Algoritmer, datastrukturer och komplexitet, hösten 2013
Lösningsförslag till mästarprov 1: algoritmer

1. **Hur kort kan texten bli?**
Den självklara giriga algoritmen placerar orden på raderna uppifrån och ned och byter rad först när ett ord inte får plats på samma rad.

```
TextPacking({w_1, ..., w_n}, len) =
    if n = 0 then return 0
    line ← 1
    pos ← w_1
    for i ← 2  to n do
        if pos + 1 + w_i > len then
            line ← line + 1
            pos ← w_i
        else pos ← pos + 1 + w_i
    return line
```

För att bevisa att en girig algoritm är optimal brukar man antingen visa att algoritmen i varje läge ligger före varje optimal algoritm (se beviset för att girig intervallschemaläggning är optimalt i avsnitt 4.1 i Kleinberg-Tardos) eller visa att varje optimal lösning kan transformeras till den giriga lösningen (se beviset för att Kruskals och Prims algoritmer ger optimala spännande träd i avsnitt 4.5 i Kleinberg-Tardos eller föreläsningsanteckningarna till föreläsning 12). Båda metoderna kan användas för att visa att girig textpackning är optimalt Låt oss genomföra den andra metoden.

Om $n = 0$ returnerar algoritmen 0 vilket är optimalt.

Anta att U_k är en optimal lösning (dvs utplacering av ord på optimalt antal rader) som överensstämmer med algoritmens utplacering av dom k första orden. Anta att ord $k+1$ i U_k har placerats på rad r_U med början i position p_U medan algoritmen placerat ordet på rad r_A med början i position p_A. Eftersom föregående ord placerades på samma sätt så vet vi att $r_U \geq r_A$ (för om ord $k+1$ hade fått plats på samma rad som ord k så hade algoritmen gjort det) och att om $r_U = r_A$ så är $p_U \geq p_A$ (för algoritmen har placerat ord $k+1$ så långt till vänster som möjligt på raden).

Om $r_U = r_A$ och $p_U > p_A$ så kan vi skjuta ordet till vänster så att det börjar på p_A. Denna modifierade U_k är då fortfarande en optimal lösning eftersom antalet rader inte ändrats och inga ord överlappar på grund av förskjutningen.

Om $r_U > r_A$ så måste $r_U = r_A + 1$ (för det kan aldrig vara optimalt att ha en tom rad i texten). I så fall kan vi flytta upp ord $k+1$ till rad r_A med start i position p_A, för ordet fick uppenbarligen plats där. Denna modifierade U_k är då fortfarande en optimal lösning eftersom antalet raderinte kan ha ökat och inga ord överlappar på grund av omplaceringen.

På detta sätt kan vi alltså alltid modifiera U_k till en lösning U_{k+1}, alltså en optimal lösning som överensstämmer med algoritmens utplacering av dom $k+1$ första orden.

Med induktion över k från 0 till n så kommer vi fram till att algoritmens utplacering av orden måste vara optimal med avseende på antalet rader.

Tidskomplexiteten är $O(n)$ eftersom algoritmen bara innehåller en enda slinga och den går $n-1$ varv där varje varv tar konstant tid med enhetskostnad.

2. **Optimal kedjevikning**

Problemet kan lösas med dynamisk programmering på många olika sätt. Här beskrivs ett enkelt sätt som tar kubisk tid, men det finns också kvadratiska algoritmer.

Låt $H[i,j]$ vara maximala antalet länkar som kan läggas där den sista länken går åt *höger* på rad i och slutar i position j. Låt $V[i,j]$ vara maximala antalet länkar som kan läggas där den sista länken går åt *vänster* på rad i och slutar i position j.

Låt $H[0,j] = 0$ och $V[0,j] = 0$ för alla j, eftersom inga länkar kan läggas på rad 0 (som är en fiktiv rad ovanför första raden).

Vi uttrycker $H[i,j]$ rekursivt genom att söka det antal länkar k på rad i som ger längst kedja. Att det på rad i ligger k länkar (åt höger) som slutar i position j betyder att kedjan måste ha vikts ner från föregående rad i position $j-3k+1$. Ingen position mellan $j-3k+1$ och j på rad i får då vara blockerad. Varje länk är 3 lång, varför $k \leq \lfloor j/3 \rfloor$. Länkarna i kedjan på föregående rad måste ha gått åt vänster, och längsta sådana kedjan som slutar i den sökta positionen står i $V[i-1, j-3k+1]$. Därmed utökas kedjan med k länkar på rad i. Om $V[i-1, j-3k+1] = 0$ betyder det att kedjan börjar på rad i.

För $V[i,j]$ får vi med samma idé k länkar åt vänster på rad i som börjar i position $j+3k-1$ och ingen position mellan j och $j+3k-1$ får vara blockerad. Eftersom raden är n lång måste $k \leq \lfloor (n-j01)/3 \rfloor$. Längsta kedjan som slutar åt höger i position $j+3k-1$ på föregående rad har $H[i-1, j+3k-1]$ länkar.

Ovanstående resonemang motiverar följande rekursioner:

$$H[i,j] = \begin{cases} 0 & \text{då } i = 0 \\ \max\limits_{1 \leq k \leq \lfloor j/3 \rfloor, B[i, j-3k+1..j]=0} (V[i-1, j-3k+1] + k) & \text{då } i > 0 \end{cases}$$

$$V[i,j] = \begin{cases} 0 & \text{då } i = 0 \\ \max\limits_{1 \leq k \leq \lfloor (n-j+1)/3 \rfloor, B[i, j..j+3k-1]=0} (H[i-1, j+3k-1] + k) & \text{då } i > 0 \end{cases}$$

Den längsta kedjan kan sluta var som helst i rutan, så det sökta maximala antalet länkar uttrycks som

$$\max_{1 \leq i \leq n, 1 \leq j \leq n} (V[i,j], H[i,j]).$$

Beräkningsordningen är radvis uppifrån och ned, parallellt i H och V.

```
Folding(B[1..n, 1..n]) =
  lmax ← 0
  for j ← 1 to n do
    H[0, j] ← 0; V[0, j] ← 0
  for i ← 1 to n do
    for j ← 1 to n do
      INV V och H beräknade för alla rader < i och på rad i för alla positioner < j;
          lmax är max av dessa.
      H[i, j] ← 0
      for k ← 1 to ⌊j/3⌋ do
        INV B[i, j − 3(k − 1) + 1..j] = 0; H[i, j] = max_{1≤p<k}(V[i − 1, j − 3p + 1] + p)
        if B[i, j − 3k + 3] = 1 or B[i, j − 3k + 2] = 1 or B[i, j − 3k + 1] = 1 then break
        if V[i − 1, j − 3k + 1] + k > H[i, j] then H[i, j] ← V[i − 1, j − 3k + 1] + k
      if H[i, j] > lmax then lmax ← H[i, j]
      V[i, j] ← 0
      for k ← 1 to ⌊(n − j + 1)/3⌋ do
        INV B[i, j..j + 3(k − 1) − 1] = 0; V[i, j] = max_{1≤p<k}(H[i − 1, j + 3p − 1] + p)
```

if $B[i, j+3k-3] = 1$ **or** $B[i, j+3k-2] = 1$ **or** $B[i, j+3k-1] = 1$ **then break**
if $H[i-1, j+3k-1] + k > V[i, j]$ **then** $V[i, j] \leftarrow H[i-1, j+3k-1] + k$
if $V[i, j] > lmax$ **then** $lmax \leftarrow V[i, j]$
return $lmax$

Korrektheten för algortimen bevisas i två steg. Den inledande argumentationen ovan visar att rekursionerna och uttrycket för det optimala värdet är korrekta. Att pseudokoden beräknar rekursionerna korrekt och att beräkningsordningen är okej ses enkelt med hjälp av invarianterna för slingorna.

Tidskomplexitet: $O(n^3)$ eftersom vi som mest har två nästlade slingor som går upp till n och en upp till $n/3$.

3. **Kan ämnet framställas?**
Det är mycket svårt att konstruera en rekursiv lösning på detta problem, och det svåra är att man måste ta hand om cykler då ett ämne tycks behöva sig själv för att framställas. Istället kan en iterativ lösning konstrueras med hjälp av en kö för alla ämnen som kan bildas av reaktioner som kan utföras (eftersom alla ämnen som behövs finns tillgängliga, antingen som råvaror eller som resultat av en redan gjord reaktion). För att vi ska veta vilka reaktioner som beror på ett ämne lagrar vi alla sådana beroenden i en kö för varje ämne. Och för att vi ska veta när alla ämnen som en reaktion behöver finns tillgängliga så inför vi en räknare för antalet behövda ämnen för varje reaktion. När vi tar hand om ett ämne i huvudkön (som alltså kan produceras) slår vi upp specialkön för det ämnet och går igenom den. För varje reaktion i specialkön minskar vi räknaren av behövda ämnen med ett, och om räknaren kommer ner till noll så stoppas reaktionens resultatämne in i kön.

Algoritmen blir så här:

ProduceSubstance($a[1..p], b, r[1..s]$) =
// Varje reaktion $r[i]$ består av resultatämnet $r[i].output$ och en mängd ämnen $r[i].input$
// som är nödvändiga för att reaktionen ska kunna utföras.
// $rawMaterial[i]$ är en boolesk array som talar om ifall ämne i är en råvara.
// $subsQueue[]$ är en kö för varje ämne som kan behöva produceras. Kön består av dom
// reaktioner som beror på ämnet, dvs har ämnet i sin input.
// $neededSubs[i]$ är heltalsarray som anger antalet återstående ämnen som behövs för
// att reaktion i ska kunna utföras.
// $mainQueue$ är en kö för ämnen som kan genereras och som därför behöver gås igenom.
$mainQueue \leftarrow$ **new** Queue
for $react \leftarrow 1$ **to** s **do**
foreach $substance \in r[react].input$ **do**
$rawMaterial[substance] \leftarrow$**false**
$subsQueue[substance] \leftarrow$ **new** Queue
for $i \leftarrow 1$ **to** p **do**
$rawMaterial[a[i]] \leftarrow$**true**
assert $rawMaterial[i]$ är sann för alla råvaror i och falsk för alla andra ämnen som är med
if $rawMaterial[b]$ **then return true**
for $react \leftarrow 1$ **to** s **do**
$noOfNeededSubstances \leftarrow 0$
foreach $substance \in r[react].input$ **do**
if not $rawMaterial[substance]$ **then**
$noOfNeededSubstances \leftarrow noOfNeededSubstances + 1$
$subsQueue[substance].put(react)$
if $noOfNeededSubstances = 0$ **then** $mainQueue.put(r[react].output)$
else $neededSubs[react] \leftarrow noOfNeededSubstances$
while not $mainQueue.isEmpty()$ **do**
$substance \leftarrow mainQueue.get()$

```
        if substance = b then return true
        while not subsQueue[substance].isEmpty() do
            react ← subsQueue[substance].get()
            neededSubs[react] ← neededSubs[react] − 1
            if neededSubs[react] = 0 then mainQueue.put(r[react].output)
    return false
```

Första nästlade slingan går igenom alla ämnen som ingår som indata i varje reaktion, vilket tar linjär tid i n. Andra slingan går igenom råvarunamnen i tid $O(p) \subseteq O(n)$. Tredje slingan går igenom alla ämnen i alla reaktioner igen. Sista nästlade slingan går igenom alla ämnen som kan genereras och för varje sånt ämne alla reaktioner det ämnet är indata till. Det är också linjärt i n. Tidskomplexiteten är alltså linjär i n.

Detta är förstås optimalt, för en trivial undre gräns är n, eftersom vi behöver titta på alla tal i indata för att kunna lösa uppgiften.

Om det sökta ämnet b är en råvara hittar algoritmen det (på raden efter **assert**). Alla reaktioner som kan genomföras med bara råvaror som behövda ämnen kommer att läggas in i *mainQueue* av slingan efter **assert**. Samtidigt lagras antalet behövda ämnen i reaktion *react* i *neededSubs*[*react*] och *react* lagras i dom behövda ämnenas specialköer. Slutligen gås *mainQueue* igenom. Varje ämne som ligger i kön kan genereras, och när det behandlas i kön kommer alla beroenden av det ämnet som indata i reaktioner att lösas upp. Om det sista beroendet för en reaktion därmed löses upp så läggs resultatet av den reaktionen in i *mainQueue*. På detta sätt går algoritmen systematiskt igenom alla reaktioner som kan genomföras och stoppar när det sökta ämnet b behandlas i kön. Om kön blir tom utan att b behandlats betyder det att det inte går att konstruera b, varför algoritmen helt korrekt returnerar falskt.