ID1020: Analysis of Algorithms

Dr. Per Brand
pbrand@kth.se

Slides adapted from Algorithms 4^{th} Edition, Sedgewick.

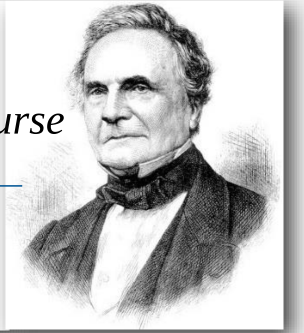
Analysis av algorithms

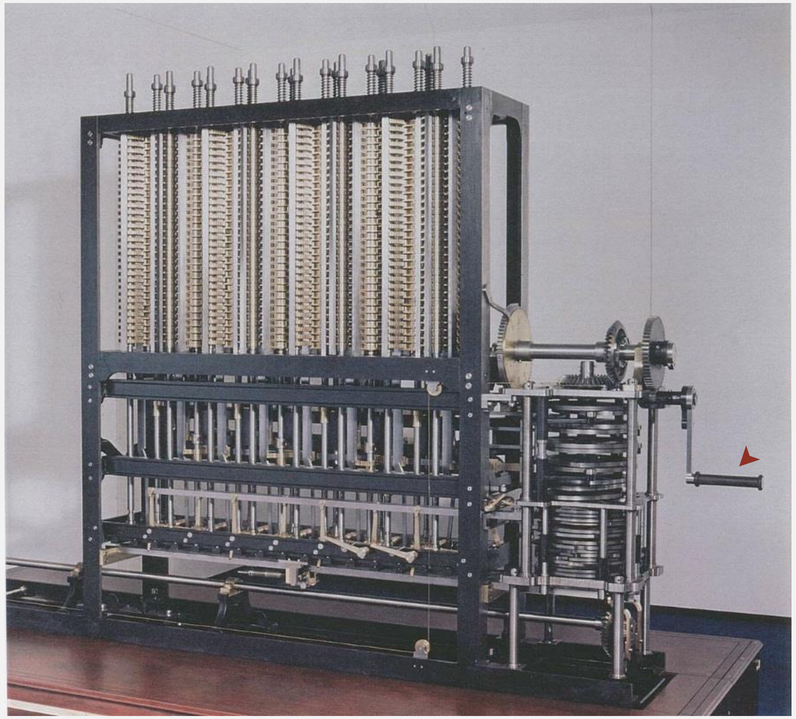
- Introduction
- Observations
- Mathematical models
- Order-of-growth classifications
- Theory of algorithms
- Memory complexity

How many times must we turn the crank?

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 (denial-of-service) attacks

“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”

Källe: The Risks Digest (catless.ncl.ac.uk/Risks)

Cast of characters

Programer need to develop
a working solution.

You might play any or
all of these roles someday

Customer want to solve
problem efficiently.

Theoretician wants to
understand...

Purpose of analysis

- Estimate running time of a program
- To be able to compare algorithms in the context of application needs
- Put focus on code segments that are most frequently run
- Choose an algorithm that suits the problem
  - Provide performace guarantees
- Understand the theory

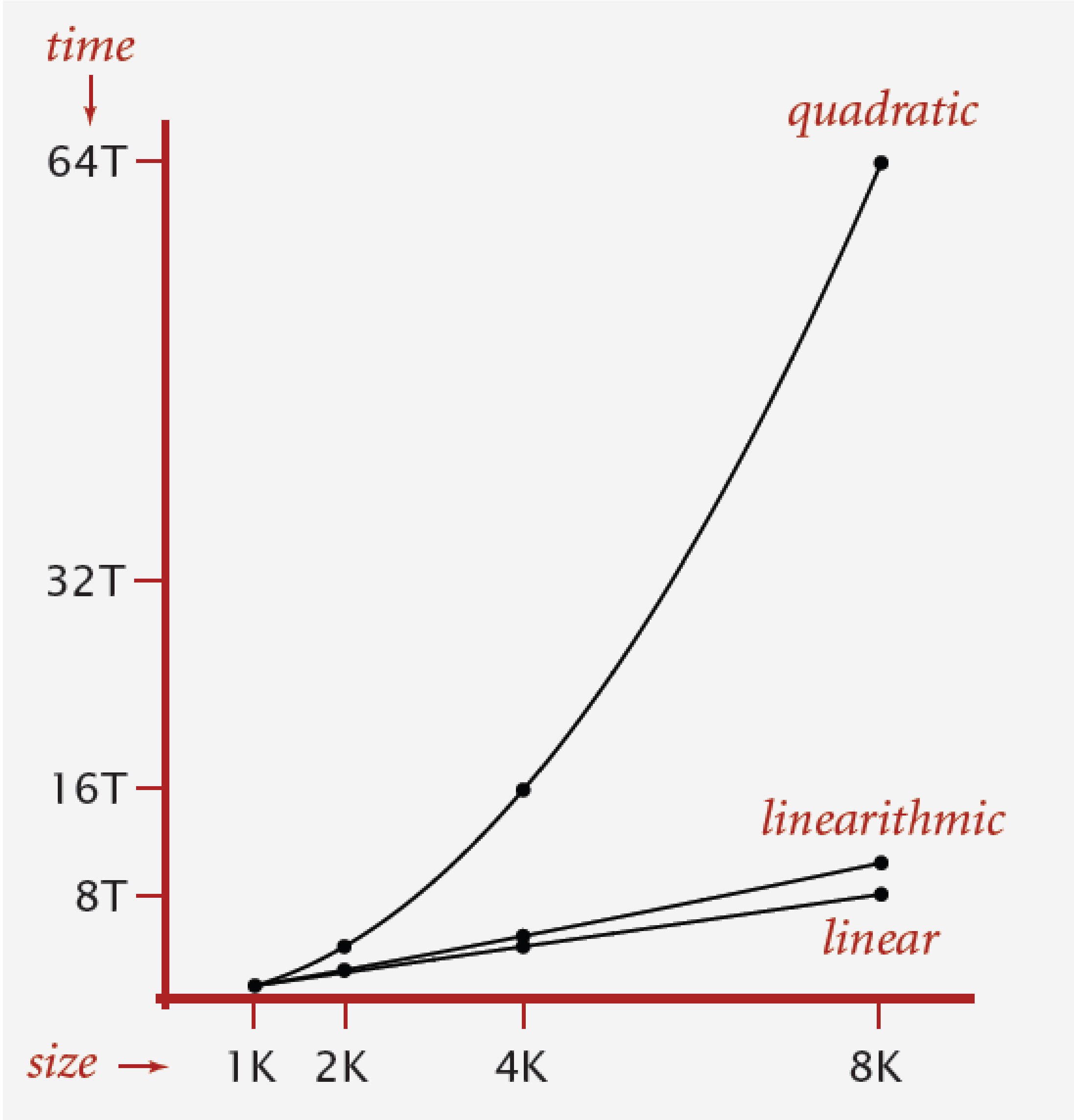
Courses focussed on the theory of algorithms at KTH:
DD1352, DD2352, DD2440

Algorithmic success

- N-body simulation.
- Simulate the gravitational forces involving *N* bodies.

- Brute force: N^2 steps

- Barnes-Hut algorithm: $N \log N$ steps
  => enables new research



Difficult problem

- Fibonaccis sequence
  - Every number is the sum of the two preceding Fibonacci numbers
    - t.ex., 0, 1, 1, 2, 3, 5, 8, 13, 21…

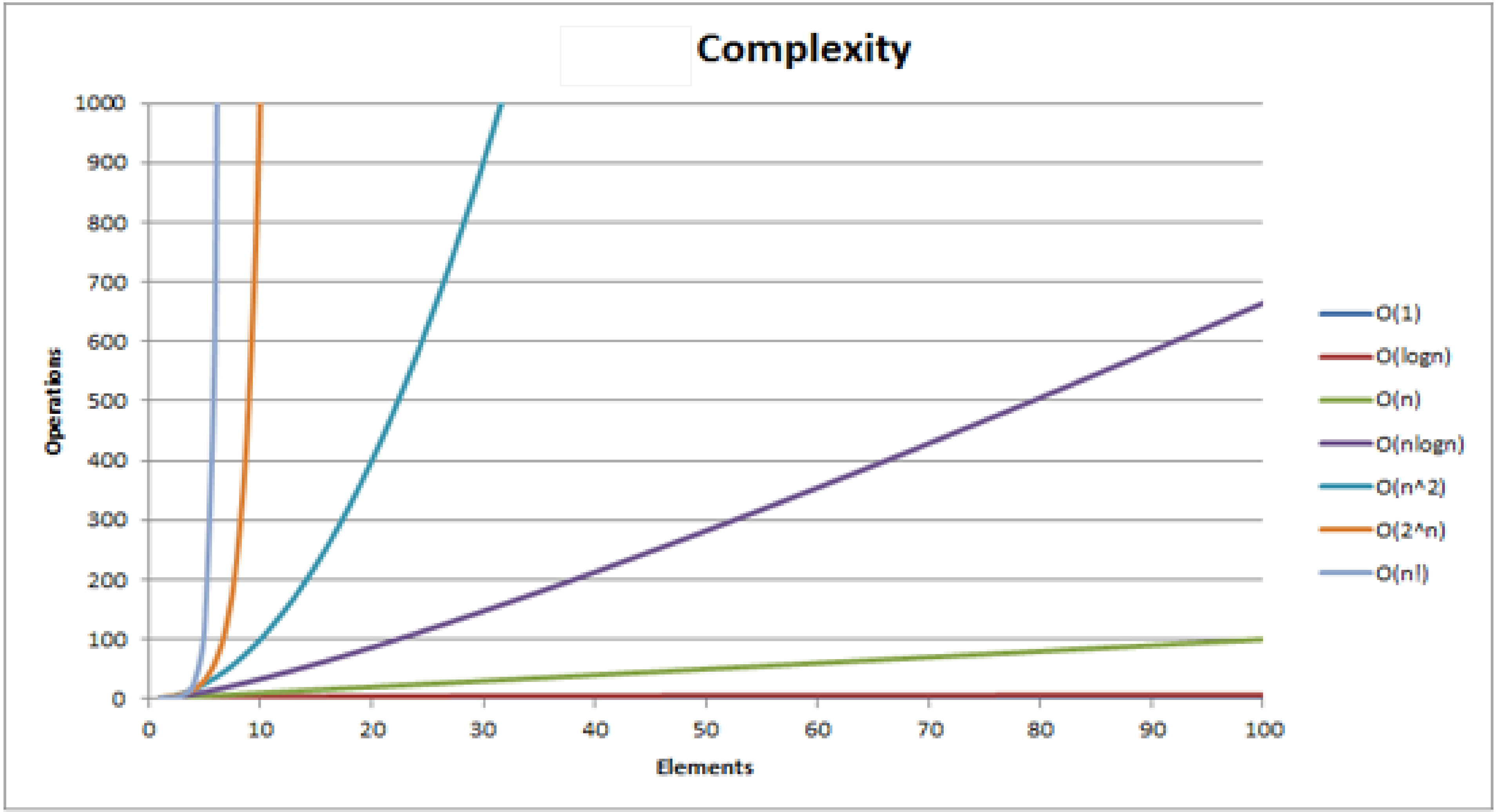
      fibonacci(0) = 0
      fibonacci(1) = 1
      fibonacci(n) = fibonacci(n - 1) + fibonacci(n – 2)

    - fibonacci(0) och fibonacci(1) are the base cases

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

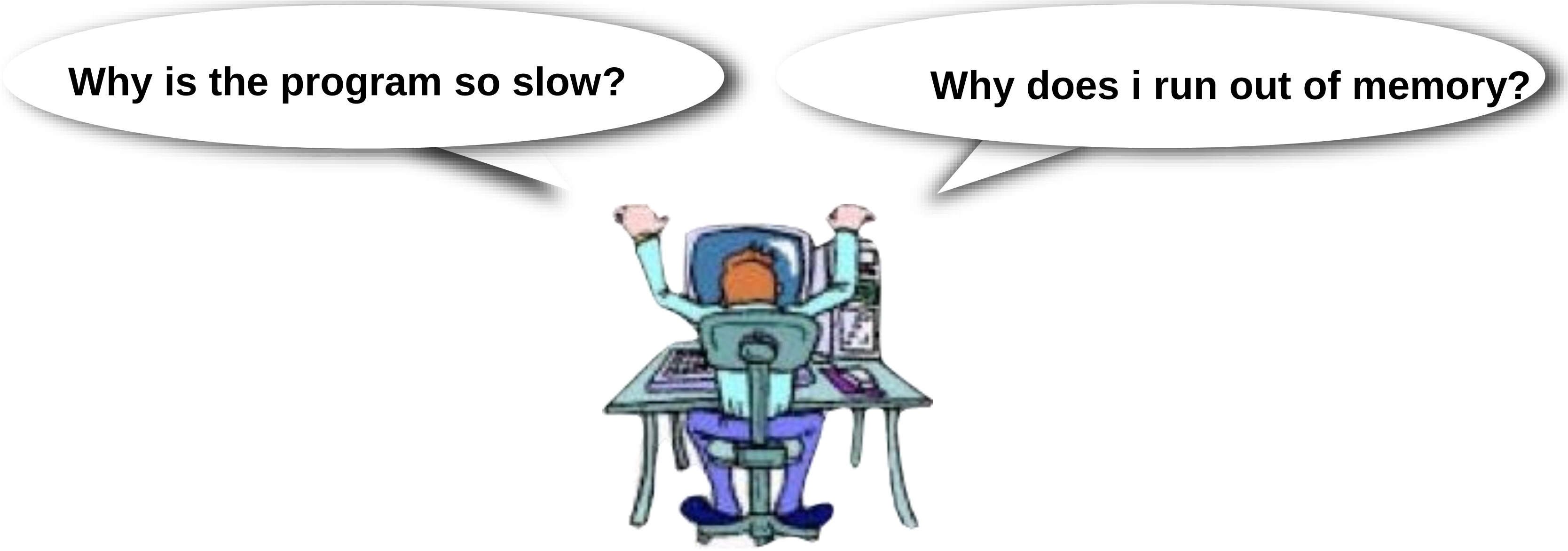
Exponential time complexity is bad news

- Fibonacci exemple – recursive Fibonacci cannot even compute fib(50)



Challenge

- Can my program handle large input?



- Insight. [Knuth 1970s] Use the scientific method to understand performance.

Analys av algorithms

- Introduction
- Observations
- Mathematical models
- Order-of-growth classifications
- Theory of algorithms
- Memory complexity

Scientific method applied to understanding algorithms

- We want of predict the performance of algorithms.
- Scientific method:
  - Observe some feature of the natural world
  - Hypothesize a model that is consistent with observations
  - Predict new events with the hypothesis
  - Verify the predictions
  - Validate repeat until hypothesis and observations agree
- Principles
  - An experiment must be reproducible (by others too!)
  - A hypothesis must be falsifiable
- Feature of the natural world
  - The computer itself

Example: 3-SUM

- 3-SUM. Given *N* integers, how many triplets sum to exactly zero?

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

>java ThreeSum 8ints.txt
4
```

	a[i]	a[j]	a[k]	sum
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));
   }
}
```

Test each triplet

Simplified: Integer overflow ignored

Measure running time

- Don’t measure by hand!
  - Let the computer do it

```
<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);
}
```

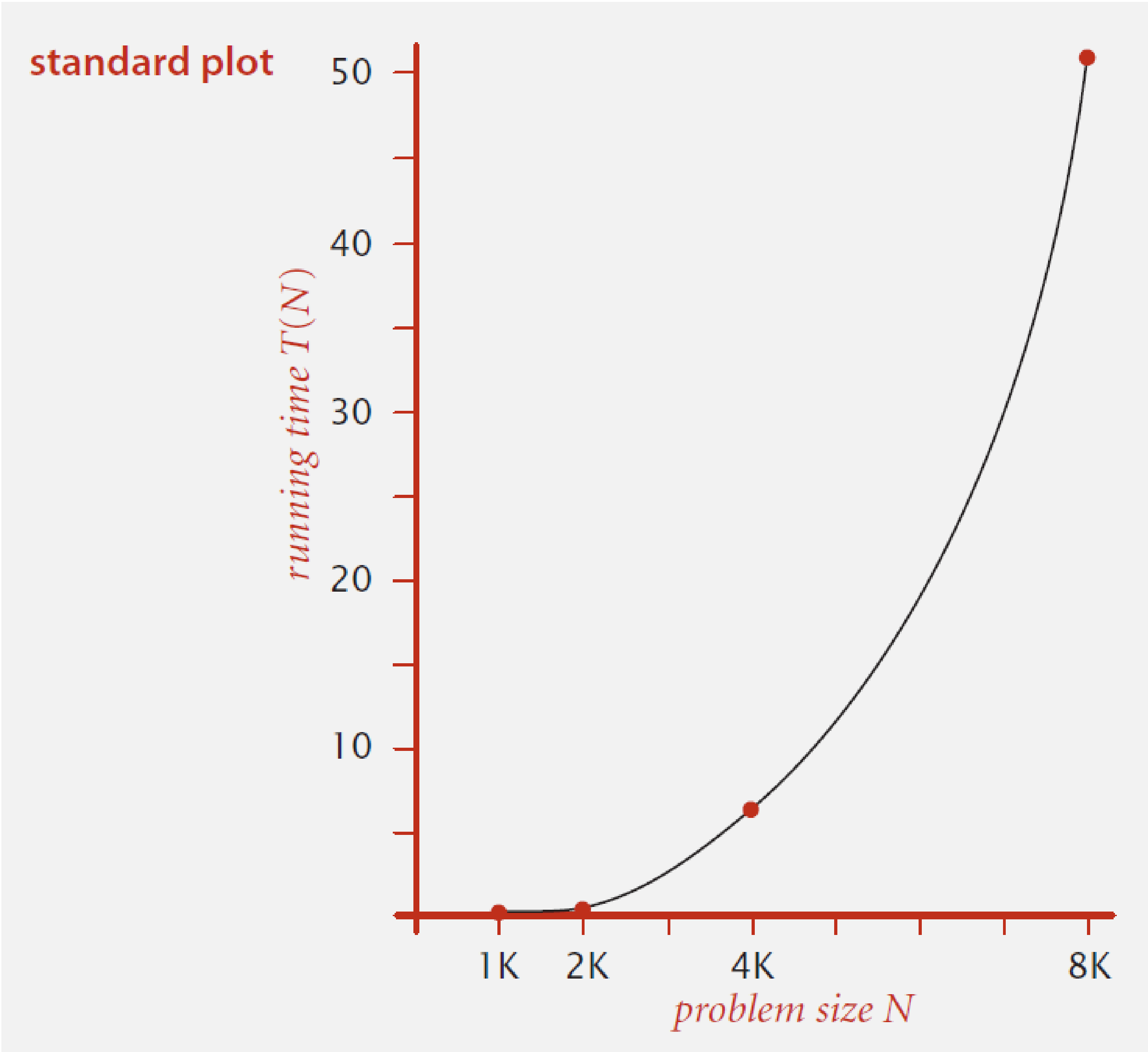
Empirical analysis

- Run the 3-SUM program for different sizes of input and measure the running time.

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	?

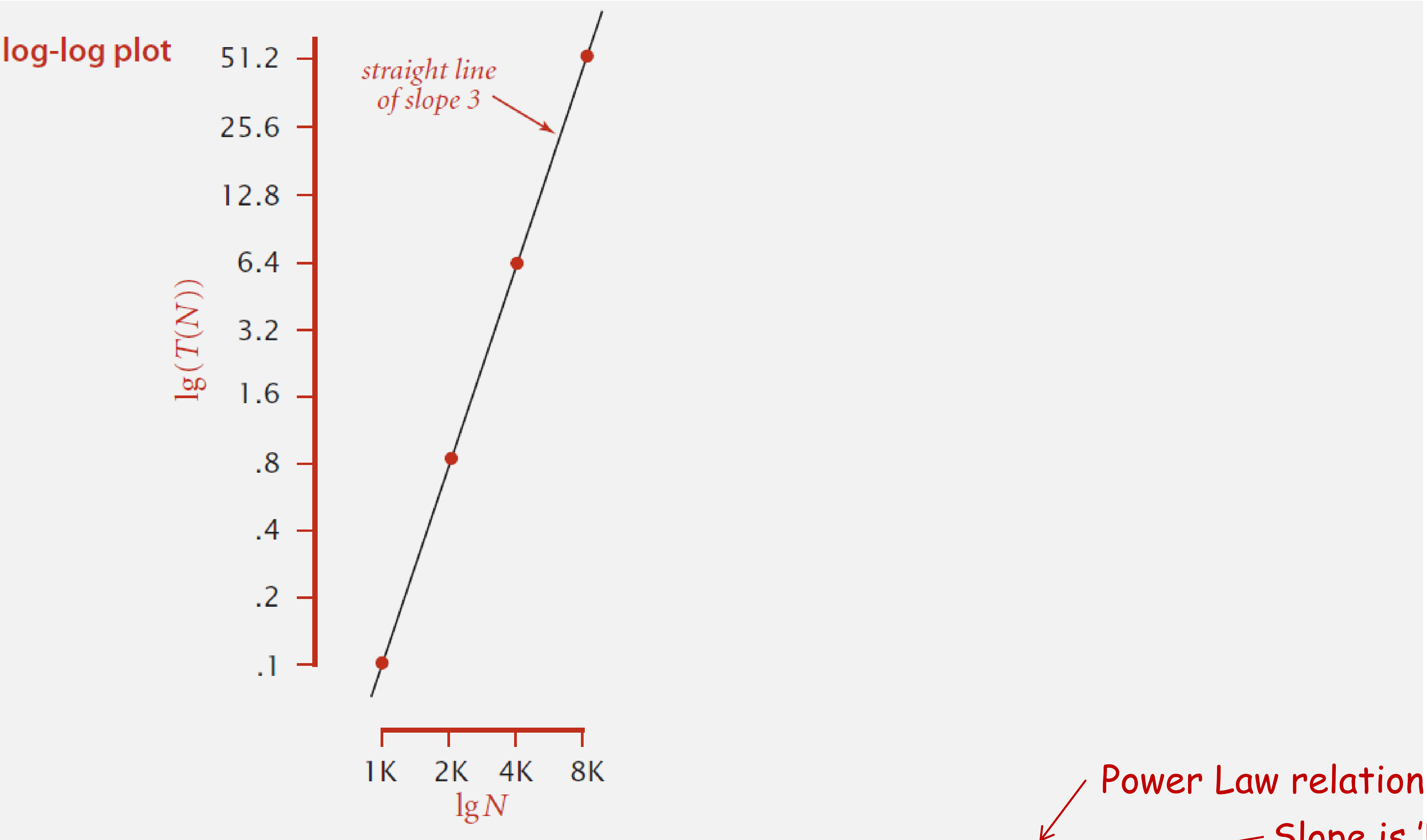
Data Analysis

Standard plot: Running time $T(N)$ vs. input size N.



Data Analysis

Log-log graph: Draw log(T (N)) vs. log(N) .



Power Law relation

Slope is 'b'

Regression. Fit a straight line through data points: $a\ N^{\ b}$

Hypothesis. Running time is approx. $1.006 \times 10^{-10} \times N^{2.999}$ seconds

Prediction and validation

- Hypothesis. Running time is approx. $1.006 \times \times 10^{-10} \times N^{2.999}$ sekunder.
  - Order-of-growth of running time is approximately N^3

- Prediction
  - Last measurement - 51.0 seconds för N = 8.000.
  - We predict - 408.1 seconds for N = 16.000.

- Observationer

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

Bingo! Observation validates hypothesis.

Doubling hypothesis

- The *doubling hypothesis* is a quick way to estimate *b* in a power-law relation.
- Run the program and double the input size!

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
$b \approx 3$

- Hypothesis. Running time $\approx a\, N^{b}$, with b = lg ratio
- Limitation. Cannot identify logarithmic factors

Doubling hypothesis

- Hur do we estimate *a* (assuming that we know *b*) ?
  - Run program for sufficiently large input and compute *a*.

N	time (seconds)
8.000	51.1
8.000	51.0
8.000	51.1

$$51.1 = a \times 80003$$
$$\Rightarrow a = 0.998 \times 10^{-10}$$

- Hypothesis. Running time is approx. $0.998 \times 10^{-10} \times N^{3}$ seconds.

Experimental algorithmics

- System independent factors
  - algorithm
  - input data

determines the exponent ”b” in power law

- System dependent factors:
  - Hardware: CPU, memory, cache, …
  - Software: compiler, JVM, …
  - System:operating system, network other apps , …

determines ”a” in power law

- Difficult to get precise and reproducible observations
  - But much easier than in other sciences!

Analys av algorithms

- Introduction
- Observations
- Mathematical models
- Order-of-growth classifications
- Theory of algorithms
- Memory complexity

Mathematical model

$$running\ time = \sum_{o \in operations} cost_o \times frequency_o$$

- Running time is the sum of the cost of all alla operations times the frequency.
- Need to analyze program/algorithm
- Cost depends on computer, compilet, JVM, etc..
- Frequency depends on the algorithm, and input data.
  - Need to find loops/recursive calls to identify the hotspots

Cost of primitive operations

operation	example	nanoseconds
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 Math.sin(theta)	arctangent	91.3
…	…	…

[from the book, Running OS X on Macbook Pro 2.2GHz with 2GB RAM]

Cost of primitive operations (2)

operation	example	nanoseconds
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	$c_{10}N$*

*Beginner mistake– string concatenation is expensive.

Example: 1-SUM

- How many instructions are executed as a function of the input size *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

Example: 2-SUM

How many instructions are executed as a function of the input size *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++;
        }
    }
}
```

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

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)

Can we simplify the analysis

- Even in 2-SUM, complex calculation

- Can we simplify without losing precision?
  - Yes we can!

Simplication 1: cost model

Cost model. Choose on primitive operation as a proxy for running time.

```
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++;
            }
      }
}
```

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

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)

cost model =
array-accesser

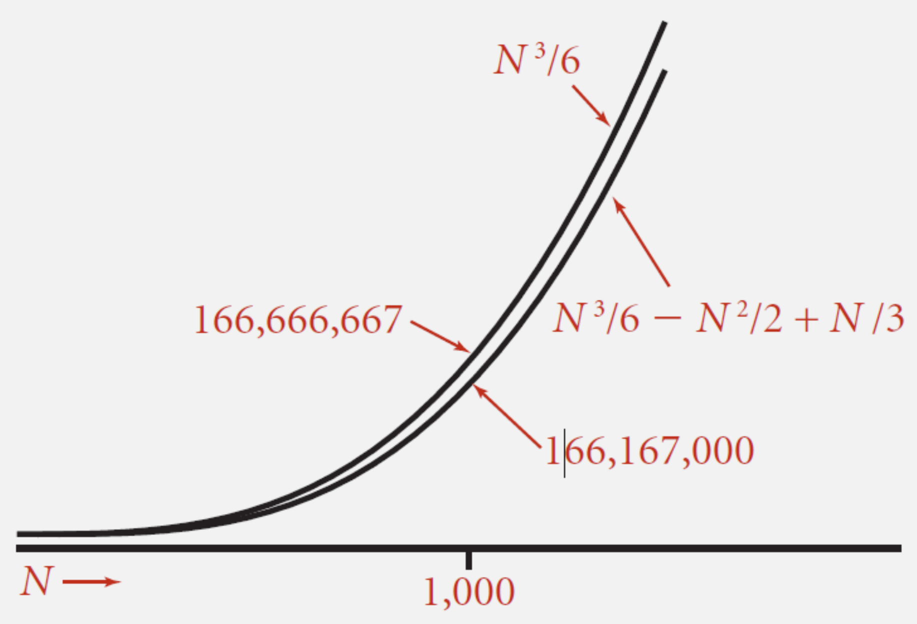
Simplification 2: tilde notation

- Estimate running time (or memory) as a function of the input size N.
- Ignore lower-order terms
  - When N is large, lower order terms are negligible
  - When N is small, we don't care

T.ex.

$$\tfrac{1}{6} N^3 \underbrace{- \tfrac{1}{2} N^2 + \tfrac{1}{3} N}_{\text{ignore lower order terms}} \approx \tfrac{1}{6} N^3$$

(t.ex., N=1000: 500.000 vs 166 miljon



Leading-term approximation

Mathematical definition: $f(N) \sim g(N)$ means $lim_{N \rightarrow \infty} \dfrac{f(N)}{g(N)} = 1$

Example: 2-SUM

- How many array accesses are needed as a function of the input size 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++;
            }
      }
}
```

inner loop

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

Example: 3-SUM

- How many array accesses are needed as a function of the input size *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

$\binom{N}{3} \approx$
${}^1/_6 N^3$

Approximating a discrete sum

- How can we approximate a discrete sum?
  - Take a course in discrete mathematics.
  - Replace the sum with an integral and use calculus.

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

$1 + 1/2 + \ldots + 1/N$ $\quad\quad$ $\sum_{i'=1}^{N} \frac{1}{i} \sim \int_{x=1}^{N} \frac{1}{x}. dx \sim ln$ N

3-SUM triple loop $\quad\quad$ $\sum_{i'=1}^{N} \sum_{j'=i}^{N} \sum_{k'=j}^{N} 1 \sim$

$\int_{x=1}^{N} \int_{x=1}^{N} \int_{x=1}^{N} dz. dy. dx \quad \sim$

$1/6\, N^3$

Mathematical models for running time

- I principle accurate mathematical models are available
- I practice
  - Formulas can be complicated
  - Advance maths may be needed
  - Let the mathematicians deal with it ☺

```
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 = cost
A..E = frequency

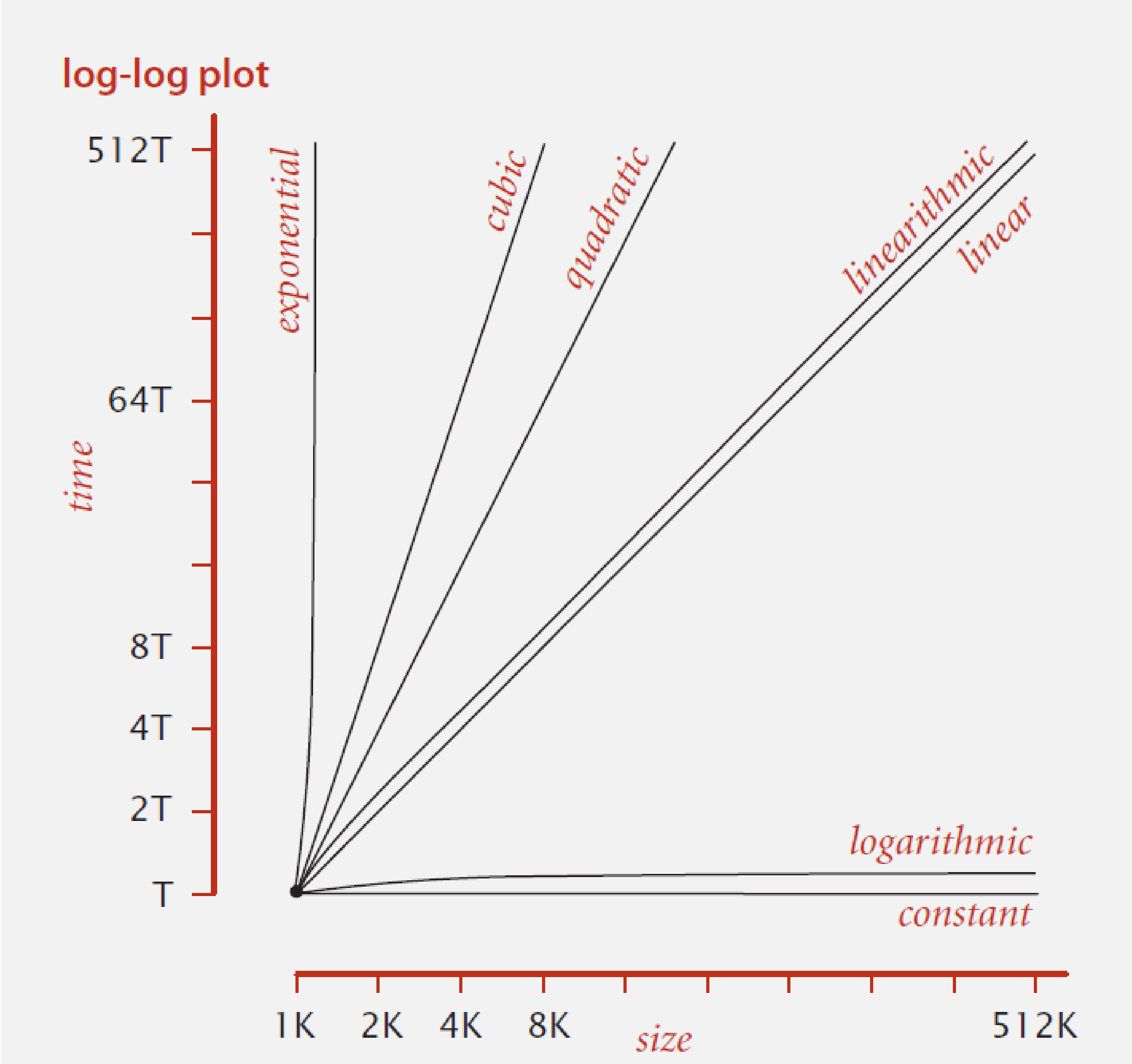
- Vi use approximate models in ID1020: $T(N) \sim c\,N\,3$.

Analysis av algorithms

- Introduction
- Observations
- Mathematical models
- Order-of-growth classifications
- Theory of algorithms
- Memory complexity

Common orders-of-growth

- Small set of functions suffices for most algorithms
- 1, log N, N, N log N,
- N^2, N^3, 2^N



Typical orders of growth

Common order-of-growth classifications

Running time	Name	Example
~1	constant	sum two integers
~logN	logarithmic	binary search
~N	linear	find maximum
~NlogN	linearithmic	mergesort
$\sim N^2$	quadratic	check all pairs
$\sim N^3$	cubic	check all triplets
$\sim 2^N$	exponential	check all subsets

Common order-of-growth classifications

Order of growth	Code example	Description
1	`a = b + c;`	statement
Log N	`while (N > 1)` `{ N = N / 2; ... }`	binary search
N	`for (int i = 0; i < N; i++)` `{ ... }`	single loop
N log N	[se mergesort föreläsning]	divide and conquer

N^2	`for (int i = 0; i < N; i++)`
	`for (int j = 0; j < N; j++)`
	`{ ... }`
		double loop

N^3	`for (int i = 0; i < N; i++)`
	`for (int j = 0; j < N; j++)`
	`for (int k = 0; k < N; k++)`
	`{ ... }`
		triple loop

2^N	[compinatorial search]	exhaustive search

Implications when varying input size

Size of problem that can be solved in a few minutes

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 need linear of linearithmic algorithms to keep pace with Moores law.

Implications when varying input size (2)

Time need when the size of the input is in the millions

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 need linear of linearithmic algorithms to keep pace with Moores law.

More implications

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^2	quadratic	not practical for large problems	several hours	10x
N^3	cubic	not practical for medium problems	several weeks	4–5x
2^N	exponential	useful only for tiny problems	forever	1x

Analysics example: binary search

Binary Search in a list

- I'm thinking of word in a long alphabetically ordered list. Can you guess which one?
  - How many guesses are needed?

  (There are roughly two hundred thousand words in the Swedish Academy Wordlist.)

LAT

No, it comes before LAT.

FUL

No, it comes after FUL

- - - (15 guesses later) - - -

17 guesses in total

As the number of words is approximately 2^{17} only one word is left after 17 guesses.

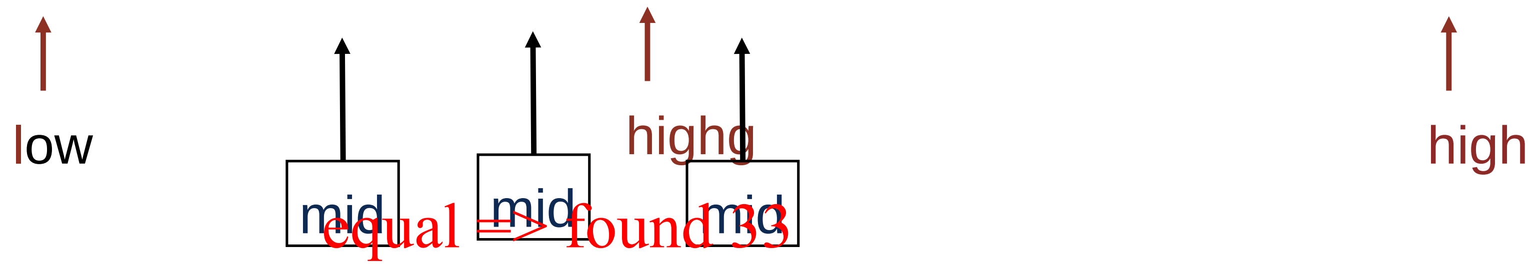
Binary search

- Goal. Given a sorted array and a key, find the index of the key in the array.

- Binary search. Compare key with median of remaining candidates.
  - If too low, go right.
  - If too high go left.
  - If low == high then we found the word (if it exists).

For example, search for 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14



Binary search: Java implementation

- Is the algorithm easy to implement?
  - Algorithm first published in 1946.
  - First bug-free implementation in 1962.
  - A bug was found in Java Arrays.binarySearch() as recently as 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;
}
```

3-way compare

Invariant. IF `key` exists in the array `a[]` THEN `a[lo]` ≤ `key` ≤ `a[hi]`

Binary search: mathematical analysis

- **Theorem.** Binary search compares keys at most $1 + \lg N$ times to search in a sorted array of size N.
- **Definition.** $T(N)$ $\equiv$ number of key comparisons in a sorted subarray of size $\leq N$.
- **Binary search "recurrence".** $T(N) \leq T(N/2) + 1$ för $N > 1$, med $T(1) = 1$.

Left or right half

- **Proof**
  [assume *N* is a power of 2]

$$
\begin{aligned}
T(N) &\leq T(N/2) + 1 && \text{[given]} \\
&\leq T(N/4) + 1 + 1 && \text{[apply recurrence to first term]} \\
&\leq T(N/8) + 1 + 1 + 1 && \text{[apply recurrence to first term]} \\
&\vdots \\
&\leq T(N/N) + 1 + 1 + \ldots + 1 && \text{[stop applying, T(1) = 1]} \\
&= 1 + \lg N
\end{aligned}
$$

A N^2 log N algorithm for 3-SUM

- Algorithm.
  - Step 1: Sort the N numbers.
  - Step 2: For every pair of numbers a[i] och a[j], do a binary search för -(a[i] + a[j]).

- Analysis. Time complexity is $N^2 \log N$.
  - Steg 1: N^2 with"insertion sort".
  - Steg 2: $N^2 \log N$ with binary search.

input

30 -40 -20 -10 40 0 10 5

sort

-40 -20 -10 0 5 10 30 40

Binary search

(-40, -20) 60
(-40, -10) 50
(-40, 0) 40
(-40, 5) 35
(-40, 10) 30
⋮ ⋮
(-40, 40) 0
⋮ ⋮
(-10, 0) 10
⋮ ⋮
(-20, 10) ~~30~~
⋮ ⋮
(10, 30) ~~-40~~
(10, 40) -50
(30, 40) -70

Only search if a[i] < a[j] < a[k] to avoid doing it twice

Comparing programs

- Conclusion. The sorting-based $N^2 \log N$ algorithm for 3-SUM is noticeably faster in practice compared to brute-force N^3 algorithm.

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

Rule of thumb. Better order of growth=> faster in practice

Analysis av algorithms

- Introduction
- Observations
- Mathematical models
- Order-of-growth classifications
- Theory of algorithms
- Memory complexity

Types of analysis

- Best case. Lower bound on cost.
  - Easiest input – fastest possible.

- Worst case. Upper bound on cost.
  - Most difficult input – slowest possible
  - Gives a guarantee for all possible inputs.

- Average (expected) case. Expected case for "random" input.
  - Need a model for what random means.
  -

T.ex 1. Array accesses for 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. Comparisons with binary search.

Best: ~ 1

Average: $\sim \lg N$

Worst: $\sim \lg N$

Types of analysis

- Best case. Lower bound on cost
- Worst case. Upper bound on cost
- Average case. Expected cost

- What happens if given input does not match are input model?
  - Vi need to understand the characteristics of the input.

- Method 1: design for worst case.
- Method 2: randomize and/or trust in probabilistical guarantees.

Theory of algorithms

- Goal.
  - Determine the difficulty of the problem
  - Develop optimal algorithms

- Approach.
  - Suppress details in the analysis: look for result within a constant factor
  - Focus on worst case (eliminating input variability)

- Upper bound. Performance guarantee for all input.
- Lower bound. Prove that there exists no faster algorithm (within constant factor).
- Optimal algorithm. Lower bound == upper bound (within a constant factor).

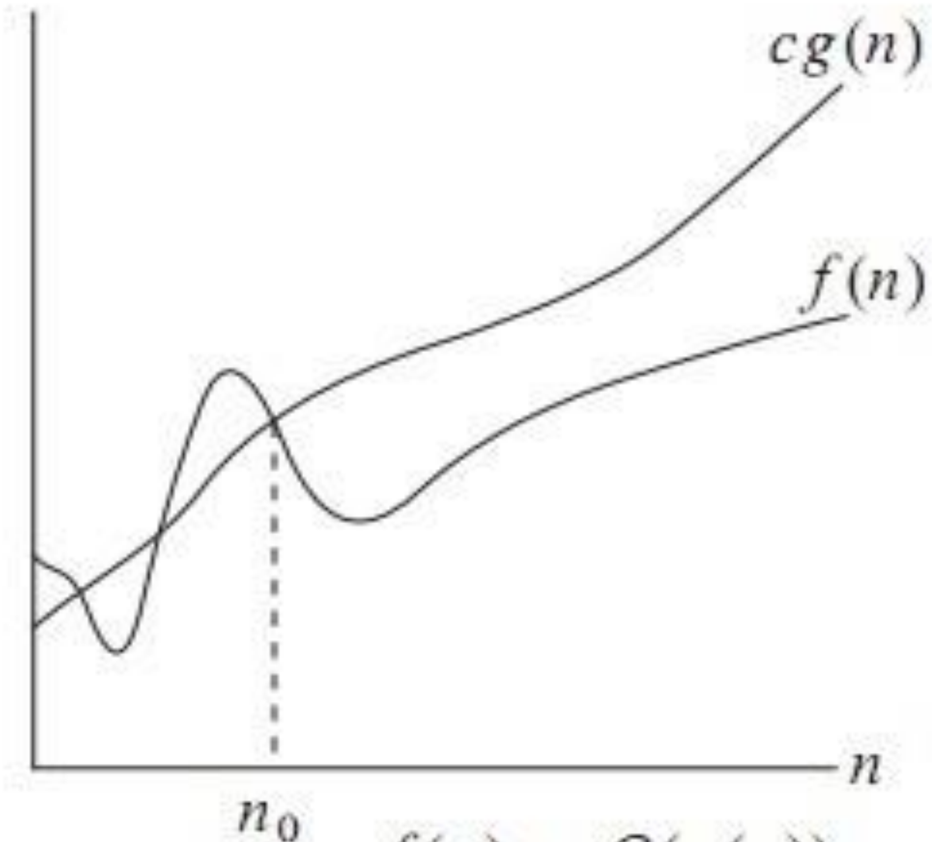
Notations used in the theory of algorithms

notation	describes	example	Shorthand for	used
Big Theta	Asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ $\vdots$	Classify algorithms
Big Oh (Stora ordo)	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ $\vdots$	Develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ $\vdots$	Develop lower bounds

Big-Oh (O)

- Big-Oh determine the upper bound.

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

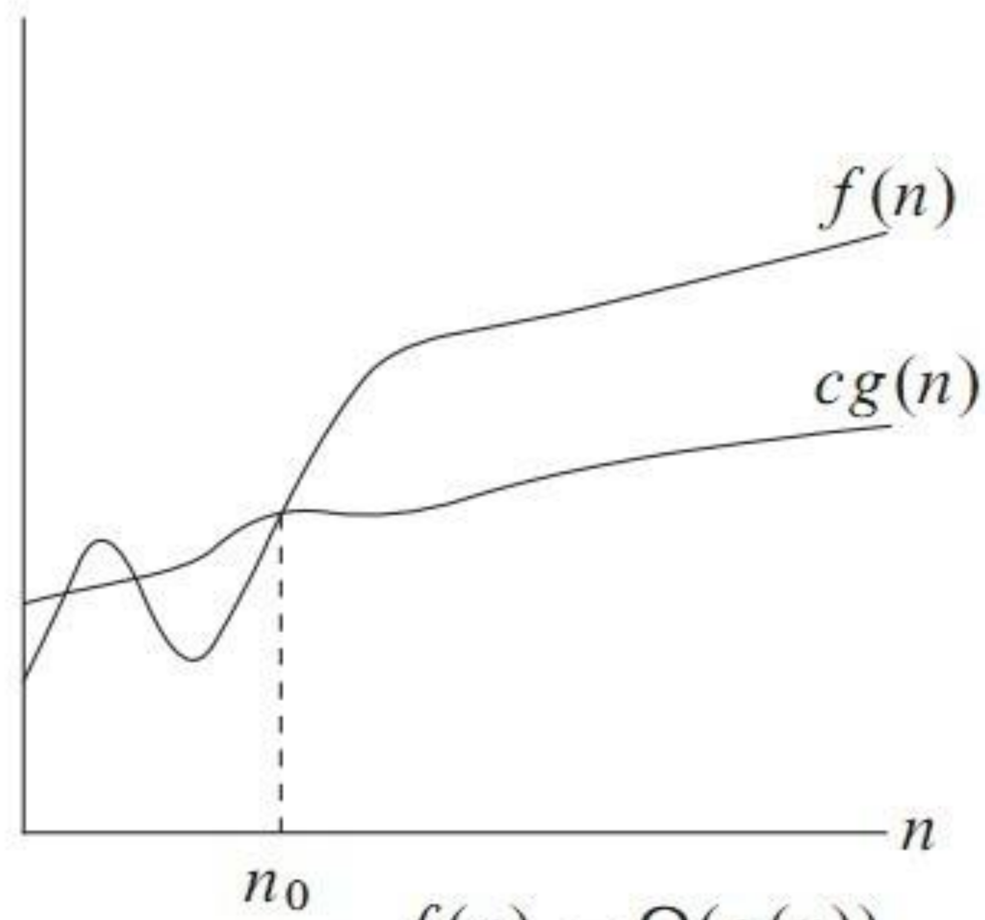


$f(n) = O(g(n))$

Big-Omega Notation

- Big-Omega determines the lower bound.

$$\Omega\big(g(n)\big) = \{f(n) \exists\, c > 0 \; och \; n_0 > 0 \;\; där \; 0 \le cg(n) \le f(n) \; \forall \, n \ge n_0\}$$

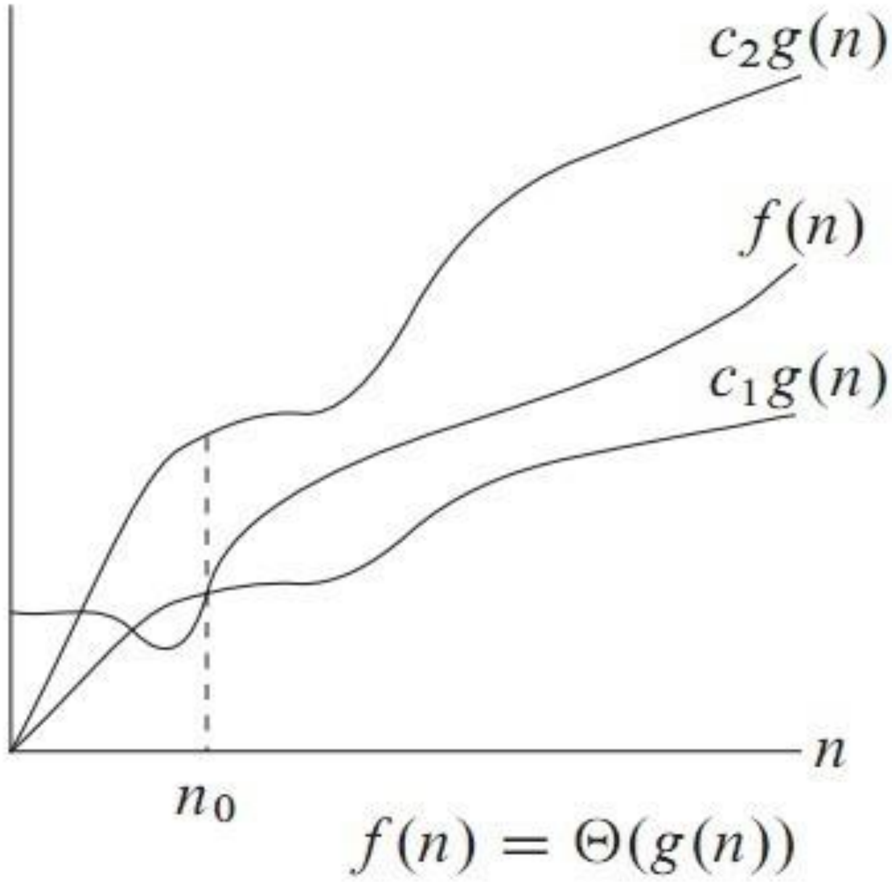


$f(n) = \Omega(g(n))$

Theta Notation

- Theta notation provides both lower and upper bound.

$$\Theta\big(g(n)\big) = \{f(n)\exists c_1, c_2, n_0 > 0 \; d\ddot{a}r \; 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall \, n \geq n_0\}$$



Class P

- Class P consists of problems that can be solved in polynomial time.
- Polynomial timen $O(n^k)$ for a constant *k*, where n is input size.
- Note both $\sim N^2$ and $\sim N^{3000}$ belong to P !!

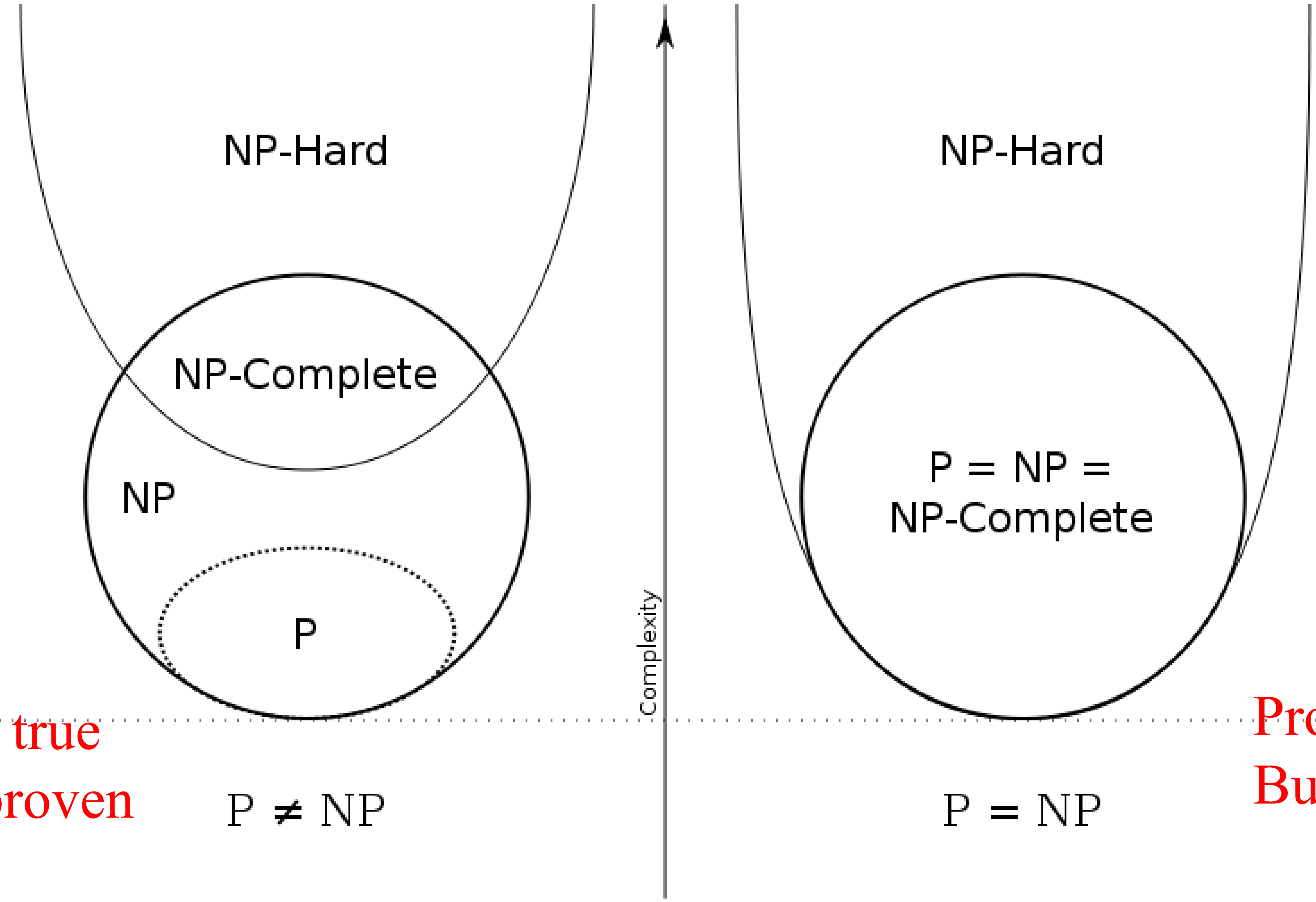
NP problems

- Time to solve NP-complete problems is proportional to number of compinations (2^n)
  - NP != Non-polynomial time
  - NP = *Non-deterministic polynomial time*

- NP is the class of problmes that can be solved in polynomial tidme *with perfect guesses*.
  - NP are those problems that can be solved in polynomical time by a non-deterministic Turingmashine.

- A problem is NP-complete when it I both in NP and the NP-hard classes.
  - NP-hard problem is as difficult as the most difficult problem in NP

P=NP?



Probably true
But not proven

Probably wrong
But not proven

[source: wikipedia]

Theory of algorithms - example

- Goal.
  - Establish “difficulty” of a problem and develop “optimal” algorithmsr.
  - For example. 1-SUM.

- Upper bound. A specific algorithm.
  - Ex. Brute-force algorithm for 1-SUM: Look at every array entry..
  - Running time of the optimal algorithm for 1-SUM is $O(N)$.

- Lower bound. Proof that no algorithm can do better
  - Ex. Have to examine all N entries.
  - Running time of the optimal algorithm for 1-SUM is $\Omega(N)$.

- Optimal algorithm? .
  - Lower bound equal upper bound (within constant factor).
  - Running time of the optimal algorithm for 1-SUM is $O(N)$.

Theory of algorithms – example 2

- Goal.
  - Establish "difficulty" of a problem and develop "optimal" algorithmsr.
  - For example. 3-SUM.

- Upper bound. A specific algorithm.
  - Ex. Brute-force algorithm for 3-SUM: $O(N^3)$
  - Better algorithm with bindary search: $O(N^2 \log N)$

- Lower bound. Proof that no algorithm can do better
  - Ex. Have to examine all N entries.
  - Running time of the optimal algorithm for 1-SUM is $\Omega(N)$.

- Optimal algorithm?
  - Open problem?
  - Between N log N and $N^2 \log N$
  - Can lower bound be raised can upper bound (better algorithm be lowered)?

Algorithm design

- Start
  - Develop and algorithm
  - Prove a lower bound

- Is there a difference?
  - Lower the upper bound (discover new algorithm).
  - Raise the lower bound (more difficult).

- The golden age of algorithm design.
  - 1970s.
  - Time of raise lower bounds and lowering upper bounds
  - Many optimal algorithms discovered

- To consider.
  - Is it too pessimistic to focus on *worst case*?
  - We need more precision “within a constant factor” to predict performance.

ID1020: We focus on the Tilde-notation

- Common error. Interpret big-Oh as an approximation model .
- ID1020. Focus on approximation models : use Tilde-notation

notation	beskriver	exempel	kort för	används för att
Tilde	leading term	$\sim 10N^2$	$10N^2$ $10N^2 + 22N\log N$ $10N^2 + 2N + 37$	Approximative models
Big Theta	asymptotisk tillväxtsordning	$\Theta(N^2)$	$\frac{1}{2}N^2$ $10N^2$ $5N^2 + 22N\log N + 3N$	klassificiera algoritms
Big Oh	$\Theta(N^2)$ och mindre	$O(N^2)$	$10N^2$ $100N$ $22N\log N + 3N$ ⋮	hitta övre gräns
Big Omega	$\Theta(N^2)$ och större	$\Omega(N^2)$	$\frac{1}{2}N^2$ N^5 $N^3 + 22N\log N + 3N$ ⋮	hitta lägre gräns

In practice other factors may be important

- Large constants
  - Lower order terms can be important
- Non-dominant inner loops
- Hardware
  - Caching, etc.
- System disturbances
  - Garbage collection
  - Other processes
- Stong dependence on input characteristics

Memory complexity

Fundamentals

NIST Datorvetenskper

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

- 64-bits computer. Vi will assume 64-bit computer with 8-byte pointers
  - To addrss more memory
  - Pointers use more memory.

Memory sizes

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

Primitive types

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

One dimensional arrays

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

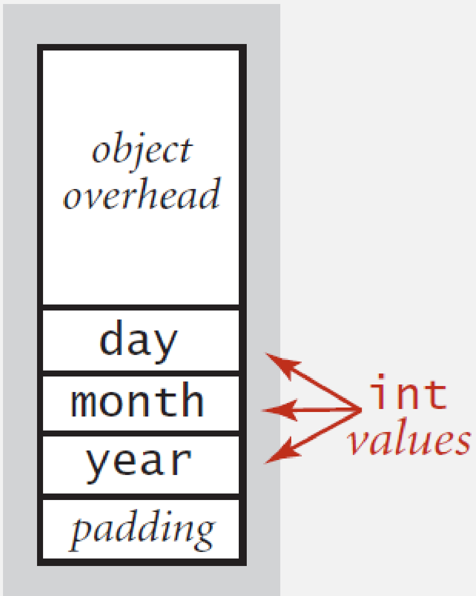
Two dimensional arrays

Memory sizes

- Objekt overhead. 16 bytes.
- Reference. 8 bytes.
- Padding. Every object must use a multiple of 8 bytes.

- Es. A Date object uses 32 bytes memory.

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



16 bytes (object overhead)

4 bytes (int)

4 bytes (int)

4 bytes (int)

4 bytes (padding)

32 bytes

Example

- How much memory does WeightedQuickUnionUF use a function of N ?
- Use tilde-notation.

```
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

Memory usage: $8N + 88 \sim 8N$ bytes.

Conclusion

- Empirical analysis.
  - Run a program as an experiment.
  - Assume a ”power law relation” develop hypothesis
  - Model enables prediction of programs performance

- Mathematical analysis.
  - Analyze by counting frequency of operations.
  - Use tilde-notation to simplify analysis.
  - Model enables us to predict behavior.

- Scientific method.
  - The mathematical model is independent of computer and system.
  - Empirical analysis is necessary to predict interesting properties precisely (e.g. performance).