Algoritmer, datastrukturer och komplexitet, hösten 2017
Lösningsförslag till mästarprov 1: algoritmer

1. **Turistplanering**
Vi använder enligt ledningen problemet bipartit matchning. Vi skapar en bipartit graf med ett vänsterhörn för varje dag och ett högerhörn för varje utflykt. Vi har en kant mellan hörnet för dag d och utflykt e om det är tillåtet att genomföra utflykt e dag d. I en matchning finns högst en kant för varje dag och utflykt, så en maximal matchning ger oss den bästa utflyktsplaneringen.

```
TouristPlanning(n, m, NotOkDaysPerExcursion : [1..m] → Subset of {1..n})=
    // Skapa en bipartit graf G = (U, V, E)
    U ← {u_1, ..., u_n}
    V ← {v_1, ..., v_m}
    E ← ∅
    for excursion← 1  to m do
        notOkDays←NotOkDaysPerExcursion(excursion)
        okDays← complement(notOkDays) // Komplementmängden är dagarna då utflykten kan göras
        for okDay ∈ okDays do
            E ← E ∪ {(okDay, excursion)}
    G ← (U, V, E)

    maxMatching← MaxBipartiteMatching(G) // Anropa algoritmen i Kleinberg-Tardos s 368
    // Sortera med avseende på dag, dvs indexet för kantens vänstra ändpunkt, Kleinberg-Tardos s 50
    MergeSort(maxMatching)
    for (u_d, v_e) ∈maxMatching do
        print 'Dag ', d, ' görs utflykt ', e
```

För varje utflykt finns n tänkbara dagar att utföra den. Antalet kanter i grafen är därför högst mn. Antalet hörn är $m + n$. Enligt kursboken Kleinberg-Tardos sida 370 kan en maximal bipartit matching hittas i tid $O(\text{antal kanter} \cdot \text{antal hörn})$, vilket betyder att tidskomplexiteten blir $O(mn(m + n))$. Att skapa grafen tar tid $O(mn)$ och att sortera matchingen tar tid $O(n \log n)$. Hela algoritmen går alltså i tid $O(mn(m + n))$.

2. **Linjär skogsavverkning**
Till att börja med låter vi $S[j]$ vara sammanlagda mängden virke som avverkas i ett kalhygge som slutar i punkt j, det vill säga $S[j] = \sum_{i=j-m+1}^{j} T[i]$. Detta är lätt att beräkna för alla j i linjär tid med hjälp av relationen $S[i-1] + T[i] = T[i-m] + S[i]$.

För att lösa problemet använder vi dynamisk programmering. Låt delproblemet $M[x, j]$ vara den maximala avverkningen för dom x första punkterna (över $0..x-1$) med exakt j stycken kalhyggen. Om $x < (m+1)j - 1$ finns det inte någon lösning.

När $M[x, j]$ är beräknat för alla x och j så är den optimala avverkningen $M[n, k]$.

Basfallet är $M[x, 0] = 0$ (då inget kalhygge finns). Vi får en lösning över $0..x-1$ med j kalhyggen, där det sista kalhygget slutar i punkt $x-1$, genom att utgå från en optimal lösning över $0..(x-m-2)$ med $j-1$ kalhyggen. Rekursionen blir

$$M[x,j] = \begin{cases} 0 & \text{om } j = 0 \\ S[m-1] & \text{om } x = m \text{ och } j = 1 \\ M[x-m-1, j-1] + S[x-1] & \text{om } x = (m+1)j - 1 \text{ och } j > 1 \\ \max(M[x-1,j], M[x-m-1, j-1] + S[x-1]) & \text{om } x > (m+1)j - 1 \end{cases}$$

Vi beräknar värdena i matrisen M kolumnvis från vänster till höger (från $j = 0$ upp till $j = k$) och inom varje kolumn uppifrån och ner (från $x = (m+1)j - 1$ upp till $x = n$).

Så här blir algoritmen:

$sum \leftarrow 0$
for $i \leftarrow 0$ **to** $m-1$ **do** $sum \leftarrow sum + T[i]$
$S[m-1] \leftarrow sum$
for $i \leftarrow m$ **to** $n-1$ **do** $S[i] \leftarrow S[i-1] + T[i] - T[i-m]$
for $x \leftarrow 1$ **to** n **do** $M[x,0] \leftarrow 0$
assert Basfallen ($j = 0$) beräknade enligt rekursionen
for $j \leftarrow 1$ **to** k **do**
  $x \leftarrow (m+1) \cdot j - 1$
  if $j = 1$ then $M[x,j] \leftarrow S[x-1]$ **else** $M[x,j] \leftarrow M[x-m-1, j-1] + S[x-1]$
  assert $M[(m+1)j-1, j]$ uträknat enligt rekursionen, liksom $M[x,i]$ för $1 \leq i < j$ och $(m+1)j - 1 \leq x \leq n$
  for $x \leftarrow (m+1) \cdot j$ **to** n **do**
    $M[x,j] \leftarrow \max(M[x-1,j], M[x-m-1, j-1] + S[x-1])$
  assert $M[x,i]$ uträknat enligt rekursionen för $1 \leq i \leq j$ och $(m+1)j - 1 \leq x \leq n$
return $M[n,k]$

Att beräkna $S[i]$ för alla i tar $O(n)$. Beräkning av varje $M[x,j]$ tar konstant tid. M-matrisen innehåller $(k+1)n \in O(kn)$ element. Totalt blir komplexiteten $O(kn)$.

Rekursionens korrekthet och beräkningsordningen motiveras ovan och det är enkelt att se att beräkningen av basfall och rekursionssteg sker korrekt enligt assertion-satserna i pseudokoden.

3. **Union av intervallmängder**
Sortera alla ändpunkter

Börja med att sortera de $2n$ start- och ändpunkterna från alla intervall. Håll reda på om en punkt är en start- eller ändpunkt och sortera så att att startpunkterna kommer före ändpunkterna vid samma värde.

Sorteringen kan göras på $O(n)$ tid med hjälp av samsortering (som i merge sort) om vi utnyttjar det faktum att vi startar med redan sorterade listor:

- Samsortera listorna två och två. Detta tar $O(n)$ tid. I pseudokod:

```
MergeEndPointSets(EndPointSet eSet[k]):
    EndPointSet res[]
    if isOdd(k): eSet[k] = empty set
    i = 0
    while i < k:
        res[i/2] = Merge(eSet[i], eSet[i+1])
        i = i + 2
    return res
```

- Fortsätt på detta sätt tills bara en sorterad lista återstår.

```
while length(eSet) > 1:
   eSet[] = MergeEndPointSets(eSet[])
```

Detta behöver upprepas högst 7 gånger eftersom vi startar med högst 100 listor och $\log_2 100 < 7$. Den totala sorteringstiden blir alltså $O(n)$.

Producera svaret

Gå igenom den sorterade listan och håll reda på hur många intervall (count) som täcker den aktuella punkten.

- När count går från 0 till 1 så går vi in i unionen.

- När count går från 1 till 0 så lämnar vi unionen.

Här är algoritmen i pseudokod:

```
P = eSet[0] // sorted list of interval endpoints
S = empty set
count = 0

for p in P:
    assert count stycken intervall täcker området precis till vänster om punkten p.
           Om count>0 anger start startpunkten för det aktuella resultatintervallet.
           Intervall med slutpunkt till vänster om p är avklarade och unionen ligger i S.
    case count == 0:
        start = p
        count++
    case count == 1:
        if p is a startpoint
            count++
        else
            add interval [start, p) to S
            count--
    case count >= 2:
        if p is a startpoint
            count++
        else
            count--

return S
```

Analys

Sorteringen tar $O(n)$ tid. I den andra delen av algoritmen går vi igenom $2n$ punkter och gör konstant arbete vid varje iteration. Det totala tidskomplexiteten blir alltså $O(n)$.

Detta är optimalt eftersom varje algoritm som löser detta problem måste kunna producera ett resultat som består av n intervall.

Korrektheten av algoritmen är enkel att visa med hjälp av den angivna invarianten.