Algoritmer, datastrukturer och komplexitet, hösten 2019
Lösningsförslag till mästarprov 1: algoritmer

1. **Nattparkering av långt godståg**
   Låt oss bygga en girig algoritm som ledningen säger.

   Ta så många vagnar som möjligt och ställ på första spåret. Fortsätt sedan på samma sätt med resten av vagnarna och resten av spåren.

   ParkTrainPossible$(n, d, v_1, \ldots, v_n, m)$ =
   $used \leftarrow 0$
   $noOfParts \leftarrow 0$
   for $i \leftarrow 1$ **to** n **do**
   INV Vagn 1 till $i-1$ har parkerats på spår 1 till $noOfParts+1$,
   längsta parkerade tågdelen$\leq m$,
   på spår $noOfParts+1$ används utrymmet $used$.
   if $v_i > m$ **then return false**
   if $used + v_i > m$ **then** $noOfParts \leftarrow noOfParts + 1$; $used \leftarrow 0$
   $used \leftarrow used + v_i$
   if $used > 0$ **then** $noOfParts \leftarrow noOfParts + 1$
   return $(noOfParts \leq d)$

   Slingan i ParkTrainPossible går n varv, och i varje varv görs ett konstant antal operationer. Med enhetskostnad tar ett varv i slingan alltså konstant tid och tidskomplexiteten för hela ParkTrainPossible blir därmed $O(n)$.

   För att visa korrektheten för denna giriga algoritmen behöver vi visa att algoritmen för varje probleminstans (korrekt indata) avgör om det går att nattparkera tåget i probleminstansen på d spår av längd m. Vi ska dels visa att om algoritmen svarar ja (true) så finns det en nattparkering av längd m och dels att om det finns en nattparkering av längd m så svarar algoritmen ja (true).

   För det första, om algoritmen svarar ja så finns det en uppdelning där varje tågdel är högst m lång. Algoritmen konstruerar nämligen en sådan uppdelning. (Det kan vi se genom att notera att invarianten är sann i varje varv i slingan och även efter slingan. Om en vagn som är längre än m upptäcks är det omöjligt att nattparkera och nej (falskt) returneras omedelbart.)

   Eftersom vi kräver att det ska vara exakt d delar så kan vi, om antalet delar är mindre än detta, enkelt dela upp tågdelar som består av flera vagnar i flera delar ända tills det totalt är precis d delar. Dom nya delarna är alltid kortare än dom ursprungliga, så längsta tågdelen är fortfarande högst m lång. Första delen av beviset är därmed klart.

   Nu behöver vi bara visa att om det finns en uppdelning av tåget i d delar där varje tågdel är högst m lång så svarar algoritmen ja. Anta att vi har en sån uppdelning. Vi ska nu försöka modifiera denna uppdelning så att den överensstämmer med algoritmens lösning. Om det är samma som den algoritmen hittat så är saken klar. Om inte så antar vi att första spåret där algoritmens uppdelning skiljer sig från den givna är spår p. Eftersom algoritmen packat spår p girigt så finns det minst en vagn till på spår p i algoritmens uppdelning än i den givna uppdelningen. Det betyder att vi kan återskapa algoritmens lösning på dom p första spåren från den givna uppdelningen genom att flytta en eller ett par vagnar till spår p. Denna uppdelning måste fortfarande uppfylla att högst d spår används och att längsta tågdelen är högst m. Vi kan göra om denna operation gång på gång ända tills algoritmens lösning och den modifierade givna uppdelningen är identiska. Eftersom högst d spår används så kommer algoritmen att svara ja.

2. **Förvirrad skalbagge**

Pseudokod

Indata är ett naturligt tal n och en boolesk matris H[x,y] som för varje position anger om det finns ett hinder på den platsen.

Förberäkna G(u, v), antalet grannar till ruta (u, v) som inte innehåller något hinder. (Även grannar utanför nätet räknas: i figuren i uppgiftslydelsen är alltså G(3, 6) = 3.)

```
// Antal grannar till (u, v) som inte innehåller hinder
G(u, v):
    res := 0
    if u = x or not H[u+1, v] then res += 1
    if v = y or not H[u, v+1] then res += 1
    if u = 1 or not H[u-1, v] then res += 1
    if v = 1 or not H[u, v-1] then res += 1
    return res
```

Beräkna sannolikheten att skalbaggen befinner sig på en viss ruta vid ett visst tidssteg.

```
// Sannolikheten att skalbaggen befinner sig i ruta (u, v) vid tidsteg n.
P(u, v, n):
    if H[u, v] then return 0

    if n = 0 then
        if (u, v) = (1, 1) then
            return 1
        else
            return 0

    if 1<u<x and 1<v<y and H[u+1,v] and H[u,v+1] and H[u-1,v] and H[u,v-1] then
        return P(u, v, n-1) // hinder i alla grannrutor

    S := summan av P(u', v', n-1) / G(u', v'),
         där (u', v') är de grannrutor som ligger i
         rutnätet och inte innehåller något hinder.

    return S
```

Den sökta sannolikheten P(n) beräknas genom att summera alla P(u, v, n) för detta n.

```
// Sannolikheten att skalbaggen är kvar på rutnätet vid tidssteg n.
P(n):
    res := 0
    for u := 1 to x
        for v := 1 to y
            res += P(u, v, n)
    return res
```

Korrekthet

Vi visar att algoritmen är korrekt med hjälp av induktion.

Värdena P(u, v, 0) är uppenbarligen korrekta.

Skalbaggen kan endast befinna sig i ruta u, v vid tidssteg n om den befann sig i någon av de möjliga grannrutorna vid tid n-1. P(u, v, n) kan därför beräknas genom att väga samman

sannolikheterna för att skalbaggen befann sig i dessa rutor vid tid n-1 multiplicerat med sannolikheten att de går till ruta (u, v). Vilket är precis vad algoritmen gör.

Tidskomplexitet

Förberäkningen av G tar O(xy) tid och summeringen av delsannolikheterna vid tid n tar O(xy) tid.

För att uppnå polynomisk tidskomplexitet när vi beräknar P(u, v, n) använder vi dynamisk programmering och beräknar en tabell som innehåller samtliga värden P(u, v, n).

Detta kan vi göra med memoisering: vi lagrar resultatet av ett funktionsanrop P(u, v, n) och återanvänder det lagrade resultatet om samma parametrar återkommer. Med den tekniken kommer beräkningen att ta O(xyn) tid eftersom det totala antalet möjliga funktionsanrop är xyn, och varje sådant funktionsanrop beräknas på O(1) tid.

Den totala tidskomplexiteten (och minneskomplexiteten) blir alltså O(xyn).

Det går också att använda tabellmetoden för dynamisk programmering. I det fallet behöver man lagra samtliga resultat P(u, v, n-1) för att kunna beräkna värdena P(u, v, n). Beräkningsordningen är då ökande tid från 0 till n. Om bara föregående tidssteg lagras blir minnesåtgången bara O(xy).

3. **Optimal nattparkering av godståg**
Problemet kan lösas i omkring kubisk tid med dynamisk programmering (till exempel med delproblemen $M[a, b]$ som anger längsta tågdelens minsta längd då dom a första vagnarna parkeras på dom b första parkeringsspåren).

Men en mycket snabbare lösning fås med en binärsökning efter optimala längsta parkerade tågdelens längd, som vi kan kalla OPT. Om tågets totala längd är $L = \sum_{j=1}^{n} v_j$ så vet vi att $OPT \geq L/d$ (vilket uppnås om tåget går att dela i d precis lika långa delar) och att $OPT \leq L$ (om hela tåget parkeras på ett parkeringsspår). Vi kan alltså använda L/d och L som undre och övre gränser i en binärsökning, som efter $\log L$ steg kommer att ha ringat in den optimala lösningen, alltså OPT.

Givet en längd m kan vi avgöra om det går att göra en uppdelning av tåget i d delar så att längsta delen har längd högst m genom att använda algoritmen ParkTrainPossible från uppgift 1.

OptimalTrainParking$(n, d, v_1, \ldots, v_n)$ =
    $L \leftarrow 0$
    for $i \leftarrow 1$ **to** n **do**
        INV $L = \sum_{j=1}^{i-1} v_j$
        $L \leftarrow L + v_i$
    $minx \leftarrow \lceil L/d \rceil - 1$
    $maxx \leftarrow L$
    while $maxx - minx > 1$ **do**
        INV Det går att nattparkera med spårlängd $maxx$ men inte med spårlängd $minx$
        $midx \leftarrow (minx + maxx)/2$
        if ParkTrainPossible$(n, d, v_1, \ldots, v_n, midx)$ **then** $maxx \leftarrow midx$
        else $minx \leftarrow midx$
    return $maxx$

För att visa korrektheten för algoritmen behöver vi visa att för varje probleminstans (korrekt indata) returnerar algoritmen det minsta x så att det går att nattparkera tåget i probleminstansen på spår av längd x. Det ska alltså gå att nattparkera på spår av längd x men inte på spår av längd $x - 1$.

I lösningen till uppgift 1 ovan visades att ParkTrainPossible är korrekt, vilket vi kan använda oss av i detta bevis.

Invarianten för for-slingan visar att L är korrekt beräknad när slingan avslutas.

När while-slingan börjar är dess invariant uppfylld, eftersom det säkert går att nattparkera tåget på spår av längd L men inte på spår av längd $\lceil L/d \rceil - 1$. I slingan sätts $midx$ till medelvärdet av $minx$ och $maxx$, så $minx < midx < maxx$. Om det går att nattparkera på spår av längd $midx$ sätts $maxx$ om till detta värde (som är mindre än det tidigare värdet på $maxx$), annars sätts $minx$ om till detta värde (som är större än det tidigare värdet på $minx$). Invarianten är därmed fortfarande uppfylld.

I varje varv i while-slingan minskar $maxx - minx$ och när skillnaden är högst 1 så bryts slingan. Invarianten säger då att det går att nattparkera med spårlängd $maxx$ men inte med spårlängd $minx$. Eftersom $maxx - minx \leq 1$ så måste optimala lösningen för problemet vara $maxx$, vilket också är det värde som algoritmen returnerar. Därmed är korrektheten visad.

Låt oss nu analysera tidskomplexiteten. Talet L är summan av n stycken tal som vart och ett är mindre än n^2, varför $L < n^3$ och alla tal som används i algoritmen är begränsade av n^3, det vill säga har $O(\log n^3) = O(\log n)$ bitar. Bitkomplexiteten för slingan som beräknar L är därför $O(n \log n)$.

För att se hur många varv den andra slingan kan gå noterar vi att differensen $maxx - minx$ kommer att halveras i varje varv. Det gör att slingan högst kan gå $\log(L - (L/d - 1)) < \log L \in O(\log n)$ varv.

I varje varv i while-slingan görs en beräkning av $midx$ i tid $O(\log n)$ med bitkostnad samt ett anrop av den giriga algoritmen ParkTrainPossible.

Vi behöver analysera tidskomplexiteten för ParkTrainPossible med bitkostnad. Slingan i ParkTrainPossible går n varv, och i varje varv görs ett konstant antal taloperationer på tal som är högst $L \in O(n^3)$ stora. Med bitkostnad tar ett varv i slingan alltså $O(\log L) \subseteq O(\log(n^3) = O(\log n)$. ParkTrainPossible tar alltså tid $O(n \log n)$.

Vart och ett av dom $O(\log n)$ varven i while-slingar tar alltså tid $O(n \log n)$, så den totala tidskomplexiteten med bitkostnad blir $O(n \log^2 n)$.

Eftersom indata består av n tal som är mindre än n^2 och ett tal som är mindre än n så är antalet bitar i indata $O(n \log(n^2) + \log n) = O(n \log n)$. Varje algoritm som löser problemet måste titta på alla tal i indata (för svaret beror på alla dessa) så är en undre gräns för tidskomplexiteten indatas längd, alltså $O(n \log n)$. Algoritmens komplexitet är bara en faktor $\log n$ större än undre gränsen, vilket är tillåtet i denna uppgift.