# ILV Datenstrukturen und Algorithmen

## 09: Laufzeitanalyse, Algorithmentechniken

Stefan Huber

FH Salzburg, Studiengang MMT / 2012

*Licensed under CreativeCommons Attribution-NonCommercial-ShareAlike 3.0*

Kapitel

# Laufzeitanalyse

# Motivation

Algorithmen benötigen zwei Arten von Ressourcen:

- Zeit
- Speicher

Zentrale Fragen in der Informatik:

- Wieviel Zeit/Speicher braucht ein Algorithmus?

Zwei Möglichkeiten:

- Empirische Untersuchung
- Theoretische Analyse

# Empirische Untersuchung

Gegeben sind zwei Algorithmen (resp. Datenstrukturen) für das gleiche Problem. Welcher ist schneller?

Einfacher Ansatz: Man macht Laufzeitmessungen.

Mit welchem Input?

- **Randomisierte Daten:** Liefert Aussagen über die mittlere Laufzeit. Bei großen Eingaben kommen besonders gute oder schlechte Eingaben meist mit hoher Wahrscheinlichkeit nicht vor.
- **Reale Daten für konkrete Anwendungen:** Liefert Aussagen über reale Laufzeiten. Können von den mittleren Laufzeiten deutlich abweichen. Für manche Probleme sind reale Daten schwer zu bekommen.[1]
- **Unnatürliche Daten:** Liefert eventuell Hinweise darauf, wann ein Algorithmus besonders schnell oder langsam ist.
  Mögliche Eingaben für das Sortierproblem: aufsteigend sortierte, absteigend sortierte und konstante Zahlenfolgen.

[1] Beispiel: ein pattern matching Algorithmus zur Erkennung von Krebsbefall von menschlichem Gewebe.

# Empirische Untersuchung: Vergleichbarkeit

Wir wollen Algorithmen bewerten und nicht deren Implementierung, den Programmierer oder die Hardware!

Sind die gewonnen Laufzeitmessungen vergleichbar?

- Ist die eigene Implementierung des eigenen Algorithmus sorgfältiger implementiert und optimiert?
  - Wurde die eigene Implementierung (unbewusst?) für die eigenen Testdaten optimiert?
- Sind die Implementierungen in der gleichen **Programmiersprache** mit den gleichen **Compilern** und den gleichen **Compiler-Optionen** übersetzt worden?
- Sind die Messungen mit einem identischen Computer (Architektur, Prozessortakt, Caches, RAM, Swap), mit einem identischen Betriebssystem und ohne zusätzliche Last durch andere Prozesse durchgeführt worden?
- Wie wurde die Zeit gemessen?
- Wurde mit ausreichend vielen Testdaten getestet?
  - Waren die Testdaten hinreichend interessant?
  - Waren die Testdaten ausreichend groß?

# Empirische Untersuchung: Nachteile

Eine empirische Untersuchung hat diverse Nachteile:

- Algorithmen/Datenstrukturen müssen zunächst implementiert werden.
  - Bei komplexen und umfangreichen Algorithmen hätte man gerne vor der Implementierung einschätzen wollen, was zu erwarten ist.
  - Es gibt theoretische Algorithmen zu denen es keine Implementierung gibt, da die Algorithmen zu komplex sind.
- Empirische Ergebnisse werden durch eine Vielzahl von Parametern beeinflusst.
- Man bekommt keine definitive Auskunft über die günstigsten und ungünstigsten Inputs für einen Algorithmus.
- Aussagekräftige Laufzeitmessungen können vielleicht viele Tage oder Monate dauern.

# Statische Codeanalyse

```
1  int min(int* array, unsigned n)
2  {
3    int min=array[0];
4    unsigned i=1;
5    while( i < n )
6    {
7      if( array[i] < min )
8        min = array[i];
9      i++;
10   }
11   return min;
12 }
```

Wie lange braucht `min`?

- Vom Compiler übersetzen lassen und den Maschinencode disassemblieren.
- Für jede Zeile die Zahl der Vergleiche, Additionen, Speicherzugriffe, Sprünge im Programmcode, . . . im Assembler-Code zählen.
  - Die Zeilen 6–10 werden $(n-1)$-mal durchgeführt, die Zeile 5 sogar $n$-mal.
- Dennoch, eine exakte Vorhersage, wieviele Sekunden etwa die Zeile 3 brauchen wird, ist auf einem modernen System praktisch unmöglich.

# Theoretische Analyse: Einführung

```
1  int min(int* array, unsigned n)
2  {
3    int min=array[0];
4    unsigned i=1;
5    while( i < n )
6    {
7      if( array[i] < min )
8        min = array[i];
9      i++;
10   }
11   return min;
12 }
```

- Wir interessieren uns für die Laufzeit in Abhängigkeit der Eingabegröße $n$.
  - Wenn wir $n$ vergrößern/verkleinern, dann braucht jede Zeile für sich nicht länger oder weniger lang.
  - Bezogen auf $n$ braucht jede einzelne Zeile für sich braucht nur eine konstante Zeit.
  - Ob die Zeile 3 viermal solange braucht wie die Zeile 4 ist irrelevant.
- Insgesamt braucht also `min` mindestens $a_0 + a_1 \cdot n$ und maximal $b_0 + b_1 \cdot n$ Zeiteinheiten für passende Konstanten $a_0, a_1, b_0, b_1$.

# Theoretische Analyse: Einführung

- Problem: teste auf *element uniqueness*, d.h., ob alle Elemente in einem Array verschieden sind.

```
1  bool allUnique(int* array, unsigned n)
2  {
3    unsigned i = 0;
4    bool unique = true;
5    while( i < n ) // runs n-times
6    {
7      j = i+1;
8      while( j < n ) // runs (n-(i+1))-times
9      {
10       if( array[i] == array[j] )
11         unique = false;
12       j = j+1;
13     }
14     i = i+1;
15   }
16   return unique;
17 }
```

# Theoretische Analyse: Einführung

- Die Laufzeit ist für passende Konstanten $a_0, a_1, a_2 \neq 0$ wie folgt:

$$a_0 + \underbrace{\sum_{i=0}^{n-1} \left( a_1 + \sum_{j=i+1}^{n-1} \overbrace{a_2}^{\text{Z. 9–13}} \right)}_{\text{Zeile 6–15}} = a_0 + \sum_{i=0}^{n-1} (a_1 + a_2(n-i-1))$$

$$= a_0 + a_1 n + a_2 \sum_{i=0}^{n-1} (n-i-1)$$

$$= a_0 + a_1 n + a_2 \sum_{i=0}^{n-1} i$$

$$= a_0 + a_1 n + a_2 \frac{(n-1)n}{2}$$

$$= a_0 + \left( a_1 - \frac{a_2}{2} \right) n + \frac{a_2}{2} n^2$$

- Das heißt, es gibt Konstanten $c_0, c_1, c_2$, mit $c_2 \neq 0$, sodass die Laufzeit von folgender Form ist:

$$c_0 + c_1 n + c_2 n^2$$

# Wachstum von Funktionen

Wir haben zu einem Problem zwei Algorithmen mit den Laufzeiten $a_0 + a_1 n + a_2 n^2$ und $b_0 + b_1 n$, wobei $a_2, b_1 > 0$. Wenn $n$ groß genug ist, dann braucht der erste Algorithmus stets mehr Zeit als der zweite:

$$b_0 + b_1 n < a_0 + a_1 n + a_2 n^2$$
$$0 < (a_0 - b_0) + (a_1 - b_1)n + a_2 n^2$$

Es sei $\Delta = (a_1 - b_1)^2 - 4a_2(a_0 - b_0)$. Falls $\Delta < 0$, dann ist die Ungleichung für alle $n$ erfüllt. Ansonsten gilt die Ungleichung für

$$n > \frac{b_1 - a_1 + \sqrt{\Delta}}{2a_2}.$$

# Wachstum von Funktionen

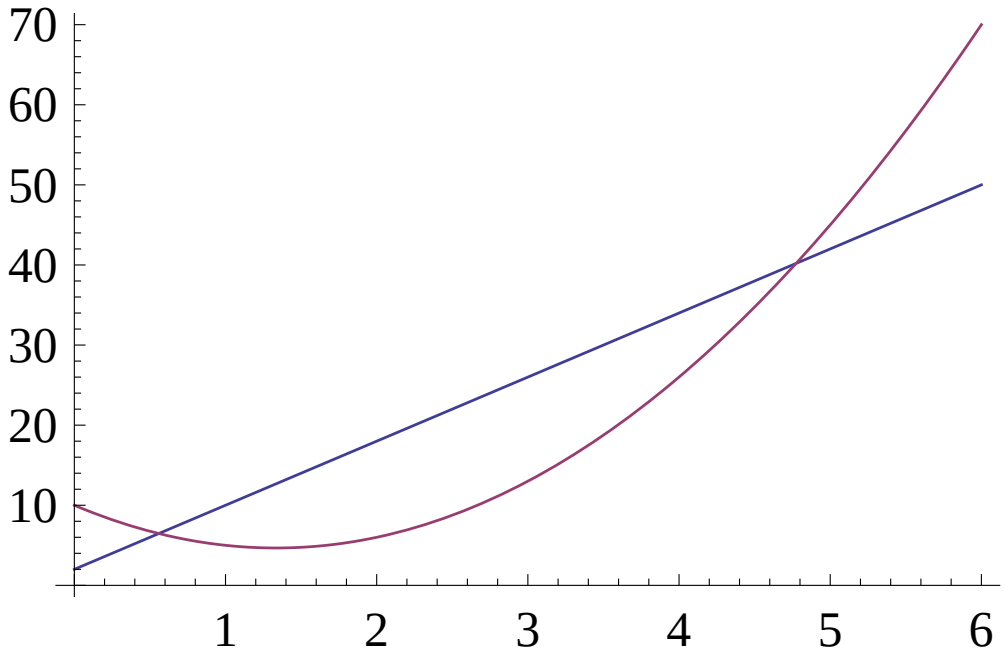



Abbildung : Die Graphen von $2 + 8n$ und $10 - 8n + 3n^2$.

Die quadratische Funktion wächst stärker als die lineare Funktion.

- Konstante Faktoren spielen keine Rolle.
- Wenn $n$ groß genug ist, dann hat die quadratische Funktion stets größere Werte als die lineare Funktion.

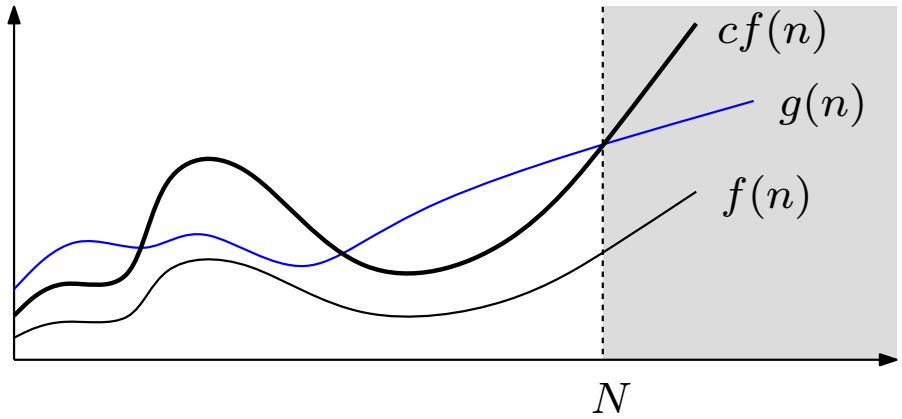
# Big-O Notation

## Definition

Man sagt, die Funktion $g$ ist in $O(f)$, wenn es zwei Konstanten $c$ und $N$ gibt, sodass für alle $n > N$ gilt

$$g(n) \leq c \cdot f(n).$$

Wir schreiben dafür $g \in O(f)$.




- $g \in O(f)$ bedeutet, dass $g$ **höchstens so stark wächst** wie $f$.
- Wenn $g \in O(f)$ und $f \in O(h)$, dann ist auch $g \in O(h)$.

# Big-O Notation

## Lemma

*Für* $f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_k n^k$ *gilt* $f \in O(n^k)$.

*Beweis.* Es gilt $\sum_{i=0}^{k} a_i n^i \leq \sum_{i=0}^{k} |a_i| n^k$. Mit $c = \sum_{i=0}^{k} |a_i|$ und $N = 0$ gilt also $f(n) \leq cn^k$ für alle $n > N$. □

Beispiele:

- $20 + 100n + 5n^2 \in O(n^2)$.
- $10 + 27n \in O(n)$ und insbesondere auch $10 + 27n \in O(n^2)$.
- Merke: Konstante Faktoren spielen keine Rolle!

# Big-O Notation

## Definition

Man sagt, die Funktion $g$ ist in $\Omega(f)$, wenn es zwei Konstanten $c$ und $N$ gibt, sodass für alle $n > N$ gilt

$$g(n) \geq c \cdot f(n).$$

Wir schreiben dafür $g \in \Omega(f)$.

- $g \in \Omega(f)$ bedeutet, dass $g$ **mindestens so stark wächst** wie $f$.
- Wenn $g \in \Omega(f)$ und $f \in \Omega(h)$, dann ist auch $g \in \Omega(h)$.

## Definition

Man sagt, die Funktion $g$ ist in $\Theta(f)$, wenn $g \in O(f)$ und $g \in \Omega(f)$. Wir schreiben dafür $g \in \Theta(f)$.

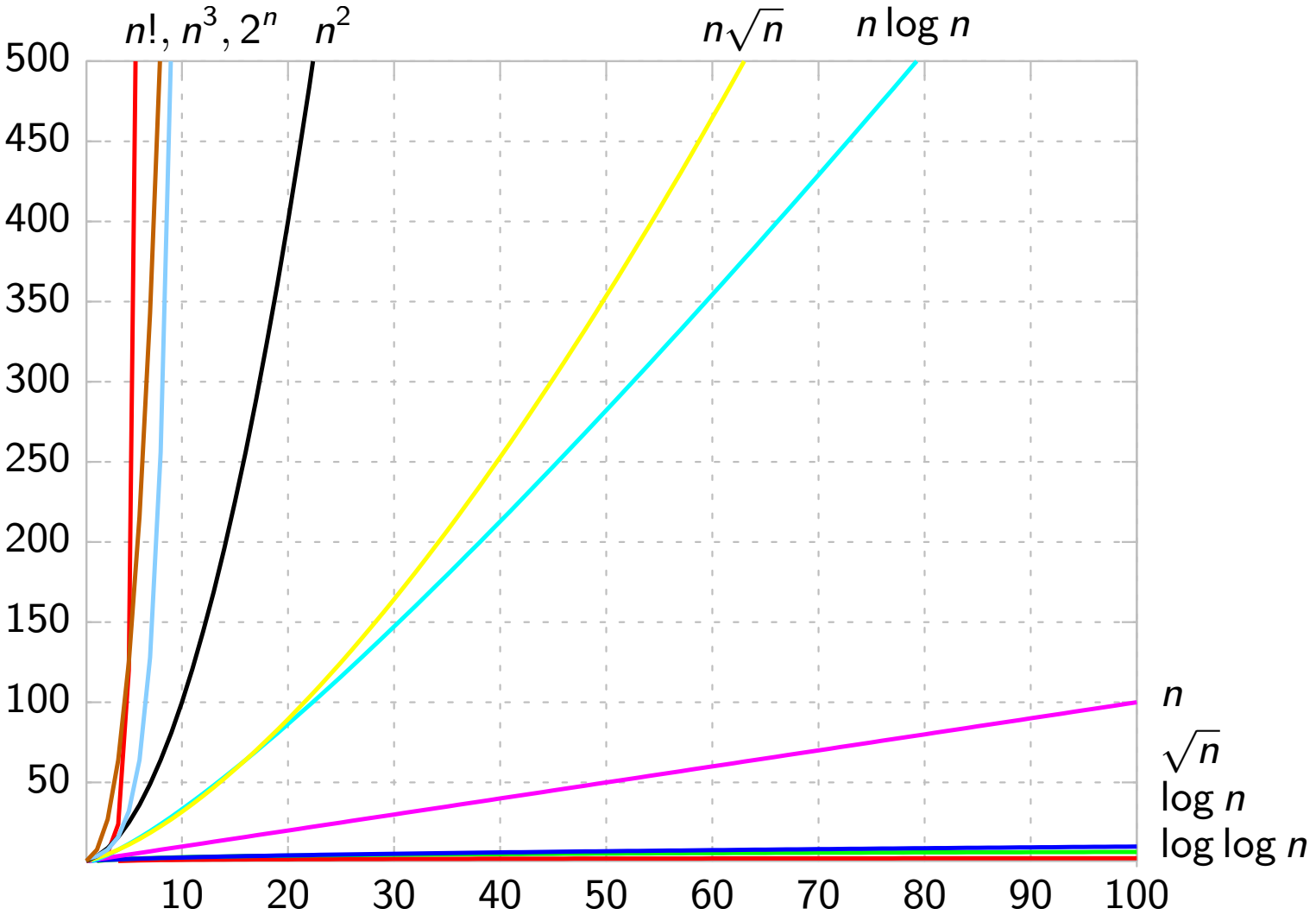
- $g \in \Theta(f)$ bedeutet, dass $g$ **genauso stark wächst** wie $f$.
- Wenn $g \in \Theta(f)$ und $f \in \Theta(h)$, dann ist auch $g \in \Theta(h)$.

# Big-O Notation

- $1 + 20n + 40n^2 \in \Omega(n^2)$ und $1 + 20n + 40n^2 \in \Omega(n)$.
- $1 + 20n + 40n^2$ ist in $\Theta(n^2)$, aber nicht in $\Theta(n)$ oder $\Theta(n^3)$.
- $O(g) \subseteq O(h)$ heißt, dass für jede Funktion $f$ mit $f \in O(g)$ auch $f \in O(h)$ gilt.
- $O(g) = O(h)$ heißt, dass $O(g) \subseteq O(h)$ und $O(h) \subseteq O(g)$.

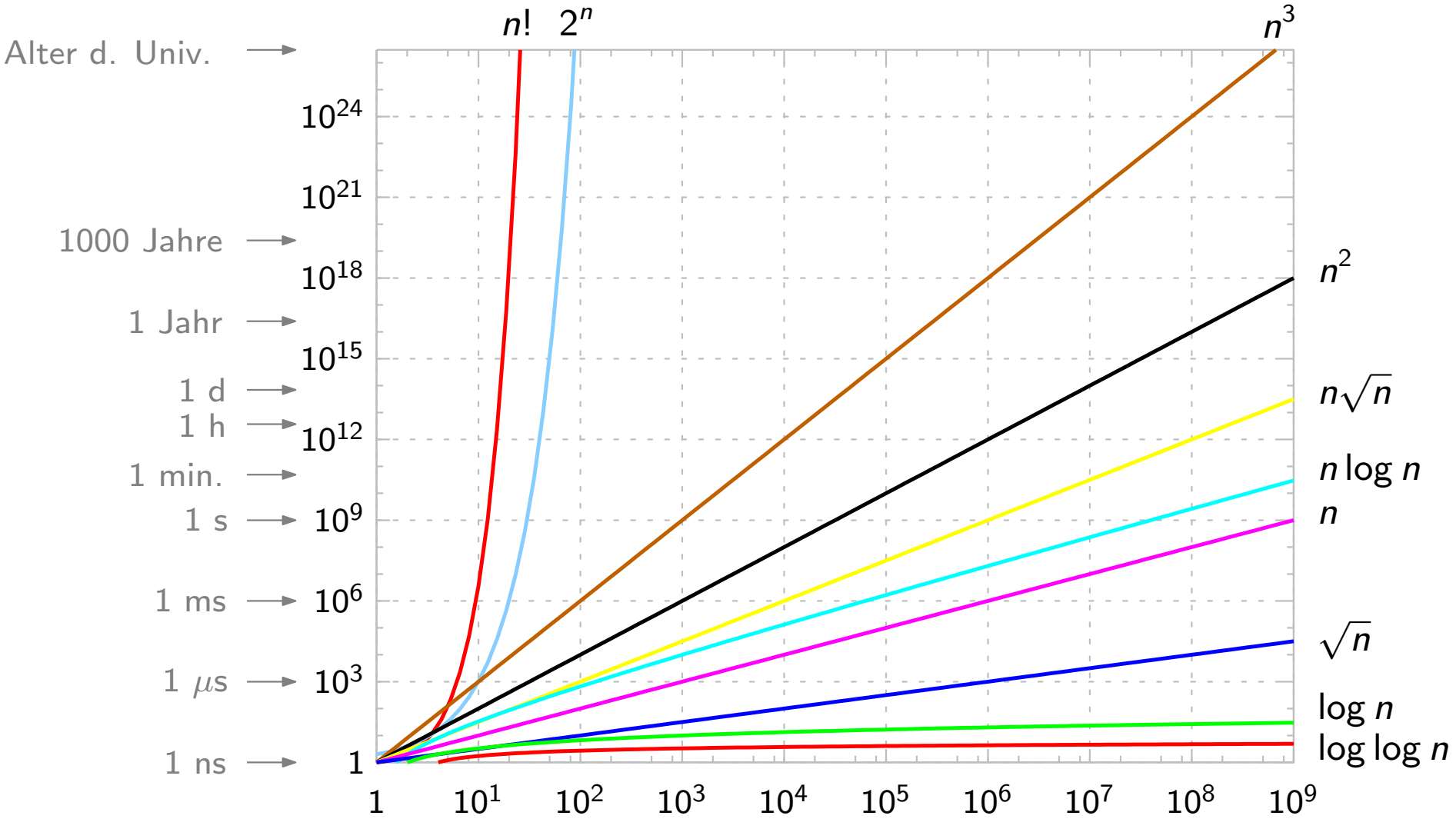
$$O(1) \subseteq O(\log\log n) \subseteq O(\log n) \subseteq O(\sqrt{n}) \subseteq O(n) \subseteq O(n \log n) \subseteq$$
$$\subseteq O(n\sqrt{n}) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n) \subseteq O(n!)$$

# Komplexitäten



# Komplexitäten

Bei $10^9$ Operationen pro Sekunde



# Best case, average case, worst case

Das Konzept der O-Notation wird verwendet um die Laufzeit von Algorithmen zu beschreiben. Dabei werden oft folgende Fälle unterschieden:

- **Best case**: Wie schnell läuft der Algorithmus für die günstigste Eingabe?
- **Worst case**: Wie schnell läuft der Algorithmus für die ungünstigste Eingabe?
- **Average case**: Wie schnell läuft der Algorithmus durchschnittlich?
- **Expected runtime**: Was ist der Erwartungswert der Laufzeit? (Bei randomisierten Algorithmen von Interesse; oder wenn man eine Wahrscheinlichkeitsverteilung für die Eingabe annimmt.)

- Läuft ein Algorithmus im worst case in $O(f)$ Zeit, dann läuft er in jedem Fall in $O(f)$ Zeit.
- Läuft ein Algorithmus im best case in $\Omega(f)$ Zeit, dann läuft der Algorithmus in jedem Fall in $\Omega(f)$ Zeit.
- Ein Algorithmus läuft genau dann in $\Theta(f)$ Zeit, wenn er im best-case in $\Omega(f)$ Zeit und im worst-case in $O(f)$ Zeit läuft.

# Element uniqueness

```
1  bool allUnique(int* array, unsigned n)
2  {
3    unsigned i = 0;
4    bool unique = true;
5    while( i < n ) // runs n-times
6    {
7      j = i+1;
8      while( j < n ) // runs (n-(i+1))-times
9      {
10       if( array[i] == array[j] )
11         unique = false;
12       j = j+1;
13     }
14     i = i+1;
15   }
16   return unique;
17 }
```

- Worst-case Laufzeit: $O(n^2)$, sogar in $\Theta(n^2)$
- Best-case Laufzeit: $\Omega(n^2)$, sogar in $\Theta(n^2)$
- In jedem Fall also in $\Theta(n^2)$.

# Element uniqueness 2

```
1  bool allUnique(int* array, unsigned n)
2  {
3    unsigned i = 0;
4    bool unique = true;
5    while( i < n && unique ) // runs at most n-times
6    {
7      j = i+1;
8      while( j < n && unique ) // runs at most (n-(i+1))-times
9      {
10       if( array[i] == array[j] )
11         unique = false;
12       j = j+1;
13     }
14     i = i+1;
15   }
16   return unique;
17 }
```

- Worst-case Laufzeit: $O(n^2)$, sogar in $\Theta(n^2)$
- Best-case Laufzeit: $\Omega(1)$

# Binäre Suche

Problem: ist die Zahl $k$ in einem sortierten Array der Größe $n$ enthalten?

```
1  bool binary_search(int* array, unsigned n, int k)
2  {
3    int l = 0, r=n-1;
4    while( l<=r )
5    {
6      const int m = l + (r-l)/2;
7      if( array[m] == k )
8        return true;
9      if( array[m] < k )
10       l = m+1;
11     if( array[m] > k )
12       r = m-1;
13   }
14   return false;
15 }
```

- Im ersten Schleifendurchlauf ist $r - l = n - 1$. Mit jedem weiteren Schleifendurchlauf wird $r - l$ mehr als halbiert.
- Laufzeit: $O(\log n)$
  - Best case: $O(1)$
  - Worst case: $\Theta(\log n)$

# Laufzeiten von Listen und Arrays

$n$ sei die Größe der folgenden Datenstrukturen.

Doppelt verkettete Listen:

- `push_back`, `push_front`, `pop_back`, `pop_front`: $O(1)$
- finde $i$-tes Element: $O(n)$

Einfach verkettete Listen:

- `push_front`, `pop_front`: $O(1)$
- finde $i$-tes Element: $O(n)$

Array:

- `push_front`, `pop_front`: $\Theta(n)$
- `push_back`: $O(n)$
- `pop_back`: $O(1)$
- finde $i$-tes Element: $O(1)$

# Amortisierte Laufzeit

- Wir betrachten ein dynamisches Array mit Verdopplungsstrategie:
  - Wenn `push_back` auf ein volles Array aufgerufen wird, dann wird die doppelte Größe reserviert und das alte Array kopiert.
- `push_back` hat folgende Laufzeiten:
  - best case: $O(1)$, wenn noch Platz ist.
  - worst case: $\Theta(n)$, wenn das Array umkopiert werden muss.
- Wieviel Zeit beansprucht das Anfügen von $m$ Elementen in ein Array mit $n$ Elementen? Es sei $N := n + m$ die endgültige Größe.
  - Es kostet jedenfalls nicht mehr als $O(N^2)$ Zeit.
  - Aber: nur bei $O(\log N)$-vielen Schritten müssen wir verdoppeln. Also brauchen wir nicht mehr als $O(N + N \log N) = O(N \log N)$ Zeit.
  - Können wir eine noch bessere Schranke finden?

# Amortisierte Laufzeit

- Wir brauchen höchstens so viel Zeit, als wenn wir ein leeres Array mit $N$ Elementen befüllt hätten.
  - Wie lange dauert das Füllen eines leeren Arrays mit $N$ Elementen?
  - Im worst case ist $N$ von der Form $2^k + 1$, dann benötigen wir folgende Laufzeit für das Kopieren durch Verdoppelungen:

$$O(1 + 2 + 4 + \cdots + 2^k) = O(2^{k+1} - 1) = O(2 \cdot 2^k - 1) = O(N)$$

- Um also $m$ Elemente an ein Array mit $n$ Elementen anzufügen brauchen wir $O(n + m)$ Zeit. Wenn $m \in \Omega(n)$, wenn wir also mindestens linear viele Elemente anfügen, dann kostet ein Anfügen im Durchschnitt folgende Zeit:

$$\frac{1}{m} O(n + m) = O\left(1 + \frac{n}{m}\right) = O(1)$$

- Man sagt: **amortisiert** kostet das Anfügen eines Elements an ein Array $O(1)$ Zeit.
  - Umgekehrt: Befüllen wir ein leeres Array mit $n$ Elementen, dann kostet das $O(n)$ Zeit, da das Anfügen eines Elements amortisiert in $O(1)$ Zeit geht.
- Amortisierte Laufzeiten eines Algorithmus spielen also eine Rolle, wenn dieser oft aufgerufen wird.

Kapitel

# Algorithmentechniken

# Divide and conquer

- Rekursive Algorithmen beruhen auf dem Prinzip, dass ein Teil des Problems von gleicher Gestalt ist.
  - Traversieren von Bäumen, Berechnen von rekursiven Funktionen, ...
- Dieses Prinzip heißt **divide and conquer** (teile und herrsche) und kann als Leitfaden zur Herleitung von Algorithmen verwendet werden.

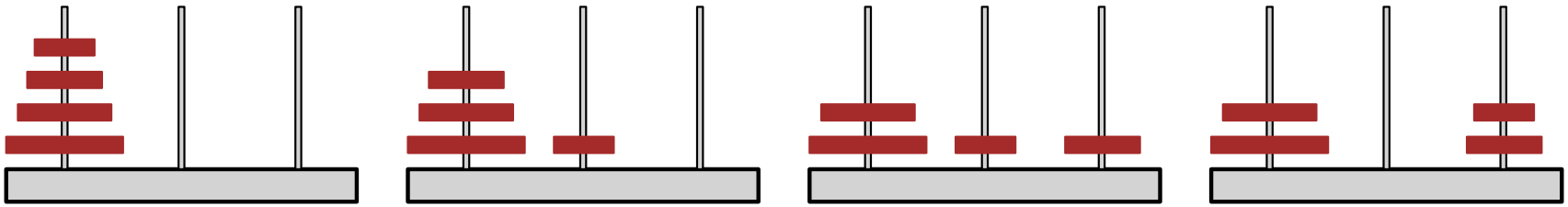
Divide and conquer Prinzip:

1. Teile das Problem in gleichgestaltige Teilprobleme.
2. Löse die Teilprobleme rekursiv.
3. Füge die Teillösungen zu einer Lösung des Gesamtproblems zusammen.

# Divide and conquer: Towers of Hanoi

Problemstellung:

- Ein Stapel mit $n$ der Größe nach sortierten Scheiben liegt auf der ersten Stange und soll auf die dritte Stange verschoben werden.
- Gültige Züge: eine Scheibe, die ganz oben liegt, kann auf einen anderen Stapel gelegt werden, wenn alle darunter liegenden Scheibe größer sind.

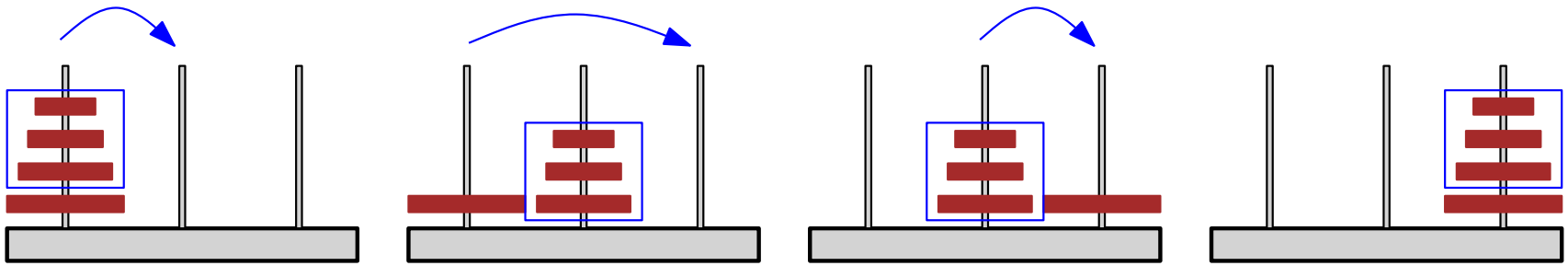


Wie lautet ein Algorithmus, welcher die zu spielenden Züge ausgibt?

# Divide and conquer: Towers of Hanoi

Divide:

- Wenn wir einen Turm mit $n-1$ Scheiben verschieben könnten, dann könnten wir wie folgt vorgehen:
  - Man verschiebt die $n-1$ oberen Scheiben auf die 2. Stange (Hilfsstange).
  - Man verschiebt die $n$-te Scheibe auf die 3. Stange (Zielstange).
  - Man verschiebt die $n-1$ Scheiben von der Hilfsstange auf die Zielstange.



Conquer:

- Doch wie verschiebt man die $n-1$ Scheiben?
  - Genau gleich!
  - Die oberen $n-2$ Scheiben auf eine Hilfsstange.
  - Die größte Scheibe von den $n-1$ Scheiben auf die Zielstange.
  - Die $n-2$ Scheiben von der Hilfsstange auf die Zielstange.

# Divide and conquer: Towers of Hanoi

```
1  void hanoi(unsigned n, unsigned src, unsigned dest)
2  {
3    if( n==0 )
4      return;
5
6    unsigned help=0;
7    while( help==src || help==dest)
8      help++;
9
10   hanoi(n-1, src, help);
11   cout << "move top disk from " << src << " to " << dest << endl;
12   hanoi(n-1, help, dest);
13 }
```

Beachte:

- Beim Aufruf von `hanoi` werden stets Teil-Türme mit den kleinsten Scheiben verschoben. Deshalb müssen wir nicht darauf achten, dass wir eine größere Scheibe auf eine kleinere geben.

# Divide and conquer: Towers of Hanoi

Wie effizient ist der Algorithmus?

- Sei $f(n)$ die Zahl der Verschiebeoperationen:

$$\begin{aligned} f(n) &= 2f(n-1)+1 \\ &= 2(2f(n-2)+1)+1 = 4f(n-2)+3 \\ &= 4(2f(n-3)+1)+3 = 8f(n-3)+7 \\ &= 8(2f(n-4)+1)+7 = 16f(n-4)+15 \\ &= \dots \\ &= 2^n f(0) + 2^n - 1 = 2^n - 1 \end{aligned}$$

- Die Laufzeit des gesamten Algorithmus ist in $\Theta(f(n))$, also in $\Theta(2^n)$.

# Dynamisches Programmieren

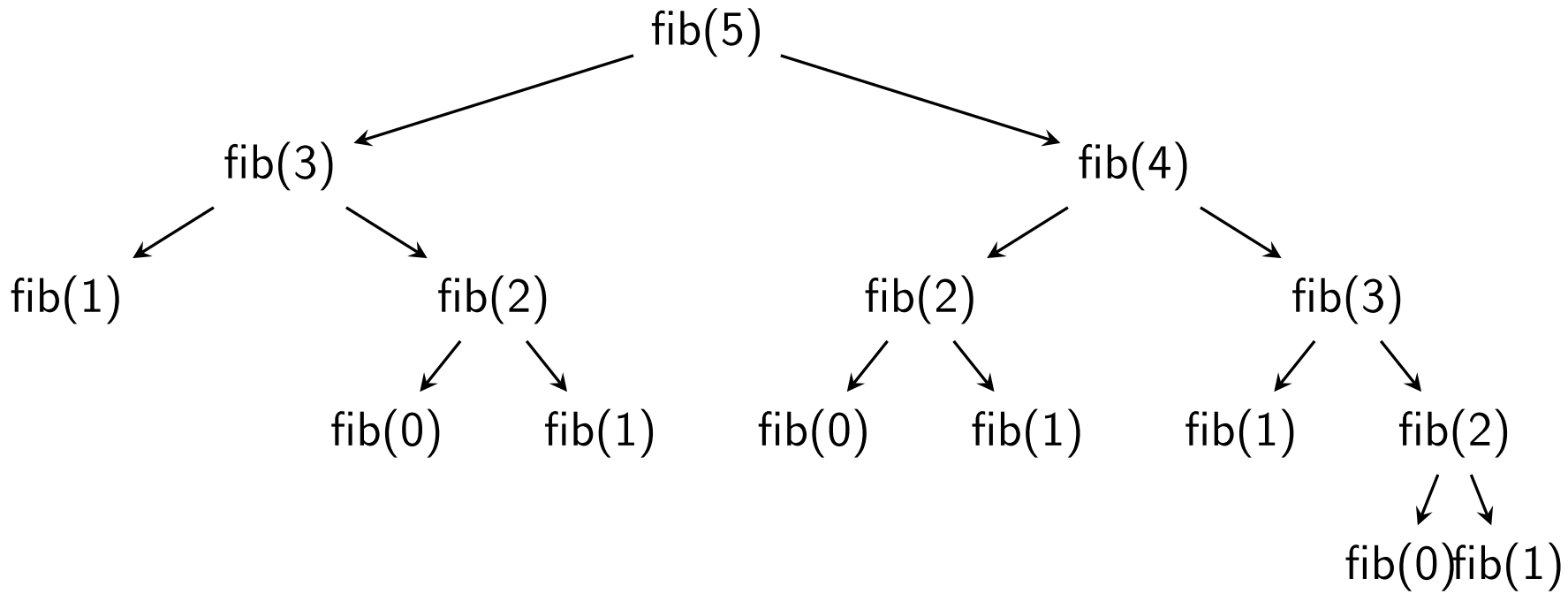
- Rekursive Algorithmen führen oft zu sehr einfachen aber ineffizienten Algorithmen.
- Beispiel: Die Berechnung der $n$-ten Fibonaccizahl $F_n$:

```
1 unsigned fib(unsigned n)
2 {
3   if( n<1 )
4     return 0;
5   if( n==1 )
6     return 1;
7   return fib(n-1) + fib(n-2);
8 }
```

- Wie lange braucht `fib(n)` zum Rechnen?
  - Die Anzahl der rekursiven Aufrufe für `fib(n)` ist $F_{n+1}$.
  - $F_n \approx \frac{1}{\sqrt{5}} \left( \frac{1+\sqrt{5}}{2} \right)^n$
  - Das heißt, dieser Algorithmus läuft in exponentieller Zeit!

# Dynamisches Programmieren

- `fib(5)` führt zu folgendem Aufrufbaum:




- Viele Teilergebnisse werden immer wieder neu berechnet!
- Idee: speichere Lösungen zu Teilproblemen und berechne `fib(n)` nur, wenn dieses Rechnung noch nicht durchgeführt wurde.

# Dynamisches Programmieren: bottom-up

- Hier wird jedes Teilergebnis nur einmal berechnet.
- Dieser Algorithmus läuft in $\Theta(n)$ Zeit!
- Da jeweils nur die letzten beiden Teilergebnisse gebraucht werden könnte man sich das Array gleich sparen. (Siehe iterative Lösung in den alten Slides.)

```
1  struct Fib
2  {
3    unsigned fib[100]; // fib(99) > 2^64
4    unsigned compute(unsigned n)
5    {
6      assert( n < 100 );
7      fib[0] = 0;
8      fib[1] = 1;
9      for(unsigned i=2; i<=n; ++i)
10       fib[i] = fib[i-1] + fib[i-2];
11     return fib[n];
12   }
13 };
```

- Man nennt diese Technik **bottom-up dynamic programming**:
  - Berechne sukzessive Lösungen von größeren Problemen indem man die gespeicherten Resultate der Teilprobleme verwendet.

# Dynamisches Programmieren: top-down

- **Top-down dynamic programming (a.k.a. memoisation)**:
  - Im rekursiven Aufruf wird geprüft, ob eine Lösung für dieses Teilproblem schon berechnet wurde und ggf. dieses zurückgeliefert.
  - Im Gegensatz zum zum bottom-up Ansatz werden nur jene Teilergebnisse berechnet, die auch gebraucht werden.

```
1  struct Fib
2  {
3    unsigned fib[100]; bool known[100];
4    Fib() {
5      known[0] = known[1] = true;
6      fib[0] = 0; fib[1] = 1;
7      for(unsigned i=2; i<100; ++i)
8        known[i] = false;
9    }
10   unsigned compute(unsigned n) { // Runs in O(n) time.
11     if( !known[n] ) {
12       fib[n] = compute(n-1) + compute(n-2);
13       known[n] = true;
14     }
15     return fib[n];
16   }
17 };
```

# Dynamisches Programmieren

- Bottom-up läuft im Aufrufbaum der Funktionsaufrufe von den Blättern zur Wurzel.
- Top-down läuft im Aufrufbaum von der Wurzel zu den Blättern.

# Knapsack Problem

Problem:

- Ein Dieb hat einen Rucksack, in dem Wertgegenstände mit einer Gesamtgröße von $M$ passen.
- In einem Tresor befinden sich Gegenstände von $n$ Arten. Jede Art hat einen bestimmten Werten und eine bestimmte Größe und von jeder Art sind beliebig viele Gegenstände vorhanden.
- Wieviele Gegenstände soll er von jeder Art mitnehmen, sodass sein Rucksack maximalen Gesamtwert erreicht?

Beispiel:

- Rucksack hat die Größe 3. Folgende Arten von Gegenständen stehen zur verfügung:

| Gegenstand | A | B | D |
|---|---|---|---|
| Größe | 2 | 1.4 | 1 |
| Wert | 5 | 3 | 0.5 |

- Lösung: zweimal Gegenstand B bringt den maximalen Gewinn.
  - Achtung: Das beste Preis/Gewicht-Verhältnis hätte A!

# Knapsack Problem

Algorithmus A:

- Zähle alle möglichen Kombinationen auf, wie man von jeder Art zwischen 0 und einer Maximalzahl von Gegenständen wählen kann. Wähle jene Kombination, die in den Rucksack passt und maximalen Gesamtwert hat.
- Der Algorithmus braucht exponentielle Laufzeit, da es mindestens $2^n$ Möglichkeiten gibt. (Unter der Annahme, dass der Rucksack überhaupt jede Art von Gegenstand unterbringen kann. Ansonsten kann man diese Art gleich ignorieren.)

Algorithmus B:

- Verwende dynamische Programmierung.
  - Hat das Knapsack Problem Teilprobleme gleicher Gestalt?
  - Wenn wir einen Gegenstand vom Typ $i$ wählen, dann können wir rekursiv den Rucksack mit der Restgröße optimal füllen lassen.
  - Wir probieren alle Gegenstände durch und prüfen bei welchem wir zusammen mit der optimalen Teillösung für den Restrucksack auf maximalen Profit kommen.
  - Das gibt uns eine optimale Füllung des Gesamtrucksacks.

# Knapsack problem: Divide and Conquer

```
1  /** Return value of optimal knapsack of size 'cap'. We are given
2   * n item types with values v[i] and sizes s[i].
3   * If you want to know how the knapsack is filled, return the
4   * content of the knapsack, too.
5   * */
6  int knapsack(int cap, int v[], int s[], int n)
7  {
8    int max = 0;
9    for(int i=0; i<n; ++i) {
10     if( cap >= s[i] ) {
11       const int restcap = cap - s[i];
12       const int value = v[i] + knapsack(restcap, v, s, n);
13       if( value > max )
14         max = value;
15     }
16   }
17   return max;
18 }
```

- Einfache Lösung, aber exponentielle Laufzeit.
  - Rekursive Version von Algorithmus A.
  - `knapsack` wird für die gleichen Größen immer wieder aufgerufen → dynamic programming löst das Problem!

# Knapsack problem: top-down DP

```
1  struct Knapsack
2  {
3    int n, *v, *s; // n item types, with values v[i] and sizes s[i]
4    bool known[maxcap+1]; int max[maxcap+1]; // Memoized results
5
6    Knapsack(int maxcap, int v[], int s[], int n) {...}
7
8    int compute(int cap)
9    {
10     if( !known[cap] ) { // Do not compute results we already know
11       max[cap] = 0; known[cap] = true;
12       for(int i=0; i<n; ++i) { // Determine correct value
13         if( cap >= s[i] ) { // Knapsack can be filled with item i
14           const int restcap = cap - s[i];  //Remaining capacity
15           const int value = v[i] + compute(restcap);
16           if( value > max[cap] )
17             max[cap] = value; // Better solution
18         }
19       }
20     }
21     return max[cap]; // Now max[cap] contains the correct value
22   }
23 };
```

# Knapsack problem: top-down DP

- Im worst case haben wir für jede der $O(M)$ Teil-Rucksackgrößen $O(n)$ Aufwand in der Schleife, da nie eine Rucksackgröße zweimal behandelt wird.
- Also bekommen wir eine $O(n \cdot M)$ Laufzeit.
- Beachte: Manche Rucksackgrößen werden evtl. nie betrachtet!
- Natürlich hätten wir auch bottom-up dynamic programming anwenden können:
  - Dann hätten wir für *alle* Rucksackgrößen in aufsteigender Reihenfolge die optimalen Lösungen bestimmt.
  - Das führt zu einem $\Theta(n \cdot M)$ Aufwand.
  - Im Allgemeinen ist der top-down Ansatz schneller. Im worst-case sind die beiden Ansätze gleich schnell.

# Knapsack problem: bottom-up DP

```
1  struct Knapsack
2  {
3    int n, *v, *s; // n item types, with values v[i] and sizes s[i]
4    int max[maxcap+1];
5
6    Knapsack(int maxcap, int v[], int s[], int n) {...}
7
8    int compute(int cap)
9    {
10     for(unsigned c=0; c<=cap; ++c) // Run through all sizes
11     {
12       max[c] = 0; // Determine optimal knapsack of size c
13       for(int i=0; i<n; ++i) {
14         if( c >= s[i] ) {
15           const int restcap = c - s[i]; // Remaining capacity
16           const int value = v[i] + max[restcap];
17           if( value > max[c] )
18             max[c] = value; // Better solution
19         }
20       }
21     }
22     return max[cap];
23   }
24 };
```

# Varianten des Knapsack Problems

- Gegeben sei ein Fracht-Container und Gegenstände von bestimmter Größe. Wie soll der Container optimal befüllt werden?
- Gegeben seien Münzgrößen. Wie kann man mit möglichst wenig Münzen auf einen gegebenen Betrag kommen?