

# 2 Sortieren

## 2.1 Einführung

Das Sortieren von Datensätzen ist ein wichtiger Bestandteil von vielen Anwendungen. Laut IBM werden in kommerziellen Rechenanlagen 25% der Rechenzeit für das Sortieren aufgewendet [Mehlhorn 88]. Die Effizienzansprüche an die Algorithmen sind dementsprechend hoch. Als Einstieg in die Problematik werden in diesem Kapitel zunächst einige grundlegende, sog. *elementare Sortierverfahren* vorgestellt und untersucht.

### Elementare Sortierverfahren:

- SelectionSort (Sortieren durch Auswahl)
- InsertionSort (Sortieren durch Einfügen)
- BubbleSort
- BucketSort

Elementare Sortierverfahren sind dadurch charakterisiert, daß sie im Mittel eine (in der Größe der Eingabe) quadratische Laufzeit besitzen<sup>1</sup>. Im Unterschied hierzu weisen die *höheren Sortierverfahren* eine im Mittel überlineare Laufzeit  $O(N \log N)$  auf:

### Höhere Sortierverfahren:

- MergeSort
- QuickSort
- HeapSort

### Konvention

Für die nachfolgenden Betrachtungen sei stets eine Folge  $a[1], \dots, a[N]$  von Datensätzen (Records) gegeben. Jeder Datensatz  $a[i]$  besitzt eine Schlüsselkomponente  $a[i].key$  ( $i = 1, \dots, N$ ). Darüber hinaus können die Datensätze weitere Informationseinheiten

---

<sup>1</sup>BucketSort nimmt hier einen Sonderstatus ein. Zwar vermag das Verfahren eine Schlüsselreihe in nur linearer Zeit zu sortieren, jedoch muß die Schlüsselmenge zusätzliche Eigenschaften besitzen, die bei einem allgemeinen Sortierverfahren nicht gefordert sind.

(z.B. Name, Adresse, PLZ, etc.) enthalten. Die Sortierung erfolgt ausschließlich nach der Schlüsselkomponente *key*. Hierzu muß auf der Menge aller Schlüssel eine *Ordnung* definiert sein.

### Definition: Sortierproblem

Gegeben sei eine Folge  $a[1], \dots, a[N]$  von Datensätzen (Records) mit einer Schlüsselkomponente  $a[i].key$  ( $i = 1, \dots, N$ ) und eine Ordnung  $\leq$  auf der Menge aller Schlüssel. Das *Sortierproblem* besteht darin, eine Permutation  $\pi$  der ursprünglichen Folge zu bestimmen, so daß gilt:

$$a[\pi_1].key \leq a[\pi_2].key \leq \dots \leq a[\pi_{N-1}].key \leq a[\pi_N].key$$

### Beispiele:

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

**Beachte:** Da wir es mit einer Folge (und keiner Menge) von Datensätzen zu tun haben, können Schlüssel oder ganze Datensätze mehrfach auftreten.

### Unterscheidungskriterien

Sortieralgorithmen können nach verschiedenen Kriterien klassifiziert werden:

- Sortiermethode
- Effizienz:  $O(N^2)$  für elementare Sortierverfahren,  $O(N \log N)$  für höhere Sortierverfahren
- intern (alle Records im Arbeitsspeicher) oder extern (Platten)
- direkt/indirekt (d.h. mit Pointern oder Array-Indizes)
- im Array oder nicht
- in situ (d.h. in einem einzigen Array ohne zusätzliches Hilfsfeld) oder nicht
- allgemein/speziell  
(z.B. fordert BucketSort zusätzliche Eigenschaften der Schlüsselmenge)
- stabil (Reihenfolge von Records mit gleichem Schlüssel bleibt erhalten) oder nicht

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/größten Elemente

Weitere (hier jedoch nicht behandelte) Sortierverfahren sind:

- BinaryInsertionSort
- ShellSort
- ShakerSort (Variante von BubbleSort)
- MergeSort (vgl. Abschnitt 1.4.1)

### Deklarationsteil

Die folgende Typdeklaration ist exemplarisch für die zu sortierenden Datensätze:

```
TYPE KeyType = REAL;
TYPE ItemType = RECORD
    . . .
    key : KeyType;
END;

VAR a : ARRAY[1..N] OF ItemType;
```

Für `KeyType` können auch andere Datentypen auf denen eine Ordnung definiert ist verwendet werden (z.B. `INTEGER`). Da stets nur nach der Schlüsselkomponente `key` sortiert wird, werden wir zur Vereinfachung der Algorithmen Felder vom Typ `KeyType` verwenden:

```
VAR a : ARRAY[1..N] OF KeyType;
```

## 2.2 Elementare Sortierverfahren

### 2.2.1 SelectionSort

Sortieren durch Auswahl

Gegeben sei eine Folge  $a[1], \dots, a[N]$  von Schlüsselementen.

**Prinzip:** Wir betrachten den  $i$ -ten Durchgang der Schleife  $i = 1, \dots, N$ :

- Bestimme den Datensatz mit dem kleinsten Schlüssel aus  $a[i], \dots, a[N]$
- vertausche dieses Minimum mit  $a[i]$

Vorteil von SelectionSort: Jeder Datensatz wird höchstens einmal bewegt.

**Achtung:** Es existieren Varianten von SelectionSort mit anderer Anzahl von Vertauschungen.

**Programm:**

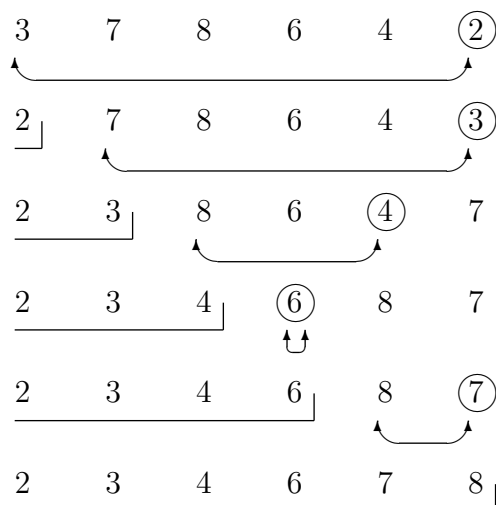
```

PROCEDURE SelectionSort (VAR a : ARRAY[1..N] OF KeyType) =
  VAR min : CARDINAL;
      t   : KeyType;
  BEGIN
    FOR i := 1 TO N-1 DO
      min := i;
      FOR j := i+1 TO N DO
        IF a[j] < a[min] THEN min := j; END;
      END;
      t := a[min]; a[min] := a[i]; a[i] := t;
    END;
  END SelectionSort;

```

**Beispiel:**

Markiert ist jeweils das kleinste Element  $a[\text{min}]$  der noch unsortierten Teilfolge.



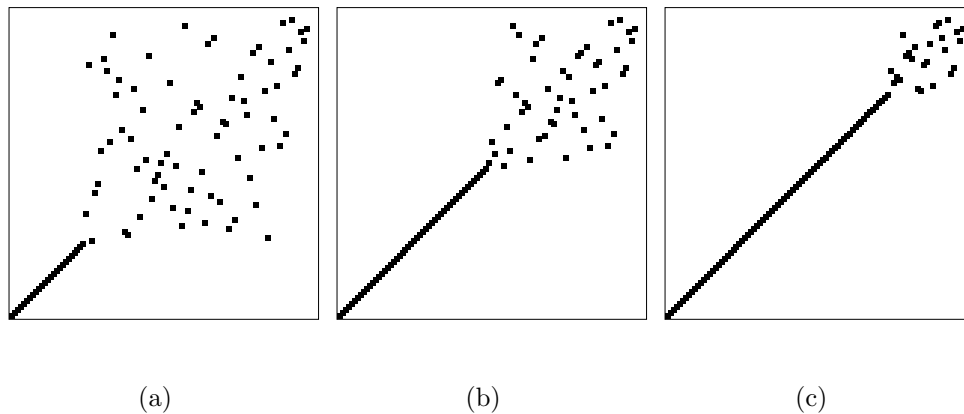


Abbildung 2.1: SelectionSort einer zufälligen Permutation von  $N$  Schlüsselementen, nachdem (a)  $N/4$ , (b)  $N/2$  und (c)  $3N/4$  der Schlüssel sortiert worden sind.

### Komplexitätsanalyse

Zum Sortieren der gesamten Folge  $a[1], \dots, a[N]$  werden  $N - 1$  Durchläufe benötigt. Pro Schleifendurchgang  $i$  gibt es eine Vertauschung, die sich aus je drei Bewegungen und  $N - i$  Vergleichen zusammensetzt, wobei  $N - i$  die Anzahl der noch nicht sortierten Elemente ist. Insgesamt ergeben sich:

- $3 \cdot (N - 1)$  Bewegungen und
- $(N - 1) + (N - 2) + \dots + 2 + 1 = \frac{N \cdot (N - 1)}{2}$  Vergleiche.

Da die Anzahl der Bewegungen nur linear in der Anzahl der Datensätze wächst, ist SelectionSort besonders für Sortieraufgaben geeignet, in denen die einzelnen Datensätze sehr groß sind.

### 2.2.2 InsertionSort

Sortieren durch Einfügen.

Gegeben sei eine Folge  $a[1], \dots, a[N]$  von Schlüsselementen.

**Prinzip:** Wir betrachten den  $i$ -ten Durchgang der Schleife  $i = 2, \dots, N$ . Dabei sei die  $i - 1$ -elementige Teilfolge  $a[1], \dots, a[i - 1]$  bereits sortiert:

- Füge den Datensatz  $a[i]$  an der korrekten Position der bereits sortierten Teilfolge  $a[1], \dots, a[i - 1]$  ein.

**Programm:**

```

PROCEDURE InsertionSort (VAR a : ARRAY[0..N] OF KeyType) =
  VAR j : CARDINAL;
      v : KeyType;
  BEGIN
    a[0] :=  $-\infty$ ; (* in Modula-3: a[0]:=FIRST(KeyType); *)
    FOR i := 2 TO N DO
      v := a[i];
      j := i;
      WHILE a[j-1]>v DO
        a[j] := a[j-1];
        DEC(j);
      END;
      a[j] := v;
    END;
  END InsertionSort;

```

**Beachte:** Um in der WHILE-Schleife eine zusätzliche Abfrage auf die linke Feldgrenze zu vermeiden, wird ein sog. *Sentinel-Element* (=Wärter; Anfangs- oder Endmarkierung)  $a[0] := -\infty$  verwendet.

**Bemerkung:** Auf die Verwendung eines *Sentinel-Elementes* kann bei *nicht-strikter* Auswertung (auch *lazy evaluation*) des Compilers verzichtet werden:

```

      WHILE 1<j AND a[j-1]>v DO

```

Dabei wird zunächst nur der linke Teil  $1<j$  der Abbruchbedingung ausgewertet. Liefert die Auswertung den Wert **FALSE**, so wird die Schleife verlassen, ohne daß der rechte Teilausdruck  $a[j-1]>v$  ausgewertet wird. (Sprechweise: „Der Operator **AND** ist *nicht-strikt*“ bzw. „**AND** ist ein *short-circuit operator*“) Die Auswertung des rechten Teilausdrucks erfolgt nur dann, wenn das linke Argument von **AND** zu **TRUE** ausgewertet werden kann.

Bei *striker* Übersetzung des Compilers kann die Abbruchbedingung der WHILE-Schleife auch innerhalb einer LOOP-Anweisung formuliert werden:

```

  LOOP
    IF 1<j THEN
      IF a[j-1]>v THEN
        a[j] := a[j-1];
        DEC(j);
      ELSE EXIT; END;
    ELSE EXIT; END;
  END;

```

**Beispiel:**

InsertionSort durchläuft die zu sortierende Folge von links nach rechts. Dabei ist die Anfangsfolge des Feldes zwar in sich sortiert, die Elemente befinden sich jedoch noch nicht notwendigerweise an ihrer endgültigen Position.

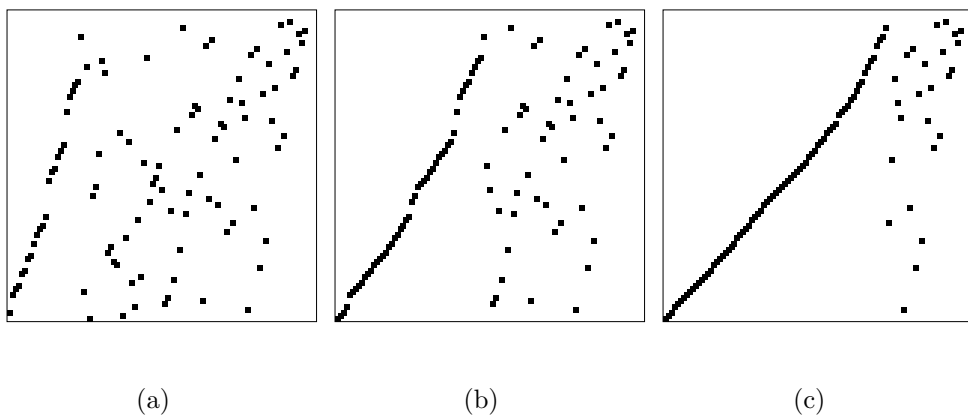
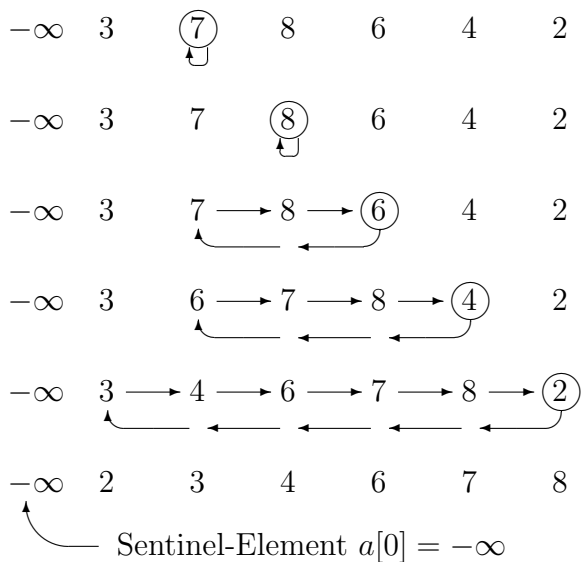


Abbildung 2.2: InsertionSort einer zufälligen Permutation von  $N$  Schlüsselementen, nachdem (a)  $N/4$ , (b)  $N/2$  und (c)  $3N/4$  der Schlüssel sortiert worden sind.

**Komplexitätsanalyse**

**Vergleiche:** Das Einfügen des Elements  $a[i]$  in die bereits sortierte Anfangsfolge  $a[1], \dots, a[i - 1]$  erfordert mindestens einen Vergleich, höchstens jedoch  $i$  Vergleiche.

Im Mittel sind dies  $i/2$  Vergleiche, da bei zufälliger Verteilung der Schlüssel die Hälfte der bereits eingefügten Elemente größer ist 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:

$$\begin{aligned} \sum_{i=2}^N i &= \left( \sum_{i=1}^N i \right) - 1 = \frac{N(N+1)}{2} - 1 \\ &= \frac{N^2}{2} + \frac{N}{2} - 1 \end{aligned}$$

- *Average Case* – Die Anzahl der Vergleiche ist etwa  $N^2/4$

**Bewegungen:** Im Schleifendurchgang  $i$  ( $i = 2, \dots, N$ ) wird bei bereits sortierter Anfangsfolge  $a[1], \dots, a[i-1]$  das einzufügende Element  $a[i]$  zunächst in die Hilfsvariable  $v$  kopiert (eine Bewegung) und anschließend mit höchstens  $i$  Schlüsseln (beachte das Sentinel-Element), wenigstens jedoch mit einem Schlüssel und im Mittel mit  $i/2$  Schlüsseln verglichen. Bis auf das letzte Vergleichselement werden die Datensätze um je eine Position nach rechts verschoben (jeweils eine Bewegung). Anschließend wird der in  $v$  zwischengespeicherte Datensatz an die gefundene Position eingefügt (eine Bewegung). Für den Schleifendurchgang  $i$  sind somit mindestens zwei Bewegungen, höchstens jedoch  $i + 1$  und im Mittel  $i/2 + 2$  Bewegungen erforderlich. Damit ergibt sich insgesamt der folgende Aufwand:

- *Best Case* – Bei vollständig vorsortierten Folgen  $2(N - 1)$  Bewegungen
- *Worst Case* – Bei umgekehrt sortierten Folgen  $N^2/2$  Bewegungen
- *Average Case* –  $\sim N^2/4$  Bewegungen

Für „fast sortierte“ Folgen verhält sich InsertionSort nahezu linear. Im Unterschied zu SelectionSort vermag InsertionSort somit eine in der zu sortierenden Datei bereits vorhandene Ordnung besser auszunutzen.

### 2.2.3 BubbleSort

Sortieren durch wiederholtes Vertauschen unmittelbar benachbarter Array-Elemente.

Gegeben sei eine Folge  $a[1], \dots, a[N]$  von Schlüsselementen.

**Prinzip:** Wir betrachten den  $i$ -ten Durchgang der Schleife  $i = N, N - 1, \dots, 2$ :

- Schleife  $j = 2, 3, \dots, i$ : ordne  $a[j - 1]$  und  $a[j]$

**Programm:**

```

PROCEDURE BubbleSort (VAR a : ARRAY[1..N] OF KeyType) =
  VAR t : KeyType;
BEGIN
  FOR i:= N TO 1 BY -1 DO
    FOR j:= 2 TO i DO
      IF a[j-1]>a[j] THEN
        t := a[j-1];
        a[j-1] := a[j];
        a[j] := t;
      END;
    END;
  END;
END BubbleSort;

```

Durch wiederholtes Vertauschen von unmittelbar benachbarten Schlüsselementen wandern die größeren Schlüssel nach und nach an das rechte Ende des zu sortierenden Feldes. Nach jedem Durchgang der äußersten Schleife nimmt das größte Element der noch unsortierten Teilfolge seine endgültige Position im Array ein. Dabei wird zwar im Verlauf der Sortierung auch der Ordnungsgrad aller noch unsortierten Schlüsselemente der Folge erhöht, jedoch ist BubbleSort nicht in der Lage, hieraus für die weitere Sortierung einen Nutzen zu ziehen.

**Beispiel:**

```

3   7   8   6   4   2
      ^-----^ ^-----^
3   7   6   4   2   |8
      ^-----^ ^-----^
3   6   4   2   |7   8
      ^-----^
3   4   2   |6   7   8
      ^-----^
3   2   |4   6   7   8
^-----^
2   |3   4   6   7   8

```

**Komplexitätsanalyse**

**Vergleiche:** Die Anzahl der Vergleiche ist unabhängig vom Vorsortierungsgrad der

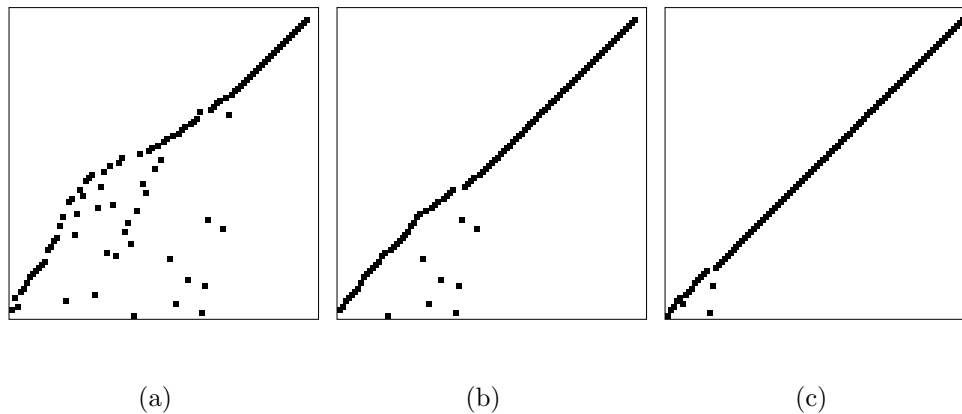


Abbildung 2.3: BubbleSort einer zufälligen Permutation von  $N$  Schlüsselementen, nachdem (a)  $N/4$ , (b)  $N/2$  und (c)  $3N/4$  der Schlüssel sortiert worden sind.

Folge. Daher sind der *worst case*, *average case* und *best case* identisch, denn es werden stets alle Elemente der noch nicht sortierten Teilfolge miteinander verglichen. Im  $i$ -ten Schleifendurchgang ( $i = N, N - 1, \dots, 2$ ) enthält die noch unsortierte Anfangsfolge  $N - i + 1$  Elemente, für die  $N - i$  Vergleiche benötigt werden.

Um die ganze Folge zu sortieren, sind  $N - 1$  Schritte erforderlich. Die Gesamtzahl der Vergleiche wächst damit quadratisch in der Anzahl der Schlüsselemente:

$$\begin{aligned} \sum_{i=1}^{N-1} (N - i) &= \sum_{i=1}^{N-1} i \\ &= \frac{N(N - 1)}{2} \end{aligned}$$

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

- im *Best Case*: 0 Bewegungen
- im *Worst Case*:  $\sim \frac{3N^2}{2}$  Bewegungen
- im *Average Case*:  $\sim \frac{3N^2}{4}$  Bewegungen.

**Vergleich elementarer Sortierverfahren**

Anzahl der Vergleiche elementarer Sortierverfahren:

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$N^2/2$	$N^2/2$	$N^2/2$
InsertionSort	$N$	$N^2/4$	$N^2/2$
BubbleSort	$N^2/2$	$N^2/2$	$N^2/2$

Anzahl der Bewegungen elementarer Sortierverfahren:

Verfahren	Best Case	Average Case	Worst Case
SelectionSort	$3(N - 1)$	$3(N - 1)$	$3(N - 1)$
InsertionSort	$2(N - 1)$	$N^2/4$	$N^2/2$
BubbleSort	0	$3N^2/4$	$3N^2/2$

**Folgerungen:**

BubbleSort: ineffizient, da immer  $N^2/2$  Vergleiche

InsertionSort: gut für fast sortierte Folgen

SelectionSort: gut für große Datensätze aufgrund konstanter Zahl der Bewegungen, jedoch stets  $N^2/2$  Vergleiche

**Fazit:** InsertionSort und SelectionSort sollten nur für  $N \leq 50$  eingesetzt werden. Für größere  $N$  sind höhere Sortierverfahren wie z.B. QuickSort und HeapSort besser geeignet.

## 2.2.4 Indirektes Sortieren

Bei sehr großen Datensätzen kann das Vertauschen der Datensätze (=Records) den Rechenaufwand für das Sortieren dominieren. In solchen Fällen ist es sinnvoll, statt der Datensätze selbst nur Verweise auf diese zu sortieren.

### Verfahren:

1. Verwende ein Index-Array  $p[1..N]$ , das mit  $p[i] := i$  ( $i = 1, \dots, N$ ) initialisiert wird
2. für Vergleiche erfolgt der Zugriff auf einen Record mit  $a[p[i]]$
3. Vertauschen der Indizes  $p[i]$  statt der Array-Elemente  $a[p[i]]$
4. optional werden nach dem Sortieren die Records selbst umsortiert (Aufwand:  $O(N)$ )

### Programm:

```

PROCEDURE InsertionSort_Indirect (VAR a : ARRAY[0..N] OF KeyType;
                                VAR p : ARRAY[0..N] OF [0..N]) =
  VAR j, v : CARDINAL;
  BEGIN
    a[0] :=  $-\infty$ ;
    FOR i := 0 TO N DO
      p[i] := i;
    END;

    FOR i := 2 TO N DO
      v := p[i];
      j := i;
      WHILE a[p[j-1]] > a[v] DO
        p[j] := p[j-1];
        DEC(j);
      END;
      p[j] := v;
    END;
  END InsertionSort_Indirect;

```

Mit Hilfe der indirekten Sortierung kann jedes Sortierverfahren so modifiziert werden, daß nicht mehr als  $N$  Record-Vertauschungen nötig sind.

**Umsortierung der Datensätze:**

Sollen die Datensätze im Anschluß an die indirekte Sortierung selbst umsortiert werden, so gibt es hierzu zwei mögliche Varianten:

- (a) Permutation mit zusätzlichem Array  $b[1 : N]$ :  $b[i] := a[p[i]] \quad (i = 1, \dots, N)$
- (b) Permutation ohne zusätzliches Array: *in situ, in place* (lohnt nur bei großen Records):

Ziel:  $p[i] = i \quad (i = 1, \dots, N)$

- falls  $p[i] = i$ : nichts zu tun;
- sonst: zyklische Vertauschung durchführen:

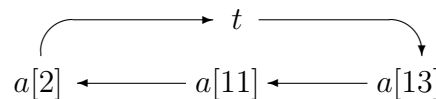
1. kopiere Record:  $t := a[i]$

Ergebnis: Platz (Loch) an Position  $i$ ;

2. „iterieren“

Beispiel:  $t = a[2]; \quad a[2] = a[11]; \quad a[11] = a[13]; \quad a[13] = t;$

Ersetzungsreihenfolge des Ringtausches:



**Beispiel:** Indirektes Sortieren

Vor dem Sortieren:

$i$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$a[i]$	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
$p[i]$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Nach dem indirekten Sortieren:  $a[p[i]] \leq a[p[i + 1]]$

$i$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$a[i]$	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
$p[i]$	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

Durch Permutation mittels  $p[i]$  ergibt sich das sortierte Array:  $b[i] := a[p[i]]$

$i$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$b[i]$	A	A	E	E	G	I	L	M	N	O	P	R	S	T	X
$p[i]$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Das folgende Programm führt eine in-situ-Permutation durch wiederholte verkettete Ringtauschoperationen aus:

**Programm:**

```

PROCEDURE InsituPermutation (VAR a : ARRAY[1..N] OF KeyType;
                             VAR p : ARRAY[1..N] OF CARDINAL) =
  VAR t : KeyType;
      j, k : CARDINAL;
  BEGIN
    FOR i := 1 TO N DO
      IF p[i] <> i THEN
        t := a[i];
        k := i;
        REPEAT
          j := k; a[j] := a[p[j]];
          k := p[j]; p[j] := j;
        UNTIL k=i;
        a[j] := t;
      END;
    END;
  END InsituPermutation;

```

**2.2.5 BucketSort**

**Andere Namen:** Bin Sorting, Distribution Counting, Sortieren durch Fachverteilen, Sortieren mittels Histogramm

**Voraussetzung:** Schlüssel können als ganzzahlige Werte im Bereich  $0, \dots, M - 1$  dargestellt werden, so daß sie als Array-Index verwendet werden können.

$$a[i] \in \{0, \dots, M - 1\} \quad \forall i = 1, \dots, N$$

**Prinzip:**

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 (mit rückläufigem Index) an ihre errechnete Position.

**Programm:**

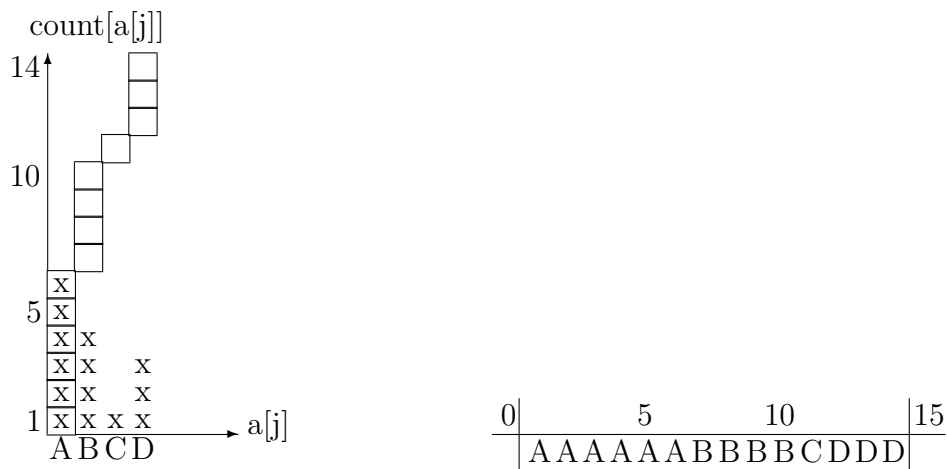
```

TYPE KeyType = [0..M-1];

PROCEDURE BucketSort (VAR a : ARRAY[1..N] OF KeyType) =
  VAR count : ARRAY KeyType OF CARDINAL;
      b      : ARRAY [1..N] OF KeyType;
  BEGIN
    FOR j := 0 TO M-1 DO      (* Initialisierung *)
      count[j] := 0;
    END;
    FOR i := 1 TO N DO      (* Erstelle Histogramm *)
      count[a[i]] := count[a[i]] + 1;
    END;
    FOR j := 1 TO M-1 DO    (* Berechne Position für jeden Schlüsselwert *)
      count[j] := count[j-1] + count[j];
    END;
    FOR i := N TO 1 BY -1 DO (* Bewege Record an errechnete Position *)
      b[count[a[i]]] := a[i];
      count[a[i]] := count[a[i]] - 1;
    END;
    FOR i := 1 TO N DO
      a[i] := b[i];
    END;
  END BucketSort;
  
```

**Beispiel:**

ABBAC|ADABB|ADDA



**Eigenschaften:**

- Wegen des rückläufigen Index ist BucketSort stabil.
- Für die Zeit- und Platzkomplexität gilt:

$$\begin{aligned} T(N) &= O(N + M) \\ &= O(\max\{N, M\}) \end{aligned}$$

- BucketSort arbeitet in dieser Form **nicht** in situ.
- Eine in-situ Variante ist möglich, jedoch verliert man dabei die Stabilität.

## 2.3 QuickSort

QuickSort wurde 1962 von C.A.R. Hoare entwickelt.

**Prinzip:** Das Prinzip folgt dem Divide-and-Conquer-Ansatz:

Gegeben sei eine Folge  $F$  von Schlüsselementen.

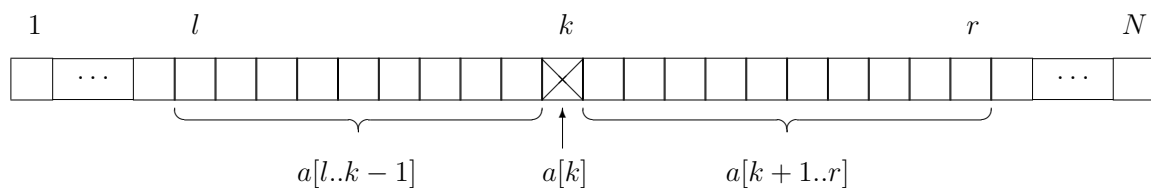
1. Zerlege  $F$  bzgl. eines partitionierenden Elementes (engl.: *pivot* = Drehpunkt)  $p \in F$  in zwei Teilfolgen  $F_1$  und  $F_2$ , so daß gilt:

$$\begin{aligned} x_1 &\leq p && \forall x_1 \in F_1 \\ p &\leq x_2 && \forall x_2 \in F_2 \end{aligned}$$

2. Wende dasselbe Schema auf jede der so erzeugten Teilfolgen  $F_1$  und  $F_2$  an, bis diese nur noch höchstens ein Element enthalten.

QuickSort realisiert diese Idee folgendermaßen:

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





Um die Folge  $F$  in 1-elementige Teilfolgen zu zerlegen, werden  $\lg N$  Schritte benötigt. Für das Vertauschen der Schlüsselemente sind in jedem Schritt  $N$  Vergleiche notwendig, so daß sich insgesamt

$$T(N) = N \cdot \lg N$$

Vergleiche ergeben.

Das folgende Programm bildet das Grundgerüst des Quicksort-Algorithmus. Die Parameter  $l$  und  $r$  definieren die linke bzw. rechte Feldgrenze der zu sortierenden Teilfolge.

### Programm:

```
PROCEDURE QuickSort(l, r : CARDINAL) =
  VAR k : CARDINAL;
  BEGIN
    IF l < r THEN
      k := Partition(l, r);
      QuickSort(l, k-1);
      QuickSort(k+1, r);
    END;
  END QuickSort;
```

Die Prozedur `Partition(l, r)` muß die folgende Eigenschaft besitzen:

Sei  $a[k]$  das Pivot-Element, dann werden die Elemente im Array  $a[l..r]$  so umsortiert, daß die folgende Bedingung erfüllt ist:

$$\begin{aligned} & \forall i \in \{l, \dots, k-1\} : a[i] \leq a[k] \\ \text{und } & \forall j \in \{k+1, \dots, r\} : a[k] \leq a[j] \end{aligned}$$

Als Konsequenz ergibt sich hieraus, daß für  $k := \text{Partition}(l, r)$  das Pivot-Element  $a[k]$  bereits seine endgültige Position im Array eingenommen hat.

Beachte, daß sich das Pivot-Element im allgemeinen erst **nach** der Umsortierung durch den Partitionierungsschritt an der Position  $k$  befindet.

**Algorithmus:**

Der folgende Algorithmus beschreibt informell die Funktionsweise der Partition-Prozedur.

```

PROCEDURE Partition(l, r : CARDINAL) : CARDINAL =
  Beachte: a[0]= $-\infty$  wird als Sentinel verwendet.
  BEGIN
    i := l-1;
    j := r;
    wähle Pivot-Element: v:=a[r];

    REPEAT
      durchsuche Array von links (i:=i+1), solange bis a[i]  $\geq$  v;
      durchsuche Array von rechts (j:=j-1), solange bis a[j]  $\leq$  v;
      vertausche a[i] und a[j];
    UNTIL j  $\leq$  i (* Zeiger kreuzen *)

    rückvertausche a[i] und a[j];
    vertausche a[i] und a[r]; (* positioniere Pivot-Element *)

    RETURN i; (*  $\hat{=}$  endgültige Position des Pivot-Elements *)
  END Partition;

```

**Programm:**

```

PROCEDURE Partition(l, r : CARDINAL) : CARDINAL =
  VAR i, j : CARDINAL;
      v, t : KeyType;
  BEGIN
    i := l-1;
    j := r;
    v := a[r]; (* wähle Pivot-Element *)

    REPEAT
      REPEAT INC(i) UNTIL a[i]>=v;
      REPEAT DEC(j) UNTIL a[j]<=v; (* mit a[0]= $-\infty$  als Sentinel *)
      t := a[i]; a[i] := a[j]; a[j] := t; ❶
    UNTIL j<=i; (* Zeiger kreuzen *)

    (* Rückvertauschung und Positionierung des Pivot-Elements *)
    a[j] := a[i]; ❷
    a[i] := a[r]; ❸
    a[r] := t; ❹
  END Partition;

```

```

RETURN i;
END Partition;

```

### Anmerkungen zum QuickSort-Algorithmus:

(a) **Allgemeine Warnung:**

In der Literatur sind zahlreiche Varianten des QuickSort-Algorithmus zu finden, die sich in der Wahl des Pivot-Elementes und der Schleifenkontrolle unterscheiden.

(b) In der hier vorgestellten Implementierung wird in jeder der beiden Schleifen

```

REPEAT INC(i) UNTIL a[i]>=v;
REPEAT DEC(j) UNTIL a[j]<=v;

```

ein (explizites oder implizites) Sentinel-Element verwendet, d.h. für die Korrektheit der Schleifenkontrollen muß stets gewährleistet sein, daß es Elemente  $a[l - 1]$  und  $a[r]$  gibt, so daß für das Pivot-Element  $v$  gilt:

$$a[l - 1] \leq v \leq a[r]$$

Die folgende Fallunterscheidung zeigt, daß diese Forderung stets erfüllt ist.

**Obere Grenze:** Wegen  $v := a[r]$  gilt auch  $v \leq a[r]$

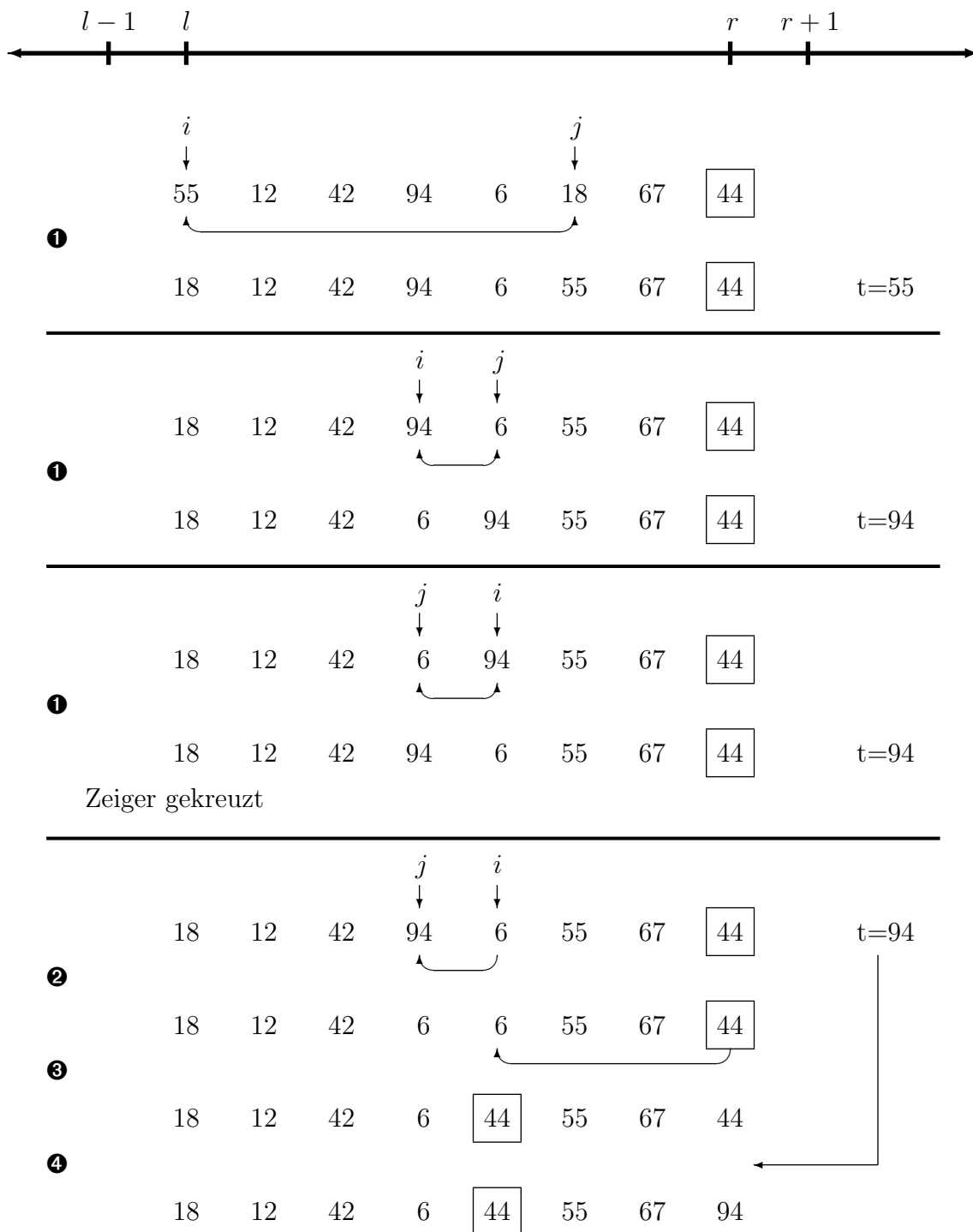
**Untere Grenze:** falls  $l = 1$ :  $a[0] := -\infty$   
 falls  $l > 1$ : Das Element  $a[l - 1]$  existiert aufgrund der Konstruktion des QuickSort-Algorithmus, da das Teilarray  $a[1..l-1]$  vor dem Array  $a[l..r]$  abgearbeitet wird. Dadurch befindet sich das Element  $a[l-1]$  bereits an seiner endgültigen Position und es gilt  $a[l - 1] \leq a[i]$  für  $i = l, \dots, N$ . Insbesondere ist dann auch  $a[l - 1] \leq v$ .

(c) Beim Kreuzen der Zeiger  $i$  und  $j$  wird eine Vertauschung zuviel durchgeführt, die nach Abbruch der äußeren Schleife rückgängig zu machen ist:

- Mache die letzte Vertauschung wieder rückgängig:  
vertausche  $a[i]$  und  $a[j]$
- Positioniere das Pivot-Element:  
vertausche  $a[i]$  und  $a[r]$

Diese Anweisungen werden in den Zeilen ②, ③ und ④ realisiert.

**Beispiel:** Die folgende Abbildung illustriert die Funktionsweise der Prozedur **Partition** für ein Array in den Grenzen  $l, \dots, r$ . Der Schlüssel 44 ist das Pivot-Element:



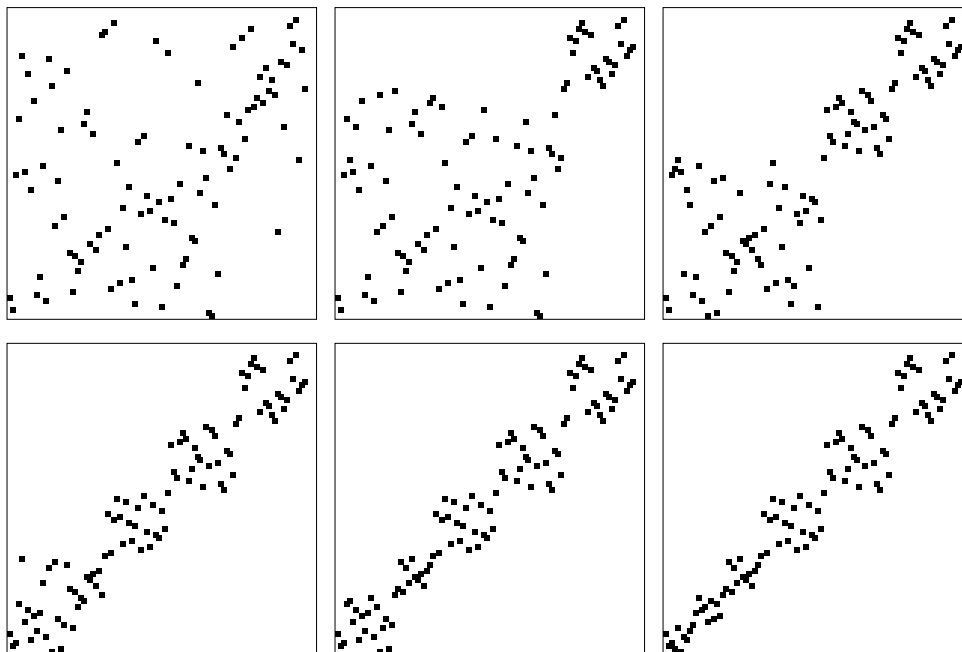


Abbildung 2.4: Anordnung der Schlüsselemente bei der Sortierung durch QuickSort nach 0, 1, 2, 3, 4 bzw. 5 Partitionierungen. Jede Partition enthält mindestens 10 Elemente.

**Zeitkomplexität:**

**Vergleiche:** Wird mit  $T(N)$  die Zahl der Vergleiche für Teilarrays der Größe  $N$  bezeichnet, dann gilt:

1. *Best Case* – Exakte Halbierung jedes Teilarrays:

$$T(N) = (N + 1) + \underbrace{\min_{1 \leq k \leq N} \{ T(k - 1) + T(N - k) \}}_{\text{günstigster Fall}}$$

$$= (N + 1) + 2 \cdot T\left(\frac{N + 1}{2}\right)$$

Lösung:

$$T(N) = (N + 1) \cdot \text{ld}(N + 1)$$

2. *Worst Case* – Ungünstigste Aufteilung der Teilarrays:

$$T(0) = T(1) = 0$$

$$T(N) = (N + 1) + \underbrace{\max_{1 \leq k \leq N} \{ T(k - 1) + T(N - k) \}}_{\text{ungünstigster Fall}}$$

Dann gilt:

$$T(N) \leq \frac{(N + 1) \cdot (N + 2)}{2} - 3$$

Beweis durch Vollständige Induktion über  $N$

Diese Schranke ist scharf, denn für ein aufsteigend sortiertes Array ohne Duplikate (d.h. ohne Mehrfachvorkommen gleicher Schlüssel) gilt:

$$\begin{aligned} T(N) &= (N + 1) + N + (N - 1) + \dots + 3 \\ &= \frac{(N + 1) \cdot (N + 2)}{2} - 3 \end{aligned}$$

3. *Average Case* – Zur Bestimmung der mittleren Anzahl von Vergleichen wird über alle möglichen Pivot-Elemente  $a[k]$  ( $k = 1, \dots, N$ ) gemittelt:

$$\begin{aligned} N \geq 2 : \quad T(N) &= \frac{1}{N} \cdot \sum_{k=1}^N \left[ N + 1 + T(k - 1) + T(N - k) \right] \\ &= N + 1 + \frac{1}{N} \cdot \underbrace{\left[ \sum_{k=1}^N T(k - 1) + \sum_{k=1}^N T(N - k) \right]} \\ &= N + 1 + \frac{2}{N} \sum_{k=1}^N T(k - 1) \end{aligned}$$

Elimination der Summe:

$$N \cdot T(N) = N \cdot (N + 1) + 2 \cdot \sum_{k=1}^N T(k - 1) \quad \textcircled{1}$$

$$(N - 1) \cdot T(N - 1) = (N - 1) \cdot N + 2 \cdot \sum_{k=1}^{N-1} T(k - 1) \quad \textcircled{2}$$

$$N \cdot T(N) - (N - 1) \cdot T(N - 1) = 2 \cdot N + 2 \cdot T(N - 1) \quad \textcircled{1} - \textcircled{2}$$

$$N \cdot T(N) = 2 \cdot N + (N + 1) \cdot T(N - 1)$$

$$\frac{T(N)}{N + 1} = \frac{2}{N + 1} + \frac{T(N - 1)}{N}$$

Sukzessives Substituieren:

$$\begin{aligned} \frac{T(N)}{N + 1} &= \frac{2}{N + 1} + \frac{T(N - 1)}{N} \\ &= \frac{2}{N + 1} + \frac{2}{N} + \frac{T(N - 2)}{N - 1} \\ &\quad \vdots \\ &= \sum_{k=2}^N \frac{2}{k + 1} + \frac{T(1)}{2} \\ &= 2 \cdot \sum_{k=3}^{N+1} \frac{1}{k} + \frac{T(1)}{2} \end{aligned}$$

Mit der Ungleichung

$$\ln \frac{N + 1}{m} \leq \sum_{k=m}^N \frac{1}{k} \leq \ln \frac{N}{m - 1}$$

ergeben sich folgende Schranken (Beweis: siehe Abschnitt 2.3.1):

$$\frac{T(N)}{N + 1} \leq 2 \cdot \ln \frac{N + 1}{2} + \frac{T(1)}{2}$$

$$\text{und} \quad 2 \cdot \ln \frac{N + 2}{3} + \frac{T(1)}{2} \leq \frac{T(N)}{N + 1}$$

Ergebnis:

$$\begin{aligned} T(N) &= 2 \cdot (N + 1) \cdot \ln(N + 1) + \Theta(N) \\ &= 1.386 \cdot (N + 1) \cdot \text{ld}(N + 1) + \Theta(N) \end{aligned}$$

### Zusammenfassung

Analyse von Quicksort für die hier vorgestellte Implementierung:

- *Best Case*:  $T(N) = (N + 1) \cdot \lg(N + 1)$
- *Average Case*:  $T(N) = 1.386 \cdot (N + 1) \cdot \lg(N + 1)$
- *Worst Case*:  $T(N) = \frac{(N + 1) \cdot (N + 2)}{2} - 3$

### Varianten und Verbesserungen

Die Effizienz von QuickSort beruht darauf, daß in der innersten und somit am häufigsten ausgeführten Schleife nur ein Schlüsselvergleich durchgeführt wird. Im worst-case (d.h. für aufsteigend sortierte Folgen) wächst die Zahl der Vergleiche jedoch quadratisch in der Größe der Eingabefolge. Zudem ist QuickSort aufgrund des Rekursionsoverheads für kleine Folgen nicht geeignet. In der Literatur finden sich daher einige Verbesserungsvorschläge für die hier vorgestellte Implementierung:

- andere Wahl des Pivot-Elements:
  - $v := (a[l] + a[r])/2$
  - median-of-three: wähle das mittlere Element dreier zufällig ausgewählter Elemente
  - Vorteile: + keine expliziten Sentinel-Elemente erforderlich  
+ worst-case wird weniger wahrscheinlich  
+ insgesamt: 5% kürzere Laufzeit
- Beobachtung: QuickSort ist ineffizient bei kleinen Arrays, z.B.  $M = 12$  oder  $22$ .  
Abhilfe: 

```
IF M<(r-1) THEN QuickSort(l, r);
                ELSE InsertionSort(l, r);
                END;
```
- Der Speicherplatzbedarf wird aufgrund der Rekursion indirekt über die Größe der Aktivierungsblöcke bestimmt. Wenn jedoch stets das kleinere der beiden Teilarrays zuerst bearbeitet wird, dann ist die Größe der Aktivierungsblöcke nach oben durch  $2 \cdot \lg N$  beschränkt.
- Iterative Variante von Quicksort: Vermeidet die Rekursion (erfordert jedoch Stack!)

### 2.3.1 Beweis der Schranken von $\sum_{k=m}^n 1/k$ mittels Integral-Methode

Es sei

$$f : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \\ x \mapsto f(x)$$

eine monoton fallende Funktion, z.B.  $f(x) = \frac{1}{x}$ . Durch Bildung der Unter- und Obersumme erhält man die folgenden Schranken:

$$\int_k^{k+1} f(x) dx \leq 1 \cdot f(k) \leq \int_{k-1}^k f(x) dx$$

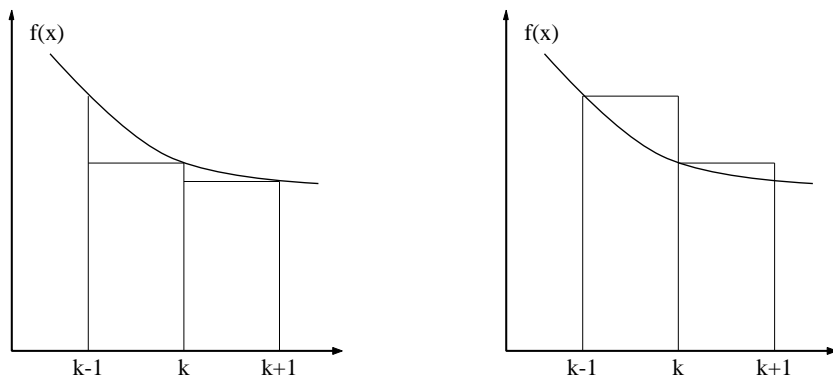


Abbildung 2.5: Unter- und Obersummen

Summation von  $k = m, \dots, n$ :

$$\int_m^{n+1} f(x) dx \leq \sum_{k=m}^n f(k) \leq \int_{m-1}^n f(x) dx$$

Insbesondere gilt dann für  $f(x) = \frac{1}{x}$  und  $m \geq 2$

$$\ln \frac{n+1}{m} \leq \sum_{k=m}^n \frac{1}{k} \leq \ln \frac{n}{m-1}$$

Für  $m = 1$  ergeben sich die sog. *Harmonischen Zahlen*.

**Definition: Harmonische Zahlen**

Als Harmonische Zahlen bezeichnet man die Reihe

$$H_n := \sum_{k=1}^n \frac{1}{k}, \quad n \in \mathbb{N}$$

Mit  $H_n = \sum_{k=1}^n \frac{1}{k} = 1 + \sum_{k=2}^n \frac{1}{k}$  erhält man die Ungleichung

$$\ln(n+1) \leq H_n \leq \ln n + 1$$

Es gilt (ohne Beweis):

$$\begin{aligned} \lim_{n \rightarrow \infty} (H_n - \ln n) &= \gamma \quad (\text{Eulersche Konstante}) \\ &\cong 0.5772 \end{aligned}$$

## 2.4 HeapSort

J.W.J. Williams 1964 und R.W. Floyd 1994

Erweiterung von SelectionSort mittels eines Heaps.

**Definition: Heap, Heap-Eigenschaft**

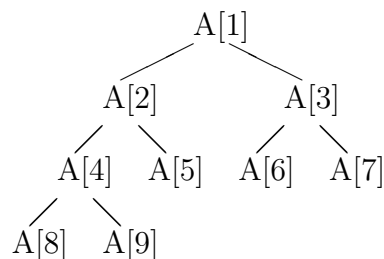
Ein *Heap* ist ein links-vollständiger Binärbaum, der in ein Array eingebettet ist (vgl. Abschnitt 1.3.5):

- Ein Array  $a[1..N]$  erfüllt die *Heap-Eigenschaft*, falls gilt:

$$a\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \geq a[i] \quad \text{für } i = 2, \dots, N$$

- Ein Array  $a[1..N]$  ist ein *Heap beginnend in Position*  $l = 1, \dots, N$ , falls:

$$a\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \geq a[i] \quad \text{für } i = 2l, \dots, N$$

**Beispiel:**

Die Heap-Eigenschaft bedeutet demnach, daß der Schlüssel jedes inneren Knotens größer ist als die Schlüssel seiner Söhne.

Insbesondere gilt damit für einen Heap im Array  $a[1..N]$ :

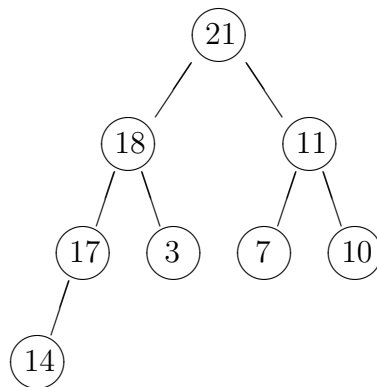
$$a[1] = \max \{ a[i] \mid i = 1, \dots, N \}$$

**Korrolar:** Jedes Array  $a[1..N]$  ist ein Heap beginnend in Position  $l = \lfloor \frac{N}{2} \rfloor + 1$

Um die Arbeitsweise von HeapSort zu illustrieren, betrachten wir das folgende Beispiel:

### Beispiel

Das folgende Array erfüllt die Heap-Eigenschaft:



Wir entfernen die Wurzel (21) und setzen das letzte Element des Arrays (14) auf die Wurzelposition. Um die Heap-Eigenschaft wieder herzustellen, wird das Element 14 solange mit dem größeren seiner Söhne vertauscht, bis alle Sohnknoten nur noch kleinere Schlüssel enthalten oder aber keine weiteren Sohnknoten existieren. Dieser Schritt heißt auch *Versickern*, *DownHeap* oder *ReHeap*.



Durch das Platzieren des Schlüssels 14 auf die Wurzelposition gewinnen wir einen freien Speicherplatz im Array, in den wir das Element 21 ablegen können.

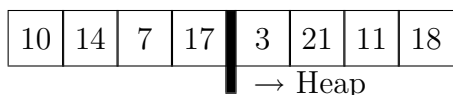
**Algorithmus:**

1. Wandle das Array  $a[1..N]$  in einen Heap um.
2. FOR  $i := 1$  TO  $N - 1$  DO
  - (a) Tausche  $a[1]$  (=Wurzel) und  $a[N - i + 1]$
  - (b) Stelle für das Rest-Array  $a[1..(N - i)]$  die Heap-Eigenschaft wieder her

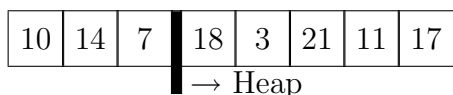
Der erste Schritt, d.h. der *Heap-Aufbau* bleibt zu klären. Wie erwähnt, ist die Heap-Eigenschaft für jedes Array ab der Position  $l = \lfloor N/2 \rfloor + 1$  bereits erfüllt. Indem man nun sukzessive die Elemente  $a[i]$  mit  $i = l - 1, \dots, 1$  versickern läßt, ergibt sich ein vollständiger Heap.

**Beispiel**

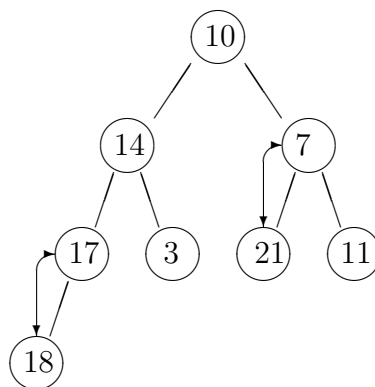
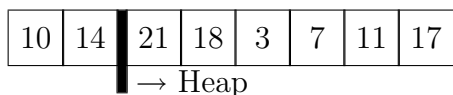
Ausgangssituation:



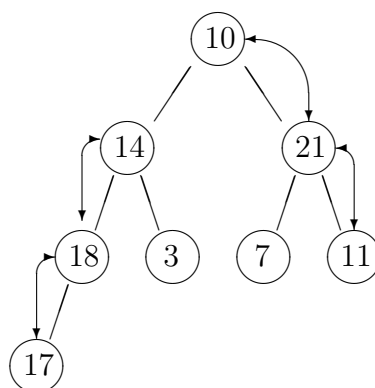
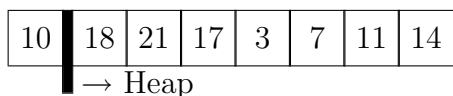
Vertausche  $17 \leftrightarrow 18$



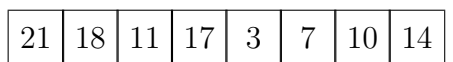
Vertausche  $7 \leftrightarrow 21$



Versickere  $14 \leftrightarrow 18 \leftrightarrow 17$



Versickere zuletzt  $10 \leftrightarrow 21 \leftrightarrow 11$



**Programm:**

Gegeben sei ein Array  $a[1..N]$ . Die Prozedur  $DownHeap(i, k, a)$  läßt das Element  $a[i]$  in dem Teilarray  $a[i..k]$  versickern.

```

PROCEDURE HeapSort(VAR a : ARRAY[1..N] OF KeyType) =
  VAR k : CARDINAL;
      t : KeyType;
BEGIN

  (* Heap-Aufbau *)
  FOR i := (N DIV 2) TO 1 BY -1 DO
    DownHeap(i, N, a);
  END;

  (* Sortierung *)
  k := N;
  REPEAT
    t := a[1];
    a[1] := a[k];
    a[k] := t;
    DEC(k);
    DownHeap(1, k, a);
  UNTIL k<=1;

END HeapSort;

PROCEDURE DownHeap(i, k : CARDINAL; VAR a : ARRAY OF KeyType) =
  VAR j : CARDINAL;
      v : KeyType;
BEGIN
  v := a[i];
  LOOP
    IF i<=(k DIV 2) THEN
      j := 2*i;          (* Berechne linken Sohn *)
      IF j<k THEN       (* Existiert rechter Sohn? *)
        IF a[j]<a[j+1] THEN (* Wähle größeren Sohn *)
          INC(j);
        END;
      END;
      IF a[j]<=v THEN EXIT; (* Beide Söhne kleiner? *)
      END;
      a[i] := a[j];
      i := j;
    ELSE EXIT;         (* Blatt erreicht! *)
    END;
  END;
  a[i] := v;
END DownHeap;

```

Erläuterungen zu der Prozedur  $\text{DownHeap}(i, k, a)$ :

- Start in Position  $i$
  - Falls nötig, wird  $a[i]$  mit dem größeren der beiden Söhne  $a[2i]$  bzw.  $a[2i + 1]$  vertauscht
- Abfragen:
- existieren beide Söhne?
  - Blatt erreicht?
- ggf. mit dem Sohn fortfahren

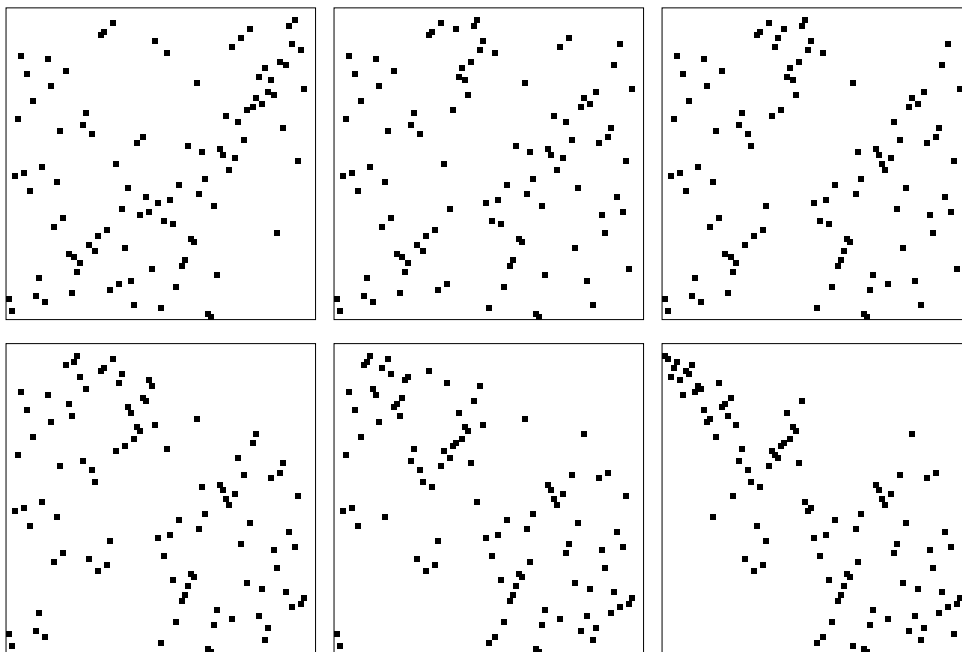


Abbildung 2.6: HeapSort einer zufälligen Permutation von Schlüsselementen: Aufbau des Heaps

### 2.4.1 Komplexitätsanalyse von HeapSort

[Mehlhorn]

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  $k = 5$ ).

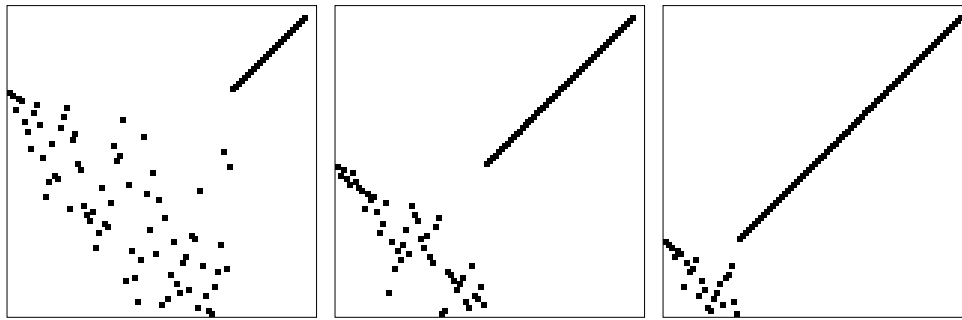
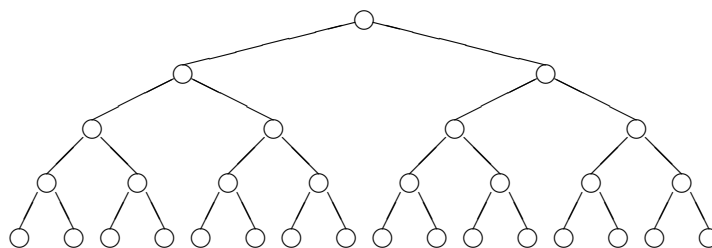


Abbildung 2.7: HeapSort einer zufälligen Permutation von Schlüsselementen: Sortierphase

**Beispiel:**



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

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

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

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

$$\sum_{i=0}^{k-2} 2 \cdot (k - 1 - i) \cdot 2^i = 2^{k+1} - 2(k + 1) \tag{2.1}$$

Beweis durch vollständige Induktion über  $k$ .

**Sortierphase**

Nach dem 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:

$$\sum_{i=0}^{k-1} 2i \cdot 2^i = 2(k-2) \cdot 2^k + 4 \quad (2.2)$$

Beweis durch vollständige Induktion über  $k$

**Zusammen:**

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

$$\begin{aligned} T(N) &\leq 2^{k+1} - 2(k+1) + 2(k-2) \cdot 2^k + 4 \\ &= 2k \cdot (2^k - 1) - 2(2^k - 1) \\ &= 2N \operatorname{ld}(N+1) - 2N \end{aligned}$$

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

$$2N \operatorname{ld}(N+1) - 2N$$

Vergleichen.

**Bemerkung:** In [Güting, S. 196] wird eine Bottom-Up-Variante von HeapSort beschrieben, die die Zahl der erforderlichen Vergleiche auf nahezu  $1 \cdot N \operatorname{ld}(N+1)$  senkt.

## 2.5 Untere und obere Schranken für das Sortierproblem

**bisher:** Komplexität eines Algorithmus

**jetzt:** Komplexität eines Problems (Aufgabenstellung)

**Ziel:** Sei  $T_{\mathcal{A}}(N) :=$  Zahl der Schlüsselvergleiche um eine  $N$ -elementige Folge von Schlüsselementen mit Algorithmus  $\mathcal{A}$  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

$$T_0(N) \leq T_{\mathcal{A}}(N) \quad \forall \mathcal{A}$$

gilt (d.h. jeder *denkbare* Algorithmus braucht in diesem Falle mindestens  $T_0(N)$  Vergleiche) ?

**Suche nach einer oberen Schranke:**

Wir wählen einen (möglichst effizienten) Sortieralgorithmus  $\mathcal{A}$  mit Komplexität  $T_{\mathcal{A}}(N)$ .

**Sprechweise:**  $T_{\mathcal{A}}(N)$  Vergleiche reichen, um jedes Sortierproblem zu lösen.

**Zusammen:**

$$T_0(N) \leq T_{\min}(N) \leq T_{\mathcal{A}}(N)$$

**Wunsch:**  $T_0(N)$  und  $T_{\mathcal{A}}(N)$  sollen möglichst eng zusammen liegen

**Konkret:**

Im folgenden betrachten wir für das Sortierproblem nur *Vergleichsoperationen*, d.h. auf der Suche nach einer unteren und oberen Schranke für das Sortierproblem werden wir uns nur auf solche Algorithmen beschränken, die ihr Wissen über die Anordnung der Eingabefolge allein durch (binäre) Vergleichsoperationen erhalten. Dabei werden wir sehen, daß BucketSort die von uns ermittelte untere Schranke durchbricht. Dies hängt damit zusammen, daß BucketSort zusätzliche Bedingungen an die Schlüsselmenge knüpft und somit kein *allgemeines Sortierverfahren* ist.

**Obere Schranke:**

Wir wählen als „effizienten“ Algorithmus MergeSort.

$$\begin{aligned} T_{\mathcal{A}}(N) &= N \lceil \lg N \rceil - 2^{\lceil \lg N \rceil} + 1 \\ &\leq N \lceil \lg N \rceil - N + 1 \end{aligned}$$

**Untere Schranke:**

Gegeben sei eine  $N$ -elementige Folge.

Sortieren:  $\hat{=}$  Auswahl einer Permutation dieser Folge

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

**Beispiel:**

Der binäre Entscheidungsbaum aus Abbildung 2.8 „sortiert“ ein 3-elementiges Array  $a[1..3]$ . Da  $3! = 6$ , muß der Entscheidungsbaum 6 Blätter besitzen. Wegen  $\lg 6 \cong 2.58$  existiert in dem Baum mindestens ein Pfad der Länge 3.

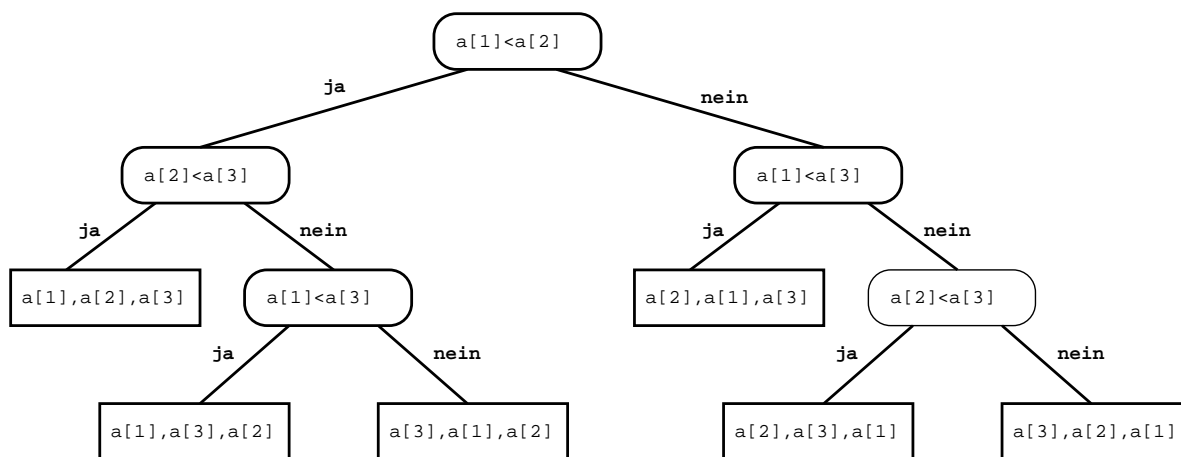


Abbildung 2.8: Binärer Entscheidungsbaum zur Sortierung eines 3-elementigen Arrays

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

$$\lg N! \leq \lceil \lg N! \rceil \leq T_0(N)$$

Mit der Ungleichung (Beweis: siehe Abschnitt 2.6)

$$N \lg N - N \lg e \leq \lg N!$$

erhalten wir das Ergebnis:

$$N \lg N - N \lg e \leq T_{\min}(N) \leq N \lceil \lg N \rceil - N + 1$$

## 2.6 Schranken für $n!$

Die Fakultät wird oft benötigt. Eine direkte Berechnung ist aber gleichzeitig umständlich und analytisch schwierig zu behandeln. Aus diesem Grund werden enge Schranken für  $n!$  benötigt.

### 1. Einfache Schranken:

- Obere Schranke:

$$\begin{aligned} n! &= \prod_{i=1}^n i \\ &\leq \prod_{i=1}^n n \\ &= n^n \end{aligned}$$

- Untere Schranke:

$$\begin{aligned} n! &= \prod_{i=1}^n i \\ &\geq \prod_{i=\lceil n/2 \rceil}^n i \\ &\geq \prod_{i=\lceil n/2 \rceil}^n \lceil n/2 \rceil \\ &\geq (n/2)^{n/2} \end{aligned}$$

- Zusammen:

$$(n/2)^{n/2} \leq n! \leq n^n$$

### 2. Engere Schranken:

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

Für die Fakultätsfunktion gilt:

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

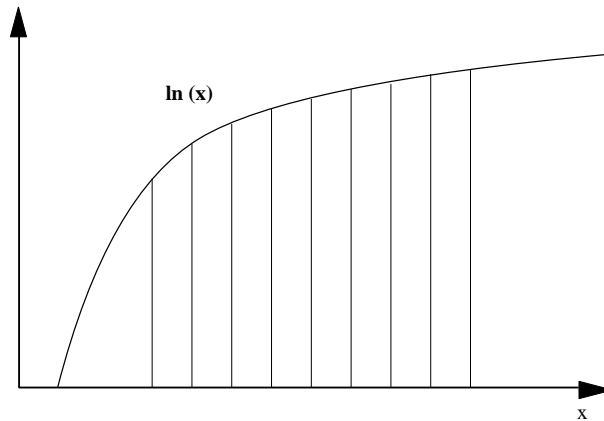


Abbildung 2.9: Graph zu  $\ln x$

**Untere Schranke:**

Die Logarithmusfunktion  $x \mapsto f(x) := \ln x$  ist monoton und konvex. Das Integral über  $f(x)$  in den Grenzen  $i - 1/2$  und  $i + 1/2$  bildet daher eine untere Schranke für die Trapez-Obersumme, deren Flächenmaßzahl durch  $\ln i$  gegeben ist (vgl. Abbildung 2.10)

$$\int_{i-1/2}^{i+1/2} \ln x \, dx \leq \ln i$$

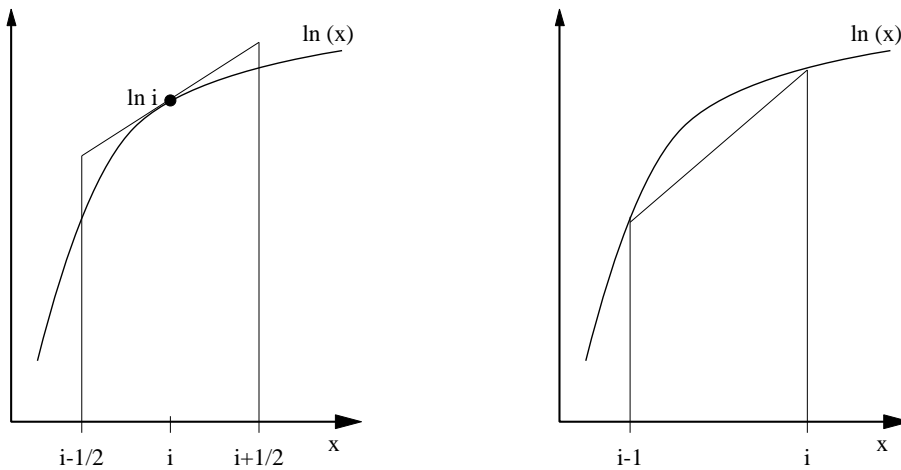


Abbildung 2.10: Trapezober- und Trapezuntersumme

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

$$\int_{1/2}^{n+1/2} \ln x \, dx \leq \sum_{i=1}^n \ln i = \ln n!$$

Mit

$$\begin{aligned} \int_a^b \ln x \, dx &= x \cdot \ln x - x \Big|_a^b \\ &= b \cdot \ln b - a \cdot \ln a \end{aligned}$$

folgt die untere Schranke für  $\ln n!$

$$n \cdot \ln \frac{n}{e} + \frac{1}{2} \cdot \ln 2 \leq \ln n!$$

### Obere Schranke:

Das Integral über  $\ln x$  in den Grenzen  $i - 1$  und  $i$  ist eine obere Schranke der zugehörigen Trapezuntersumme (vgl. Abbildung 2.10):

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

Summation von  $i = 2, \dots, n$  ergibt:

$$\begin{aligned} \frac{1}{2} \cdot \ln n! - \ln n &\leq \int_1^n \ln x \, dx = n \cdot \ln \frac{n}{e} + 1 \\ \ln n! &\leq n \cdot \ln \frac{n}{e} + \frac{1}{2} \cdot \ln n + 1 \end{aligned}$$

### Zusammen:

$$n \cdot \ln \frac{n}{e} + \frac{1}{2} \cdot \ln n + \frac{1}{2} \cdot \ln 2 \leq \ln n! \leq n \cdot \ln \frac{n}{e} + \frac{1}{2} \cdot \ln n + 1$$

Durch die Näherungen ergibt sich die folgende Schranke:

$$n \leq 2 : \quad \sqrt{2} \leq \frac{n!}{(n/e)^n \cdot \sqrt{n}} \leq e$$

**3. Engste Schranken und Stirlingsche Formel** (ohne Beweis)

Es gilt:

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

mit

$$\frac{1}{12n+1} < g(n) < \frac{1}{12n}$$

zum Beweis: siehe U. Krengel, *Einführung in die Wahrscheinlichkeitstheorie und Statistik*, 3. Auflage, Vieweg Studium, S.78 ff.

Der Fehler  $g(n)$  strebt gegen 0:

$$\lim_{n \rightarrow \infty} g(n) = 0$$

**2.7 MergeSort**

MergeSort wurde bereits in Abschnitt 1.4.1 behandelt. Die wichtigsten Ergebnisse waren:

- worst case = average case:  $N \lg N$
- zusätzlicher Speicherplatz:  $O(N)$ , nicht in situ, aber sequentiell

**2.8 Zusammenfassung**

Die folgende Tabelle faßt die Rechenzeiten der in diesem Kapitel behandelten Sortierverfahren zusammen. Sämtliche Algorithmen wurden dabei in RAM-Code (vgl. Kapitel 1.2.1) umgesetzt. Als Kostenmaß wurde das Einheitskostenmaß verwendet.

Verfahren	Komplexität	Abschnitt	Referenz
SelectionSort	$2.5N^2 + 3(N+1) \lg N + 4.5N - 4$	(2.2.1)	[Mehlhorn, S. 40]
QuickSort	$9(N+1) \lg(N+1) + 29N - 33$	(2.3)	[Mehlhorn, S. 51]
HeapSort	$20N \lg N - N - 7$	(2.4)	[Mehlhorn, S. 47]
MergeSort	$12N \lg N + 40N + 97 \lg N + 29$	(1.4.1)	[Mehlhorn, S. 58]