Übersicht

Übersicht

Einführung

- Suchen und Sortieren sind elementare Aufgaben, die
  - sich (nicht nur) in der Informatik *sehr häufig* stellen
  - sich auf viele unterschiedliche Arten lösen lassen
- Suche
  - *sequentiell* in unsortierten Folgen
  - *binäre Suche* in sortierten Folgen
- Sortieren
  - Verschiedene elementare Algorithmen
  - Teils basierend auf *Rekursion*
- Wir entwerfen Algorithmen
- Wir versuchen, Aufwand abzuschätzen
  - Komplexität von Algorithmen

Voraussetzungen

- Wir betrachten lineare Folgen $(a_i), 0 \leqslant i < n$, für die gilt
  - a_i bezeichnet den i. Datenwert
  - Es gibt eine Ordnungsrelation $<$ auf dem Datentyp
  - Zugriff auf *jedes* Element a_i der Folge möglich
- Beispiel: Lexikon oder Telefonbuch
  - *Lexikographische* Ordnung (z.B. „Aal“ $\leqslant$ „Aberglaube“)
  - Zugriff nur auf einzelne Seiten
- Wir beschränken uns vorerst auf Folgen von ganzen Zahlen:
  - d.h. in Java: Felder `a` vom Typ `int[]` mit `a.length==n`
  - und anders als für Objekte können wir mit `<,==,>` vergleichen
- Wir nehmen hier an, dass gleiche Einträge *nicht mehrfach* vorkommen.

Suche

Definiere Funktion

$$\texttt{find}(a, x) = \begin{cases} i & \text{falls } a_i = x \\ \bot & \text{falls } \nexists i : a_i = x \end{cases}$$

- Index `i` des gesuchten Eintrags oder
- Undefiniert falls kein entsprechendes Element existiert

- Falls Einträge mehrfach vorkommen können, würden wir eine Menge von Indices erwarten. *Wir schließen das hier aus.*

Übersicht

Sequentielle Suche

- Gegeben ist eine Folge von *unsortierten Daten*, d.h.
- Keine Annahme über Verteilung und Auftreten von Werten
  - z.B. Rezeptsammlung aus einzelnen, unsortierten Blättern
- Es müssen *alle* Werte der Folge durchsucht werden!
  - Muss entscheiden, ob x in (a_i) vorkommt
  - Einfachste Möglichkeit: *sequentiell* suchen von $i = 0, \ldots, n-1$
- Demnach könnten wir `find`(a, x) wie folgt implementieren

```
static final int UNDEF = -1;   // ⊥

public static int find(int[] a,int x) {
  for (int i=0;i<a.length;++i)
    if (a[i]==x) return i;
  return UNDEF;
}
```

  - $\perp$ wird hier durch ungültigen Index `UNDEF==-1` ausgedrückt

Aufwand für sequentielle Suche

- Definiere den Aufwand als *Anzahl der Vergleiche* `a[i]==x`
- Wir betrachten verschiedene Szenarien

Szenario	Aufwand
bester Fall	1
schlechtester Fall	n
Durchschnitt (erfolglose Suche)	n
Durchschnitt (erfolgreiche Suche)	$\lfloor \frac{n+1}{2} \rfloor$

  - Im besten Fall gilt „*zufällig*“ `a[0]==x`
  - Schlechtester Fall = erfolglose Suche *oder* `a[n-1]==x`
- Durchschnittlicher Aufwand
  - Annahme: Gleichverteilung
  - Aufwand $1, 2, \ldots, n-1, n$ jeweils gleich wahrscheinlich
  - Im Mittel $\frac{1}{n} \cdot (1+2+\cdots+n-1+n) = \frac{1}{n} \cdot \frac{(n+1)n}{2} = \frac{n+1}{2}$

Suche in geordneter Folge

- Was ändert sich für eine geordnete Folge?
  - Hier gilt für $0 \leqslant i, j < n$: $\quad i \leqslant j \Rightarrow a_i \leqslant a_j$
- Beispiel: Suche nach Eintrag in Telefonbuch
- Idee: Wende den *Teile und herrsche* Grundsatz an
  - Teile in zwei Teile
  - Rekursion über den Teil, in dem gesuchter Eintrag liegt

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	4	9	11	12	17	18	19	**20**	23	24	25	26
1	2	4	9	11	12	17	18	19	**20**	23	24	25	26
1	2	4	9	11	12	17	18	19	**20**	23	24	25	26
1	2	4	9	11	12	17	18	19	**20**	23	24	25	26

Binäre Suche in Java (rekursiv)

```
public static int find(int[] a,int x) {
  return _find(a,0,a.length-1,x);
}
static int _find(int[] a,int l,int r,int x) {
  if (l>r)              return UNDEF;
  int m=(l+r)/2;
  if       (x==a[m]) return m;
  else if (x<a[m])   return _find(a,l,m-1,x);
  else               return _find(a,m+1,r,x);
}
```

- `int l,int r` bezeichnen *linke* und *rechte* Grenze der Partition
- Initial `l=0` und `r=a.length-1=n-1`
- `m=(l+r)/2` bezeichnet die *Mitte*
- Rekursion endet
  - wenn `l>r` — dann war kein Abstieg möglich, oder
  - wenn `a[m]==x`

Binäre Suche in Java (iterativ)

```
public static int find(int[] a,int x) {
  int l=0, r=a.length-1, m;
  do {
    m=(l+r)/2;
    if (x<a[m])
      r=m-1;
    else
      l=m+1;
  } while (x!=a[m] && l<=r);

  return (x==a[m]) ? m : UNDEF;
}
```

- Partition in `while` Schleife mit Anpassung der Grenzen `l` und `r`
- Abbruch der Schleife
  - wenn `a[m]==x`, *oder*
  - wenn `l>r` — dann keine weitere Partition möglich

Analyse des Aufwands für binäre Suche

- Eingabe: durchsuche Feld mit n Elementen
- *Annahme: erfolglose Suche (schlechtester Fall)*
- Nach dem 1. Schritt: durchsuche noch $\frac{n}{2}$ Elemente

  Nach dem 2. Schritt: durchsuche noch $\frac{n}{4}$ Elemente

  ...

  Nach dem k. Schritt: durchsuche noch $\frac{n}{2^k}$ Elemente
- *In jedem Schritt halbiert sich die Anzahl der Elemente*
- Solange bis nur noch ein Element in Teilliste: `left==right`
  Das heißt

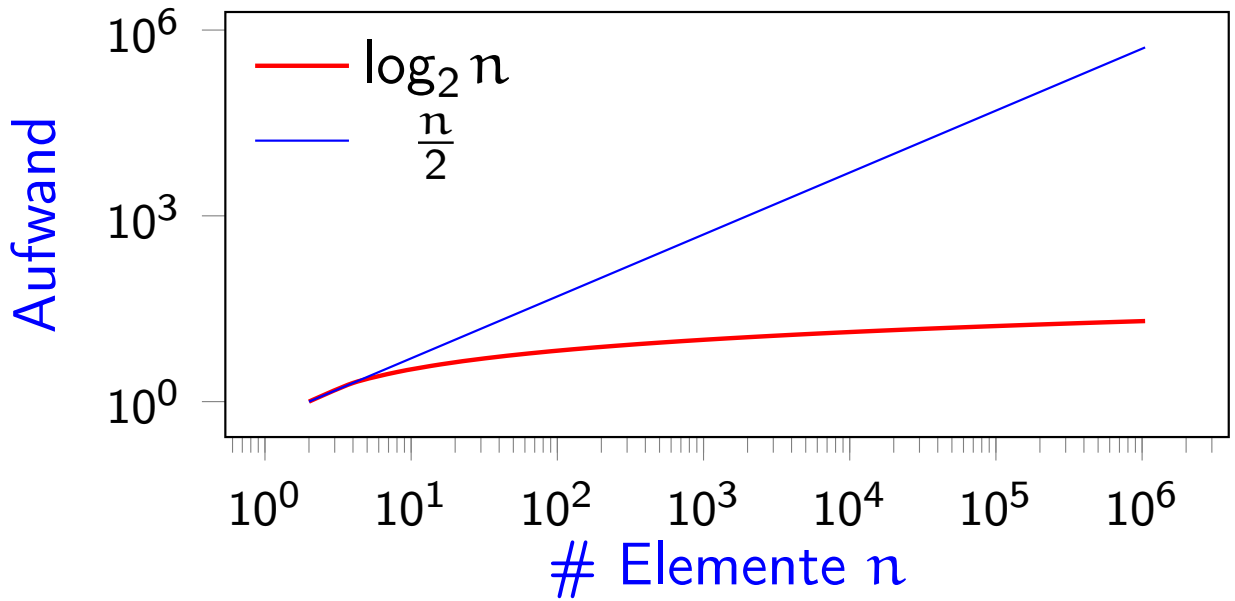
$$1 = \frac{n}{2^k} \Leftrightarrow 2^k = n \Leftrightarrow k = \log_2 n$$

  Es werden maximal $\log_2 n$ Schritte benötigt!
  - *Schritt* = Rekursion oder Iteration
  - Das sind maximal $\log_2 n + 1$ *Vergleiche*

Aufwand für binäre Suche

Szenario	Aufwand
bester Fall	1
schlechtester Fall	$\log_2 n + 1$
Durchschnitt (erfolglose Suche)	$\log_2 n + 1$
Durchschnitt (erfolgreiche Suche)	$\approx \log_2 n + 1$

Zum Vergleich mit sequentieller Suche



Sequentielle vs binäre Suche

- Offenbar ist der Aufwand der binären Suche geringer
  - Macht sich vor allem für große n bemerkbar
  - Für $n = 1.000.000$ im Mittel 20 Vergleiche gegenüber 500.000
  - Doppelte Anzahl $n \quad \Rightarrow \quad +1$ Vergleich!
- Schlüssel zum Erfolg: *Teile und herrsche* Grundsatz
  - Rekursion
  - In diesem Fall einfach durch Schleife ersetzbar
- *Anmerkung*: Lohnt sich binäre Suche für *kleine* n? z.B. $n = 10$
  - Hier kann sequentielle Suche effizienter sein.
  - Abhängig von Datentyp, Sprache/Compiler *und* Rechnerarchitektur
  - *Ausprobieren!*
- Für binäre Suche benötigen wir eine *sortierte* Folge ...
  Als nächstes: Sortieralgorithmen

Übersicht

Sortieralgorithmen

- *Wir betrachten eine Auswahl von Algorithmen*
- Sortieren durch Auswahl: *Selection Sort*
- Sortieren durch Einfügen: *Insertion Sort*
- Sortieren durch Aufsteigen: *Bubblesort*
- *Quicksort**
- Sortieren durch „Mischen“: *Mergesort**

* *Teile und herrsche*

- *Zuerst zu den Spielregeln . . .*

Sortieren

- Aufgabe: Sortiere ein Feld (a_i) so, dass gilt $i \leqslant j \Rightarrow a_i \leqslant a_j$
- Wir betrachten wieder Felder von ganzen Zahlen (Typ `int[]`)
  - Tatsächlich beliebiger Typ
  - Benötige Ordnungsrelation $<$ ggf. auf *Schlüssel*
  - Oft Typ = Paar (s_i, w_i) aus Schlüssel s_i, Wert w_i (*key*/*value*)
- Elementare Operationen
  - Vergleich $a_i < a_j$
  - Vertauschen (*swap*) von zwei Einträgen: ändert Reihenfolge
- Beobachtung
  - Ggf. unterschiedliche Kosten (Vergleich/Vertauschen)
  - Nötige Folge von Vertauschungen nicht eindeutig (*Viele Wege führen zum Ziel.*)

Ordnungsrelation

- Es reicht $<$ z.B. in Form einer Funktion `bool less(T a,T b)`
- Denn

$$
\begin{aligned}
a > b &\Leftrightarrow b < a \\
a = b &\Leftrightarrow \neg\big((a < b) \vee (a > b)\big) \Leftrightarrow \neg(a < b) \wedge \neg(a > b) \\
a \leqslant b &\Leftrightarrow (a < b) \vee (a = b) \Leftrightarrow (a < b) \vee \neg(a > b)
\end{aligned}
$$

- Bemerkung
  - C++ Standardbibliothek definiert auf diese Weise Operatoren `== < <= > >=` für beliebige Datentypen „automatisch“.
  - Das ist in Java so nicht möglich.
  - *Interface* `int Comparable::compareTo(Object other)`

Rückgabewert	<0	$\Leftrightarrow$	`this < other`
Rückgabewert	$=0$	$\Leftrightarrow$	`this == other`
Rückgabewert	>0	$\Leftrightarrow$	`this > other`

key-value Paare

- Oft werden Paare von Schlüsseln und Werten sortiert
  - Schlüssel (*key*) definiert die Sortierreihenfolge
  - Wert (*value*) = Daten (irrelevant für Sortierung)
- Anwendungsbeispiele
  - (Matrikelnummer, (Name,Vorname,Adresse,. . .))
  - ((Name,Vorname), Matrikelnummer)
- z.B. In Java: nur ein „Teil“ eines Objekts dient als Schlüssel
- *Wir bleiben vorerst bei* `int` (als Schlüssel *und* Wert)
- Sonderfall: gleicher Schlüssel, verschiedener Wert
  - Daten sind unterscheidbar trotz gleicher Schlüssel
  - z.B. $(1, \mathtt{a}), \neq (1, \mathtt{b})$ (Zahl als Schlüssel)
  - Reihenfolge als Eigenschaft des Sortierverfahrens . . .

Stabiles Sortierverfahren

Definition (Stabiles Sortierverfahren)

Ein Sortierverfahren heißt *stabil*, wenn es die relative Reihenfolge für gleiche Schlüssel beibehält.

Beispiel

- Gegeben ist die Sequenz $\big[(2,\mathtt{x}),(1,\mathtt{b}),(1,\mathtt{a})\big]$ und die
- Ordnungsrelation $(s,v) < (s',v') \quad \Leftrightarrow \quad s < s'$
- D.h. wir haben *Schlüssel* $\{1,2\}$ und *Werte* $\{\mathtt{a},\mathtt{b},\mathtt{x}\}$
- *Beide* Folgen wurden sortiert ...

  $\big[(1,\mathtt{a}),(1,\mathtt{b}),(2,\mathtt{x})\big]$ *nicht* stabil

  $\big[(1,\mathtt{b}),(1,\mathtt{a}),(2,\mathtt{x})\big]$ stabil

Internes und externes Sortieren

Definition (Internes und externes Sortieren)

Ein Sortierverfahren heißt *intern*, wenn alle zu sortierenden Daten (und nötige Hilfsdaten) in den Arbeitsspeicher passen.
Ein Sortierverfahren heißt *extern*, wenn Teile der Daten auf einen anderen, externen Speicher ausgelagert sind.

- Arbeitsspeicher = Speicher in dem Vergleich/Vertauschen stattfinden, z.B. RAM
- Externer Speicher = i.d.R. Massenspeicher wie z.B. Festplatten, Magnetbänder
- Betrachte sowohl kleine als auch sehr große Datenmengen
- Daneben noch Unterscheidung
  - Überschreibe Eingabe durch Ausgabe (*in-place* oder *in situ*)
  - Ausgabe (und/oder Hilfsdaten) in neuem Feld (*out-of-place*)

Trivialer Fall: 2 Elemente

```
public static void sort2(int[] a) {
  assert (a.length==2); // use with "java -ea"

  if (a[0]>a[1]) {
    int t=a[0]; a[0]=a[1]; a[1]=t;      // swap
  }

  assert (a[0]<=a[1]);
}
```

- Vertauschen (*swap*) falls Bedingung $i \leqslant j \Rightarrow a_i \leqslant a_j$ verletzt
- *Bemerkung*: `assert condition;`
  - *to assert* = „versichern, dass [Bedingung gilt]“
  - Abbruch mit Fehlermeldung, wenn Bedingung verletzt
  - Überprüfung nur mit `java -ea [Class]` (*enable assertions*)
  - Oft besser als ein Kommentar (allein)!
  - *Später mehr* zu Vor- und Nachbedingungen, Invarianten

Sortiere Feld mit 3 Elementen

```
public static void sort3(int[] a) {
  assert (a.length==3);

  if (a[0]>a[1]) {
    int t=a[0]; a[0]=a[1]; a[1]=t;
  }
  if (a[0]>a[2]) {
    int t=a[0]; a[0]=a[2]; a[2]=t;
  }
  if (a[1]>a[2]) {
    int t=a[1]; a[1]=a[2]; a[2]=t;
  }

  assert (a[0]<=a[1] && a[1]<=a[2]);
}
```

- Stabil? Möglichkeiten $(1,1,1), (1,2,2), (2,1,2), (2,2,1)$

Sortiere Feld mit 3 Elementen (stabil)

```
public static void sort3(int[] a) {
  assert (a.length==3);

  if (a[1]>a[2]) {
    int t=a[0]; a[0]=a[1]; a[1]=t;
  }
  if (a[0]>a[1]) {
    int t=a[0]; a[0]=a[2]; a[2]=t;
  }
  if (a[1]>a[2]) {
    int t=a[1]; a[1]=a[2]; a[2]=t;
  }

  assert (a[0]<=a[1] && a[1]<=a[2]);
}
```

- Stabil? ✓ Möglichkeiten $(1,1,1), (1,2,2), (2,1,2), (2,2,1)$

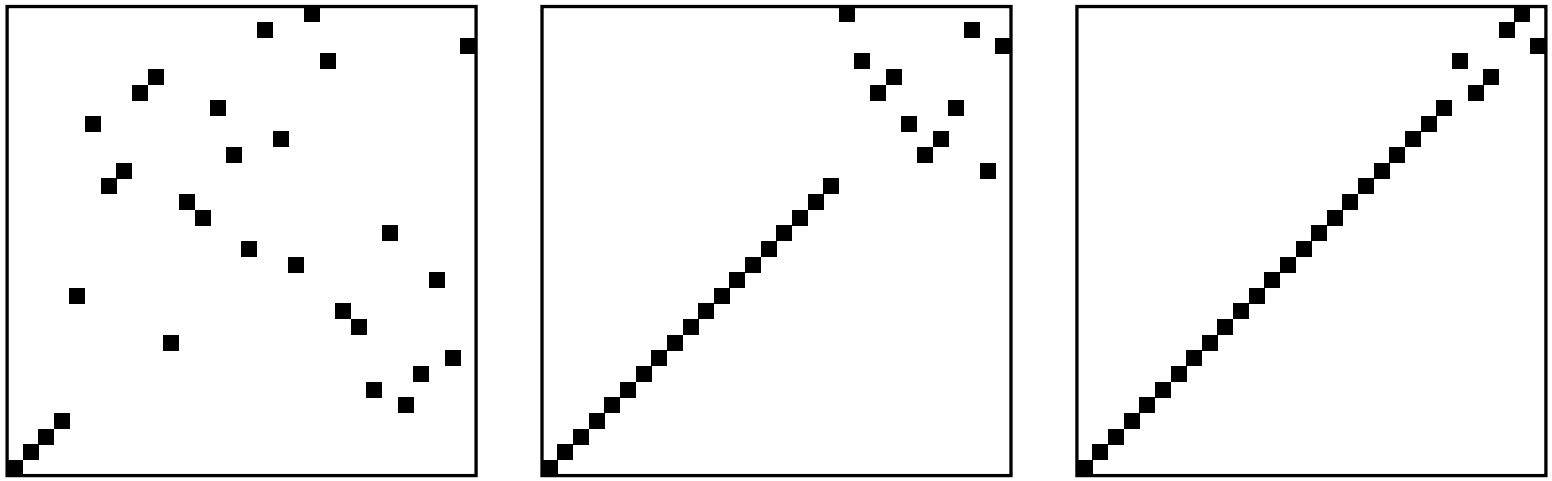
Selection Sort: Sortieren durch Auswahl

- Idee
  1. Finde kleinsten Eintrag
  2. Stelle ihn an den Anfang
  3. Wiederhole das gleiche für restliche Einträge
- Umsetzung in Java

```
public static void selectionsort(int[] a) {
  int n=a.length;
  for (int i=0;i<n-1;++i) {                    // (3)
    int imin=i;
    for (int j=i+1;j<n;++j)                    // (1)
      if (a[j]<a[imin])
        imin=j;
    int t=a[imin]; a[imin]=a[i]; a[i]=t; // (2)
  }
}
```

Selection Sort

- $n = 30$



Eigenschaften von Selection Sort

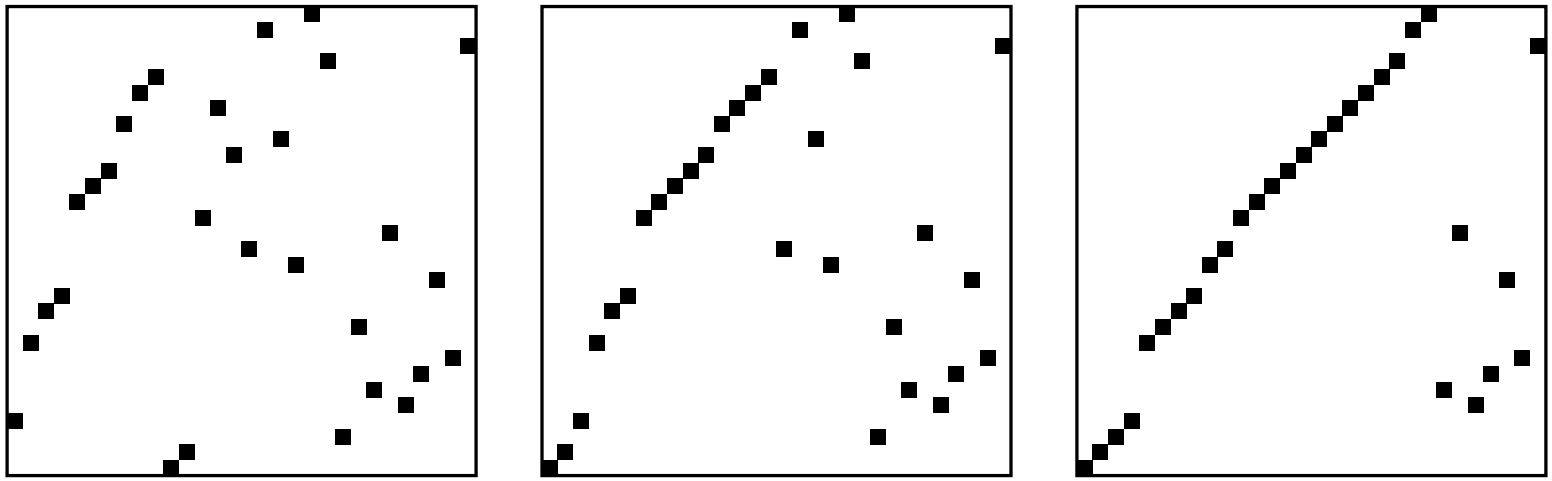
- Es werden genau $n-1$ Werte *vertauscht*.
  - Denn äußere Schleife wird $n-1$ mal durchlaufen.
- Es werden $\frac{n(n-1)}{2} \approx \frac{n^2}{2}$ *Vergleiche* benötigt.
  - $n-1$ Durchgänge mit $n-i$ Vergleichen im i. Durchgang
  - Also $(n-1)+(n-2)+\cdots+2+1 = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$
- Identisch für besten/mittleren/schlechtesten Fall
- *Selection Sort* ist *nicht* stabil !
  - Problem: Vertauschen `a[imin]` $\leftrightarrow$ `a[i]`
  - Kann Reihenfolge gleicher S. ändern z.B. $[\underline{2}, 2, 1] \rightarrow [1, 2, \underline{2}]$
  - *out-of-place* Variante von *Selection Sort* ist stabil

Insertion Sort: Sortieren durch Einfügen

- Idee
  1. Nimm nächsten Eintrag
  2. Füge ihn in *bereits sortierte* Teilfolge ein
  3. Wiederhole, bis alle Einträge einsortiert sind
- Beispiel: Sortieren eines Kartenspiels
  - Zwei Stapel: noch nicht sortiert / schon sortiert
  - Sortiere jeweils die nächste Karte ein
  - In dieser Formulierung *out-of-place*, wir betrachten *in-place*
- Elementare Operation: Einfügen in sortierte Teilfolge
  - Schiebe größere Einträge um eins nach rechts
  - Einfügen in freien Platz
  - $(a_1, \ldots, a_j, a_{j+1}, \ldots, a_n, x) \to (a_1, \ldots, a_j, x, a_{j+1}, \ldots, a_{n-1})$ mit $a_j < x < a_{j+1}$ und $i \leqslant j \Rightarrow a_i \leqslant a_j$

Insertion Sort

- $n = 30$



Umsetzung in Java

```
public static void insertionsort(int[] a) {
  int n=a.length;
  for (int i=1;i<n;++i) {

    int x=a[i];          // insert x into (a_0,...,a_{i-1})
    int j;
    for (j=i;j>0 && a[j-1]>x;--j)
      a[j]=a[j-1];

    a[j]=x;
  }
}
```

- `a[j-1]` wird für $j = 0$ nicht ausgewertet!
- Grund: *short circuit evaluation* von Termen in Bedingungen

Analyse von Insertion Sort (1)

- Immer $n-1$ Iterationen in äußerer Schleife („alle einfügen“)
- Betrachte innere Schleife („Einfügeposition finden“)

```
for (j=i;j>0 && a[j-1]>x;--j) a[j]=a[j-1];
```

- Bester Fall
  - Folge ist bereits sortiert.
  - Ein Vergleich pro Element $i \Rightarrow$ insgesamt $n-1 \approx n$ Vergleiche
  - Kein Einfügen/Verschieben nötig (aber `x=a[i];...a[j]=x;`)

Insertion Sort stellt fest, dass Folge sortiert ist und bricht ab!

Analyse von Insertion Sort (2)

- Immer $n-1$ Iterationen in äußerer Schleife („alle einfügen“)
- Betrachte innere Schleife („Einfügeposition finden“)

```
for (j=i;j>0 && a[j-1]>x;--j) a[j]=a[j-1];
```

- Schlechtester Fall
  - Liste ist umgekehrt sortiert: $i \leqslant j \Rightarrow a_i \geqslant a_j$
  - D.h. Einfügeposition immer bei $j = 0$, also je $i-1$ Iterationen
  - Insgesamt $\sum_{i=1}^{n}(i-1) = \frac{n\,(n-1)}{2} \approx \frac{n^2}{2}$ Vergleiche
  - Gleiche Anzahl an Schiebe-Operationen `a[j]=a[j-1]`
  - Zähle 1 Schiebe-Operation $\approx \frac{1}{2} \times$ Vertauschen
- Im Mittel
  - Unsortiert, Annahme: Gleichverteilung $\Rightarrow$ Einfügen bei $\frac{i-1}{2}$
  - Damit etwa $\sum_{i=1}^{n} \frac{(i-1)}{2} = \frac{n\,(n-1)}{4} \approx \frac{n^2}{4}$ Vergleiche

Eigenschaften von Insertion Sort

- Im *Mittel* etwa $\frac{n^2}{4}$ Vergleiche und $\frac{n^2}{8}$ Vertauschungen.
- Im *schlechtesten Fall* etwa doppelter Aufwand,
  Folge ist umgekehrt sortiert.
- Im *besten Fall* nur n Vergleiche für sortierte Folgen!
  Erwarte linearen Aufwand für „fast“ sortierte Folgen.
- Insertion Sort ist *stabil*!
- Variante: Finde Einfügeposition mit *binärer Suche*
  - Lohnend, wenn Aufwand für Vergleiche relativ hoch
  - z.B. für lange Zeichenketten (verschiebe nur Referenzen)
  - Es bleibt bei $\frac{n^2}{8}$ Vertauschungen.
- Variante: Shell Sort (nach Donald Shell)
  - Erlaubt Austausch über “größere” Nachbarschaften
  - *Nicht* stabil

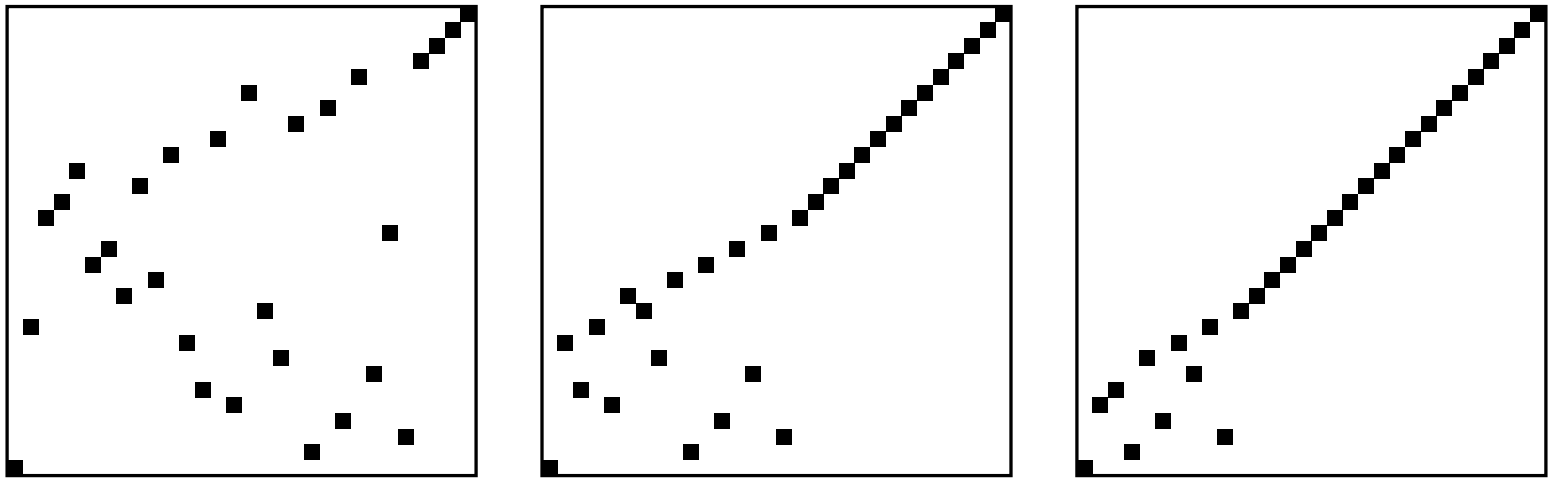
Bubble Sort

- Idee
  1. Iteriere über alle Einträge
  2. Vertausche je zwei benachbarte Einträge, wenn nötig
  3. Wiederhole Verfahren, bis nichts mehr vertauscht wurde
- *Große Einträge „perlen“ nach oben, kleine „sinken“ nach unten.*
- Beobachtung
  - Erste Iteration schiebt größte Zahl ans Ende der Folge
  - Verbesserung: betrachte in jeder Iteration nur Teilfolge

```
public static void bubblesort(int[] a) {
  int n=a.length;
  for (int i=n-1;i>=0;--i)
    for (int j=1;j<=i;++j)
      if (a[j-1]>a[j]) {
        int t=a[j]; a[j]=a[j-1]; a[j-1]=t;
      }
} // Missing: Terminate if noting was swapped
```

Bubble Sort

- $n = 30$



Eigenschaften von Bubble Sort

- Im *schlechtesten Fall* etwa $\frac{n^2}{2}$ Vergleiche und $\frac{n^2}{2}$ Vertauschungen.
- Gleiches gilt *im Mittel*.
- Im *besten Fall* nur n Vergleiche für sortierte Folgen
- Einfache Idee. Aber *eher nicht praktikabel.*
  - Insertion Sort ist i.d.R. effizienter.

> *In short, the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems.*

D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*

Quicksort

- Anwendung des *Teile und Herrsche* Grundsatzes
- Idee
  1. Wähle einen Eintrag $p \in \{a_i\}$, das sogenannte *Pivotelement*
  2. Zerlege Folge in Teile $\{a_i \,|\, a_i < p\}$, p, $\{a_i \,|\, a_i > p\}$
  3. *Rekursive* Anwendung auf nicht-leere Teile
- *pivot* (französisch/englisch) — *Dreh- und Angelpunkt*
- Quicksort Algorithmus in Java

```
public static void quicksort(int[] a) {
  _quicksort(a,0,a.length-1);      // [a_0,...,a_n-1]
}
static void _quicksort(int[] a,int l,int r) {
  if (r>l) {                       // [a_l,...,a_r]
    int m=partition(a,l,r);        // pivot p=a_m
    _quicksort(a,l,m-1);
    _quicksort(a,m+1,r);
  }
}
```

Zentraler Schritt: Zerlegung

- *In-place* Zerlegung
  1. Wähle Pivot p (hier immer der letzte Eintrag $p = a_{r-1}$)
  2. Durchsuche Folge von links ($i = 0, 1, \ldots$) nach $a_i > p$
  3. Durchsuche Folge von rechts ($j = n-1, \ldots$) nach $a_j < p$
  4. Vertausche ggf. Einträge $a_i \leftrightarrow a_j$ und wiederhole bis $i > j$

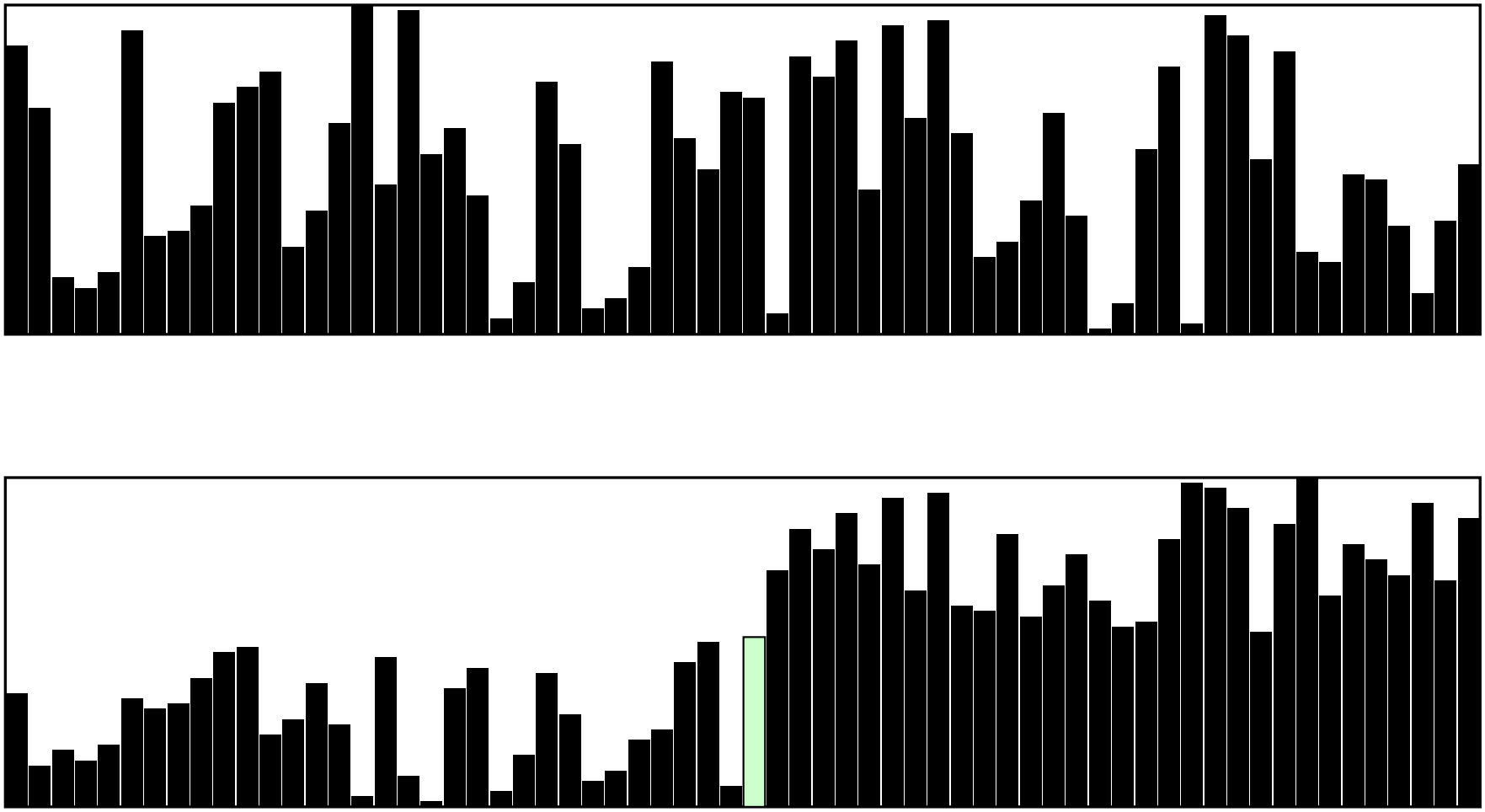
5	3	6	7	1	2	4
0	1	2	3	4	5	6
2	3	1	4	6	5	7

Zerlegung in Java

```
static int partition(int[] a,int l,int r) {
  assert (l<=r);                    // l-eft, r-ight

  int p=a[r], t;                              // pivot
  int i=l-1, j=r;
  do {
    do ++i; while (a[i]<p);                   // find
    do --j; while (j>l && a[j]>p);

    t=a[i]; a[i]=a[j]; a[j]=t;                // swap
  } while (i<j);

  a[j]=a[i]; a[i]=a[r]; a[r]=t;

  return i;                       // new index of pivot
}
```

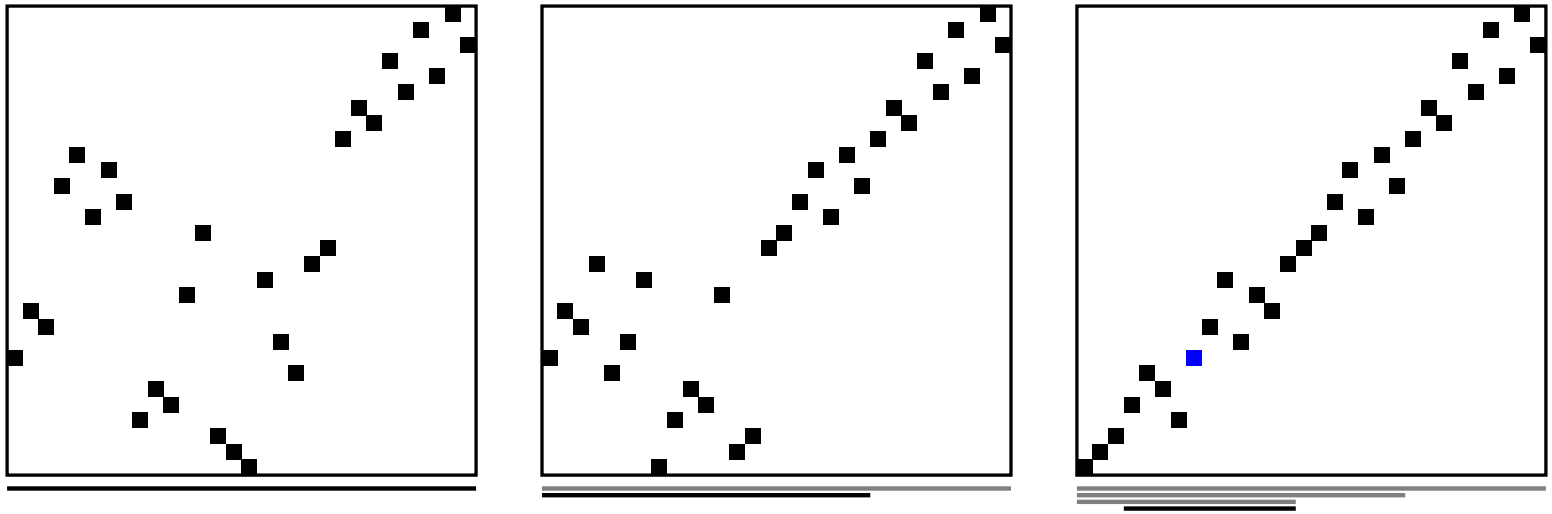
- Rückgabewert = Index des Pivot in Partition

Zerlegung: ein größeres Beispiel



Quicksort

- $n = 30$



Pivotelement

- Wahl des Pivotelements bestimmt Zerlegung
  - Bester Fall: zwei *gleichgroße* Hälften (*Median* als Pivot)
  - Schlechtester Fall: eine Hälfte *leer*
- Median = *mittleres* Element einer *sortierten* Folge

  Hier muss der Median ein Element der Folge sein, d.h. bei ungerader Anzahl das "linke" ("abrunden")!
- Schlechtester Fall (in jedem Rekursionsschritt)
  - Folge ist *invers sortiert* (Pivot wandert nach links)
  - Folge ist *sortiert* (Pivot bereits in Position)
- Bessere Wahl des Pivotelements
  - Zufällige Wahl
  - Median einer kleinen *Teilmenge* z.B. *median-of-three*: median$(a_0, a_{\lfloor n/2 \rfloor}, a_{n-1})$, (typischerweise $\approx 5\%$ Ersparnis [Sedgewick])

Sortieren von Teilfolgen

- Quicksort sortiert Teilfolgen rekursiv (*divide and conquer*)
- Für kleine Teilfolgen kann rekursiver Aufruf relativ teuer sein
  - z.B. Teil besteht nur aus 2 oder 3 Elementen
  - Dann sind `sort2` oder `sort3` sicher effizienter
- Erklärung
  - Quicksort vertauscht Einträge auch über große Distanz.
  - Dieser Vorteil verschwindet bei kleinen Teilfolgen.
  - Gleichzeitig eine gewisse „Vorsortierung“ auf Teilfolgen
  - In diesem Fall ist Insertion Sort linear.
- In der Praxis oft „Umschalten“ auf ein anderes Verfahren
  - z.B. Aufruf von Insertion Sort, wenn $r - l < m$
  - Typischerweise $\approx 20\%$ Ersparnis für $m = 5, \ldots, 25$ [Sedgewick]

Exkurs: Funktionale Programmierung

- *These*: Funktionale P. vereinfacht Entwurf von Algorithmen
- Beispiel: Eine Quicksort Implementierung in Haskell

```
quicksort []     = []
quicksort (p:xs) =
  (quicksort lesser) ++ [p] ++ (quicksort greater)
     where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

- Der Code scheint intuitiv lesbar.
- Keine Indices, stattdessen Listen-Operationen.
- Funktionale Programmierung beschreibt *out-of-place* Variante

Eigenschaften von Quicksort

- Rekursiver Algorithmus nach *Teile und Herrsche* Grundsatz
- *Analyse folgt im Anschluss an Merge Sort*
- Im *Mittel* etwa $1,38\,n \log_2 n$ Vergleiche
- Im *besten Fall* $n \log_2 n$ Vergleiche
- Im *schlechtesten Fall* n^2 Vergleiche
- Quicksort ist *nicht* stabil (Vertauschen bei Zerlegung)
- *Verschiedene Verbesserungen sind möglich*
- Es ist nicht einfach, Quicksort gut zu implementieren!

> *[Quicksort] is fragile: a simple mistake in the implementation can go unnoticed and can cause it to perform badly for some files.*

[Sedgewick]

Mergesort: Sortieren durch „Mischen“

- Anwendung des *Teile und Herrsche* Grundsatzes
- Idee
  1. Teile Folge in zwei gleich große Teile
  2. Sortiere beide Teile unabhängig voneinander
  3. Zusammenführen der sortierten Teilfolgen
- Wo steckt die Rekursion? Sortieren ist ein *rekursiver* Prozess
- Zusammenführen (= *to merge*, auch *verschmelzen*, *mischen*)
  - Vergleiche jeweils die beiden *kleinsten* Einträge
  - „Schiebe“ den kleineren ans Ende der neuen Folge
- Konsequenz: Benötige Zwischenspeicher (*out-of-place*)
- Beobachtung
  - Im Gegensatz zu Quicksort *immer* bestmögliche Aufteilung
  - Einträge in Teilfolgen werden *sequentiell* verarbeitet, d.h. immer einer (a_i) nach dem *nächsten* (a_{i+1}).

Zusammenführen (*merging*)

- Zusammenführen von zwei *sortierten* Folgen (a_i) und (b_j)
- `merge` liefert sortierte Folge (c_k) aus allen Einträgen von a, b

```
static void merge(int[] a,int[] b,int[] c) {
  assert (c.length>=a.length+b.length);

  int i=0,j=0;
  for (int k=0;k<a.length+b.length;++k) {
    if       (i>=a.length) c[k]=b[j++];
    else if (j>=b.length) c[k]=a[i++];
    else if (a[i]<=b[j])  c[k]=a[i++];
    else                  c[k]=b[j++];
  }
}
```

- Anmerkung: Hier wäre der Einsatz von *sentinels* sinnvoll!
- Offensichtlich ist `merge(a,b,c)` stabil!

Mergesort

- Mit Hilfe von `merge` können wir `mergesort` einfach ausdrücken

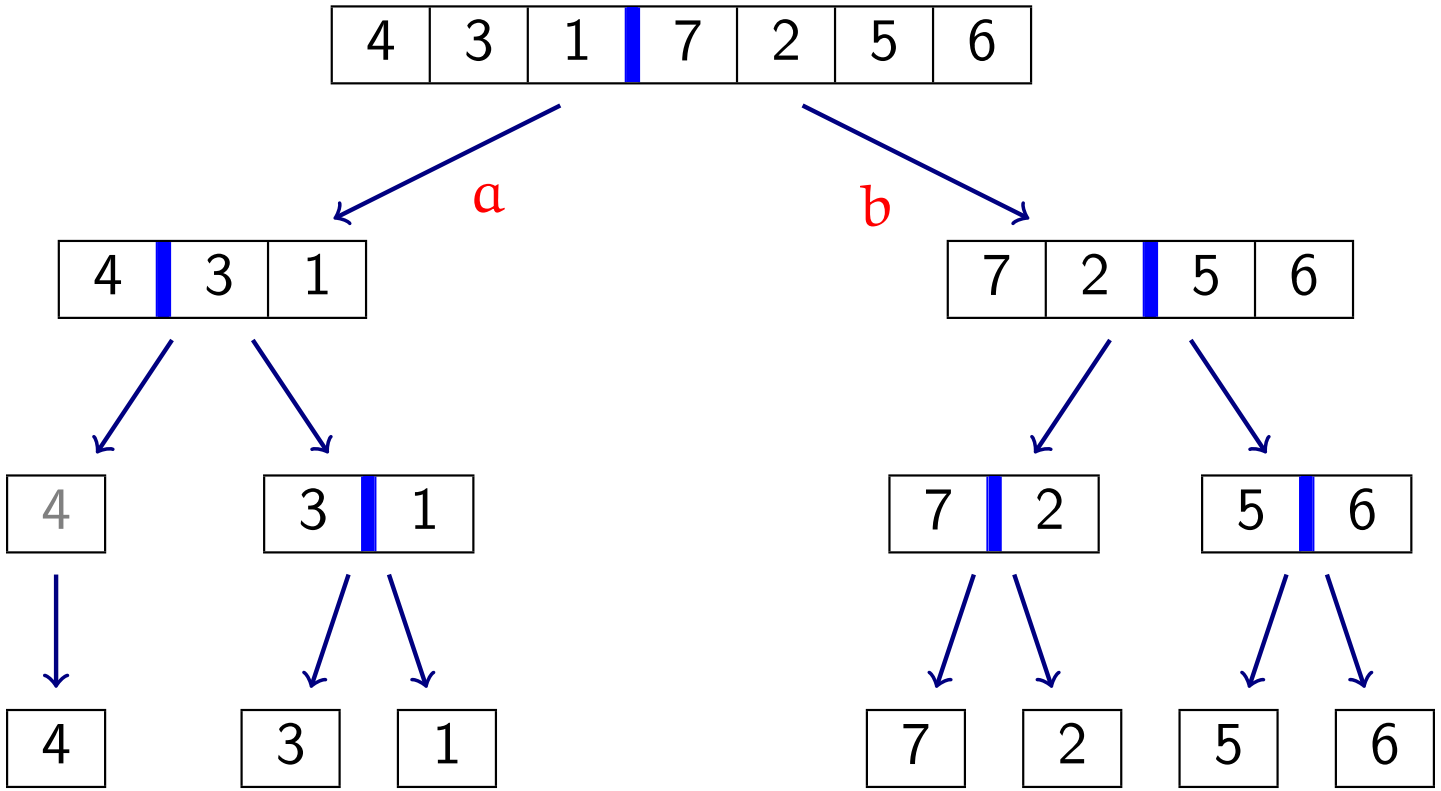
```
public static void mergesort(int[] c) {
  int na=c.length/2;
  int nb=c.length-na;  // mind integer division

  int[] a=new int[na]; // split c: copy parts
  for (int i=0;i<na;++i) a[i]=c[i];

  int[] b=new int[nb];
  for (int j=0;j<nb;++j) b[j]=c[j+na];

  if (a.length>1) mergesort(a); // recursive
  if (b.length>1) mergesort(b); //  sort

  merge(a,b,c);                 // merge
}
```

Umsetzung von Mergesort

- Was ist nicht so gut an dieser Implementierung?
- $\mathtt{c}$ wird in *jedem* Rekursionsschritt aufgeteilt
  - Dabei wird *jedesmal* wieder Speicher für $\mathtt{a}, \mathtt{b}$ angefordert.
  - Das ist unnötig.
- *Annahme*: Wir sortieren eine Folge der Länge $\mathtt{n}$.
- Dann genügt *ein* Feld der Länge $\mathtt{n}$ als Zwischenspeicher!
  - Für *alle* Rekursionsschritte
  - Abgrenzung links/rechts mit Indices $\mathtt{l}, \mathtt{r}$ wie bei Quicksort
  - Übung!

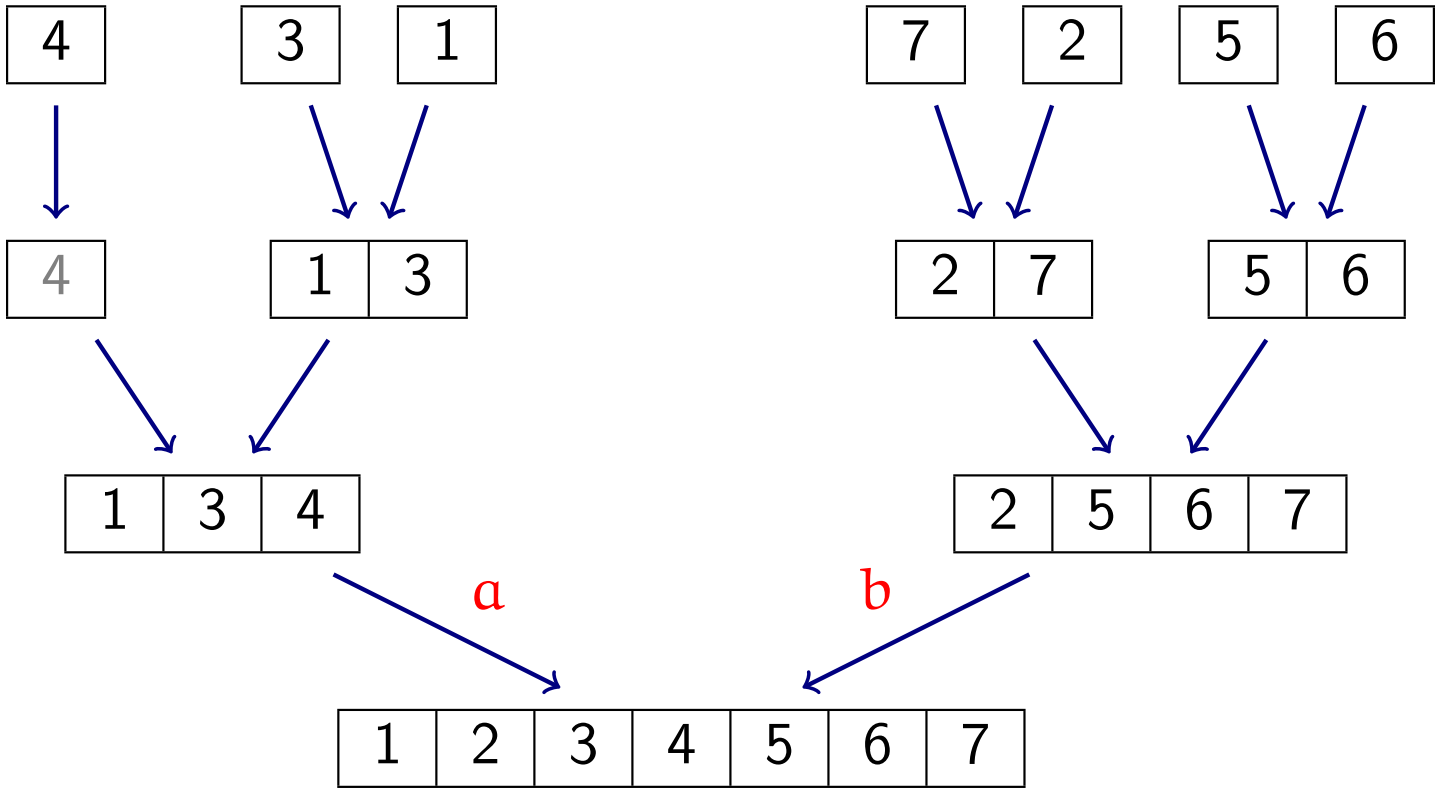
Rekursion in Mergesort: *split*

- 1. Schritt: **Teilen** (*split*)
  - *Entspricht im Java Code Kopieren der Teilfolgen* a, b *aus* c



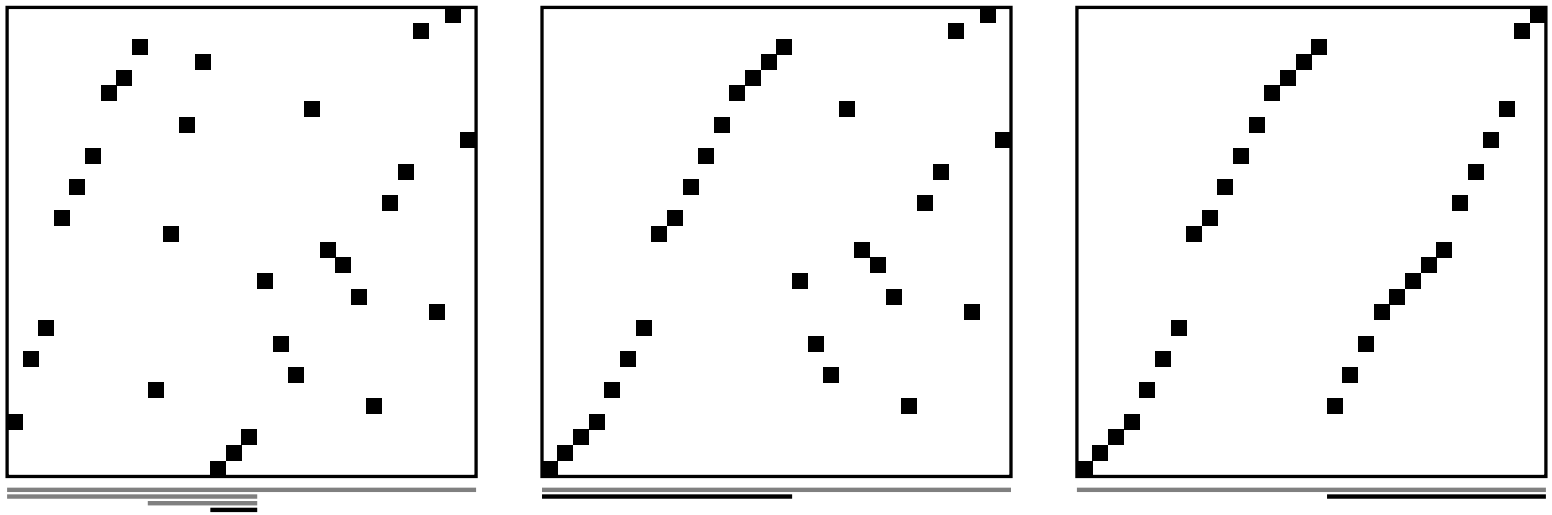
Rekursion in Mergesort: *merge*

- 2. Schritt: **Zusammenführen** (*merge*)
  - *Entspricht Aufruf von* `merge` *im Java Code*



Mergesort

- $n = 30$



Analyse von Mergesort

- `merge` benötigt n Vergleiche für zwei Folgen der Länge je $\frac{n}{2}$
- *Annahme*: Wir sortieren Folge der Länge $n = 2^N$
  - $2\times$ rekursiver Aufruf von `mergesort` für je $\frac{n}{2} = 2^{N-1}$ Einträge
  - `merge` benötigt $n = 2^N$ Vergleiche
- Sei C_N die Anzahl Vergleiche, die zum Sortieren nötig ist. Dann gilt

$$C_N \;=\; 2\,C_{N-1} + 2^N \qquad \text{für } N \geqslant 1 \text{ mit } C_0 = 0\ .$$

- Diese Formel ist ebenfalls *rekursiv* definiert.
  - Für kleine N können wir C_N berechnen, z.B. $C_1 = 2C_0 + 2 = 2, C_2 = 8, C_3 = 24, C_4 = 64, C_5 = 160, \ldots$
- Wir wollen eine *geschlossene* Darstellung von C_N.

Analyse von Mergesort

- Es gilt

$$C_N = 2C_{N-1} + 2^N \quad \text{für } N \geqslant 1 \text{ mit } C_0 = 0$$

- Betrachte

$$\begin{aligned}
C_N &= 2C_{N-1} + 2^N \\
\frac{C_N}{2^N} &= 2\frac{C_{N-1}}{2^N} + 1 = \frac{C_{N-1}}{2^{N-1}} + 1 \\
&= \frac{2C_{N-2} + 2^{N-1}}{2^{N-1}} + 1 = \frac{C_{N-2}}{2^{N-2}} + 1 + 1 \\
&= \frac{C_{N-3}}{2^{N-3}} + 3 = \dots \\
&= N
\end{aligned}$$

- D.h. es gilt $C_N = N \cdot 2^N$
- Wir setzen ein $n = 2^N \Leftrightarrow N = \log_2 n$ und erhalten

$$C(n) = n \log_2 n$$

Analyse von Mergesort und Quicksort

- Mergesort benötigt $n \log_2 n$ Vergleiche
  - Unabhängig von der Reihenfolge der Eingabedaten!
- Gleiche Abschätzung für Quicksort im *besten* Fall
  - Partitionierung immer in gleich große Teile
- Abschätzung für Quicksort im *Mittel*
  - Ähnliche Rekursion, ähnliche Idee zur Auflösung
  - Aber insgesamt komplizierter
  - Es gilt $C_0 = C_1 = 0$ und für $n > 1$

$$C_n = n + 1 + \frac{1}{n} \sum_{1 \leqslant k \leqslant n} \left(C_{k-1} + C_{n-k} \right)$$

  - $n + 1$ Vergleiche für Zerlegung
  - Teile der Größe $k - 1$ und $n - k$ je mit Wahrscheinlichkeit $\frac{1}{n}$
  - Man kann zeigen, dass $C_n \approx 2n \ln n \approx 1,38 n \log_2 n$
  - Bei Interesse: Beweis z.B. in [Sedgewick]

Weitere Eigenschaften von Mergesort

- Mergesort benötigt einen Zwischenspeicher der Größe n.
- Mergesort ist *stabil*.
- Kombination mit anderen Verfahren möglich (wie Quicksort)

- *Out-of-place* Sortierverfahren
- Mergesort ist vor allem als externes Sortierverfahren geeignet.
  - `merge` liest sequentiell und schreibt sequentiell
  - Zwei Eingabeströme und ein Ausgabestrom
  - z.B. Magnetband oder `tar`-Archiv (oder `.tar.gz`)
  - Es muss nur ein kleiner Teil der Daten im Hauptspeicher sein.

Überblick über Sortierverfahren

- Sortieren von n Elementen
- Aufwand = (ungefähre) Anzahl Vergleiche

Algorithmus	bestens	Mittel	schlechtest	stabil
Selection Sort	$n^2/2$	$n^2/2$	$n^2/2$	–
Insertion Sort	n	$n^2/4$	$n^2/2$	✓
Bubble Sort	n	$n^2/2$	$n^2/2$	✓
Quicksort	$n \log_2 n$	$1,38\, n \log_2 n$	n^2	–
Mergesort	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$	✓

- Selection Sort benötigt immer n Vertauschungen.
- Insertion Sort ist linear für (fast) sortierte Folgen.
- Quicksort und Mergesort: *Teile und Herrsche* Grundsatz
- Aufwand für Mergesort ist unabhängig von Eingabereihenfolge.
- Mergesort benötigt Zwischenspeicher (n Elemente).

Übersicht

Anmerkungen zum Suchen und Sortieren

- Permutation von Folgen
  - „Indirektes Sortieren“
- Ordnungsrelationen: lexikographische Ordnung
  - z.B. Vergleich von Zeichenketten
- Sortieren in der Praxis (in Java)
  - Vergleich von Objekten
  - *Interface* `java.lang.Comparable`
  - Erste Berührung mit Java *generics*

Permutationen und Sortierverfahren

- Permutation = Anordnung einer Reihe von Objekten
  - Im Sinn von Veränderung der Anordnung durch Vertauschen
  - Formale Definition als bijektive Abbildung (*siehe Mathematik-Vorlesung*)
- *Mögliche* Darstellung einer Permutation von $\mathbf{a} \in \mathbb{Z}^n$
  - Anordnung der Zahlen $1, 2, \ldots, n$ $n!$ *Möglichkeiten!*
  - in einem Vektor $\mathbf{p} \in \mathbb{N}^n$,
  - so dass $\mathbf{a}_i \to \mathbf{a}_{\mathbf{p}_i}$
- In Java
  - Darstellung als `int[] p` mit `p.length==a.length`
  - Zugriff auf `a[p[i]]` (statt `a[i]`)

Sortieren mit Hilfe von Permutationen

- *Annahme*: Vertauschen von Objekten ist sehr aufwendig
- Erzeuge Permutation, die Sortierreihenfolge herstellt
  - Sortiere „indirekt“: dabei Vertauschen nur auf Feld $\mathtt{p}$
  - Danach $\mathtt{n}$ Vertauschungen (*out-of-place*, d.h. neues Feld!)
- Beispiel

$\mathtt{i}$	0	1	2	3	4
$\mathtt{p}_\mathtt{i}$	0	1	2	3	4
$\mathtt{a}_\mathtt{i}$	8	7	5	9	6

`psort` →

$\mathtt{i}$	0	1	2	3	4
$\mathtt{p}_\mathtt{i}$	2	4	1	0	3
$\mathtt{a}_\mathtt{i}$	8	7	5	9	6

  - Vergleiche dabei `a[p[i]]<a[p[j]]` statt `a[i]<a[j]`
  - Ergebnis $(\mathtt{a}_2, \mathtt{a}_4, \mathtt{a}_1, \mathtt{a}_0, \mathtt{a}_3) = (5, 6, 7, 8, 9)$
- Weitere Anwendungen
  - Schlüssel und Werte in verschiedenen Feldern
  - Zufälliges Mischen (*shuffle*) einer Folge

Lexikographische Ordnung

- Lineare Ordnung für zusammengesetzte Objekte (z.B. Tupel)
  - Beispiel: Zeichenketten
  - `"Aal"` $<$ `"Aberglaube"` oder `"Aal"` $<$ `"Aalfilet"`
- Allgemeines Prinzip
  - Gegeben sind $\mathbf{a}, \mathbf{b} \in \mathcal{X}^n$. Dann soll gelten

$$\mathbf{a} < \mathbf{b} \quad \Leftrightarrow \quad \big(\exists k \geqslant 0\big)\big[a_k < b_k \ \wedge \ \forall(\ell < k)(a_\ell = b_\ell)\big]$$

- In Java (für `char` Felder)

```
bool static less(char[] a,char[] b) {
  for (int i=0;i<a.length;++i)
    if      (i>=b.length) return false;
    else if (a[i]<b[i])   return true;
    else if (a[i]>b[i])   return false;

  return true; // a shorter than b
}
```

Sortieren von Objekten in Java

- Bisher haben wir nur Folgen von `int` betrachtet.
- Wir wollen Folgen von Java *Objekten* sortieren.
  - Dazu benötigen wir Definition einer Ordnungsrelation $<$
- Wir kennen schon den Test von *Gleichheit* von Objekten
  - == vergleicht *Referenzen* (Identität)! — i.d.R. nicht erwünscht
  - `boolean Object.equals(y)` vergleicht „Inhalt“ (Zustand)
  - Idee: *überschreibe* Methode `equals`
- *Interface* `Comparable` definiert `int compareTo(y)`
  - `x.compareTo(y)< 0` $\Leftrightarrow$ $x < y$
  - `x.compareTo(y)==0` $\Leftrightarrow$ $x = y$
  - `x.compareTo(y)> 0` $\Leftrightarrow$ $x > y$
- Viele Klassen implementieren diese Schnittstelle bereits
  - z.B. `String`, `Integer`, `Double`, ... siehe Dokumentation
- Definitionen von `compareTo` und `equals` sollten konform sein!

Sortieren von Objekten in Java

- Sortiere Instanzen der Klasse `Student` nach Matrikelnr. (`id`)

```
class Student implements Comparable<Student> {
  int     id;
  String name;

  public int compareTo(Student other) {
    return this.id-other.id;
  }
}
```

- Sortierverfahren *(1. Versuch)*
  - Ersetze `a[i]<a[j]` durch `a[i].compareTo(a[j])<0`.

```
class SortStudents {
  public static void sort(Student[] a) { ... }
}
```

Sortieren von Objekten in Java

- `SortStudents` kann nur `Student` Objekte sortieren
- Das geht besser bzw. allgemeiner — *2. Versuch*!

```
class Sort {
  public static void sort(Comparable[] a) { ... }
}
```

- Methode `Sort.sort()` sortiert Felder von *beliebigen* Objekten
- Voraussetzung an Klasse: *Implementierung* von `Comparable`
- Alternative: *interface* `java.lang.Comparator`
  - `sort()` sortiert Feld von `java.lang.Object`
  - `Comparator` *Objekt* als zusätzliches Argument an `sort()`
  - Damit *unterschiedliche* Sortierkriterien einfach realisierbar, z.B.

```
Sort.sort(students,new CompareStudentsById());
```

```
Sort.sort(students,new CompareStudentsByName());
```

Generische Schnittstelle `Comparable<T>`

- Gleicher Algorithmus für Vielzahl von Daten (Typen, Klassen)
- Generische Programmierung erlaubt „variable“ Typen
- Beispiel in Java: `Comparable` (ähnlich `Comparator`)
  - Klasse `Student` implementiert `Comparable`
  - D.h. `Student.compareTo()` nimmt als Argument `Student`
  - *Das ist bei der Definition der Schnittstelle nicht bekannt!*
- Definition einer generischen Schnittstelle

```
interface Comparable<T> {
  int compareTo(T other);
}
```

- Anwendung als `Comparable<MyClass>`
  - Ähnlich für Klassen und Methoden
  - Ohne Angabe von `T` implizit `Comparable<Object>`
- Mehr zu *generics* siehe z.B. Java Tutorial

Zusammenfassung

- Binäre Suche: *Teile und Herrsche*
- Sortieralgorithmen
  - Verschiedene Vorgehensweisen
  - Algorithmen mit verschiedenen Eigenschaften
    - Stabilität
    - intern/extern
    - *in-place*/*out-of-place*
- Analyse des Aufwands
  - In Abhängigkeit von der Eingabe
  - Gemessen in Elementaren Operationen (z.B. Vergleiche)
  - Betrachte besten Fall, Mittel und schlechtesten Fall (*best case*, *average*, *worst case*)
- Voraussetzung: Ordnungsrelation
  - In Java: objektorientiert (Überschreiben) und generisch

- Literatur

  [Saake&Sattler] [Sedgewick] [Goodrich&Tamassia]

  Beispiele in Java z.T. online