Algoritmer, datastrukturer och komplexitet, hösten 2015
Lösningsförslag till mästarprov 1: algoritmer

1. **Hantverkare**
   Låt $t_1, t_2, \ldots, t_n$ vara en tidsordning för att utföra jobben. Den totala kostnaden för denna ordning blir
   $$100(nt_1 + (n-1)t_2 + (n-2)t_3 + \ldots + t_n).$$
   Summan har sitt minsta värde när $t_1 \leq t_2 \leq \ldots \leq t_n$. Om $t_i > t_{i+1}$ blir nämligen summan mindre om vi byter plats på dessa två värden:
   $$\begin{aligned}(n-i+1)t_i + (n-i)t_{i+1} &= (n-i)(t_i + t_{i+1}) + t_i > \\ (n-i)(t_i + t_{i+1}) + t_{i+1} &= (n-i+1)t_{i+1} + (n-i)t_i.\end{aligned}$$
   Kostnaden blir alltså minimal om vi låter hantverkarna arbeta i växande tidsordning.

   Denna algoritm kan vi implementera genom att till exempel sortera tiderna med mergesort och sedan beräkna kostnaden enligt formeln ovan. Tidskomplexiteten blir då $O(n \log n)$.

   MergeSort($\{t_i\}$)
   return $100(nt_1 + (n-1)t_2 + (n-2)t_3 + \ldots + t_n)$

2. **Höravstånd**
   För att lösa problemet använder vi dynamisk programmering. Låt delproblemet $F[i,j]$ vara den minimala förflyttningen för dom i första personerna där person i flyttas till j. Först noterar vi att det aldrig kan vara optimalt för personer att flytta till vänster om $P[1]$ eller till höger om $P[n]$ (för det är bättre att stanna i $P[1]$ än att gå till vänster om den punkten, och motsvarande för $P[n]$). Man kan heller aldrig tjäna på att låta två personer gå om varandra, dvs om person i ursprungligen står till vänster om person j så kommer han efter flyttningarna fortfarande att stå till vänster om j.

   När $F[i,j]$ är beräknat för alla i, j så är den optimala förflyttningen $\min_{P[1] \leq x \leq P[n]} F[n,x]$.

   Basfallet är $F[1,j] = |j - P[1]|$. Om vi för alla x vet kostnaden för att flytta person 1 till och med $i-1$ så att person $i-1$ hamnar på x så får vi minimala kostnaden för att flytta person $1-i$ så att person i hamnar på j som $\min_{j-k \leq x \leq j} F[i-1,x] + |j - P[i]|$. Rekursionen blir
   $$F[i,j] = \begin{cases} |j - P[1]| & \text{om } i = 1 \\ \min_{j-k<x\leq j} F[i-1,x] + |j - P[i]| & \text{annars} \end{cases}$$

   Vi beräknar värdena i matrisen F radvis uppifrån och ner och inom varje rad från vänster till höger. Då finns alltid dom värden vi behöver använda redan beräknade.

   Så här blir algoritmen:

   for $j \leftarrow P[1]$ **to** $P[n]$ **do** $F[1,j] \leftarrow j - P[1]$
   assertion Basfallen beräknade enligt rekursionen
   for $i \leftarrow 2$ **to** n **do**
       invariant $F[z,j]$ beräknad enligt rekursionen för $1 \leq z < i$ och $P[1] \leq j \leq P[n]$
       for $j \leftarrow P[1]$ **to** $P[n]$ **do**
           $t \leftarrow \infty$
           $a \leftarrow \max(P[1], j-k+1)$
           for $x \leftarrow a$ **to** j **do**
               invariant $t = \min_{a \leq y < x} F[i-1,y]$
               if $F[i-1,x] < t$ **then** $t \leftarrow F[i-1,x]$
           $F[i,j] \leftarrow t + |j - P[i]|$
   $t \leftarrow \infty$
   for $x \leftarrow P[1]$ **to** $P[n]$ **do**
       invariant $t = \min_{P[1] \leq y < x} F[n,y]$
       if $F[n,x] < t$ **then** $t \leftarrow F[n,x]$
   return t

Tidskomplexiteten domineras av dom tre nästlade slingorna som går $n-1$, $P[n]-P[1]+1 \leq 2kn+1$ respektive $k-1$ varv. Totalt blir det $O(n^2k^2)$.

Rekursionens korrekthet motiveras ovan och det är enkelt att se att algoritmen implementerar rekursionen korrekt med hjälp av invarianterna.

3. **Mängder av anagram!**
Anta att ordlistan lagras i arrayen $w[1..m]$ där varje element är en post med fältet word som innehåller ordet och fältet key som beräknas av algoritmen. Orden består av bokstäverna a till ö i det svenska alfabetet.

Algoritmen består av fyra steg:
1. Sortera hela w med avseende på fältet word.
2. Sortera varje ords bokstäver och lagra i fältet key (*kolla* blir alltså *akllo*).
3. Sortera hela w med avseende på fältet key med en stabil sorteringsalgoritm.
4. Mata ut alla ord som ligger efter varandra och har samma värde på key

I pseudokod blir detta:

```
Sort(w[1..m]) med avseende på fältet word
for i ← 1  to m do
    w[i].key ← CopyOf(w[i].word)
    Sort(w[i].key) med avseende på bokstäverna
Sort(w[1..m]) med avseende på fältet key
prev ← w[1].key
for i ← 1  to m do
    if prev ≠ w[i].key then
        print newline
    print w[i].word+" "
print newline
```

Om mergesort eller heapsort används för sortering blir tidskomplexiteten för steg 1 och 3 $O(nm \log m)$ (m ord av längd högst n sorteras ihop), för steg 2 $O(mn \log n)$ (m ord av längd högst n sorteras var för sig) och för steg 3 $O(nm)$, alltså totalt $O(nm \log nm)$.

På föreläsning 16 beskrevs sorteringsalgoritmerna räknesortering och radixsortering. Algoritmen kan göras snabbare om man använder räknesortering (med f(a)=1, f(b)=2, ..., f(ö)=29) i steg 2, som då tar tid $O(nm)$. Steg 1 och 3 kan snabbas upp genom att man använder radixsortering (med n stycken iterationer av räknesortering), vilket tar tid $O(nm)$. Hela algoritmen tar därmed tid $O(nm)$. Detta är optimalt, för eftersom indatas storlek är nm i värsta fallet och man måste läsa hela indata för att lösa problemet så är $\Omega(nm)$ en undre gräns för problemet.

Korrekthet: Två ord är anagram om och endast om (bokstäverna i) orden sorterade är lika. Om man sorterar dom sorterade orden (som i steg 3) kommer anagramord därför att komma efter varandra. Steg 4 skriver ut alla ord som kommer efter varandra och som ser likadana ut när dom är sorterade på samma rad, dvs varje anagrammängd kommer på en egen rad. Orden kommer att skrivas ut i bokstavsordning eftersom dom sorterades i steg 1 och eftersom steg 3 inte förändrar ordningen mellan ord som har samma värde i fältet key.