7 Sortieren

7.1 Überblick

In diesem Kapitel werden wir uns intensiv mit der Theorie des *Sortierens* auseinandersetzen. Wir werden verschiedene Algorithmen kennenlernen, wichtige Problemlösestrategien diskutieren und sogar eine untere Schranke für die Laufzeit *aller* vergleichsbasierten Sortieralgorithmen beweisen.

Nach einer (etwas älteren) Schätzung von **IBM** werden ca. 25% der Rechenzeit in typischen kommerziellen Anwendungen für das Sortieren aufgewandt.

Einige wichtige Anwendungen sind z.B.:

- Zusammenfassen „verwandter"Dinge (z.B. Sortieren einer Buchdatenbank nach Autoren)
- Suche von Duplikaten
- Beschleunigen wichtiger Suchalgorithmen
- Auswerten von DNA-Arrays in der Bioinformatik

7.2 Das Sortierproblem

Wir beginnen mit der allgemeinen Form des Sortierproblems.

Definition 7.1 (Sortierproblem). Gegeben sei eine Menge A von Elementen aus einem Universum U, sowie eine Ordnungsrelation „$\preceq$" auf U. Gesucht ist eine Anordnung der Elemente von A in aufsteigender (oder fallender) Reihenfolge bezüglich der Ordnungsrelation $\preceq$.

Beispiele:

- $U =$ Englische Wörter, $\preceq$ = lexikographische Ordnung
- $U =$ Studenten der Universität des Saarlandes,
  $x \preceq y \Leftrightarrow$ Matrikelnummer$(x) \leq$ Matrikelnummer(y)

Um die Notation zu vereinfachen, wollen wir uns im Folgenden darauf beschränken, ganze oder reelle Zahlen zu sortieren, also $U = \mathbb{R}$, " $\preceq$ " $\equiv$ " $\leq$ ".

7.2.1 Ein triviales Sortierverfahren

Definition von TrivialSort

Ein offensichtlicher Algorithmus zum Sortieren eines Feldes a_i funktioniert folgendermassen:

```
TrivialSort:

Gegeben ein Array A = {A[0]...A[n-1]}.

(1) i = 0
(2) Solange i < n-1:
(3)   Suche das minimale Element m in {A[i]...A[n-1]}
(4)   Vertausche A[i] und m
(5)   Erhoehe i um 1
(6)   Zurueck zu Zeile 1
```

In C++ können wir dies für `integer`–Zahlen z.B. wie folgt implementieren:

```
1  /** Sucht im Teilfeld A[min_index]...A[max_index] nach dem
2      minimalen Element A[i].
3      Liefert den Index i dieses Elements zurueck.
4   */
5  int index_of_minimum(vector<int>& a, int min_index, int max_index)
6  {
7    int index = min_index;
8    for (int i=min_index; i < max_index; i++)
9    {
10     if (a[index] > a[i]) index = i;
11   }
12
13   return index;
14 }
15
16 /** Sortiert einen vector<int> aufsteigend.
17  */
18 void trivial_sort(vector<int>& a)
19 {
20   for (int i=0; i<a.size(); i++)
21   {
22     int j = index_of_minimum(a, i+1, a.size());
23     int k = a[i];
24     if (k>a[j])
25     {
26       a[i] = a[j];
```

```
27            a[j] = k;
28        }
29    }
30 }
```

Laufzeit von TrivialSort

Analysieren wir nun die Laufzeit von TrivialSort.
Offenbar benötigt die Funktion `index_of_minimum` $\Theta($`max_index-min_index`$)$ Schritte. Da `index_of_minimum` in der `for`-Schleife (Zeile 21–29) im i-ten Schritt mit `max_index = a.size() = n` und `min_index = i+1` aufgerufen wird, benötigt jeder Durchlauf der `for`-Schleife (Zeile 21–29) $\Theta(n-i)$ Schritte. Also ergibt sich für die Laufzeit von *TrivialSort*:

Satz 7.1 (Laufzeit von TrivialSort). *Die Laufzeit von TrivialSort liegt in* $\Theta(n^2)$.

Beweis. Für die Laufzeit von TrivialSort gilt nach den obigen Überlegungen mit einer Konstanten $c > 0$:

$$\begin{aligned} T_{\text{TS}}(n) &= c\sum_{i=0}^{n-1}(n-i) \\ &= c\left(\sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i\right) \\ &= c\left(n^2 - \frac{n^2-n}{2}\right) \\ &= c\frac{n^2+n}{2} \\ &\in \Theta(n^2) \end{aligned}$$

■

Diese hohe Laufzeit lässt sich damit erklären, dass unser hier gewählte Ansatz zur Lösung des Sortierproblems die folgende Form hat:

Problem A(n) = Konkatenation(Problem B(n), Problem A(n-1))

wobei in unserem Fall das Problem B(n) im Suchen des Minimums von n Zahlen besteht. Ein solches Schema führt zu der folgenden Rekursion für die Laufzeit $T_A(n)$ von Problem A:

$$T_A(n) = \begin{cases} d & n = 1 \\ c + T_B(n) + T_A(n-1) & n > 1 \end{cases}$$

wobei d eine positive Konstante ist.
In unserem Fall hat Problem B(n) – die Suche nach dem Minimum von n Zahlen – die worstcase Laufzeit $\Theta(n)$, und daher gilt für die obige Rekursion:

$$T_A(n) = \begin{cases} d & n = 1 \\ c_1 + c_2 n + T_A(n-1) & n > 1 \end{cases}$$

mit positiven Konstanten d, c_1, c_2.
Wir haben also die Lösung eines Problems der Größe n auf die Lösung eines Problems der Größe $n-1$ zurückgeführt! Häufig ist es allerdings sinnvoll, die Lösung des „*grossen*" Problems auf die Lösung mehrerer wesentlich kleinerer Teilprobleme zurückzuführen.

7.2.2 Divide & Conquer – Verfahren

Die obigen Überlegungen führen auf die sogenannte **Divide & Conquer**–Strategie:

Definition 7.2 (Divide&Conquer). Zerlege das ursprüngliche Problem der Größe n in kleinere Teilprobleme. Löse diese Teilprobleme und setze die Gesamtlösung aus den Lösungen der Teilprobleme zusammen.

Häufig lässt sich ein Problem der Größe n z.B. in zwei Teilprobleme der Größe $\frac{n}{2}$ zerlegen.

Problem A(n) = Konkatenation(Problem A(n/2), Problem A(n/2))

7.2.3 MergeSort

Ein Sortieralgorithmus, der auf dem oben angegebenen Schema basiert, ist *MergeSort*.

Definition von MergeSort

```
MergeSort:

Gegeben ein Array A = {A[0]...A[n-1]}.

(1) Zerlege A in zwei ungefähr gleich grosse Hälften A1, A2 der Länge
n/2
(2) Sortiere A1 und A2 rekursiv mit MergeSort
(3) Mische die sortierten Halbfelder zu einem sortierten Feld und gib
    dieses als Ergebnis zurück
```

Implementieren wir zunächst die Funktion `merge` in C++:

```
1  /** Mischt die sortierten Teilfelder {A[s1],...,A[e1]}
2      und {A[e1+1],...,A[e2]} zu einem sortierten Feld
3      {A[s1],...,A[e2]}.
4   */
5  void merge(vector<int>& a, int s1, int e1, int e2)
6  {
7    vector<int> b(e2-s1+1); // Hilfsfeld zum Mischen
8    int i=s1, j=e1+1, k=0;
9
10   for (; i<=e1 && j<=e2; k++)
```

```
11    {
12      if (a[i] <= a[j]) {b[k] = a[i]; i++;}
13      else              {b[k] = a[j]; j++;}
14    }
15
16    // Leere den Rest des ersten Teilfelds
17    for (;i<=e1;k++,i++) b[k]=a[i];
18
19    // Leere den Rest des zweiten Teilfelds
20    for (;j<=e2;k++,j++) b[k]=a[j];
21
22    // Kopiere das Ergebnis zurueck nach A
23    for (k=0,i=s1; i<=e2; i++,k++) a[i]=b[k];
24  }
```

Satz 7.2 (Korrektheit und Laufzeit von merge). *Die Funktion* `merge` *sortiert zwei schon sortierte Teilfelder* $A[s1], \ldots, A[e1]$ *und* $A[e1+1], \ldots, A[e2]$ *der Länge* $\frac{n}{2}$ *korrekt in Zeit* $\Theta(n)$.

Beweis. Zu zeigen haben wir Korrektheit und Laufzeit:

- Korrektheit: die beiden Teilfelder $A[s1], \ldots, A[e1]$ und $A[e1+1], \ldots, A[e2]$ sind bereits korrekt sortiert. Daher sind die Indizes i und j die Indizes der jeweils kleinsten noch nicht in b einsortierten Elemente der beiden Teilfelder. Also ist $\min(A[i], A[j])$ offenbar die kleinste noch nicht verarbeitete Zahl in $A[s1, \ldots, e2]$, und muss daher an die Stelle $b[k]$ einsortiert werden.

- Laufzeit: trivial ersichtlich aus den Definitionen der `for`–Schleifen

■

Damit können wir nun die Funktion `merge_sort` implementieren:

```
1  void merge_sort(vector<int>& a, int min, int max)
2  {
3    if (max–min > 0) // sind wir schon fertig?
4    {
5      merge_sort(a, min, min+((max–min)/2));
6      merge_sort(a, min+((max–min)/2)+1, max);
7      merge(a, min, min+((max–min)/2), max);
8    }
9  }
```

Laufzeit von MergeSort

Satz 7.3 (Laufzeit von MergeSort). *MergeSort sortiert ein Array der Größe n in Zeit $\Theta(n \log n)$.*

Beweis. Für die Laufzeit der Funktion `merge_sort` erhalten wir mit Satz 7.2 die folgende Rekursion:

$$T_{\text{MS}}(n) = \begin{cases} c & n = 1 \\ 2\,T_{\text{MS}}\left(\frac{n}{2}\right) + dn & n > 1 \end{cases}$$

Da $\log_2(2) = 1$, und damit $dn \in \Theta(n^{\log_2(2)}) = \Theta(n)$, können wir Fall (2) des Mastertheorems 4.1 anwenden, und es folgt $T_{\text{MS}}(n) \in \Theta(n \log_2(n))$. ■

7.2.4 HeapSort

Ein weiteres, häufig verwendetes Sortierverfahren ist *HeapSort*. Wir werden feststellen, dass auch *HeapSort* in Zeit $\mathcal{O}(n \log_2(n))$ läuft, also asymptotisch gesehen eigentlich keine Verbesserung gegenüber *MergeSort* darstellt. Allerdings sortiert *HeapSort "in-place"*, d.h. es wird kein zusätzliches Hilfsfeld benötigt, in das die Daten beim Sortieren kopiert werden. Daher läuft *HeapSort* in vielen praktischen Anwendungen beträchtlich schneller als *MergeSort*. Der *HeapSort* Algorithmus basiert auf einer speziellen Datenstruktur, dem sogenannten *Heap*.

Definition 7.3 (Heap). Ein binärer *MinHeap* ist ein binärer Baum mit den folgenden Eigenschaften:

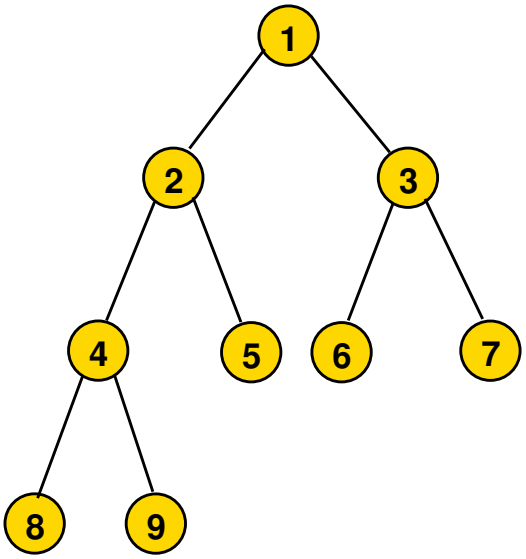
1. Für jeden Knoten `v` des Heaps mit einem Knoten `v.parent != NULL` gilt: `v.key` $\geq$ `v.parent->key` (*Heapeigenschaft*)

2. Der Heap ist – abgesehen von der untersten Ebene – immer vollständig gefüllt

Für die Definition eines binären *MaxHeaps* wird in Punkt 1. „$\geq$" durch „$\leq$" ersetzt.

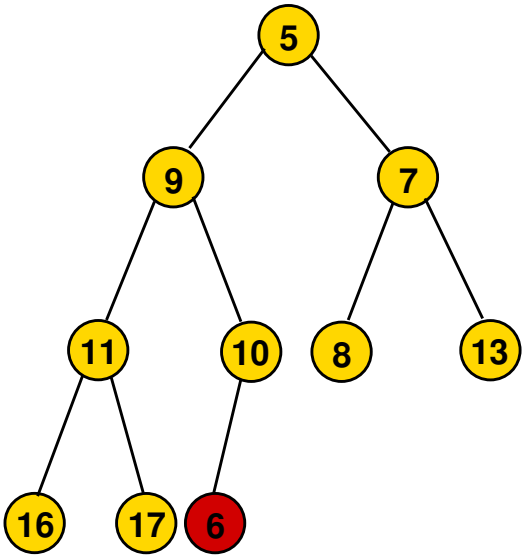
Anmerkung 7.1. Der Einfachheit halber werden wir in diesem Kapitel den *binären MinHeap* kurz als *Heap* bezeichnen.

Beispiel 7.1 (MinHeap).



Wie können wir Elemente in einen Heap einfügen? Wenn wir das neue Element einfach an die nächste freie Position im Heap anhängen, kann es passieren, dass wir die Heapeigenschaft verletzen!

Beispiel 7.2 (Einfügen in einen MinHeap).

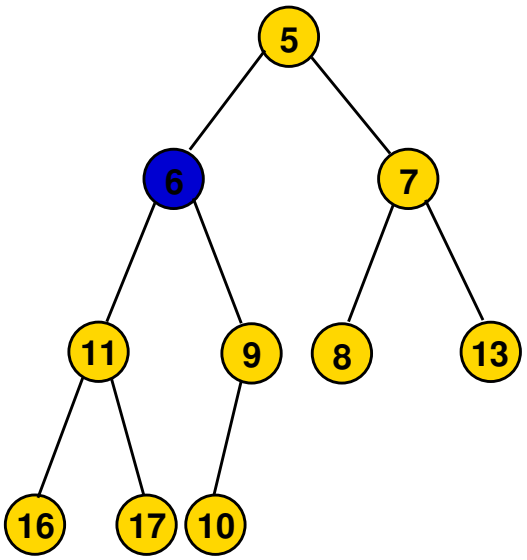


Wir müssen also nach dem *Einhängen* des Knotens den Heap *reparieren*! Dazu nehmen wir an, dass `v` ein Zeiger auf den eingefügten Knoten ist. Die Reparatur erfolgt nun, indem wir den soeben eingefügten Knoten so lange "*nach oben schieben*" indem wir jeweils Vater und Sohn miteinander vertauschen, bis die Heapeigenschaft wiederhergestellt ist:

```
(1) Solange v->key < v->parent->key:
(2)   Vertausche Vater und Sohn:
(3)     int help = v->key;
```

```
(3)     v->key = v->parent->key;
(4)     v->parent->key = help;
(5)     v = v->parent
```

Beispiel 7.3 (Reparatur des MinHeaps aus Beispiel 7.2).



Damit können wir die Funktion `heap_insert` in Pseudocode folgendermaßen definieren:

```
heap_insert:
(1) Hänge den Knoten an die erste freie Position im Heap
(2) Repariere die Heapeigenschaft
```

Satz 7.4 (Laufzeit von `heap_insert`). `heap_insert` *benötigt für das Einfügen eines Knotens in einen Heap mit* n *Elementen* $\mathcal{O}(\log(n))$ *Operationen.*

Beweis. Da der Heap ein – eventuell bis auf die letzte Ebene – vollständiger binärer Baum ist, beträgt die Höhe des Heaps zu jedem Zeitpunkt $\log(n)$. Da bei jeder Einfügeoperation im schlimmsten Fall der Pfad des neu eingefügten Knotens bis zur Wurzel hochgewandert werden muss, und da jede einzelne Knotenvertauschung während der Reparatur in konstanter Zeit abläuft, benötigt `heap_insert` im worst-case $\mathcal{O}(\log(n))$ Operationen. ■

Offenbar können wir einen Heap H aus einem Array A aufbauen, indem wir mit einem leeren Heap starten, und dann nacheinander die Elemente von A einfügen:

```
build_heap:
(1) H = ∅
(2) Solange A nicht leer:
(3)   füge das erste Element von A per heap_insert in H ein
(4)   lösche dieses Element aus A
```

Satz 7.5 (Laufzeit von `build_heap`). `build_heap` *benötigt* $\mathcal{O}(n \log(n))$ *Schritte zum Aufbau eines Heaps aus einem Array der Länge* n*.*

Beweis. Die Einfügeoperation im i-ten Schritt läuft in $\mathcal{O}(\log(i)) \subset \mathcal{O}(\log(n))$ Operationen. Da wir n Einfügeoperationen durchführen, erhalten wir eine Laufzeit von $\mathcal{O}(n \log(n))$. ■

HeapSort

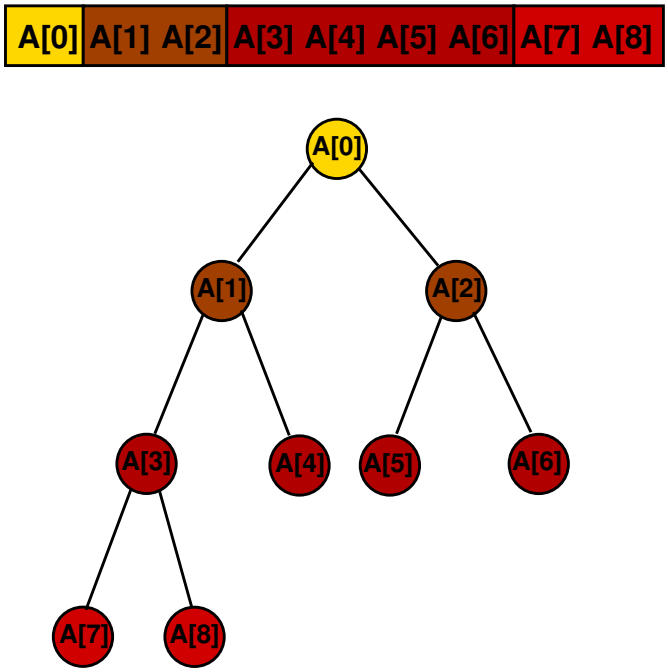
Mit Hilfe eines Heaps H können wir nun die in ihm gespeicherten Elemente wie folgt in ein Feld A sortieren:

```
heap_sort:
(1) A = ∅
(2) Solange H ≠ ∅:
(3)   Entnehme das Element in der Wurzel (das kleinste im Heap)
(4)   Hänge dieses Element an das Feld der bereits sortierten
      Elemente A an
(5)   Verschiebe das zuletzt eingefügte Element in die Wurzel
(6)   Schiebe die neue Wurzel solange nach unten, bis die Heap-
      eigenschaft wiederhergestellt ist
```

Bevor wir uns überlegen, wie wir `heap_sort` genau implementieren können, müssen wir uns zunächst überlegen, wie wir einen Heap möglichst einfach repräsentieren können. Da es sich beim Heap um einen – bis auf die letzte Ebene – vollständigen Binärbaum handelt, lässt er sich sehr einfach und effizient in einem Array simulieren, indem einfach nacheinander die verschiedenen Ebenen des Heaps von links nach rechts im Array abgelegt werden. Dann gilt für einen Knoten mit Index i:

Index des linken Kindes: $2i + 1$, Index des rechten Kindes: $2i + 2$

Beispiel 7.4 (Simulation eines Heaps als Array).



Wie bewegt man nun ein in die Wurzel verschobenes Element auf seinen korrekten Platz im Heap?

```
1  /** Schiebt einen Knoten mit Index i nach unten auf seinen
2      Platz im Heap
3   */
4  void shift_down(vector<int>& a, int i, int last)
5  {
6    bool finished = false;
7    int k;
8
9    while (!finished)
10   {
11     // Hat der betrachtete Knoten noch 2 Kinder?
12     if ((2*i+2) < last)
13     {
14       // Wenn ja, finde den Index des groesseren Kindes
15       if (a[2*i+2] < a[2*i+1]) k = 2*i + 2;
16       else                     k = 2*i + 1;
17     }
18     else
19     {
20       // Hat der Knoten genau ein Kind?
21       if ((2*i+1) < last) k = 2*i + 1; // ja
22       else                return;      // nein->wir sind fertig
23     }
```

```
24
25      // ist die Heapeigenschaft noch verletzt?
26      if (a[i] > a[k])
27      {
28        // ja->schiebe Knoten nach unten
29        int help = a[i];
30        a[i] = a[k];
31        a[k] = help;
32        i = k;
33      }
34      else
35      {
36        // nein->wir sind fertig
37        finished = true;
38      }
39    }
40  }
```

Satz 7.6 (Laufzeit von `shift_down`). `shift_down` *benötigt im worst case* $\mathcal{O}(\log(n))$ *Operationen.*

Beweis. Es wird maximal ein Pfad von der Wurzel bis zu einem Blatt durchlaufen. In jedem Schritt werden zum Vertauschen der Knoten nur konstant viele Operationen benötigt. Da der Heap Höhe $\mathcal{O}(\log(n))$ hat, folgt die Behauptung. ■

Mit der oben definierten Funktion `shift_down` ist es sehr einfach, ein gegebenes Array A so umzusortieren, dass es einen Heap repräsentiert:

```
1  /** Sortiert die Elemente von a so um, dass sie einen
2      Heap simulieren.
3   */
4  void heapify(vector<int>& a)
5  {
6    for (int i=a.size()-1; i>= 0; i--)
7      shift_down(a, i, a.size());
8  }
```

`heapify` erstellt den Heap indem Schicht für Schicht die Heapeigenschaft hergestellt wird. Dabei wird auf der Ebene der Blätter begonnen.

Satz 7.7 (Laufzeit von `heapify`). *Die Funktion* `heapify` *sortiert ein Array* A *der Länge* n *in* $\mathcal{O}(n\log(n))$ *Schritten zu einem Heap um.*

Beweis. `heapify` ruft offenbar n-mal die Funktion `shift_down` auf. Diese läuft nach Satz 7.6 in $\mathcal{O}(\log(n))$. Damit folgt die Behauptung. ■

Anmerkung 7.2. Die Funktion `heapify` arbeitet *in-place*, d.h. das Array A wird direkt umsortiert, ohne dass ein zusätzliches Hilfsfeld benötigt wird.

Anmerkung 7.3. Offenbar können die *Blätter* nicht weiter nach unten durchgereicht werden. Daher könnte in der Funktion `heapify` die `for`–Schleife durch eine effizientere ersetzt werden, die erst auf der von unten gesehen zweiten Ebene beginnt.

Implementierung von HeapSort

Damit haben wir nun alles beisammen, um *HeapSort* implementieren zu können. Da wir uns in diesem Kapitel immer auf *MinHeaps* bezogen haben, liefert die unten angegebene Implementierung eine *absteigende* Sortierung des Arrays. Zum *aufsteigenden* Sortieren kann man statt dessen *MaxHeaps* verwenden, auf die sich alle hier erzielten Resultate sehr einfach übertragen lassen.

```
1  /** Sortiert den vector a absteigend.
2   */
3  void heap_sort(vector<int>& a)
4  {
5    // Zunaechst bauen wir den Heap
6    heapify(a);
7
8    for (int i=a.size()-1; i>0; i--)
9    {
10     // Wir retten den Schluessel von a
11     int help = a[i];
12
13     // Die Wurzel ist das kleinste noch nicht sortierte
14     // Element, daher koennen wir sie zu den bereits
15     // sortierten Elementen packen.
16     a[i] = a[0];
17
18     // den alten Wert von a[i] stecken wir nun in die
19     // Wurzel
20     a[0] = help;
21
22     // Und reparieren die Heapeigenschaft
23     shift_down(a, 0, i);
24   }
25 }
```

Satz 7.8 (Laufzeit von `heap_sort`). `heap_sort` *sortiert ein Array A der Länge n in Zeit $\mathcal{O}(n \log(n))$.*

Beweis. Nach Satz 7.7 läuft `heapify` in Zeit $\mathcal{O}(n \log(n))$. In jedem Schleifendurchlauf wird einmal `shift_down` aufgerufen, was nach Satz 7.6 Zeit $\mathcal{O}(\log(n))$ benötigt. Da die Schleife $n-1$-mal ausgeführt wird, ergibt sich die behauptete Laufzeit von $\mathcal{O}(n \log(n))$. ■

7.2.5 QuickSort

HeapSort ist zwar ein äusserst schneller Algorithmus mit optimaler asymptotischer worst-case Laufzeit (wie wir später beweisen werden). In der Praxis ist es jedoch meist möglich, noch schneller zu sortieren. Dazu wird ein Algorithmus eingesetzt, dessen *worst-case* Laufzeit zwar wesentlich schlechter ist als die von *HeapSort*, dessen *erwartete* Laufzeit allerdings auch in $\Theta(n \log(n))$ liegt, wobei allerdings die in der $\mathcal{O}$-Notation "*versteckten*" Konstanten wesentlich kleiner sind als im Fall von *HeapSort*. Dadurch wird in vielen tatsächlich auftretenden Anwendungsfällen ein wesentlich schnelleres Sortieren ermöglicht.

Die Idee von QuickSort

QuickSort ist – ähnlich *MergeSort* – ein **Divide&Conquer**–Algorithmus. Allerdings wird hierbei das Array nicht in zwei gleich große Hälften zerlegt. Statt dessen wählt man sich – z.B. zufällig – ein Element e aus dem Array A (das sogenannte *Pivotelement*) und vergleicht alle anderen Elemente von A mit e. Damit zerlegt man dann das Array $A \setminus \{e\}$ in zwei Teilmengen $A_{\leq}, A_{>}$ mit der Eigenschaft:

$$\forall x \in A_{\leq} : x \leq e$$
$$\forall x \in A_{>} : x > e$$

Dann führt man die gleiche Prozedur rekursiv für die Arrays $A_{\leq}, A_{>}$ durch. Daher können wir mit *QuickSort* ein Array folgendermaßen sortieren:

$$\boxed{\text{QS}(A) = \text{Konkatenation}(\text{QS}(A_{\leq}), e, \text{QS}(A_{>}))}$$

Da die Definition der Mengen $A_{\leq}$ und $A_{>}$ von der Wahl von e abhängt, ist natürlich auch die Zahl der Elemente in $A_{\leq}$ und $A_{>}$, und damit auch die Laufzeit von *QuickSort*, e-abhängig. Im worst-case ist in jedem Schritt eine der beiden Teilmengen leer, so dass der Algorithmus die gleiche Komplexität wie *TrivialSort* – nämlich $\Theta(n^2)$ – aufweist (es wird nämlich ein Problem der Größe n auf ein Problem der Größe $n-1$ zurückgeführt).
Schafft man es allerdings, durch die Wahl von e in jedem Schritt ungefähr gleich mächtige Mengen zu produzieren, dann gilt für die Laufzeit:

$$\boxed{\text{QS}(n) = 2\,\text{QS}(\tfrac{n}{2}) + cn = \Theta(n \log(n))}$$

Implementierung von QuickSort

In Pseudocode können wir *QuickSort* folgendermassen angeben:

```
QuickSort(A,l,r)

(1) falls (l < r)
(2)   q = partition(A,l,r)
(3)   QuickSort(A,l,q-1)
(4)   QuickSort(A,q+1,r)
```

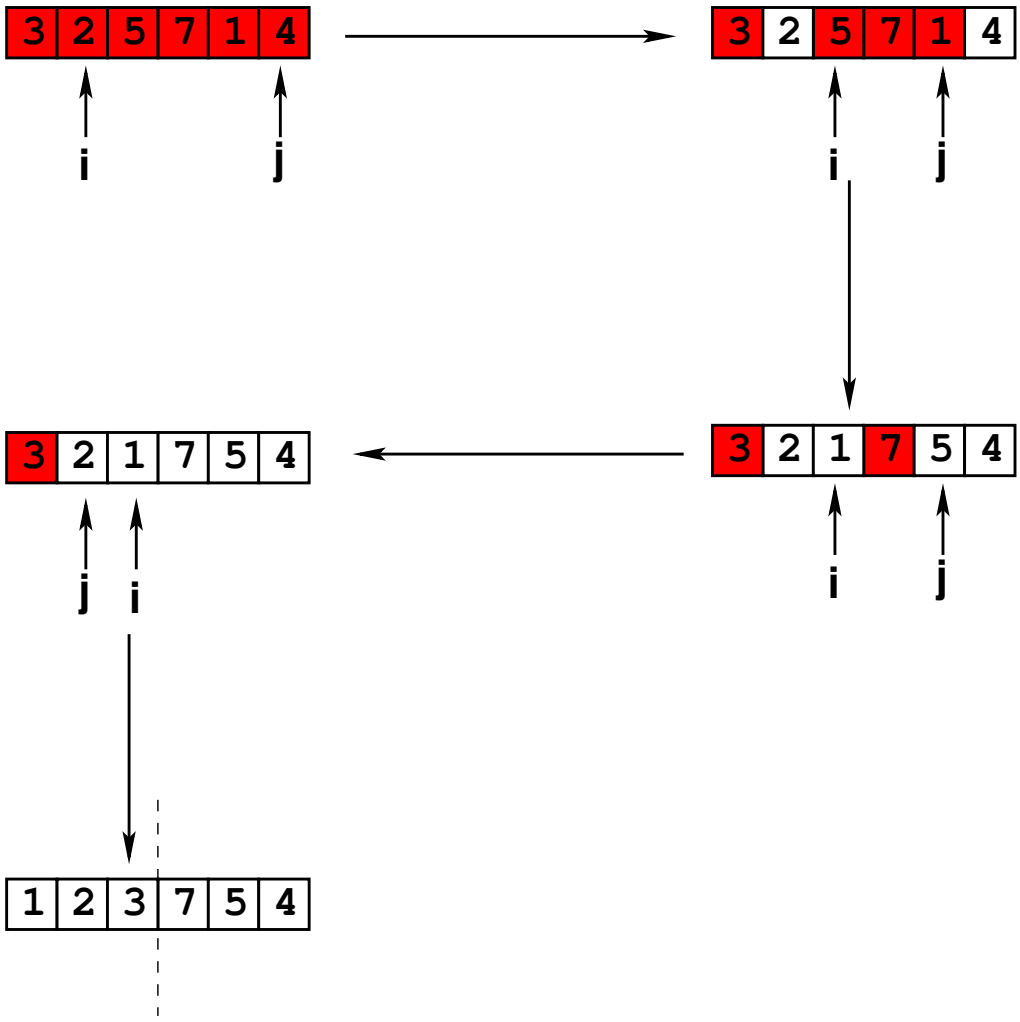
dabei berechnet die Funktion `partition` die Zerlegung in die Teilmengen $A_{\leq}$ und $A_{>}$, d.h. A wird so umsortiert, dass gilt:

$$A = \{ \underbrace{A_{\leq}}_{A_0,\cdots,A_{q-1}}, \quad e, \quad \underbrace{A_{>}}_{A_{q+1},\cdots,A_{n-1}} \}$$

Um die Menge A in die gewünschten Teilmengen zu zerlegen, gehen wir mit zwei Zeigern i, j von rechts und links im Array A aufeinander zu. Dabei bewegen wir jeweils den rechten Zeiger j einen Schritt nach links und überprüfen, ob das Element auf das aktuell verwiesen wird, die gewünschte Bedingung verletzt (d.h. $A[j] \leq e$). Falls nein, gehen wir einen weiteren Schritt nach links. Falls ja, bewegen wir den linken Zeiger nach rechts, bis wir auch für die linke Hälfte ein Element finden, dass die Bedingung verletzt (d.h. $A[i] > e$). Nun müssen wir zwei Fälle unterscheiden: befindet sich der rechte Zeiger nun links vom linken Zeiger (d.h. $j \leq i$), dann haben wir eine erlaubte Partition erzeugt und müssen nur noch das Pivotelement $e = A[0]$ mit dem Element an der Stelle $A[q]$ vertauschen. Andernfalls müssen wir die beiden gefunden Elemente einfach vertauschen und können weiter nach unerlaubten Paaren suchen.

Dieses Verhalten lässt sich am einfachsten anhand eines Beispiels nachvollziehen: (wir wählen hier als Pivotelement $A[0]$)

Beispiel 7.5 (Ablauf von `partition`).



Anmerkung 7.4. Da nach Ablauf von `partition` alle Elemente in der linken Hälfte kleiner oder gleich dem Pivotelement sind und alle Elemente in der rechten Hälfte grösser, können wir jetzt die beiden Teilarrays getrennt voneinander betrachten, wobei das Pivotelement schon seinen endgültigen Platz im sortierten Array angenommen hat, also in allen weiteren Schritten nicht mehr betrachtet werden muss.

Wenn wir nun *QuickSort* implementieren wollen, müssen wir uns zuvor noch überlegen, wie wir in jedem Schritt das zum Vergleichen herangezogene Pivotelement $e \in A$ wählen können. Offensichtlich ist die erwartete Laufzeit von *QuickSort* – wie in den Überlegungen des letzten Abschnittes gezeigt – stark von der Wahl von e abhängig.

Um die Implementierung möglichst einfach zu halten, entscheiden wir uns dafür, in jedem Rekursionsschritt das jeweils *erste* Element des aktuell zu sortierenden Teilarrays zu verwenden. Mit dieser einfachen *Pivotstrategie* können wir nun schliesslich *QuickSort* implementieren:

```
1 /** Diese Funktion uebernimmt das Partitionieren
2  *  in Teilarrays V1 und V2, so dass alle Elemente
3  *  von V1 kleiner als alle Elemente von V2 sind.
```

```
4   */
5  unsigned int partition(vector<int>& v, int p, int q)
6  {
7    // Wir nehmen das erste Element als Pivotelement
8    int pivot = v[p];
9
10   // die beiden Zeiger
11   int links  = p;
12   int rechts = q;
13
14   // das eigentliche Vertauschen
15   while (links < rechts)
16   {
17     if (v[rechts] > pivot) { rechts--; };
18     else
19     {
20       if (v[links] <= pivot) { links++; }
21       else
22       {
23         int tmp   = v[links];
24         v[links]  = v[rechts];
25         v[rechts] = tmp;
26       }
27     }
28   }
29
30   // Jetzt muss nur noch das aktuell betrachtete Element
31   // mit dem Pivot vertauscht werden
32   v[p]      = v[rechts];
33   v[rechts] = pivot;
34
35   // und die richtige Position zurueckgegeben werden
36   return rechts;
37 }
38
39 /** Diese Funktion implementiert den Quicksort-
40  *  Algorithmus fuer vector<int>.
41  */
42 void quick_sort(vector<int>& v, int p, int q)
43 {
44   // sind wir vielleicht schon fertig? oder ist
45   // unterwegs was seltsames passiert?
46   if ( p < q )
47   {
```

```
48      // na gut, wir muessen doch noch was machen
49      unsigned int r = partition(v, p, q);
50      quick_sort(v, p, r-1);
51      quick_sort(v, r+1, q);
52    }
53  }
```

Laufzeit von QuickSort

Anmerkung 7.5. Im folgenden Abschnitt werden wir zur Vereinfachung der Beweise annehmen, dass alle Elemente von A paarweise verschieden sind, d.h. dass für alle Paare $i \neq j, i = 1 \dots n, j = 1 \dots n$ gilt: $a_i \neq a_j$.

Offenbar ist die Laufzeit von *QuickSort* proportional der Anzahl der Vergleiche unterschiedlicher Feldelemente.

Lemma 7.1. *Jedes Paar $A[i], A[j]$ im Array A wird höchstens einmal miteinander verglichen!*

Beweis. Alle vorkommenden Vergleiche sind Vergleiche mit dem Pivotelement. Da das Pivotelement von Schritt i in Schritt $i + 1$ in den rekursiven Aufrufen von `quick_sort` gar nicht mehr auftaucht, folgt die Behauptung. ■

Es sei $Z = \{z_1, \dots, z_n\}$ das Ergebnis der Sortierung von A, d.h. es gilt $z_i \in A \forall i$, und $z_1 < z_2 < \dots < z_n$. Mit Z_{ij} bezeichnen wir die Menge $Z_{ij} := \{z_i, \dots, z_j\}$.

Um die *erwartete Laufzeit* von *QuickSort* zu bestimmen, führen wir nun die folgenden Zufalls(Indikator-)variablen ein:

$$X_{ij} = \begin{cases} 1 & \text{falls } z_i \text{ und } z_j \text{ miteinander verglichen werden} \\ 0 & \text{sonst} \end{cases}$$

Um die Anzahl an Vergleichen zu bestimmen, müssen wir über i und j summieren:

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

Anmerkung 7.6. Die zweite Summe beginnt bei $j = i+1$, da ansonsten jeder Vergleich doppelt gezählt würde.

Da die Laufzeit von *QuickSort* nach der obigen Anmerkung proportional zur Anzahl an Vergleichen ist, können wir die erwartete Laufzeit mit Hilfe des *Erwartungswertes* der Ver-

gleichsanzahl berechnen! Für diese erwartete Vergleichsanzahl gilt:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} p(\{X_{ij}=1\}). \end{aligned}$$

Dabei ist nach dem oben Gesagten $p(\{X_{ij}=1\})$ die Wahrscheinlichkeit dafür, dass Element z_i mit Element z_j verglichen wird.[1]

Im Verlauf des Algorithmus wählen wir immer neue Pivotelemente aus der Menge A. Dabei wird in irgendeinem Schritt t_k zum ersten Mal ein Element aus der Menge $Z_{ij} = \{Z_i, \ldots, Z_j\}$ ausgewählt werden.[2] Obwohl nun in allen vorherigen Schritten die ursprüngliche Menge A rekursiv in viele unabhängig voneinander zu betrachtende Teilmengen unterteilt wurde, macht man sich leicht klar, dass alle Elemente von Z_{ij} zum Zeitpunkt t_k noch in die selbe Teilmenge eingeordnet wurden:

Lemma 7.2. *Zum Zeitpunkt t_k liegen alle Elemente der Menge Z_{ij} in der selben noch rekursiv zu zerteilenden Teilmenge $A_i \subset A$.*

Beweis. Alle in den Schritten $t_1 \ldots t_{k-1}$ ausgewählten Pivotelemente y stammen entweder aus der Menge $Z_{1\ldots i-1}$ oder aus der Menge $Z_{j+1\ldots n-1}$. Daher sind alle Elemente von Z_{ij} entweder grösser als y – dann landen sie in der "rechten" Teilmenge – oder kleiner als y – dann landen sie in der "linken" Teilmenge. Insgesamt werden also in jedem vorherigen Schritt t_1 bis t_{k-1} alle Elemente aus Z_{ij} der selben Menge zugeordnet. ■

Nun können wir zwei mögliche Fälle für x unterscheiden:

1. $x = z_i$ oder $x = z_j$
2. $((x \neq z_i) \wedge (x \neq z_j)) \Rightarrow x \in Z_{i+1,j-1}$

Da nach Lemma 7.2 alle Elemente aus Z_{ij} in der selben noch zu sortierenden Teilmenge liegen, werden sie auch alle mit dem gewählten Pivotelement x verglichen. Also gilt in Fall 1. $X_{ij} = 1$, denn da in diesem Fall entweder z_i oder z_j das Pivotelement ist, wird das z_j bzw. z_i mit ihm verglichen. In Fall 2. hingegen gilt $X_{ij} = 0$, denn da in diesem Fall $z_i < x$ und $z_j > x$

[1] Aus dieser Formel können wir auch sehr einfach die worst-case Laufzeit herleiten: da im schlimmsten Fall jedes Element mit jedem verglichen wird, gilt $T_{\text{QS}}^{\text{worst\_case}} = \mathcal{O}(n^2)$

[2] Man beachte, dass wir *nicht* wissen, welches Element aus Z_{ij} als Pivotelement ausgewählt wird, obwohl wir uns für eine Pivotstrategie entschieden haben. Dies liegt daran, dass wir unsere Pivotstrategie natürlich auf die Menge A anwenden, und wir für ein gegebenes Element aus A vor dem Sortieren natürlich noch nicht wissen, an welcher Position es in Z stehen wird!

gilt, landen z_i und z_j in unterschiedlichen Teilmengen, so dass sie in keinem späteren Schritt $t_{k+1}, \ldots$ miteinander verglichen werden können. Damit können wir nun folgendes Lemma beweisen:

Lemma 7.3.

$$p(\{X_{ij} = 1\}) = \frac{2}{j - 1 + 1}$$

Beweis. Es gilt:

$$\begin{aligned} p(\{X_{ij} = 1\}) &= p(z_i \text{ wird mit } z_j \text{ verglichen}) \\ &= p(z_i \text{ oder } z_j \text{ wird als erstes Pivotelement aus } Z_{ij} \text{ gewählt}) \\ &= p(z_i \text{ wird als erstes Pivotelement aus } Z_{ij} \text{ gewählt}) \\ &\quad + p(z_j \text{ wird als erstes Pivotelement aus } Z_{ij} \text{ gewählt}) \\ &=: p_i + p_j \end{aligned}$$

Da jedes Element aus Z_{ij} mit gleicher Wahrscheinlichkeit als erstes aus Z_{ij} gezogen wird, gilt:

$$p_i = p_j = \frac{1}{|Z_{ij}|} = \frac{1}{j - i + 1}$$

und damit

$$p(\{X_{ij} = 1\}) = p_i + p_j = 2\,p_i = \frac{2}{j - i + 1}.$$

■

Damit können wir nun endlich $E[X]$ berechnen:

Satz 7.9 (Erwartete Laufzeit von *QuickSort*). *Die erwartete Laufzeit von* QuickSort $E[X]$ *liegt in* $\mathcal{O}(n \log(n))$.

Beweis.

$$\begin{aligned} E[X] &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} p(\{X_{ij} = 1\}) \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\ &= \sum_{i=1}^{n} \sum_{k=1}^{n-i} \frac{2}{k + 1} \\ &< \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k} \\ &< \sum_{i=1}^{n} c \log(n) \\ &= c \sum_{i=1}^{n} \log(n) \in \mathcal{O}(n \log(n)) \end{aligned}$$

■

7.2.6 BucketSort

Keiner der bisher vorgestellten Sortieralgorithmen lief schneller als $\mathcal{O}(n \log(n))$, und wir werden später noch sehen dass dies tatsächlich eine untere Schranke für die Laufzeit aller *vergleichsbasierten* Sortierverfahren darstellt. Besitzt die Eingabemenge allerdings bestimmte vorher bekannte Eigenschaften, so ist unter Umständen tatsächlich ein Sortieren in *linearer Zeit* möglich!

Stellen wir uns zum Beispiel vor, wir müssten Zettel, die mit den Zahlen von 1 bis n beschriftet sind der Grösse nach sortieren. Dazu könnten wir einfach n "Zettelablagen" in einer Reihe aufstellen, jeweils einen Zettel "ziehen" und ihn in die passende Ablage legen. Danach könnten wir die Zettel der Reihe nach aus den Ablagen entnehmen, und hätten in Zeit $2n$ sortiert.

Diese Idee liegt *BucketSort* zu Grunde. Gegeben sei hier ein Array A von reellen Zahlen a_i, die alle aus einem endlichen Intervall $[a, b) \subset \mathbb{R}$ stammen und *uniform verteilt* sind:

Definition 7.4 (Uniforme Verteilung). Eine Menge von reellen Zahlen A aus einem Intervall $[a, b) \subset \mathbb{R}$ heißt *uniform verteilt*, wenn gilt:

$$\forall x \in A, \forall [c, d) \subset [a, b) : p(x \in [c, d)) = \frac{d - c}{b - a}.$$

Intuitiv bedeutet dies, dass die Wahrscheinlichkeit, dass ein zufällig gewähltes Element in einem bestimmten Teilintervall $[c, d) \subset [a, b)$ liegt, nur von der Grösse dieses Teilintervalls abhängt.
Ohne Beschränkung der Allgemeinheit können wir (durch geeignetes Skalieren) immer davon ausgehen, dass $[a, b) = [0, 1)$, d.h. dass die Zahlen aus A aus dem Einheitsintervall stammen. Dann gilt für die in Definition 7.4 definierte Wahrscheinlichkeit:

$$p(x \in [c, d)) = d - c.$$

Damit erhalten wir für eine uniform verteilte Menge von n Elementen:

$$p\left(x \in \left[\frac{i}{n}, \frac{i+1}{n}\right)\right) = \frac{i - (i+1)}{n} = \frac{1}{n} \quad \forall i \in \{0, \ldots, n-1\}$$

Idee von *BucketSort*

BucketSort verwendet eine Art von Hashing mit Chaining zur Kollisionsauflösung: Wir teilen das Intervall $[0, 1)$ in n gleich große Teilintervalle, die sogenannten "Buckets"[3]. Man ordne nun die n zu sortierenden Elemente in ihren jeweiligen Bucket ein (d.h. man hänge sie einfach an die Liste der in ihrem Bucket gespeicherten Elemente an), und sortiere anschliessend alle Buckets mit Hilfe von *TrivialSort*. Zum Schluss muss nur noch einmal über alle Buckets iteriert werden, um die in ihnen gespeicherten Elemente hintereinander zu hängen.

[3] diese entsprechen natürlich den "Zettelablagen" aus dem obigen Beispiel

An dieser Stelle sieht man auch deutlich, warum die Eingabemenge uniform verteilt sein muss, um eine ”gute” Laufzeit zu erreichen: bei uniformer Verteilung werden die Elemente gleichmässig über die Buckets verteilt, so dass die einzelnen Aufrufe von *TrivialSort* nur sehr kleine Teilmengen sortieren müssen.
Aufgrund der offensichtlichen Verwandtschaft von *BucketSort* zum Hashing mit Chaining können wir auch eine typische Hashtabelle als Datenstruktur verwenden, d.h. wir verwenden ein Feld von n Listen,[4] in denen dann die eigentlichen reellen Zahlen abgelegt werden. Die verwendete Hashfunktion ist hier trivial:

$$h(x) = i \Leftrightarrow \chi_{[\frac{i}{n}, \frac{i+1}{n})}(x) = 1$$

Dabei ist $\chi_{[\frac{i}{n}, \frac{i+1}{n})}$ die *charakteristische Funktion* des Intervalls $[\frac{i}{n}, \frac{i+1}{n})$, die durch

$$\chi_{[a,b)}(x) = 1 :\Leftrightarrow x \in [a, b)$$

definiert wird.
Die Anzahl der im Bucket i gespeicherten Elemente bezeichnen wir mit n_i. Dann gilt offenbar für die Laufzeit von *BucketSort* auf einem Array der Länge n:

Lemma 7.4.

$$T_{BS}(n) = cn + \sum_{i=0}^{n-1} dn_i^2$$

Beweis. Zuerst müssen wir jedes Element von A in die Hashtabelle einsortieren. Die Hashfunktion lässt sich offensichtlich in $\mathcal{O}(1)$ berechnen, daher benötigen wir für das Einsortieren aller n Elemente $\Theta(n)$ Schritte. Dann wird jeder Bucket i mit *TrivialSort* sortiert, was nach Satz 7.1 in Zeit $\Theta(n_i^2)$ läuft. Das Zusammenhängen der einzelnen in den Buckets gespeicherten Listen zu einem sortierten Array benötigt wiederum $\mathcal{O}(n)$ Schritte. Aufsummieren all dieser Terme liefert die Behauptung. ■

Satz 7.10 (Erwartete Laufzeit von *BucketSort*). *Die erwartete Laufzeit $E[T_{BS}(n)]$ von* Bucket-Sort *auf einem Array von uniform verteilten reellen Zahlen aus einem Intervall $[a, b)$ liegt in $\mathcal{O}(n)$*

Beweis. Zunächst folgt mit Lemma 7.4:

$$\begin{aligned} E[T_{\text{BS}}(n)] &= E\left[cn + \sum_{i=0}^{n-1} dn_i^2\right] \\ &= E[cn] + d\sum_{i=0}^{n-1} E[n_i^2] \\ &= cn + d\sum_{i=0}^{n-1} E[n_i^2] \end{aligned} \tag{7.1}$$

[4] Man beachte, dass in diesem Fall die Länge der Hashtabelle der Anzahl der zu speichernden Elemente entspricht!

Zur Bestimmung von $E[n_i^2]$ führen wir nun neue Indikatorvariablen ein:

$$X_{ij} = \begin{cases} 1 & \text{falls } a_j \text{ Bucket } b_i \text{ zugeordnet wird} \\ 0 & \text{sonst} \end{cases}$$

wobei $i = 0, \ldots, n-1, j = 0, \ldots, n-1$ Da n_i die Anzahl an Elementen in Bucket i beschreibt, gilt:

$$n_i = \sum_{j=0}^{n-1} X_{ij}.$$

Damit können wir nun den Erwartungswert $E[n_i^2]$ umformen:

$$\begin{aligned} E[n_i^2] &= E\left[\left(\sum_{j=0}^{n-1} X_{ij}\right)^2\right] \\ &= E\left[\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} X_{ij}X_{ik}\right] \\ &= E\left[\sum_{j=0}^{n-1} X_{ij}^2 + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} X_{ij}X_{ik}\right] \\ &= \sum_{j=0}^{n-1} E\left[X_{ij}^2\right] + \sum_{j=0}^{n-1}\sum_{\substack{k=0 \\ k\neq j}}^{n-1} E[X_{ij}X_{ik}]. \end{aligned}$$

Da es sich bei den X_{ij} um Indikatorvariablen handelt, und da wir uniforme Verteilung der Elemente angenommen haben, lässt sich der hier auftretende Term $E[X_{ij}^2]$ sehr einfach bestimmen:

$$\begin{aligned} E[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\ &= \frac{1}{n} && (7.2) \\ &= E[X_{ij}]. && (7.3) \end{aligned}$$

Weiterhin können wir für $k \neq j$ den Erwartungswert $E[X_{ij}X_{ik}]$ zu $E[X_{ij}]E[X_{ik}]$ auseinanderziehen, da in diesem Fall die Zufallsvariablen X_{ij} und X_{ik} offensichtlich unabhängig voneinander sind.

Damit können wir nun weiter umformen:

$$
\begin{aligned}
E[n_i^2] &= \sum_{j=0}^{n-1} E\left[X_{ij}^2\right] + \sum_{j=0}^{n-1} \sum_{\substack{k=0 \\ k\neq j}}^{n-1} E[X_{ij}X_{ik}] \\
&\underset{(7.2)}{=} \sum_{j=0}^{n-1} \frac{1}{n} + \sum_{j=0}^{n-1} \sum_{\substack{k=0 \\ k\neq j}}^{n-1} E[X_{ij}]E[X_{ik}] \\
&\underset{(7.3)}{=} \sum_{j=0}^{n-1} \frac{1}{n} + \sum_{j=0}^{n-1} \sum_{\substack{k=0 \\ k\neq j}}^{n-1} \frac{1}{n}\frac{1}{n} \\
&= \sum_{j=0}^{n-1} \frac{1}{n} + \sum_{j=0}^{n-1} \sum_{\substack{k=0 \\ k\neq j}}^{n-1} \frac{1}{n^2} \\
&= n\,\frac{1}{n} + n(n-1)\frac{1}{n^2} \\
&= 1 + 1 - \frac{1}{n} \\
&= 2 - \frac{1}{n}. \qquad (7.4)
\end{aligned}
$$

Durch Einsetzen von (7.4) in Gleichung (7.1) folgt schliesslich die Behauptung:

$$
\begin{aligned}
E[T_{\text{BS}}(n)] &= cn + d\sum_{i=0}^{n-1} E[n_i^2] \\
&\underset{(7.4)}{=} cn + d\sum_{i=0}^{n-1} \left(2 - \frac{1}{n}\right) \\
&= cn + d\sum_{i=0}^{n-1} 2 - d\sum_{i=0}^{n-1} \frac{1}{n} \\
&= cn + 2dn - dn\,\frac{1}{n} \\
&= cn + 2dn - d \\
&= (c + 2d)n - d \in \mathcal{O}(n).
\end{aligned}
$$

■

7.2.7 Untere Laufzeitschranken für das Sortierproblem

Mit *BucketSort* haben wir ein Sortierverfahren kennengelernt, welches Elemente mit bestimmten angenehmen Eigenschaften in linearer Zeit sortiert. Diese notwendigen Eigenschaften – alle zu sortierenden Elemente sollen uniform verteilt aus einem Intervall $[a, b) \subset \mathbb{R}$ stammen – schränken natürlich die Anwendbarkeit von *BucketSort* stark ein. Insbesondere ist damit

BucketSort im strikten Sinne unserer ursprünglichen Definition 7.1 **keine** Lösung des allgemeinen Sortierproblems!

Die einzige zum Sortieren "verwertbare" Eigenschaft der Eingabemenge des allgemeinen Sortierproblems 7.1 ist offenbar die Ordnungsrelation $\preceq$. Einen Sortieralgorithmus, der ausschliesslich diese Ordnungsrelation benötigt um die Eingabemenge zu sortieren, also keine weiteren Voraussetzungen an die Struktur der Eingabedaten stellt, nennen wir einen "*vergleichsbasierten*" Sortieralgorithmus.

Präziser:

Definition 7.5 (Vergleichsbasiertes Sortierverfahren). Gegeben sei die Eingabemenge $A = \{a_0, \ldots, a_{n-1}\}$ mit der Ordnungsrelation $\preceq$. Ein Sortierverfahren $\mathcal{S}$ heißt *vergleichsbasiert*, wenn für die Bestimmung der endgültigen Position jedes Elementes a_i im sortierten Array $\mathcal{S}(A)$ ausschliesslich die Ergebnisse von Vergleichen der Form $a_i \preceq x$ herangezogen werden, wobei x ein beliebiges Element aus dem Definitionsbereich von $\preceq$ darstellt (z.B. ein anderes Arrayelement a_j).

Beispiel 7.6. Beispiele für vergleichsbasierte Sortierverfahren sind u.a.:

- *TrivialSort*
- *MergeSort*
- *HeapSort*
- *QuickSort*

Anmerkung 7.7. *BucketSort* ist offensichtlich **kein** vergleichsbasiertes Sortierverfahren, da zum Einsortieren der Elemente zusätzlich zur Ordnungsrelation $\leq$ eine Hashfunktion h herangezogen wird!

Keines der bisher vorgestellten vergleichsbasierten Sortierverfahren hat eine bessere worstcase Laufzeit als $\mathcal{O}(n \log(n))$, d.h. ihre Laufzeiten liegen in $\Omega(n \log(n))$. Wir werden nun zeigen, dass dies kein Zufall ist: $n \log(n)$ ist eine untere Schranke für die Laufzeit vergleichsbasierter Sortierverfahren!

Anmerkung 7.8. Im restlichen Verlauf dieses Kapitels werden wir annehmen, dass alle Eingabeelemente a_i paarweise verschieden sind, d.h. aus $i \neq j$ folgt schon $a_i \neq a_j$. In diesem Fall gilt: $a_i \prec a_j \Leftrightarrow a_i \preceq a_j$ und $a_i \succ a_j \Leftrightarrow a_i \succeq a_j$. Man macht sich leicht klar, dass dann die Vergleiche $a_i \preceq a_j, a_i \prec a_j, a_i \succeq a_j, a_i \succ a_j$ alle den selben Informationsgehalt haben, d.h. aus der Kenntnis des Ergebnisses eines dieser Vergleiche kann man die Ergebnisse der anderen drei sofort ableiten.

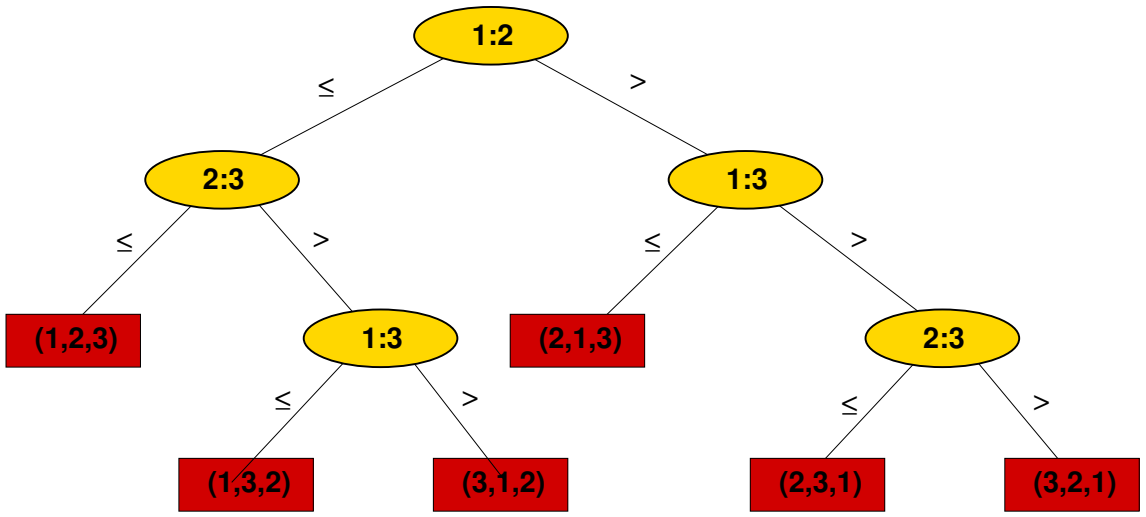
Um überhaupt eine Aussage für **alle** vergleichsbasierten Sortierverfahren treffen zu können, müssen wir soweit wie möglich vom konkreten Verfahren abstrahieren. Dies gelingt mit dem sogenannten *Entscheidungsbaummodell*.

Das Entscheidungsbaummodell

Mit Hilfe sogenannter *Entscheidungsbäume* lässt sich der Ablauf eines vergleichsbasierten Sortierverfahrens abstrakt darstellen.

Definition 7.6 (Entscheidungsbaum). Ein *Entscheidungsbaum* $\mathcal{B}$ ist ein binärer Baum, der alle möglichen Folgen von Vergleichen eines vergleichenden Sortierverfahrens für alle möglichen Eingabemengen einer bestimmten Größe n repräsentiert. Dabei ist jeder innere Knoten v beschriftet mit einem String der Form $i : j$, jedes Blatt w ist beschriftet mit einem String der Form $(\pi_1, \ldots, \pi_n)$ mit $\pi_i \in \{0, \ldots, n-1\}$. Von den beiden ausgehenden Kanten e_1, e_2 eines jeden inneren Knoten v ist genau eine mit $\preceq$ beschriftet, die jeweils andere mit $\succ$.
$\mathcal{B}$ enthält genau dann einen inneren Knoten v mit Beschriftung $i : j$, wenn im Verlauf des zugehörigen Sortierverfahrens Element a_i mit Element a_j verglichen wird. $\mathcal{B}$ enthält für jede Permutation $\pi := (\pi_1, \ldots, \pi_n)$ der Indizes der Eingabeelemente $\{0, \ldots, n-1\}$ genau ein Blatt w mit Beschriftung π.

Beispiel 7.7 (Entscheidungsbaum der Länge 3 für ein triviales Sortierverfahren).



Lemma 7.5. *Es sei $\mathcal{S}$ ein beliebiges korrektes vergleichsbasiertes Sortierverfahren. Dann gilt: der Entscheidungsbaum der Länge n für $\mathcal{S}$ hat mindestens $n!$ Blätter.*

Beweis. $\mathcal{S}$ sotiert eine beliebige Eingabe A der Länge n korrekt zu $\mathcal{S}(A)$ um. Da die Elemente in A in beliebiger Reihenfolge stehen können, muss $\mathcal{S}$ auch jede mögliche Permutation der Eingabeelemente als Ausgabe erzeugen können. Da das Entscheidungsbaummodell den Ablauf von $\mathcal{S}$ auf einer beliebigen Eingabemenge einer fixen Größe n darstellt, muss also auch im Entscheidungsbaum für eine Eingabe von n Elementen **jede** der $n!$ möglichen Permutationen der Eingabeindizes als Blatt vorkommen. ■

Diese Information können wir verwenden, um eine untere Schranke für die worst-case Laufzeit eines allgemeinen vergleichsbasierten Sortieralgorithmus auf einer Eingabe der Länge n abzuleiten. Dazu bemerken wir zunächst, dass zu jeder möglichen Eingabe A der Länge n genau ein Blatt $\mathcal{S}(A)$ im Baum existiert, dass diese Eingabe korrekt sortiert.

Lemma 7.6. *Es sei A eine beliebig aber fest gewählte Eingabe für ein vergleichsbasiertes Sortierverfahren $\mathcal{S}$. $\mathcal{S}(A)$ sei das zu dieser Eingabe gehörige Blatt im Entscheidungsbaum der Länge n für $\mathcal{S}$, welches A korrekt sortiert. $h(\mathcal{S}(A))$ sei die Höhe dieses Blattes im Entscheidungsbaum. Dann gilt: die Anzahl an Vergleichen, die ein beliebiger vergleichsbasierter Sortieralgorithmus benötigt, um A zu sortieren, liegt in $\Omega(h(\mathcal{S}(A)))$.*

Beweis. Jede Kante im Entscheidungsbaum repräsentiert einen Vergleich zweier Eingabelemente. Man benötigt also mindestens $h(\mathcal{S}(A))$ Vergleiche, um das Blatt $\mathcal{S}(A)$ ausgehend von der Wurzel erreichen zu können. Dies entspricht der minimalen Vergleichsanzahl mit der entschieden werden kann, ob $\mathcal{S}(A)$ eine korrekte Sortierung von A darstellt. ■

Lemma 7.7. *Es sei $h_n^{\mathcal{S}}$ die Höhe des Entscheidungsbaums der Länge n für ein Sortierverfahren $\mathcal{S}$. Dann gilt: die worst-case Laufzeit von $\mathcal{S}$ auf einer Eingabe der Länge n liegt in $\Omega(h_n^{\mathcal{S}})$.*

Beweis. Nach Lemma 7.6 wissen wir, dass $\mathcal{S}$ mindestens $\Omega(h(\mathcal{S}(A))$ Vergleiche benötigt, um das Ergebnis $\mathcal{S}(A)$ zu berechnen. Im worst-case entspricht das korrekte Ergebnis einem Blatt mit maximaler Höhe im Entscheidungsbaum, d.h. es werden mindestens $\Omega(h_n^{\mathcal{S}})$ Vergleiche benötigt, um das korrekte Ergebnis zu berechnen. Alle anderen eventuell noch nicht berücksichtigten Operationen – wie z.B. das Umkopieren von Elementen oder die Speicherallokation – können die Laufzeit natürlich höchstens noch in die Höhe treiben. Damit folgt die Behauptung. ■

Bislang haben wir also folgendes erreicht: wenn wir eine untere Schranke für die Höhe aller möglichen Entscheidungsbäume zu einer bestimmten Eingabegröße n bestimmen können, dann können wir nach Lemma 7.7 damit eine untere Schranke für die Laufzeit aller möglichen vergleichsbasierten Sortierverfahren berechnen. Aus Lemma 7.5 ist uns schon bekannt, dass **jeder** Entscheidungsbaum der Länge n mindestens $n!$ Blätter besitzen muss. Diese Informationen kombinieren wir zum folgenden zentralen Satz dieses Kapitels:

Satz 7.11 (Laufzeit beliebiger vergleichsbasierter Sortierverfahren). *Die worst-case Laufzeit jedes vergleichsbasierte Sortieralgorithmus $\mathcal{S}$ beim Sortieren einer beliebigen Eingabe A der Länge n liegt in $\Omega(n \log(n))$.*

Beweis. $\mathcal{B}_n^{\mathcal{S}}$ sei der Entscheidungsbaum der Länge n für $\mathcal{S}$. Dann gilt nach Lemma 7.5: $\mathcal{B}_n^{\mathcal{S}}$ hat $\Omega(n!)$ Blätter. Andererseits hat $\mathcal{B}_n^{\mathcal{S}}$ als binärer Baum der Höhe $h(\mathcal{B}_n^{\mathcal{S}})$ maximal $2^{h(\mathcal{B}_n^{\mathcal{S}})}$ Blätter. Es gilt also die Abschätzung:

$$n! \leq 2^{h(\mathcal{B}_n^{\mathcal{S}})}$$

Logarithmieren ergibt:

$$h(\mathcal{B}_n^{\mathcal{S}}) \geq \log(n!) \tag{7.5}$$

Die Fakultät können wir mit Hilfe der *Stirlingformel*

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\alpha_n}$$

mit

$$\frac{1}{12\,n+1} < \alpha_n < \frac{1}{12}$$

umformen zu:

$$\begin{aligned}
\log(n!) &= \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n}\right) \\
&= \log\left(\sqrt{2\pi n}\right) + \log\left(\left(\frac{n}{e}\right)^n\right) + \log\left(e^{\alpha_n}\right) \\
&= \frac{1}{2}\log(2\pi n) + n\log\left(\frac{n}{e}\right) + \alpha_n \log(e) \\
&\geq n\log\left(\frac{n}{e}\right) \\
&= n\left(\log(n) - \log(e)\right)
\end{aligned}$$

Da wir nur an einer asymptotischen Aussage interessiert sind, können wir verlangen, dass $\log(n) > 2\log(e)$, also $n > e^2$. In diesem Fall gilt dann weiter:

$$\begin{aligned}
\log(n!) &\geq n\left(\log(n) - \log(e)\right) \\
&\geq n\log(n) - \frac{1}{2}n\log(n) \\
&= \frac{1}{2}n\log(n)
\end{aligned}$$

Also erhalten wir mit Gleichung (7.5):

$$h(\mathcal{B}_n^{\mathcal{S}}) \in \Omega(n\log(n))$$

Mit Lemma (7.7) folgt die Behauptung. ■

Satz 7.12. HeapSort *und* MergeSort *sind asymptotisch optimale vergleichsbasierte Sortierverfahren.*

Beweis. Nach Satz 7.8 und Satz 7.3 liegt die Laufzeit von *HeapSort* und *MergeSort* jeweils in $\mathcal{O}(n\log(n))$. Da laut Satz 7.11 beide Laufzeiten auch in $\Omega(n\log(n))$ liegen müssen, folgt die Behauptung. ■