Begriffe:
stabil: Elemente mit dem selben Sortierkriterium bleiben nach Sortierung in Ihrer ursprünglichen Reihenfolge.
in-situ (in-place): Es wird kein zusätzlicher Speicherplatz benötigt

- vergleichsbasierte Sortierverfahren

| Bezeichnung | Best Case | Average Case | Worst Case | stabil | in situ |
|---|---|---|---|---|---|
| Einfache Sortierverfahren | | | | | |
| Sortieren durch Auswählen (SelectionSort) | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | (leicht mgl.) | (wenn mit Array) |
| Sortieren durch Einfügen (InsertionSort) | O(n) | $O(n^2)$ | $O(n^2)$ | X | (wenn mit Array) |
| Blasensortierung (BubbleSort) | O(n) | $O(n^2)$ | $O(n^2)$ | X | X |
| Verfeinertes Auswählen | | | | | |
| Haldensortieren (HeapSort) | O(n log n) | O(n log n) | **O(n log n)** | | X |
| Teile-&-Herrsche/Divide-&-Conquer-Verf. | | | | | |
| Sortieren durch Verschmelzen (MergeSort) | O(n log n) | O(n log n) | **O(n log n)** | X | (in der Regel nicht) |
| Sortieren durch Zerlegen (QuickSort) | O(n log n) | O(n log n) | $O(n^2)$ | | X |

- nicht-vergleichsbasierte Sortierverfahren

| Sortieren durch Fachverteilen | Zeit | stabil | in situ |
|---|---|---|---|
| BucketSort | O(n + m) | X | |
| RadixSort | O(n · k) | X | |

Beweisbar, dass vergleichsbas. Sortieren nicht besser als in O(n log n) möglich ist!

Bedeutung von m und k folgt im entsprechenden Abschnitt.

mm Algorithmen und Datenstrukturen • Philippsen/Stamminger/Pflaum/

SelectionSort:
Gehe durch die Liste und lösche das Maximum und füge dieses vorne an die Ergebnisliste an.
In-Situ: Wenn man Arrays statt Listen nimmt und das Maximum nach hinten tauscht.
stabil: Indem man immer das letzte Maximum löscht und an die Ergebnisliste anhängt.
Entweder Rekursiv oder Iterativ

```
public LinkedList<E> selsort() {
    // Kopie der zu sortierenden Liste l0
    LinkedList<E> l = new LinkedList<E>();
    l.addAll(this);
    // Erzeuge leere Ergebnisliste ls
    LinkedList<E> ls = new LinkedList<E>();
    // Aufruf der Rekursion
    return selsortRec(l, ls);
    }
LinkedList<E> selsortRec(LinkedList<E> l, LinkedList<E> ls) {
    if (!l.isEmpty()) {
        E max = getMaximum(l);
        l.remove(max);
        ls.addFirst(max);
        return selsortRec(l, ls);
    } else return ls;
}
```

```
//entrekursiviert:
while (!l.isEmpty()) {
    E max = getMaximum(l); // Maximum auswaehlen
    l.remove(max); // Maximum entfernen
    ls.addFirst(max); // Maximum an Ergebnisliste vorne anfuegen
}
return ls;
```

O-Notation: $O(n^2)$ [n mal vergleichen bei n-Durchläufen]

InsertionSort:
Gehe durch die Liste, nehme das nächste Element und füge es in der Ergebnisliste an der Richtigen Stelle ein. In-situ: Nehme das nächste (rechte) Element und vergleiche nach links bis das linke Element kleiner ist.
stabil: Da man immer so lange nach links geht bis das linke Element der Ergebnisliste kleiner-gleich ist bleibt die Ordnung erhalten.

ex-situ, mit Listen

```
public LinkedList<E> insort() {
    LinkedList<E> l = new LinkedList<E>();
    l.addAll(this);
    LinkedList<E> ls = new LinkedList<E>();
    while (!l.isEmpty()) {
        E e = l.get(0);
        l.remove(e);
        // suche in ls e', e" aufeinanderfolgend
        // mit e' ≤ e < e";
        // fuege in ls e zwischen e', e" ein
        }
    return ls;
}
```

in-situ, mit Array

```
public void insort() {
    E e;
    int j;
    E[] a;
    for (int i = 1; i < a.length; i++) {
        e = a[i]; // entnommenes Element merken
        j = i - 1;
        while (j >= 0 && e.compareTo(a[j]) < 0) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = e; // entnommenes Element einsetzen
    }
}
```

O-Notation: O($n^2$)

Anmerkung: Im Best-Case nur O(n), da die Liste einfach einmal durchlaufen wird.

BubbleSort:

Das Array wird immer wieder durchlaufen und dabei werden benachbarte Elemente in die richtige Reihenfolge gebracht. Größere Elemente überholen so die kleineren und drängen an das Ende der Folge (wie aufsteigende Luftblasen):

```
static <E extends Comparable<E>> void bubblesort(E[] a) {
    // Variable zum merken, ob beim aktuellen Durchlauf der Sequenz eine
    // Vertauschung stattfand
    boolean swapped;
    // oberster Index des Arrays a, bis zu dem noch korrekte Ordnung geprueft
    // werden muss
    int upper = a.length - 1;
    do { swapped = false;
    for (int i = 0; i < upper; i++) {
        if (a[i].compareTo(a[i + 1]) > 0) {
            // tausche im Array a die
            // Eintraege an den Indizes
            // i und i + 1 swap(a, i, i + 1);
            // merke: es wurde getauscht
            swapped = true;
            }
        } upper--;
        } while (swapped);
    }
}
```

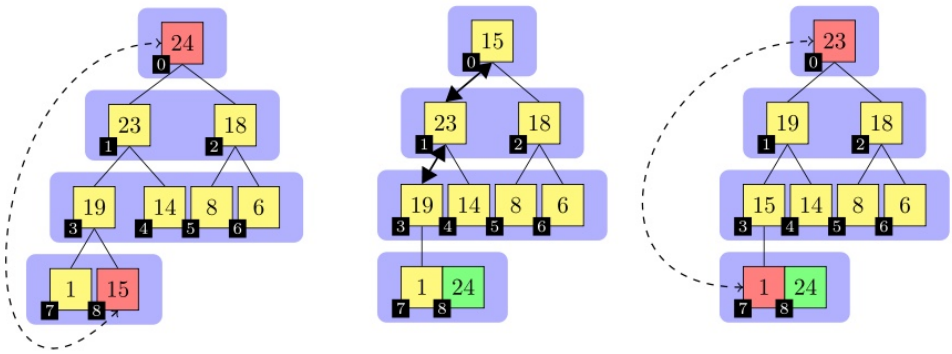
Aufwand: O($n^2$)

Durch die swapped Funktion sinkt der Aufwand im Best Case auf O(n) (falls vorsortiert)

BubbleSort wird daher verwendet um annähernd sortierte Listen mit einzelnen Swapfehlern (zB bei Grafikberechnungen), in linearer Laufzeit zu korrigieren.

HeapSort

Die Liste wird in einen MaxHeap geschrieben. Dann entnimmt man den Wert des obersten Knotens (Im MaxHeap der höchste Wert) und tauscht es mit dem niedrigsten (das rechteste der letzten Reihe im Heap). Dann lässt man diesen Wert „versickern“, das heißt es wird immer mit dem höherwertigen Kind getauscht, bis der Heap wieder die Max-Bedingung erfüllt:




usw.

```
public void heapsort(E[] a) {
    int n = a.length;
    // Phase 1: Halde aufbauen
    for (int i = n / 2; i >= 0; i--) {
        reheap(a, i, n - 1);
    }
    // Phase 2: jeweils Maximum entnehmen und sortierte Liste am Ende aufbauen
    for (int i = n - 1; i > 0; i--) {
        // Maximum ans Ende des Haldenbereichs tauschen
        swap(a, 0, i);
        // nach vorne getauschtes Element einsinken lassen und
        // max-Halden-Eigenschaft wieder herstellen
        reheap(a, 0, i - 1);
    }
}
protected void reheap(E[] a, int i, int k) {
    int leftKidIdx = 2 * i + 1;
    int rightKidIdx = leftKidIdx + 1;
    int kidIdx;
    if (leftKidIdx <= k && rightKidIdx > k) {
    // nur ein (= linkes) Kind
        if (a[leftKidIdx].compareTo(a[i]) > 0) {
            swap(a, leftKidIdx, i);
        }
    } else {
        if (rightKidIdx <= k) {
            // in kidIdx groesseren der beiden Kinder erfassen
            kidIdx = a[leftKidIdx].compareTo(a[rightKidIdx]) > 0
                    ? leftKidIdx : rightKidIdx;
            if (a[kidIdx].compareTo(a[i]) > 0) { // gfs. tauschen
                swap(a, i, kidIdx);
                reheap(a, kidIdx, k);
            }
        }
    }
}
```

Durch den Haldenaufbau wird jegliche Sortierung gleicher Keys ignoriert => nicht stabil
Phase 1: O(n)
Phase 2: O(n log n) (Binärbaum)
=> ergibt O(n log n) für jeden Case

MergeSort:
Unterteile die Liste der Länge n in n Sublisten der Länge 1. Merge benachbarte Listen bis es wieder eine geordnete Liste gibt.

```
static <E extends Comparable<? super E>> LinkedList<E>
    mergesort(LinkedList<E> l) {
    // Basisfall
    if (l.size() < 2) return l;
    // zerlege l in 2 ungefaehr gleich lange Teillisten
    LinkedList<E> left = new LinkedList<>(l.subList(0, l.size() / 2));
    LinkedList<E> right = new LinkedList<>(l.subList(l.size() / 2, l.size()));
    // sortiere beide Teillisten und verschmelze sie
    return merge(mergesort(left), mergesort(right));
}
```

```java
static <E extends Comparable<? super E>> LinkedList<E>
    merge(LinkedList<E> l, LinkedList<E> r) {
    LinkedList<E> ret = new LinkedList<>();
    while (l.size() > 0 && r.size() > 0) { // Reissverschluss
        E left = l.getFirst(), right = r.getFirst();
        if (left.compareTo(right) <= 0) { //stabil wegen =
            ret.addLast(left);
            l.removeFirst();
        } else {
            ret.addLast(right);
            r.removeFirst();
        }
    }
    while (l.size() > 0) { // uebrige Werte von links
        ret.addLast(l.getFirst());
        l.removeFirst();
    }
    while (r.size() > 0) { // uebrige Werte von rechts
        ret.addLast(r.getFirst());
        r.removeFirst();
    }
    return ret;
}
```

MergeSort ist Stabil und im Schnitt schneller als QuickSort, da weniger Vergleiche. Aufgrund der Rekursion ist es jedoch Speicheraufwändiger.
Die Laufzeit beträgt durch das Aufspalten und mergen in jedem Fall O(n log n)

<u>QuickSort</u>
Zerlege die Liste anhand eines Pivots („Drehpunkt“) in Teillisten. Sortiere diese Teillisten bis Pivot auf einelementiger Liste basiert.

```java
protected int partition(E[] a, int m, int n) {
    // {P: (0 ≤ m ≤ n < a.length)}
    // a' := java.util.Arrays.copyOf(a, a.length); <- „virtual copy“
    E p = a[n];
    int i = m - 1;
    int j = m;

    while (j < n) {
        // {I: m - 1 ≤ i < j ≤ n ∧ perm(a[m..n],a'[m..n]) ∧
        //      ∀ k: ((m ≤ k ≤ i ⇒ a[k] ≤ p) ∧ (i < k < j ⇒ p < a[k]))}
        if (a[j].compareTo(p) <= 0) {
            i++;
            swap(a, i, j);
        }
        j++;
    }
    // {S: I ∧ ¬b}
    int r = i + 1;
    swap(a, r, n);
    // {Q: (m ≤ r ≤ n) ∧ perm(a[m..n],a'[m..n]) ∧
    //      ∀ k:((m ≤ k ≤ r ⇒ a[k] ≤ a[r]) ∧ (r < k ≤ n ⇒ a[r] < a[k]))}
    return r;
}
```

Falls ein anderes Element als Pivot ausgesucht wurde, muss es vorher an die Stelle `a[n]` getauscht werden.

nach Umsortierung: Pivot-Element an die Nahtstelle zwischen dem Bereich mit Werten $\leq p$ und Werten $> p$ tauschen.

rekursiv:

```
public void quicksort(E[] a) {
    quicksortRec(a, 0, a.length - 1);
}
protected void quicksortRec(E[] a, int m, int n) {
    if (n > m) {
        int r = partition(a, m, n);
        quicksortRec(a, m, r - 1);
        quicksortRec(a, r + 1, n);
    }
}
```

Aufwand QuickSort: Normalerweise O(n $\log_2 n$). Es findet sich allerdings zu jeder Pivotwahl ein Worst-Case mit O($n^2$).
Pivotwahl, um Wahrscheinlichkeit des O($n^2$)-Falls zu reduzieren:

- 3-Median-Strategie: Wähle drei Elemente vom linken und rechten Rand und aus der Mitte des zu sortierenden Ausschnitts. Wähle als Pivot-Element das mittlere dieser drei Elemente.
- Zufallsstrategie: Wähle mit Zufallszahlengenerator (gleichverteilt) die Stelle des Pivot-Elements im zu sortierenden Ausschnitt aus.

Radix-Sort

Statt für jeden in Frage kommenden Wert ein Fach anzulegen, wird der Wert in (kürzere) Segmente aufgeteilt (Buckets). z.B. bei Zahlen: 10 Buckets (für 0-9)
Dann werden entweder von rechts nach links (Least Significant Digit zuerst) oder andersherum (Most Significant Digit zuerst) die Werte aus der Liste in die Buckets zugeteilt und zum Schluss der Reihe nach in die Folgeliste übertragen, solange bis jedes Segment bearbeitet wurde.
Somit ergibt sich ein Aufwand von O(n*k) für n Elemente mit k Segmenten.