Algorithmen & Datenstrukturen

Patrick Bucher

Contents

1 Einführung

1.1 Was ist ein Algorithmus?

Ein Algorithmus ist ein präzise festgelegtes *Verfahren zur Lösung eines Problems* bzw. einer Problemklasse; ein Lösungsverfahren (Rezept, Anleitung).

Eigenschaften eines Algorithmus:

1. schrittweises Verfahren
2. ausführbare Schritte
3. eindeutiger nächster Schritt (determiniert)
4. endet nach endlich vielen Schritten (terminiert)

Beispiele für Algorithmen:

- Berechnung des ggT
- Zeichnen von Verbindungslinien
- Sortierung von Zahlen
- Finden des kürzesten Weges zwischen zwei Punkten
- Primzahltest
- Berechnung eines Integrals
- Finden einer Lösung in einem Lösungsraum

Themenbereiche von Algorithmen:

- Algorithmentheorie: Finden guter Lösungsalgorithmen für bestimmte Problemstellungen
- Komplexitätstheorie: Ressourcenverbrauch (Rechenzeit, Speicherbedarf)
- Berechenbarkeitstheorie: Was ist mit einer Maschine lösbar/nicht lösbar?

1.2 Was ist eine Datenstruktur?

Eine Datenstruktur ist ein Konzept zur *Speicherung und Organisation von Daten.* Sie ist durch die Operationen charakterisiert, welche Zugriffe und Verwaltung realisieren.

Beispiele für Datenstrukturen:

- Array: direkter Zugriff, fixe Grösse
- Liste: sequentieller Zugriff, flexible Grösse

1.3 Was ist ein Programm?

Ein Programm kombiniert Algorithmen und Datenstrukturen.

Der Ressourcenverbrauch eines Algorithmus (Laufzeit und Speicherbedarf) hängt von der Verwendung geeigneter Datenstrukturen ab.

Algorithmen operieren auf Datenstrukturen und Datenstrukturen bedingen spezifische Algorithmen.

1.4 Fragen betreffend Algorithmen/Datenstrukturen

1. Für kleine oder grosse Probleme adäquat?

2. Selber entwickeln oder aus einer Bibliothek?
3. Einfach oder schwierig zu verstehen, implementieren und warten?
4. Geringe Laufzeit mit grossem Speicherbedarf oder umgekehrt?

1.5 Beispiel: Berechnung des ggT

Gegeben sind zwei Zahlen, A und B.

1. A sei die grössere der beiden Zahlen (andernfalls tauschen).
2. Setze `A = A - B`
3. Wenn `A != B`: Schritt 1, wenn `A = B`: Fertig

Iterative Lösung (mit impliziter Vertauschung):

```
public static int ggT(int a, int b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

Iterative Lösung (mit Modulo-Operator "abgekürzt"):

```
public static int ggT(int a, int b) {
    while ((a != b) && (b != 0)) {
        if (a > b) {
            a = a % b;
        } else {
            b = b % a;
        }
    }
    return (a + b);
}
```

Rekursive Lösung:

```
public static int ggT(int a, int b) {
    if (a > b) {
        ggT(a - b, b);
    } else {
        if (a < b) {
            return ggT(a, b - a);
```

```
        } else {
            return a;
        }
    }
}
```

1.6 Gleichwertigkeit

- Alle Implementierungen führen zum Ziel und liefern die gleichen Resultate. Sie sind *gleichwertig*.
- Die Anzahl Schleifendurchläufe, arithmetische Operationen und Methodenaufrufe – und somit Laufzeit und Speicherbedarf – unterscheiden sich jedoch.

1. Zu jeder Problemklasse gibt es verschiedene konkrete Probleme.
2. Zu jedem konkreten Problem gibt es verschiedene Algorithmen.
3. Zu jedem Algorithmus gibt es verschiedene Implementierungen.

Die Gleichwertigkeit von Algorithmen kann nicht bewiesen werden (Halting-Problem).

1.7 Komplexität

- Die Komplexität (oder Aufwand, Kosten) eines Algorithmus besagt, wie der Ressourcenbedarf von den Eingabedaten abhängt.
- Der Ressourcenbedarf ist eine Funktion der Eingabedaten: `R=f(E)`
  - Rechenzeit: *Zeitkomplexität*
  - Speicherbedarf: *Speicherkomplexität*
- Abhängigkeit von Eingabedaten:
  - Grösse der *Datenmenge* (z.B. Anzahl zu sortierender Elemente)
  - Grösse der *Datenwerte* (z.B. Grössse zu prüfender Primzahlen)

Bei der Komplexität eines Algorithmus geht es nicht um die exakte Rechenzeit (einer Implementierung), sondern um das *Anwachsen des Ressourcenbedarfs* in Abhängigkeit von wachsenden Eingabedaten.

1.7.1 Beispiel

Laufzeit eines Programms abhängig vom Eingabeparameter `n`:

```
public static void task(int n) {
    task1();
    task1();
    task1();
    task1();
```

```
    for (int i = 0; i < n; i++) {
        task2();
        task2();
        task2();
        for (int j = 0; j < n; j++) {
            task3();
            task3();
        }
    }
}
```

Annahmen:

- Die Methoden `task1`, `task2` und `task3` haben eine konstante und vergleichbare Rechenzeit und sind *nicht* vom Eingabeparameter `n` abhängig.
- Die Ausführungszeiten der Schleifensteuerung sind vernachlässigbar.

Berechnung: Die Rechenzeit `T` von `task(n)` beträgt $T=f(n)\sim 4+3n+2n^2$

Folgerung:

- Für grosse `n` dominiert der Anteil von n^2.
- Die Funktion ist von der Ordnung $O(n^2)$

1.7.2 Big-O-Notation

- `O`, das landausche Symbol, bringt zum Ausdruck, dass eine Funktion `f(n)` höchstens so schnell wächst wie eine andere Funktion `g(n)`.
- Wird `n` genügend gross gewählt, unterscheiden sich `f(n)` und `g(n)` nur noch durch eine Konstante.

Wichtige Ordnungsfunktionen:

Bezeichnung	Notation	Beispiele
Konstant	`O(1)`	Hashing
Logarithmisch	`O(ln(n))`	binäres Suchen
Linear	`O(n)`	Suchen in Text
`n*log(n)`	`O(n*log(n))`	schlaues Sortieren
Polynominal	`O(n^m)`	einfaches Sortieren
Exponential	`O(m^n)`	Optimierungen
Fakultät	`O(n!)`	Permutationen, Travelling Salesman

1.7.3 Abschliessende Bemerkungen

- Die Ordnung macht keine Aussage über das Verhalten bei kleinen `n`.
- Konstante Faktoren können bei kleinen `n` relevant sein.
- Die exakte mathematische Analyse vieler Algorithmen ist schwierig oder sogar unmöglich.
- Bei der Analyse muss darum differenziert werden:
  - bester Fall (best case)
  - schlechtester Fall (worst case)
  - mittlerer Fall (average case)

2 Rekursion

Viele Algorithmen und Datenstrukturen sind von Natur aus *selbstähnlich* bzw. *selbstbezüglich*.

- Der ggT von 21 und 15 ist gleich dem ggt von 21-15 und 15.
- Ein Verzeichnis enthält Daten und andere Verzeichnisse.
- Ein Ausschnitt einer Matrix, einer Liste, eines Baumes, eines Graphen ist wiederum eine Matrix, eine Liste, ein Baum, ein Graph.

2.1 Rekursion vs. Iteration (am Beispiel der Fakultät)

2.1.1 Iterativer Ansatz

Iterative Definition: `n! = 1*2*3*...*n`

Iterative Implementierung:

```
public static factorial(int n) {
    int result = 1;
    for (int i = 0; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

2.1.2 Rekursiver Ansatz

Rekursive Definition:

- Rekursionsbasis:
  - `0! = 1`
  - `1! = 1`
- Rekursionsvorschrift: `n!=n*(n-1)!`

- Rekursion: Aufzeigen eines Lösungsweges, wie ein schwieriges Problem auf ein gleichartiges aber einfacheres Problem zurückgeführt werden kann.

Beispiel: `5!` wird auf die Rekursionsbasis zurückgeführt:

```
- `5!=5*4!`
- `5!=5*(4*3!)`
- `5!=5*(4*(3*2!))`
- `5!=5*(4*(3*(2*1!)))=5*(4*(3*(2*1)))=120`
```

Rekursive Implementierung:

```
public static factorial(int n) {
    if (n == 0 || n == 1) {
        // Iterationsbasis
        return 1;
    } else {
        // Rekursionsvorschrift (Rückführung)
        return n * factorial(n - 1);
    }
}
```

2.2 Mächtigkeit der Rekursion

- Rekursion und Iteration sind *gleich mächtig*.
- Die Menge der berechenbaren Problemstellungen bei Verwendung von Rekursion und Iteration ist gleich.
- Eine iterative Implementierung lässt sich immer in eine rekursive Implementierung überführen (und umgekehrt).
- Vorteile der Rekursion:
  - Rekursive Ansätze sind oft einfach und elegant.
  - Die Korrektheit rekursiver Definitionen lässt sich oft einfacher aufzeigen.
  - Es gibt rein rekursive Programmiersprachen, z.B. LISP und Prolog
- Nachteile der Rekursion:
  - Es ergeben sich schnell sehr viele Methodenaufrufe.
  - Die Programmausführung ist dadurch tendenziell langsamer.
  - Es besteht grosser Speicherbedarf auf dem Call-Stack und die Gefahr für einen Stack Overflow.

2.2.1 Beispiel: Fibonacci-Zahl

- Fibonacci-Zahlen sind in der Mathematik rekursiv definiert:
  - `f(0)=0`
  - `f(1)=1`

– `f(n)=f(n-1)+f(n-2)`, für `n>=2`

Rekursive Implementierung:

```
public int fib(n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

Probleme einer rekursiven Implementierung:

- Viele rekursive Aufrufe: Komplexitätsklasse `O(2^n)`.
- Mehrfache Berechnung der gleichen Zahlen. (Lösungsansatz: Caching)
- Besser eine iterative Lösung finden!
- *Nicht jedes Problem, das sich rekursiv präsentiert, sollte rekursiv umgesetzt werden!*

2.3 Typen von Rekursion

- linear vs. nichtlinear:
  - *lineare Rekursion*: eine Methode ruft sich intern selber auf (z.B. Fakultät)
    - `m() -> m()`
  - *nichtlineare Rekursion*: ein Methodenaufruf führt zu mehreren rekursiven Aufrufen (z.B. Fibonacci)
    - nicht geschachtelt (Fibonacci): *primitiv rekursiv*, `m() -> m(); m();`
    - geschachtelt: *nicht primitiv rekursiv*, `m() -> m(m())`
- direkt vs. indirekt:
  - *Direkte Rekursion*: eine Methode ruft sich *direkt* selber auf.
  - *Indirekte Rekursion*: eine rekursive Methode ruft sich *indirekt* selber auf, `m() -> m(n(m()))`

2.4 Beispiel: Permutation

Ein sechs Zeichen langes Passwort bestehend aus den Buchstaben von `A` bis `F`, wobei jeder Buchstabe nur einmal vorkommt, ist gesucht. Gefragt sind die *Permutationen* (alle möglichen Kombinationen) dieser Buchstaben. Es gibt `6!=720` mögliche Kombinationen:

- `ABCDEF`
- `BACDEF`
- ...
- `FEDCBA`

Lösungsansatz: Rückführung des Problems (Permutation von `n` Zeichen auf Permutation von einem Zeichen)!

- `A` kombiniert mit den Permutationen von `BCDEF`
  - `B` kombiniert mit den Permutationen von `CDEF`
    - `C` kombiniert mit den Permutationen von `DEF`

TODO: Beispielimplementierung

3 Glossar

- Algorithmus: präzise festgelegtes Verfahren zur Lösung eines Problems bzw. einer Problemklasse; ein Lösungsverfahren (Rezept, Anleitung)-Operator)
- Datenstruktur: ein Konzept zur *Speicherung und Organisation von Daten*. Sie ist durch die Operationen charakterisiert, welche Zugriffe und Verwaltung realisieren.
- Rekursion: Aufzeigen eines Lösungsweges, wie ein schwieriges Problem auf ein gleichartiges aber einfacheres Problem zurückgeführt werden kann.