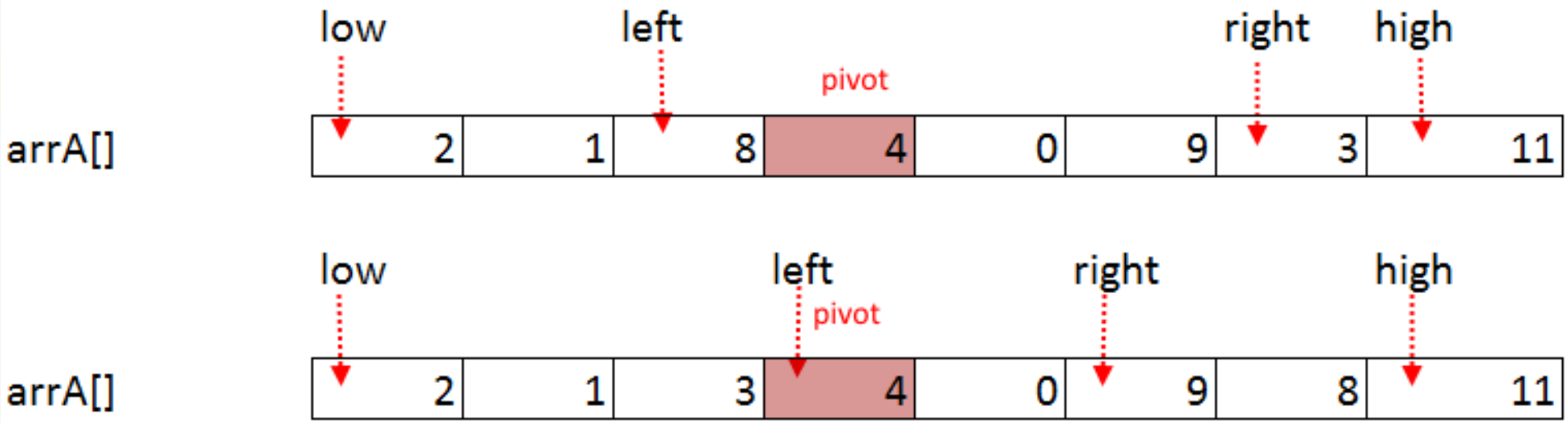
Karlsruher Institut für Technologie

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 σ, 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?

```
 1: function SORT(A : Number[1, ..., n])
 2:     for i from 1 to n − 1 do
 3:         k ← i
 4:         for j from i + 1 to n do
 5:             if A[k] > A[j] then
 6:                 k ← j
 7:             end if
 8:         end for
 9:         SWAP(A, i, k)
10:     end for
11: end function
```

Ein Sortieralgorithmus

Wie heißt der Algorithmus?

```
 1: function SelectionSort(A : Number[1, ..., n])
 2:     for sorted from 1 to n − 1 do
 3:         minIndex ← sorted
 4:         for index from sorted + 1 to n do
 5:             if A[minIndex] > A[index] then
 6:                 minIndex ← index
 7:             end if
 8:         end for
 9:         Swap(A, sorted, minIndex)
10:     end for
11: end function
```

Ein Sortieralgorithmus

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

```
 1: function SelectionSort(A : Number[1, ..., n])
 2:     for sorted from 1 to n − 1 do
 3:         minIndex ← sorted
 4:         for index from sorted + 1 to n do
 5:             if A[minIndex] > A[index] then
 6:                 minIndex ← index
 7:             end if
 8:         end for
 9:         Swap(A, sorted, minIndex)
10:     end for
11: end function
```

Ein Sortieralgorithmus

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

```
 1: function SelectionSort(A : Number[1, ..., n])
 2:     for sorted from 1 to n − 1 do
 3:         minIndex ← sorted
 4:         for index from sorted + 1 to n do
 5:             if A[minIndex] > A[index] then
 6:                 minIndex ← index
 7:             end if
 8:         end for
 9:         Swap(A, sorted, minIndex)
10:     end for
11: end function
```

Ein Sortieralgorithmus

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

```
 1: function SelectionSort(A : Number[1, ..., n])
 2:     for sorted from 1 to n − 1 do
 3:         minIndex ← sorted
 4:         for index from sorted + 1 to n do
 5:             if A[minIndex] > A[index] then
 6:                 minIndex ← index
 7:             end if
 8:         end for
 9:         Swap(A, sorted, minIndex)
10:     end for
11: end function
```

Ein Sortieralgorithmus

Wie ist der zusätzliche Speicherbedarf im O-Kalkül?

```
 1: function SelectionSort(A : Number[1, ..., n])
 2:     for sorted from 1 to n − 1 do
 3:         minIndex ← sorted
 4:         for index from sorted + 1 to n do
 5:             if A[minIndex] > A[index] then
 6:                 minIndex ← index
 7:             end if
 8:         end for
 9:         Swap(A, sorted, minIndex)
10:     end for
11: end function
```

Ein Sortieralgorithmus

Ist der Sortieralgorithmus stabil?

```
 1: function SelectionSort(A : Number[1, . . . , n])
 2:     for sorted from 1 to n − 1 do
 3:         minIndex ← sorted
 4:         for index from sorted + 1 to n do
 5:             if A[minIndex] > A[index] then
 6:                 minIndex ← index
 7:             end if
 8:         end for
 9:         Swap(A, sorted, minIndex)
10:     end for
11: end function
```

Noch ein Sortieralgorithmus

Wie heißt der Algorithmus?

```
 1: function SORT(A : Number[1, ..., n])
 2:     A[0] ← −∞
 3:     for i from 2 to n do
 4:         j ← i − 1
 5:         while A[j] > A[j + 1] do
 6:             SWAP(A, j, j + 1)
 7:             j − −
 8:         end while
 9:     end for
10: end function
```

Noch ein Sortieralgorithmus

Wie heißt der Algorithmus?

```
 1: function InsertionSort(A : Number[1, ..., n])
 2:     A[0] ← −∞
 3:     for sorted from 2 to n do
 4:         swapIndex ← sorted − 1
 5:         while A[swapIndex] > A[swapIndex + 1] do
 6:             Swap(A, swapIndex, swapIndex + 1)
 7:             swapIndex − −
 8:         end while
 9:     end for
10: end function
```

Noch ein Sortieralgorithmus

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

```
1:  function InsertionSort(A : Number[1, ..., n])
2:      A[0] ← −∞
3:      for sorted from 2 to n do
4:          swapIndex ← sorted − 1
5:          while A[swapIndex] > A[swapIndex + 1] do
6:              Swap(A, swapIndex, swapIndex + 1)
7:              swapIndex − −
8:          end while
9:      end for
10: end function
```

Noch ein Sortieralgorithmus

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

```
 1: function InsertionSort(A : Number[1, ..., n])
 2:     A[0] ← −∞
 3:     for sorted from 2 to n do
 4:         swapIndex ← sorted − 1
 5:         while A[swapIndex] > A[swapIndex + 1] do
 6:             Swap(A, swapIndex, swapIndex + 1)
 7:             swapIndex − −
 8:         end while
 9:     end for
10: end function
```

Noch ein Sortieralgorithmus

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

```
 1: function InsertionSort(A : Number[1, ..., n])
 2:     A[0] ← −∞
 3:     for sorted from 2 to n do
 4:         swapIndex ← sorted − 1
 5:         while A[swapIndex] > A[swapIndex + 1] do
 6:             Swap(A, swapIndex, swapIndex + 1)
 7:             swapIndex − −
 8:         end while
 9:     end for
10: end function
```

Noch ein Sortieralgorithmus

Wie ist der zusätzliche Speicherbedarf im O-Kalkül?

```
 1: function InsertionSort(A : Number[1, ..., n])
 2:     A[0] ← −∞
 3:     for sorted from 2 to n do
 4:         swapIndex ← sorted − 1
 5:         while A[swapIndex] > A[swapIndex + 1] do
 6:             Swap(A, swapIndex, swapIndex + 1)
 7:             swapIndex − −
 8:         end while
 9:     end for
10: end function
```

Noch ein Sortieralgorithmus

Ist der Sortieralgorithmus stabil?

```
 1: function InsertionSort(A : Number[1, ..., n])
 2:     A[0] ← −∞
 3:     for sorted from 2 to n do
 4:         swapIndex ← sorted − 1
 5:         while A[swapIndex] > A[swapIndex + 1] do
 6:             Swap(A, swapIndex, swapIndex + 1)
 7:             swapIndex − −
 8:         end while
 9:     end for
10: end function
```

Schon wieder ein Sortieralgorithmus

Wie heißt der Algorithmus?

```
 1: function SORT(A : Number[1, ..., n])
 2:     if n = 1 then return A
 3:     else
 4:         (A_1, A_2) ← SPLITTOEQUALSIZE(A)
 5:         (B_1, B_2) ← (SORT(A_1), SORT(A_2))
 6:         B ← new Number[1, ..., n]
 7:         i_1 ← i_2 ← 1
 8:         for j from 1 to n do
 9:             B[j] ← MIN(B_1[i_1], B_2[i_2])
10:             Increment i_{1/2} if it was chosen
11:         end for
12:     end if
13:     return B concat ∞
14: end function
```

Schon wieder ein Sortieralgorithmus

Wie heißt der Algorithmus?

```
 1: function MergeSort(A : Number[1, ..., n])
 2:     if n = 1 then return A
 3:     else
 4:         (A_1, A_2) ← SplitToEqualSize(A)
 5:         (B_1, B_2) ← (MergeSort(A_1), MergeSort(A_2))
 6:         B ← new Number[1, ..., n]
 7:         lowest_1 ← lowest_2 ← 1
 8:         for index from 1 to n do
 9:             B[index] ← Min(B_1[lowest_1], B_2[lowest_2])
10:             Increment lowest_{1/2} if it was chosen
11:         end for
12:     end if
13:     return B concat ∞
14: end function
```

Schon wieder ein Sortieralgorithmus

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

```
 1: function MergeSort(A : Number[1, ..., n])
 2:     if n = 1 then return A
 3:     else
 4:         (A_1, A_2) ← SplitToEqualSize(A)
 5:         (B_1, B_2) ← (MergeSort(A_1), MergeSort(A_2))
 6:         B ← new Number[1, ..., n]
 7:         lowest_1 ← lowest_2 ← 1
 8:         for index from 1 to n do
 9:             B[index] ← Min(B_1[lowest_1], B_2[lowest_2])
10:             Increment lowest_{1/2} if it was chosen
11:         end for
12:     end if
13:     return B concat ∞
14: end function
```

Schon wieder ein Sortieralgorithmus

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

```
 1: function MergeSort(A : Number[1, ..., n])
 2:     if n = 1 then return A
 3:     else
 4:         (A_1, A_2) ← SplitToEqualSize(A)
 5:         (B_1, B_2) ← (MergeSort(A_1), MergeSort(A_2))
 6:         B ← new Number[1, ..., n]
 7:         lowest_1 ← lowest_2 ← 1
 8:         for index from 1 to n do
 9:             B[index] ← Min(B_1[lowest_1], B_2[lowest_2])
10:             Increment lowest_{1/2} if it was chosen
11:         end for
12:     end if
13:     return B concat ∞
14: end function
```

Schon wieder ein Sortieralgorithmus

Wie ist der zusätzliche Speicherbedarf im O-Kalkül?

```
 1: function MergeSort(A : Number[1, ..., n])
 2:     if n = 1 then return A
 3:     else
 4:         (A_1, A_2) ← SplitToEqualSize(A)
 5:         (B_1, B_2) ← (MergeSort(A_1), MergeSort(A_2))
 6:         B ← new Number[1, ..., n]
 7:         lowest_1 ← lowest_2 ← 1
 8:         for index from 1 to n do
 9:             B[index] ← Min(B_1[lowest_1], B_2[lowest_2])
10:             Increment lowest_{1/2} if it was chosen
11:         end for
12:     end if
13:     return B concat ∞
14: end function
```

Schon wieder ein Sortieralgorithmus

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

```
 1: function MergeSort(A : Number[1, ..., n])
 2:     if n = 1 then return A
 3:     else
 4:         (A_1, A_2) ← SplitToEqualSize(A)
 5:         (B_1, B_2) ← (MergeSort(A_1), MergeSort(A_2))
 6:         B ← new Number[1, ..., n]
 7:         lowest_1 ← lowest_2 ← 1
 8:         for index from 1 to n do
 9:             B[index] ← Min(B_1[lowest_1], B_2[lowest_2])
10:             Increment lowest_{1/2} if it was chosen
11:         end for
12:     end if
13:     return B concat ∞
14: end function
```

Schon wieder ein Sortieralgorithmus

Ist der Sortieralgorithmus stabil?

```
 1: function MergeSort(A : Number[1, ..., n])
 2:     if n = 1 then return A
 3:     else
 4:         (A_1, A_2) ← SplitToEqualSize(A)
 5:         (B_1, B_2) ← (MergeSort(A_1), MergeSort(A_2))
 6:         B ← new Number[1, ..., n]
 7:         lowest_1 ← lowest_2 ← 1
 8:         for index from 1 to n do
 9:             B[index] ← Min(B_1[lowest_1], B_2[lowest_2])
10:             Increment lowest_{1/2} if it was chosen
11:         end for
12:     end if
13:     return B concat ∞
14: end function
```

Schon wieder ein Sortieralgorithmus

Wie macht man jeden nicht-stabilen Sortieralgorithmus stabil?

```
 1: function MergeSort(A : Number[1, ..., n])
 2:     if n = 1 then return A
 3:     else
 4:         (A_1, A_2) ← SplitToEqualSize(A)
 5:         (B_1, B_2) ← (MergeSort(A_1), MergeSort(A_2))
 6:         B ← new Number[1, ..., n]
 7:         lowest_1 ← lowest_2 ← 1
 8:         for index from 1 to n do
 9:             B[index] ← Min(B_1[lowest_1], B_2[lowest_2])
10:             Increment lowest_{1/2} if it was chosen
11:         end for
12:     end if
13:     return B concat ∞
14: end function
```

INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): // COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```