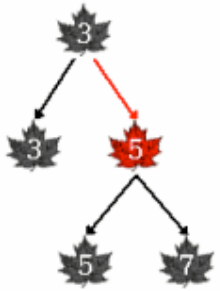
Algoritmedesign

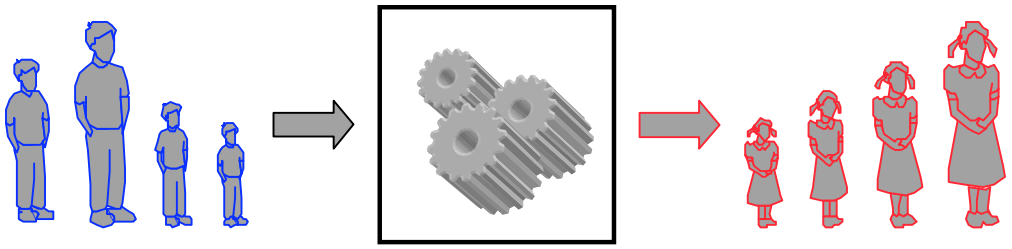
med internetanvendelser

ved

Keld Helsgaun



Analyse af algoritmer



Input **Algoritme** **Output**

En **algoritme** er en trinvis metode til løsning af et problem i endelig tid

Algoritmebegrebet

D. E. Knuth (1968)

En **algoritme** er en *endelig*, *præcis* og *effektiv procedure* med *output*

- *Endelig*: Algoritmen vil terminere efter et endeligt antal trin
- *Præcis*: Hvert trin skal være defineret i præcise og utvetydige termer
- *Effektiv*: Hvert trin skal kunne udføres i endelig tid
- *Procedure*: Algoritmen består af trin
- *Output*. Algoritmen må producere et resultat

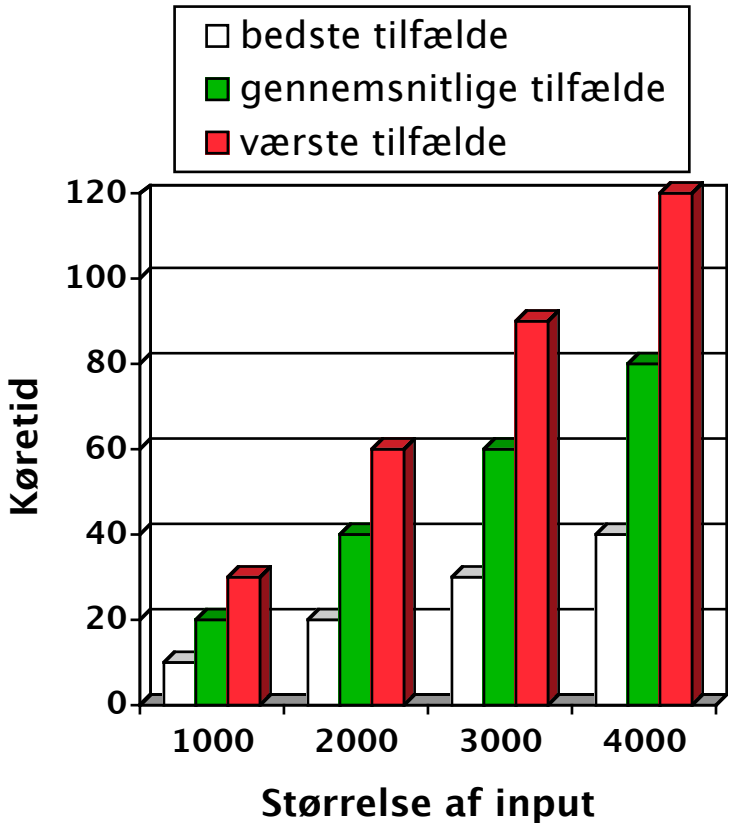
Køretid

De fleste algoritmer omformer input til output

Køretiden for en algoritme vokser typisk med størrelsen af input

Gennemsnitlig køretid er ofte vanskelig at bestemme

Vi fokuserer på værste tilfælde, da det
(1) er lettere at analysere
(2) ofte er vigtigt (ja måske kritisk)



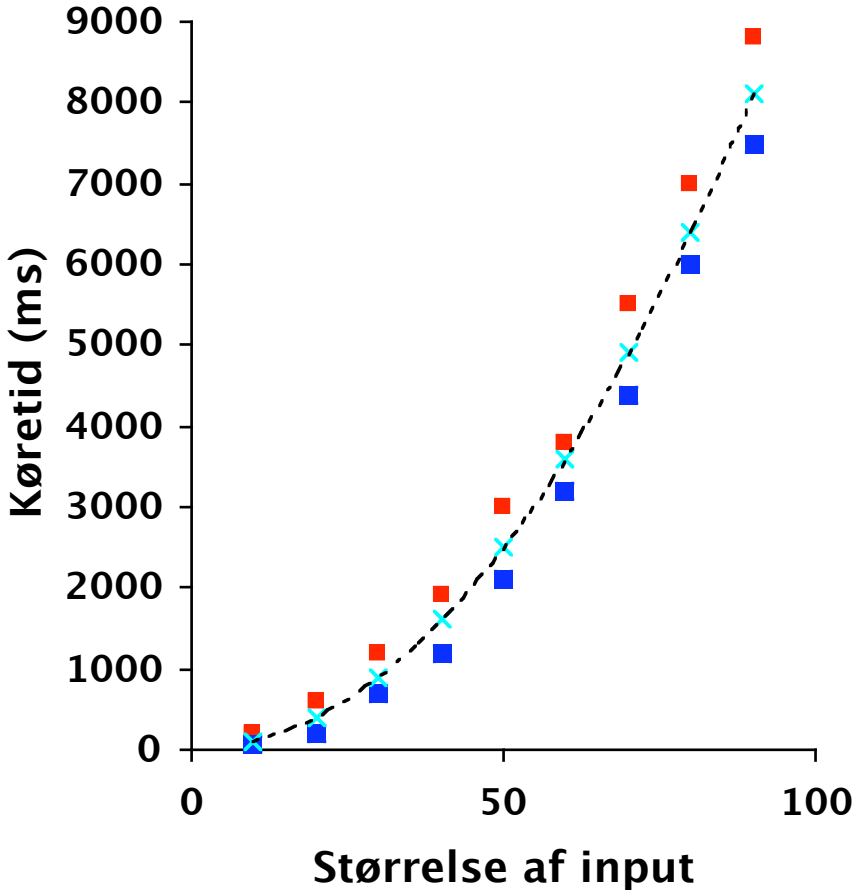
Eksperimentel analyse

(1) Skriv et program, der implementerer algoritmen

(2) Kør programmet med input af varierende størrelse

(3) Brug en metode som `System.currentTimeMillis()` i Java til at måle køretider

(4) Plot resultaterne



Ulemper ved eksperimentel analyse

- Det er nødvendigt at implementere algoritmen, hvilket kan være svært
- Resultaterne behøver ikke at være retningsgivende for input, der ikke er medtaget i eksperimenterne
- Hvis to algoritmer skal sammenlignes, skal det ske med det samme materiel og programmel

Teoretisk analyse

Benytter en højniveau-beskrivelse af algoritmen i stedet for en implementering

Karakteriserer køretiden som en funktion af inputstørrelsen

Tager hensyn til **alle mulige** input

Muliggør vurdering af en algoritmes hastighed uafhængigt af materiel og programmel

Pseudokode

Eksempel: find det største element i et array

Højniveau-beskrivelse af en algoritme

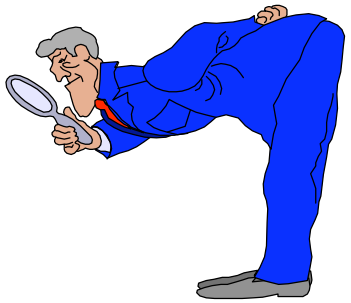
Mere struktureret end prosa

Mindre detaljeret end programkode

```
Algorithm arrayMax(A, n)
  Input array A of n integers
  Output maximum element of A

  currentMax ← A[0]
  for i ← 1 to n − 1 do
    if A[i] > currentMax then
      currentMax ← A[i]
  return currentMax
```

Pseudokode-detaljer



Kontrolstrukturer

if … **then** … [**else** …]

while … **do** …

repeat … **until** …

for … **do** …

Indrykning erstatter parenteser

Metodeerklæring

Algorithm ***method***(***arg*** [, ***arg***…])

Input …

Output …

Metodekald

method(***arg*** [, ***arg***…])

Returnering af værdi

return ***expression***

Udtryk

Tildeling

← (svaret til = i Java)

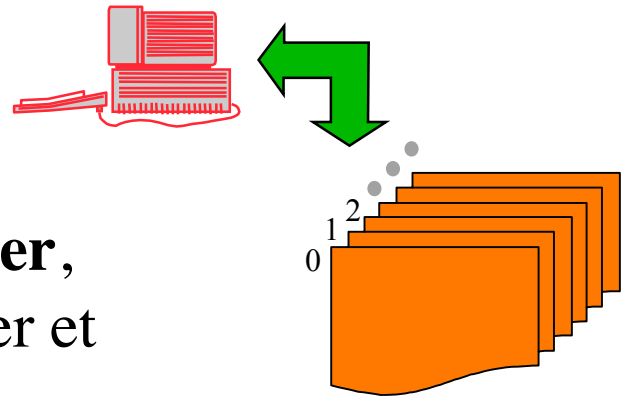
Test for lighed

= (svarer til == i Java)

n^2 Matematisk notation er tilladt

Random Access Machine (RAM) model

- En **CPU**
- En ubegrænset mængde **lagerceller**, som hver kan indeholde et tal eller et tegn



Lagercellerne er nummererede, og en tilgang til en hvilken som helst celle tager en tidsenhed

Primitive operationer

De basale operationer foretaget af en algoritme

Kan identificeres i pseudokoden

Stort set uafhængige af programmeringssprog

Antages at tage konstant tid i RAM-modellen

Eksempler:

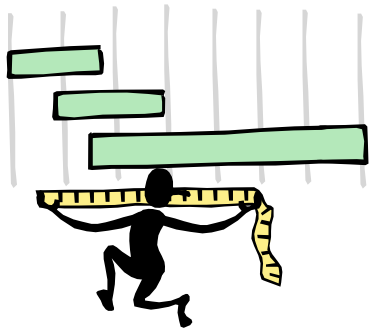
- Evaluering af et udtryk
- Tildeling af en værdi til en variabel
- Indeksering i et array
- Kald af en metode
- Returnering fra en metode

Optælling af primitive operationer

Ud fra pseudokoden kan vi bestemme det maksimale antal primitive operationer, der udføres af en algoritme, som en funktion af inputstørrelsen

Algorithm ***arrayMax***(A, n)	# operationer (max))
currentMax $\leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$1 + n$
if $A[i] >$ ***currentMax*** **then**	$2(n - 1)$
currentMax $\leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return ***currentMax***	1
	I alt $7n - 2$

Estimering af køretid



Algoritmen ***arrayMax*** udfører i værste tilfælde $7n - 2$ primitive operationer

Definer

a = Den tid, det tager at udføre den hurtigste primitive operation

b = Den tid, det tager at udføre den langsomste primitive operation

og lad $T(n)$ være den værste tid for ***arrayMax***

Da gælder

$a(7n - 2) \leq T(n) \leq b(7n - 2)$

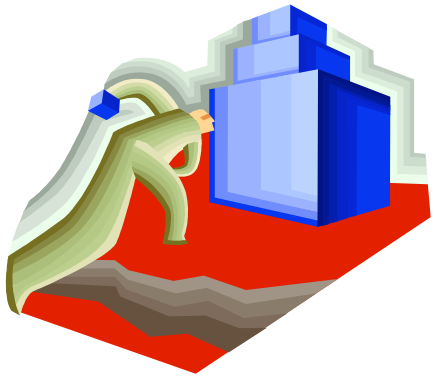
Køretiden er altså opadtil og nedadtil begrænset af to lineære funktioner

Vækstraten for køretiden

Et skift til en anden maskine

påvirker $T(n)$ med en konstant faktor,
men ændrer ikke på vækstraten for $T(n)$

Lineær vækst af køretiden $T(n)$ er en fundamental egenskab ved algoritmen ***arrayMax***

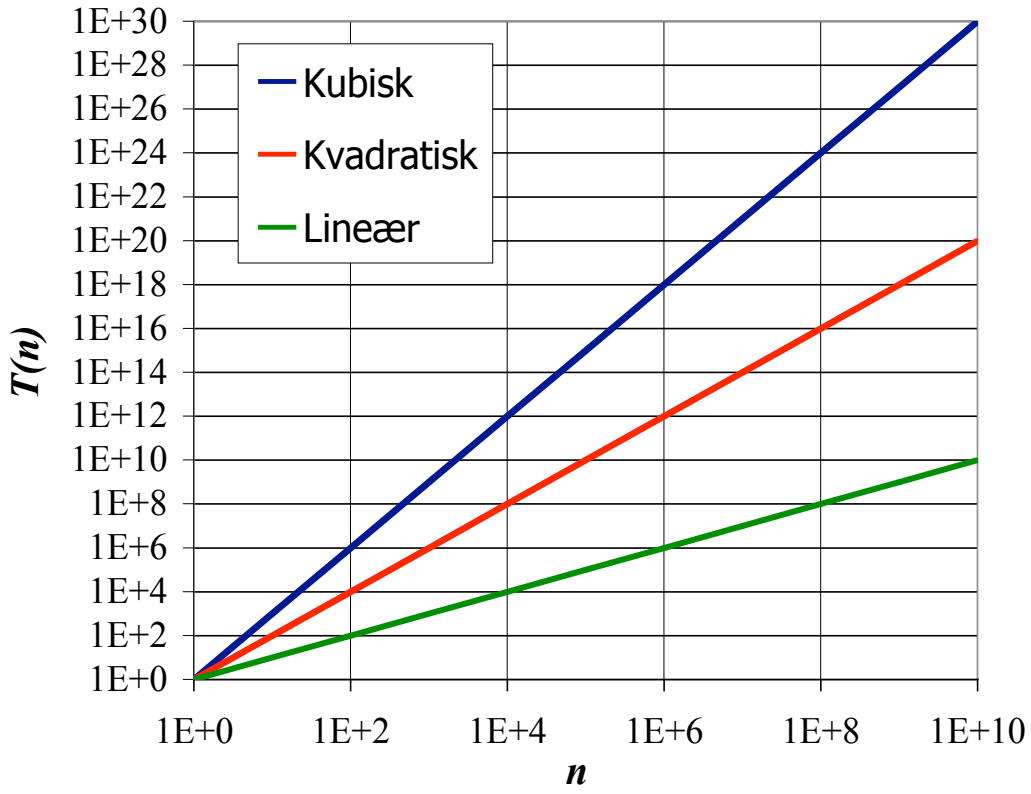


Vækstrater

Eksempler på vækstrater:

- Lineær ≈ n
- Kvadratisk ≈ n^2
- Kubisk ≈ n^3

I et log-log diagram svarer linjens hældningskoefficient til funktionens vækstrate



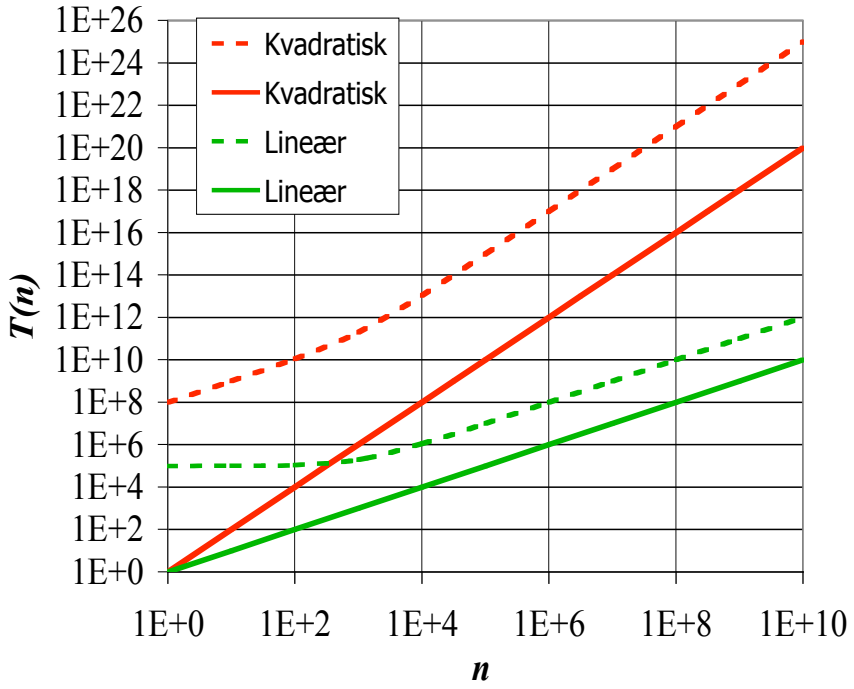
Konstante faktorer

Vækstraten påvirkes ikke af konstante faktorer og lavere-ordens termer

Eksempler:

$10^2n + 10^5$ er en lineær funktion

$10^5n^2 + 10^8n$ er en kvadratisk funktion



O-notation (store-*O*)

Givet funktionerne $f(n)$ og $g(n)$

Vi siger, at $f(n)$ er $O(g(n))$, hvis der findes positive konstanter c og n_0, således at

$$f(n) \leq cg(n) \quad \text{for } n \geq n_0$$

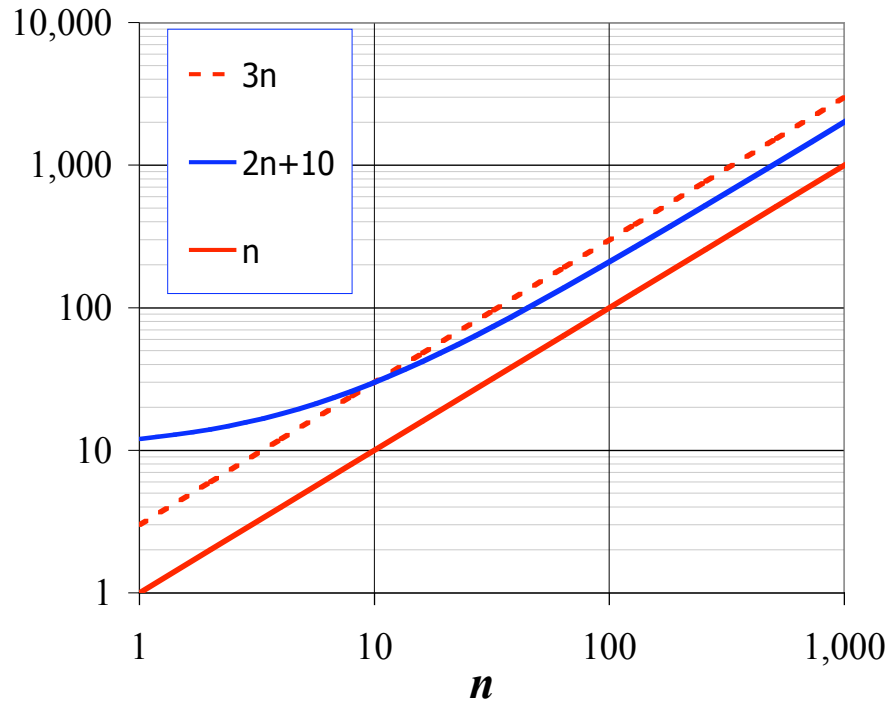
Eksempel: $2n + 10$ er $O(n)$

$$2n + 10 \leq cn$$

$$(c - 2)\, n \geq 10$$

$$n \geq 10/(c - 2)$$

Vælg $c = 3$ og $n_0 = 10$



Eksempel

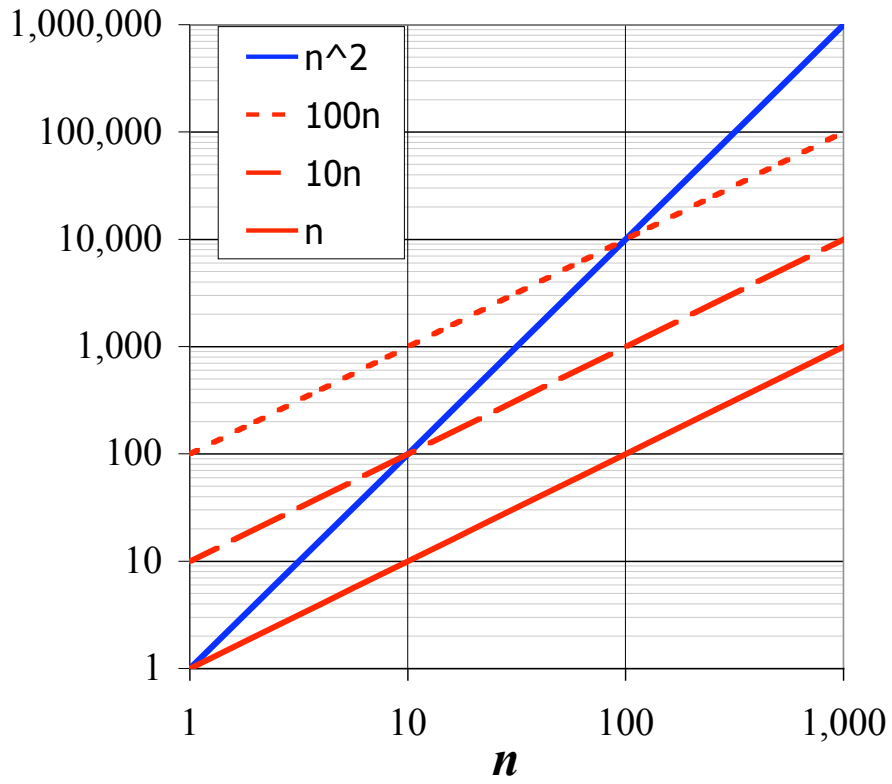
Eksempel:

Funktion n^2 er ikke $O(n)$

$n^2 \leq cn$

$n \leq c$

Denne ulighed kan ikke tilfredsstilles, da c skal være en konstant



Flere eksempler

$7n$-2 er $O(n)$

Bestem $c > 0$ og $n_0 \geq 1$, så $7n\text{-}2 \leq cn$ for $n \geq n_0$

Det gælder for $c = 7$ og $n_0 = 1$

$3n^3 + 20n^2 + 5$ er $O(n^3)$

Bestem $c > 0$ og $n_0 \geq 1$, så $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$

Det gælder for $c = 4$ og $n_0 = 21$

$3 \log n + \log \log n$ er $O(\log n)$

Bestem $c > 0$ og $n_0 \geq 1$, så $3 \log n + \log \log n \leq c \log n$ for $n \geq n_0$

Det gælder for $c = 4$ og $n_0 = 2$

O og vækstrater

O-notationen giver en øvre grænse for vækstraten for en funktion

Udsagnet "$f(n)$ er $O(g(n))$" betyder, at vækstraten for $f(n)$ ikke er større end vækstraten for $g(n)$

	$f(n)$ er $O(g(n))$	$g(n)$ er $O(f(n))$
$g(n)$ vokser hurtigst	Ja	Nej
$f(n)$ vokser hurtigst	Nej	Ja
Samme vækstrate	Ja	Ja

Regler

Hvis $f(n)$ er et polynomium af orden d, så gælder, at $f(n)$ er $O(n^d)$.

Fjern lavere-ordens led

Fjern konstante faktorer

Benyt så lille orden som muligt:

Sig "$2n$ er $O(n)$" i stedet for "$2n$ er $O(n^2)$"

Benyt så simpelt et udtryk som muligt:

Sig "$3n + 5$ er $O(n)$" i stedet for "$3n + 5$ er $O(3n)$"

Sig "$75 + 25$ er $O(1)$" i stedet for "$75+ 25$ er $O(100)$"

Asymptotisk algoritmeanalyse

Asymptotisk analyse af en algoritme udtrykker køretiden i ***O***-notation

(1) Find det værste tilfælde og bestem antallet af primitive operationer som en funktion af inputstørrelsen
(2) Udtryk denne funktion ved ***O***-notation

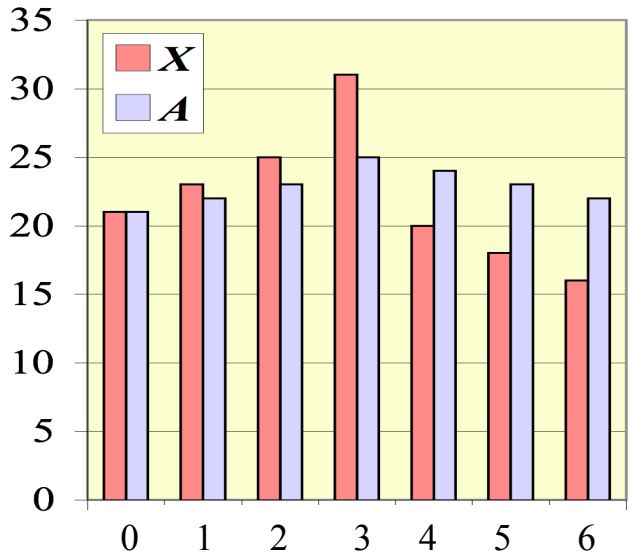
Eksempel: Vi afgør, at algoritmen ***arrayMax*** højst udfører $7n - 2$ primitive operationer. Vi siger, at algoritmen ***arrayMax*** "udføres i $O(n)$ tid"

Eftersom konstante faktorer og lavere-ordens led udelades, kan vi se bort fra dem, når vi tæller antallet af primitive operationer

Eksempel på asymptotisk analyse
Beregning af løbende gennemsnit

Det ***i***'te løbende gennemsnit, $A[i]$, af et array X er gennemsnittet af de første $(i+1)$ elementer i X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$



Løbende gennemsnit (i kvadratisk tid)

Algorithm *prefixAverages1*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	#operationer
$A \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow X[0]$	n
for $j \leftarrow 1$ **to** i **do**	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$A[i] \leftarrow s / (i + 1)$	n
return A	1

Summen af de første n heltal er $n(n+1)/2$

Køretiden er derfor $O(n^2)$

Løbende gennemsnit (i lineær tid)

Nedenstående algoritme bestemmer løbende gennemsnit i lineær tid ved at vedligeholde en løbende sum.

Algorithm *prefixAverages2*(*X*, *n*)	
Input array *X* of *n* integers	
Output array *A* of prefix averages of *X*	#operationer
$A \leftarrow$ new array of n integers	n
$sum \leftarrow 0$	1
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$sum \leftarrow sum + X[i]$	n
$A[i] \leftarrow sum / (i + 1)$	n
return *A*	1

Slægtninge til store-*O*

Store-Omega

f(n) er Ω(g(n)), hvis der findes en konstant $c > 0$
og en heltalskonstant $n_0 \geq 1$, således at f(n) $\geq$ cg(n) for $n \geq n_0$

Store-Theta

f(n) er Θ(g(n)), hvis der findes konstanter $c' > 0$ og $c'' > 0$, samt en heltalskonstant $n_0 \geq 1$, således at c'g(n) $\leq$ f(n) $\leq c''$g(n) for $n \geq n_0$

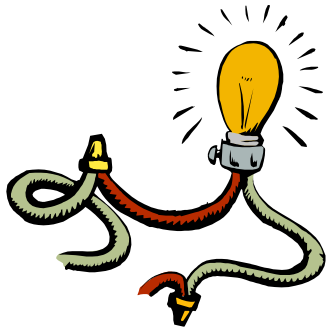
Lille-o

f(n) er o(g(n)), hvis der for enhver konstant $c > 0$ findes en heltalskonstant $n_0 \geq 0$, således at f(n) $\leq c$g(n) for $n \geq n_0$

Lille-omega

f(n) er ω(g(n)), hvis der for enhver konstant $c > 0$ findes en heltalskonstant $n_0 \geq 0$, således at f(n) $\geq$ cg(n) for $n \geq n_0$

Intuitiv beskrivelse af asymptotisk notation



Store-O

f(***n***) er ***O***(g(***n***)), hvis f(***n***) er asymptotisk **mindre end eller lig med** g(***n***)

Store-Omega

f(***n***) er ***Ω***(g(***n***)), hvis f(***n***) er asymptotisk **større end eller lig med** g(***n***)

Store-Theta

f(***n***) er ***Θ***(g(***n***)), hvis f(***n***) er asymptotisk **lig med** g(***n***)

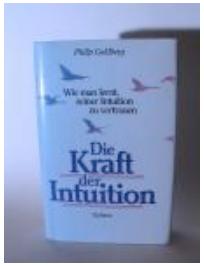
Lille-o

f(***n***) er ***o***(g(***n***)), hvis f(n) er asymptotisk **skarpt mindre end** g(***n***)

Lille-omega

f(***n***) er ***ω***(g(***n***)), hvis f(***n***) er asymptotisk **skarpt større end** g(***n***)

Intuitiv beskrivelse af asymptotisk notation



For en funktion f er $o(f)$, $O(f)$, $\Theta(f)$, $\Omega(f)$ og $\omega(f)$ mængder af funktioner karakteriseret på følgende måde:

$o(f)$: vokser langsommere end f
$O(f)$: vokser højst så hurtigt som f
$\Theta(f)$: vokser som f
$\Omega(f)$: vokser mindst så hurtigt som f
$\omega(f)$: vokser hurtigere end f

Citat

“People who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.”

Donald E. Knuth