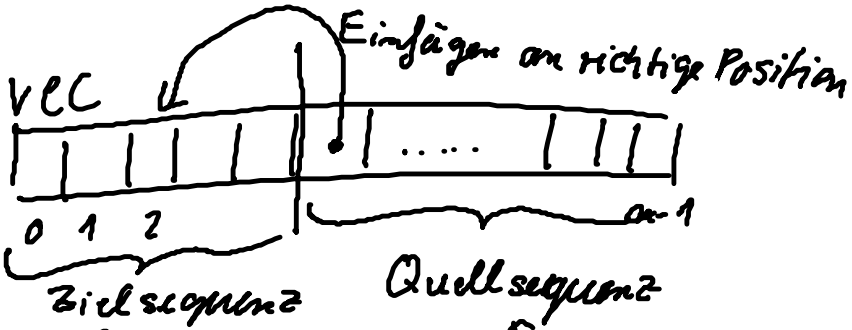
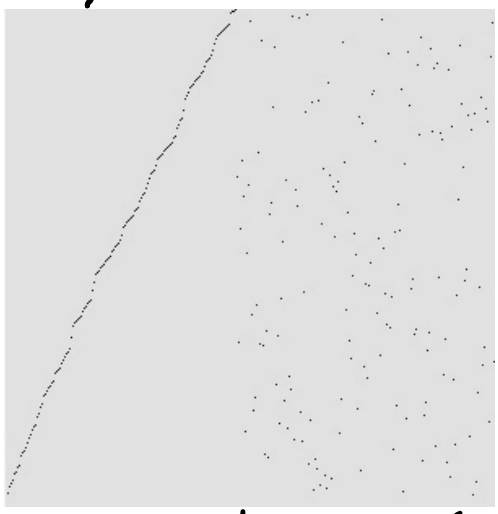


Insertion Sort

„Karten auf der Hand"
sortiert

„Stapel auf Tisch"
unsortiert

Snapshot



Zielsequenz Quellsequenz

Analyse:

$C(n)$

↑

Vergleiche

Suchen der Einfügeposition möglich über

A) sequentielle Suche von vorn nach hinten

B) binäre Suche (möglich, da Zielsequenz bereits vorsortiert)

zu A):

in Phase 1:	1 Vergleich	$O(n^2)$
in Phase 2:	2 Vergleiche	
⋮		
in Phase n-1:	n-1 Vergleiche	

Beispiele zeigen: $C(n) \in \Theta(n^2)$

zu B): Einfügen in Phase i im Zielsegment der Länge i erfordert $\leq \lfloor \log i \rfloor + 1$ Vergleiche (siehe VL über binäre Suche)

$$C(n) \leq \sum_{i=1}^{n-1} (\lfloor \log i \rfloor + 1)$$

$$\leq \sum_{i=1}^{n-1} (\lfloor \log (n-1) \rfloor + 1)$$

$$\leq (n-1)(\log(n-1)+1) \quad \boxed{\in O(n \log n)}$$

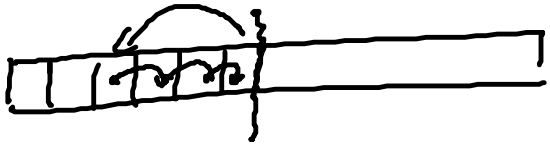
Frage: Bessere obere Schranke durch feinere Abschätzung?

Antwort später: nein, jedes vergleichsbezogene Sortierverfahren benötigt $\Omega(n \log n)$ Vergleiche.

Analyse von $A(n)$: ← # Zuweisungen

in beiden Varianten A und B muss im Zielarray alles nach Einfügeposition verschoben werden.

$\Rightarrow$ in Phase i im Worst Case $i+1$ Zuweisungen.



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

$$\in O(n^2)$$

Im Worst Case benötigt man so viele Zuweisungen, d.h.

$$A(n) \in \Theta(n^2)$$

Alternative: Verwende statt Array eine Liste:

$$\Rightarrow A(n) \in \mathcal{O}(n)$$

ABER: Binäre Suche in Liste nicht möglich, da kein direkter Zugriff auf Element an Position k.

$$\Rightarrow C(n) \in \Theta(n^2)$$ bei Verwendung einer Liste.

Alle bislang betrachteten Sortieralgorithmen haben Worst Case Laufzeit $\Omega(n^2)$.

Jetzt: 3 intelligentere Methoden

4) Merge Sort
5) Quick Sort
6) Heap Sort

} basieren auf Rekursion und „nicht-lokalen" Vergleichen

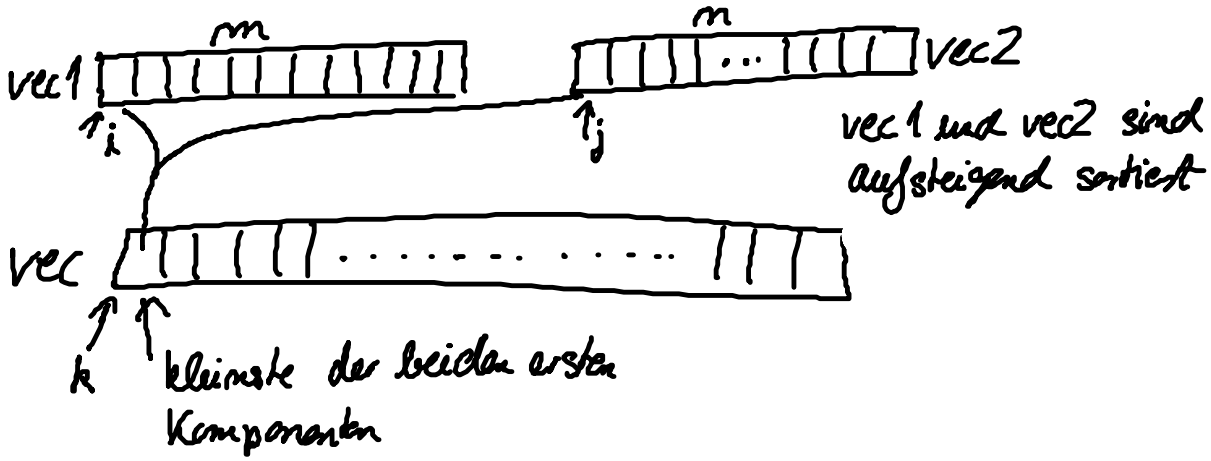
4) Merge Sort

"merge" bedeutet „verschmelzen" oder „mischen"

Idee:
- Teile zu sortierende Array in zwei möglichst gleich große Teile auf
- Sortiere rekursiv die beiden Teil-Arrays
- Verschmelze die sortierten Teil-Arrays zu einem sortierten Array.

Wie verschmelzen?

Annahme: Gesamt-Array besteht aus zwei vorsortierten Teil-Arrays:



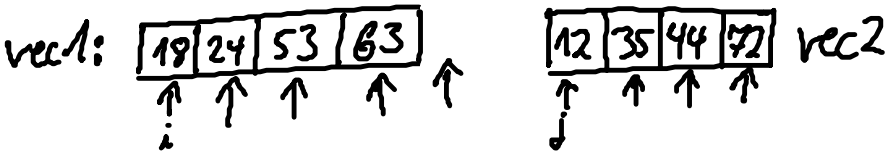
Genauere Beschreibung dieser merge-Operation:

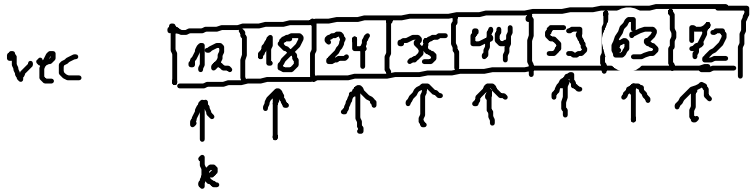
Dazu durchlaufen wir `vec1` und `vec2` von links nach rechts mit zwei Indexzeigern `i` und `j` wie folgt:

1. Initialisierung: `i = 0; j = 0; k= 0;` (k = i+j)
2. Wiederhole Schritt 3 bis `i = m` oder `j = n`.
3. Falls `vec1[i].key < vec2[j].key`, so kopiere `vec1[i]` an die Position `k` von `vec` und erhöhe `i` und `k` um 1.

   Andernfalls kopiere `vec2[j]` an die Position `k` von `vec` und erhöhe `j` und `k` um 1.
4. Ist `i = m` und `j < n`, so übertrage die restlichen Komponenten von `vec2` nach `vec`.
5. Ist `j = n` und `i < m`, so übertrage die restlichen Komponenten von `vec1` nach `vec`.

Beispiel:





Es sei $C(n,m) =$ # Vergleiche für merge-Operation im Worst Case

Dann ist $\boxed{C(n,m) = n+m-1}$

denn: Für jeden Eintrag im vec (außer für den letzten) muss im Worst Case ein Vergleich durchgeführt werden. Der letzte Eintrag wird immer ohne Vergleich übertragen, möglicherweise gilt das bereits für frühere Einträge (jedoch nicht im Worst Case!)

Idee für Merge Sort:

$n = 1 \Rightarrow$ nichts zu tun

$n > 1$:
- Teile das Array in zwei gleich große Hälften auf (Größenunterschied ≤ 1)
- Wende Merge Sort rekursiv auf beide Hälften an ($\rightarrow$ beide Hälften aufsteigend sortiert)
- wende die merge-Operation an.

Analysieren jetzt $C(n) :=$ # Vergleiche von Merge Sort im Worst Case:

$$C(2n) = 2 \cdot C(n) + 2n - 1 \quad \text{für } n \geq 2$$
$$C(2) = 1$$

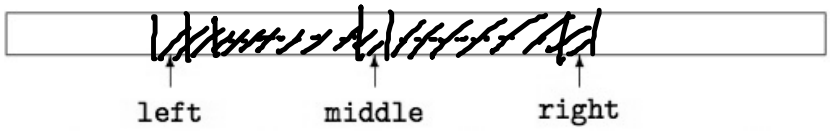
Rekursionsformel für # Vergleiche.

$A(m,n)$ = # Zuweisungen bei merge-Operation
$= m+n$

Für die Rekursion benötigen wir das folgende Verhalten:

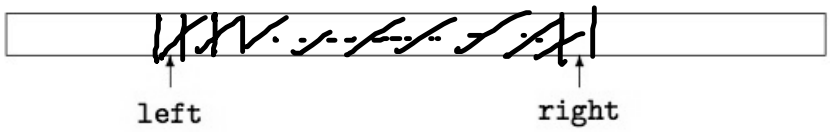
Wir verlangen also folgendes Input/Output Verhalten:

Input: vec



mit den sortierten Bereichen `vec[left]...vec[middle]` und `vec[middle+1]...vec[right]`.

Output: vec



mit dem aufsteigend sortierten Bereich `vec[left]...vec[right]`.

Umsetzung in Java:

```
private static void merge(Item[] vec, int left, int middle, int right) {
        int i, j;
        int m = middle - left + 1; // length of first array region
        int n = right - middle;    // length of second array region
```

```
// make copies of array regions to be merged
// (only the references to the items)
Item[] copy1 = new Item[m];
Item[] copy2 = new Item[n];
for (i = 0; i < m; i++) copy1[i] = vec[left + i];
for (j = 0; j < n; j++) copy2[j] = vec[middle + 1 + j];
```

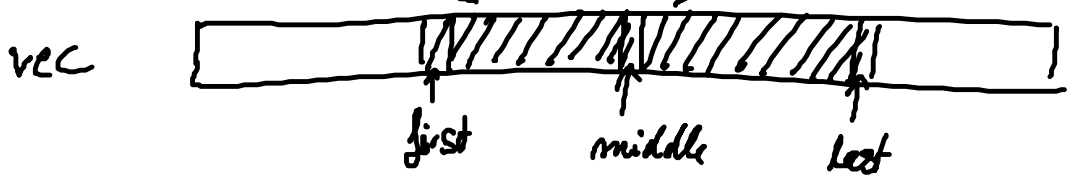

```
i = 0; j = 0;
// merge copy1 and copy2 into  vec[left...right]
while (i < m && j < n) {
        if (copy1[i].key < copy2[j].key) {
                vec[left+i+j] = copy1[i];
                i++;
        } else {
                vec[left+i+j] = copy2[j];
                j++;
        }//endif
}//endwhile
if (j == n) { // second array region is completely handled,
              // so copy rest of first region
        while (i < m) {
                vec[left+i+j] = copy1[i];
                i++;
        }
}
// if (i == m) do nothing,
// rest of second region is already in place
```

Diese Methode merge wird in der Implementation des Alg. MergeSort verwendet:

```
public static void mergeSort(Item vec[])
                        throws NullPointerException {
        if (vec == null) throw new NullPointerException();
        mergeSort(vec, 0, vec.length - 1);
}
```

↑ Aufruf einer weiteren Methode, die dafür ausgelegt ist ein Teilarray des Bezugs-Arrays vec zu sortieren, das durch die linke und rechte Grenze spezifiziert wird.

```
private static void mergeSort(Item[] vec, int first, int last) {
        if (first == last) return;
        // devide vec into 2 equal parts
        int middle = (first + last) / 2;
        mergeSort(vec, first, middle);   // sort the first part
        mergeSort(vec, middle+1, last);  // sort the second part
        merge(vec, first, middle, last); // merge the 2 sorted parts
}
```



Programmierersicht: delegiere Sortierung der kleineren Teilarrays durch rekursive Aufrufe.

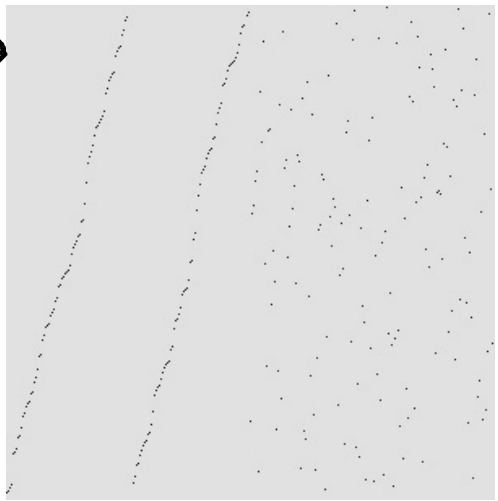
Ablauf der Rekursion auf dem Laufzeitstack ≙ „Ausführungssicht"

mergeSort(a,0,7)	63	24	12	53	72	18	44	35
• mergeSort(a,0,3)	63	24	12	53	72	18	44	35
— mergeSort(a,0,1)	63	24	12	53	72	18	44	35
• mergeSort(a,0,0)	63	24	12	53	72	18	44	35
• mergeSort(a,1,1)	63	24	12	53	72	18	44	35
• merge(a,0,0,1)	24	63	12	53	72	18	44	35
— mergeSort(a,2,3)	24	63	12	53	72	18	44	35
• mergeSort(a,2,2)	24	63	12	53	72	18	44	35
• mergeSort(a,3,3)	24	63	12	53	72	18	44	35
• merge(a,2,2,3)	24	63	12	53	72	18	44	35
— merge(a,0,1,3)	12	24	53	63	72	18	44	35
• mergeSort(a,4,7)	12	24	53	63	72	18	44	35
mergeSort(a,4,5)	12	24	53	63	72	18	44	35
mergeSort(a,4,4)	12	24	53	63	72	18	44	35
mergeSort(a,5,5)	12	24	53	63	72	18	44	35
merge(a,4,4,5)	12	24	53	63	18	72	44	35
mergeSort(a,6,7)	12	24	53	63	18	72	44	35
mergeSort(a,6,6)	12	24	53	63	18	72	44	35
mergeSort(a,7,7)	12	24	53	63	18	72	44	35
merge(a,6,6,7)	12	24	53	63	18	72	35	44
merge(a,4,5,7)	12	24	53	63	18	35	44	72
merge(a,0,3,7)	12	18	24	35	44	53	63	72

← Input

□ ≙ sortierte Teilarrays

Snapshot:



Analyse von MergeSort:

Vergleiche: $C(2) = 1 \qquad (C(1) = 0)$

$$C(2n) = \underbrace{2 \cdot C(n)}_{\text{sortiere Teilarrays durch Rekursion}} + \underbrace{2n-1}_{\text{Vergleiche beim Verschmelzen}}$$

Lösen der Rekursionsgleichung:

Um einen Ansatz zu finden, nehmen wir zunächst an, dass $n = 2^q$ für $q \in \mathbb{N}$ ist.

Wiederholte Anwendung der Rekursionsformel:

$$\begin{aligned}
C(n) &= 2 \cdot C\left(\tfrac{n}{2}\right) + 2 \cdot \tfrac{n}{2} - 1 \\
&= 2 \cdot C\left(\tfrac{n}{2}\right) + n - 1 && \text{1-mal angewandt} \\
&= 2 \cdot \left(2 \cdot C\left(\tfrac{n}{4}\right) + 2 \cdot \tfrac{n}{4} - 1\right) + n - 1 \\
&= 4 \cdot C\left(\tfrac{n}{4}\right) + 2n - 3 && \text{2-mal angewandt} \\
&= 4 \cdot \left(2 \cdot C\left(\tfrac{n}{8}\right) + 2 \cdot \tfrac{n}{8} - 1\right) + 2n - 3 \\
&= 8 \cdot C\left(\tfrac{n}{8}\right) + 3n - 7 && \text{3-mal angewandt} \\
&\;\;\vdots \\
&\overset{!}{=} 2^{q-1} \cdot C(2) + (q-1)n - (2^{q-1} - 1) && \text{insgesamt } q-1 \text{ mal angew.} \\
&\overset{!}{=} (q-1) \cdot 2^q + 1
\end{aligned}$$

Ansatz zur Lösung der Rekursionsgleichung:

$$\boxed{C(2^q) = (q-1) \cdot 2^q + 1}$$

Verifikation dieses Ansatzes durch vollst. Ind. für allg. q:

$q = 1 \Rightarrow C(2) = 1 = (1-1)\cdot 2^1 + 1$ ✓

Ind.-schluss auf $q+1$:

$$C(2^{q+1}) = C(2\cdot 2^q) \overset{\text{Rek.Bd.}}{=} 2\cdot C(2^q) + 2\cdot 2^q - 1$$

$$\overset{\text{I.V.}}{=} 2\cdot\left((q-1)\cdot 2^q + 1\right) + 2\cdot 2^q - 1$$

$$= (q-1)\cdot 2^{q+1} + 2 + 2^{q+1} - 1$$

$$= q\cdot 2^{q+1} + 1$$

∎