DHBW Lörrach
Duale Hochschule Baden-Württemberg

Algorithmen und Komplexität
TIF 21 A/B
Dr. Bruno Becker

1. Einführung – Probleme und Algorithmen

Einführung - Probleme und Algorithmen

- **Definition Algorithmus, Datenstruktur**
- Eigenschaften eines Algorithmus
- Beispiel: Suchen in linearer Liste
- Binäre Suche

Was ist ein Problem in der Informatik?

- **Problem:** Klasse gleichartiger Aufgabenstellungen, mit
  - Zulässigen Inputs
  - Möglichen Outputs
  - Deren Beziehung (Relation)
- Probleme möglichst allgemein formulieren
  - *Finde Maximum in einer Folge von Werten*
  - *Finde kürzesten Weg von <Start> nach <Ziel>*
- **Probleminstanz:** Konkreter Satz von Input-Daten für ein Problem
- **Problemgröße:** Meistens Inputgröße

Algorithmus und Datenstruktur

- **Algorithmus:** Verfahren zur systematischen Lösung eines Problems
- **Datenstruktur:** Verfahren zur Speicherung von Informationen
- Algorithmus benötigt Zugriff auf Daten
- Wahl der Datenstrukturen oft entscheidend für Machbarkeit und Effizienz eines Algorithmus

➔ **Algorithmen und Datenstrukturen hängen eng zusammen**

Probleme und Algorithmen

- Definition Algorithmus, Datenstruktur
- **Eigenschaften eines Algorithmus**
- Beispiel: Suchen in linearer Liste
- Binäre Suche

Eigenschaften eines Algorithmus

- **Korrektheit:** Algorithmus löst Problem (für alle zulässigen Inputs)
  - *Navigation führt immer mit <schnellster, kürzester,..> Route zum <Ziel> …*
  - *… oder meldet, dass es keinen Weg mit dem <Verkehrsmittel> dorthin gibt*
- **Robust:**
  - Funktioniert auch für falsche oder unvollständige Inputs
- **Effizienz:** Möglichst geringe Kosten (in Abhängigkeit von Input)
  - Laufzeit -> Schnell (z.B. *Navigation in Deutschland innerhalb 30 sec.)*
    -> Geringe Anzahl von Rechenoperationen
  - Speicherplatz

Beschreibung von Algorithmen

- **Pseudo-Code:** *Algorithmus* ist *Idee* eines Programms, nicht vollständiges, syntaktisch und semantisch korrektes Programm
  - Endliche Folge eindeutiger Anweisungen
  - Prinzipiell maschinenmäßig ausführbar

- Implementierung eines Algorithmus in Programmiersprache (Java, Python, C++, …)

Fragestellungen zu Algorithmen

- Ist ein Problem (immer) algorithmisch lösbar?
  - Wie schwierig ist ein Problem?
  - Welchen Aufwand benötigt Algorithmus?
  - Gibt es einen „besten“ Algorithmus für ein Problem?
- Analyse von Algorithmen
  - Korrektheit (für jeden Input)?
  - Ressourcenverbrauch?; Geht es besser?
- Entwurf von Algorithmen
  - Designmuster?
  - Welche Datenstrukturen?

Probleme und Algorithmen

- Definition Algorithmus, Datenstruktur
- Eigenschaften eines Algorithmus
- **Beispiel: Suchen in linearer Liste**
- Binäre Suche

Suchen in linearer Liste

- Input: Zahlenfolge $a_0, a_1, a_2, \ldots, a_{n-1}$, Zahl X
- Output: Der Index i, falls $a_i = X$; *sonst -1* (Fehler/Exception)

Idee: Vergleiche X nacheinander mit $a_0, a_2, \ldots, a_{n-1}$ und stoppe,
falls $a_i = X$ gefunden. Gebe i zurück. Wenn am Ende der Folge nicht gefunden, gebe -1 zurück

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

➔Ist dieser Algorithmus „gut“? Geht es besser?

➔Übung: Variante *Finde alle i mit a[i]=x*

Analyse von Algorithmen

- **Ziel**: Kosten (Ressourcenverbrauch) des Algorithmus in Abhängigkeit von Problemgröße und Input bestimmen

- **Empirisch**: Implementieren und Messen…
- **Theoretisch**: Zähle Operationen/Speicher auf Basis eines vereinfachten Maschinenmodells
  - Unabhängig von konkreten Implementierungen und Maschinen
  - Für alle Inputs
  - Maschinenmodell realistisch aber doch vereinfacht (z.B. zählen nur gewisse Operationen)

Analyse der linearen Suche

- **Maschinenmodell:** Zähle nur Vergleichsoperationen
- **Input:** Länge *n* der Folge

- **Ergebnis:**
  - **Best Case (Bester Fall):** ***1***
  - **Average Case (Durchschnitt):** ***n/2*** für erfolgreiche Suche
  - **Worst Case (Schlimmster Fall):** ***n*** für erfolglose Suche oder letztes Element

Average Case - Analyse

- **…ist oft schwierig**

- **Durchschnitt über**
  - Welche Inputs?
  - Welche Wahrscheinlichkeitsverteilung?
  - Geschlossene Formeldarstellung? (Alternative Simulation)

Untere Schranken für Aufwandsbetrachtung

- Gibt es für Suchproblem einen im *worst case* besseren Algorithmus?
- **NEIN!**

Beweis durch Widerspruch:

Angenommen, es gäbe Algorithmus A, der nach max. *n-1* Vergleichen Index findet

Verwende Folge, für die a_i <> X für alle i, für die A Vergleich durchführt und a_j = X für ein j, für dessen Feld A *keinen* Vergleich durchführt

→A findet X nicht, ist also nicht korrekt

→ So einen Algorithmus A gibt es nicht

Probleme und Algorithmen

- Definition Algorithmus, Datenstruktur
- Eigenschaften eines Algorithmus
- Beispiel: Suchen in linearer Liste
- **Binäre Suche**

Binäre Suche

- **Suche in sortierter Zahlenfolge ist deutlich einfacher!**
- **Input:** $a_1 \leq a_2 \leq \ldots \leq a_n$ **, X**
- **Output:** Ein Index **i** mit a_i**=X**

- **Idee:** Suche Element in Mitte der Folge. Wert **M.**

  X=M → Fertig

  X< M → Suche in linker Hälfte

  X> M → Suche in rechter Hälfte

Binäre Suche

- **Algorithmus** nach der „Divide & Conquer“-Strategie
- **Rekursive Variante** (Funktion ruft sich selbst wieder auf)
- **Geht auch iterativ (**Schrittweise innerhalb der Funktion)

Binäre Suche – Iterativer Algorithmus

```
int binarysearch (int[] a, int x) {
   int unten = 0;
   int oben = a.length-1;
   while (unten <= oben) {
            mitte = (unten + oben)/2;
            if (x < a[mitte])  // suche links
                 oben = mitte -1;
            else if (x >  a[mitte])  // suche rechts
                 unten  = mitte + 1;
            else
                 return mitte; // gefunden
    }
    return -1 // nicht gefunden
}
```

Übung: Rekursiver Algorithmus

Binäre Suche – Rekursiver Algorithmus

```
int binarysearch (int[] a, int unten, int oben,  int x) {
   if (unten > oben)
     return -1; // nicht gefunden
   mitte = (unten + oben)/2;
   if (x < a[mitte])  // suche links
      return binarysearch(a,unten,mitte-1,x)
    if (x > a[mitte])  // suche rechts
      return binarysearch(a,mitte+1, oben, x)
    // sonst gefunden
                return mitte;
}
```

Initialer Aufruf : binarysearch(a,0,a.length-1,x)

Rekursion nach dem ***Divide & Conquer-Prinzip (Teile & Herrsche)***

- Teile Problem solange, bis Lösung offensichtlich („beherrschbar“)
- Konstruiere dann Lösung aus der Lösung der Teilprobleme

Binäre Suche - Aufwandsbetrachtung

Kosten: Vergleich zählt eine Einheit

Folge der Länge $N = 2^n-1$:

$n=1$; $T(1) = 1$

$n=2$; $T(3) \leq 2$

$n=3$; $T(7) \leq ?$

Allgemein: $T(2^n-1) \leq T(2^{n-1}-1) +1$

➔ $T(2^n-1) \leq n +1$

➔ Binäre Suche benötigt $\log_2 N +1$ Schritte

Kein relevanter Unterschied zwischen rekursiver- und iterativer Lösung

Binäre Suche

- **Vergleich mit linearer Suche**

N	Lineare Suche	Binäre Suche
4	4	3
16	16	5
1.024	1.024	11
1.048.576	1.048.576	21
1.073.741.824	1.073.741.824	31

- **Besser als mit $\log_2 N$ Schritten geht Suche nicht**