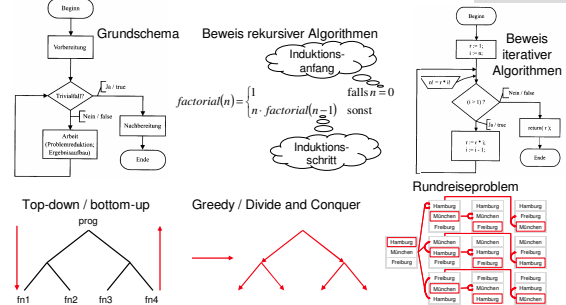


Suchen

Thomas Röfer

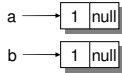
Flache/Tiefe Gleichheit
Lineare Suche
Binäre Suche
Asymptotische Komplexität

Rückblick „Algorithmenkonstruktion“



Gleichheit

- Identität**
 - Es handelt sich um dasselbe Objekt
 - In Java: == für Referenzen
- Flache Gleichheit**
 - Alle Elemente zweier Objekte sind „flach“ gleich
 - In Java: == für alle Elemente der Objekte

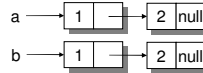


```
class List
{
    List next;
    int value;

    boolean shallowEquals(List other)
    {
        return other != null &&
            value == other.value &&
            next == other.next;
    }
}
```

Gleichheit

- Tiefe Gleichheit**
 - Alle Elemente zweier Objekte sind „tief“ (rekursiv) gleich
 - In Java: equals() auf Objekten, wenn equals() für alle beteiligten Klassen geeignet definiert ist
 - equals() in Klasse Object entspricht nur ==, muss also überschrieben werden



```
class List
{
    List next;
    int value;

    boolean equals(List other)
    {
        return other != null &&
            value == other.value &&
            (next == other.next ||
            next != null &&
            next.equals(other.next));
    }
}
```

Tiefe Kopie

- Ziel**
 - Erstellen einer vollständigen Kopie, die unabhängig vom Original ist
- In Java**
 - Die Klasse Object enthält bereits eine Funktion clone() zu diesem Zweck
 - Sie erzeugt aber nur eine CloneNotSupportedException, es sei denn, eine Klasse implementiert die Schnittstelle Cloneable
- Implementierung**
 - Cloneable implementieren
 - In clone()
 - Neues Objekt erstellen
 - Alle Basistypen-Attribute kopieren
 - Für alle Objekt-Attribute wiederum clone() aufrufen

```
class List implements Cloneable
{
    List next;
    int value;

    public Object clone()
    {
        List copy = new List();
        copy.value = value;
        if(next != null)
            copy.next = (List) next.clone();
        return copy;
    }
}
```

Lineare Suche

- Voraussetzungen**
 - Suche in Reihung oder Liste
 - Gleichheit der Elemente muss definiert sein
- Ansatz**
 - Vergleiche Suchwert der Reihe nach mit allen Elementen der Reihung/Liste
- Komplexität**
 - Bester Fall: Erstes Element ist das gesuchte
 - Schlechtester Fall: Das gesuchte Element ist nicht vorhanden
 - Im Mittel wird die halbe Reihung oder Liste durchsucht

```
int linearSearch(Object[] a, Object key)
{
    int i = 0;
    while(i < a.length &&
        !a[i].equals(key))
        ++i;
    return i == a.length ? -1 : i;
}
```

Lineare Suche mit Wächter/Stopper

- Ansatz
 - Gesuchtes Element wird ans Ende der Reihung/Liste gestellt, damit es auf jeden Fall gefunden wird
 - Dadurch entfällt der Vergleich mit der Länge der Reihung/Liste
- Nachteile
 - Vorbereitung dauert länger
 - Für Listen nur sinnvoll, wenn man direkten Zugriff auf das Ende hat
 - Reihung/Liste wird zeitweilig verändert, mögliche Probleme bei paralleler Ausführung

```
int guardSearch(Object[] a, Object key)
{
    Object last = a[a.length - 1];
    if (last == key)
        return a.length - 1;
    else
    {
        a[a.length - 1] = key;
        int i = 0;
        while (!a[i].equals(key))
            ++i;
        a[a.length - 1] = last;
        return i == a.length - 1 ? -1 : i;
    }
}
```

Lineare Suche ohne Längentest

- Grund
 - Java prüft ohnehin die Gültigkeit eines Index bzw. einer Referenz
- Ansatz
 - Nicht auf Länge testen
 - Java-Ausnahme fangen
- Nachteile
 - Erzeugung und Fangen von Ausnahme kostet Zeit

```
int exceptionSearch(Object[] a, Object key)
{
    try
    {
        int i = 0;
        while (!a[i].equals(key))
            ++i;
        return i;
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
        return -1;
    }
}
```

Ordnungsrelationen

- Definition
 - Eine Ordnungsrelation definiert eine Ordnung zwischen Elementen eines Typs
 - Typische Ordnungsrelationen sind $<$ und $>$
 - Ordnungsrelationen sind transitiv:
 - $a > b \wedge b > c \rightarrow a > c$
 - Dadurch können Werte auch implizit miteinander verglichen werden
 - Eine Ordnung kann über allen Typen definiert werden, ist aber nicht immer intuitiv
 - Apfel > Birnen?
- In Java
 - Operatoren $<$, $>$, $<=$, $>=$ für alle Basistypen außer boolean
 - Methode `compareTo()` in Schnittstelle `Comparable`

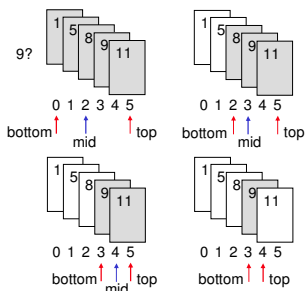
```
void test(Comparable a, Comparable b)
{
    switch (a.compareTo(b))
    {
        case -1:
            System.out.println("a < b");
            break;
        case 0:
            System.out.println("a == b");
            break;
        case 1:
            System.out.println("a > b");
            break;
    }
}
```

Binäre Suche

- Voraussetzungen
 - Suche in Reihung
 - Ordnungsrelation muss für Elemente definiert sein
 - Elemente müssen entsprechend der Ordnungsrelation sortiert sein
- Ansatz
 - Divide and Conquer
 - Durch Sortierung muss nicht mit allen Elementen verglichen werden

```
int binarySearch(Comparable[] a, Comparable key)
{
    if (a.length == 0)
        return -1;
    else
    {
        int bottom = 0,
            top = a.length;
        while (bottom + 1 != top)
        {
            int mid = (top + bottom) / 2;
            if (a[mid].compareTo(key) > 0)
                top = mid;
            else
                bottom = mid;
        }
        return a[bottom].equals(key)
            ? bottom : -1;
    }
}
```

Binäre Suche – Beispiel



```
int binarySearch(Comparable[] a, Comparable key)
{
    if (a.length == 0)
        return -1;
    else
    {
        int bottom = 0,
            top = a.length;
        while (bottom + 1 != top)
        {
            int mid = (top + bottom) / 2;
            if (a[mid].compareTo(key) > 0)
                top = mid;
            else
                bottom = mid;
        }
        return a[bottom].equals(key)
            ? bottom : -1;
    }
}
```

Komplexität

- Definition
 - Die Komplexität eines Algorithmus ist die Anzahl der benötigten Programmschritte (der *Aufwand*) in Abhängigkeit von der *Größe* der Eingabe
- Arten
 - Zeitkomplexität
 - ist üblicherweise mit „Komplexität“ gemeint
 - Platzkomplexität
- Zu betrachtende Fälle
 - Schlechtester Fall (worst case)
 - Bester Fall (best case)
 - Durchschnittlicher Fall (average case)

```
int sqr(int x)
{
    return x * x;
}

int factorial(int n)
{
    return n > 0 ? factorial(n - 1) : 1;
}
```

Exakte Bestimmung – Multiplikationen

	Z(n)	V(n)	M(n)	I(n)
1	2	1	0	0
2	3	3	0	1
3	5	6	1	3
4	8	10	3	6
5	12	15	6	10
6	17	21	10	15
7	23	28	15	21
8	30	36	21	28
9	38	45	28	36
10	47	55	36	45

```
int f1(int n)
{
    int res = 1;
    for(int j = 1; j < n; ++j)
        for(int i = 1; i < j; ++i)
            res = res * i;
    return res;
}
```

$$\begin{aligned}
 M(n) &= (n-2) + M(n-1) \\
 &= (n-2) + (n-3) + M(n-4) \\
 &= \sum_{k=1}^{n-2} k \\
 &= \frac{(n-1)(n-2)}{2}
 \end{aligned}$$

Exakte Bestimmung – Inkremente

	Z(n)	V(n)	M(n)	I(n)
1	2	1	0	0
2	3	3	0	1
3	5	6	1	3
4	8	10	3	6
5	12	15	6	10
6	17	21	10	15
7	23	28	15	21
8	30	36	21	28
9	38	45	28	36
10	47	55	36	45

```
int f1(int n)
{
    int res = 1;
    for(int j = 1; j < n; ++j)
        for(int i = 1; i < j; ++i)
            res = res * i;
    return res;
}
```

$$\begin{aligned}
 I(n) &= (n-2) + 1 + I(n-1) \\
 &= \sum_{k=1}^{n-1} k \\
 &= \frac{n(n-1)}{2}
 \end{aligned}$$

Exakte Bestimmung – Zuweisungen

	Z(n)	V(n)	M(n)	I(n)
1	2	1	0	0
2	3	3	0	1
3	5	6	1	3
4	8	10	3	6
5	12	15	6	10
6	17	21	10	15
7	23	28	15	21
8	30	36	21	28
9	38	45	28	36
10	47	55	36	45

```
int f1(int n)
{
    int res = 1;
    for(int j = 1; j < n; ++j)
        for(int i = 1; i < j; ++i)
            res = res * i;
    return res;
}
```

$$\begin{aligned}
 Z(n) &= 1 + 1 + I(n) \\
 &= 2 + \frac{n(n-1)}{2}
 \end{aligned}$$

Exakte Bestimmung – Vergleiche

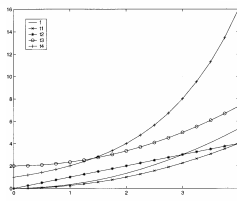
	Z(n)	V(n)	M(n)	I(n)
1	2	1	0	0
2	3	3	0	1
3	5	6	1	3
4	8	10	3	6
5	12	15	6	10
6	17	21	10	15
7	23	28	15	21
8	30	36	21	28
9	38	45	28	36
10	47	55	36	45

```
int f1(int n)
{
    int res = 1;
    for(int j = 1; j < n; ++j)
        for(int i = 1; i < j; ++i)
            res = res * i;
    return res;
}
```

$$\begin{aligned}
 V(n) &= n + V(n-1) \\
 &= \sum_{k=1}^n k \\
 &= \frac{n(n+1)}{2} \\
 G(n) &= M(n) + I(n) + V(n) + Z(n)
 \end{aligned}$$

Asymptotische Komplexität

- Definition
 - Sei $f : \mathbb{N} \rightarrow \mathbb{R}^+$
 - Die Ordnung von f ist die Menge $O(f(n)) = \{t : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : t(n) \leq c \cdot f(n)\}$
 - $O(f(n))$ charakterisiert das Wachstum von f
 - Sie enthält alle Funktionen, deren Graph maximal so stark wächst wie der Graph von f
 - bis auf Umbezeichnung der Einheiten auf der y-Achse
- Beispiel
 - $f(n) = 1/3 n^2$
 - $t_1(n) = 1/4 n^2 \in O(f(n)), n_0 = 1, c = 1$
 - $t_2(n) = n \in O(f(n)), n_0 = 3, c = 1$
 - $t_3(n) = 1/3 n^2 + 2 \in O(f(n)), n_0 = 3, c = 2$
 - $t_4(n) = 2^n \notin O(f(n))$



Beweise

- $n^2 \in O(n^3)$?
 - Zu zeigen: $\exists c \in \mathbb{R}^+$ und $\exists n_0 \in \mathbb{N}$ so dass für alle $n > n_0$ gilt: $n^2 \leq c \cdot n^3$
 - Dies gilt z.B. ab $c = 1, n_0 = 1$
- $n^3 \in O(n^2)$?
 - Zu zeigen: $\exists c \in \mathbb{R}^+$ und $\exists n_0 \in \mathbb{N}$ so dass für alle $n > n_0$ gilt: $n^3 \leq c \cdot n^2$
 - Dies erfordert ein c , so dass für alle n gilt: $n \leq c$
 - Ein solches c kann es nicht geben, daher gilt: $n^3 \notin O(n^2)$
- Weitere Aussagen
 - $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$
 - $O(f(n)) \subseteq O(g(n)) \Leftrightarrow f(n) \in O(g(n))$
 - $O(f(n)) = O(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge g(n) \in O(f(n))$
 - $O(f(n)) \subset O(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge g(n) \notin O(f(n))$

Typische Aufwandsklassen

- ▶ **$O(1)$**
 - ▶ Z.B. `sqr()`
- ▶ **$O(\log n)$**
 - ▶ Z.B. `binarySearch()`, $n = a.length$
- ▶ **$O(n)$**
 - ▶ Z.B. `linearSearch()`, $n = a.length$
- ▶ **$O(n \log n)$**
 - ▶ Sortieren mit Divide and Conquer
- ▶ **$O(n^2)$, $O(n^3)$...**
 - ▶ f1 (Folie 13ff)
- ▶ **$O(2^n)$, $O(3^n)$, ...**
- ▶ **$O(n!)$**
 - ▶ Rundreiseproblem