Tutoriumsvorbereitung 3

Prof. Dr. Olaf Hellwich und Mitarbeiter

07.05.2018 - 11.05.2018

Inhaltsverzeichnis

0 Organisatorisches

Der abschließende Online-Test wird am **Montag, den 14.05.2018 um 10:00 Uhr** freigeschaltet und läuft bis **Sonntag, den 20.05.2018 um 23:59 Uhr**.

1 Suche

1.1 Motivation

Ein grundlegendes Problem der Informatik ist die Handhabung großer Datenmengen. Diese werden zumeist in Datenbanken abgelegt. Dabei kommt es auf eine gezielte und effiziente Verwaltung der Daten an. In diesem Kontext muss man sich der Auswahl geeigneter Suchalgorithmen stellen. In diesem Tutorium werden wir zwei Suchalgorithmen miteinander vergleichen.

1.2 Iterative Suche

Algorithmus

- Das Array wird von links nach rechts durchgegangen.
- Jeder Slot wird mit dem gesuchten Element verglichen.
- Die Suche endet erfolgreich sobald das Element gefunden ist.

Handsimulation

Aufgabe: Gegeben sei folgendes Array:

3	5	7	9	12	16	19	24	36	72

Finden Sie die 7 mittels iterativer Suche.

Lösung:

Beginnen wir nun mit der Suche nach dem Element 7:

3	5	7	9	12	16	19	24	36	72

⇓

3	5	7	9	12	16	19	24	36	72

⇓

3	5	7	9	12	16	19	24	36	72

⇓

3	5	7	9	12	16	19	24	36	72

Das Element 7 wurde am Index 2 gefunden.

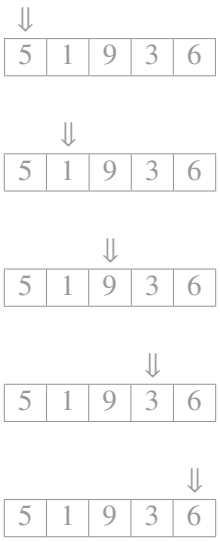
Anmerkungen

Aufgabe: Kann man die iterative Suche nach der 7 auf das folgende Array anwenden? Und wenn ja was ist das Ergebnis?

5	1	9	3	6

Lösung:

- Ja, die iterative Suche kann auf das unsortierte Array angewendet werden.
- Die Lösung wird auf folgendem Weg erhalten:



- Das Element ist nicht enthalten, da das Array passiert wurde, ohne dass das Element gefunden wurde.

Umsetzung in Java

Dazu ist zu überlegen, was als Parameter übergeben werden soll, wie lange die Suche laufen soll, was überprüft wird und was sinnvolle Rückgabewerte der Methode sind (z.B. bei nicht erfolgreicher Suche).

Aufgabe: Geben Sie die nötigen Teile des Quellcodes an.

Lösung:

Methodenstruktur:

- Rückgabetyp: int (für den Index, an dem wir das Element gefunden haben)
- Parameter: Wir müssen das gesuchte Element übergeben.
- Mit einer for-Schleife durchlaufen wir das Array und überprüfen für jeden Durchgang, ob der Inhalt des aktuellen Slots mit dem gesuchten Element übereinstimmt.
- Finden wir eine Übereinstimmung, geben wir den Index unseres aktuellen Slots zurück.
- Läuft die Schleife einmal komplett ohne dass eine Übereinstimmung gefunden wurde, geben wir eine -1 zurück.

Hinweis: Gute Praxis ist es, diese Methodenstruktur als Kommentar in den Quellcode zu integrieren.

Aufgabe: Setzen Sie nun den Quellcode in Java um.

Lösung:

```
1  public class IterativeSuche{
2     private int [] array;
3     public IterativeSuche(int [] array){
4        this.array = array;
5     }
6     public int suche( int gesucht) {
7        for (int i = 0; i < array.length; i++) {
8           if (array[i] == gesucht) {
9              return i;
10          }
11       }
12       return -1;
13    }
14 }
```

1.3 Binäre Suche

Binäre Suche ist ein geschickter Algorithmus, der fast immer schneller zum Ergebnis kommt. Für diesen Algorithmus gibt es jedoch eine wichtige Voraussetzung, damit dieser funktioniert: Das Array muss immer sortiert sein!

Algorithmus

- Das Array wird bei jedem Schritt in der Mitte geteilt und das Element in der Mitte mit dem gesuchten Element verglichen.
- Je nachdem ob das gesuchte Element kleiner oder größer ist, wird dieser Schritt beim linken oder beim rechten Teilarray wiederholt, bis das gesuchte Element gefunden wurde.
- Vorsicht: In Java wird abgerundet! Index 5/2 = 2

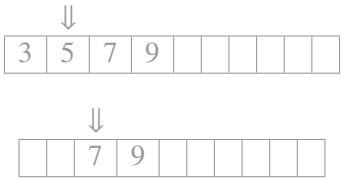
Handsimulation

Aufgabe: Gegeben sei das gleiche Array:

3	5	7	9	12	16	19	24	36	72

Finden Sie die 7 mittels binärer Suche.

Lösung:



Das Element 7 wurde am Index 2 gefunden.

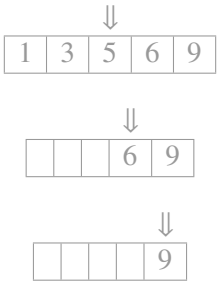
Anmerkungen

Aufgabe: Kann man die binäre Suche nach der 7 auf das folgende Array anwenden? Und wenn ja: Was ist das Ergebnis?

1	3	5	6	9

Lösung:

- Die binäre Suche kann angewendet werden.



- Kein weiteres Teilen möglich: Die Zahl ist nicht enthalten.

Aufgabe: Kann man die binäre Suche nach der 7 auf das folgende Array anwenden? Und wenn ja, was ist das Ergebnis?

5	1	9	3	6

Lösung:

- Die binäre Suche kann nicht auf unsortierte Arrays angewendet werden!

Algorithmusskizze

Dazu ist zu überlegen, was sinnvolle Rückgabewerte der Methode sind, was als Parameter übergeben werden soll, wie lange die Suche laufen soll, was überprüft wird und was zurückgegeben werden soll, wenn die Suche nicht erfolgreich war. Und zusätzlich welche lokalen Variablen notwendig werden, was getestet werden muss und wie die Update-Regeln aussehen.

Aufgabe: Geben Sie die nötigen Teile des Quellcodes an.

Lösung:

Methodenstruktur:

- Rückgabetyp: int (für den Index an dem wir das Element gefunden haben)
- Parameter: Wir müssen das gesuchte Element übergeben.
- Lokale Variablen: Wir brauchen Variablen, um das Ergebnis und den Index des ersten und letzten Elements zu speichern (Grenzen des Teilarrays). Die Ergebnis-Variable wird mit -1 initialisiert, sodass sie nicht extra gesetzt werden muss, falls das Element nicht enthalten ist.
- while-Schleife: Wir suchen solange bis ein Ergebnis gefunden wurde oder bis es keinen Teilarray mehr gibt, der durchsucht werden kann.
- Als erstes ermitteln wir das Element in der Mitte.
- Ist dieses Element kleiner oder größer, werden die Grenzen verändert, sodass wir im nächsten Durchlauf den jeweiligen Teilarray betrachten.
- Ist das Element in der Mitte gleich dem gesuchten Element, wird in unsere Ergebnisvariable der Index des Elements in der Mitte geschrieben.

2 Komplexität

2.1 Motivation

- Neben der Qualität der Ergebnisse ist auch die Zeit, die zum Berechnen dieser Ergebnisse benötigt wird, entscheidend.
- Die Aufwandsfunktion $T(n)$ beschreibt hierbei den zeitlichen Aufwand in Abhängigkeit vom Umfang der Eingangsdaten n.[1] Diese wird in dieser Veranstaltung mit der Anzahl von Befehlsaufrufen und Zuweisungen[2] angegeben. Die Effizienz bzw. Geschwindigkeit des Rechners geht in dieser Veranstaltung nur als skalarer Faktor ein.
- Es interessieren daher nicht der absolute Wert, sondern wie der Aufwand mit dem Umfang der Eingangsdaten n für große Datenmengen skaliert.

2.2 Komplexitätsklassen

- Die exakte mathematische Beschreibung des Zeitaufwands von Algorithmen ist i.A. schwierig.
- Man gibt sich mit verlässlichen Ober- und Unterschranken für das asymptotische Verhalten der Laufzeit zufrieden und kann so jedem Algorithmus eine Komplexitätsklasse zuweisen.
- Diese Zugehörigkeit zu einer Komplexitätsklasse drückt man mit Hilfe verschiedener Notationen aus. Die gebräuchlichste ist die *Groß-O-Notation*.

Definition: Es seien zwei Funktionen $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$. Die Komplexitätsklasse $O(f)$ (*Groß-O von* f) ist dann definiert durch:

$$g \in O(f) \qquad \exists k > 0, n_0 \in \mathbb{N} \quad \text{sodass} \quad k \cdot f(n) \geq g(n) \qquad \forall n \geq n_0$$

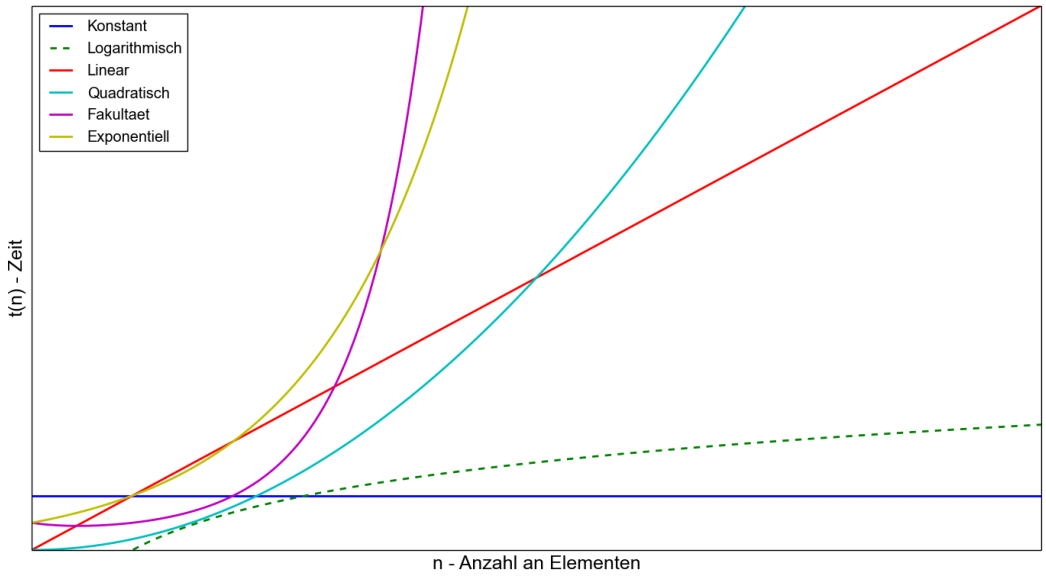
[1] Es kann auch die Speicherkomplexität betrachtet werden, sehen Sie dazu die Vorlesung „Suche & Sortieren“.

[2] Dies ist eine starke Vereinfachung, da verschiedene Befehlsaufrufe, Zuweisungen usw. (auch abhänging von der Hardware oder dem Betriebssystem) verschieden lang dauern. Man könnte genauer mit Zeiteinheiten für verschiedene Operationen rechnen.

Folgende Komplexitätsklassen sind für uns am wichtigsten:

Notation	**Bezeichnung**
$O(k)$	konstant, $k \in \mathbb{N}$
$O(log(n))$	logarithmisch
$O(n)$	linear
$O(n \cdot log(n))$	$n \cdot log(n)$
$O(n^2)$	quadratisch
$O(n^3)$	kubisch
$O(n^a)$	polynomiell
$O(2^n)$	exponentiell

Wobei gilt, dass $O(k) \subset O(log(n)) \subset O(n) \subset O(n \cdot log(n)) \ldots$.



Es sind Beispiele von Komplexitätsklassen dargestellt.[3] Betrachten Sie bspw. die braune (exponentiell) und die rote (linear) Funktion. Es gibt verschiedene Bereiche, in welchen sich ändern kann, welche Funktion größer ist. I.d.R. ist aber entscheidend, welchen Verlauf die Funktionen für große n also große Datenmengen annehmen.

2.3 Best-, Worst- und Average-Case und Abhängigkeit von der Programmsteuerung

- Für den **Best-Case** werden Fallentscheidungen stets zugunsten der Laufzeit gefällt, d.h. es wird immer zum „kürzesten“ (am wenigsten aufwendigsten) Weg im Programmcode verzweigt.
- Für den **Worst-Case** werden Fallentscheidungen stets zulasten der Laufzeit gefällt, d.h. es wird immer zum „längsten“ (aufwendigsten) Weg im Programmcode verzweigt.
- Für den **Average-Case** wird anhand von sinnvollen (fallabhängigen) Kriterien entschieden, wie oft Verzweigungsbedingungen im Mittel erfüllt sein werden. Dies muss bei jeder Betrachtung explizit angegeben werden, da der Average-Case nicht eindeutig ist.

[3] Der Schnittpunkt mit der y-Achse (Konstante addiert) und der Anstieg (Konstante multipliziert) können variieren. Beispiel: $2n^2 + 1$ und n^2 und $n^2 - 30$ gehören alle zu $O(n^2)$.

2.4 Bestimmung der Anzahl der Zuweisungen bei einfachen Implementierungen

Aufgabe: Welche Funktionalität hat folgende Methode?

```
1  int whatami( int[] array, int a ) {
2      int i = 0, result = 0;
3      while( i < array.length ) {
4          if( array[i] == a ) {
5              result ++;
6              array[i] = 0;
7          }
8          i++;
9      }
10     return result;
11 }
```

Lösung:

Sie gibt zurück, wie oft das Element a im Array enthalten ist und ersetzt jedes Vorkommen von a im Array mit 0.

Aufgabe: In Abhängigkeit der Problemgröße n (hier Arraylänge) soll nun die Anzahl der Zuweisungen $T(n)$ nach Aufruf der Methode im Worst-Case (Best-Case, Average-Case) ermittelt werden.
Vorgehen:

- im Worst-Case annehmen, dass stets zum „längsten“ (heißt aufwendigsten) Weg im Programmcode verzweigt wird (hier: if-Bedingung immer erfüllt)
- alle Zuweisungen außerhalb von Schleifen zählen
- Zuweisungen innerhalb von Schleifen zählen und mit der Anzahl der Schleifendurchläufe multiplizieren (evtl. mehrmals bei verschachtelten Schleifen!)
- insbesondere aufpassen, keine „versteckten“ Zuweisungen zu übersehen, wie z.B. Inkrementoperator (result++) und Schleifencounter

Lösung:

1. außerhalb der Schleife in Zeile 2: 2 Zuweisungen
2. Zeile 5: 1 Zuweisung
3. Zeile 6: 1 Zuweisung
4. Zeile 8: 1 Zuweisung
5. (2),(3),(4) liegen innerhalb der Schleife, die n mal durchlaufen wird, also $3 \cdot n$ Zuweisungen in der Schleife
6. insgesamt sind es also $3 \cdot n + 2$ Zuweisungen nach Aufruf der Methode
7. man wird später sehen, dass Summanden, die den Faktor n nicht enthalten, vernachlässigt werden können, also sind es $\approx 3 \cdot n$ Zuweisungen für große n

Für die anderen Fälle ergeben sich:

- Best-Case: Nur die Zuweisung in Zeile 8 (i++) wird erreicht plus die 2 Zuweisungen in Zeile 2. Somit $n + 2$.
- Average-Case: Analog zu Worst-Case, aber nur 50% der Verzweigungen berücksichtigen ($\frac{2n}{2} + n + 2$)
  - Der Average-Case ist von der Anwendung des Algorithmus’ abhängig. Er wird im Rahmen dieser Lehrveranstaltung nur grob geschätzt.

Wenn man nicht sofort alles erkennt, kann man sich die Methode auch mit for-Schleife umschreiben, sodass die Zuweisungen und die Anzahl der Schleifendurchläufe noch offensichtlicher werden:

```java
int whatami( int[] array, int a ) {
      int result = 0;
      for( int i = 0; i < array.length; i++ ) {
         if( array[i] == a ) {
            result = result + 1;
            array[i] = 0;
         }
      }
      return result;
   }
```

2.5 Einordnung und Bestimmung der Komplexitätsklasse

- Die resultierende Funktion $T(n)$ wird zu einer der gegebenen Komplexitätsklassen zugeordnet
- Dazu wird die höchstwertigste Potenz in $T(n)$ in Betracht gezogen

Aufgabe: Welcher Klasse gehört $T(n) = n^3 + 2999 \cdot n^2 \cdot \log(n) - 2$ an?

Lösung:
$T(n) \in O(n^3)$

3 Komplexitätsbetrachtung von iterativer und binärer Suche

Um die beiden Suchverfahren miteinander zu vergleichen, wollen wir den Aufwand (Komplexität) $T(n)$ der beiden Methoden ermitteln.

Was soll überhaupt beim Aufwand (der Komplexität) ermittelt werden?
Es wird ermittelt wie viele Schritte man in Abhängigkeit von der Anzahl der Elemente braucht, um das gesuchte Element zu finden.

- Generell hängt beim Suchen die Anzahl der Schritte davon ab, wo sich das Element im Array befindet.
- Deshalb gilt es zu unterscheiden zwischen Best-Case, Worst-Case und Average-Case (in diesem Fall als Durchschnitt von Best und Worst-Case, da selbe Wahrscheinlichkeit).
- Der Aufwand wird in Abhängigkeit von der Anzahl n der zu verarbeitenden Daten (hier: Anzahl der Elemente des Arrays) angegeben.

Aufgabe: Geben Sie den Best-Case und den Worst-Case für die iterative und binäre Suche an.

Lösung:

- Iterative Suche:
  - Best-Case: 1 (gesuchtes Element ist das erste Element)
  - Worst-Case: n (letztes Element: bei z.B. 16 Elementen 16 Durchläufe)
- Binäre Suche:
  - Best-Case: 1 (gesuchtes Element ist das mittlere (erstes getestete) Element)
  - Worst-Case: $log_2(n)$ (da wir das Array immer teilen, gehen wir 2er-logarithmisch vor, Element im letzten Durchlauf)

In beiden Fällen lässt sich auch der Average-Case aus dem Durchschnitt von Best- und Worst-Case bestimmen (siehe dazu auch die Vorlesung „Suche & Sortieren“):

- Iterative Suche:
  - Average: $\frac{n}{2}$
- Binäre Suche:
  - Average: $log_2(n+1) - 1$

Wir sehen, dass im betrachteten Average-Case die binäre Suche schneller ist (n zu $log_2(n)$). Außer für Spezialfälle (z.B. Array ist unsortiert oder sehr kleine Datenmenge), würde es sich immer lohnen, die binäre Suche zu benutzen.

4 Potenzberechnung

Das Potenzieren einer Basis a mit einem Exponenten n (also a^n) kann algorithmisch nicht nur durch n-faches Aufmultiplizeren von a (wie in Hausaufgabe 2 Teilaufgabe 3 vorgeschlagen) gelöst werden.

Aufgabe: Schreiben Sie einen Algorithmus, der eine positive Zahl a mit dem natürlichen Exponenten n potenziert. Der Algorithmus soll effizienter als $O(n)$ sein.

```
1  /*
2   * Klasse, die eine statische Methode zur Potenzberechnung bereitstellt.
3   */
4  class Potenz {
5
6     /* Berechnet die Potenz a^n rekursiv mit logarithmischer Komplexitaet. */
7     public static double pow(double a, int n) {
8
9
10
11
12    }
13
14 }
```

Lösung:

```
1  /*
2   * Klasse, die eine statische Methode zur Potenzberechnung bereitstellt.
3   */
4  class Potenz {
5
6     /* Berechnet die Potenz a^n rekursiv mit logarithmischer Komplexitaet. */
7     public static double pow(double a, int n) {
8        if (n <= 0) return 1.0;
9        double b = pow(a, n/2);
10       if (n % 2 == 0) return b*b;
11       else    return b*b*a;
12    }
13
14 }
```