Karlsruher Institut für Technologie

# Algorithmen I - Tutorium 5

Sebastian Schmidt – *isibboi@gmail.com*

Arbeitsgruppe Kryptographie und Sicherheit

# Sortieren

- Eines der verbreitetsten Probleme der Informatik
- In jeder Sprache in der Standardbibliothek vorhanden
- Wichtig für schnelle Indexdatenstrukturen jeder Art
- Quicksort beliebtes Thema von Bewerbungsgesprächen
- Easy to learn - Hard to master

# Sortieren

- Eines der verbreitetsten Probleme der Informatik
- In jeder Sprache in der Standardbibliothek vorhanden
- Wichtig für schnelle Indexdatenstrukturen jeder Art
- Quicksort beliebtes Thema von Bewerbungsgesprächen
- Easy to learn - Hard to master

# Sortieren

- Eines der verbreitetsten Probleme der Informatik
- In jeder Sprache in der Standardbibliothek vorhanden
- Wichtig für schnelle Indexdatenstrukturen jeder Art
- Quicksort beliebtes Thema von Bewerbungsgesprächen
- Easy to learn - Hard to master

# Sortieren

- Eines der verbreitetsten Probleme der Informatik
- In jeder Sprache in der Standardbibliothek vorhanden
- Wichtig für schnelle Indexdatenstrukturen jeder Art
- Quicksort beliebtes Thema von Bewerbungsgesprächen
- Easy to learn - Hard to master

# Sortieren

- Eines der verbreitetsten Probleme der Informatik
- In jeder Sprache in der Standardbibliothek vorhanden
- Wichtig für schnelle Indexdatenstrukturen jeder Art
- Quicksort beliebtes Thema von Bewerbungsgesprächen
- Easy to learn - Hard to master

# Das Problem

Gegeben eine Folge von $n$ Elementen $\alpha_1, ..., \alpha_n$.
Finde eine Permutation $\sigma$, sodass gilt:
$\forall i \in \{1, \ldots, n-1\} : \alpha_{\sigma(i)} \leq \alpha_{\sigma(i+1)}$

# Die Lösungen

- Binary Tree Sort (höhen-balanciert)
- Binary Tree Sort
- Bogosort
- Bubblesort
- Bucketsort
- Combsort
- Countingsort
- Flashsort
- Gnomesort
- Heapsort
- Insertionsort
- Introsort
- Merge Insertion
- Mergesort
- Natural Mergesort
- Quicksort
- Radixsort (LSB)
- Radixsort (MSB)
- Selectionsort
- Shakersort (Cocktailsort)
- Shellsort
- Slowsort
- Smoothsort
- Stoogesort
- Swap-Sort
- Timsort

$\Rightarrow$ Es machen sich sehr viele Leute sehr viele Gedanken darüber, wie man schnell sortiert.

# Die Lösungen

- Binary Tree Sort (höhen-balanciert)
- Binary Tree Sort
- Bogosort
- Bubblesort
- Bucketsort
- Combsort
- Countingsort
- Flashsort
- Gnomesort
- Heapsort
- Insertionsort
- Introsort
- Merge Insertion
- Mergesort
- Natural Mergesort
- Quicksort
- Radixsort (LSB)
- Radixsort (MSB)
- Selectionsort
- Shakersort (Cocktailsort)
- Shellsort
- Slowsort
- Smoothsort
- Stoogesort
- Swap-Sort
- Timsort

$\Rightarrow$ Es machen sich sehr viele Leute sehr viele Gedanken darüber, wie man schnell sortiert.

# Die Lösungen

- Binary Tree Sort (höhen-balanciert)
- Binary Tree Sort
- Bogosort
- Bubblesort
- Bucketsort
- Combsort
- Countingsort
- Flashsort
- Gnomesort
- Heapsort
- Insertionsort
- Introsort
- Merge Insertion
- Mergesort
- Natural Mergesort
- Quicksort
- Radixsort (LSB)
- Radixsort (MSB)
- Selectionsort
- Shakersort (Cocktailsort)
- Shellsort
- Slowsort
- Smoothsort
- Stoogesort
- Swap-Sort
- Timsort

$\Rightarrow$ Es machen sich sehr viele Leute sehr viele Gedanken darüber, wie man schnell sortiert.

# Ein Sortieralgorithmus

Wie heißt der Algorithmus?

```
function SORT(A : Number[1, ..., n])
    for i from 1 to n − 1 do
        k ← i
        for j from i + 1 to n do
            if A[k] > A[j] then
                k ← j
            end if
        end for
        SWAP(A, i, k)
    end for
end function
```

# Ein Sortieralgorithmus

Wie heißt der Algorithmus?

```
function SelectionSort(A : Number[1, ..., n])
    for sorted from 1 to n − 1 do
        minIndex ← sorted
        for index from sorted + 1 to n do
            if A[minIndex] > A[index] then
                minIndex ← index
            end if
        end for
        Swap(A, sorted, minIndex)
    end for
end function
```

# Ein Sortieralgorithmus

Welche Laufzeit hat der Algorithmus im Worst-Case im $O$-Kalkül? Was ist eine Worst-Case-Eingabe?

---

```
function SelectionSort(A : Number[1, ..., n])
    for sorted from 1 to n − 1 do
        minIndex ← sorted
        for index from sorted + 1 to n do
            if A[minIndex] > A[index] then
                minIndex ← index
            end if
        end for
        Swap(A, sorted, minIndex)
    end for
end function
```

---

# Ein Sortieralgorithmus

Welche Laufzeit hat der Algorithmus im Best-Case im $O$-Kalkül? Was ist eine Best-Case-Eingabe?

---

```
function SelectionSort(A : Number[1, ..., n])
    for sorted from 1 to n − 1 do
        minIndex ← sorted
        for index from sorted + 1 to n do
            if A[minIndex] > A[index] then
                minIndex ← index
            end if
        end for
        Swap(A, sorted, minIndex)
    end for
end function
```

---

# Ein Sortieralgorithmus

Welche Laufzeit hat der Algorithmus im Average-Case im $O$-Kalkül?

---

```
function SelectionSort(A : Number[1, ..., n])
    for sorted from 1 to n − 1 do
        minIndex ← sorted
        for index from sorted + 1 to n do
            if A[minIndex] > A[index] then
                minIndex ← index
            end if
        end for
        Swap(A, sorted, minIndex)
    end for
end function
```

---

# Ein Sortieralgorithmus

Wie ist der zusätzliche Speicherbedarf im $O$-Kalkül?

```
function SelectionSort(A : Number[1, ..., n])
    for sorted from 1 to n − 1 do
        minIndex ← sorted
        for index from sorted + 1 to n do
            if A[minIndex] > A[index] then
                minIndex ← index
            end if
        end for
        Swap(A, sorted, minIndex)
    end for
end function
```

# Ein Sortieralgorithmus

Ist der Sortieralgorithmus stabil?

```
function SelectionSort(A : Number[1, ..., n])
    for sorted from 1 to n − 1 do
        minIndex ← sorted
        for index from sorted + 1 to n do
            if A[minIndex] > A[index] then
                minIndex ← index
            end if
        end for
        Swap(A, sorted, minIndex)
    end for
end function
```

# Noch ein Sortieralgorithmus

Wie heißt der Algorithmus?

```
function SORT(A : Number[1, ..., n])
    A[0] ← −∞
    for i from 2 to n do
        j ← i − 1
        while A[j] > A[j + 1] do
            SWAP(A, j, j + 1)
            j − −
        end while
    end for
end function
```

# Noch ein Sortieralgorithmus

Wie heißt der Algorithmus?

```
function InsertionSort(A : Number[1, ..., n])
    A[0] ← −∞
    for sorted from 2 to n do
        swapIndex ← sorted − 1
        while A[swapIndex] > A[swapIndex + 1] do
            Swap(A, swapIndex, swapIndex + 1)
            swapIndex − −
        end while
    end for
end function
```

# Noch ein Sortieralgorithmus

Welche Laufzeit hat der Algorithmus im Worst-Case im $O$-Kalkül? Was ist eine Worst-Case-Eingabe?

```
function InsertionSort(A : Number[1, ..., n])
    A[0] ← −∞
    for sorted from 2 to n do
        swapIndex ← sorted − 1
        while A[swapIndex] > A[swapIndex + 1] do
            Swap(A, swapIndex, swapIndex + 1)
            swapIndex − −
        end while
    end for
end function
```

# Noch ein Sortieralgorithmus

Welche Laufzeit hat der Algorithmus im Best-Case im $O$-Kalkül? Was ist eine Best-Case-Eingabe?

---

**function** INSERTIONSORT($A$ : *Number*$[1, \ldots, n]$)
    $A[0] \leftarrow -\infty$
    **for** *sorted* from 2 to $n$ **do**
        $swapIndex \leftarrow sorted - 1$
        **while** $A[swapIndex] > A[swapIndex + 1]$ **do**
            SWAP($A$, $swapIndex$, $swapIndex + 1$)
            $swapIndex - -$
        **end while**
    **end for**
**end function**

---

# Noch ein Sortieralgorithmus

Welche Laufzeit hat der Algorithmus im Average-Case im $O$-Kalkül?

```
function InsertionSort(A : Number[1, ..., n])
    A[0] ← −∞
    for sorted from 2 to n do
        swapIndex ← sorted − 1
        while A[swapIndex] > A[swapIndex + 1] do
            Swap(A, swapIndex, swapIndex + 1)
            swapIndex − −
        end while
    end for
end function
```

# Noch ein Sortieralgorithmus

Wie ist der zusätzliche Speicherbedarf im $O$-Kalkül?

```
function InsertionSort(A : Number[1, ..., n])
    A[0] ← −∞
    for sorted from 2 to n do
        swapIndex ← sorted − 1
        while A[swapIndex] > A[swapIndex + 1] do
            Swap(A, swapIndex, swapIndex + 1)
            swapIndex − −
        end while
    end for
end function
```

# Noch ein Sortieralgorithmus

Ist der Sortieralgorithmus stabil?

```
function InsertionSort(A : Number[1, ..., n])
    A[0] ← −∞
    for sorted from 2 to n do
        swapIndex ← sorted − 1
        while A[swapIndex] > A[swapIndex + 1] do
            Swap(A, swapIndex, swapIndex + 1)
            swapIndex − −
        end while
    end for
end function
```

# Schon wieder ein Sortieralgorithmus

Wie heißt der Algorithmus?

```
function SORT(A : Number[1, ..., n])
    if n = 1 then return A
    else
        (A_1, A_2) ← SPLITTOEQUALSIZE(A)
        (B_1, B_2) ← (SORT(A_1), SORT(A_2))
        B ← new Number[1, ..., n]
        i_1 ← i_2 ← 1
        for j from 1 to n do
            B[j] ← MIN(B_1[i_1], B_2[i_2])
            Increment i_{1/2} if it was chosen
        end for
    end if
    return B concat ∞
end function
```

# Schon wieder ein Sortieralgorithmus

Wie heißt der Algorithmus?

---

```
function MergeSort(A : Number[1, ..., n])
    if n = 1 then return A
    else
        (A_1, A_2) ← SplitToEqualSize(A)
        (B_1, B_2) ← (MergeSort(A_1), MergeSort(A_2))
        B ← new Number[1, ..., n]
        lowest_1 ← lowest_2 ← 1
        for index from 1 to n do
            B[index] ← Min(B_1[lowest_1], B_2[lowest_2])
            Increment lowest_1/2 if it was chosen
        end for
    end if
    return B concat ∞
end function
```

---

# Schon wieder ein Sortieralgorithmus

Gibt es einen Unterschied zwischen Worst-Case- und Best-Case-Laufzeit im $O$-Kalkül?

---

```
function MergeSort(A : Number[1, ..., n])
    if n = 1 then return A
    else
        (A_1, A_2) ← SplitToEqualSize(A)
        (B_1, B_2) ← (MergeSort(A_1), MergeSort(A_2))
        B ← new Number[1, ..., n]
        lowest_1 ← lowest_2 ← 1
        for index from 1 to n do
            B[index] ← Min(B_1[lowest_1], B_2[lowest_2])
            Increment lowest_1/2 if it was chosen
        end for
    end if
    return B concat ∞
end function
```

# Schon wieder ein Sortieralgorithmus

Welche Laufzeit hat der Algorithmus im Worst-Case im $O$-Kalkül?

---

```
function MergeSort(A : Number[1, ..., n])
    if n = 1 then return A
    else
        (A_1, A_2) ← SplitToEqualSize(A)
        (B_1, B_2) ← (MergeSort(A_1), MergeSort(A_2))
        B ← new Number[1, ..., n]
        lowest_1 ← lowest_2 ← 1
        for index from 1 to n do
            B[index] ← Min(B_1[lowest_1], B_2[lowest_2])
            Increment lowest_1/2 if it was chosen
        end for
    end if
    return B concat ∞
end function
```

---

# Schon wieder ein Sortieralgorithmus

Wie ist der zusätzliche Speicherbedarf im $O$-Kalkül?

---

**function** MERGESORT($A$ : *Number*$[1, \ldots, n]$)
  **if** $n = 1$ **then return** $A$
  **else**
    $(A_1, A_2) \leftarrow$ SPLITTOEQUALSIZE($A$)
    $(B_1, B_2) \leftarrow$ (MERGESORT($A_1$), MERGESORT($A_2$))
    $B \leftarrow$ **new** *Number*$[1, \ldots, n]$
    $lowest_1 \leftarrow lowest_2 \leftarrow 1$
    **for** *index* from 1 to $n$ **do**
      $B[index] \leftarrow$ MIN($B_1[lowest_1], B_2[lowest_2]$)
      Increment $lowest_{1/2}$ if it was chosen
    **end for**
  **end if**
  **return** $B$ concat $\infty$
**end function**

---

# Schon wieder ein Sortieralgorithmus

Bekommt man auch $O(n)$ zusätzlichen Speicher hin?

---

**function** MERGESORT($A$ : *Number*$[1, \ldots, n]$)
  **if** $n = 1$ **then return** $A$
  **else**
    $(A_1, A_2) \leftarrow$ SPLITTOEQUALSIZE($A$)
    $(B_1, B_2) \leftarrow$ (MERGESORT($A_1$), MERGESORT($A_2$))
    $B \leftarrow$ **new** *Number*$[1, \ldots, n]$
    $lowest_1 \leftarrow lowest_2 \leftarrow 1$
    **for** *index* from 1 to $n$ **do**
      $B[index] \leftarrow$ MIN($B_1[lowest_1], B_2[lowest_2]$)
      Increment $lowest_{1/2}$ if it was chosen
    **end for**
  **end if**
  **return** $B$ concat $\infty$
**end function**

---

# Schon wieder ein Sortieralgorithmus

Ist der Sortieralgorithmus stabil?

---

```
function MergeSort(A : Number[1, ..., n])
    if n = 1 then return A
    else
        (A_1, A_2) ← SplitToEqualSize(A)
        (B_1, B_2) ← (MergeSort(A_1), MergeSort(A_2))
        B ← new Number[1, ..., n]
        lowest_1 ← lowest_2 ← 1
        for index from 1 to n do
            B[index] ← Min(B_1[lowest_1], B_2[lowest_2])
            Increment lowest_1/2 if it was chosen
        end for
    end if
    return B concat ∞
end function
```

---