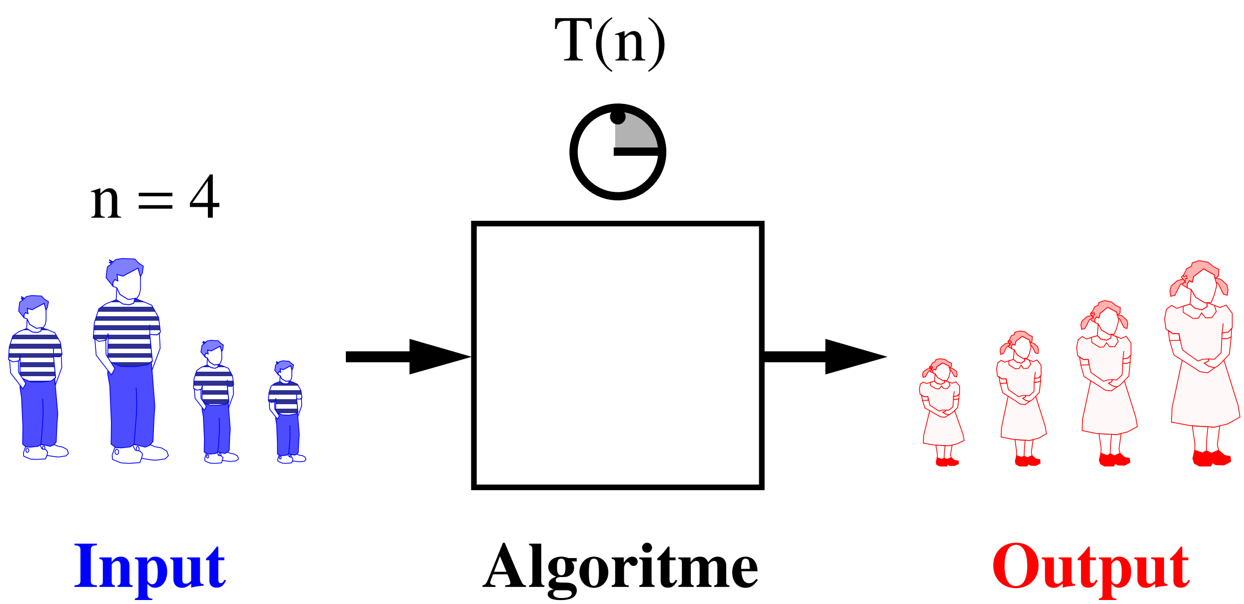
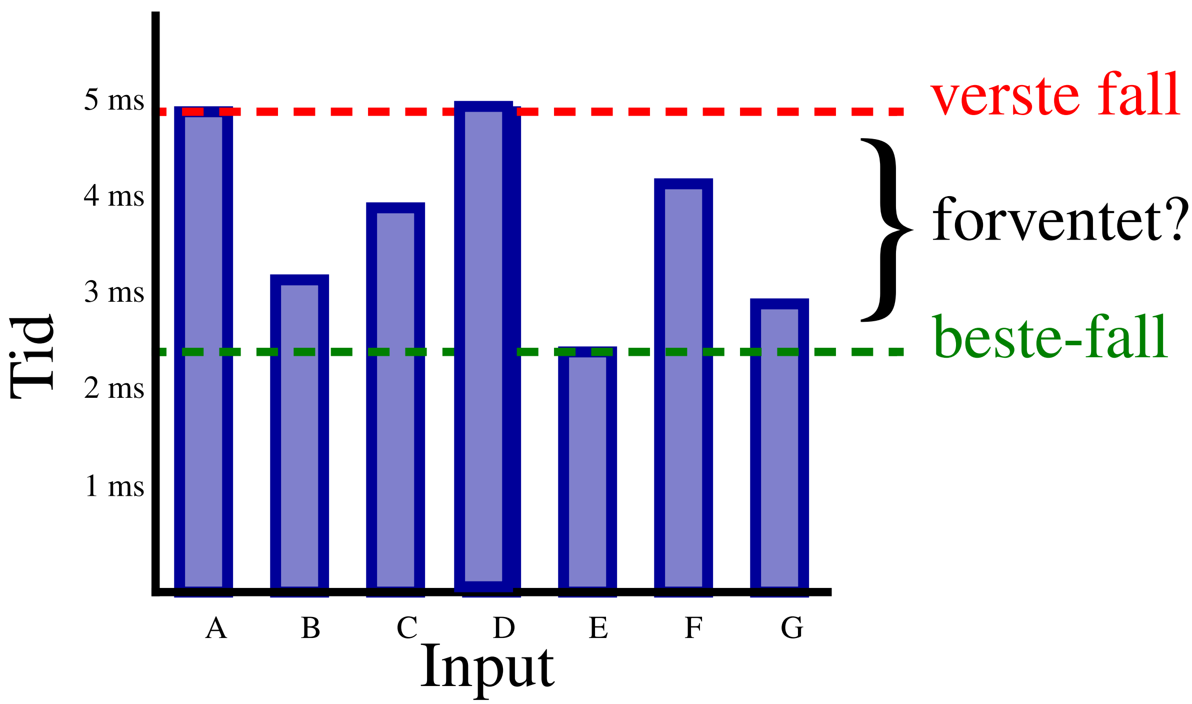
ALGORITMEANALYSE

- Rask matematisk gjennomgang
- Kjøretid
- Pseudokode
- Algoritmeanalyse
- Asymptotisk notasjon
- Asymptotisk analyse



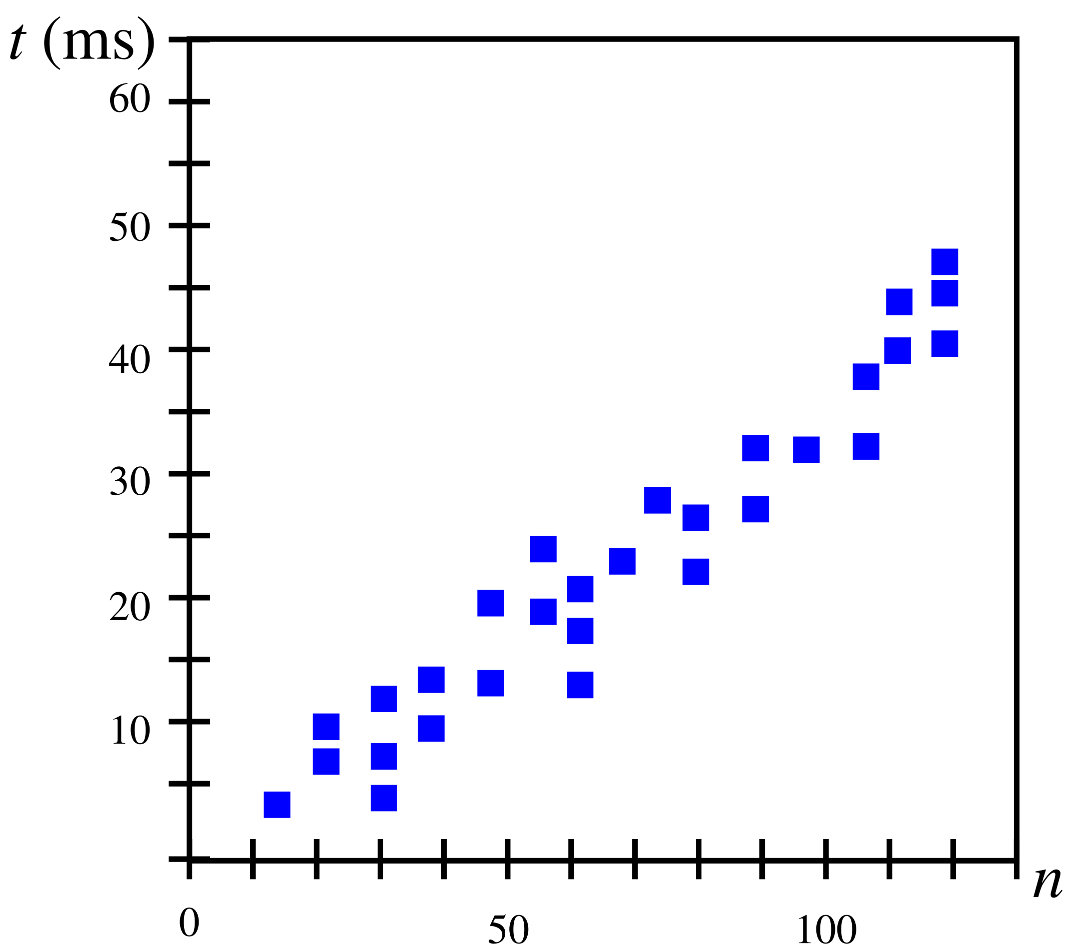
Forventet vs Verste Fall Kjøretid for en Algoritme

- En algoritme kan være rask på noen input og sein på andre.
- Å finne den forventede kjøretid kan dog være vanskelig, derfor måles helst algoritmer etter deres verste fall tidskompleksitet.
- For mange applikasjoner (overvåking av flytrafikk, medisinsk operasjonsutstyr) så er verste fall tidskompleksitet avgjørende!



Måling av Kjøretid

- Hvordan måle kjøretiden til en algoritme?
- Praktisk eksperimentering
  - Skriv et program som implementerer algoritmen
  - Kjør programmet for varierende input av lengde n, og for forskjellige verdier av n.
  - Bruk en metode som System.currentTimeMillis() til å måle nøyaktig kjøretid.
  - Datsettet vil f.eks. se slikt ut:



Bedre enn eksperimentering

- Eksperimentelle studier har begrensninger:
  - Man må implementere og teste algoritmene for å bestemme kjøretid.
  - Eksperimenter kan gjøres kun på et begrenset antall input, og forventet/verste fall er kanskje ikke med.
  - For å sammenlikne to algoritmer, må samme hardware og software omgivelser brukes.
- Vi bruker istedet en generell metode for kjøretidsanalyse av algoritmer. Denne
  - Bruker en høynivå beskrivelse av algoritmen, istedet for en bestemt implementasjon.
  - Tar høyde for alle mulige input.
  - Gir evaluering av effektiviteten til en algoritme uavhengig av hardware og software omgivelser.

Pseudokode

- Pseudokode er en beskrivelse av en algoritme som er mer strukturert enn vanlig prosa, men mindre formell enn et programmeringsspråk.
- Eksempel: Finn største element i en array.

Algorithm arrayMax(A, *n*):
  Input: En array A som lagrer *n* heltall.
  Output: Største elementet i A.
 currentMax ← A[0]
 for *i* ← 1 **to** *n* −1 **do**
  if *currentMax* < A[*i*] **then**
   currentMax ← A[*i*]
 return *currentMax*

- Pseudokode foretrekkes for algoritmebeskrivelser.
- Pseudokode gjemmer bort mange design valg.

Hva er pseudokode?

- En blanding av naturlig språk og høynivå programmeringsbegrep som beskriver det viktigste i en generisk implementasjon av en datastruktur eller algoritme.
  - Uttrykk: bruk standard matematiske symboler for numeriske og boolske uttrykk:
    - bruk $\leftarrow$ for tildeling ("=" i Java)
    - bruk **=** for å teste likhet ("==" i Java)
  - Metodedeklarasjoner:
    - **Algorithm** name(*param1*, *param2*)
  - Programmeringskonstruksjoner:
    - decision structures: **if** ... **then** ... **[else** ... **]**
    - while-loops: **while** ... **do**
    - repeat-loops: **repeat** ... **until** ...
    - for-loop: **for** ... **do**
    - array indeksering: **A[*i*]**
  - Metode:
    - kall: object method(args)
    - returverdi: **return** value

AlgoritmeAnalyse

- Primitive Operasjoner: Lavnivå beregninger som er uavhengige av programmeringsspråk, f.eks.:
  - kalle en metode og returnere fra en metode
  - utføre en aritmetisk operasjon(f.eks. addisjon)
  - sammenlikne to tall, etc.

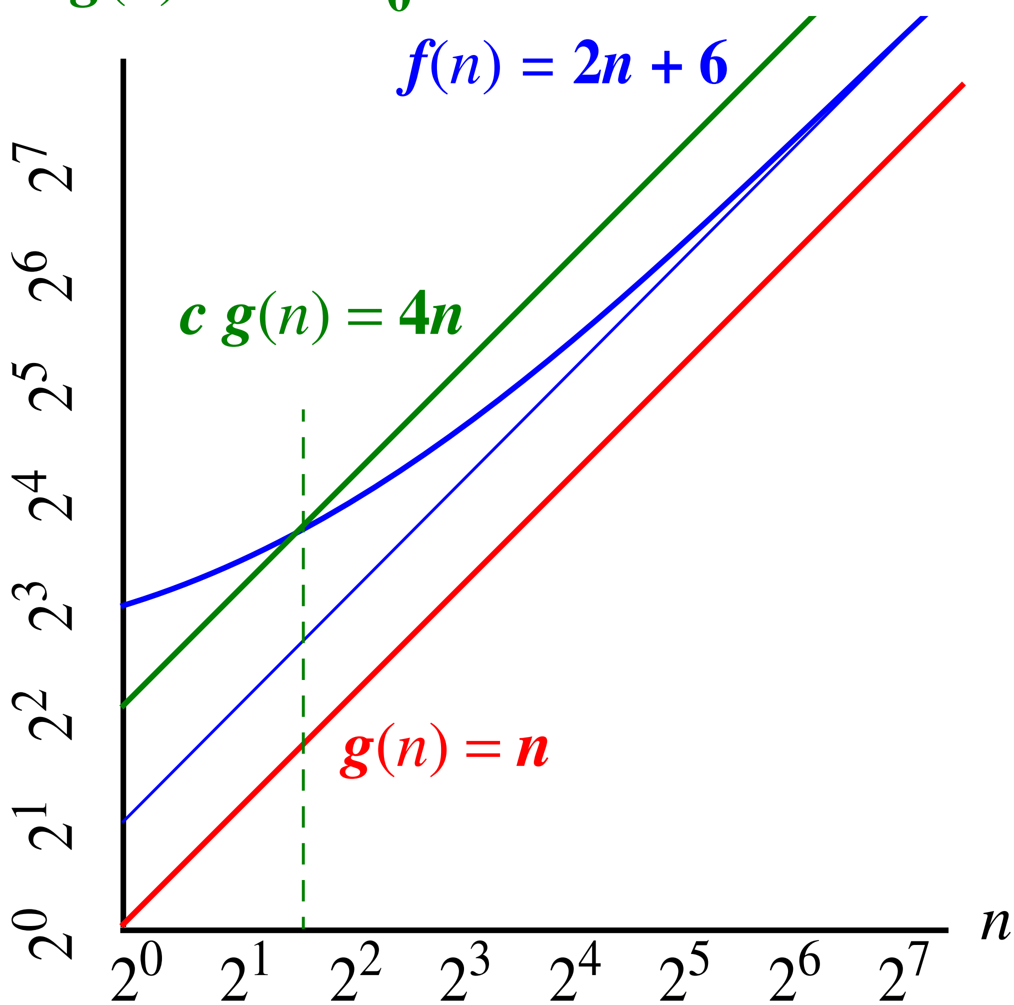
- Ved å se på pseudokoden kan vi telle antall primitive operasjoner en algoritme vil utføre.

- Eksempel:

Algorithm arrayMax(A, *n*):
    Input: En tabell A med *n* heltall.
    Output: Det største elementet i A.
currentMax ← A[0]
for *i* ← 1 **to** *n* −1 **do**
    if *currentMax* < A[*i*] **then**
        currentMax ← A[*i*]
return *currentMax*

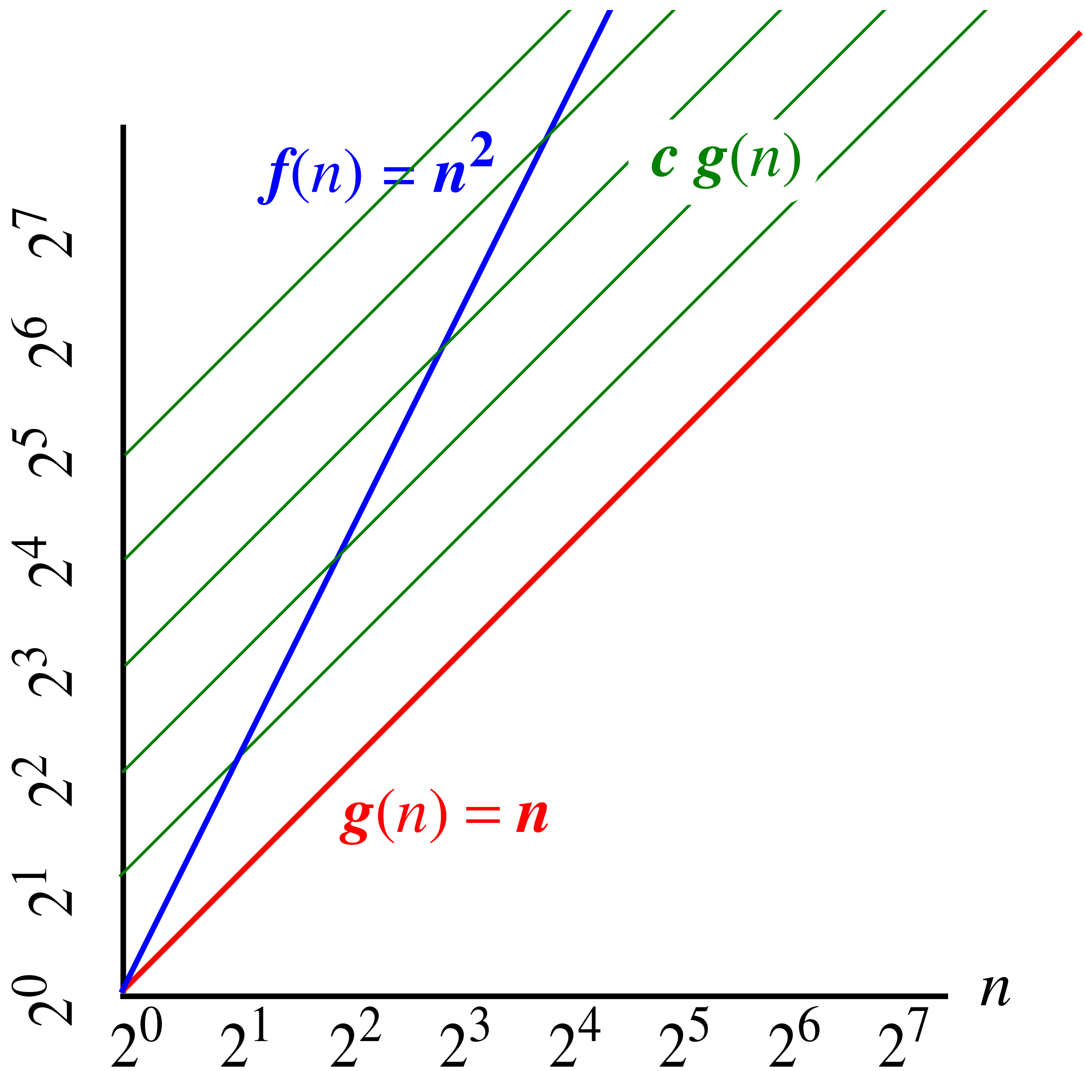
Asymptotisk notasjon

- Målet: Forenkle kjøretidsanalysen ved å fjerne irrelevant informasjon
  - som “avrunding” 1,000,001 ≈ 1,000,000
  - på en formell måte vil vi si f.eks. $3n^2 \approx n^2$
- Stor-O notasjon/“Big-Oh” Notation
  gitt to funksjoner $f(n)$ og $g(n)$, sier vi at
  $f(n)$ er $O(g(n))$ **hvis og bare hvis**
  det eksisterer positive konstanter c og n_0 s.a.
  $f(n) \leq c\, g(n)$ for n n_0



Nytt eksempel

- n^2 **er ikke** O(n)
- vi kan ikke finne c og n_0 s.a.
  $n^2 \leq c\, n$ for n n_0



Asymptotisk Notasjon (forts.)

- NB: Selv om det er korrekt å si "7n - 3 er $\boldsymbol{O}(n^3)$", er det bedre å si "7n - 3 er $\boldsymbol{O}(n)$", dvs 'avrundingen' skal være så presis som mulig.

- Enkel regel: Dropp lavereordens termer og konstanter:
  - $7n - 3$ er $\boldsymbol{O}(n)$
  - $8n^2 \log n + 5n^2 + n$ er $\boldsymbol{O}(n^2 \log n)$

- Typiske kjøretider:
  - logaritmisk: $\boldsymbol{O}(\log n)$
  - lineært $\boldsymbol{O}(n)$
  - kvadratisk $\boldsymbol{O}(n^2)$
  - polynomisk $\boldsymbol{O}(n^k)$, k 1
  - eksponensiellt $\boldsymbol{O}(a^n)$, $n > 1$

- Andre notasjoner tilsvarende Stor-O (brukes i I234):
  – $\Omega(f(n))$: Stor-Omega
  – $\Theta(f(n))$: Stor-Theta

Asymptotisk analyse av kjøretid

- Bruk stor-O notasjon for å angi antall primitive operasjoner en algoritme utfører, som en funksjon av inputstørrelse.
- F.eks. sier vi at arrayMax algoritmen har kjøretid ***O***(n), dvs den er lineær. Underforstått at dette er i verste fall, dvs ‘det finnes en konstant k s.a. for enhver input av størrelse n (i dette tilfelle n heltall) så utfører arrayMax algoritmen ikke mer enn kn primitive operasjoner.’
- Sammenlikning av asymptotisk kjøretid:
  - En ***O***(n) algoritme er bedre enn en ***O***(n^2) algoritme
  - ***O***(log n) er bedre enn ***O***(n)
  - hierarki av funksjoner:
  - log n << n << n log n << n^2 << n^3 << 2^n
- Advarsel!
  - Asympotisk notasjon gjemmer bort store konstante faktorer. En algoritme med kjøretid 100,000 *n*, dvs ***O***(*n*), er for mange datasett mindre effektiv enn en algoritme med kjøretid $2n^2$, dvs ***O***(n^2). Hvor stor må n være før den siste algoritmen blir mest effektiv?

Eksempel på asymptotisk analyse

- En algoritme som beregner prefiks snitt:

Algorithm prefixAverages1(*X*):
  Input: En *n*-element array *X* av tall.
  Output: En *n*-element array A av tall s.a.
    A[*i*] er snittet av elementene *X*[0], ... , *X*[*i*].
Let A be an array of *n* numbers.
for $i \leftarrow 0$ **to** $n - 1$ **do**
  $a \leftarrow 0$
  for $j \leftarrow 0$ **to** i **do**
    $a \leftarrow a + X[j]$
  $A[i] \leftarrow a/(i + 1)$
return array A

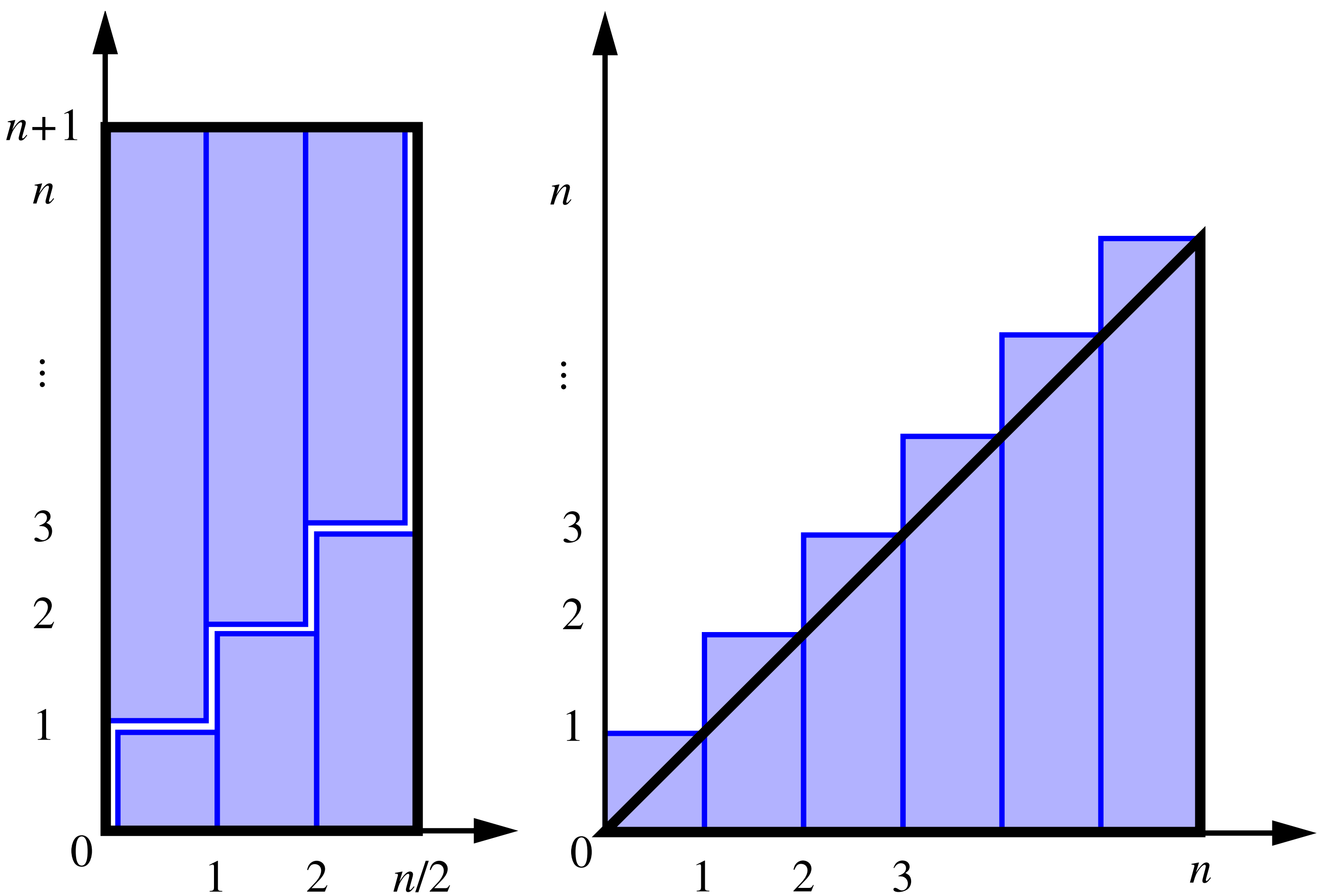
- Analyse ...

• Rask Matte-bakgrunn

- Aritmetiske progresjoner:
  - Eksempel:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n^2 + n}{2}$$

  - to visuelle representasjoner:



Nytt Eksempel

- En bedre algoritme for å beregne prefix snitt:

Algorithm prefixAverages2(X):
Input: En n-element array X av tall.
Output: En n-element array A av tall s.a.
A[i] er snittet av elementene X[0], ... , X[i].
Let A be an array of n numbers.
$s \leftarrow 0$
for $i \leftarrow 0$ **to** n - 1 **do**
$s \leftarrow s + X[i]$
A[i] $\leftarrow s/(i + 1)$
return array A

- Analyse ...

Mere matte-bakgrunn

- Gulv (Floor)

  $\lfloor x \rfloor$ = det største heltall $\leq$ x

- Tak (Ceiling)

  $\lceil x \rceil$ = det minste heltall   x

- Summasjoner
  - generell definisjon:

$$\sum_{i=s}^{t} f(i) = f(s) + f(s+1) + f(s+2) + \ldots + f(t)$$

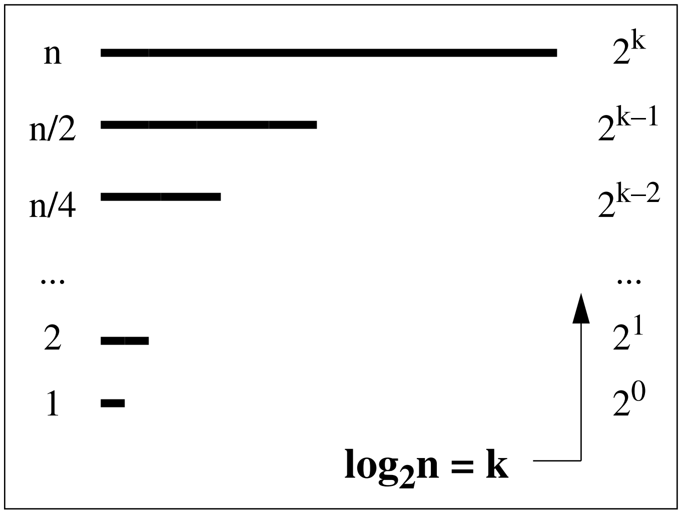
  - hvor f er en funksjon, s er startindeks, og t er sluttindeks.
  - Geometrisk progresjon: $f(i) = a^i$
  - gitt heltall n 0 og et reellt tall $0 < a \neq 1$

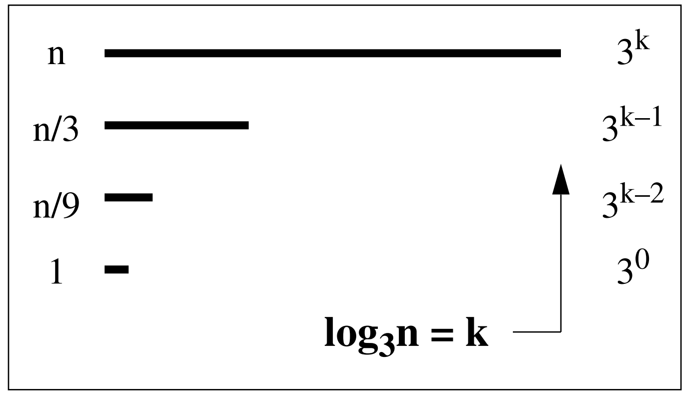
$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \ldots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

  - geometriske progresjoner utviser eksponensiell vekst

Noen enkle fakta

$\mathbf{log_b n = k}$ hviss $\mathbf{b^k = n}$





$$\mathbf{log_b n = \frac{log_c n}{log_c b}}$$

$$\log_3 n = \frac{\log_2 n}{\log_2 3} \quad =o$$

$$\mathbf{\log(n * m) \quad = \quad \log(n) + \log(m)}$$

$$\log(n^c) \quad = \quad c * \log(n) \qquad =o$$

$$\log(n/m) \quad = \quad \log(n) - \log(m)$$

$$\mathbf{1 + 2 + 3 + ... + n} = \sum_{k=1}^{n} k \quad = \quad \frac{n^2 + n}{2}$$

$$\mathbf{a^0 + a^1 + ... + a^n} = \sum_{k=0}^{n} a^k \quad = \quad \frac{1 - a^{n+1}}{1 - a} \qquad 0 < a \neq 1$$

Anta vi har 3 algoritmer A, B og C med kjøretid hhv O(n), O(nlogn) og O(n^2), men at konstanten som skjules er forskjellig for hver algoritme, dvs faktisk er disse hhv 400*n, 20*nlogn og 2*n^2 .
Stor-O forteller oss at asymptotisk, dvs når n vokser, så er A er bedre enn B bedre enn C. I praksis vil dette avhenge av inputstørrelsen n. Beregn antall primitive operasjoner for n=10, n=100 og n=1000.
I det lange løp vil alltid stor-O notasjon skille korrekt. I særdeleshet dersom vi vil se på virkningen av å skaffe en raskere maskin:

.

O( )	faktisk	problemstørrelse – løses på 1 sekund		
		dagens maskin	60-gngr raskere	3600-gngr raskere
n	400 * n	2 500	150 000	9 000 000
n * log(n)	20*n*log(n)	4 000	170 000	8 000 000
n^2	2 * n^2	700	5 500	42 000
n^4	n^4	30	90	250
2^n	2^n	20	25	30

Med den lineære algoritmen A ser vi at med en maskin som er 3600-ganger raskere så løser vi et problem 3600 ganger større.
Med den kvadratiske algoritmen C ser vi at med en maskin 3600 ganger raskere løser vi et problem kun 60 ganger større.

Er algoritmen eksponensiell i n, O(2^n), så er økningen kun additiv! Dvs 10 enheter pluss for 3600-ganger raskere maskin.

• Noen enkle bevisteknikker

- Ved eksempel
  - Finn et eksempel
  - Finn et moteksempel
- Ved motsigelse
  - Anta utsagnet ikke korrekt og vis at dette medfører en motsigelse. Ergo må utsagnet være korrekt.
  - Utsagn: 'Om ab oddetall, så er a og b oddetall.' Bevis: Anta utsagnet ikke stemmer, dvs ab er oddetall men a=2c (samme om vi sier b=2c). Da er ab=2bc et partall. Dette motsier at ab er oddetall.
- Induksjon og Løkke-Invarianter
  - Induksjon
    - 1) Bevis base-tilfellet, f.eks. n=0.
    - 2) Bevis at 'om tilfelle n korrekt, så er tilfelle n+1 korrekt'
  - Løkke-invarianter
    - Bevis forbetingelsen S_0
    - Vis at S_{i-1} impliserer S_i efter iterasjon i