trappor (heapar) med samma element, men där den ena är ordnad med minsta elementet i roten och den andra med största elementet i roten eller också med hjälp av ett balanserat sökträd som utökas med två extra variabler, en för det minsta värdet i trädet och en för det största.

Implementationerna är i båda fallen enkla. I binärträdsfallet är det trivialt att skriva `findmin` och `findmax`. För dom tre övriga operationerna använder man motsvarande vanliga balanserade sökträdsoperationer men efterbehandlar genom att kopiera dom minsta och största värdena i trädet till dom två variablerna. □

Lösning till Bloomfilter med borttagning
Låt Bloomfiltret bestå av tal (till exempel 8- eller 16-bits heltal) istället för bitar. Talet noll motsvarar bitvärdet 0 och varje positivt tal motsvarar bitvärdet 1. Istället för att sätta en bit till 1 så ökar Insert istället motsvarande tal med ett. Remove fungerar precis som Insert men minskar med ett istället för att öka med ett. Nu räknar heltalen hur många element som är hashade till varje position. När ett heltal kommit ner till noll så betyder det att inget element längre hashas till den positionen.

Storleken på heltalen måste avpassas till antalet element som beräknas ligga i Bloomfiltret. Man får nämligen problem om en räknare når heltalstypens tak. En möjlighet är att införa en extra bit per position som håller reda på om taket någon gång nåtts för den positionen. När en räknare med den biten satt kommer ner till noll så måste hela Bloomfiltret beräknas på nytt.

Ett alternativ är att den räknare som slår i taket får sitta fast i taket även om Remove anropas. Det betyder att den räknaren aldrig kommer att nå noll, även om alla element som hashas till den positionen tas bort. Det kommer förmodligen aldrig att inträffa eftersom det var så många element som hashats till den positionen, och om det inträffar så ökar sannolikheten för fel bara för dom ord som hashas dit, och det förmodligen inte så mycket. □

Lösning till Suffixregeluppslagning i Stava
Det man slår upp på är vänsterleden i reglerna. Vi kallar dessa för ingångssuffixen. Alla regler som har tomt ingångssuffix lagrar vi i en lista för sig. Övriga regler lagrar vi i en array sorterade efter ingångssuffixet läst baklänges. Vi använder latmannahashning på en bokstav, det vill säga vi har en bokstavsindexerad array som anger indexet till första ingångssuffixet som slutar på den bokstaven.

Vid sökning kollar man först i listan med regler med tomt ingångssuffix. Därefter slår man upp sista bokstaven (*a* i *dockorna*) med latmannahashning och går igenom dom regler som har den enda bokstaven som ingångssuffix. Sedan binärsöker man efter näst sista bokstaven (*n*) bland dom ingångssuffix som slutar med *a* och går igenom dom regler som har *na* som ingångssuffix. Därefter söker vi efter nästnäst sista bokstaven (*r*) bland dom ingångssuffix som slutar med *na* och går igenom dom regler som har *rna* som ingångssuffix och så vidare.

På detta sätt blir det färre och färre ingångssuffix att binärsöka bland. □

Lösning till Grafsökning – egen övning!

Lösning till Topologisk sortering av DAG
Om man tittar på i vilken ordning djupetförstsökningen passerar hörnen på tillbakavägen i grafen (det vill säga i vilken ordning hörnen färdigbehandlas av sökningsproceduren) så ser man att det är precis omvänd topologisk ordning. Om vi vill sortera hörnen topologiskt behöver vi alltså bara lägga till en sats `Push(u)` sist i proceduren `DFSVISIT(u)`. När djupetförstsökningen är genomförd kan vi poppa hörnen ett i taget från stacken, och då kommer dom topologiskt sorterade. □

Lösning till Klick
Ledning: studera komplementgrafen (som har kanter bara där ursprungsgrafen inte har kanter). □

Lösning till Kantmatrisprodukt
Diagonalelementet (i, i) anger hur många kanter som har sin ändpunkt i hörn i. Ickediagonalelementet (i, j) är det negerade antalet kanter som går mellan hörnen i och j. □

Lösning till Bipartithet
Använd djupetförstsökning men färga hörnen alternerande med rött och grönt. Om en kant mellan två hörn av samma färg upptäcks är grafen inte bipartit. □

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 3

Dekomposition och dynamisk programmering

Max och min med dekomposition I vektorn $v[1..n]$ ligger n tal. Konstruera en dekompositionsalgoritm som tar reda på det största och det minsta talet i v. Algoritmen ska använda högst $\lceil 3n/2 \rceil - 2$ jämförelser mellan v-element. Antalet tal i v behöver inte vara en tvåpotens.

Matrismultiplikation Strassens algoritm multiplicerar två $n \times n$-matriser i tid $O(n^{2.808})$ genom dekomposition i 2×2-blockmatriser. Anledningen till att Strassens algoritm går snabbare än $O(n^3)$ är att den gör sju multiplikationer istället för åtta för att bilda produktmatrisen.

En annan idé är att göra dekomposition i 3×3-blockmatriser istället. Viggo försökte för ett par år sedan hitta det minimala antalet multiplikationer som krävs för att multiplicera två 3×3-matriser. Han lyckades nästan komma fram till 22 multiplikationer. Om han hade lyckats, vilken tidskomplexitet hade det gett för multiplikation av två $n \times n$-matriser?

Majoritet med dekomposition Indata är i denna uppgift en array A med n element. Konstruera och analysera en algoritm som tar reda på om något element i arrayen A är i majoritet, det vill säga förekommer mer än $n/2$ gånger, och i så fall returnerar det. Algoritmen ska vara en dekompositionsalgoritm och ha tidskomplexiteten $O(n \log n)$. Enda tillåtna jämförelseoperationen på element i A är $=$. Det finns alltså ingen ordningsrelation mellan elementen.

Mobil Konstruera en algoritm som balanserar en mobil! Mobilen beskrivs som ett binärträd. Löven motsvarar kulor som hänger underst i mobilen. Inre noder motsvarar tunna raka pinnar i vars ändar är fastknutet trådar som sönerna hänger i. Indata är ett binärträd med vikter i noderna. Vikten i varje löv är motsvarande kulas massa. Vikten i varje inre nod är motsvarande pinnes längd i cm. Resultatet av algoritmen ska vara samma binärträd men där det i varje inre nod också står hur många centimeter från vänstra ändpunkten tråden som fäster pinnen i ovanförliggande pinne ska sitta för att mobilen ska bli balanserad.

Talföljder Anta att du har fått rekursionen för en talföljd presenterad för dig. Skriv pseudokoden för en dynamisk programmeringsalgoritm för att beräkna S_n om

a)

$$S_n = \begin{cases} 1 & \text{om } n = 0, \\ 2(S_{n-1}) + 1 & \text{annars.} \end{cases}$$

b)

$$S_n = \begin{cases} 4 & \text{om } n = 1, \\ 5 & \text{om } n = 2, \\ \max(S_{n-1}, S_{n-2}, S_{n-1} - S_{n-2} + 7) & \text{annars.} \end{cases}$$

c)

$$S_n = \begin{cases} 4 & \text{om } n = 1, \\ 5 & \text{om } n = 2, \\ \max(S_{n-2}, S_{n-1} - S_{n-2} + 7) & \text{annars.} \end{cases}$$

Skriv ned de 7 första talen. Vad behöver man spara under beräkningens gång?

Generaliserade talföljder

a) Tvådimensionell rekursion utan indata

Givet följande rekursion, konstruera en algoritm som beräknar $M[i,j]$ med dynamisk programmering. Argumentera för att algoritmens beräkningsordning fungerar.

$$M[i,j] = \begin{cases} 0 & \text{om } i = 0 \text{ eller } j = 0, \\ M[i-1,j-1] + i & \text{annars.} \end{cases}$$

b) 2D-rekursion med indata

Givet två strängar a och b av längd n respektive m, låt a_i vara tecknet på position i i sträng a och på motsvarande sätt b_j vara tecknet på position j i b, hitta en beräkningsordning och skriv en algoritm som utför följande rekursion med hjälp av dynamisk programmering.

$$M[i,j] = \begin{cases} 0 & \text{om } i = 0 \text{ eller } j = 0, \\ M[i-1,j-1] + 1 & \text{om } a_i = b_j, \\ 0 & \text{annars.} \end{cases}$$

Lösningar

Lösning till Max och min med dekomposition
När man bara har två tal räcker det med en enda jämförelse för att både hitta det största och det minsta talet.

```
MinMax(v,i,j)=
  if i=j then return (v[i],v[i])
  else if i+1=j then
     if v[i]<v[j] then return (v[i],v[j])
     else return (v[j],v[i])
  else
     m:=Floor((j-i)/2)
     if Odd(m) then m:=m+1;
     (min1,max1):=MinMax(v,i,i+m-1);
     (min2,max2):=MinMax(v,i+m,j);
     min:=(if min1<min2 then min1 else min2);
     max:=(if max1>max2 then max1 else max2);
     return (min,max);
```

Beräkningsträdet kommer att få $\lceil n/2 \rceil$ löv och $\lceil n/2 \rceil - 1$ inre noder. Om n är jämnt är alla $n/2$ löv tvåelementsföljder och gör därför en jämförelse. Om n är udda är ett löv (det högraste) ett enstaka element som inte kräver någon jämförelse. I löven görs alltså $\lfloor n/2 \rfloor$ jämförelser. I varje inre nod görs två jämförelser, alltså sammanlagt $2\lceil n/2 \rceil - 2$ stycken. Totalt får vi $\lfloor n/2 \rfloor + 2\lceil n/2 \rceil - 2 = \lceil 3n/2 \rceil - 2$ stycken jämförelser.

Det går faktiskt att visa att problemet inte kan lösas med färre jämförelser. □

Lösning till Matrismultiplikation
Rekursionsekvationen blir $T(n) = 22 \cdot T(n/3) + O(n^2)$. Mästarsatsen ger lösningen $T(n) = O(n^{\log_3 22}) = O(n^{2.814})$. □

Lösning till Majoritet med dekomposition
Om det finns ett majoritetselement måste det vara i majoritet i åtminstone ena halvan av arrayen. Rekursiv tanke: Kolla majoritet rekursivt i vänstra och högra halvan och räkna sedan hur många gånger halvarraysmajoritetselementen förekommer i hela arrayen. Om något element är i total majoritet returneras det.

```
Majority(A[1..n]) =
    if n = 1 then return A[1]
    mid← ⌈(n + 1)/2⌉
    majInLeft← Majority(A[1..mid−1])
    majInRight← Majority(A[mid..n])
    if majInleft=majInRight then return majInLeft
    noOfMajInLeft← 0
    noOfMajInRight← 0
    for i ← 1  to n do
       if A[i] =majInLeft then noOfMajInLeft←noOfMajInLeft+1
       else if A[i] =majInRight then noOfMajInRight←noOfMajInRight+1
    if noOfMajInLeft≥ m then return majInLeft
    if noOfMajInRight≥ m then return majInRight
    else return NULL
```

Tidskomplexitet: Två rekursiva anrop med halva arrayen följt av efterarbetet $O(n)$ ger med mästarsatsens hjälp tidskomplexiteten $O(n \log n)$. □

Lösning till Mobil
Betrakta en pinne av längd l i mobilen. Anta att vikten v hänger i vänstra änden och vikten w i den högra. Låt x vara avståndet från pinnens vänstra ände till tråden som fäster pinnen i ovanförliggande pinne. För att momenten ska ta ut varandra krävs enligt mekaniken att $xv = (l - x)w$, det vill säga att $x = lw/(v + w)$. Vi kan nu beräkna x för varje pinne rekursivt nerifrån och upp i trädet. Låt den rekursiva funktionen returnera vikten av mobilen som hänger i den aktuella pinnen. Algoritmen tar linjär tid.

```
Balance(p)=
        if p.left =NIL then return p.num
        left ← Balance(p.left)
        right ← Balance(p.right)
        p.x ← p.num · right/(left + right)
        return left + right
```

□

Lösning till Talföljder
a) 1, 3, 7, 15, 31, 63, 127

För att beräkna ett tal behöver man bara känna till talet innan. Beräkningsordningen blir från basfallet och vidare ”uppåt”.

```
Talfoljda(n) =
    v = 1
    for i = 1 to n do
      v = 2*v+1
    return v
```

b) 4, 5, 8, 10, 10, 10, 10

För att beräkna ett tal behöver man känna till de två föregående talen. Beräkningsordningen blir från basfallet och vidare ”uppåt”.

```
Talfoljdb(n) =
  if n = 1 then return 4
  elif n = 2 then return 5
  else
    v0 = 4
    v1 = 5
    for i = 2 to n do
      v = max(v1, v0, v1-v0+7)
      v0 = v1
      v1 = v
    return v
```

c) 4, 5, 8, 10, 9, 10, 9

För att beräkna ett tal behöver man känna till de två föregående talen. Beräkningsordningen blir från basfallet och vidare ”uppåt”.

```
Talfoljdc(n) =
  if n = 1 then return 4
  elif n = 2 then return 5
  else
    v0 = 4
    v1 = 5
    for i = 2 to n do
      v = max(v0, v1-v0+7)
      v0 = v1
      v1 = v
    return v
```

□

Lösning till Generaliserade talföljder

a) 2D-rekursion utan indata

Eftersom vi inte vet vad vi letar efter, låter vi algoritmen returnera hela M. Det är inte vad man typiskt skulle göra om algoritmen skulle användas för att svara på någon specifik fråga. Oavsett vad frågan skulle vara, är det (första) vi behöver kunna göra just att beräkna M. När hela matrisen är ifylld kan vi svara på olika frågor om den, som vilket det största värdet i matrisen är, eller hur många gånger det största värdet förekommer.

Vi måste börja med ett basfall, men om vi tänker oss en matris M som ska fyllas i med rätt värden, så har hela översta raden (där $i = 0$) och hela vänstra kolumnen (där $j = 0$) värdet 0, som inte beräknas rekursivt. En typisk beräkningsordning för ett program skulle vara en rad i taget uppifrån eller en kolumn i taget från vänster till höger. Vi väljer rad för rad, och sätter värdena på första raden. Därefter loopar vi över alla andra rader och sätter värdena kolumn för kolumn. Beräkningsordningen fungerar, eftersom den information vi behöver ha för att beräkna ett nytt värde utom basfallen bara är vad som stått på tidigare rader, samt indexet för den rad vi befinner oss på. Vi kommer att kunna beräkna hela M. Här är vår algoritm:

for $j \leftarrow 0$ **to** n
  $M[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m

```
    M[i, 0] ← 0
    for j ← 1 to n
        M[i, j] ← M[i − 1, j − 1] + i
return M
```

Tiden domineras av den nästlade for-slingan och är alltså $\Theta(nm)$.

b) 2D-rekursion med indata
Vi börjar med basfallen. Alla element på översta raden och alla i kolumnen längst till vänster uppfyller villkoret för basfall, och kan därför sättas till 0. Vi väljer att börja med första raden. Därefter beräknar vi värdena på alla följande rader uppifrån och ner, kolumn för kolumn. Enligt rekursionen är värdet 0 om $a_i \neq b_j$, och annars beräknas det med hjälp av tidigare uträknade värden. Vi kan alltid titta på föregående rad $(i - 1)$, eftersom vi beräknar en rad i taget. På den raden kan vi titta på vilka element vi vill, eftersom hela raden är ifylld, speciellt är det tillåtet att titta på det i kolumn $j - 1$. Beräkningsordningen fungerar alltså. Algoritmen kan se ut så här:

```
for j ← 0 to n
    M[0, j] ← 0
for i ← 1 to m
    M[i, 0] ← 0
    for j ← 1 to n
        if a_i = b_j then
            M[i, j] ← M[i − 1, j − 1] + 1
        else M[i, j] ← 0
return M
```

Tiden blir precis som i det förra exemplet $\Theta(nm)$. □

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 4

Dynamisk programmering

På denna övning är det också **inlämning av skriftliga lösningar av teoriuppgifterna till labb 2** och muntlig redovisning av teoriuppgifterna.

Träskvandring Tina ska gå genom ett träsk som representeras av ett $n \times n$-rutmönster från vänsterkanten till högerkanten. I varje steg kan hon gå ett steg rakt till höger, snett uppåt höger eller snett nedåt höger. Det är olika jobbigt att gå på olika ställen i träsket. Att hamna på ruta (i, j) kostar arbetet $A[i, j]$, som är ett positivt heltal.

Beskriv en algoritm som hittar det minimala arbetet att ta sig igenom träsket i följande olika fall.

a) Anta att Tina får starta var som helst på vänsterkanten och får sluta var som helst på högerkanten.

b) Anta att Tina alltid startar i ruta $(n/2, 1)$.

c) Anta att Tina dessutom måste sluta i ruta $(n/2, n)$.

d) Skriv ut den minst jobbiga väg Tina kan ta genom träsket i c).

Kombinera mynt och sedlar Givet en (konstant) uppsättning mynt- och sedelslag med valörerna $v_1, \ldots, v_k$ kronor och ett maximalt belopp n. (Alla tal i indata är positiva heltal.) Formulera en rekursion för på hur många sätt man kan bilda varje summa mellan 0 och n kronor med hjälp av denna uppsättning mynt och sedlar.

Längsta gemensamma delsträng Strängarna `ALGORITM` och `PLÅGORIS` har den gemensamma delsträngen `GORI`. Den *längsta gemensamma delsträngen* hos dessa strängar har alltså längd 4. I en delsträng måste tecknen ligga i en sammanhängande följd.

Konstruera en effektiv algoritm som givet två strängar $a_1 a_2 \ldots a_m$ och $b_1 b_2 \ldots b_n$ beräknar och returnerar längden hos den längsta gemensamma delsträngen. Algoritmen ska bygga på dynamisk programmering och gå i tid $O(nm)$.

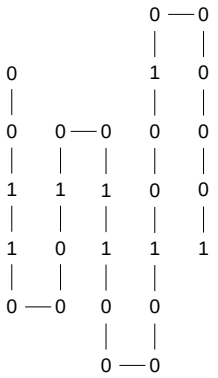
Här följer två roliga men mer komplicerade exempel som vi nog inte hinner med på övningen:

Proteinvikning Ett protein är en lång kedja av aminosyror. Proteinkedjan är inte rak som en pinne utan hopvikt på ett intrikat sätt som minimerar den potentiella energin. Man vill väldigt gärna kunna räkna ut hur ett protein kommer att vika sig. I denna uppgift ska vi därför studera en enkel modell av proteinvikning där aminosyrorna är antingen *hydrofoba* eller *hydrofila*. Hydrofoba aminosyror tenderar att klumpa ihop sig.

För enkelhets skull ser vi proteinet som en binär sträng där ettor motsvarar hydrofoba aminosyror och nollor hydrofila aminosyror. Strängen (proteinet) ska sedan vikas i ett tvådimensionellt kvadratiskt gitter. Målet är att få dom hydrofoba aminosyrorna att klumpa ihop sig, det vill säga att få så många ettor som möjligt att ligga nära varandra. Vi har alltså ett optimeringsproblem där målfunktionen är antalet par av ettor som ligger intill varandra i gittret (lodrätt eller vågrätt) utan att vara intill varandra i strängen.

Du ska konstruera en algoritm som med hjälp av dynamisk programmering konstruerar en optimal *dragspelsvikning* av en given proteinsträng av längd n. En dragspelsvikning är en vikning där strängen först går en sträcka rakt nedåt, sedan en sträcka rakt uppåt, sedan en sträcka rakt nedåt, och så vidare. I en sådan vikning kan man notera att lodräta par av intilliggande ettor alltid kommer i följd i strängen, så det är bara vågräta par av ettor som bidrar till målfunktionen.

I följande figur är strängen 00110001001100001001000001 dragspelsvikt på ett sådant sätt att målfunktionen blir 4.



Definition av problemet PROTEINDRAGSPELSVIKNING:

INMATNING: En binär sträng med n tecken.

PROBLEM: Hitta den dragspelsvikning av indatasträngen som ger det största värdet på målfunktionen, alltså det största antalet par av ettor som ligger bredvid varandra men inte direkt efter varandra i strängen.

Konstruera och analysera tidskomplexiteten för en algoritm som löser proteindragspelsvikningsproblemet med dynamisk programmering.

Du får gärna använda dig av nedanstående algoritm, som beräknar antalet par ettor i ett varv (dvs mellan två sträckor) i en dragspelsvikning som ligger bredvid varandra (men inte direkt efter varandra i strängen). Anta att proteinet lagras i en array `p[1..n]`. Parametrarna `a` och `b` anger index i arrayen för den första sträckans ändpunkter. Parametern `c` anger index för den andra sträckans slutpunkt. Se figuren nedanför till höger.

```
profit(a,b,c) =
  shortest←min(b-a,c-(b+1));
  s←0;
  for i←1 to shortest do
    if p[b-i]=1 and p[b+1+i]=1 then
      s←s+1;
  return s;
```

Not: Proteinvikningsproblemet är ett viktigt algoritmiskt problem som studeras i bioinformatiken. Det behandlas tillsammans med många andra problem med biologisk anknytning i den valfria kursen *Algoritmisk bioinformatik* som går i period 4 varje år.

Analysator för kontextfri grammatik En *kontextfri grammatik* brukar användas för att beskriva syntax för bland annat programspråk. En kontextfri grammatik i *Chomskynormalform* beskrivs av

- en mängd slutsymboler T (som brukar skrivas med små bokstäver),
- en mängd ickeslutsymboler N (som brukar skrivas med stora bokstäver),

- startsymbolen S (en av ickeslutsymbolerna i mängden N),
- en mängd omskrivningsregler som antingen är på formen $A \to BC$ eller $A \to a$, där $A, B, C \in N$ och $a \in T$.

Om $A \in N$ så definieras $\mathcal{L}(A)$ genom

$$\mathcal{L}(A) = \{bc : b \in \mathcal{L}(B) \text{ och } c \in \mathcal{L}(C) \text{ där } A \to BC\} \cup \{a : A \to a\}.$$

Språket som genereras av grammatiken definieras nu som $\mathcal{L}(S)$, vilket alltså är alla strängar av slutsymboler som kan bildas med omskrivningskedjor som börjar med startsymbolen S.

Exempel: Betrakta grammatiken med $T = \{a, b\}$, $N = \{S, A, B, R\}$, startsymbolen S och reglerna $S \to AR$, $S \to AB$, $A \to a$, $B \to b$, $R \to SB$. Vi kan se att strängen *aabb* tillhör språket som genereras av grammatiken med hjälp av följande kedja av omskrivningar:

$$S \to AR \to aR \to aSB \to aSb \to aABb \to aaBb \to aabb.$$

I själva verket kan man visa att det språk som genereras av grammatiken är precis alla strängar som består av k stycken a följt av k stycken b där k är ett positivt heltal.

Din uppgift är att *konstruera* och *analysera* en effektiv algoritm som avgör ifall en sträng tillhör det språk som genereras av en grammatik. Indata är alltså en kontextfri grammatik på Chomskynormalform samt en sträng av slutsymboler. Utdata är sant eller falskt beroende på om strängen kunde genereras av grammatiken eller inte. Ange tidskomplexiteten för din algoritm uttryckt i antalet regler m i grammatiken och längden n av strängen.
Mer om grammatiker kan man läsa i kursen *Automater och språk*.

Lösningar

Lösning till Träskvandring
a) Vi låter $W[i, j]$ vara det minimala arbetet som krävs för att komma till ruta (i, j). Tina får starta var som helst på vänsterkanten. Basfallen blir att hamna på en första ruta (från någonstans utanför träsket). Det kostar endast $A[i, 0]$ för varje i. Vi kommer att vilja uttrycka hur jobbigt det är att komma till ruta i, j från ruta $i, j-1$, vilket är arbetet $A[i, j]$ plus det det kostade att komma till ruta $i, j-1$. Vi vill hitta det *minst jobbiga* sättet att komma till varje ruta i, j, så i allmänhet finns det tre ställen Tina kan ha kommit ifrån, alla i kolumnen $j-1$, och vi måste jämföra hur jobbigt det är att komma från vart och ett och välja det minst jobbiga. Nu verkar det rimligt att börja beräkna kolumnvis hur jobbigt det är att komma till varje ruta. Att komma till en ruta i ovankanten uppifrån går inte, så det värdet sätts till oändligheten för att vår jämförelse inte ska premiera den riktningen. På samma sätt hanterar vi underkanten av matrisen. Därför kommer vi att ha följande rekursion: $W[i, j] = \min(W[i-1, j-1], W[i, j-1], W[i+1, j-1]) + A[i, j]$. Vi beräknar alla värdena i den nya matrisen W.

Basfallen:
for $j \leftarrow 1$ **to** n
  $W[1, j] = A[1, j]$

Fyll i matrisen:
for $j \leftarrow 2$ **to** n
  for $i \leftarrow 1$ **to** n
    $fromabove = W[i-1, j-1]$ **if** $i > 1$, **else** ∞
    $fromsame = W[i, j-1]$
    $frombelow = W[i+1, j-1]$ **if** $i < n$, **else** ∞
    $W[i, j] = \min(fromabove, fromsame, frombelow) + A[i, j]$

Tina fick lämna träsket från vilken ruta som helst på högerkanten. Därför kan vi gå igenom sista kolumnen i vår matris och leta efter det minsta värdet där.

Hitta svaret:
$opt \leftarrow \infty$
for $i \leftarrow 1$ **to** n
  if $W[i,n] < opt$ **then**
    $opt \leftarrow W[i,n]$
return opt

Om vi sätter ihop dessa tre steg får vi en dynamisk programmering-algoritm som beräknar det minst jobbiga sättet att ta sig från vänster sida av träsket till höger sida.

b) Om Tina alltid startar i ruta $(n/2, 1)$ har vi lite annorlunda basfall. Eftersom Tina aldrig får förflytta sig rakt uppåt eller rakt nedåt, är alla andra rutor i den vänstra kolumnen omöjliga att nå, dvs arbetet för att ta sig dit är oändligt. Rekursionen blir likadan.

c) Om Tina dessutom alltid slutar i rutan $(n/2, n)$, kommer vi när vi är ute efter svaret bara att vara intresserade av $W[n/2, n]$, oavsett vad som står i de andra rutorna i sista kolumnen. Det enda som ändras är vad vi ska returnera, där sökandet efter *opt* ersätts med att returnera $W[n/2, n]$. Rekursionen fungerar likadant.

d) Nu vill vi konstruera en bästa väg. Vi kan antingen spara varifrån vi kom i varje steg, dvs låta W innehålla både ett jobbighetsvärde och en pekare till rutan man kommit ifrån, eller ta vår slutruta, och subtrahera dess värde från A-matrisen, och se vilken av rutorna som vi kan ha kommit ifrån som hade detta värde, och sedan upprepa processen tills vi kommer till startrutan. (Om det är mer än en väg till en ruta som var lika jobbig, gör det inget. Vi vill bara ha *en* minst jobbig väg genom träsket, och det kan finnas flera som är lika jobbiga.)

Eftersom vi började med värdet i slutrutan, och tog reda på varifrån vi kunde ha kommit för att få det värdet där, och eftersom vi upprepar detta för varje ruta i stigen, kommer vi att nå startrutan. (Annars kan vi inte ha något värde i slutrutan.) I varje ruta står nämligen hur jobbigt det är att komma dit från startrutan, den enklaste vägen, och om vi subtraherar bort det som hörde till just denna ruta, återstår hur jobbigt det var att komma till förra rutan. Det värdet måste stå i minst en av rutorna som vi kunde ha kommit från, och det betyder att vi kunde nå den rutan till det priset. Vi kommer alltid att nå startrutan utan att passera någon onåbar ruta, eftersom $W[i,j] - A[i,j] \neq \infty$ såvida inte $W[i,j] = \infty$. Eftersom vi tittar på stigen baklänges, kan vi spara värdena i en stack för att skriva ut dem i rätt ordning. □

Lösning till Kombinera mynt och sedlar

Låt $N[b,j]$ vara antalet sätt att bilda beloppet b med valörerna $v_1, \ldots, v_j$ kronor.

N kan till exempel definieras rekursivt på följande sätt:

$$N[b,j] = \begin{cases} 1 & \text{om } b = 0 \text{ eller } (j = 1) \wedge (b \mod v_1 = 0), \\ 0 & \text{om } j = 1 \text{ och } b \mod v_1 \neq 0, \\ \sum_{i=0}^{\lfloor b/v_j \rfloor} N[b - i \cdot v_j, j-1] & \text{om } 0 < b \leq n \end{cases}$$

En annan möjlig rekursiv formulering bygger på tanken att en kombination av valörerna $v_1, \ldots, v_j$ antingen innehåller valören v_j eller bara innehåller valörerna $v_1, \ldots, v_{j-1}$:

$$N[b,j] = \begin{cases} 1 & \text{om } b = 0, \\ 0 & \text{om } j = 0 \text{ och } b > 0, \\ N[b, j-1] & \text{om } 0 < b < v_j \text{ och } j > 0 \\ N[b - v_j, j] + N[b, j-1] & \text{om } v_j < b \leq n \text{ och } j > 0 \end{cases}$$

□

Lösning till Längsta gemensamma delsträng
För varje par av tecken ifrån var sin sträng, låt $M[i,j]$ vara antal bokstäver till vänster om (och inklusive) a_i som överensstämmer med lika många bokstäver till vänster (och inklusive) b_j. Längden av den längsta gemensamma strängen är då det största talet i matrisen M.

M kan definieras rekursivt på följande sätt:

$$M[i,j] = \begin{cases} 0 & \text{om } i = 0 \text{ eller } j = 0, \\ M[i-1, j-1] + 1 & \text{om } a_i = b_j, \\ 0 & \text{annars.} \end{cases}$$

Här kan vi se att den rekursion vi kommit fram till är samma som vi tittade på förra övningen. Då kom vi fram till följande algoritm:

for $j \leftarrow 0$ **to** n
  $M[0,j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
  $M[i,0] \leftarrow 0$
  for $j \leftarrow 1$ **to** n
    if $a_i = b_j$ **then**
      $M[i,j] \leftarrow M[i-1, j-1] + 1$
    else $M[i,j] \leftarrow 0$
return M

Det vi behöver göra utöver detta, för att svara på frågan i denna uppgift, är att hitta det största talet i M. Vi kan förstås först beräkna M och sedan gå igenom matrisen och jämföra värden, men det går bra att utöka algoritmen så att den jämför värden under beräkningarnas gång. Då slipper vi få hela matrisen som utdata. Följande algoritm beräknar hela M och returnerar det största talet i M.

$max \leftarrow 0$
for $j \leftarrow 0$ **to** n
  $M[0,j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
  $M[i,0] \leftarrow 0$
  for $j \leftarrow 1$ **to** n
    if $a_i = b_j$ **then**
      $M[i,j] \leftarrow M[i-1, j-1] + 1$
      if $M[i,j] > max$ **then** $max \leftarrow M[i,j]$
    else $M[i,j] \leftarrow 0$
return max

Tiden domineras även nu av den nästlade for-slingan och är alltså $\Theta(nm)$. □

Lösning till Proteinvikning
Låt $q_{a,b}$ vara det maximala värdet på målfunktionen man kan få för en vikning av delen $p[a..n]$ av proteinet, där den första sträckan i vikningen har ändpunkterna a och b. Vi kan uttrycka $q_{a,b}$ rekursivt på följande sätt:

$$q_{a,b} = \max_{b+1<c\leq n} (\texttt{profit}(a,b,c) + q_{b+1,c}).$$

Basfallen är $q_{a,n} = 0$ för $1 \leq a < n$. Svaret hittar vi sedan som $\max_{1<b\leq n} q_{1,b}$.
Nu gäller det bara att beräkna $q_{a,b}$ enligt dessa formler i rätt ordning:

```
for a←1 to n-1 do q[a,n]←0;
for b←n-1 downto 2 do
```

```
  for a←1 to b-1 do
    t←-1;
    for c←b+2 to n do
      v←profit(a,b,c)+q[b+1,c];
      if v>t then t←v;
    q[a,b]←t;
max←0;
for b←2 to n do
  if q[1,b]>max then max←q[1,b];
return max;
```

Eftersom vi som mest har tre nästlade `for`-slingor och ett anrop till `profit` tar tid $O(n)$ blir tidskomplexiteten uppenbarligen $O(n^4)$. □

Lösning till Analysator för kontextfri grammatik

Vi använder dynamisk programmering ungefär som i problemet där man letar efter optimal matriskedjemultiplikationsordning. Här ska vi istället bestämma i vilken ordning och på vilken delsträng reglerna ska tillämpas.

Indata är en uppsättning regler R och en vektor `w[1..n]` som alltså indexeras från 1 till n. Låt oss bygga upp en matris `M[1..n,1..n]` där elementet `M[i,j]` anger dom ickeslutsymboler från vilka man med hjälp av kedjor av omskrivningar kan härleda delsträngen `w[i..j]`.

Rekursiv definition av `M[i,j]`:

$$\texttt{M}[i,j] = \begin{cases} \{X : (X \to \texttt{w}[i]) \in R\} & \text{om } i = j \\ \{X : (X \to AB) \in R \wedge \exists k : A \in \texttt{M}[i,k-1] \wedge B \in \texttt{M}[k,j]\} & \text{om } i < j \end{cases}$$

Eftersom varje position i matrisen är en mängd av ickeslutsymboler så måste vi välja en lämplig datastruktur också för detta. Låt oss representera en mängd av ickeslutsymboler som en bitvektor som indexeras med ickeslutsymboler. 1 betyder att symbolen är med i mängden och 0 att den inte är med i mängden. Exempel: Om `M[i,j][`B`]`$= 1$ så är ickeslutsymbolen B med i mängden `M[i,j]`, vilket betyder att det finns en kedja av omskrivningsregler från B till delsträngen `w[i..j]`.

Algoritm som beräknar matrisen `M[i,j]` och returnerar sant ifall strängen tillhör språket som genereras av grammatiken:

```
for i←1 to n do
  M[i,i]←0; /* alla bitar nollställs */
  för varje regel X →w[i] do
    M[i,i][X]←1;
for len←2 to n do
  for i←1 to n-len+1 do
    j←i+len-1;
    M[i,j]←0;
    for k←i+1 to j do
      för varje regel X → AB do
        if M[i,k-1][A]=1 and M[k,j][B]=1 then
          M[i,j][X]←1;
return M[1,n][S]= 1;
```

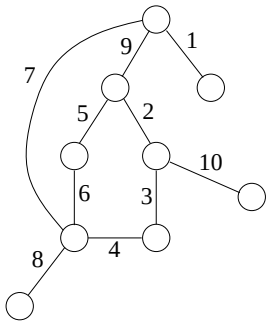
Tid: $O(n^3m)$. Minne: $O(n^2m)$ (eftersom m är en övre gräns för antalet ickeslutsymboler). □

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 5

Grafalgoritmer och undre gränser

Spännande träd Visa hur ett minimalt spännande träd hittas i följande graf med både Prims och Kruskals algoritmer.



Förändrat flöde

a) Beskriv en effektiv algoritm som hittar ett nytt maximalt flöde om kapaciteten längs en viss kant *ökar* med en enhet.

   Algoritmens tidskomplexitet ska vara linjär, alltså $O(|V| + |E|)$.

b) Beskriv en effektiv algoritm som hittar ett nytt maximalt flöde om kapaciteten längs en viss kant *minskar* med en enhet.

   Algoritmens tidskomplexitet ska vara linjär, alltså $O(|V| + |E|)$.

Eulercykel Givet en oriktad sammanhängande graf $G = \langle V, E \rangle$ där alla hörn har jämnt gradtal. Hitta en Eulercykel, dvs en sluten stig som passerar varje kant i E exakt en gång. Algoritmen ska gå i linjär tid.

Julklappsfördelning En pappa ska ge sina n barn var sin julklapp. Varje barn har skrivit en önskelista. Pappan vill ge varje barn en julklapp från barnets önskelista, men han vill inte ge samma julklapp till flera barn.

Konstruera och analysera en effektiv algoritm för detta problem. Du kan anta att det finns högst m saker på varje önskelista.

Undre gräns för sökning i sorterad array Binärsökning i en sorterad array med n tal tar som bekant tiden $O(\log n)$. Bevisa att $\Omega(\log n)$ också är en undre gräns för antalet jämförelser som krävs för detta problem (i värsta fallet).

Laga julgransbelysning Jonas julgransbelysning med n seriellt kopplade lampor lyser inte. Det räcker att en lampa är trasig för att belysningen inte ska lysa. Han har skaffat n nya lampor som säkert fungerar. Han vill inte ersätta några hela lampor med nya, för det är dumt att slösa på nya lampor. Han vill naturligtvis göra så få lampprovningar som möjligt.

Konstruera och analysera en effektiv algoritm för hur Jonas ska få belysningen att fungera med så få lampprovningar (ur- och iskruvningar) som möjligt. Analysera värstafalletkomplexiteten för algoritmen exakt (inte bara ordoklass). En urskruvning följt av en iskruvning av en lampa tar tiden 1. Ju snabbare algoritm desto bättre.

Ge också en undre gräns för värstafalletkomplexiteten som matchar din algoritm.

Lösningar och ledningar

Lösning till Spännande träd – egen övning!

Lösning till Förändrat flöde

a) Anta att kanten från u till v får sin kapacitet ökad med ett. Eftersom tidskomplexiteten ska vara linjär kan vi inte beräkna en helt ny lösning utan måste utnyttja lösningen på det tidigare flödesproblemet. Låt Φ vara denna lösning (Vad består Φ av?) och gör bara en ny iteration i Ford-Fulkersons algoritm med den ändrade grafen: I restflödesgrafen ökas kapaciteten i kanten (u, v) med ett. Gör en grafsökning (i tid $O(|V| + |E|)$) för att se om det nu finns någon stig i restflödesgrafen längs vilken flödet kan öka. Om det finns det måste det vara ett flöde av storleken 1 (eftersom alla flöden är heltalsflöden). Om det inte finns något flöde i restflödesgrafen är Φ fortfarande det maximala flödet.

b) Anta att kanten från u till v får sin kapacitet minskad med ett. Om det tidigare maximala flödet Φ inte utnyttjade hela kapaciteten i (u, v) så förändras inte flödet alls. I annat fall måste vi uppdatera flödet på följande sätt:

Eftersom det kommer in en enhet större flöde till u än som går ut och det kommer in en enhet mindre flöde till v än det går ut så måste vi försöka leda en enhets flöde från u till v någon annan väg. Sök alltså i restflödesgrafen efter en stig från u till v längs vilken flödet kan öka med ett. Detta görs med en grafsökning i tid $O(|V| + |E|)$. Om det finns en sån stig uppdaterar vi Φ med det flödet.

Om det inte finns en sån stig måste vi minska flödet från s till u och från v till t med en enhet. Det gör vi genom att hitta en stig från u till s i restflödesgrafen längs vilken flödet kan öka med ett och en stig från t till v i restflödesgrafen längs vilken flödet kan öka med ett. (Det måste finnas såna stigar eftersom vi hade ett flöde från s till t via (u, v).) Uppdatera sedan Φ med dessa två flöden.

□

Lösning till Eulercykel
Idén är att börja i ett hörn v och söka sig igenom grafen tills v nås igen. Sökstigen P kommer då antingen att ha besökt alla kanter eller så återstår det, om vi tar bort P, en eller flera sammanhängande komponenter $G_1, G_2, \ldots, G_n$. I det senare fallet kan vi göra om samma sorts sökning för varje komponent så vi får en Eulercykel för varje G_i, och sedan sätter vi in alla dessa Eulercykler i P så att vi får en total Eulercykel.

För att kunna genomföra detta i linjär tid så hittar vi på en algoritm som använder två spelare P_1 och P_2 som tillsammans går igenom grafen. Båda spelarna startar i v. Spelare P_1 skapar stigen P genom att gå längs kanter hur som helst. Han märker varje kant han passerar och stannar när han kommer tillbaka till v igen. Sedan följer P_2 efter längs P, men varje gång han kommer till ett nytt hörn u så kollar han om alla kanter från u är märkta. I så fall fortsätter han längs P. Om det finns omärkta kanter (det finns alltid ett jämnt antal) så skickar han ut P_1 för att hitta en ny stig som börjar och slutar i u. P_2 går sedan denna nya stig innan han fortsätter från u. Efter att ha upprepat detta kommer P_2 till slut att ha gått längs varje kant exakt en gång och därför skapat en Eulercykel. P_1 har inte heller gått längs samma kant mer än en gång så totala körtiden blir linjär.

I algoritmen nedan kallas P_1 för $PathFinder$ och P_2 för $Straggler$.

```
EulerCycle(G) =
    cycle ← {1}
    choose edge (1, t)
    mark (1, t)
    path ← PathFinder(G, 1, t)
    Straggler(path)
    return cycle

PathFinder(G, start, cur) =
    append(cur, path)
    while cur ≠ start do
        choose unmarked edge (cur, v)
        mark (cur, v)
        append(v, path)
        cur ← v
    return path

Straggler(path) =
    while path ≠ ∅ do
        u ← next(path)
        append(u, cycle)
        for all edges (u, v) do
            if unmarked (u, v) then
                mark (u, v)
                p ← PathFinder(G, u, v)
                Straggler(p)
```

□

Lösning till Julklappsfördelning

Problemet är ett bipartit matchningsproblem. Det gäller att hitta en matchning i en graf där vänsterhörnen motsvaras av barn (n stycken hörn) och högerhörnen av saker. Det går en kant mellan ett barn och en sak om saken står med på barnets önskelista. Bipartit matchning kan lösas som ett flödesproblem i en graf med $O(nm)$ kanter. Eftersom flödet ökar med ett i varje varv i Ford-Fulkersons algoritm och matchningen (flödet) är högst n så går slingan högst n varv och komplexiteten blir $O(n^2m)$. □

Ledning till Undre gräns för sökning i sorterad array

Konstruera ett beslutsträd för en godtycklig algoritm som söker i en sorterad array. Undersök hur många löv trädet måste ha och kom fram till hur högt det då måste vara.

Lösning till Laga julgransbelysning
Numrera julgransbelysningens lampplatser från 1 till n.

byt ut alla lampor utom den på plats 1 mot nya
lägg dom gamla lamporna i en säck
$i \leftarrow 1$
while (lampor kvar i säcken) **do**
    if (belysningen lyser) **then**
        $i \leftarrow i + 1$
        skruva ur den nya lampan på plats i
    else
        skruva ur den gamla trasiga lampan på plats i
    ta en lampa från säcken och skruva i den på plats i
if not (belysningen lyser) **then**
    skruva ur den gamla trasiga lampan på plats i
    skruva i en ny lampa på plats i

Det är lätt att se att algoritmen är korrekt med hjälp av följande invariant för slingan (invarianten är sann i början av varje varv i slingan): Alla lampor på plats $< i$ är gamla och fungerar, lampan på plats i är gammal, alla lampor på plats $> i$ är nya.

I värsta fall är alla lampor trasiga i belysningen, och då gör algoritmen $(n-1)+(n-1)+1 = 2n-1$ lampbyten.

Att $2n-1$ också är en undre gräns kan man inse så här: Enda sättet att avgöra ifall en viss lampa i belysningen är hel eller trasig är att försäkra sig om att alla andra lampor är hela och då se om belysningen lyser. En tuff motståndare ser till att belysningen fortfarande inte lyser efter att $n-1$ nya lampor skruvats i helt enkelt genom att låta den enda återstående originallampan vara trasig. Alla $n-1$ bortskruvade originallampor måste sedan provas igen, för den tuffa motståndaren har ännu inte tillåtit att någon som helst information om dessa lampor har getts, och det kräver $n-1$ lampbyten till. Slutligen ser motståndaren till att just den sist provade originallampan är trasig, och då krävs det ytterligare ett lampbyte för att fixa den.

Den tuffa motståndaren behöver alltså bara två trasiga lampor för att få algoritmen att ta $2n-1$ lampbyten! □

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 6

Algoritmkonstruktion

På denna övning är det också **inlämning av skriftliga lösningar av teoriuppgifterna till labb 3** och muntlig redovisning av teoriuppgifterna.

Inuti eller utanför? Låt P vara en konvex n-hörnig polygon beskriven som en array av hörnen $p_1, p_2, \ldots p_n$ i cyklisk ordning. Konstruera en algoritm som talar om ifall en given punkt q är inuti P. Algoritmen ska gå i tid $O(\log n)$ i värsta fallet.

Sortering av små heltal Konstruera en algoritm som sorterar n heltal som alla ligger i intervallet $[1..n^3]$ i tid $O(n)$ med enhetskostnad.
Tips: tänk på räknesortering och radixsortering.

Hitta det saknade talet På en datafil ligger 999 999 999 tal, nämligen alla heltal mellan 1 och 1 000 000 000 utom ett. Vilket tal är det som saknas? Konstruera en algoritm som löser detta problem i konstant minne och linjär tid i en beräkningsmodell som har 32-bitsaritmetik (dvs kan räkna med tal av storlek mindre än $2^{31} = 2147483648$).

Komplex multiplikation Om man multiplicerar två komplexa tal $a + bi$ och $c + di$ på det vanliga sättet krävs fyra multiplikationer och två additioner av reella tal. Eftersom multiplikationer är dyrare än additioner (och subtraktioner) lönar det sig att minimera antalet multiplikationer om man ska räkna med stora tal. Hitta på en algoritm som bara använder tre multiplikationer (men fler additioner) för att multiplicera två komplexa tal.

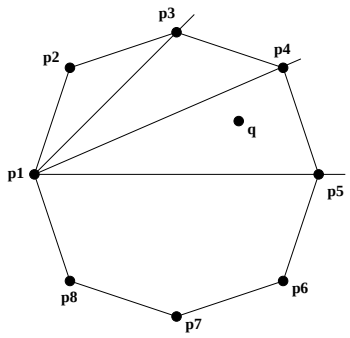
Tvådimensionell Fouriertransform Anta att du har tillgång till en FFT-implementation för det endimensionella fallet, men att du behöver transformera tvådimensionella data, till exempel transformera en bild given som en matris $(a_{ij})_{i,j\in\{0,\ldots,N-1\}}$ till ett tvådimensionellt frekvensspektrum. Hur bör du utnyttja FFT för detta? Vad blir tiden?

Definitionen av 2D-Fouriertransformen:

$$\hat{a}_{kl} = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a_{ij} w^{ik+jl}$$

där w är en lämplig enhetsrot ($w^N = 1$).

Binärträd med speglad struktur Två binära träd sägs ha *speglad struktur* om det ena är en spegelbild av det andra, det vill säga att om man byter vänster och höger överallt i det ena trädet så blir träden strukturekvivalenta. Konstruera och analysera tidskomplexiteten för en effektiv dekompositionsalgoritm som avgör ifall två binärträd har speglad struktur.



Figur 1: En konvex polygon och linjerna som algoritmen använder för att halvera den.

Partyproblemet Indata är en lista med n personer, ett heltal k och en lista med vilka personer som känner varandra. Du vill bjuda så många av personerna som möjligt på ditt party, men för att alla ska trivas vill du att varje inbjuden gäst ska känna minst k av dom övriga gästerna. Konstruera och analysera en algoritm som löser detta problem i linjär tid i indatas storlek.

Lösningar

Lösning till Inuti eller utanför?
Om polygonen inte hade varit konvex hade vi räknat antalet skärningar polygonen har med en linje från q till en punkt utanför P, vilket tar tid $O(n)$, men det har vi inte tid med här. Istället kommer vi att använda en intervallhalveringsliknande sökning för att utesluta hälften av (den återstående) polygonen i taget ända tills bara en triangel återstår. Därefter kan vi lätt (med ett konstant antal jämförelser) avgöra ifall q ligger i P. Se figur 1.

$InsideConvex(P, q, l, u) =$
  if $u = l + 1$ **then** $\quad /* \text{ en triangel } */$
    välj en punkt q' utanför triangeln p_1–p_l–p_u
    if linjen q–q' skär exakt en av kanterna i triangeln **then**
      return inuti
    else
      return utanför
  else
    $mid \leftarrow \lceil \frac{l+u}{2} \rceil$
    if q är på samma sida om linjen p_1–p_{mid} som p_{mid+1} **then**
      return $InsideConvex(P, q, mid, u)$
    else
      return $InsideConvex(P, q, l, mid)$

Algoritmen anropas med $InsideConvex(P, q, 2, n)$.

Om vi antar att $InsideConvex(P, q, 2, n)$ tar tid $T(n)$ så får vi rekursionsekvationen

$$T(n) = T\left(\frac{n}{2}\right) + c$$

som har lösningen $c \log n$. Tidskomplexiteten blir alltså $T(n) \in O(\log n)$. □

Lösning till Sortering av små heltal
Betrakta heltalen skrivna i bas n. Alla tal har då (högst) tre siffror, utom n^3 som skrivs som 1000 och kan hanteras separat. (Gå igenom alla tal, plocka ut alla förekomster av n^3 och lägg dom sist i den sorterade följden.) Lägg till inledande nollor på tal som är mindre än n^2. Gör nu radixsortering på talen. Eftersom det är tresiffriga tal blir det tre omgångar i radixsorteringen, där man i varje omgång ska sortera talen efter en siffra i intervallet $[0..n-1]$. En sån sortering görs i linjär tid med räknesortering (som i implementationen från föreläsning 16 är stabil).

Analys: Tre omgångar görs och varje omgång tar tid $O(n)$. Specialhanteringen av n^3 tar inte heller mer än $O(n)$. Totalt blir tiden alltså $O(n)$. □

Ledning till Hitta det saknade talet
Summera alla tal modulo till exempel 1 000 000 000.

Lösning till Komplex multiplikation
Egen övning. □

Lösning till Tvådimensionell Fouriertransform
Notera först att faktorn w^{ik} kan flyttas ur den innersta summan. Om vi nu ser den innersta summan som en endimensionell Fouriertransform $\hat{b}_{il}$ så är $\hat{a}_{kl}$ en endimensionell transform av $\hat{b}_{il}$.

$$\hat{b}_{il} = \frac{1}{N}\sum_{j=0}^{N-1} a_{ij}w^{jl}, \qquad \hat{a}_{kl} = \frac{1}{N}\sum_{i=0}^{N-1} \hat{b}_{il}w^{ik}$$

Vi har alltså utnyttjat att Fouriertransformen är separabel. Ovanstående ekvationer säger att vi kan Fouriertransformera tvådimensionella data genom att först transformera varje kolumn i matrisen (a_{ij}) och sedan transformera raderna i den resulterande matrisen (b_{il}) (eller tvärtom om man vill det). Algoritmen blir då

2D-FFT$(a_{0,0}, a_{0,1}, \ldots, a_{N-1,N-1}, N) =$
    for $i \leftarrow 0$ **to** $N-1$
        $b_{i,0}, \ldots, b_{i,N-1} \leftarrow FFT(a_{i,0}, a_{i,1}, \ldots, a_{i,N-1}, N)$
    for $i \leftarrow 0$ **to** $N-1$
        $c_{0,i}, \ldots, c_{N-1,i} \leftarrow FFT(b_{0,i}, b_{1,i}, \ldots, b_{N-1,i}, N)$
    return $c_{0,0}, c_{0,1}, \ldots, c_{N-1,N-1}$

FFT anropas $2N$ gånger. Eftersom varje anrop tar $O(N \log N)$ blir totaltiden $O(N^2 \log N)$. □

Lösning till Binärträd med speglad struktur
Vi använder dekomposition för att stega oss ner i båda träden samtidigt. För att träden ska vara varandras spegelbilder måste vänstra delträdet till första trädet vara spegelbilden av högra delträdet till andra trädet och högra delträdet till första trädet vara spegelbilden av vänstra delträdet till andra trädet. Basfallet blir att minst ett av träden är tomt. I så fall är träden spegelbilder av varandra bara om båda träden är tomma.

SpegladeTräd(T_1, T_2**)=**
    if T_1 **=NIL or** T_2 **=NIL then**
        return T_1 **=NIL and** T_2 **=NIL**
    return SpegladeTräd(T_1**.left,**T_2**.right) and SpegladeTräd(**T_1**.right,**T_2**.left)**

Eftersom funktionen kommer att anropas (högst) en gång för varje delträd (rot) och arbetet i ett funktionsanrop förutom rekursionen är konstant så blir komplexiteten linjär i trädens storlek.

Det är lätt att inse att algoritmen är korrekt med hjälp av så kallad strukturell induktion, det vill säga induktion över trädens struktur. Vi kollar först att basfallet stämmer (om ena trädet är tomt så måste också det andra trädet vara tomt för att dom ska vara spegelbilder) och kollar sedan att induktionssteget stämmer (om första trädets vänstra delträd är spegelbilden av andra trädets högra delträd och första trädets högra delträd är spegelbilden av andra trädets vänstra delträd så måste träden vara varandras spegelbilder). □

Lösning till Partyproblemet

Anta att listan med vilka gäster som känner varandra består av m stycken rader. Det är enkelt att representera indata som en graf $G = (V, E)$ där varje person motsvarar ett hörn och varje kant (x, y) motsvarar att person x och y känner varandra. Representera grafen med kantlistor.

Den sökta lösningen är den största (inducerade) delgrafen där varje hörn har gradtal minst k. Vi kan hitta denna delgraf genom att plocka bort varje hörn som har gradtal mindre än k. När ett hörn plockas bort tas samtidigt alla kanter till hörnet bort, varför andra hörn får lägre gradtal och kan behöva plockas bort. Detta implementeras enklast med en variabel d_x i varje hörn x som håller reda på det aktuella gradtalet och som uppdateras varje gång en kant till det hörnet plockas bort. Låt oss också ha en kö Q där vi lägger alla hörn som har gradtal mindre än k i väntan på att deras kanter ska plockas bort.

```
foreach x ∈ V do
   if d_x < k then Q.Put(x)
while not Q.Empty() do
    x ← Q.Get()
    foreach (x, v) ∈ x.kantlista do
        if d_v ≥ k then
            d_v ← d_v − 1
            if d_v < k then Q.Put(v)
write 'Lösningen består av:'
foreach x ∈ V do
   if d_x ≥ k then write x
```

Notera först att så fort d_v blir mindre än k så läggs hörnet v in i kön och efter att det hamnat där så kommer inte d_v att ändras mer (på grund av if-satsen). Eftersom det finns n hörn och varje hörn behandlas ett konstant antal gånger (initiering av Q, köinsättning, köutplockning, utskrift), och eftersom det finns m kanter och varje kant behandlas högst två gånger (den förekommer i två kantlistor) så blir totala komplexiteten $O(n + m)$, alltså linjärt i indatas storlek.

Det är enkelt att se att algoritmen är korrekt eftersom varje hörn som skrivs ut i lösningen fortfarande har minst k grannar kvar i grafen, och inget hörn som inte har färre än k grannar kvar i grafen har plockats bort under algoritmens gång. □

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 7

Probabilistiska algoritmer, reduktioner

Skipplistor Sannolikheten p att ett element på nivå i i en skipplista också finns på nivå $i+1$ kan man variera efter behov. På sida 5 i skipplistartikeln analyseras hur sökstigens förväntade längd och det förväntade antalet pekare i ett element beror av p. Där framgår att kortaste sökstigen fås om $p = 1/e \approx 0.3679$. Varför kan det trots detta vara bättre att välja $p = 1/2$ eller $p = 1/4$?

Verifikation av matrismultiplikation Snabbaste kända metoden för multiplikation av två $n \times n$-matriser är $O(n^{2,376})$. Visa att det går att probabilistiskt verifiera en matrismultiplikation i kvadratisk tid. Närmare bestämt, konstruera en Montecarloalgoritm som i tid $O(kn^2)$ verifierar att $AB = C$ och har en felsannolikhet på högst 2^{-k}.

Reduktion som ger negativt resultat Att hitta medianelementet (det $\lceil n/2 \rceil$e minsta elementet) av n positiva heltal kräver minst $2n$ jämförelser i värsta fallet. Utnyttja detta resultat för att bestämma en undre gräns för antalet jämförelser som krävs för att hitta det $\lceil n/2 \rceil + 10$e minsta elementet.

Reduktion som ger positivt resultat Ett ofta användbart sätt att lösa problem på är att hitta en reduktion till ett problem som man redan vet hur man ska lösa. Du ska nu använda denna metod för att lösa kantsammanhängandegradsproblemet som definieras på följande sätt.

> INMATNING: En sammanhängande oriktad graf $G = (V, E)$ och ett positivt heltal K mellan 1 och $|V|$.
>
> PROBLEM: Är det tillräckligt att ta bort K kanter från grafen G för att göra den osammanhängande (dvs uppdelad i flera sammanhängande komponenter)?

Formulera optimeringsproblem som beslutsproblem Jobbmaskningsproblemet definieras som ett optimeringsproblem på följande sätt:

Indata: Arbetsdagens längd T och n arbetsuppgifter. Uppgift i tar tiden t_i och kräver arbetsinsatsen w_i. Alla tal är positiva heltal.

Lösning: Ett urval av arbetsuppgifterna $D \subseteq [1..n]$ som tillsammans fyller ut en arbetsdag, det vill säga $\sum_{i \in D} t_i \geq T$.

Målfunktion: Den arbetsinsats som krävs för att utföra dagsverket D, alltså $\sum_{i \in D} w_i$.

Mål: Minimera målfunktionen, alltså hitta det dagsverke som kräver minst arbetsinsats.

a) Formulera optimeringsproblemet som ett beslutsproblem. Utvidga indata med ett målvärde I för arbetsinsatsen och formulera en fråga som har svaret ja eller nej.

b) Turingreducera optimeringsproblemet till beslutsproblemet, det vill säga visa hur man kan lösa optimeringsproblemet med hjälp av en algoritm som löser beslutsproblemet.

Reduktioner mellan besluts-, optimerings- och konstruktionsproblem Anta att algoritmen GraphColouring(G,k) på tid $T(n)$ (där n är antalet hörn i G) svarar 1 omm hörnen i G kan färgas med k färger utan att någon kant har likfärgade ändpunkter.

a) Konstruera en algoritm som givet en graf G med n hörn bestämmer det minimala antalet färger som behövs för att färga G. Tiden ska vara $O(\log n \cdot T(n))$.

b) Konstruera en algoritm som givet en graf G med n hörn färgar hörnen med minimalt antal färger i tid $O(P(n)T(n))$ där $P(n)$ är ett polynom.

Lösningar

Lösning till Skipplistor
Det beror på att man vill snabbt kunna beräkna RandomLevel, alltså på vilken nivå ett nytt element ska in. Om $p = 2^{-k}$ så kan detta göras effektivt. Man läser helt enkelt k bitar av en slumpsträng (en ström av bitar) och kollar till exempel om alla k bitarna är 1. □

Lösning till Verifikation av matrismultiplikation
Idé: slumpa fram en n-vektor $\mathbf{r} \in \{0,1\}^n$, beräkna $\mathbf{a} = C \circ \mathbf{r}$ och $\mathbf{b} = A \circ (B \circ \mathbf{r})$ och verifiera att $\mathbf{a} = \mathbf{b}$.

Om $AB = C$ så blir alltid $\mathbf{a} = \mathbf{b}$. Om $AB \neq C$ så är $D = AB - C \neq 0$. Då måste det finnas något index i i $\mathbf{r}$ som påverkar värdet av $D\mathbf{r}$ om man byter $\mathbf{r}[i]$ så att det blir 1 om det var 0 och 0 om det var 1. Alltså, om algoritmen slumpar fram elementen i vektorn $\mathbf{r}$ med sannolikhet $1/2$ för 0 och $1/2$ för 1 så kommer $AB\mathbf{r} \neq C\mathbf{r}$ med sannolikhet minst $1/2$. Om vi upprepar detta k gånger kommer sannolikheten för att algoritmen blir lurad alla k gångerna vara mindre än $(1/2)^k$.

Algoritmen blir alltså:

```
VerifyMult(n, A, B, C, k) =
        for i ← 1  to k do
            r ← slumpvektor∈ {0,1}^n
            a ← C ∘ r
            b ← A ∘ (B ∘ r)
            if a ≠ b then return false
        return true
```

Tidskomplexiteten blir $O(kn^2)$ eftersom algoritmen gör k varv och i varje varv görs tre matris-vektormultiplikationer som tar $O(n^2)$. □

Lösning till Reduktion som ger negativt resultat
Reducera medianproblemet till det nya problemet genom att lägga till så många ettor till indata så att det tidigare medianelementet nu är det $\lceil m/2 \rceil + 10$e minsta elementet, där m är antalet element efter det att ettorna lagts till. Det betyder att vi måste lägga till 20 ettor och $m = n + 20$. Den undre gränsen $2n$ blir därmed uttryckt i m: $2n = 2m - 40$. □

Lösning till Reduktion som ger positivt resultat

Först bör vi försöka förstå problemet. Anta att X är en minimal kantmängd vars borttagande gör G osammanhängande. (Detta ska tolkas som att ingen strikt delmängd av X uppfyller detsamma.) Då gäller att G uppdelas i två komponenter, och alla kanterna i X går mellan dessa två. (Annars skulle ju X inte vara minimalt.) X svarar alltså mot ett snitt i grafen (en uppdelning av hörnmängden i två delar). Antalet kanter i X kan sägas vara snittets storlek. Detta innebär att det minsta antalet kanter som vi måste ta bort för att G ska bli osammanhängande — kalla detta $\lambda(G)$ — är lika med storleken på ett minsta snitt (V_1, V_2) i G.

Minimalt snitt är ju samma som maximalt flöde. Vi ska därför försöka reducera kantsammanhängandegradsproblemet till maximalt flöde mellan två hörn s och t i en graf. Om vi utökar G till en flödesgraf G' genom att ge varje kant dubbelriktad kapacitet 1, så kan vi alltså finna $\lambda(G)$ genom att beräkna maxflödet från s till t för olika s och t. Vi behöver dock inte variera båda dessa. Om vi väljer s godtyckligt så måste det höra till någon av mängderna V_1 och V_2 ovan. Genom att variera t över alla hörn utöver s kommer så vi garanterat att träffa något hörn i den andra mängden. Slutsatsen blir alltså att för ett godtyckligt hörn s gäller

$$\lambda(G) = \min_{t \in V - \{s\}} \{\text{MaxFlow}(G', s, t)\}.$$

Om vi ska vara noggranna så var nu uppgiften att avgöra om $K \geq \lambda(G)$. Vi kan alltså svara NEJ om $K < \text{MaxFlow}(G', s, t)$ för alla $t \in V - \{s\}$ och JA annars.

Flödesalgoritmen anropas $|V| - 1$ gånger. Varje flödesberäkning kan göras i tid $O(|V|^3)$ (vilket man inte behöver kunna utantill). Alltså blir komplexiteten för vår algoritm $O(|V|^4)$. □

Lösning till Formulera optimeringsproblem som beslutsproblem

a) **Indata:** Arbetsdagens längd T, målet I och n arbetsuppgifter. Uppgift i tar tiden t_i och kräver arbetsinsatsen w_i.

Fråga: Finns det en delmängd $D \subseteq [1..n]$ så att $\sum_{i \in D} t_i \geq T$ och $\sum_{i \in D} w_i \leq I$?

b) Anta att Jobbmaskning$(T, I, n, \{t_i\}, \{w_i\})$ är en algoritm som löser beslutsproblemet. Optimala totala arbetsinsatsen måste vara ett heltal mellan 0 och $\sum_{1 \leq i \leq n} w_i$. Gör en binärsökning mellan dessa extremvärden efter det minsta värde I som Jobbmaskning$(T, I, n, \{t_i\}, \{w_i\})$ svarar ja för.

□

Lösning till Reduktioner mellan besluts-, optimerings- och konstruktionsproblem

a) Vi vet att antalet färger är mellan 1 och n. Använd binärsökning i detta intervall för att med hjälp av algoritmen GraphColouring hitta ett k så att GraphColouring$(G, k) = 1$ och GraphColouring$(G, k-1) = 0$. Detta innebär nämligen att minimala färgningen har k färger. Det behövs $\log n$ halveringar av n för att komma ner till 1. Tidskomplexiteten blir därför $O(\log n \cdot T(n))$.

b) Hitta först det minimala antalet färger k med metoden ovan. Vi vill färga hörnen i G med färger med nummer 1 till k. Följande algoritm gör det:

CreateColouring$(G = (V, E),\ k)$=
$u \leftarrow$ första hörnet i V
$C \leftarrow \{u\}; u.colour \leftarrow k$
foreach $v \in V - \{u\}$ **do**
  if $(u, v) \notin E$ **then**
    if GraphColouring$((V, E \cup \{(u, v)\}), k) = 1$ **then** $E \leftarrow E \cup \{(u, v)\}$
    else $C \leftarrow C \cup \{v\}; v.colour \leftarrow k$
if $k > 0$ **then** *CreateColouring*$((V - C, E), k - 1)$

GraphColouring anropas högst en gång för varje par av hörn i grafen. Tidskomplexiteten för hela algoritmen blir därför $O(\log n \cdot T(n) + n^2 \cdot T(n)) = O(n^2 \cdot T(n))$. □

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 8

Oavgörbarhet

Övningens ena timme ägnas åt genomgång av mästarprov 1.

Oavgörbarhet 1 Problemet *orm i kakel* tar som indata en mängd T av kakelplattstyper och två punkter p_1 och p_2 i planet. Frågan är om det går att använda endast kakeltyper i T kakla en orm som ringlar sig från p_1 till p_2, som inte bryter kakelmönstret någonstans och som hela tiden håller sig i det övre halvplanet.

Är följande varianter av ormproblemet avgörbara eller oavgörbara?

a) Indata utvidgas med en fjärde parameter, en kakeltyp $t \in T$. Första kakelplattan (den som täcker p_1) måste vara av typ t.

b) Indata utvidgas med ett tal N och frågan utvidgas med bivillkoret att ormen måste bestå av högst N plattor.

c) Indata utvidgas med ett tal N och frågan utvidgas med bivillkoret att ormen måste bestå av åtminstone N plattor.

Oavgörbarhet 2 Finns det något explicit program P så att det givet y är avgörbart huruvida P stannar på indata y?

Oavgörbarhet 3 Finns det något explicit program P så att det givet y är oavgörbart huruvida P stannar på indata y?

Oavgörbarhet 4 Om programmet x stannar på tomt indata så låter vi $f(x)$ vara antalet steg innan det stannar. Annars sätter vi $f(x) = 0$. Definiera nu $MR(y)$, den maximala körtiden över alla program vars binära kodning är mindre än y.

$$MR(y) = \max_{x \leq y} f(x),$$

Är MR beräkningsbar?

Oavgörbarhet 5 Visa att funktionen MR i föregående uppgift växer snabbare än varje rekursiv funktion. Visa närmare bestämt att för varje rekursiv funktion g finns ett y så att $MR(y) > g(y)$.

Oavgörbarhet 6 Anta att en turingmaskin M, indata X (som står på bandet från början) och en heltalskonstant K är givna. Är följande problem avgörbart eller oavgörbart?

Stannar M på indata X efter att ha använt högst K rutor på bandet (en använd ruta får skrivas och läsas flera gånger)?

Oavgörbarhet 7 En *rekursivt uppräknelig mängd* definierades på föreläsningen som ett språk som kan kännas igen av en funktion vars ja-lösningar kan verifieras ändligt. En alternativ definition är en mängd som är uppräknelig (elementen kan numreras med naturliga tal) och kan produceras av en algoritm. Använd den senare definitionen och visa att varje rekursiv mängd också är rekursivt uppräknelig.

Oavgörbarhet 8 Den *diagonaliserade stoppmängden* består av alla program p som stannar på indata p. Visa att denna mängd är rekursivt uppräknelig.

Lösningar

Lösning till Oavgörbarhet 1

a) Oavgörbart. Vi reducerar *orm-i-kakel* till *orm-med-speciell-början* på följande sätt.

orm-i-kakel$(T, p_1, p_2) =$
    for $t \in T$ **do**
        if *orm-med-speciell-början*(T, p_1, p_2, t) **then**
            return true
    return false

b) Avgörbart. Vi behöver ”bara” testa alla tänkbara ormar av längd högst N som börjar i p_1 och se om dom duger. Eftersom detta är ett ändligt antal (begränsat av $(3|T|)^N$) så är algoritmen ändlig och problemet avgörbart.

c) Oavgörbart. Vi reducerar *orm-i-kakel* till *orm-med-minsta-längd* på följande sätt.

orm-i-kakel$(T, p_1, p_2) =$
    return *orm-med-minsta-längd*$(T, p_1, p_2, 1)$

□

Lösning till Oavgörbarhet 2
Ja, det finns massor av sådana program. Ta till exempel det enkla programmet $P(y) =$ **return**. Det är lätt att se att P stannar på alla indata. □

Lösning till Oavgörbarhet 3
Ja. Låt till exempel P vara en interpretator och indata y vara ett program x följt av indata y' till det programmet. Det betyder att $P(y)$ beter sig precis som programmet x skulle göra på indata y', och det är ju oavgörbart huruvida ett program x stannar på ett visst indata y'. □

Lösning till Oavgörbarhet 4
Nej, *MR* är inte beräkningsbar. Vi vet att det är oavgörbart huruvida ett program stannar på blankt indata (tomma strängen). Reducera därför detta problem till *MR*. Notera att *MR*(y) är det maximala antalet steg som varje program av ”storlek” högst y behöver innan det stannar om det startas med blankt indata.

StopBlank(y) =
    $s \leftarrow MR(y)$
    if $s = 0$ **then return false**
    else
        simulera y på blankt indata i s steg (eller tills y stannar)
        if y stannade **then return true**
        else return false

Om inte y har stannat inom s steg så vet vi att den aldrig stannar. Eftersom StopBlank inte är beräkningsbar så kan inte MR vara det heller. □

Lösning till Oavgörbarhet 5
Anta motsatsen, dvs att det finns en rekursiv funktion g så att $g(y) \geq MR(y)$ för alla y. Då skulle vi kunna byta ut första satsen $s \leftarrow MR(y)$ i programmet i lösningen till föregående uppgift mot satsen $s \leftarrow g(y)$ och programmet skulle ändå fungera. Dessutom skulle programmet då bli beräkningsbart vilket vi vet att det inte kan vara. Därför har vi en motsägelse och vårt antagande var alltså falskt. □

Lösning till Oavgörbarhet 6
Problemet är avgörbart. Eftersom turingmaskinen måste hålla sig inom K rutor på bandet så finns det bara ett ändligt antal konfigurationer som turingmaskinen kan befinna sig i. Om maskinen har m tillstånd så är antalet konfigurationer $m \cdot K \cdot 3^K$ (antalet tillstånd gånger antalet möjliga positioner för läs- och skrivhuvudet gånger antalet möjliga bandkonfigurationer). Simulera därför turingmaskinen i $m \cdot K \cdot 3^K + 1$ steg och kontrollera hela tiden att den inte rör sig utanför dom K rutorna på bandet. Om turingmaskinen stannar inom denna tid svarar vi *ja*. Om turingmaskinen inte stannat efter denna tid så måste den ha återkommit till en konfiguration som den tidigare varit i, och därmed är den inne i en oändlig slinga och vi kan tryggt svara *nej*. □

Lösning till Oavgörbarhet 7
Anta att S är en rekursiv mängd. Det betyder att det finns en algoritm $A(x)$ som talar om ifall $x \in S$. Elementen i S lagras naturligtvis som allt annat som binära strängar. Följande program genererar mängden S:

for $i \leftarrow 0$ **to** ∞ **do**
    if $A(i)$ **then write** i

□

Lösning till Oavgörbarhet 8

for $i \leftarrow 0$ **to** ∞ **do**
    for $p \leftarrow 0$ **to** i **do**
        simulera beräkningen $p(p)$ under i steg
        if $p(p)$ stannar inom i steg **then write** p

Samma p kommer att skrivas ut många gånger. Om man vill undvika det får man se till att bara skriva ut p om antingen $p(p)$ stannar på exakt i steg eller $p = i$. □

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 9

NP-fullständighetsbevis

På denna övning är det också **inlämning av skriftliga lösningar av teoriuppgifterna till labb 4** och muntlig redovisning av teoriuppgifterna.

Vad säger reduktionerna? A, B, C, D och E är beslutsproblem. Anta att B är NP-fullständigt och att det finns polynomiska Karpreduktioner mellan problemen så här (en reduktion av A till B tecknas här $A \to B$):

$$\begin{array}{ccccccc} \text{A} & \to & \text{B} & \leftrightarrow & \text{C} & \leftarrow & \text{D} \\ & & \downarrow & & & & \\ & & \text{E} & & & & \end{array}$$

Vad vet man då om komplexiteten för A, C, D och E? Sätt ett kryss i tabellen nedan för det man säkert vet och en ring i för det som är möjligt men som man inte vet säkert.

	ligger i NP	är NP-fullständigt	är NP-svårt
A			
C			
D			
E			

Frekvensallokering Inom mobiltelefonin behöver man lösa *frekvensallokeringsproblemet* som lyder på följande sätt. Det finns ett antal sändare utplacerade. Varje sändare kan sända på någon av en given uppsättning frekvenser. Olika sändare kan ha olika frekvensuppsättningar. Vissa sändare är så nära varandra att dom inte kan sända på samma frekvens, för då skulle dom störa varandra. (Detta beror faktiskt inte så mycket på avståndet som på geografin — om det finns något berg, hus eller liknande som skärmar av.)
Man vet från början vilka sändare som finns, vilken frekvensuppsättning varje sändare har och vilka par av sändare som skulle störa varandra om dom sände på samma frekvens. Problemet är att avgöra ifall det finns något möjligt val av frekvenser så att inte någon sändare stör någon annan.

a) Formulera detta problem som ett grafproblem
b) Visa att problemet ligger i NP genom att:
b1) Föreslå vad en lösning kan vara.
b2) Visa att om svaret är ja så kan lösningen verifieras.
b3) Visa att verifikationen tar polynomisk tid.
c) Visa att problemet är NP-svårt genom att:
c1) Hitta ett känt NP-fullständigt problem att reducera.
c2) Beskriv reduktionen mellan det kända problemet och frekvensallokeringsproblemet.
c3) Visa att reduktionen är polynomisk.
c4) Visa att reduktionen är korrekt.

Därefter vet du att problemet är NP-fullständigt!

Hamiltonsk stig i graf Visa att problemet Hamiltonian Path är NP-fullständigt. Problemet är att avgöra ifall det finns någon enkel stig som passerar alla hörn i en graf.

Spännande träd med begränsat gradtal Visa att följande problem är NP-fullständigt: Givet en oriktad graf $G = (V, E)$ och ett heltal k, avgör ifall G innehåller ett spännande träd T så att varje hörn i trädet har gradtal högst k.

Polynomisk reduktion Konstruera en polynomisk reduktion av 3CNF-SAT till EQ-GF[2], satisfierbarhetsproblemet för ett system av polynomekvationer över GF[2] (alltså heltal modulo 2).

Är en Eulergraf k-färgbar? Många problem av typen *avgör om grafen G har egenskapen e* kan eventuellt förenklas om vi antar att grafen har någon speciell annan egenskap. Vi ska studera ett specialfall. Vi säger att en sammanhängande graf är en *Eulergraf* om varje hörn i grafen har jämn grad. Vi vill nu avgöra om en sådan graf är k-färgbar. Vi har närmare bestämt följande problem:

> INDATA: En Eulergraf G och ett heltal k.
> UTDATA: **JA** om grafen är k-färgbar. **NEJ** annars.

För $k \leq 2$ finns det en polynomisk algoritm för att avgöra färgbarhet för generella grafer. Vi antar därför att $k \geq 3$. Avgör nu om det finns en polynomisk algoritm för att lösa ovanstående problem eller om problemet är NP-fullständigt.

Lösningar

Lösning till Vad säger reduktionerna?

	ligger i NP	är NP-fullständigt	är NP-svårt
A	x	o	o
C	x	x	x
D	x	o	o
E	o	o	x

□

Lösning till Frekvensallokering

a) Formulera frekvensallokeringsproblemet som ett grafproblem.

Låt hörnen motsvara sändare och kanter motsvara sändare som stör varandra. Varje hörn v_i är märkt med en frekvensuppsättning F_i. Frågan är om det går tilldela varje hörn en frekvens från dess frekvensuppsättning så att inga hörn som är förbundna med en kant har samma frekvens.

b) Visa att grafproblemet ligger i NP.
b1) En lösning är lämpligen en frekvenstilldelning till varje hörn.
b2) Gå igenom varje hörn och verifiera att dess frekvens är med i frekvensuppsättningen. Gå också igenom varje kant och verifiera att ändpunkternas frekvenser är olika.
b3) Verifikationen tar linjär tid i grafens storlek.

c) Visa att grafproblemet är NP-svårt.
c1) Vi provar att reducera k-färgningsproblemet.
c2)

k-färgning(G, k) =
    för varje hörn v_i i grafen G
        $F_i \leftarrow \{1, \ldots, k\}$
    return frekvensallokering$(G, \{F_i\})$

c3) Reduktionen skapar bara en k-mängd för vart och ett av hörnen. Den är alltså uppenbart polynomisk.
c4) Visa nu att det finns en k-färgning av grafen G om och endast om det finns en tillåten frekvenstilldelning till G där alla hörn har frekvensuppsättningen $\{1, \ldots, k\}$.

Anta att vi har en k-färgning av G. Numrera färgerna 1 till k. Om ett hörn har fått färg i så låter vi motsvarande hörn (sändare) i frekvensallokeringsproblemet få frekvensen i. Detta blir en tillåten frekvenstilldelning eftersom vi har utgått från en tillåten k-färgning.

Åt andra hållet: anta att vi har en tillåten frekvenstilldelning. Vi får en k-färgning genom att låta ett hörn få färg i om motsvarande sändare har fått frekvens i.

□

Solution to Hamiltonsk stig i graf
A Hamiltonian path is a simple open path that contains each vertex in a graph exactly once. The Hamiltonian Path problem is the problem to determine whether a given graph contains a Hamiltonian path.

To show that this problem is NP-complete we first need to show that it actually belongs to the class NP and then find a known NP-complete problem that can be reduced to Hamiltonian Path.

For a given graph G we can solve Hamiltonian Path by nondeterministically choosing edges from G that are to be included in the path. Then we traverse the path and make sure that we visit each vertex exactly once. This obviously can be done in polynomial time, and hence, the problem belongs to NP.

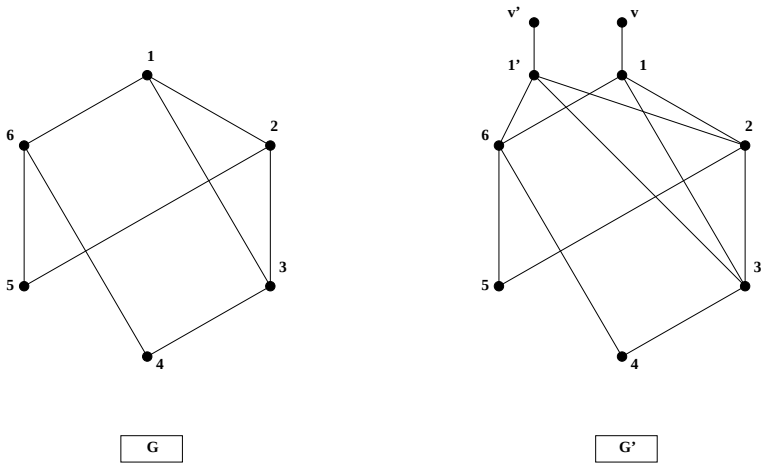
Now we have to find an NP-complete problem that can be reduced to Hamiltonian Path. A closely related problem is the problem to determine whether a graph contains a Hamiltonian cycle, that is, a Hamiltonian path that begin and end in the same vertex. Moreover, we know that Hamiltonian Cycle is NP-complete, so we may try to reduce this problem to Hamiltonian Path.

Given a graph $G = \langle V, E \rangle$ we construct a graph G' such that G contains a Hamiltonian cycle if and only if G' contains a Hamiltonian path. This is done by choosing an arbitrary vertex u in G and adding a copy, u', of it together with all its edges. Then add vertices v and v' to the graph and connect v with u and v' with u'; see Figure 2 for an example.

Suppose first that G contains a Hamiltonian cycle. Then we get a Hamiltonian path in G' if we start in v, follow the cycle that we got from G back to u' instead of u and finally end in v'. For example, consider the left graph, G, in Figure 2 which contains the Hamiltonian cycle $1, 2, 5, 6, 4, 3, 1$. In G' this corresponds to the path $v, 1, 2, 5, 6, 4, 3, 1', v'$.

Conversely, suppose G' contains a Hamiltonian path. In that case, the path must necessarily have endpoints in v and v'. This path can be transformed to a cycle in G. Namely, if we disregard v and v', the path must have endpoints in u and u' and if we remove u' we get a cycle in G if we close the path back to u instead of u'.

The construction won't work when G is a single edge, so this has to be taken care of as a special case. Hence, we have shown that G contains a Hamiltonian cycle if and only if G' contains a Hamiltonian path, which concludes the proof that Hamiltonian Path is NP-complete. □



Figur 2: En graf G och hamiltonstigsreduktionens konstruerade graf G'.

Solution to Spännande träd med begränsat gradtal

In this solution we use the alternative (original) definition of NP as *nondeterministic polynomial time*. First note that the problem can be solved by the following nondeterministic algorithm:

1. For each edge in E, choose nondeterministically if it is to be included in T.
2. Check that T is a tree and that each vertex has degree less than k.

This means that the problem is in NP. Now we need to reduce a problem known to be NP-complete to our spanning tree problem. In this way we can state that determining whether a graph has a k-spanning tree is at least as hard as every other problem in NP.

Consider the problem Hamiltonian Path that was shown to be NP-complete in the previous exercise. Can we reduce this problem to the spanning tree problem? That is, can we solve Hamiltonian Path if we know how to solve the spanning tree problem? We claim that we can and that G has a Hamiltonian path if and only if it has a spanning tree with vertex degree ≤ 2.

It is easy to see that such a spanning tree is a Hamiltonian path. Since it has degree ≤ 2 it cannot branch and since it is spanning only two vertices can have degree < 2. So the spanning tree is a Hamiltonian path. If, on the other hand, G contains a Hamiltonian path, this path must be a spanning tree since the path visits every node and a path trivially is a tree.

We have reduced Hamiltonian Path to the spanning tree problem and, therefore, our problem is NP-complete. □

Solution to Polynomisk reduktion

Suppose we have a formula $\varphi \in$ 3Cnf-sat, for example

$$\varphi(x_1, x_2, x_3, x_4) = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_1} \vee x_3 \vee x_4)$$

Reducing 3Cnf-sat to Eq-gf[2] means that, for every formula φ, we can create an equation that is solvable if and only if φ is satisfiable. Now, φ is only satisfiable if every clause can be satisfied simultaneously, so let's start with finding an equation for an arbitrary clause $(x \vee y \vee z)$, where x, y and z are considered as literals rather than variables. For a start, the clause is satisfiable with one of the literals set to true, so in that case the equation $x + y + z = 1$ is also solved. But this fails if two literals are true! We can fix this by adding three more terms, where one and only one is 1 if two variables are 1. We get

$$x + y + z + xy + yz + xz = 1$$

Again, this is not complete. if all variables are set to 1, it fails. Of course, the solution is to add a term for this case, xyz. Our result is

$$x + y + z + xy + yz + xz + xyz = 1$$

If one of the literals in the clause happened to be inverted, e.g. we have $\overline{x}$ instead of x, replace x by $(1 + x)$ in the equation. The first clause in the example would be turned into

$$x_1 + x_2 + (1 + x_3) + x_1x_2 + x_1(1 + x_3) + x_2(1 + x_3) + x_1x_2(1 + x_3) = 1$$

We are now ready to create an equation from an arbitrary 3-CNF formula φ. For every clause φ_i in φ, create a corresponding equation Q_i. Our claim is that this system of equations is solvable if and only if φ is satisfiable and we must prove that this is correct.

To begin with, we note that $(1+x)$ is a proper way to handle inverses, $1+1 = 0$ and $1+0 = 1$, so in the following we will only consider literal values and not variables.

1. ($\Leftarrow$) If an equation in Q_i is satisfiable, then there exists an assignment to the variables such that each left hand expression is summed to 1. Hence, at least one of the literals is 1 and the corresponding literal in the boolean formula set true would make it's clause satisfied. The equation system Q_i being satisfied means that each equation is satisfied and consequently is each clause in φ satisfied.

2. ($\Rightarrow$) When φ is satisfied, we have an assignment on the variables such that each clause is satisfied. There are three cases for the clauses, (i) one literal is true, (ii) two literals are true and (iii) three literals are true. By construction, the corresponding equations are all satisfied.

□

Lösning till Är en Eulergraf k-färgbar?
Problemet är NP-fullständigt. Det är enkelt att verifiera om en lösning är en korrekt färgning så problemet ligger i NP. För att visa fullständighet reducerar vi det generella k-färgningsproblemet till vårt problem. Vi antar att $k \geq 3$. Anta att vi har en graf G. Vi gör om den till en Eulergraf G' på följande sätt: Det måste finnas ett jämnt antal hörn med udda grad. Dela in dessa hörn i par. För varje par införs ett nytt hörn. Till detta hörn dras två kanter, en från vart och ett av de gamla hörn som genererat det nya hörnet. Grafen G plus de nya hörnen och kanterna utgör grafen G'. Det är lätt att se att G' är en Eulergraf.

Vi visar nu att G är k-färgbar $\Leftrightarrow$ G' är k-färgbar. Anta att f är en färgning av G, det vill säga att varje hörn x får en färg $f(x)$. Vi definierar då en k-färgning f' av G' på följande sätt: Om x är ett hörn i G' som också tillhör G sätts $f'(x) = f(x)$. Om x inte tillhör G så finns det två grannhörn y och z i G. Vi sätter då $f(x)$ till någon annan ledig färg än $f(y)$ och $f(z)$. (Det är möjligt att göra det eftersom $k \geq 3$.) Implikationen åt andra hållet är trivial. □

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 10

NP-fullständiga problem

Konstruera kappsäckslösning Kappsäcksproblemet är ett välkänt NP-fullständigt problem (se föreläsning 21). Indata är en mängd P med prylar med vikt w_i och värde v_i, en kappsäcksstorlek S och ett mål K. Frågan i kappsäcksproblemet är ifall det går att välja ut prylar av sammanlagt värde minst K så att deras sammanlagda vikt är högst S. Alla tal i indata är ickenegativa heltal.

Anta nu att vi har en algoritm A som löser ovanstående beslutsproblem. Konstruera en algoritm som med hjälp av anrop till A löser det konstruktiva kappsäcksproblemet, det vill säga med samma indata som A dels besvarar kappsäcksproblemet och dels, ifall svaret är ja, talar om precis vilka prylar som ska packas ned i kappsäcken. Algoritmen får anropa A $O(|P|)$ gånger men får i övrigt inte ta mer än polynomisk tid.

Du ska alltså konstruera och analysera en turingreduktion av det konstruktiva kappsäcksproblemet till det vanliga kappsäcksproblemet (som är ett beslutsproblem).

Tillförlitlighet hos Internet Det händer alltför ofta att datorer eller enskilda förbindelser mellan datorer är trasiga. För att detta inte ska störa möjligheten att kommunicera inom resten av Internet vill man att det ska finnas alternativa vägar mellan varje par av datorer i nätet. Om det till exempel finns tre olika vägar genom Internet från datorn A till datorn B och dessa vägar dessutom är helt disjunkta (utom ändpunkterna) så kan två stycken datorer, vilka som helst, försvinna utan att möjligheten att kommunicera mellan A och B försvinner.

I det här problemet tänker vi oss Internet som en oriktad graf där hörnen motsvarar datorerna i Internet och varje kant (X, Y) motsvarar en *möjlig direktförbindelse* mellan datorerna X och Y. För varje par av datorer (X, Y) har vi också en önskad tillförlitlighet $T(X, Y)$ som anger hur många disjunkta vägar det ska finnas i Internet mellan X och Y. Till sist finns det en budget B som anger hur många direktförbindelser man har råd att konstruera.

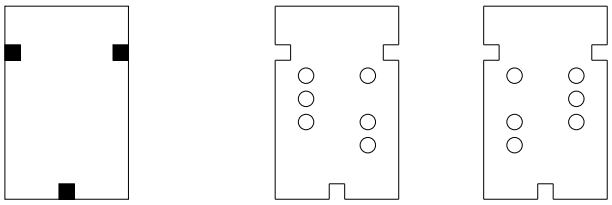
Frågan är: går det att plocka ut en delgraf bestående av B kanter så att det för alla par av hörn (X, Y) finns minst $T(X, Y)$ disjunkta stigar i delgrafen mellan X och Y.

Visa att detta problem är NP-fullständigt!

Håstads leksaksaffär Håstads leksaksaffär säljer ett pussel som består av en låda och ett antal kort, se figur 3. Kort kan placeras i lådan på två sätt, med framsidan upp eller med baksidan upp, eftersom det finns urgröpningar i korten som måste passa in i lister som sitter i lådan. På varje kort finns två kolumner med hål, men vissa av hålen är igenfyllda. Målet med pusslet är att lägga korten i lådan på ett sånt sätt att hela botten täcks; för varje hålposition måste alltså åtminstone ett av korten ha ett igenfyllt hål.

Bevisa att språket $\textsc{Toys} = \{\langle c_1, c_2, \ldots, c_n\rangle$: ($c_i$ beskriver kort) $\wedge$ (det går att lösa pusslet)$\}$ är NP-fullständigt.

Processorschemaläggning Givet n jobb som ska exekveras och ett tillräckligt antal processorer. Varje jobb tar en tidsenhet att utföra. Det finns också villkor på formen *jobb i kan inte exekveras samtidigt som jobb j*. Vi kallar problemet att avgöra ifall det går att schemalägga



Figur 3: Håstads pussel. Lådan med tre lister, ett kort med hål och urgröpningar samt samma kort lagt med baksidan upp.

jobben så att alla jobb kan exekveras på tre tidsenheter för SCHEDULE. Visa att detta problem är NP-fullständigt.

Är problemet fortfarande NP-fullständigt om vi kräver att jobben ska vara exekverade efter k tidsenheter där $k > 3$? Är det sant för $k = 2$?

Lösningar

Lösning till Konstruera kappsäckslösning

```
Kappsäck(P,S,K)=
  if not A(P,S,K) then return "Ingen lösning"
  foreach p in P do
    if A(P-p,S,K) then P:=P-{p}
  return P
```

Algoritmen anropar A högst $|P| + 1$ gånger. Den returnerade mängden P är en lösning för den uppfyller följande två kriterier.

- *P innehåller inga överflödiga element* eftersom forslingan löper över alla element i P, och i varje varv kollas om elementet p är överflödigt; om det är det plockas det bort ur P.
- *P innehåller en lösning* eftersom detta är en invariant för slingan. Om man kommer till slingan är detta uppfyllt, och efter varje varv i slingan är det uppfyllt.

□

Ledning till Tillförlitlighet hos Internet
Reducera problemet *Hamiltonsk krets*.

Solution to Håstads leksaksaffär
Given cards in the box you can just by a glance verify that every hole position is covered. Thus the problem is in NP.

What remains is to show that every problem in NP can be reduced to TOYS. We do that by reducing CNF-SAT, known to be NP-complete, to TOYS. More precise, we must show that every CNF-formula

$$\varphi = (l_1 \vee l_2 \vee \ldots \vee l_i) \wedge (l_j \vee \ldots \vee l_k) \wedge \ldots \wedge (\ldots)$$

with n variables and m clauses reduced to a problem γ in TOYS is satisfiable if and only if γ is solvable. Consider the following rules.

1. Variable x_i corresponds to a card c_i.

2. There are two columns with m positions on every card, so there is one row for each clause.

3. If x_i occurs in clause j then the hole in position j in the left column is covered.

4. If $\bar{x}_i$ occurs in clause j then the hole in position j in the right column is covered.

5. There are holes in all other positions.

6. There is an additional card c_{n+1} with holes only in the left column.

If φ is satisfied for an assignment of $x_1, \ldots, x_n$ then the corresponding puzzle γ is solvable, because if $x_i = 1$ then card c_i can be put down "right side up" and thus covering the holes in the left column corresponding to the clauses x_i satisfies. If $\bar{x}_i = 1$ then the card c_i should be placed "up-side-down" and in the same way covering holes. Since the formula was satisfied, every clause must be satisfied and therefore the left holes in every row are covered. The extra card c_{n+1} is used for covering the holes in the right column that might occur when all variables in a clause are set to 1, so all right holes are covered too.

Conversely, if the puzzle γ is solved, we can easily find an assignment to $x_1, \ldots, x_n$ satisfying φ from the following rule:

$$x_i = \begin{cases} 1, & \text{if } c_i \text{ is right side up} \\ 0, & \text{otherwise} \end{cases}$$

We also need to check the placing of the extra card; if it is up-side-down we simply invert the assignment of all variables. When all holes are covered we also have, by construction, that all clauses are satisfied. We have shown that CNF-SAT can be reduced to TOYS. □

Solution to Processorschemaläggning

First we need to show that the problem is in NP. This is true because we can guess a scheduling of the jobs and then easily verify that it is valid.

Next, to prove that SCHEDULE is NP-complete, we have to reduce an NP-complete problem to it. We choose to reduce the NP-complete problem 3-COLORING to SCHEDULE. This means that, given a graph $G = (V, E)$, we want to create an instance of SCHEDULE such that the graph can be colored with 3 colors if and only if the jobs can be scheduled in 3 time units.

We create a job for each vertex in the graph and if the vertex i is a neighbor to the vertex j we add the condition that job i can not be executed at the same time as job j. Let each color correspond to a time unit. Then we claim that the graph can be colored with 3 colors if and only if the jobs can be executed in 3 time units. First, if the graph can be colored with 3 colors let C_1 be the vertices that is colored with the first color. There is no edge between the vertices in C_1 and hence the corresponding jobs can all be executed in the first time unit. The same is true for colors 2 and 3 which gives that all jobs can be executed in 3 time units. On the other hand, if the jobs can be executed in 3 time units, the vertices that correspond to the jobs in each time unit can be given one color so that the graph can be 3-colored.

In this way we have reduced 3-COLORING to SCHEDULE and, consequently, SCHEDULE is NP-complete.

With the same argument we can reduce k-COLORING to SCHEDULE with k time units so that the latter problem is NP-complete for $k \geq 3$.

If we have 2 time steps, the problem can be solved by performing the reduction in the opposite way. That is, given the jobs and the conditions, we construct a graph which we then try to color with 2 colors. This can be done in linear time with depth first search, since if a vertex v has color 1, all neighbors to v must have color 2. □

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 11

Approximationsalgoritmer

Approximation av oberoende mängd Låt Independent Set$-B$ vara problemet att hitta en maximal mängd oberoende hörn i en graf vars gradtal (i varje hörn) är begränsat av konstanten B. Visa att detta problem ligger i APX, det vill säga kan approximeras inom någon konstant i polynomisk tid.

Probabilistisk alla-inte-lika-satisfiering Det NP-svåra problemet Max Not-All-Equal 3-CNF Sat definieras på följande sätt.

> Inmatning: En CNF-formel som består av klausulerna $c_1, c_2, \ldots, c_m$ där varje klausul är en disjunktion av exakt tre literaler (variabler eller negerade variabler). Variablerna heter $x_1, x_2, \ldots, x_n$.
> Lösning: En variabeltilldelning.
> Målfunktion: Antalet klausuler som innehåller minst en sann literal och minst en falsk literal.
> Problem: Maximera målfunktionen.

Eftersom detta problem är NP-svårt så vill vi ha en algoritm som approximerar det inom en konstant i polynomisk tid. Konstruera en probabilistisk approximationsalgoritm för problemet som ger en förväntad approximationskvot på $4/3$. Analysera din algoritms tidskomplexitet och förväntad approximationskvot. Du behöver inte slumpeliminera algoritmen.

Approximation av linjära olikheter Max Sat LR$^{\geq}$ (maximum satisfiable linear subsystem) är problemet att, givet en uppsättning linjära olikheter av typen $\geq$ (till exempel $2x_1 - 8x_3 + 3x_8 \geq 3$), hitta en variabeltilldelning som satisfierar så många olikheter som möjligt. Konstruera en approximationsalgoritm som approximerar Max Sat LR$^{\geq}$ inom faktorn 2.

Övre gräns för approximation av homogena bipolära olikheter
Max Hom Bipolar Sat LR$^{\geq}$ (maximum homogeneous bipolar satisfiable linear subsystem) är samma problem som Max Sat LR$^{\geq}$ men variablerna får bara anta värdena 1 och -1, och alla olikheter är homogena, det vill säga saknar konstanttermer (har noll i högerledet).

Visa att Max Hom Bipolar Sat LR$^{\geq}$ kan approximeras inom faktorn 2 och att Max Hom Bipolar Sat LR$^{>}$ kan approximeras inom 4.

Undre gräns för approximation av binära olikheter Max Binary Sat LR$^{\geq}$ (maximum binary satisfiable linear subsystem) är samma problem som Max Sat LR$^{\geq}$ där variablerna bara får anta värdena 0 och 1.

Visa att Max Binary Sat LR$^{\geq} \notin$APX.

Lösningar

Lösning till Approximation av oberoende mängd
Givet en graf $G = (V, E)$ vars gradtal är begränsat av B, konstruera en oberoende mängd hörn på följande sätt.

$V' \leftarrow \emptyset$
$W \leftarrow V$
for $v \in W$ **do**
  $V' \leftarrow V' \cup \{v\}$
  $W \leftarrow W - \{w \in W : (w, v) \in E\} - \{v\}$

Efter algoritmen är V' en oberoende mängd hörn. Det är också lätt att se att V' är en dominerande mängd. Den optimala oberoende mängden hörn V'_{opt} kan högst vara B gånger större än $|V'|$. För att se detta behöver man bara titta på ett av hörnen i V' och dess grannar i V. Om alla dess grannar är med i V'_{opt} så blir det B stycken, och värre kan det inte bli. Eftersom V' är en dominerande mängd måste varje hörn i V'_{opt} antingen vara med i V' eller vara granne till ett hörn i V'. Det betyder att V'_{opt} är högst B gånger större än V', det vill säga att approximationskvoten är B, som är en konstant. □

Lösning till Probabilistisk alla-inte-lika-satisfiering
Algoritmen är helt enkelt: sätt varje variabel till ett slumpmässigt valt värde. Det ska vara lika sannolikt att variabeln sätts till sant som till falskt. Om vi nu tittar på en godtycklig klausul så är det bara i två fall av åtta (nämligen falskt-falskt-falskt och sant-sant-sant) som den inte ska räknas. Väntevärdet för antalet klausuler som räknas är

$$\begin{aligned}
&E[\text{antal klausuler som har både sanna och falska literaler}] = \\
&= m \cdot Pr[\text{en godtycklig klausul är varken helt sann eller falsk}] = \\
&= m \cdot (1 - Pr[\text{en godtycklig klausul är antingen helt sann eller helt falsk}]) = \\
&= m \cdot (1 - 2/8) = 3m/4.
\end{aligned}$$

Eftersom högst m klausuler kan räknas blir den förväntade approximationskvoten

$$\frac{OPT}{APPROX} \geq \frac{m}{3m/4} = \frac{4}{3}.$$

Algoritmen tar linjär tid och kräver linjär slump i antalet variabler. □

Lösning till Approximation av linjära olikheter
Anta att inmatningen ges som en mängd X med rationella variabler och en mängd E med linjära olikheter över X.

while $E \neq \emptyset$ **do**
  if (det finns olikheter i E med en enda variabel) **then**
    $U \leftarrow \{x \in X : x$ är ensam variabel i minst en olikhet i $E\}$
    Välj godtyckligt $y \in U$.
    $F(y) \leftarrow \{e \in E : e$ bara innehåller variabeln $y\}$
    Ge y ett värde som satisfierar så många olikheter i $F(y)$ som möjligt.
    $E \leftarrow E - F(y)$
  else
    Välj godtyckligt $y \in X$.
    $y \leftarrow 0$
  Evaluera om olikheterna i E som innehåller y.
  $X \leftarrow X - \{y\}$

Algoritmen tilldelar alltid y ett värde som satisfierar minst hälften av olikheterna i $F(y)$. Alltså konstruerar den en lösning som satisfierar minst hälften av olikheterna i indata. Eftersom högst alla olikheterna kan satisfieras blir approximationskvoten för algoritmen 2. Tidskomplexiteten är polynomisk eftersom varje variabel och term bara behandlas en gång. □

Lösning till Övre gräns för approximation av homogena bipolära olikheter
Vi börjar med approximation av Max Hom Bipolar Sat LR$^{\geq}$. Ta en godtycklig bipolär vektor $\mathbf{x}$ och titta på antalet satisfierade olikheter om variablerna sätts till $\mathbf{x}$ respektive $-\mathbf{x}$. Om vänsterledet av en olikhet är positivt för $\mathbf{x}$ så är det negativt för $-\mathbf{x}$ och vice versa. Därför kommer en av dessa två vektorer att satisfiera minst häften av olikheterna, det vill säga uppnå approximationskvoten 2.

Denna triviala algoritm fungerar inte för Max Hom Bipolar Sat LR$^{>}$, för många relationer kan vara noll för båda vektorerna. Därför börjar vi med att söka upp en lösning med många nollskilda relationer. Approximationsalgoritmen för Max Sat LR$^{\geq}$ ovan kan modifieras så att den hittar en lösning $\mathbf{x}$ för vilken minst hälften av relationerna är nollskilda. Samma sak gäller då också $-\mathbf{x}$, och någon av dessa två vektorer måste då satisfiera minst en fjärdedel av alla relationer. □

Lösning till Undre gräns för approximation av binära olikheter
Visa detta genom att reducera Max Clique till Max Binary Sat LR$^{\geq}$ med en approximationsbevarande reduktion. Eftersom vi vet att Max Clique inte tillhör APX så kommer det att innebära att Max Binary Sat LR$^{\geq}\notin$APX.

Låt $G = (V, E)$ vara indata till Max Clique. För varje hörn $v_i \in V$ så konstruerar vi en variabel x_i och olikheten

$$x_i - \sum_{j \in N(v_i)} x_j \geq 1$$

där j är med i $N(v_i)$ om $v_j \neq v_i$ och $(v_i, v_j) \notin E$ (det vill säga v_j inte är granne med v_i). Nu har vi ett system med $|V|$ olikheter. Observera att den i-te olikheten är satisfierad om och endast om $x_i = 1$ och $x_j = 0$ för alla $j \in N(v_i)$.

Det är lätt att verifiera att om man får en s-klick $V' \subseteq V$ så kan man få en binär lösning som satisfierar dom s motsvarande olikheterna genom att sätta $x_i = 1$ om $v_i \in V'$ och $x_i = 0$ annars. Å andra sidan, om man får en binär lösning $\mathbf{x}$ som satisfierar s olikheter så kan man få en s-klick genom att låta V' bestå av alla hörn v_i som motsvarar satisfierade olikheter.

Denna reduktion är inte bara *approximationsbevarande* utan också *kostnadsbevarande* eftersom den bevarar målfunktionens värde helt och hållet.

Notera att denna reduktion också fungerar för $A\mathbf{x} > \mathbf{0}$ och $A\mathbf{x} = \mathbf{1}$, så vi kan alltså visa att Max Binary Sat LR$^{>}\notin$APX och att Max Binary Sat LR$^{=}\notin$APX. □

Algoritmer, datastrukturer och komplexitet, hösten 2015

Uppgifter till övning 12

Komplexitetsklasser och repetition

Uppgifter på komplexitetsklasser

co-NP-fullständighet Ett diskret tekniskt diagnosproblem kan modelleras på följande sätt: komponenttillstånd, systemtillstånd och omgivningstillstånd representeras av booleska variabler och själva systemet definieras med en boolesk formel φ. Man vet att i alla *möjliga* (fungerande) världar kommer systemvariablerna att ha värden så att φ är sann.

Anta att komponenttillstånden alla är tvåvärda och att en komponent c därför representeras av en boolesk variabel x_c (där sant betyder att komponenten fungerar och falskt att den är trasig). Vi vet att en komponent c är trasig om den formel som vi får om vi i φ sätter in värden för alla variabler som representerar kända komponent-, system- och omgivningstillstånd samt sätter x_c till sann inte är satisfierbar.

Visa att det är co-NP-fullständigt att avgöra ifall c är trasig.

Komplexitetsklassrelation PSPACE är komplexitetsklassen som består av alla språk för vilka det existerar en *deterministisk* turingmaskin som känner igen språket i polynomiskt *minne*. EXPTIME består av alla språk för vilka det existerar en *deterministisk* turingmaskin som känner igen språket i exponentiell tid. Visa att PSPACE$\subseteq$EXPTIME.

Blandade uppgifter från gamla tentor

Första uppgiften är en typisk första uppgift på teoritentan. Övriga uppgifter är typiska uppgifter på muntliga tentan.

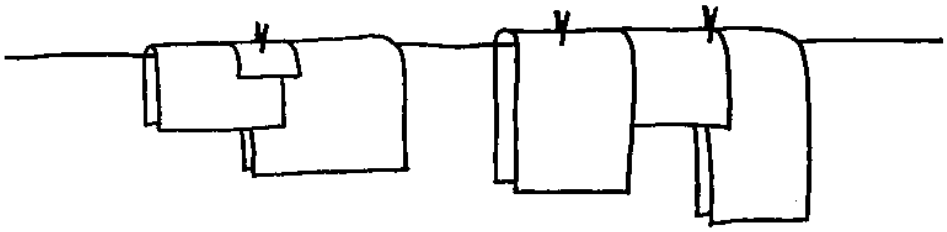
1. Är följande påståenden sanna eller falska? För varje deluppgift ger riktigt svar 1 poäng och ett övertygande bevisat riktigt svar 2 poäng,

   a) Problemet att avgöra ifall ett tal med n siffror är ett primtal ligger i komplexitetsklassen co-NP.

   b) Det finns en konstant $c > 1$ så att $n^3 \in O(c^{\log n})$.

   c) Binära träd implementerar man vanligen genom att införa två pekare i varje post (`left` och `right`). När man implementerar ternära träd (där varje nod har tre söner) går det inte att klara sig med mindre än tre pekare i varje post.

2. (algoritmkonstruktion, betyg C) På ett klädstreck har Viggo hängt n stycken handdukar av olika storlek. Viggo vill att handdukarna ska göra ett harmoniskt intryck för betraktaren. Därför var han mycket noga vid utplaceringen och har tagit mått på exakt var handdukarna hänger längs strecket, så att ingen ska flytta på dom. För att inte vinden ska störa ordningen vill Viggo säkra handdukarna med klädnypor, men han vill inte använda fler klädnypor än absolut nödvändigt. Det räcker ju att varje handduk kläms åt av en nypa, och två eller flera handdukar som hänger över varandra kan dela på samma nypa.



Beskriv en algoritm (i ord eller med pseudokod) som så snabbt som möjligt beräknar var på strecket man ska sätta nyporna för att det ska gå åt så få nypor som möjligt. Motivera att algoritmen är korrekt och analysera med enhetskostnad.

Indata är n stycken par av tal (v_1, h_1), (v_2, h_2), $\ldots$, (v_n, h_n) där varje par talar om var vänsterkanten respektive högerkanten på en handduk är. Alla mått avser avståndet från början av klädstrecket till en handdukskant. Paren är inte sorterade på något sätt. Utdata ska vara ett antal tal som anger var man bör sätta klädnyporna (en klädnypa antas ha bredd 0).

För att du ska få maximal poäng måste din algoritm gå i tid $O(n \log n)$.

3. (komplexitet, betyg C) I en stor organisation som Teknis finns det många grupper av personer, t ex lärare på Nada, lärare på F, elever på kursen Algoritmer, datastrukturer och komplexitet, medlemmar i Teknologkören, etc. Varje individ är med i minst en grupp, men kan vara med i många grupper. Nu vill rektor skapa en grupp av informatörer som snabbt ska kunna föra ut information till alla individer på Teknis. Han vill att varje grupp ska vara representerad i informatörsgruppen (dvs minst en medlem i varje grupp ska vara med i informatörsgruppen), men han vill samtidigt att informatörsgruppen ska vara så liten som möjligt.

   Detta är ett exempel på ett allmänt problem där man givet en uppsättning grupper vill hitta en så liten skara av representanter som möjligt.

   a) Formulera problemet matematiskt som ett mängdproblem och beskriv det samtidigt som ett beslutsproblem.

   b) Visa att problemet är NP-fullständigt.

4. (algoritmkonstruktion, betyg A; komplexitet, betyg C; svår uppgift!) Tumstocksproblemet kan beskrivas på följande sätt. En tumstock består av en kedja av träbitar som är fastsatta med gångjärn i ändarna. Det är möjligt (men inte nödvändigt) att vika tumstocken vid varje gångjärn. Det kan vara så att träbitarna som tumstocken är gjord av har olika längd. Då är det inte längre självklart hur man ska vika ihop tumstocken så att den blir så kort som möjligt (och går ner i snickarens ficka). Problemet är alltså att, givet en tumstock, vika ihop den så att den blir så kort som möjligt.

   a) Formulera tumstocksproblemet matematiskt som ett beslutsproblem.

   b) Visa att tumstocksproblemet är NP-fullständigt. Reducera till exempel mängdpartitioneringsproblemet till tumstocksproblemet.

   c) Anta att träbitarna har heltalslängder och att den längsta träbiten är $17n$, där n är antalet träbitar som tumstocken är sammansatt av. Bestäm komplexiteten för detta problem.

5. (komplexitet, betyg C) MAX 2$\wedge$SAT är ett optimeringsproblem som definieras som beslutsproblem på följande sätt.

INMATNING: Ett positivt heltal K mellan 1 och n samt n klausuler där varje klausul består av en enda literal eller två literaler kombinerade av operatorn $\wedge$. Exempel: $x_1 \wedge \overline{x}_3$, $x_2 \wedge x_3$.

PROBLEM: Finns det en variabeltilldelning som satisfierar minst K klausuler?

Visa att denna beslutsproblemsversion av MAX 2$\wedge$SAT är NP-fullständig. Du kan exempelvis använda dej av att MAX 2SAT – det motsvarande problemet med operatorn $\vee$ istället för $\wedge$ – är NP-fullständigt.

Lösningar till uppgifter på komplexitetsklasser

Lösning till co-NP-fullständighet
För att visa att problemet ligger i co-NP så ska vi visa att vi i polynomisk tid kan verifiera en nej-lösning, det vill säga att c inte är trasig. Vi vet att c inte är trasig om och endast om det finns en variabeltilldelning som satisfierar formeln. Om vi gissar variabelvärden behöver vi alltså bara verifiera att formeln med denna variabeltilldelning är sann. Detta går att göra i polynomisk tid.

För att visa att problemet är co-NP-svårt reducerar vi co-SAT, alltså problemet att avgöra ifall en given boolesk formel ϕ *inte* är satisfierbar. Eftersom SAT är NP-fullständigt så är per definition co-SAT co-NP-fullständigt.

Givet ϕ, konstruera ett system som innehåller den extra komponenten c (representerad av x_c) och som definieras av formeln $\varphi = \phi \vee \neg x_c$.

Notera nu att om x_c sätts till sann blir $\varphi = \phi$ så problemet att avgöra ifall c är trasig är precis samma som problemet att avgöra ifall ϕ inte är satisfierbar. Alltså är reduktionen korrekt. □

Lösning till Komplexitetsklassrelation
Om en turingmaskin använder polynomiskt minne finns det en konstant k så att antalet använda rutor på bandet är $O(n^k)$ där n är indatas längd. Om alfabetet består av tre tecken (0, 1, blank) så är antalet olika möjliga konfigurationer på bandet begränsat av $3^{O(n^k)}$, antalet möjliga platser för läs/skrivhuvudet är $O(n^k)$ och antalet möjliga tillstånd i turingmaskinen är ändligt, dvs $O(1)$. Det totala antalet konfigurationer för en turingmaskin som använder polynomiskt minne är alltså $O(n^k) \cdot 3^{O(n^k)}$, dvs exponentiellt i n. Eftersom turingmaskinen inte kan återkomma till samma konfiguration flera gånger (då skulle den gå i en oändlig slinga) så är detta också en övre gräns på tiden. Alltså kan varje problem som kan lösas med polynomiskt minne lösas i exponentiell tid. □

Lösningar till blandade uppgifter från gamla tentor

1. a) *Sant.* Problemet ligger i co-NP om komplementproblemet ligger i NP. Komplementproblemet är i detta fall att avgöra ifall ett tal med n siffror kan faktoriseras i minst två faktorer (större än 1). Detta problem ligger i NP eftersom en lösning (dvs en faktorisering av talet) kan verifieras i polynomisk tid (genom att man multiplicerar ihop faktorerna och kollar att produkten blir det givna talet).

   b) *Sant.* Om vi antar att $\log n$ är logaritmen i basen 2 så vet vi att $c^{\log n} = 2^{\log c^{\log n}} = 2^{\log n \log c} = n^{\log c}$. Om vi väljer $c \geq 8$ så är $\log c \geq 3$ och $n^3 \in O(n^{\log c}) = O(c^{\log n})$.

   c) *Falskt.* För allmänna träd räcker det med två pekare i varje post (`firstson` och `next`).

2. En girig algoritm löser problemet i tid $O(n \log n)$.

   1. Sortera alla paren dels med avseende på vänsterkanten och dels med avseende på högerkanten. Lagra resultaten i två listor, *Vsort* och *Hsort*. Håll under sorteringen reda på var varje post i den ena listan hamnar i den andra listan. Med heapsort tar detta tid $O(n \log n)$.

2. Så länge det finns några par kvar i listan *Hsort*:

   2.1 Ta det första återstående paret, säg (v, h). Eftersom paren har sorterats efter högerkanterna är (v, h) den handduk vars högerkant ligger längst till vänster.

   2.2 Sätt en klädnypa längst till höger på denna handduk, dvs precis till vänster om h.

   2.3 Ta bort alla par vars vänsterkant är till vänster om h, dvs ta bort alla handdukar som den just ditsatta klädnypan nyper fast. Detta gör man genom att plocka par från början av *Vsort* som ju är sorterad i stigande vänsterkantsordning. För varje par som plockas bort från *Vsort* ska samma par plockas bort från *Hsort*.

Till slut har alla par plockats bort och algoritmen avslutas. I steg 2 behandlas varje par bara en gång. Tiden för steg 2 blir alltså $O(n)$. Hela algoritmen tar därför tid $O(n \log n)$.

Korrekthet: Eftersom bara par som fått en klädnypa i sig blir borttagna så kommer alla par att sitta fast när algoritmen har genomförts. Nu ska vi bara visa att antalet använda klädnypor är minimalt. Den handduk som har den vänstraste högerkanten (dvs den som är allra först i *Hsort*) måste få en klädnypa i sig. Om man sätter fast handduken i punkten p så kommer alla handdukar med vänsterkant mindre än p att sitta fast, eftersom det inte finns någon handduk som har sin högerkant till vänster om p. För att sätta fast så många handdukar som möjligt så ska man välja p så stort som möjligt, dvs så nära handdukens högerkant som möjligt. Samma resonemang tillämpas sedan på dom övriga handdukarna.

3. a) Låt k vara ett positivt heltal, S vara mängden av personer och $C = \{C_1, \ldots, C_m\}$ vara dom m grupperna. Problemet är att hitta en delmängd $S' \subseteq S$ med högst k element så att $S' \cap C_i \neq \emptyset$ för $1 \leq i \leq m$. Detta problem har på engelska namnet *hitting set*.

   b) Problemet ligger i NP eftersom man kan gissa vilka k element som ska ligga i S' och verifiera att $S' \cap C_i \neq \emptyset$ för $1 \leq i \leq m$ i polynomisk tid.

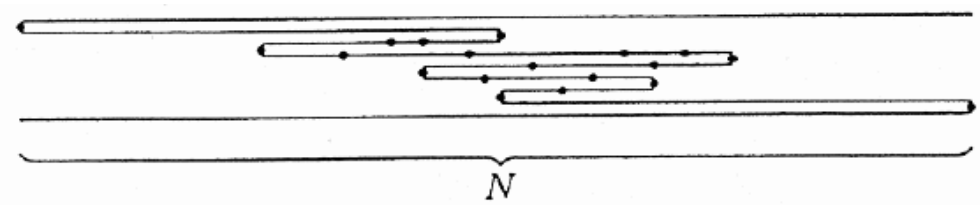
   Problemet är NP-svårt eftersom det är en generalisering av hörntäckningsproblemet som är känt NP-fullständigt. Givet en graf $G = (V, E)$, låt $S = V$ och $C = E$. En hörntäckning av storlek k motsvarar precis en delmängd $S' \subseteq S$ av storlek k som innehåller minst ett element från varje C_i.

4. a) Indata är n positiva tal $l_1, \ldots, l_n$, som i tur och ordning anger längderna av dom n träbitarna som ingår i tumstocken, samt ett positivt tal K som anger längden av snickarens ficka. Frågan är om det går att välja riktningar $r_1, \ldots, r_n$, där $r_i \in \{-1, 1\}$, för dom n träbitarna så att om man viker tumstocken enligt riktningarna så blir dess längd högst K, dvs $\max_{j \in [0..n]} \left\{ \sum_{i=1}^{j} r_i l_i \right\} - \min_{j \in [0..n]} \left\{ \sum_{i=1}^{j} r_i l_i \right\} \leq K$.

   b) Tumstocksproblemet tillhör NP eftersom man kan gissa värdena på $r_1, \ldots, r_n$ och verifiera att tumstockens längd högst är K i polynomisk tid.

   Vi visar att det är NP-svårt genom att reducera partitioneringsproblemet som är välkänt NP-fullständigt. Indata till partitioneringsproblemet är n positiva tal $t_1 \ldots, t_n$ och frågan är om talen kan delas upp i två grupper så att deras summa är lika.

   Givet $t_1 \ldots, t_n$, konstruera indata till tumstocksproblemet på följande sätt: $l_1 = N$, $l_2 = N/2$, $l_{i+2} = t_i$ för $1 \leq i \leq n$, $l_{n+3} = N/2$, $l_{n+4} = N$ och $K = N$, där $N = \sum_{i=1}^{n} t_i$. Visa nu att det finns en jämn partitionering om och endast om det finns en vikning av tumstocken så att längden blir högst N.



Om vi har en partitionering så kan vi vika tumstocken så att träbitar som motsvarar tal i den första gruppen viks åt vänster (riktning -1), bitar som motsvarar tal i den andra gruppen viks åt höger (riktning $+1$), $r_1 = -1$, $r_2 = 1$, $r_{n+3} = 1$, $r_{n+4} = -1$. Det är lätt att kontrollera att längden av tumstocken blir precis N, se figuren.

Omvänt, om vi har en vikning som gör att tumstocken ryms i fickan av storlek N, så kan vi anta att $r_1 = 1$. Om $r_1 = -1$ så tar vi först och byter tecken på alla riktningar, dvs vänder på tumstocken, som förstås fortfarande har samma längd. Eftersom längden av den första träbiten är N så måste nästa bit ha riktning $r_2 = -1$ och föra tillbaka till en position mitt på den första träbiten. På samma sätt ser vi att den sista och den näst sista träbiten måste vara riktade åt olika håll och att den näst sista träbiten börjar i en position som är mitt på den sista träbiten. För att längden av tumstocken ska vara högst N så måste uppenbarligen den andra träbitens ändes position överensstämma med den näst sista träbitens början. Detta innebär att dom mellanliggande bitarna måste börja och sluta på samma position, dvs $\sum_{i=3}^{n+2} r_i l_i = 0$, vilket kan skrivas som $\sum_{i:3\le i\le n+2 \wedge r_i=1} l_i - \sum_{i:3\le i\le n+2\wedge r_i=-1} l_i = 0$. Eftersom dessa träbitars längder precis motsvarar tal i partitioneringsproblemet så får vi en lösning till detta problem genom att vi partitionerar efter hur dom motsvarande träbitarna är riktade.

c) Problemet kan lösas i polynomisk tid med dynamisk programmering genom att man håller reda på alla delmängdssummor som kan uppkomma. Om den längsta träbiten är $17n$ vet vi att det för alla delsummor måste gälla att $-17n^2 \le \sum_{i=1}^{j} r_i l_i \le 17n^2$. Skapa en boolesk array $b[0..n, -17n^2..17n^2]$ där alla element från början är falska. Algoritmen ska fylla arrayen så att $b[j,k]$ är sant omm det finns värden på r_i så att $\sum_{i=1}^{j} r_i l_i = k$. Rekursionsekvationen blir $b[j,k] = b[j-1, k+l_j] \vee b[j-1, k-l_j]$. Skapa också en boolesk array $ends[0..n, -17n^2..17n^2, -17n^2..0, 0..17n^2]$ där $ends[j,k,l,r]$ är sant om det finns någon tumstocksvikning av dom j första träbitarna som slutar i position k, har sin vänstraste punkt i position l och sin högraste punkt i position r.

$b[0,0] \leftarrow$ sant
for $j \leftarrow 1$ **to** n **do for** $k \leftarrow -17n^2$ **to** $17n^2$ **do**
  if $b[j-1,k]$ **then**
    $b[j, k-l_j] \leftarrow b[j, k+l_j] \leftarrow$ sant
    for $l \leftarrow -17n^2$ **to** 0 **do for** $r \leftarrow 0$ **to** $17n^2$ **do**
      if $ends[j-1,k,l,r]$ **then**
        $ends[j, k-l_j, min(l, k-l_j), r] \leftarrow ends[j, k+l_j, l, max(r, k+l_j)] \leftarrow$ sant
$minlength \leftarrow 17n^2$
for $k \leftarrow -17n^2$ **to** $17n^2$ **do**
  if $b[n,k]$ **then**
    for $l \leftarrow -17n^2$ **to** 0 **do for** $r \leftarrow 0$ **to** $17n^2$ **do**
      if $ends[j,k,l,r] \wedge (r - l < minlength)$ **then** $minlength \leftarrow r - l$
return $(minlength)$

Det är lätt att verifiera att algoritmen returnerar den minsta tumstockslängd som kan uppnås. Algoritmen tar tid $O(n^7)$, vilket är polynomiskt (men ändå ganska långsamt!).

5. MAX 2∧SAT ligger i NP eftersom det är lätt att verifiera att en variabeltilldelning (en lösning) satisfierar minst K av klausulerna. Vi visar att det är NP-svårt genom att reducera MAX 2-SAT. Varje 2-SAT-klausul med en enda literal flyttar vi oförvanskad över till MAX 2∧SAT-probleminstansen. För varje klausul $l_i \vee l_j$ i MAX 2-SAT-instansen konstruerar vi tre klausuler i MAX 2∧SAT-probleminstansen: $l_i \wedge l_j$, $\overline{l}_i \wedge l_j$ och $l_i \wedge \overline{l}_j$. Vi ser att $l_i \vee l_j$ är sann omm en av dom tre konstruerade klausulerna är sann. Om $l_i \vee l_j$ är falsk så är alla dom tre konstruerade klausulerna falska. Alltså motsvarar antalet satisfierade MAX 2-SAT-klausuler precis antalet satisfierade MAX 2∧SAT-klausuler. Välj alltså målet K för det konstruerade problemet till samma värde som K för MAX 2-SAT-problemet.