Sortieralgorithmen und Binäre Suche

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.

Lernziele dieses Kapitels:

- Eine formale Definition des Sortierproblems zu nennen.
- Die klassischen Sortieralgorithmen Sortieren durch Auswählen, Sortieren durch Einfügen, Mergesort und Quicksort zu beschreiben und zu implementieren.
- Binäre Suche in verschiedenen Kontexten anzuwenden und zu implementieren.

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.

2 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 }
```

3 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  }
```

4 Sortieren durch Mischen (merge sort)

MergeSort 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 den Übungen.

5 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, die kleiner als e sind.

  - Der zweite Teil enthält die Elemente, die größer oder gleich e sind.

- Wende nun Quicksort rekursiv auf die Teillisten an.
- Füge anschließend die resultierenden, nun sortierten Teillisten und das Pivotelement wieder zu einer sortierten Liste 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:

  $$\texttt{f[left]} \geq \texttt{pivot} \text{ und } \texttt{f[right]} < \texttt{pivot}$$

  → In jedem Schritt gilt:

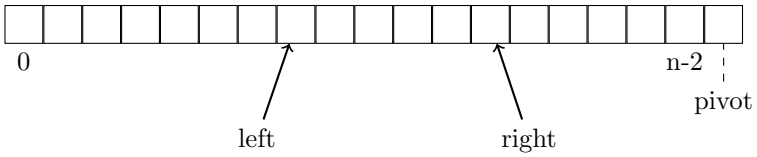
  * Für alle i in $[\texttt{0, left-1}]$: $\texttt{f[i]} < \texttt{pivot}$
  * Für alle i in $[\texttt{right+1, n-2}]$: $\texttt{pivot} \leq \texttt{f[i]}$

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

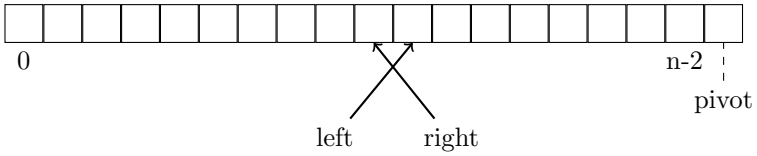
  – Vertausche `f[left]` und `f[right]`

  – 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 bereits 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.

6 Binäre Suche

Kapitel 4.1 aus *Sedgewick, Wayne: Einführung in die Programmierung mit Java*

Frage 2: Ein kleines Ratespiel: Ich stelle mir eine Zahl zwischen 0 und 127 vor; Sie müssen diese Zahl erraten und dürfen mir dazu Fragen stellen. Die einzige erlaubte Frage ist dabei "Ist die Zahl kleiner als _ ?"; die Antwort ist entweder "Ja" oder "Nein".

- Wie gehen Sie vor?
- Wie viele Fragen müssen Sie maximal stellen, um die Zahl zu ermitteln?

Wir betrachten im Folgenden eine Verallgemeinerung dieser Fragestellung:

Wie kann man mit möglichst wenigen Fragen eine Zahl $x \in \mathbb{N}$ zwischen 0 und $N-1$, also aus dem Intervall $[0, N)$ erraten?

Wir betrachten folgende algorithmische Idee zur Lösung des Ratespiels:

- Wir verwenden ein Intervall $[l, h)$ mit $x \in [l, h)$, das in jedem Schritt halbiert wird. Dabei gilt für die gesuchte Zahl x in jedem Schritt: $l \leq x$ und $x < h$
- Als Anfangsintervall wählen wir das Intervall $[0, N)$.
- Rekursive Strategie:

  Basisfall: Enthält das Intervall nur noch eine Zahl ($h - l = 1$), dann ist die gesuchte Zahl $x = l$

  Rekursiver Schritt:

  * Frage, ob die Zahl kleiner ist als die Mitte des Intervalls, $m = l + (h - l)/2$.
  * Falls ja, suche die Zahl im linken Teilintervall $[l, m)$.
  * Andernfalls, suche die Zahl im rechten Teilintervall $[m, h)$.

Man kann dieses Ratespiel folgendermaßen implementieren:

```
1  #include<stdio.h>
2
3  // Sucht die Zahl im Interval [lo,hi)
4  int search(int lo, int hi) {
5    // Basisfall
6    if ((hi-lo) == 1) {
7      return lo;
8    }
```

```
9       // Rekursiver Schritt
10      int mid = lo + (hi -lo) / 2;
11      printf("Ist die Zahl kleiner als %d ? (j/n)\n", mid);
12      char answer;
13      scanf("%c", &answer);
14      getchar(); // liest das naechste Zeichen, d.h. den
        Zeilenumbruch
15      if (answer == 'j') {
16        return search(lo,mid);
17      } else {
18        return search(mid,hi);
19      }
20    }
21
22    int main(void)
23    {
24      printf("Denke dir eine Zahl zwischen 0 und 127!\n");
25      int x = search(0, 128);
26      printf("Die gesuchte Zahl ist %d\n", x);
27    }
```

Frage 3: Wie kann man das Programm erweitern, damit in einem beliebigen Intervall gesucht werden kann?

Korrektheit des Verfahrens Wie können wir *beweisen*, dass unser Suchverfahren das richtige Ergebnis liefert?
Wir machen hierzu die vereinfachende Annahme, dass $N = 2^n$ für ein $n \in \mathbb{N}$ ist. Diese Annahme erleichtert das Halbieren der Intervallgröße im rekursiven Fall. Für jeden Aufruf von `search(h,l)` gilt (Beweis durch Induktion):

- Das Intervall enthält immer die gesuchte Zahl.
- Die Intervallgrößen, d.h. die Differenz $h - l$, sind Zweierpotenzen, die mit jedem rekursiven Aufruf kleiner werden, beginnend mit 2^n und endend mit $2^0 = 1$.

Damit ist sichergestellt, dass das Verfahren terminiert, da der Basisfall mit Intervallgröße 1 immer erreicht wird. Außerdem enthält das Intervall immer, also auch im Basisfall, die gesuchte Zahl.

Analyse Wie viele Fragen werden benötigt, um die gesuchte Zahl zu ermitteln?

- Nach Annahme ist $N = 2^n$ für ein $n \in \mathbb{N}$, d.h. $n = \log_2(N)$.
- Sei $T(N)$ die Anzahl der Fragen, die für die Suche im Intervall der Größe N gestellt werden müssen.

- In jedem Rekursionsschritt wird eine Frage gestellt und das Problem auf einem Intervall halber Größe gelöst:

$$T(N) = 1 + T(N/2)$$

- Im Basisfall ist keine Frage mehr nötig:

$$T(1) = 0$$

- Insgesamt erhalten wir:

$$\begin{aligned} T(N) &= T(2^n) \\ &= T(2^{n-1}) + 1 = T(2^{n-2}) + 2 = \cdots = T(2^{n-n}) + n \\ &= T(2^0) + n = T(1) + n \\ &= n \end{aligned}$$

Man muss also n-mal fragen; beim letzten Aufruf von `search` befindet man sich im Basisfall. Insgesamt benötigt man $n + 1 = \log_2(N) + 1$ Aufrufe von `search`.

6.1 Binäre Suche in sortiertem Array

Die Idee der binären Suche ist auch beim Suchen in sortierten Datensammlungen anwendbar, die einen direkten Zugriff auf einen Eintrag erlauben. Im Alltag verwendet man den Algorithmus beispielsweise bei der Suche in Wörterbüchern oder Telefonbüchern. Wir wenden nun die gleiche algorithmische Idee wie im Ratespiel an, um Einträge in einem sortierten Array zu suchen.

Problembeschreibung Gegeben sei eine Folge $s_0, \ldots, s_{N-1}$ von sortierten *Datensätzen*. Jeder Datensatz s_j besteht aus einem Schlüssel k_j und einem zugehörigen Wert (hier: String). Wir gehen hier davon aus, dass die Schlüssel Integer sind. Wir repräsentieren einen Datensatz durch die Struktur `dataset_t`:

```
5   typedef struct dataset {
6     int key;
7     char *data;
8   } dataset_t;
```

Für sortierte Datensätze gilt:

$$k_0 \leq k_1 \leq \cdots \leq k_{N-1}$$

```
12  /* Hilfsfunktion fuer die rekursive Variante */
13  /* Sucht den Datensatz mit Schluessel x im Indexbereich [lo,hi)
       */
14  dataset_t *search(int x, dataset_t *f, int lo, int hi) {
15    if (hi <= lo) {
```

```
16        // Element nicht gefunden
17        return NULL;
18      }
19      int mid = lo + (hi-lo) /2;
20      if (x < f[mid].key) {
21        // Suche rekursiv im Intervall [lo, mid)
22        return search(x, f, lo, mid);
23      }
24      if (x > f[mid].key) {
25        // Suche rekursiv im Intervall [mid+1, hi)
26        return search(x, f, mid+1, hi);
27      }
28      // Hier gilt nun: x == f[mid].key, d.h. Element gefunden
29      return &f[mid];
30    }
31
32    /* Liefert NULL, wenn Datensatz mit Schluessel x nicht in f
        enthalten ist; andernfalls die Referenz auf den gefundenen
        Datensatz */
33    /* rekursive Variante */
34    dataset_t *binsearch_recursive (int x, dataset_t *f, int n) {
35      return search(x, f, 0, n);
36    }
```

Hinweis: Dieser Algorithmus kann auch in der Implementierung von `searchIndex` für die Maps basierend auf Arrays in leicht abgewandelter Form angewendet werden, um den passenden Array-Index für einen Schlüssel zu ermitteln.Dabei muss sichergestellt sein, dass die Einträge im Array sortiert verwaltet werden (d.h. neue Einträge werden nicht am Ende, sondern sortiert am richtigen Index eingefügt, damit die Schlüssel in der Map immer aufsteigend (bzw. absteigend) sortiert sind).

Wir zeigen hier außerdem zum Vergleich die iterative Variante:

```
40    /* Liefert null, wenn Datensatz mit Schluessel x nicht in f
        enthalten ist; andernfalls die Referenz auf den gefundenen
        Datensatz */
41    /* iterative Variante */
42    dataset_t *binsearch_iterative(int x, dataset_t *f, int n) {
43      int lo = 0;
44      int hi = n - 1;
45
46      // fuer alle int i < lo gilt elems[i].key < x
47      // fuer alle int i > hi gilt elems[i].key > x
48      while (lo <= hi) {
49        int mid = lo + (hi-lo)/2;
50        if (x < f[mid].key) {
```

```
51          hi = mid - 1;
52        } else if (x > f[mid].key) {
53          lo = mid + 1;
54        } else {
55          return &f[mid];
56        }
57      }
58      // lo == hi + 1
59      // fuer alle int i < lo gilt elems[i].key < x
60      // fuer alle int i >= lo gilt elems[i].key > x
61
62      return NULL;
63    }
```

7 Ausblick: Algorithmenanalyse

In diesem Kapitel haben wir verschiedene Algorithmen zum Sortieren von Datensätzen betrachtet.

- Welcher dieser Algorithmen ist denn der Beste?
- Welcher Algorithmus sollte in einem konkreten Anwendungsfall angewendet werden?

In der Vorlesung "Entwicklung und Analyse von Algorithmen (EAA)" werden Sie lernen Algorithmen analysieren, um sie bezüglich ihrer Laufzeit und ihres Speicherbedarfs vergleichen zu können. Diese Analyse kann sowohl durch Messungen bei der Ausführung auf verschiedenen Eingaben als auch durch theoretische Überlegungen anhand der Beschreibung erfolgen. Dabei werden Modelle entwickelt, mit denen sich Laufzeit (bzw. Speicherbedarf) in Abhängigkeit der Eingabe abzuschätzen lassen. Die theoretische Analyse des Algorithmus ist wichtig, da sie tiefere Einblicke gibt und eventuell auch Fälle aufzeigt, die nur selten auftreten und durch Tests evtl. nicht abgedeckt werden. Messungen sind dabei nötig, um das Modell zu validieren und wichtige Kenngrößen zu ermitteln.

7.1 Laufzeitmessungen

Messung der Laufzeit ist auf verschiedene Arten möglich. Eine einfache Möglichkeit die Laufzeit eines Programms zu messen ist das Programm `time`, welches auf den meisten Linux- und Mac-Systemen bereits vorinstalliert ist.[1]
Das Programm `time` nimmt als Programm-Argumente einen Befehl, der ausgeführt und dessen Laufzeit gemessen wird. Um die Laufzeit des Aufrufs `./a.out 30000` zu testen, kann `time` beispielsweise wie folgt aufgerufen werden:

[1] Unter Windows kann alternativ zum Beispiel der `Measure-Command`-Befehl in der PowerShell verwendet werden.

```
$ time ./a.out 30000

real    0m1.167s
user    0m1.160s
sys     0m0.000s
```

Wir können `time` zusammen mit einer `main`-Funktion verwenden, welche ein Array mit gegebener Größe erstellt, mit zufälligen Werten füllt und dann mit `selectionsort` sortiert:

```
// Erstellt ein Array der Größe size mit zufälligen Werten
int *random_array(int size)
{
    int *ar = malloc(size*sizeof(int));
    if (ar == NULL)
    {
        printf("OUT of memory, size = %d\n", size);
        abort();
    }
    for (int i=0; i<size; i++)
    {
        ar[i] = rand();
    }
    return ar;
}

int main(int argc, char **argv) {
    int size = 10000;
    if (argc > 1)
    {
        // Größe von Programmparameter lesen
        size = atoi(argv[1]);
    }
    int *ar = random_array(size);
    selectionsort(ar, size);
    free(ar);
    return 0;
}
```

Beim Entwerfen von Programmen zum Messen der Laufzeit sollte man bedenken, dass Optimierungen, die der Compiler anwenden kann, eventuell das Ergebnis verfälschen könnten. Zum Beispiel könnte ein hinreichend intelligenter Compiler erkennen, dass das sortierte Array nie gelesen wird und daher das Sortieren aus dem Programm entfernen.

Ein Nachteil des Messens mit `time` ist die geringe Genauigkeit und die fehlende Möglichkeit das Generierens der Eingabe vom eigentlich zu messenden Sortieren zu trennen.

Alternativ zum Programm `time` kann die Zeit auch im Programm selbst gemessen werden. Dazu bietet C beispielsweise die Funktion `clock_t clock(void)` [2] an, welche die Prozessor-Zeit für den aktuellen Prozess angibt. Diese Zeit ist unabhängig von der realen Zeit und läuft mal schneller oder langsamer, je nachdem wieviel Prozessorzeit der aktuelle Prozess verwendet. Mit Hilfe dieser Funktion können wir ein Programm schreiben, welches den SelectionSort-Algorithmus mit verschiedenen Eingabegrößen testet und jeweils die Größe und Laufzeit in Millisekunden (ms) ausgibt. Wir verdoppeln hier die Eingabegröße so lange, bis ein Test 5000ms oder länger benötigt.

```
int main(int argc, char **argv) {
    int size = 1;
    double max_time = 5000;
    double time;
    do {
        int *ar = random_array(size);
        // Zeit vor Sortieren:
        clock_t start_time = clock();
        // Sortieren:
        selectionsort(ar, size);
        // Zeit nach Sortieren:
        clock_t end_time = clock();
        // Verbrauchte Zeit in ms berechnen:
        time = (end_time - start_time)*1000.0 / CLOCKS_PER_SEC;
        printf("%d, %lf\n", size, time);
        free(ar);
        size = size * 2;
    } while (time < max_time);
    return 0;
}
```

Ausführen des Programms ergibt z.B. die folgende Ausgabe:

```
1, 0.000000
2, 0.000000
4, 0.001000
8, 0.001000
16, 0.000000
32, 0.001000
64, 0.004000
128, 0.010000
256, 0.033000
512, 0.114000
1024, 0.412000
2048, 1.562000
4096, 10.494000
8192, 27.875000
16384, 102.111000
32768, 366.685000
65536, 1450.968000
131072, 5901.001000
```

[2] Siehe auch `http://en.cppreference.com/w/c/chrono/clock`. Zum Verwenden der Funktion wird der Import `#import <time.h>` benötigt.

Durch weitere Experimenten und Messungen ergeben sich die folgenden Beobachtungen:

- Die Laufzeit ist bei allen Sortieralgorithmen abhängig von der Größe N des zu sortierenden Arrays (insbesondere für große N).
- Die Laufzeit variiert leicht für *gleiche* Eingabe bei verschiedenen Durchläufen.
- Es gibt keine offensichtliche Abhängigkeit von den zu sortierenden Zahlen, insbesondere scheint der Algorithmus insensitiv in Bezug auf Vorsortierung des Arrays zu sein.
- Die Laufzeit ändert sich, wenn andere Hardware verwendet wird, der Trend bleibt aber gleich.

Typischerweise beeinflusst eine (bisweilen auch mehrere) ***Problemgröße*** die Laufzeit eines Programms. Die Laufzeit sollte mit wachsender Problemgröße zunehmen. In der Regel ist dies die Größe der Eingabe oder auch der Wert von Befehlszeilenargumenten. Im Fall von Selectionsort ist die Problemgröße die Größe N des zu sortierenden Arrays
.
Um die Frage zu beantworten, wie viel tatsächlich die Laufzeit mit wachsender Arraygröße zunimmt, wird häufig die ***Verdopplungshypothese*** verwendet.
Man erhält diese Hypothese als Antwort auf die folgende Frage:

Welche Auswirkungen hat es auf die Laufzeit,
wenn wir die Problemgröße verdoppeln?

Zum Beantworten dieser Frage betrachten wir verschiedene Messungen, mit jeweils verdoppelter Eingabegröße und die Veränderung der Laufzeit im Vergleich zum vorherigen Durchlauf.

N	Laufzeit (ms)	Veränderung
1	0	
2	0	
4	0.001	
8	0.001	
16	0	
32	0.001	
64	0.004	4.00
128	0.01	2.50
256	0.033	3.30
512	0.114	3.45
1024	0.412	3.61
2048	1.562	3.79
4096	10.494	6.72
8192	27.875	2.66
16384	102.111	3.66
32768	366.685	3.59
65536	1450.968	3.96
131072	5901.001	4.07

Wir beobachten, dass sich bei großen Eingabegrößen die Laufzeit ungefähr um den Faktor 4 erhöht, wenn sich die Eingabegröße verdoppelt.

7.2 Komplexitätsklassen

Um Algorithmen zu vergleichen, werden sie bezüglich ihrer Komplexität klassifiziert.

Klasse	Verdoppeln	Bezeichnung	Beispiel
$O(1)$	*1	*konstant*	Schaltjahrberechnung, Hashverfahren
$O(\log(N))$	-	*logarithmisch*	binäre Suche in Bäumen
$O(N)$	*2	*linear*	Mittelwert von Arrayeinträgen
$O(N \cdot \log(N))$	-	*linearithmetisch*	Mergesort
$O(N^2)$	*4	*quadratisch*	Sortieren durch Auswahl
$O(N^3)$	*8	*kubisch*	Matrixmultiplikation
$O(2^N)$	-	*exponentiell*	diverse Optimierungsverfahren

Die O-Notation zur Bezeichnung der Komplexitätsklassen wird in der Informatik verwendet, um das Verhalten bei großen Eingabegrößen abzuschätzen. Die O-Notation erlaubt eine (vergleichsweise einfache) theoretische Einordnung und Vergleich von Algorithmen. Sie liefert aber nur eine grobe Klassifizierung durch eine obere Schranke. Der Fokus bei der Verwendung der O-Notation liegt auf dem asymptotischen Verhalten bei Programmen bei großen Eingaben.
So wächst bei SelectionSort der Aufwand quadratisch mit Größe des zu sortierenden Arrays: Wenn die Größe verdoppelt wird($N \rightarrow 2 * N$), benötigt man 4-mal soviel Zeit ($N^2 \rightarrow (2N)^2 = 4N^2$).

7.3 Lösbarkeit großer Probleme

Angenommen, ein Programm benötigt für ein Problem der Größe N auf einem Rechner wenige Sekunden. Wie verhält sich die Laufzeit des Programm, wenn man die Problemgröße erhöht? Um wie viel kann ein schnellerer Rechner die Berechnungszeit verringern?
Wenn man die Problemgröße um den Faktor 100 erhöht, wächst die Laufzeit von einigen Sekunden auf ...

Klasse	Laufzeitabschätzung
linear	einige Minuten
linearithmetisch	einige Minuten
quadratisch	mehrere Stunden
kubisch	einige Wochen
exponentiell	Milliarden von Jahren

Hilft uns die Investition in besserer Hardware? Wenn man einen Computer verwendet, der um den Faktor 10 schneller rechnet, kann in der gleichen Zeit eine Probleminstanz gelöst werden, die um den Faktor ... größer ist.

Klasse	Faktor der Erhöhung der Problemgröße
linear	10
linearithmetisch	10
quadratisch	3-4
kubisch	2-3
exponentiell	1

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  }
```