Algoritmer, datastrukturer och komplexitet, hösten 2022
Lösningsförslag till mästarprov 1: algoritmer

1. **Elnät för notställsbelysning**
Vi representerar problemet enligt ledningen med en graf och använder Prims algoritm för att hitta det minimala spännande trädet i denna graf. Vi bygger upp grafen med ett hörn för varje notställ. Dra en kant mellan varje par av hörn och låt kantvikten vara längden på sladden som skulle behövas för att koppla ihop motsvarande notställ, enligt matrisen. Lägg till ett hörn för vägguttaget och en kant med vikt 150 mellan detta hörn och det hörn som motsvarar notställ n.

   Prims algoritm beskrivs och analyseras i Kleinberg-Tardos, avsnitt 4.5. Den tar som indata en graf (V, E) representerad med grannlistor där varje kant e har en vikt c_e, samt ett rothörn som i detta fall är vägguttagets hörn. Tidskomplexiteten för Prims algoritm är $O(|E| \log |V|)$.

   Dom riktade kanterna i det minimala spännande trädet kommer att motsvara lösningens sladdar och antalet uttag i grendosan blir gradtalet i motsvarande nod i trädet (vilket alltså är utgradtalet plus 1).

```
OptimalMusicStandLightning(M[1..n, 1..n])=
    NL[0] ← ((n, 150)) // grannlistan för hörn 0 innehåller bara en kant till hörn n med vikt 150
    for i ← 1  to n do
        NL[i] ← emptylist
        for j ← 1  to n do
            if M[i, j] > 0 then
                NL[i].append((j, M[i, j]))
    Tree ←Prim(NL, 0) // Anropar Prims algoritm med grafen och hörn 0 som rot
    for i ← 0  to n do
        outdeg[i] ← 0
        for (j, weight) ∈ Tree[i] do
            outdeg[i] ← outdeg[i] + 1
    Sol ← emptylist
    for i ← 0  to n do
        for (j, weight) ∈ Tree[i] do
            Sol.append(weight, outdeg[j] + 1, i, j)
    return Sol
```

   Tidskomplexitet

   Den skapade grafen har $n + 1$ hörn och $n(n - 1) + 1$ kanter. Att skapa grafen tar tid $O(n^2)$ och att beräkna minimala spännande trädet tar tid $O(n^2 \log n)$. Det spännande trädet har $n+1$ noder och n kanter. Efterarbetet består av att beräkna utgradtalet för varje nod, vilket tar tid $O(n)$, och att gå igenom varje kant i trädet och konstruera motsvarande sladd med grenuttag, vilket tar tid $O(n)$. Totala tidskomplexiteten blir $O(n^2 \log n)$.

   Korrekthet

   Algoritmen är korrekt om den uppfyller sin specifikation, dvs för varje tillåten indatamatris ska den producera en lista med sladdspecifikationer som motsvarar det minimala elnätet för notställsbelysning. Eftersom det finns positiva element i hela matrisen, utom på diagonalen, så finns det alltid en lösning till problemet. Det kan finnas flera optimala lösningar, men det är bara en av dessa som ska returneras.

2. **Viktiga möten**
Låt $start[i], slut[i]$ och $prio[i]$ vara starttid, sluttid och prioritet för möte i. Låt dessutom $prio[0], start[0]$ och $slut[0]$ vara 0.

Uppgift 2.1

Vi löser uppgiften genom att använda dynamisk programmering och dela upp problemet i delproblem $P[n]$, där $P[n]$ är den optimala totala prioriteten för möte 1 till n.

Delproblemen $P[n]$ kan uttryckas med hjälp av följande rekursion:

För $n = 0$ är $P[0] = 0$.

För $n > 0$ är $P[n] = \max(P[n-1], prio[n] + P[k])$, där k är det största $k < n$ för vilket $start[n] \geq slut[k]$.

Uppgift 2.2

En lämplig beräkningsordning är att beräkna arrayen P från vänster till höger. Följande pseudokod gör denna beräkning:

```
// Returnerar den maximala totala prioriteten för möte 1 till n.
MaxMeetingPriority(prio[1..n], start[1..n], slut[1..n], description[1..n])=
    prio[0] ← 0; start[0] ← 0; slut[0] ← 0
    P[0] ← 0
    for i ← 1  to n do
        k ← i − 1
        while start[i] < slut[k] do
              k ← k − 1
        P[i] ← max(P[i − 1], prio[i] + P[k])
    return P[n]
```

Tidskomplexitet

Den inre loopen exekveras max i gånger och den totala tidskomplexiteten blir därför mindre än summan $1 + 2 + \cdots + n$, vilket är $O(n^2)$.

Korrekthet

I basfallet, där antalet möten är noll, returneras 0, ett tal som är mindre än alla förekommande prioriteter.

Om $i > 0$ så kommer en optimal schemaläggning antingen att innehålla möte i eller inte. Anropet till max returnerar den största möjliga totala prioriteten i dessa två fall.

Observera att i fallet där möte i ingår så kan inget annat överlappande möte schemaläggas. Eftersom mötena är sorterade på sluttid så kommer while-loopen att hoppa över precis dessa möten.

Uppgift 2.3

Vi hittar lösningen på delproblem genom uppslagning i arrayen P. Observera att dessa värden då redan är beräknade.

Pseudokod för att hitta en optimal lösning

```
// S[i] anger om möte i är med i en optimal lösning
for i ← 1  to n do
   S[i] ← false
i ← n
while i > 0 do
      k ← i − 1
      while start[i] < slut[k] do
            k ← k − 1
      if P[i − 1] < prio[i] + P[k] then
         S[i] ← true
```

```
        i ← k
      else
        i ← i − 1
for i ← 1 to n do
  if S[i] then print i, description[i]
```

3. **Optimal analys av ringar i vas**

När en ring med radien c kastas ner i vasen kommer den att fastna på den den översta höjd i vasen som har en radie som är mindre än c. Det spelar ingen roll om vasen längre ner blir bredare igen. Därför börjar algoritmen med att sålla bort väggpunkter (x_i, y_i) där $x_i \geq x_d$ för något $d > i$, dvs $x_i \geq \min_{i<d\leq m} x_d$. Genom en binärsökning efter c mellan kvarvarande punkter hittar vi snabbt den översta väggpunkten (x_b, y_b) i indata där $x_b < c$. Vi kan sedan räkna ut vilken höjd ringen fastnar genom att sätta in i ekvationen för linjen mellan (x_b, y_b) och (x_{b+1}, y_{b+1}). Höjden blir då $y_b+(c-x_b)(y_{b+1}-y_b)/(x_{b+1}-x_b)$. Sållningen av indatapunkterna behöver bara göras en gång. Binärsökningen och höjdberäkningen görs en gång för varje ring som kastas. Eftersom utdata ska vara sorterade efter växande höjd gör vi slutligen en mergesortering av dom framräknade höjderna.

Här är pseudokod för denna algoritm:

```
RingsInVase((x_1, y_1), ..., (x_m, y_m), c_1, ..., c_n)=
  y_0 ← 0; x_0 ← 0
  s ← m; k[m] ← x_m; p[m] ← m
  for i ← m − 1 downto 1 do
    inv För i < j < s är x_j ≥ x_s och för s ≤ j ≤ m är k[j] = x_{p[j]} = min_{p[j]≤d≤m} x_d
    if x_i < x_s then
      s ← s − 1; k[s] ← x_i; p[s] ← i
  s ← s − 1; k[s] ← 0; p[s] ← 0
  assert För s ≤ j ≤ m är k[j] = x_{p[j]} = min_{p[j]≤d≤m} x_d och k[s] = 0
  for i ← 1 to n do
    inv För 1 ≤ j < i är r[j] lösningen (dvs höjden i vasen) för ring c_j
    if c_j > x_m then
      r[i] ← y_m
    else
        assert k[s] = 0 < c_i ≤ x_m = k[m] och k[s..m] är sorterad
        t ← BinarySearch(k[s..m], c_i) // hittar t så att k[t − 1] < c_i ≤ k[t]
        b ← p[t − 1]
        r[i] ← y_b + (c_i − x_b)(y_{b+1} − y_b)/(x_{b+1} − x_b)
  Mergesort(r[1..n])// Kleinberg-Tardos, avsnitt 2.4
  return r

BinarySearch(k[a..b], c)=
  PRE k[a] < c ≤ k[b] och k[a..b] är sorterad
  POST k[RES−1] < c ≤ k[RES] och a < RES ≤ b
  while b > a + 1 do
      inv k[a] < c ≤ k[b] och k[a..b] är sorterad
      m ← ⌊(a + b)/2⌋
      if c < k[m] then b ← m else a ← m
  return b
```

Indata består av m punkter och n ringar. Sållningen tar linjär tid i m. För varje ring görs en binärsökning, som tar tid $O(\log m)$. Mergesorteringen tar $O(n \log n)$, se Kleinberg-Tardos, avsnitt 5.1. Totalt får vi komplexiteten $O(m+n \log m+n \log n) = O(n \log n)$ eftersom $m \leq n$.

För att visa att $n \log n$ också är en undre gräns visar vi att vi kan sortera n godtyckliga positiva rationella tal med hjälp av ett anrop till algoritmen, och eftersom sortering har en undre gräns på $n \log n$ så kan inte algoritmen gå snabbare än $n \log n$.

Vi låter varje tal i indata motsvaras av en ring med talet som radie. Vi bygger en enkel vas med bara en diagonal sträcka som vägg med samma radie och höjd som dem största ringen. Den konstruerade vasen har radie h på höjd h och alla ringar ryms i vasen. Det betyder att en ring med radie c kommer att fastna på höjd c i vasen, så resultatet av anropet av RingsInVase blir en sorterad följd av ringarnas radier, dvs av talen i indata. Detta är därför en reduktion av sortering till ringproblemet, där själva reduktionen tar linjär tid i antalet indata.

Sort($a[1..n]$)=
    $x_1 \leftarrow \max_{1 \leq i \leq n} a_i$
    $y_1 \leftarrow x_1$
    for $i \leftarrow 1$ **to** n **do**
        $c_i \leftarrow a_i$
    return RingsInVase($(x_1, y_1), c_1, \ldots, c_n$)

För att motivera korrektheten för algoritmen RingsInVase behöver vi visa att för varje probleminstans (väggpunkter och ringradier) returnerar algoritmen en sorterad lista av höjderna som ringarna skulle fastna på i vasen.

Vi behöver undersöka invarianterna för forslingorna. Det är enkelt att visa att invariantuttrycken i pseudokoden ovan är invarianter och att invarianterna är sanna i första varvet i slingan. Slutligen konstaterar vi att om invarianten för sista slingan är sann när $i = n + 1$ så ligger korrekt beräknade höjder i $r[1..n]$, och efter anropet av Mergesort är dessa sorterade.

En alternativ algoritm är att först sortera ringarna efter minskande radie och sedan gå igenom vasens sida uppifrån och ned och behandla ringarna allteftersom vasens radie minskar. På det sättet behövs ingen extra datastruktur k och ingen binärsökning. Komplexiteten är densamma.