Laufzeitanalyse

Richard Göbel

Hochschule Hof
University of Applied Sciences

Sortieren eines Arrays mit Zahlen - Analyse der Laufzeit verschiedener Algorithmen

Hochschule Hof
University of Applied Sciences

- Bubblesort als einfacher Algorithmus für das Sortieren
- Überprüfe in dem Array, ob eine kleinere Zahl auf eine größere Zahl folgt
- Wenn dies der Fall ist, dann vertausche diese beiden Zahlen
- Ist dies nicht der Fall, dann ist die Liste sortiert

13	14	15	17	18

Implementierung eines Bubblesort


```
public void sort(int[] array) {
  int h;
  boolean found = true;

  while (found) {
    found = false;
    for (int i = 0; i < array.length-1; i++) {
      if (array[i] > array[i+1]) {
        h = array[i+1];
        array[i+1] = array[i];
        array[i] = h;
        found = true;
      }
    }
  }
}
```

Hochschule Hof
University of Applied Sciences

```
public class AnalyzeTimeSorting {
    SortingAlgorithm algorithm;

    public AnalyzeTimeSorting(SortingAlgorithm algorithm) {
        this.algorithm = algorithm;

    }


    public int[] genArray(int length) {
        int[] result = new int[length];
        Random random = new Random(1);
        for (int i = 0; i < length; i++) {
            result[i] = random.nextInt();
        }
        return result;
    }

    public long durationSort(int[] array) {
        long start = System.currentTimeMillis();
        algorithm.sort(array);
        return System.currentTimeMillis() - start;
    }

    public void tryDifferentSizes(int start, int step, int end, PrintStream stream) {
        for (int i = start; i <= end; i+=step) {
            System.out.println(i);
            int[] array = genArray(i);
            stream.println(durationSort(array));
        }
    }

    public static void main(String[] args) throws FileNotFoundException {
        SortingAlgorithm bubble = new Bubblesort();
        SortingAlgorithm quick = new Quicksort();

        AnalyzeTimeSorting sort = new AnalyzeTimeSorting(bubble);
        String file = "Log" +(System.currentTimeMillis()/1000) + ".csv";
        PrintStream ps = new PrintStream(new FileOutputStream(file));
        sort.tryDifferentSizes(10000, 10000, 100000, ps);
    }
}
```

Vergleich Bubblesort mit Quicksort

- Quicksort aus der Java Library
  `Arrays.sort(int[] a)`
- Arrays mit verschiedenen Größen erzeugen
- Laufzeit für beide Verfahren mit den verschieden großen Arrays messen

Hochschule Hof
University of Applied Sciences

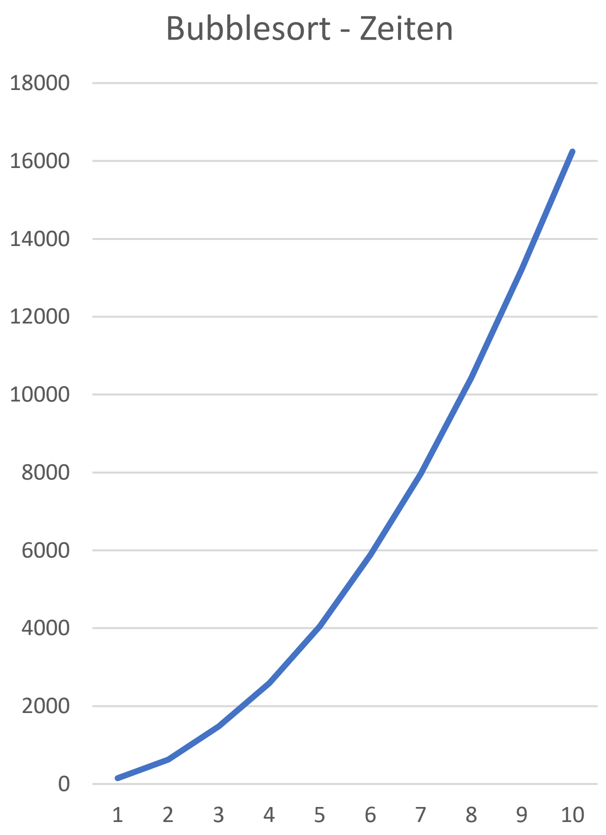
Vergleich der Zeiten für verschiedene Größen

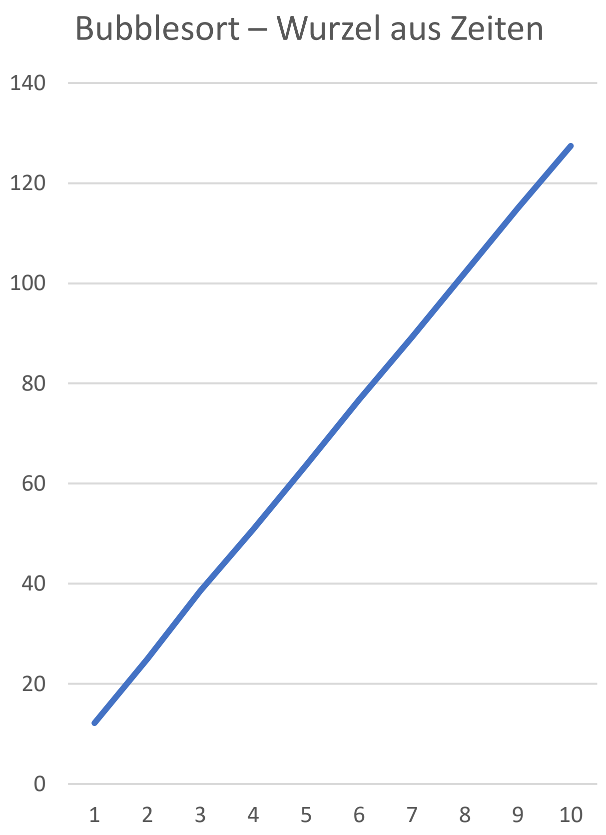
Bubblesort	Millisekunden
10.000	148
20.000	624
30.000	1.484
40.000	2.588
50.000	4.051
60.000	5.889
70.000	7.968
80.000	10.429
90.000	13.226
100.000	16.242

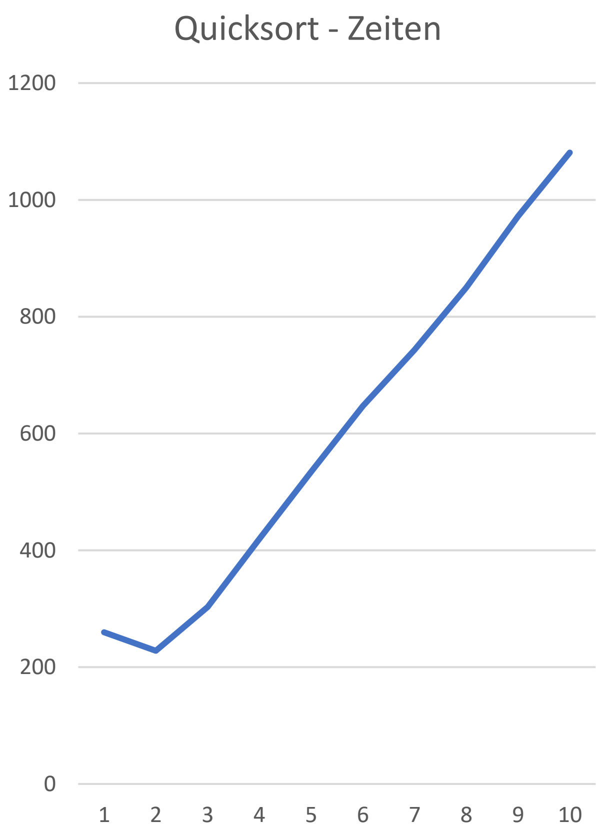
Quicksort	Millisekunden
1.000.000	260
2000.000	228
3.000.000	303
4.000.000	419
5.000.000	534
6.000.000	647
7.000.000	743
8.000.000	850
9.000.000	972
10.000.000	1.081

Vergleich des funktionalen Zusammenhangs zwischen der Größe des Arrays und der Laufzeit

Hochschule Hof
University of Applied Sciences







Schlussfolgerungen

_ Der Aufwand für das Sortieren mit dem Bubblesort scheint quadratisch mit der Länge n des Arrays zu wachsen

_ Der Aufwand für das Sortieren mit dem Quicksort scheint linear mit der Länge n des Arrays zu wachsen …
*… tatsächlich wächst der Aufwand mit n * log(n)*

_ Das Wachstum des (Zeit-) Aufwands mit der Größe der Eingabe kann aus dem Programm abgeleitet werden

Zeitaufwand für ein Array mit *n* Elementen

```
public void sort(int[] array) {
  int h;
  boolean found = true;

  while (found) {
    found = false;
    for (int i = 0; i < array.length-1; i++) {
      if (array[i] > array[i+1]) {
        h = array[i+1];
        array[i+1] = array[i];
        array[i] = h;
        found = true;
      }
    }
  }
}
```

Schwieriger zu analysieren!

- Ungünstigster Fall: die kleinste Zahl ist an der letzten Position des Arrays
- In diesem Fall muss die Schleife n-1 mal durchlaufen werden

Die Schleife wird n-1 mal durchlaufen

Als Konsequenz werden Anweisungen in der inneren For-Schleife im ungünstigsten Fall (n-1) * (n-1) mal durchlaufen.

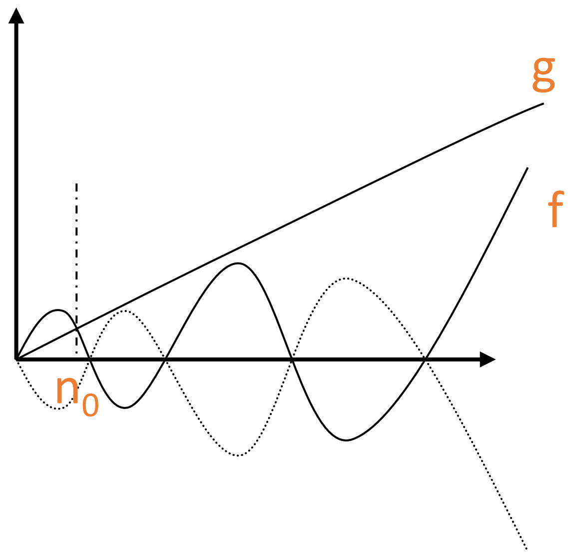
Vergleich des Laufzeitverhaltens verschiedener Algorithmen

- Für den Bubblesort wächst die Laufzeit mit $(n – 1) * (n – 1) = n^2 - 2*n + 1$
- Bei großen Arrays wird die Laufzeit von dem am schnellsten wachsenden Term bestimmt …
  … in diesem Fall also von n^2
- Für zwei andere Algorithmen könnte der Aufwand mit $2*n^3 + 4*n + 12$ sowie $4*n +76$ abgeschätzt werden
- Die am schnellsten wachsende Terme in dieser Summe sind $2 * n^3$ sowie $4 * n$.
- Für den Vergleich spielen auch die Koeffizienten 2 und 4 keine Rolle. Entsprechend werden hier nur die Potenzen für den Vergleich betrachtet: n^3, n^2, n^1
- Tatsächlich sind die Koeffizienten sogar irreführend, da die genauen Ausführungszeiten von dem Rechenaufwand für einzelnen Anweisungen in den Schleifen abhängen
- Mit dieser Analyse kann also nur die **Form der Kurve** festgestellt werden, welche den Zusammenhang zwischen Größe der Eingabe und Laufzeit repräsentiert → **Vorauswahl der Algorithmen**
- Algorithmen mit identischen Formen von Kurven (zum Beispiel quadratisch) müssen dann mit Hilfe von **Benchmarks** verglichen werden (Implementierungen mit definierten Eingaben unterschiedlicher Größe)

Ansatz

_ Die Laufzeit ist abhängig von
  _ der Menge der Daten
  _ technologischen Parametern
_ Absolute Messungen sind nur bei direktem Vergleich mit definierter Umgebung sinnvoll
_ Der Zusammenhang zwischen Laufzeit und Datenmenge …
_ … lässt sich (fast) unabhängig von technologischen Parametern analysieren
_ Mögliche Angaben
  _ Untergrenze
  _ Durchschnitt
  _ **Obergrenze**

Formalisierung – Mathematische Formulierung



- Die O-Notation (auch Landau-Notation oder Landau-Symbole) gibt eine qualitative Abschätzung für den Verlauf einer Funktion
  - $f \in O(g): \exists\, c, n_0 : \forall\, n : n \geq n_0 \Rightarrow |f(n)| \leq c \bullet |g(n)|$
  - $f \in \Omega(g): \exists\, c, n_0 : \forall\, n : n \geq n_0 \Rightarrow |f(n)| \geq c \bullet |g(n)|$
- Für eine Funktion *f* wird der qualitative Aufwand mit einer einfacheren Funktion *g* angegeben
- Ansatz
  - Berücksichtige nur den am schnellsten wachsenden Term
  - Ignoriere Koeffizienten

Typische Beispiele für den Verlauf einer Kurve

- $O(1)$: Konstant
- $O(\log(n))$: Logarithmisch
- $O(n)$: Linear
- $O(\log(n) \bullet n)$: Superlinear
- $O(n^2)$: Quadratisch
- $O(n^3)$: Kubisch
- $O(2^n)$: Exponentiell

- Abschätzung des qualitativen Verlaufs

  $3 \bullet n^3 - 17 \bullet n^2 + \log(n) + 328$
- Nur den am schnellsten wachsenden Term berücksichtigen

  $3 \bullet n^3$
- Koeffizienten ignorieren

  n^3
- Ergebnis

  $3 \bullet n^3 - 17 \bullet n^2 + \log(n) + 328 \in O(n^3)$

Analyse von Algorithmen

_ Schleifen analysieren

_ Keine Schleife: O(1)

_ Einzelne For-Schleife: O(n) (aber nicht immer)

```
for (int i = 0; i < array.length; i++) { … }
```

_ Zwei ineinander geschachtelte For-Schleifen: O(n^2) (es gibt Ausnahmen)

```
for (int i = 0; i < array.length; i++) {
  for (int  j = i; j < array.length; j++) {
… }}
```

_ Analyse einer While-Schleife ist kompliziert

```
while ( … ) { … }
```

Beispiel: Finde das kleinste Element: O(n)

```
public int minElem(int data[]) {
  int min = Integer.MAX_VALUE;
  for (int i = 1; i < data.length; i++) {
    if (min > data[i]) {
      min = data[i];
    }
  }
  return min;
}
```

Kleinste Differenz von zwei Zahlen: $O(n^2)$

```
// Quadratische Zeit, das geht auch besser!
public int smallestDist (int data[]) {
  int dist = Integer.MAX_VALUE;
  for (int i = 1; i < data.length - 1; i++) {
    for (int j = i; j < data.length; j++) {
      if (Math.abs(data[i] - data[j]) < dist) {
        dist = Math.abs(data[i] - data[j]);
      }
    }
  }
  return dist;
}
```

Vereinfachtes Rucksackproblem

- Rucksack mit Gewichtsgrenze (maximales Gewicht)
- Anzahl von n Gegenstände mit unterschiedlichen Gewichten
- Gibt es eine Auswahl an Gegenständen, so das genau das Grenzgewicht erreicht wird?

Hochschule Hof
University of Applied Sciences

Lösungsansatz

9 kg	8 kg	6 kg	5 kg	4 kg	3 kg
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	0	1
0	0	0	1	1	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	0	1
0	0	1	0	1	0
…					

- Probiere alle Kombinationen durch
- Wähle die Kombination, welche genau das Zielgewicht hat
- Eventuell gibt es keine entsprechende Kombination
- Aufwand:
  - Für jeden Gegenstand wird entschieden, ob er im Rucksack ist oder nicht (binäre Entscheidung)
  - Daraus ergeben sich 2^n Kombinationen

Programm für das vereinfachte Rucksackproblem – Darstellung der Gegenstände

```
class Item {
  int weight;
  boolean selected = false;

  Item(int weight) {
    this.weight = weight;
  }
}
```

Programm für das vereinfachte Rucksackproblem – Klasse Knapsack Teil 1

```
public class Knapsack {
  ArrayList<Item> items = new ArrayList<Item>(); // alle Gegenstände

  void addItem(int weight) {
    items.add(new Item(weight));
  }

  void printSelectedItems() {
    for (Item item : items) {
      if (item.selected) {
        System.out.println(item.weight);
      }
    }
  }
    …
```

Programm für das vereinfachte Rucksackproblem – Klasse Knapsack Teil 2

Hochschule Hof
University of Applied Sciences

…

```
 int calcWeight() {
   int sum = 0;
   for (Item item : items) {
     if (item.selected) {
       sum += item.weight;
     }
   }
   return sum;
 }
```

…

Programm für das vereinfachte Rucksackproblem – Klasse Knapsack Teil 3

```
    …
boolean findCombination(int limit) {
  boolean result = false; // Ergebnis gefunden
  boolean flag = true; // probiere weitere Kombinationen
  while (flag) {
    // erzeuge eine neue Kombination durch Ändern der Attribute „selected“
    if (calcWeight() == limit) {
      result = true;
      flag = false;
      break;
    }
  }
  return result;
}
    …
```

Hochschule Hof
University of Applied Sciences

Erzeuge eine neue Kombination

```
flag = false;
for (Item item : items) {
  if (item.selected == false) {
    item.selected = true;
    flag = true;
    break;
  }
  else {
    item.selected = false;
  }
}
```

Aufwand für das Verfahren

- Das Verfahren hat einen exponentiellen Aufwand
- Es gibt sicher effizientere Lösungen
- Gibt es eine Lösung, die den exponentiellen Aufwand vermeidet?

- *Bisher wurde noch keine solche Lösung gefunden*

- Statt der Laufzeit eines Programms …
- … kann auch die minimale mögliche Laufzeit einer Aufgabenstellung …
- … relativ zur Größe der Eingabe untersucht werden
- Dann existiert kein Algorithmus, der die Aufgabenstellung schneller löst

- **Komplexitätstheorie**