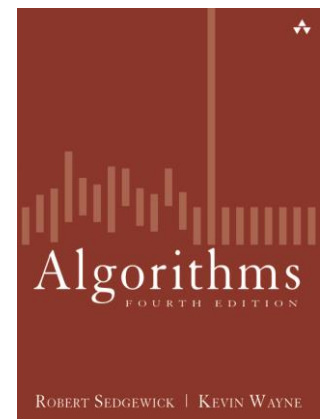


# ID1020: Analys av Algoritmer

Dr. Per Brand  
[pbrand@kth.se](mailto:pbrand@kth.se)



Slides adapted from *Algorithms* 4<sup>th</sup> Edition, Sedgwick.

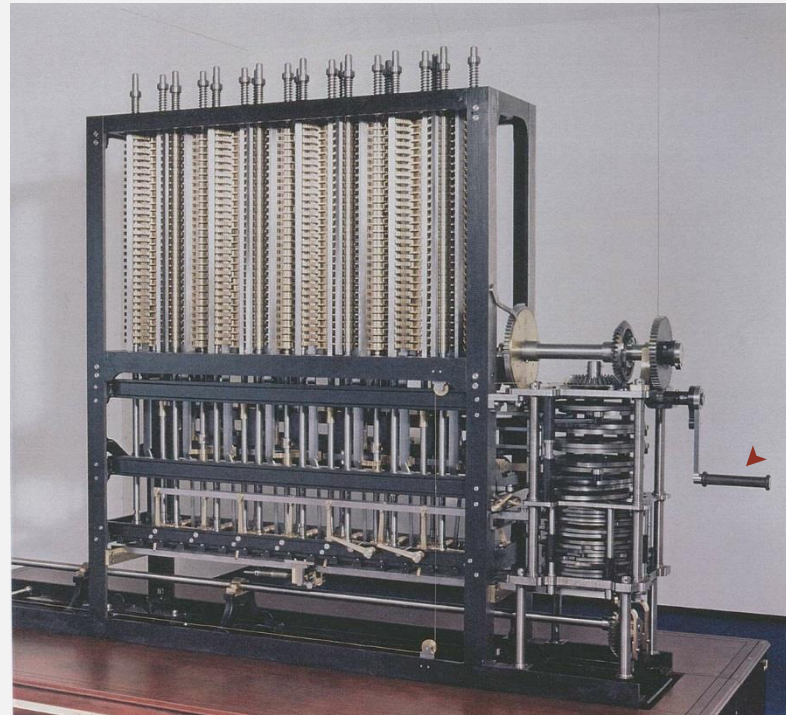
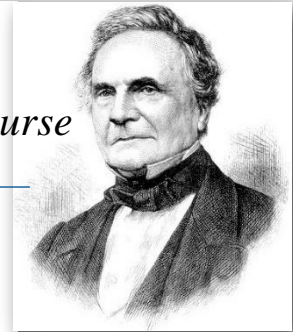
# Analys av algoritmer

- Introduktion
- Observationer
- Matematiska modeller
- Tidskomplexitetsklasser (order-of-growth)
- Komplexitetsteori
- Minneskomplexitet

# Hur många gånger måste vi veva ?

*As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? ”*

*Charles Babbage (1864)*



# DoS med beräkningskomplexitet attackar

“We present a new class of low-bandwidth denial of service attacks that exploit algorithmic deficiencies in many common applications' data structures. **Frequently used data structures have “average-case” expected running time that's far more efficient than the worst case. For example, both binary trees and hash tables can degenerate to linked lists with carefully chosen input.** We show how an attacker can effectively compute such input, and we demonstrate attacks against the hash table implementations in .. the Squid web proxy”

Källa: The Risks Digest ([catless.ncl.ac.uk/Risks](http://catless.ncl.ac.uk/Risks))

# Olika synpunkter



**Programmerare** behöver utveckla en fungerande lösning.



**Kunden** vill lösa problemet på ett effektivt sätt.

**Ni** kan behöva spela alla dessa roller någon dag.



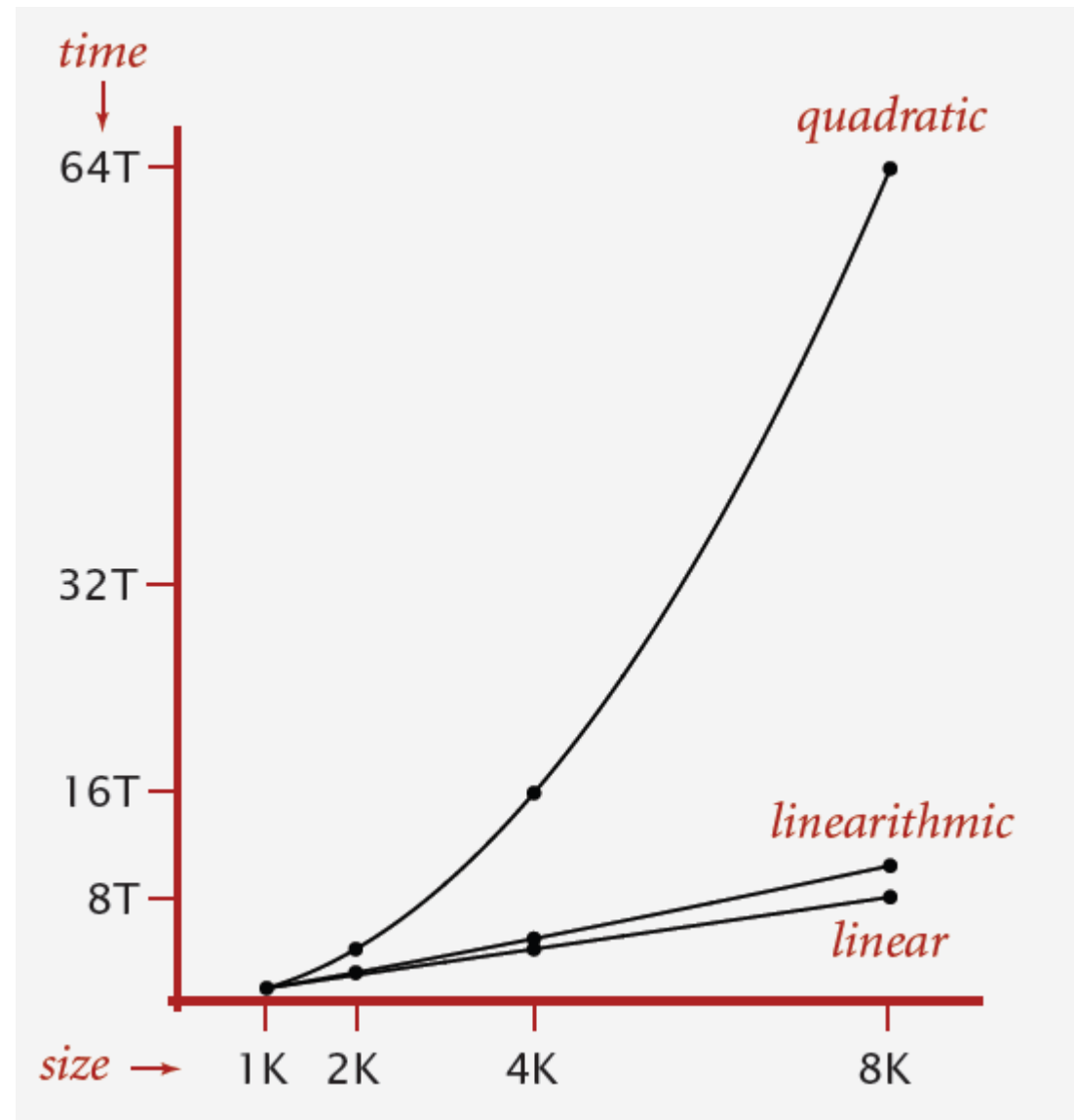
**Teoretikern** vill förstå...

# Syftet med analys

- Uppskatta körtiden av ett program
- Jämför prestanda av olika algoritmer
- Lägg fokus på kodblockar som exekveras mest
  - undvik prestandabrister
- Välj en algoritm för att lösa problemet
  - Ge prestandagarantier
- Förstår teori bakom beräkningskomplexitet
  - └ Kurser mer inriktad på komplexitetsteori på KTH:  
DD1352, DD2352, DD2440

# Algoritmiska framgångar

- N-body simulering.
- Simulera gravitationskrafter mellan  $N$  kroppar.
- Brute force:  $N^2$  steg
- Barnes-Hut algoritm:  $N \log N$  steg  
=> möjliggör ny forskning



# Fibonaccis talföljd

- Fibonaccis talföljd

- Varje tal är summan av de två föregående Fibonaccitalen

- t.ex., 0, 1, 1, 2, 3, 5, 8, 13, 21...

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

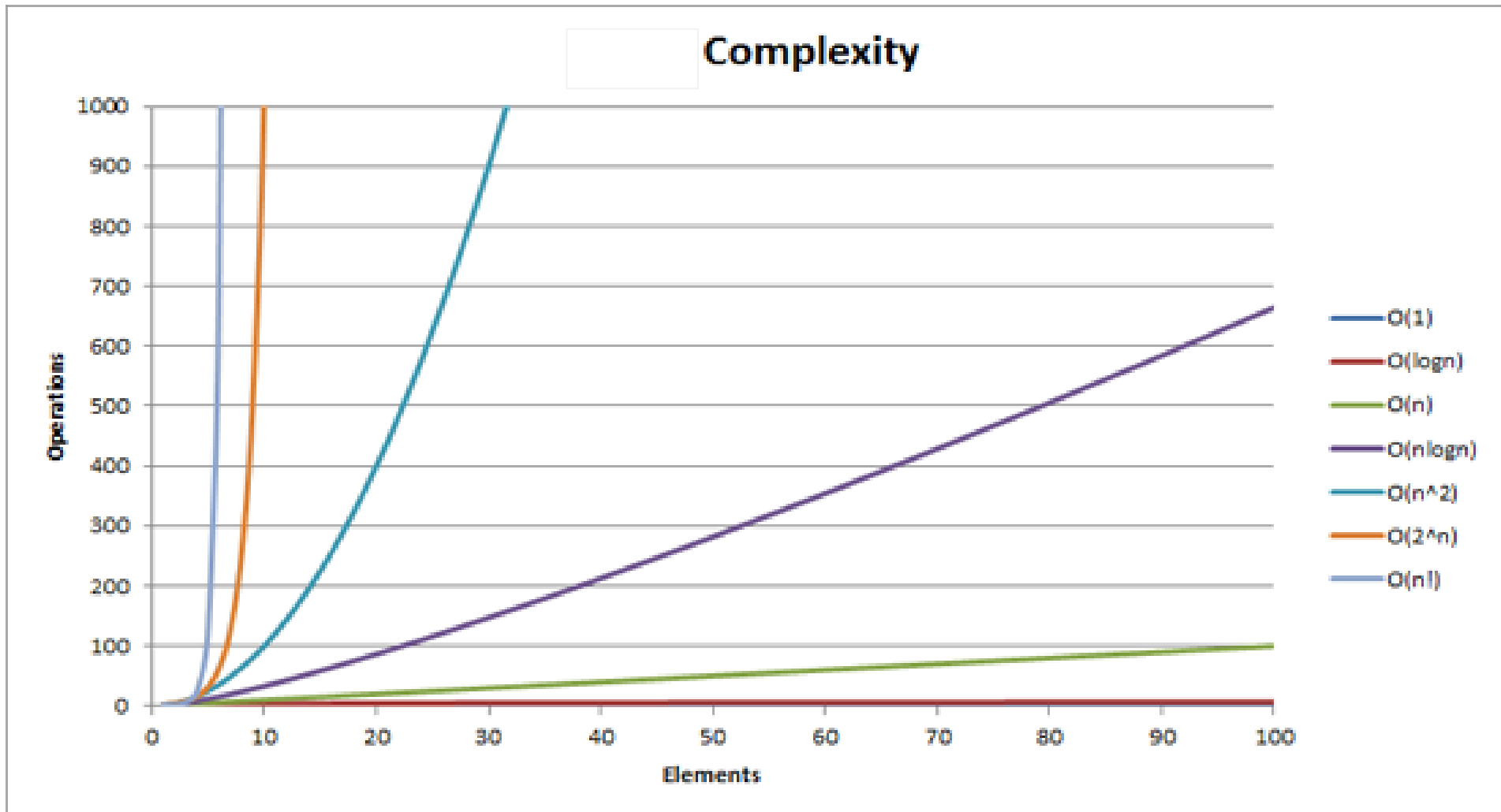
- fibonacci(0) och fibonacci(1) är basfallen

$$F(n) = \begin{cases} 0 & \text{om } n = 0; \\ 1 & \text{om } n = 1; \\ F(n - 1) + F(n - 2) & \text{om } n > 1. \end{cases}$$



# Hur skalar exponentiell komplexitet?

- Fibonacci exempel – rekursiv Fibonacci kan inte ens beräkna fib(50)



# Utmaningen

- Kan mitt program hantera ett stort (vanlig) input?

Varför är programmet långsamt?

Varför tar minnet slut?



- **Insikt.** [Knuth 1970s] Använd den **vetenskapliga metoden** för att förstå prestanda.

# Analys av algoritmer

- Introduktion
- Observationer
- Matematiska modeller
- Tidskomplexitetsklasser (order-of-growth)
- Komplexitetsteori
- Minneskomplexitet

# Analys av algoritmer med vetenskapliga metoden

- Vi vill uppskatta prestandet av algoritmer.
- Vi använder den naturvetenskapliga metoden:
  - Observera någonting ut i världen
  - Hitta en hypotes som är konsekvent med vad man har observerat
  - Förutse nya händelser med hypotesen
  - Kontrollera förutsägelser (hypotesprövning)
    - Försök falsifiera hypotesen
- Principer
  - Ett experiment måste vara reproducerbart av andra
  - En hypotes ska kunna bevisas vara falsk (falsifierbar)

# En exempel-algoritm: 3-SUM

- 3-SUM. Givet  $N$  unika heltal, hur många tripplar summerar till precis noll?

```
>more 8ints.txt
8
30 -40 -20 -10 40 0 10 5

>java ThreeSum 8ints.txt
4
```

	<b>a[i]</b>	<b>a[j]</b>	<b>a[k]</b>	<b>sum</b>
1	30	-40	10	0
2	30	-20	-10	0
3	-40	40	0	0
4	-10	0	10	0

# 3-SUM: Brute-force algorithm

```
public class ThreeSum
{
    public static int count(int[] a)
    {
        int N = a.length;
        int count = 0;
        for (int i = 0; i < N; i++) {
            for (int j = i+1; j < N; j++) {
                for (int k = j+1; k < N; k++) {
                    if (a[i] + a[j] + a[k] == 0)
                        count++;
                }
            }
        }
        return count;
    }
    public static void main(String[] args) {
        int[] a = In.readInts(args[0]);
        StdOut.println(count(a));
    }
}
```

← testa varje trippel  
← förenklad: Integer overflow ignoreras

# Mäta körtiden

- Mäta inte körtiden för hand!
  - Kodblocket nedan mäter körtiden för 3-SUM

```
<dependency>  
  <groupId>edu.princeton.cs.introcs</groupId>  
  <artifactId>stdlib-package</artifactId>  
  <version>1.0</version>  
</dependency>
```



```
public static void main(String[] args) {  
    int[] a = In.readInts(args[0]);  
    Stopwatch stopwatch = new Stopwatch();  
    StdOut.println(ThreeSum.count(a));  
    double time = stopwatch.elapsedTime();  
    StdOut.println("Körtiden var: " + time);  
}
```

# Empirisk analys

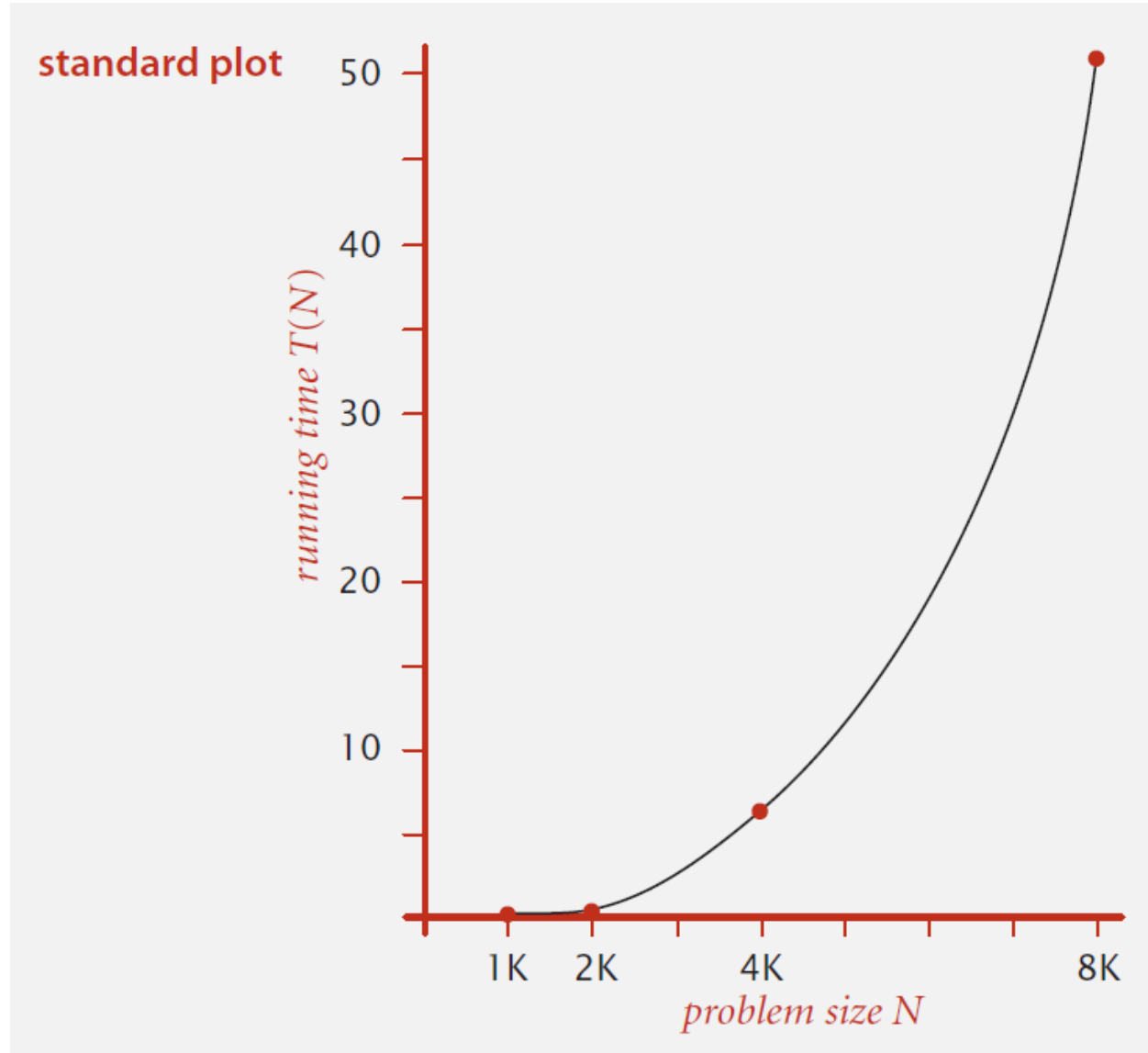
- Kör 3-SUM programmet för olika input-storlekar och mäta körtiden.

N	tid (sekunder)
250	0.0
500	0.0
1.000	0.1
2.000	0.8
4.000	6.4
8.000	51.1
16.000	?



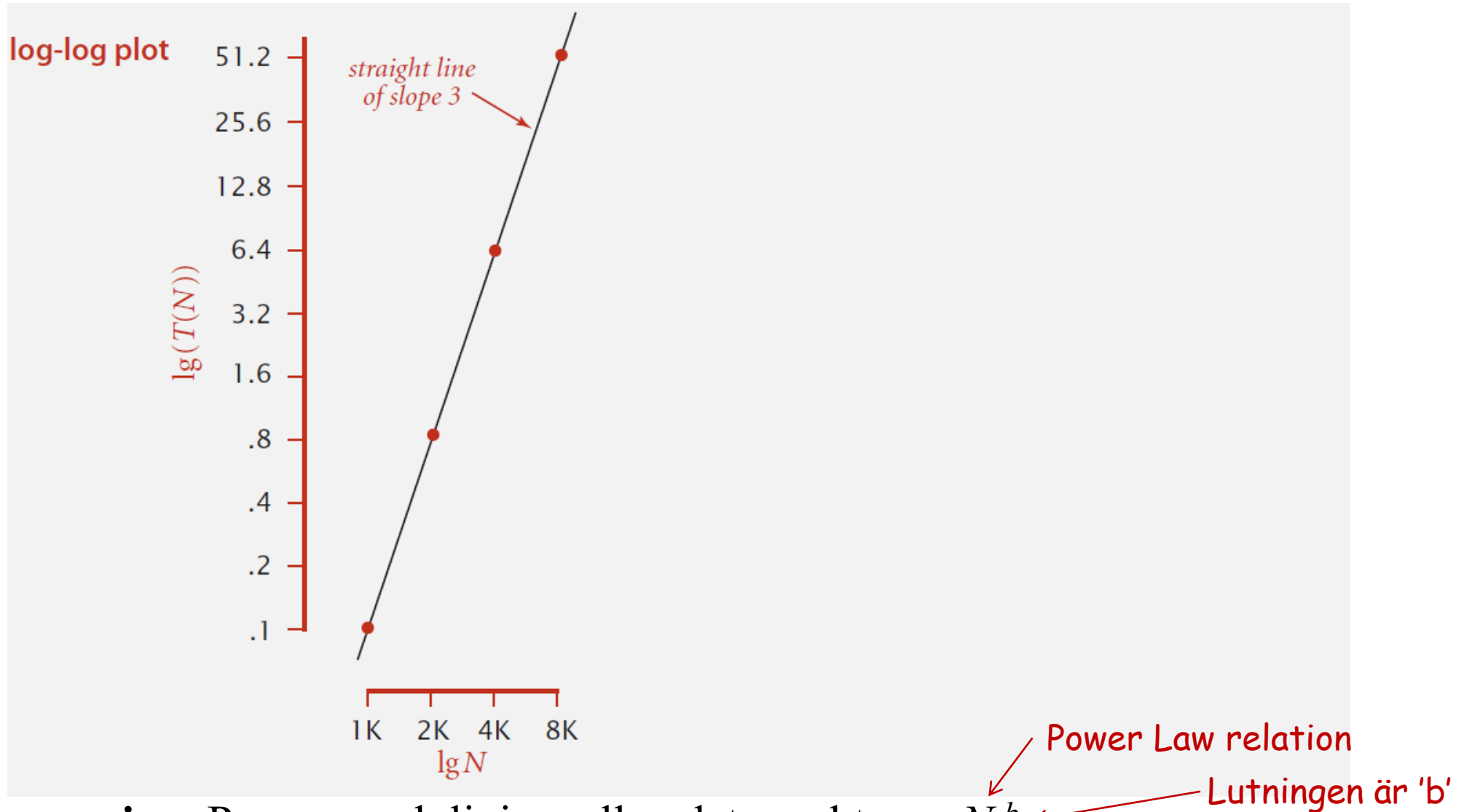
# Data Analys

Vanlig graf: Rita körtiden  $T(N)$  mot input-storlek  $N$ .



# Data Analys

Log-log graf: Rita körtiden  $T(N)$  vs. input-storlek  $N$  med log-log skala.



**Regression.** Passa en rak linje mellan datapunkter:  $a N^b$

**Hypotes.** Körtiden är ungefär  $1.006 \times 10^{-10} \times N^{2.999}$  sekunder.

# Förutsägelser och Validering

- Hypotes. Körtiden är ungefär  $1.006 \times 10^{-10} \times N^{2.999}$  sekunder.
  - Tidskomplexitet (*order-of-growth*) av körtiden är ungefär  $N^3$
- Förutsägelser
  - 51.0 sekunder för  $N = 8.000$ .
  - 408.1 sekunder för  $N = 16.000$ .
- Observationer

N	tid (sekunder)
8.000	51.1
8.000	51.0
8.000	51.1
16.000	410.8

Bingo! Mätningen  
bekräftar hypotesen.

# Fördubblings-hypotes

- Fördubblings-hypotesen (*doubling hypothesis*) är ett snabbt sätt att approximera  $b$  i en power-law relation.
- Man kör programmet och fördubbla input-storleken!

N	tid (sekunder)	Kvot	lg ratio
250	0.0	-	-
500	0.0	4.8	2.3
1.000	0.1	6.9	2.8
2.000	0.8	7.7	2.9
4.000	6.4	8.0	3.0
8.000	51.1	8.0	3.0

← konvergera till  
ett konstant  
 $b \approx 3$

- Hypotes. Körtiden  $\approx a N^b$ , med  $b = \lg \text{kvot}$
- Begränsning. Kan inte identifiera logaritmisk faktorerna med fördubblings-hypotesen.

# Fördubblings-hypotes

- Fördubblings-hypotes: Ett snabb sätt att approximera  $b$  in en power-law relation.
- Hur uppskattar man  $a$  (anta att vi vet  $b$ ) ?
  - Kör programmet (för tillräckligt stort  $N$ ) och räkna ut  $a$ .

N	tid (sekunder)
8.000	51.1
8.000	51.0
8.000	51.1

$$51.1 = a \times 8000^3$$
$$\Rightarrow a = 0.998 \times 10^{-10}$$

- Hypotes. Körtiden är ungefär  $0.998 \times 10^{-10} \times N^3$  sekunder.

# Experiment med algoritmer

- System oberoende faktorer

- algoritm
  - input data
- avgör exponent "b" i  
power law

- System beroende faktorer:

- Hårdvara: CPU, minne, cache, ...
  - Programvara: kompilator, JVM, ...
  - System: operativsystem, nätverk, andra VMs/appar, ...
- avgör konstant "a"  
i power law

- Svårt att få exakta reproducerbara mätningar

- Men mycket lättare för oss att köra ett experiment igen än vad det är i många andra vetenskaper

# Analys av algoritmer

- Introduktion
- Observationer
- Matematiska modeller
- Tidskomplexitetsklasser (order-of-growth)
- Komplexitetsteori
- Minneskomplexitet

# Matematiska modeller för körtid

$$körtid = \sum_{o \in operations} kostnad_o \times frekvens_o$$

- Körtiden för ett program är summan av kostnader av alla operationer gånger frekvensen av operationerna.
- Man behöver analysera programmet för att identifiera alla operationer.
- Kostnaden beror på datorn, kompilatorn, JVM, osv..
- Frekvensen beror på algoritmen, input data.
  - Hitta looparna/rekursion för att identifiera möjliga hotspots
- I princip, finns noggranna matematiska modeller tillgängliga.



# Kostnaden av primitiva operationer

operation	exempel	nanosekunder
integer add	$a + b$	2.1
integer multiply	$a * b$	2.4
integer divide	$a / b$	5.4
floating-point add	$a + b$	4.6
floating-point multiply	$a * b$	4.2
floating-point divide	$a / b$	13.5
sine <code>Math.sin(theta)</code>	arctangent	91.3
...	...	...

[från boken, Running OS X on Macbook Pro 2.2GHz with 2GB RAM]

# Kostnaden av primitiva operationer

operation	exempel	nanosekunder
variable declaration	int a	$C_1$
assignment statement	a = b	$C_2$
integer compare	a < b	$C_3$
array element access	a[i]	$C_4$
array length	a.length	$C_5$
1D array allocation	new int[N]	$C_6N$
2D array allocation	new int[N][N]	$C_7N^2$
string length	s.length()	$C_8$
substring extraction	s.substring(N/2, N)	$C_9$
string concatenation	s + t	$C_{10}N^*$

\*Nybörjare misstag – sträng konkatenering är dyr.

# Exempel: 1-SUM

- Hur många instruktioner exekveras som funktion av input-storleken  $N$  ?

```
int count = 0;
for (int i = 0; i < N; i++) {
    if (a[i] == 0) {
        count++;
    }
}
```

operation	frekvens
variable declaration	2
assignment statement	2
less than compare	$N+1$
equal to compare	$N$
array access	$N$
increment	$N$ till $2N$

# Exempel: 2-SUM

Hur många instruktioner exekveras som funktion av input-storlek  $N$  ?

```
int count = 0;
for (int i = 0; i < N; i++) {
    for (int j = i+1; j < N; j++) {
        if (a[i] + a[j] == 0) {
            count++;
        }
    }
}
```

←  $0 + 1 + 2 + \dots + N =$   
 $(0.5)(N)(N-1) = \binom{N}{2}$

operation	frekvens
variable declaration	$N+2$
assignment statement	$N+2$
less than compare	$(0.5)(N+1)(N+2)$
equal to compare	$(0.5)(N)(N-1)$
array access	$N(N-1)$
increment	$(0.5)(N)(N-1)$ till $N(N-1)$

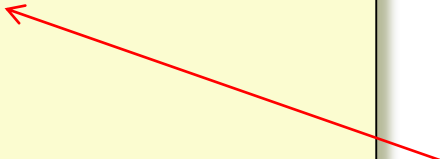
# Förenkla beräkningarna

- Även/redan i 2-SUM, finns det (för) många beräkningar
- Kan vi förenkla den matematiska modellen och ändå vara noggranna?
  - Yes we can!

# Förenkling 1: kostnadsmodell

Kostnadsmodell. Välj en primitiv operation som en proxy för körtiden.

```
int count = 0;
for (int i = 0; i < N; i++) {
    for (int j = i+1; j < N; j++) {
        if (a[i] + a[j] == 0) {
            count++;
        }
    }
}
```


$$0 + 1 + 2 + \dots + N = (0.5)(N)(N-1) = \binom{N}{2}$$

operation	frekvens
variable declaration	$N+2$
assignment statement	$N+2$
less than compare	$(0.5)(N+1)(N+2)$
equal to compare	$(0.5)(N)(N-1)$
array access	$N(N-1)$
increment	$(0.5)(N)(N-1)$ till $N(N-1)$



kostnadsmodell =  
array-accesser

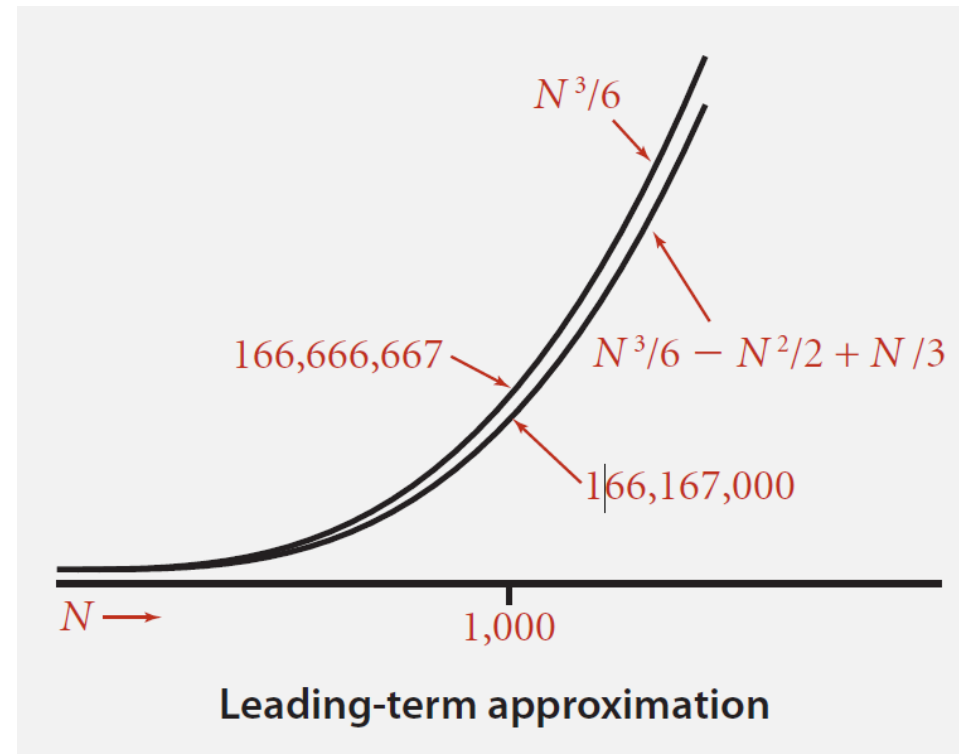
# Förenkling 2: tilde-notation

- Uppskatta körtiden (eller minne) som funktion av input-storlek  $N$ .
- Ignorera lägre ordningens termer (*lower-order terms*)
  - när  $N$  är stort, är lägre ordningens termer försumbara
  - när  $N$  är liten, bryr vi oss inte

T.ex.

$$\frac{1}{6} N^3 - \underbrace{\frac{1}{2} N^2 + \frac{1}{3} N}_{\text{slänga lägre ordningens termer}} \approx \frac{1}{6} N^3$$

slänga lägre ordningens termer  
(t.ex.,  $N=1000$ : 500.000 vs 166 miljon)



Matematisk definition:  $f(N) \sim g(N)$  betyder  $\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = 1$

# Exempel: 2-SUM

- Ungefär hur många array-accesser behövs som funktion av input-storleken  $N$  ?
  - $N^2$  array-accesser.

```
int count = 0;
for (int i = 0; i < N; i++) {
    for (int j = i+1; j < N; j++) {
        if (a[i] + a[j] == 0) {
            count++;
        }
    }
}
```

inre loop

$$0 + 1 + 2 + \dots + N = (0.5)(N)(N-1) = \binom{N}{2}$$

Använda kostnadsmodell och tilde-notation för att förenkla beräkningar



# Exempel: 3-SUM

- Ungefär hur många array-accesser behövs som funktion av input-storleken  $N$  ?

$1/2 N^3$  array-accesser.

```
int count = 0;
for (int i = 0; i < N; i++) {
    for (int j = i+1; j < N; j++) {
        for (int k = j+1; k < N; k++) {
            if (a[i]+a[j]+ a[k] == 0) {
                count++;
            }
        }
    }
}
```

inre loop

$$\binom{N}{3} \approx \frac{1}{6}N^3$$

Använda kostnadsmodell och tilde-notation för att förenkla beräkningar

# Approximera en diskret summa

- Hur approximerar man en diskret summa?
  - Tar en kurs i diskretmatematik.
  - Eller ersätt summan med en integral, och använd matematisk analys.

$$1 + 2 + \dots + N$$

$$\sum_{i'=1}^N i \sim \int_{x=1}^N x dx \sim 1/2 N^2$$

$$1 + 1/2 + \dots + 1/N$$

$$\sum_{i'=1}^N \frac{1}{i} \sim \int_{x=1}^N \frac{1}{x} \cdot dx \sim \ln N$$

$$3\text{-SUM trippel loop}$$

$$\sum_{i'=1}^N \sum_{j'=i}^N \sum_{k'=j}^N 1 \sim \int_{x=1}^N \int_{x=1}^N \int_{x=1}^N dz \cdot dy \cdot dx \sim 1/6 N^3$$

# Matematiska modeller för körtid

- I princip kan man komma på noggranna matematiska modeller
- I praktik
  - formeln kan vara komplicerade
  - avancerad matematik kan behövs
  - låta matematiker tar hand om noggranna modeller istället ☺

$$TN = c1 A + c2 B + c3 C + c4 D + c5$$

*E*

*A* = array access

*B* = integer add

*C* = integer compare

*D* = increment

*E* = variable assignment

*c1..c5* = kostnader  
*A..E* = frekvenser  
av operationer

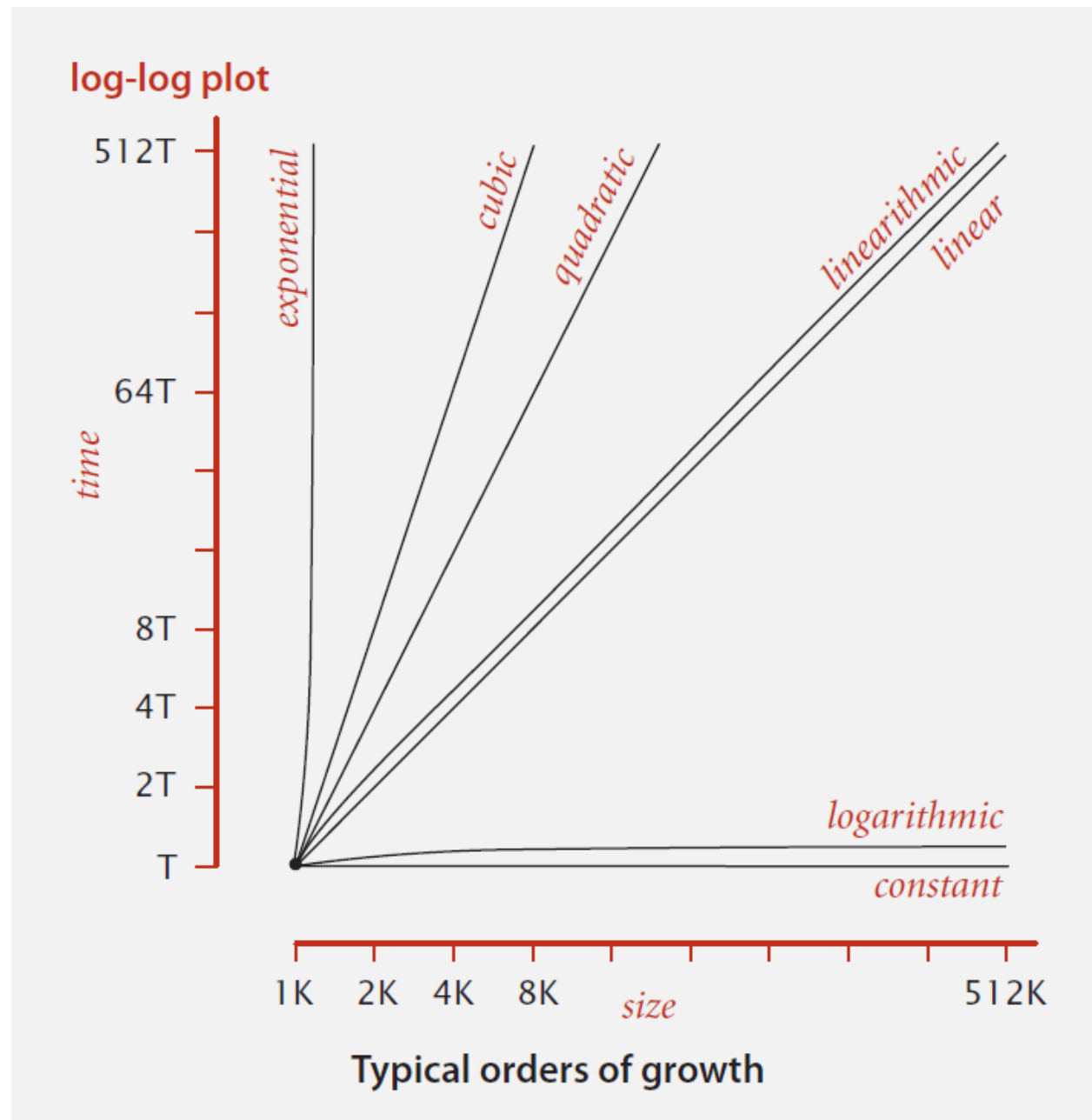
- Vi använder ungefärliga modeller i ID1020:  $T(N) \sim c N^3$ .

# Analys av algoritmer

- Introduktion
- Observationer
- Matematiska modeller
- Tidskomplexitetsklasser (order-of-growth)
- Komplexitetsteori
- Minneskomplexitet

# Vanliga tidskomplexitetsklasser

- Det finns ett få antal funktioner vi behöver veta om som beskriver beräkningstider av vanliga algoritmer



# Vanliga tidskomplexitetsklasser

Beräkningstid	Namn	Exempel
$\sim 1$	konstant	summa två heltal
$\sim \log N$	logaritmisk	binär sökning
$\sim N$	lineär	hitta maximal värde
$\sim N \log N$	linearitmisk	mergesort
$N^2$	kvadratisk	jämför alla par
$N^3$	cubik	jämför alla trippel
$2^N$	exponentiell	jämför alla delmängd

# Vanliga tidskomplexitetsklasser

Beräkningstid	Kodexempel	Beskrivning
1	<code>a = b + c;</code>	statement
Log N	<code>while (N &gt; 1) { N = N / 2; ... }</code>	binärsökning
N	<code>for (int i = 0; i &lt; N; i++) { ... }</code>	loopa och accessa alla element
N log N	[se mergesort föreläsning]	divide and conquer
$N^2$	<code>for (int i = 0; i &lt; N; i++) for (int j = 0; j &lt; N; j++) { ... }</code>	dubbel loop
$N^3$	<code>for (int i = 0; i &lt; N; i++) for (int j = 0; j &lt; N; j++) for (int k = 0; k &lt; N; k++) { ... }</code>	trippel loop
$2^N$	[kombinatorisk sökning]	exhaustive search

# Implikationer av beräkningstider: input-storleken

Storleken på problem som kan lösas inom minuter

Beräkningstid	1970s	1980s	1990s	2000s
1	any	any	any	any
log N	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
N logN	hundreds of thousands	millions	millions	hundreds of millions
$N^2$	hundreds	thousand	thousands	tens of thousands
$N^3$	hundred	hundreds	thousand	thousands
$2^N$	20	20s	20s	30

Vi behöver lineär eller linearithmic algoritmer för att hänga med Moores lag.



# Implikationer av beräkningstider: tiden som behövs

Tiden som behövs för att bearbeta en input-storlek av miljoner

Beräkningstid	1970s	1980s	1990s	2000s
1	instant	instant	instant	instant
log N	instant	instant	instant	instant
N	minutes	seconds	second	instant
N logN	hour	minutes	tens of seconds	seconds
$N^2$	decades	years	months	weeks
$N^3$	never	never	never	millennia

**Vi behöver lineär eller linearithmic algoritmer för att hänga med Moores lag.**

# Sammanfattning av beräkningstider implikationer

Order-of-Growth Fn	Namn	description	effect on a program that runs for a few seconds	
			time for 100x more data	size for 100x faster computer
1	constant	independent of input size	—	—
log N	logarithmic	nearly independent of input size	—	—
N	linear	optimal for N inputs	a few minutes	100x
N log N	linearithmic	nearly optimal for N inputs	a few minutes	100x
N <sup>2</sup>	quadratic	not practical for large problems	several hours	10x
N <sup>3</sup>	cubic	not practical for medium problems	several weeks	4–5x
2 <sup>N</sup>	exponential	useful only for tiny problems	forever	1x

# Exempel analys: binärsökning

# Binärsökning i lista: exempel

- Jag tänker på ett ord i ordlistan. Kan du gissa vilket?
  - Hur många gissningar borde det ta att gissa ordet?(Det finns nästan tvåhundrausen ord i Svenska Akademiens Ordlista.)

**LAT**

Nej, det kommer före LAT.

**FUL**

Nej, det kommer efter FUL

- - - (femton gissningar senare) - - -

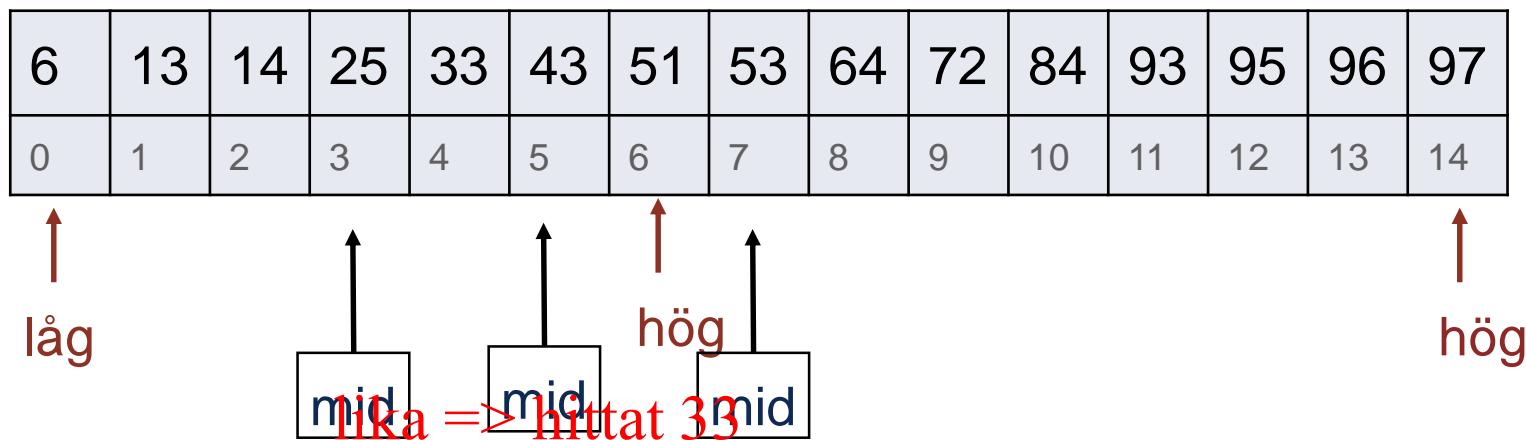
**SJUTTON gissningar totalt.**

Eftersom antalet ord i SAOL är ungefär 2 höjt till 17 är man nere i ett enda möjligt ord efter sjutton halveringar.

# Binärsökning demo

- **Mål.** Givet en sorterad array och en nyckel, hitta indexen av nyckeln i array:n.
- **Binärsökning.** Jämför nyckeln med median elementet.
  - Om värdet är för lågt, gå till vänster.
  - Om värdet är för högt, gå till höger.
  - Om lågt och högt är lika, har vi hittat nyckeln.

T.ex. Leta efter 33 i array nedan med binärsökning



# Binary search: Java implementation

- Är algoritmen lätt att implementera?
  - Första binärsökning algoritmen publicerades 1946.
  - Första utan buggar 1962.
  - En bugg hittades i Java Arrays.binarySearch() så sent som 2006.

```
public static int binarySearch(int[] a, int key) {
    int lo = 0, hi = a.length-1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (key < a[mid]) { hi = mid - 1; }
        else if (key > a[mid]) { lo = mid + 1; }
        else { return mid; }
    }
    return -1;
}
```

en "3-vägs"  
jämförelse

**Invariant.** IF  $key$  finns i array  $a[]$  THEN  $a[lo] \leq key \leq a[hi]$

# Binärsökning: matematiska analys

- **Teorem.** Binärsökning jämför nycklar högst  $1 + \lg N$  gånger för att söka i en sorterad array av storlek  $N$ .
- **Definition.**  $T(N) \equiv$  antal nyckeljämförelser för att binärsöka i en sorterad subarray av storlek  $\leq N$ .

- **Binärsök "recurrence".**  $T(N) \leq T(N/2) + 1$  för  $N > 1$ , med  $T(1) = 1$ .  

$\uparrow$   
vänster eller höger

$\uparrow$   
kan implementeras med bara en  
2- vägs jämförelse (istället för 3-vägs)

- **Bevis (handviftande).**  
[antar att  $N$  är upphöjd till 2]

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && [ \text{givet} ] \\ &\leq T(N/4) + 1 + 1 && [ \text{applicera recurrence till första termen} ] \\ &\leq T(N/8) + 1 + 1 + 1 && [ \text{applicera recurrence till första termen} ] \\ &\vdots \\ &\leq T(N/N) + 1 + 1 + \dots + 1 && [ \text{sluta applicera , } T(1) = 1 ] \\ &= 1 + \lg N \end{aligned}$$

# En $N^2 \log N$ algoritm för 3-SUM

- **Algoritm.**

- Steg 1: Sortera  $N$  (unika) siffror.
- Steg 2: För varje par siffror  $a[i]$  och  $a[j]$ , binärsökning för  $-(a[i] + a[j])$ .

- **Analys.** Tidskomplexiteten är  $N^2 \log N$ .

- Steg 1:  $N^2$  med "insertion sort".
- Steg 2:  $N^2 \log N$  med binärsökning.

**input**

30 -40 -20 -10 40 0 10 5

**sortera**

-40 -20 -10 0 5 10 30 40

**binärsökning**

(-40, -20)	60
(-40, -10)	50
(-40, 0)	40
(-40, 5)	35
(-40, 10)	30
⋮	⋮
(-40, 40)	0
⋮	⋮
(-10, 0)	10
⋮	⋮
(-20, 10)	<del>30</del>
⋮	⋮
(10, 30)	<del>-40</del>
(10, 40)	-50
(30, 40)	-70

räkna bara om  
 $a[i] < a[j] < a[k]$   
för att undvika  
att räkna två  
gångar



# Att jämföra program

- **Hypotes.** Sorterings-baserad  $N^2 \log N$  algoritm för 3-SUM är märkbart snabbare i praktiken än brute-force  $N^3$  algoritm.

NN	tid (sekunder)
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

ThreeSum.java

N	tid (sekunder)
1,000	0.14
2,000	0.18
4,000	0.34
8,000	0.96
16,000	3.67
32,000	14.88
64,000	59.16

ThreeSumDeluxe.java

**Tumregeln.** Bättre tidskomplexitet => snabbare (och billigare) algoritmer.

# Analys av algoritmer

- Introduktion
- Observationer
- Matematiska modeller
- Tidskomplexitetsklasser (order-of-growth)
- Komplexitetsteori
- Minneskomplexitet

# Typ av analys

- **Best case.** Nedre gräns på kostnaden.
    - "Lättaste" input med snabbaste resultat.
    - Ett mål föra alla input.
  - **Worst case.** Övre gräns på kostnaden.
    - "Svåraste" input med långsammaste resultat.
    - Ger en garanti för alla inputs.
  - **Average case.** Förväntad kostnaden för slump input.
    - Behöver en modell för "slump" input.
    - Ett sätt att förutsäga prestanda.
- inte examinerad i ID1020

**T.ex 1.** Array läsningar för brute-force 3-SUM.

Best:  $\sim \frac{1}{2} N^3$

Average:  $\sim \frac{1}{2} N^3$

Worst:  $\sim \frac{1}{2} N^3$

**T.ex 2.** Jämförelser med binärsökning.

Best:  $\sim 1$

Average:  $\sim \lg N$

Worst:  $\sim \lg N$

# Typ av analys

- **Best case.** Nedre gräns (*lower bound*) på kostnaden.
- **Worst case.** Övre gräns (*upper bound*) på kostnaden.
- **Average case.** Förväntad (*expected*) kostnaden.
  
- **Vad händer om input inte matchar input-modellen?**
  - Vi behöver först förstå input för att kunna bearbeta input på ett effektivt sätt.
  
- **Metod 1:** designa för worst case.
- **Metod 2:** randomisera, lita på probabilistiska garantier.

# Komplexitetsteori

- Mål.

- Fastställ "svårighetsgrad" av ett problem.
- Utveckla "optimala" algoritmer.

- Metodik.

- Ta bort detaljer i analys : analysera för ett resultat som gäller inom en konstant faktor
- Ta bort variation i input modellen: lägg fokus på worst case.

- Övre gräns. Prestandsgaranti för algoritmen för alla möjliga input.

- Lägre gräns. Bevis att ingen algoritm är snabbare.

- Optimal algoritm. Lägre gräns = övre gräns (inom en konstant faktor).

# Notation i komplexitetsteori

notation	beskriver	exempel	kort för	används för att
<b>Big Theta</b>	asymptotisk beräkningskomplexitet	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ $\vdots$	klassificiera algoritmer
<b>Big Oh (Stora ordo)</b>	$\Theta(N^2)$ och mindre	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ $\vdots$	hitta övre gränser
<b>Big Omega</b>	$\Theta(N^2)$ och större	$\Omega(N^2)$	$\frac{1}{2} N^2$ $N^5$ $N^3 + 22 N \log N + 3 N$ $\vdots$	hitta lägre gränser

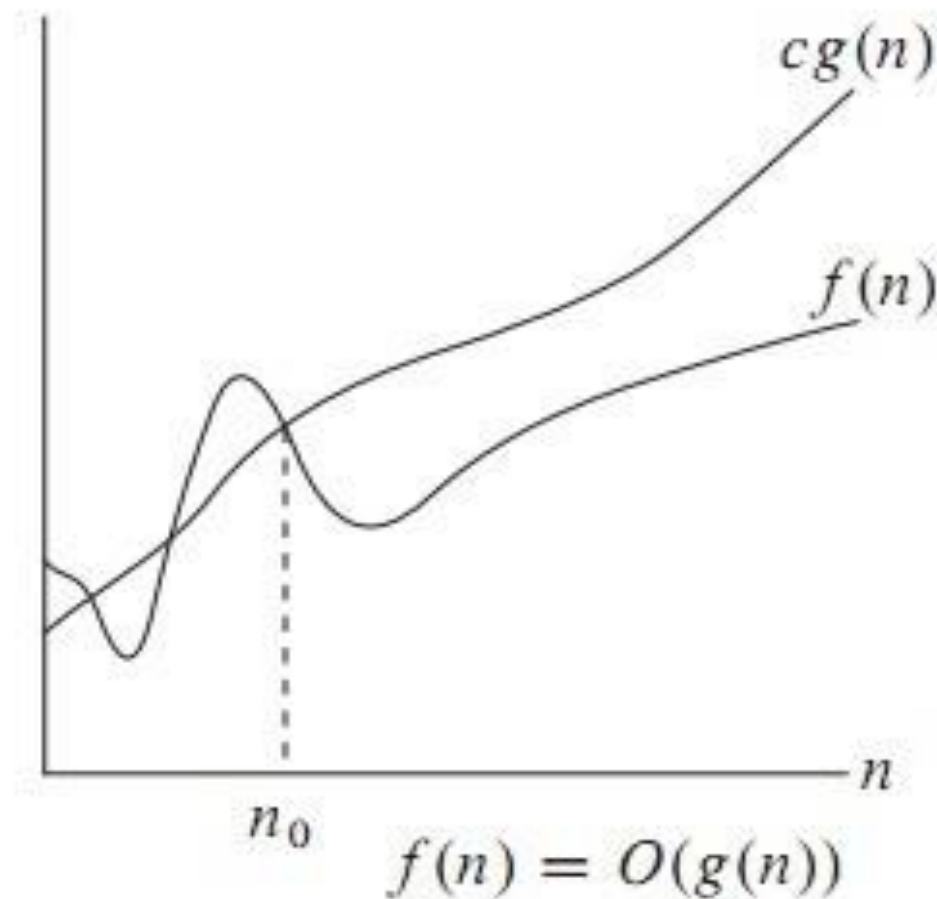
# Tidskomplexitet

- Tidskomplexitet av ett problem är tidskomplexiteten av den snabbaste algoritm som löser problemet.
- $O(n)$ : det finns ingen algoritm som löser problemet snabbare än denna tid
  - måste gälla för *alla input*, dvs, worst case
- $\Omega(n)$ : vi kan bevisa att ingen annan algoritm kan lösa problem i mindre tid.
  - bara en input behöver ta så här mycket tid
- $\Theta(n)$ : Löser både  $O(n)$  och  $\Omega(n)$ .

# Big-Oh ( $O$ )

- Big-Oh bestämmer **övre gränsen** på en funktion.

$f(n) = O(g(n))$  if  $f \exists c > 0$  och  $n_0 > 0$  där  $f(n) \leq cg(n) \forall n \geq n_0$

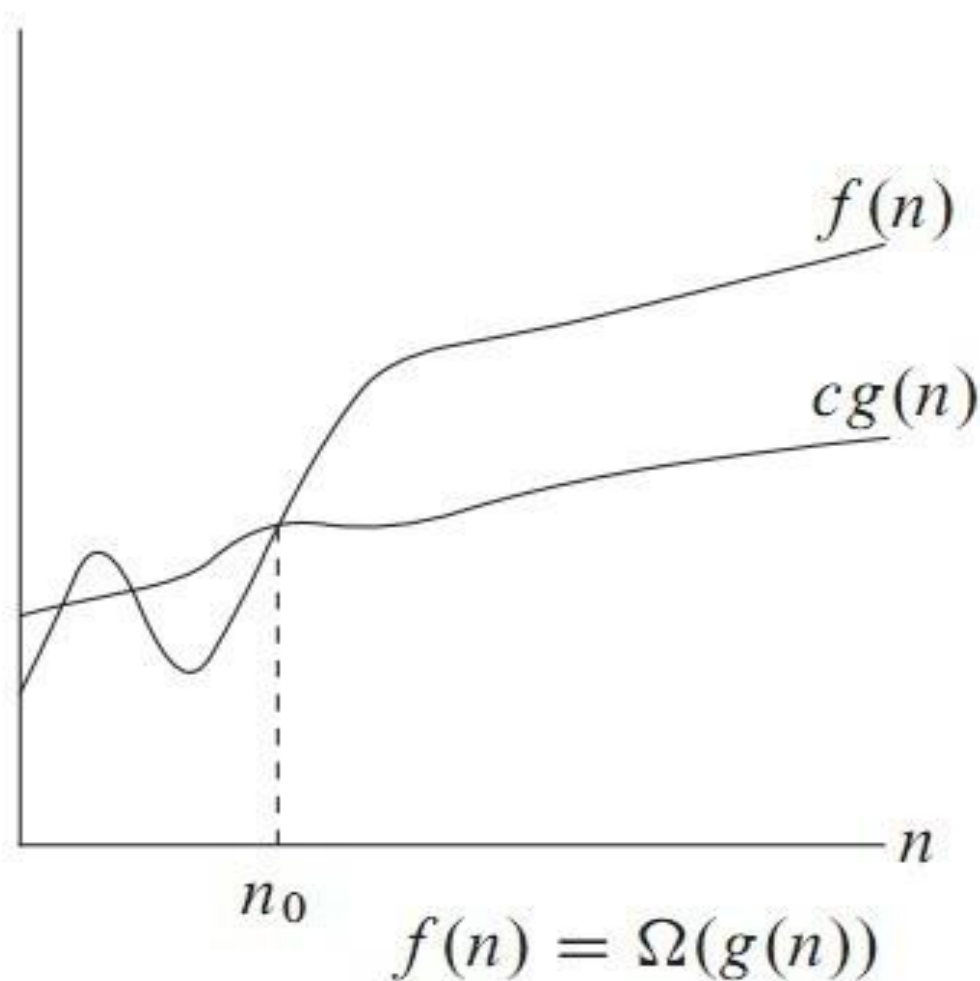




# Big-Omega Notation

- Big-Omega bestämmer **lägre gränsen** på en funktion.

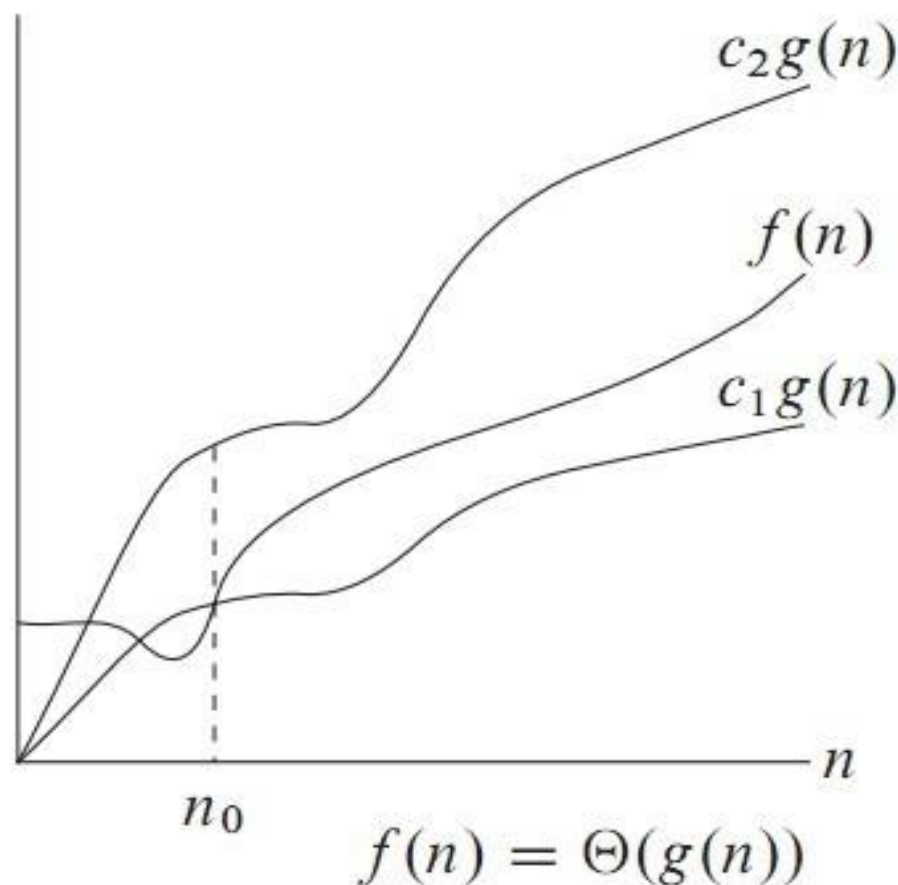
$$\Omega(g(n)) = \{f(n) \mid \exists c > 0 \text{ och } n_0 > 0 \text{ där } 0 \leq cg(n) \leq f(n) \forall n \geq n_0\}$$



# Theta Notation

- Theta notation används när en funktion  $f$  passar inom de lägre och övre gränserna av en annan funktion  $g$ .

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0 \text{ där } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}$$



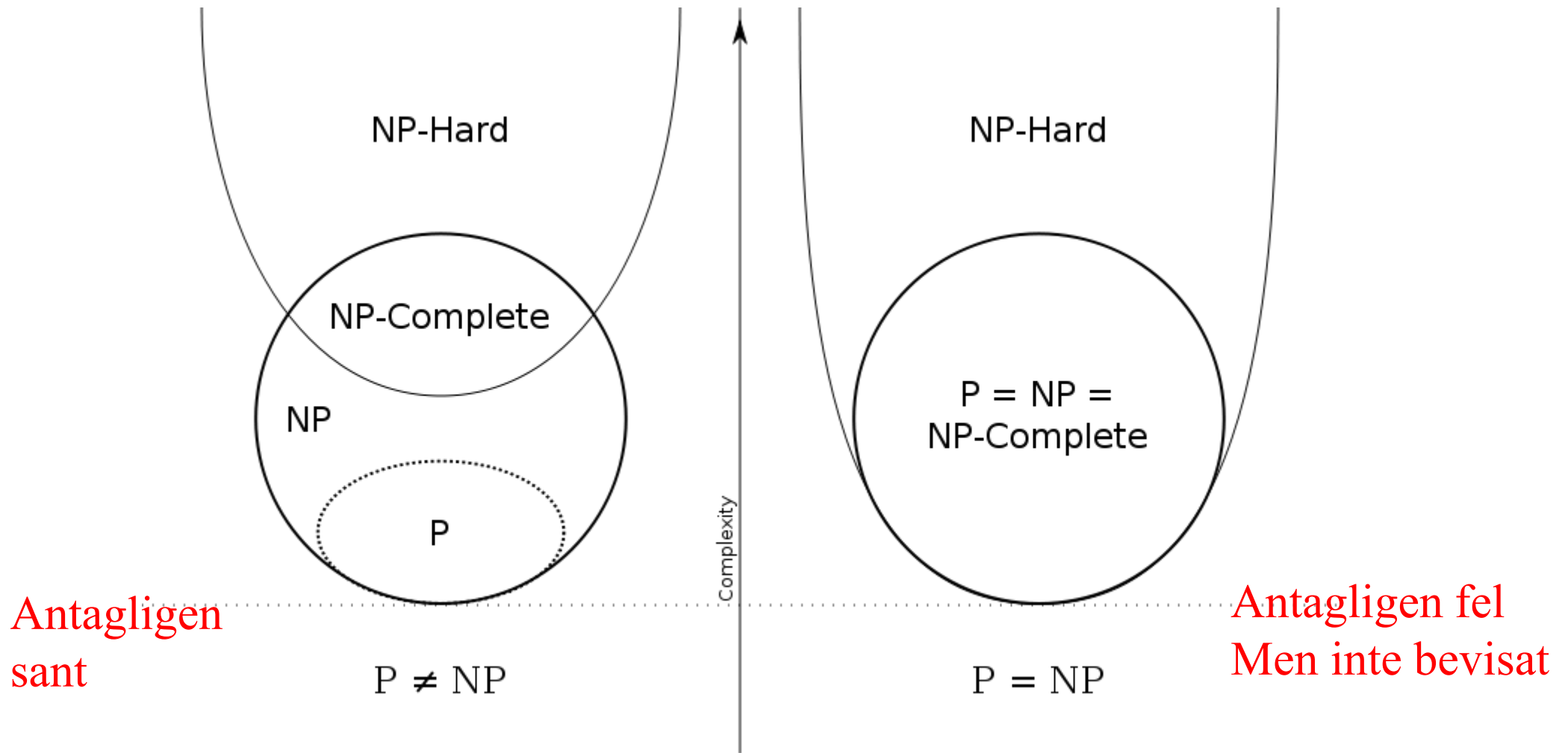
# Klass P

- Klass P består av problem som kan lösas i polynomiell tid.
- Polynomiell tid definieras som en tid  $O(n^k)$  för en konstant  $k$ , där  $n$  är input-storleken till problemet.

# NP(-komplett) problem

- Tiden det tar att lösa NP-komplett problem är proportionell mot antalet kombinationer ( $2^n$ )
  - NP  $\neq$  Non-polynomial time
  - NP = Icke-deterministisk polynomtid (*Non-deterministic Polynomial time*)
- NP är alla tal som kan lösas i polynom tid *om man kan gissa perfekt*.
  - Om man kan gissa "perfekt" löses problemet på kort tid. Dvs, det finns en lösning som tar polynomtid.
  - Med andra ord: NP betecknar problem som kan lösas i polynomiell tid av en icke-deterministisk Turingmaskin.
- Ett problem är NP-komplett när det finns i både NP och NP-hard klasser.
  - NP-hard problem är minst lika svårt som svåraste problem i NP

# P=NP?



[source: wikipedia]

# Komplexitetsteori: exempel

- Mål.

- Fastställ "svårighetsgrad" av ett problem och utveckla "optimal" algoritmer.
- T.ex. 3-SUM.

- Över gräns. En specifik algoritm.

- T.ex. Brute-force algoritm för 3-SUM.
- Körtiden av den optimala algoritm för 3-SUM är  $O(N^3)$ .

# Komplexitetsteori: exempel

- **Mål.**
  - Fastställ "svårighetsgrad" av ett problem och utveckla "optimal" algoritmer.
  - T.ex. 3-SUM.
- **Över gräns.** En specifik algoritm.
  - T.ex. en **förbättrad** algoritm för 3-SUM.
  - Körtiden av den optimala algoritm för 3-SUM är  $O(N^2 \log N)$ .
- **Lägre gräns.** Hitta en bevis att ingen algoritm är snabbare.
  - T.ex. vi behöver examinera alla  $N$  element för att lösa 3-SUM.
  - Körtiden av den optimala algoritm för att lösa 3-SUM är  $\Omega(N)$ .
- **Öppna problem.**
  - Vad är den optimala algoritm för 3-SUM?
  - Finns det en sub-kvadratisk algoritm för 3-SUM?
  - Är kvadratisk tid en lägre gräns för 3-SUM?

# Algorithm design metodik

- Börja.
  - Utveckla en algoritm.
  - Bevisa en lägre gräns.
- Finns det en skillnad mellan den lägre gränsen och den övre gräns?
  - Sänka den övre gräns (upptäck en ny algoritm).
  - Höja den lägre gräns (svårare).
- Den gyllene tiden för algoritm design.
  - 1970-talet.
  - Nedstigande övre gräns för många viktiga problem.
  - Och uppstigande nedre gräns
  - Många kända optimala algoritmen upptäckta.
- Att tänka på.
  - Är det alltför pessimistisk att lägga fokus på *worst case*?
  - Vi behöver mer noggranna mätningar än "inom en konstant faktor" för att förutse prestanda.



# ID1020: vi lägger fokus på Tilde-notation

- Vanlig misstag. Tolka big-Oh som en approximationsmodell.
- ID1020. Fokus på approximationsmodeller: använda Tilde-notation

notation	beskriver	exempel	kort för	används för att
<b>Tilde</b>	leading term	$\sim 10N^2$	$10 N^2$ $10 N^2 + 22 N \log N$ $10 N^2 + 2 N + 37$	bygga approximations- modeller
<b>Big Theta</b>	asymptotisk tillväxtsordning	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$	klassificiera algoritms
<b>Big Oh</b>	$\Theta(N^2)$ och mindre	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$	hitta övre gräns
<b>Big Omega</b>	$\Theta(N^2)$ och större	$\Omega(N^2)$	$\frac{1}{2} N^2$ $N^5$ $N^3 + 22 N \log N + 3 N$ $\vdots$	hitta lägre gräns

# Andra faktorer kan vara viktiga

- Stora konstanter
  - Lägre ordningens term kan vara viktiga
- Icke dominant inre slinga
- Instruktionstider
  - Caching,
- Systemet
  - Minnesreklamation
  - Andra program/aktiviteter
- Starkt beroende av typen av input

# Minneskomplexitet

# Grundkunskap

- Bit. 0 or 1.
- Byte. 8 bitar.
- Megabyte (MB). 1 miljon eller  $2^{20}$  bytes.
- Gigabyte (GB). 1 miljard eller  $2^{30}$  bytes.

NIST  
↓

Datorvetenskaper  
↓



- 64-bitars dator. Vi antar en 64-bitars dator med 8-byte pekare.
  - Kan adressera mer minne.
  - Pekare använder mer minne.



↑  
vissa JVMs "komprimerar" vanliga objekt  
pekare till 4 bytes för att undvika kostnaden

# Vanliga minnesstorlekar för primitiva typer och arrayer

typ	bytes
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8

primitiva typer

typ	bytes
char[]	$2 N + 24$
int[]	$4 N + 24$
double[]	$8 N + 24$

endimensionell arrayer

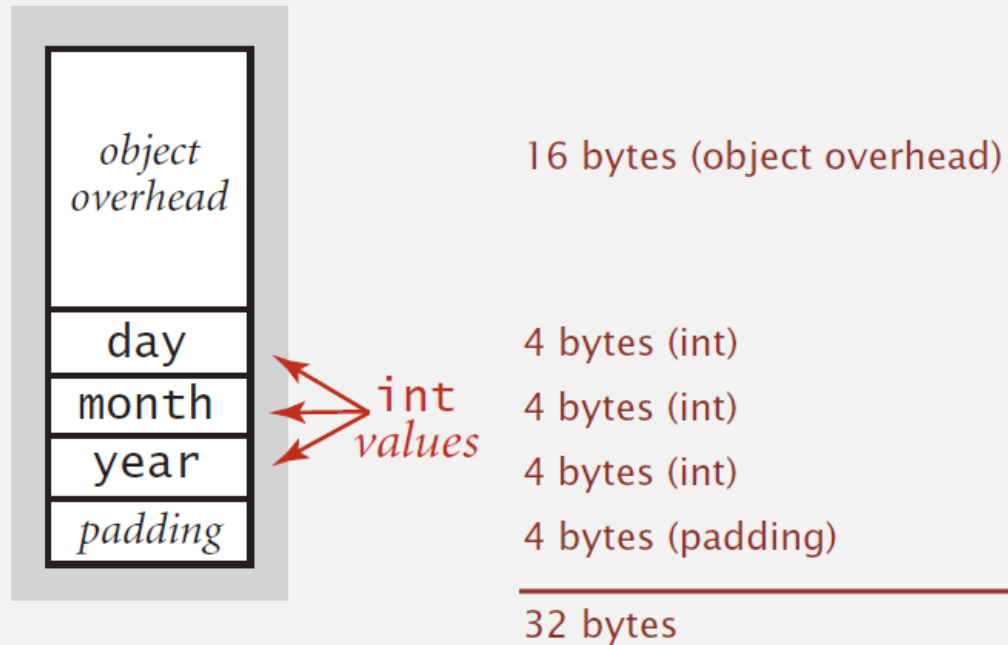
typ	bytes
char[][]	$\sim 2 M N$
int[][]	$\sim 4 M N$
double[][]	$\sim 8 M N$

tvådimensionell arrayer

# Vanliga minnesstorlekar för ett objekt i Java

- Objekt overhead. 16 bytes.
  - Referens. 8 bytes.
  - Padding. varje objekt använder en multipel av 8 bytes.
- 
- T.ex 1. Ett Date objekt använder 32 bytes minne.


```
public class Date
{
    private int day;
    private int month;
    private int year;
    ...
}
```



# Vanliga minnesstorlekar

- **Total minnesstorlekar för en datatyp:**
  - Primitivtyp: 4 bytes för int, 8 bytes för double, ...
  - Objekt referens: 8 bytes.
  - Array: 24 bytes + minne för varje array element.
  - Objekt: 16 bytes + minne för varje attribut.
  - Padding (vadering): runda av uppåt till en multipel av 8 bytes.

+ 8 extra bytes per innre klass objekt  
(en referens till outer klassen)



- **Shallow memory usage:** Räkna inte refererade objekt.
- **Deep memory usage:** Om ett array-element eller attribut är en referens, räkna minnet (rekursivt) för refererade objekt.

# Exempel

- Hur mycket minne använder WeightedQuickUnionUF som funktion av  $N$ ?
- Använd tilde-notation för att förenkla ditt svar.

```
public class WeightedQuickUnionUF {  
    private int[] id;  
    private int[] sz;  
    private int count;  
  
    public WeightedQuickUnionUF(int N) {  
        id = new int[N];  
        sz = new int[N];  
        for (int i = 0; i < N; i++) id[i] = i;  
        for (int i = 0; i < N; i++) sz[i] = 1;  
    }  
    ...  
}
```

← 16 bytes (objekt overhead)

← 8 + (4N + 24) bytes varje  
← (referens + int[] array)

← 4 bytes (int)

← 4 bytes (padding)

---

8N + 88 bytes

Minnet använt:  $8N + 88 \sim 8N$  bytes.



# Sammanfattning

- Empirisk analys.
  - Kör ett program med någon input för att exekvera ett experiment.
  - Anta en "power law relation" och kom på en hypotes för körtiden.
  - Modellen gör det möjligt att **förutsäga programmets prestanda (för olika input-storlekar)**.
- Matematisk analys.
  - Analysera en algoritm genom att räkna frekvensen av operationer.
  - Använd tilde-notation för att förenkla analys.
  - Modellen gör det möjligt att **förutsäga beteende**.
- Vetenskapliga metoden.
  - Matematiska modellen är oberoende av datorsystemet.
  - Empirisk analys är nödvändig för att validera matematiska modeller och för att förutsäga något av intresse (t.ex. prestanda).