Kapitel 2 - Sortieren

Elementare Sortierverfahren
SelectionSort
InsertionSort
BubbleSort

Höhere Sortierverfahren
MergeSort
QuickSort
HeapSort

Untere und obere Schranken für das Sortierproblem

Spezielle Sortierverfahren
CountingSort

Voraussetzungen

- Für nachfolgende Betrachtungen sei jeweils eine Folge a[1],..., a[n] von Datensätzen gegeben.
- Jeder Datensatz a[i] besitzt eine Schlüsselkomponente a[i].key
- Datensätze enthalten außer der Schlüsselkomponente in der Regel weitere Informationseinheiten (z.B. Name, Adresse, PLZ, etc.).
- Sortierung erfolgt ausschließlich nach der Schlüsselkomponente key.
- Für Sortierung muss auf der Menge aller Schlüssel eine totale Ordnung definiert sein.

Totale/lineare Ordnung

Sei M eine nicht leere Menge und "$\leq$" $\subseteq M \times M$ eine binäre Relation auf M.

Das Paar $(M, \leq)$ heißt eine *totale (bzw. lineare) Ordnung auf* M, genau dann wenn "$\leq$" die folgenden Eigenschaften erfüllt:

reflexiv:	$\forall x \in M$:	$x \leq x$
transitiv:	$\forall x, y, z \in M$:	$x \leq y \wedge y \leq z \Rightarrow x \leq \mathrm{z}$
antisymmetrisch:	$\forall x, y \in M$:	$x \leq y \wedge y \leq x \Rightarrow x = y$
total / linear:	$\forall x, y \in M$:	$x \leq y \vee y \leq x$

Beispiel: Totale Ordnung

1. Bsp.: $\leq$ auf den natürlichen Zahlen bildet eine totale Ordnung.

2. Bsp.: Die *lexikographische Ordnung* $\leq_{lex}$ ist eine totale Ordnung.
   Sei $\leq$ die totale Ordnung auf den Buchstaben A < B < C <…< Y < Z
   Für zwei endliche Wörter $u = u_1 u_2 \ldots u_m$ und $v = v_1 v_2 \ldots v_n$ mit $u_i, v_j \in$ {A, B, C, …, Z} gilt $u \leq_{lex} v$ gdw.

   $$(u \text{ leer}) \vee \left(u, v \text{ nicht}-\text{leer} \wedge \left(u_1 < v_1 \vee (u_1 = v_1 \wedge u_2 \ldots u_m \leq_{lex} v_2 \ldots v_n)\right)\right)$$

   Beispiel: FLO $\leq_{lex}$ FLORIAN $\leq_{lex}$ PAUL $\leq_{lex}$ PETER

3. Bsp. (keine totale Ordnung): Teilmengenrelation $\subseteq$ auf Potenzmenge von $\mathbb{N}$
   Beweis: $\subseteq$ ist nicht total, da weder $\{1\} \subseteq \{2\}$ noch $\{2\} \subseteq \{1\}$ gilt.

Sortierproblem

- Gegeben sei eine Folge $a[1], \ldots, a[n]$ von Datensätzen mit einer Schlüsselkomponente $a[i].\mathrm{key}$ $(i = 1, \ldots, n)$ und eine totale Ordnung $\leq$ auf der Menge aller Schlüssel.

- Das Sortierproblem besteht darin, eine *Permutation* π der ursprünglichen Folge zu bestimmen, so dass gilt:

$$a[\pi_1].\mathrm{key} \leq a[\pi_2].\mathrm{key} \leq \ \ldots \ \leq a[\pi_{n-1}].\mathrm{key} \leq a[\pi_n].\mathrm{key}$$

Beispiel

Liste	**Schlüsselelement**	**Ordnung**
Telefonbuch	Nachname	lexikographische Ordnung
Klausurergebnisse	Punktezahl	$\leq$ auf rationalen Zahlen
Lexikon	Stichwort	lexikographische Ordnung
Studentenverzeichnis	Matrikelnummer	$\leq$ auf $\mathbb{N}$
Entfernungstabelle	Distanz	$\leq$ auf $\mathbb{R}$
Fahrplan	Abfahrtszeit	„früher als“

Unterscheidungskriterien

Sortieralgorithmen können nach verschiedenen Kriterien klassifiziert werden:

- Berechnungsaufwand
  - $O(n^2)$, $O(n \log n)$, $O(n)$
- Worst Case vs. Average Case
  - Ausnutzen von Vorsortierungen
- Speicherbedarf
  - In-place / in situ
  - Kopieren
- stabil (Reihenfolge von Records mit gleichem Schlüssel bleibt erhalten) oder nicht

Weitere Aufgaben

Viele Aufgaben sind mit dem Sortieren verwandt und können auf das Sortierproblem zurückgeführt werden:

- Bestimmung des Median (der Median ist definiert als das Element an der mittleren Position der sortierten Folge)
- Bestimmung der k kleinsten bzw. größten Elemente

Typen von Sortieralgorithmen

Einfache

SelectionSort, InsertionSort, BubbleSort, ...

Höhere

MergeSort, QuickSort, HeapSort, ...

Spezielle

BucketSort, ...

Swap

Im Folgenden wird häufiger der Aufruf swap(a, i, j) verwendet, wobei a ein Array ist und i, j int-Werte.

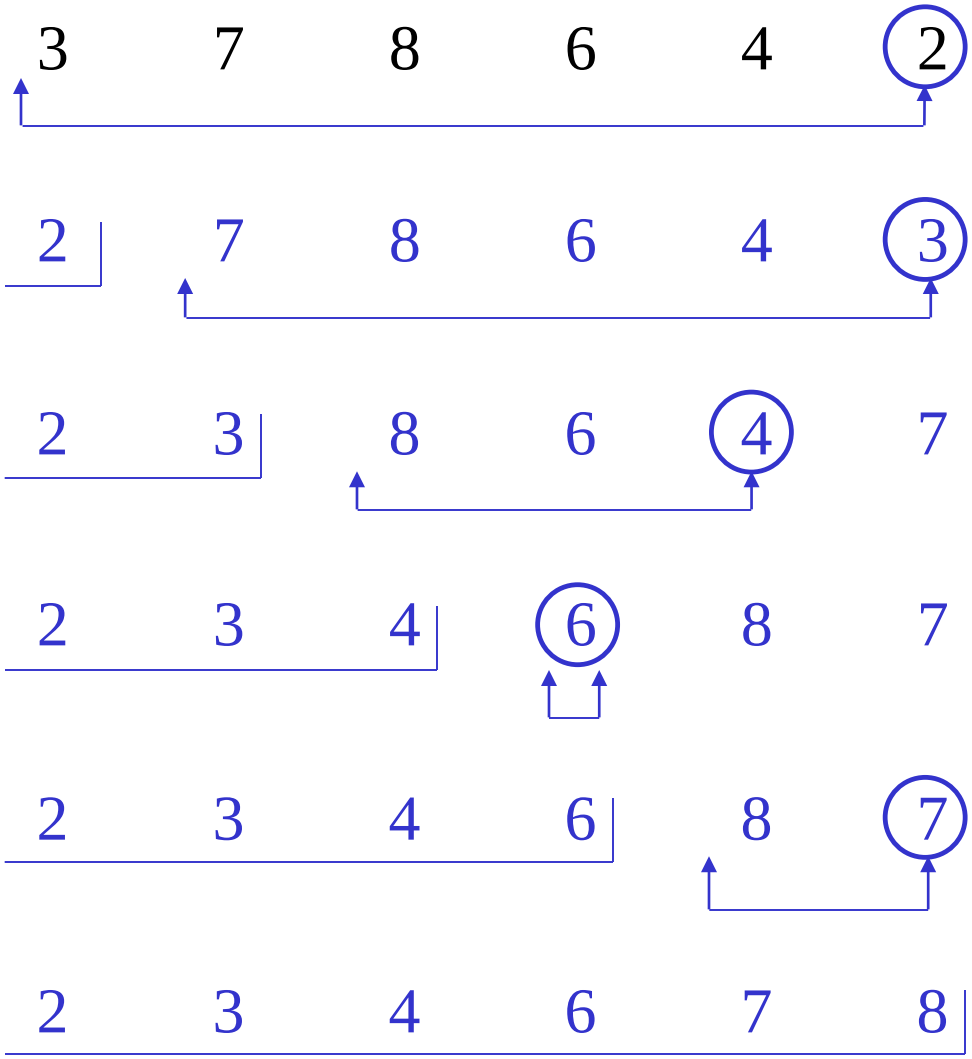
Der Aufruf ersetzt folgende drei Zuweisungen:

```
(int) h = a[i];
a[i] = a[j];
a[j] = h;
```

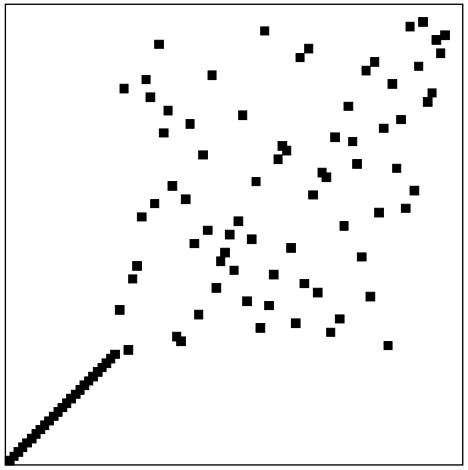
SelectionSort: Prinzip

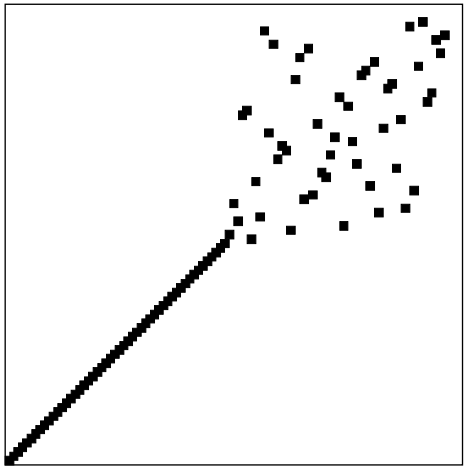
1. Suche kleinstes Element
2. Vertausche es mit dem Schlüssel an der ersten Stelle
3. Wende denselben Algorithmus auf die restlichen $(n-1)$ Elemente an

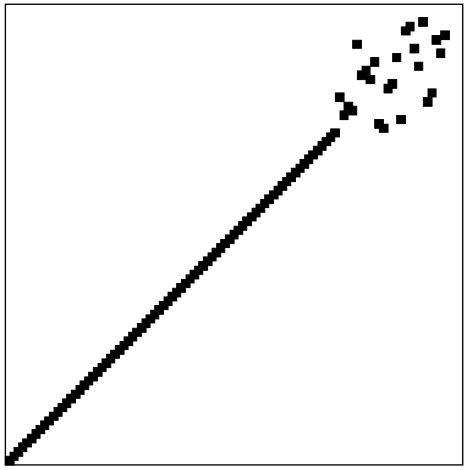
SelectionSort: Beispiel



SelectionSort: Visualisierung







SelectionSort: Implementierung

```
static void selectionSort (int[] a) {
    for (int i = 1; i <= a.length-1; ++i) {

      int min = i;
      for (int j = i+1; j <= a.length ; ++j) {
        if  (a[j] < a[min]) min = j;
      }

      swap(a, i, min);
    }
}
```

SelectionSort: Komplexitätsanalyse

- Zum Sortieren der gesamten Folge a[1], ..., a[n] werden n -1 Durchläufe benötigt.

- Pro Schleifendurchgang i gibt es eine Vertauschung, die sich aus je einer Ausführung von swap und n - i Vergleichen zusammensetzt, wobei n-i die Anzahl der noch nicht sortierten Elemente ist.

- Insgesamt ergeben sich im best, worst und average case:
  - $(n-1)$ swaps und $\in O(n)$
  - $(n-1)+(n-2)+\ \ldots\ +2+1=\frac{n\cdot(n-1)}{2}$ Vergleiche $\in O(n^2)$
  - Durch spezielle if-Abfrage (min == i) kann Anzahl der swaps verringert werden (best case: 0 swaps); dann jedoch mehr Vergleiche notwendig

- Anzahl der Vertauschungen wächst nur linear in der Anzahl der Datensätze

  $\Rightarrow$ SelectionSort besonders für Sortieraufgaben geeignet, in denen die einzelnen Datensätze sehr groß sind

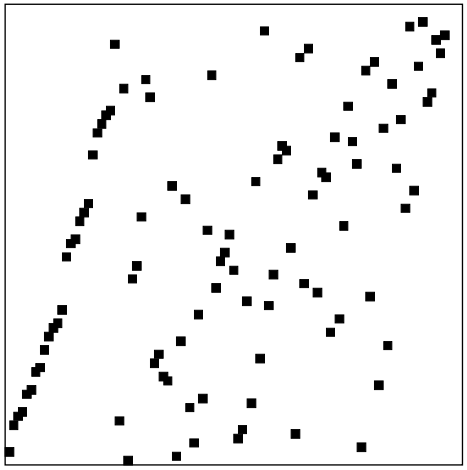
InsertionSort: Prinzip

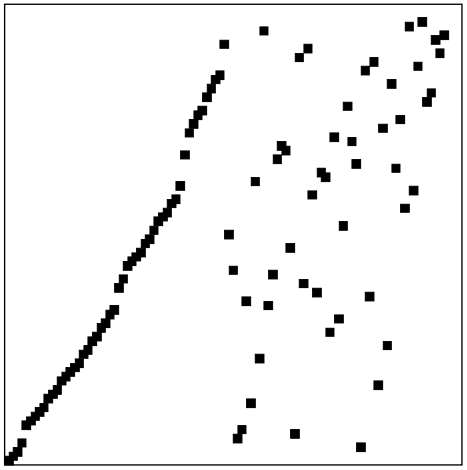
- nehme sortierte Teilfolge a[1],...,a[i-1] an (beginne mit i = 2)
- füge nächsten Datensatz a[i] an korrekte Position der Teilfolge ein

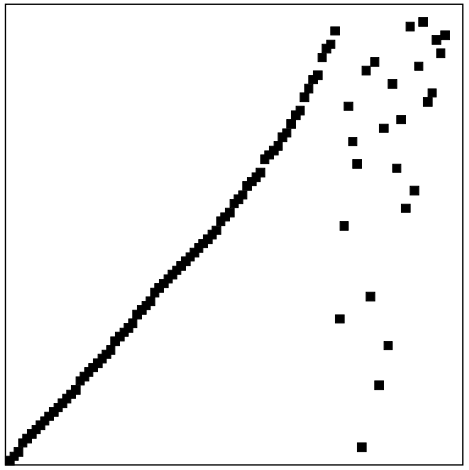
→ erhalte sortierte Teilfolge a[1],...,a[i]

→ für vollständige Sortierung $n - 1$ Schritte $(i = 2, \ldots, n)$ durchführen

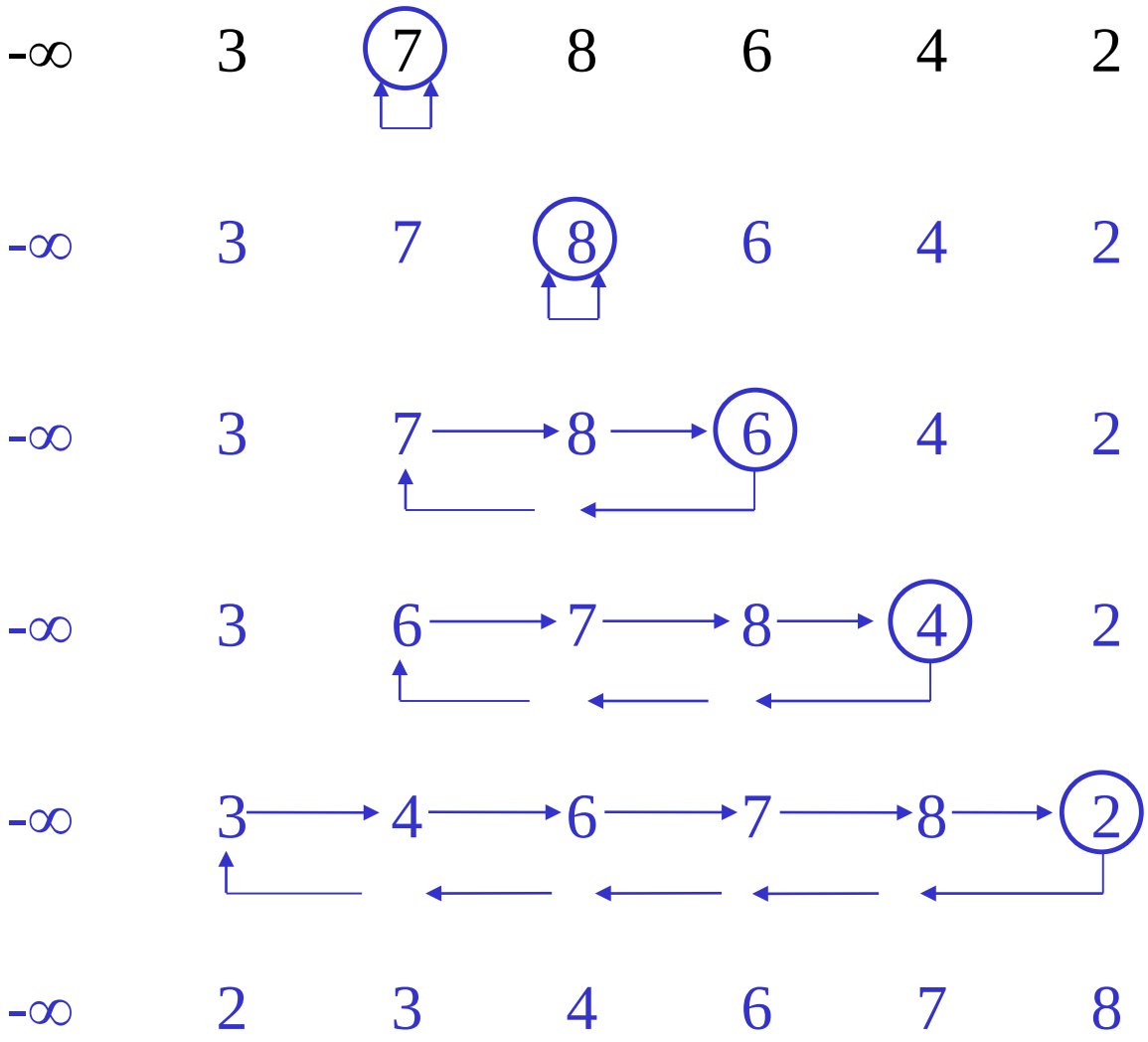
InsertionSort: Visualisierung







InsertionSort: Beispiel



InsertionSort: Implementierung

```
static void insertionSort (int[] a) {

   for (int i = 2; i <= a.length; ++i) {
        int v = a[i];
        int j = i;
        while ((a[j-1]  > v) && (j >= 2)) {
            a[j] = a[j-1];
            j--;
        }
        a[j] = v;
   }
}
```

InsertionSort: Variante zur Implementierung

- **Variante**

  Verwende **Sentinel**, um in der while-Schleife eine zusätzliche Abfrage auf die linke Feldgrenze zu vermeiden (Sentinel-Element entspricht Wärter; Anfangs- oder Endmarkierung)

- Aufruf wie folgt:

  a[0] := -∞

  dann: Array um 1 verlängern; der Inhalt liegt dann in a[1], …, a[n]

InsertionSort: Komplexitätsanalyse

Vergleiche

- Einfügen des Elements a[i] in die bereits sortierte Anfangsfolge a[1], ...,a[i-1] erfordert mindestens einen Vergleich und höchstens i Vergleiche.
- Im Mittel: i/2 Vergleiche, denn bei zufälliger Verteilung der Schlüssel ist die Hälfte der bereits eingefügten Elemente größer als das Element a[i].

- Best Case
  Bei vollständig vorsortierten Folgen ergeben sich $n-1$ Vergleiche.

- Worst Case
  Bei umgekehrt sortierten Folgen gilt für die Anzahl der Vergleiche:

$$\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1 = \frac{n(n+1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1$$

- Average Case: Die Anzahl der Vergleiche ist etwa $\frac{n^2}{4}$

InsertionSort: Komplexitätsanalyse

Bewegungen

- Best Case
  Bei vollständig vorsortierten Folgen $2 \cdot (n-1)$ Bewegungen
  - Zuweisungen v = a[i] und a[j] = v in jedem Schritt notwendig
  - durch zusätzliche if-Abfragen verbesserbar, dann jedoch mehr Vergleiche

- Worst Case
  Bei umgekehrt sortierten Folgen $\frac{n^2}{2}$ Bewegungen

- Average Case
  $\sim \frac{n^2}{4}$ Bewegungen

BubbleSort: Prinzip

- Grundidee ähnlich zu SelectionSort
- beim Vergleichen (zum Finden des Minimums bei SelectionSort) die Objekte direkt paarweise korrekt anordnen

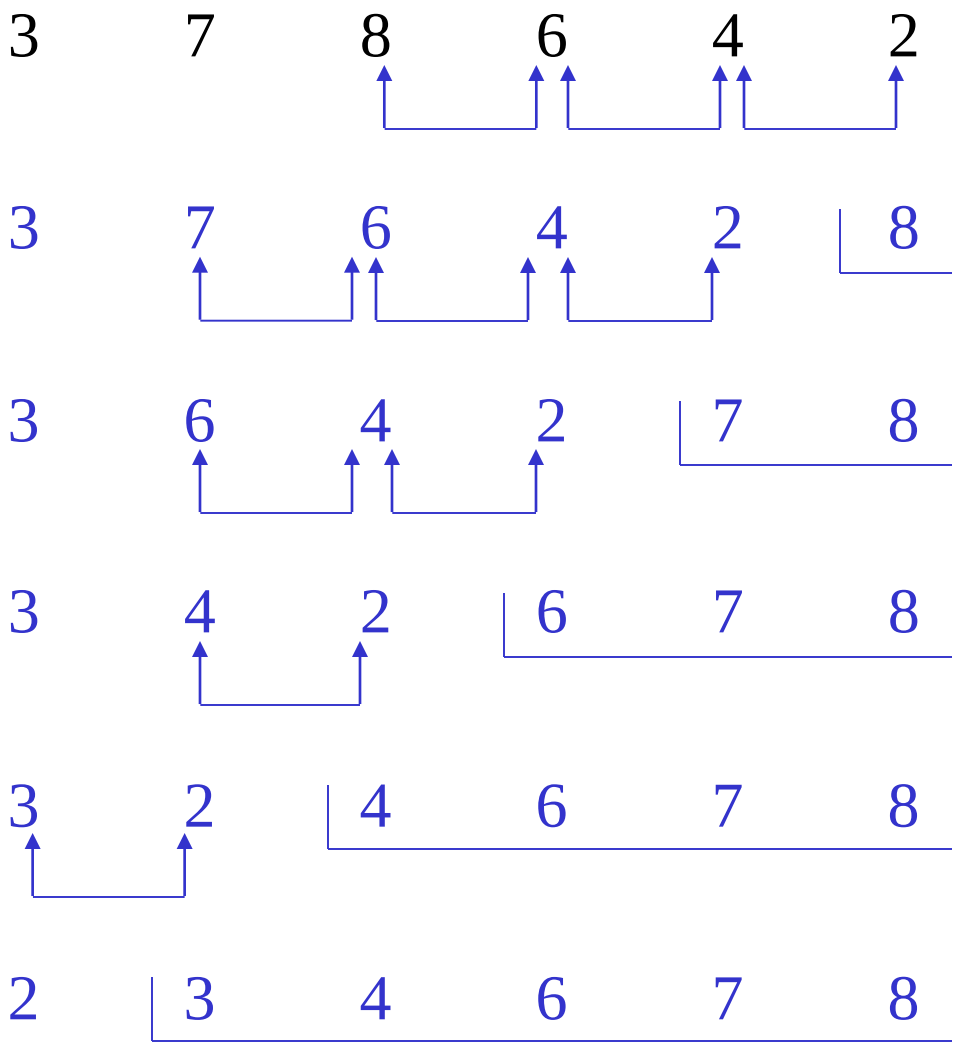
→ Nachziehen von Objekten; Aufsteigen von Blasen

- $n-1$ Schritte, $i = n, n-1, \ldots, 2$ (i herunterzählen)
- Für jeden dieser Schritte:

  $i-1$ Schritte, $j = 2, \ldots, i$

  ordne a[j-1] und a[j]

BubbleSort: Beispiel



BubbleSort: Visualisierung

BubbleSort: Komplexitätsanalyse

Vergleiche

- Anzahl der Vergleiche ist unabhängig vom Vorsortierungsgrad der Folge

  $\Rightarrow$ worst case, average case und best case sind identisch

  (es werden stets alle Elemente der noch nicht sortierten Teilfolge miteinander verglichen)

- Im i-ten Schleifendurchgang enthält die noch unsortierte Anfangsfolge $n - i + 1$ Elemente
  - für diese werden $n - i$ Vergleiche benötigt
- Um ganze Folge zu sortieren, sind $n - 1$ Schritte erforderlich
- Gesamtzahl der Vergleiche wächst damit quadratisch in der Anzahl der Schlüsselelemente:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2} \in O(n^2)$$

BubbleSort: Komplexitätsanalyse

Bewegungen/Swaps

Aus der Analyse der Bewegungen für den gesamten Durchlauf ergeben sich:

- Best Case
  0 Swaps
- Worst Case
  $\sim n^2 / 2$ Swaps
- Average Case
  $\sim n^2 / 4$ Bewegungen

Kapitel 2 - Sortieren

Elementare Sortierverfahren
SelectionSort
InsertionSort
BubbleSort

Höhere Sortierverfahren
MergeSort
QuickSort
HeapSort

Untere und obere Schranken für das Sortierproblem

Spezielle Sortierverfahren
CountingSort

Höhere Sortierverfahren

- Bislang alle Algorithmen $O(n^2)$ Vergleiche und/oder Bewegungen
- Geht es auch besser?

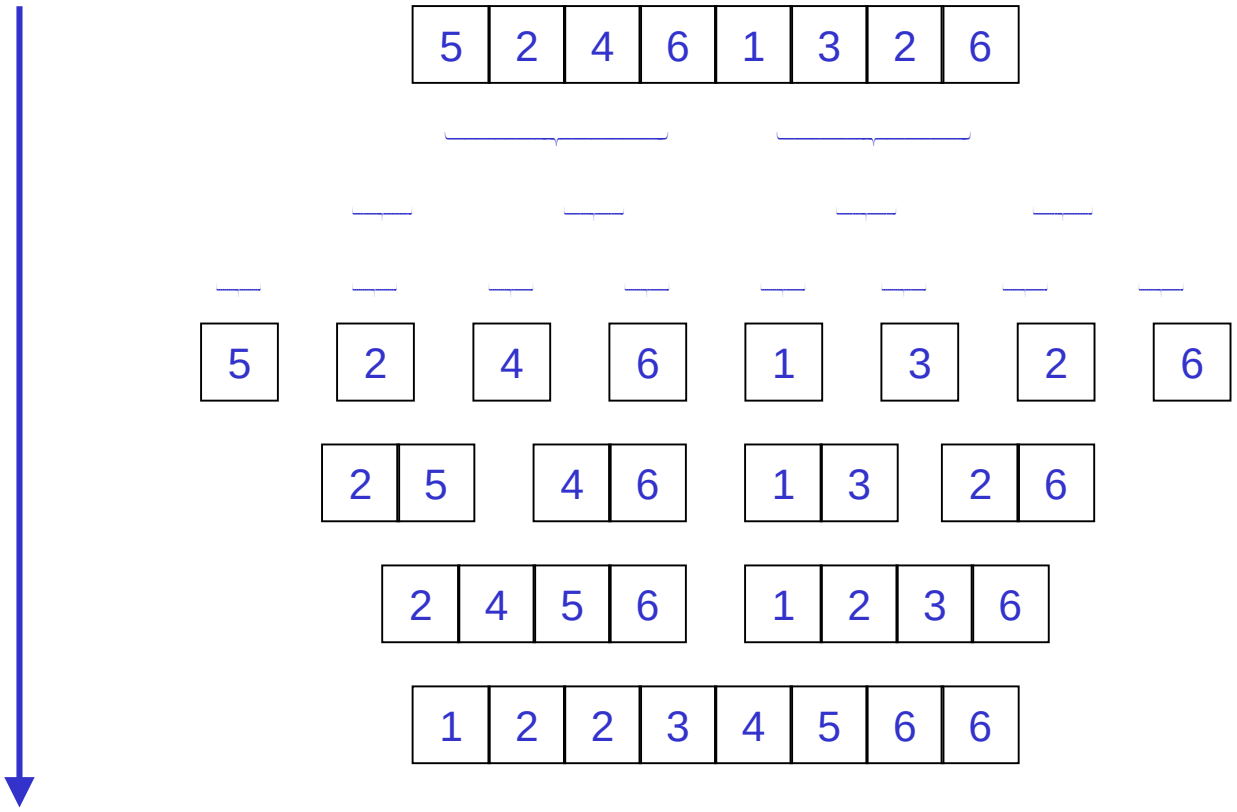
- Prinzip von MergeSort
  Gegeben sei eine Folge F = a[1] … a[n] von Schlüsselelementen.

  1. **Divide:** Zerlege F in zwei etwa gleichlange Folgen F_1=a[1] … a[$\lceil$ n/2 $\rceil$] und F_2= a[$\lceil$ n/2 $\rceil$+1] … a[n].

  2. **Conquer:** Sortiere F_1 und F_2 mittels MergeSort, falls $|F_1|$>1 bzw. $|F_2|$>1.

  3. **Merge:** Verschmelze sortierte Teilfolgen F_1 und F_2 zu sortierter Gesamtfolge

MergeSort: Beispiel



MergeSort: Implementierung

```
MergeSort(Array, left, right){
    if (left < right) {
        middle = ⌊(left + right)/2⌋;
        MergeSort(Array, left, middle);
        MergeSort(Array, middle+1, right);
        Merge(Array, left, middle, right);
    }
}
```

- Schnittstelle für die Benutzung:

```
MergeSort (Array a) {
    MergeSort(a, 0, a.length – 1);
}
```

Merge

```
Merge (Array, left, middle, right) {
    int[ ] B = new int[right – left + 1];
    int i = 0, j = left, k = middle+1;

    while ((j <= middle) and (k <= right)){
        if (Array[j] <= Array[k])   { B[i++] = Array[j++]; }
        else                        { B[i++] = Array[k++]; }
    } /*  j > middle or k > right  */

    while (j <= middle)             { B[i++] = Array[j++]; }
    while (k <= right)              { B[i++] = Array[k++]; }
    for (i = left; i <= right; ++i) { Array[i] = B[i-left]; }
}
```

Paradigma: Divide-and-Conquer

- Divide-and-Conquer (Teile-und-Herrsche, Divide-et-impera)
  - Ansatz: Zerlegung von Problemen in Teilprobleme
  - Resultiert üblicherweise in rekursiven Algorithmen
- Divide: Zerlege das Problem in Teilprobelem
  - bis „elementar“: z.B. Liste mit nur einer Zahl
- Conquer: Rekursives Lösen (Beherrschen) der Teilprobleme
- Merge: Setze Teillösungen zusammen zur Gesamtlösung
- Top-Down-Verfahren
  - Im Gegensatz zu Bottom-Up (Bsp. dynamische Programmierung)

MergeSort: Komplexitätsanalyse

Erster Ansatz: Bestimmung der Laufzeit mit Hilfe des Master-Theorems

Um das Master-Theorem anwenden zu können, ist es notwendig, die Laufzeit durch eine entsprechende Rekursionsformel zu beschreiben.

Sei T(n) die Anzahl der Operationen (im wesentlichen Vergleiche). Dann gilt:

$$T(n) = \begin{cases} c_1 & \text{für } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c_2 \cdot n & \text{für } n > 1 \end{cases}$$

für entsprechende Konstanten c_1, $c_2 > 0$. Dabei charakterisiert

- c_1 den Aufwand für die Lösung des trivialen Problems ($n = 1$) und
- $n \cdot c_2$ denjenigen für des Verschmelzen zweier Listen.

Mit Hilfe des Master-Theorems erhalten wir:

$$T(n) \in O(n \log n)$$

MergeSort: Komplexitätsanalyse

Sei $T(n)$ die Anzahl der Vergleiche. Die Folge wird in zwei Teilfolgen aufgeteilt:

Eine Teilfolge mit $\lceil \frac{n}{2} \rceil$ Elementen und eine mit $\lfloor \frac{n}{2} \rfloor$ Elementen.

Zur Verschmelzung werden $\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor - 1 = n - 1$ Vergleiche benötigt.

Daher ergibt sich folgende Rekursionsgleichung:

$$T(n) = \begin{cases} 0 & \text{für } n = 1 \\ n - 1 + T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) & \text{für } n > 1 \end{cases}$$

MergeSort: Komplexitätsanalyse

Behauptung:

$$T(n) = n \cdot \lceil ld(n) \rceil - 2^{\lceil ld(n) \rceil} + 1$$

Beweis:

Die Rekursion wird verifiziert mittels vollständiger Induktion.

Induktionsanfang: $n = 1$ … offensichtlich korrekt

Induktionsvoraussetzung: Die Behauptung gelte für alle natürlichen Zahlen m mit $1 \leq m < n$.

Induktionsschluss: …

MergeSort: Komplexitätsanalyse

Fall 1: $n \neq 2^k + 1$

Dann gilt $\lceil ld \lceil n/2 \rceil \rceil = \lceil ld \lfloor n/2 \rfloor \rceil = \lceil ld(n) \rceil - 1$, und daher

$$T(n) = n - 1 + \left\lceil \frac{n}{2} \right\rceil \cdot \left\lceil ld \left\lceil \frac{n}{2} \right\rceil \right\rceil - 2^{\left\lceil ld \left\lceil \frac{n}{2} \right\rceil \right\rceil} + 1$$

$$+ \left\lfloor \frac{n}{2} \right\rfloor \cdot \left\lceil ld \left\lfloor \frac{n}{2} \right\rfloor \right\rceil - 2^{\left\lceil ld \left\lfloor \frac{n}{2} \right\rfloor \right\rceil} + 1$$

$$= n + n\left(\left\lceil ld(n) - 1 \right\rceil\right) - 2^{\lceil ld(n) \rceil} + 1$$

$$= n \cdot \lceil ld(n) \rceil - 2^{\lceil ld(n) \rceil} + 1$$

MergeSort: Komplexitätsanalyse

Fall 2: $n = 2^k + 1$

Dann gilt $ld\lfloor n/2 \rfloor = \lceil ld\lfloor n/2 \rfloor \rceil = \lceil ld\lceil n/2 \rceil \rceil - 1 = \lceil ld(n) \rceil - 2$, und daher

$$T(n) = n - 1 + \left\lceil \frac{n}{2} \right\rceil \cdot \left(\lceil ld(n) \rceil - 1 \right) - 2^{\lceil ld(n) \rceil - 1} + 1$$

$$+ \left\lfloor \frac{n}{2} \right\rfloor \cdot \left(\lceil ld(n) \rceil - 2 \right) - 2^{\lceil ld(n) \rceil - 2} + 1$$

$$= n \cdot \left(\lceil ld(n) \rceil \right) - \left\lfloor \frac{n}{2} \right\rfloor - 2^{\lceil ld(n) \rceil - 1} - 2^{\lceil ld(n) \rceil - 2} + 1$$

$$= n \cdot \lceil ld(n) \rceil - 2^{\lceil ld(n) \rceil} + 1$$

QuickSort: Prinzip

Gegeben sei eine Folge F von Schlüsselelementen.

1. **Divide:** Zerlege F bzgl. eines partitionierenden Elementes (Pivotelement) $p \in \mathrm{F}$ in zwei Teilfolgen F_1 und F_2, so daß gilt:

$$x_1 \leq p \quad \forall x_1 \in F_1$$
$$p \leq x_2 \quad \forall x_2 \in F_2$$

2. **Conquer:** Sortiere F_1 und F_2 durch rekursiven Aufruf und erhalte dadurch die sortierten Folgen F_1^s und F_2^s

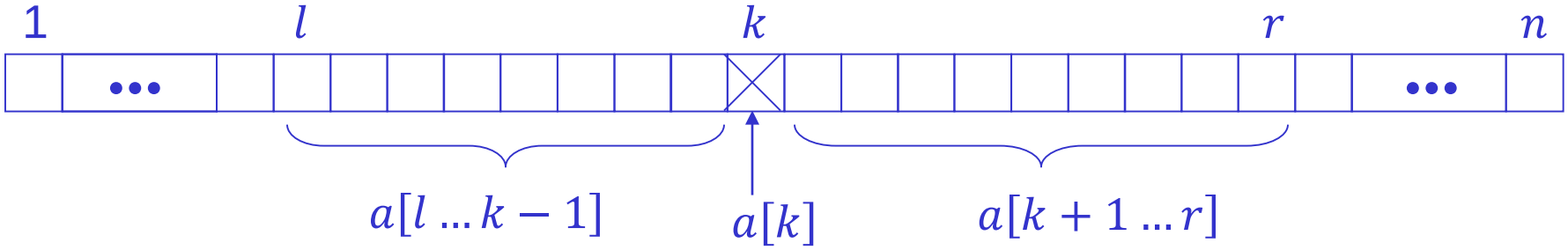
3. **Merge:** Setze Teilfolgen zusammen: $F_1^s \circ p \circ F_2^s$

Ankündigungen

- Übungsabgabe
  - Programmcode als .java abgeben (nicht als .jpg oder .pdf -- Bewertbarkeit ist sonst nicht gewährleistet)
- Übungsgruppen
  - Übungen Mo + Do 14-16 und 16-18 in Amalienstraße überlastet
  - Parallele Übungen in Freimann leer (Edmund-Rumpler-Str. 9)
  - U6 bis Freimann (9 Minuten ab Universität), dann wenige Minuten zu Fuß
- Klausurtermin
  - Di 8.8. 17-20h im Hauptgebäude

QuickSort: Erklärung der Idee

Ziel:
Zerlegung (Partitionierung) des (Teil-)Arrays $a[l \ldots r]$ bzgl. eines *Pivot-Elementes* $a[k]$ in zwei Teilarrays $a[l \ldots k-1]$ und $a[k+1 \ldots r]$.

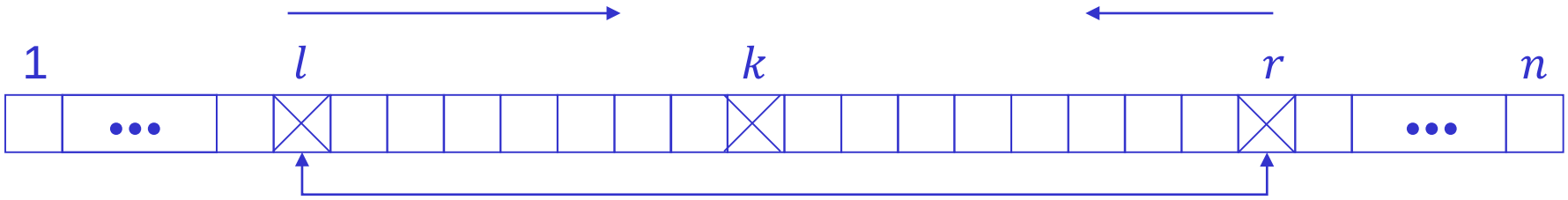


so dass gilt:

$$\forall i \in \{l, \ldots, k-1\}: \; a[i] \leq a[k]$$
$$\forall i \in \{k+1, \ldots, r\}: \; a[k] \leq a[j]$$

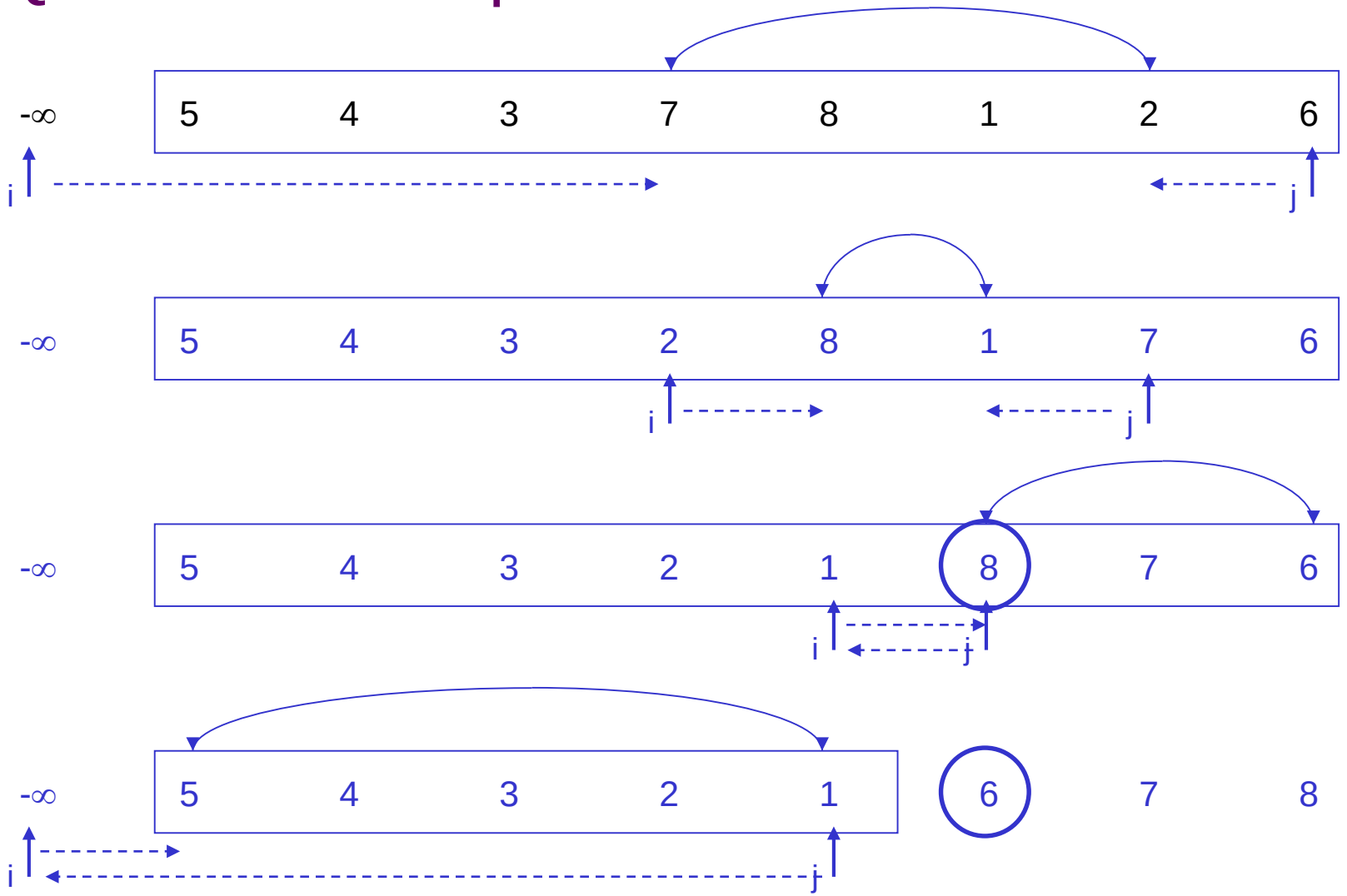
QuickSort: Erklärung der Idee

Methode:
Laufe mit zwei Indizes gegenläufig von l bzw. r zur Mitte
Tausche gegebenenfalls Schlüssel zwischen den Teilarrays aus

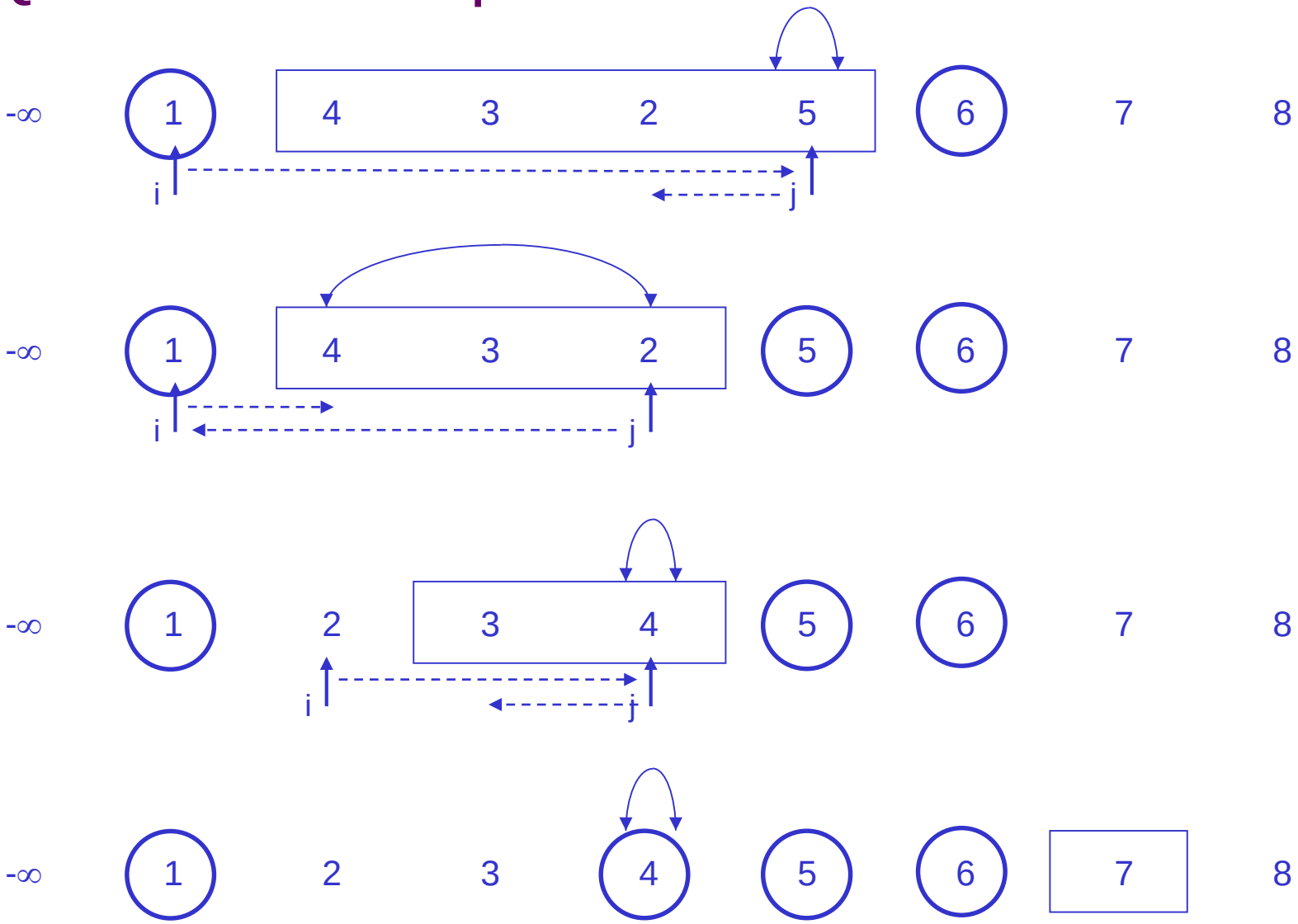


Rekursion:
linkes Teilarray $a[l \ldots k-1]$ und rechtes Teilarray $a[k+1 \ldots r]$ bearbeiten

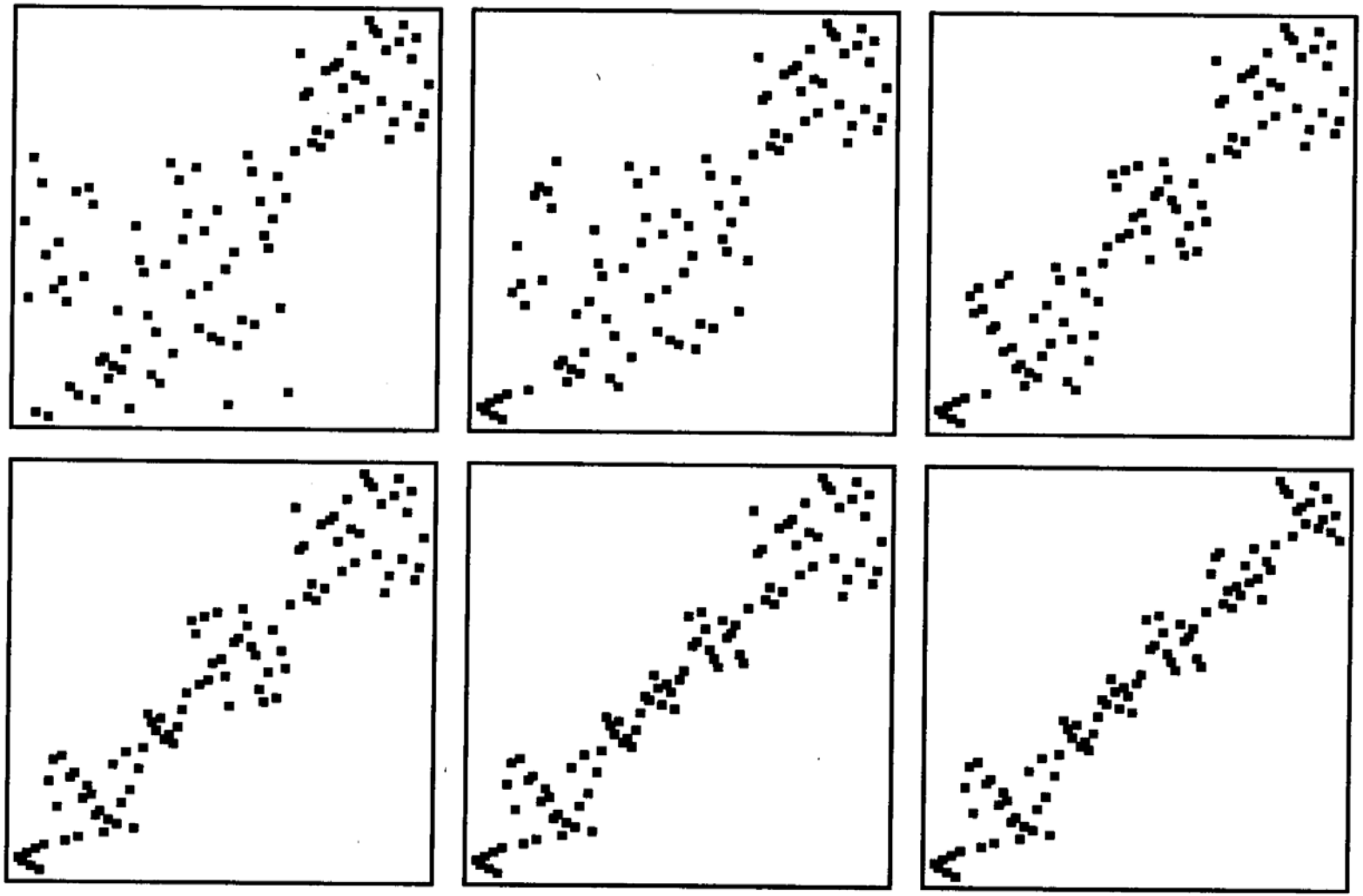
QuickSort: Beispiel



QuickSort: Beispiel



QuickSort: Visualisierung



QuickSort: Implementierung

```
static void quickSort (int[] a, int l, int r) {
    if (l < r) {
      int m = partition(a, l, r);
      quickSort(a, l, m-1);
      quickSort(a, m+1, r);
    }
}
```

Aufruf mit: `quickSort(a, 1, n);`

QuickSort: Implementierung

```
static int partition (int[] a, int l, int r){
    int i = l-1;
    int j = r;
    while (i < j) {
       i++; while (a[i] < a[ r ]) {++i;}
       j--; while (a[ j ] > a[ r ]) {--j;}
       swap(a, i, j);
    }
    swap(a, i, j);
    swap(a, i, r);

    return i;
}
```

Technischer Grund

QuickSort: Implementierung

- **Achtung**:

  Verwende **Sentinel**, um korrekte Terminierung der inneren Schleife zu garantieren.

- Aufruf mit:

  a[0] := -∞

  QuickSort(a, 1, n);

- while (a[i] < a[r]) i++;

  while (a[j] > a[r]) j--;

Bricht spätestens für j = r ab.

Bricht spätestens für j = l-1 ab.

QuickSort: Komplexitätsanalyse

Best Case: Folge zerfällt immer in zwei gleich große Teile

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) \quad \in O(n \log n)$$

Worst Case: Nur jeweils ein Element wird abgespalten. Bei Pivotwahl vorne oder hinten: falls die Folge bereits auf- oder absteigend sortiert ist:

$$T(n) = T(n-1) + O(n) \quad \in O(n^2)$$

Average Case:

$$T(n) \in O(n \log n)$$

Bei der Analyse fließen folgende Annahmen ein:

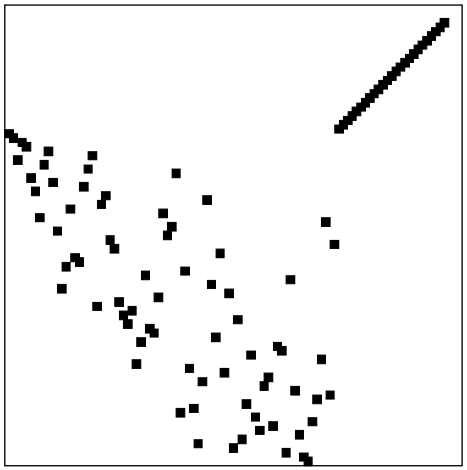
- Gleichverteilung für den Index des Pivotelements
  $\rightarrow$ Verwendung des arithmetischen Mittels für Zeiten für die Rekursionsaufrufe
- lineare Zeit für restliche Operationen

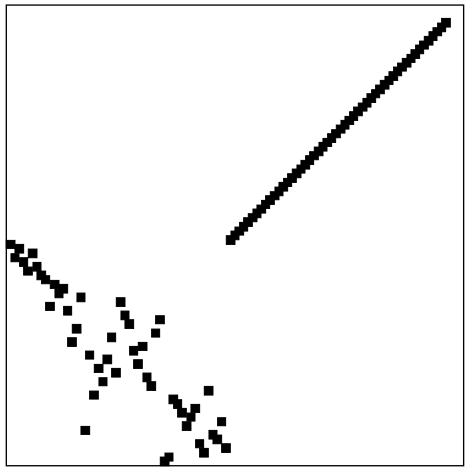
Dies führt zu folgendem Ansatz:

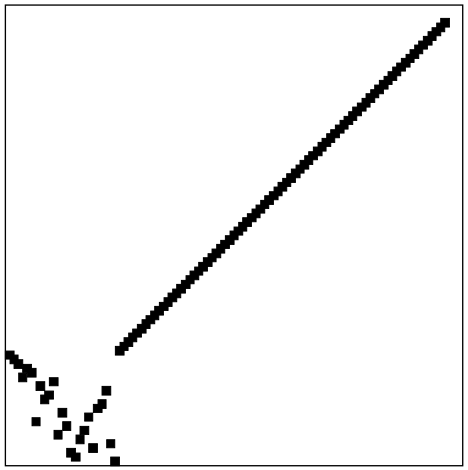
$$T(n) = O(n) + \frac{1}{n}\left(\sum_{i=0}^{n-1} T(i) + T(n-1-i)\right) = O(n) + \frac{2}{n} \cdot \sum_{i=0}^{n-1} T(i)$$

HeapSort: Verbessertes SelectionSort

- SelectionSort: Wähle n-mal das jeweils nächste Maximum in $O(n)$ Zeit
- Falls das immer in $O(\log n)$ ginge, hätten wir ein $O(n \log n)$-Verfahren
- Idee: Stelle das jeweils nächste Maximum in $O(\log n)$ ganz nach vorne







Heap: Struktur

- Ein Heap ist ein **links-vollständiger Binärbaum**, der in ein Array eingebettet ist und folgende Eigenschaft erfüllt:

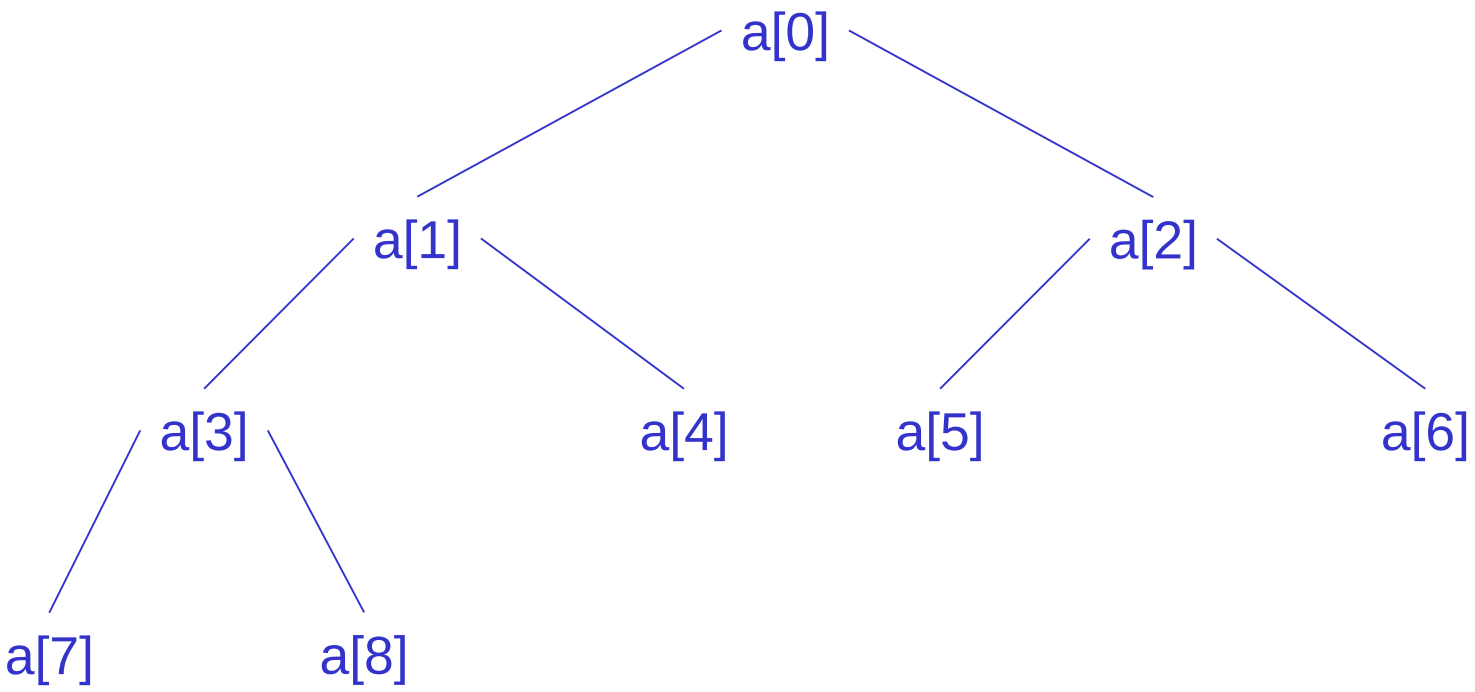
- Ein Array a erfüllt die Heap-Eigenschaft, falls gilt:

$$a\left[\left\lfloor\frac{i-1}{2}\right\rfloor\right] \geq a[i] \quad \text{für } i = 1, \ldots, a.\text{length} - 1$$

- Ein Array a ist ein Heap beginnend in Position $l = 0, \ldots, a.\text{length} - 1$, falls:

$$a\left[\left\lfloor\frac{i-1}{2}\right\rfloor\right] \geq a[i] \quad \text{für } i = 2l - 1, \ldots, a.\text{length} - 1$$

Heap: Beispiel für Einbettung



Folgerungen der Heapeigenschaft

- In einem Heap ist der Schlüssel jedes inneren Knotens größer ist als die Schlüssel seiner Kinder.

- Insbesondere gilt damit für einen Heap im Array:

$$a[0] = \max\{a[i]\colon\ i = 0, \ldots, a.\text{length} - 1\}$$

- Außerdem gilt folgendes Korollar:

  Jedes Array $a[1 \ldots n]$ ist ein Heap beginnend in Position

$$l = \left\lfloor \frac{a.\text{length}}{2} \right\rfloor + 1$$

HeapSort: Ablauf

1. Heap-Aufbau-Phase

   Wandle das Array a in einen Heap um.

2. Sortierphase

   for i := 0 to (a.length – 2) do
   a) Tausche Wurzel a[0] und aktuell letztes Element a[a.length – i]
   b) Stelle für das Rest-Array a[0 … (a.length – i – 1)] die Heap-Eigenschaft wieder her (durch Versickern)

HeapSort: Beispiel für Vertauschen

Ausgangssituation:

10	14	7	17	3	21	11	18

→ Heap

Vertausche 17 ↔ 18

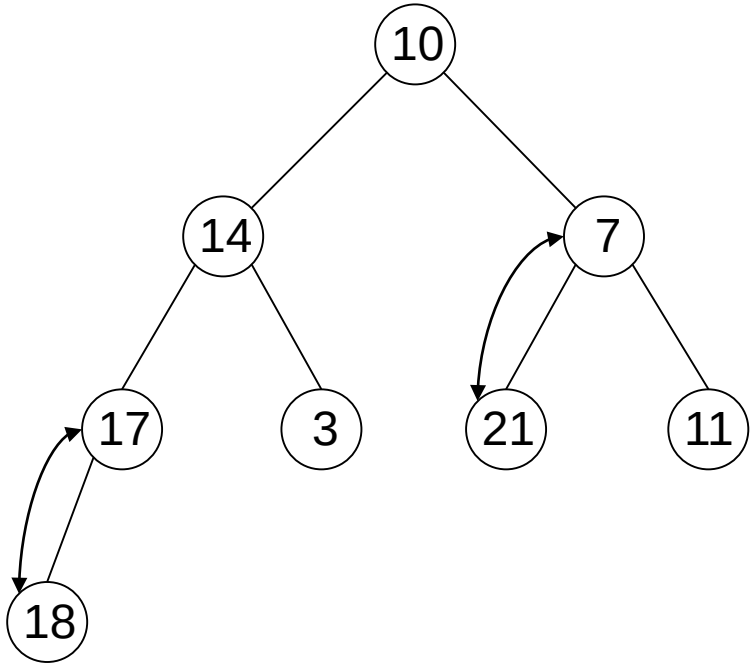
10	14	7	18	3	21	11	17

→ Heap

Vertausche 7 ↔ 21

10	18	21	17	3	7	11	14

→ Heap



HeapSort: Beispiel für Versickern

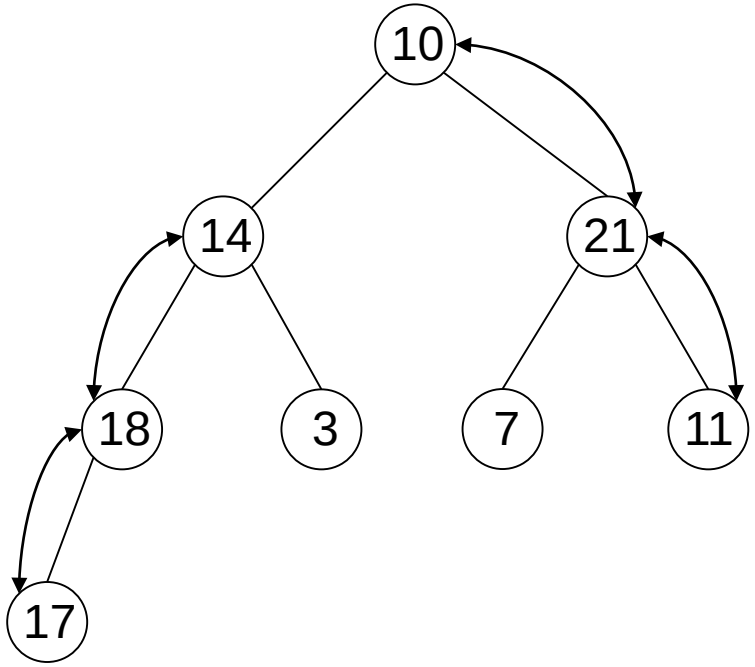
Versickere 14 ↔ 18 ↔ 17:

10	18	21	17	3	7	11	14

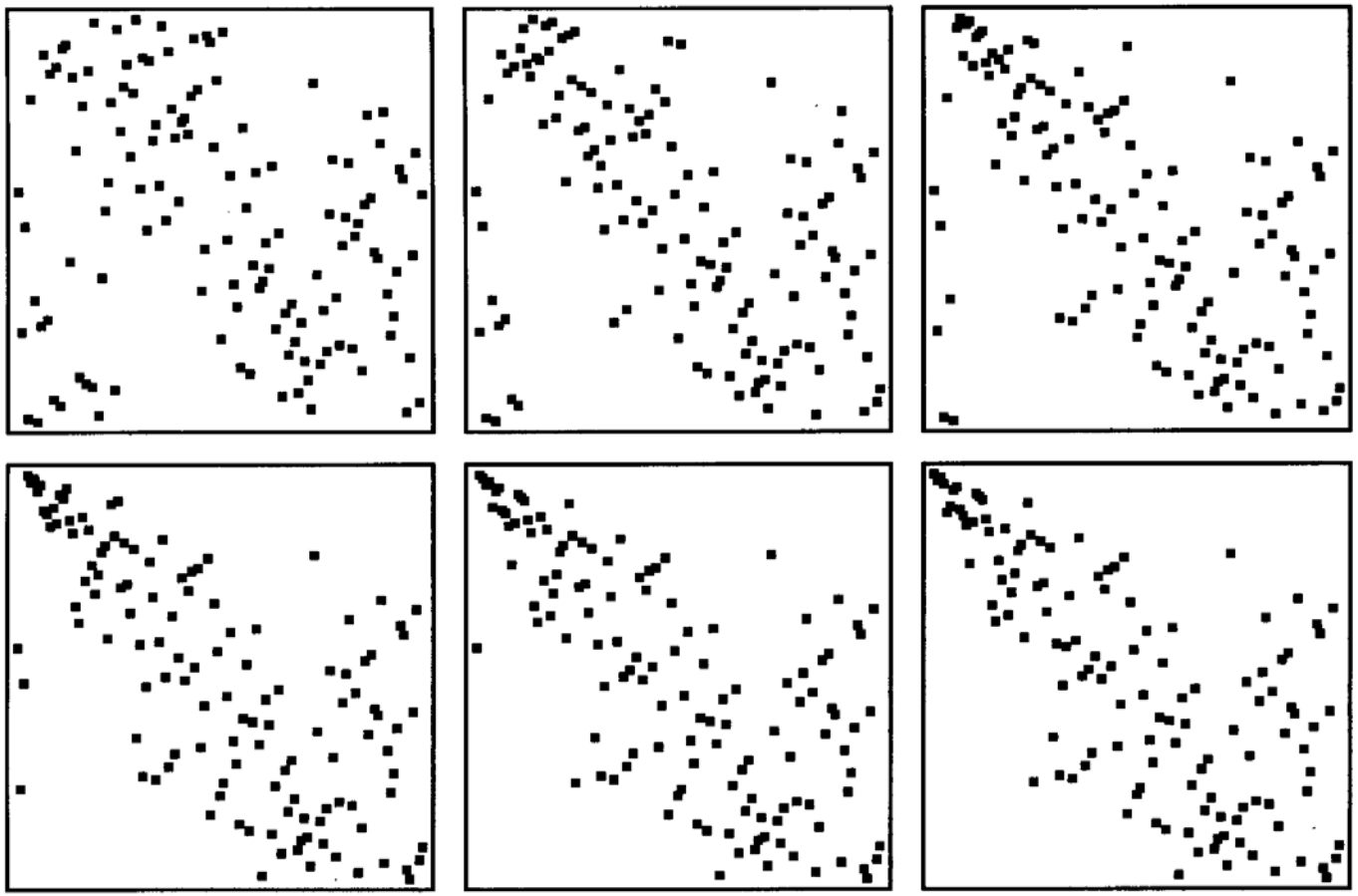
→ Heap

Versickere zuletzt 10 ↔ 21 ↔ 11:

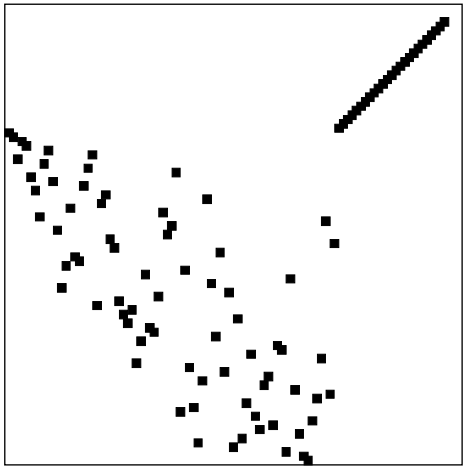
21	14	11	17	3	7	10	14

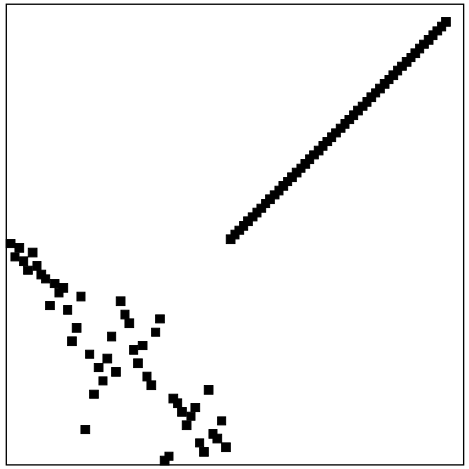


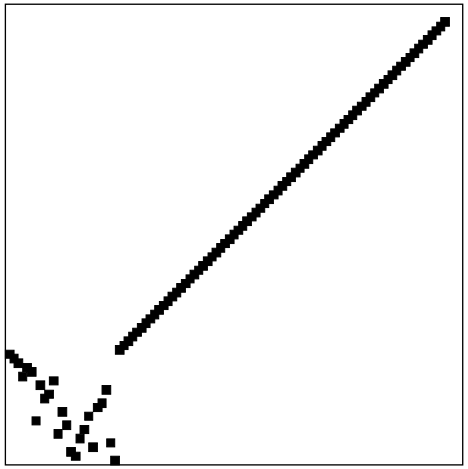
HeapSort: Aufbau des Heaps



HeapSort: Sortierphase







HeapSort: Implementierung Heapaufbau

```
static void convertToHeap (int[] a, int n) {
    for (int i = n/2;  i >= 0; --i) {
        sink(a, n, i);
    }
}
```

Wenn die Schleife bis i = p durchgelaufen ist, hat das Array a ab Index p die Heap-Eigenschaft.

HeapSort: Implementierung Versickern

```
static void sink (int[] a, int n, int i){
    while (i < n/2){
      int j = 2*i;
      if ((j < n) && (a[j+1] > a[j])) j++;
      if (a[j] > a[i]) {
        swap(a, j, i);
        i = j;
      } else {
        i = n;
      }
    }
}
```

HeapSort: Implementierung (gesamt)

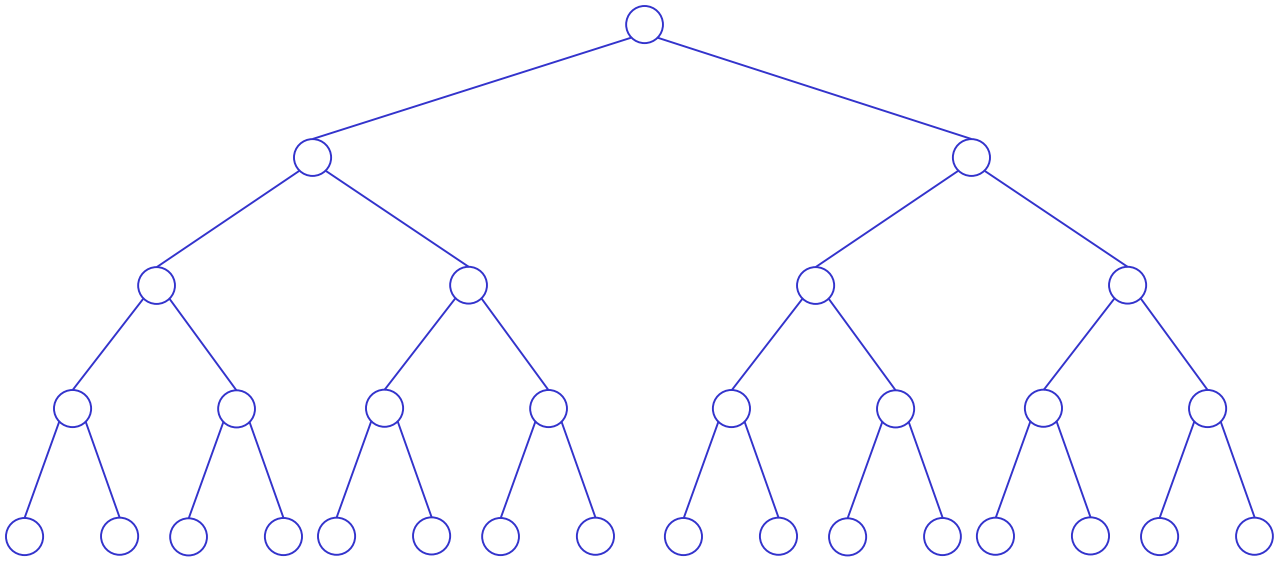
```
static void heapSort(int[] a){
    convertToHeap(a, a.length – 1);

    for (int i = a.length – 1; i >= 0; --i){
      swap(a, 0, i);
      sink(a, i, 0);
    }
}
```

HeapSort: Komplexitätsanalyse

Wir betrachten die Anzahl der Vergleiche, um ein Array der Größe $n = 2^k - 1, k \in \mathbb{N}$ zu sortieren.

Zur Veranschaulichung betrachten wir die Analyse exemplarisch für ein Array der Größe $n = 2^5 - 1 = 31$ (also k = 5).



i = 0: $2^0 = 1$ Knoten

i = 1: $2^1 = 2$ Knoten

i = 2: $2^2 = 4$ Knoten

i = 3: $2^3 = 8$ Knoten

i = 4: $2^4 = 16$ Knoten

HeapSort: Komplexitätsanalyse

Heap-Aufbau für Array mit $n = 2^k - 1$ Knoten

- Auf der Ebene $i, i = 0, \ldots, k-1,$ gibt es 2^i Knoten.
- Wir fügen ein Element auf dem Niveau $i = (k-2), (k-3), \ldots, 0$ hinzu
- Dieses Element kann maximal auf Niveau $k-1$ sinken.
- Pro Niveau werden dazu höchstens zwei Vergleiche benötigt:
  1. Vergleich: IF a[j]<=v THEN ...
     Wird bei jedem Durchlauf der WHILE-Schleife ausgeführt, die ihrerseits bei jedem Prozeduraufruf DownHeap(i, k, a) mindestens einmal durchlaufen wird.
  2. Vergleich: IF a[j]<a[j+1] THEN ...
     Wird ausgeführt, falls zweites Kind existiert.

Für die Gesamtzahl der Vergleiche ergibt sich damit die obere Schranke:

$$\sum_{i=0}^{k-2} 2 \cdot (k-1-i) = 2^{k+1} - 2(k+1)$$

Beweis durch vollständige Induktion über k.

HeapSort: Komplexitätsanalyse

Sortierphase

• Nach Aufbau des Heaps muß noch die endgültige Ordnung auf dem Array hergestellt werden

– Dazu wird ein Knoten der Tiefe i auf die Wurzel gesetzt
– Dieser Knoten kann mit DownHeap maximal um i Niveaus sinken
– Pro Niveau sind hierzu höchstens zwei Vergleiche erforderlich
– Damit ergibt sich für die Anzahl der Vergleiche die obere Schranke:

$$\sum_{i=0}^{k-1} 2i \cdot 2^i = 2(k-2)\cdot 2^k + 4$$

Beweis durch vollständige Induktion über k

HeapSort: Komplexitätsanalyse

Zusammen

Sei $n = 2^k - 1$, dann gilt für die Anzahl $T(n)$ der Vergleiche:

$$T(n) \leq 2^{k+1} - 2(k+1) + 2(k-2) \cdot 2^k + 4 = 2k \cdot \left(2^k - 1\right) - 2 \cdot \left(2^k - 1\right)$$
$$= 2n \cdot ld(n+1) - 2n \in O(n \log n)$$

Für $n \neq 2^k - 1$ erhält man ein ähnliches Ergebnis. Die Rechnung gestaltet sich jedoch umständlicher.

Resultat

HeapSort sortiert jede Folge a[1..n] mit höchstens $2n \cdot ld(n+1) - 2n \in O(n \log n)$ Vergleichen.

Kapitel 2 - Sortieren

Elementare Sortierverfahren
SelectionSort
InsertionSort
BubbleSort

Höhere Sortierverfahren
MergeSort
QuickSort
HeapSort

Untere und obere Schranken für das Sortierproblem

Spezielle Sortierverfahren
CountingSort

Untere und obere Schranken für das Sortierproblem

bisher:

Komplexität eines Algorithmus

jetzt:

Komplexität eines Problems (Aufgabenstellung)

Ziel:

$T_A(n)$:= Zahl der Schlüsselvergleiche, um eine n-elementige Folge von Schlüsselelementen mit Algorithmus A zu sortieren.

$T_{min}(n)$:= Zahl der Vergleiche für den effizientesten Algorithmus

Suche nach einer unteren Schranke

Gibt es ein $T_0(n)$, so dass

$$T_0(n) \leq T_A(n) \qquad \forall A$$

gilt, d.h. jeder *denkbare* Algorithmus braucht in diesem Fall mindestens $T_0(n)$ Vergleiche?

Suche nach einer oberen Schranke

Wir wählen einen (möglichst effizienten) Sortieralgorithmus A mit Komplexität $T_A(n)$.

Sprechweise:

$T_A(n)$ Vergleiche reichen, um jedes Sortierproblem zu lösen.

Untere und obere Schranke zusammen

$$T_0(n) \leq T_{min}(n) \leq T_A(n)$$

Wunsch:

$T_0(n)$ und $T_A(n)$ sollen möglichst eng zusammen liegen.

Konkret:

- Im folgenden betrachten wir für das Sortierproblem nur *Vergleichsoperationen*
  - – Beschränkung auf Algorithmen, die Wissen über die Anordnung der Eingabefolge allein durch (binäre) Vergleichsoperationen erhalten

Obere Schranke

Wir wählen als effizienten Algorithmus MergeSort.

$$
\begin{aligned}
T_A(n) &= n \lceil \mathrm{ld}\, n \rceil - 2^{\lceil \mathrm{ld}\, n \rceil} + 1 \\
&\leq n \lceil \mathrm{ld}\, n \rceil - n + 1
\end{aligned}
$$

Untere Schranke

Gegeben sei eine n-elementige Folge.

Sortieren: entspricht Auswahl einer Permutation dieser Folge

Es gibt n! Permutationen, aus denen die „richtige“ auszuwählen ist.

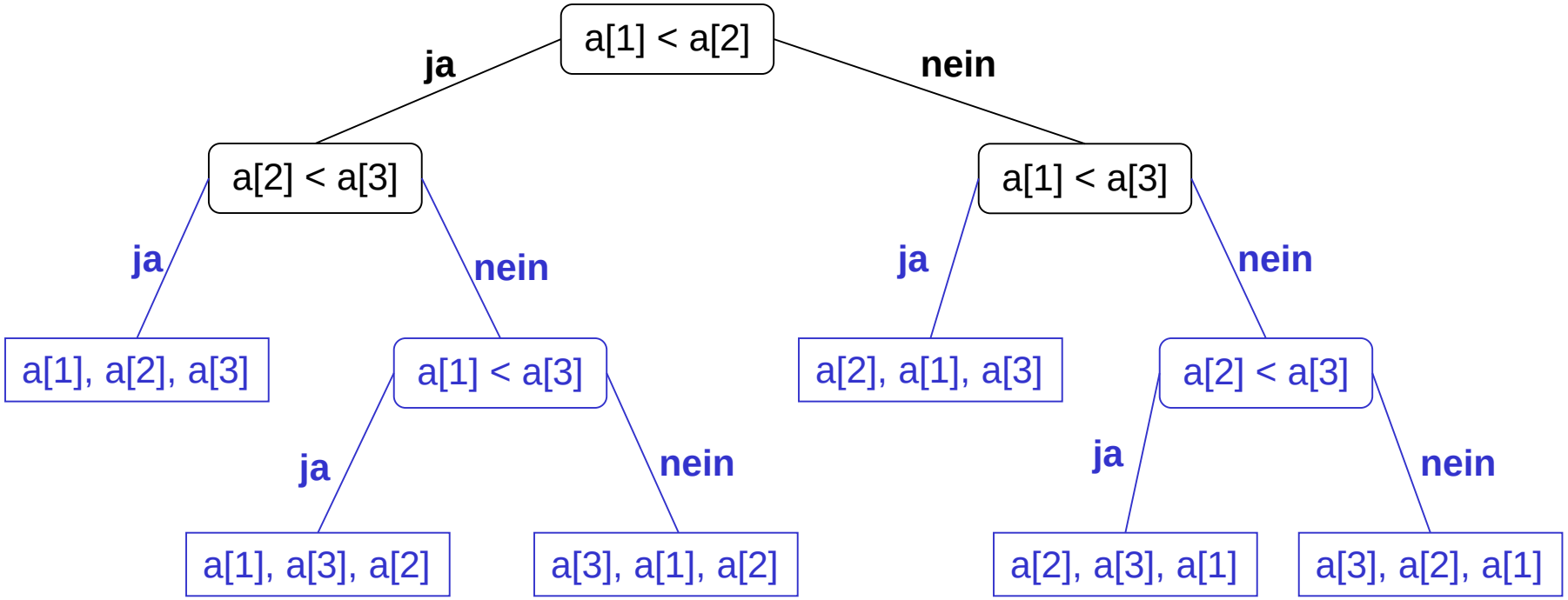
Entscheidungsbaum

Der Ablauf eines nur auf Vergleichen basierenden Sortieralgorithmus kann durch einen Entscheidungsbaum dargestellt werden:

- innerer Knoten entspricht Vergleich
- Blatt entspricht Permutation

Entscheidungsbaum - Beispiel

Der folgende binäre Entscheidungsbaum „sortiert“ ein 3-elementiges Array a[1...3]. Da 3! = 6, muß der Entscheidungsbaum 6 Blätter besitzen. Wegen ld 6 ≈ 2.58 existiert in dem Baum mindestens ein Pfad der Länge 3.



Entscheidungsbaum

Ein binärer Entscheidungsbaum für das Sortierproblem besitzt genau n! Blätter. Damit ergibt sich als untere Schranke für das Sortierproblem:

$$\text{ld } n! \leq \lceil \text{ld } n! \rceil \leq T_0(n)$$

Mit der Ungleichung (Beweis: siehe folgende Folien)

$$n \text{ ld } n - n \text{ ld } e \leq \text{ld } n!$$

erhalten wir das Ergebnis:

$$n \text{ ld } n - n \text{ ld } e \leq T_{min}(n) \leq n \lceil \text{ld } n \rceil - n + 1$$

Einfache Schranken für n!

Obere Schranke:

$$n! = \prod_{i=1}^{n} i \;\leq\; \prod_{i=1}^{n} n = n^n$$

Untere Schranke:

$$n! = \prod_{i=1}^{n} i \;\geq\; \prod_{i=\lceil \frac{n}{2} \rceil}^{n} i \;\geq\; \prod_{i=\lceil \frac{n}{2} \rceil}^{n} \left\lceil \frac{n}{2} \right\rceil \;\geq\; \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

Zusammen:

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} \;\leq\; n! \;\leq\; n^n$$

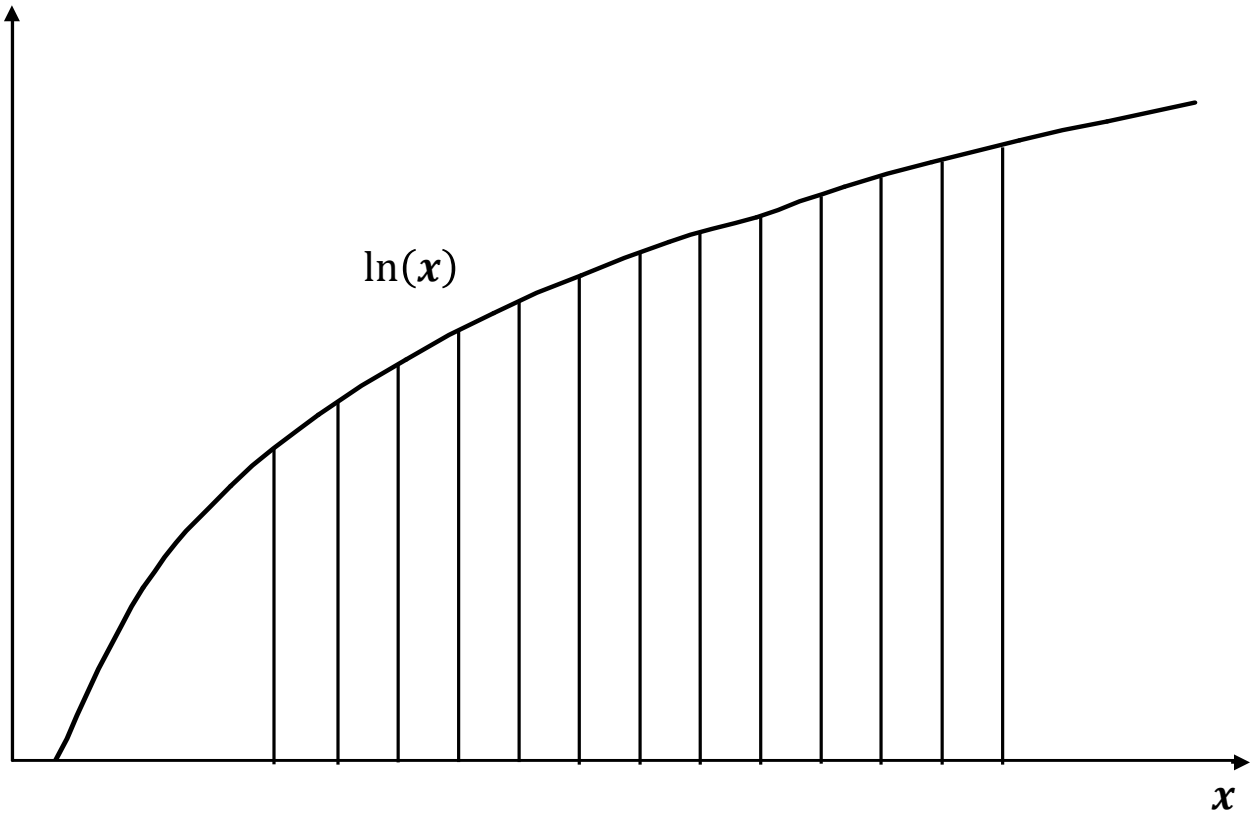
Integral-Methode

- Engere Schranken für $n!$ mittels der *Integral-Methode*
  – das Flächenintegral monotoner und konvexer Funktionen wird von oben und unten durch Trapezsummen approximiert.

- Für die Fakultätsfunktion gilt:

$$\ln(n!) = \ln \prod_{i=1}^{n} i = \sum_{i=1}^{n} \ln i$$

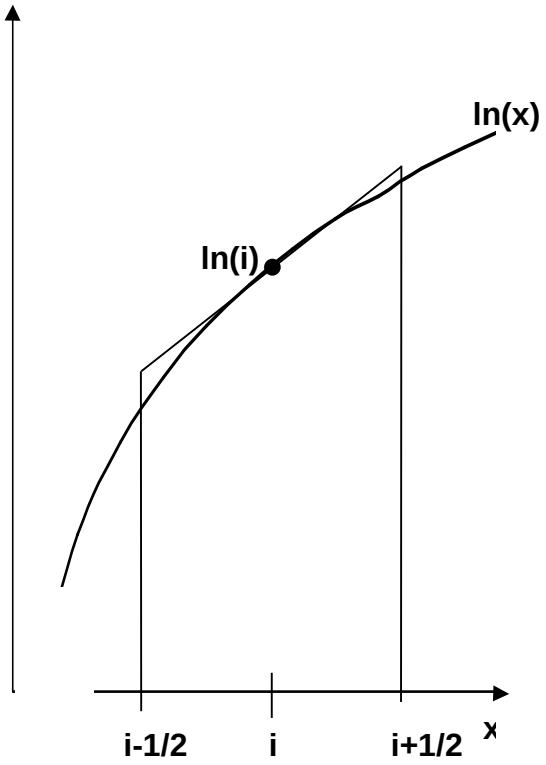
Graph zu $\ln(x)$



Untere Schranke

Die Logarithmusfunktion $x \to f(x) := \ln(x)$ ist monoton und konvex.
Daher:

$$\int_{i-\frac{1}{2}}^{i+\frac{1}{2}} \ln(x)\, dx \;\leq\; \ln(i)$$



Untere Schranke für n!

Summation von $i = 1, \ldots, n$ ergibt:

$$\int_{\frac{1}{2}}^{n+\frac{1}{2}} \ln(x)\, dx \;\leq\; \sum_{i=1}^{n} \ln(i) \;=\; \ln(n!)$$

Mit $\int_a^b \ln(x)\, dx = x \cdot \ln(x) - x|_a^b$ folgt die untere Schranke für $\ln(n!)$:

$$\ln(n!) \geq \left(n + \frac{1}{2}\right) \cdot \ln\left(n + \frac{1}{2}\right) - n - \frac{1}{2} - \frac{1}{2} \cdot \ln\left(\frac{1}{2}\right) + \frac{1}{2}$$
$$= \mathrm{n} \cdot \ln\left(n + \frac{1}{2}\right) + \ln\left(n + \frac{1}{2}\right) - n \cdot \ln(e) + \frac{1}{2} \cdot \ln(2)$$
$$\geq n \cdot \ln(n) - n \cdot \ln(e)$$

→ $n\ ld\ n - n\ ld\ e \;\leq\; ld\ n!$ (wie vorne genutzt)

Obere Schranke für n!

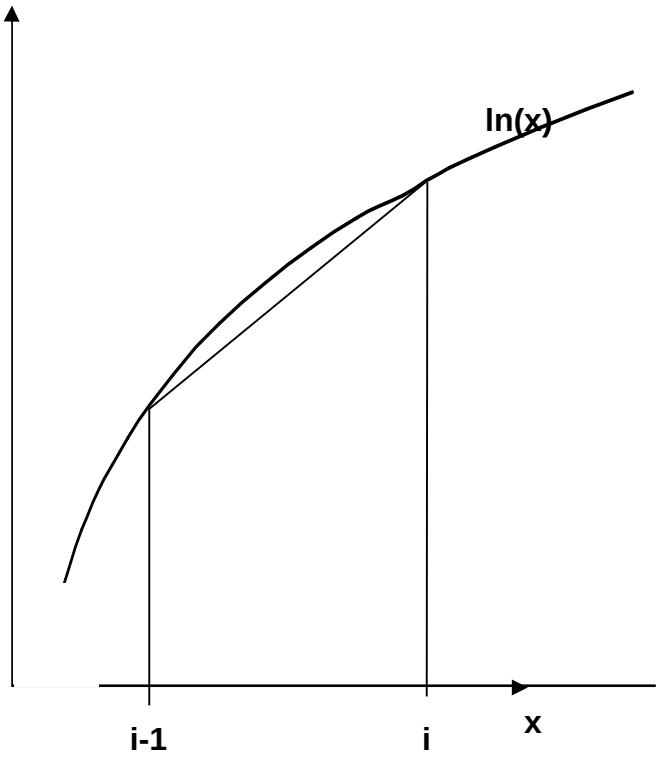
Das Integral über ln(x) in den Grenzen i-1 und i ist eine obere Schranke der zugehörigen Trapezuntersumme:

$$\frac{1}{2} \cdot [\ln(i-1) + \ln(i)] \leq \int_{i-1}^{i} \ln(x)\, dx$$

Summation von i = 2, ..., n ergibt:

$$\ln(n!) - \frac{1}{2}\ln(n) \leq \int_{1}^{n} \ln(x)\, dx = n \cdot \ln(n) - n\ln(e) + 1$$

$$\Rightarrow \quad \ln(n!) \leq n \cdot \ln\left(\frac{n}{e}\right) + \frac{1}{2} \cdot \ln(n) + 1$$



Beide Schranken zusammen

Aus den vorherigen Berechnungen ergibt sich

$$n \cdot \ln\left(\frac{n}{e}\right) + \frac{1}{2} \cdot \ln(n) + \frac{1}{2} \cdot \ln(2) \leq \ln(n!) \leq n \cdot \ln\left(\frac{n}{e}\right) + \frac{1}{2} \cdot \ln(n) + 1$$

$$\Leftrightarrow \quad \ln\left(\left(\frac{n}{e}\right)^n\right) + \ln(\sqrt{n}) + \ln(\sqrt{2}) \leq \ln(n!) \leq \ln\left(\left(\frac{n}{e}\right)^n\right) + \ln(\sqrt{n}) + \ln(e)$$

Und somit:

$$\left(\frac{n}{e}\right)^n \sqrt{2n} \leq n! \leq \left(\frac{n}{e}\right)^n \sqrt{2n}\,\frac{e}{\sqrt{2}}$$

Stirlingsche Formel

Es gilt:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\Theta\left(\frac{1}{n}\right)}$$

mit

$$\frac{1}{12n+1} < \Theta\left(\frac{1}{n}\right) < \frac{1}{12n}$$

Der Fehler $\Theta(1/n)$ strebt sehr schnell gegen 0:

$$\lim_{n\to\infty} \Theta\left(\frac{1}{n}\right) = 0$$

Kapitel 2 - Sortieren

Elementare Sortierverfahren
SelectionSort
InsertionSort
BubbleSort

Höhere Sortierverfahren
MergeSort
QuickSort
HeapSort

Untere und obere Schranken für das Sortierproblem

Spezielle Sortierverfahren
CountingSort

Spezielle Sortierverfahren: CountingSort

Voraussetzung:

Schlüssel sind darstellbar als ganzzahlige Werte im Bereich 0, ..., M-1, sodass sie als Array-Index verwendet werden können.

$$a[i] \in \{0, ..., M - 1\} \quad \forall\, i=1,\ldots,n$$

Arbeitsweise:

1. Erstelle ein Histogramm, d.h. zähle für jeden Schlüsselwert, wie häufig er vorkommt.
2. Berechne aus dem Histogramm die Position für jeden Record.
3. Bewege die Records an ihre errechnete Position.

CountingSort: Beispiel

- 4 Schlüsselwerte: A, B, C und D
- Array mit 14 Einträgen:

A	B	B	A	C	A	D	A	B	B	A	D	D	A

1. Zählen der Häufigkeit jedes Schlüssels

A	B	C	D
6	4	1	3

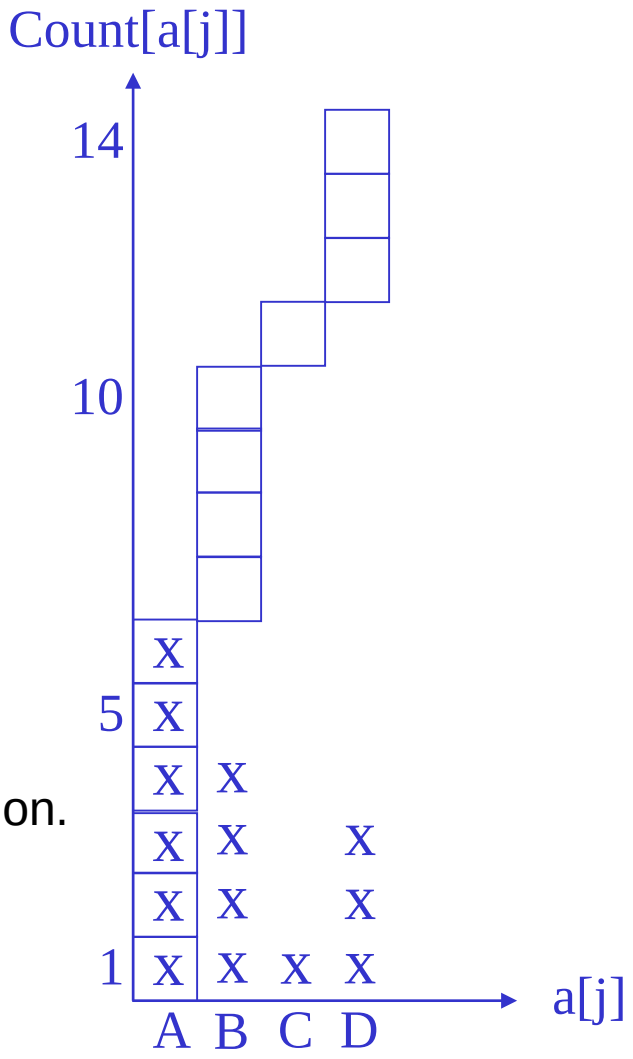
2. Berechnen der höchsten Position jedes Eintrags

A	B	C	D
6	10	11	14

3. Einfügen der Elemente an der berechneten Position.

A	A	A	A	A	A	B	B	B	B	C	D	D	D

Rückwärtiges Durchlaufen gewährleistet Stabilität



CountingSort: Implementierung

```
static int[] countingSort(int[] array, int keysCount){
   int[] histogram = new int [keysCount];
   for (int i = 0; i < array.length ; i++){
     histogram[ array[i] ]++;
   }
   for (int i = 1; i < keysCount; i++){
     histogram[i] += histogram[i-1]
   }
   int[] result = int [array.length]
   for (int i = array.length; i >= 0; i--){
     result [histogram[array[i]]] = array[i];
     histogram [array[i]]--;
  }
}
```

O(n)

O(M)

O(n)

CountingSort hat eine Zeitkomplexität von $T(n) \in O(n + M) = O(\max\{ n, M \})$

CountingSort: Komplexität

- Komplexität von CountingSort:

  $$T(n) \in O(n + M) = O(\max\{ n, M \})$$

- Für kleine Schlüsselmengen ist CountingSort besser als die von uns bewiesene Schranke! Wie ist das möglich?

- CountingSort beruht **nicht** auf **Vergleichen**.

- Beim Beweis der unteren Schranke haben wir vorausgesetzt, dass Elemente durch Vergleichen sortiert werden.

BucketSort

- Für große oder nicht endliche Anzahl von Schlüsseln
- Aufteilung der Schlüssel in Gruppen (Buckets)
- Grobe Sortierung der Elemente anhand der Gruppen
- Algorithmus entsprechend CountingSort

- Beispiele:
  - Sortierung von Klausuren anhand der ersten Ziffern der Matrikelnummer
  - Sortierung von Briefen anhand der ersten zwei Ziffern der PLZ
  - Sortierung nach Namen anhand des Anfangsbuchstaben

Kapitel 2 - Sortieren

Elementare Sortierverfahren
SelectionSort
InsertionSort
BubbleSort

Höhere Sortierverfahren
MergeSort
QuickSort
HeapSort

Untere und obere Schranken für das Sortierproblem

Spezielle Sortierverfahren
CountingSort