Aufgabe 2

Bearbeitungszeit: 2 Wochen (bis Freitag, 4. Mai 2018)

Mathematischer Hintergrund: Sortieralgorithmen, Rekursion

Elemente von C++: Schleifen, Funktionen, Felder, Zeiger, dynamische Speicherverwaltung

---

Bei der Behandlung vieler mathematischer Probleme ist es notwendig, eine Liste von Elementen zu sortieren. Daher beschäftigt sich diese Aufgabe mit Algorithmen zum Sortieren von Listen.

Um zum Sortieren eine Reihenfolge der Elemente einer Menge M festzulegen, benötigt man eine strenge schwache Ordnung auf M: Eine Relation $R \subset M \times M$ heißt strenge schwache Ordnung auf der Menge M, falls sie folgende Axiome erfüllt: Transitivität (Gilt für beliebige $a, b, c \in M$ sowohl $(a, b) \in R$ als auch $(b, c) \in R$, so gilt auch $(a, c) \in R$.), Irreflexivität (Für beliebiges $a \in M$ gilt $(a, a) \notin R$.) und negative Transitivität (Gilt für beliebige $a, b, c \in M$ sowohl $(a, b) \notin R$ als auch $(b, c) \notin R$, so gilt auch $(a, c) \notin R$.). Gilt $(a, b) \in R$ für $a, b \in M$, so sagen wir im Folgenden auch a ist kleiner als b bezüglich der Ordnung R.

Eine strenge schwache Ordnung erlaubt mehrere Elemente gleichen Ranges, das heißt, es gilt für $a, b \in M$ weder $(a, b) \in R$ noch $(b, a) \in R$. Ansonsten legt sie aber eine eindeutige Reihenfolge fest. Das Sortierproblem besteht dann darin, eine gegebene Liste mit Elementen aus der Menge M in die richtige durch die Ordnung vorgegebene Reihenfolge zu bringen.

Die Relation „kleiner als“ $(<)$ erfüllt auf der Menge $\mathbb{Z}$ der ganzen Zahlen diese Axiome und ist daher ein Beispiel für eine strenge schwache Ordnung. Im Folgenden betrachten wir zur Erklärung der Sortieralgorithmen beispielhaft $M = \mathbb{Z}$ mit $<$, wenn nichts Gegenteiliges angegeben ist.

Die Liste wird im Rechner durch ein *Feld* (engl. array) modelliert, auf dessen Elemente über Indizes zugegriffen werden kann. Die Indizes eines Feldes der Länge n sind in C++ nichtnegative ganze Zahlen von 0 bis $n - 1$.

Sortieralgorithmen

Das Problem des Sortierens kann auf sehr unterschiedliche Weise gelöst werden. Besonders wenn zusätzliche Informationen über die Menge vorliegen, können sich einige Algorithmen[1] als wesentlich effizienter erweisen als andere. Im Folgenden betrachten wir jedoch den allgemeinen Fall, in dem keine weiteren Kenntnisse über die Struktur des Sortierproblems vorhanden sind.

Das wichtigste Kriterium bei der Auswahl eines Sortieralgorithmus ist der Aufwand, den man für die Sortierung eines gegebenen Feldes zu erwarten hat. Dieser hängt aber bei allen Algorithmen von der Vorsortierung der Daten ab. Oft betrachtet man daher den günstigsten, den ungünstigsten und den durchschnittlichen Fall. Bei allen Sortierverfahren, ist es üblich, den Aufwand anhand der Anzahl der durchgeführten Vergleichsoperationen zu messen.

Darüber hinaus ist auch der benötigte Speicherplatz zu beachten. Viele Sortierverfahren arbeiten *ortsfest* (engl. in-place), das heißt, zusätzlich zu dem zu sortierenden Feld selbst wird nur weiterer Speicherplatz fester Größe unabhängig von der Länge des zu sortierenden Feldes benötigt. Andere Verfahren verlangen hingegen weiteren Speicherplatz in der Größenordnung des zu sortierenden Feldes.

[1] zum Beispiel Bucketsort

Ein weiteres wichtiges Merkmal, ist die Stabilität eines Sortieralgorithmus. Ein Sortierverfahren heißt *stabil*, wenn für zwei Elemente gleichen Ranges $a, b \in M$, das heißt, es gilt $(a, b) \notin R$ und $(b, a) \notin R$, die Reihenfolge stets beibehalten wird.

Bubblesort

Der Algorithmus Bubblesort stellt eines der einfachsten Sortierverfahren dar. Er durchläuft das gesamte Feld in Vorwärtsrichtung. Für jedes Element i durchläuft der Algorithmus das Teilfeld mit den Indizes j mit $j > i$ rückwärts. Erreicht er hier Element j des zu sortierenden Feldes, so vergleicht er es mit dem linken Nachbarn und vertauscht die beiden Elemente, falls das Element kleiner ist als der linke Nachbar. Der Name stammt daher, dass die großen Elemente des Feldes aufsteigen wie Luftblasen (engl. bubble) im Wasser.

Bubblesort ist bei geeigneter Implementierung ein stabiles Sortierverfahren. Der Aufwand zum Sortieren eines Feldes mit n Elementen mit dem Algorithmus Bubblesort, gemessen durch die Anzahl der durchgeführten Vergleiche, beträgt im günstigsten Fall (vorsortiertes Feld) die Ordnung $\mathcal{O}(n)$, im ungünstigsten Fall (entgegengesetzt vorsortiertes Feld) und im durchschnittlichen Fall $\mathcal{O}(n^2)$. Daher ist dieses Verfahren in der Praxis besonders für große zu sortierende Datenmengen ungeeignet.

Auswahlsortieren

Das *Auswahlsortieren* (engl. selection sort) ist ein weiteres stabiles Sortierverfahren und stellt eine Verbesserung des Bubblesort-Algorithmus dar. Statt des Austausches mit dem Nachbarelement wird in der inneren Schleife zunächst das kleinste Element des verbliebenen Feldes gesucht und dann das aktuelle Element gezielt mit diesem vertauscht. Der Aufwand beträgt in jedem Fall $\mathcal{O}(n^2)$. Obwohl der Algorithmus im Durchschnitt den gleichen Aufwand wie Bubblesort besitzt, zeigt er in der Praxis oft etwas günstigeres Laufzeitverhalten als Bubblesort.

Einfügesortieren

Das *Einfügesortieren* (engl. insertion sort) baut wie die anderen bisher erklärten Verfahren am Anfang des Feldes ein sortiertes Teilfeld auf, in das nach und nach alle restlichen Elemente eingefügt werden. Dazu durchläuft es das gesamte Feld in Vorwärtsrichtung. Beim Element i angekommen sind die Elemente mit den Indizes 0 bis $i-1$ bereits sortiert und das Element mit Index i wird an der richtigen Stelle in dieses geordnete Teilfeld eingefügt, wobei alle nachfolgenden Elemente des Teilfeldes um einen Platz nach rechts verschoben werden. Die Komplexitätsabschätzung liefert das gleiche Ergebnis wie bei Bubblesort: Im günstigsten Fall werden $\mathcal{O}(n)$, im ungünstigsten und im durchschnittlichen Fall $\mathcal{O}(n^2)$ Vergleiche benötigt.

Heapsort

Der Algorithmus Heapsort wurde von Robert W. Floyd [3] und J. W. J. Williams [10] vorgeschlagen. Ihm liegt der *Heap* (engl. Halde, Haufen) zugrunde. Dies ist eine spezielle Datenstruktur, die so konstruiert ist, dass sich leicht das größte Element entnehmen lässt.[2]

[2] Ein Heap lässt sich leicht als spezieller binärer Baum darstellen. Wir klären daher zunächst die Begriffe: Ein Wurzelbaum ist ein spezieller gerichteter Graph, bei dem sich ausgehend von einem speziellen Knoten, der Wurzel, alle anderen Knoten des Graphen erreichen lassen. In der Informatik ist mit einem binären Baum ein Wurzelbaum gemeint, bei dem man von jedem Knoten aus höchstens zwei andere Knoten (Kindknoten) erreichen kann. Die Knoten, die keine Kindknoten besitzen, werden als Blätter bezeichnet.

Darstellung eines Feldes als binärer Baum In einem *binären Baum* bezeichnet man als *Level* eines Knotens die Anzahl der Kanten, die man durchlaufen muss, um von der Wurzel, hier der Knoten 0, zu diesem Knoten zu gelangen. Sind zwei Knoten im binären Baum direkt verbunden, so bezeichnet man den mit dem kleineren Level als *Vater* und den mit dem größeren Level als *Kind*.

Um ein Feld sortieren zu können, interpretieren wir es zunächst als binären Baum, indem wir die Knoten des Baumes levelweise hintereinander den Einträgen des Feldes zuordnen. Die Abbildung 1 zeigt diese Zuordnung für ein Beispiel. Zur Erklärung des Algorithmus betrachten wir im Folgenden statt des Feldes den zugehörigen binären Baum und interpretieren ihn am Ende umgekehrt wieder als Feld.

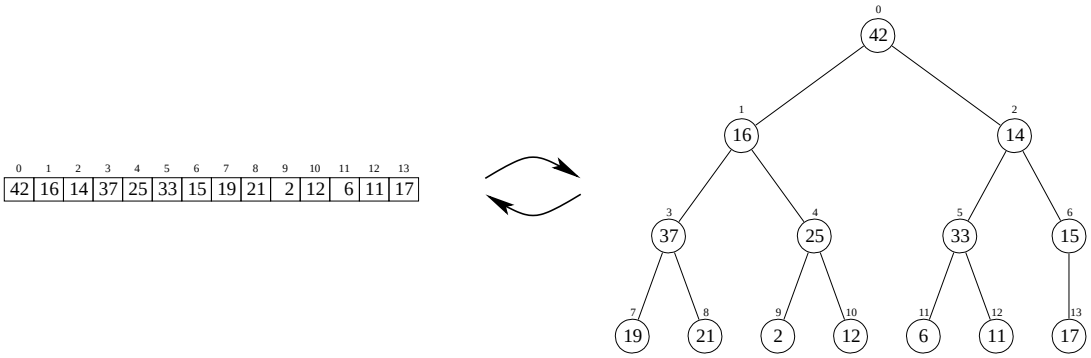

Abbildung 1: Darstellung des Beispielfeldes als binärer Baum

Überführen eines binären Baumes in einen Heap Ein Heap ist ein binärer Baum, der zusätzlich die Heap-Bedingung erfüllt. Diese besagt, dass ein Element bezüglich der Ordnung nicht kleiner ist als seine Kinder. Im Allgemeinen ist dies natürlich nicht der Fall, zum Beispiel erfüllt Knoten 2 des binären Baumes in Abbildung 1 diese Bedingung nicht. Jedoch lässt sich der dort gegebene Binärbaum in einen Heap umwandeln.

Dazu betrachten wir zunächst den Fall, dass der an einem Knoten i beginnende binäre Teilbaum kein Heap ist, die beiden Teilbäume, die an den Kindern des Knotens i beginnen, jedoch bereits die Heap-Bedingung erfüllen. In Abbildung 1 ist dies bei Knoten 1 erfüllt, denn der hier beginnende Teilbaum erfüllt die Heap-Bedingung nicht, die bei Knoten 3 und 4 beginnenden Teilbäume jedoch schon.

Nun bestimmt man das bezüglich der Ordnung größte Element aus dem Knoten i und seinen höchstens zwei Kindern und tauscht dieses Element gegen den Knoten i. Dieser Tauschalgorithmus wird auch als `heapify(i)` benannt, wenn er auf Knoten i angewendet wird.[2] Im Beispiel in Abbildung 1 vertauscht man also Knoten 1 mit Knoten 3.

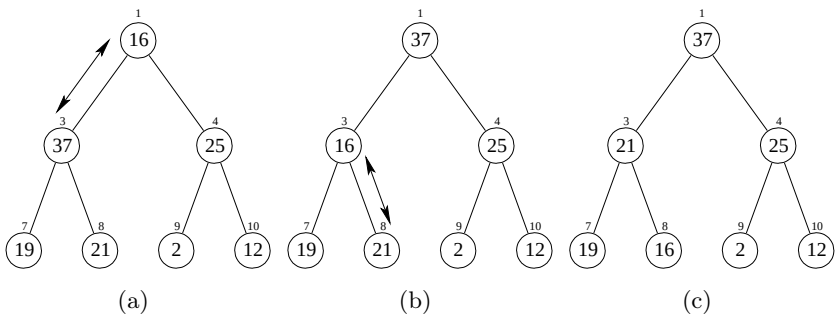

Abbildung 2: Aufbau eines Heaps aus zwei Teilheaps: (a) Aufruf von `heapify(1)` (b) Knoten 3 wurde verändert, daher rekursiver Aufruf von `heapify(3)` (c) Ergebnis

War vorher die Heap-Bedingung an jedem Kindknoten von Knoten i erfüllt, so wurde möglicherweise nun einer der Kindknoten verändert. Dann muss sichergestellt werden, dass dieser Teilbaum weiterhin die Heap-Bedingung erfüllt und daher ruft sich `heapify` in diesem Fall mit dem Index des veränderten Kindknotens

als Argument nochmals auf. Diese Selbstaufrufe setzen sich kaskadenartig fort. Es liegt ein rekursiver Algorithmus vor, der terminiert, sobald bei einem Aufruf kein Kindknoten verändert wird oder alle Kindknoten auch Blattknoten sind. Am Ende wurde der gesamte Teilbaum in einen Heap umgewandelt. Die Abbildung 2 zeigt diese Schritte anhand des Teilbaums, der in Abbildung 1 bei Knoten 1 beginnt.

Um nun einen binären Baum insgesamt in einen Heap umzuwandeln, durchläuft man alle Knoten, die keine Blätter sind, levelweise vom zweithöchsten Level[3] bis zur Wurzel auf Level 0 und wendet auf den jeweiligen Knoten i die Funktion `heapify(i)` an. Am Ende erhält man den gewünschten Heap.

Extrahieren des größten Elementes aus einem Heap Das bezüglich der Ordnung größte Element ist in einem Heap immer an der Wurzel zu finden und kann daher leicht entnommen werden. Nach dem Entnehmen dieses Elementes verschiebt man ein Blattelement an diese Stelle und stellt die Heap-Eigenschaft durch Aufruf der Funktion `heapify(0)` wieder her, wenn Knoten 0 die Wurzel des Heaps bezeichnet. Wegen des geringen Aufwands für diesen Vorgang eignet sich ein Heap auch zur Modellierung einer Warteschlange, denn man kann leicht das Element, das die höchste Priorität hat[4] und daher am wenigsten lange warten soll, erkennen und abarbeiten.

Heapsort-Algorithmus Der Heapsort-Algorithmus beruht auf der Entnahme des größten Elementes in jedem Schritt. Dazu vertauscht man die Wurzel mit dem letzten Element im Heap und stellt sich vor, dass das nun letzte Element nicht mehr zum binären Baum gehört. Anschließend sorgt ein Aufruf von `heapify(0)` wieder dafür, dass die Heap-Bedingung im verkleinerten Baum wieder gilt. So entsteht beginnend mit dem größten Element von hinten nach vorne eine sortiertes Feld.

Man kann zeigen, dass der Aufwand des Algorithmus Heapsort die Ordnung $\mathcal{O}(n \log(n))$ besitzt.[2] Heapsort ist nicht stabil.

Quicksort

Die nun folgenden Verfahren basieren auf der Idee *Teile und herrsche* (lat. divide et impera, engl. divide and conquer): Man führt das Problem des Sortierens eines langen Feldes auf das Sortieren von zwei kürzeren Feldern zurück. Diese Rekursion endet, wenn die zu sortierenden Felder nur noch aus einem Element bestehen.

Der Algorithmus Quicksort wurde von Charles A. R. Hoare entwickelt [4]. Man sucht sich im zu sortierenden Feld zunächst ein Trennelement (auch Pivotelement) aus. Wir wählen hier das letzte Element des Feldes. Das Feld wird nun so umsortiert, dass vor dem Pivotelement im Feld nur Elemente sind, im Vergleich zu denen das Pivotelement nicht kleiner ist, und nach dem Pivotelement nur solche folgen, die nicht kleiner als das Pivotelement sind (vgl. Tabelle 1).

Rekursions-	Feld													
tiefe	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	(42	16	14	37	25	33	15	19	21	2	12	6	11	[17])
1	(11	16	14	6	12	2	[15])	17	(21	33	25	37	42	[19])
2	(11	2	14	6	[12])	15	([16])	17	19	(33	25	37	42	[21])
3	(11	2	[6])	12	([14])	15	16	17	19	21	(25	37	42	[33])
4	([2])	6	([11])	12	14	15	16	17	19	21	([25])	33	(42	[37])
5	2	6	11	12	14	15	16	17	19	21	25	33	37	42

Tabelle 1: Quicksort für das Beispielfeld bezüglich der strengen schwachen Ordnung „<“

Dazu legt man zwei Indizes an. Der erste wird mit dem ersten, der zweite mit dem vorletzten Element des Feldes initialisiert. Solange das Element, auf das der erste Index verweist, nicht größer als das Pivotelement

[3] Die Knoten des höchsten Levels sind Blätter.
[4] Bei einer Warteschlange entspricht das größte Element der Aufgabe mit der höchsten Priorität.

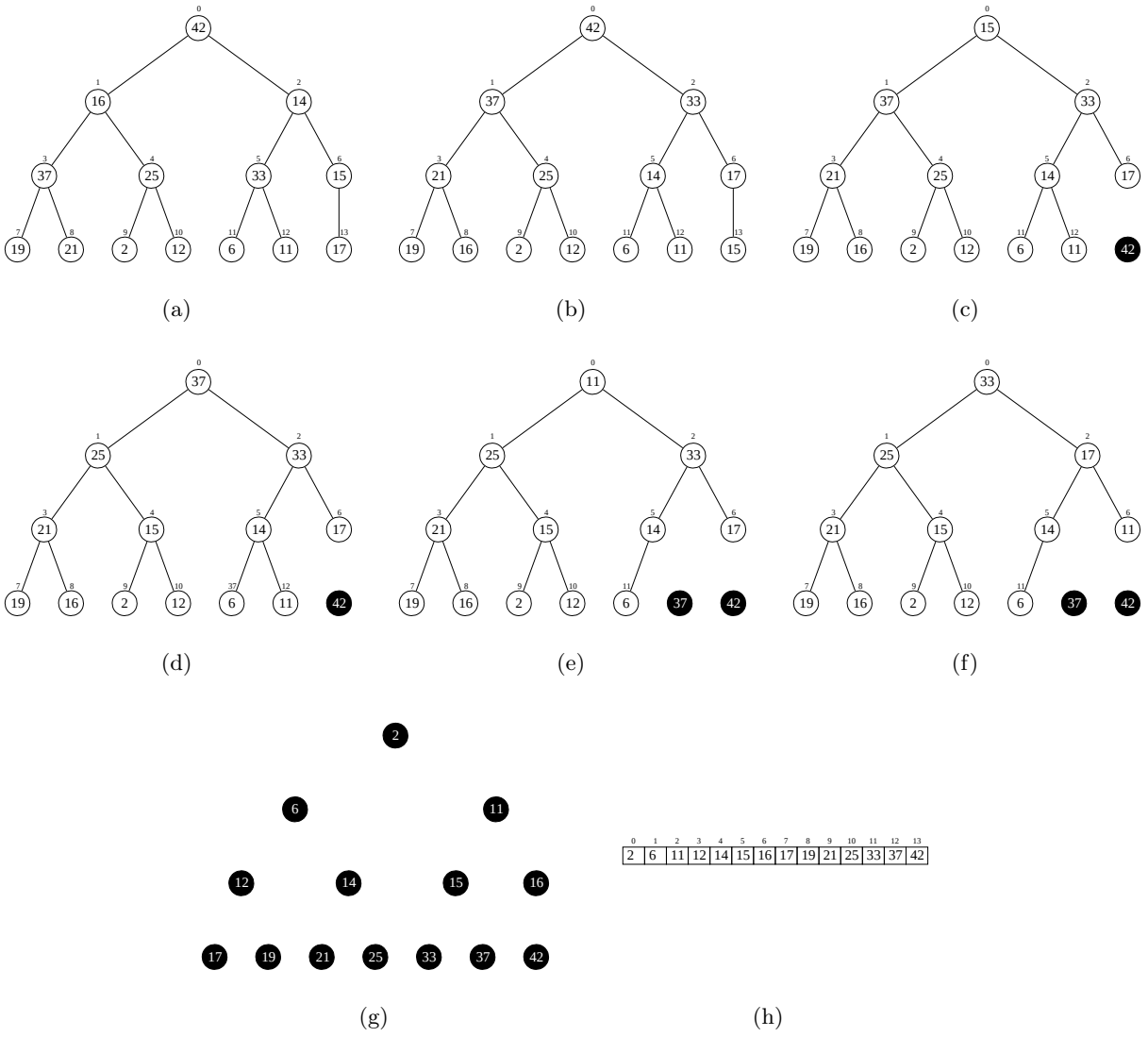

Abbildung 3: Heapsort: (a) Binärbaum aus Abbildung 1 (b) Umwandlung in einen Heap (c) Tausch des ersten Elementes mit dem letzten und Verkleinerung des Binärbaums (d) `heapify(0)` (e) Erneut Tausch des ersten Elementes mit dem letzten und Verkleinerung des Binärbaums (f) Erneut `heapify(0)` (g) Heapsort abgeschlossen (h) Sortiertes Feld

ist, wird der Index um Eins erhöht. Ebenso wird der zweite Index um Eins erniedrigt, solange das Element, auf das der zweite Index verweist nicht kleiner als das Pivotelement ist. Sind beide Bedingungen erfüllt, so werden die Elemente, auf die die beiden Indizes verweisen, vertauscht und anschließend der Vergleich mit dem Pivotelement fortgesetzt. Ist der zweite Index kleiner als der erste, so liegt die endgültige Position des Pivotelementes fest, es kann an diese Position getauscht werden und die Elemente kleiner oder gleich dem Pivotelement befinden sich im Feld davor, diejenigen größer oder gleich dem Pivotelement dahinter.

Man kann das gesamte Feld nun als aus dem Pivotelement und zwei Teilfeldern bestehend interpretieren, das erste Teilfeld beginnt mit dem ersten Element und endet vor dem Pivotelement, die zweite beginnt mit dem Nachfolger des Pivotelementes und endet mit dem letzten Element. Anschließend wird jedes der beiden Teilfelder durch rekursiven Aufruf von Quicksort sortiert. Die Rekursion terminiert, da einelementige Felder stets sortiert sind.[5] Da die beiden Teilfelder ortsfest sortiert werden, ist kein Aufwand zum Zusammenfügen mehr nötig, das Feld ist bereits fertig sortiert. Die Tabelle 1 zeigt die Elemente des Feldes vor dem rekursiven Aufruf. Die Pivotelemente sind durch Kästen markiert.

Im ungünstigsten Fall benötigt Quicksort $\mathcal{O}(n^2)$, im mittleren Fall aber nur $\mathcal{O}(n \log(n))$ Vergleiche und die

[5] Man kann auch bereits den Spezialfall des zweielementigen Feldes behandeln, indem man falls nötig die beiden Elemente tauscht.

Rekursions-	Feld													
tiefe	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	(42	16	14	37	25	33	15	19	21	2	12	6	11	[17])
1	(42	16	14	11	25	33	15	6	21	2	[12])	17	(37	[19])
2	(2	21	[11])	12	(25	33	15	6	16	42	[14])	17	([37])	19
3	([21])	11	([2])	12	(42	[33])	14	(6	16	25	[15])	17	37	19
4	21	11	2	12	([42])	33	14	([25])	15	(16	[6])	17	37	19
5	21	11	2	12	42	33	14	25	15	([6])	16	17	37	19
5	21	11	2	12	42	33	14	25	15	6	16	17	37	19

Tabelle 2: Quicksort für das Beispielfeld bezüglich $R := \{(a, b) \subset \mathbb{Z} \times \mathbb{Z} : (a \mod 10) < (b \mod 10)\}$

in der Landau-Notation versteckten Konstanten erweisen sich in der Praxis als recht klein. Andererseits lässt sich nur mit wesentlich mehr Aufwand eine rekursionsfreie Variante implementieren. Quicksort ist nicht stabil, wie man auch am Beispiel in Tabelle 2 ablesen kann.

Median von Drei Die Effizienz des Algorithmus hängt wesentlich von der Wahl des Pivotelementes ab. Dieses sollte so gewählt werden, dass die beiden entstehenden Teilfelder möglichst gleich groß sind. Man erkennt deutlich, dass die Wahl der Pivotelemente in Tabelle 1 nicht optimal ist. Die Suche des optimalen Pivotelementes ist aber zu aufwändig. In der Praxis wählt man oft aus dem ersten, dem letzten und dem in der Mitte im unsortierten Feldes gespeicherten Elementen das bezüglich der Ordnung mittlere Element aus und wählt es als Pivotelement. Gegenüber der Standardwahl erhält man im durchschnittlichen Fall eine messbare Verringerung des Aufwandes. Jedoch gibt es weiterhin ungünstige Vorsortierungen, in denen $\mathcal{O}(n^2)$ Vergleiche nötig sind, statistisch treten diese aber seltener auf.

Introsort Durch geeignete Kombination von Quicksort mit Heapsort, das in für Quicksort ungünstigen Fällen eine Komplexität von $\mathcal{O}(n \log(n))$ sicherstellt, erhält man den Algorithmus Introsort [8] den David Musser 1997 vorschlug. Er gilt als zur Zeit schnellstes instabiles Sortierverfahren und wird unter anderem auch in der C++ Standard Library [1] heute eingesetzt.[6]

Mergesort

Statt des Tausches von Elementen basiert der Algorithmus Mergesort auf der Beobachtung, dass sich zwei sortierte Felder leicht zu einem sortierten Gesamtfeld zusammenfassen lassen. Dazu vergleicht man die jeweils kleinsten Elemente der beiden Teilfelder und fügt das jeweils kleinere Element an das neue Feld an. Mergesort wurde laut Donald E. Knuth [7] im Jahr 1945 von John von Neumann vorgeschlagen.

Das Problem ein geeignetes Pivotelement festzulegen entfällt, denn man kopiert das gegebene Feld in zwei Teilfelder, die sich in der Größe höchstens um ein Element unterscheiden. Anschließend erfolgt wiederum der rekursive Aufruf für jedes der Teilfelder. Wieder terminiert die Rekursion im Falle des einelementigen Feldes.

Der Ablauf des Algorithmus Mergesort für das Beispielfeld ist in Tabelle 3 dargestellt. Anders als bei Quicksort wird die Hauptarbeit bei Mergesort nicht beim Trennen des Feldes in Teile sondern beim Zusammenfügen der Teile geleistet. Mergesort besitzt im günstigsten wie auch im ungünstigsten Fall einen Aufwand von $\mathcal{O}(n \log(n))$ und ist ein stabiles Verfahren. Im Gegensatz zu allen bisher betrachteten Algorithmen arbeitet Mergesort nicht am Platz und benötigt daher zusätzlichen Speicherplatz der Größenordnung $\mathcal{O}(n)$. Die bisher aufgeführten Verfahren begnügen sich mit einer von der Länge des zu sortierenden Feldes unabhängigen Menge zusätzlichen Speicherplatzes (Größenordnung $\mathcal{O}(1)$).

[6] Die Implementierung der Standard Template Library von SGI nutzt Introsort seit Juni 2000.

Rekursions-tiefe	Feld													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	(42	16	14	37	25	33	15	19	21	2	12	6	11	17)
1	(42	16	14	37	25	33	15)(	19	21	2	12	6	11	17)
2	(42	16	14	37)(	25	33	15)(	19	21	2	12)(	6	11	17)
3	(42	16)(	14	37)(	25	33)(	15)(	19	21)(	2	12)(	6	11)(	17)
4	(42)(	16)(	14)(	37)(	25)(	33)(	15)(	19)(	21)(	2)(	12)(	6)(	11)(	17)
3	(16	42)(	14	37)(	25	33)(	15)(	19	21)(	2	12)(	6	11)(	17)
2	(14	16	37	42)(	15	25	33)(	2	12	19	21)(	6	11	17)
1	(14	15	16	25	33	37	42)(	2	6	11	12	17	19	21)
0	(2	6	11	12	14	15	16	17	19	21	25	33	37	42)

Tabelle 3: Mergesort für das Beispielfeld

Aufgabenstellung

Implementieren Sie die Algorithmen für Bubblesort, Auswahlsortieren, Einfügesortieren, Heapsort, Quicksort (einfach und mit Median aus Drei-Erweiterung) und Mergesort zum Sortieren eines gegebenen Feldes von nichtnegativen ganzen Zahlen (Typ `unsigned int`) in C++. Benutzen Sie als Datenstruktur für das Feld ein C-Array.

Aufbau des Programms

Die Praktikumsumgebung stellt bereits einige Funktionalität für Sie zur Verfügung. Mit der Funktion

```
void start(const unsigned int beispiel, size_t &laenge, unsigned int *&
   feld);
```

erhalten Sie ein neues zu sortierendes Feld. In der Variable `laenge` können Sie eine gewünschte Länge des Feldes vorgeben, die Funktion liefert Ihnen nach dem Aufruf die tatsächliche Länge des zu sortierenden Feldes zurück, die von der gewünschten abweichen kann, da einige der Beispiele nur eine fixe Länge unterstützen. Um die Elemente i und j im Feld zu vertauschen, benutzen Sie bitte die Funktion

```
void tausche(unsigned int *feld, size_t i, size_t j);
```

Nach abgeschlossener Sortierung geben Sie bitte das sortierte Feld mit

```
bool ergebnis(unsigned int *feld);
```

zurück. Anschließend wird Ihr Sortierergebnis analysiert und Sie erhalten eine Bewertung.

Unterteilen Sie ihr Programm sinnvoll in einzelne Funktionen, so dass Sie mindestens eine Funktion pro Sortieralgorithmus erstellen.

Dynamische Speicherverwaltung

Benötigen Sie in einer Funktion ein lokales Feld, dessen Größe sie zuvor nicht kennen, so fordern Sie vom Betriebssystem hierfür dynamisch Speicherplatz an. Benutzen Sie dazu den C++-Befehl `new`. Sobald das Feld nicht mehr verwandt wird, geben Sie das Feld mit `delete` wieder frei. Bitte sehen Sie die Syntax der Befehle in einem C++-Handbuch oder einer Referenz[7] nach, bevor Sie mit dem Programmieren beginnen.

[7] zum Beispiel `http://www.cppreference.com`

Typische Programmfehler

Beim Umgang mit Feldern oder Zeigern ist unbedingt darauf zu achten, dass der erlaubte Speicherbereich nicht überschritten wird. Wird der Speicherbereich, der einem Programm zugeteilt ist, überschritten, das heißt, versucht ein Programm außerhalb dieses Bereiches zu lesen oder zu schreiben, wird das Programm vom Betriebssystem abgebrochen und dem Anwender der Fehler *Bereichsüberschreitung* (engl. segmentation fault) gemeldet.

Bitte beachten Sie, dass falls dieser Fehler nicht auftritt, das keineswegs heißen muss, dass alle Zugriffe im erlaubten Bereich erfolgen, denn der Bereich eines Feldes ist im Allgemeinen deutlich kleiner als der gesamte einem Programm zugeteilte Speicherbereich. Daher ist hier sorgfältige Programmierung besonders wichtig.

Fordert Ihr Programm vom Betriebssystem mit dem Befehl `new` Speicherplatz an, so ist es wichtig zu prüfen, ob der Speicher für Ihr Programm auch erfolgreich bereitgestellt wurde. Bitte achten Sie auch genau darauf, dass dieser auch auf jeden Fall mit `delete` wieder freigegeben wird, wenn das Programm endet. Mischen Sie keinesfalls die C++-Befehle `new` und `delete` mit den C-Befehlen `malloc` und `free` und bevorzugen Sie hier die C++-Befehle. Beachten Sie, dass ein mit `new` angelegtes Feld mit `delete []` wieder freigegeben werden muss (eckige Klammern, darin keine Größenangabe).

Literatur

[1] *C/C++-Referenz.* `http://www.cppreference.com/index.html`.

[2] Cormen, T. H., C. E. Leiserson und R. L. Rivest: *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1994.

[3] Floyd, R. W.: *Algorithm 245 - Treesort 3.* Communications of the ACM, 7(12):701, 1964.

[4] Hoare, C. A. R.: *Quicksort.* Computer Journal, 5:10–15, 1962.

[5] Horowitz, E., S. Sahni und S. Anderson-Freed: *Grundlagen von Datenstrukturen in C.* International Thomson Publishing, Bonn, 1994.

[6] JaJa, J. F.: *A Perspective on Quicksort.* Computing in Science & Engineering, 2(1):43–49, 2000.

[7] Knuth, D. E.: *The Art of Computer Programming. Vol. 3: Sorting and Searching.* Addison-Wesley Publishing Company, Reading, MA, 2. Auflage, 1997.

[8] Musser, D. R.: *Introspective Sorting and Selection Algorithms.* Software Practice and Experience, 27(8):983–993, 1997.

[9] Sedgewick, R.: *Algorithms.* Addison-Wesley Publishing Company, Reading, MA, 1988.

[10] Williams, J. W. J.: *Algorithm 232 - Heapsort.* Communications of the ACM, 7(6):347–348, 1964.

[11] Wirth, N.: *Algorithmen und Datenstrukturen, Pascal-Version.* Leitfäden der Informatik. B.G. Teubner, Wiesbaden, 1998.