Sortieralgorithmen

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Suchen und Sortieren sind Problemstellungen, die in einer Vielzahl von Programmen auftreten. Beispiele sind das Verwalten von Musiksammlungen oder Profile in sozialen Netzwerken, die Berechnung des Median einer Datenmenge oder das Erstellen von Histogrammen. Liegen große Datenmengen vor, müssen effiziente Algorithmen verwendet werden, welche die Laufzeit und den Speicherbedarf der Programme optimieren.
Themen dieses Kapitels:

- Sortieren durch Auswählen, Sortieren durch Einfügen, Mergesort, Quicksort

1 Sortieren

Sortieren ist eine Standardaufgabe, die Teil vieler spezieller oder auch umfassenderer Aufgabenstellungen ist. Sie ist u.a. eine Voraussetzung für die binäre Suche, die wir uns im nächsten Abschnitt anschauen werden.

> Untersuchungen zeigen, dass „mehr als ein Viertel der kommerziell verbrauchten Rechenzeit auf Sortiervorgänge entfällt". (Ottmann, Widmayer: Algorithmen und Datenstrukturen, Kap. 2).

Formalisierung des Sortierproblems Gegeben ist eine Liste von Elementen S und eine totale Ordnung auf diesen Elementen. Gesucht ist eine Permutation π von S, so dass π folgende Reihenfolge aufweist:

$$\pi(0) \leq \pi(1) \leq \cdots \leq \pi(N-1)$$

Dabei ist eine Permutation π einer Liste S eine Liste, deren Elemente genau den Elementen von S entspricht, jedoch kann sich die Reihenfolge der Elemente unterscheiden.
Wir betrachten hier vier klassische Sortieralgorithmen: Sortieren durch Auswählen, Sortieren durch Einfügen, Mergesort und Quicksort.
Für jeden dieser Algorithmen stellen wir zunächst die algorithmische Grundidee vor. Wir präsentieren außerdem eine Implementierung, die Elemente von Listen bzw. Arrays sortiert. Dabei verwenden wir zur besseren Übersicht int-Elemente; die gleiche Vorgehensweise kann aber auch gewählt werden, um Datensätze anhand der Schlüssel zu sortieren.

1.1 Sortieren durch Einfügen (insertion sort)

Algorithmische Grundidee

- Füge die noch nicht sortierten Elemente nacheinander in die bereits sortierte Teilliste an der richtigen Stelle ein.
- Beginne mit der leeren Liste (welche bereits sortiert ist)

Implementierung für einfach verkettete Liste Beim Sortieren von Listen müssen die Listenknoten (evtl.) in eine neue Reihenfolge gebracht werden. Die Knoten selbst bleiben dabei aber erhalten.
Wir betrachten zunächst die Funktion `insert_sorted_rec`, welche einen Zeiger `first` auf den ersten Knoten einer bereits sortierten Liste nimmt und den Knoten `n` dann so in die Liste einfügt, dass die Liste wieder sortiert ist. Die Funktion gibt einen Zeiger auf den neuen ersten Knoten der Liste zurück.

```
105 node_t *insert_sorted_rec(node_t *first, node_t *n)
106 {
107   if (first == NULL || n->value < first->value)
108   {
109     n->next = first;
110     return n;
111   }
112   else
113   {
114     first->next = insert_sorted_rec(first->next, n);
115     return first;
116   }
117 }
```

Wir können nun diese Funktion verwenden, um alle Knoten der Liste nacheinander sortiert in eine anfangs leere Liste (`target`) einzufügen. Dabei ist `target` der Zeiger auf den ersten Knoten der bereits sortierten Teilliste.

```
121 void insertion_sort_list_r(linked_list_t *list)
122 {
123   node_t *target = NULL;
124   node_t *n = list->first;
125   while (n)
126   {
127     node_t *next = n->next;
128     target = insert_sorted_rec(target, n);
129     n = next;
130   }
131   list->first = target;
132 }
```

Statt Änderungen am Zeiger auf das erste Element wie oben durch den Rückgabewert zu behandeln, lässt sich in C auch ein Zeiger auf den Zeiger auf das erste Elemente der Liste verwenden. Diese Variante kann dann wie folgt implementiert werden:

```
136 void insert_element_sorted (node_t **target_p, node_t *n)
137 {
138   /* p ist ein Zeiger auf die (potentiellen) Einfuegestelle */
139   node_t** p = target_p;
140   /* Iteriere an die richtige Einfuegestelle:
141      entweder am Ende der Liste oder vor den ersten Knoten,
142      der ein groesseres Element enthaelt */
143   while (*p != NULL && (*p)->value < n->value)
144   {
145     p = &((*p)->next);
146   }
147   /* Setze den neuen Nachfolger */
148   n->next = *p;
149   /* Fuege den Knoten n an der Einfuegestelle ein */
150   *p = n;
151 }
152
153 void insertion_sort_list (linked_list_t *ll)
154 {
155   /* Verweist auf das erste Element einer verlinkten Knotenliste,
156      in welche die Knoten aus ll sortiert eingefügt werden */
157   node_t *target = NULL;
158   /* Iteriere ueber die Knoten der unsortierten Liste,
159      jeweils aktueller Knoten ist *curr  */
160   node_t **curr = &(ll->first);
161   while (*curr) {
162     /* Entferne den aktuellen Knoten aus der unsortierten Liste
        */
163     node_t *n = *curr;
164     *curr = (*curr)->next;
165     /* Fuege ihn in die Hilfsliste sortiert ein */
166     insert_element_sorted(&target, n);
167   }
168   /* Setze den Zeiger von der Eingabeliste auf die sortierte
169      Hilfsliste */
170   ll->first = target;
171 }
```

1.2 Sortieren durch Auswahl (Selection Sort)

Algorithmische Idee

- Entferne ein minimales Element `min` aus der Liste der unsortierten Elemente.
- Sortiere nun rekursiv die Liste der übrigen Elemente, aus der `min` entfernt wurde.
- Füge `min` dann als erstes Element vorne an die sortierte Ergebnisliste an.

Statt ein minimales Element auszuwählen und vorne an die sortierte Folge der Elemente anzufügen, kann man analog auch ein maximales Element wählen und es hinten anfügen.

Implementierung auf Arrays

1. Finde einen Index $imin$ des Arrays f, so dass $f[imin]$ ein minimales Element von $f[0]$ bis $f[N-1]$ enthält
2. Vertausche $f[imin]$ und $f[0]$
3. Sortiere dann den Bereich $f[1]$ bis $f[N-1]$ auf gleiche Art

```
15 /* Sortieren durch Auswahl auf einem Array von int */
16 void selectionsort (int *f, int n) {
17   /* Iteriere ueber die Indizes des Arrays */
18   for (int i = 0; i < n - 1; i++) {
19     /* Finde Index fuer minimales Element */
20     int imin = i;
21     for (int j = i+1; j < n; j++) {
22       if (f[j] < f[imin]) {
23         imin = j;
24       }
25     }
26     /* Vertausche Elemente */
27     swap(&f[i], &f[imin]);
28   }
29 }
```

1.3 Sortieren durch Mischen (merge sort)

Merge ist ein typischer Algorithmus, der eine Divide-and-Conquer-Strategie verfolgt:

- Zerlege das Problem in Teilprobleme.
- Wende den Algorithmus rekursiv auf die Teilprobleme an.
- Füge die Teilergebnisse wieder zusammen.

Algorithmische Idee

1. Hat die zu sortierende Liste mehr als ein Element, teile die Liste in zwei gleich große Teile auf (bzw. mit einem Größenunterschied von 1).
2. Sortiere die zwei Teillisten.
3. Füge die zwei sortierten Teillisten nun wieder zu einer sortierten Liste zusammen.

Beispiel:

- Gegeben sei die Liste $[17, 12, 6, 19, 23, 8, 5, 10]$.
- Die Liste wird aufgeteilt in $[17, 12, 6, 19]$ und $[23, 8, 5, 10]$.
- Die zwei Teillisten werden durch rekursive Aufrufe sortiert:
  $[6, 12, 17, 19]$ und $[5, 8, 10, 23]$
- Die sortierten Teillisten können zusammengeführt werden, indem immer das kleinste Element vorne weggenommen wird:

Ergebnis	Teilliste 1	Teilliste 2
$[]$	$[6, 12, 17, 19]$	$[5, 8, 10, 23]$
$[5]$	$[6, 12, 17, 19]$	$[8, 10, 23]$
$[5, 6]$	$[12, 17, 19]$	$[8, 10, 23]$
$[5, 6, 8]$	$[12, 17, 19]$	$[10, 23]$
$[5, 6, 8, 10]$	$[12, 17, 19]$	$[23]$
$[5, 6, 8, 10, 12]$	$[17, 19]$	$[23]$
$[5, 6, 8, 10, 12, 17]$	$[19]$	$[23]$
$[5, 6, 8, 10, 12, 17, 19]$	$[]$	$[23]$
$[5, 6, 8, 10, 12, 17, 19, 23]$	$[]$	$[]$

MergeSort eignet sich besonders gut zur Implementierung auf verketteten Listen. Wird MergeSort auf Arrays implementiert, wird ein Hilfsarray benötigt, um die sortierten Teillisten wieder zusammenzufügen. D.h. bei der Implementierung auf Arrays wird zusätzlich Speicherplatz für ein Array von gleicher Größe wie die Eingabe reserviert. Die Implementierung von Mergesort behandeln wir in der nächsten Übung.

1.4 Quicksort

Wie MergeSort basiert auch Quicksort auf der Divide-and-Conquer-Strategie.

Algorithmische Grundidee

- Wähle ein beliebiges Element e aus der Liste aus (***Pivotelement***).
- Teile die Elemente in zwei Teillisten:
  - Der erste Teil enthält alle Elemente $< e$.
  - Der zweite Teil enthält die Elemente $\geq e$
- Wende nun Quicksort rekursiv auf die Teillisten an.
- Füge anschließend die resultierenden, nun sortierten Teillisten und das Pivotelement wieder zusammen.

Algorithmische Idee für Partitionierung von Arrays: Wir wollen später den Algorithmus auf Arrays implementieren und beim Aufteilen (Partitionieren) des Array in zwei Teile keinen zusätzlichen Speicherplatz verwenden. Das folgende Beispiel zeigt, wie wir dies durch geschicktes Vertauschen von Elementen im Array umsetzen können. Gegeben sei das folgende Array:

17	12	6	19	23	8	5	10

Wir wählen als Pivotelement das letzte Element (10):

17	12	6	19	23	8	5	10

Als nächstes partitionieren wir das Array; d.h. wir vertauschen die Elemente paarweise, so dass die Elemente, die kleiner als das Pivotelement sind, vorne im Array sind, die Elemente, die größer als das Pivotelement sind, im hinteren Teil. Dazu wählen wir das erste Element im Array (17) und das vorletzte Elemente (5).

17	12	6	19	23	8	5	10

Da 5 kleiner als der Pivot ist und 17 größer, vertauschen wir die beiden:

5	12	6	19	23	8	17	10

Danach fahren wir mit den nächsten Elementen 12 und 8 fort. Auch diese werden vertauscht:

5	8	6	19	23	12	17	10

Da 6 kleiner als das Pivotelement ist und 23 größer, sind keine weiteren Vertauschungen mehr möglich:

5	8	6	19	23	12	17	10

Nun tauschen wir das Pivotelement zwischen die beiden Partitionen:

5	8	6	10	23	12	17	19

Nun gilt die gewünschte Eigenschaft: Alle Element links des Pivotelements sind kleiner, alle Element rechts davon größer oder gleich.

Implementierung auf Arrays

- Bearbeite rekursiv Teilbereiche des Arrays (`n`-Elemente, startend bei `f`)
- Realisiere das Teilen der Liste durch Vertauschen
  - → Indexzähler `left`, `right` laufen von links bzw. rechts und suchen Einträge, die vertauscht werden können.
  - → Für die zu tauschenden Einträge gilt:

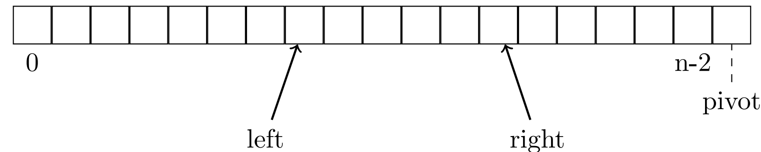
`f[left]` $\geq$ `pivot` und `f[right] < pivot`

→ In jedem Schritt gilt:

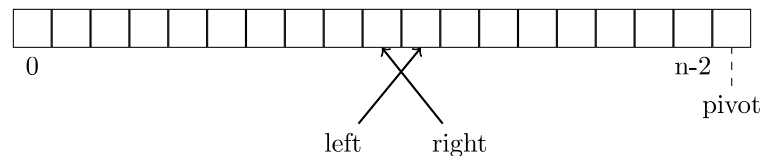
  * Für alle i in [`0, left-1`] : `f[i] < pivot`
  * Für alle i in [`right+1, n-2`] : `pivot` $\leq$ `f[i]`

Wenn ein Paar zum Vertauschen gefunden wurde, gilt `left < right`:

- Vertausche `f[left]` und `f[right]`
- Inkrementiere `left` und dekrementiere `right`
- Fahre dann mit der Suche nach vertauschbaren Elementpaaren fort.



Die Umsortierung des (Teil-)Arrays ist abgeschlossen, sobald `left > right`.



Zum Schluss wird das `pivot` an die richtige Stelle getauscht, indem das Element an Position `left` und das an Position `n-1` vertauscht werden.

Implementierung in C Wir können Quicksort auf Arrays folgendermaßen implementieren:

```
34 // Als Pivot wird das letzte Element gewaehlt;
35 // Hilfsfunktion zum Partitionieren eines Arrays
36 // liefert die neue Position des Pivotelements
37 int partition (int *f, int n){
38    int left = 0;
39    int right = n - 2;
40
41    int pivot = f[n-1];
42    while (1) {
43      while (f[left] < pivot) { left++; }
44      while (left <= right && f[right] >= pivot){ right--; }
45      if( left > right ) {
46        break;
47      } else {
48        swap(&f[left], &f[right]);
49        left++; right--;
50      }
51    }
52    // Pivotelement kommt zwischen die beiden Partitionen
```

```
53      swap(&f[left], &f[n-1]);
54      return left;
55    }
56
57
58    // Quicksort auf einem Array von int
59    void quicksort (int *f,int n) {
60      if (n > 1) {
61        int ixsplit = partition(f, n);
62
63        /* Es gilt:
64        0 <= ixsplit <= n-1
65          && ( fuer alle i:
66               0 <= i < ixsplit => f[i] <= f[ixsplit] )
67          && ( fuer alle i:
68               ixsplit < i <= n-1 => f[i] >= f[ixsplit] )  */
69          quicksort( f, ixsplit);
70          quicksort( &f[ixsplit+1],  (n - 1) - ixsplit );
71        }
72      }
```

Frage 1: Bei einer Implementierung in Java kann man für den rekursiven Aufruf nicht einen Zeiger auf das Teil-Array nutzen. Geben Sie eine Implementierung von Quicksort in Java, die stattdessen die Array-Grenzen für das jeweilige Teilarray berücksichtigt!

Optimierungen Die vorgestellte Quicksort-Fassung arbeitet schlecht auf schon sortierten Arrays. Dies kann man durch folgende Strategien verhindern:

→ Shuffle des Arrays, um Vorsortierung aufzuheben

→ Randomisierte Auswahl des Pivot-Elements (Beispiel: Median von 3 Elementen)

Der Aufwand für die rekursiven Funktionsaufrufe ist außerdem vergleichsweise hoch, wenn die Teil-Arrays nur noch wenige Elemente enthalten. Man kann die Anzahl dieser Aufrufe reduzieren, in dem andere Sortierverfahren wie InsertionSort zur Sortierung von Teil-Arrays mit nur wenigen Elemente verwendet wird.

Hinweise zu den Fragen

Hinweise zu Frage 1:

```
36  // Quicksort auf einem Array von Datensaetzen
37  public static void quicksort (DataSet[] f) {
38    quicksort(f, 0, f.length - 1);
39  }
40
41  private static void quicksort (DataSet[] f, int lo, int hi) {
42    if (lo < hi) {
43      int ixsplit  = partition(f, lo, hi);
44
45      /* Es gilt:
46      lo <= ixsplit <= hi && f[ixsplit].key == pivotKey
47        && ( fuer alle i: lo <= i < ixsplit => f[i].key <= pivotKey )
48        && ( fuer alle i: ixsplit < i <= hi => f[i].key >= pivotKey )  */
49
50        quicksort( f, lo, ixsplit-1 );
51        quicksort( f, ixsplit+1, hi );
52      }
53    }
54
55  // Hilfsmethode zum Partitionieren des Arrays zwischen Index lo und hi
56  // Liefert die Position des Pivotelements
57  private static int partition (DataSet[] f, int lo, int hi){
58    int left = lo;
59    int right = hi - 1;
60
61    int pivot = f[hi].key;
62    while (true) {
63      while (f[left].key < pivot) { left++; }
64      while (left <= right && f[right].key >= pivot){ right--; }
65      if( left > right ) {
66        break;
67      } else {
68        swap(f, left, right);
69        left++; right--;
70      }
71    }
72    // Pivotelement kommt zwischen die beiden Partitionen
73    swap(f, left, hi);
74    return left;
75  }
76
77
78  // Hilfsmethode zum Tauschen zweier Arrayeintraege
79  static void swap (DataSet[] f, int i, int j) {
80    DataSet tmp = f[i];
81    f[i] = f[j];
82    f[j] = tmp;
83  }
```