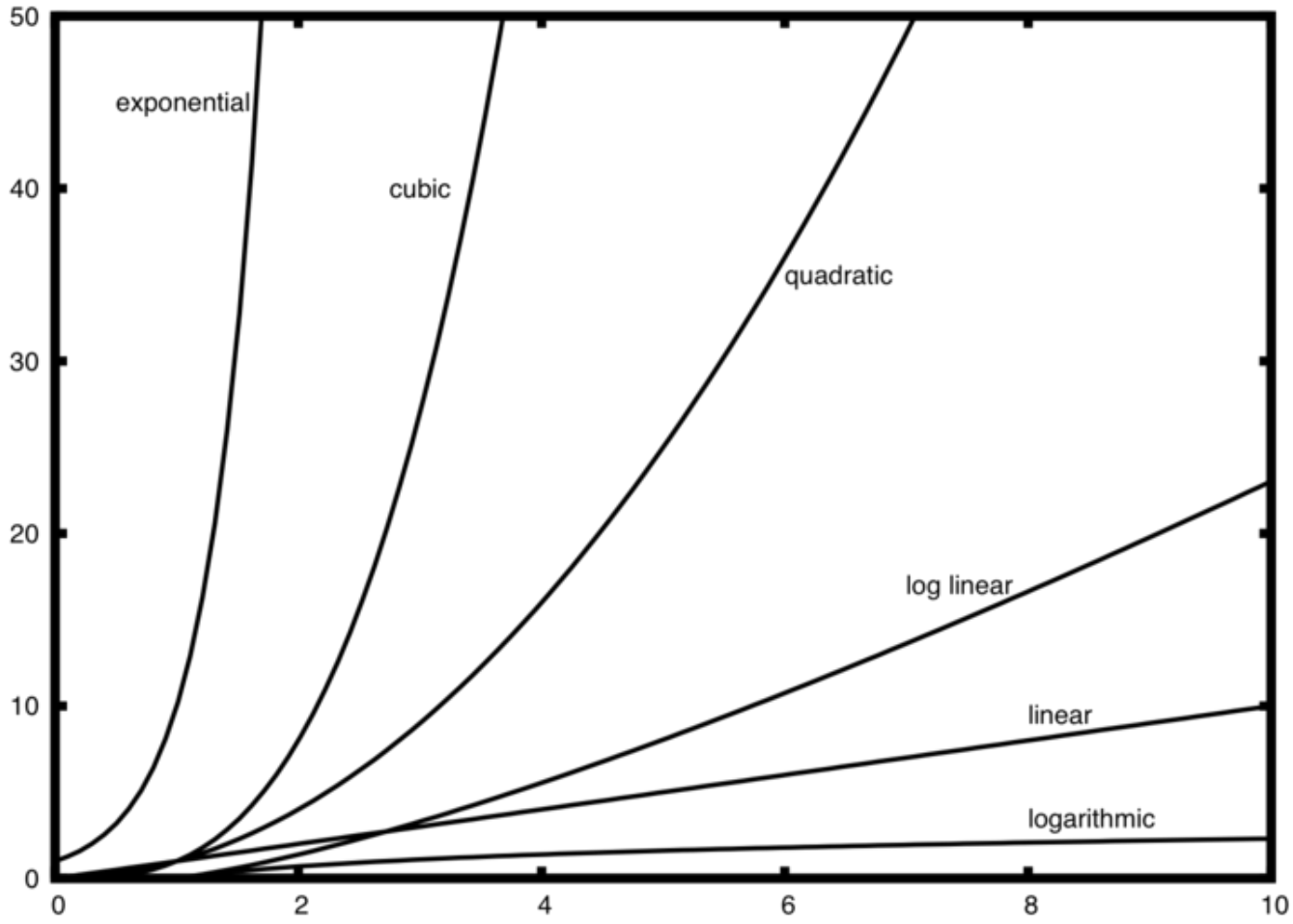
Algoritmeanalyse



Algoritmer - definisjon

- En algoritme* er en beskrivelse av hvordan man løser et **veldefinert** problem med en presist formulert **sekvens** av et endelig antall enkle, utvetydige og tidsbegrensede **steg**.
- De fleste dagligdagse gjøremål kan beskrives som en algoritme, f.eks. matoppskrifter.
- Alle 'vanlige' dataprogrammer kan sees på som beskrivelser/implementasjoner av en eller flere algoritmer.

*: Abū ʿAbdallāh Muḥammad ibn Mūsā al-Khwārizmī (Persia, 800-tallet)

Eksempel – Euklids algoritme

1. Gitt to positive heltall A og B
2. Hvis A er mindre enn B:
   2.1 Sett et heltall TMP lik A
   2.2 Sett A lik B
   2.3 Sett B lik TMP
3. Hvis B er lik 0:
   3.1 Gå til 7
4. Sett et heltall D lik resten man får ved å dele A på B (D = A mod B)
5. Sett A lik D
6. Gå til 2
7. Verdien av A er løsningen
8. Stopp

Krav til en algoritme

- **Nøyaktig spesifikasjon av input:**
  - Hva kreves for at algoritmen skal produsere korrekt resultat?
- **Veldefinerte steg:**
  - Hver enkelt operasjon må være klar og entydig.
- **Korrekthet:**
  - Algoritmen må løse problemet på riktig måte.
- **Terminering:**
  - Algoritmen må bli ferdig/avslutte hvis input er lovlig.
- **Beskrivelse av resultat/effekt:**
  - Det må være klart hva algoritmen er ment å gjøre.

Algoritmeanalyse

- En undersøkelse av hvor **effektiv** en algoritme er
- Tidsforbruk:
  - Hvor lang tid bruker en algoritme på å løse et problem i forhold til en annen algoritme?
  - Hvordan **øker** tidsbruken når problemet vokser seg stort?
- Forbruk av lagerplass:
  - Hvor mye minne (RAM og evt. disk etc.) bruker algoritmen i prosessen med å løse et problem av en viss størrelse?
  - Hvordan øker minnebruken med problemstørrelsen?

To ulike tilnærminger til algoritmeanalyse

- Gjennomsnittlig oppførsel:
  - Hvordan er effektiviteten i “det lange løp” når algoritmen/ programmet kjøres for et stort antall ulike datasett?
  - Er ofte meget komplisert å analysere
- “Worst case”:
  - Hvor effektiv er algoritmen for de “verste tilfellene” av input?
  - Gir en “garanti” for at vi ikke overstiger øvre grenser for tid- og plassforbruk
  - Er vanligst brukt og oftest enklest å analysere

Effektivitet og problemstørrelse

- Ønsker å finne ut hvordan tidsforbruket til en algoritme **vokser** når vi øker størrelsen på problemet
- Problemstørrelse er oftest et heltall (en parameter):
  - Betegnes med *n*
  - *n* kan f.eks. være antall elementer som skal sorteres eller antall elementer i en mengde vi skal søke i
- Vi er ikke interessert i **nøyaktige** kjøretider, men i **størrelsesorden** (eller **funksjonstype**) for tidsforbruket *t* som funksjon av *n* :

  $t = t(n)$

Eksempel: Nedlasting av filer

- Anta at det tar 2 sekunder å sette opp en internett-forbindelse, og at man deretter kan laste ned 2.5 MB/sekund.
- Hvis filen som skal lastes ned er på *n* megabytes, vil tiden det tar å laste den ned kunne uttrykkes som:

  $t(n) = n/2.5 + 2$

- For store filer vil det første leddet **dominere** uttrykket, vi kan i praksis se bort fra bidraget fra konstantleddet.
- Kjøretiden blir tilnærmet proporsjonal med filstørrelsen, nedlastingen av filer er derfor en **lineær** algoritme.

Eksempel: Minste element i en tabell

Algoritme:

- Input: Tabell med **n** heltall
- Output: Den minste elementet i tabellen, **min**
- Lokale variabler: Indeksvariabel, **i**

1. Sett **min** lik tabellens første element
2. Sett indeksvariabel **i** = 1
3. Så lenge som **i** er mindre enn **n**
   - 3.1 Hvis verdien til element nr. **i** er mindre enn **min**
     - 3.1.1 Set **min** lik element nr. **i**
   - 3.2 Øk verdien til **i** med 1
4. Returner verdien til **min**

Javakode

```
public static int finnMin(int[] tab, int n)
{
  int min = tab[0];                  // 1
  for (int i = 1; i < n ; i++) // 2, 3, 3.2
  {
    if (tab[i] < min )               // 3.1
    {
      min = tab[i];                  // 3.1.1
    }
  }
  return min;                        // 4
}
```

Opptelling av antall operasjoner

1. Tilordning og tabelloppslag: 2 operasjoner
2. 1 operasjon
3. I for-løkka utføres
   - Sammenlikningen ($i < n$): n - 1 operasjoner
   - Oppdatering av indeksvariabel: n - 1 operasjoner
   - Setning 3.1, ett tabelloppslag og en sammenlikning, utføres begge n - 1 ganger: $2(n - 1)$ operasjoner
   - Tilordningen i setning 3.1.1 utføres hver gang betingelsen er sann: x operasjoner
4. Retur av verdien til `min`: 1 operasjon

Totalt antall operasjoner: $4 + 4(n - 1) + x = 4n + x$

Asymptotisk analyse i stedet for detaljer

- Teller oftest ikke opp operasjoner så detaljert som dette
- For store verdier av *n* **dominerer** operasjonene som skjer inne i for-løkka
- Antall ganger de dominerende operasjonene utføres, definerer algoritmens **orden**
- Nøyaktig antall operasjoner ($4n + x$) er relativt uinteressant, det viktige er hvordan kjøretiden øker når antall elementer, *n*, øker
- Vi sier at algoritmen er lineær, den har orden (eller størrelsesorden) *n*, fordi kjøretiden er tilnærmet proporsjonal med *n*

Eksempel: Potensberegning

- x reelt tall, n heltall > 0
- x^n : Potensfunksjon med eksponent n ("x i n'te")
- $x^n = x \cdot x \cdot x \cdot x \cdot \ldots \cdot x$ (n ganger)
- x: grunntall, n: eksponent
- Eksempler:

$$2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16$$

$$10^6 = 10 \cdot 10 \cdot 10 \cdot 10 \cdot 10 \cdot 10 = 1\,000\,000$$

Potensberegning: “Rett-frem” algoritme

```
double potens1(double x, long n)
{
   double result = 1.0;
   for (long i = 1; i <= n; i++)
     result *= x;
   return result;
}
```

Operasjoner i enkel potensberegning

- Utenom for-løkken:
  - 2 parameteroverføringer, 2 tilordninger, 1 retur av verdi
  - Konstant tid, avhenger ikke av *n*
  - Anta at dette tar tiden C_1
- I hvert gjennomløp av for-løkken:
  - 1 sammenligning, 2 tilordninger, 1 addisjon, 1 multiplikasjon
  - Anta at et gjennomløp tar tiden C_2

Tidsforbruk, enkel potensberegning

- For-løkken utføres n ganger
- Den totale kjøretiden blir:

  $t = t(n) = C_2 \cdot n + C_1$

- For store verdier av n dominerer første ledd:

  $t(n) \approx C_2 \cdot n$

- Konstanten C_2 er avhengig av maskin/kompilator
- Vi sier at algoritmen er av orden n og skriver:

  $t = O(n)$

Mer effektiv potensberegning

- Bruker at:

  $x^{2n} = (x^2)^n$ – eksponenten er partall

  $x^{2n+1} = x \cdot (x^2)^n$ – eksponenten er oddetall

- "Rett-frem" beregning med 16 multiplikasjoner:

  $2^{16} = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 65536$

- Mer effektiv beregning med 4 multiplikasjoner:

  $2^{16} = (2^2)^8 = (4)^8 = (4^2)^4 = (16)^4 = (16^2)^2 =$

  $256^2 = 65536$

Eksempel: Effektiv potensberegning

- Beregning av x^{37} med bare 7 multiplikasjoner:

$$x^{37} = x \cdot (x^2)^{18}$$
$$x^{18} = (x^2)^9$$
$$x^9 = x \cdot (x^2)^4$$
$$x^4 = (x^2)^2$$
$$x^2 = x \cdot x$$

Potensberegning: Effektiv algoritme

```
double potens2(double x, long n)
{
    double result = x;
    while (n > 1)
    {
        result = result * result;
        if (n % 2 != 0)
            result = result * x;
        n = n / 2;
    }
    return result;
}
```

Operasjoner i effektiv potensberegning

- Utenom while-løkken:
  - 1 tilordning, 1 retur av verdi
  - Konstant, avhenger ikke av *n*
  - Anta at dette tar tiden C_1
- I hvert gjennomløp av while-løkken:
  - 2 sammenligninger, 1 tilordning, 1 divisjon, 1 eller 2 multiplikasjoner
  - Anta at et gjennomløp høyst tar tiden C_2

Antall iterasjoner

- Hvor mange ganger utføres while-løkken i en effektiv potensberegning?
- Antall iterasjoner blir lik antall ganger som *n* kan deles på 2 (halveres)
- Dette er det samme som tallet vi må opphøye 2 i for å få *n*
- Per definisjon er dette 2-er logaritmen til *n*
- Antall iterasjoner blir lik $\log_2 n$ (eller bare log *n*)

Tidsforbruk, effektiv potensberegning

- For-løkken utføres log n ganger
- Den totale kjøretiden blir:

  $t = t(n) = C_2 \cdot \log n + C_1$

- For store verdier av n dominerer første ledd:

  $t(n) \approx C_2 \cdot \log n$

- Konstanten C_2 er avhengig av maskin/kompilator
- Vi sier at algoritmen er av orden log n og skriver:

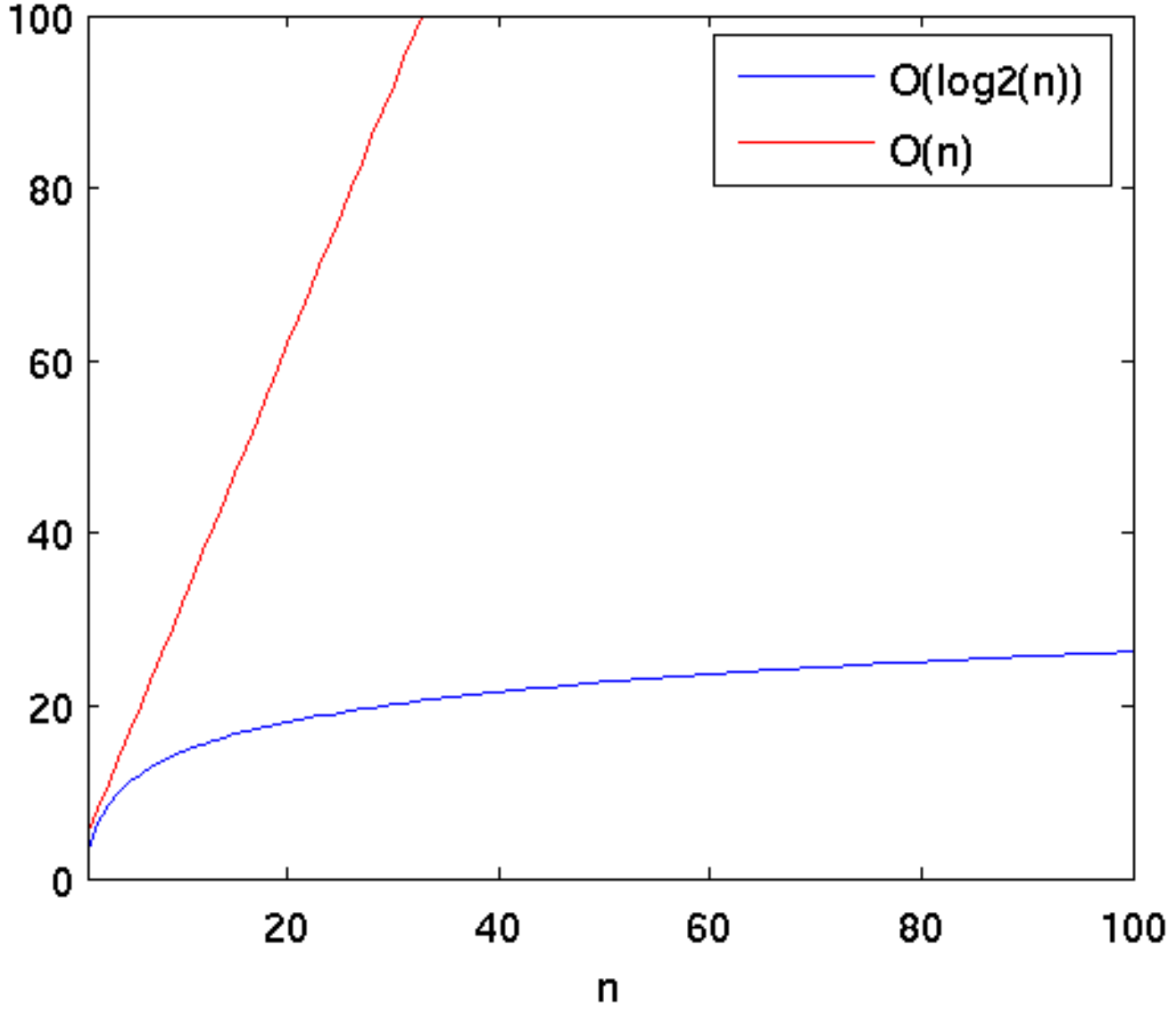
  $t = O(\log n)$

Smart potensberegning er *mye* raskere enn "rett frem"-beregning

- Tidsforbruket til den smarte algoritmen vokser som en logaritmefunksjon – veldig sakte!

- Eksempler:

$16 = 2^4$	$\log_2 16 = 4$
$256 = 2^7$	$\log_2 256 = 7$
$1000000 \approx 2^{20}$	$\log_2 1000000 \approx 20$


100
80
60
40
20
0
20
40
60
80
100
n
O(log2(n))
O(n)

Kjøretider: Potensberegning

Anta at et gjennomløp av løkken i begge de to metodene for beregning av x^n tar samme tid $C = 10^{-6}$ s

n	$\log_2 n$	$t_1(n) \approx C \cdot n$	$t_2(n) \approx C \cdot \log_2 n$	t_1 / t_2
2	1	$2 \cdot 10^{-6}$ s	10^{-6} s	2
16	4	$1.6 \cdot 10^{-5}$ s	$4 \cdot 10^{-6}$ s	4
256	8	$2.56 \cdot 10^{-4}$ s	$8 \cdot 10^{-6}$ s	32
4096	12	$4.1 \cdot 10^{-3}$ s	$1.2 \cdot 10^{-5}$ s	342
10^6	≈20	1.0 s	$2 \cdot 10^{-5}$ s	50000
10^9	≈30	16.5 min	$3 \cdot 10^{-5}$ s	$33 \cdot 10^6$

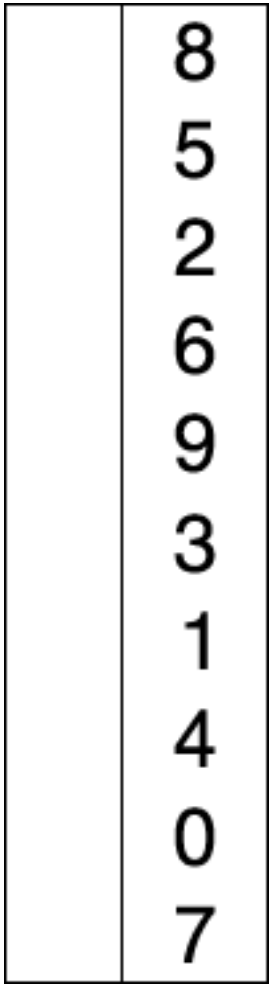
Sorteringsalgoritme: Utplukksortering

- Sorterer en array med lengde *n* i stigende rekkefølge
- Algoritmen bruker *n* - 1 gjennomløp
- I gjennomløp nummer *i* :
  - De *i* - 1 minste elementene ligger fremst i arrayen og *er* sortert riktig
  - Finner det minste av elementene blant de *n* – *i* + 1 som *ikke* er sortert
  - Setter dette på plass nummer *i,* lengden på sortert delarray *øker* med 1
- Hele arrayen vil være sortert etter *n* - 1 gjennomløp

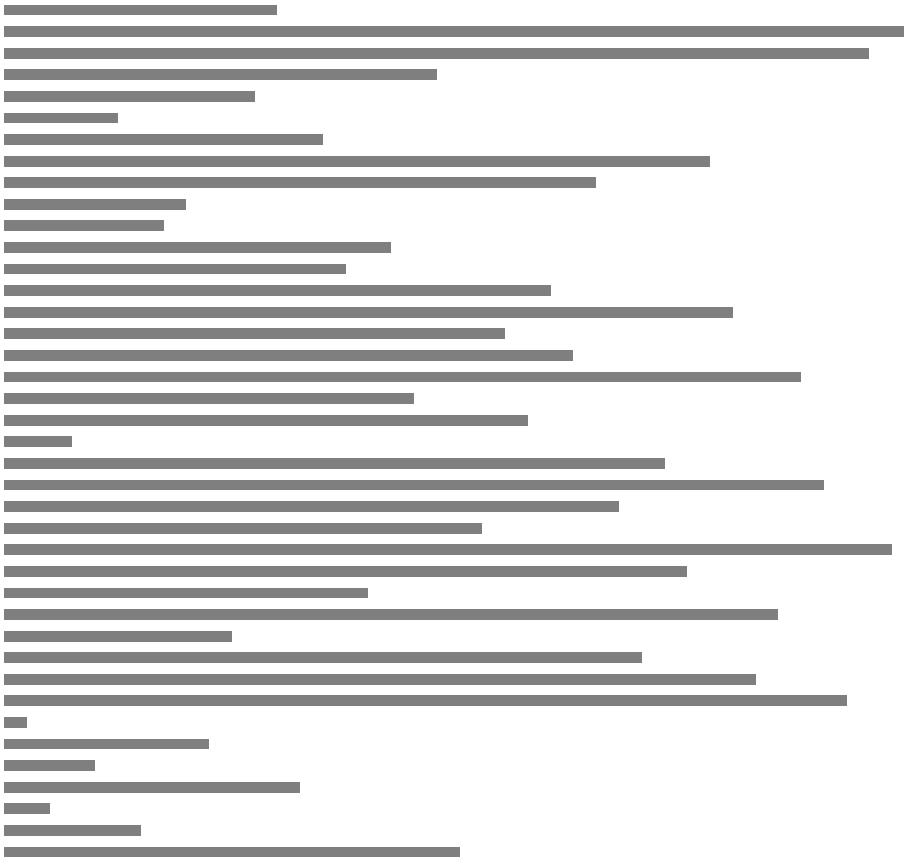
Utplukksortering: Enkelt eksempel

Selection sort animation:

- Red is current min
- Yellow is sorted list
- Blue is current item



Utplukksortering: Animasjon



Utplukksortering av heltall: Kode

```
void selectionSort(int a[])
{
   int n = a.length, tmp = 0;
   for (int i = 0; i < n - 1; i++)
   {
      int min = i;
      for (int j = i + 1; j < n; j++)
      {
         if (a[j] < a[min])
            min = j;
      }
      tmp = a[min];
      a[min] = a[i];
      a[i] = tmp;
   }
}
```

Operasjoner i utplukksortering

- Utenfor løkkene:
  - Parameterovering og tilordninger, konstant tid
  - Anta at dette tar tiden C_1
- Den ytre løkken går n – 1 ganger:
  - I hvert gjennomløp gjøres det tilordninger og ombytting av elementer *utenfor* den indre løkken
  - Anta at operasjonene utenfor indre løkke hver gang tar tiden C_2
  - Hvor mange ganger går den indre løkken i utplukksortering?

Gjennomløp av den indre løkken

- Ytre løkke går $n - 1$ ganger med i lik 0, 1, 2,… , $n - 2$
- Indre løkke går:

$i = 0$	$n - 1$ ganger
$i = 1$	$n - 2$ ganger
$i = 2$	$n - 3$ ganger
.	
.	
.	
$i = n - 3$	2 ganger
$i = n - 2$	1 gang

- Totalt antall gjennomløp av indre løkke i utplukksortering * :

$$1 + 2 + 3 + \ldots + (n - 2) + (n - 1) = n \cdot (n - 1) / 2$$

*: Resultat fra grunnleggende matematikk om sum av naturlige tall, kan bevises ved å bruke induksjon

Tidsforbruk for utplukksortering

- Anta at et gjennomløp av indre løkke tar tiden C_3
- Den totale kjøretiden blir:

$$t = t(n) = C_3\, n\, (n - 1)/2 + C_2\, n + C_1 =$$

$$C_3\, n^2/2 + C_3\, n/2 + C_2\, n + C_1$$

- For store verdier av n dominerer første ledd:

$$t(n) \approx C_3\, n^2/2$$

- Vi sier at algoritmen er av <u>orden</u> n^2 og skriver:

$$t = O(n^2)$$

(Store) *O*-notasjon (Big-Oh)

- Matematisk definisjon, der *t* og *f* begge er funksjoner av *n*:

  t(*n*) er *O*(*f*(*n*)) hvis det finnes positive konstanter *c* og n_0 slik at *t*(*n*) ≤ *cf*(n) når $n \geq n_0$

- Av definisjonen kan vi lese at O-notasjon gir en *øvre grense* for kjøretidsfunksjonen *t*(*n*)

- Kalles også for *asymptotisk tilnærming*

- *O*-notasjon er en veldig grov måte for å klassifisere tidsforbruk. Vi forkorter så mye som mulig, og stryker både lavere ordens ledd og konstanter.

O-notasjon: Noen eksempler og resultater

- $t(n) = 100n$ — *t(n)* er $O(n)$ fordi $t(n) \leq 100\ n$
- $t(n) = 4n + 2000$ — *t(n)* er $O(n)$ fordi $t(n) \leq 5n$ for $n \geq 2000$
- $t(n) = 50$ — *t(n)* er $O(1)$ fordi $t(n) \leq 50 \cdot 1$
- $t(n) = 3n^2 - 100n$ — *t(n)* er $O(n^2)$ fordi for stor n er $n^2 >>$ n

- Hvis $t(n)$ er et polynom av grad r, så er $t(n)$ alltid $O(n^r)$
- Alle logaritmer vokser saktere enn alle potensfunksjoner med positiv eksponent:

  log n er O(n^k) for alle $k > 0$, men n^k er aldri $O(\log n)$

O-notasjon: Kodeeksempler

- $O(n^3)$:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
      System.out.print(“Hællæ”);
```

- $O(n^3)$:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n * n; j++)
    sum++;
```

O-notasjon: Kodeeksempler

- $O(n \cdot \log n)$:

```
for (int i = 0; i < n; i++)
   for (int j = n; j > 1; j = j/2)
      System.out.print("Hællæ");
```

- $O(n \cdot \sqrt{n})$:

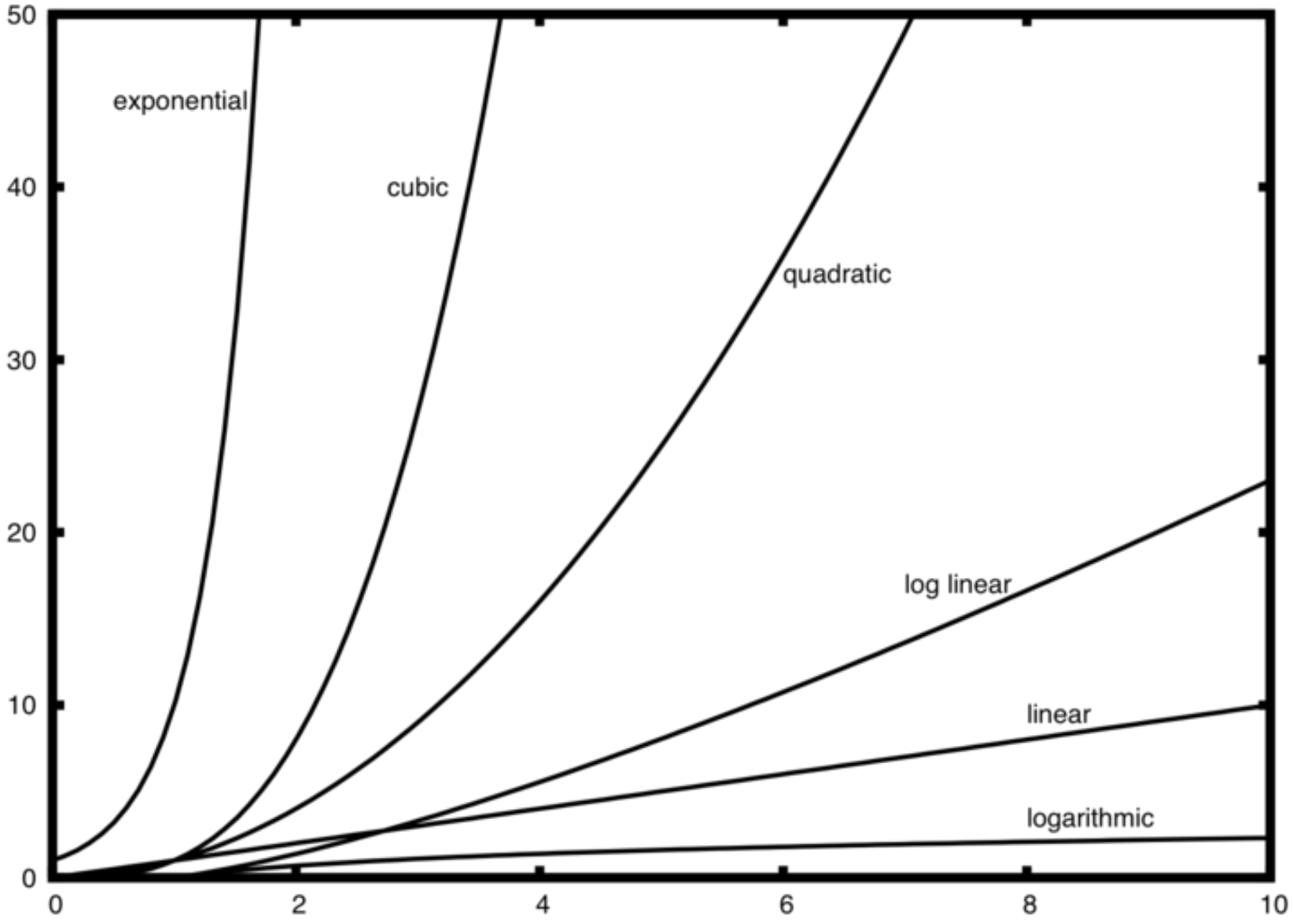
```
for (int i = 0; i < n; i++)
{
    j = 1;
    while (j * j < i)
    {
       sum++; j++;
    }
}
```

Vanlige funksjoner i algoritmeanalyse

• Konstant	$O(1)$	“ingen løkker”
• Lineær	$O(n)$	“enkle løkker”
• Kvadratisk	$O(n^2)$	“to løkker i hverandre”
• Kubisk	$O(n^3)$	“tre løkker i hverandre”
• Logaritmisk	$O(\log n)$	“smarte søk”
• Superlineær	$O(n \log n)$	“smarte sorteringer”
• Eksponentiell	$O(k^n)$	“rå kraft”
• Kombinatorisk	$O(n^n)$, $O(n!)$	“ubrukelig”

Sammenligning av vekst

n	1	$\log_2 n$	n	nlogn	n^2	n^3	2^n	n^n
1	1	0	1	0	1	1	2	1
10	1	3.3	10	33	100	1000	1024	$3 \cdot 10^6$
100	1	6.6	100	664	10 000	10^6	$\sim 10^{30}$	$\sim 10^{158}$
1000	1	9.9	1000	9970	10^6	10^9	$\sim 10^{301}$	$\sim 10^{2567}$


50
40
30
20
10
0
0
2
4
6
8
10
exponential
cubic
quadratic
log linear
linear
logarithmic

Sammenligning av kjøretider

- n = 100 000
- Antar 10^6 (en million) operasjoner i sekundet:

Orden	**Kjøretid**
log n	$1.2 \cdot 10^{-5}$ sekunder
n	0.1 sekunder
n log n	1.2 sekunder
n^2	2.8 timer
n^3	31.7 år
2^n	mer enn et århundre

Hvordan måle kjøretid til et program?

- Java tilbyr metoden `System.currentTimeMillis()`:

```
start = System.currentTimeMillis();
// Algoritmen man skal måle tiden for...
end = System.currentTimeMillis();
tid = end - start;
```

- Siden vi måler kun hele millisekunder, vil vi få 0 som kjøretid hvis algoritmen kjører raskt (f.eks. for små *n*).
- Kan da legge på en løkke, kjøre algoritmen (med samme *n*) et stort antall ganger og beregne en gjennomsnittlig kjøretid

Kontroll av algoritmeanalyse

- Hvis beregnet arbeidsmengde for en algoritme er *O(f(n))* og reell kjøretid er *t(n)*, kan vi se på hvordan **forholdet** *t(n)/f(n)* endres for økende verdier av *n*:
  - Øker: Arbeidsmengden er underestimert
  - Går mot null: Arbeidsmengden er overestimert
  - Går mot en konstant verdi: Godt estimat
- Eksempel: Utplukksortering
- En variant av dette kommer i en av de obligatoriske oppgavene...

Eksempel på over- og underestimering

(fra *Data Structures & Problem solving using Java* av Mark Allen Weiss)

N	T (ms)	T/n	T/n^2	T/(nlogn)
10 000	100	0.01000000	0.00000100	0.00075257
20 000	200	0.01000000	0.00000050	0.00069990
40 000	440	0.01100000	0.00000027	0.00071953
80 000	930	0.01162500	0.00000015	0.00071373
160 000	1960	0.01225000	0.00000008	0.00070860
320 000	4170	0.01303125	0.00000004	0.00071257
640 000	8770	0.01370313	0.00000002	0.00071046

Mangler ved O-notasjon

- Ingen informasjon om algoritmens kompleksitet
- Kan skjule store forskjeller:
  - Algoritme 1: $100000 \cdot n = O(n)$
  - Algoritme 2: $1.0 \cdot n^2 = O(n^2)$
  - Algoritme 2 er raskere så lenge n < 100000 !
- Mer presist:
  - Estimér høyeste ordens ledd og bruk O-notasjon på resten*:

$$t(n) = 100\, n^2 + 20\, n + 30 = 100\, n^2 + O(n)$$

*: Skal implementeres for sorteringsalgoritmer i oblig. 2

Når er O-notasjon feil målestokk?

- Asymptotisk kjøretid er oftest det beste målet for algoritmisk effektivitet, med noen unntak:
  - Bruk-og-kast programmer
  - Datamengdene er alltid små
  - Den raskeste algoritmen er for kompleks og upålitelig
  - Den raskeste algoritmen bruker for mye av RAM eller andre ressurser
  - (Numerisk) nøyaktighet er viktigere enn kjøretid