Teil II: Unterrichtssequenz

Inhalt

1 Einleitung und Begriffe

Lernziel
Nach der Bearbeitung dieses Kapitels
- können Sie die zentralen Begriffe des Themenfeldes Sortierung erklären.

Zu Beginn eine kleine Geschichte:

Der Mistkäfer Willi möchte Ordnung in seine Mistkugelsammlung bringen. Am liebsten hätte er sie schön der Grösse nach sortiert, die kleinste Kugel links, die grösste rechts. Leider hat Willi keine Ahnung, wie er das anstellen könnte. Er setzt sich auf eine der Mistkugeln und überlegt, aber es fällt ihm keine Lösung ein. Da wählt er in seinem Frust zufällig zwei Mistkugeln aus, bei denen die Reihenfolge nicht stimmt, und vertauscht die beiden. Willi ist natürlich klar, dass er damit seine Kugelreihe noch lange nicht sortiert hat. In seiner wachsenden Verzweiflung wählt er ein weiteres Kugelpaar, das nicht richtig geordnet ist, und vertauscht auch dieses. Diesen Vorgang wiederholt er nun laufend. Nach vielen solchen Vertauschungen stutzt Willi plötzlich: Er findet kein einziges "falsches" Paar mehr! Enttäuscht wendet er sich ab und versinkt wieder ins Grübeln. Nach einer Weile schaut er auf und betrachtet die Reihe seiner Mistkugeln...

(Fortsetzung folgt)

1.1 Wozu sortieren?

Wir sortieren Farbstifte, Rechnungen, Adressen, Kleider, Bankbelege, Personen und vieles mehr. Wozu eigentlich? Die Antwort ist klar: Sortieren erleichtert das Wiederfinden. Stellen wir uns vor, im Telefonbuch wären die Einträge in zufälliger Reihenfolge abgedruckt. Das Telefonbuch wäre nahezu nutzlos.

Der Nutzen der Sortierung ist also wohl unbestritten. Für Informatiker interessant ist die Tatsache, dass es sehr viele verschiedene Sortierverfahren gibt. Diese sind keineswegs gleichwertig, sondern weisen gewaltige Unterschiede bezüglich ihrer Effizienz auf.

Wir werden in diesem Kapitel drei Sortierverfahren kennen lernen und implementieren. Ausserdem werden wir ein Mass für die Effizienz dieser Verfahren einführen und so in der Lage sein, sie miteinander zu vergleichen.

Aufgabe (1.1)
Beschreiben Sie den Unterschied zwischen Inhaltsverzeichnis und Index eines Sachbuches bezüglich der Reihenfolge der Einträge.

1.2 Sortierkriterium und Vergleichsfunktion

Je nach Anwendungsgebiet sind verschiedene Sortierkriterien denkbar: Rechnungen sortieren wir nach Datum oder nach Absender, Bankbelege nach Datum oder nach Bank. Eine Gruppe von Personen kann man z.B. nach Grösse, Gewicht oder nach Alter sortieren. Farbstifte können wir z.B. nach Helligkeit sortieren, nach Rotanteil oder gemäss der Reihenfolge, die auf der Farbstiftschachtel aufgedruckt ist.

Es gibt auch kombinierte Sortierkriterien. Nehmen wir zum Beispiel ein Telefonbuch. Die Einträge sind in erster Linie nach der Ortschaft sortiert, innerhalb einer Ortschaft nach Name sortiert, bei gleichen Namen nach Vorname.

Die Wahl des Kriteriums ist entscheidend für die Nützlichkeit der Sortierung. Oder wer könnte wohl etwas anfangen mit einem Telefonbuch, das nach Telefonnummern sortiert ist?

Um eine Liste nach einem bestimmten Kriterium sortieren zu können, müssen je zwei Elemente gemäss diesem Kriterium mit einander verglichen werden können. Von zwei beliebigen Elementen muss entscheidbar sein, welches das grössere bzw. das kleinere ist. Natürlich gibt es auch den Fall der Gleichheit zweier Elemente.

Diese Vergleichbarkeit ist z.B. bei Zahlen selbstverständlich. Bei Buchstaben ebenfalls, da gilt die alphabetische Ordnung. Der Vergleich zweier Buchstabenfolgen (also Wörter) gemäss alphabetischer Ordnung ist hingegen schon etwas komplizierter.

Wir werden uns hier auf das Sortieren von Arrays ganzer Zahlen beschränken.

Aufgabe (1.2)
In welcher Reihenfolge stehen folgende Namen im Telefonbuch?

Vontobel
Vogel
Völki
Vock
Vollenweider
von Salis
Vögeli

Aufgabe (1.3)
Welche Probleme ergeben sich, wenn man eine Schulklasse nach
a) Haarfarbe
b) Geschlecht
c) Hobbies
sortieren möchte?

1.3 Sortierrichtung

Sortieren ist immer in zwei Richtungen möglich: Bei aufsteigender Sortierung kommt das kleinste Element zuerst, das grösste zuletzt. Bei absteigender Sortierung ist es genau umgekehrt.

Wir werden hier praktisch ausschliesslich aufsteigend sortieren.

1.4 Wann ist ein Array sortiert?

Bei dieser Frage hilft uns die Geschichte des Mistkäfers Willi. Nachdem er viele Paare vertauscht hat, bei denen die linke Kugel grösser war als die rechte, fand er irgendwann kein solches "falsches" Paar mehr. Natürlich war zu diesem Zeitpunkt die Reihe vollständig sortiert.

Warum funktioniert dieses Verfahren eigentlich? Nun, jede Vertauschung bringt eine grössere Kugel ein Stück nach rechts und einen kleinere Kugel ein Stück nach links. Auf diese Weise trägt jede Vertauschung ein kleines Stück zur Sortierung bei. Jede Vertauschung macht also die Sortierung ein bisschen besser. Die Sortierung ist perfekt, wenn es keine falschen Paare mehr gibt.

Wir halten also fest: Ein Array ist sortiert, wenn es keine zwei Elemente mit falscher Reihenfolge gibt.

Es ist leicht einzusehen, dass auch die folgende Formulierung gilt: Ein Array ist sortiert, wenn es keine zwei *benachbarten* Elemente mit falscher Reihenfolge gibt.

Aufgabe (1.4)
Programmieren Sie eine Funktion

```
bool istSortiert(int[] a)
```

welche herausfindet, ob ein gegebener Array sortiert ist. Testen Sie die Funktion mit verschiedenen Arrays.

1.5 Begriff des Algorithmus

Unter einem **Algorithmus** versteht man ein Problemlösungsverfahren mit folgenden Merkmalen (sinngemäss aus *J. Hromkovič: Sieben Wunder der Informatik*):

- Das Verfahren muss so einfach und präzise formuliert sein, dass es auch von jemandem angewendet werden kann, der kein Experte für das zu lösende Problem ist.
- Das Verfahren muss nicht nur einen einzelnen Problemfall lösen können, sondern alle Fälle eines bestimmten Problemtyps.
- Das Verfahren muss in jedem Fall in endlicher Zeit das korrekte Ergebnis liefern.

Beispiele von Algorithmen aus dem Alltag sind Kochrezepte oder eine Bauanleitung für ein Modellflugzeug.

Algorithmen kann man in natürlicher Sprache oder in speziell dafür entwickelten Ablaufdiagrammen formulieren. Für Informatiker sind natürlich vor allem solche Algorithmen von Interesse, welche sich mit Hilfe einer Programmiersprache auf einem Computer umsetzen lassen.

Aufgabe (1.5)
Nennen Sie zwei weitere Beispiele von Algorithmen aus dem Alltag.

1.6 Zusammenfassung

In diesem Abschnitt haben uns zunächst in Erinnerung gerufen, welche wichtige Bedeutung sortierte Daten in unserem Alltag haben.

Wenn wir Daten oder Gegenstände sortieren wollen, müssen wir uns stets überlegen, nach welchem *Kriterium* dies geschehen soll. Ein Merkmal ist nur dann als Sortierkriterium geeignet, wenn zwei beliebige Elemente bezüglich dieses Merkmals miteinander *vergleichbar* sind.

In den folgenden Abschnitten werden wir verschiedene Sortierverfahren entwickeln. Um diese Verfahren auf dem Computer implementieren zu können, müssen wir sie *präzise* formulieren. Wir werden auch darauf achten, dass sie möglichst *allgemeingültig* sind. Das bedeutet, dass ein Verfahren Zahlenreihen beliebiger Länge sortieren können muss, und zwar unabhängig davon, in welcher Reihenfolge die Zahlen vor der Sortierung angeordnet sind. Ausserdem müssen unsere Verfahren in jedem Fall in *endlicher Zeit* die Lösung finden. Diese gewünschten Merkmale unserer Sortierverfahren haben wir unter dem Begriff *Algorithmus* zusammengefasst.

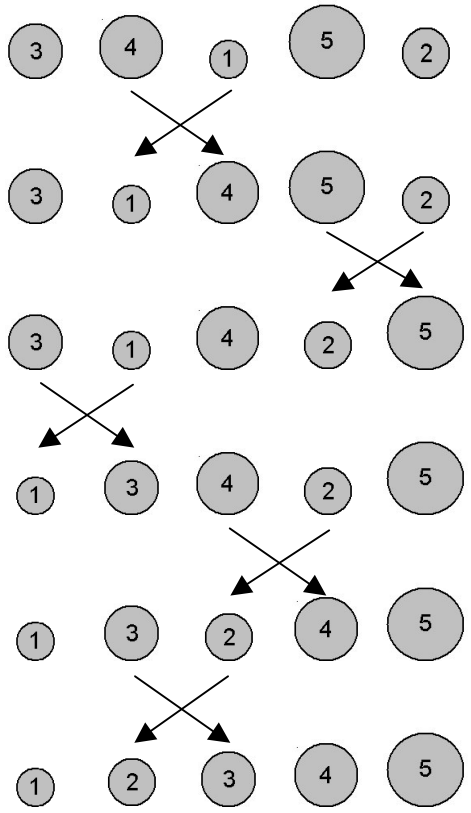
2 BubbleSort

Lernziele
Nach der Bearbeitung dieses Kapitels
- verstehen Sie die Funktionsweise eines einfachen Sortierverfahrens namens BubbleSort.
- können Sie dieses Sortierverfahren und Varianten davon programmieren.
- kennen Sie eine Methode, wie man die "Geschwindigkeit" eines Sortierverfahrens messen kann, und können entsprechenden Code in ihr Programm einbauen.

2.1 Die Idee

Oje, Willi hat seine Mistkugeln auf offenem Feld aufgereiht. Ein heftiger Windstoss und die Kugeln rollen davon und liegen schon wieder in einem Durcheinander. Also nochmals sortieren. Willi ist noch ganz erschöpft von seiner ersten Sortier-Aktion. Vor allem das Suchen der "falschen" Paare war sehr anstrengend! Er musste dabei immer die ganze Kugelreihe überblicken. Gut, ich machs nochmals, sagt er sich, aber diesmal ein bisschen systematischer. Er sucht die "falschen" Paare jetzt nicht mehr nach dem Zufallsprinzip, sondern geht von links nach rechts durch die Reihe. Dabei vergleicht er immer zwei benachbarte Kugeln. Sobald er zwei Kugeln mit falscher Reihenfolge gefunden hat, vertauscht er diese. Nachdem er die ganze Reihe so abgearbeitet hat, beginnt er wieder vorn vorne, d.h. von links. Wenn es schon mit Zufall geklappt hat, denkt sich Willi, dann muss es doch mit System erst recht klappen. Vielleicht sogar noch schneller als vorher? Tatsächlich findet er nach einigen Durchgängen kein "falsches" Paar mehr...

Wir spielen ein Beispiel durch. Der Einfachheit halber nehmen wir an, dass alle Kugeln verschieden gross sind und entsprechend ihrer Grösse beschriftet sind:

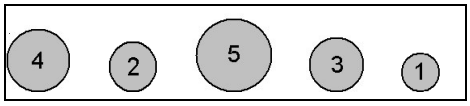
Beginn erster Durchgang	
Beginn zweiter Durchgang	
Beginn dritter Durchgang	
Nach dem dritten Durchgang gibt es kein "falsches" Paar mehr. Fertig!	

Schauen wir uns an, was genau passiert ist:

Erster Durchgang: Zuerst wurde die 4 mit der 1 vertauscht, dann noch die 5 mit der zwei. Am Ende des ersten Durchgangs liegt also die 5 ganz rechts
Zweiter Durchgang: Zuerst wird die 3 mit der 1 vertauscht, dann noch die 4 mit der 2. Am Ende des zweiten Durchgangs liegt also die 4 an der zweiten Stelle von rechts.
Dritter Durchgang: Die 3 wird mit der 2 vertauscht. Am Ende des dritten Durchgangs liegt also die 3 an der dritten Stelle von rechts.

Wenn man sich anstelle der Mistkugeln Luftblasen in einem Aquarium vorstellt und das Ganze um 90° gegen den Uhrzeigersinn dreht, sieht man in Gedanken, wie die *Luftblasen* im Wasser *aufsteigen*. Zuerst die grösste ganz nach oben, dann die zweitgrösste usw. Die Vorstellung von den steigenden Luftblasen hat diesem Sortierverfahren seinen Namen gegeben: **BubbleSort** (*bubble (engl.) = Blase*).

Aufgabe (2.1)
Sortieren Sie folgende Reihe mit dem BubbleSort-Verfahren. Füllen Sie dazu wie im obigen Beispiel eine Tabelle aus.



2.2 Der Algorithmus

Wir sehen im Beispiel auf Seite 7 sofort, dass die Reihe nach dem dritten Durchgang sortiert ist und wir damit fertig sind. Wie aber kann ein Programm herausfinden, ob eine Reihe sortiert ist? Richtig: Es muss alle Elemente durchgehen und "falsche" Paare finden (siehe Einleitung, Abschnitt "Wann ist ein Array sortiert?"). Das ist aber genau das, was in einem Durchgang geschieht. Um zu entscheiden, ob die Reihe sortiert ist, kann unser Programm also einfach einen weiteren Durchgang ausführen. Wenn in einem Durchgang kein "falsches" Paar mehr gefunden wird, sind wir fertig.

Diese Überlegungen führen uns auf folgenden Algorithmus:

Algorithmus

Schritt	Was tun
(1)	Wähle die ersten beiden Elemente von links.
(2)	Ist beim gewählten Paar das linke Element grösser als das rechte? Wenn ja, vertausche die beiden.
(3)	Hat es rechts des soeben untersuchten Paares noch weitere Kugeln? Wenn ja, wähle das nächste Paar benachbarter Elemente und gehe zu (2).
(4)	Wir haben einen Durchgang beendet. Wurde in diesem Durchgang mindestens ein "falsches" Paar gefunden und vertauscht? Wenn ja, gehe zu (1), d.h. starte einen weiteren Durchgang. Wenn nein, sind wir fertig.

Das Beispiel von Seite 7 mit Zahlen und streng nach Anleitung:

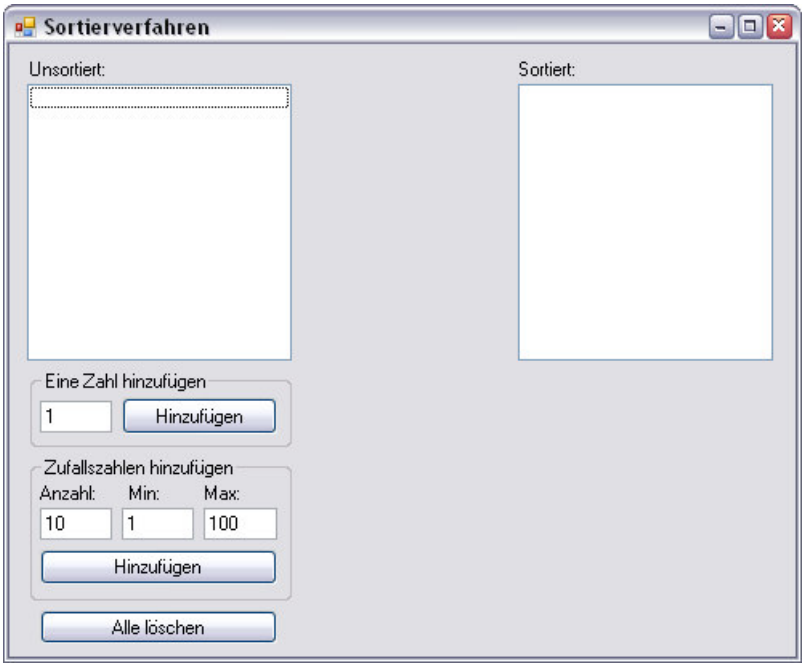
Schritt	Was tun	Ergebnis dieses Schrittes				
	Ausgangslage	3	4	1	5	2
(1)	Wähle das erste Paar: 3 und 4	3	4	1	5	2
(2)	3 ist nicht grösser als 4, also nichts tun	3	4	1	5	2
(3)	Wähle nächstes Paar: 4 und 1	3	4	1	5	2
(2)	4 ist grösser als 1, also vertauschen	3	1	4	5	2
(3)	Wähle nächstes Paar: 4 und 5	3	1	4	5	2
(2)	4 ist nicht grösser als 5, also nichts tun	3	1	4	5	2
(3)	Wähle nächstes Paar: 5 und 2	3	1	4	5	2
(2)	5 ist grösser als 2, also vertauschen	3	1	4	2	5
(3)	Es hat weiter rechts kein Paar mehr.	3	1	4	2	5
(4)	In diesem Durchgang wurden 2 Vertauschungen gemacht, also noch ein Durchgang.	3	1	4	2	5
(1)	Wähle das erste Paar: 3 und 1	3	1	4	2	5
(2)	3 ist grösser als 1, also vertauschen	1	3	4	2	5
(3)	Wähle nächstes Paar: 3 und 4	1	3	4	2	6
(2)	3 ist nicht grösser als 4, also nichts tun	1	3	4	2	5
(3)	Wähle nächstes Paar: 4 und 2	1	3	4	2	5
(2)	4 ist grösser als 2, also vertauschen	1	3	2	4	5
(3)	Wähle nächstes Paar: 4 und 5	1	3	2	4	5
(2)	4 ist nicht grösser als 5, also nichts tun	1	3	2	4	5
(3)	Es hat weiter rechts kein Paar mehr.	1	3	2	4	5
(4)	In diesem Durchgang wurden 2 Vertauschungen gemacht, also noch ein Durchgang.	1	3	2	4	5
(1)	Wähle das erste Paar: 1 und 3	1	3	2	4	5
(2)	1 ist nicht grösser als 3, also nichts tun	1	3	2	4	5
(3)	Wähle nächstes Paar: 3 und 2	1	3	2	4	5
(2)	3 ist grösser als 2, also vertauschen	1	2	3	4	5
(3)	Wähle nächstes Paar: 3 und 4	1	2	3	4	5
(2)	3 ist nicht grösser als 4, also nichts tun	1	2	3	4	5
(3)	Wähle nächstes Paar: 4 und 5	1	2	3	4	5
(2)	4 ist nicht grösser als 5, also nichts tun	1	2	3	4	5

(3)	Es hat weiter rechts kein Paar mehr.	1	2	3	4	5
(4)	In diesem Durchgang wurde 1 Vertauschung gemacht, also noch ein Durchgang.	1	2	3	4	5
(1)	Wähle das erste Paar: 1 und 2	1	2	3	4	5
(2)	1 ist nicht grösser als 2, also nichts tun	1	2	3	4	5
(3)	Wähle nächstes Paar: 2 und 3	1	2	3	4	5
(2)	2 ist nicht grösser als 3, also nichts tun	1	2	3	4	5
(3)	Wähle nächstes Paar: 3 und 4	1	2	3	4	5
(2)	3 ist nicht grösser als 4, also nichts tun	1	2	3	4	5
(3)	Wähle nächstes Paar: 4 und 5	1	2	3	4	5
(2)	4 ist nicht grösser als 5, also nichts tun	1	2	3	4	5
(3)	Es hat weiter rechts kein Paar mehr.	1	2	3	4	5
(4)	In diesem Durchgang wurden keine Vertauschungen gemacht, also sind wir fertig.	1	2	3	4	5
	Fertig	1	2	3	4	5

Aufgabe (2.2)
Spielen Sie den BubbleSort-Algorithmus an folgendem Zahlenbeispiel durch. Halten Sie sich dabei genau an die Anleitung und füllen sie wie im obigen Beispiel eine Tabelle aus.

11 3 6 7 3 6

Aufgabe (2.3)
An dieser Stelle wollen wir das Gerüst für ein grösseres Programm erstellen. Dieses Programm werden wir nach und nach erweitern, indem wir alle hier behandelten Sortierverfahren einbauen.
Erzeugen Sie zunächst in einem neuen Windows-Projekt folgendes Formular:



Klick auf einen "Hinzufügen"-Button soll einen bzw. mehrere Einträge zu der linken ListBox hinzufügen. Klick auf den "Alle löschen"-Button soll die linke ListBox leeren. Programmieren Sie die Buttons entsprechend.

Aufgabe (2.4)
Erstellen Sie in ihrem Programm eine Funktion

```
void bubbleSort(int[] a)
```

welche den BubbleSort-Algorithmus gemäss obiger Anleitung implementiert. Verwenden Sie Schleifen, keine goto-Befehle.

Erzeugen Sie im Formular zwischen den beiden Listboxen einen Button "BubbleSort", welcher aus dem Inhalt der linken ListBox einen Array von Zahlen erzeugt, damit die Funktion bubbleSort() aufruft und den sortierten Array in der rechten Listbox ausgibt.

Tipp: Erstellen Sie Funktionen für die Erzeugung eines Arrays aus der linken ListBox sowie für die Ausgabe eines Arrays in die rechte ListBox. Wir werden sie noch öfters brauchen.

Testen Sie ihr erstes Sortierprogramm mit verschiedenen Arrays!

Aufgabe (2.5)
Was passiert, wenn man im Schritt 2 des BubbleSort-Algorithmus den Begriff "grösser als" durch
a) "kleiner als"
b) "grösser als oder gleich"
c) "kleiner als oder gleich"
ersetzt?

2.3 Optimierungen

Erste Optimierung

Wie wir im Beispiel auf Seite 7 gesehen haben, wird im ersten Durchgang die grösste Kugel an ihren richtigen Platz transportiert (nämlich ganz nach rechts), im zweiten Durchgang die zweitgrösste (nämlich an die zweite Position von rechts) usw.
Bei einem Array der Grösse n werden also in den ersten n-1 Durchgängen die n-1 grössten Elemente (das sind alle ausser das kleinste) an ihren endgültigen Platz gebracht. Wenn aber alle Elemente bis auf das kleinste am richtigen Platz sind, dann muss zwangsläufig das kleinste auch schon am richtigen Platz sein, nämlich ganz links. Eine andere Möglichkeit gibt es nicht!
Mit anderen Worten: Bei einem Array der Grösse n wird im n-ten Durchgang sicher nichts mehr vertauscht. Es sind also höchstens n-1 Durchgänge nötig. Man kann auch direkt eine feste Zahl von n-1 Durchgängen ausführen. Dadurch erspart man sich die Abfrage, ob in einem Durchgang etwas vertauscht wurde.

Aufgabe (2.6)
Was ist ein möglicher Nachteil, wenn wir eine fixe Anzahl (nämlich n-1) Durchgänge ausführen?

Zweite Optimierung

Nach dem ersten Durchgang steht das grösste Element ganz rechts. Nach dem zweiten Durchgang steht das zweitgrösste Element an zweiter Stelle von rechts. Nach i Durchgängen steht das i-t-grösste Element an der i-ten Stelle von rechts, und die i Elemente rechts sind sortiert. Es muss also nicht jedes Mal der ganze Array durchlaufen werden, sondern im i-ten Durchgang nur die ersten (n-i+1) Elemente.

Aufgabe (2.7)
Streichen Sie in der Tabelle, welche Sie in der Aufgabe (2.2) erstellt haben, alle Zeilen durch, welche gemäss unserer zweiten Optimierung unnötig sind.

Aufgabe (2.8)
Erstellen Sie in ihrem Programm eine Funktion

```
void bubbleSortOpt(int[] a)
```

welche BubbleSort mit den beiden besprochenen Optimierungen implementiert. Erzeugen Sie einen Button "BubbleSort Opt", welche die neue Funktion aufruft und das Ergebnis in die rechte ListBox ausgibt.

Aufgabe (2.9)
Erstellen Sie in ihrem Programm eine Funktion

```
void bubbleSortAbsteigend(int[] a)
```

welche den Array mit BubbleSort absteigend sortiert.

2.4 Aufwandmessungen

Um den Zeitaufwand bzw. die "Geschwindigkeit" eines Sortierverfahrens zu messen, sind verschiedene Möglichkeiten denkbar: Man könnte zum Beispiel die Laufzeit in Sekunden messen. Dies wäre allerdings abhängig von der verwendeten Maschine und möglicherweise auch von anderen, gleichzeitig laufenden Programmen.

Wir wollen den Aufwand unserer Verfahren messen, indem wir zwei Grössen bestimmen: Die Anzahl Vergleiche und die Anzahl Vertauschungen von Array-Elementen. Sämtliche hier behandelten Verfahren basieren nämlich ausschliesslich auf diesen beiden Operationen. Den übrigen Aufwand, z.B. für die Ablaufsteuerung von Schleifen, ignorieren wir grosszügig.

Aufgabe (2.10)
Bauen Sie Ihr Programm so um, dass Vergleiche und Vertauschungen von Array-Elementen gezählt werden. Deklarieren dazu zwei entsprechende globale Variablen:

```
int anzahlVergleiche = 0;
int anzahlVertauschungen = 0;
```

Programmieren Sie geeignete Funktionen für den Vergleich und das Vertauschen von zwei Array-Elementen, welche als Nebeneffekt den entsprechenden Zählen erhöhen:

```
bool istGroesser(int x, int y)
```

und

```
void vertauschen(int[] a, int i, int j)
```

Ersetzen Sie anschliessend alle Vergleiche und Vertauschungen in den Sortierfunktionen bubbleSort() und bubbleSortOpt() durch entsprechende Funktionsaufrufe.

Sorgen Sie dafür, dass die Zähler zu Beginn der Sortierung auf 0 gesetzt werden und nach der Sortierung im Formular angezeigt werden.

Probieren Sie das Programm mit dem Zählmechanismus mit Zufalls-Arrays verschiedener Grössen aus und füllen Sie folgende Tabellen aus:

Anzahl Vergleiche					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
BubbleSort					
BubbleSort Opt					

Anzahl Vertauschungen					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
BubbleSort					
BubbleSort Opt					

Was fällt auf?

Zusatzaufgabe (2.11)
Implementieren Sie das vom Mistkäfer Willi im ersten Kapitel entdeckte Sortierverfahren, welches auf der zufälligen Auswahl falsch sortierter Paare beruht. Erstellen Sie dazu die Funktion

```
void randomSort(int[] a)
```

und bauen Sie sie in ihr Programm ein.

2.5 Zusammenfassung

Mit BubbleSort haben wir ein einfaches Sortierverfahren kennengelernt und implementiert. Gleichzeitig haben wir auch Vorbereitungen für die folgenden Kapitel getroffen, indem wir ein Programmgerüst entwickelt haben, in welches wir auch weitere Sortiermethoden einbauen können.

Als Mass für den Aufwand eines Sortier-Algorithmus haben wir die Anzahl Vergleiche und Vertauschungen eingeführt und in unser Programm eingebaut. Mit diesem Instrument werden wir der Lage sein, den Aufwand verschiedener Verfahren miteinander zu vergleichen. Wir haben bereits erfahren, wie sich einfache Optimierungen von BubbleSort günstig auf die Laufzeit auswirken.

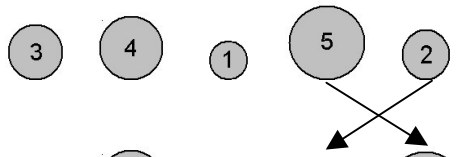
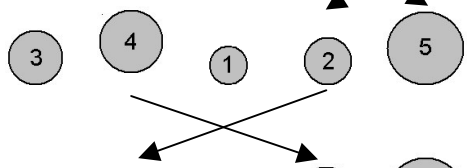
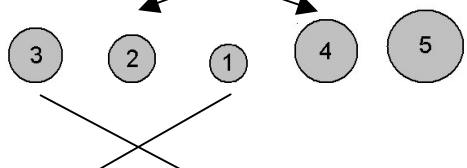
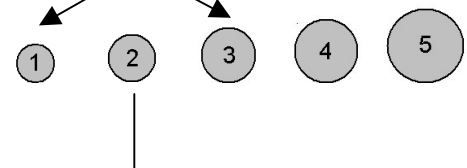
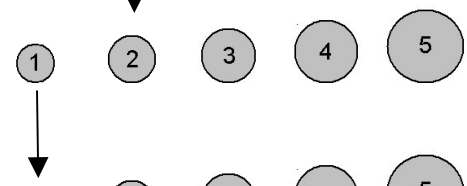
3 SelectionSort

Lernziele
Nach der Bearbeitung dieses Kapitels
- verstehen Sie die Funktionsweise eines Sortierverfahrens namens SelectionSort.
- können Sie dieses Sortierverfahren und Varianten davon programmieren.

3.1 Die Idee

Ein furchtbares Gewitter ist durch das Land gezogen und hat Willis fein säuberlich sortierte Mistkugelsammlung total durcheinander gebracht. Diesmal ist Willi aber guten Mutes. Er weiss ja nun, wie er die Kugeln wieder in die richtige Reihenfolge bringen kann. Schon will er ans Werk gehen. Aber halt! Die Kugeln sind vom Regen noch ganz schmutzig. Muss er wirklich jede Kugel so oft anfassen wie beim ersten Mal? Nicht dass er sich vor seinen Mistkugeln ekeln würde, aber die Kugeln könnten dabei Schaden nehmen. Er setzt sich hin und überlegt. Am Schluss muss doch die grösste Kugel ganz rechts liegen, das ist klar. Warum suche ich nicht einfach die grösste Kugel und lege sie ganz nach rechts? Gedacht, getan. Das ist aber gar nicht so einfach, denn die beiden grössten Kugeln sind fast gleich gross. Willi nimmt sein Messband und misst die Umfänge der beiden Kandidaten. Das Resultat ist eindeutig. Er bringt also die grösste Kugel an die Position ganz rechts, indem er sie mit der Kugel vertauscht, welche vorher dort war. Die zweitgrösste Kugel bekommt den Platz direkt links neben der grössten, ebenfalls durch Vertauschen. So fährt Willi nun fort: Er bestimmt die drittgrösste Kugel und bringt sie an die dritte Position von rechts und so weiter. Es dauert gar nicht lange, bis er fertig ist. Zufrieden schaut er sein Werk an und überlegt sich, wieviel Mal er jede Kugel anfassen musste...

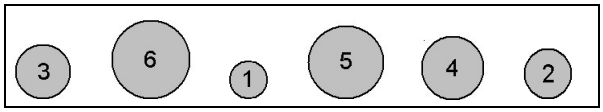
Unser Beispiel aus dem ersten Kapitel, sortiert mit Willis neuen Verfahren:

Vertausche grösste Kugel (5) mit der Kugel ganz rechts (2)	
Vertausche zweitgrösste Kugel (4) mit der zweiten Kugel von rechts (2)	
Vertausche drittgrösste Kugel (3) mit der dritten Kugel von rechts (1)	
Vertausche viertgrösste Kugel (2, zweitkleinste) mit der vierten Kugel von rechts (2, zweite von links) - nichts tun	
Vertausche fünftgrösste (kleinste) Kugel (1) mit der fünften Kugel von rechts (ganz links, 1) - nichts tun	
Es gibt kein "falsches" Paar mehr. Fertig!	

In diesem Beispiel sind wir mit drei Vertauschungen ausgekommen. Die letzten beiden Kugeln waren schon am richtigen Ort. War das Zufall? Bei der vorletzten schon, aber bei der letzten nicht. Wenn nämlich alle Kugeln ausser die kleinste an ihren richtigen Platz verschoben wurden, dann muss die kleinste zwangsläufig auch schon an ihrem richtigen Platz stehen, nämlich ganz links.

Dieses Verfahren trägt den Namen **SelectionSort**, weil in jedem Durchgang das nächstgrösste Element des Arrays *ausgewählt* und an seinen endgültigen Platz gesetzt wird (*selection (engl.) = Auswahl*).

Aufgabe (3.1)
Sortieren Sie folgende Reihe mit dem SelectionSort-Verfahren. Füllen Sie dazu wie im obigen Beispiel eine Tabelle aus. Lassen Sie den letzten, unnötigen Durchgang weg.



3.2 Der Algorithmus

Den Algorithmus für einen Array mit n Elementen könnte man so formulieren:

Algorithmus

Schritt	Was tun
(1)	Suche das grösste Element des ganzen Arrays und vertausche es mit dem Element ganz rechts.
(2)	Suche das zweitgrösste Element des ganzen Arrays und vertausche es mit dem zweiten Element von rechts.
...	... und so weiter...
(n-1)	Suche das zweitkleinste Element und vertausche es mit dem zweiten Element von links.

Wir wollen uns aber mit dieser saloppen "und so weiter"-Formulierung nicht zufrieden geben. Präziser ist die folgende Darstellung:

Algorithmus

Schritt	Was tun
(1)	Setze einen Zähler i auf 1.
(2)	Suche das i-t-grösste Element (*).
(3)	Vertausche es mit dem i-ten Element von rechts.
(4)	Erhöhe die den Zähler i um 1. Wenn i kleiner als die Arraylänge ist, gehe zu (2), sonst sind wir fertig.

(*) mit dem "i-grössten Element" ist folgendes gemeint:

i	"i-t-grösstes Element"
1	grösstes
2	zweitgrösstes
3	drittgrösstes
...	...

Willis Beispiel:

Schritt	Was tun	i	Ergebnis dieses Schrittes				
	Ausgangslage		3	4	1	5	2
(1)	Setze i auf 1	1	3	4	1	5	2
(2)	Suche das grösste Element: 5	1	3	4	1	5	2
(3)	Vertausche es mit dem Element ganz rechts: 2	1	3	4	1	2	5
(4)	Erhöhe i um 1	2	3	4	1	2	5
(2)	Suche das zweitgrösste Element: 4	2	3	4	1	2	5
(3)	Vertausche es mit dem zweiten Element von rechts: 2	2	3	2	1	4	5
(4)	Erhöhe i um 1	3	3	2	1	4	5
(2)	Suche das drittgrösste Element: 3	3	3	2	1	4	5
(3)	Vertausche es mit dem dritten Element von rechts: 1	3	1	2	3	4	5
(4)	Erhöhe i um 1	4	1	2	3	4	5
(2)	Suche das viertgrösste Element: 2	4	1	2	3	4	5
(3)	Vertausche es mit dem vierten Element von rechts (Vertauschung mit sich selber, d.h. nichts tun)	4	1	2	3	4	5
(4)	Erhöhe i um 1. i ist nicht kleiner als die Arraylänge, also sind wir fertig.	5	1	2	3	4	5
	Fertig		1	2	3	4	5

Bevor wir diesen Algorithmus programmieren, wollen wir noch ein wichtiges Detail klären: Im Schritt (2) müssen wir das i-t-grösste Element des Arrays bestimmen. Dieses Problem allgemein zu lösen ist gar nicht so einfach. Glücklicherweise ist es aber in unserem Fall nicht schwierig.
Schauen wir uns nochmals den obigen Ablauf an: Zu Beginn des zweiten Durchgangs ist das grösste Element ganz rechts, zu Beginn des dritten Durchgangs ist zusätzlich das zweitgrösste an zweiter Stelle von rechts. Allgemein: Zu Beginn des Durchgangs i befinden sich die (i-1) grössten Elemente auf der rechten Seite des Arrays, und zwar bereits in sortierter Reihenfolge und damit an ihrer endgültigen Position. An diesem Teil des Arrays müssen wir also gar nichts mehr ändern. Dieser bereits sortierte Bereich ist in der obigen Tabelle grau schraffiert.
Wie finden wir nun das i-t-grösste Element? Ganz einfach: Es ist das grösste Element des linken, noch unsortierten Teils unseres Arrays.

Hier die vorläufig letzte Version unseres Algorithmus. Sie sieht vielleicht ein bisschen komplizierter aus als die vorherige, ist aber nachher einfacher zu programmieren:

Algorithmus

Schritt	Was tun
(1)	Setze den Zähler i auf 1.
(2)	Suche das grösste Element der (n-i+1) Elemente von links.
(3)	Vertausche es mit dem i-ten Element von rechts.
(4)	Erhöhe den Zähler i um 1. Wenn i kleiner als die Arraylänge ist, gehe zu (2), sonst sind wir fertig.

Aufgabe (3.2)
Spielen Sie den SelectionSort-Algorithmus an folgendem Zahlenbeispiel durch. Halten Sie sich dabei genau an die Anleitung und füllen sie wie im obigen Beispiel eine Tabelle aus.

9 3 8 8 2 6

Aufgabe (3.3)
Implementieren Sie den SelectionSort-Algorithmus gemäss obiger Ableitung und bauen Sie ihn in ihr Programm ein. Verwenden Sie Schleifen und keine goto-Anweisungen. Bauen Sie von Anfang an die Zählung der Anzahl Vergleiche und Vertauschungen ein.

Zusatzaufgabe (3.4)
a) Wie oft wird das grösste Element bei SelectionSort höchstens vertauscht? Wie oft das kleinste?
b) Wieviel mal wird ein Element im Durchschnitt vertauscht?

Hinweis: Gehen Sie bei dieser Aufgabe davon aus, dass Vertauschungen "stur" ausgeführt werden, d.h. es gibt auch unnötige Vertauschungen eines Elements mit sich selber. Diese werden ebenfalls gezählt.

3.3 Aufwandmessungen

Aufgabe (3.5)
Wenden Sie SelectionSort auf Zufalls-Arrays verschiedener Grössen an und tragen Sie die Anzahl Vergleiche und Vertauschungen in eine Tabelle ein.

Anzahl Vergleiche					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
SelectionSort					

Anzahl Vertauschungen					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
SelectionSort					

Zusatzaufgabe (3.6)
Folgender naheliegender Gedanke führt uns zu einer rekursiven Variante von SelectionSort: Man suche das grösste Element des Arrays und bringe es an die Position ganz rechts. Anschliessend sortiere man den verbleibenden Array, also ohne das grösste Element.

Implementieren Sie die rekursive Variante von SelectionSort. Für eine einfache Anwendbarkeit empfiehlt es sich, zwei Funktionen zu erstellen:

Die rekursive Funktion

```
void selectionSortRek(int[] a, int n)
```

implementiert den eigentlichen rekursiven SelectionSort-Algorithmus. Diese Funktion hat zwei Parameter: den Array a und die Länge n, bis zu der sortiert werden soll.

Die Funktion

```
void selectionSort3(int[] a)
```

kann von aussen ganz normal aufgerufen werden. Sie enthält nur eine einzige Anweisung, nämlich den Aufruf von selectionSortRek() mit dem ganzen Array a.

Zusatzaufgabe (3.7)
In der Praxis müssen häufig Listen sortiert werden, welche nicht Zahlen enthalten, sondern andere, unter Umständen zusammengesetzte Elemente.

Erstellen Sie ein Programm, mit dem der Benutzer eine Liste von Adressen erfassen und diese nach Name, Postleitzahl oder Ort sortieren kann.

3.4 Zusammenfassung

SelectionSort stellt im Vergleich zu BubbleSort eine wesentliche Verbesserung in bezug auf die Laufzeit dar. Der Grund liegt darin, dass bei SelectionSort ein Element direkt an seinen definitiven Platz gesetzt wird, anstatt wie bei BubbleSort durch fortlaufendes Vertauschen benachbarter Elemente schrittweise dorthin gebracht zu werden. Wir haben gesehen, wie dadurch die Anzahl Vertauschungen dramatisch reduziert wird.

Trotz der massiven Verbesserung ist SelectionSort gegenüber BubbleSort nicht etwa komplizierter oder schwerer zu verstehen. Im Gegenteil: Für viele dürfte die Funktionsweise von SelectionSort sogar noch einleuchtender sein als die von BubbleSort.

4 QuickSort

Lernziele

Nach der Bearbeitung dieses Kapitels

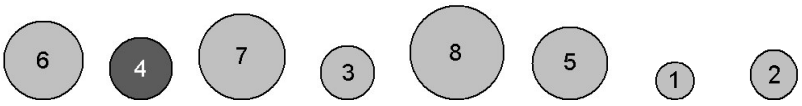
- verstehen Sie die Funktionsweise eines der schnellsten Sortierverfahren.
- können Sie die wichtigsten Begriffe erklären, welche in Zusammenhang mit diesem Verfahren von Bedeutung sind.

4.1 Die Idee

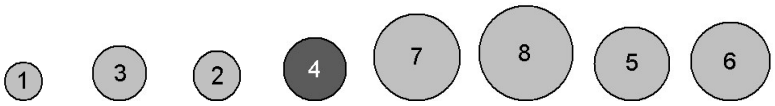
Der Sommer war lang und heiss. Willi konnte seine Mistkugelsammlung um viele schöne Exemplare erweitern. Leider hat er in seinem Sammeleifer die Ordnung ein bisschen vernachlässigt. Wieder einmal liegen die Kugeln unsortiert in einer Reihe. Nun möchte Willi die Ordnung wieder herrichten. Aber so viele Kugeln sortieren? Willi seufzt bei dem Gedanken. Da fällt sein Blick auf eine besonders schöne Kugel. Sie ist etwa von mittlerer Grösse und etwas dunkler als die anderen. An welcher Position wird wohl diese Kugel stehen, wenn einst alle sortiert sind? Wahrscheinlich ungefähr in der Mitte. Und wenn ich es genau wissen möchte? sinniert Willi. Er überlegt eine Weile. Plötzlich steht er auf, nimmt seine Lieblingskugel und platziert sie ungefähr in der Mitte der Kugelreihe. Dann sucht er alle Kugeln heraus, welche kleiner sind als die Lieblingskugel und schafft sie auf die linke Seite. Anschliessend bringt er alle Kugeln, die grösser sind als seine Lieblingskugel, auf die rechte Seite. Moment mal, überlegt sich Willi. Nun liegen also alle kleineren Kugeln links von meiner Lieblingskugel und alle grösseren rechts von von ihr. Das bedeutet doch nichts anderes als dass meine Lieblingskugel jetzt schon an ihrem endgültigen Platz liegt! Ob ich wohl damit schon einen kleinen Beitrag zur Sortierung geleistet habe? fragt sich Willi und betrachtet lange seine Mistkugelreihe.

Da kommt Willis Freund, der Frosch, des Weges. "Na, wieder mal am sortieren?", fragt der Frosch scherzhaft. "Leider komme ich nicht weiter", klagt Willi. "Ich habe diesen Sommer so viele Kugeln gesammelt. Ich glaube ich schaffe es nicht, sie je in die richtige Reihenfolge zu bringen." "Aber so schlecht sieht das doch gar nicht aus", sagt der Frosch. "Du hast ja schon einen Anfang gemacht. Oh, und was ist denn das dort für ein Prachtsexemplar?" fragt der Frosch und zeigt auf Willis Lieblingskugel. "Ja, das ist eine besonders schöne. Sie ist mir vorhin aufgefallen, und da habe ich - halb aus Spass, halb aus Verzweiflung - die anderen Kugeln so angeordnet. Aber sortiert sind sie deswegen noch lange nicht.", seufzt Willi. "Weisst du was?", sagt der Frosch nach einer kurzen Pause. "Ich helfe dir. Geteilte Arbeit ist halbe Arbeit. Ich sortiere den linken Teil deiner Reihe, also den mit den kleinen Kugeln, und du machst das gleiche mit dem rechten Teil, dort wo die grossen liegen. Und deine Lieblingskugel lassen wir schön brav an ihrem Platz liegen, damit sie ja nicht kaputt geht." "Danke, das ist sehr nett von dir", sagt Willi. "Aber wie willst du das anstellen?" "Weiss ich noch nicht so genau", sagt der Frosch, "aber vielleicht hole ich noch meine Freundin, die Schnecke, zu Hilfe...".

Die Ausgangslage (unsortiert):



Nach Willis Umstellung: Alle Kugeln kleiner als 4 sind links von der 4, alle grösseren rechts.



Die Überlegungen von Willi und dem Frosch führen uns zu **QuickSort.** Wir werden bald herausfinden, ob dieses Verfahren seinen Namen verdient hat (*quick (engl.) = schnell*).

4.2 Der Algorithmus

Algorithmus

Schritt	Was tun
(1)	Wähle ein beliebiges, idealerweise ungefähr mittelgrosses Element aus dem Array.
(2)	Ordne den Array so um, so dass sich alle Elemente, die kleiner sind als das ausgewählte, links von diesem befinden und alle, die grösser sind, rechts desselben.
(3)	Sortiere den Teil-Array links des gewählten Elementes.
(4)	Sortiere den Teil-Array rechts des gewählten Elementes.

Natürlich gibt es dazu noch einige Fragen zu beantworten bzw. Teilprobleme zu lösen:

Frage 1: Was bedeutet "ungefähr mittelgross"? Wie finden wir ein solches Element?

Das exakt mittlere Element eines Arrays heisst **Median**. Das ist dasjenige Element, zu dem es gleich viele kleinere wie grössere Elemente gibt. Er ist nicht zu verwechseln mit dem arithmetischen Mittel aller Elemente! Den Median eines unsortierten Arrays zu bestimmen ist in der Tat relativ aufwändig.

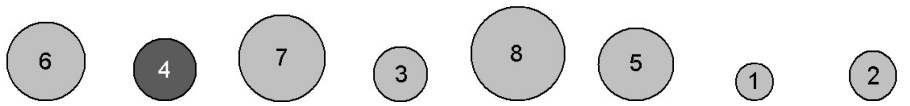
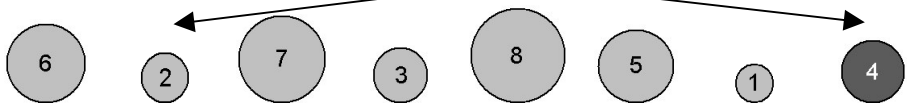
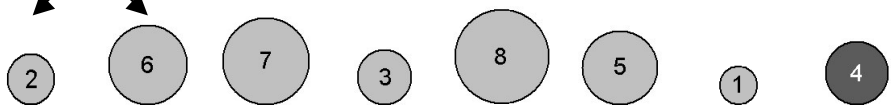
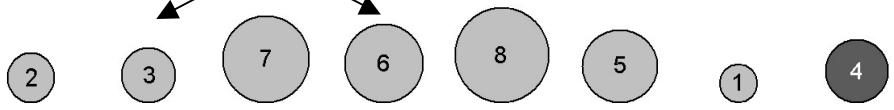
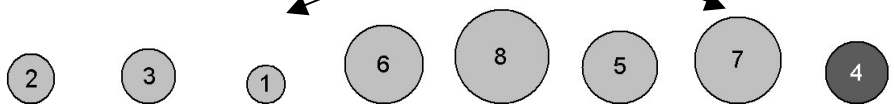
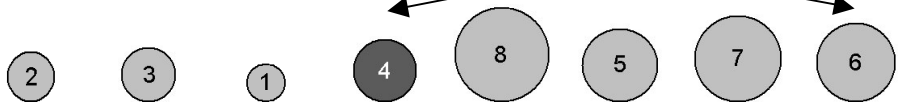
Aufgabe (4.1)
Finden Sie den Median der folgenden Reihen:
a) 3 5 6 2 7 2 4
b) 3 6 4 11 24 3 6 7 1 5

Für die Programmierung von QuickSort brauchen wir uns zum Glück um die Bestimmung des Medians keine Sorgen zu machen. Wir können ein *beliebiges* Element wählen. Der Algorithmus funktioniert trotzdem! Und jetzt noch ein Fachbegriff: Das ausgewählte "Lieblingselement" nennt man **Pivot-Element** oder kurz **Pivot**.

Frage 2: Wie können wir den Array so umordnen, dass sich links des ausgewählten Elementes alle kleineren und rechts davon alle grösseren Elemente befinden?

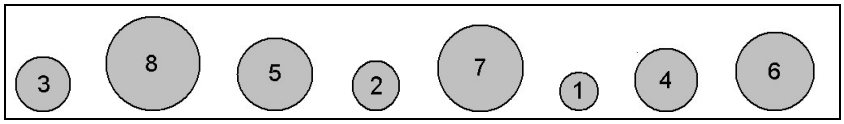
Auch für diesen Vorgang gibt es einen Fachbegriff: Man nennt ihn **Partitionierung**. Die Partitionierung lässt sich zum Beispiel wie folgt erledigen: Zuerst schaffen wir das Pivot-Element aus dem Weg, indem wir es ganz auf die rechte Seite bringen.
Dann suchen wir im ganzen Array (ausser natürlich im letzten Feld) alle Elemente, die kleiner sind als das Pivot-Element und füllen sie von links her auf, indem wir sie mit dem entsprechenden Element vertauschen. Zuletzt bringen wir das Pivot-Element an die richtige Position.

Wir spielen die Partitionierung an Willis Kugelreihe durch:

Schaffe Pivot aus dem Weg	
Bringe erstes Element < Pivot auf Platz 1	
Bringe zweites Element < Pivot auf Platz 2	
Bringe drittes Element < Pivot auf Platz 3	
Es gibt kein weiteres Element < Pivot; bringe Pivot auf Platz 4	
Die Reihe ist jetzt partitioniert.	

Wir haben hier ein anderes Ergebnis bekommen als Willi. Er hat ja die Partitionierung auch nicht genau nach unserem Verfahren durchgeführt. Natürlich sind beide Ergebnisse richtig.

Aufgabe (4.2)
Bestimmen Sie für die folgende Reihe zwei verschiedene Pivot-Elemente ihrer Wahl und führen Sie für jedes die Partitionierung gemäss obigem Verfahren durch:



Nun können wir den Algorithmus für die Partitionierung formulieren. Wir bezeichnen die Schritte mit (2.1), (2.2) usw., da es sich hier um eine Verfeinerung von Schritt (2) des obigen Algorithmus handelt:

Algorithmus

Schritt	Was tun
(2.1)	Bringe das Pivot-Element durch Vertauschen an die letzte Array-Position.
(2.2)	Setze eine Variable k für die Einfügeposition auf die Position ganz links.
(2.3)	Wähle das erste Array-Element.
(2.4)	Ist das gewählte Element kleiner als das Pivot-Element? Wenn ja vertausche es mit dem Element an der Position k und erhöhe dann k um 1.
(2.5)	Gibt es rechts neben dem ausgewählten Element (ausser dem Pivot) noch weitere Elemente? Wenn ja, wähle das nächste aus und gehe zu (2.4).
(2.6)	Bringe das Pivot-Element durch Vertauschen an die Position k.

Hier nochmals die Partitionierung von Willis Beispiel, dieses Mal allerdings mit Zahlen und streng nach Anleitung:

Schritt	Was tun	k	Ergebnis dieses Schrittes							
	Ausgangslage		6	**4**	7	3	8	5	1	2
(2.1)	Bringe Pivot an letzte Stelle		6	2	7	3	8	5	1	**4**
(2.2)	Setze Einfügeposition k auf 1	1	6	2	7	3	8	5	1	**4**
(2.3)	Wähle erstes Element	1	6	2	7	3	8	5	1	**4**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	1	6	2	7	3	8	5	1	**4**
(2.5)	Wähle nächstes Element	1	6	2	7	3	8	5	1	**4**
(2.4)	kleiner als Pivot, also vertauschen mit Pos. 1 und dann k um 1 erhöhen	2	2	6	7	3	8	5	1	**4**
(2.5)	Wähle nächstes Element	2	2	6	7	3	8	5	1	**4**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	2	2	6	7	3	8	5	1	**4**
(2.5)	Wähle nächstes Element	2	2	6	7	3	8	5	1	**4**
(2.4)	kleiner als Pivot, also vertauschen mit Pos. 2 und dann k um 1 erhöhen	3	2	3	7	6	8	5	1	**4**
(2.5)	Wähle nächstes Element	3	2	3	7	6	8	5	1	**4**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	3	2	3	7	6	8	5	1	**4**
(2.5)	Wähle nächstes Element	3	2	3	7	6	8	5	1	**4**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	3	2	3	7	6	8	5	1	**4**
(2.5)	Wähle nächstes Element	3	2	3	7	6	8	5	1	**4**
(2.4)	kleiner als Pivot, also vertauschen mit Pos. 3 und dann k um 1 erhöhen	4	2	3	1	6	8	5	7	**4**
(2.5)	Es gibt kein weiteres Element mehr	4	2	3	1	6	8	5	7	**4**
(2.6)	Bringe Pivot an Position 4	4	2	3	1	**4**	8	5	7	6
	Fertig		2	3	1	**4**	8	5	7	6

Aufgabe (4.3)
Spielen Sie den Partitionierungs-Algorithmus anhand der folgenden Arrays und der gegebenen Pivot-Elemente durch. Füllen Sie dazu wie oben eine Tabelle aus:

a)	5	7	11	3	8	6	9	2,	Pivot = 7
b)	5	8	4	6	3	8	3	1,	Pivot = 3

Aufgabe (4.4)
a) Entwickeln Sie innerhalb ihres Sortierprogramms eine Funktion zur Partinionierung eines Arrays bei gegebenem Pivot-Element:

```
int partitionierenGanz(int[] a, int pivotIndex)
```

Diese Funktion bekommt als Parameter den Array und den Index des Pivot. Sie soll den Index des Pivots nach der Partitionierung, also seine endgültige Position, zurückgeben.

Wir werden diese Funktion später benötigen, um den ganzen QuickSort-Algorithmus zu bauen. Verwenden Sie die weiter oben entwickelten Funktionen zum Vergleichen und Vertauschen zweier Elemente. Auf diese Weise bereiten Sie ihren Code bereits für die Aufwandmessung vor. Testen Sie die Funktion an einigen Beispielen.

b) Eigentlich benötigen wir eine Funktion, welche nicht den ganzen Array partitioniert, sondern nur einen bestimmten Ausschnitt daraus. Erweitern Sie die obige Funktion um zwei Parameter, welche den zu partitionierenden Ausschnitt festlegen:

```
int partitionieren(int[a], int pivotIndex,
                   int vonIndex, int bisIndex)
```

Sie können davon ausgehen, dass vonIndex <= pivotIndex <= bisIndex. Testen Sie ihre Funktion an einigen Beispielen.

Nun noch zur wohl alles entscheidenden

Frage 3: Wie sortieren wir den linken und den rechten Teil-Array?

Die Antwort mag zugleich banal und verblüffend erscheinen: Natürlich auch mit QuickSort! Konkret bedeutet das, dass wir zur Sortierung des linken und des rechten Teil-Arrays wieder QuickSort aufrufen. Jeder Teil-Array wird dann wieder in zwei Teile geteilt, welche wieder in zwei Teile geteilt wird und so weiter. Diese Strategie, ein grosses Problem kleinere Einheiten zu zerlegen und diese nacheinander abzuarbeiten, wird *"Divide et impera!"* genannt. Das ist lateinisch, auf deutsch heisst es *"Teile und herrsche!"*.

QuickSort in dieser Form ist also ein **rekursives Verfahren**. Das Abbruchkriterium der Rekursion ist dann erfüllt, wenn ein Array der Grösse 1 sortiert werden soll. Da gibt es nämlich nichts zu tun.

Aufgabe (4.5)
Hinweis: Diese Aufgabe ist für SchülerInnen gedacht, welche die Zusatzaufgabe (3.6) (SelectionSort rekursiv) nicht gelöst haben. Wenn Sie (3.6) gelöst haben, lösen Sie bitte die Aufgabe (4.6) an Stelle von dieser.

Mit der Partitionierung haben Sie einen wesentlichen Teil von QuickSort selbst entwickelt. Den Rest bekommen Sie geschenkt. Er besteht aus zwei Funktionen.

Die erste Funktion implementiert den eigentlichen rekursiven Algorithmus. Diese Funktion hat drei Parameter: den Array a und die Indexgrenzen des Ausschnitts, der sortiert werden soll:

```
void quickSortRek(int[] a, int vonIndex, int bisIndex) {
    if (vonIndex >= bisIndex) { return; }       // Abruchbedingung
    int pivotIndex = (vonIndex + bisIndex) / 2; // Pivot-Wahl
    int pivotIndexDef = partitionieren(a, pivotIndex,
                                        vonIndex, bisIndex);
    quickSortRek(a, vonIndex, pivotIndexDef - 1);
    quickSortRek(a, pivotIndexDef + 1, bisIndex);
}
```

Die zweite Funktion kann von aussen ganz normal aufgerufen werden. Sie enthält nur eine einzige Anweisung, nämlich den Aufruf von quickSortRek() mit dem ganzen Array a:

```
void quickSort(int[] a) {
    quickSortRek(a, 0, a.Length - 1);
}
```

Bauen Sie den QuickSort-Algorithmus in ihr Sortierprogramm ein, indem Sie über einen neuen Button "QuickSort" die Funktion quickSort() aufrufen.

Aufgabe (4.6)
Hinweis: Diese Aufgabe ist für SchülerInnen gedacht, welche die Zusatzaufgabe (3.6) (SelectionSort rekursiv) gelöst haben. Wenn Sie (3.6) nicht gelöst haben, lösen Sie bitte die Aufgabe (4.5) an Stelle von dieser.

Implementieren Sie den QuickSort-Algorithmus. Wie schon beim rekursiven SelectionSort sollen zwei Funktionen erstellt werden:

Erstens die rekursive Funktion

```
void quickSortRek(int[] a, int vonIndex, int bisIndex)
```

Sie implementiert den eigentlichen rekursiven QuickSort-Algorithmus. Diese Funktion hat drei Parameter: Den Array a und und die zwei Indizes vonIndex und bisIndex, welche festlegen, welcher Bereich des Arrays sortiert werden soll. Wir erinnern uns, dass diese Funktion ja auch Teil-Arrays sortieren können muss. Diese Funktion soll die vorhin entwickelte Funktion zur Partitionierung aufrufen.

Zweitens die Funktion

```
void quickSort(int[] a)
```

Sie kann von aussen ganz normal aufgerufen werden. Sie enthält nur eine einzige Anweisung, nämlich den Aufruf von quickSortRek(), wobei der Array a sowie der tiefste und der höchste Index von a übergeben werden.

Bauen Sie den QuickSort-Algorithmus in ihr Sortierprogramm ein, indem Sie ihn über einen neuen Button "QuickSort" aufrufen.

4.3 Aufwandmessungen

Aufgabe (4.7)
Sortieren Sie Zufalls-Arrays verschiedener Grössen mit QuickSort und tragen Sie die Anzahl Vergleiche und Vertauschungen in eine Tabelle ein.

Anzahl Vergleiche					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
QuickSort					

Anzahl Vertauschungen					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
QuickSort					

Zusatzaufgabe (4.8)
Gegeben ist folgende Variante von QuickSort:

```
void quickSortRek2(int[] a, int vonIndex, int bisIndex) {
    if (vonIndex >= bisIndex) { return; }
    int ix = (vonIndex + bisIndex) / 2;
    int x = a[ix];
    int l = vonIndex;
    int r = bisIndex;
    do {
        while (istGroesser(x, a[l])) { l++; }
        while (istGroesser(a[r], x)) { r--; }
        if (l <= r) {
            vertauschen(a, l, r);
            l++;
            r--;
        }
    } while (l <= r);
    quickSortRek2(a, vonIndex, r);
    quickSortRek2(a, l, bisIndex);
}
```

Bauen Sie auch diese Variante in ihr Sortierprogramm ein (Button "QuickSort2") und vergleichen Sie die beiden Varianten bezüglich Anzahl Vergleiche und Vertauschungen. Können Sie die Unterschiede erklären?

Zusatzaufgabe (4.9)
Programmieren Sie eine Funktion zur Ermittlung des Medians eines Arrays.

4.4 Zusammenfassung

Hier nochmals eine Zusammenfassung der Funktionsweise von Quicksort: Wir wählen ein Element des Arrays und nennen dieses *Pivot-Element*. Anschliessend führen wir eine sogenannte *Partionierung* durch, indem wir den Array so umordnen, dass links vom Pivot nur kleinere Elemente, rechts von ihm nur grössere stehen. Im Idealfall, nämlich wenn wir als Pivot gerade den *Median* auswählen würden, würde auf diese Weise der Array gerade in zwei gleich grosse Hälften aufgeteilt. Die Bestimmung des Medians ist allerdings aufwändig. Glücklicherweise funktioniert das Verfahren auch dann, wenn wir ein *beliebiges* Element als Pivot wählen. Man beachte, dass die beiden Teil-Arrays links und rechts des Pivots nach dem Partitionierungsvorgang immer noch unsortiert sind. Die Sortierung des linken und des rechten Teil-Arrays erfolgt nach dem Prinzip *"Teile und herrsche!"*: Wir wenden das gleiche Verfahren auf die beiden Teil-Arrays an, von denen natürlich jeder kleiner ist als der ganze Array. Indem wir diesen Mechanismus *rekursiv* fortsetzen, bis wir nur noch Arrays der Grösse 1 zu sortieren haben, erreichen wir schliesslich unser Ziel, nämlich die Sortierung des ganzen Arrays.

Wir ahnen bereits, dass QuickSort bezüglich Laufzeit besser abschneidet als die beiden vorher behandelten Verfahren. Im nächsten Kapitel werden wir die drei Verfahren miteinander vergleichen und die Ergebnisse anschaulich darstellen.

5 Vergleich der Verfahren

Lernziel
Nach der Bearbeitung dieses Kapitels
- wissen Sie, welches der hier besprochenen Sortierverfahren das schnellste ist.

5.1 Daten sammeln und darstellen

Aufgabe (5.1)
Sie haben zu jedem Sortierverfahren eine Tabelle mit der Anzahl Vergleiche und Vertauschungen für verschieden grosse Arrays erstellt. Nun wollen wir die drei Verfahren miteinander vergleichen.

a) Führen Sie die Tabellen der Anzahl Vergleiche und Vertauschungen zusammen und erweitern Sie sie - soweit die Rechnerleistung es zulässt - um Spalten für die Arraygrössen 3000, 10000, 30000 und 100000:

Anzahl Vergleiche									
Sortierverfahren	*Arraygrösse*								
	10	*30*	*100*	*300*	*1000*	*3000*	*10000*	*30000*	*100000*
BubbleSort									
BubbleSort Opt									
SelectionSort									
QuickSort									

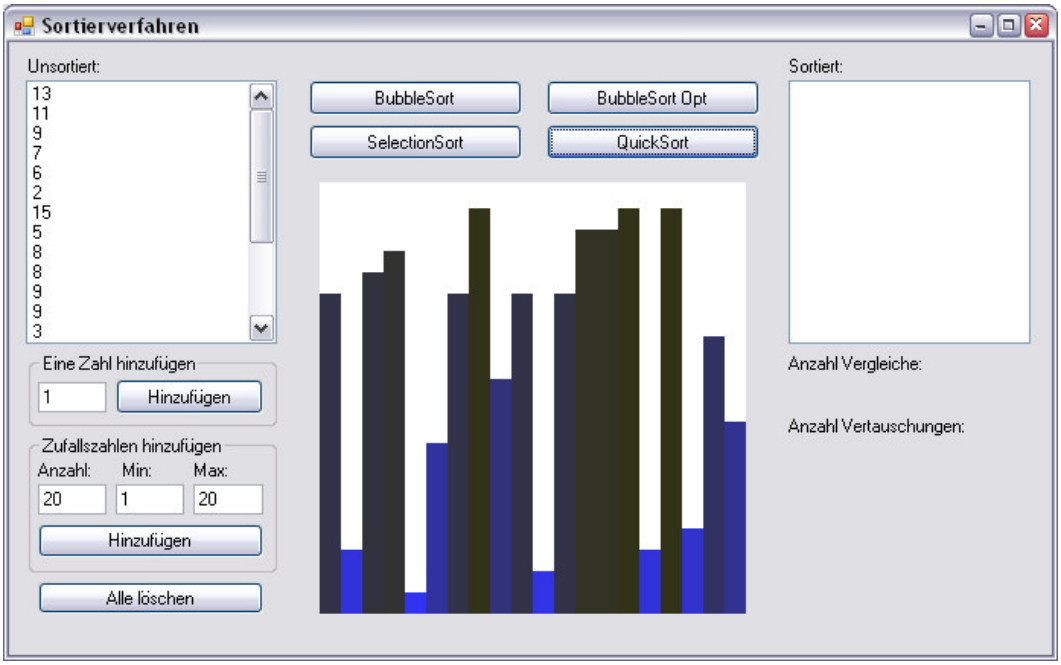
Anzahl Vertauschungen									
Sortierverfahren	*Arraygrösse*								
	10	*30*	*100*	*300*	*1000*	*3000*	*10000*	*30000*	*100000*
BubbleSort									
BubbleSort Opt									
SelectionSort									
QuickSort									

b) Erstellen Sie mit Hilfe eines Tabellenkalkulations-Programmes (z.B. Excel) Diagramme aus den beiden obigen Tabellen. Beachten Sie, dass die Rubriken-Achse durch die vorgegebene Wahl der Arraygrössen (10, 30, 100, 300,...) eine logarithmische Skala besitzt. Wählen Sie auch für die y-Achse eine logarithmische Skala. Kommentieren Sie das Ergebnis.

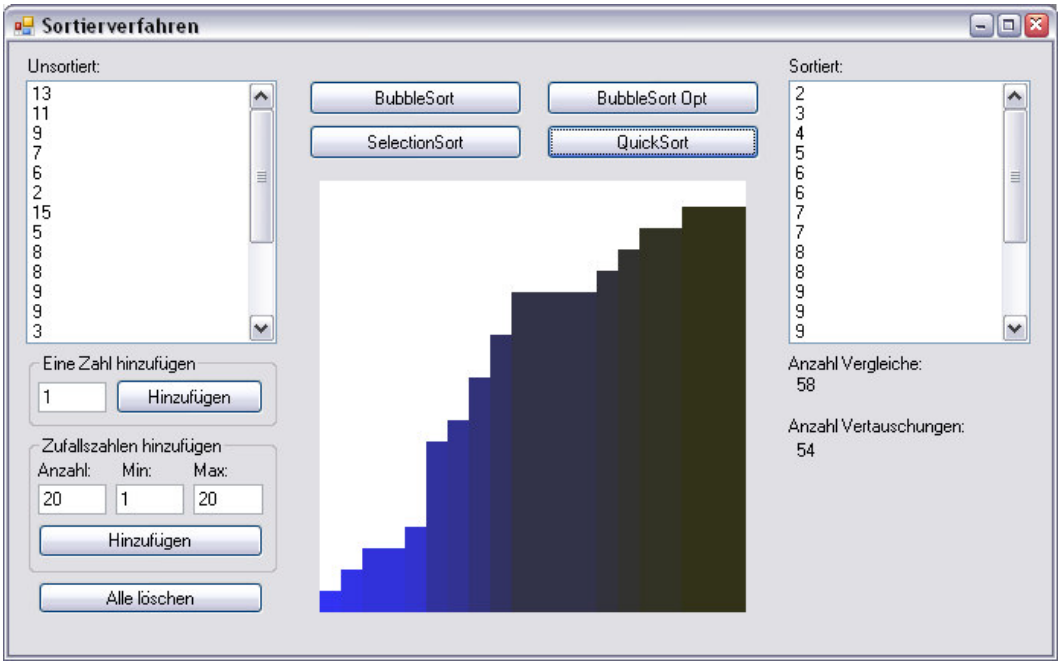
c) Unter der Annahme, dass eine Vertauschung ungefähr gleich viel Rechenzeit braucht wie ein Vergleich, macht die Summe dieser beiden Grössen eine Aussage über die Effizienz eines Verfahrens als Ganzes. Erstellen Sie ein Diagramm mit der Summe von Anzahl Vergleichen und Vertauschungen.

Zusatzaufgabe (5.2)
Erweitern Sie ihr Programm so, dass es eine grafische Animation des Sortiervorganges anzeigt. Die Werte sollen durch Balken entsprechender Länge dargestellt werden. Jeder Vertauschvorgang soll in der Grafik dargestellt werden.

Beispiel vor der Sortierung:



Nach der Sortierung:



Tipp: Für diese Erweiterung brauchen Sie die eigentlichen Sortierfunktionen gar nicht anzufassen. Vertauschungen werden ja bereits zwecks Zählung durch die Funktion vertauschen() erledigt. Die Animation können Sie als zusätzlichen Nebeneffekt in diese Funktion einbauen.

5.2 Zusammenfassung

Von den vielen Sortierverfahren, die es gibt, haben wir drei ausführlich besprochen, implementiert und miteinander verglichen. Diese Verfahren funktionieren alle korrekt, unterscheiden sich aber in ihrem Laufzeit-Aufwand um mehrere Grössenordnungen. Während bei kleinen Arrays die Unterschiede kaum spürbar sein dürften, zeigen sich die Differenzen bei grossen Arrays um so mehr. Für den praktischen Einsatz ist es äusserst wichtig, einen schnellen Algorithmus zu wählen. Von *BubbleSort* ist also dringend abzuraten. *SelectionSort* ist zwar wesentlich schneller als BubbleSort, für grosse Arrays aber trotzdem nicht zu empfehlen. Mit *QuickSort* haben wir eines der schnellsten Sortierverfahren kennen gelernt, die es überhaupt gibt.

6 Lösungsvorschläge zu den Aufgaben

6.1 Lösungen zu Aufgaben aus Kapitel 1

Aufgabe (1.1)

Das Inhaltsverzeichnis eines Sachbuches ist nicht sortiert, sondern (in der Regel hierarchisch) nach Themen gegliedert. Der Index hingegen ist alphabetisch sortiert. Dadurch ermöglicht er das rasche Auffinden von Begriffen unabhängig von ihrer thematischen Zugehörigkeit.

Aufgabe (1.2)

Aus dem Telefonbuch von Frauenfeld:
Vock
Vogel
Vögeli
Völki
Vollenweider
Vontobel
von Salis

Aufgabe (1.3)

a) Bei zwei Farben ist nicht ohne weiteres klar, welches die "kleinere" ist. Man könnte versuchen, als Kriterium die Helligkeit zu nehmen. Das könnte aber z.B. zu folgender Ordnung führen: blond, hellbraun, rot, kastanienbraun, dunkelrot, schwarz. Dies wäre wohl unbefriedigend, weil die beiden Rothaarigen nicht nebeneinander stehen.
b) Es ist nicht klar, welches das "kleinere" Geschlecht ist. Ausserdem ist die Sortierung nicht eindeutig, da es nur zwei verschiedene Geschlechter gibt.
c) Dieses Kriterium ist für die Sortierung gänzlich ungeeignet, weil erstens zwei Hobbies nicht vergleichbar sind und weil zweitens ein(e) Schüler(in) mehrere Hobbies haben kann.

Aufgabe (1.4)

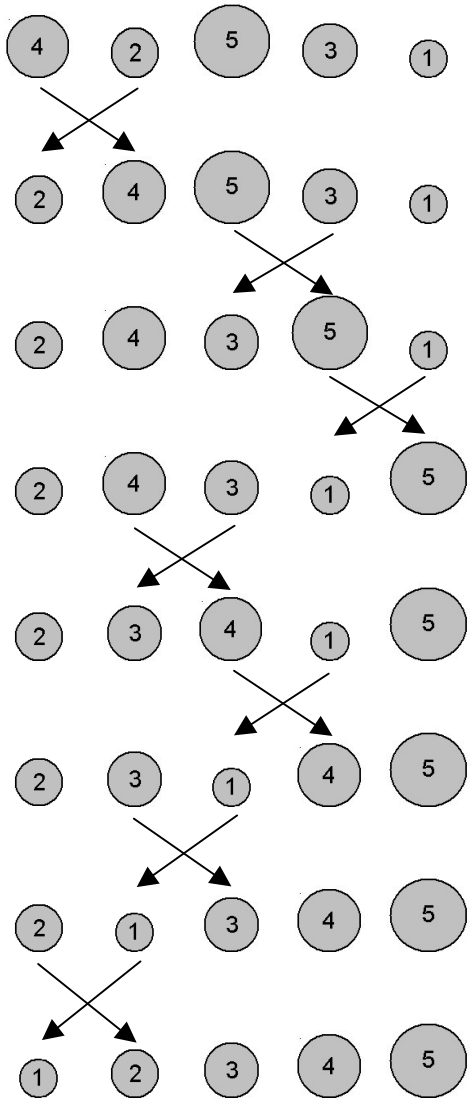
```
bool istSortiert(int[] a) {
    for (int i = 0; i < a.Length - 1; i++) {
        if (a[i] > a[i + 1]) { return false; }
    }
    return true;
}
```

Aufgabe (1.5)

Weitere Beispiele von Algorithmen sind:
- Wegbeschreibung einer Routenplaner-Software
- Installationsanleitung für ein Gerät, z.B. einen Drucker

6.2 Lösungen zu Aufgaben aus Kapitel 2

Aufgabe (2.1)

Beginn erster Durchgang	
Beginn zweiter Durchgang	
Beginn dritter Durchgang	
Beginn vierter Durchgang	
Fertig.	

Aufgabe (2.2)

Schritt	Was tun	Ergebnis dieses Schrittes					
	Ausgangslage	11	3	6	7	3	6
(1)	Wähle erstes Paar	11	3	6	7	3	6
(2)	Vertauschen	3	11	6	7	3	6
(3)	Wähle nächstes Paar	3	11	6	7	3	6
(2)	Vertauschen	3	6	11	7	3	6
(3)	Wähle nächstes Paar	3	6	11	7	3	6
(2)	Vertauschen	3	6	7	11	3	6
(3)	Wähle nächstes Paar	3	6	7	11	3	6
(2)	Vertauschen	3	6	7	3	11	6
(3)	Wähle nächstes Paar	3	6	7	3	11	6
(2)	Vertauschen	3	6	7	3	6	11
(3)	Es hat weiter rechts kein Paar mehr.	3	6	7	3	6	11
(4)	In diesem Durchgang wurden 5 Vertauschungen gemacht, also noch ein Durchgang.	3	6	7	3	6	11
(1)	Wähle erstes Paar	3	6	7	3	6	11
(2)	Nichts tun	3	6	7	3	6	11
(3)	Wähle nächstes Paar	3	6	7	3	6	11
(2)	Nichts tun	3	6	7	3	6	11
(3)	Wähle nächstes Paar	3	6	7	3	6	11
(2)	Vertauschen	3	6	3	7	6	11
(3)	Wähle nächstes Paar	3	6	3	7	6	11
(2)	Vertauschen	3	6	3	6	7	11
(3)	Wähle nächstes Paar	3	6	3	6	7	11
(2)	Nichts tun	3	6	3	6	7	11
(3)	Es hat weiter rechts kein Paar mehr.	3	6	3	6	7	11
(4)	In diesem Durchgang wurden 2 Vertauschungen gemacht, also noch ein Durchgang.	3	6	3	6	7	11
(1)	Wähle erstes Paar	3	6	3	6	7	11
(2)	Nichts tun	3	6	3	6	7	11
(3)	Wähle nächstes Paar	3	6	3	6	7	11
(2)	Vertauschen	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11

(2)	Nichts tun (da 6 nicht grösser als 6 ist!)	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Es hat weiter rechts kein Paar mehr.	3	3	6	6	7	11
(4)	In diesem Durchgang wurde 1 Vertauschung gemacht, also noch ein Durchgang.	3	3	6	6	7	11
(1)	Wähle erstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Es hat weiter rechts kein Paar mehr.	3	3	6	6	7	11
(4)	In diesem Durchgang wurden keine Vertauschungen gemacht, also sind wir fertig.	3	3	6	6	7	11
	Fertig	3	3	6	6	7	11

Aufgabe (2.3)

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Sortierverfahren {

/// <summary>
/// Sortierverfahren
/// mb, 09.08.2007
/// </summary>
public partial class Form1 : Form {

public Form1() {
    InitializeComponent();
}

void eineZahlHinzufuegenButton_Click(object sender, EventArgs e) {
    try {
        int zahl = int.Parse(elementTextBox.Text);
        unsortiertListBox.Items.Add(zahl.ToString());
    } catch {
    }
}

void zufallsZahlenHinzufuegenButton_Click(object sender, EventArgs e) {
    try {
        int n = int.Parse(anzahlElementeTextBox.Text);
        int min = int.Parse(minTextBox.Text);
        int max = int.Parse(maxTextBox.Text);
        Random r = new Random();
        for (int i = 0; i < n; i++) {
            unsortiertListBox.Items.Add(r.Next(min, max).ToString());
        }
    } catch {
    }
}

void alleLoeschenButton_Click(object sender, EventArgs e) {
    unsortiertListBox.Items.Clear();
}

}  // end class
}  // end namespace
```

Aufgabe (2.4)

Es werden nur die zusätzlichen Funktionen abgedruckt:

```
void bubbleSort(int[] a) {
    int n = a.Length;
    bool etwasVertauscht;
    do {
        etwasVertauscht = false;
        for (int k = 0; k < n - 1; k++) {
            if (a[k] > a[k + 1]) {
                int temp = a[k]; a[k] = a[k + 1]; a[k + 1] = temp;
                etwasVertauscht = true;
            }
        }
    } while (etwasVertauscht);
}

int[] arrayAusListBoxErzeugen() {
    int n = unsortiertListBox.Items.Count;
    int[] a = new int[n];
    for (int i = 0; i < n; i++) {
        a[i] = int.Parse(unsortiertListBox.Items[i].ToString());
    }
    return a;
}

void arrayAusgeben(int[] a) {
    sortiertListBox.Items.Clear();
    for (int i = 0; i < a.Length; i++) {
        sortiertListBox.Items.Add(a[i].ToString());
    }
}

void bubbleSortButton_Click(object sender, EventArgs e) {
    int[] a = arrayAusListBoxErzeugen();
    bubbleSort(a);
    arrayAusgeben(a);
}
```

Aufgabe (2.5)

a) Diese Änderung führt zu einer absteigenden Sortierung.

b) Diese Änderung hat keinen Einfluss, solange der Array nicht zwei gleiche Elemente enthält. Wenn es aber zwei (oder mehr) gleiche Elemente gibt, geschieht folgendes: Die beiden gleichen Elemente werden irgendwann nebeneinander stehen. Von da an werden Sie in jedem Durchgang miteinander vertauscht. Das Verfahren hört also nie auf und ist damit gar kein richtiger Algorithmus (vergleiche Definition des Begriffs Algorithmus im Kapitel 1).

c) wie b), aber absteigende Sortierung

Aufgabe (2.6)

n-1 Durchgänge sind unter Umständen mehr als nötig. Im Extremfall eines Arrays, der in der Ausgangslage schon sortiert ist, würde die ursprüngliche Variante mit einem einzigen Durchgang auskommen und wäre damit viel schneller als die "optimierte" Version.

Aufgabe (2.7)

Schritt	Was tun	Ergebnis dieses Schrittes					
	Ausgangslage	11	3	6	7	3	6
(1)	Wähle erstes Paar	11	3	6	7	3	6
(2)	Vertauschen	3	11	6	7	3	6
(3)	Wähle nächstes Paar	3	11	6	7	3	6
(2)	Vertauschen	3	6	11	7	3	6
(3)	Wähle nächstes Paar	3	6	11	7	3	6
(2)	Vertauschen	3	6	7	11	3	6
(3)	Wähle nächstes Paar	3	6	7	11	3	6
(2)	Vertauschen	3	6	7	3	11	6
(3)	Wähle nächstes Paar	3	6	7	3	11	6
(2)	Vertauschen	3	6	7	3	6	11
(3)	Es hat weiter rechts kein Paar mehr.	3	6	7	3	6	11
(4)	In diesem Durchgang wurden 5 Vertauschungen gemacht, also noch ein Durchgang.	3	6	7	3	6	11
(1)	Wähle erstes Paar	3	6	7	3	6	11
(2)	Nichts tun	3	6	7	3	6	11
(3)	Wähle nächstes Paar	3	6	7	3	6	11
(2)	Nichts tun	3	6	7	3	6	11
(3)	Wähle nächstes Paar	3	6	7	3	6	11
(2)	Vertauschen	3	6	3	7	6	11
(3)	Wähle nächstes Paar	3	6	3	7	6	11
(2)	Vertauschen	3	6	3	6	7	11
(3)	Wähle nächstes Paar	3	6	3	6	7	11
(2)	Nichts tun	3	6	3	6	7	11
(3)	Es hat weiter rechts kein Paar mehr.	3	6	3	6	7	11
(4)	In diesem Durchgang wurden 2 Vertauschungen gemacht, also noch ein Durchgang.	3	6	3	6	7	11
(1)	Wähle erstes Paar	3	6	3	6	7	11

(2)	Nichts tun	3	6	3	6	7	11
(3)	Wähle nächstes Paar	3	6	3	6	7	11
(2)	Vertauschen	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun (da 6 nicht grösser als 6 ist!)	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Es hat weiter rechts kein Paar mehr.	3	3	6	6	7	11
(4)	In diesem Durchgang wurde 1 Vertauschung gemacht, also noch ein Durchgang.	3	3	6	6	7	11
(1)	Wähle erstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Wähle nächstes Paar	3	3	6	6	7	11
(2)	Nichts tun	3	3	6	6	7	11
(3)	Es hat weiter rechts kein Paar mehr.	3	3	6	6	7	11
(4)	In diesem Durchgang wurden keine Vertauschungen gemacht, also sind wir fertig.	3	3	6	6	7	11
	Fertig	3	3	6	6	7	11

Aufgabe (2.8)

Es werden nur die zusätzlichen Funktionen abgedruckt:

```
void bubbleSortOpt(int[] a) {
    int n = a.Length;
    for (int i = 0; i < n - 1; i++) {
        for (int k = 0; k < n - 1 - i; k++) {
            if (a[k] > a[k + 1]) {
                int temp = a[k]; a[k] = a[k + 1]; a[k + 1] = temp;
            }
        }
    }
}

void bubbleSortOptButton_Click(object sender, EventArgs e) {
    int[] a = arrayAusListBoxErzeugen();
    bubbleSortOpt(a);
    arrayAusgeben(a);
}
```

Aufgabe (2.9)

Für eine absteigende Sortierung muss nur der Vergleich zweier Elemente umgekehrt werden, also

```
            if (a[k] < a[k + 1])
```

statt

```
            if (a[k] > a[k + 1])
```

Aufgabe (2.10)

Formular:

Code (nur neue und geänderte Teile):

```
int anzahlVergleiche = 0;
int anzahlVertauschungen = 0;

bool istGroesser(int x, int y) {
    anzahlVergleiche++;
    return (x > y);
}

void vertauschen(int[] a, int i, int j) {
    anzahlVertauschungen++;
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

void bubbleSort(int[] a) {
    int n = a.Length;
    bool etwasVertauscht;
    do {
        etwasVertauscht = false;
        for (int k = 0; k < n - 1; k++) {
            if (istGroesser(a[k], a[k + 1])) {
                vertauschen(a, k, k + 1);
                etwasVertauscht = true;
            }
        }
    } while (etwasVertauscht);
}

void bubbleSortOpt(int[] a) {
    int n = a.Length;
    for (int i = 0; i < n - 1; i++) {
        for (int k = 0; k < n - 1 - i; k++) {
            if (istGroesser(a[k],a[k + 1])) {
                vertauschen(a, k, k + 1);
            }
        }
    }
}

void zaehlerZuruecksetzen() {
    anzahlVergleiche = 0;
    anzahlVertauschungen = 0;
}

void zaehlerAusgeben() {
    anzahlVergleicheLabel.Text = anzahlVergleiche.ToString();
    anzahlVertauschungenLabel.Text = anzahlVertauschungen.ToString();
}

void bubbleSortButton_Click(object sender, EventArgs e) {
    int[] a = arrayAusListBoxErzeugen();
    zaehlerZuruecksetzen();
    bubbleSort(a);
    arrayAusgeben(a);
    zaehlerAusgeben();
}
```

```
void bubbleSortOptButton_Click(object sender, EventArgs e) {
    int[] a = arrayAusListBoxErzeugen();
    zaehlerZuruecksetzen();
    bubbleSortOpt(a);
    arrayAusgeben(a);
    zaehlerAusgeben();
}
```

Tabelle: Da der unsortierte Array mit Zufallszahlen gefüllt wurde, können ihre Zahlen von diesen abweichen.

Anzahl Vergleiche					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
BubbleSort	81	725	9306	86710	948051
BubbleSort Opt	45	435	4950	44850	499500

Anzahl Vertauschungen					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
BubbleSort	31	271	2553	22732	246784
BubbleSort Opt	31	271	2553	22732	246784

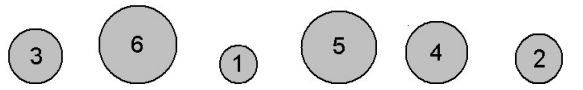
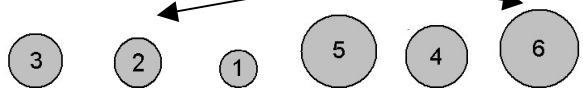
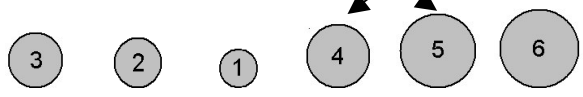
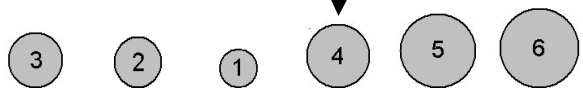
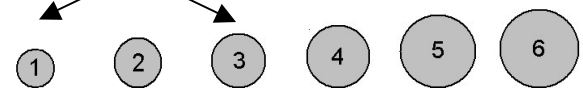
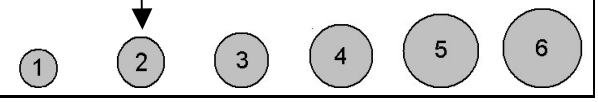
Was auffällt: BubbleSortOpt braucht für den gleichen Array genau gleich viele Vertauschungen wie BubbleSort, aber weniger Vergleiche. Das ist natürlich genau der gewünschte Effekt der Optimierung.

Zusatzaufgabe (2.11)

```
void randomSort(int[] a) {
    Random r = new Random();
    // Aufruf der früher programmierten Funktion istSortiert():
    while (!istSortiert(a)) {
        int i; int k;
        // Suche Paar mit falscher Reihenfolge,
        // d.h. i,k mit i<k und a[i]>a[k]
        do {
            i = r.Next(a.Length);
            k = r.Next(a.Length);
            if (i > k) {
                int temp = i; i = k; k = temp;
            }
        } while (!(istGroesser(a[i], a[k])));
        vertauschen(a, i, k);
    }
}
```

6.3 Lösungen zu Aufgaben aus Kapitel 3

Aufgabe (3.1)

Vertausche grösste Kugel mit der Kugel ganz rechts	
Vertausche zweitgrösste Kugel mit der zweiten Kugel von rechts	
Vertausche drittgrösste Kugel mit der dritten Kugel von rechts (nichts tun)	
Vertausche viertgrösste Kugel mit der vierten Kugel von rechts	
Vertausche fünftgrösste (zweitkleinste) Kugel mit der fünften Kugel von rechts (nichts tun)	
Fertig (kleinste Kugel ist sicher ganz links)	

Aufgabe (3.2)

Schritt	Was tun	i	Ergebnis dieses Schrittes					
	Ausgangslage		9	3	8	8	2	6
(1)	Setze i auf 1	1	9	3	8	8	2	6
(2)	Suche das grösste Element	1	9	3	8	8	2	6
(3)	Vertausche es mit dem Element ganz rechts	1	6	3	8	8	2	9
(4)	Erhöhe i um 1	2	6	3	8	8	2	9
(2)	Suche das grösste der ersten fünf Elemente	2	6	3	8	8	2	9
(3)	Vertausche es mit dem zweiten Element von rechts	2	6	3	2	8	8	9
(4)	Erhöhe i um 1	3	6	3	2	8	8	9
(2)	Suche das grösste der ersten vier Elemente	3	6	3	2	8	8	9
(3)	Vertausche es mit dem dritten Element von rechts (nichts tun)	3	6	3	2	8	8	9
(4)	Erhöhe i um 1	4	6	3	2	8	8	9
(2)	Suche das grösste der ersten drei Elemente	4	6	3	2	8	8	9
(3)	Vertausche es mit dem vierten Element von rechts	4	2	3	6	8	8	9
(4)	Erhöhe i um 1	5	2	3	6	8	8	9
(2)	Suche das grösste der ersten zwei Elemente	5	2	3	6	8	8	9
(3)	Vertausche es mit dem fünften Element von rechts (nichts tun)	5	2	3	6	8	8	9
(4)	Erhöhe i um 1. i ist nicht kleiner als die Arraylänge, also sind wir fertig.	6	2	3	6	8	8	9
	Fertig		2	3	6	8	8	9

Aufgabe (3.3)

Nur Sortierfunktion (zwei Varianten):

```
void selectionSort(int[] a) {
    int n = a.Length;
    for (int i = 1; i < n; i++) {
        int maxIndex = 0; // Index des bisher grössten Elements
        for (int k = 1; k <= n - i; k++) {
            if (istGroesser(a[k], a[maxIndex])) {
                maxIndex = k;
            }
        }
        vertauschen(a, maxIndex, n - i);
    }
}

// Variante: wie oben, aber mit j := n-i
void selectionSort2(int[] a) {
    int n = a.Length;
    for (int j = n - 1; j > 0; j--) {
        int maxIndex = 0; // Index des bisher grössten Elements
        for (int k = 1; k <= j; k++) {
            if (istGroesser(a[k], a[maxIndex])) {
                maxIndex = k;
            }
        }
        vertauschen(a, maxIndex, j);
    }
}
```

Zusatzaufgabe (3.4)

a) das grösste genau einmal, das kleinste höchstens (n-1) mal, nämlich dann, wenn es am Anfang ganz rechts steht.

b) Es gibt (n-1) Vertauschungen. An jeder Vertauschung sind zwei Elemente beteiligt. Also ist jedes Element im Durchschnitt 2(n-1)/n mal, also knapp zweimal an einer Vertauschung beteiligt.

Aufgabe (3.5)

Da SelectionSort unabhängig von der Ausgangslage immer genau gleich abläuft, sollten ihre Zahlen exakt mit diesen übereinstimmen. Das gilt aber natürlich nur, wenn Vertauschungen "stur" ausgeführt werden, d.h. wenn auch eine (unnötige) Vertauschung eines Elements mit sich selber gezählt wird.

Anzahl Vergleiche					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
SelectionSort	45	435	4950	44850	499500

Anzahl Vertauschungen					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
SelectionSort	9	29	99	299	999

Zusatzaufgabe (3.6)

```
void selectionSortRek(int[] a, int n) {
    if (n == 1) { return; } // Abbruchbedingung
    int maxIndex = 0;       // Index des bisher grössten Elements
    for (int k = 1; k < n; k++) {
        if (istGroesser(a[k], a[maxIndex])) {
            maxIndex = k;
        }
    }
    vertauschen(a, maxIndex, n - 1);
    selectionSortRek(a, n - 1);
}

void selectionSort3(int[] a) {
    selectionSortRek(a, a.Length);
}
```

Zusatzaufgabe (3.7)

ohne Lösung

6.4 Lösungen zu Aufgaben aus Kapitel 4

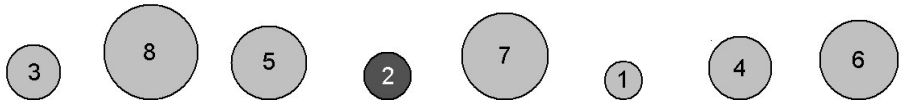
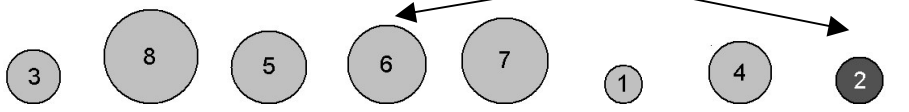
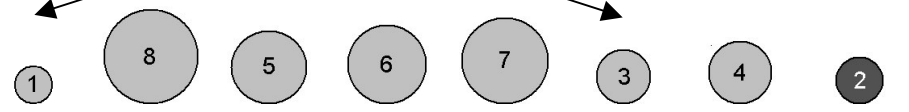
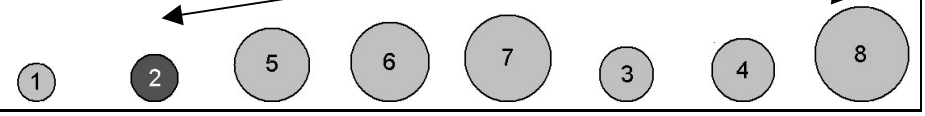
Aufgabe (4.1)

a)
Reihe sortiert: 2 2 3 4 5 6 7
Median = 4

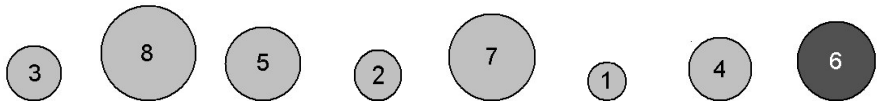
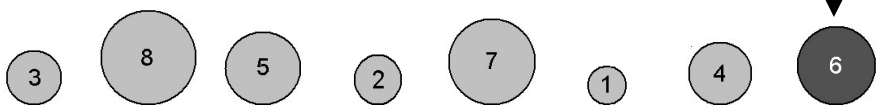
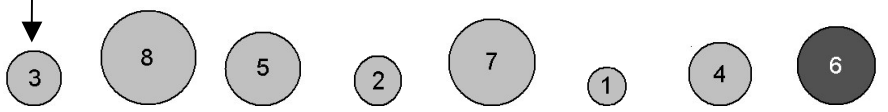
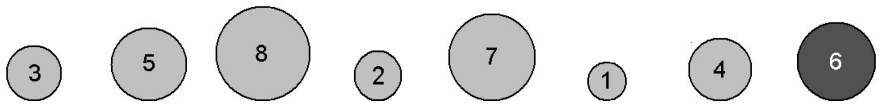
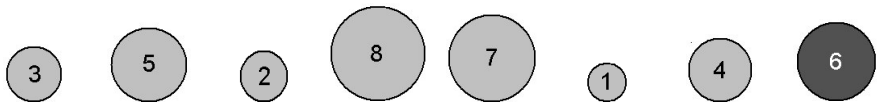
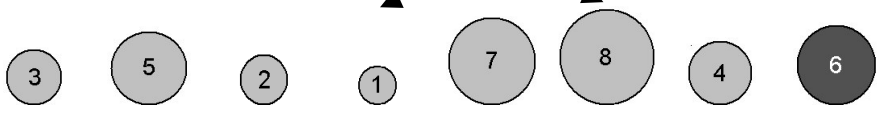
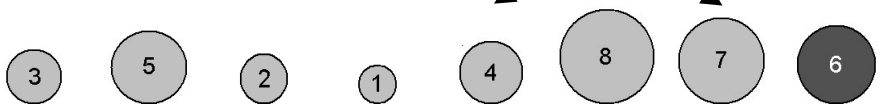
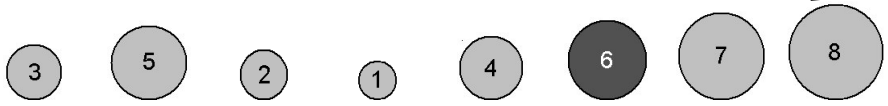
b)
Reihe sortiert: 1 3 3 4 5 6 6 7 11 24
Median = (5+6)/2 = 5.5
Bei gerader Anzahl Elemente ist der Median der Durchschnitt der beiden mittleren Werte.

Aufgabe (4.2)

Erste Pivot-Wahl: 2

Schaffe Pivot aus dem Weg	
Bringe erstes Element < Pivot auf Platz 1	
Es gibt keine weiteren Elemente < Pivot; bringe Pivot auf Platz 2	
Die Reihe ist jetzt partitioniert.	

Zweite Pivot-Wahl: 6

Schaffe Pivot aus dem Weg (nichts tun)	
Bringe erstes Element < Pivot auf Platz 1 (nichts tun)	
Bringe zweites Element < Pivot auf Platz 2	
Bringe drittes Element < Pivot auf Platz 3	
Bringe viertes Element < Pivot auf Platz 4	
Bringe fünftes Element < Pivot auf Platz 5	
Es gibt keine weiteren Elemente < Pivot; bringe Pivot auf Platz 6	
Die Reihe ist jetzt partitioniert.	

Aufgabe (4.3)

a)

Schritt	Was tun	k	Ergebnis dieses Schrittes							
	Ausgangslage		5	**7**	11	3	8	6	9	2
(2.1)	Bringe Pivot an letzte Stelle		5	2	11	3	8	6	9	**7**
(2.2)	Setze Einfügeposition k auf 1	1	5	2	11	3	8	6	9	**7**
(2.3)	Wähle erstes Element	1	5	2	11	3	8	6	9	**7**
(2.4)	kleiner als Pivot, also vertauschen mit Pos. 1 (nichts tun) und dann k um 1 erhöhen	2	5	2	11	3	8	6	9	**7**
(2.5)	Wähle nächstes Element	2	5	2	11	3	8	6	9	**7**
(2.4)	kleiner als Pivot, also vertauschen mit Pos. 2 (nichts tun) und dann k um 1 erhöhen	3	5	2	11	3	8	6	9	**7**
(2.5)	Wähle nächstes Element	3	2	5	11	3	8	6	9	**7**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	3	2	5	11	3	8	6	9	**7**
(2.5)	Wähle nächstes Element	3	2	5	11	3	8	6	9	**7**
(2.4)	kleiner als Pivot, also vertauschen mit Pos. 3 und dann k um 1 erhöhen	4	2	5	3	11	8	6	9	**7**
(2.5)	Wähle nächstes Element	4	2	5	3	11	8	6	9	**7**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	4	2	5	3	11	8	6	9	**7**
(2.5)	Wähle nächstes Element	4	2	5	3	11	8	6	9	**7**
(2.4)	kleiner als Pivot, also vertauschen mit Pos. 4 und dann k um 1 erhöhen	5	2	5	3	6	8	11	9	**7**
(2.5)	Wähle nächstes Element	5	2	5	3	6	8	11	9	**7**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	5	2	5	3	6	8	11	9	**7**
(2.5)	Es gibt kein weiteres Element mehr	5	2	5	3	6	8	11	9	**7**
(2.6)	Bringe Pivot an Position 5	5	2	5	3	6	**7**	11	9	8
	Fertig		2	5	3	6	**7**	11	9	8

b)

Schritt	Was tun	k	Ergebnis dieses Schrittes							
	Ausgangslage		5	8	4	6	**3**	8	3	1
(2.1)	Bringe Pivot an letzte Stelle		5	8	4	6	1	8	3	**3**
(2.2)	Setze Einfügeposition k auf 1	1	5	8	4	6	1	8	3	**3**
(2.3)	Wähle erstes Element	1	5	8	4	6	1	8	3	**3**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	1	5	8	4	6	1	8	3	**3**
(2.5)	Wähle nächstes Element	1	5	8	4	6	1	8	3	**3**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	1	5	8	4	6	1	8	3	**3**
(2.5)	Wähle nächstes Element	1	5	8	4	6	1	8	3	**3**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	1	5	8	4	6	1	8	3	**3**
(2.5)	Wähle nächstes Element	1	5	8	4	6	1	8	3	**3**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	1	5	8	4	6	1	8	3	**3**
(2.5)	Wähle nächstes Element	1	5	8	4	6	1	8	3	**3**
(2.4)	kleiner als Pivot, also vertauschen mit Pos. 1 und dann k um 1 erhöhen	2	1	8	4	6	5	8	3	**3**
(2.5)	Wähle nächstes Element	2	1	8	4	6	5	8	3	**3**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	2	1	8	4	6	5	8	3	**3**
(2.5)	Wähle nächstes Element	2	1	8	4	6	5	8	3	**3**
(2.4)	Gewähltes Element ist nicht kleiner als Pivot, also nichts tun	2	1	8	4	6	5	8	3	**3**
(2.5)	Es gibt kein weiteres Element mehr	2	1	8	4	6	5	8	3	**3**
(2.6)	Bringe Pivot an Position 2	2	1	**3**	4	6	5	8	3	8
	Fertig		1	**3**	4	6	5	8	3	8

Man beachte, dass der rechte Teil durchaus Elemente enthalten kann, die gleich dem Pivot sind (hier die 3). Wichtig ist, dass keine *kleineren* Elemente rechts des Pivot stehen und keine *grösseren* links von ihm.

Aufgabe (4.4)

a)

```
int partitionierenGanz(int[] a, int pivotIndex) {
    int n = a.Length;
    int pivot = a[pivotIndex];
    // Pivot ganz nach rechts setzen:
    vertauschen(a, pivotIndex, n - 1);
    int k = 0;
    for (int i = 0; i < n - 1; i++) {
        if (istGroesser(pivot, a[i])) {
            vertauschen(a, k, i);
            k++;
        }
    }
    // Pivot an seine endgültige Position setzen:
    vertauschen(a, n - 1, k);
    return k;
}
```

b)

```
int partitionieren(int[] a, int pivotIndex, int vonIndex, int bisIndex) {
    int pivot = a[pivotIndex];
    // Pivot ganz nach rechts setzen:
    vertauschen(a, pivotIndex, bisIndex);
    int k = vonIndex;
    for (int i = vonIndex; i < bisIndex; i++) {
        if (istGroesser(pivot, a[i])) {
            vertauschen(a, k, i);
            k++;
        }
    }
    // Pivot an seine endgültige Position setzen:
    vertauschen(a, bisIndex, k);
    return k;
}
```

Aufgabe (4.5)

Diese Aufgabe besteht nur aus dem Zusammensetzen vorgegebener und früher entwickelter Programmteile. Aus diesem Grund wird hier auf die Angabe einer Lösung verzichtet.

Aufgabe (4.6)

Siehe Aufgabenstellung Aufgabe (4.5)

Aufgabe (4.7)

Ihre Zahlen können von diesen hier abweichen, da ja die Arrays mit Zufallszahlen gefüllt werden und QuickSort sich je nach Ausgangslage unterschiedlich verhält.

Anzahl Vergleiche					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
QuickSort	27	128	636	2626	10343

Anzahl Vertauschungen					
Sortierverfahren	*Arraygrösse*				
	10	*30*	*100*	*300*	*1000*
QuickSort	20	90	471	1695	6116

Zusatzaufgabe (4.8)

In der zweiten Variante geschieht der Partionierungsvorgang von beiden Seiten her gleichzeitig (über die Indizes r und l). Dadurch kann die Anzahl Vertauschungen reduziert werden.

Zusatzaufgabe (4.9)

```
double median(int[] a) {
    // Array mit beliebigem Verfahren sortieren:
    quickSort(a);
    int n = a.Length;
    return n % 2 == 0 ? (a[n/2] + a[n/2 - 1]) / 2.0 : a[(n-1) / 2];
}
```

6.5 Lösungen zu Aufgaben aus Kapitel 5

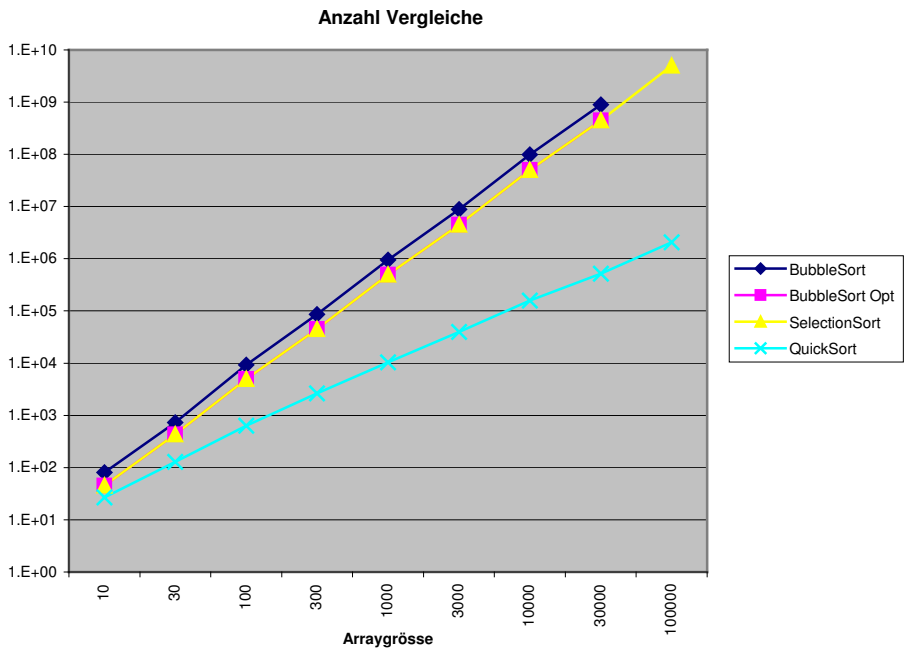
Aufgabe (5.1)

a)

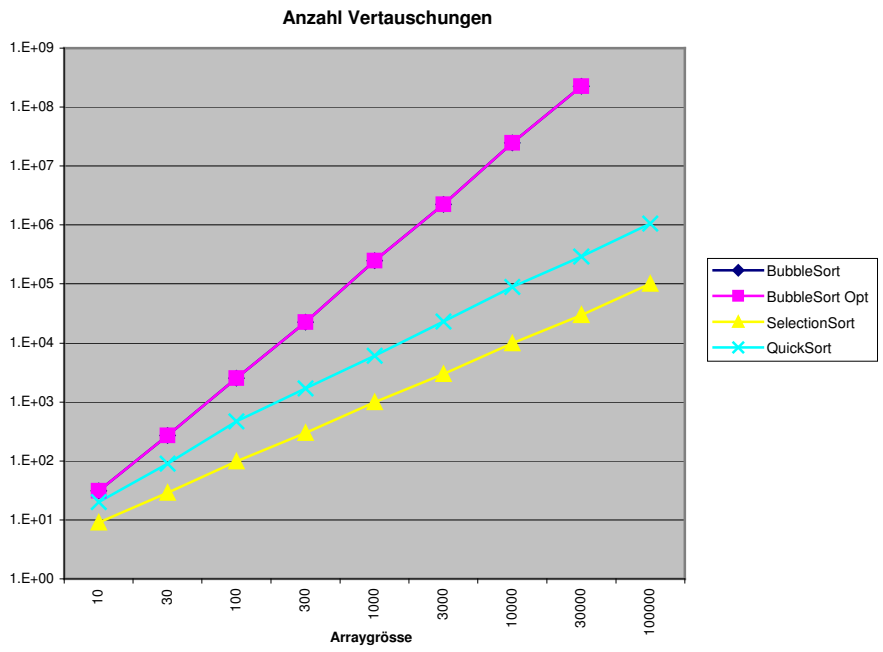
Anzahl Vergleiche									
Sortierverfahren	*Arraygrösse*								
	10	*30*	*100*	*300*	*1000*	*3000*	*10000*	*30000*	*100000*
BubbleSort	81	725	9306	86710	948051	8823058	98350164	8.97E+08	
BubbleSort Opt	45	435	4950	44850	499500	4498500	49995000	4.5E+08	
SelectionSort	45	435	4950	44850	499500	4498500	49995000	4.5E+08	5E+09
QuickSort	27	128	636	2626	10343	39474	157114	519625	2060849

Anzahl Vertauschungen									
Sortierverfahren	*Arraygrösse*								
	10	*30*	*100*	*300*	*1000*	*3000*	*10000*	*30000*	*100000*
BubbleSort	31	271	2553	22732	246784	2247527	24826831	2.25E+08	
BubbleSort Opt	31	271	2553	22732	246784	2247527	24826831	2.25E+08	
SelectionSort	9	29	99	299	999	2999	9999	29999	99999
QuickSort	20	90	471	1695	6116	23205	88831	289926	1054436

b)

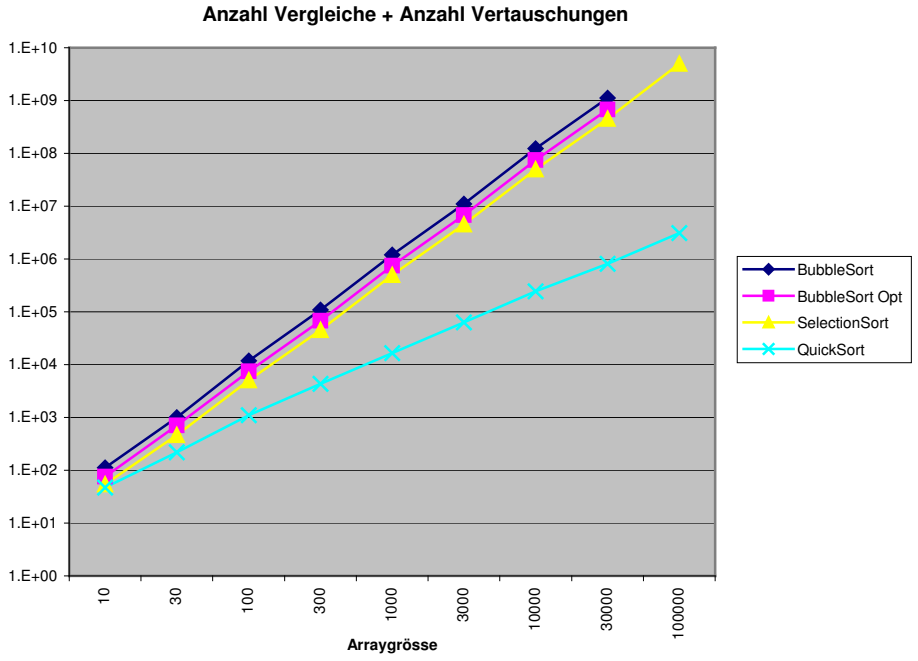


Bei den Vergleichen sind SelectionSort und der optimierte BubbleSort identisch, der ursprüngliche BubbleSort ist leicht schlechter. QuickSort ist schon für kleine Arrays am besten und wächst viel weniger schnell als alle anderen.



SelectionSort ist bei den Vertauschungen unschlagbar, QuickSort ist etwa eine Grössenordnung schlechter, BubbleSort weit abgeschlagen.

c)



...and the winner is: **QuickSort!**

Zusatzaufgabe (5.2)

ohne Lösung