

Algorithms and Data Structures

- Lesson 2 -

Michael Schwarzkopf

<https://www.uni-weimar.de/de/medien/professuren/medieninformatik/grafische-datenverarbeitung>

Bauhaus-University Weimar

May 16, 2018

Overview

...of things you should definitely know about if you want a very good grade

- Complexity
- Sorting for Beginners (avg $O(n^2)$)
 - Selection
 - Insertion
 - Bubble
- Sorting for Pros (faster than $O(n^2)$)
 - Shell
 - Quick
 - Heap
- Hashing
- Searching
- ...

Complexity

What do expressions like

$$f(n) = O(n^2)$$

try to tell us?

- n : Length of an Input
- $f(n)$: total number of Steps depending on n
- $O(\textit{SomeFunktion}(n))$: An upper bound

Some Pseudo – Code Example

```
def smallestElement(n)
{
    int a = n[0]
    int location = 0

    for i = 0,1, ... , |n|
    {
        if n[i] < a
        {
            a = n[i]
            location = i
        }
    }
    return location;
}
```

Finding the location
of smallest Element in
Array

Some Pseudo – Code Example

```
def smallestElement(n)
{
    int a = n[0]
    int location = 0 } 2 Steps
```

```
    for i = 0,1, ... , |n|
    {
        if n[i] < a } 1 Step
        {
            a = n[i]
            location = i } 2 Steps } n Steps
        }
    }
    return location; } 1 Step
}
```

$3 + 3n$ Steps
in total.

Some Pseudo – Code Example

- For quite big n , I don't care about constants
 - 10000 or 10003 steps doesn't matter
- n or $3n$ also makes no difference
 - Lets just buy a machine 3 times as fast

$3 + 3n$ Steps
in total.

$$\underline{\rightarrow 3 + 3n = O(n)}$$

Useful techniques

- Professor gave you formal definition of Big O notation, but it's complicated
- Easier:
 - Just remove all the constants. (exception: c^n)
 - Kick out functions which grow slower than others (Search for the biggest exponent)
- $2n + 6n^2 + 5 \rightarrow n + n^2 = O(n^2)$

Try it!

Remove constants, only keep most growing functions:

$$500 + n + 0.05 * n^2$$

$$2n * 4n * 0.5n + n^2$$

$$\log_2(n) + \log_{10}(2n)$$

$$n^{100} * 2^n$$

$$2n * 2 * \log_2(n)$$

Try it!

Remove constants, only keep most growing functions:

$$500 + n + 0.05 * n^2 = O(n^2)$$

$$2n * 4n * 0.5n + n^2 = O(n^3)$$

$$\log_2(n) + \log_{10}(2n) = O(\log(n))$$

$$n^{100} * 2^n = O(2^n)$$

$$2n * 2 * \log_2(n) = O(n \log(n))$$

Sorting

- Almost guaranteed: one Question in Exam like:

„Explain Sorting Algorithm XYsort and discuss its Complexity“

- 6 Algorithms each with
 - Best case
 - average case → Where complexity can differ, but does not have to.
 - worst case

Selection Sort

- **Select** smallest element in unsorted set and put it at the end of the sorted one
 - Finding smallest Element, as we know: $O(n)$
 - Set shrinks each Iteration by one, so:
$$(n - 1) + (n - 2) + \dots + 2 + 1 = n*(n-1)/2$$
$$= n^2/2 - n/2 = O(n^2)$$
- Best = Average = Worst case = $O(n^2)$

Insertion Sort

- Take an Element in unsorted set and **Insert** at the right place in the sorted set
 - Taking the first Element: $O(1)$
 - Finding right place to insert: depends.
 - At the beginning: $O(1)$
 - At the end: $O(n)$ (Like W.C. in Selection Sort)
 - In the middle (average): $n/4 = O(n)$

Insertion Sort

- As we have to do this for every element
→ n times:
 - Best case: $O(n)$
 - Average case: $O(n^2)$
 - Worst case: $O(n^2)$
- That makes it slightly more efficient than Selection Sort

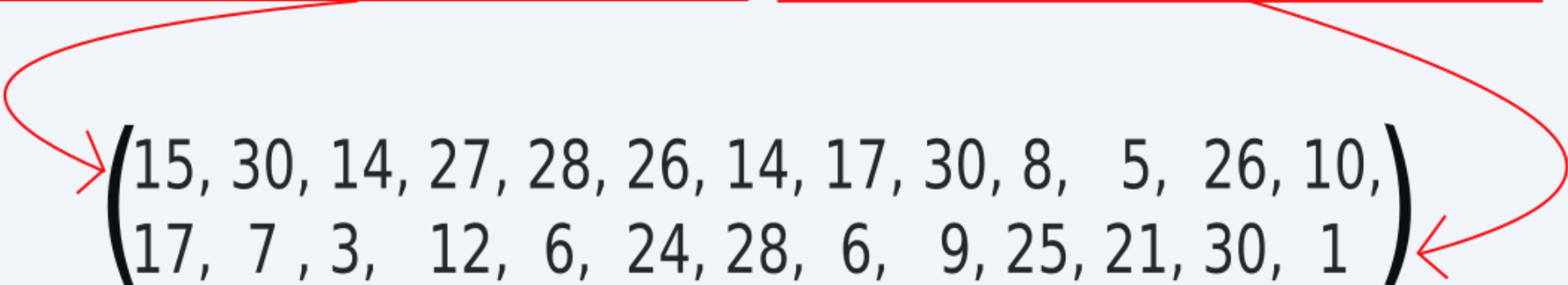
Bubble Sort

- From $i = 0 \dots n - 1$, compare Element i with $i + 1$ and swap, if Element _{i} is Bigger than Element _{$i+1$}
 - Bigger values raise the Array like a **Bubble!**
 - Stops when Array is sorted
- Best Case: only check if sorted $\rightarrow O(n)$
- Avg. Case: $(n + (n-1) + \dots + 2 + 1) - c = O(n^2)$
 - c ops. you don't do if sorted soon. Usually small
- Worst Case: $(n + (n-1) + \dots + 2 + 1) = O(n^2)$

Shell Sort


- Defined Sequence: 1, 4, 13, 40, 121, ... ($\rightarrow s_i = 3s_{i-1} + 1$)
- String of Length $n = 26$. \rightarrow 13 $\leq n < 40$
 - Organize String in Matrix with 13 columns:

15, 30, 14, 27, 28, 26, 14, 17, 30, 8, 5, 26, 10, 17, 7, 3, 12, 6, 24, 28, 6, 9, 25, 21, 30, 1



$\begin{pmatrix} 15, 30, 14, 27, 28, 26, 14, 17, 30, 8, 5, 26, 10, \\ 17, 7, 3, 12, 6, 24, 28, 6, 9, 25, 21, 30, 1 \end{pmatrix}$

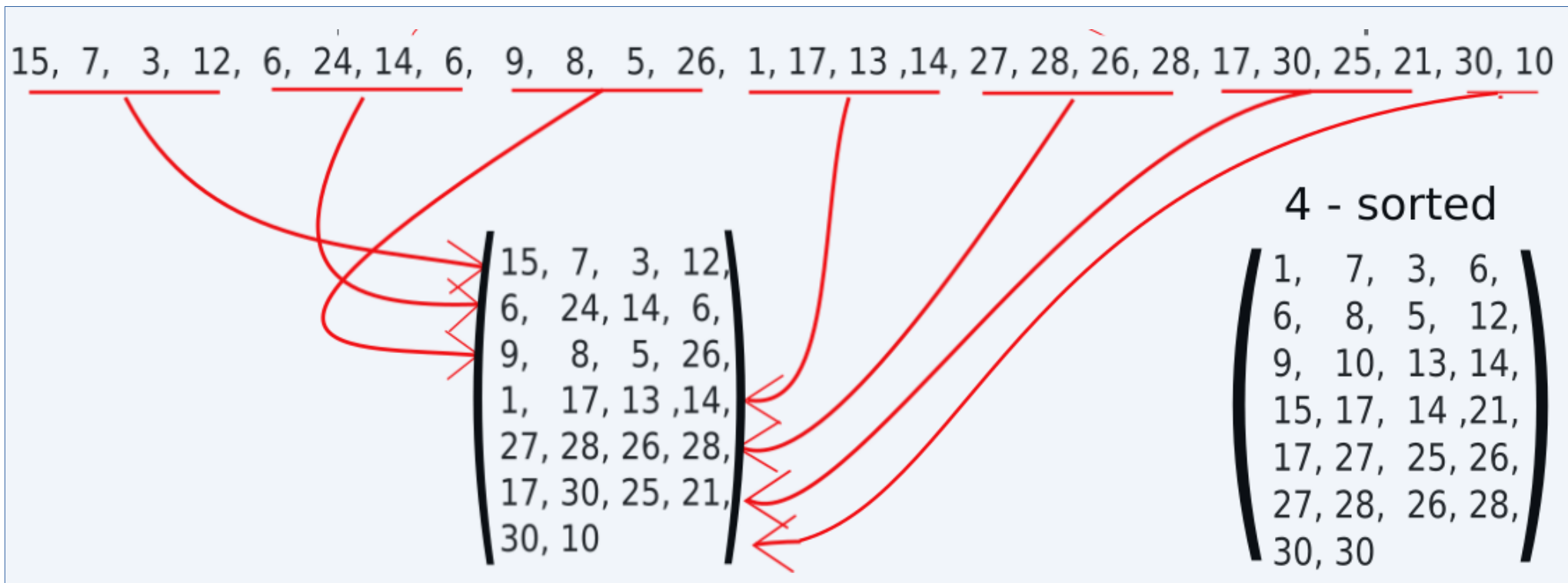
- Sort Columns



$\begin{pmatrix} 15, 7, 3, 12, 6, 24, 14, 6, 9, 8, 5, 26, 1, \\ 17, 13, 14, 27, 28, 26, 28, 17, 30, 25, 21, 30, 10 \end{pmatrix}$

Shell Sort

- Defined Sequence: 1, 4, 13, 40, 121, ... ($\rightarrow s_i = 3s_{i-1} + 1$)
- String is 13 – Sorted. Next, do same with 4 Column Matrix
 - Organize String in Matrix with 4 columns & sort them using Insertion Sort:



Shell Sort

- Defined Sequence: 1, 4, 13, 40, 121, ... ($\rightarrow s_i = 3s_{i-1} + 1$)
- String is 4 – Sorted. Treating it now like a 1 – Column - Matrix just means sorting the resulting String with Insertion

1, 7, 3, 6, 6, 8, 5, 12, 9, 10, 13, 14, 15, 17, 14, 21, 17, 27, 25, 26, 27, 28, 26, 28, 30, 30

↓
Insertion Sort

1, 3, 5, 6, 6, 7, 8, 9, 10, 12, 13, 14, 14, 15, 17, 17, 21, 25, 26, 26, 27, 27, 28, 28, 30, 30

- But why is this faster than just using Insertion?

Shell Sort - complexity

- In last step we needed only 17 Swaps
- Remember: Insertion needs $n^2/4$ steps in average case: $n = 26 \rightarrow$ about 130 swaps.
 - \rightarrow We did Insertion sort either on very small sets or on pre – sorted ones.
- Scientists believe it's somewhere between $O(n^{3/2})$ in worst case and $O(n^{5/4})$ in average case.
- Polynomial runtime, we can do better!

Quick Sort

- Devide and Conquer!
- Choose an Element randomly, its called „Pivot“
- Elements smaller than Pivot go to the left, bigger ones to the right
- Do the procedure recursively on left and right set.
- Break if in the set are two or less Elements
- Quick!

Quick Sort - Complexity

- Comparing all Elements with Pivot $\rightarrow O(n)$
- 1. Step: One Set \rightarrow devide it
- 2. step: Two Sets \rightarrow devide them
- 3. Step: Four Sets $\rightarrow \dots$
- ...
- m-th Step: 2^m Sets
 $\rightarrow \log_2(n)$ Steps: Whole Array is sorted!
- Thus Quick Sort is in $O(n \log (n))$ in average
- unless you choose unlucky Pivot $\rightarrow O(n^2)$

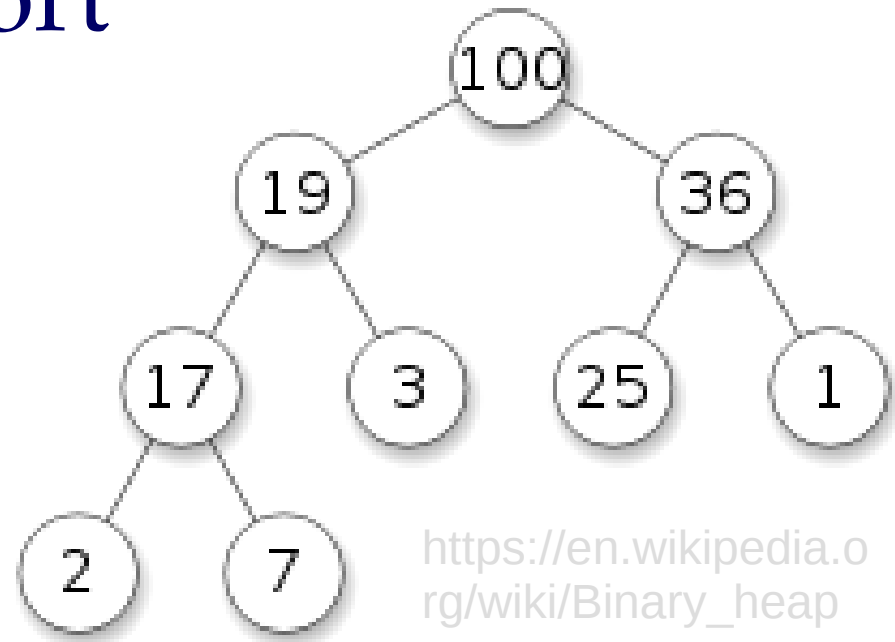
Heap Sort

- 2 Steps: Heap & Sort
- Heap:

- Complete binary tree

- every level, except possibly
the last, is completely filled

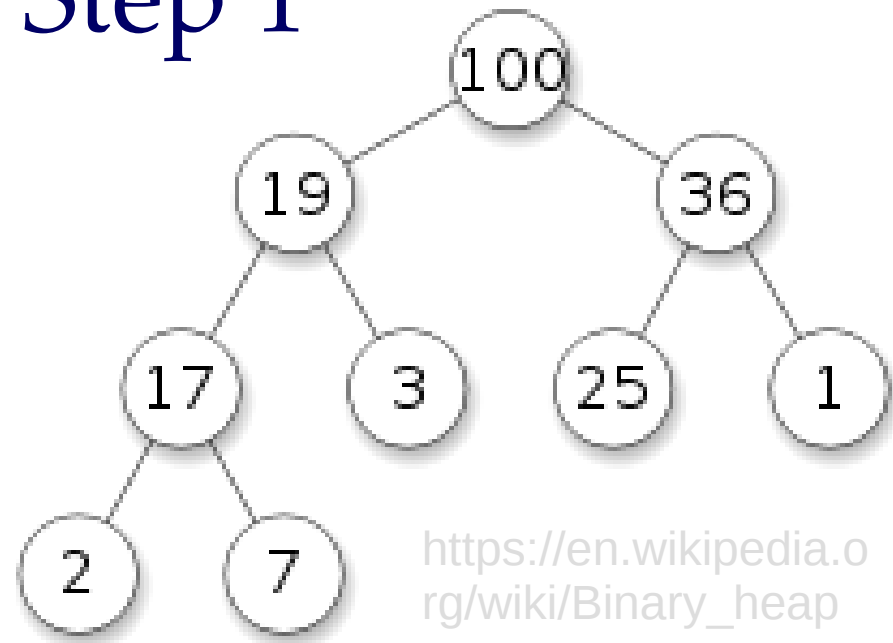
- Children of each node are smaller or equal
than father.



https://en.wikipedia.org/wiki/Binary_heap

Heap Sort – Step 1

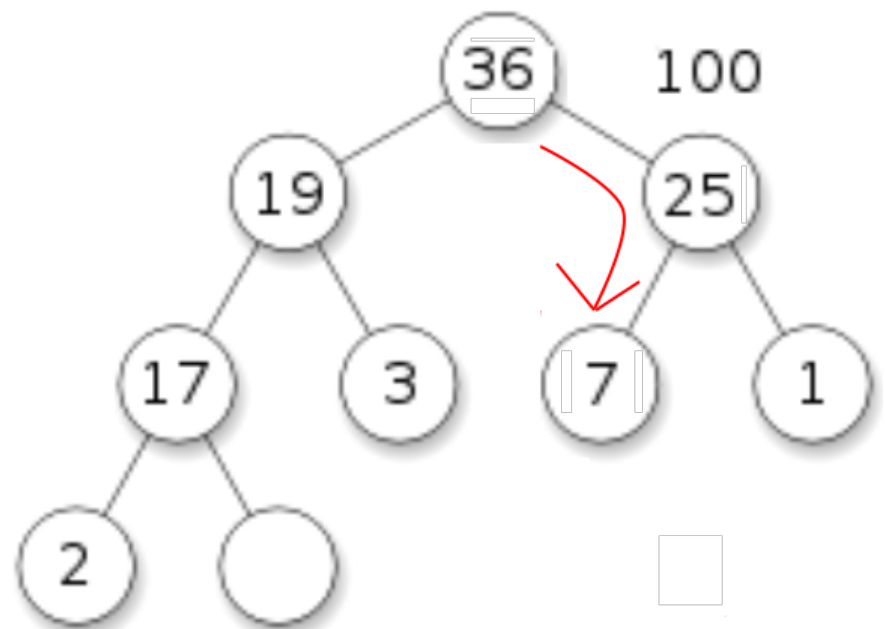
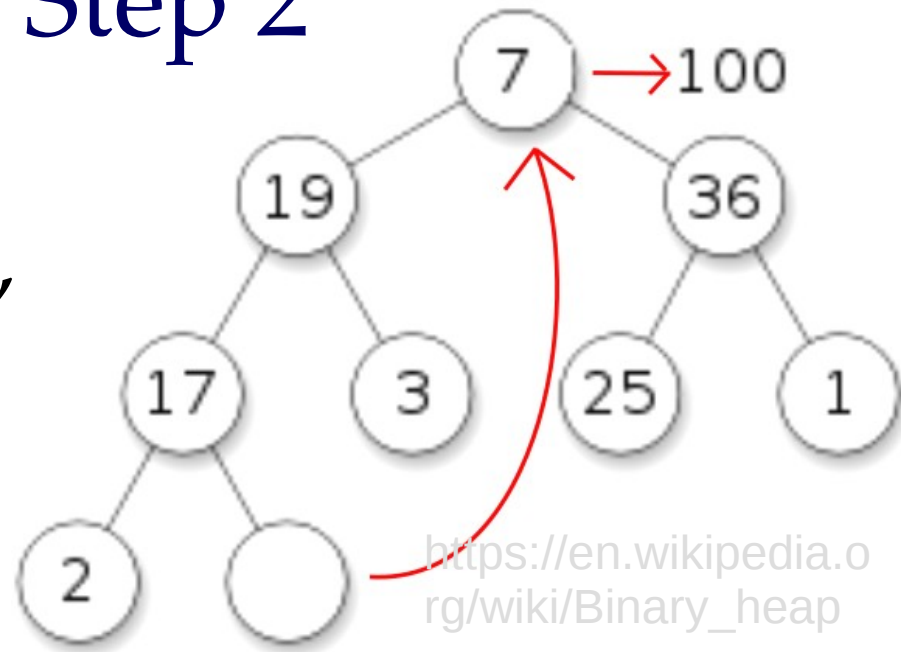
- How to achieve a Heap:
 - organize elements arbitrarily in a complete Tree



- do depth search, if child is greater than father, swap, check again until condition is fulfilled
- Trees have less Edges than Vertices $|E| < n$
- Depth of a binary Tree is always $\log_2(n)$ or less
→ $O(n \log(n))$ steps to build the Heap

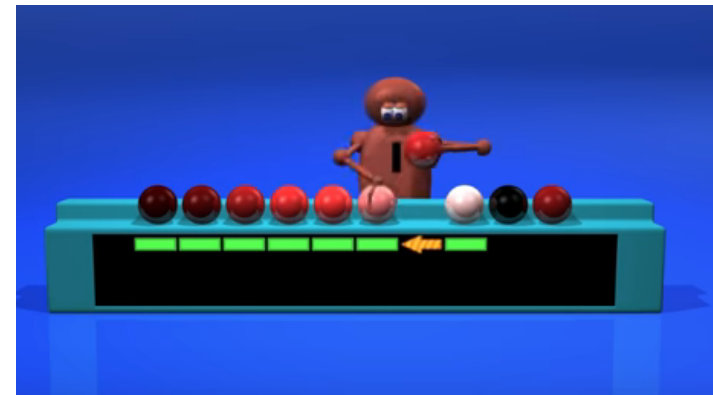
Heap Sort – Step 2

- Take out the root Element, its the biggest now
- Put element at lowest level to the root
- Swap with biggest child until Heap condition is fulfilled again
- Do as long as there are Elements in the Heap
Again: $O(n \log n)$



Homework

- If you haven't yet, watch these three cute little videos:
 - <https://youtu.be/TZRWRjq2CAg>
 - „Insertion Sort vs Bubble Sort + Some analysis!“
 - <https://youtu.be/H5kAcmGOn4Q>
 - „Heaps and Heap Sort“
 - <https://youtu.be/aXXWXz5rF64>
 - „Visualization of Quick sort“



Assignment

- You have now the tools to do proper sorting!
- Write a program that reads a list of numbers or characters as input, uses Quick sort and returns the sorted list as output.
- Deadline: 29. May 2018 23:59
- Mailto: michael.schwarzkopf@uni-weimar.de

Good luck!