Prof. Dr. Sven Strickroth

Einführung in die Programmierung

Wintersemester 2021/22

Grundkonzepte der Programmierung IV: Sortieralgorithmen und Komplexität

Übersicht

1. Sortieren
2. Laufzeitkomplexität
3. Suchen

Motivation

- Sortieren ist ein grundlegendes Problem in der Informatik

- Aufgabe:
  - Gegeben Datensatz mit Objekten, die einen Schlüssel enthalten
  - Umordnen der Datensätze, so dass eine klar definierte Ordnung (nummerisch, lexikographisch, etc.) der Schlüssel besteht

- Schlüssel: Attribut/Wert nach dem sortiert werden soll
  - ```
    class Student {
       String name;
       String vorname;
       long matrikelNr;
       int alter;
    }
    ```
  - Name, Matrikelnummer, etc. sind alles potentielle Schlüssel

- Vereinfachung: nur Betrachtung der Schlüssel, z.B. Array von Integer-Werten

- Beispiel:
  - Adressbuch, sortieren von Personen nach Namen
    ```

Sortieren durch Einfügen (InsertionSort)

- Prinzip: Wie das Sortieren eines Kartenstapels durch einen Menschen

Sortieren durch Einfügen (InsertionSort)

- Prinzip: Wie das Sortieren eines Kartenstapels durch einen Menschen

- 1. Aktuelles Element mit bereits sortiertem Feld vergleichen
- 2. An richtiger Stelle einfügen, Rest verschieben

Index:	0	1	2	3	4	5
zu sortierenden Feld:	5	1	8	3	9	2

Sortieren durch Einfügen (InsertionSort)

- Prinzip: Wie das Sortieren eines Kartenstapels durch einen Menschen

- 1. Aktuelles Element mit bereits sortiertem Feld vergleichen
- 2. An richtiger Stelle einfügen, Rest verschieben

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	5	8	3	9	2

Sortieren durch Einfügen (InsertionSort)

- Prinzip: Wie das Sortieren eines Kartenstapels durch einen Menschen

- 1. Aktuelles Element mit bereits sortiertem Feld vergleichen
- 2. An richtiger Stelle einfügen, Rest verschieben

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	5	8	3	9	2

Sortieren durch Einfügen (InsertionSort)

- Prinzip: Wie das Sortieren eines Kartenstapels durch einen Menschen

- 1. Aktuelles Element mit bereits sortiertem Feld vergleichen
- 2. An richtiger Stelle einfügen, Rest verschieben

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	3	5	8	9	2

Sortieren durch Einfügen (InsertionSort)

- Prinzip: Wie das Sortieren eines Kartenstapels durch einen Menschen

- 1. Aktuelles Element mit bereits sortiertem Feld vergleichen
- 2. An richtiger Stelle einfügen, Rest verschieben

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	3	5	8	9	2

Sortieren durch Einfügen (InsertionSort)

- Prinzip: Wie das Sortieren eines Kartenstapels durch einen Menschen

- 1. Aktuelles Element mit bereits sortiertem Feld vergleichen
- 2. An richtiger Stelle einfügen, Rest verschieben

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	2	3	5	8	9

InsertionSort: Algorithmus (Pseudocode)

```
InsertionSort(A):
  for i = 1 to A.size-1 {
    einzusortierender_wert = A[i] // aktuelles Element
    j = i
    while (j > 0 && A[j-1] > einzusortierender_wert) {
      A[j] = A[j-1]
      j = j–1
    }
    A[j] = einzusortierender_wert
  }
```

- Hier implementiert als in-place Verfahren (arbeitet direkt auf dem Array).
- Es ist auch möglich einen zweiten Array als Ergebnis-Array zu benutzen → doppelter Speicherplatz notwendig.

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	0	1	2	3	4	5
zu sortierenden Feld:	5	1	8	3	9	2

1. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	**0**	**1**	**2**	**3**	**4**	**5**
zu sortierenden Feld:	1	5	8	3	9	2

1. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	5	8	3	9	2

1. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	**0**	**1**	**2**	**3**	**4**	**5**
zu sortierenden Feld:	1	5	3	8	9	2

1. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	5	3	8	9	2

1. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	5	3	8	2	9

2. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	**0**	**1**	**2**	**3**	**4**	**5**
zu sortierenden Feld:	1	5	3	8	2	9

2. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	**0**	**1**	**2**	**3**	**4**	**5**
zu sortierenden Feld:	1	3	5	8	2	9

2. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	**0**	**1**	**2**	**3**	**4**	**5**
zu sortierenden Feld:	1	3	5	8	2	9

2. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	**0**	**1**	**2**	**3**	**4**	**5**
zu sortierenden Feld:	1	3	5	2	8	9

3. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	**0**	**1**	**2**	**3**	**4**	**5**
zu sortierenden Feld:	1	3	5	2	8	9

3. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	3	5	2	8	9

3. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	3	2	5	8	9

4. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	**0**	**1**	**2**	**3**	**4**	**5**
zu sortierenden Feld:	1	3	2	5	8	9

4. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	2	3	5	8	9

5. Durchlauf

BubbleSort: Prinzip

- Durchlaufe Array von links nach rechts
  - vergleiche aktuelles Element mit Nachbarn
  - vertausche Elemente, falls nicht in richtiger Reihenfolge
  - wiederhole solange, bis keine Vertauschung mehr notwendig
- Größte Zahl rutsche in jedem Durchlauf automatisch an das Ende der Liste
- Im Durchlauf j reicht die Untersuchung bis Position n-(j-1)

Index:	0	1	2	3	4	5
zu sortierenden Feld:	1	2	3	5	8	9

6. Durchlauf

BubbleSort: Algorithmus (Pseudocode)

```
bubbleSort(Array A):
  n = A.size
  do {
    swapped = false
    for (i=0; i<n-1; ++i) {
      if (A[i] > A[i+1]) {
        A.swap(i, i+1)
        swapped = true
      }
    }
    n = n-1
  } while (swapped)
```

Übersicht

1. Sortieren
2. Laufzeitkomplexität
3. Suchen

Laufzeitmessung

Array-Länge	InsertionSort Random	BubbleSort random
100	0 ms	0 ms
1.000	3 ms	6 ms
10.000	52 ms	125 ms
20.000	199 ms	505 ms
30.000	438 ms	1.295 ms
40.000	768 ms	2.195 ms
100.000	4.757 ms	14.121 ms
…		

Ressourcenbedarf von Algorithmen

- Algorithmen verbrauchen zwei Ressourcen:
  - Rechenzeit
  - Speicherplatz

- Interessante Fälle:
  - beste Fall (best case): oft sehr leicht zu bestimmen, kommt in der Praxis jedoch selten vor
  - schlechteste Fall (worst case): liefert garantierte Schranken, meist relativ leicht zu bestimmen, oft zu pessimistisch
  - durchschittlicher Fall (average case): Abschätzung, wie viele Schritte „normalerweise“ zu erwarten sind

- Im folgenden werden wir nur die Rechenzeit/Laufzeit betrachten.

Laufzeitanalyse (1)

- 1. Ansatz: Direktes Messen der Laufzeit (z.B. in ms).
  - Abhängig von vielen Parametern (Rechnerkonfiguration, Rechnerlast, Compiler, Betriebssystem, ...)
  - → kaum übertragbar und ungenau

- 2. Ansatz: Zählen der benötigten Elementaroperationen des Algorithmus in Abhängigkeit von der Größe *n* der Eingabe.
  - Das algorithmische Verhalten wird als Funktion $f : \mathbb{N} \to \mathbb{N}$ der benötigten Elementaroperationen dargestellt.
  - Die Charakterisierung dieser elementaren Operationen ist abhängig von der jeweiligen Problemstellung und dem zugrunde liegenden algorithmischen Modell.
  - Beispiele für Elementaroperationen: Zuweisungen, Vergleiche, arithmetische Operationen, Arrayzugriffe, ...

Beispiel: Minimum in einem Array suchen (1)

- Eingabe: Array mit n Zahlen ($a_0, a_1, \ldots, a_{n-1}$).
- Ausgabe: Index i, so dass $a_i \leq a_j$ für alle Indizes $0 \leq j \leq n - 1$.
- Beispiel:
  - Eingabe: 28, 51, 13, 8, 6, 12, 99
  - Ausgabe: 4

```java
public static int min(int[] a) {
  int min = 0;
  for (int i=1; i < a.length; ++i) {
    if (a[i] < a[min]) {
      min = i;
    }
  }
  return min;
}
```

Beispiel: Minimum in einem Array suchen (1)

```
public static int min(int[] a) {
  int min = 0;
  for (int i=1; i < a.length; ++i) {
    if (a[i] < a[min]) {
      min = i;
    }
  }
  return min;
}
```

Kosten	Anzahl
c_1	1
c_2	n-1
c_3	n-1
c_4	n-1

- Zusammen: Zeit: $T(n) = \mathrm{c}_1 + (\mathrm{n}-1)(\mathrm{c}_2 + \mathrm{c}_3 + \mathrm{c}_4 + \mathrm{c}_5) < \mathrm{C} * \mathrm{n}$
- Eingabegröße: Länge n des Arrays

Laufzeitanalyse (2)

- Das Maß für die Größe n der Eingabe ist abhängig von der Problemstellung, z.B.
  - Suche des Minimums in einem Array: n = Länge des Arrays
  - Sortierung einer Liste von Zahlen: n = Anzahl der Zahlen
  - Multiplikation zweier Matrizen: n = Dimension der Matrizen

- Benutzung eines Einheitskostenmaßes
  - alle Elementaroperationen „dauern“ gleich lange

- Laufzeit:
  - benötigte Elementaroperationen bei einer bestimmten Eingabelänge n
- Speicherplatz:
  - benötigter Speicher bei einer bestimmten Eingabelänge n

Analyse des InsertionSort Algorithmus (1)

- Wesentliche Parameter für den Aufwand bei unseren Sortieralgorithmen sind
  - Anzahl der Vertauschungen
  - Anzahl der Vergleiche

- Anzahl der Vergleiche dominiert die Anzahl der Vertauschungen, d.h. es werden deutlich mehr Vergleiche durchgeführt als Vertauschungen
  - → wir betrachten nur die Anzahl der Vergleiche

Analyse des InsertionSort Algorithmus (2)

- Wir müssen auf jeden Fall das Array einmal durchgehen (von 1 bis n-1; for-Schleife)
- Finden der Einfügeposition
  - Bester Fall:
    1 Vergleich
    (Liste ist bereits sortiert)
    → n-1 Vergleiche insgesamt (linear)
  - Schlechtester Fall:
    Einfügeposition ist immer am Anfang der Liste
    (Liste ist rückwärts sortiert)
    → in jedem Durchlauf müssen wir eine Position weitergehen:
    $1 + 2 + 3 + \cdots + n - 1 = \frac{n(n-1)}{2} < n^2$
    (quadratisch)

```
InsertionSort(A):
  for i = 1 to A.size-1 {
    e = A[i]
    j = i
    while (j > 0 && A[j-1] > e) {
       A[j] = A[j - 1]
        j = j - 1
    }
    A[j] = e
  }
```

Asymptotisches Laufzeitverhalten

- „Kleine“ Probleme (z. B. n = 5) sind uninteressant:
  - Die Laufzeit des Programms ist eher bestimmt durch die Initialisierungskosten (Betriebssystem, Programmiersprache etc.) als durch den Algorithmus selbst.
- Interessanter:
  - Wie verhält sich der Algorithmus bei sehr großen Problemgrößen?
  - Wie verändert sich die Laufzeit, wenn ich die Problemgröße variiere (z.B. Verdopplung der Problemgröße)?
  - Das bringt uns zum Konzept: asymptotisches Laufzeitverhalten

Definition $\mathcal{O}$-Notation

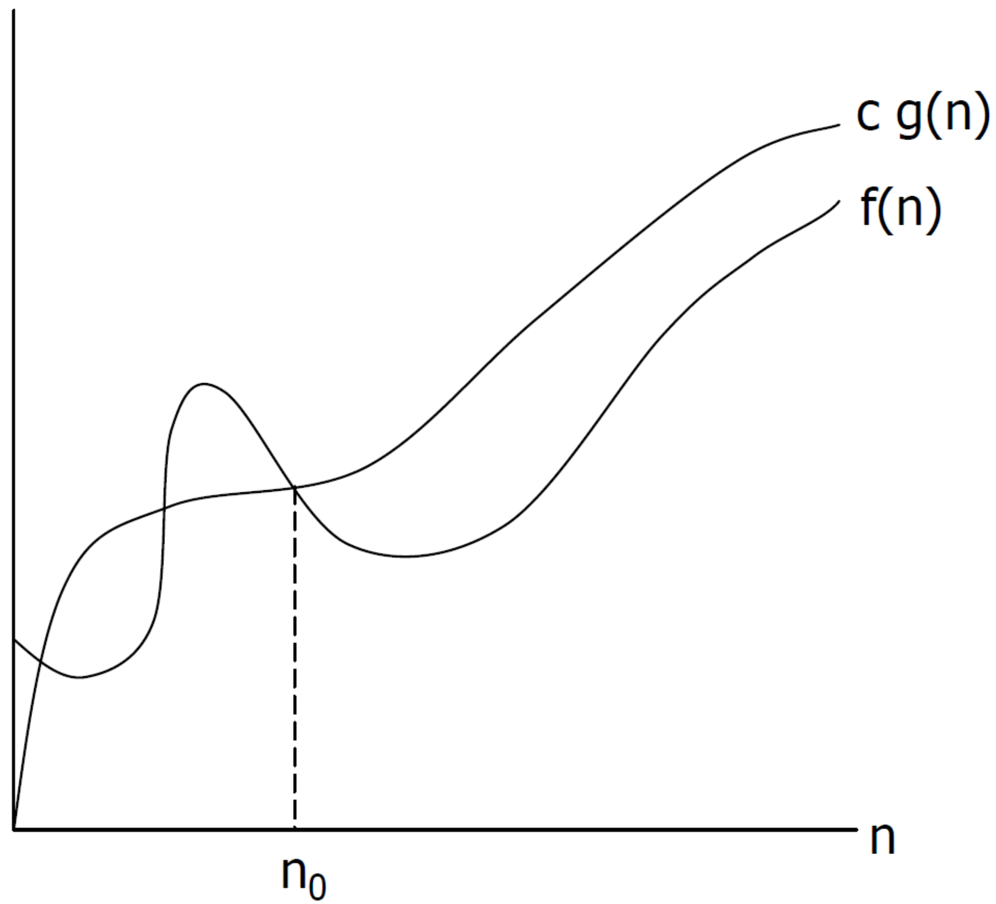
- Mit der $\mathcal{O}$-Notation haben Informatiker einen Weg gefunden, die asymptotische Komplexität (bzgl. Laufzeit oder Speicherplatzbedarf) eines Algorithmus zu charakterisieren.

- Definition $\mathcal{O}$-Notation:
  Seien $f: \mathbb{N} \to \mathbb{N}$ und $g: \mathbb{N} \to \mathbb{N}$ zwei Funktionen

  Die Funktion f ist von der Größenordnung $\mathcal{O}(g)$, geschrieben $\mathrm{f} \in \mathcal{O}(g)$, wenn es ein $n_0 \in \mathbb{N}$ und ein $\mathrm{c} \in \mathbb{N}$ gibt, so dass gilt: Für alle $\mathrm{n} \in \mathbb{N}$ mit $\mathrm{n} \geq n_0$ ist $f(n) \leq c \cdot g(n)$.
- Man sagt auch: f wächst höchstens so schnell wie g
- Man findet in der Literatur häufig auch $\mathrm{f} = \mathcal{O}(g)$ statt $\mathrm{f} \in \mathcal{O}(g)$

Veranschaulichung der $\mathcal{O}$-Notation

- Die Funktion f gehört zur Menge $\mathcal{O}(g)$, wenn es eine positive Konstante c gibt, so dass $f(n)$ ab n_0 unterhalb $c \cdot g(n)$ liegt.



„Rechnen“ mit der $\mathcal{O}$-Notation (1)

- Ziel: „ungefähres Rechnen in Größenordnungen“

- Elimination von Konstanten:
  - $2n \in \mathcal{O}(n)$
  - $\frac{n}{2} + 1 \in \mathcal{O}(n)$
- Bei einer Summe zählt nur der am stärksten wachsende Summand (d.h. mit dem höchsten Exponenten):
  - $2n^3 + 5n^2 + 5 \in \mathcal{O}(n^3)$
  - $n^2(n^2 + 2n) \in \mathcal{O}(n^4)$
  - $\mathcal{O}(1) \subseteq \mathcal{O}(\log n) \subseteq \mathcal{O}(n) \subseteq \mathcal{O}(\mathrm{n}\log n) \subseteq \mathcal{O}(n^2) \subseteq \mathcal{O}(n^3) \subseteq \cdots \subseteq \mathcal{O}(2^n)$
- Basis des Logarithmus ist unerheblich
  - $\mathcal{O}(\log_2 n) \subseteq \mathcal{O}(\log n)$
  - Basiswechsel ist Multiplikation mit einer Konstante:
$$\log_b n = \log_a n \cdot \log_b a$$

„Rechnen" mit der $\mathcal{O}$-Notation (2)

- Aufwand von „doSomething()" ist konstant, d.h. liegt in $\mathcal{O}(1)$

- ```
  for (i=0; i < 15; ++i)
      doSomething()
  ```
  $\Rightarrow \mathcal{O}(1)$

- ```
  for (i = 0; i < n; ++i)
      doSomething()
  ```
  $\Rightarrow \mathcal{O}(n)$

- ```
  for (i = 0; i < n; ++i)
      for (j = 0; j < n; ++j)
          doSomething()
  ```
  Innere Schleife: $\mathcal{O}(n)$; gesamt: $\mathcal{O}(n) * \mathcal{O}(n) \Rightarrow \mathcal{O}(n^2)$

- ```
  for (i = 0; i < n/2; ++i)
      doSomething()
  for (i = n/2; i < n; ++i)
      for (j = 0; j < n; ++j)
          doSomething()
  ```
  Erste Schleife: $\mathcal{O}(n)$; innere Schleife: $\mathcal{O}(n)$; zweite Schleife: $\mathcal{O}(n) * \mathcal{O}(n) = \mathcal{O}(n^2)$; gesamt: $\mathcal{O}(n) + \mathcal{O}(n^2) \Rightarrow \mathcal{O}(n^2)$

Wichtige Klassen von Funktionen

Komplexitätsklasse	Sprechweise	Typische Algorithmen/Operationen
$\mathcal{O}(1)$	konstant	Zuweisungen
$\mathcal{O}(\log n)$	logarithmisch	Suchen in sortierter Menge
$\mathcal{O}(n)$	linear	Lineare Suche
$\mathcal{O}(n \cdot \log n)$		gute Sortierverfahren
$\mathcal{O}(n^2)$	quadratisch	primitive Sortierverfahren
$\mathcal{O}(n^k), k > 1$	polynomiell	
$\mathcal{O}(2^n)$	exponentiell	Ausprobieren von Kombinationen

- Die $\mathcal{O}$-Notation hilft insbesondere bei der Beurteilung, ob ein Algorithmus für großes n noch geeignet ist bzw. erlaubt einen Effizienz-Vergleich zwischen verschiedenen Algorithmen für große n.
- Schlechtere als polynomielle Laufzeit gilt als nicht effizient, kann aber für viele Probleme das best-mögliche sein.

Skalierbarkeiten

- Annahme: 1 Rechenschritt ≙ 0.001 Sekunden → Maximale Eingabelänge bei gegebener Rechenzeit:

Laufzeit	1 Sekunde	1 Minute	1 Stunde
n	1000	60000	3600000
n log n	140	4895	204094
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

- Annahme: Wir können einen 10-fach schnelleren Rechner verwenden → Statt eines Problems der Größe p kann in gleicher Zeit dann berechnet werden:

Laufzeit T(n)	Neue Problemgröße
n	10p
n log n	fast 10p
n^2	3,16p
n^3	2,15p
2^n	3,32+p

Ausblick

- Analyse weiterer Algorithmen(klassen)
→ Vorlesung „Algorithmen und Datenstrukturen”
- „bessere“ Sortierverfahren
→ Vorlesung „Algorithmen und Datenstrukturen“
- Erforschung von oberen und unteren Schranken von Problemklassen
→ Theoretische Informatik

Übersicht

1. Sortieren
2. Laufzeitkomplexität
3. Suchen

Suchen

- Suchen ist eine der häufigsten Aufgaben in der Informatik

- Gegeben: Array a mit ganzen Zahlen; Element x
- Gesucht: An welcher Position kommt x in a vor?

- Idee:
  - Durchgehen von a von a[0] bis zum Ende
  - Finden wir ein a[i] == x, dann gib i zurück
  - Ansonsten geben wir -1 zurück

Sequentielles Suchen

```
public static int find(int[] a, int x) {
  for (int i = 0; i < a.length; ++i) {
    if (a[i] == x) {
      return i;
    }
  }
  return -1;
}
```

Beispiel: Sequentielle Suche

- Gesucht: 1

Index:	0	1	2	3	4	5
Feld:	5	4	9	1	8	2

a[0] == 1: nein

Beispiel: Sequentielle Suche

- Gesucht: 1

Index:	0	1	2	3	4	5
Feld:	5	4	9	1	8	2

a[1] == 1: nein

Beispiel: Sequentielle Suche

- Gesucht: 1

Index:	0	1	2	3	4	5
Feld:	5	4	9	1	8	2

a[2] == 1: nein

Beispiel: Sequentielle Suche

- Gesucht: 1

Index:	0	1	2	3	4	5
Feld:	5	4	9	1	8	2

a[3] == 1: ja

Sequentielle Suche

- Im Beispiel haben wir 4 Vergleiche benötigt

- Im schlimmsten Fall, benötigen wir bei einem Array der Länge n sogar *n* Vergleiche.

- Im Durchschnitt, wenn *x* im Array vorhanden ist, benötigen wir *(n+1)/2* Vergleiche.

→ $\mathcal{O}$(n)

- Um in einer bereits sortierten Array zu suchen, bietet sich die effiziente binäre Suche an

Binäre Suche

- Idee:
  - Ist das Feld bereits sortiert, können wir davon profitieren
  - Vergleiche x mit dem Wert, der in der Mitte steht.
  - Liegt Gleichheit vor, sind wir fertig.
  - Ist x kleiner, brauchen wir nur noch links weitersuchen.
  - Ist x größer, brauchen wir nur noch rechts weiter suchen.

- → binäre Suche

Binäre Suche: Prinzip

- Gesucht: 12

Index:	0	1	2	3	4	5	6	7	8	9
Feld:	1	2	4	12	13	14	16	19	22	29

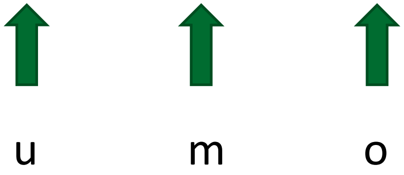
1. Schritt

u m o

Binäre Suche: Prinzip

- Gesucht: 12

Index:	0	1	2	3	4	5	6	7	8	9
Feld:	1	2	4	12	13	14	16	19	22	29



2. Schritt

Binäre Suche: Prinzip

- Gesucht: 12

Index:	0	1	2	3	4	5	6	7	8	9
Feld:	1	2	4	$\boxed{12}$	13	14	16	19	22	29

u m o

3. Schritt

Binäre Suche: Algorithmus (Pseudocode)

```
binarySearch(Array A, int x)
  u = 0
  o = A.size - 1
  while (u<o) {
    m = (u+o)/2
    if (a[m] == x) {
      return m;
    } else if (a[m] > x) {
      u = m + 1
    } else {
      o = m - 1
    }
  }
  return -1;
```

Binäre Suche: Analyse

- Aufwand für die Suche bei *n* Elementen
  - nach dem ersten Teilen der Folge: *n/2* Elemente
  - nach dem zweiten Schritt: *n/4*
  - nach dem dritten Schritt: *n/8*
  - …

  - Allgemein: nach dem i-ten Schritt: $n/(2^i)$
  - → Obere Schranke: $log_2\, n$

  - Worst Case = Average Case = $\log_2$ n Vergleiche → $\mathcal{O}(\log_2 n)$

Verfahren	n=10	n=100	n=1.000	n=10.000
Sequentiell	5	50	500	5000
Binär	3,3	6,6	9,9	13,3

Zusammenfassung

- Sortierverfahren
  - BubbleSort und InsertionSort

- Laufzeitkomplexität
  - groß $\mathcal{O}$-Notation
  - wort-case-Abschätzung von Algorithmen
  - Komplexitätsklassen
  - „Rechnen mit Größenordnungen“

- Suchverfahren
  - lineare und binäre Suche

Ausblick: Allgemeines Sortieren

- Zum Sortieren beliebiger Objekte kann die vordefinierte Schnittstelle Comparable genutzt werden:

```
public interface Comparable {
  int compareTo(Object other);
}
```

- Wenn die Klasse von o Comparable implementiert und o mit other vergleichbar ist, dann sollte gelten:
  - `o.compareTo(other) < 0`, falls o kleiner other
  - `o.compareTo(other) == 0`, falls o gleich other
  - `o.compareTo(other) > 0`, falls o größer als other

- Ersetze im Sortier- oder Suchalgorithmus `x < y` durch `x.compareTo(y) < 0`

- Beispiel: Die Klasse String implementiert dieses Interface.
  - Die Ordnung ist dabei die lexikographische Ordnung.
  - Der Ausdruck `"AAAaaa".compareTo("Test")` hat einen Wert < 0.

Prof. Dr. Sven Strickroth

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67
80538 München

Telefon: +49-89-2180-9300
sven.strickroth@ifi.lmu.de