

Programmieren / Algorithmen & Datenstrukturen

Einfaches Suchen und Sortieren



Prof. Dr. Skroch

Universitatea
BABEȘ-BOLYAI

Einfaches Suchen und Sortieren

Inhalt.

- ▶ Zur Algorithmenanalyse
- ▶ Sequentielle Suche
- ▶ Binäre Suche und Interpolationssuche
- ▶ Sortieren durch Auswahl, Sortieren durch Einfügung
- ▶ Zu den Eingangsdaten

Algorithmen

Warum befassen wir uns überhaupt damit?

- ▶ Menschen beschäftigen sich seit deutlich mehr als 2000 Jahren mit Algorithmen (spätestens seit Euklid, ca. 300 v. Chr.)
- ▶ Algorithmen haben heute in vielen Lebensbereichen fundamentale Bedeutung (z.B.: Internet / Kommunikation, Biologie / Genetik, Physik, Industrieproduktion, Verkehr).
- ▶ Algorithmen können helfen, praktische Probleme zu lösen, denen man mit anderen Herangehensweisen nicht beikommen würde.
- ▶ In Forschung und Technik ersetzen algorithmische Modelle (Programme) zunehmend die traditionellen mathematischen Modelle (Formeln).

The image shows a side-by-side comparison of scientific representations. On the left, under the heading '20th century science (formula based)', are three mathematical formulas: $E = mc^2$, $F = ma$ and $F = \frac{Gm_1m_2}{r^2}$, and the Schrödinger equation $\left[-\frac{\hbar^2}{2m} \nabla^2 + V(r) \right] \Psi(r) = E \Psi(r)$. On the right, under the heading '21st century science (algorithm based)', is a snippet of C++ code for a physics simulation:

```
for (double t = 0.0; true; t = t + dt)
for (int i = 0; i < N; i++)
{
    bodies[i].resetForce();
    for (int j = 0; j < N; j++)
        if (i != j)
            bodies[i].addForce(bodies[j]);
}
```

- ▶ Viele Menschen empfinden die Beschäftigung mit Algorithmen als eine bereichernde Tätigkeit in sich selbst, die ihnen Spaß macht.

Algorithmenanalyse

Praktische Bedeutung und Grundprinzipien.

- ▶ Die Laufzeit von Programmen kann durch geeignete Entwurfstechniken um einen Faktor im Bereich von Tausenden oder Millionen (!) verbessert werden.
- ▶ Ziel muss also sein, Algorithmen gründlich zu verstehen.
 - Um ihre wesentlichen Leistungsdaten zu kennen.
 - Um sie untereinander vergleichen zu können.
 - Um sie effizient auf praktische Probleme anzuwenden.
- ▶ Man kann zwei grundsätzliche Herangehensweisen unterscheiden.
 - *Implementierung und empirische Untersuchung.*
 - *Mathematisch-theoretische Analyse.*
 - Ergänzen sich gegenseitig.

Algorithmenanalyse

Praktische Bedeutung und Grundprinzipien.

► Empirische Untersuchung

- Zur Lösung eines real existierenden Problems wird ein Algorithmus implementiert und eingesetzt.
- Laufzeiten und/oder Häufigkeiten für die Ausführung der einzelnen Anweisungen werden gemessen.
- Schwierig: eine möglichst gute (i.Allg. effiziente) Implementierung ist erforderlich.
 - "Handwerkliche" Herausforderung.
- Schwierig: die Experimente selbst können aufwändig werden.
 - Das ist aber bei kontrollierten Experimenten fast immer schwierig, in der Informatik sind die Aufwände oft noch vergleichsweise überschaubar.

► Mathematische Analyse

- Ein formales, mathematisches Modell (des Problems) wird entwickelt.
- Der Algorithmus wird im Rahmen des Modells theoretisch durchdacht.
- Schwierig: ein möglichst gut passendes Modell ist erforderlich.
 - Das ist aber bei der formalen Modellierung immer schwierig, in der Informatik ist ein üblicher Ansatz die Annahme von *zufälligen* Situationen.
- Schwierig: die mathematischen Hilfsmittel müssen existieren und korrekt eingesetzt werden.
 - Auch das ist eine übliche Schwierigkeit in vielen Disziplinen.

► *Wissenschaft* bedeutet: *beides* ist erforderlich.

Algorithmenanalyse

Implementierung und empirische Untersuchung:
das Prinzip.

- ▶ Das Prinzip ist simpel.
 - Mehrere Algorithmen lösen das selbe Problem.
 - Sie werden implementiert und unter ansonsten gleichen Bedingungen ausgeführt.
 - Die Laufzeiten werden gemessen.
- ▶ Bei sorgfältiger Implementierung und typischen Eingabedaten ergeben sich gemessene Leistungswerte, die
 - zum tieferen Verständnis des Algorithmus entscheidend beitragen können,
 - direkt auf die Effizienz des Algorithmus schließen lassen,
 - Vergleichsinformationen liefern,
 - mathematische Analysen bestätigen können.


Algorithmenanalyse

Implementierung und empirische Untersuchung:
einige Faustregeln.

► Das Leistungsverhalten mit *großen* Datenmengen testen.

- Beispiel, was stimmt mit dieser Quellcode-Zeile (in C geschrieben) nicht:

```
for( int i=0; i < strlen( s ); ++i ) { /* mache etwas mit s[i] */ }
```

 Liefert die Länge der
Zeichenkette s (C-Bibliotheksfunktion)

- Problem: der Quellcode war Teil eines Internet Log-Analyzers, d.h. der Quellcode wurde für *sehr* viele *sehr* lange Zeichenketten verwendet und hat das Programm stark verlangsamt.
- Das ist erst beim realen Einsatz aufgefallen, hätte aber durch vorherige Messung mit realistisch großen Mengen an Testdaten festgestellt werden können.

► Messen, wie viel Zeit die kritischen Testfälle benötigen.

- Welche sind das? Eine andere wichtige Frage...

► Treten Performance-Probleme auf, dann suchen wir die Ursachen zunächst in schlecht konzipierten und/oder ungeeigneten Algorithmen.

► "Don't optimize *yet*", d.h. Vorsicht vor übereilten "Optimierungen" des Quellcodes: Ursache vieler Fehler, fast immer unnötiger Zeitverlust.

Algorithmenanalyse

Implementierung und empirische Untersuchung:
die zwei vielleicht größten Fehler.

► Der vielleicht größte Fehler bei der Algorithmenwahl:

- Leistungseigenschaften werden ignoriert.
 - Programmierer geben sich mit langsamen Algorithmen zufrieden, um sich nicht mit den Schwierigkeiten bei der Analyse des Problems und beim Programmieren fortgeschrittener Algorithmen auseinandersetzen zu müssen.
 - Obwohl manchmal extreme Einsparungen mit nur wenigen Codezeilen erreicht werden können.

► Der vielleicht zweitgrößte Fehler bei der Algorithmenwahl:

- Leistungseigenschaften werden übermäßig betont.
 - Der insgesamt erforderliche Aufwand für die Implementierung eines fortgeschrittenen, komplizierten Algorithmus ist i.Allg. beträchtlich höher im Vergleich zur Entwicklung eines einfachen Algorithmus.
 - Unter Umständen lohnt sich der hohe zusätzliche Aufwand einfach nicht.

Algorithmenanalyse

Implementierung und empirische Untersuchung:
"handwerkliche" Herausforderungen.

- ▶ Nicht alle Performance-Probleme sind auf schlecht konzipierte und/oder ungeeignete Algorithmen zurückzuführen.
 - Tatsächlich stellt der überwiegende Teil vieler Quellcodes keine besonderen algorithmischen Herausforderungen,
 - sondern "handwerkliche".
 - Zu den entsprechenden nicht-algorithmischen bzw. handwerklichen Ursachen für Performance-Probleme gehören
 - vermeidbare (wiederholte) Berechnung von Daten,
 - vermeidbare (wiederholte) Überprüfung von Bedingungen,
 - vermeidbare (wiederholte) Zugriffe auf externe Datenquellen (Festplatte, Internet),
 - **speziell in den inneren (innersten) von verschachtelten Schleifen.**
 - Wir werden, wenn wir Quellcode für das Sortieren durch direkte Einfügung besprechen, noch näher auf einige dieser handwerklichen Aspekte eingehen.

Algorithmenanalyse

Implementierung und empirische Untersuchung:
Fragestellungen und Problembereiche.

- ▶ Externe Einflussfaktoren?
 - Das Wesen der Eingabedaten (tatsächliche, zufällige oder extreme).
 - Der Compiler (gemessen wird die Performance des ausführbaren *Maschinencodes*, nicht die des Quellcodes).
 - Die Systemumgebung (v.a. Hardware).
- ▶ Korrekte und vollständige Implementierung der zu untersuchenden Algorithmen?
 - Gerade die interessanten Algorithmen sind oft durchaus anspruchsvoll in der Implementierung.
 - Alle Algorithmen sollten aber aus Gründen der Vergleichbarkeit unvoreingenommen und gleich sorgfältig implementiert sein.
- ▶ Aufwand der Untersuchung und Messung?
 - Es ist immer auch eine Abwägung der Vor- und Nachteile (Kosten-Nutzen) erforderlich, bevor ein Programmteil einer Laufzeitmessung unterzogen wird.
 - Es ist nicht sinnvoll, stundenlang auf das Ende eines Programms zu warten, nur um damit festzustellen, dass es langsam läuft.

Algorithmenanalyse

Implementierung und empirische Untersuchung:
C++ Sprachmittel.

► Zeitmessungen in C++

- Verwenden Sie die im Header `chrono` verfügbare C++ Funktion `std::chrono::high_resolution_clock::now()`.
- `now()` ist dazu gedacht, kurze Zeitintervalle innerhalb einer einzelnen Ausführung eines Programms zu messen.
- `now()` liefert immer ein Ergebnis vom Typ `time_point` zurück (erzeugt keine Ausnahme).
- Zur Umrechnung in Millisekunden subtrahieren Sie zuerst zwei `time_point` Werte und verwenden dann `duration_cast<milliseconds>` zur Umwandlung des Ergebnisses.

► Laufzeitmessungen mit derart instrumentiertem Quellcode sollten Sie grundsätzlich mehrmals wiederholen und die Ergebnisse sollten weniger als ca. 10 Prozent voneinander abweichen.

► Denken Sie immer daran, wie *schnell* moderne Computer sind.

Algorithmenanalyse

Implementierung und empirische Untersuchung:
std::chrono.

- Quellcode-Schema mit C++ Sprachmitteln aus dem chrono Header.

```
using std::chrono::high_resolution_clock;
using std::chrono::milliseconds;
using std::chrono::duration_cast;

high_resolution_clock::time_point my_end {};
high_resolution_clock::time_point my_start {
    high_resolution_clock::now(); // static time_point now() noexcept
};

    // Performance-kritischer Teil

my_end = high_resolution_clock::now();

milliseconds ms { duration_cast<milliseconds>( my_end - my_start ) };

// Die Syntax duration_cast<>() benennt eine sog. Templatefunktion, die hier zur
// Umwandlung von unterschiedlichen Einheiten der Zeitmessung eingesetzt wird.
// Wir werden Templates noch genauer besprechen...

std::cout << ms.count() << " Millisekunden";
```

Übung

Empirische Untersuchung der Laufzeit:

wie lange dauert es, um bis 100 Millionen – und noch weiter – zu zählen?

- Bestimmen Sie für $n_1=100,1000,10000, \dots$ und $n_2=100,1000,10000, \dots$ die Zeiten, die die folgenden Programmzeilen in Ihrer Programmierumgebung benötigen (ggf. auftretenden Überlauf können Sie ignorieren) .

```
volatile int counter{0};
int innen{0};
for( int aussen{0}; aussen < n1; ++aussen )
    for( innen = 0; innen < n2; ++innen )
        ++counter;
```

- Finden Sie heraus, welche Funktionen zur Optimierung von Programmen der Compiler Ihrer IDE hat und stellen Sie fest, ob diese für das Programm wirksam sind.

Das Schlüsselwort `volatile`

- Die Optimierungsfunktionen moderner Compiler werden womöglich feststellen, dass `counter` zwar in der inneren Schleife inkrementiert aber ansonsten im Programm nicht verwendet wird. Womöglich wird der Maschinencode dann so generiert, dass `counter` nicht mehr existiert und die Laufzeit null ist...
- Mit `volatile` bestimmt der Programmierer, dass ein Objekt im Programm in einer Weise verwendet wird, die der Compiler nicht feststellen kann (z.B. über bestimmte Zeiger), wodurch verhindert wird, dass der Compiler `volatile` Objekte aus dem Maschinencode wegoptimiert.
- Lassen Sie den Quellcode auch ohne `volatile` und mit großen `n1` und `n2` laufen, und prüfen Sie, ob Sie "null" Laufzeit feststellen...

LEERE SEITE

Algorithmenanalyse

Mathematische Analyse.

- ▶ Die mathematisch-theoretische Analyse von Algorithmen
 - erfordert nur wenige mathematische Hilfsmittel,
 - ist dennoch meist nicht ganz einfach, und man muss sich konzentrieren,
 - kann sehr aufschlussreich sein und entscheidend zum Verständnis eines Algorithmus beitragen.
- ▶ Viele bekannte Algorithmen sind gründlich erforscht.
 - Es sind Formeln bekannt, aus denen sich die Laufzeit in praktischen Situationen vorhersagen lässt.
- ▶ Die Leistungseigenschaften vieler interessanter Algorithmen sind aber (noch) nicht bekannt, z.B. weil
 - (derzeit) ungelöste mathematische Fragen bei der Analyse eine Rolle spielen,
 - die Algorithmen so kompliziert sind, dass die Analyse zu aufwändig wird,
 - die Eingangsdaten sich nicht vernünftig charakterisieren lassen.

Algorithmenanalyse

Mathematische Analyse, erster Schritt: die Grundoperationen, aus denen der Algorithmus aufgebaut ist.

- ▶ Der erste Schritt der Analyse ist die Trennung des Algorithmus von seiner Implementierung.
 - D.h. die Identifikation der abstrakten Maschinen-Operationen, auf denen der Algorithmus wesentlich basiert, egal in welcher Quellcodesprache und Umgebung er implementiert wird.
 - D.h.: es liegt ein **Maschinenmodell** zugrunde.
 - Was ist überhaupt maschinell ("mechanisch") berechenbar, was nicht?
Ausblick: Church-Turing These, theor. Informatik.
- ▶ Obwohl an einem Algorithmus viele solche Operationen beteiligt sein können, hängt seine Leistung prinzipiell nur von wenigen Größen ab.
 - Die wichtigsten dieser Größen lassen sich oft leicht identifizieren.
 - Beispiele folgen gleich.

Algorithmenanalyse

Mathematische Analyse, zweiter Schritt: das Wesen der vom Algorithmus zu verarbeitenden Daten.

- ▶ Der zweite Schritt ist eine Vorstellung von den Daten, mit denen der Algorithmus arbeiten soll.
 - Zur Untersuchung der *durchschnittlichen* Leistung des Programms werden i.Allg. zufällige Daten verwendet.
 - *Vorhersagen* zur voraussichtlichen Laufzeit werden normalerweise unter der Annahme solcher zufälligen Daten getroffen.
 - Für manche Algorithmen unkompliziert, führt dann zu nützlichen analytischen Ergebnissen.
 - Für viele Algorithmen sind *zufällige Daten* aber nicht einfach zu definieren, der sog. Durchschnittsfall muss dann nicht repräsentativ sein.
 - Mit extremen Eingangsdaten lassen sich der *günstigste Fall* und der *ungünstigste Fall* untersuchen.
 - *Garantien* zur maximalen Laufzeit können nur unter der Annahme des ungünstigsten Falls gegeben werden.
 - Möglicherweise abstruse Konstruktionen von Daten, die so in der Praxis nicht vorkommen.
 - Können aber nützliche Erkenntnisse zum Verständnis liefern.

Algorithmenanalyse

Mathematische Analyse: die Größe der zu verarbeitenden Datenmenge.

- ▶ Die meisten Algorithmen haben einen *primären Parameter* N , der die Laufzeit entscheidend beeinflusst.
 - Dieser Parameter ist sehr oft direkt proportional zur Größe bzw. Menge der zu verarbeitenden Daten.
 - N kann z.B. die Anzahl der Zeichen in einem Text beschreiben, oder die Anzahl von Objekten einer Klasse, oder die Anzahl der Elemente in einem `vector`.
 - Ziel ist, den Ressourcenbedarf des Algorithmus (i.Allg. die Laufzeit) in Form von diesem einen Parameter N auszudrücken.
 - Durch möglichst einfache Formeln.
 - Besonders für den Fall, dass N immer größer wird.
 - Bemerkung: gibt es mehr als einen Parameter, dann werden die weiteren Parameter in der Analyse oft auf nur einen Parameter reduziert.
 - Indem etwa ein Parameter als Funktion des anderen ausgedrückt wird, oder jeweils nur ein Parameter betrachtet wird und die anderen konstant bleiben.

Algorithmenanalyse

Mathematische Analyse: erstes Beispiel.

► Beispiel

```
void g( /*...*/ ) { // Annahme: N == v.size()
    int counter{};
    for( int i{}; i < N; ++i )
        if( v[i] == 0 ) ++counter;
}
```

► Wie viele Operationen werden in `g()` abhängig von N ausgeführt? (Ohne die Aufrufparameter.)

▪ Definitionen	2 (nicht von N abhängig)
▪ Initialisierungen	2 (nicht von N abhängig)
▪ Vergleiche <code>==</code>	N
▪ Vergleiche <code><</code>	$N+1$
▪ <code>vector</code> Zugriffe	N
▪ Inkremente <code>++i</code>	N
▪ Inkremente <code>++counter</code>	zwischen 0 und N

► *Beobachtung:* die Analyse hängt nicht von der eingesetzten Quellsprache ab, sondern vom zugrunde liegenden Maschinenmodell.

Algorithmenanalyse

Mathematische Analyse: zweites Beispiel.

► Beispiel

```
void g( /*...*/ ) { // Annahme: N == v.size()
    int counter{};
    for( int i{}; i < N; ++i )
        for( int j{i+1}; j < N; ++j )
            if( v[i] == v[j] ) ++counter;
}
```

► Wie viele Operationen werden in `g()` abhängig von `N` ausgeführt? (Ohne die Aufrufparameter.)

▪ Definitionen	$N+2$
▪ Initialisierungen	$N+2$
▪ Vergleiche <code>==</code>	$\frac{1}{2} N (N-1) = 0+1+2+\dots+(N-1)$
▪ Vergleiche <code><</code>	$\frac{1}{2} (N+1) (N+2) = 0+1+2+\dots+(N+1)$
▪ <code>vector</code> Zugriffe	$N (N-1)$ zweimal pro Vergleich <code>==</code>
▪ Inkremente <code>++i</code>	N
▪ Inkremente <code>++j</code>	$\frac{1}{2} N (N-1)$ einmal pro Vergleich <code>==</code>
▪ Inkremente <code>++counter</code>	zwischen 0 und $\frac{1}{2} N (N-1)$

Die Ermittlung solcher
Terme ist oft mühsam
und fehlerträchtig...

Algorithmenanalyse

Mathematische Analyse: zweites Beispiel.

► Beispiel

```
void g( /*...*/ ) {  
    int counter {};  
    for( int i {}; i < N; ++i )  
        for( int j {i+1}; j < N; ++j )  
            if( v[i] == v[j] ) ++counter;  
}
```

Definitionen $\underline{N+2}$

Initialisierungen $\underline{N+2}$

Algorithmenanalyse

Mathematische Analyse: zweites Beispiel.

► Beispiel

```
void g( /*...*/ ) {  
    int counter {};  
    for( int i {}; i < N; ++i )  
        for( int j {i+1}; j < N; ++j )  
            if( v[i] == v[j] ) ++counter;  
}
```

<i>N</i>	==
0	0
1	0
2	1
3	2+1
4	3+2+1
5	4+3+2+1
...	$\underbrace{\hspace{1.5cm}}$ $= \frac{1}{2} N (N-1)$

Zur Erläuterung:

$$\begin{aligned} 1+2+\dots+10 &= (1+10) + (2+9) + (3+8) + (4+7) + (5+6) = \\ &= 11 + 11 + 11 + 11 + 11 \\ &= 5 \cdot 11 = 55 = \frac{1}{2} \cdot 10 \cdot (10+1) \end{aligned}$$

Allgemein:

$$1+2+\dots+N = \frac{1}{2} \cdot N \cdot (N+1)$$

Algorithmenanalyse

Mathematische Analyse: zweites Beispiel.

► Beispiel

```
void g( /*...*/ ) {  
    int counter {};  
    for( int i {}; i < N; ++i )  
        for( int j {i+1}; j < N; ++j )  
            if( v[i] == v[j] ) ++counter;  
}
```

N	i < N	j < N	<
0	1	0	1 + 0
1	2	1	2 + 1
2	3	2+1	3 + 2+1
3	4	3+2+1	4 + 3+2+1
4	5	4+3+2+1	5 + 4+3+2+1
5	6	5+4+3+2+1	6 + 5+4+3+2+1
...			$\underbrace{\hspace{10em}}$ $= \frac{1}{2} (N+1)(N+2)$

Algorithmenanalyse

Mathematische Analyse: zweites Beispiel.

► Beispiel

```
void g( /*...*/ ) {  
    int counter {};  
    for( int i {}; i < N; ++i )  
        for( int j {i+1}; j < N; ++j )  
            if( v[i] == v[j] ) ++counter;  
}
```

<code>++i</code>	N
<code>++j</code>	$\frac{1}{2} N (N-1)$
<code>++counter</code>	zwischen 0 und $\frac{1}{2} N (N-1)$

Algorithmenanalyse

Mathematische Analyse: zweites Beispiel, Tildennotation.

- Welche Laufzeit als Funktion von N wird der Quellcode haben?

Operation	Häufigkeit	Zeit pro Op.	Gesamtzeit
Definition	$\sim N$	c_1	$\sim c_1 N$
Initialisierung	$\sim N$	c_2	$\sim c_2 N$
Vergleich ==	$\sim N^2$	c_3	$\sim c_3 N^2$
Vergleich <	$\sim N^2$	c_4	$\sim c_4 N^2$
vector Zugriff	$\sim N^2$	c_5	$\sim c_5 N^2$
Inkrement	$\sim N^2$	c_6	$\sim c_6 N^2$
Gesamt			$\sim c N^2$

Das Symbol \sim heißt *Tilde* :
nur der führende (größte)
Term wird beachtet
("Proportionalität").

$$g(N) \sim f(N) \Leftrightarrow \lim_{N \rightarrow \infty} \left| \frac{g(N)}{f(N)} \right| = c$$

$$\sim \underbrace{c_1 N + c_2 N}_{\text{Eingeschlossener Term}} + \underbrace{c_3 N^2 + c_4 N^2 + c_5 N^2 + c_6 N^2}_{\text{Führender Term}}$$

$$\sim (c_1 + c_2) N + \underbrace{(c_3 + c_4 + c_5 + c_6) N^2}_{\sim c N^2}$$

Eingeschlossener Term
(wird weggelassen)

Führender Term

Algorithmenanalyse

Mathematische Analyse: Wachstum der Laufzeit in Abhängigkeit von der Größe der zu verarbeitenden Datenmenge N .

- ▶ Die Laufzeit eines Programms kann also analytisch dargestellt werden als
 - eine *Konstante* c
 - *multipliziert* mit einem größten Term $f(N)$, dem sog. *führenden Term* (wie etwa N^2)
 - *plus* einigen kleineren Termen $e(N)$, die sog. *eingeschlossenen Terme*.
- ▶ Bei immer größer werdendem N entscheidet der Einfluss des führenden Terms praktisch allein.
 - Begründung: wächst $f(N)$ asymptotisch stärker als $e(N)$ so gilt $\lim_{N \rightarrow \infty} \frac{e(N)}{f(N)} = 0$
 - Wir interessieren uns besonders für immer größer werdende N und lassen daher die eingeschlossenen Terme weg.
- ▶ Bemerkungen:
 - Bei kleinen N können mehrere Terme zur Laufzeit beitragen.
 - Auch bei sehr fortgeschritten gestalteten Algorithmen können mehrere Terme zur Laufzeit beitragen...

Algorithmenanalyse

Exkurs: einige empirische Messwerte für die Dauer von Grundoperationen (eine Nanosekunde entspricht 10^{-9} Sekunden).

operation	example	nanoseconds [†]
integer add	<code>a + b</code>	2.1
integer multiply	<code>a * b</code>	2.4
integer divide	<code>a / b</code>	5.4
floating point add	<code>a + b</code>	4.6
floating point multiply	<code>a * b</code>	4.2
floating point divide	<code>a / b</code>	13.5
sine	<code>Math.sin(theta)</code>	91.3
arctangent	<code>Math.atan2(y, x)</code>	129.0
...

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

Algorithmenanalyse

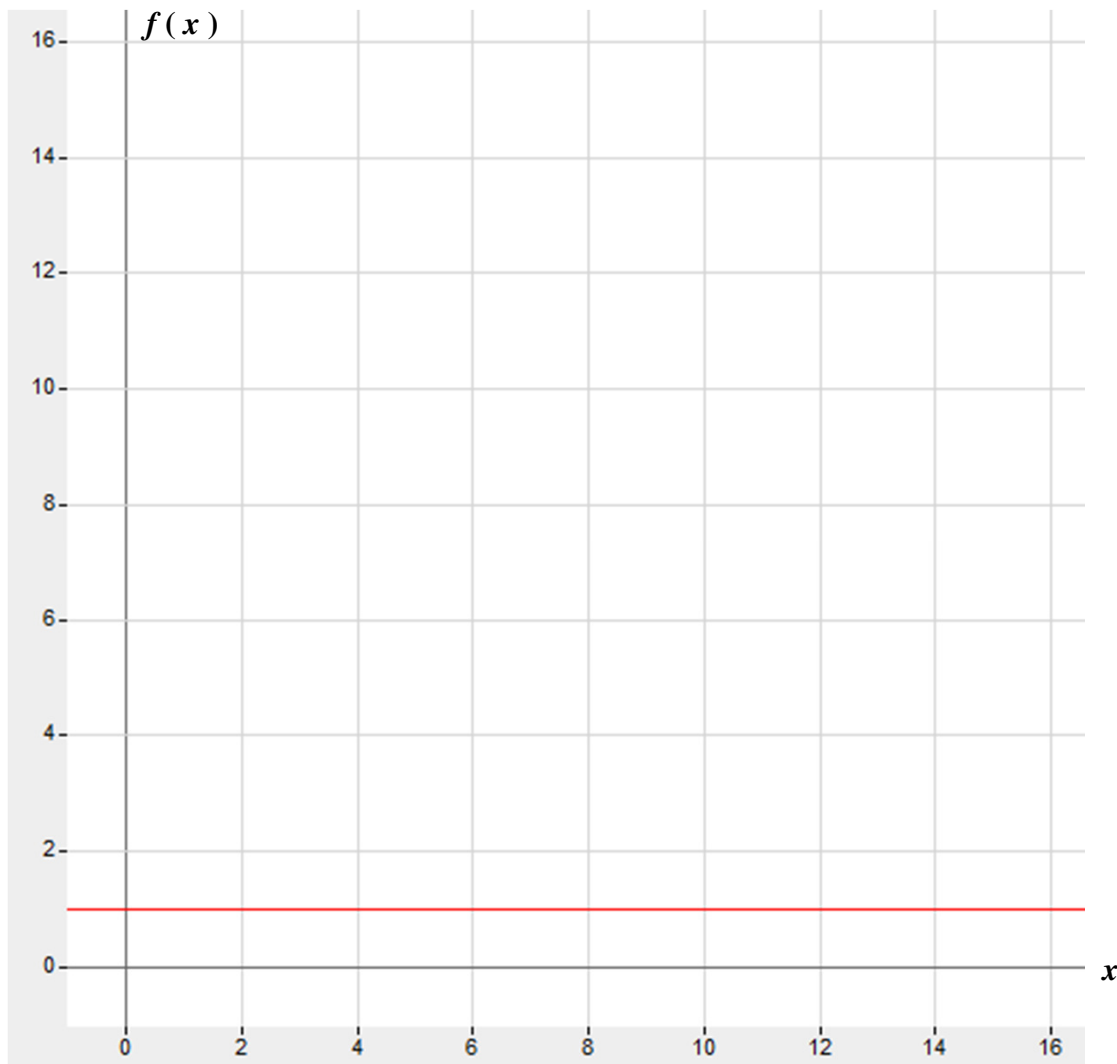
Mathematische Analyse: Wachstum von Funktionen.

- ▶ Viele Algorithmen haben in der Regel ein Laufzeitverhalten, das zu einer der folgenden Funktionen proportional ist.
 - **1** – unabhängig von der Menge der Eingangsdaten werden die Anweisungen nur einmal oder wenige Male ausgeführt: *konstantes* Laufzeitverhalten.
 - **$\log N$** – Das Programm wird mit wachsender Menge an Eingangsdaten N allmählich langsamer: *logarithmisches* Laufzeitverhalten.
 - **N** – Verdoppelt/verdreifacht/... sich N , trifft dies auch auf die Laufzeit zu: *lineares* Laufzeitverhalten.
 - **$N \log N$** – Wenn N sich verdoppelt/verdreifacht/... wird die Laufzeit etwas mehr (aber nicht viel mehr) als doppelt/dreifach/... so groß.
 - **N^2, N^3** – Z.B. doppelt (N^2) und dreifach (N^3) verschachtelte Schleifen, solche Algorithmen laufen schon mit mäßig wachsendem N deutlich länger und sind daher nur für kleine N geeignet: *quadratisches / kubisches*, allgemein *polynomiales* Laufzeitverhalten (*gilt oft als praktische Grenze der Einsetzbarkeit*).
 - **2^N** – Diese Algorithmen kommen für praktische Zwecke mit großen Datenmengen kaum in Frage: *exponentielles* Laufzeitverhalten.

Algorithmenanalyse

Mathematische Analyse: Wachstum von Funktionen.

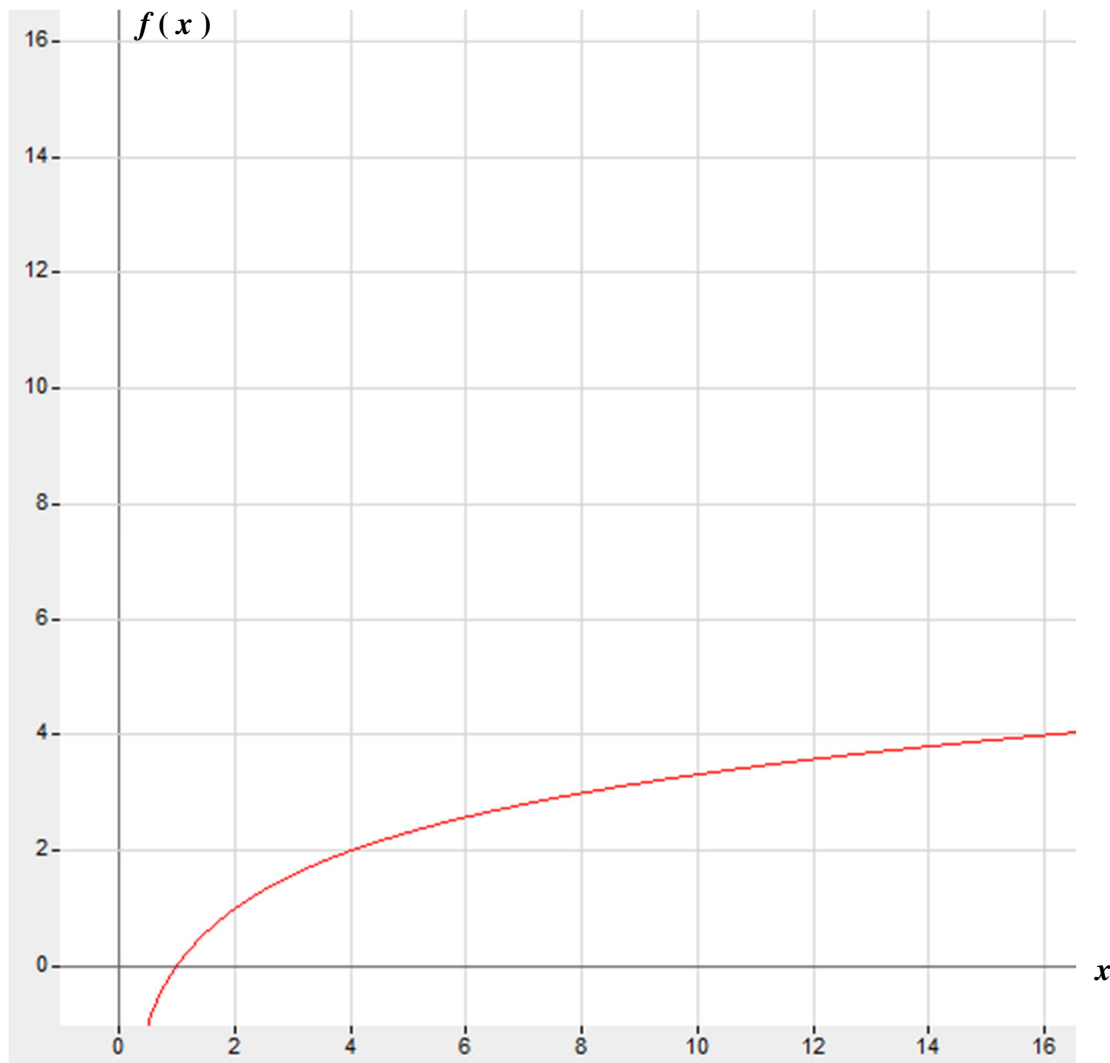
- Graph einer konstanten Funktion $f(x) = 1$



Algorithmenanalyse

Mathematische Analyse: Wachstum von Funktionen.

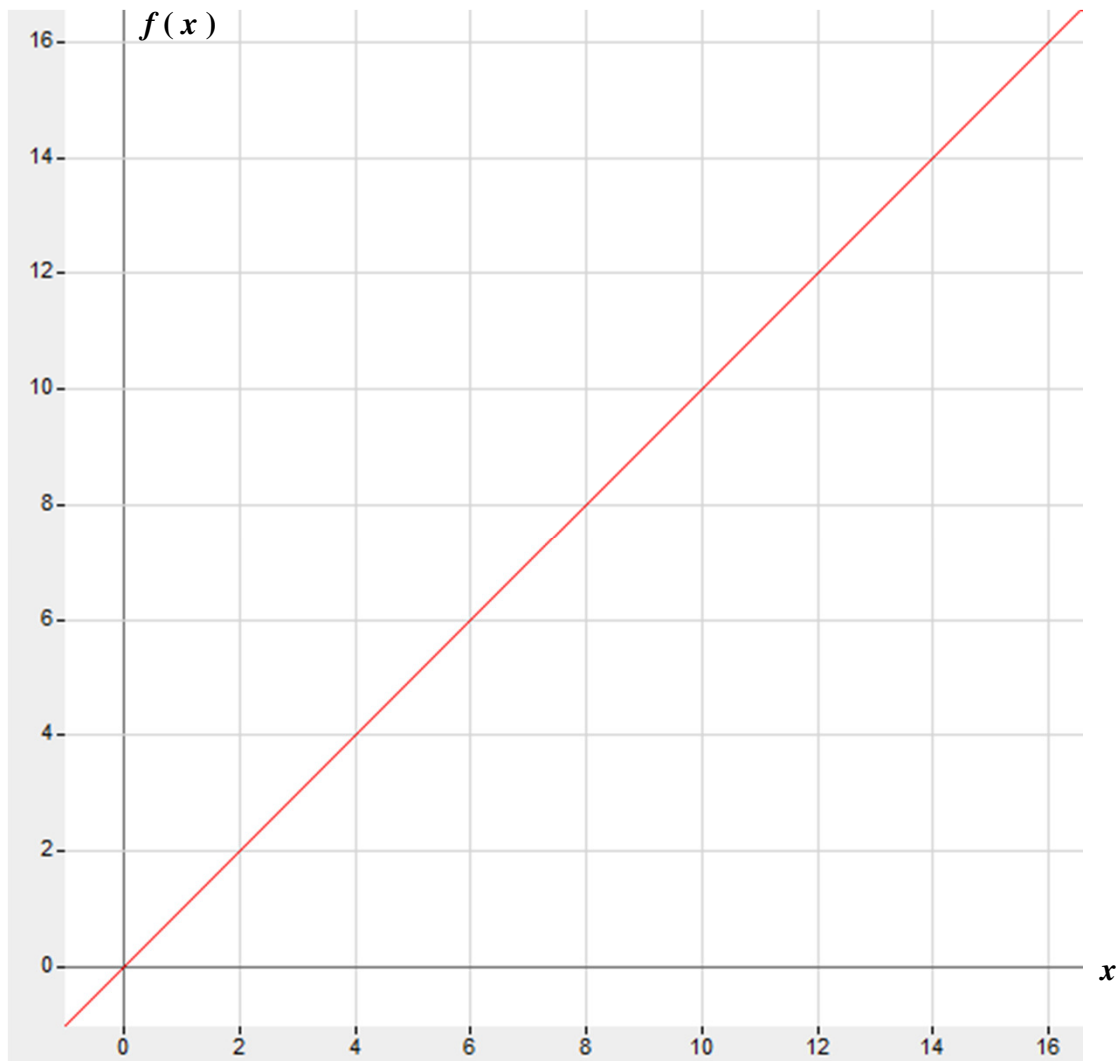
- Graph einer logarithmisch wachsenden Funktion $f(x) = \log_2 x$



Algorithmenanalyse

Mathematische Analyse: Wachstum von Funktionen.

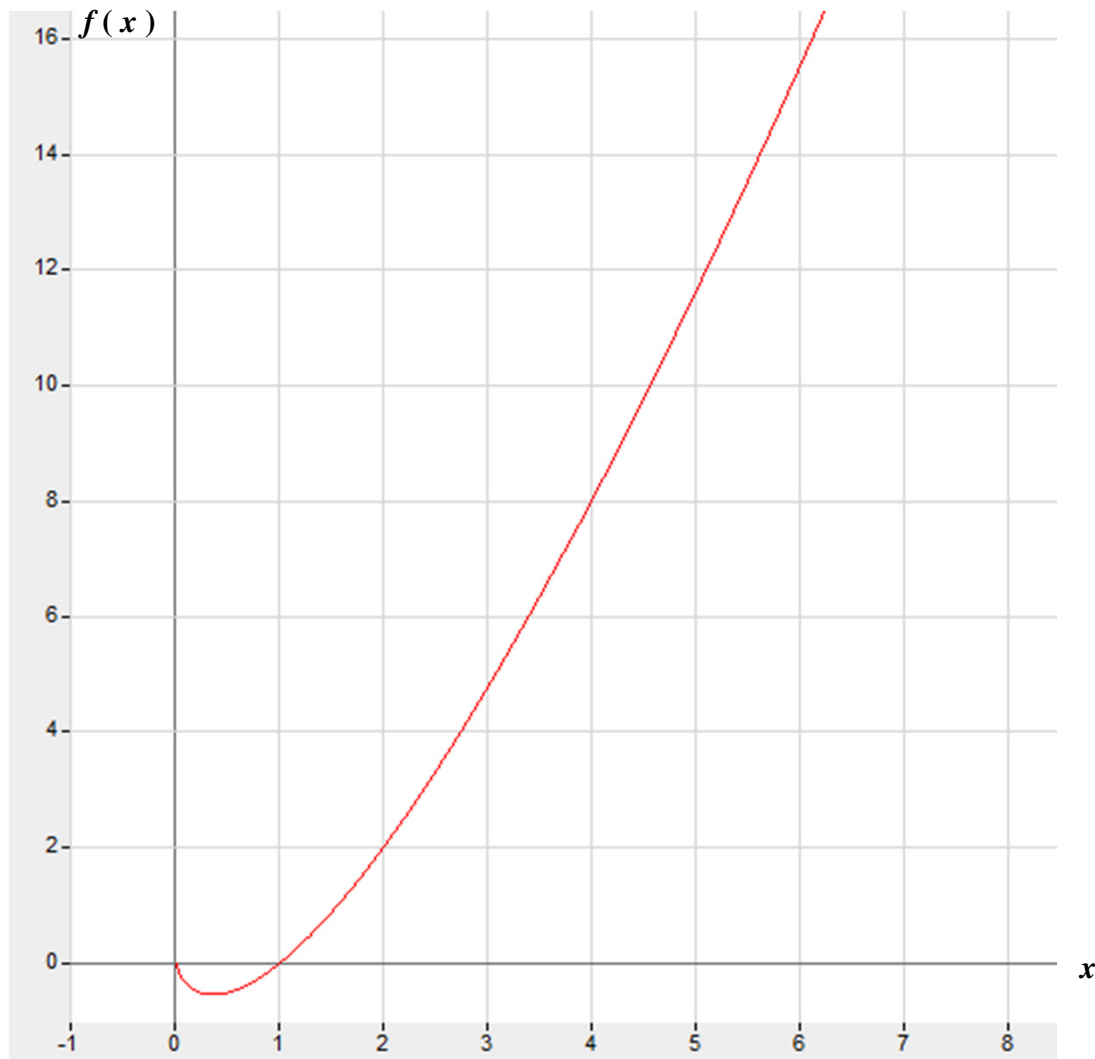
- Graph einer linear wachsenden Funktion $f(x) = x$



Algorithmenanalyse

Mathematische Analyse: Wachstum von Funktionen.

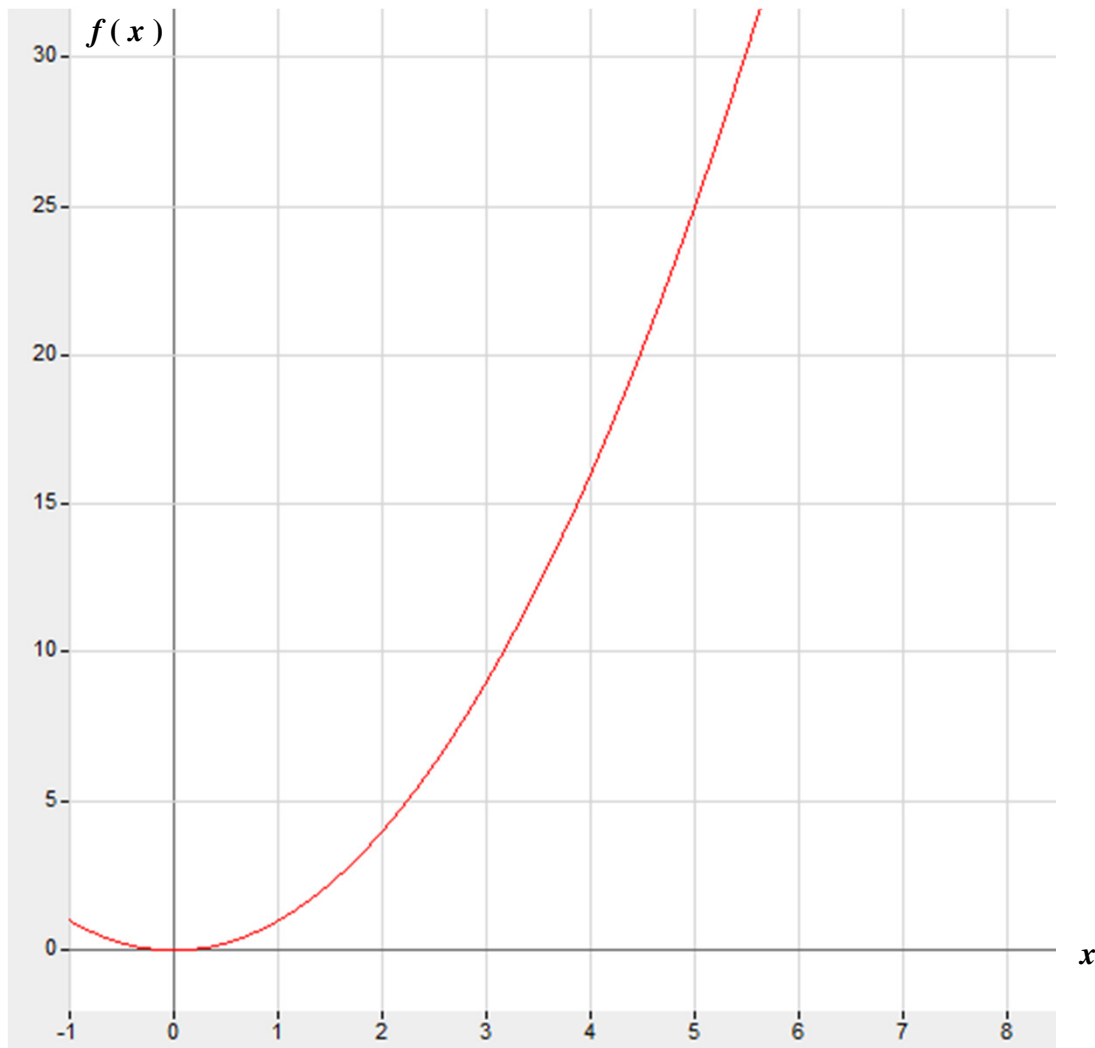
- Graph einer "log-linear" wachsenden Funktion $f(x) = x \log_2 x$



Algorithmenanalyse

Mathematische Analyse: Wachstum von Funktionen.

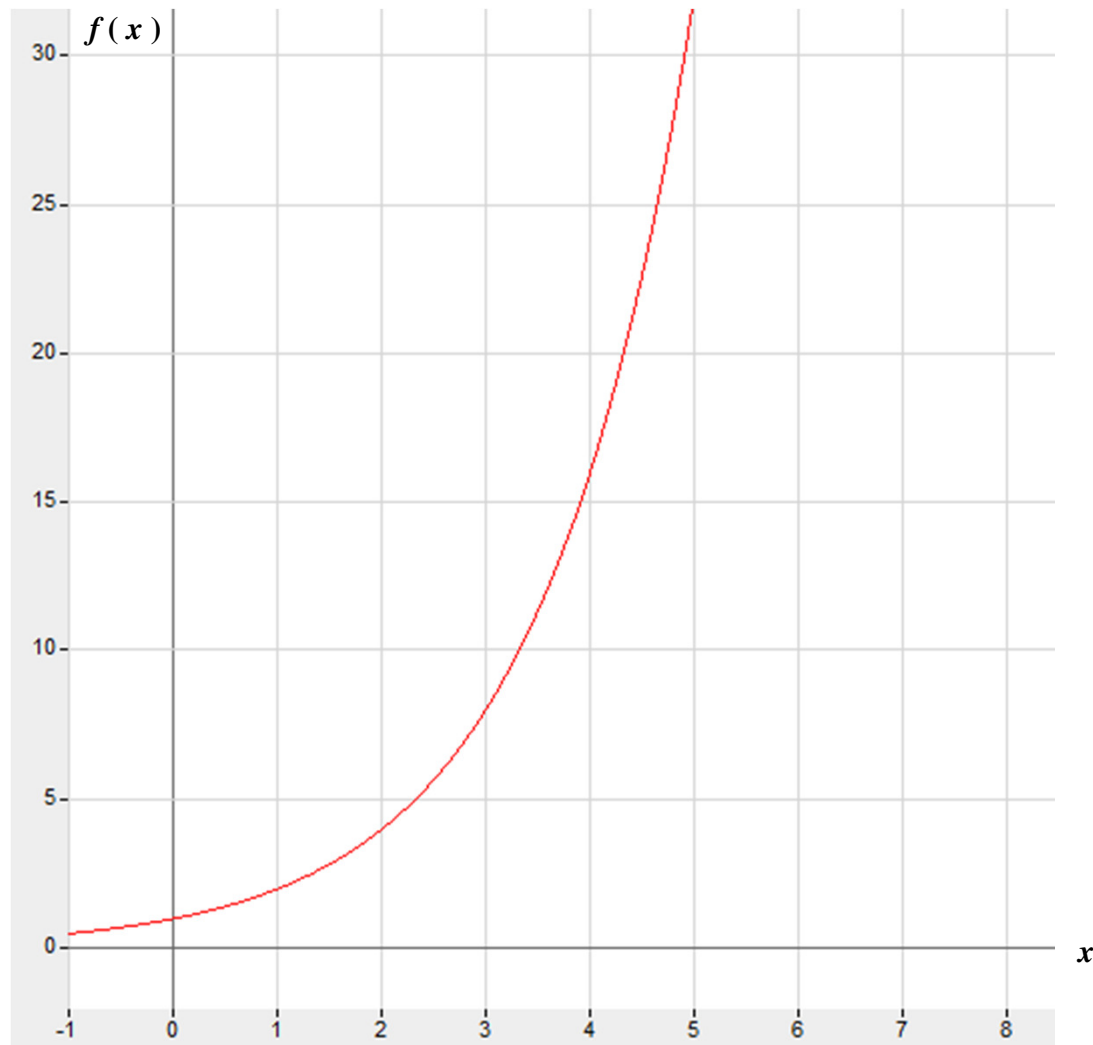
- Graph einer polynomial wachsenden Funktion $f(x) = x^2$



Algorithmenanalyse

Mathematische Analyse: Wachstum von Funktionen.

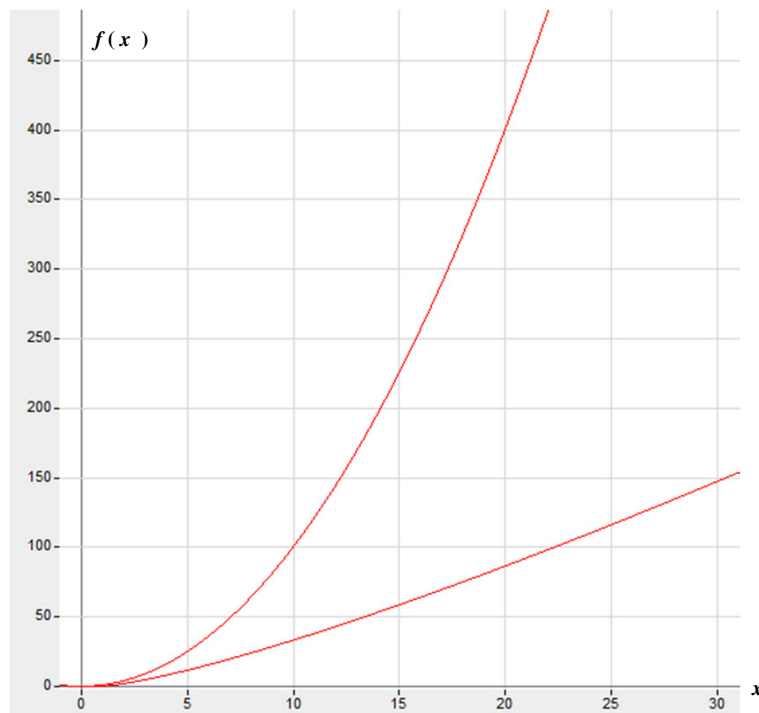
- Graph einer exponentiell wachsenden Funktion $f(x) = 2^x$



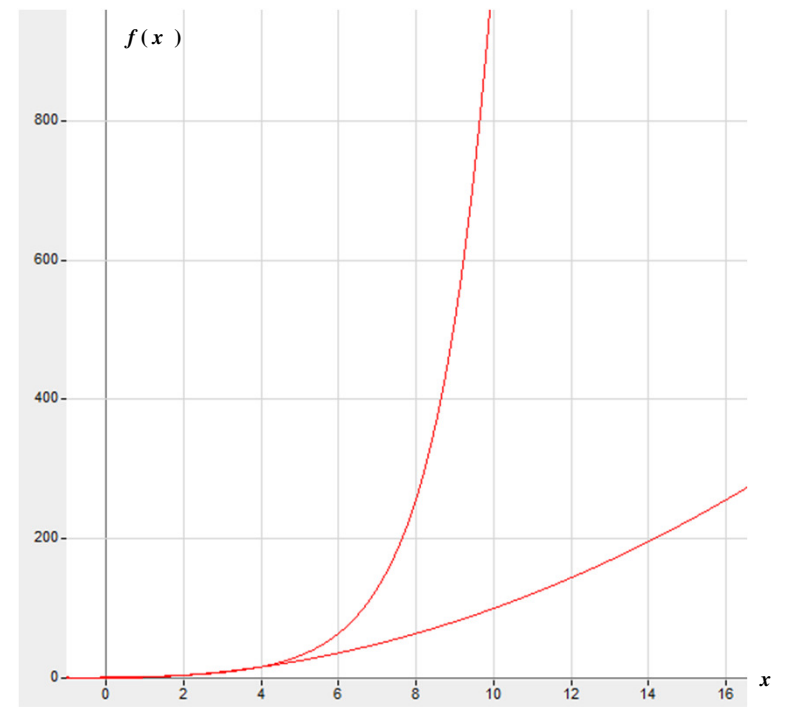
Algorithmenanalyse

Mathematische Analyse: Wachstum von Funktionen.

- Wachstum polynomial x^2 und "log-linear" $x \log_2 x$



- Wachstum exponentiell 2^x und polynomial x^2



Algorithmenanalyse

Mathematische Analyse: O -Notation ['gro:s o: no:tatsi:o:n] zur Darstellung von asymptotischen Laufzeit-Obergrenzen.

- ▶ Definition: eine Funktion $g(N)$ gehört zu $O(f(N))$
 - wenn es Konstanten c_0 und N_0 gibt,
 - so dass $g(N) \leq c_0 f(N)$ für alle $N > N_0$ gilt. Bzw.: $\lim_{N \rightarrow \infty} \left| \frac{g(N)}{f(N)} \right| < \infty$;
- ▶ Schreibweise: $g(N) \in O(f(N))$.
- ▶ Aussprache: "g wächst nicht schneller als f".
 - Umgangssprachlich sagt man bei $O(\dots)$ "... Obergrenze", also z.B. bei $O(N^2)$ "quadratische Obergrenze".
- ▶ Die O -Notation beschreibt eine asymptotische obere Wachstumsschranke.
 - Man erhält vernünftige, akzeptabel angenäherte Aussagen zu Laufzeit-Obergrenzen für große N , ohne jeden einzelnen Term der Algorithmenanalyse mitzuführen.
 - Indem man wie bei der Tildennotation den größten ("führenden") Term festhält und die kleineren ("eingeschlossenen") Terme ignoriert.
 - Die O -Notation entspricht einfach der in den beiden Beispielen verwendeten Tildennotation ohne die Konstanten c .
 - Die O -Notation ist sowohl intuitiv/einfach als auch mathematisch genau definiert.

Algorithmenanalyse

Mathematische Analyse: O , Ω und Θ .

► O -Notation ("Groß O")

- Für asymptotische Obergrenzen.

► Ω -Notation ("Groß Omega")

- Entsprechend für asymptotische Untergrenzen.
- Eine Funktion $g(N)$ heißt $\Omega(f(N))$ wenn es Konstanten c_0 und N_0 gibt, so dass $g(N) \geq c_0 f(N)$ für alle $N > N_0$ gilt.
- $g(N) \in \Omega(f(N))$, d.h. "g wächst nicht langsamer als f".

► Θ -Notation ("Groß Theta")

- $g(N) \in \Theta(f(N)) \Leftrightarrow g(N) \in O(f(N)) \wedge g(N) \in \Omega(f(N))$

Algorithmenanalyse

Mathematische Analyse: zu den asymptotischen Grenzen.

► Einige Rechenoperationen

- $f(N) \in O(f(N))$
 - $c \cdot O(f(N)) \in O(f(N))$
 - $O(c \cdot f(N)) \in O(f(N))$
 - $O(f(N)) + O(f(N)) \in O(f(N))$
 - $O(f(N)) \cdot O(g(N)) \in O(f(N) \cdot g(N))$
 - $O(f(N)) + O(g(N)) \in O(\max(f(N), g(N)))$
 - $O(O(f(N))) \in O(f(N))$
 - $f(N) \in O(g(N)) \Rightarrow O(f(N)) + O(g(N)) \in O(g(N))$
-
- $\Theta(f(N)) \subseteq O(f(N))$
 - $\Theta(f(N)) \subseteq \Omega(f(N))$
 - $\Theta(f(N)) = O(f(N)) \cap \Omega(f(N))$

Algorithmenanalyse

Mathematische Analyse: einige Aspekte der O -Notation.

- ▶ Algebraische Ausdrücke kann man mittels der O -Notation erweitern, z.B. ausmultiplizieren (als wäre O gar nicht vorhanden), und man kann dann alle außer den größten O -Term fallen lassen.
 - Beispiel:
$$\begin{aligned} & (N + O(1)) (N + O(\log N) + O(1)) = \\ & = N^2 + O(N) + O(N \log N) + O(\log N) + O(N) + O(1) = \\ & = N^2 + O(N \log N) \end{aligned}$$
- ▶ Die O -Notation enthält versteckt die beiden Konstanten c_0 und N_0 hinter denen sich ggf. wichtige Implementierungsdetails verbergen.
 - Ein $O(f(N))$ Laufzeitverhalten sagt nichts aus, falls $N < N_0$ ist und/oder c_0 viel Implementierungs-Overhead beinhaltet.
- ▶ Womöglich würde man für ein konkretes Problem ein Verfahren, das für die Verarbeitung von N Elementen N^2 Nanosekunden läuft, einem Verfahren vorziehen, das dafür $\log N$ Jahre braucht.
 - Die Algorithmenanalyse mittels O -Notation liefert aber keinerlei Hinweise für eine solche Entscheidung.

Algorithmenanalyse

Mathematische Analyse: Wachstum von Funktionen.

growth rate	name	typical code framework	description	example	$T(2N) / T(N)$
1	constant	<code>a = b + c;</code>	statement	add two numbers	1
$\log N$	logarithmic	<pre>while (N > 1) { N = N / 2; ... }</pre>	divide in half	binary search	~ 1
N	linear	<pre>for (int i = 0; i < N; i++) { ... }</pre>	loop	find the maximum	2
$N \log N$	linearithmic		divide and conquer	mergesort	~ 2
N^2	quadratic	<pre>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }</pre>	double loop	check all pairs	4
N^3	cubic	<pre>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }</pre>	triple loop	check all triples	8
2^N	exponential		exhaustive search	check all possibilities	$T(N)$

Algorithmenanalyse

Mathematische Analyse: Wachstum von Funktionen.

growth rate	name	description	effect on a program that runs for a few seconds	
			time for 100x more data	
1	constant	independent of input size	-	
$\log N$	logarithmic	nearly independent of input size	-	
N	linear	optimal for N inputs	a few minutes	
$N \log N$	linearithmic	nearly optimal for N inputs	a few minutes	
N^2	quadratic	not practical for large problems	several hours	
N^3	cubic	not practical for medium problems	several weeks	
2^N	exponential	useful only for tiny problems	forever	

Übung

Mathematische Analyse.

► Aufgabe:

- Entwickeln Sie eine Formel der Art

$$c_0 + c_1 N + c_2 N^2$$

welche die Laufzeit des Programms aus der letzten Übung (empirische Untersuchung der Laufzeit) möglichst gut beschreibt. Geben Sie an, in welcher Größenordnung die Laufzeit abhängig von N wächst.

- Vergleichen Sie die mit dieser Formel vorhergesagten Zeiten mit den von Ihnen tatsächlich gemessenen Zeiten.

► "Wartezeit-Tabelle" für Sekunden

10^2	in zwei Minuten	10^7	in vier Monaten
10^3	nach einer Viertelstunde	10^8	nach drei Jahren
10^4	in drei Stunden	10^9	nach drei Jahrzehnten
10^5	nach einem Tag	10^{10}	nach drei Jahrhunderten
10^6	nach anderthalb Wochen	10^{11}	nie

- ### ► Versuchen Sie auch, ein gewisses Gefühl für Größenordnungen zu entwickeln.

Algorithmenanalyse

Exkurs: Größenordnungen.

Beispiel	Größenordnung Sekunden ca.	Größenordnung Jahre ca.	
Planck-Zeit	10^{-43} 2^{-143}	10^{-51} 2^{-169}	(phys. Grenze)
Taktzeit Gigahertz Prozessor	10^{-9} 2^{-30}	10^{-17} 2^{-55}	(Nanosekunde)
Ruhepuls	10^0 2^0	10^{-8} 2^{-25}	
24 Stunden	10^5 2^{17}	10^{-3} 2^{-9}	
365 Tage	10^7 2^{23}	10^0 2^0	
Kapitalwert-Halbierung bei 4% p.a.	10^9 2^{30}	10^1 2^3	(ca. 18 Jahre)
Kreide-Tertiär-Grenze	10^{15} 2^{51}	10^7 2^{24}	(ca. 65 Mio. Jahre)
Beginn des Kambriums	10^{16} 2^{54}	10^8 2^{27}	(ca. 540 Mio. Jahre)
Erdalter	10^{17} 2^{57}	10^9 2^{30}	(ca. 4,7 Mrd. Jahre)
Alter des Universums	10^{18} 2^{60}	10^{10} 2^{33}	(ca. 14 Mrd. Jahre)

Algorithmenanalyse

Mathematische Analyse: grundlegende Differenzengleichungen.

- ▶ Viele Algorithmen zerlegen ihre Aufgabe rekursiv in Teilprobleme, und setzen deren Teillösungen danach wieder rekursiv zur Gesamtlösung zusammen (sog. "divide-and-conquer").

- ▶ Beispiel: Summe der ganzen Zahlen von 1 bis n

```
// rekursiv
int sum_rec( int n ) {
    if( n == 1 ) return 1;
    return n + sum_rec( n-1 );
}
```

```
int sum( int n ) { // iterativ
    int s{};
    for( int i{1}; i <= n; ++i )
        s += i;
    return s;
}
```

- ▶ Beispiel: größter gemeinsamer Teiler der ganzen Zahlen m und n

```
// rekursiv
int ggT_rec( int m, int n ) {
    if( n == 0 ) return m;
    return ggT_rec( n, m%n );
}
```

```
int ggT( int m, int n ) { // iterativ
    int r{};
    while( n != 0 ) {
        r = m%n; m = n; n = r;
    }
    return m;
}
```

- ▶ Mit sog. Differenzengleichungen (Rekursionsgleichungen) kann die mathematische Beschreibung und Analyse solcher Funktionen erfolgen.

Algorithmenanalyse

Mathematische Analyse: grundlegende Differenzengleichungen.

- Differenzengleichung, wenn ein Programm rekursiv durch N Input-Objekte geht und in jedem Durchgang eines davon separiert.

$$C_N = C_{N-1} + N; \quad \text{mit } N \geq 2 \text{ und } C_1 = 1$$

$$C_N = C_{N-1} + N$$

$$= C_{N-2} + (N - 1) + N$$

$$= C_{N-3} + (N - 2) + (N - 1) + N$$

$$= \dots$$

$$= C_1 + 2 + \dots + (N - 1) + N$$

$$= 1 + 2 + \dots + (N - 1) + N$$

$$= \frac{N(N+1)}{2};$$

- $O(N^2)$

Algorithmenanalyse

Mathematische Analyse: grundlegende Differenzengleichungen.

- Differenzengleichung, wenn ein Programm N Input-Objekte in jedem Durchgang rekursiv halbiert.

$$C_N = C_{N/2} + 1; \quad \text{mit } N \geq 2 \text{ und } C_1 = 1$$

$$C_{2^n} = C_{2^{n-1}} + 1$$

$$= C_{2^{n-2}} + 1 + 1$$

$$= C_{2^{n-3}} + 1 + 1 + 1$$

$$= \dots$$

$$= C_{2^0} + n$$

$$= \mathbf{n + 1};$$

- $O(\log_2 N)$

Es sei $N = 2^n$ damit die Halbierungen ganzzahlig bleiben, also $\mathbf{n = \log_2 N}$

Hinweis:

Im Rahmen dieser Lehrveranstaltung bleiben bei den Differenzengleichungen und ihren Lösungen bestimmte rigorose Details unberücksichtigt, soweit es

- dem einfacheren Verständnis zugute kommt,
- das Problem nicht unangemessen vereinfacht
- und die Lösung (zumindest asymptotisch) nicht verfälscht.

Z.B. lässt sich die Lösung hier mathematisch nicht ohne weiteres für $N \neq 2^n$ verallgemeinern. Man müsste dafür etwa annehmen, dass $N/2$ auf $\lfloor N/2 \rfloor$ abgebildet sein soll usw.

Der Blick würde sich durch solche (zweifelloos auch relevanten) Fragen aber von den Kernpunkten entfernen, die zum grundlegenden Verständnis wichtig erscheinen.

Algorithmenanalyse

Mathematische Analyse: grundlegende Differenzengleichungen.

- Differenzengleichung, wenn ein Programm N Input-Objekte behandelt, halbiert und rekursiv mit allen Objekten (aus einer Hälfte) weitermacht.

$$C_N = C_{N/2} + N; \quad \text{mit } N \geq 2 \text{ und } C_1 = 0$$

$$C_{2^n} = C_{2^{n-1}} + 2^n$$

$$= C_{2^{n-2}} + 2^{n-1} + 2^n$$

$$= C_{2^{n-3}} + 2^{n-2} + 2^{n-1} + 2^n$$

$$= C_{2^0} + \dots + 2^{n-2} + 2^{n-1} + 2^n$$

$$= 0 + \dots + \frac{N}{4} + \frac{N}{2} + N$$

$$= 2N;$$

Es sei wieder $N = 2^n$ damit die Halbierungen ganzzahlig bleiben, also $n = \log_2 N$

- $O(N)$

Algorithmenanalyse

Mathematische Analyse: grundlegende Differenzengleichungen.

- Differenzengleichung, wenn ein Programm N Input-Objekte behandelt, halbiert und rekursiv mit allen Objekten (aus beiden Hälften) weitermacht.

$$C_N = 2 \cdot C_{N/2} + N; \quad \text{mit } N \geq 2 \text{ und } C_1 = 0$$

$$C_{2^n} = 2 \cdot C_{2^{n-1}} + 2^n;$$

Es sei wieder $N = 2^n$ damit die Halbierungen ganzzahlig bleiben, also $n = \log_2 N$

$$\frac{C_{2^n}}{2^n} = \frac{C_{2^{n-1}}}{2^{n-1}} + 1$$

beide Seiten durch 2^n teilen

$$= \frac{C_{2^{n-2}}}{2^{n-2}} + 1 + 1$$

$$= \dots$$

$$= n;$$

- $O(N \log_2 N)$

Algorithmenanalyse

Mathematische Analyse: grundlegende Differenzengleichungen.

- Differenzengleichung, wenn ein Programm N Input-Objekte rekursiv halbiert und immer jeweils ein Objekt pro Hälfte behandelt.

$$C_N = 2 \cdot C_{N/2} + 1; \quad \text{mit } N \geq 2 \text{ und } C_1 = 0$$

$$C_{2^n} = 2 \cdot C_{2^{n-1}} + 1;$$

$$C_{2^{n-1}} = 2 \cdot C_{2^{n-2}} + 1;$$

$$C_{2^n} = 2 \cdot (2 \cdot C_{2^{n-2}} + 1) + 1$$

$$= 2 \cdot (2 \cdot (2 \cdot C_{2^{n-3}} + 1) + 1) + 1$$

$$= 2^0 + 2^1 + 2^2 + \dots + 2^{n-1} + 2^n$$

$$= 2N;$$

Es sei wieder $N = 2^n$ damit die Halbierungen ganzzahlig bleiben, also $n = \log_2 N$

- $O(N)$

Einfaches Suchen und Sortieren

Inhalt.

- ▶ Zur Algorithmenanalyse
- ▶ **Sequentielle Suche**
- ▶ Binäre Suche und Interpolationssuche
- ▶ Sortieren durch Auswahl, Sortieren durch Einfügung
- ▶ Zu den Eingangsdaten

Sequentielle Suche

Vergleicht jedes Objekt der Eingabedaten, eines nach dem anderen, mit einem Suchschlüssel.

- ▶ Die zu durchsuchenden Objekte (der Einfachheit halber Nummern vom `int` Typ) speichern wir in einem `vector<int>`.
- ▶ Der Algorithmus `seqSearch()` überprüft, ob die gesuchte Nummer `x` zu der vorher im `vector` Objekt abgespeicherten Menge von Nummern gehört.
- ▶ Dazu wird der Wert von `x` nacheinander mit den Werten aller N `vector` Elemente verglichen.
- ▶ Wird `x` gefunden, so gibt `seqSearch()` den `vector`-Index der Position zurück, an der die gesuchte Nummer gefunden wurde, andernfalls gibt `seqSearch()` den Wert `-1` zurück (*keine* Ausnahme).
- ▶ Rudimentärer Quellcode:

```
int seqSearch( const vector<int>& vV, int x, int li, int re ) {  
    for( int i{li}; i<=re; ++i )  
        if( x == vV.at(i) ) return i;  
    return -1;  
}
```

Sequentielle Suche

Vergleicht jedes Objekt der Eingabedaten, eines nach dem anderen, mit einem Suchschlüssel.

- ▶ Wichtigste analytische Erkenntnis: die Laufzeit hängt von den Daten ab.
 - Davon, ob das gesuchte Objekt unter den zu durchsuchenden Objekten vorhanden ist oder nicht.
 - Die **erfolglose Suche** endet immer erst, wenn alle N Objekte verglichen wurden.
 - Eine **erfolgreiche Suche** kann schon mit einem Treffer beim Objekt am ersten verglichenen `vector`-Index beendet sein.
- ▶ Um die Laufzeit vorherzusagen muss man also Annahmen über die zu durchsuchenden Eingangsdaten treffen.
 - Es erscheint sinnvoll, zunächst von zufälligen Eingangsdaten, also in unserem Beispiel per Zufall gesetzten `int` Werten im `vector` auszugehen.
 - Was bedeutet *zufällig*, wenn es nicht mehr um vereinfachte Beispiele geht?
 - Bei weiterer Überlegung erkennt man, dass die Frage nach dem Wesen der zu durchsuchenden Daten bei Suchverfahren allgemein eine entscheidende Rolle spielt.

Sequentielle Suche

Vergleicht jedes Objekt der Eingabedaten, eines nach dem anderen, mit einem Suchschlüssel.

- ▶ Die sequentielle Suche prüft N Werte für jede erfolglose und im Mittel $(N+1)/2$ Werte für jede erfolgreiche Suche.
 - $O(N)$.
 - Geht man davon aus, dass der Vergleich zweier Werte immer gleich lang dauert, so wird das Verfahren bei doppelt so großem N (d.h. bei doppelt so vielen zu durchsuchenden Werten) doppelt so lang dauern.
- ▶ Die erfolglose sequentielle Suche kann offensichtlich beschleunigt werden, indem die zu durchsuchenden Objekte vor der Suche in eine geeignete Reihenfolge gebracht werden.
 - Grund: bei aufsteigend (absteigend) sortierten Werten kann die sequentielle Suche beim ersten Wert abgebrochen werden, der größer (kleiner) als der gesuchte Wert ist.
- ▶ Die sequentielle Suche prüft aber auch auf sortierten Objekten für jede Suchoperation im ungünstigsten Fall N Werte und im Durchschnitt $(N+1)/2$ Werte für jeden Suchtreffer.
 - Auch $O(N)$.
 - Die erfolglose sequentielle Suche auf sortierten Objekten prüft aber durchschnittlich $(N+1)/2$ Werte, d.h. nur halb so viele wie auf unsortierten Objekten.

Sequentielle Suche

Anzahl der Vergleichsoperationen.

► N Eingabedaten unsortiert

	Suchtreffer	erfolglose Suche
günstigster Fall	1	N
ungünstigster Fall	N	N
durchschnittlicher Fall	$(N+1)/2$	N

► N Eingabedaten sortiert

	Suchtreffer	erfolglose Suche
günstigster Fall	1	1
ungünstigster Fall	N	N
durchschnittlicher Fall	$(N+1)/2$	$(N+1)/2$

Sequentielle Suche

Die Methode ist für größere Datenmengen nicht geeignet.

- ▶ Die Laufzeit der sequentiellen Suche ist proportional zu $M \cdot N$, wenn in einem Bestand von N Werten nach M Werten gesucht werden soll.
 - Verdoppelt sich entweder die Anzahl zu durchsuchender Werte oder die Anzahl zu suchender Werte, wird die Laufzeit doppelt so lang.
 - Verdoppeln sich beide, nimmt die Laufzeit quadratisch zu.
- ▶ Das Verfahren ist also in der Praxis bei größeren Datenmengen ungeeignet, Beispiel:
 - Es seien einige (im Bereich von 10^4) Kreditkartendaten abhanden gekommen.
 - Man überprüft alle in Frage kommenden Transaktionen während des fraglichen Zeitraums (deren Anzahl N sei im Bereich von 10^8), ob eine der fraglichen M Kartenummern auftaucht.
 - Die Untersuchung einer einzigen Transaktion dauert z.B. c Mikrosekunden (eine Mikrosekunde entspricht 10^{-6} Sekunden).
 - Die Laufzeit durch alle Transaktionen liegt dann im Bereich von $\frac{1}{2} c \cdot 10^6$ Sekunden, also ca. c Wochen.

Einfaches Suchen und Sortieren

Inhalt.

- ▶ Zur Algorithmenanalyse
- ▶ Sequentielle Suche
- ▶ Binäre Suche und Interpolationssuche
- ▶ Sortieren durch Auswahl, Sortieren durch Einfügung
- ▶ Zu den Eingangsdaten

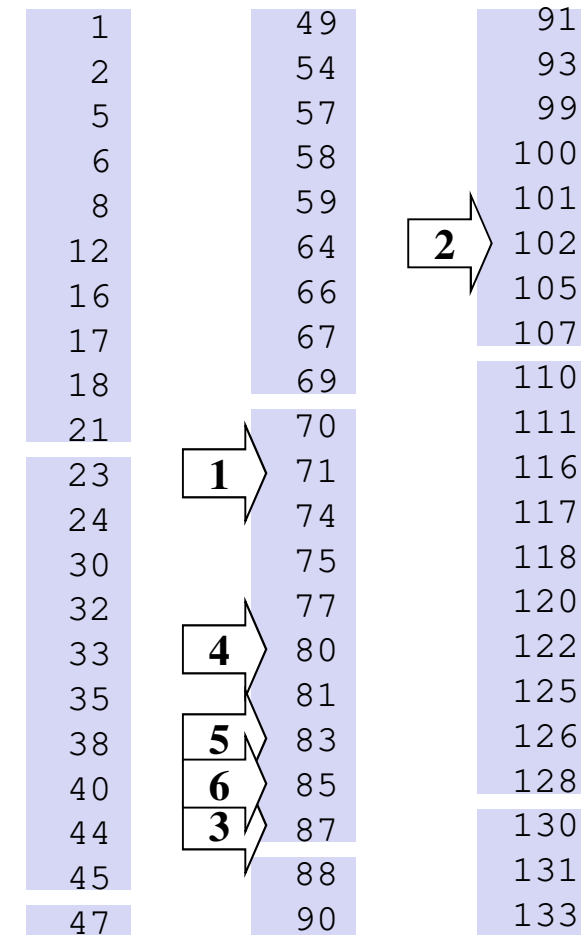
Binäre Suche

Ein naheliegendes Verfahren für die Suche in vorsortierten Objekten.

- Suche die 85 in den sortierten Werten.

Index u	Index o	Index m	Wert	zu 85
0	62	31	71	<
32	62	47	102	>
32	46	39	87	>
32	38	35	80	<
36	38	37	83	<
38	38	38	85	==

- $m = ((o-u) / 2) + u; \quad i_m = \lfloor (i_o - i_u) / 2 \rfloor$
- Für 63 Werte sind im ungünstigsten Fall sechs Vergleiche erforderlich.
 - Grund: N Vergleiche (Binäroperationen) können 2^N Werte absuchen, $63 < 2^6$.



**$N=63$ aufsteigend
sortierte Werte**

Binäre Suche

Ein naheliegendes Verfahren für die Suche in vorsortierten Objekten.

- ▶ Die zu durchsuchenden Objekte (der Einfachheit halber ganzzahlige Werte vom `int` Typ) speichern wir *aufsteigend sortiert* in einem `vector<int>`.
- ▶ Der Algorithmus `binSearch()` stellt genau die gleiche Funktionalität wie `seqSearch()` zur Verfügung.
- ▶ Das binäre Suchprinzip, nach dem `binSearch()` arbeitet, wurde eben erläutert.
- ▶ Rudimentärer Quellcode

```
int binSearch( const vector<int>& vV, int x, int ui, int oi ) {  
    int mid {0};  
    while( ui<=oi ) {  
        mid = ui+((oi-ui)/2); // somit int Ueberlauf vermeiden  
        if( x == vV.at(mid) ) return mid;  
        if( x < vV.at(mid) ) oi = mid-1;  
        else ui = mid+1;  
    }  
    return -1;  
}
```

Binäre Suche

Beachten Sie die Indexgrenzen beim Implementieren des binären Suchverfahrens.

► Quellcode für z.B. 101 Elemente:

```
int m1{0}, li1{m1}, re1{100};  
while( li1<=re1 ) {  
    m1 = li1+((re1-li1)/2);  
    cout << li1 << " " << re1 << " " << m1 << endl;  
    li1 = m1+1;  
}
```

```
int m2{0}, li2{m2}, re2{100};  
while( li2<=re2 ) {  
    m2 = li2+((re2-li2)/2);  
    cout << li2 << " " << re2 << " " << m2 << endl;  
    re2 = m2-1;  
}
```

Ausgabe:

```
0 100 50  
51 100 75  
76 100 88  
89 100 94  
95 100 97  
98 100 99  
100 100 100
```

```
0 100 50  
0 49 24  
0 23 11  
0 10 5  
0 4 2  
0 1 0
```

Binäre Suche

Ein naheliegendes Verfahren für die Suche in vorsortierten Objekten.

- ▶ Die binäre Suche prüft nie mehr als $\lfloor \log_2 N \rfloor + 1$ Werte.
 - $O(\log_2 N)$, Erläuterung:
 - Die binäre Suche mit N Elementen macht nach jeder Halbierung im ungünstigsten Fall mit genau $\lfloor N/2 \rfloor$ Elementen weiter, und am Ende muss noch der letzte Wert überprüft werden
 - Die Anzahl der Bits in der Binärdarstellung von N ist $\lfloor \log_2 N \rfloor + 1$ und man kann sich vorstellen, dass eine Iteration der binären Suche das niedrigste Bit der Binärdarstellung entfernt, d.h. "das Suchfeld halbiert" (analog gilt, dass in der dezimalen Darstellung von N die Anzahl der Dezimalstellen $\lfloor \log_{10} N \rfloor + 1$ ist, wird durch 10 geteilt, so wird die niedrigste Dezimalstelle entfernt).
- ▶ Mit binärer Suche kann man große Suchprobleme auf bis zu einer Milliarde Werten mit nur maximal 30 Vergleichen pro gesuchtem Wert lösen.
 - Viele Computer benötigen vermutlich mehr Zeit, um zwei `int` Werte aus dem Hauptspeicher zu lesen, als um dreißig `int` Vergleiche durchzuführen.
- ▶ Suchprobleme haben beim Programmieren eine so immense Bedeutung, dass Verfahren entwickelt wurden, die sogar noch schneller sind.

Interpolationssuche

Eine Anpassung der binären Suche.

- Suche die 85 in den sortierten Werten

Index u	Index o	Index m	Wert	zu 85
0	62	39	87	>
0	38	38	85	==

- Entspricht weitestgehend der binären Suche, aber der teilende Index wird nun durch Interpolation bestimmt.
- Prinzip:

$$m = u + ((x - v[u]) / (v[o] - v[u])) (o - u)$$
- Im Beispiel:
 - 1. Iteration: $0 + ((85 - 1) / (133 - 1)) (62 - 0) = 39,45$
 - 2. Iteration: $0 + ((85 - 1) / (85 - 1)) (38 - 0) = 38$

1	49	91
2	54	93
5	57	99
6	58	100
8	59	101
12	64	102
16	66	105
17	67	107
18	69	110
21	70	111
23	71	116
24	74	117
30	75	118
32	77	120
33	80	122
35	81	125
38	83	126
40	85	128
44	87	130
45	88	131
47	90	133

N=63 aufsteigend
sortierte Werte

Interpolationssuche

Eine Anpassung der binären Suche.

- ▶ Die Interpolationssuche wählt einen anderen Teilungsindex als die binäre Suche, arbeitet aber ansonsten gleich.
 - Die Interpolation beruht auf der Annahme, dass die zu durchsuchenden Eingangsdaten numerisch und gleichmäßig verteilt sind.
 - Aufgrund der Vorsortierung werden große Werte eher hinten in der Sequenz (also bei größeren Indizes) erwartet, kleine Werte eher vorn.
 - In einer alphabetisch sortierten Wortliste (Lexikon) sucht man beispielsweise auch so: man schlägt für "Zarathustra" eher das Ende auf, für "Abraham" eher den Anfang.
- ▶ Die Untersuchung des Laufzeitverhaltens der Interpolationssuche hat sich als schwierig herausgestellt.
 - Es fällt ein zusätzlicher Overhead für die Berechnung der Interpolation an.
 - Sind die Daten über den Wertebereich recht gleichmäßig verteilt, ist die Strategie, den teilenden Index zu interpolieren, sehr effizient.
 - Man konnte für zufällige Eingangsdaten $O(\log_2(\log_2 N))$ zeigen (vgl. weiterführend bei Sedgewick).
 - D.h. praktisch konstant, denn in diesem Universum ist $O(\log_2(\log_2 N)) < 7$.

Interpolationssuche

Eine Anpassung der binären Suche.

- ▶ Die zu durchsuchenden Objekte (der Einfachheit halber Nummern vom `int` Typ) speichern wir wieder *aufsteigend sortiert* in einem `vector<int>`.
- ▶ Der Algorithmus `polSearch()` implementiert genau die gleiche Funktionalität wie `seqSearch()` oder `binSearch()`.
- ▶ Das Suchprinzip der Interpolation wurde eben erläutert.
- ▶ Rudimentärer Quellcode analog zur binären Suche (vgl. weiter vorne).
 - Die Ermittlung des teilenden Index durch Halbierung
`mid = ui + ((oi-ui) / 2);`
muss zunächst ersetzt werden durch Interpolation
`mid = ui + ((x-vV.at(ui) * (oi-ui)) / (vV.at(oi) - vV.at(ui)));`
 - Das allein reicht aber nicht für eine lauffähige Implementierung dieses Verfahrens...
 - Was muss berücksichtigt werden, damit es zu keiner Endlosschleife kommen kann?
 - Was muss berücksichtigt werden, damit es zu keinem Speicherzugriff außerhalb des erlaubten Bereichs kommen kann?

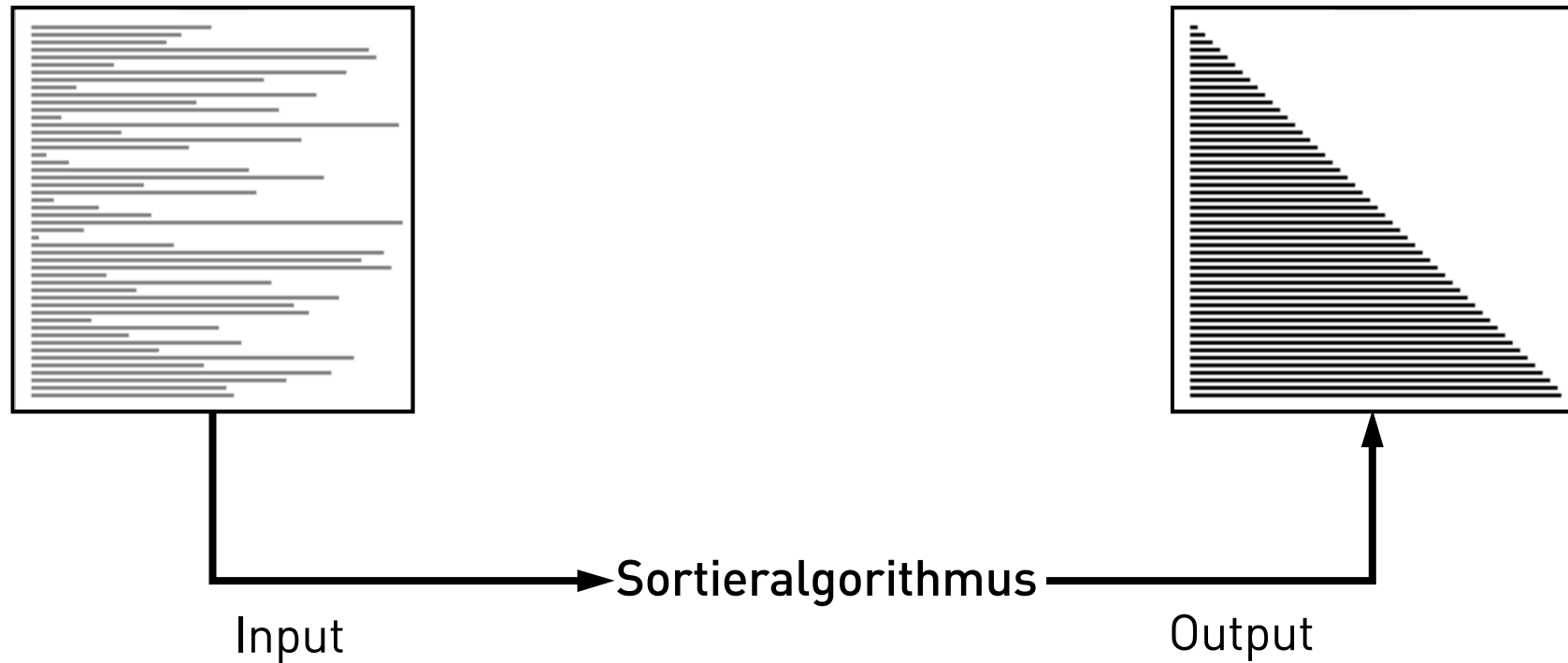
Einfaches Suchen und Sortieren

Inhalt.

- ▶ Zur Algorithmenanalyse
- ▶ Sequentielle Suche
- ▶ Binäre Suche und Interpolationssuche
- ▶ Sortieren durch Auswahl, Sortieren durch Einfügung
- ▶ Zu den Eingangsdaten

Sortieren

Das Sortieren von Daten gehört zu den grundlegendsten Aufgaben beim Programmieren.



Sortieren

Einige Begriffe und grundlegende Annahmen.

- ▶ Wir betrachten Verfahren zum Sortieren von *Daten*, die aus *Elementen* bestehen, die *Schlüssel* (Schlüsselwerte) enthalten.
 - Die Schlüssel, die ein (meist kleiner) Teil der Elemente sind, steuern den Sortiervorgang.
- ▶ Das Sortiervorgehen soll die Elemente neu ordnen.
 - So, dass die Schlüssel nach einer klaren Sortiervorschrift angeordnet sind (gewöhnlich die numerische oder alphabetische Reihenfolge).
- ▶ Wir betrachten *internes* Sortieren.
 - Die zu sortierenden Daten lassen sich im Hauptspeicher unterbringen.
 - D.h. schneller Zugriff auf die Elemente.
- ▶ Davon zu unterscheiden ist *externes* Sortieren.
 - Die zu sortierenden Daten sind auf einem persistenten Speicher (Festplatte, USB-Stift, Magnetband).
 - D.h. Zugriff auf die Elemente langsam, nur sequentiell oder höchstens in Blöcken.

Sortieren

Einige Begriffe und grundlegende Annahmen.

► Speicherplatz-Bedarf

- Nach der schon besprochenen Laufzeit das zweitwichtigste Leistungskriterium.
- Es lassen sich hierbei drei grundsätzliche Typen von Sortieralgorithmen unterscheiden:
 - 1) Kein zusätzlicher Speicherbedarf wird benötigt.
(Abgesehen von ggf. einer Hilfstabelle oder einem Stack,
mit höchstens $O(\log_2 N)$ Größe.)
D.h. Sortierung *in situ* (an Ort und Stelle).
 - 2) Zusätzlicher Speicherplatz wird benötigt, um alle N Indizes abzulegen.
 - 3) Zusätzlicher Speicherplatz wird benötigt, um eine Kopie aller N zu sortierenden Elemente abzulegen.

Sortieren

Einige Begriffe und grundlegende Annahmen.

► Stabilität

- Ein Sortieralgorithmus arbeitet *stabil*, wenn er die relative Reihenfolge der Elemente mit gleichen Schlüsselwerten bewahrt.
- Relevanz am Beispiel:

1. Key	2. Key
Berry C	1
Orbison R	1
Burdon E	2
Marley B	2
Santana C	2
Waits T	3
Young N	3
Cale JJ	4
Mercury F	4
Reed L	4

Ergebnisdaten:

Mit *stabilem* Algorithmus
nach dem zweiten Schlüssel
sortiert

1. Key	2. Key
Berry C	1
Burdon E	2
Cale JJ	4
Marley B	2
Mercury F	4
Orbison R	1
Reed L	4
Santana C	2
Waits T	3
Young N	3

Eingangsdaten:

nach dem ersten
Schlüssel vorsortiert

1. Key	2. Key
Berry C	1
Orbison R	1
Santana C	2
Marley B	2
Burdon E	2
Young N	3
Waits T	3
Reed L	4
Cale JJ	4
Mercury F	4

Ergebnisdaten:

Mit *instabilem* Algorithmus
nach dem zweiten Schlüssel
sortiert

Sortieren

Einige Begriffe und grundlegende Annahmen.

- ▶ Zwei grundlegende Funktionen, die wir für unsere Sortieralgorithmen brauchen werden
 - Der folgende, rudimentäre Quellcode vertauscht zwei Elemente aus den zu sortierenden Daten:

```
void swap( T& a, T& b ) { T tmp{a}; a=b; b=tmp; }  
void swap_if( T& a, T& b ) { if( b<a ) swap( a,b ); }
```
 - `T` ist der Typ der Schlüsselwerte der zu sortierenden Elemente (z.B. `int` oder `string`).
 - `T&` ist eine L-Referenz (ein Alias) für ein Element vom Typ `T`.
- ▶ Verwenden Sie in Ihren Programmen `std::swap()`, welches ähnlich wie oben definiert ist.
 - `swap_if()` gibt es in der C++11 StdLib nicht (muss ggf., z.B. wie oben, selbst definiert werden).

Sortieren durch direkte Auswahl

Englischer Begriff: Selection Sort.

- ▶ Suche zuerst den kleinsten Schlüsselwert und tausche das gefundene Element mit dem ersten Element.
- ▶ Suche dann den zweitkleinsten Schlüsselwert und tausche das gefundene Element mit dem zweiten Element.
- ▶ Usw. mit dem drittkleinsten, ... bis zum zweitgrößten Schlüsselwert (das letzte Element enthält dann bereits den größten Schlüsselwert).
- ▶ Das Verfahren heißt Sortieren durch direkte Auswahl, weil in jeder Iteration der kleinste verbleibende Schlüsselwert ausgewählt wird.
- ▶ *Erklären Sie die Arbeitsweise des Sortierens durch direkte Auswahl am Beispiel rechts.*

Ⓐ	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S	O	R	T	I	N	G	E	X	Ⓐ	M	P	L	E
A	A	O	R	T	I	N	G	Ⓔ	X	S	M	P	L	E
A	A	E	R	T	I	N	G	O	X	S	M	P	L	Ⓔ
A	A	E	E	T	I	N	Ⓖ	O	X	S	M	P	L	R
A	A	E	E	G	Ⓘ	N	T	O	X	S	M	P	L	R
A	A	E	E	G	I	N	T	O	X	S	M	P	Ⓖ	R
A	A	E	E	G	I	L	T	O	X	S	Ⓜ	P	N	R
A	A	E	E	G	I	L	M	O	X	S	T	P	Ⓝ	R
A	A	E	E	G	I	L	M	N	X	S	T	P	Ⓞ	R
A	A	E	E	G	I	L	M	N	O	S	T	Ⓟ	X	R
A	A	E	E	G	I	L	M	N	O	P	T	S	X	Ⓡ
A	A	E	E	G	I	L	M	N	O	P	R	Ⓢ	X	T
A	A	E	E	G	I	L	M	N	O	P	R	S	X	Ⓣ
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

Bildquelle: R. Sedgewick.

Sortieren durch direkte Auswahl

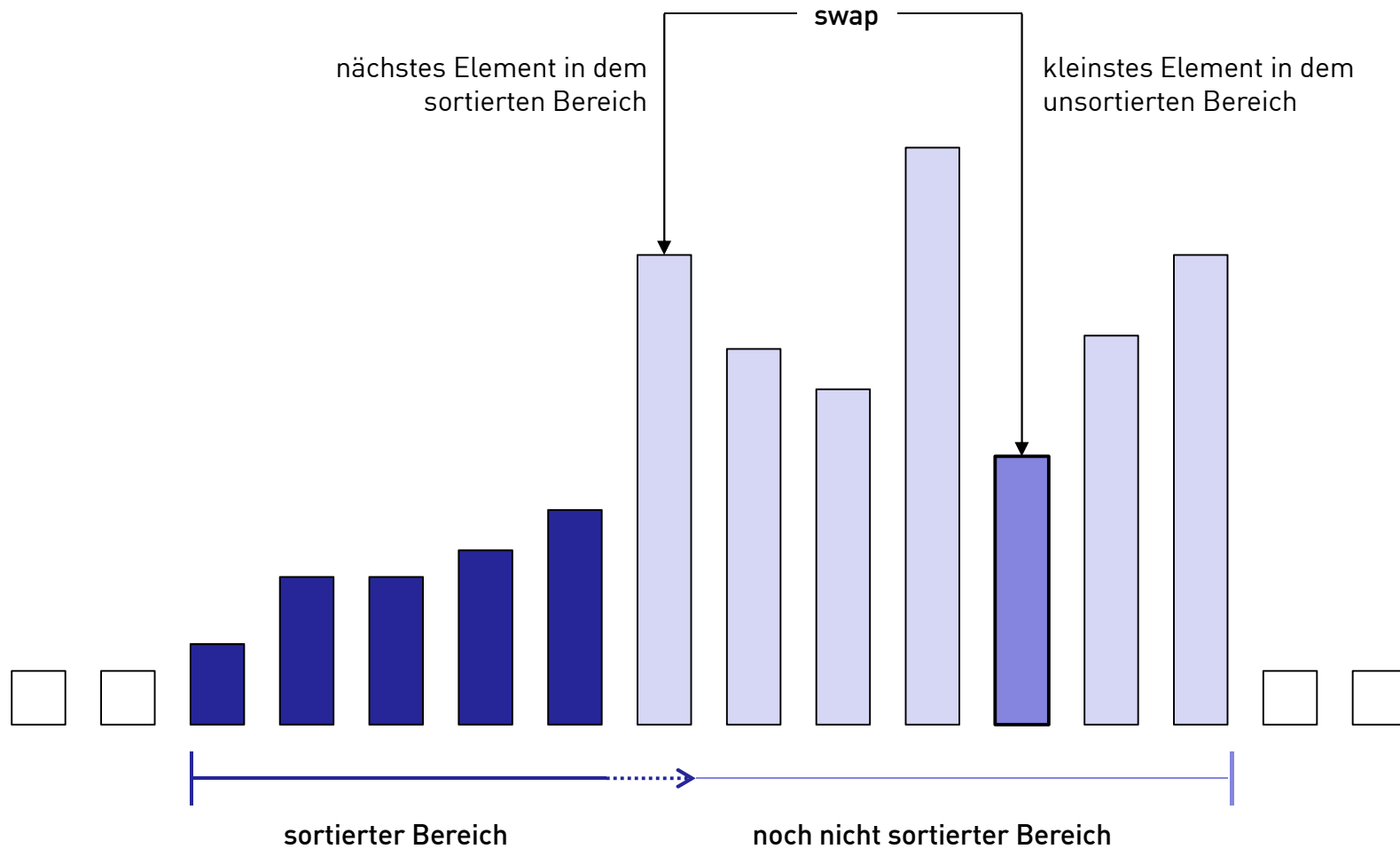
Mögliche Implementierung für `vector<int>`.

- Rudimentärer Quellcode für `int` Werte im `vector`

```
void selSort( vector<int>& vV, int ui, int oi )
{
    int min{0};
    for( int i{ui}; i<oi; ++i )
    {
        min = i;
        for( int j{i+1}; j<=oi; ++j )
            if( vV.at(j)<vV.at(min) ) min=j;
        swap( vV.at(i), vV.at(min) );
    }
}
```

Sortieren durch direkte Auswahl

Veranschaulichung.



Sortieren durch direkte Auswahl

Einige Eigenschaften.

- ▶ Die Anzahl der Tauschoperationen ist $N-1$,
die Anzahl der Vergleiche ist $(N-1) + (N-2) + \dots + 2 + 1 = N(N-1) / 2$
 - Das Laufzeitverhalten wird größten Term bestimmt, d.h. hier von den Vergleichen, also $O(N^2)$.
- ▶ Implementierungen sind oft instabil, das Verfahren kann aber auch stabil implementiert werden.
- ▶ Das Verfahren läuft für zufällige Eingangsdaten in etwa gleich lang wie für bereits sortierte Daten oder für Daten, die lauter gleiche Schlüssel haben.
- ▶ Die Methode hat im Vergleich zu den meisten (auch ausgeklügelteren) Verfahren einen großen Vorteil:
 - Sortieren durch direkte Auswahl kommt mit wenigen Tauschoperationen zurecht: *jedes Element wird nur höchstens einmal bewegt*.
 - D.h. die Anzahl an Tauschoperationen wächst höchstens in linearer Größenordnung.
 - Ist bei den zu sortierenden Daten der Aufwand für das Umkopieren von Elementen weit höher ist als für den Vergleich von Schlüsselwerten (etwa bei Daten mit sehr großen Elementen und kleinen Schlüsseln), kann das Verfahren gut geeignet sein.

Sortieren durch direkte Einfügung

Englischer Begriff: Insertion Sort.

- ▶ Nehme das zweite Element.
 - Tausche es mit dem ersten Element, falls dessen Schlüsselwert größer ist.
- ▶ Nehme dann das dritte Element.
 - Tausche es mit dem zweiten Element, falls dessen Schlüsselwert größer ist.
 - Tausche es dann mit ersten Element, falls dessen Schlüsselwert größer ist.
- ▶ Nehme dann das vierte Element.
 - Tausche es mit dem dritten Element, falls dessen Schlüsselwert größer ist.
 - Tausche es dann mit dem zweiten Element, falls dessen Schlüsselwert größer ist.
 - Tausche es dann mit dem ersten Element, falls dessen Schlüsselwert größer ist.
- ▶ Usw. bis zum letzten Element.
- ▶ *Erklären Sie die Arbeitsweise des Sortierens durch direkte Einfügung am Beispiel rechts.*

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E										
A	(S)	O	R	T	I	N	G	E	X	A	M	P	L	E										
A	(O)	S	R	T	I	N	G	E	X	A	M	P	L	E										
A	O	(R)	S	T	I	N	G	E	X	A	M	P	L	E										
A	O	R	S	(T)	I	N	G	E	X	A	M	P	L	E										
A	(I)	O	R	S	T	I	N	G	E	X	A	M	P	L	E									
A	I	(N)	O	R	S	T	I	N	G	E	X	A	M	P	L	E								
A	(G)	I	N	O	R	S	T	I	N	G	E	X	A	M	P	L	E							
A	(E)	G	I	N	O	R	S	T	I	N	G	E	X	A	M	P	L	E						
A	E	G	I	N	O	R	S	T	(X)	I	N	G	E	X	A	M	P	L	E					
A	(A)	E	G	I	N	O	R	S	T	X	I	N	G	E	X	A	M	P	L	E				
A	A	E	G	I	(M)	N	O	R	S	T	X	I	N	G	E	X	A	M	P	L	E			
A	A	E	G	I	M	N	O	(P)	R	S	T	X	I	N	G	E	X	A	M	P	L	E		
A	A	E	G	I	(L)	M	N	O	P	R	S	T	X	I	N	G	E	X	A	M	P	L	E	
A	A	E	(E)	G	I	L	M	N	O	P	R	S	T	X	I	N	G	E	X	A	M	P	L	E
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X	I	N	G	E	X	A	M	P	L	E

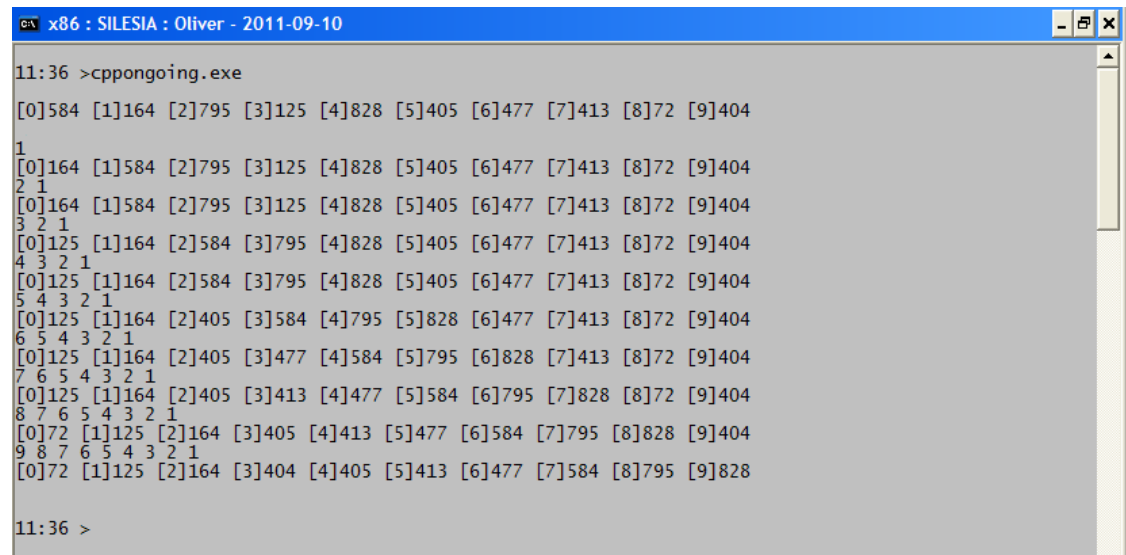
Bildquelle: R. Sedgewick.

Sortieren durch direkte Einfügung

Mögliche Implementierung für `vector<int>`.

► Rudimentärer Quellcode für `int` Werte im `vector`

```
void insSort_v0( vector<int>& vV, int ui, int oi ) {  
    for( int i{ui+1}; i<=oi; ++i )  
        for( int j{i}; j>ui; --j )  
            swap_if( vV.at(j-1), vV.at(j) );  
}
```



The screenshot shows a terminal window titled "x86 : SILESIA : Oliver - 2011-09-10". The command prompt shows the execution of "cppingoing.exe". The output displays a series of array states during the sorting process. Each line shows an array of 9 integers, with indices [0] to [9] in brackets. The arrays represent the state of the vector at different points in the insertion sort algorithm. The first line shows the initial array: [0]584 [1]164 [2]795 [3]125 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404. Subsequent lines show the array after inserting elements at their sorted positions, with the sorted portion of the array being [0]164 [1]164 [2]795 [3]125 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404. The final line shows the array after the last insertion: [0]72 [1]125 [2]164 [3]404 [4]405 [5]413 [6]477 [7]584 [8]795 [9]828.

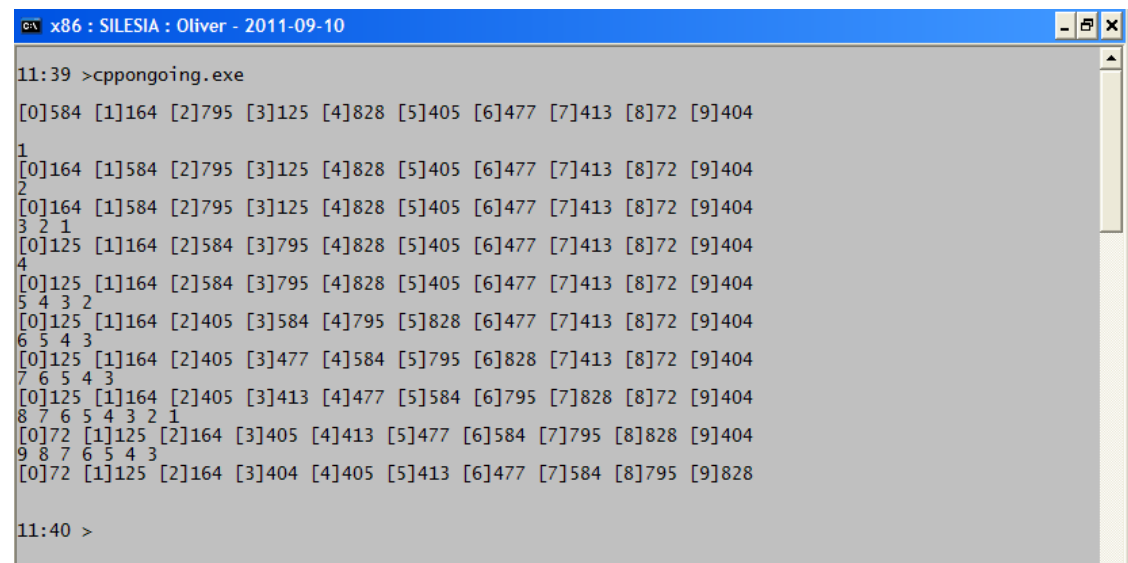
```
11:36 >cppingoing.exe  
[0]584 [1]164 [2]795 [3]125 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404  
1  
[0]164 [1]584 [2]795 [3]125 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404  
2 1  
[0]164 [1]584 [2]795 [3]125 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404  
3 2 1  
[0]125 [1]164 [2]584 [3]795 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404  
4 3 2 1  
[0]125 [1]164 [2]584 [3]795 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404  
5 4 3 2 1  
[0]125 [1]164 [2]405 [3]584 [4]795 [5]828 [6]477 [7]413 [8]72 [9]404  
6 5 4 3 2 1  
[0]125 [1]164 [2]405 [3]477 [4]584 [5]795 [6]828 [7]413 [8]72 [9]404  
7 6 5 4 3 2 1  
[0]125 [1]164 [2]405 [3]413 [4]477 [5]584 [6]795 [7]828 [8]72 [9]404  
8 7 6 5 4 3 2 1  
[0]72 [1]125 [2]164 [3]405 [4]413 [5]477 [6]584 [7]795 [8]828 [9]404  
9 8 7 6 5 4 3 2 1  
[0]72 [1]125 [2]164 [3]404 [4]405 [5]413 [6]477 [7]584 [8]795 [9]828  
11:36 >
```

Sortieren durch direkte Einfügung

Handwerklich etwas bessere Implementierung des selben Algorithmus.

► Quellcode, beschleunigte Version

```
void insSort_v1( vector<int>& vV, int ui, int oi ) {  
    int j{};  
    for( int i{ui+1}; i<=oi; ++i ) {  
        for( j=i; j>ui; --j ) {  
            if( vV.at(j-1) < vV.at(j) ) break;  
            swap( vV.at(j-1), vV.at(j) );  
        }  
    }  
}
```



```
x86 : SILESIA : Oliver - 2011-09-10  
11:39 >cpongoining.exe  
[0]584 [1]164 [2]795 [3]125 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404  
1  
[0]164 [1]584 [2]795 [3]125 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404  
2  
[0]164 [1]584 [2]795 [3]125 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404  
3 2 1  
[0]125 [1]164 [2]584 [3]795 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404  
4  
[0]125 [1]164 [2]584 [3]795 [4]828 [5]405 [6]477 [7]413 [8]72 [9]404  
5 4 3 2  
[0]125 [1]164 [2]405 [3]584 [4]795 [5]828 [6]477 [7]413 [8]72 [9]404  
6 5 4 3  
[0]125 [1]164 [2]405 [3]477 [4]584 [5]795 [6]828 [7]413 [8]72 [9]404  
7 6 5 4 3  
[0]125 [1]164 [2]405 [3]413 [4]477 [5]584 [6]795 [7]828 [8]72 [9]404  
8 7 6 5 4 3 2 1  
[0]72 [1]125 [2]164 [3]405 [4]413 [5]477 [6]584 [7]795 [8]828 [9]404  
9 8 7 6 5 4 3  
[0]72 [1]125 [2]164 [3]404 [4]405 [5]413 [6]477 [7]584 [8]795 [9]828  
11:40 >
```

Sortieren durch direkte Einfügung

Handwerklich fortgeschrittene Implementierung des selben Algorithmus, die das Verfahren deutlich beschleunigt.

► Quellcode, weitere Beschleunigung

```
01 void insSort_v2( vector<int>& vV, int ui, int oi ) {  
02     int i{0}; int j{0}; int tmp{0};  
03     for( i=oi; ui<i; --i ) swap_if( vV.at(i-1), vV.at(i) );  
04     for( i=ui+2; i<=oi; ++i ) {  
05         tmp = vV.at(i);  
06         for( j=i; tmp<vV.at(j-1); --j )  
07             vV.at(j)=vV.at(j-1);  
08         vV.at(j) = tmp;  
09     }  
10 }
```

► Diese Implementierung läuft bei zufälligen Eingangsdaten *fast doppelt so schnell* wie unsere ursprüngliche Version _v0.

Sortieren durch direkte Einfügung

Fortgeschrittene Implementierung, Anmerkungen zum Quellcode `_v2`.

- ▶ Im Quellcode `_v1` kann man erkennen, dass in der inneren `for`-Schleife die Vergleichsoperation `j > u[i]` *nur* positiv ist, wenn das einzufügende Element (das sich an der aktuellen Indexposition `i` befindet) das bisher kleinste ist *und* wenn es den Anfang der Daten (die unterste Indexposition 0) erreicht.
- ▶ Die fortgeschrittene Implementierung sorgt dafür, dass kein Element in der inneren Iteration jemals die Indexposition 0 erreicht.
 - In einer Extra-Iteration vorab (Zeile 3) wird das Element mit dem minimalen Wert schon an den Anfang der Daten gebracht.
 - Diese Art von Kniff wird allgemein beim Programmieren häufig verwendet, das besondere Element wird meist als "Sentinel" (Wächter) bezeichnet.
 - Dadurch kann die entsprechende Vergleichsoperation in der inneren Schleife entfallen, und die Implementierung wird asymptotisch insgesamt schneller.

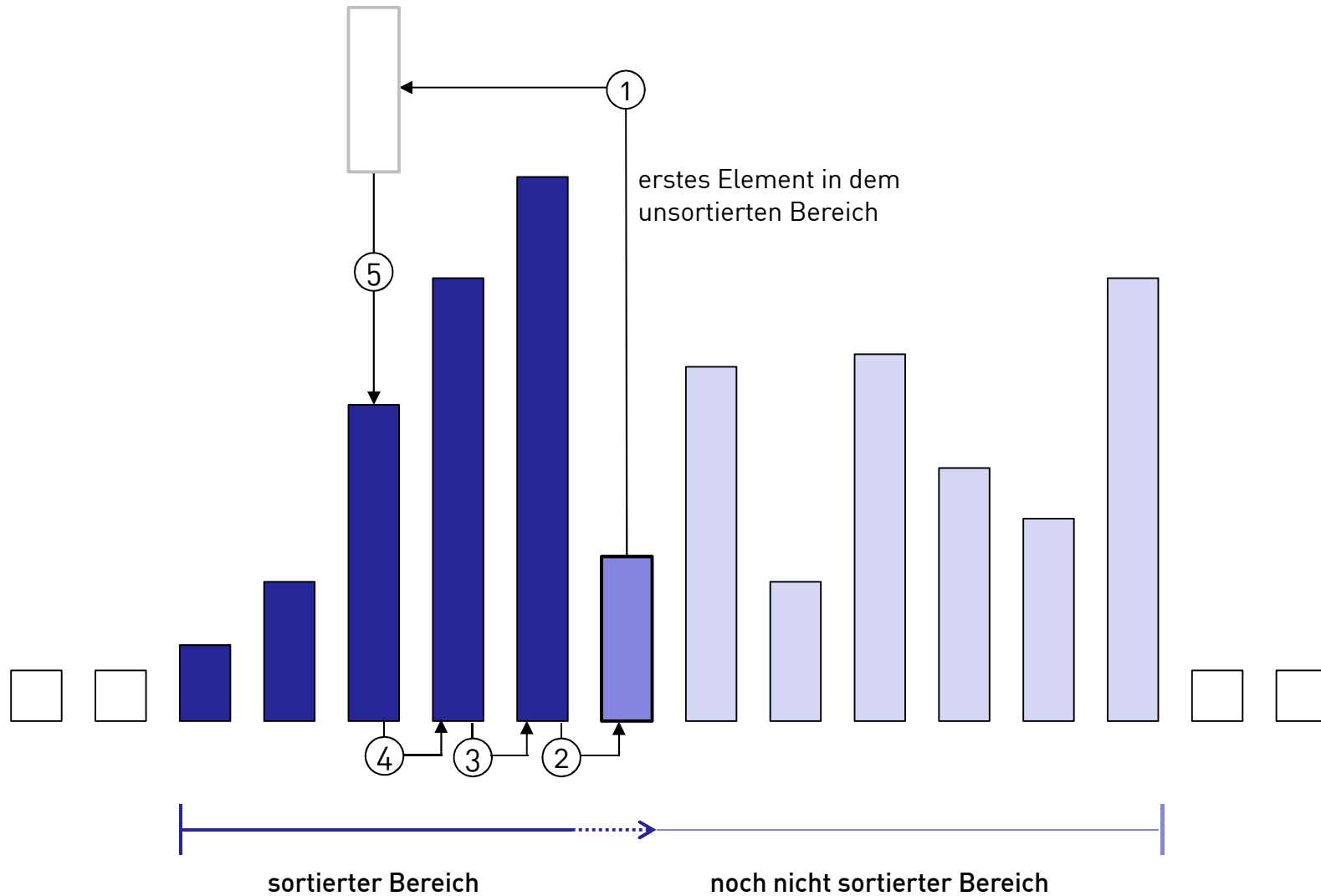
Sortieren durch direkte Einfügung

Fortgeschrittene Implementierung, Anmerkungen zum Quellcode `_v2`.

- ▶ Genau wie in `_v1` terminiert die innere Schleife wieder, sobald das einzufügende Element seine endgültige Position erreicht hat (Zeile 6).
- ▶ Genau wie in `_v1` gibt es keine Definitionen in Schleifen mehr, wodurch der Quellcode schneller läuft (alle Definitionen sind in Zeile 2).
- ▶ Die fortgeschrittene Implementierung geht bei der Umordnung der Elemente etwas anders vor.
 - Statt das nächste Element mit dem verbleibenden kleinsten zu tauschen werden alle Elemente, die in dem bereits sortierten Teil größer sind, um eine Position nach oben verschoben (Zeile 7).
 - Danach wird das aktuell einzufügende Element aus `tmp` an seine endgültige, nun freie Position gebracht (Zeile 8).
 - Durch diese Art der Umordnung kann die innere Schleife ohne eigentliche Tauschoperation programmiert werden, wodurch der Quellcode schneller läuft.

Sortieren durch direkte Einfügung

Veranschaulichung.



Sortieren durch direkte Einfügung

Einige Eigenschaften.

- ▶ Sortieren durch Einfügen benötigt
 - im Durchschnitt ungefähr $N^2/4$ Vergleiche und $N^2/4$ Bewegungen,
 - im ungünstigsten Fall doppelt so viele.
- ▶ Das Verfahren ist oft stabil implementiert, es kann aber auch instabil umgesetzt werden.
- ▶ Im Unterschied zum Sortieren durch direkte Auswahl hängt die Laufzeit beim Sortieren durch direkte Einfügung stark von der Reihenfolge der Eingangsdaten ab.
- ▶ Wenn z.B. die Datenmenge nicht zu groß und *beinahe* sortiert ist, dann ist das Sortieren durch direkte Einfügung schnell (linear).
 - In der Praxis kommen solche Daten häufig vor.
 - Es werden auch für solche Anwendungen oft Mehrzweck-Sortierverfahren (wie Quicksort, wird noch besprochen) eingesetzt, die aber dann im Vergleich oft weniger effizient sind.

Bubble Sort

Ein kurzer Blick auf dieses eigenartige Suchverfahren, das zumeist langsamer als die beiden bisher behandelten Algorithmen läuft.

- ▶ Durchlaufe die Daten vom ersten bis zum letzten Element und vertausche benachbarte Elemente, falls diese sich zueinander noch nicht in der richtigen Reihenfolge befinden.
- ▶ Wiederhole diese Durchläufe so lange, bis die Daten sortiert sind.
- ▶ *Man muss vielleicht etwas nachdenken um sich davon zu überzeugen, dass dieser Weg überhaupt zum Ziel führt...*

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	(A)	S	O	R	T	I	N	G	E	X	(E)	M	P	L
A	A	(E)	S	O	R	T	I	N	G	E	X	(L)	M	P
A	A	E	(E)	S	O	R	T	I	N	G	(L)	X	(M)	(P)
A	A	E	E	(G)	S	O	R	T	I	N	(L)	(M)	X	(P)
A	A	E	E	G	(I)	S	O	R	T	(L)	N	(M)	(P)	X
A	A	E	E	G	I	(L)	S	O	R	T	(M)	N	(P)	X
A	A	E	E	G	I	L	(M)	S	O	R	T	N	(P)	X
A	A	E	E	G	I	L	M	(N)	S	O	R	T	(P)	X
A	A	E	E	G	I	L	M	N	(O)	S	(P)	R	T	(X)
A	A	E	E	G	I	L	M	N	O	(P)	S	(R)	T	(X)
A	A	E	E	G	I	L	M	N	O	P	(R)	S	T	(X)
A	A	E	E	G	I	L	M	N	O	P	R	(S)	T	(X)
A	A	E	E	G	I	L	M	N	O	P	R	S	(T)	(X)
A	A	E	E	G	I	L	M	N	O	P	R	S	T	(X)

Bubble Sort

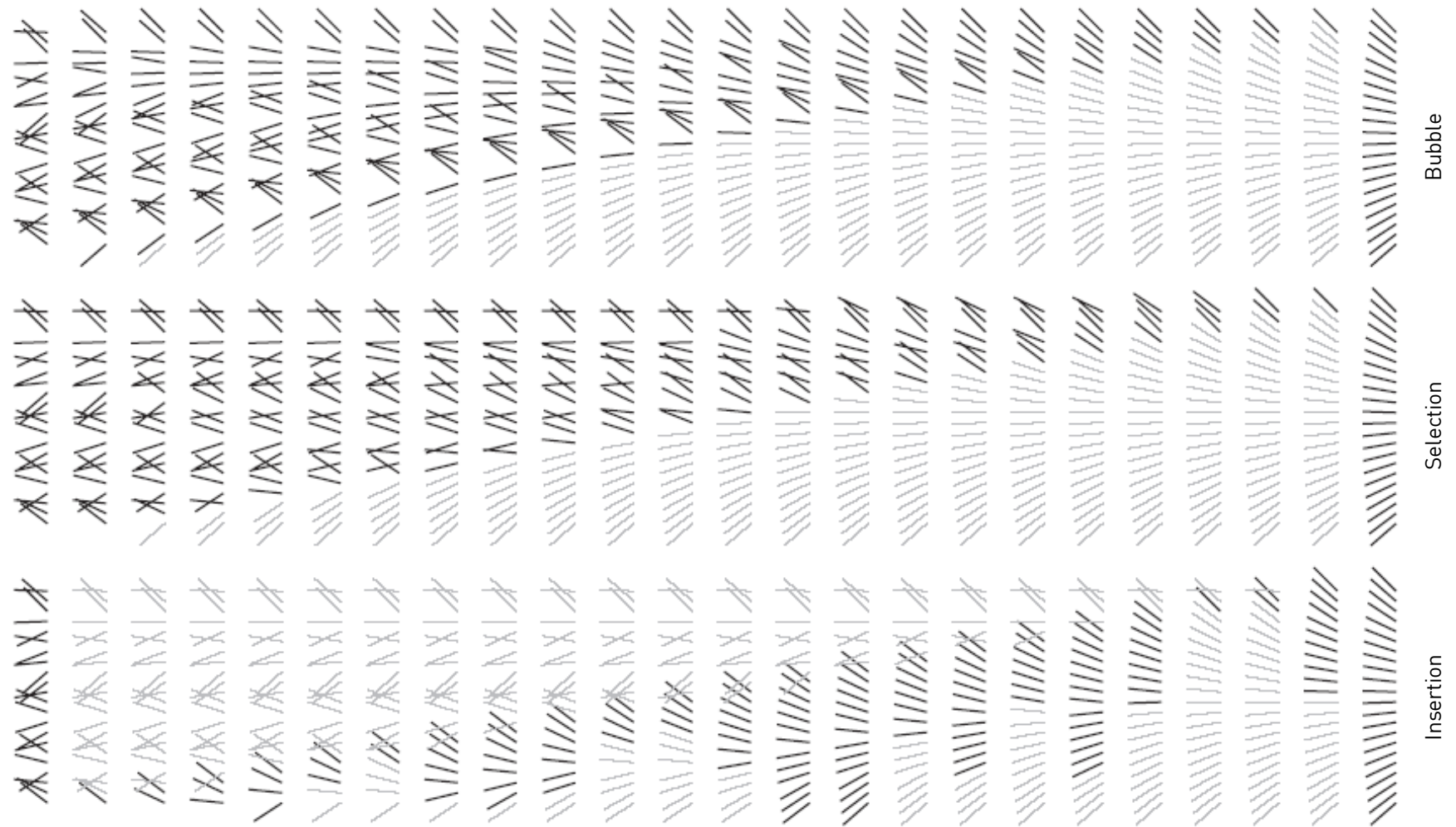
Ein kurzer Blick auf dieses Suchverfahren, das zumeist langsamer als die beiden bisher behandelten Algorithmen läuft.

- ▶ Bubble Sort läuft wie eine Entartung des Sortierens durch direkte Auswahl, mit viel mehr Operationen, um ein Element an seine Position zu bringen.
- ▶ Rudimentärer Quellcode für `int` Werte im `vector`

```
void bubble( vector<int>& vV, int ui, int oi ) {  
    for( int i{ui}; i<oi; ++i )  
        for( int j{oi}; j>i; --j )  
            swap_if( vV.at(j-1), vV.at(j) );  
}
```

Sortieren

Visualisierung der Abläufe bei Bubble Sort, Sortieren durch direkte Auswahl und Sortieren durch direkte Einfügung.



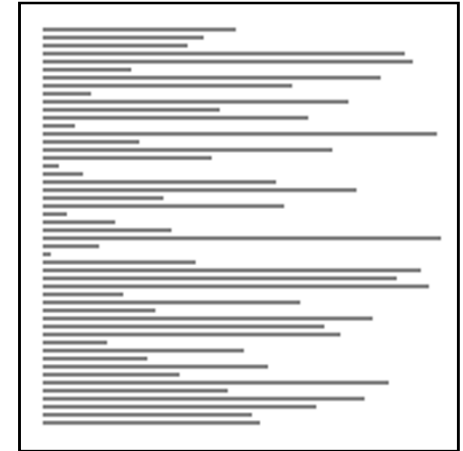
Einfaches Suchen und Sortieren

Inhalt.

- ▶ Zur Algorithmenanalyse
- ▶ Sequentielle Suche
- ▶ Binäre Suche und Interpolationssuche
- ▶ Sortieren durch Auswahl, Sortieren durch Einfügung
- ▶ Zu den Eingangsdaten

Eingangsdaten

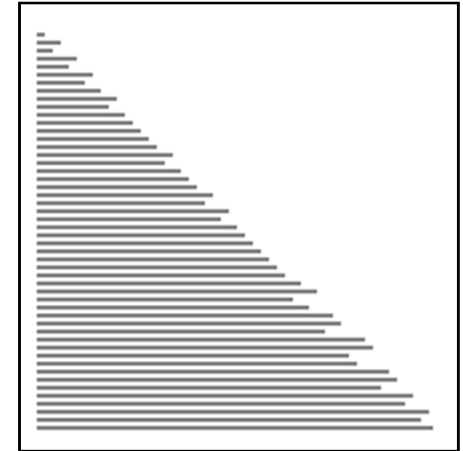
Random initial order: zufällig.



- ▶ Ein völlig zufällig gemischter Anfangszustand der zu sortierenden Daten wird oft verwendet, um Sortieralgorithmen zu bewerten.
- ▶ Gründe
 - Solche Eingangsdaten sollen den typischen Fall darstellen.
 - Die mathematische Analyse ist einfacher.
- ▶ Der erste Grund ist allerdings nicht für alle Anwendungsbereiche zutreffend.
 - Schlussfolgerungen sollten also nicht vorschnell verallgemeinert werden.
- ▶ Bei solchen Eingangsdaten zeigt sich mit wachsendem N deutlich der große Leistungsunterschied zwischen den typisch etwa $O(N^2)$ laufenden, elementaren Sortieralgorithmen (direkte Einfügung, direkte Auswahl) und weiter fortgeschrittenen Verfahren.

Eingangsdaten

Nearly sorted initial order: beinahe sortiert.



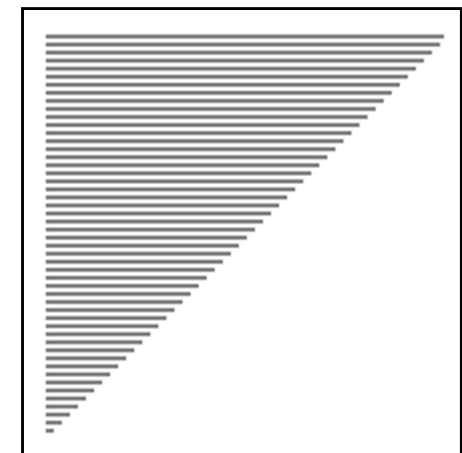
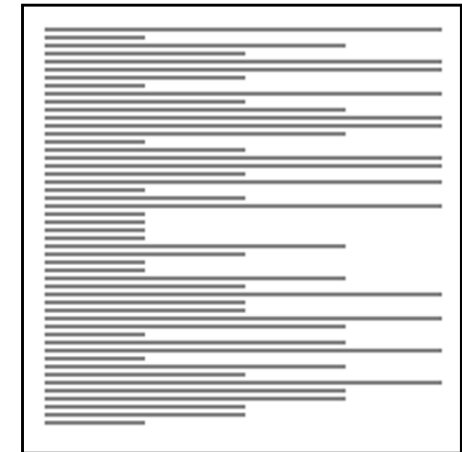
- ▶ Das Sortieren von vorher schon beinahe sortierten Daten kommt in der Praxis häufig vor.
 - Einige Daten aus einer großen Menge werden bearbeitet, wodurch sich die Schlüssel ändern (etwa "Kundenadresse ändern").
 - Einige Daten werden zu einer großen, sortierten Menge hinzugefügt (etwa "Neukunden anlegen").
- ▶ Einige Erkenntnisse für beinahe sortierte Anfangsdaten.
 - Sortieren durch direkte Einfügung ist bei solchen Daten sehr schnell.
 - Im ungünstigsten Fall zwar $O(N^2)$,
 - auf beinahe sortierten Anfangsdaten aber fast $O(N)$.
 - Quicksort (gilt als *das* Universal-Sortierverfahren, wird noch behandelt) ist für beinahe sortierte Daten nicht gut geeignet.
- ▶ Ein für beinahe sortierte Eingangsdaten sehr effizienter Algorithmus ist Smoothsort von E. Dijkstra (soll hier nur erwähnt werden).

Eingangsdaten

Few unique keys: kategorisierte Daten.

Reversed initial order: falsch herum vorsortiert.

- ▶ Eingangsdaten, die zu *wenigen Kategorien* zusammengefasst sind (im gezeigten Beispiel 4), sind in der Praxis recht häufig.
- ▶ Für Quicksort (gilt als *das* Universalverfahren, wird noch behandelt) sind solche Eingangsdaten ungünstig.
- ▶ Das *Umdrehen einer Sortierreihenfolge* ist eine Aufgabe, die beim Programmieren oft vorkommt.
- ▶ Dieses Szenario ist der ungünstigste Fall für das Sortieren durch direkte Einfügung.



Einige Beispielfragen

Einfaches Suchen und Sortieren.

- ▶ Welche *praktische* Relevanz hat die Untersuchung von Algorithmen?
- ▶ Welche zwei grundsätzlichen Herangehensweisen der Untersuchung des Laufzeitverhaltens von Algorithmen kennen Sie?
- ▶ Wie instrumentieren Sie C++ Quellcode zur Messung kurzer Zeitintervalle innerhalb einer einzelnen Ausführung eines Programms? Welche Funktion(en) verwenden Sie? Beschreiben Sie Schritt für Schritt, was zu tun ist.
- ▶ Was geschieht bei der Trennung eines Algorithmus von seiner Implementierung?
- ▶ Welche Vorstellung von der Struktur der Eingangsdaten hat man bei der Analyse der durchschnittlichen Laufzeit eines Algorithmus? Warum?
- ▶ Bringen Sie die folgenden Terme in eine dem Wachstum nach aufsteigende Reihenfolge:

$N \log_2 N$ $\log_2 \log_2 N$ 1 2^N N^2 $\log_2 N$ N^3

Einige Beispielfragen

Einfaches Suchen und Sortieren.

- ▶ Nennen Sie einen Algorithmus mit einer asymptotischen Laufzeit-Obergrenze von logarithmischer Größenordnung.
- ▶ Wie heißt der asymptotische Ausdruck für Laufzeitobergrenzen von Algorithmen, den Sie in der Vorlesung kennen gelernt haben. Geben Sie die Notation an. Welchen Sinn hat dieser asymptotische Ausdruck?
- ▶ $O(f(N)) - O(f(N)) \in O(0)$, stimmt das, und warum bzw. warum nicht (und wie lautet dann die rechte Seite)?
- ▶ Würden Sie 10^7 Millisekunden darauf warten, dass Ihr/e WG-Partner/in morgens im Bad fertig ist?
- ▶ Erklären Sie das Prinzip der sequentiellen Suche mit eigenen Worten.
- ▶ Wie hängt die Laufzeit der sequentiellen Suche von der Struktur der zu durchsuchenden Objekte ab? Erklären Sie den Sachverhalt anhand eines *eigenen* Beispiels.
- ▶ Welches Laufzeitverhalten hat die sequentielle Suche?

Einige Beispielfragen

Einfaches Suchen und Sortieren.

- ▶ Ist die sequentielle Suche für große Datenmengen geeignet? Warum, bzw. warum nicht?
- ▶ Erklären Sie, welches Suchszenario die sequentielle Suche sehr schnell erfolgreich enden lässt und warum das so ist.
- ▶ Warum kann man auf vorsortierten Daten viel schneller suchen als auf zufällig angeordneten Daten?
- ▶ Welcher wesentliche Unterschied besteht zwischen sequentiellen und binären Suchszenarien?
- ▶ Von welcher Größenordnung ist die asymptotische Laufzeit-Obergrenze der binären Suche? Der Interpolationssuche? Was bedeutet das, ist das gut oder nicht so gut? Warum?
- ▶ Welcher wesentliche Unterschied besteht zwischen der binären Suche und der Interpolationssuche?
- ▶ Erklären Sie in eigenen Worten (ohne mathematische Formeln, ohne Quellcode), wie bei der Interpolationssuche der teilende Index bestimmt wird.

Einige Beispielfragen

Einfaches Suchen und Sortieren.

- ▶ Warum lösen unsere Implementierungen der Suchalgorithmen bei erfolgloser Suche keine Ausnahme aus?
- ▶ Welchen *praktischen* Zweck haben Sortialgorithmen?
- ▶ Erklären Sie den Unterschied zwischen internem und externem Sortieren.
- ▶ Erklären Sie den Unterschied zwischen stabilem und nicht stabilem Sortieren.
- ▶ Nennen Sie den nach der Laufzeit zweitwichtigsten Leistungsfaktor von Algorithmen. Welche Typen von Sortialgorithmen lassen sich hierbei unterscheiden?
- ▶ Erklären Sie die Arbeitsweise des Sortierens durch direkte Auswahl mit eigenen Worten.
- ▶ Wie groß ist die maximale und die durchschnittliche Anzahl von Tauschoperationen für ein einzelnes Element beim Sortieren durch direkte Auswahl?

Einige Beispielfragen

Einfaches Suchen und Sortieren.

- ▶ Zeigen Sie Schritt für Schritt, wie Sie LEICHTEAUFGABE durch direkte Auswahl sortieren.
- ▶ Beschreiben Sie den großen praktischen Vorteil, den das Sortieren durch direkte Auswahl hat. Für welche Sortierszenarien ist dieser Vorteil bedeutend?
- ▶ Erklären Sie die Arbeitsweise des Sortierens durch direkte Einfügung mit eigenen Worten.
- ▶ Zeigen Sie Schritt für Schritt, wie Sie LEICHTEAUFGABE durch direktes Einfügen sortiert.
- ▶ Welche entscheidende Änderung hat die in der Vorlesung besprochene Implementierung von `insSort_v1()` im Vergleich zu `insSort_v0()`? Wie genau verbessert diese Änderung die Performance?
- ▶ Welche weiteren Änderungen hat die in der Vorlesung besprochene Implementierung von `insSort_v2()` im Vergleich zu `insSort_v1()`? Wie genau verbessern diese Änderungen die Performance noch weiter?

Einige Beispielfragen

Einfaches Suchen und Sortieren.

- ▶ Beschreiben Sie den praktischen Vorteil, den das Sortieren durch direkte Einfügung hat. Für welche Sortierszenarien ist dieser Vorteil bedeutend?
- ▶ Warum werden zufällig verteilte Eingangsdaten zur Untersuchung des durchschnittlichen Falls der Performance verwendet, obwohl solche Daten in den wenigsten praktischen Fällen vorliegen?
- ▶ Wie beurteilen Sie das Sortieren durch Einfügen mit beinahe sortierten Eingangsdaten? Begründen Sie Ihre Antwort.
- ▶ Welche der rudimentären Implementierungen der beiden Sortierverfahren direkte Auswahl bzw. direkte Einfügung läuft schneller
 - für Daten, in denen alle Schlüssel identisch sind?
 - für genau umgekehrt vorsortierte Daten?

Nächste Einheit:

Grundlagen, Teil (ii)