TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa
Peter Zeller

AG Softech
FB Informatik
TU Kaiserslautern

Quicksort

Anwenden der Divide-and-Conquer-Strategie zum Entwurf von Quicksort

- Zerlege das Problem in kleinere Teilprobleme.
- Wende den Algorithmus rekursiv auf die Teilprobleme an.
- Füge die Teilergebnisse wieder zusammen.

Quicksort

Anwenden der Divide-and-Conquer-Strategie zum Entwurf von Quicksort

- Zerlege das Problem in kleinere Teilprobleme.
  - Wähle ein beliebiges Element p aus der Liste aus (**Pivotelement**)
  - Teile die restlichen Elemente in zwei Teillisten:
    Teil 1: die Elemente $< p$.
    Teil 2: die Elemente $\geq p$
- Wende den Algorithmus rekursiv auf die Teilprobleme an.

- Füge die Teilergebnisse wieder zusammen.

Quicksort

Anwenden der Divide-and-Conquer-Strategie zum Entwurf von Quicksort

- Zerlege das Problem in kleinere Teilprobleme.
  - Wähle ein beliebiges Element p aus der Liste aus (**Pivotelement**)
  - Teile die restlichen Elemente in zwei Teillisten:
    Teil 1: die Elemente $< p$.
    Teil 2: die Elemente $\geq p$
- Wende den Algorithmus rekursiv auf die Teilprobleme an.
  - Liefert sortierten Teil 1 und Teil 2
- Füge die Teilergebnisse wieder zusammen.

Quicksort

Anwenden der Divide-and-Conquer-Strategie zum Entwurf von Quicksort

- Zerlege das Problem in kleinere Teilprobleme.
  - Wähle ein beliebiges Element p aus der Liste aus (**Pivotelement**)
  - Teile die restlichen Elemente in zwei Teillisten:
    Teil 1: die Elemente $< p$.
    Teil 2: die Elemente $\geq p$
- Wende den Algorithmus rekursiv auf die Teilprobleme an.
  - Liefert sortierten Teil 1 und Teil 2
- Füge die Teilergebnisse wieder zusammen.
  - Erst Teil 1, dann das Pivotelement p, dann Teil 2

Quicksort Beispiel

- Gegeben sei die Liste $[17, 12, 6, 19, 23, 8, 5, 10]$.
- Wähle zum Beispiel 10 als Pivotelement.
- Teil 1: Elemente < 10:
  zum Beispiel: $[5, 8, 6]$
- Teil 2: Elemente ≥ 10:
  zum Beispiel: $[23, 12, 17, 19]$
- Sortiere die 2 Teile:
  Teil 1: $[5, 6, 8]$
  Teil 2: $[12, 17, 19, 23]$
- Zusammenfügen: $[5, 6, 8, \quad 10, \quad 12, 17, 19, 23]$

Beispiel: Partitionierung auf Arrays

17	12	6	19	23	8	5	10

Wähle als Pivotelement das letzte Element (10):

17	12	6	19	23	8	5	10

Aufteilen der übrigen Elemente:

17	12	6	19	23	8	5	10

Vertausche 5 und 17:

5	12	6	19	23	8	17	10

Vertausche 8 und 12:

5	8	6	19	23	12	17	10

Keine Vertauschung mehr möglich:

5	8	6	19	23	12	17	10

Tausche nun das Pivotelement zwischen die beiden Partitionen:

5	8	6	10	23	12	17	19

Partitionierung auf Arrays I

- Bearbeite rekursiv Teilbereiche des Arrays (`n` Elemente, startend bei `f`)
- Realisiere das Teilen der Liste durch Vertauschen
  - → Indexzähler `left`, `right` laufen von links bzw. rechts und suchen Einträge, die vertauscht werden können.
  - → Für die zu tauschenden Einträge gilt:

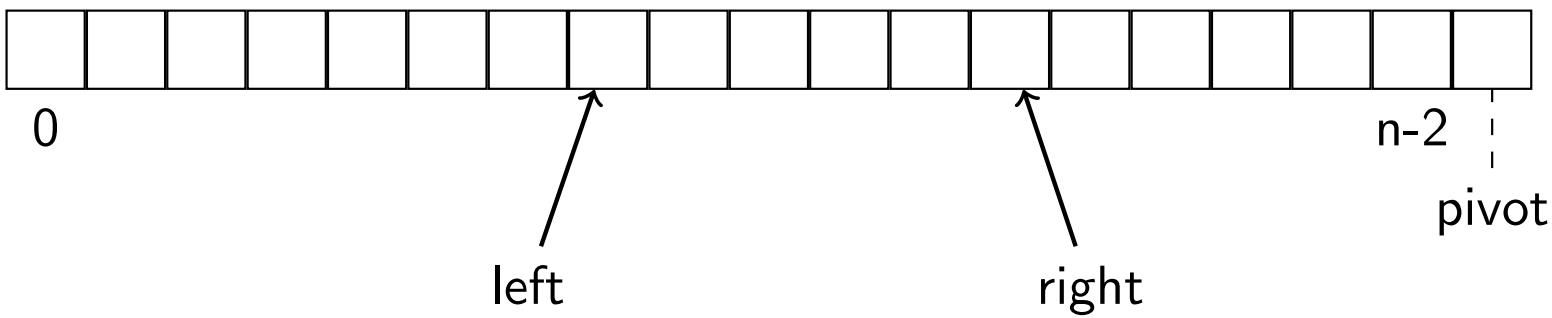
    $$\texttt{f[left]} \geq \texttt{pivot} \text{ und } \texttt{f[right]} < \texttt{pivot}$$

  - → In jedem Schritt gilt:
    - Für alle i in [0, `left-1`] : `f[i]` < `pivot`
    - Für alle i in [`right+1`, `n-2`] : `pivot` $\leq$ `f[i]`

Partitionierung auf Arrays II

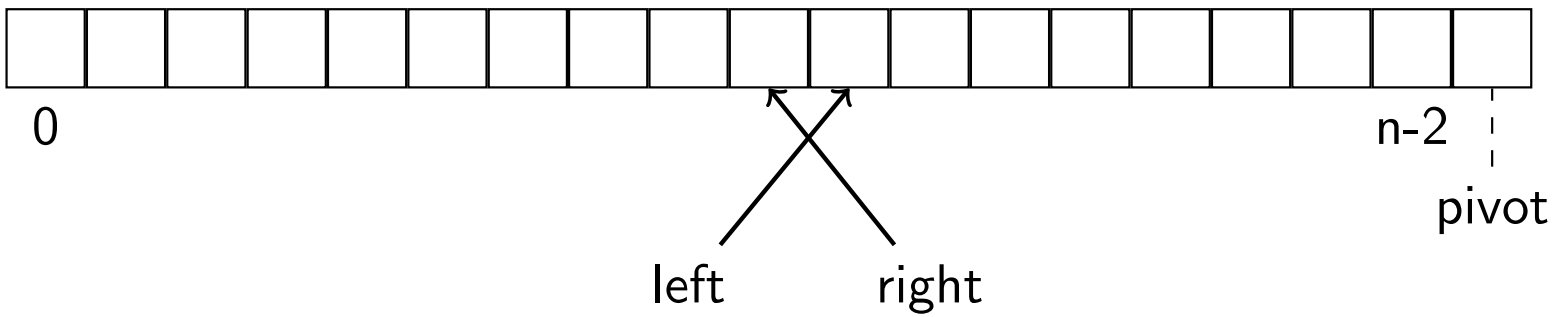
Wenn ein Paar zum Vertauschen gefunden wurde gilt `left < right`:

- Vertausche `f[left]` und `f[right]`
- Inkrementiere `left` und dekrementiere `right`
- Fahre dann mit der Suche nach vertauschbaren Elementpaaren fort.



Partitionierung auf Arrays III

Die Umsortierung des (Teil-)Arrays ist abgeschlossen, sobald `left > right`.



Zum Schluss wird das `pivot` an die richtige Stelle getauscht, indem das Element an Position `left` und das an Position `n-1` vertauscht werden.

Implementierung der Partitionierung

```
int partition(int *f, int n){
  int left = 0;
  int right = n - 2;

  int pivot = f[n-1];
  while (true) {
    while (f[left] < pivot) { left++; }
    while (left <= right && f[right] >= pivot){ right--; }
    if( left > right ) {
      break;
    } else {
      swap(&f[left], &f[right]);
      left++; right--;
    }
  }
  // Pivotelement kommt zwischen die beiden Partitionen
  swap(&f[left], &f[n-1]);
  return left;
}
```

Implementierung von Quicksort

```
void quicksort(int *f, int n) {
  if (n > 1) {
    int ixsplit = partition(f, n);
    quicksort(f, ixsplit);
    quicksort(&f[ixsplit+1], n - 1 - ixsplit);
  }
}
```

Optimierungen

Die vorgestellte Quicksort-Fassung arbeitet schlecht auf schon sortierten Arrays. Dies kann man durch folgende Strategien verhindern:

- → Shuffle des Arrays, um Vorsortierung aufzuheben
- → Randomisierte Auswahl des Pivot-Elements (Beispiel: Median von 3 Elementen)

Der Aufwand für die rekursiven Funktionsaufrufe ist außerdem vergleichsweise hoch, wenn die Teil-Arrays nur noch wenige Elemente enthalten. Man kann die Anzahl dieser Aufrufe reduzieren, in dem andere Sortierverfahren wie InsertionSort zur Sortierung von Teil-Arrays mit nur wenigen Elemente verwendet wird.

Anwenden von Sortierung: Binäre Suche I

Ein kleines Ratespiel:

Ich denke mir eine natürliche Zahl x aus der Menge $\{0, 1, \ldots 999\}$ aus.

Sie müssen diese Zahl erraten und dürfen mir dazu Fragen stellen.

Die einzige erlaubte Frage ist dabei "Ist die Zahl kleiner als ___ ?".
Die Antwort ist entweder "Ja" oder "Nein".

- Wie gehen Sie vor?
- Wie viele Fragen müssen Sie maximal stellen, um die Zahl zu ermitteln?
- Wie lässt sich Ihre Strategie allgemein als Algorithmus beschreiben?

Anwenden von Sortierung: Binäre Suche II

Beispiel:

Ist die Zahl kleiner als 500 ? Ja!
Ist die Zahl kleiner als 250 ? Nein!
Ist die Zahl kleiner als 375 ? Ja!
Ist die Zahl kleiner als 312 ? Ja!
Ist die Zahl kleiner als 281 ? Nein!
Ist die Zahl kleiner als 296 ? Ja!
Ist die Zahl kleiner als 288 ? Nein!
Ist die Zahl kleiner als 292 ? Nein!
Ist die Zahl kleiner als 294 ? Ja!
Ist die Zahl kleiner als 293 ? Ja!
Die gesuchte Zahl ist 292

Anwenden von Sortierung: Binäre Suche III

Algorithmische Idee zur Lösung des Ratespiels:
Suche x aus $\{0, 1, \ldots N-1\}$

- Intervall $[lo, hi)$ mit $x \in [lo, hi)$, das in jedem Schritt halbiert wird.
  Für die gesuchte Zahl x gilt immer $lo \leq x$ und $x < hi$
- Als Anfangsintervall wählen wir das Intervall $[0, N)$.
- Rekursive Strategie:

  Basisfall: Enthält das Intervall nur noch eine Zahl ($hi - lo = 1$), dann ist die gesuchte Zahl $x = lo$
  Rekursiver Schritt:
  - Frage, ob die Zahl kleiner ist als die Mitte des Intervalls, $m = lo + (hi - lo)/2$.
  - Falls ja, suche die Zahl im linken Teilintervall $[lo, m)$.
  - Andernfalls, suche die Zahl im rechten Teilintervall $[m, hi)$.

Implementierung des Ratespiels

```
int search(int lo, int hi) {
  // Basisfall
  if ((hi-lo) == 1) {
    return lo;
  }
  // Rekursiver Schritt
  int mid = lo + (hi -lo) / 2;
  printf("Ist die Zahl kleiner als %d ? (j/n)\n", mid);
  char answer;
  scanf("%c", &answer);
  getchar(); // liest naechstes Zeichen (Zeilenumbruch)
  if (answer == 'j') {
    return search(lo,mid);
  } else {
    return search(mid,hi);
  }
}
```

Korrektheit des Verfahrens I

Für Aufrufe von `search(lo,hi)` gilt:

- Das Intervall $[lo, hi)$ enthält immer die gesuchte Zahl.
  [Beweis durch Induktion]
- Bei Rückgabe enthält das Intervall nur die Zahl `lo`.
  Das ist dann die gesuchte Zahl.

Korrektheit des Verfahrens II

Terminierung:

- Terminiert für Aufrufe mit `hi > lo`
  Abbruchfall: $(hi - lo) = 1$
- Mit jedem rekursiven Aufruf wird der Abstand von $hi - lo$ zu 1 kleiner:
  Bei rekursiven Aufrufen gilt: $hi - lo > 1$
  Daher gilt:

$$mid = lo + \left\lfloor \frac{hi - lo}{2} \right\rfloor = \left\lfloor \frac{hi + lo}{2} \right\rfloor$$

$$mid < hi$$

$$mid > lo$$

Laufzeitanalyse: Wie viele Fragen werden benötigt?

- Annahme: $N = 2^n$ für ein $n \in \mathbb{N}$, d.h. $n = \log_2(N)$.
- Sei $T(N)$ die Anzahl der Fragen für Intervallgröße N
- In jedem Rekursionsschritt wird eine Frage gestellt und das Problem auf einem Intervall halber Größe gelöst:

$$T(N) = T(N/2) + 1$$

- Im Basisfall ist keine Frage mehr nötig: $T(1) = 0$
- Insgesamt:

$$\begin{aligned} T(N) &= T(2^n) \\ &= T(2^{n-1}) + 1 = T(2^{n-2}) + 2 = \cdots = T(2^{n-n}) + n \\ &= T(2^0) + n = T(1) + n \\ &= n \\ &= log_2(N) \end{aligned}$$

- Insgesamt benötigt man $n + 1 = \log_2(N) + 1$ Aufrufe von `search`.

Binäre Suche in sortiertem Array

- Idee der Binären Suche auch beim Suchen in sortierten Datensammlungen anwendbar, die direkten Zugriff auf einen Eintrag erlauben
- Alltagsbeispiele: Wörterbuch, Telefonbuch

Binäre Suche in sortiertem Array

Problembeschreibung:
Gegeben sei eine Folge $s_0, \ldots, s_{N-1}$ von sortierten *Datensätzen*.

Jeder Datensatz s_j hat einen zugehörigen Schlüssel k_j.
Wir gehen hier davon aus, dass die Schlüssel Integer sind.

Für sortierte Datensätze gilt:

$$k_0 \leq k_1 \leq \cdots \leq k_{N-1}$$

```
typedef struct dataset {
  int key;
  char *data;
} dataset_t;
```

Implementierung (rekursiv)

```
dataset_t *search(int x, dataset_t *f, int lo, int hi) {
  if (hi <= lo) {
    // Element nicht gefunden
    return NULL;
  }
  int mid = lo + (hi-lo) /2;
  if (x < f[mid].key) {
    // Suche rekursiv im Intervall [lo, mid)
    return search(x, f, lo, mid);
  }
  if (x > f[mid].key) {
    // Suche rekursiv im Intervall [mid+1, hi)
    return search(x, f, mid+1, hi);
  }
  // Element gefunden: x == f[mid].key
  return &f[mid];
}
```

Zusammenfassung: Binäre Suche

Voraussetzung für binäre Suche sind **sortierte Daten** und schneller Zugriff über Index.

Mit der binären Suche kann in jedem Schritt der Suchbereich **halbiert** werden.

In einem sortierten Array mit N Elementen kann ein Eintrag mit etwa $log_2(N)$ Vergleichen gefunden werden.

N	Anzahl Vergleiche für Suche
1024	10
82 175 684	27
7 450 000 000	33

Ausblick: Algorithmenanalyse

- Welcher der vorgestellten Sortieralgorithmen ist denn der Beste?
- Welcher Algorithmus sollte in einem konkreten Anwendungsfall angewendet werden?

Herangehensweise

- Durch Messen auf Testeingaben
- Durch theoretische Analyse mit Hilfe eines Modells

Laufzeit messen (Demo)

Beispiel: Selectionsort

- `time` Programm
- `clock` Funktion

Laufzeit messen (Demo)

Beispiel: Selectionsort

- `time` Programm
- `clock` Funktion

Achtung:
Beim Entwerfen von Programmen zum Messen der Laufzeit sollte man bedenken, dass **Optimierungen**, die der Compiler anwenden kann, eventuell das Ergebnis verfälschen könnten. Zum Beispiel könnte ein hinreichend intelligenter Compiler erkennen, dass das sortierte Array nie gelesen wird und daher das Sortieren aus dem Programm entfernen.

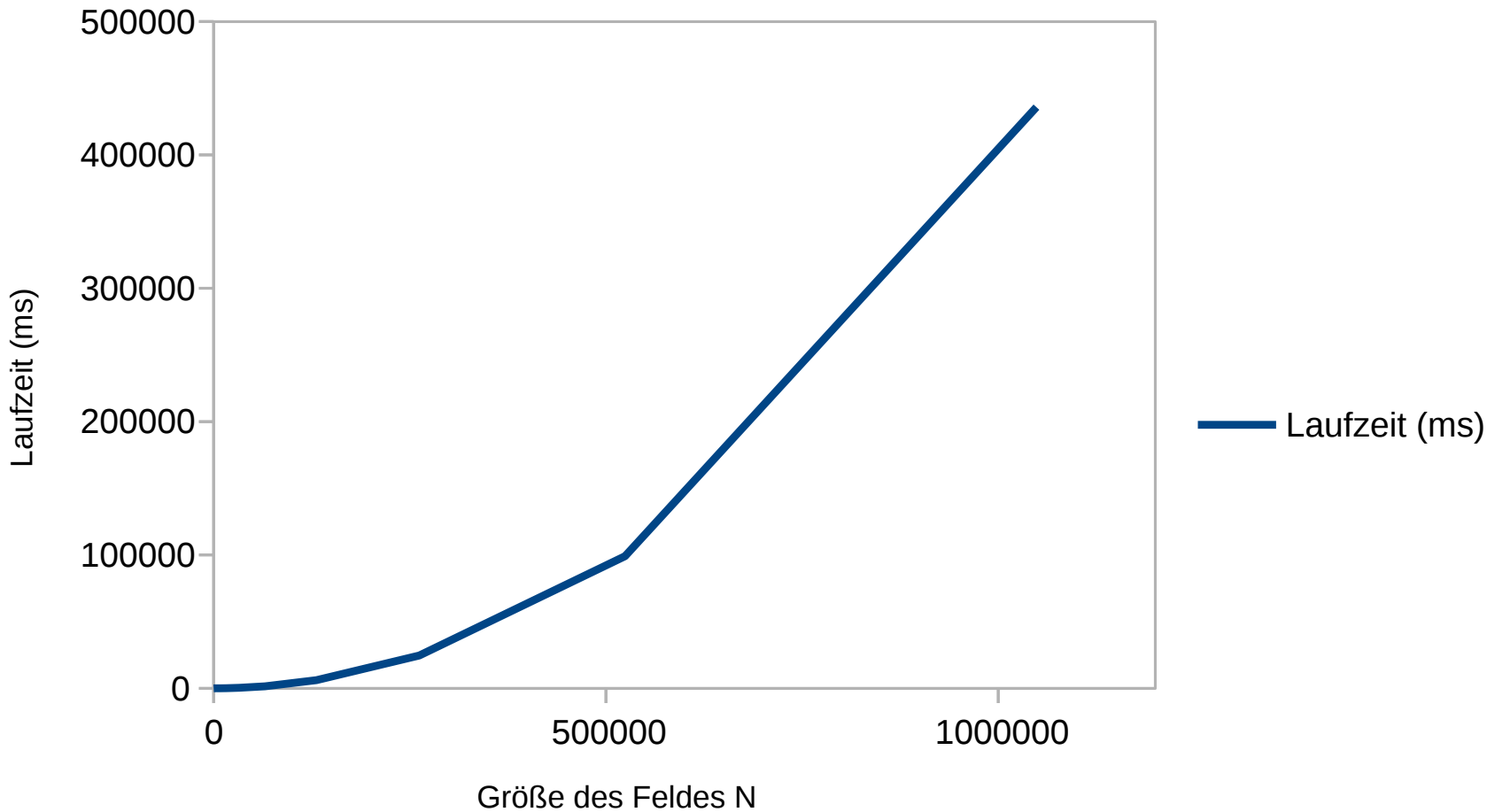
Beobachtungen

- Die Laufzeit ist abhängig von der Größe N des zu sortierenden Arrays (insbesondere für große N).
- Die Laufzeit variiert leicht für *gleiche* Eingabe bei verschiedenen Durchläufen.
- Die Laufzeit ändert sich, wenn andere Hardware verwendet wird, der Trend bleibt aber gleich.

Problemgrößen

- Typischerweise beeinflusst eine (bisweilen auch mehrere) **Problemgröße** die Komplexität eines Programms.
- Die Laufzeit sollte mit der wachsender Problemgröße zunehmen.
- Typischerweise ist dies die Größe der Eingabe oder auch der Wert von Befehlszeilenargumenten.
- Im Beispiel: Größe N des zu sortierenden Arrays

Laufzeit von Selectionsort für verschiedene Eingabegrößen:



Laufzeit von Selectionsort für verschiedene Eingabegrößen:

N	Laufzeit (ms)
1	0
2	0
4	0.001
8	0.001
16	0
32	0.001
64	0.004
128	0.01
256	0.033
512	0.114
1024	0.412
2048	1.562
4096	10.494
8192	27.875
16384	102.111
32768	366.685
65536	1450.968
131072	5901.001

Laufzeit von Selectionsort für verschiedene Eingabegrößen:

N	Laufzeit (ms)	Veränderung
1	0	
2	0	
4	0.001	
8	0.001	
16	0	
32	0.001	
64	0.004	4.00
128	0.01	2.50
256	0.033	3.30
512	0.114	3.45
1024	0.412	3.61
2048	1.562	3.79
4096	10.494	6.72
8192	27.875	2.66
16384	102.111	3.66
32768	366.685	3.59
65536	1450.968	3.96
131072	5901.001	4.07

Hypothese zum Laufzeitverhalten von Sortieren durch Auswahl

Die Laufzeit erhöht sich ungefähr um den Faktor 4, wenn sich die Eingabegröße verdoppelt.

Komplexitätsklassen zur Klassifizierung von Algorithmen

Klasse	Verdoppeln	Bezeichnung	Beispiel
$O(1)$	*1	*konstant*	Schaltjahrberechnung
$O(\log(N))$		*logarithmisch*	binäre Suche
$O(N)$	*2	*linear*	Mittelwert von Int-Array
$O(N \cdot \log(N))$		*linearithmetisch*	Mergesort
$O(N^2)$	*4	*quadratisch*	Sortieren durch Auswahl
$O(N^3)$	*8	*kubisch*	Matrixmultiplikation
$O(2^N)$		*exponentiell*	diverse Optimierungen

Algorithmen mit einem Aufwand in $O(N^k)$ nennt man **polynomiell** (engl. *polynomial*).

Laufzeitabschätzung basierend auf Komplexitätsklassen

Annahme: Programm benötigt für ein Problem der Größe N auf einem Rechner wenige Sekunden.

- Wie verhält sich die Laufzeit des Programm, wenn man die Problemgröße erhöht?
- Um wieviel kann ein schnellerer Rechner die Berechnungszeit verringern?

Lösbarkeit großer Probleme

Wenn man die Problemgröße um den Faktor 100 erhöht, wächst die Laufzeit von einigen Sekunden auf ...

Klasse	Laufzeitabschätzung
linear	einige Minuten
linearithmetisch	einige Minuten
quadratisch	mehrere Stunden
kubisch	einige Wochen
exponentiell	Milliarden von Jahren

Nutzung schnellerer Rechner

Wenn man einen Computer verwendet, der um den Faktor 10 schneller rechnet, kann in der gleichen Zeit eine Probleminstanz gelöst werden, die um den Faktor . . . größer ist.

Klasse	Faktor der Erhöhung der Problemgröße
linear	10
linearithmetisch	10
quadratisch	3-4
kubisch	2-3
exponentiell	1

Bemerkungen

- O-Notation liefert nur eine grobe Klassifizierung durch obere Schranke
- Fokus auf asymptotisches Verhalten bei großen Eingaben
- (Relativ einfache) theoretische Einordnung und Vergleichsbasis
- In der Praxis können “konstante” Faktoren entscheidend sein $\Rightarrow$ Kostenmodell!