

Algoritmer, datastrukturer och komplexitet

Övning 8

Anton Grensjö
grensjo@csc.kth.se

5 november 2015

Kursplanering

F21: Introduktion till komplexitet

F22: Formella definitioner, turingmaskiner

F23: Oavgörbarhet

Ö8: Mästarprov 1, oavgörbarhet

F24: Cooks sats

Labb 3 bonusdatum 10/11

F25: NP-fullständigheidsbevis

F26: NP-reduktionsvisualisering

Ö9: NP-fullständigheidsbevis

Idag

- Genomgång av Mästarprov 1
- Oavgörbarhet

Uppgift 1: Hantverkare

- Ditt kök ska renoveras! Till din hjälp har du n stycken hantverkare med varsitt tilldelat uppdrag.
- Hantverkarnas respektive uppdrag tar tid t_1, t_2, \dots, t_n tid att utföra.
- Endast ett uppdrag kan utföras åt gången.
- Alla hantverkare dyker upp samtidigt, och du debiteras både för arbetstid och väntetid.
- Varje hantverkare kostar 100 kr per timme.

Uppgift Designa en girig algoritm som beräknar den minimala kostnaden för renoveringen.

Beskriv algoritmen med pseudokod, förklara varför den fungerar och analysera tidskomplexiteten.

Uppgift 1: Hantverkare

Lösning

- Hantverkarna måste utföra sina uppdrag i någon ordning. Låt k_i vara indexet för det i :te uppdraget som utförs.
- Det uppdrag som utförs först tar tiden t_{k_1} , och under den tiden är det 1 person som arbetar och $n - 1$ personer som väntar.
- Kostnaden under denna tid är alltså $100 \cdot n \cdot t_{k_1}$.
- Under nästa uppdrag är det en person mindre som behöver vänta, så kostnaden under denna tid blir $100(n - 1)t_{k_2}$.
- Den totala kostnaden för denna ordning blir:

$$K = 100 (nt_{k_1} + (n - 1)t_{k_2} + (n - 2)t_{k_3} + \dots + t_{k_n})$$

Uppgift 1: Hantverkare

Lösning

$$K = 100 (nt_{k_1} + (n-1)t_{k_2} + (n-2)t_{k_3} + \dots + t_{k_n})$$

- Frågan är alltså vilken ordning $\{k_i\}_{i=1}^n$ som minimerar K .
- K har sitt minimala värde när uppdragen utförs i ordning av ökande tid, dvs:

$$t_{k_1} \leq t_{k_2} \leq \dots \leq t_{k_n}$$

- Algoritm: Sortera listan av tider och beräkna kostnaden enligt formeln ovan.

function HANTVERKARE(t_1, t_2, \dots, t_n)

MERGESORT(t_1, t_2, \dots, t_n)

$cost \leftarrow 0$

for $i = 1$ **to** n **do**

$cost \leftarrow cost + (n - i + 1)t_i$

return $cost$

Uppgift 1: Hantverkare

Tidskomplexitet

```
function HANTVERKARE( $t_1, t_2, \dots, t_n$ )  
  MERGESORT( $t_1, t_2, \dots, t_n$ )  
   $cost \leftarrow 0$   
  for  $i = 1$  to  $n$  do  
     $cost \leftarrow cost + (n - i + 1)t_i$   
  return  $cost$ 
```

- Mergesort har tidskomplexiteten $\Theta(n \log n)$.
- Vi räknar med enhetskostnad, så varje steg i for-slingen går på konstant tid. Vi har n iterationer, så for-slingen går på tid $\Theta(n)$.
- Mergesort dominerar, så totala tidskomplexiteten är $\Theta(n \log n)$.

Uppgift 1: Hantverkare

Korrekthetsresonemang

- Korrekthet: varför ger den sorterade ordningen $(t_{k_1} \leq t_{k_2} \leq \dots \leq t_{k_n})$ minst kostnad?
 - Antag att vi har en ordning som ger minimalt K , men där $t_{k_i} > t_{k_{i+1}}$ för något i .
 - Undersök vad som händer om vi byter plats på det i :te och $i + 1$:te uppdraget.

$$(n - i + 1)t_{k_i} + (n - i)t_{k_{i+1}} = (n - i)(t_{k_i} + t_{k_{i+1}}) + t_{k_i} >$$

$$(n - i)(t_{k_i} + t_{k_{i+1}}) + t_{k_{i+1}} = (n - i + 1)t_{k_{i+1}} + (n - i)t_{k_i}$$

- Kostanden blir alltså lägre om vi byter plats på uppdragen.
- Motsägelse! Ordningen kan därmed inte ha varit optimal.
- Slutsats: En ordning som ger minimalt K måste uppfylla $t_{k_i} \leq t_{k_{i+1}}$ för varje i , dvs vi måste ha: $t_{k_1} \leq t_{k_2} \leq \dots \leq t_{k_n}$.

Uppgift 2: Höravstånd

- n personer står på en tallinje, på heltalskoordinater mellan 0 och $2kn$.
- Två personer kan höra varandra om deras avstånd är **strikt** mindre än k .
- Vi vill flytta på personerna på ett sådant sätt att alla par av personer kan kommunicera (möjligen med hjälp av en eller flera mellanhänder).
- Vilken är den minimala summan av förflyttningar som behövs för att detta ska bli uppfyllt?

Indata ges som n, k och $P[1..n]$, där $P[i]$ beskriver ursprungspositionen för person i . Det är givet att $P[1] \leq P[2] \leq \dots \leq P[n]$.

Konstruera en algoritm som löser problemet, analysera tidskomplexitet i n och k samt motivera korrekthet.

Uppgift 2: Höravstånd

Insikter

- Notera att det aldrig lönar sig för personerna längst ut att flytta sig längre bort.
- Det lönar sig heller aldrig för två personer att flytta sig förbi varandra.
- Antag att vi redan har placerat ut personerna $1, 2, \dots, i - 1$ på ett korrekt sätt. Det enda som just nu spelar roll för placeringen av person i är var person $i - 1$ står (avståndet får inte vara för stort).
- Om vi bestämmer oss för att placera ut person i på plats j så kan vi enkelt beräkna kostnaden för detta: $|j - P[i]|$.
- Men hur ska vi veta vilken plats j vi ska välja?
- Vi kanske kan tänka rekursivt här? Dela in i delproblem på något sätt?
- Låt delproblemet vara: Vad blir den minimala förflyttningen för de i första personerna, givet att den i :te personen placeras på plats j ?

Uppgift 2: Höravstånd

Rekursion

- Det här börjar likna dynamisk programmering!
- Låt $F[i, j]$ = den minimala förflyttningen för de i första personerna, om person i flyttas till plats j .
- Basfall: $F[1, j] = |j - P[1]|$.
- Vad blir $F[i, j]$ generellt?
 - Vi ska placera ut person i på plats j . Det kostar $|j - P[i]|$.
 - Vad kostar det att placera ut de $i - 1$ första personerna? Det beror på var person $i - 1$ placerats.
 - Om person $i - 1$ placerades på positionen x är kostnaden $F[i - 1, x]$.
 - Iterera över alla möjliga positioner x för person $i - 1$ (dvs $j - k < x \leq j$), och välj positionen med lägst $F[i - 1, x]$.

$$F[i, j] = \begin{cases} |j - P[1]| & \text{om } i = 1 \\ \min_{j-k < x \leq j} F[i - 1, x] + |j - P[i]| & \text{annars} \end{cases}$$

Uppgift 2: Höravstånd

Rekursion

$F[i, j]$ = den minimala förflyttningen för de i första personerna, om person i flyttas till plats j .

$$F[i, j] = \begin{cases} |j - P[1]| & \text{om } i = 1 \\ \min_{j-k < x \leq j} F[i-1, x] + |j - P[i]| & \text{annars} \end{cases}$$

- Svaret är $\min_{P[1] \leq x \leq P[n]} F[n, x]$.
- Beräkningsordning? Radvis fungerar bra.

Uppgift 2: Höravstånd

Pseudokod

function HÖRAVSTÅND($n, k, P[1..n]$)

for $j \leftarrow P[1]$ **to** $P[n]$ **do**

▷ Basfall.

$F[1, j] \leftarrow j - P[1]$

for $i \leftarrow 2$ **to** n **do**

▷ Beräkna $F[i, j]$ radvis.

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**

▷ Iterera över alla möjliga val av position x för föregående person.

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**

▷ Hitta det slutliga svaret.

if $F[n, x] < t$ **then**

$t \leftarrow F[n, x]$

Uppgift 2: Höravstånd

Tidskomplexitet

```

function HÖRAVSTÅND( $n, k, P[1..n]$ )
  for  $j \leftarrow P[1]$  to  $P[n]$  do
     $F[1, j] \leftarrow j - P[1]$ 
  for  $i \leftarrow 2$  to  $n$  do
    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
        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
    if  $F[n, x] < t$  then
       $t \leftarrow F[n, x]$ 

```

- De nästlade slingorna kommer dominera.
- Den yttre kör $n - 1$ varv.
- Den mellersta kör maximalt $P[n] - P[1] + 1 \leq 2kn + 1$ varv.
- Den innersta kör $k - 1$ varv.
- Tidskomplexitet: $\mathcal{O}(n^2k^2)$.

Uppgift 3: Mängder av anagram

- Givet: En lista av m stycken ord (endast små bokstäver, a-ö), med längd $\leq n$.
- m och n är oberoende av varandra.
- Skriv ut alla mängder av anagram i ordlistan, med en mängd per rad.
 - Dvs alla ord i listan som är anagram av varandra ska stå på samma rad.
 - Orden på varje rad ska stå i bokstavsordning.
 - Ordningen mellan raderna spelar ingen roll.
 - Ett ensamt anagram ska stå på en egen rad.
- Beskriv algoritmen med pseudokod, analysera tidskomplexiteten, motivera korrekthet.
- Algoritmen ska vara optimal, visa därför att övre och undre gräns sammanfaller.

Uppgift 3: Mängder av anagram

Lösning

- Hur kontrollerar vi om två ord är anagram?
 - Sortera bokstäverna i respektive ord, kolla om resultatet blev lika.
- Låt orden lagras i en array $w[1..m]$, där varje element har ett fält word som innehåller ordet, samt ett fält key.
 - 1 Sortera w med avseende på fältet word.
 - 2 Sortera varje ords bokstäver och lagra i fältet key (key kommer alltså vara samma för ord som är anagram).
 - 3 Sortera w med avseende på fältet key med en stabil sorteringsalgorithm.
 - 4 Gå igenom w och skriv ut alla ord, där de med samma key skrivs ut på samma rad.

Uppgift 3: Mängder av anagram

Pseudokod

```
function ANAGRAMMÄNGDER( $w[1..m]$ )  
  SORT( $w[1..m]$ ) med avseende på fältet word.  
  for  $i \leftarrow 1$  to  $m$  do  
     $w[i].key \leftarrow w[i].word$   
    SORT( $w[i].key$ ) med avseende på bokstäverna.  
  SORT( $w[1..m]$ ) med avseende på fältet key.  
   $pref \leftarrow w[1].key$   
  for  $i \leftarrow 1$  to  $m$  do  
    if  $prev \neq w[i].key$  then  
      PRINT(newline)  
      PRINT( $w[i].word + " "$ )  
  PRINT(newline)
```

Uppgift 3: Mängder av anagram

Tidskomplexitet

- 1 Sortera w med avseende på fältet $word$.
 - 2 Sortera varje ords bokstäver och lagra i fältet key .
 - 3 Sortera w med avseende på fältet key med en stabil sorteringsalgoritm.
 - 4 Gå igenom w och skriv ut alla ord, där de med samma key skrivs ut på samma rad.
- Om vi använder t.ex. mergesort blir tidskomplexiteten för steg 1 och 3 $\mathcal{O}(nm \log m)$, för steg 2 $\mathcal{O}(mn \log n)$ och för steg 4 $\mathcal{O}(nm)$. Totalt: $\mathcal{O}(nm \log nm)$.
 - Detta är inte optimalt. En lösning med denna tidskomplexitet gav “mindre fel” på mästarprovet.
 - Om vi istället använder radixsort i steg 1 och 3, samt räknesortering i steg 2 slipper vi log-faktorerna och får tidskomplexitet $\mathcal{O}(nm)$.

Uppgift 3: Mängder av anagram

Undre gräns

- En övre gräns för vår algoritms tidskomplexitet är alltså $\mathcal{O}(nm)$.
- Vi vill visa att en undre gräns för varje möjlig algoritm är $\Omega(nm)$, vilket skulle innebära att vår algoritm är optimal.
- Insikt: För att kunna lösa problemet måste vi titta på alla bokstäver. Varför?
 - Om vi inte tittar på alla bokstäver i något ord kan vi inte avgöra vilka ord det är anagram med.
- Således är $\Omega(nm)$ en undre gräns för problemet, vilket vi ville visa.

Oavgörbarhet

Teori

- Ett problem som har en algoritm som för *alla instanser* kan hitta en lösning i ändlig tid kallas **avgörbart**.
- Ett problem som inte kan lösas i ändlig tid av någon algoritm kallas **oavgörbart**.
- För beslutsproblem talar man om avgörbarhet. För övriga problem talar man istället om beräkningsbarhet (definieras analogt).
- Motsägelsebevis: För att bevisa ett påstående p , antag $\neg p$ och härled motsägelse. Då måste p vara sant.

Oavgörbarhet

Teori

Stopproblemet

Givet ett program P och indata X , kommer P någonsin att terminera om det startas med X ?

Stopproblemet är oavgörbart.

Bevis.

Antag motsatsen, dvs att det finns en algoritm $\text{STOPP}(P, X)$ som löser stopproblemet. Konstruera följande program:

function $\text{META}(P)$	Vad händer om vi anropar $\text{META}(\text{META})$?
if $\text{STOPP}(P, P)$ then	Om META terminerar så går programmet in
while true do	i en oändlig loop. Omöjligt!
else	Om META inte terminerar så går det in i
return	else-satsen och terminerar. Omöjligt!

Således måste antagandet vara falskt. □

Oavgörbarhet

Teori

Att bevisa oavgörbarhet med hjälp av reduktion:

- Säg att vi vill visa att ett problem P är oavgörbart.
- Antag att P är avgörbart.
- Reducera stopproblemet till P . Vi kan nu lösa stopproblemet genom att lösa P !
- ...men det är bevisat att stopproblemet är oavgörbart.
- Detta är alltså en motsägelse, och P måste vara oavgörbart.

Problem 1

Problemet **orm i kakel** tar som indata en mängd T av kakelplattstyper och två punkter p_1 och p_2 i planet. Frågan är om det går att användande endast kakeltyper i T kakla en orm som ringlar sig från p_1 till p_2 , som inte bryter kakelmönstret någonstans och som hela tiden håller sig i det övre halvplanet.

Detta problem är oavgörbart.

Är följande varianter av ormproblemet avgörbara eller oavgörbara?

- a) Indata utvidgas med en fjärde parameter, en kakeltyp $t \in T$. Första kakelplattan (den som täcker p_1) måste vara av typ t .

Antag att denna variant är avgörbar. Då kan vi lösa det ursprungliga ormproblemet genom att göra ett anrop till detta för varje kakeltyp.
Motsägelse.

Svar: Oavgörbart.

Problem 1

b) Indata utvidgas med ett tal N och frågan utvidgas med bivillkoret att ormen måste bestå av **högst** N plattor.

Avgörbart, ty vi kan nu lösa problemet med totalsökning, då det bara finns ett ändligt antal möjliga ormar.

c) Indata utvidgas med ett tal N och frågan utvidgas med bivillkoret att ormen måste bestå av **åtminstone** N plattor.

Oavgörbart. **Bevis:**

- Antag att denna variant är avgörbar.
- Då kan vi lösa det ursprungliga problemet genom att anropa denna variant med $N = 1$.
- Men det ursprungliga problemet är oavgörbart, så det är en motsägelse.
- Således kan inte denna variant vara avgörbar.



Problem 2

Fråga: Finns det något explicit program P så att det givet y är avgörbart huruvida P stannar på indata y ?

Betrakta följande program:

```
1: function  $P(y)$   
2:   return
```

Uppenbarligen så finns det massor av sådana program.

Problem 3

Fråga: Finns det något explicit program P så att det givet y är oavgörbart huruvida P stannar på indata y ?

Låt P vara en interpretator och indata y vara ett program x följt av indata y' till det programmet.

$P(y)$ beter sig alltså precis som programmet x skulle göra på indata y' , och det är ju oavgörbart huruvida ett program x stannar på ett visst indata y' . Svaret är alltså ja.

Problem 4

Fråga: Om programmet x stannar på tomt indata så låter vi $f(x)$ vara antalet steg innan det stannar. Annars sätter vi $f(x) = 0$. Definiera nu $MR(y)$, den maximala körtiden över alla program vars binära kodning är mindre än y .

$$MR(y) = \max_{x \leq y} f(x)$$

Är MR beräkningsbar?

Nej. Vi vet att det är oavgörbart huruvida ett program stannar på blankt indata. Reducera detta problem till $MR(y)$.

- Notera att om programmet y stannar, så gör det det på högst $MR(y)$ steg.
- Simulera därför y på blankt indata i $MR(y)$ steg och kolla om det stannar. Om det inte gör det så vet vi att det aldrig stannar.

Problem 4

StopBlank(y) =

$s \leftarrow MR(y)$

if $s = 0$ **then return false**

else

simulera y på blankt indata i s steg (eller tills y stannar)

if y stannade **then return true**

else return false

Problem 5

Problem: Visa att funktionen MR i föregående uppgift växer snabbare än varje rekursiv (beräkningsbar) funktion.

Visa närmare bestämt att det för varje rekursiv funktion g finns ett y så att $MR(y) > g(y)$.

Ledtråd: Tänk på föregående uppgift. Använd motsägelsebevis.

- Antag motsatsen, dvs att det finns en rekursiv funktion g så att $g(y) \geq MR(y)$ för alla y .
- Betrakta nu lösningen till föregående problem. Eftersom $g(y) \geq MR(y)$ kan vi lika gärna simulera programmet i $g(y)$ steg istället för $MR(y)$ steg.
- Detta betyder alltså att programmet med den förändringen är beräkningsbart, men det vet vi att det inte kan vara. Motsägelse.

Problem 6

Fråga: Anta att en turingmaskin M , indata X (som står på bandet från början) och en heltalskonstant K är givna. Är följande problem avgörbart eller oavgörbart?

Stannar M på indata X efter att ha använt högst K rutor på bandet (en använd ruta får skrivas och läsas flera gånger)?

Detta är faktiskt avgörbart! Varför?

- Turingmaskinen måste hålla sig inom K rutor på bandet \implies Turingmaskinen har endast ett ändligt antal möjliga konfigurationer (över dessa rutor).
- Om maskinen har m tillstånd, så är antalet konfigurationer $m \cdot K \cdot 3^K$.
- Simulera maskinen $m \cdot K \cdot 3^K + 1$ steg. Kontrollera att den inte rör sig över fler än K rutor.
- Om den stannar inom denna tid svarar vi ja. Om den inte har stannat måste den ha återkommit till en gammal konfiguration, och vara fast i en oändlig slinga. Svara nej.

Problem 7

- En **rekursivt uppräknelig mängd** definierades på föreläsningen som ett språk som kan kännas igen av en funktion vars ja-lösningar kan verifieras ändligt.
- En alternativ definition är en mängd som är uppräknelig (elementet kan numreras med naturliga tal) och kan produceras av en algoritm.

Uppgift: Använd den senare definitionen och visa att varje rekursiv mängd också är rekursivt uppräknelig.

Vi vill visa att om en mängd S är rekursiv (det finns en beräkningsbar algoritm som kan avgöra vilka element som tillhör S) så är S uppräknelig och kan produceras av en algoritm.

Så, om S är en rekursiv mängd, så finns det en algoritm $A(x)$ som returnerar true om $x \in S$. Vi antar att elementen i S lagras som binära strängar.

```
1: for  $i \leftarrow 0$  to  $\infty$  do  
2:   if  $A(i)$  then write  $i$ 
```

Problem 8

Uppgift: Den **diagonaliserade stoppmängden** består av alla program p som stannar på indata p . Visa att denna mängd är rekursivt uppräknelig.

Visualisering: oändlig tabell.

- 1: **for** $i \leftarrow 0$ **to** ∞ **do**
- 2: **for** $p \leftarrow 0$ **to** i **do**
- 3: Simulera beräkningen $p(p)$ under i steg
- 4: **if** $p(p)$ stannar inom i steg **then write** p

Notera: samma p kommer skrivas ut många gånger. Enkelt att fixa.
Hur?

Nästa gång

- Redovisning av teoriuppgifter för labb 4
- NP-fullständighetsbevis