0.1 Suchen und Sortieren

0.1.1 Suchen

Diese Vorlesung beschäftigt sich mit zwei fundamentalen Aufgaben der Informatik: Suchen und Sortieren. Viele der Berechnungen, die in Rechenzentren durchgeführt werden, beschäftigen sich mit diesen Problemen – dementsprechend wichtig ist es, effiziente Algorithmen zu finden.

Beispiel: Finde die Prüfung von Maria Miller in einem Stapel aus 600 Prüfungen.

Formal betrachten wir das folgende Problem:

Definition 0.1 (Suche)**.** *Gegeben seien ein Array A mit n Einträgen (bei uns: Zahlen) und ein Element b. Gesucht ist ein Index k mit $A[k] = b$, oder die Rückgabe "nicht gefunden", falls b nicht in A enthalten ist.*

Wir betrachten zwei Fälle:

- Fall 1: A ist unsortiert
- Fall 2: A ist sortiert, d.h., $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Fall 1: A ist unsortiert

In diesem Fall machen wir keinerlei Annahmen über das Array A. Der einfachste Algorithmus zur Lösung des Suchproblems in diesem Fall ist die lineare Suche:

LINEAR-SEARCH$(A = (A[1], \ldots, A[n]), b)$ — LINEARE SUCHE

```
1 for i ← 1, 2, ..., n do
2     if A[i] = b then return i
3 return "nicht gefunden"
```

Die Laufzeit der linearen Suche beträgt im schlechtesten Fall offensichtlich $\Theta(n)$, denn die Schleife wird maximal n Mal durchlaufen. Tatsächlich ist die lineare Suche optimal, d.h., im schlechtesten Fall muss *jeder* Suchalgorithmus n viele Vergleiche durchführen. Um dies zu begründen, stellen wir uns vor, dass b nicht in A enthalten ist. Nachdem der Algorithmus $n-1$ viele Elemente von A gelesen hat, kann er noch nicht entscheiden, ob b in A enthalten ist, oder nicht (denn b könnte ja im letzten, noch nicht gelesenen Element von A gespeichert sein). — LAUFZEIT

Fall 2: A ist sortiert

Wir nehmen nun an, dass A sortiert ist, d.h., $A[1] \leq A[2] \leq \cdots \leq A[n]$. Die lineare Suche funktioniert natürlich auch in diesem Fall, doch wir können die zusätzliche Information, dass A sortiert ist, ausnutzen, um deutlich schneller zu sein.

Ist A sortiert, dann können wir das Suchproblem mittels Divide-and-Conquer lösen. Dazu betrachten wir die mittlere Position $m = \lfloor n/2 \rfloor$ im Array und unterscheiden drei Fälle:

1) Ist $b = A[m]$, dann haben wir den Schlüssel b an Position m gefunden und geben daher m zurück.

2) Ist $b < A[m]$, dann suchen wir rekursiv in der linken Hälfte (also auf den Positionen $1, \ldots, m-1$) weiter nach b.

3) Ist $b > A[m]$, dann suchen wir rekursiv in der rechten Hälfte (also auf den Positionen $m+1, \ldots, n$) weiter nach b.

KORREKTHEIT

Dieses Verfahren wird *binäre Suche* genannt. Die Korrektheit der binären Suche folgt direkt aus der Tatsache, dass A aufsteigend sortiert ist. Ist also $b < A[m]$, dann wissen wir bereits, dass die Positionen $m, \ldots, n$ nur Schlüssel enthalten, die grösser als b sind. Also reicht es, die Suche auf die ersten $m-1$ Positionen einzuschränken. Die Begründung für den Fall $b > A[m]$ verläuft analog. Da der Suchbereich (die Anzahl der noch möglichen Positionen) in jedem Schritt um mindestens ein Element kleiner wird, endet das Verfahren auch nach einer endlichen Zahl von Schritten.

LAUFZEIT

Wie viele Schritte sind dies im schlimmsten Fall? Dieser tritt offenbar dann ein, wenn erfolglos nach einem Schlüssel gesucht wird. Für ein Array der Länge $n = 2^k$ ergibt sich die folgende Rekurrenz zur Abschätzung der maximal durchgeführten Operationen bei einem Array A der Länge n:

$$T(n) = \begin{cases} c & \text{falls } n = 0 \text{ ist,} \\ T(n/2) + d & \text{falls } n \geq 1 \text{ ist,} \end{cases} \tag{1}$$

wobei c und d jeweils konstant sind. Mit den früher in der Vorlesung vorgestellten Methoden erhalten wir $T(n) \in \mathcal{O}(\log n)$, was sehr schnell ist. In der Realität würde man zudem die binäre Suche iterativ statt rekursiv implementieren, was nicht schwierig ist, wie der folgende Pseudocode zeigt.

BINÄRE SUCHE

BINARY-SEARCH$(A = (A[1], \ldots, A[n]), b)$

```
1 left ← 1; right ← n                          ▷ Initialer Suchbereich
2 while left ≤ right do
3     middle ← ⌊(left+right)/2⌋
4     if A[middle] = b then return middle      ▷ Element gefunden
5     else if A[middle] > b then right ← middle−1   ▷ Suche links weiter
6     else left ← middle+1                     ▷ Suche rechts weiter
7 return "Nicht vorhanden"
```

Wie im unsortierten Fall stellt sich auch hier die Frage, ob es besser geht. Die Antwort ist wieder nein – auch wenn der Beweis etwas aufwändiger ist.

UNTERE SCHRANKE

Untere Schranke Um zu beweisen, dass die Suche in sortierten Arrays im schlimmsten Fall immer $\Omega(\log n)$ viele Vergleiche ausführt, betrachten wir wieder einen *beliebigen* vergleichsbasierten Suchalgorithmus. Wir nehmen an, dass der Algorithmus die Suche durch Vergleiche ausführt, d.h., Elemente miteinander vergleicht und aufgrund des Resultats Ja/Nein-Entscheidungen trifft.

Einen solchen Algorithmus können wir als *Entscheidungsbaum* darstellen. Jeder Knoten in diesem Baum entspricht einem Vergleich, und die beiden Kindknoten entsprechen der Entscheidung, die der Algorithmus aufgrund des Vergleichs fällt (Ja/Nein). Die Blätter des Entscheidungsbaums entsprechen den Rückgabewerten des Algorithmus. Ist eine Suche nach einem Element b erfolgreich, dann kann b an

Abb. 1 Binäre Suche auf einem Array der Länge 6 als Entscheidungsbaum visualisiert. Da sich das zu suchende Element im Erfolgsfall auf sechs möglichen Positionen befinden kann, hat der Entscheidungsbaum die gleiche Anzahl von Knoten. Im ersten Schritt wird $A[3]$ mit b verglichen. Bei Gleichheit ist der Algorithmus fertig, ansonsten werden $A[1]$ bzw. $A[5]$ verglichen, usw. Da der Baum Höhe 3 hat (also aus drei "Schichten" besteht), ist die binäre Suche auf einem Array mit sechs Elementen nach spätestens drei Vergleichen fertig.

jeder der n Positionen des Arrays stehen. Für jede dieser n möglichen Ergebnisse der Suche muss der Entscheidungsbaum mindestens einen Knoten enthalten. Zusätzlich muss es einen Knoten für "nicht gefunden" geben, also muss die Gesamtanzahl der Knoten mindestens $n+1$ betragen.

Die Anzahl von Vergleichen, die ein Algorithmus im schlechtesten Fall ausführt, entspricht exakt der *Höhe* des Baums. Diese ist definiert als die maximale Anzahl von Knoten auf einem Weg von der *Wurzel* (dem "obersten" Knoten) zu einem *Blatt* (einem Knoten ohne Nachfolger). Wir beobachten, dass ein Baum der Höhe h höchstens

$$2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = 2^h - 1 < 2^h \tag{2}$$

viele Knoten hat. Daraus folgt

$$n+1 \leq \text{Anzahl Knoten im Entscheidungsbaum} < 2^h \Rightarrow h > \log_2(n+1). \tag{3}$$

Da die Anzahl der im schlimmsten Fall ausgeführten Vergleiche h beträgt und unsere Argumentation für *beliebige* Entscheidungsbäume gilt, haben wir also bewiesen dass jedes vergleichsbasierte Suchverfahren auf einem sortierten Array im schlechtesten Fall $\Omega(\log n)$ viele Vergleiche durchführt.

0.1.2 Sortieren

Wir haben also gesehen, dass Suchen auf sortierten Daten wesentlich schneller ist als auf unsortierten Daten. Wie sortieren wir Daten effizient? Formal betrachten wir das folgende Problem:

Definition 0.2 (Sortieren)**.** *Gegeben sei ein Array A mit n Zahlen. Gesucht ist eine Permutation (Umordnung) von A, die aufsteigend sortiert ist: $A[i] \leq A[j]$ für alle $1 \leq i < j \leq n$.*

Wir betrachten die folgenden elementaren Operationen:

- Vergleiche
- Vertauschungen zweier Array-Elemente.

Bevor wir uns dem Sortierproblem zuwenden, betrachten wir zunächst das Problem, zu überprüfen, ob ein gegebenes Array bereits sortiert ist. Dies können wir wie folgt erreichen:

Prüfe Sortiertheit

```
SORTED(A)
1 for i ← 1, 2, ..., n − 1 do
2     if A[i] > A[i + 1] then return false
3 return true
```

Die Laufzeit dieses Algorithmus ist $\mathcal{O}(n)$.

Bubble Sort

Eine einfache Idee zum Sortieren eines Arrays A ist die folgende: Solange es einen Index i mit $A[i] > A[i+1]$ gibt, vertausche die Elemente $A[i]$ und $A[i+1]$. Diese Idee alleine führt jedoch noch nicht zu einem korrekten Algorithmus – wir müssen diese Prozedur mehrmals hintereinander ausführen:

Bubble Sort

BUBBLE-SORT(A)

```
1 for j ← 1, 2, ..., n do
2     for i ← 1, 2, ..., n − 1 do
3         if A[i] > A[i + 1] then
4             tausche A[i] und A[i + 1]
```

Dieser Algorithmus wird Bubble Sort genannt.

Beispiel. *Betrachten wir das Array* $A = (5, 3, 7, 1, 4)$. *Bubble Sort führt nun die folgenden Schritte durch:*

$$\begin{aligned}
j = 1: \quad & (5,3,7,1,4) \to (3,5,7,1,4) \to (3,5,7,1,4) \\
& \to (3,5,1,7,4) \to (3,5,1,4,7) \\
j = 2: \quad & (3,5,1,4,7) \to (3,1,5,4,7) \to (3,1,4,5,7) \\
& \to (3,1,4,5,7) \to (3,1,4,5,7) \\
j = 3: \quad & (3,1,4,5,7) \to (1,3,4,5,7) \to (1,3,4,5,7) \\
& \to (1,3,4,5,7) \to (1,3,4,5,7)
\end{aligned}$$

Wir beobachten, dass Bubble Sort in diesem Fall bereits nach drei Durchläufen das Array sortiert hat, obwohl $n = 5$ *ist.*

Die Laufzeit von Bubble Sort ist:

- $\Theta(n^2)$ Vergleiche
- $\mathcal{O}(n^2)$ Vertauschungen.

Um zu begründen, dass Bubble Sort tatsächlich korrekt ist, betrachten wir die folgende Invariante:

> Nach j Durchläufen der äußeren Schleife sind die j größten Elemente am richtigen Ort.

Formaler ausgedrückt gilt also

$$A[n-j+1] \leq A[n-j+2] \leq \cdots \leq A[n],$$

und die Elemente $A[n-j+1], \ldots, A[n]$ werden auch in Zukunft nicht mehr verändert.

Für $j = 1$ ist dies leicht einzusehen: Das größte Element "blubbert" in jedem Schritt der inneren Schleife ein Element weiter nach rechts, bis es die letzte Position im Array erreicht hat.

Induktiv folgt die Invariante für $j+1$ aus der Invariante für j: Die j größten Elemente befinden sich bereits am richtigen Ort, und das $(j+1)$-größte Element "blubbert" während des $(j+1)$-ten Durchlaufs der äußeren Schleife an die richtige Stelle.

Am Ende gilt die Invariante für $j = n$. Also sind die n größten Elemente am richtigen Ort, d.h., das Array ist sortiert.

Selection Sort

Wir betrachten nun eine andere Invariante:

> Nach j Durchläufen der äußeren Schleife sind die Elemente $A[n-j+1], \ldots, A[n]$ sortiert (sie enthalten aber nicht zwangsläufig die j größten Elemente des Arrays).

Formaler ausgedrückt gilt also

$$A[n-j+1] \leq A[n-j+2] \leq \cdots \leq A[n],$$

doch die Elemente an diesen Positionen können sich in Zukunft noch verändern.

Wie erreichen wir die Invariante für $j+1$, falls die Invariante für j bereits gilt? Dazu müssen wir lediglich das größte Element im Teilarray $A[1, \ldots, n-j]$ finden, und dieses mit $A[n-j]$ vertauschen.

Dies führt zum folgenden Algorithmus, der Selection Sort genannt wird:

SELECTION-SORT(A) — SELECTION SORT

```
1 for j ← n − 1, n − 2, . . . , 0 do
2     k ← Index des Maximums in A[1, . . . , n − j]
3     tausche A[k] und A[n − j]
```

Beispiel. *Betrachten wir das Array $A = (5, 3, 7, 1, 4)$. Selection Sort führt nun die folgenden Schritte durch:*

$$\begin{aligned}
j = 4: &\quad (5,3,7,1,4) \to (5,3,4,1,7)\\
j = 3: &\quad (5,3,4,1,7) \to (1,3,4,5,7)\\
j = 2: &\quad (1,3,4,5,7) \to (1,3,4,5,7)\\
j = 1: &\quad (1,3,4,5,7) \to (1,3,4,5,7)\\
j = 0: &\quad (1,3,4,5,7) \to (1,3,4,5,7)
\end{aligned}$$

Wir beobachten, dass Selection Sort in diesem Fall bereits nach zwei Durchläufen das Array sortiert hat.

Die Laufzeit von Selection Sort ist:

- $\Theta(n^2)$ Vergleiche
- $\Theta(n)$ Vertauschungen.

Insertion Sort

Wir betrachten nun eine weitere Invariante:

> Nach j Durchläufen der äußeren Schleife ist das Teilarray $A[1, \ldots, j]$ sortiert (es enthält aber nicht zwangsläufig die j kleinsten Elemente des Arrays).

Formaler ausgedrückt gilt also

$$A[1] \leq A[2] \leq \cdots \leq A[j],$$

doch die Elemente an diesen Positionen können sich in Zukunft noch verändern.

Wie erreichen wir die Invariante für $j+1$, falls die Invariante für j bereits gilt? Dazu müssen wir das Element $A[j+1]$ an der richtigen Stelle im Teilarray $A[1, \ldots, j]$ einfügen. Dies erreichen wir, indem wir $A[j+1]$ so lange mit dem Element links davon vertauschen, bis es an der richtigen Stelle steht.

Dies führt zum folgenden Algorithmus, der Insertion Sort genannt wird:

Insertion Sort

INSERTION-SORT(A)

```
1  for j ← 2, 3, ..., n do
2      k ← kleinster Index in {1, ..., j−1} sodass A[j] ≤ A[k]   ▷ A[j] gehört an diese Stelle k
3      x ← A[j]                                                  ▷ merke A[j], da es gleich überschrieben wird
4      verschiebe A[k, ..., j − 1] nach A[k + 1, ..., j]         ▷ aufwendig, braucht j − k Operationen (worst case: j)
5      A[k] ← x
```

Beispiel. *Betrachten wir das Array $A = (5, 3, 7, 1, 4)$. Insertion Sort führt nun die folgenden Schritte durch:*

$$\begin{aligned} j = 2: &\quad (5,3,7,1,4) \to (3,5,7,1,4) \\ j = 3: &\quad (3,5,7,1,4) \to (3,5,7,1,4) \\ j = 4: &\quad (3,5,7,1,4) \to (1,3,5,7,4) \\ j = 5: &\quad (1,3,5,7,4) \to (1,3,4,5,7) \end{aligned}$$

Wir beobachten, dass Insertion Sort in diesem Fall vier Durchläufe benötigt.

Die Laufzeit von Insertion Sort ist:

- $\mathcal{O}(n^2)$ Vergleiche
- $\mathcal{O}(n^2)$ Vertauschungen.

Die Korrektheit von Insertion Sort folgt direkt aus der oben beschriebenen Invariante: Nach n Durchläufen ist das gesamte Array sortiert.

Merge Sort

Die drei bisher vorgestellten Sortieralgorithmen benötigen im schlechtesten Fall $\Omega(n^2)$ viele Operationen. Können wir schneller sein?

Alle drei Algorithmen betrachten immer nur ein Element des Arrays gleichzeitig. Selection Sort findet in jedem Schritt das größte Element des noch unsortierten Teilarrays und fügt es an der richtigen Stelle ein. Insertion Sort betrachtet in jedem Schritt ein neues Element und fügt es an der richtigen Stelle im bereits sortierten Teilarray ein. Bubble Sort betrachtet in jedem Schritt zwei benachbarte Elemente und vertauscht sie, falls sie nicht in der richtigen Reihenfolge sind.

Können wir durch die Verwendung von Divide-and-Conquer schneller sein? Wir könnten etwa das Array in zwei Hälften teilen, die beiden Hälften rekursiv sortieren, und die beiden sortierten Hälften dann zu einem sortierten Array zusammenfügen.

Diese Idee führt zum Algorithmus Merge Sort:

MERGE-SORT(A, l, r) — MERGE SORT

```
1
                                  ▷ sortiert den Bereich A[l, ..., r]
2  if l < r then
3      m ← ⌊(l + r)/2⌋
4      MERGE-SORT(A, l, m)        ▷ sortiere linke Hälfte
5      MERGE-SORT(A, m + 1, r)    ▷ sortiere rechte Hälfte
6      MERGE(A, l, m, r)          ▷ verschmelze beide Hälften
```

Der Basisfall dieser Rekursion ist $l = r$. In diesem Fall besteht das zu sortierende Teilarray $A[l, \ldots, r]$ nur aus einem einzigen Element, ist also bereits sortiert.

Es bleibt noch zu beschreiben, wie die Funktion MERGE funktioniert. Sie erhält als Eingabe ein Array A und drei Indizes l, m und r, sodass die Teilarrays $A[l, \ldots, m]$ und $A[m+1, \ldots, r]$ bereits sortiert sind. MERGE soll diese beiden Teilarrays zu einem sortierten Teilarray $A[l, \ldots, r]$ verschmelzen.

MERGE(A, l, m, r) — MERGE

```
1  B ← new Array with r − l + 1 cells   ▷ same size as A[l, ..., r]
2  i ← l                                ▷ erstes unbenutztes Element in linker Hälfte
3  j ← m + 1                            ▷ erstes unbenutztes Element in rechter Hälfte
4  k ← 1                                ▷ nächste Position in B
5  while i ≤ m and j ≤ r do             ▷ beide Hälften noch nicht ausgeschöpft
6      if A[i] < A[j] then
7          B[k] ← A[i]
8          i ← i + 1
9          k ← k + 1
10     else
11         B[k] ← A[j]
```

```
12        j ← j + 1
13        k ← k + 1
14  übernimm Rest links bzw. rechts        ▷ wenn die andere Hälfte ausgeschöpft ist
15  kopiere B nach A[l, ..., r]
```

Beispiel. *Betrachten wir das Array* $A = (9, 7, 3, 2, 1, 8, 4, 6)$ *und die Indizes* $l = 1$, $m = 4$ *und* $r = 8$. *Die Teilarrays* $A[1, \ldots, 4] = (9, 7, 3, 2)$ *und* $A[5, \ldots, 8] = (1, 8, 4, 6)$ *sind bereits sortiert.*

Merge Sort führt nun die folgenden Schritte durch:

$$(9, 7, 3, 2,\ 1, 8, 4, 6) \to (2, 3, 7, 9,\ 1, 4, 6, 8)$$
$$\to (1, 2, 3, 4, 6, 7, 8, 9)$$

Die Laufzeit von Merge ist $\mathcal{O}(n)$, da jedes Element von A nur einmal gelesen und einmal geschrieben wird.

Die Laufzeit von Merge Sort lässt sich durch die folgende Rekursionsgleichung beschreiben:

$$T(1) = c,$$
$$T(n) \leq 2 \cdot T(n/2) + dn \quad \text{für } n = 2^k.$$

Dabei ist c die Laufzeit für den Basisfall und dn die Laufzeit von Merge.

Mittels Induktion lässt sich zeigen, dass

$$T(n) = dn \log_2 n + cn \in \mathcal{O}(n \log n)$$

gilt. Merge Sort ist also asymptotisch schneller als die drei zuvor vorgestellten Algorithmen.

Der Nachteil von Merge Sort ist, dass der Algorithmus ein zusätzliches Array B benötigt. Ein Sortieralgorithmus, der kein zusätzliches Array benötigt, wird *in-place*-Algorithmus genannt. Bubble Sort, Selection Sort und Insertion Sort sind in-place-Algorithmen, Merge Sort hingegen nicht.

Alternative Bestimmung der Laufzeit Alternativ lässt sich die Laufzeit von Merge Sort auch wie folgt bestimmen: Jeder Aufruf von MERGE benötigt $\leq n$ Vergleiche und $2n$ Kopieroperationen. Da die Rekursion von Merge Sort $\log_2 n$ Level hat (für $n = 2^k$), ist die Gesamtlaufzeit

$$\underbrace{\log_2 n}_{\text{Level}} \cdot \underbrace{n}_{\text{Vergleiche pro Level}} \in \mathcal{O}(n \log n).$$

0.2 Quicksort

0.2.1 Motivation

In den vorherigen Vorlesungen haben wir verschiedene Sortieralgorithmen kennengelernt, darunter Bubble Sort, Selection Sort, Insertion Sort und Merge Sort. Die Tabelle **??** fasst die Laufzeiten und Eigenschaften dieser Algorithmen zusammen.

Algorithmus	Vergleiche	Bewegungen	Extr. Platz	Lokalität
Bubble Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	gut
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	gut
Insertion Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	gut
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	schlecht

Tabelle 1 Übersicht über die Laufzeiten und Eigenschaften verschiedener Sortieralgorithmen. *Bewegungen* bezeichnet die Anzahl der Schreiboperationen im Speicher. *Extr. Platz* gibt an, wie viel zusätzlichen Speicherplatz der jeweilige Algorithmus benötigt. *Lokalität* beschreibt, ob der Algorithmus im Speicher sequentiell arbeitet, oder ob er im Speicher hin und her springt.

Merge Sort zeichnet sich durch eine Laufzeit von $\mathcal{O}(n \log n)$ aus, benötigt allerdings zusätzlichen Speicherplatz der Grössenordnung $\mathcal{O}(n)$.

Die Grundidee von Merge Sort besteht darin, das zu sortierende Array in zwei Hälften aufzuteilen, die beiden Hälften rekursiv zu sortieren, und anschliessend die beiden sortierten Teillisten in einem Schritt zusammenzuführen (engl. *merge*). Die eigentliche Arbeit fällt beim Zusammenführen an, da dabei die Elemente der beiden Teillisten miteinander verglichen und in die richtige Reihenfolge gebracht werden müssen.

0.2.2 Grundidee

Quicksort verfolgt einen ähnlichen Ansatz wie Merge Sort, allerdings wird die Reihenfolge der Schritte umgekehrt: Anstatt die Teillisten erst zu sortieren und anschliessend zusammenzuführen, werden die Elemente des Arrays zunächst in zwei Gruppen aufgeteilt, so dass die Elemente der einen Gruppe kleiner sind als die Elemente der anderen Gruppe. Anschliessend werden die beiden Gruppen rekursiv sortiert.

Der Schlüssel zu dieser Idee ist die Wahl eines sogenannten *Pivotelements*. Alle Elemente, die kleiner als das Pivotelement sind, kommen in die linke Gruppe; alle Elemente, die grösser sind als das Pivotelement, kommen in die rechte Gruppe. Nachdem die Elemente in zwei Gruppen aufgeteilt wurden, muss das Pivotelement noch an die richtige Stelle zwischen den beiden Gruppen verschoben werden.

0.2.3 Der Algorithmus

Quicksort lässt sich rekursiv beschreiben. Der Algorithmus erhält als Eingabe ein Array A und zwei Indizes l und r, die den zu sortierenden Bereich festlegen. Die Aufgabe besteht darin, den Bereich von $A[l]$ bis $A[r]$ aufsteigend zu sortieren.

QUICKSORT(A, l, r) QUICKSORT

```
1 if l < r then
2     k ← Aufteilen(A, l, r)      ▷ Teile A[l..r] in zwei Gruppen auf
3     Quicksort(A, l, k − 1)      ▷ Sortiere linke Gruppe
4     Quicksort(A, k + 1, r)      ▷ Sortiere rechte Gruppe
```

Die Routine Aufteilen(A, l, r) führt die folgenden Schritte aus:

1. Wähle ein Element p als Pivotelement. In unserem Fall wählen wir das letzte Element des zu sortierenden Bereichs: $p \leftarrow A[r]$.
2. Bestimme die korrekte Position k des Pivotelements p im Array A.
3. Verschiebe alle Elemente, die kleiner oder gleich p sind, nach links (in den Bereich $A[l..k-1]$) und alle Elemente, die grösser als p sind, nach rechts (in den Bereich $A[k+1..r]$).

AUFTEILEN

AUFTEILEN(A, l, r)

```
1  i ← l                                  ▷ Index für die linke Gruppe
2  j ← r − 1                              ▷ Index für die rechte Gruppe
3  p ← A[r]                               ▷ Pivotelement
4  repeat
5      while i < r and A[i] ≤ p do
6          i ← i + 1                      ▷ Suche nächstes Element für rechte Gruppe
7      while j > l and A[j] > p do
8          j ← j − 1                      ▷ Suche nächstes Element für linke Gruppe
9      if i < j then
10         Vertausche A[i] und A[j]       ▷ Vertausche Elemente
11 until i > j                            ▷ Schleife endet, wenn sich i und j "treffen"
12 Vertausche A[i] und A[r]               ▷ Pivotelement an die richtige Stelle
13 return i                               ▷ Gib Position des Pivotelements zurück
```

Die Routine Aufteilen arbeitet mit zwei Indizes i und j, die von links bzw. rechts durch den zu sortierenden Bereich laufen. Die While-Schleifen suchen nach Elementen, die sich auf der falschen Seite des Pivotelements befinden. Werden zwei solche Elemente gefunden, werden sie vertauscht. Die Repeat-Schleife wird so lange ausgeführt, bis die Indizes i und j aneinander vorbeigelaufen sind. Am Ende der Routine befindet sich das Pivotelement an der korrekten Position $k = i$. Links davon stehen alle Elemente, die kleiner oder gleich dem Pivotelement sind; rechts davon stehen alle Elemente, die grösser als das Pivotelement sind.

0.2.4 Laufzeit von Quicksort

Die Laufzeit von Quicksort hängt davon ab, an welcher Position das Pivotelement landet. Im besten Fall landet das Pivotelement genau in der Mitte des zu sortierenden Bereichs. In diesem Fall teilt sich das Problem in zwei gleich grosse Teilprobleme auf, und wir erhalten die folgende Rekurrenz für die Laufzeit:

$$T(n) \overset{\text{I0}}{=} 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n.$$

Abb. 2 Beispiel für einen Max-Heap.

Dabei bezeichnet $c \cdot n$ die Laufzeit der Routine Aufteilen. Aus dieser Rekurrenz ergibt sich eine Laufzeit von $\mathcal{O}(n \log n)$.

Im schlechtesten Fall landet das Pivotelement am Rand des zu sortierenden Bereichs. Dies ist beispielsweise der Fall, wenn das Array bereits sortiert ist und wir das letzte Element als Pivotelement wählen. In diesem Fall erhalten wir die folgende Rekurrenz:

$$T(n) = T(n-1) + c \cdot n.$$

Diese Rekurrenz führt zu einer Laufzeit von $\mathcal{O}(n^2)$. In der Praxis ist Quicksort jedoch trotz der quadratischen Laufzeit im schlechtesten Fall einer der am häufigsten verwendeten Sortieralgorithmen. Der Grund dafür ist, dass die Laufzeit von Quicksort im Durchschnitt $\mathcal{O}(n \log n)$ beträgt, sofern das Pivotelement zufällig gewählt wird.

0.3 Heapsort

0.3.1 Motivation

Selection Sort basiert auf der Idee, in jedem Schritt das grösste Element des Arrays zu finden und an die richtige Stelle zu verschieben. Die Invariante des Algorithmus ist, dass der rechte Teil des Arrays bereits korrekt sortiert ist.

Der Flaschenhals von Selection Sort ist die Suche nach dem grössten Element, die im i-ten Schritt Zeit $\mathcal{O}(i)$ benötigt. Dies führt zu einer Gesamtlaufzeit von $\mathcal{O}(n^2)$.

Die Frage ist nun, ob sich diese Suche nach dem grössten Element beschleunigen lässt. Sofern die Daten in einem unsortierten Array gespeichert sind, ist die Antwort nein: Wir müssen jedes Element des Arrays betrachten, um das grösste Element zu finden.

Die Idee von Heapsort besteht nun darin, die Daten nicht in einem Array, sondern in einer speziellen Baumstruktur, einem sogenannten *Heap*, zu organisieren. Diese Struktur erlaubt es uns, das grösste Element effizient zu finden und zu entfernen.

0.3.2 Max-Heap

Ein *Max-Heap* ist ein binärer Baum, der die folgenden beiden Bedingungen erfüllt:

1. *Vollständiger Binärbaum*: Jeder Knoten hat genau zwei Kinder, ausser eventuell im untersten Level des Baums. Das unterste Level ist von links nach rechts aufgefüllt.

2. *Heap-Bedingung*: Der Schlüssel jedes Knotens ist grösser oder gleich den Schlüsseln seiner Kinder.

Abbildung **??** zeigt ein Beispiel für einen Max-Heap.

Terminologie

In einem Baum werden die Knoten über *Kanten* miteinander verbunden.

- *Elternknoten, Kindknoten*: Ein Knoten, der direkt über einem anderen Knoten liegt, wird als *Elternknoten* bezeichnet; der darunterliegende Knoten ist der *Kindknoten*.
- *Vorfahren, Nachkommen*: Ein Knoten v ist ein *Vorfahr* eines Knotens w, wenn es einen Pfad von v nach w gibt, der nur abwärts verläuft. In diesem Fall ist w ein *Nachkomme* von v.
- *Wurzel*: Der Knoten, der ganz oben im Baum steht und keinen Elternknoten hat, ist die *Wurzel* des Baums.
- *Blätter*: Die Knoten, die keine Kinder haben, sind die *Blätter* des Baums.

Eigenschaften des Max-Heaps

Aus der Heap-Bedingung folgt direkt, dass das grösste Element eines Max-Heaps stets in der Wurzel des Baums steht. Ausserdem gilt: Der Schlüssel eines beliebigen Vorfahren ist grösser oder gleich dem Schlüssel eines beliebigen Nachkommen.

0.3.3 Heapsort

Um Heapsort zu implementieren, benötigen wir zwei Operationen auf Max-Heaps:

1. *Daten in Heap umwandeln*: Gegeben ein Array A mit n Elementen, konstruiere einen Max-Heap, dessen Knoten die Elemente aus A enthalten.
2. *Maximum aus Heap löschen (*Extract Max*)*: Finde das grösste Element des Heaps (also den Schlüssel der Wurzel), entferne es aus dem Heap, und stelle die Heap-Bedingung wieder her.

Sobald wir diese beiden Operationen implementiert haben, lässt sich Heapsort wie folgt beschreiben:

Heapsort

HEAPSORT(A)

```
1 H ← empty Heap                          ▷ Erzeuge leeren Heap
2 for i ← 1, 2, ..., n do                 ▷ Füge Elemente in Heap ein
3     insert(H, A[i])
4 for i ← n, n − 1, ..., 1 do             ▷ Entferne Elemente aus Heap
5     A[i] ← Extract Max(H)
```

0.3.4 Maximum aus Heap löschen (Extract Max)

Die Operation Extract Max besteht aus zwei Schritten:

1. Entferne die Wurzel des Heaps. Dadurch entsteht eine Lücke an der Wurzelposition, und die Baumstruktur des Heaps wird verletzt.

Abb. 3 Beispiel für die Operation Extract Max.

Abb. 4 Beispiel für die Operation insert.

2. Verschiebe das letzte Blatt des Heaps an die Wurzelposition. Dadurch wird die Baumstruktur wieder hergestellt, allerdings ist die Heap-Bedingung möglicherweise verletzt. Stelle die Heap-Bedingung wieder her, indem du den neuen Wurzelknoten so lange mit dem grösseren seiner beiden Kinder vertauschst, bis die Heap-Bedingung erfüllt ist.

Abbildung **??** veranschaulicht die Operation Extract Max anhand eines Beispiels.

Laufzeit von Extract Max

Die Laufzeit von Extract Max wird von der Anzahl der Vertauschungen dominiert, die notwendig sind, um die Heap-Bedingung wiederherzustellen. Im schlimmsten Fall müssen wir den neuen Wurzelknoten bis zum untersten Level des Baums verschieben. Die Anzahl der Vertauschungen ist daher proportional zur *Tiefe* des Baums.

Die Tiefe eines vollständigen Binärbaums mit n Knoten ist $\mathcal{O}(\log n)$. Folglich benötigt Extract Max Zeit $\mathcal{O}(\log n)$.

0.3.5 Daten in Heap umwandeln

Es gibt verschiedene Möglichkeiten, ein Array in einen Heap umzuwandeln. Eine einfache Methode besteht darin, mit einem leeren Heap zu beginnen und die Elemente des Arrays nacheinander in den Heap einzufügen.

Einfügen in Heap (*insert*)

Die Operation insert(H, p) fügt einen neuen Knoten mit Schlüssel p in den Heap H ein. Sie besteht aus zwei Schritten:

1. Füge den neuen Knoten an der nächsten freien Stelle im Heap ein. Dadurch wird die Baumstruktur des Heaps erhalten.

2. Stelle die Heap-Bedingung wieder her, indem du den neuen Knoten so lange mit seinem Elternknoten vertauschst, bis die Heap-Bedingung erfüllt ist.

Abbildung **??** veranschaulicht die Operation insert anhand eines Beispiels.

Laufzeit von insert

Die Laufzeit von insert wird von der Anzahl der Vertauschungen dominiert, die notwendig sind, um die Heap-Bedingung wiederherzustellen. Im schlimmsten Fall müssen wir den neuen Knoten bis zur Wurzel des Baums verschieben. Die Anzahl der Vertauschungen ist daher proportional zur *Tiefe* des Baums.

Wie bereits erwähnt, ist die Tiefe eines vollständigen Binärbaums mit n Knoten $\mathcal{O}(\log n)$. Folglich benötigt insert Zeit $\mathcal{O}(\log n)$.

Abb. 5 Darstellung des Heaps aus Abbildung **??** in einem Array.

Laufzeit des Umwandelns

Um ein Array mit n Elementen in einen Heap umzuwandeln, müssen wir n mal die Operation insert ausführen. Da insert Zeit $\mathcal{O}(\log n)$ benötigt, beträgt die Gesamtlaufzeit des Umwandelns $\mathcal{O}(n \log n)$.

0.3.6 Laufzeitanalyse von Heapsort

Heapsort besteht aus den folgenden Schritten:

1. Umwandeln des Arrays in einen Heap: $\mathcal{O}(n \log n)$
2. n-maliges Ausführen von Extract Max: $n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Die Gesamtlaufzeit von Heapsort beträgt daher $\mathcal{O}(n \log n)$.

0.3.7 Darstellung eines Binärbaums im Speicher

Bisher haben wir uns nicht darum gekümmert, wie ein Binärbaum (und damit auch ein Heap) im Speicher eines Computers dargestellt wird. Eine effiziente Möglichkeit besteht darin, die Knoten des Baums in einem Array zu speichern.

Die Knoten werden Level für Level von links nach rechts im Array abgelegt. Die Wurzel des Baums steht an Position 1 des Arrays. Die Kinder des Knotens an Position k stehen an den Positionen $2k$ und $2k + 1$.

Abbildung **??** zeigt die Darstellung des Heaps aus Abbildung **??** in einem Array.

Mit dieser Darstellung lassen sich alle Heap-Operationen direkt im Array implementieren. Beispielsweise lässt sich die Heap-Bedingung für den Knoten an Position k wie folgt testen:

Test der Heap-Bedingung

SatisfiesHeapCondition(k)

```
1 if 2k > n then return true                          ▷ A[k] ist ein Blatt
2 if 2k = n and A[2k] ≤ A[k] then return true         ▷ A[k] hat nur ein Kind
3 if 2k < n and A[2k] ≤ A[k] and A[2k + 1] ≤ A[k] then return true
    ▷ A[k] hat zwei Kinder
4 return false
```

0.3.8 Vorteile und Nachteile von Heapsort

Heapsort bietet gegenüber anderen Sortieralgorithmen wie Merge Sort und Quicksort einige Vor- und Nachteile:

Vorteile

- *Laufzeit* $\mathcal{O}(n \log n)$: Heapsort hat die gleiche asymptotische Laufzeit wie Merge Sort und Quicksort.
- *In-place*: Heapsort benötigt keinen zusätzlichen Speicherplatz (ausser einer konstanten Anzahl von Hilfsvariablen).

Abb. 6 Beispiel für einen Entscheidungsbaum eines vergleichsbasierten Sortieralgorithmus.

Nachteile

- *Schlechte Lokalität*: Heapsort springt im Speicher hin und her, um auf die verschiedenen Knoten des Heaps zuzugreifen. Dies kann die Performance des Algorithmus negativ beeinflussen, da der Zugriff auf benachbarte Speicherzellen in der Regel schneller ist als der Zugriff auf weit voneinander entfernte Speicherzellen.

In der Praxis wird Heapsort daher etwas seltener verwendet als Merge Sort und Quicksort.

0.4 Untere Schranke für vergleichsbasiertes Sortieren

Wir haben nun drei Sortieralgorithmen kennengelernt – Merge Sort, Quicksort und Heapsort –, die alle eine Laufzeit von $\mathcal{O}(n \log n)$ erreichen. Die Frage ist nun, ob es einen Sortieralgorithmus gibt, der eine noch bessere Laufzeit erreicht.

Unter der Annahme, dass der Sortieralgorithmus ausschliesslich auf *Vergleichen* von Schlüsseln basiert, ist die Antwort nein. Jeder vergleichsbasierte Sortieralgorithmus benötigt im schlechtesten Fall mindestens $\Omega(n \log n)$ viele Vergleiche.

0.4.1 Beweis

Um diese Aussage zu beweisen, betrachten wir einen beliebigen vergleichsbasierten Sortieralgorithmus und stellen ihn als *Entscheidungsbaum* dar.

Jeder Knoten des Entscheidungsbaums repräsentiert einen Vergleich von zwei Schlüsseln. Die beiden Kanten, die von einem Knoten ausgehen, entsprechen den beiden möglichen Ergebnissen des Vergleichs (kleiner oder grösser).

Die Blätter des Entscheidungsbaums entsprechen den möglichen *Permutationen* des Eingabearrays. Eine Permutation ist eine bestimmte Anordnung der Elemente des Arrays. Beispielsweise sind $(2, 1, 3)$ und $(3, 1, 2)$ zwei verschiedene Permutationen des Arrays $(1, 2, 3)$.

Die Anzahl der Vergleiche, die der Algorithmus im schlechtesten Fall ausführt, entspricht der *Höhe* des Entscheidungsbaums, also der maximalen Anzahl von Knoten auf einem Pfad von der Wurzel zu einem Blatt.

Anzahl der Permutationen

Ein Array mit n Elementen hat $n!$ verschiedene Permutationen. Dies lässt sich wie folgt begründen:

Für das erste Element des Arrays gibt es n mögliche Positionen. Für das zweite Element gibt es dann nur noch $n - 1$ mögliche Positionen, da eine Position bereits belegt ist. Für das dritte Element gibt es $n - 2$ mögliche Positionen, und so weiter. Insgesamt erhalten wir $n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 = n!$ mögliche Anordnungen.

Höhe des Entscheidungsbaums

Da der Entscheidungsbaum für jede mögliche Permutation des Eingabearrays mindestens ein Blatt enthalten muss, hat er mindestens $n!$ Blätter.

Jeder Knoten des Entscheidungsbaums hat maximal zwei Kinder. Folglich hat ein Baum der Höhe h maximal 2^h Blätter.

Aus diesen beiden Tatsachen folgt:

$$\begin{aligned} & 2^h \geq n! \\ \Rightarrow \quad & h \geq \log_2(n!) \end{aligned}$$

Mit Hilfe der Stirling-Formel lässt sich zeigen, dass $\log_2(n!) \in \Omega(n \log n)$ gilt.

Folglich benötigt jeder vergleichsbasierte Sortieralgorithmus im schlechtesten Fall mindestens $\Omega(n \log n)$ viele Vergleiche.

0.5 Datenstrukturen

In der letzten Vorlesung haben wir uns mit dem Problem des Suchens in Arrays beschäftigt und verschiedene Algorithmen zur Sortierung von Arrays kennengelernt. Dabei haben wir festgestellt, dass die Effizienz der Suche stark davon abhängt, ob die Daten bereits sortiert sind.

0.5.1 Abstrakte Datentypen und Datenstrukturen

Sortieren lässt sich als ein Spezialfall eines wichtigeren Themas auffassen, nämlich der Organisation von Daten im Speicher, um effizient darauf zugreifen und sie manipulieren zu können. In diesem Zusammenhang spielen *Datenstrukturen* eine zentrale Rolle. Eine Datenstruktur ist eine bestimmte Art und Weise, Daten im Speicher eines Computers zu repräsentieren.

Mit Datenstrukturen eng verknüpft ist der Begriff des *abstrakten Datentyps* (ADT). Ein ADT beschreibt, *was* wir mit einer Menge von Daten tun wollen, ohne im Detail festzulegen, *wie* dies genau geschehen soll. Mit anderen Worten: Ein ADT definiert die Operationen, die auf einer Menge von Daten ausgeführt werden können, ohne die konkrete Implementierung dieser Operationen vorzuschreiben.

Man kann sich einen ADT als eine Art *Wunschliste* vorstellen, die festlegt, welche Funktionalitäten eine Datenstruktur bereitstellen soll. Die Datenstruktur selbst ist dann die *Implementierung* dieser Wunschliste. Es ist durchaus möglich, dass es für einen gegebenen ADT mehrere unterschiedliche Implementierungen gibt, die jeweils verschiedene Vor- und Nachteile haben.

0.5.2 Beispiel: ADT Liste

Als Beispiel betrachten wir den abstrakten Datentyp *Liste*. Eine Liste ist eine geordnete Sammlung von Objekten (Schlüsseln), bei der die Reihenfolge der Elemente relevant ist. Formaler: Eine Liste $L = (k_1, \ldots, k_n)$ enthält n Schlüssel in der Reihenfolge $k_1, \ldots, k_n$. Typische Operationen, die auf Listen definiert werden, sind:

- insert(K, L): Fügt den Schlüssel K am Ende der Liste L ein.
- get(i, L): Gibt den i-ten Schlüssel in L zurück.

- delete(o, L): Entfernt das Objekt o aus der Liste L.
- insertAfter(o, K, L): Fügt den Schlüssel K hinter dem Objekt o in die Liste L ein.

Implementierungen für den ADT Liste

Es gibt verschiedene Möglichkeiten, den ADT Liste zu implementieren. Zwei häufig verwendete Ansätze sind Arrays und verkettete Listen.

Arrays Sofern die maximale Länge der Liste bekannt ist, lässt sich der ADT Liste mittels eines Arrays implementieren. Dazu wird ein Array fester Grösse angelegt, dessen Elemente die Schlüssel der Liste speichern. Die Elemente der Liste belegen die ersten Zellen des Arrays; der restliche Teil des Arrays enthält keine relevanten Daten.

Verkettete Listen Eine *verkettete Liste* (linked list) ist eine dynamische Datenstruktur, bei der die Elemente nicht in aufeinanderfolgenden Speicherzellen abgelegt werden, sondern über *Zeiger* (Pointer) miteinander verbunden sind. Jedes Element einer verketteten Liste besteht aus dem Schlüssel und einem Zeiger, der auf das nächste Element der Liste verweist. Der letzte Zeiger der Liste hat den Wert *null*, um anzuzeigen, dass keine weiteren Elemente folgen.

In der Praxis wird oft ein zusätzlicher Zeiger verwendet, der auf das erste Element der Liste zeigt. Ausserdem gibt es auch *doppelt verkettete Listen*, bei denen jedes Element zusätzlich einen Zeiger auf den Vorgänger enthält.