

Optimierung

- ▶ Optimierungsansätze:
 1. Algorithmische/mathematische Optimierungen
 2. Wahl der Programmiersprache
 3. Compiler-spezifische Optimierungen
 4. Hardware-spezifische Optimierungen
- ▶ Optimierungsziele: Laufzeit – aber auch: Speicherplatz, etc.

Optimierung - Tradeoffs

- ▶ Optimierter Code ist meist
 - ▶ aufwändiger zu schreiben
 - ▶ schwerer zu lesen/warten
 - ▶ komplizierter zu testen/debuggen

- ▶ *Nur performanzkritischen Code optimieren!*

Fibonacci-Reihe Definition

$$f_n = f_{n-1} + f_{n-2}, \quad \text{mit } f_0 = 0, f_1 = 1$$

	f_0	f_1	f_2	f_3	f_4	f_5	f_6			
$f_0 =$	0									
$f_1 =$		1								
$f_2 =$	0	+	1	=	1					
$f_3 =$			1	+	1	=	2			
$f_4 =$				1	+	2	=	3		
$f_5 =$					2	+	3	=	5	
$f_6 =$						3	+	5	=	8

Fibonacci: Rekursiv

```
1 #include <stdint.h>
2
3 uint64_t fib1(uint64_t n) {
4     if (n <= 1) {
5         return n;    // fib(0) = 0, fib(1) = 1
6     }
7
8     return fib1(n - 1) + fib1(n - 2);
9 }
```

Laufzeit Fibonacci (Rekursiv)

n	Laufzeit
0 ... 39	< 0.50s
40	0.55s
42	1.42s
44	3.73s
46	9.74s
48	26.00s
50	67.49s
52	> 2 min
53 ... 93	sehr lange

Quiz: Laufzeit Fibonacci (Rekursiv)

Warum steigt die Laufzeit dieser Implementierung so schnell?

Die Lösung verbraucht noch zu viele Zeilen Code

Es wird mit `uint64_t` statt mit `int64_t` Werten gerechnet

Aufgrund der Berechnung mittels doppelter Rekursion

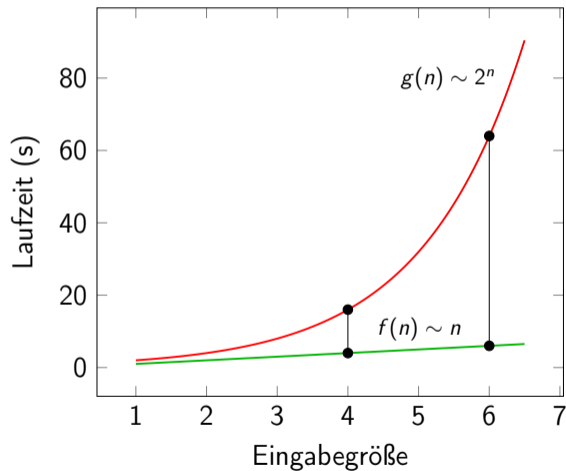
Das Programm wurde in C und nicht in Assembly geschrieben

Laufzeitklassen

- ▶ (Komplexe) Laufzeit eines Algorithmus: $f(n)$
- ▶ $f(n)$ wächst vergleichbar zu einer "simplen" Funktion $K(n)$
 - ▶ $K(n)$ ist Laufzeitklasse des Algorithmus

$K(n)$	Laufzeitklasse
2^n	exponentiell
n^2	quadratisch
n	linear
$\log n$	logarithmisch
1	konstant

Laufzeitklassen - n vs 2^n



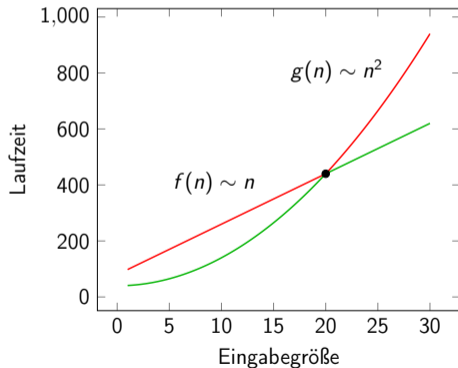
n	$f(n)$	$g(n)$
1	1s	2s
2	2s	4s
4	4s	16s
6	6s	64s
8	8s	4min
10	10s	17min
15	15s	9h
20	20s	12d

Optimierungen - Laufzeitklassen

- ▶ Laufzeitklasse des Algorithmus entscheidend
 - ▶ Erst Laufzeitkomplexität optimieren!
- ▶ Andere Optimierungen zunächst unnötig
 - ▶ Insb. von frühzeitigen Mikrooptimierungen absehen!

Optimierung für kleine Eingabewerte

- ▶ Schlechtere Laufzeitklassen möglicherweise schneller
 - ▶ Konstante Faktoren und Offsets ausschlaggebend
- ▶ Muss *individuell* getestet werden



Quiz: Laufzeitklassen (1)

Welche der folgenden Optimierungsmöglichkeiten sollte in der Regel zuerst betrachtet werden?

Speicherzugriffe und Cacheverhalten

Die Anzahl verwendeter Variablen

Die Auswahl des Algorithmus (Verbesserung der Laufzeitklasse)

Die Länge des Codes

Quiz: Laufzeitklassen (2)

Was haben (frühzeitige) Mikrooptimierungen meist zur Auswirkung?

Der Code wird schwerer zu lesen und zu warten

Die Laufzeit des Algorithmus verbessert sich um mehrere Größenordnungen

Der Code wird fehleranfälliger und komplizierter zu debuggen

Der Code wirkt professioneller, was wiederum zu einer Gehaltserhöhung führt

Quiz: Laufzeitklassen (3)

Was gilt für zwei Algorithmen A und B, wobei sich Algorithmus A in einer besseren Laufzeitklasse befindet als Algorithmus B?

Algorithmus A braucht für alle Eingabewerte weniger Zeit

Algorithmus A braucht vor allem für große Eingabewerte weniger Zeit

Für kleine Eingabewerte kann Algorithmus B schneller sein

In der Praxis kann die Verwendung von Algorithmus B oft ausreichend sein

Fibonacci: Lineare Schleife

► Doppelte Rekursion (exponentiell) → Lineare Schleife

```
1 uint64_t fib2(uint64_t n) {
2     if (n == 0) {
3         return 0;
4     }
5     if (n > 93) {
6         return UINT64_MAX;
7     }
8
9     uint64_t a = 0;
10    uint64_t b = 1;
11    ...
```

```
11    ...
12    uint64_t i = 1;
13    for (; i < n; i++) {
14        uint64_t tmp = b;
15        b += a;
16        a = tmp;
17    }
18
19    return b;
20 }
```

Ausblick: Formel von Binet

$$\text{fib}(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right)$$

- + Logarithmische Laufzeit (mit schneller Exponentiation)
- Fließkommazahlen mit begrenzten Nachkommastellen
 - ▶ Genauigkeitsverluste

Optimierung mittels Lookuptabelle (LUT)

- ▶ Nur 94 Fibonaccizahlen mit `uint64_t` darstellbar
- ▶ Vorberechnung der Zahlen mit implementiertem Algorithmus
 - ▶ Speichern in Lookuptabelle (LUT)
- ▶ Algorithmus schlägt Werte einfach in LUT nach

Fibonacci: LUT

```
1 // All 94 64-bit fibonacci numbers (n = 0,...,93)
2 uint64_t lut[] = {
3     0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,...,
4     7540113804746346429,12200160415121876738};
5
6 uint64_t fib3(uint64_t n) {
7     if (n > 93) {
8         return UINT64_MAX;
9     }
10
11     return lut[n];
12 }
```

Quiz: Optimierung mittels LUTs

Wann sind LUTs zur Optimierung meist gut geeignet?

Für eine bekannte und "überschaubare"
Menge an benötigten Werten

Für nicht-deterministische Algorithmen

Für Algorithmen mit unendlich vielen
möglichen Eingabe- und Ausgabewerten

Für häufig auftretende, identische Berechnungen

Speicherplatzoptimierung

- + Mittels LUT konstante “Berechnungszeit”
- Nur verwendbar, wenn alle gewünschten Fibonacci-Zahlen bereits vorberechnet sind
- Lookuptabelle potentiell sehr groß
 - ▶ Hoher Speicherplatzverbrauch
 - ▶ Nicht für sehr große n praktikabel/möglich

Speicherplatzoptimierung: LUT verkleinern

- ▶ Lookuptabelle in Abschnitte unterteilen
 - ▶ Erste zwei Werte jeden Abschnitts speichern
 - ▶ Restlichen Werte ab Abschnittanfang dynamisch zur Laufzeit berechnen
- ▶ Z.B. 6 Abschnitte mit je 16 (14 im letzten Abschnitt) Zahlen
 - ▶ Lookuptabelle schrumpft von 94 Einträgen auf 12

Fibonacci: Kleine LUT (1)

```
1 #include <stdint.h>
2
3 // LUT for n = {0,16,32,48,64,80}
4 uint64_t lut0[] = {
5     0,987,2178309,4807526976,10610209857723,
6     23416728348467685};
7
8 // LUT for n = {1,17,33,49,65,81}
9 uint64_t lut1[] = {
10    1,1597,3524578,7778742049,17167680177565,
11    37889062373143906};
12 ...
```

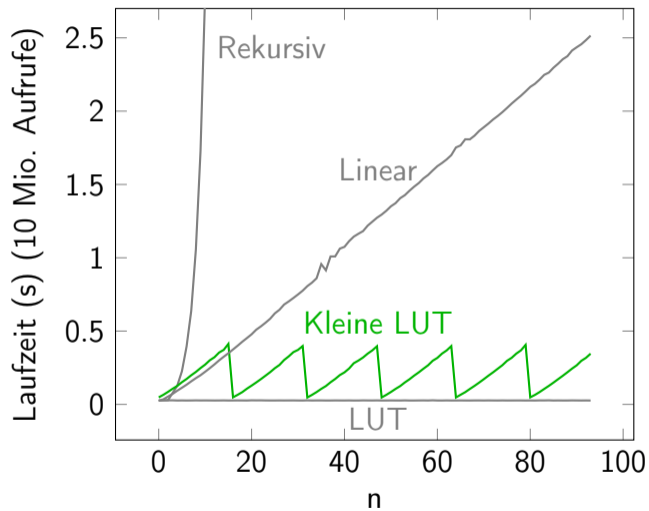
Fibonacci: Kleine LUT (2)

```
12 ...
13 uint64_t fib4(uint64_t n) {
14     if (n > 93) {
15         return UINT64_MAX;
16     }
17
18     uint64_t index = n / 16;
19     uint64_t a = lut0[index];
20     uint64_t b = lut1[index];
21     ...
```

Fibonacci: Kleine LUT (3)

```
21     ...
22     index *= 16;
23     if (index == n)
24         return a;
25
26     index++;
27     for (; index < n; index++) {
28         uint64_t tmp = b;
29         b += a;
30         a = tmp;
31     }
32
33     return b;
34 }
```

Laufzeit der Fibonacci Implementierungen im Vergleich



Laufzeitvergleich der Fibonacci Algorithmen

	Rekursiv	Schleife	LUT	Kleine LUT
Laufzeitklasse	Exponentiell	Linear	Konstant	Konstant ¹
Wiederholungen	1	—	10.000.000	—
f(40)	0.55s	1.13s	0.03s	0.23s
f(45)	6.02s	1.27s	0.03s	0.36s
f(50)	67.49s	1.40s	0.03s	0.09s
f(93)	—	2.70s	0.03s	0.36s

¹Je weniger Werte die LUT umfasst, desto mehr hat die worst-case Laufzeit linearen "Charakter"

Quiz: Fibonacci (Kleine LUT) (1)

Um welchen Faktor können wir eine LUT ungefähr verkleinern, wenn wir sie mit der eben besprochenen Speicherplatzoptimierung (speziell für die Fibonacci Zahlen) in 8 Abschnitte aufteilen?

≈ 6

≈ 12

≈ 8

≈ 16

Quiz: Fibonacci (Kleine LUT) (2)

Für welche Eingabewerte hat die Laufzeit der kleinen LUT ihr

Minimum

Für die Werte am Anfang
jedes Abschnitts

$n \in \{0, 16, 32, 48, 64, 80\}$

Für die Werte am Ende
jedes Abschnitts

$n \in \{15, 31, 47, 63, 79\}$

Maximum

Quiz: Vergleich der Implementierungen

Welche Implementierung ist jeweils am besten auf (1) Laufzeit, (2) Speicherplatz, und (3) Laufzeit *und* Speicherplatz optimiert?

	1	2	3
Rekursiv	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Lineare Schleife	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
LUT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Kleine LUT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>