IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(09 - Weitere Sortierverfahren)

Prof. Dr. Susanne Albers

Heap Sort (1)

Heap Sort:

- Effizientes Verfahren zum Sortieren (auch im worst case in O(n log n))
- Prinzip: Sortieren durch wiederholtes Auswählen des Maximums (ähnlich Selection Sort)
- Besonderheit: Verwendung einer Datenstruktur Heap, welche die Bestimmung des Maximums effizient unterstützt

Definition. *Eine Folge* $F = (k_1, k_2, \ldots, k_n)$ *von Schlüsseln ist ein* (Max-) Heap, *wenn für alle* $i \in \{1, 2, \ldots, n/2\}$ *gilt:* $k_i \geq k_{2i}$ *und* $k_i \geq k_{2i+1}$.

Bsp:

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

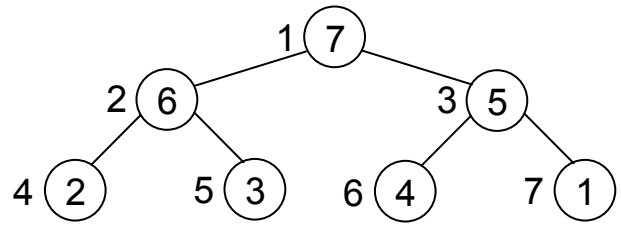
7	6	5	2	3	4	1

Heap-Bedingung

Heap Sort (2)

Veranschaulichung des Heaps durch Binärbaum mit Positionsnummern:

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



- Level *i* hat 2^i Knoten (außer dem letzten)
- Knoten sind von oben nach unten und von links nach rechts nummeriert.
- Knoten *i* hat Knoten *2i* als linken und Knoten *2i + 1* als rechten Nachfolger und Knoten $\lfloor i/2 \rfloor$ als Vorgänger (falls jeweils vorhanden).

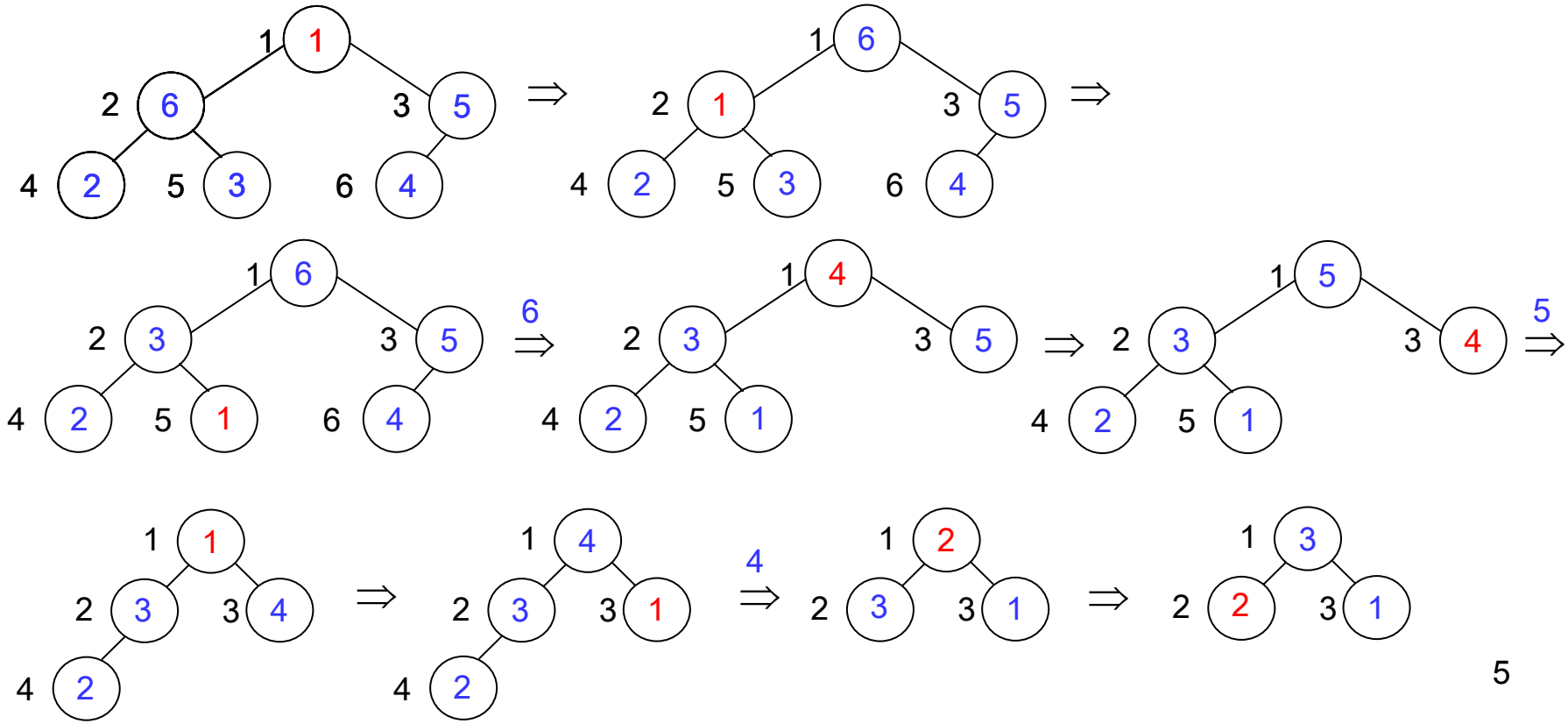
Aus der Heap-Bedingung $k_i \geq k_{2i}$ und $k_i \geq k_{2i+1}$ folgt:

Das Maximum steht an der Wurzel (an Index 1) und kann sofort entfernt werden.

Doch wie erhält man wieder einen Heap für den Rest?

Heap Sort (3)

Das Element von der größten verbleibenden Index-Position wird nach Pos. 1 bewegt und dann versickert. Sobald wieder ein Heap entstanden ist, kann das nächste Maximum entfernt werden, bis der Heap leer ist.



Heap Sort (4)

Versickern bedeutet: Vertauschen mit dem größeren der Nachfolger, solange dieser größer als das zu versickernde Element ist.

Verfahren zum Versickern des ersten Elements in einem Array a in den Grenzen *j* und *t*:

```
private static void percolate (char[] a, int j, int t){
    int h;
    while ((h=2*j) <= t){                    // h: linker Nachfolger
        if (h < t && a[h+1] > a[h]) h++;     // h: größerer Nachfolger
        if (a[h] > a[j]) {                   // versickern nötig?
            swap (a, h, j);                  // ja: vertauschen
            j = h;                           // und weiter
        }
        else break;                          // nein: fertig
    }
} // percolate
```

Anzahl der Vergleiche und Vertauschungen, um ein Element in *n* Elementen zu versickern: O(log *n*)

Heap Sort (5)

Sortieren mit Hilfe eines Heaps:

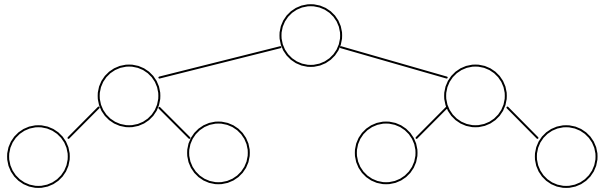
- Erst muss ein Heap erstellt werden.
- Dann kann man immer wieder das jeweilige Maximum entnehmen.
- Da bei *n* Elementen diejenigen mit den Positionen $\lfloor n/2 \rfloor + 1, \ldots, n$ bereits die Heap-Bedingung erfüllen, kann man bei der Heap-Erstellung mit dem Versickern beim Element mit der Nr. $\lfloor n/2 \rfloor$ beginnen.

```
static void heapSort (char[] a){
    int j, hi = a.length-1;
    for (j = hi/2; j >= 1;)        // alle Pos. j = [n/2] .. 1:
        percolate (a, j--, hi);    // versickere im Array j ... N
    for (j = hi ; j > 1;) {        // alle Pos. j = n ... 2:
            swap (a, 1, j);        // vertausche Maximum nach j
        percolate (a, 1, --j);     // versickere im Array 1 ... (j-1)
    }
} // heapSort
```


Frage: Wie aufwändig ist der Heap-Aufbau durch iteriertes Versickern?

Heap Sort (6)

Heap-Aufbau durch iteriertes Versickern:



0	2^0
1	2^1
2	2^2
k = 3	2^k Knoten

Anzahl der Knoten insgesamt: $n = 2^{k+1} - 1$

Anzahl der Vergleiche zum Aufbau:

$$2 * \sum_{i=0}^{k-1} 2^i (k-i) = 2 * \sum_{j=1}^{k} 2^{k-j}\, j$$

$$= 2 * (2^{k+1} - 2 - k)$$

$$= 2 * (n - 1 - k) \in \mathrm{O}(n)$$

Der Heap-Aufbau erfolgt also in linearer Zeit.

Heap Sort (7)

Laufzeit für Heap Sort insgesamt: $O(n \log n)$.

Platzbedarf: n für das Array sowie zusätzlich: $O(1)$.

Frage: Kann Heap Sort noch verbessert werden?

Beobachtungen:

- Pro Niveau, um das ein Element versickert wird, treten 2 Schlüsselvergleiche auf, um den größeren der Nachfolger zu bestimmen und um festzustellen, ob weiter versickert werden muss.
- Im schlimmsten Fall wird immer das kleinste Element bis ganz unten versickert.
- Auch die erwartete Tiefe beim Versickern ist hoch.

Verbesserungsidee:

- Bestimme den größeren der Nachfolger eines Elementes mit einem Schlüsselvergleich.
- Versickere auf jeden Fall (auch wenn das Element zu groß ist).
- Unten angekommen, lasse das Element wieder aufsteigen, soweit nötig.

Bottom-Up Heap Sort (1)

Bemerkung:

Die gefundene Stelle für das Element stimmt mit derjenigen der Versickerungsprozedur überein.

Satz. *Das verbesserte Verfahren Bottom-up Heapsort benötigt zum Sortieren einer Folge von n Schlüsseln im schlechtesten Fall nur 1.5 n log n + 2n Schlüsselvergleiche.*

Im Mittel benötigt Bottom-up Heapsort nur n log n + O(n) Schlüsselvergleiche.

```
static void bottomUpheapSort (char[] a){
    int j, hi = a.length-1;
    for (j = hi/2; j >= 1;)
        percolateb (a, j--, hi); // versickere (mod) von j bis hi
    for (j = hi ; j > 1;) {
            swap (a, 1, j);     // größeres nach j
        percolateb (a, 1, --j);  // versickere (mod)
    }
} // bottomUpheapSort
```

Bottom-Up Heap Sort (2)

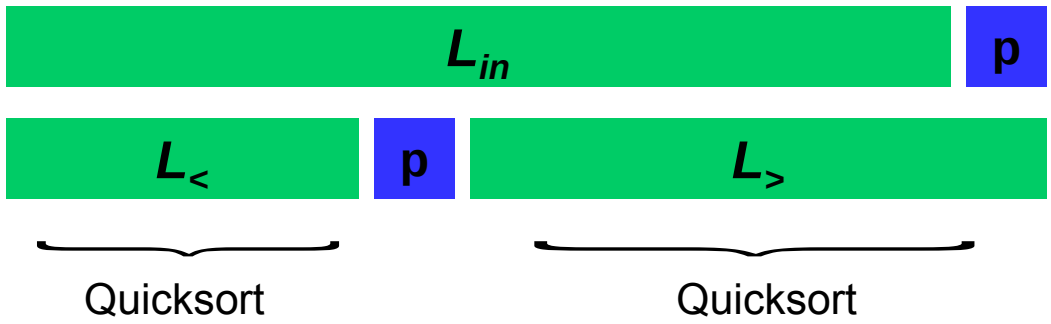
```
private static void percolateb (char[] a, int j, int t){
    int h;
    while ((h=2*j) <= t){                    // h: linker Nachfolger
        if (h < t && a[h+1] > a[h]) h++;     // h: größerer Nachfolger
        swap (a, h, j);                      // tausche auf jeden Fall
        j = h;                               // und weiter
    }
    bubbleUp (a, j);          // tausche wieder zurück wenn nötig
} // percolateb

private static void bubbleUp (char[] a, int j){
    char x = a[j];                       // Position für x gesucht
    for (; j > 1 && a[j/2] < x; j/=2) // bis nach oben, falls nötig:
        a [j] = a [j/2];                 // ziehe Elemente herunter
    a[j] = x; // positioniere x
} // bubbleUp
```

Quicksort (1): Sortieren durch Teilen

- Divide-&-Conquer Prinzip
- Sehr gute Laufzeit im Mittel, schlechte Laufzeit im worst case.

Quicksort Eingabe: unsortierte Liste *L*, Ausgabe: sortierte Liste

```
Quicksort(L):
if (|L| <= 1)
        return L
else waehle Pivotelement p aus L
        L< = {a in L | a < p}
        L> = {a in L | a > p}
        return
          [Quicksort(L<)] + [p] + [Quicksort(L>)]
```



Quicksort (2): Implementation

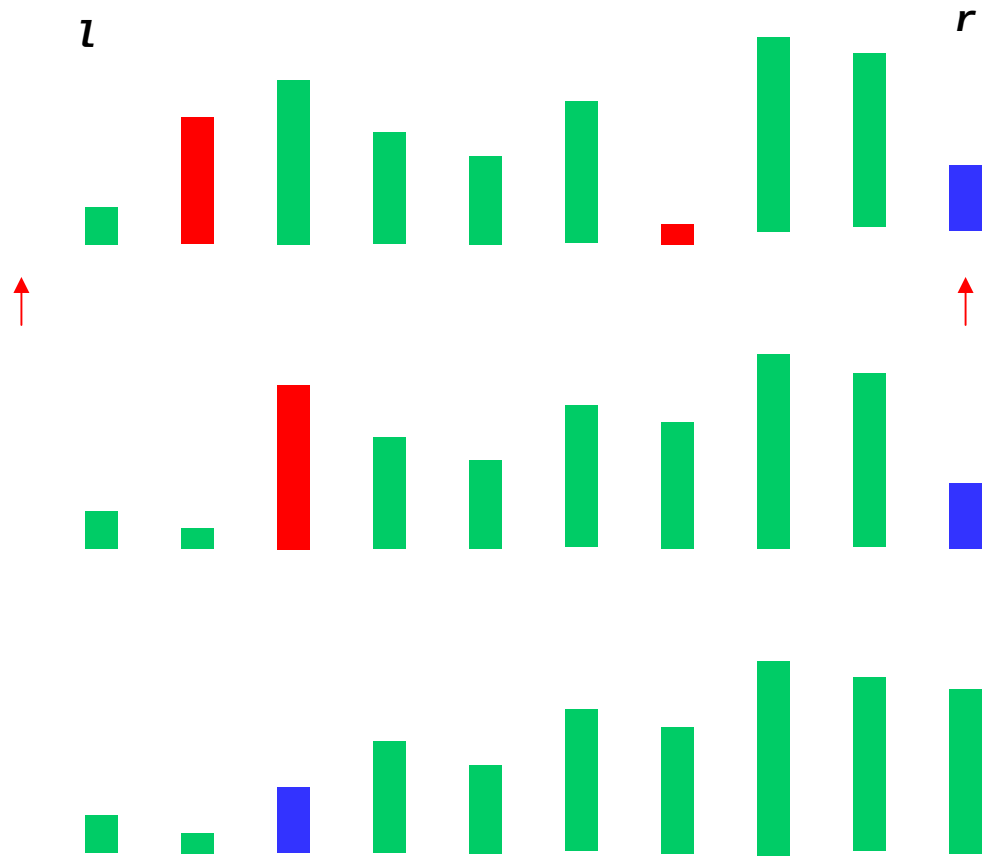
Eingliederung in ein Rahmenprogramm:

```
class QuickSort extends SortAlgorithm {

        static void sort (Orderable A[]){
                /* sortiert das ganze Array */
                sort (A, 1, A.length-1);
        }
        static void sort (Orderable A[], int l, int r){
                /* sortiert das Array zwischen Grenzen l und r */
                if (r > l) { // mind. 2 Elemente in A[l..r]
                        int i = divide(A, l, r);
                        sort (A, l, i-1);
                        sort (A, i+1, r);
                }
        }
        static int divide (Orderable A [], int l, int r) {
                /* teilt das Array zwischen l und r mit Hilfe
                des Pivot-Elements in zwei Teile auf und gibt
                die Position des Pivot-Elementes zurueck */
                ...
        }
}
```

Quicksort (3): Der Aufteilungsschritt

divide(A; l; r):

- liefert den Index des Pivotelements in *A*
- ausführbar in Zeit *O(r – l)*

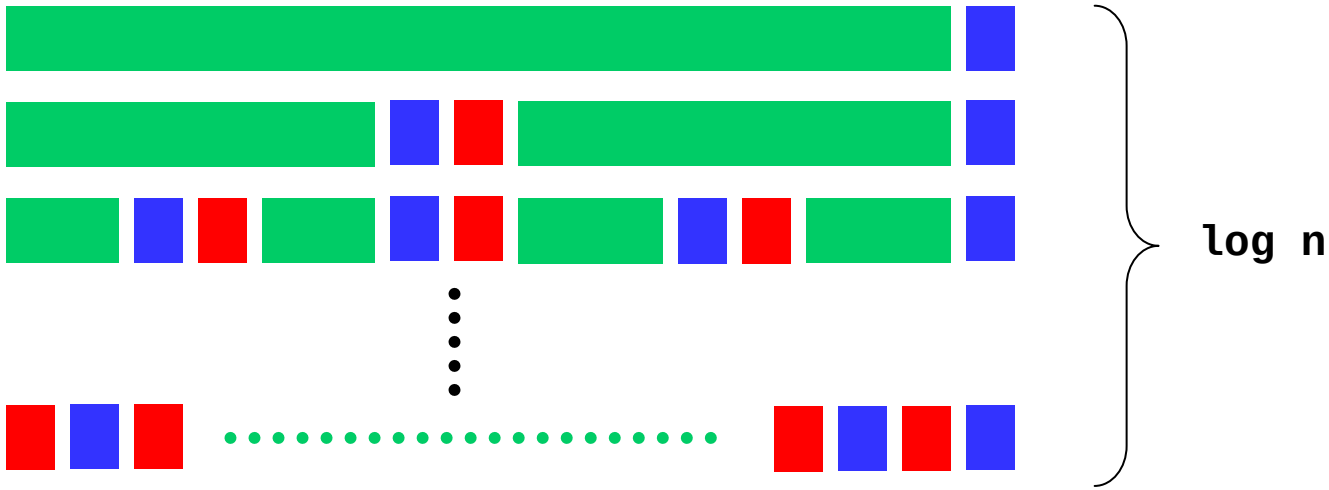


Quicksort (4): Implementation: Aufteilungsschritt

```
static int divide (Orderable A [], int l, int r) {
  // teilt das Array zwischen l und r mit Hilfe
  // des Pivot-Elements in zwei Teile auf und gibt
  // die Position des Pivot-Elementes zurueck
  int i = l-1;                    // linker Zeiger auf Array
  int j = r;                      // rechter Zeiger auf Array
  Orderable pivot = A [r]; // das Pivot-Element
  while (true){ // "Endlos"-Schleife
        do i++; while (i < j && A[i].less(pivot));
        do j--; while (i < j && A[j].greater(pivot));
        if (i >= j) {
              swap (A, i, r);
              return i;       // Abbruch der Schleife
        }
        swap (A, i, j);
  }
}
```

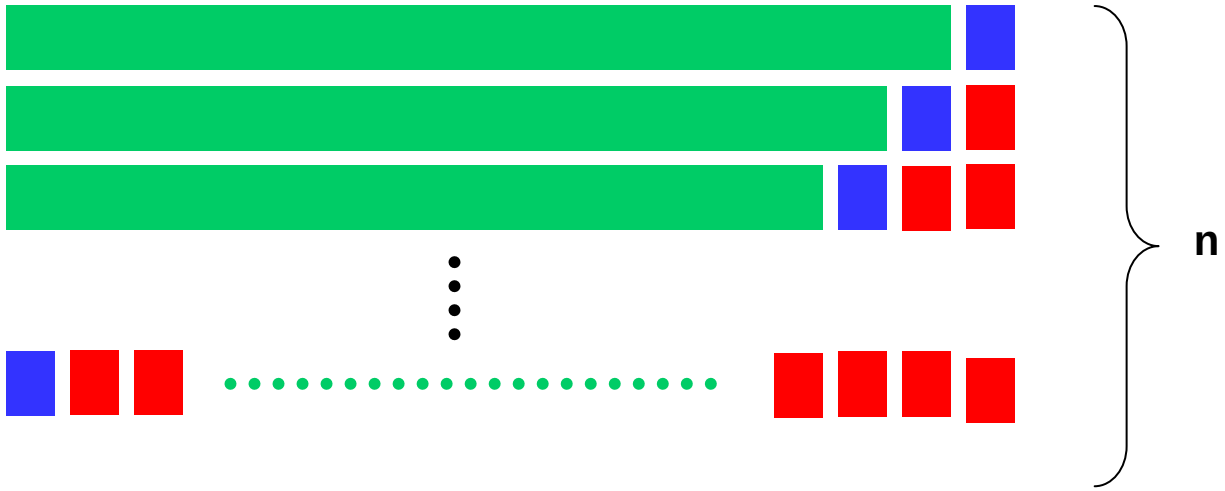
Quicksort (5): Analyse

Günstigster Fall:



$T_{min} = O(n \log n)$

Schlechtester Fall:



$T_{max} = O(n^2)$

Quicksort (6)

- Die Wahl von *w* (das Pivot-Element) ist entscheidend für die Effizienz: Sind F_l und F_r annähernd gleich groß, resultiert Laufzeit in *O(n log n).*
- Eine gute Wahl von *w* wäre ein mittleres Element, allerdings aufwändig zu bestimmen.
- Ein einfacherer Ansatz ist die Wahl des (von der Größe her) mittleren von drei Elementen, die links, rechts und in der Mitte der Folge stehen. Dabei werden sie direkt schon sortiert, womit kleine Probleme (bis zur Größe 3) bereits gelöst sind.

Quicksort (7): Medium-of-Three Variante

```
private static void quickSort (char[] a, int lo, int hi){
    int li = lo;
    int re = hi;
    int mid = (li+re)/2;
    if (a[li] > a[mid]) swap(a, li, mid);    // Kleines Problem
    if (a[mid] > a[re] ) swap(a, mid, re);   // und Vorsortieren
    if (a[li] > a[mid]) swap(a, li, mid);    // per Bubble Sort
    if ((re - li) > 2){                              // Großes Problem:
        char w = a[mid];                             // Divide:
        do{ while (a[li] < w) li++;                  // suche li
            while (a[re] > w) re--;                  // suche re
            if (li <= re) swap (a, li++, re--);      // vertausche
        } while (li <= re);                          // bis fertig
        if (lo < re) quickSort (a, lo, re);  // Conquer links
        if (li < hi) quickSort (a, li, hi);  // Conquer rechts
    } // Merge unnötig
} // quickSort
```

Quicksort (8)

Laufzeitabschätzung:

- Erfolgt die Aufteilung der Folgen jeweils gleichmäßig (best case), so gilt:
  $T(n) = 2T(n/2) + O(n)$ und damit $T(n) = O(n \log n)$
- Im ungünstigsten Fall (worst case) liegt das Pivot-Element ganz links oder rechts:
  $T(n) = T(1) + T(n - 2) + O(n)$ und dann $T(n) = O(n^2)$
- Mittelt man über sämtliche Permutationen von n Schlüsseln und nimmt an, dass die Wahl des Pivot-Elementes immer zufällig erfolgt (average case), so ergibt sich für die Anzahl von Schlüsselvergleichen:

$$T(n) = \frac{1}{n} \sum_{k=1}^{n} (T(k-1) + T(n-k)) + n - 1$$
$$\in O(n \log n)$$

Merge-Sort (1)

- Merge-Sort ist ein Divide-&-Conquer Verfahren mit simplem Divide- und etwas aufwändigerem Merge-Schritt.
- Die Datensätze der zu sortierenden Folge werden, wenn sie nicht klein genug zum direkten Sortieren sind, in zwei annähernd gleich große Teilfolgen aufgeteilt.
- Die Teilfolgen werden nach dem gleichen Verfahren sortiert.
- Anschließend werden die sortierten Teilfolgen zusammengemischt (merge).
- Beim Mischen müssen jeweils nur die ersten Elemente beider Folgen verglichen werden, das kleinere wird entfernt und kommt in die Ergebnisfolge.
- Ist eine der Teilfolgen ganz ausgeschöpft, müssen die restlichen Elemente der verbleibenden Teilfolge nur noch umkopiert werden.
- Eine Teilfolge mit nur einem Element ist bereits sortiert.

Merge-Sort (2)

- Der nicht-rekursive Einstieg in Merge-Sort:

```
static void mergeSort(char[] a){
        int hi = a.length-1;
        mergeSort (a, 0, hi); // die rekursive Variante
} // mergeSort
```

- Das Zusammenmischen der Teilfolgen kann nicht im gleichen Array erfolgen.
- Es wird ein Hilfs-Array in der Größe der Ergebnisfolge benötigt.

Merge-Sort (3)

```
private static void mergeSort (char[] a, int lo, int hi){
    if (lo < hi) { // Grosses Problem:
        int m = (lo + hi +1)/2;              // Divide
        mergeSort (a, lo, m-1);                // Conquer links
        mergeSort (a, m, hi);                  // Conquer rechts
                                             // Merge nach
        char [] temp = new char[hi-lo+1];      // Hilfs-Array
        for (int i=0, j=lo, k=m; i<temp.length;) // bis voll
            if ((k > hi) || (j < m) && (a[j] < a[k])) // lazy!
                temp[i++] = a[j++];     // von links
            else temp[i++] = a[k++];    // von rechts
        for (int i=0; i<temp.length; i++)    // Kopiere zurueck
            a[lo+i] = temp[i];                 // von Hilfs-Array
    }
} // mergeSort
```

Merge-Sort (4)

Aufwandsabschätzung:

- Die Folgen werden in allen Fällen (best, worst, average) immer gleichmäßig aufgeteilt.
- Das Zusammen-Mischen erfordert einen linearen Aufwand in der Länge der Ergebnisfolge.
- Für die Laufzeit ergibt sich:

$$T(n) \leq 2T(n/2) + O(n)$$ und somit $T(n) \in O(n \log n)$

- Platzbedarf: Es wird ein zweites Array benötigt sowie Speicherplatz (Stack) zur Verwaltung der rekursiven Aufrufe.

Merge-Sort (5)

Optimierungsansätze:

- Der Merge-Sort Algorithmus – wie angegeben – ist übersichtlich und leicht verständlich. Er kann aber zu Lasten dieser Eigenschaften weiter optimiert werden.
- Wie bei Quicksort, könnten weitere einfache Fälle ($n \in \{1,2,3,...\}$) direkt behandelt werden.
- Das ständige Neugenerieren (und Verwerfen) von Hilfs-Arrays verbraucht unnötig Zeit. Effizienter ist die (Wieder-) Verwendung eines einzigen Arrays der gleichen Größe wie das Ausgangs-Array. Zudem kann man Kopieroperationen einsparen.
- Ist eine Teilfolge beim Mischen ausgeschöpft, kann der Rest der anderen ohne unnötige Abfragen umkopiert werden.

Bemerkungen zur Anwendung:

- Im Vergleich zu anderen Internen Sortierverfahren schneidet Merge-Sort nicht so gut ab, weshalb es selten tatsächlich eingesetzt wird.
- Gut an Merge-Sort ist sein garantiertes worst case Verhalten.
- Varianten von Merge-Sort werden häufig beim Externen Sortieren eingesetzt.

Distribution-Sort (1)

- Bisher betrachtete Sortierverfahren sind Allgemeine Verfahren: Die erlaubten Operationen sind Schlüsselvergleiche und Elementvertauschungen.
- Allgemeine Sortierverfahren benötigen im worst case immer Zeit in $\Omega(n \log n)$, bei einigen kann $O(n \log n)$ garantiert werden.
- Distribution-Sort (Sortieren durch Fach-Verteilen) kommt ganz ohne Vergleiche aus.
- Dafür ist ein Zugriff auf die Struktur des Schlüssels erlaubt.
- Die zu sortierenden Elemente werden mehrfach nach jeweils einem Teil des Schlüssels in Fächer verteilt und wieder zusammengetragen.
- Vorbilder sind klassische mechanische Sortieranlagen etwa für Briefe nach Postleitzahlen.

Distribution-Sort (2)

Nachteile:

- Es ist nicht ganz leicht, das Vorbild (Fach-Verteilen von Briefen) überhaupt auf den Rechner zu übertragen.
- Im konkreten Fall kann es schwer (oder sehr aufwändig) sein, das Verfahren an einen
- bestimmten Sortierschlüssel anzupassen.

Vorteile:

- Die besondere Effizienz des Verfahrens.
- Die Laufzeit von Distribution-Sort ist linear in der Größe der Eingabe.
- Wenn anwendbar, ist Distribution-Sort auch in der Praxis den bisherigen Verfahren überlegen.

Distribution-Sort (3)

Prinzip beim Sortieren von Briefen:

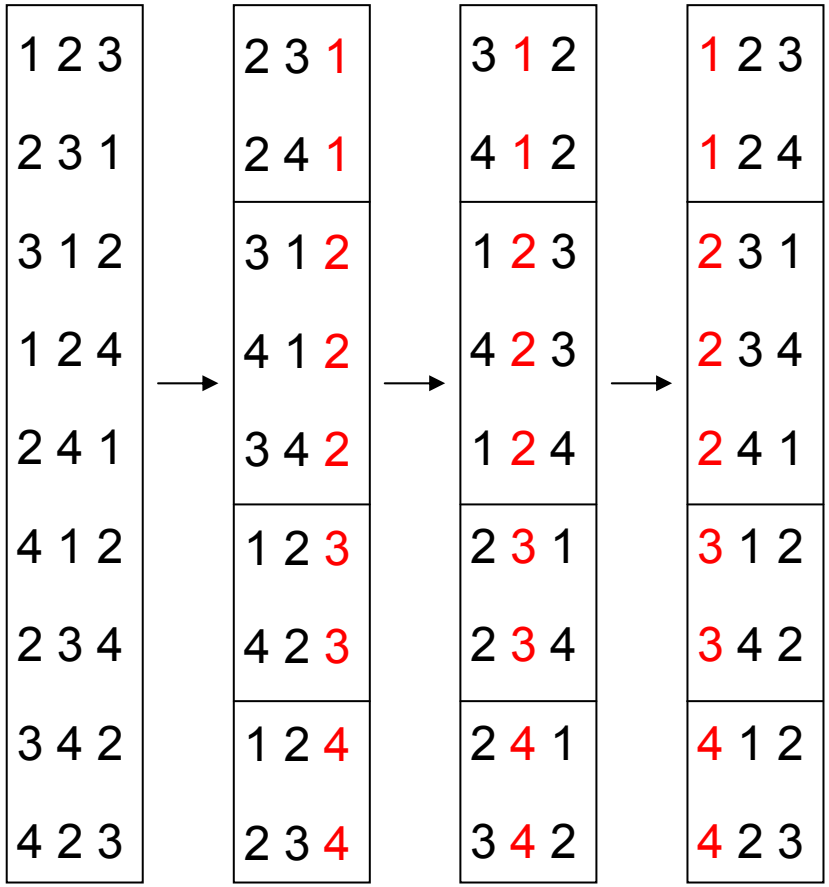
1. Die letzte Ziffer der Postleitzahl sei die aktuelle Ziffer.
2. Verteile alle Briefe in 10 Fächer mit Nummern 0, 1, . . . , 9 entsprechend der aktuellen Ziffer.
3. Lege alle Briefe unter Beibehaltung der bisherigen Ordnung zusammen.
4. Ist die aktuelle Ziffer die erste Ziffer, dann sind alle Briefe fertig sortiert.
   Ist das nicht der Fall, wird die Ziffer links der aktuellen Ziffer zur neuen aktuellen Ziffer, und es geht weiter bei Schritt 2

Distribution-Sort: Beispiel

1 2 3, 2 3 1, 3 1 2, 1 2 4, 2 4 1, 4 1 2, 2 3 4, 3 4 2, 4 2 3

Distribution-Sort (3)

Beispiel für 3-stellige Postleitzahlen:



Nach 3 (# der Ziffern) Durchgängen sind alle Zahlen sortiert.

Distribution-Sort (4)

Probleme bei der Umsetzung:

- Man muss beachten, wie der Schlüssel aufgebaut ist (Folge von Ziffern, Bytes, Bits, . . .) und ob die Sortierung nach den Teilen eine Sortierung der Schlüssel ergibt.

  Dies ist etwa für Zahlen in Zweierkomplement-Darstellung nicht gegeben, kann durch Anpassung aber erreicht werden.

- Bei Schlüsselzerlegung in Bytes z.B. benötigt man 256 Fächer, wovon jedes im Prinzip alle Datensätze aufnehmen können muss.

  Durch Vorinspektion kann aber die # der Datensätze für jedes Fach – und damit seine Index-Position im Array – ermittelt werden.

Distribution-Sort (5)

Distribution-Sort für char[] mit 256 Fächern (1 Durchgang):

```
static void distributionSort (char[] a){
    int hi = a.length-1;
    char[] b = new char[hi+1]; // Hilfs-Array zum Umkopieren
    int[] count = new int[256]; // Zaehler, 0 initialisiert
    for (int i=0; i <= hi; i++)
        count[ (int) a[i] ]++; // Zeichen zaehlen in Fach
    for (int i=1; i < 256; i++)
        count[i] += count[i-1]; // Akkumulieren
    for (int i = hi; i >= 0; i--) // von hinten nach vorne:
        b[--count[ (int) a[i] ]] = a[i]; // umsortieren von a nach b
    System.arraycopy(b,0, a,0, hi+1); // Array b nach a kopieren
} // distributionSort
```

count:	-	…	a	b	c	d	e	f	g	h	…
#	1	0	1	1	1	1	3	1	1	2	
∑#	1	1	2	3	4	5	8	9	10	12	

Distribution-Sort (6)

Diskussion:

- Der angegebene Algorithmus `distributionSort` interpretiert einzelne Zeichen (char) als Bytes, funktioniert also nur bei Codierungen im Bereich 0, . . . , 255.
- Bei längeren Schlüsseln, die wirklich zerlegt werden müssen, sind entsprechend mehr Durchgänge erforderlich.
- Die Anzahl der Durchgänge ist eine Konstante.
- Pro Durchgang werden linear viele Operationen ausgeführt.
- Die Komplexität des Verfahrens ist in $O(n)$.
- Die Bedingung einer Zerlegbarkeit der Schlüssel in Teile, deren Sortierung die Ordnung auf den Schlüsseln erhält, muss erfüllbar sein, damit das Verfahren angewandt werden kann, wie etwa bei
  – 5-stelligen positiven Zahlen (Postleitzahlen)
  – Strings der Länge ≤ 20 (fest, aber beliebig)
  – ganzen und reellen Zahlen (nach Anpassung)
- Das Verfahren lässt sich noch weiter optimieren, etwa durch Vermeiden des Rückkopierens aus dem Hilfs-Array. Statt dessen wird in aufeinander folgenden Durchgängen die Rolle der beiden Arrays vertauscht.

Distribution-Sort (7)

- Distribution-Sort für char[] mit 2 Fächern (8 Durchgänge, 1 pro Bit):

```
static void distributionSortBit (char[] a){
    int hi = a.length-1;
    char[] b = new char[hi+1];    // Hilfs-Array zum Umkopieren
    for (int j=0; j< 8; j++){     // fuer jedes Bit des Schluessels:
        int[] count = new int[2]; // Zaehler, 0 initialisiert
        for (int i=0; i <= hi; i++)
            count[ (a[i]>>>j) & 1 ]++; // Zeichen zaehlen in Fach
        for (int i=1; i < 2; i++)
             count[i] += count[i-1];     // Akkumulieren
        for (int i = hi; i >= 0; i--) // von hinten nach vorne:
             b[--count[(a[i]>>>j) & 1]] = a[i]; // umsortieren: a -> b
        char[] h = a; a = b; b = h;      // Array-Referenzen umsetzen
    } //for
} // distributionSortBit
```

Distribution-Sort (8)

Distribution-Sort für char[] mit 2 Fächern (8 × 2 Durchgänge, ohne count-Array):

```
static void distributionSortBit2 (char[] a){
    int hi = a.length-1;
    char[] b = new char[hi+1];     // Hilfs-Array zum Umkopieren
    for (int j=0; j< 8; j++){      // fuer jedes Bit j des Schluessels:
        for (int k=1, m=hi; k >= 0; k--)  // fuer k in {1,0}
            for (int i = hi; i >= 0; i--)         // i Zeiger in a:
                if ((a[i]>>>j & 1) == k)  // j-tes Bit == k?
                    b [m--] = a [i];      // umsortieren a->b
        char[] h = a; a = b; b = h;       // Array-Referenzen umsetzen
    } //for
} // distributionSortBit2
```

Auswahl von Sortier-Verfahren

Welches ist das beste?

- Bei wenigen Datensätzen (≤ 1000) ist die Laufzeit kaum problematisch, das einfachste Verfahren kann bevorzugt werden (Insertion, Selection, Bubble, . . .).
- Sind die Daten fast sortiert, sollte ein Verfahren genommen werden, das Vorsortierung ausnutzt (Insertion, Bubble, . . .).
- Sind sehr viele zufällig sortierte Elemente häufig zu sortieren, ist der Aufwand zur Anpassung von Distribution-Sort gut investiert.
- Will man möglichst flexibel bleiben und hat keine Angst vor dem worst case, kann man Quick-Sort einsetzen.
- In den restlichen Fällen wird man Shell-Sort, Merge-Sort oder Heap-Sort wählen.