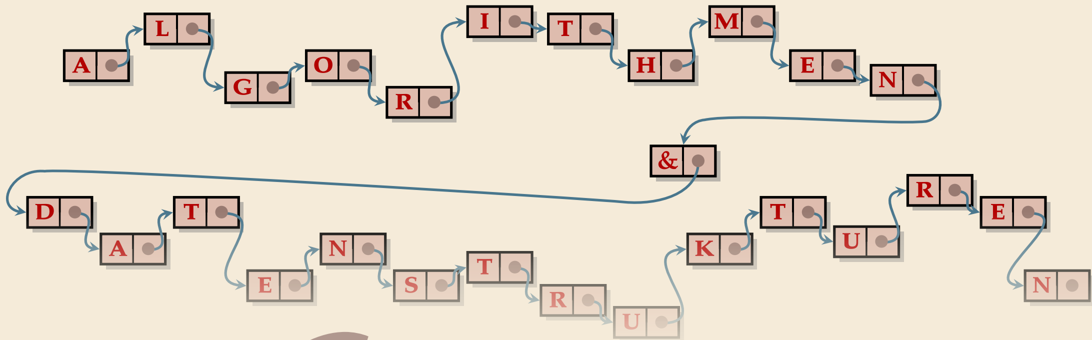


6

Algorithm Science

Algorithmen & Datenstrukturen · Sommersemester 2026

Prof. Dr. Sebastian Wild

CS210 (Sommersemester 2026)
Philipps-Universität Marburg

version `2026-05-11 21:46 H`

Outline

6 Algorithm Science

6.1 Algorithm Science

6.2 Laufzeitexperimente

6.3 Mathematische Analyse von Algorithmen

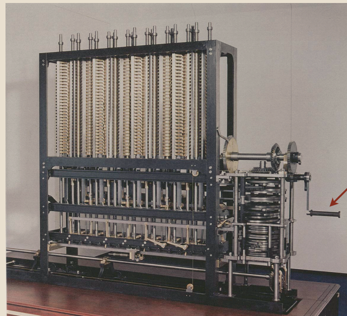
6.4 Typische Wachstumsraten

6.5 Speicherbedarf

6.1 Algorithm Science

Running time

" 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)*



how many times do you have to turn the crank?

Analytic Engine

Cast of characters

Programmer needs to develop a working solution.

Student might play any or all of these roles someday.

Client wants to solve problem efficiently.

Theoretician wants to understand.

Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

this course (COS 226)

theory of algorithms (COS 423)

Primary practical reason: avoid performance bugs.

client gets poor performance because programmer did not understand performance characteristics

The challenge

Q. Will my program be able to solve a large practical input?

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

Scientific method applied to analysis of algorithms

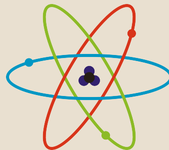
A framework for predicting performance and comparing algorithms.

Scientific method.

- Observe some feature of the natural world.
- Hypothesize a model that is consistent with the observations.
- Predict events using the hypothesis.
- Verify the predictions by making further observations.
- Validate by repeating until the hypothesis and observations agree.

Principles.

- Experiments must be reproducible.
- Hypotheses must be falsifiable.



Feature of the natural world. Computer itself.

6.2 Laufzeitexperimente

Ein (erstes) Wiedersehen: 3-SUM

3-SUM-Problem

- **Gegeben:** $A[0..n)$ with $A[i] \in \mathbb{Z}$ for $0 \le i < n$.
- **Ziel:** Anzahl c von Tripeln aus A mit Summe 0

Parameter:

- Eingabegröße: # Zahlen n

Mit der einfachsten Lösung hatten wir auch schon vorgearbeitet:

1. **Algorithmische Idee**
   Alle Tripel durchprobieren
2. </> **Pseudocode**
   (3 geschachtelte for-Schleifen)
   siehe nächste Folie
3. **Korrektheitsbeweis**
   direkt nach Definition von c
4. **Analyse**
   # Ausführungen von Zeilen 5–6
   $$\frac{(n-2)(n-1)n}{6} \sim \frac{1}{6}n^3 = \Theta(n^3)$$

Brute-Force 3-SUM Code

Pseudocode

1 $c = 0$
2 **for** $i = 0, \ldots, n-3$
3   **for** $j = i+1, \ldots, n-2$
4   **for** $k = j+1, \ldots, n-1$
5             $t = A[i] + A[j] + A[k]$
6   **if** $t == 0$ **then** $c := c + 1$
7   **end for**
8   **end for**
9 **end for**
10 **return** c

Java-Implementierung

```
1  import edu.princeton.cs.algs4.*;
2
3  public class ThreeSum {
4      public static int count(int[] a) {
5          int n = a.length;
6          int count = 0;
7          for (int i = 0; i < n-2; i++)
8              for (int j = i+1; j < n-1; j++)
9                  for (int k = j+1; k < n; k++)
10                     if (a[i] + a[j] + a[k] == 0)
11                         count++;
12         return count;
13     }
14
15     public static void main(String[] args) {
16         In in = new In(args[0]);
17         int[] a = in.readAllInts();
18         StdOut.println(count(a));
19     }
20 }
```

Measuring the running time

Q. How to time a program?

A. Manual.

```
% java ThreeSum 1Kints.txt
```

tick tick tick

```
70
% java ThreeSum 2Kints.txt
```

tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick

```
528
% java ThreeSum 4Kints.txt
```

tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick

```
4039
```

Measuring the running time

Q. How to time a program?

A. Automatic.

public class Stopwatch	(part of stdlib.jar)
Stopwatch()	*create a new stopwatch*
double elapsedTime()	*time since creation (in seconds)*

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

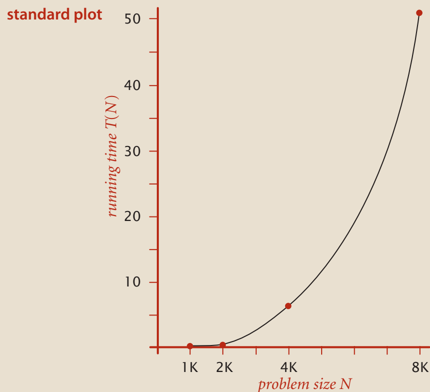
Empirical analysis

Run the program for various input sizes and measure running time.

N	time (seconds) †
250	0.0
500	0.0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

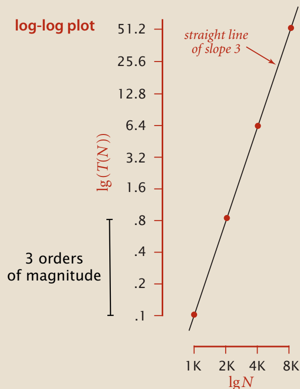
Data analysis

Standard plot. Plot running time $T(N)$ vs. input size N.



Data analysis

Log-log plot. Plot running time $T(N)$ vs. input size N using log-log scale.



$$\lg(T(N)) = b \lg N + c$$
$$b = 2.999$$
$$c = -33.2103$$

$$T(N) = a N^b, \text{ where } a = 2^c$$

power law

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

slope

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

Prediction and validation

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

"order of growth" of running time is about N^3 [stay tuned]

Predictions.

- 51.0 seconds for $N = 8{,}000$.
- 408.1 seconds for $N = 16{,}000$.

Observations.

N	time (seconds) †
8,000	51.1
8,000	51.0
8,000	51.1
16,000	410.8

validates hypothesis!

Doubling hypothesis

Doubling hypothesis. Quick way to estimate b in a power-law relationship.

Run program, doubling the size of the input.

N	time (seconds) †	ratio	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

$$\frac{T(2N)}{T(N)} = \frac{a(2N)^b}{aN^b} = 2^b$$

lg (6.4 / 0.8) = 3.0

seems to converge to a constant b ≈ 3

Hypothesis. Running time is about $a N^b$ with b = lg ratio.

Caveat. Cannot identify logarithmic factors with doubling hypothesis.

Doubling hypothesis

Doubling hypothesis. Quick way to estimate b in a power-law relationship.

Q. How to estimate a (assuming we know b) ?

A. Run the program (for a sufficient large value of N) and solve for a.

N	time (seconds) †
8,000	51.1
8,000	51.0
8,000	51.1

$51.1 = a \times 8000^3$

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

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

almost identical hypothesis
to one obtained via linear regression

Experimental algorithmics

System independent effects.

- Algorithm.
- Input data.

determines exponent in power law

System dependent effects.

- Hardware: CPU, memory, cache, …
- Software: compiler, interpreter, garbage collector, …
- System: operating system, network, other apps, …

determines constant in power law

Bad news. Difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.

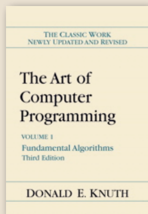
e.g., can run huge number of experiments

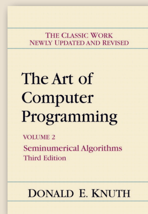
6.3 Mathematische Analyse von Algorithmen

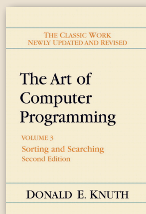
Mathematical models for running time

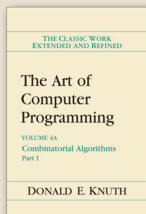
Total running time: sum of cost × frequency for all operations.

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.









Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of basic operations

Challenge. How to estimate constants.

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)	91.3
arctangent	Math.atan2(y, x)	129.0
...	...	...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Cost of basic operations

Observation. Most primitive operations take constant time.

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]`	$c_6\ N$
2D array allocation	`new int[N][N]`	$c_7\ N^2$

Caveat. Non-primitive operations often take more than constant time.

novice mistake: abusive string concatenation

Beispiel: Knuthsche Analyse

Wir betrachten folgenden Code für das (wenig spannende) ***1-SUM-Problem***

```
1 public static int oneSum(int[] a) {
2     int count = 0, N = a.length;
3     for (int i = 0; i < N; ++i)
4         if (a[i] == 0)
5             count++;
6     return count;
7 }
```

Wie viele Instruktionen als Funktion in N?

Zeile Z	Frequenz $f_Z(N)$
2	1
3	$N+1$
4	N
5	$[0..N]$
6	1

Analyse aller Operationen

Operation	Frequenz	Kosten
Variablendeklaration	2	c_1
Zuweisung	2	c_2
Vergleich $\le$	$N+1$	c_3
Vergleich $==$	N	c_4
Array-Zugriff	N	c_5
Inkrement	$[N..2N]$	c_6

$\rightsquigarrow$ Gesamtkosten

$$T(N) \le (c_3 + c_4 + c_5 + 2c_6)N + 2(c_1 + c_2) + c_3$$

alle Instruktionen zählen i.d.R. zu aufwändig

- $\rightsquigarrow$ Wählen *abstraktes Kostenmaß*: **# Arrayzugriffe**
- $\rightsquigarrow$ Kosten von 1-SUM: $\sim n$

Simplifying the calculations

> “ *It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings.* ” — *Alan Turing*

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

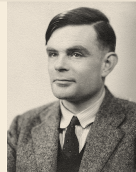
By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known ‘Gauss elimination process’, it is found that the errors are normally quite moderate: no exponential build-up need occur.



Von Code zu Analyse

Achtung: Im Allgemeinen Ausführungsfrequenz von Codestelle vielleicht unmöglich zu analysieren.

Oft geht es aber, z. B. nach einer der folgenden Regeln

<table>
<tr><th>Code</th><th>Laufzeit T</th></tr>
<tr><td><code>x = 0</code>, <code>a[i]</code>
<code>x + y</code>, <code>x * y</code>, <code>x++</code> etc.</td><td>1 (falls teil des abstrakte Kostenmaßes, sonst 0)</td></tr>
<tr><td><pre>for (int j = a; j <= b; ++j) {
    B(j) // weiterer Code
}</pre></td><td>$\sum_{j=a}^{b} T_B(j)$ für $T_B(j)$ Laufzeit des Schleifenrumpfs mit Wert j</td></tr>
<tr><td><pre>if (...) { A } else { B }</pre></td><td>$\leq \max\{T_A, T_B\}$</td></tr>
<tr><td><pre>while (n > c) {
    A(n) // weiterer Code
    n = n/2;
    B(n) // weiterer Code
}</pre></td><td rowspan="2">Für allgemeine While-Schleifen und rekursive Methoden
Rekursionsgleichung
$T(n) = T(n/2) + T_A(n) + T_B(n/2) \quad (n > c)$
$T(n) = 1 \quad (n \leq c)$
$\rightsquigarrow$ Master-Theorem</td></tr>
<tr><td><pre>void m(n) {
    if (n <= c) return;
    A(n) // weiterer Code
    m(n/2); // Rekursion
    B(n/2) // weiterer Code
}</pre></td></tr>
</table>

Example: 2-SUM

Q. Approximately how many array accesses as a function of 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++;
```

"inner loop"

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

A. $\sim N^2$ array accesses.

Bottom line. Use cost model and tilde notation to simplify counts.

Example: 3-SUM

Q. Approximately how many array accesses as a function of input size N?

```
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++;
```

"inner loop"

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \sim \frac{1}{6}N^3$$

A. $\sim \frac{1}{2} N^3$ array accesses.

Bottom line. Use cost model and tilde notation to simplify counts.

6.4 Typische Wachstumsraten

Common order-of-growth classifications

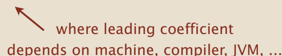
Definition. If $f(N) \sim c\, g(N)$ for some constant $c > 0$, then the order of growth of $f(N)$ is $g(N)$.

- Ignores leading coefficient.
- Ignores lower-order terms.

Ex. The order of growth of the running time of this code is N^3.

```
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++;
```

Typical usage. With running times.

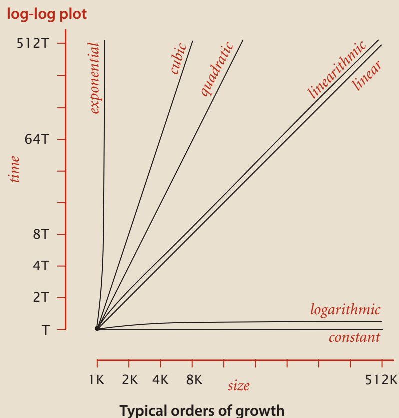


Common order-of-growth classifications

Good news. The set of functions

$1,\ \log N,\ N,\ N \log N,\ N^2,\ N^3$, and 2^N

suffices to describe the order of growth of most common algorithms.



Typical orders of growth

Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	**constant**	`a = b + c;`	statement	add two numbers	1
$\log N$	**logarithmic**	`while (N > 1)` `{ N = N / 2; ... }`	divide in half	binary search	~ 1
N	**linear**	`for (int i = 0; i < N; i++)` `{ ... }`	loop	find the maximum	2
$N \log N$	**linearithmic**	[see mergesort lecture]	divide and conquer	mergesort	~ 2

N^2	**quadratic**	`for (int i = 0; i < N; i++)`
		`for (int j = 0; j < N; j++)`
		`{ ... }`
			double loop	check all pairs	4

N^3	**cubic**	`for (int i = 0; i < N; i++)`
		`for (int j = 0; j < N; j++)`
		`for (int k = 0; k < N; k++)`
		`{ ... }`
			triple loop	check all triples	8

2^N	**exponential**	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

6.5 Speicherbedarf

Basics

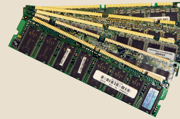
Bit. 0 or 1.

Byte. 8 bits.

Megabyte (MB). 1 million or 2^{20} bytes.

Gigabyte (GB). 1 billion or 2^{30} bytes.

NIST ↓ (1 million / 1 billion)　　most computer scientists ↓ (2^{20} / 2^{30})



64-bit machine. We assume a 64-bit machine with 8-byte pointers.

- Can address more memory.
- Pointers use more space.

some JVMs "compress" ordinary object pointers to 4 bytes to avoid this cost

Typical memory usage for primitive types and arrays

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

primitive types

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

one-dimensional arrays

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

two-dimensional arrays

Typical memory usage for objects in Java

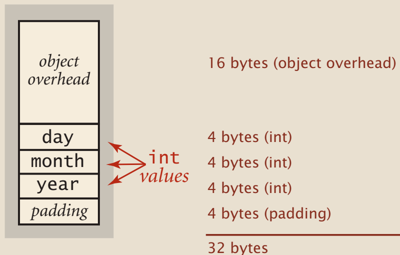
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Each object uses a multiple of 8 bytes.

Ex 1. A `Date` object uses 32 bytes of 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)

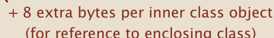
4 bytes (padding)

32 bytes

Typical memory usage summary

Total memory usage for a data type value:

- Primitive type: 4 bytes for `int`, 8 bytes for `double`, ...
- Object reference: 8 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable.
- Padding: round up to multiple of 8 bytes.



Shallow memory usage: Don't count referenced objects.

Deep memory usage: If array entry or instance variable is a reference, count memory (recursively) for referenced object.

Example

Q. How much memory does `WeightedQuickUnionUF` use as a function of N? Use tilde notation to simplify your answer.

```
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 (object overhead)

8 + (4N + 24) bytes each (reference + `int[]` array)

4 bytes (`int`)

4 bytes (padding)

8N + 88 bytes

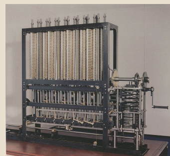
A. $8N + 88 \sim 8N$ bytes.

Turning the crank: summary

Empirical analysis.

- Execute program to perform experiments.
- Assume power law and formulate a hypothesis for running time.
- Model enables us to make predictions.

Mathematical analysis.



- Analyze algorithm to count frequency of operations.
- Use tilde notation to simplify analysis.
- Model enables us to explain behavior.

Scientific method.

- Mathematical model is independent of a particular system; applies to machines not yet built.
- Empirical analysis is necessary to validate mathematical models and to make predictions.