Komplexitet
&
Korrekthet

Klassificering och jämförelse av körtider

Klassificering och jämförelse av körtider

- Vi har två algoritmer
  - $a_1(n)$ med körtid $n^2 + 7n + 128$
  - $a_2(n)$ med körtid $n^2 + 12n + 10$
- Körtiderna är olika men domineras av n^2-termen för stora n
- Ekvivalent effektivitet iom båda körtiderna är kvadratiska uttryck av indatastorleken
- Det är ofta svårt att hitta ett exakt uttryck för körtiden
- Vi vill istället generalisera detta

Hur kan vi enkelt beskriva algoritmernas körtid på ett sätt som fångar denna typ av ekvivalens?

Klassificering och jämförelse av körtider

- Ordo-notation! (eng. Big-O notation)
- För både a_1 och a_2 från föregående slide så har vi $a_1(n) \in O(n^2)$ och $a_2(n) \in O(n^2)$
- Skrivs ibland slarvigt med likhetstecken, $f(n) = O(n^2)$

Definition:

Uttrycket $f(n) \in O(g(n))$ säger att

"$f(n)$ är begränsad överifrån av $g(n)$ (gånger någon konstant) för stora n"

Formellt:

$$f(n) \in O(g(n))$$

$$\iff$$

$$\text{det finns några } n_0 \geq 0 \text{ och } c > 0 \text{ så att } f(n) < c \cdot g(n) \text{ för alla } n \geq n_0$$

Beräkning med ordo-notation; låt säga att vi har

$$f(n) = 20 + 15n + 10n^2$$

Om $n > 1$ så

$$f(n) < 20n^2 + 15n^2 + 10n^2 = 45n^2$$

Låt då $n_0 = 2, c = 45$:

$$f(n) < 45 \cdot n^2, n \geq 2 \iff f(n) \in O(n^2)$$

- Big-O ger bara *en* övre gräns, så om $f(x) \in O(n^2)$ så har vi även $f(x) \in O(n^3)$
- dvs $O(n^2) \subset O(n^3)$
- För att ge en undre gräns använder vi Ω-notation:

$f(n) \in \Omega(g(n)) \iff f(n)$ är begränsad underifrån av $g(n)$ för stora n

- Vi kan kombinera dessa med Θ för att få en mer exakt beskrivning av körtiden:

$f(n) \in \Theta(g(n)) \iff f(n)$ är begränsad överifrån *och* underifrån av $g(n)$ för stora n

- Beskriver körtiden ”exakt”, $\Theta(n^2)$ är skilt från $\Theta(n^3)$

Tidskomplexitet

- $\{\text{Körtid för A}\} \in O(f(n)) \iff A$ har *tidskomplexitet* $O(f(n))$
- På samma sätt kan vi tala om *minneskomplexitet*

Tidskomplexitet

- Vi talar oftast om tidskomplexiteten i det *värsta fallet*, men det finns även *bästa fall* och *genomsnittlig*
- Ibland är alla dessa samma, ibland inte
- Ett extremt exempel: $\mathrm{bogosort}$
  - T_{best} :
  - T_{worst} :
  - T_{avg} :

Tidskomplexitet

- Vi talar oftast om tidskomplexiteten i det *värsta fallet*, men det finns även *bästa fall* och *genomsnittlig*
- Ibland är alla dessa samma, ibland inte
- Ett extremt exempel: $\mathrm{bogosort}$
  - $T_{\mathrm{best}}(n) : O(1)$
  - $T_{\mathrm{worst}}(n) :$
  - $T_{\mathrm{avg}}(n) :$

Tidskomplexitet

- Vi talar oftast om tidskomplexiteten i det *värsta fallet*, men det finns även *bästa fall* och *genomsnittlig*
- Ibland är alla dessa samma, ibland inte
- Ett extremt exempel: $\mathrm{bogosort}$
  - $T_{\mathrm{best}}(n) : O(1)$
  - $T_{\mathrm{worst}}(n) : O(\infty)$
  - $T_{\mathrm{avg}}(n) :$

Tidskomplexitet

- Vi talar oftast om tidskomplexiteten i det *värsta fallet*, men det finns även *bästa fall* och *genomsnittlig*
- Ibland är alla dessa samma, ibland inte
- Ett extremt exempel: $\mathrm{bogosort}$
  - $T_{\mathrm{best}}(n) : O(1)$
  - $T_{\mathrm{worst}}(n) : O(\infty)$
  - $T_{\mathrm{avg}}(n) : O(n \cdot n!)$

Tidskomplexitet - exempel

findMax(L, n)

$m \leftarrow -\infty$

for $i \leftarrow 1$ **to** n **do**

  if $L_i > m$ **then**

    $m \leftarrow L_i$

return m

- Vi gör ett konstant antal operationer utanför slingan (k)
- Vi gör ett konstant antal operationer för varje iteration (m)
- Slingan körs n gånger
- Vi gör $k + mn$ operationer
- Tidskomplexiteten blir $O(k + mn) = O(n)$
- Hur hade det sett ut om vi hade en nästlad slinga?

Mästarsatsen

Mästarsatsen

- Antag att vi har en rekursiv algoritm
- Algoritmen anropar sig själv a gånger med indata minskad med en faktor b
- Algoritmen gör ytterligare operationer som har körtid $f(n) = O(n^d)$
- Då är körtiden för algoritmen

$$T(n) = aT(n/b) + f(n)$$

Mästarsatsen

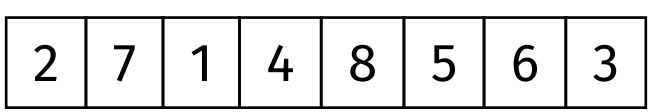
$$T(n) = aT(n/b) + f(n), \quad f(n) = O\big(n^d\big)$$

$$T(n) = \begin{cases} \Theta(n^d) & \text{om } a < b^d \\ \Theta(n^d \log n) & \text{om } a = b^d \\ \Theta(n^{\log_b a}) & \text{om } a > b^d \end{cases}$$

Merge-sort

- Algoritm för att sortera en lista
- Sorterar delar av input och sammanfogar dessa (därav *merge*)
- Är en typ av *divide-and-conquer*-algoritm

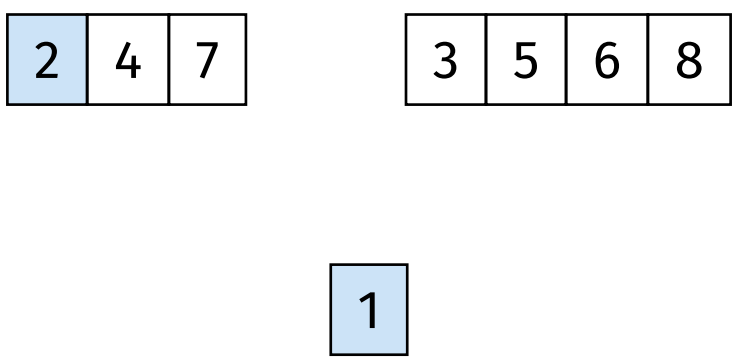
Exempel



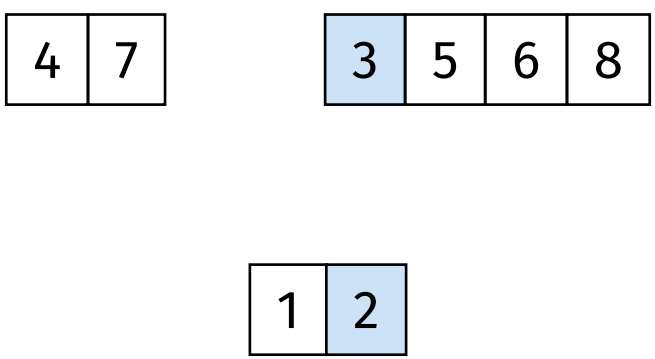
Dela indatan

Sortera varje dellista

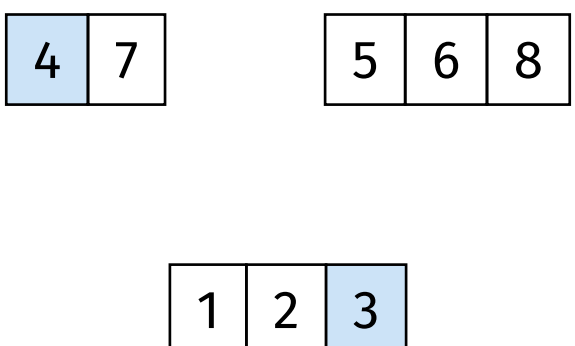
Kombinera resultatet



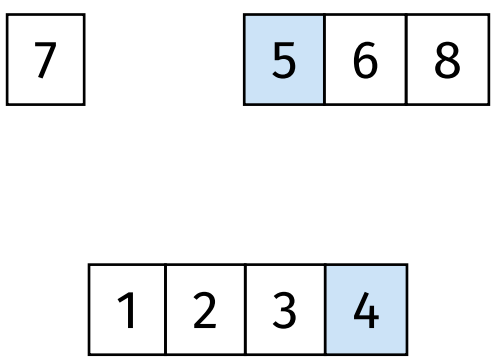
Kombinera resultatet



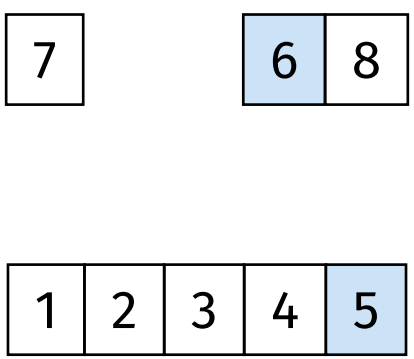
Kombinera resultatet



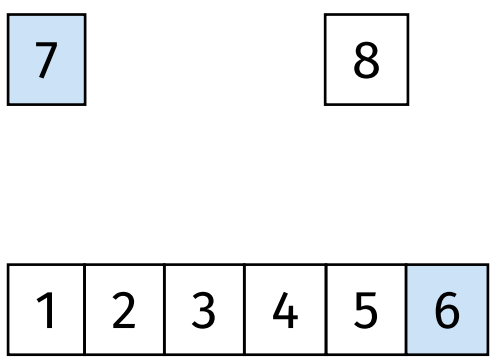
Kombinera resultatet



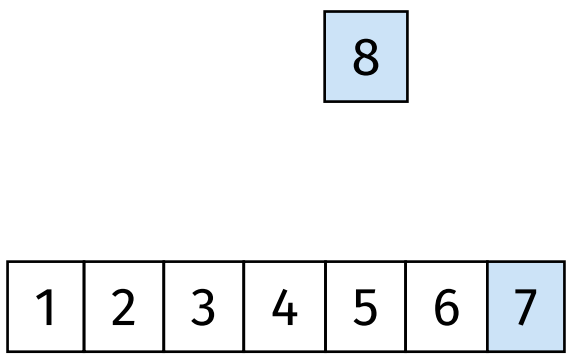
Kombinera resultatet



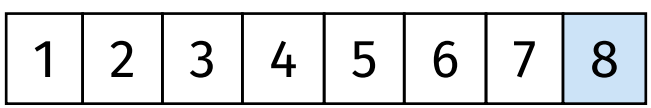
Kombinera resultatet



Kombinera resultatet



Klart!



Mästarsatsen & Merge-sort

- Vi gör två rekursiva anrop $\Rightarrow a = 2$
- Vi delar indatan i två $\Rightarrow b = 2$
- Vi sammanfogar resultaten i $O(n) \Rightarrow d = 1$

$$T(n) = \begin{cases} \Theta(n^d) & \text{om } a < b^d \\ \Theta(n^d \log n) & \text{om } a = b^d \\ \Theta(n^{\log_b a}) & \text{om } a > b^d \end{cases}$$

$$a = b^d \Rightarrow T(n) = O\big(n^d \log n\big) = \underline{O(n \log n)}$$

Komplexitetsklasser

Komplexitetsklasser

- En gruppering av problem med liknande tids- eller minneskomplexitet
- Det handlar om *beslutsproblem* (ja/nej)
- Exempel:
  - **P** = problem som kan *lösas* i polynomisk tid ($O(n^k)$)
  - **NP** = problem som kan *verifieras* i polynomisk tid

NP-problem

- Exempel på NP-problem:

$$\textit{Kan en graf } G = (V, E) \textit{ nodfärgas med } k \textit{ färger?}$$

- Det finns ingen känd algoritm i P, men vi kan verfiera en godtycklig lösning i polynomisk tid
- Indatan kallas för *instans* och lösningen kallas för *certifikat*
- Vad blir dessa i detta fall?
- Hur kan vi konstruera en *verifierare* (algoritmen som verifierar lösningen)?
- Gäller P $\subseteq$ NP?

NP-svåra problem

- **NP-svårt problem** = problem som är minst lika svårt som ett problem i NP
- Men hur vet vi om ett problem är ”svårare” än ett annat?

NP-svåra problem

- Svar: vi använder *reduktioner*
- En reduktion är en algoritm som omvandlar en instans för ett problem A till en instans för ett problem B. Om algoritmen är polynomisk säger vi att

$$A \leq_p B$$

- Exempel: Hamiltoncykel $\leq_p$ Hamiltonstig i grafer
- Ett problem B är **NP-svårt** om $A \leq_p B$ för varje problem A i NP, det vill säga, varje NP-problem kan reduceras till B.

NP-fullständig problem

- Ett problem är **NP-fullständigt** om det ligger i NP *och* är NP-svårt
- Exempel på NP-fullständiga problem:
  - *Graffärgning*: kan noder färgas med k färger så att grannar inte har samma färg?
  - *Satisifierbarhet*: t.ex. finns x_1, x_2, x_3 så att $(x_1 \wedge x_2) \vee \overline{x_3}$?
  - *Hamiltonstig*: finns en stig i en graf som går genom varje nod exakt en gång?
- Alla NP-fullständiga problem kan reduceras till varandra (varför?)

P=NP?

- Vi vet att $P \subseteq NP$
- Ingen vet om $P = NP$
- De flesta tror dock att $P \neq NP$

- Varför spelar det någon roll?
- Problem i P representerar problem som (i allmänhet) kan lösas i rimlig tid i verkligheten
- Om vi t.ex. hittar en polynomisk algoritm som löser ett NP-fullständigt problem, vet vi att $P = NP$, och vi kan lösa alla problem i NP i rimlig tid
- Primtalsfaktorisering ligger i NP, och RSA bygger på antagandet att primtalsfaktorisering tar lång tid

Korrekthet

Hur kan man visa att en algoritm gör det den ska göra?

- **findMax** är en funktion som hittar det största värdet i en lista med heltal
- Hur visar vi att den är korrekt?

findMax(L, n)

$m \leftarrow -\infty$

$i \leftarrow 1$

while $i \leq n$ **do**

  if $L_i > m$ **then**

    $m \leftarrow L_i$

  $i \leftarrow i + 1$

return m

Slinginvarianter

- Svar: vi använder en *slinginvariant*.
- Ett påstående som är sant i början och i slutet av varje iteration.
- Om det är sant innan slingan vet vi att det är sant efter slingan.

Slinginvarianter

- Vad skulle vara en bra slinginvariant i den här koden?

findMax(L, n)

$m \leftarrow -\infty$

$i \leftarrow 1$

while $i \leq n$ **do**

  if $L_i > m$ **then**

    $m \leftarrow L_i$

  $i \leftarrow i + 1$

return m

Slinginvarianter

findMax(L, n)

pre L är en lista med heltal av längd n.

post m är det största värdet i L.

$m \leftarrow -\infty$

$i \leftarrow 1$

while $i \leq n$ **do**

inv $m \geq L_k$ för $1 \leq k < i$

if $L_i > m$ **then**

$m \leftarrow L_i$

$i \leftarrow i + 1$

return m

Matematisk induktion - recap

- Bevismetod som liknar dominoeffekten
- Vi vill bevisa ett påstående $P(n)\ \forall n \geq m$
- Består av tre delar:
  - *Induktionsbas*: Vi visar att $P(m)$ gäller
  - *Induktionsantagande*: Vi antar att $P(k)$ för något k
  - *Induktionssteg*: Vi visar att $P(k) \Rightarrow P(k+1)$
- Då gäller $P(n)\ \forall n \geq m$

Matematisk induktion - exempel

Vi vill visa att

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2} \ \forall n \geq 0.$$

Matematisk induktion - exempel

Steg 1: *Induktionsbas*

$$\sum_{k=0}^{0} k = 0 = \frac{0(0+1)}{2}.$$

Steg 2: *Induktionsantagande*

$$\sum_{k=0}^{m} k = \frac{m(m+1)}{2} \text{ för något } m.$$

Matematisk induktion - exempel

Steg 3: *Induktionssteg*

$$\sum_{k=0}^{m+1} k = m+1+\sum_{k=0}^{m} k = m+1+ \overbrace{\frac{m(m+1)}{2}}^{\text{Induktionsantagande}}$$

$$= \frac{2m+2+m^2+m}{2} = \frac{m^2+3m+2}{2} = \frac{(m+1)(m+2)}{2}$$

Enligt induktionsprincipen gäller påståendet för alla $n \geq 0$.

Induktion i korrekthetsbevis

- Induktion kan även användas för att bevisa korrekthet för rekursiva funktioner.

findMaxRecursive(L, n)
  if $n = 1$ **then**
    return L_1
  $M \leftarrow$**findMaxRecursive**$(L, n-1)$
  if $L_n > M$ **then**
    return L_n
  else
    return M

Induktion i korrekthetsbevis

- Vad är påståendet vi vill bevisa?
- Vad är induktionsbas, -antagande och -steg?

findMaxRecursive(L, n)
  if $n = 1$ **then**
    return L_1
  $M \leftarrow$**findMaxRecursive**(L, $n - 1$)
  if $L_n > M$ **then**
    return L_n
  else
    return M

Induktion i korrekthetsbevis

Vi vill visa att

$$\mathbf{findMaxRecursive}(L, n) = \max_{i \leq n} \{L_i\} \ \forall n \geq 1$$

Varför $n \geq 1$*?*

Induktionsbas:

if $n = 1$ **then**
  return L_1

$$\mathbf{findMaxRecursive}(L, 1) = L_1 = \max_{i \leq 1}\{L_i\}$$

Induktion i korrekthetsbevis

Induktionsantagande:

$$\textbf{findMaxRecursive}(L, k) = \max_{i \leq k}\{L_i\} \text{ för något } k$$

Induktionssteg:

$M \leftarrow$ **findMaxRecursive**$(L, n-1)$
if $L_n > M$ **then**
  return L_n
else
  return M

$$\begin{aligned}
&\mathbf{findMaxRecursive}(L, k+1) \\
&= \max\{L_{k+1}, \mathbf{findMaxRecursive}(L, k)\} \\
&= \max\left\{ L_{k+1}, \underbrace{\max_{i \le k}\{L_i\}}_{\text{Induktionsantagande}} \right\} \\
&= \max_{i \le k+1}\{L_i\}
\end{aligned}$$