Referenztypen – Teil 3

Inhalt:

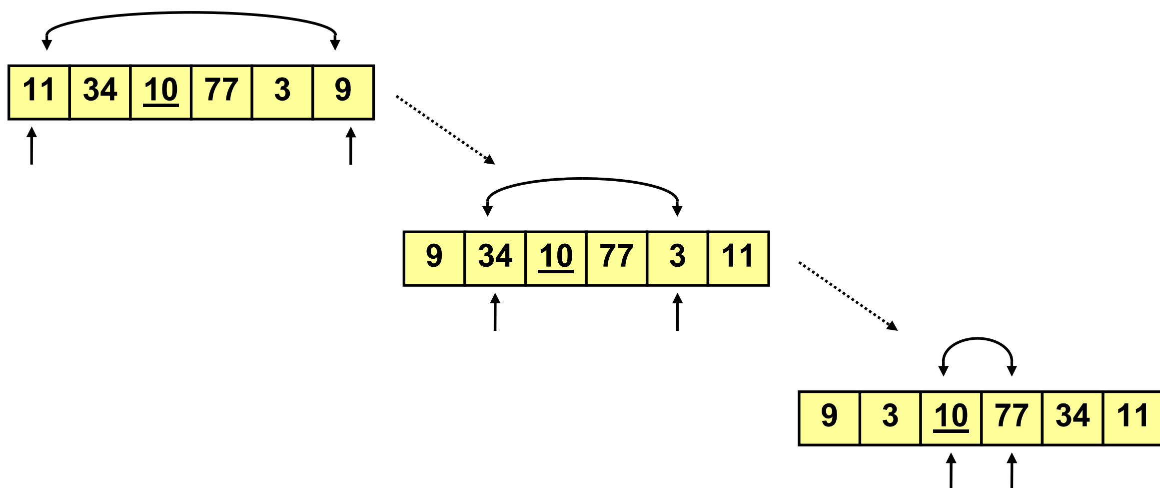
8. Beispiel: Quicksort
9. Komplexität
  9.1 Vergleich der Sortierverfahren
  9.2 Komplexität

Bezug:
G. Krüger, GotoJava 2 – HTML-Version:
Kapitel 4.2, 4.4, 4.5
Kapitel 11.3

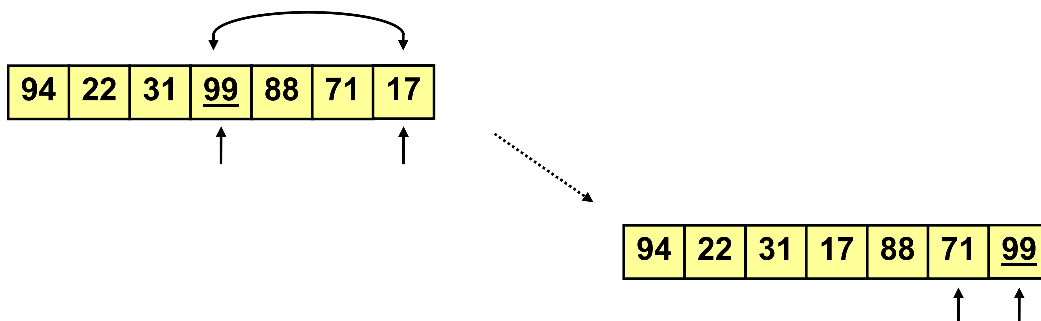
8. Beispiel: Quicksort[1]

Als Beispiel für ein effizientes Sortierverfahren wollen wir die folgende Idee verfolgen: Statt jeweils benachbarte Feldelemente zu vertauschen, sollen die Vertauschungen über größere Entfernungen so erfolgen, dass immer ein „großes" und ein „kleines" Element ihre Plätze wechseln. Benutzt man zum Vergleich ein „mittleres" Element, dann entsteht auf diese Weise ein Feld aus zwei meist unterschiedlich großen Teilen, in dem links die „kleinen", rechts die „großen" Elemente stehen. Die Größe der Teile hängt von der relativen Größe des „mittleren" Elements zu den anderen Feldelementen ab.

Als „mittleres" Element wählen wir das Element etwa in der Mitte des Feldes, das im folgenden Bild unterstrichen auftritt. Daraufhin durchsuchen wir das Feld von „links", bis wir ein größeres, dann von „rechts", bis wir ein kleineres Element als das „mittlere" gefunden haben. Diese werden vertauscht. Danach wird weiter gesucht und vertauscht. Die momentanen Plätze, bis zu denen das Feld von links bzw. rechts schon durchsucht wurde, werden durch zwei „Zeiger" gekennzeichnet, die wir als *links* und *rechts* bezeichnen und die im folgenden Bild als kleine Pfeile erscheinen. Die Doppelpfeile verbinden die zu vertauschenden Elemente. Der Prozess bricht ab, wenn sich die Zeiger treffen (nicht, wenn das mittlere Element überschritten wird).



Das Verfahren funktioniert auch, wenn als mittleres Element zufällig der kleinste oder der größte Wert des Feldes gewählt wird.

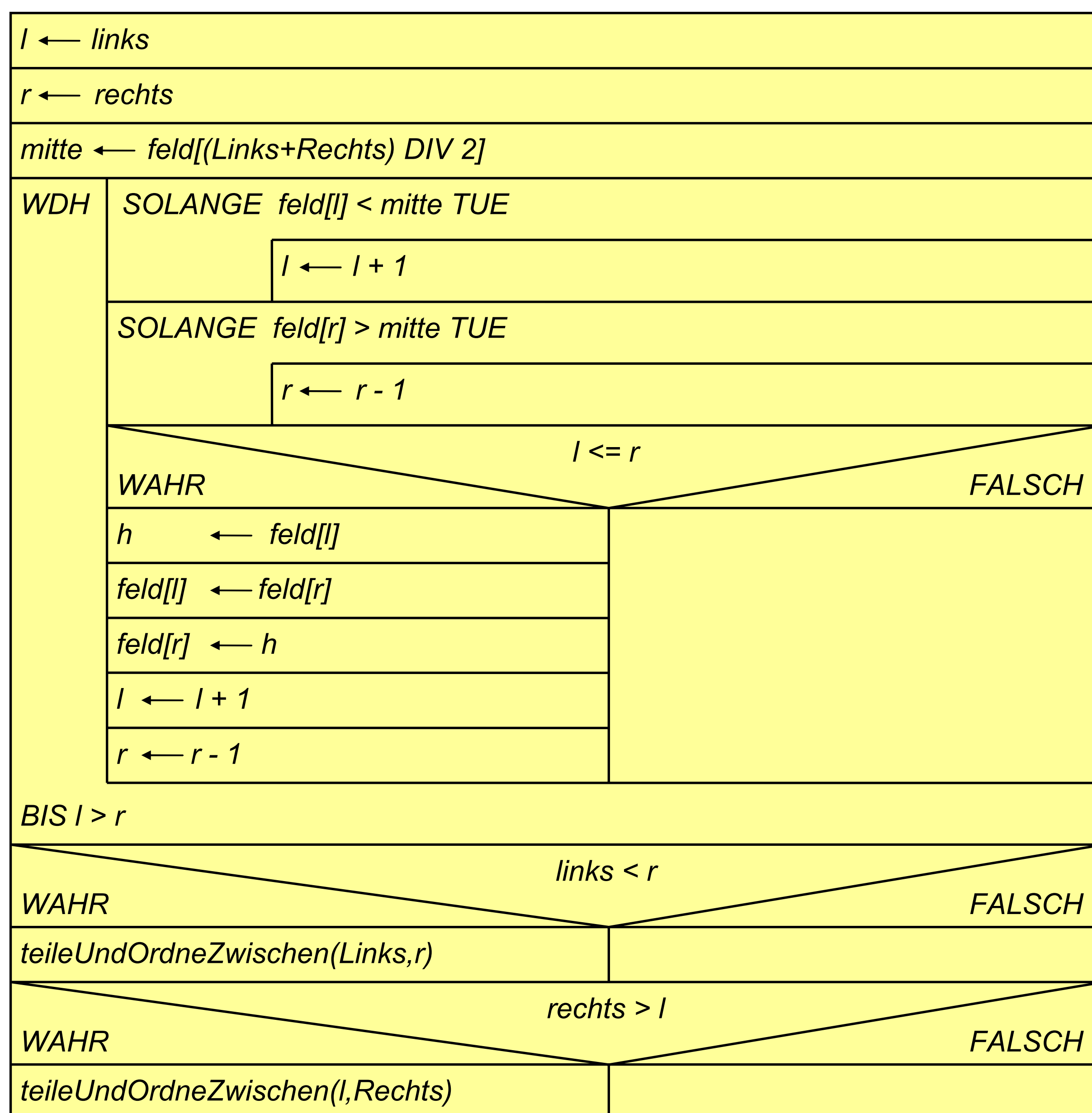


[1] Text nach E. Modrow, Dateien – Datenbanken - Datenschutz, Dümmler 1996

Natürlich ist das Feld nach einem solchen Durchgang noch nicht vollständig sortiert, denn bisher stehen nur links die Elemente kleiner als das „mittlere“, rechts die größeren. Das Feld kann weiterbearbeitet werden, indem die beiden Teile des Feldes wiederum nach dem gleichen Verfahren geteilt und geordnet werden. Als Grenzen für diesen Prozess gelten dann entweder der linke Rand und der Treffpunkt der beiden Zeiger bzw. dieser Treffpunkt und der rechte Rand. Das Sortierverfahren ist also rekursiv zu formulieren.

Im Struktogramm lautet das Verfahren bisher *wie* folgt, wenn die Grenzen, zwischen denen geordnet werden soll, als Parameter *links* und rechts übergeben werden:

`teileUndOrdneZwischen(int links, int rechts):`



Das Quicksort-Verfahren wird gestartet, indem die *TeileUndOrdneZwischen*-Prozedur mit den Grenzen des zu sortierenden Feldes aufgerufen wird:

`Quicksort:` *TeileUndOrdneZwischen(1,laenge)*

Die Umsetzung in Java kann direkt erfolgen: Wir wollen zwei Zahlenfelder vereinbaren, von denen das eine unsortierte Zufallszahlen aufnimmt und das andere die sortierten Zahlen.

```
int[] alteZahlen = new int[20], sortierteZahlen;
```

Zur Darstellung der Felder wandeln wir deren Inhalte bei Bedarf in Zeichenketten um:

```
private String toString(int[] f)
{
      String h = "[";
      for(int i=0;i<f.length-1;i++) h=h+f[i]+",";
      h=h+f[f.length-1]+"]";
      return h;
}
```

Jetzt können wir das erste Feld mit Zufallszahlen füllen und in einer Label-Komponente namens *labelAZ* anzeigen.

```
private void neueZahlen()
{
      for(int i=0;i<alteZahlen.length;i++)
            alteZahlen[i] = (int)Math.round(Math.random()*1000);
      labelAZ.setText(toString(alteZahlen));
}
```

Nach diesen Vorbereitungen können wir den Quicksort-Algorithmus implementieren …

```
private void teileUndOrdneZwischen(int links, int rechts)
{
      int l = links, r = rechts;
      int mitte = sortierteZahlen[Math.round((links+rechts)/2)];
      do
      {
            while(sortierteZahlen[l] < mitte) l++;
            while(sortierteZahlen[r] > mitte) r--;
            if(l<=r)
            {
                  int h = sortierteZahlen[l];
                  sortierteZahlen[l] = sortierteZahlen[r];
                  sortierteZahlen[r] = h;
                  l++;
                  r--;
            }
      }
            while(l<=r);
      if(links<r) teileUndOrdneZwischen(links,r);
      if(rechts>l) teileUndOrdneZwischen(l,rechts);
}
```

und bei Bedarf einsetzen:

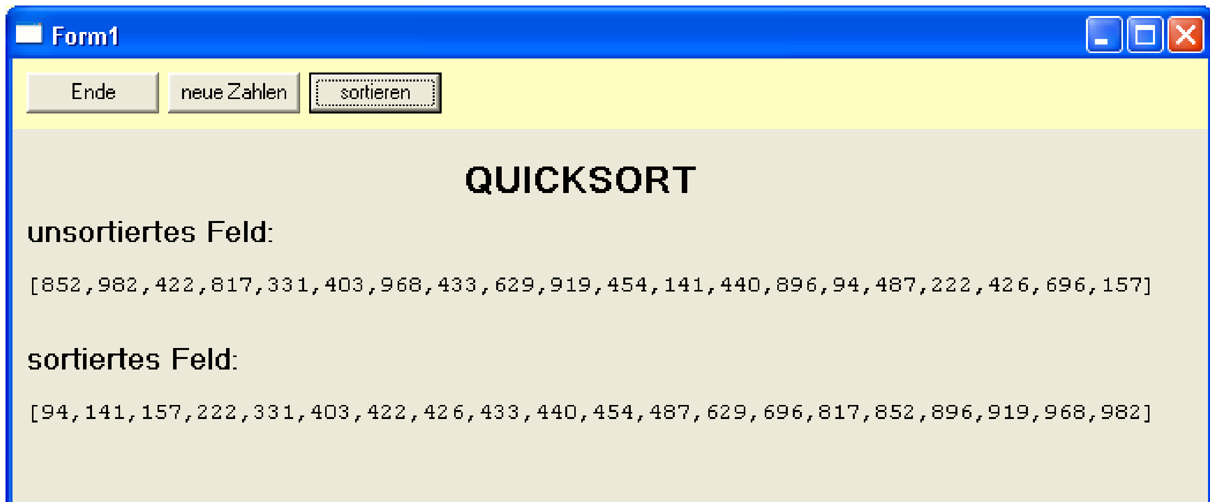
```
private void quicksort()
{
      teileUndOrdneZwischen(0,sortierteZahlen.length-1);
}
```

Natürlich müssen wir vorher das zweite Zahlenfeld initialisieren.

```
private void sortieren_click(Object source, Event e)
{
      sortierteZahlen = alteZahlen;
      quicksort();

}
```

Das Ergebnis innerhalb einer schnell zusammengeklickten Oberfläche sieht dann z. B. so aus:



Aufgaben:

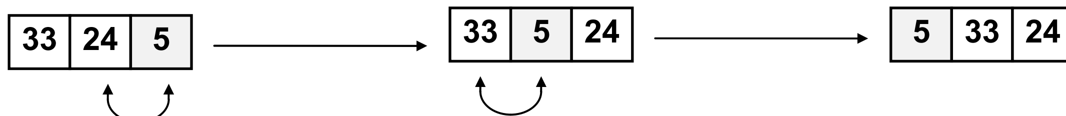
1. Wenden Sie das Quicksort-Verfahren auf Feldern mit Zeichenketten an.

2. a: Vereinbaren Sie eine Klasse *personaldaten*, die unterschiedliche Daten z. B. von Arbeitnehmern enthält.

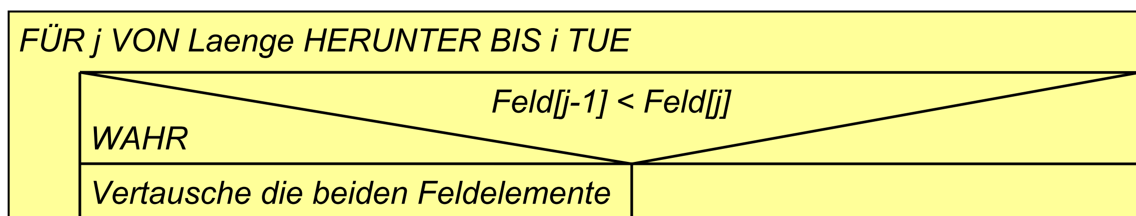
   b: Leiten Sie aus den Daten ein Schlüsselfeld (*key*) ab, das für unterschiedliche Datensätze jeweils eindeutige Schlüsselnummern ergibt.

   c: Ordnen Sie Felder mit Personaldaten mit Quicksort unter Verwendung der *Keys*.

3. Das wohl einfachste Verfahren zum Sortieren eines Feldes ist, jeweils zwei benachbarte Elemente zu vergleichen, z.B. nach der Größe, und bei Bedarf zu vertauschen. Wiederholt man diesen Vorgang mit den nächsten Elementen, dann wandert das kleinste (oder größte) Element wie eine Art Luftblase im Wasser zur entsprechenden Feldgrenze. Nachdem das Feld einmal durchlaufen wurde, befindet sich z.B. das kleinste Element am „linken" Rand. Wir beginnen im folgenden Beispiel bei einem sehr kleinen Feld am rechten Rand. Die Doppelpfeile bezeichnen die Elemente, die jeweils verglichen werden. Das Feld wird anschließend wieder im Zustand nach dem Vergleich und eventueller Vertauschung dargestellt.



Der Prozess lässt sich beschreiben als **`KleinstesElementNachLinks`:**



Zum vollständigen Sortieren muss wiederholt das kleinste Element der Restmenge nach links gebracht werden; allerdings kann das vorher erste Element dabei unberücksichtigt bleiben, da es sich schon an der richtigen Stelle befindet. Die linke Sortiergrenze i rückt deshalb nach jedem Sortiervorgang um eine Stelle nach rechts.

Bubblesort:

FÜR i VON 2 BIS Laenge TUE
KleinstesElementNachLinks

Realisieren Sie das Verfahren.

4. Realisieren Sie die folgenden Sortierverfahren für Felder mit n Elementen, indem Sie zuerst entsprechend verfeinerte Struktogramme entwickeln, die danach in Programme umgesetzt werden. Schätzen Sie den Zeitbedarf der Verfahren ab.

a. Sortieren durch direktes Einfügen:

FÜR i VON 2 BIS n TUE
Sortiere das i-te Element am richtigen Platz unter den ersten (i-1) Elementen ein

b. Sortieren durch direktes Auswählen:

FÜR i VON 1 BIS (n-1) TUE
Bestimme den Platz des kleinsten Elements zwischen i und n
Vertausche dieses Element mit dem i-ten.

c: Shakersort ist eine Abwandlung von Bubblesort, in der die Laufrichtung der “Blasen“ von Durchgang zu Durchgang gewechselt wird.

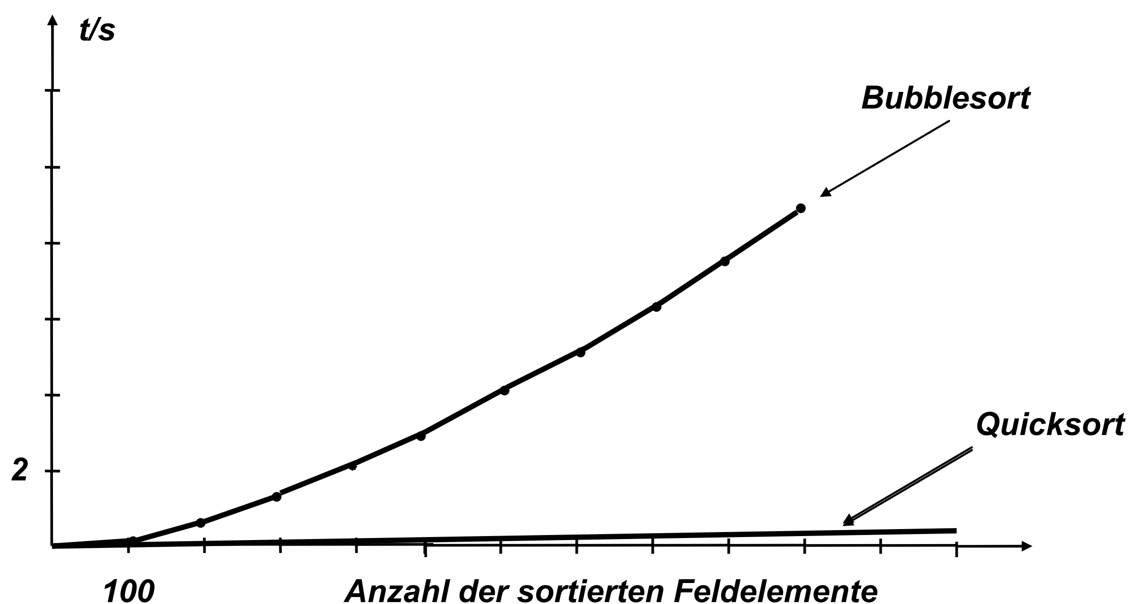
Links ← 2		
Rechts ← n		
WDH	*„Bubblesort von rechts“ zwischen Rechts und Links*	
	Es wurde getauscht	
	WAHR	*FALSCH*
	Links ← Platz rechts vom kleinsten eben vertauschten Element	*Links ← Links + 1*
	„Bubblesort von links“ zwischen Rechts und Links	
	Es wurde getauscht	
	WAHR	*FALSCH*
	Rechts ← Platz links vom größten eben vertauschten Element	*Rechts ← Rechts - 1*
BIS Links > Rechts		

9. Komplexität[2]

9.1 Vergleich der Sortierverfahren

Eigentlich ist es überraschend, dass in der Literatur außerordentlich viele unterschiedliche Sortierverfahren zu finden sind, denn im Laufe der Jahre sollte sich doch das beste Verfahren durchgesetzt haben. Wenn dem nicht so ist, dann gibt es augenscheinlich kein „bestes" Verfahren für alle möglichen Fälle. Die Effizienz der Verfahren hängt außerordentlich stark von der Art der zu sortierenden Daten ab. Sind etwa wenige neue Daten in einen schon sortierten Bestand einzuordnen, dann kann es günstig sein, diese einfach im Bubblesort-Verfahren von hinten an ihren richtigen Platz laufen zu lassen. Liegen große Mengen zufällig angeordneter Daten vor, dann ist das Quicksort-Verfahren sicherlich geeigneter. Zusätzlich müssen auch z. B. der Platzbedarf der Daten im Rechner (wenn etwa die Daten in ein zweites, gleich großes Feld eingefügt werden sollen), der Zeitbedarf für die verschiedenen Operationen (beim Vertauschen etwa) u. Ä. berücksichtigt werden. Ohne genaue Kenntnisse über spezielle Eigenschaften der zu sortierenden Daten lässt sich das geeignetste Verfahren also nicht angeben. Wir gehen deshalb im Folgenden von einer zufällig verteilten Datenmenge aus.

Wir wollen mit einem experimentellen Vergleich der Verfahren beginnen. Dazu messen wir einfach die Zeit, die die unterschiedlichen Methoden zur Sortierung derselben Felder aus Zufallszahlen benötigen. Ist diese Zeit sehr kurz, werden mehrere Sortierläufe nacheinander durchgeführt und der Mittelwert berechnet. Von dieser Zeit muss jeweils die Zeit abgezogen werden, die das Rahmenprogramm, das z.B. die Felder aus Zufallszahlen bereitstellt, selbst benötigt. Auch diese Zeit wird zuerst experimentell ermittelt. Zur Zeitmessung kann die eingebaute Uhr im Rechner benutzt werden. Fehlt diese, dann leistet eine Stoppuhr bei entsprechend vielen Durchgängen und Mittelwertbildung gute Dienste. Das Ende eines Sortiervorgangs sollte dann durch einen Ton angezeigt werden, da man auf diesen schneller reagiert als auf ein optisches Signal. Die Messwerte für die folgende Grafik wurden mit der Hand gestoppt.



Man sieht, dass Bubblesort für das Sortieren größerer, ungeordneter Datenmengen völlig ungeeignet ist (für das Sortieren sehr kleiner Datenmengen benötigt man keinen Computer), und

[2] Text nach E. Modrow, Dateien – Datenbanken - Datenschutz, Dümmler 1996

dass der Zeitbedarf etwa quadratisch mit der Anzahl der zu sortierenden Daten wächst. Der Zeitbedarf von Quicksort scheint im betrachteten Bereich bei dem verwendeten Maßstab eher linear von der Zahl der Datensätze abzuhängen. Versuchen wir also den genauen Zusammenhang herauszufinden.

Bubblesort:
Wir betrachten den ungünstigsten Fall, wenn bei jedem Vergleich auch ein Platztausch der betrachteten Elemente erforderlich wird. Dann sind alle Operationen gleichwertig. Hat unser Feld n Elemente, dann wird das letzte Element mit $(n-1)$ Elementen verglichen, bevor es seinen endgültigen Platz erreicht hat. Das nächste Element wird nur noch mit $(n-2)$ Elementen verglichen, das nächste mit $(n-3)$ usw., bis beim letzten nur noch ein einziger Vergleich benötigt wird. Insgesamt sind zum Sortieren

$$(n-1)+(n-2)+(n-3)+....+2+1=\frac{1}{2}\cdot n\cdot(n-1)$$

Vergleiche nötig. Da alle unsere Vergleiche gleichwertig sind, ergibt sich damit auch eine quadratische Abhängigkeit der Sortierzeit von der Zahl der zu sortierenden Elemente. Doch auch im günstigsten Fall eines schon sortierten Feldes, wenn keine Vertauschungen notwendig werden, müssen alle Vergleichsoperationen vollständig durchgeführt werden, so dass die Zeitabhängigkeit quadratisch bleibt, allerdings mit einer anderen Konstanten.

Quicksort:
Nach der Auswahl eines „mittleren“ Elements müssen alle anderen Elemente mit diesem verglichen werden: das erfordert $(n-1)$ Vergleiche. Die Zahl der Vertauschungen ist erheblich geringer. Falls das „mittlere“ Element wirklich den mittleren Wert der zu sortierenden Werte darstellt, werden im ungünstigsten Fall alle Elemente links von ihm mit den rechten Elementen vertauscht: das ergibt maximal $(n/2)$ Vertauschungen. Beide Ergebnisse ergeben aber eine lineare Abhängigkeit von n. Damit ist das Feld allerdings noch nicht sortiert, denn der Vorgang muss jetzt für die Teilstücke wiederholt werden. Wird das Feld bei jeder Zerlegung halbiert, dann ergibt sich die Anzahl der Zerlegungen aus $ld(n)$, dem Zweierlogarithmus von n, weil umgekehrt das Feld wieder aus 2 Verdoppelungen der entstandenen Teilstücke entsteht und der Zweierlogarithmus die Umkehrfunktion zur Zweierpotenz ist. Da die Teilstücke nun eine kleinere Länge als n haben, ist das Produkt

$$n\cdot ld(n)$$

von $ld(n)$ Durchgängen mit je etwa n Vergleichen eine obere Abschätzung des Aufwands zum Sortieren des Feldes für diesen Fall. Im ungünstigsten Fall wird das Feld allerdings nicht halbiert, sondern in ein einzelnes Element und den Rest aufgeteilt. Insgesamt ergeben sich dann also etwa n Zerlegungen und $(n*n)$ Vergleiche. Für ein Feld aus Zufallszahlen ist dieser Fall allerdings extrem unwahrscheinlich, so dass der erste Ansatz eine realistische Abschätzung liefert. Will man sicher gehen, dann kann das „mittlere“ Element nicht als zufällig in der Mitte stehendes Element, sondern als Ergebnis eines besonderen Prozesses ermittelt werden.

Aufgaben

1. Zur Ermittlung des Zeitbedarfs muss die Zeit einer Vergleichsoperation von der eines Austausches von Elementen unterschieden werden. Ermitteln Sie diese Zeiten für unterschiedliche Datensätze und schätzen Sie entsprechend die Rechenzeiten für unterschiedliche Sortierverfahren ab.

2. Ermitteln Sie die Abhängigkeit des Zeitbedarfs von Sortierverfahren von der Anzahl der zu sortierenden Elemente experimentell und theoretisch
   a. für Felder aus Zufallszahlen.
   b. für den ungünstigsten Fall, etwa in der „falschen“ Reihenfolge vorsortierte Felder.
   c. für den günstigsten Fall, etwa vorsortierte Felder.

9.2 Komplexität

Um unabhängig vom benutzen Rechner und anderen technischen Details die Komplexität von Algorithmen zu beurteilen, versuchen wir Aussagen über die *relative* Zunahme vom Ressourcenbedarf zu machen, wenn die Zahl der bearbeiteten Elemente zunimmt. Wir versuchen also den Bedarf am Zeit und Raum (Speicher) des Verfahrens zu bestimmen. Entsprechend spricht man von **Zeitkomplexität** bzw. **Raumkomplexität**. Wir erläutern die Methode am Beispiel:

Gegeben seinen drei Methoden *m1()*, *m2()* und *m3()*, die jeweils unterschiedlich viel Rechenzeit beanspruchen. Diese Methoden werden innerhalb einer vierten aufgerufen, und zwar mehrfach:

```
public void test(int n)
{
   m1();m1();m1();
   for(int i=1;i<=n;i++)
   {
      m2();
      for(int j=1;j<=n;j++)
      {
         m3();
         m3();
      }
   }
}
```

Untersuchen wir nun mal die Zahl der Ausführungen der einzelnen Methoden. Bei jedem Aufruf von *test()* wird

- $3 \cdot$ *m1()*
- $n \cdot$ *m2()*
- $2 \cdot n \cdot n \cdot$ *m3()*

ausgeführt. Messen wir die Zeit in *tics* und weisen den drei Methoden unterschiedliche Ausführungszeiten zu, z. B.

- *m1()* dauert 100 tics
- *m2()* dauert 10 tics
- *m3()* dauert 1 tic

dann erfordert die Ausführung von *test()* $(3 \cdot 100 + n \cdot 10 + 2n^2 \cdot 1)$ tics.
Die Abhängigkeit dieser Zeit von n kann man leicht nachrechnen:

n	Zeit in tics
1	312
10	600
100	21300
1.000	2010300
10.000	200100300
100.000	20001000300
1.000.000	2000010000300
10.000.000	200000100000300

Man sieht, dass für größere Werte von n (und nur die sind interessant) der Zeitbedarf praktisch ausschließlich von den Methodenaufrufen bestimmt wird, die mit der höchsten Potenz von n aufgerufen werden. Man sagt, dass die Zeitkomplexität der Methode *test()* „*von der Größenordnung n^2*“ sei und schreibt das als ***O(n^2)***. In unserem Fall bedeutet das, dass bei Verdoppelung von n der Zeitbedarf auf das Vierfache wächst: $(2n)^2/n^2=4$.

Allgemein spricht man von

- *konstanter* Komplexität bei O(1),
- *linearer* Komplexität bei O(n),
- *quadratischer* Komplexität bei $O(n^2)$,
- *logarithmischer* Komplexität bei O(log n) - oft auch bei O(n·log n),
- polynomialer Komplexität, wenn die Komplexität besser ist als $O(n^k)$
- und exponetieller Komplexität, wenn sie noch miserabler ist.

Allgemein gilt für polynomiale Komplexitäten: $\lim_{n\to\infty}\frac{f(n)}{n^k} = konst.$

Für unsere *test*-Methode z. B. wg. $O(n^2)$:

$$\lim_{n\to\infty}\frac{f(n)}{n^2} = \lim_{n\to\infty}\frac{2\cdot n^2+10\cdot n+300}{n^2} = \lim_{n\to\infty}\left(\frac{2\cdot n^2}{n^2}+\frac{10\cdot n}{n^2}+\frac{300}{n^2}\right) = 2$$

Als Beispiel wollen wir Fakuläten berechnen, zuerst iterativ:

```
public int nfak(int n)
{
  int h=1;
  for(int i=2;i<=n;i++) h = h*i;
  return h;
}
```

Die Zeitkomplexität des Verfahrens ist offensichtlich O(n), der Speicherbedarf beträgt konstant drei Integerplätze.

Jetzt formulieren wir das Verfahren rekursiv:

```
public int nfak(int n)
{
  if(n==0) return 1;
  else return n*nfak(n-1);
}
```

Wieder ist die Zeitkomplexität O(n). Dazu kommt aber eine *Speicherkomplexität*, die ebenfalls O(n) ist, weil die für die Abarbeitung der Rekursion notwendigen Daten auf dem Stack abgelegt werden müssen.

Aufgaben

1. Gegeben sei ein Feld mit gemischt positiven bzw. negativen Zahlen. Gesucht ist die ***maximale Teilsumme*** dieser Zahlen.

   a: Schreiben Sie ein entsprechendes Java-Programm.

   b: Ermitteln Sie die Zeitkomplexität ihrer Lösung.

   c: Verifizieren Sie Ihr theoretisches Ergebnis durch Messungen.

   d: Vergleichen Sie die Komplexität Ihrer Lösung mit der bekannter anderer Verfahren, die Sie zahlreich im Internet bzw. in der Literatur finden.

2. Gegeben sei eine längere Zeichenkette *text* und eine kürzere *muster*. Gesucht ist der Index des ersten Zeichens der längeren Kette, ab der *muster* in ihr enthalten ist. (Bei Misserfolg wird 0 von dieser ***Mustererkennung*** zurückgeliefert.)

   a: Schreiben Sie ein entsprechendes Java-Programm.

   b: Ermitteln Sie die Zeitkomplexität ihrer Lösung.

   c: Vergleichen Sie die Komplexität Ihrer Lösung mit der bekannter anderer Verfahren, die Sie zahlreich im Internet bzw. in der Literatur finden.

3. Gegeben sei ein sortierter Binärbaum, der n Zufallszahlen enthält. Schreiben Sie eine Methode suchen, die feststellt, ob sich eine Zahl im Baum befindet – oder nicht.

   a: Schreiben Sie ein entsprechendes Java-Programm.

   b: Ermitteln Sie die Zeitkomplexität ihrer Lösung.

4. Verfahren Sie entsprechend mit den Sortierverfahren aus Abschnitt 8.