FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
NATURWISSENSCHAFTLICHE
FAKULTÄT

# Introduction

Lecture “Mathematics of Learning (Maths of Data Science 1)”

Andreas Bärmann
Friedrich-Alexander-Universität Erlangen-Nürnberg
Winter semester 2022

WIRTSCHAFTS
MATHEMATIK

# Preliminaries

- These lecture notes are based on the material by Frauke Liers from the summer semester 2022. It is a compilation from different sources that will be mentioned for further references during the lecture.
- Among the different sources: *Many thanks to Frauke Liers, Martin Burger, Daniel Tenbrinck, and Philipp Wacker for allowing me to use their material from earlier lectures!*
- We will start from this state for the first chapter, then adapt/edit, etc., according to needs.

# Preliminaries

- These lecture notes are based on the material by Frauke Liers from the summer semester 2022. It is a compilation from different sources that will be mentioned for further references during the lecture.
- Among the different sources: *Many thanks to Frauke Liers, Martin Burger, Daniel Tenbrinck, and Philipp Wacker for allowing me to use their material from earlier lectures!*
- We will start from this state for the first chapter, then adapt/edit, etc., according to needs.
- Lecture: each Wednesday, 18:15 PM - 19:45 PM German time, H11, videos from SS 2022 are available.
- Forum on StudOn to ask your questions, will be read regularly.

# Preliminaries

- These lecture notes are based on the material by Frauke Liers from the summer semester 2022. It is a compilation from different sources that will be mentioned for further references during the lecture.
- Among the different sources: *Many thanks to Frauke Liers, Martin Burger, Daniel Tenbrinck, and Philipp Wacker for allowing me to use their material from earlier lectures!*
- We will start from this state for the first chapter, then adapt/edit, etc., according to needs.
- Lecture: each Wednesday, 18:15 PM - 19:45 PM German time, H11, videos from SS 2022 are available.
- Forum on StudOn to ask your questions, will be read regularly.
- Exercises: Jan Krause and Ehsan Waiezi, take place online in the first week

# Preliminaries

- Exercises consist of both mathematical and implementation tasks

# Preliminaries

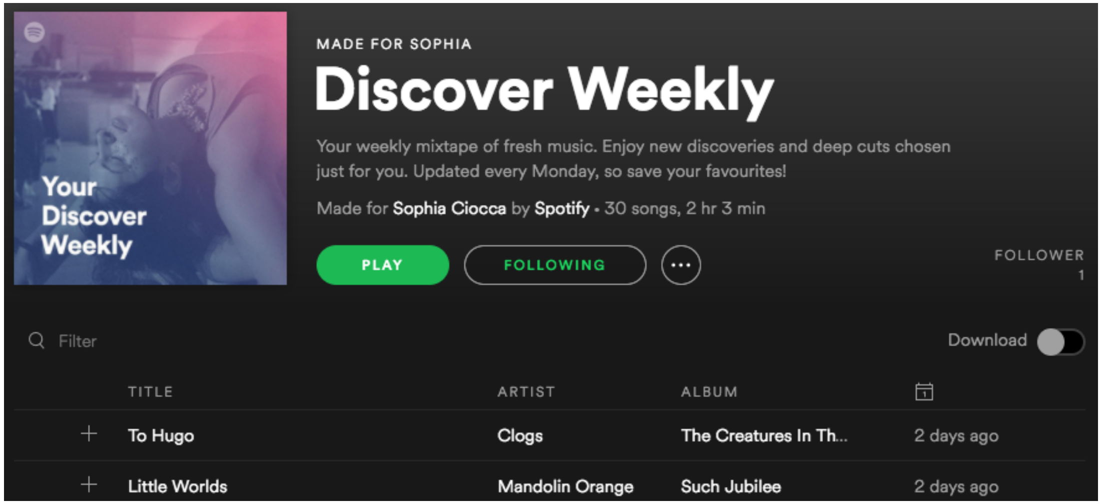
- Exercises consist of both mathematical and implementation tasks
- Students need to pass a written exam in presence at the end of the semester.
- Passing this course is mandatory for Master Data Science.

# What is Data Science?

- relatively **new field** due to recent boost in digital transformation → *digitization, IoT, paperless office, ...*
- needs mathematics, computer science, and domain knowledge
- well-known examples for successfully harvesting and interpreting data:
  → Google
  → Spotify
  → Amazon

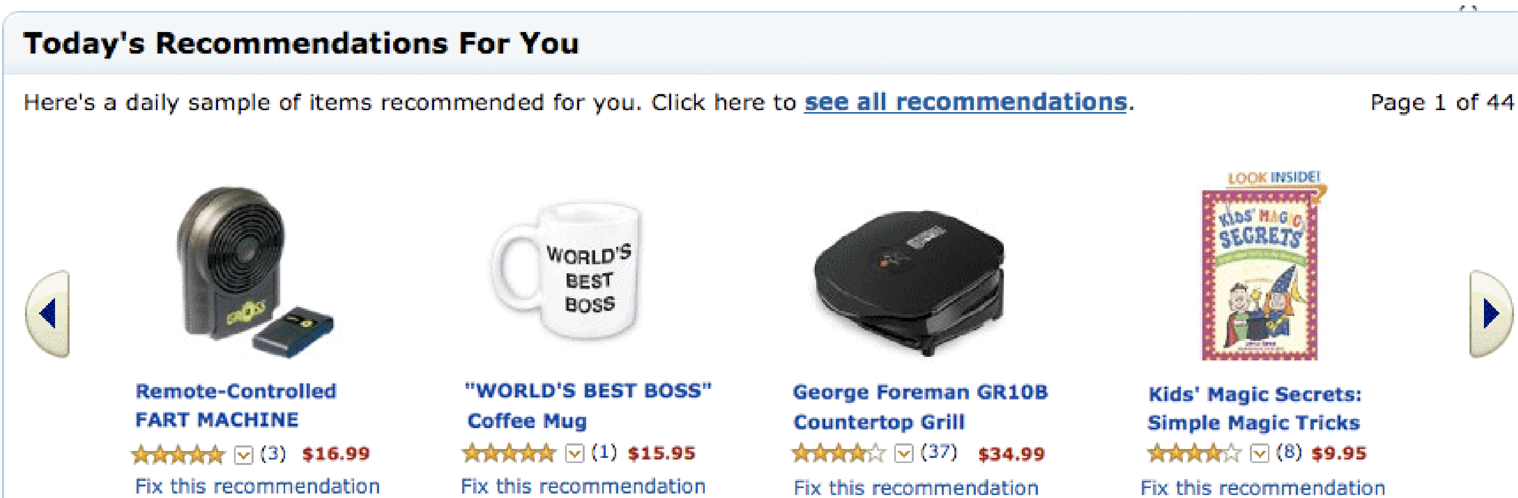

# What is Data Science?

- relatively **new field** due to recent boost in digital transformation $\rightarrow$ *digitization, IoT, paperless office, ...*
- needs mathematics, computer science, and domain knowledge
- well-known examples for successfully harvesting and interpreting data:
  - $\rightarrow$ Google
  - $\rightarrow$ Spotify
  - $\rightarrow$ Amazon



# What is Data Science?

- relatively **new field** due to recent boost in digital transformation $\rightarrow$ *digitization, IoT, paperless office, ...*
- needs mathematics, computer science, and domain knowledge
- well-known examples for successfully harvesting and interpreting data:
  - $\rightarrow$ Google
  - $\rightarrow$ Spotify
  - $\rightarrow$ Amazon



# Data Science Cycle

**iteratively...**

1. extract and process / correct data
2. extract hypotheses
3. *develop approaches, models, algorithms, implementations (from statistics, optimization, numerics and simulation, math theory, databases, AI,...)*
4. use approaches to explore and understand data
5. derive predictions, decisions, consequences, recommendations for application domain
6. visualize data and results, possibly iterate

Your study programme covers these aspects, with specializations possible.

Many applications: logistics, mobility, health care, energy networks, ...

# Data Science Cycle

**Goals of this course**

- learn fundamental math-based data science concepts and algorithms, know where to look it up
- understanding the underlying mathematical reasons why certain algorithms work well and others do not
- solve realistic problems

# Data Science Cycle

**Goals of this course**

- learn fundamental math-based data science concepts and algorithms, know where to look it up
- understanding the underlying mathematical reasons why certain algorithms work well and others do not
- solve realistic problems

**Topics: unsupervised learning and supervised learning methods, e.g.**

- clustering
- PCA, kernel methods, kernel-PCA
- statistical methods
- machine learning via neural networks
- graphical models,...

# Some Data Science Examples

- prediction of solar injection in energy networks
  and best-possible curtailment decisions beforehand
  to avoid breakdown of the network
- learn customer / market preferences, produce accordingly
  and forecast price developments
- predict shelf life of food
- forecast development of chronic diseases,
  determine best possible medication and treatment
- forecast development of Covid-19, determine best possible reactions
- many more

# Further Reading

- Hastie, Tibshirani, Friedman: The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Springer
- www.deeplearningbook.org

# Excursion: Some Background in Algorithms

This excursion is meant to give some brief and abstract introduction into algorithms. Further reading, e.g.:

- Thomas Cormen, Charles Leiserson, Ronald Rivest, and Cliff Stein: Introduction to Algorithms, MIT Press
- Thomas Ottmann and Peter Widmayer: Algorithmen und Datenstrukturen,
- Robert Sedgewick, Kevin Wayne: Algorithms, Addison-Wesley,
- Donald Knuth: The Art of Computer Programming,

and many others.

For exemplary purposes, we consider a basic operation: sorting numbers.

# Sorting: Notation

Sorting is a basic operation in computer science.

- **Input:** $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Output:** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

# Sorting: Notation

Sorting is a basic operation in computer science.

- **Input:** $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Output:** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- **instance**: each concrete series of numbers, e.g.,
  $\langle 32, 25, 13, 48, 39 \rangle \Rightarrow \langle 13, 25, 32, 39, 48 \rangle$
  (in general: all input that is necessary for determining a solution.)

# Sorting: Notation

Sorting is a basic operation in computer science.

- **Input:** $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Output:** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- **instance**: each concrete series of numbers, e.g.,
  $\langle 32, 25, 13, 48, 39 \rangle \Rightarrow \langle 13, 25, 32, 39, 48 \rangle$
  (in general: all input that is necessary for determining a solution.)

Depending on the application, different algorithms are suited best:

- How many elements?
- Is already partially sorted?
- In main memory, or on disk?

⋮

# Sorting: Notation

Sorting is a basic operation in computer science.

- **Input:** $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Output:** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- **instance**: each concrete series of numbers, e.g.,
  $\langle 32, 25, 13, 48, 39 \rangle \Rightarrow \langle 13, 25, 32, 39, 48 \rangle$
  (in general: all input that is necessary for determining a solution.)

Depending on the application, different algorithms are suited best:

- How many elements?
- Is already partially sorted?
- In main memory, or on disk?

$\vdots$

An algorithm is called **correct**, if it **terminates** for all instances with a correct solution. It then **solves** the problem.

# Insertion Sort


Figure 2.1 Sorting a hand of cards using insertion sort.

Aus: Cormen et al. (2001) "Algorithms", chpt. 2;
MIT Press, Cambridge (MA)

# Insertion Sort

As parameters, it has the array `A` and its length `length(A)`. In the for-loop, the *j*-th element of the sequence is inserted in the correct position that is determined by the while-loop. In the latter we compare the element to be inserted (`key`) from 'right' to 'left' with each element from the sorted subsequence stored in `A[0],...,A[j-1]`. If `key` is smaller, it has to be insert further left. Therefore, we move `A[i]` one position to the right and decrease `i` by one in line 7. If the while-loop stops, `key` is inserted.

```
insertion_sort(A)
  for j = 1 to (length(A)-1) do
    key = A[j]
    // insert A[j] into the sorted sequence A[1...j-1]
    i = j
    while (i > 0 and (A[i-1] > key) ) do
      A[i] = A[i-1]
      i = i-1
    end while
    A[i] = key
    end for
```

# Insertion sort for the example sequence $\langle 5, 2, 4, 6, 1, 3 \rangle$

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] |
|---|---|---|---|---|---|
| 5 | 2 | 4 | 6 | 1 | 3 |
| 2 | 5 | 4 | 6 | 1 | 3 |
| 2 | 4 | 5 | 6 | 1 | 3 |
| 2 | 4 | 5 | 6 | 1 | 3 |
| 1 | 2 | 4 | 5 | 6 | 3 |
| 1 | 2 | 3 | 4 | 5 | 6 |

# Correctness

- At beginning of the `for`-loop, `A[1..j-1]` is always sorted.

# Correctness

- At beginning of the `for`-loop, `A[1..j-1]` is always sorted.
- Within the `for`-loop, `A[j-2]`, `A[j-3]`, `A[j-4]`, ... are moved one position to the right until the correct position for `key=A[j]` is found and assigned to `key`.

# Correctness

- At beginning of the `for`-loop, `A[1..j-1]` is always sorted.
- Within the `for`-loop, `A[j-2]`, `A[j-3]`, `A[j-4]`, ... are moved one position to the right until the correct position for `key=A[j]` is found and assigned to `key`.
- When the `for`-loop stops, we have `j=n` and `A[0..n-1]` is sorted.

# Correctness

- At beginning of the `for`-loop, `A[1..j-1]` is always sorted.
- Within the `for`-loop, `A[j-2]`, `A[j-3]`, `A[j-4]`, `...` are moved one position to the right until the correct position for `key=A[j]` is found and assigned to `key`.
- When the `for`-loop stops, we have `j=n` and `A[0..n-1]` is sorted.
- Termination: The `while`- and the `for`-loop always terminate.

# Used Ressources

- space
- **time**

# Used Ressources

- space
- **time**

**RAM: “random access machine”**:

- 1 processor
- data in main memory
- All memory accesses need the same time.

# Used Ressources

- space
- **time**

**RAM: “random access machine”**:

- 1 processor
- data in main memory
- All memory accesses need the same time.

time:

- number of primitive operations, e.g., $a = b$, $a = b + c$, etc.

# Used Ressources

- space
- **time**

**RAM: “random access machine”**:

- 1 processor
- data in main memory
- All memory accesses need the same time.

time:

- number of primitive operations, e.g., $a = b$, $a = b + c$, etc.
- assumption: each such operation takes equal time.

# Used Ressources

- space
- **time**

**RAM: “random access machine”**:

- 1 processor
- data in main memory
- All memory accesses need the same time.

time:

- number of primitive operations, e.g., $a = b$, $a = b + c$, etc.
- assumption: each such operation takes equal time.

## “Worst Case”-Running Time

- **longest possible running time** for predescribed input size (here $n$).

## “Worst Case”-Running Time

- **longest possible running time** for predescribed input size (here $n$).
- **upper bound** for running time for solving an arbitrary instance of this size.

## “Worst Case”-Running Time

- **longest possible running time** for predescribed input size (here $n$).
- **upper bound** for running time for solving an arbitrary instance of this size.
- For some applications, “worst-case” appears more often than for others, e.g. when searching for a non-existent entry in a data base.

## “Worst Case”-Running Time

- **longest possible running time** for predescribed input size (here $n$).
- **upper bound** for running time for solving an arbitrary instance of this size.
- For some applications, “worst-case” appears more often than for others, e.g. when searching for a non-existent entry in a data base.

Question: Does the worst-case running time of insertion_sort grow linearly, quadratically, . . ., or even exponentially in $n$?

# Running Time Insertion Sort

```
void insertion_sort(A){
  for j = 1 to < length(A-1) do
    key = A[j]
    // insert A[j] into the sorted sequence A[1...j-1]
    i = j
    while (i > 0 and A[i-1] > key) do
      A[i] = A[i-1]
      i= i-1
    end while
    A[i] = key
  end for
```

# Running Time Insertion Sort

```
void insertion_sort(A){
  for j = 1 to < length(A-1) do
    key = A[j]
    // insert A[j] into the sorted sequence A[1...j-1]
    i = j
    while (i > 0 and A[i-1] > key) do
      A[i] = A[i-1]
      i= i-1
    end while
    A[i] = key
  end for
```

**For a sequence sorted in reverse order (worst case):**

- The `while`-loop stops only when $i = 0$
- $j = 1$: 1 assignment $A[i] = A[i-1]$
- $j = 2$: 2 assignments
- $\ldots$
- $j = n-1$: $n-1$ assignments

# Running Time Insertion Sort

```
void insertion_sort(A){
  for j = 1 to < length(A-1) do
    key = A[j]
    // insert A[j] into the sorted sequence A[1...j-1]
    i = j
    while (i > 0 and A[i-1] > key) do
      A[i] = A[i-1]
      i= i-1
    end while
    A[i] = key
  end for
```

**For a sequence sorted in reverse order (worst case):**

- The `while`-loop stops only when $i = 0$
- $j = 1$: 1 assignment $A[i] = A[i-1]$
- $j = 2$: 2 assignments
- $\ldots$
- $j = n-1$: $n-1$ assignments

in total: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ many assignments; quadratically many

# ...more formally

## Worst-Case Running Time

Idea: Consider only the characteristic behaviour as a function of the input size; ignore constants and terms of lower order.

# ...more formally

## Worst-Case Running Time

Idea: Consider only the characteristic behaviour as a function of the input size; ignore constants and terms of lower order.
If the values of a function $f$ are "less" than those of another function $g$ for all $n$ larger than some constant $n_0$ (up to a constant factor $c$), then asymptotically $g$ is an upper bound for $f$, denoted by $f \in O(g)$.

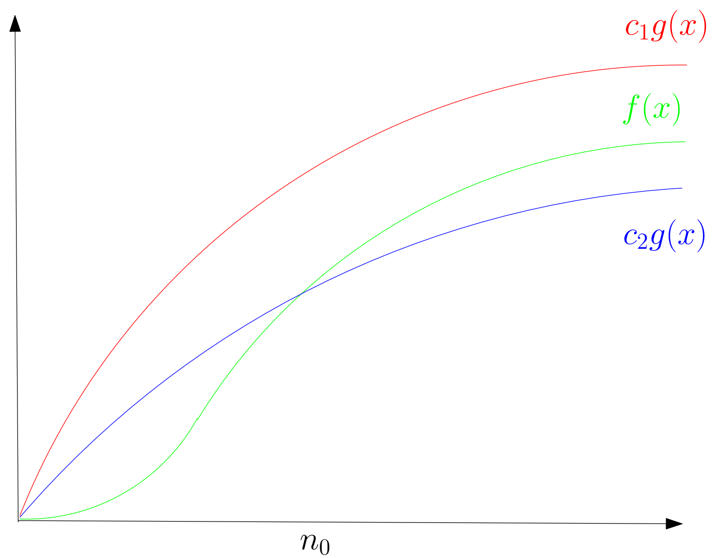
More generally, the following symbols are often used for a function $g \colon \mathbb{N} \to \mathbb{R}$ (for upper bounds $O$, for lower bounds $\Omega$ and for equally fast growth $\Theta$):

$$O(g) := \{f \colon \mathbb{N} \to \mathbb{R} \mid (\exists c, n_0 > 0)(\forall n \geq n_0) : 0 \leq f(n) \leq cg(n)\}$$

$$\Omega(g) := \{f \colon \mathbb{N} \to \mathbb{R} \mid (\exists c, n_0 > 0)(\forall n \geq n_0) : 0 \leq cg(n) \leq f(n)\}$$

$$\Theta(g) := \{f \colon \mathbb{N} \to \mathbb{R} \mid (\exists c_1, c_2, n_0 > 0)(\forall n \geq n_0) : c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

# Worst-Case Running Time




- The *worst case* running time of insertion sort is $O(n^2)$.

# Design of Algorithms

insertion sort: *incremental method.*
different principle: *'divide and conquer'*

- **divide** problem in subproblems
- **conquer** the subproblems through recursive solution. (If small enough, solve them directly.)
- **combine** the solutions of the subproblems to a solution for the original problem.

# Merge Sort

- **divide:** divide sequence of $n$ numbers in the middle into two sub sequences.
- **conquer:** sort the subsequences recursively using *merge sort*.
- **combine:** merge the two subsequences to a sorted sequence.

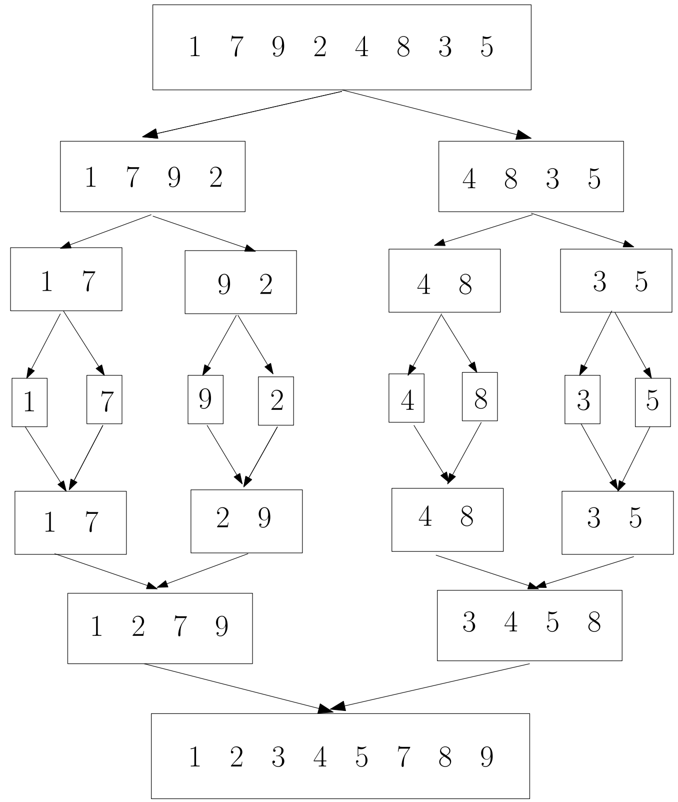
$\longrightarrow$ for sequences containing one element only nothing has to be done.

# Merge sort

Sort the sequence stored in `A[p]...A[r]`:

```
void merge_sort(int[] A, int p, int r) {
  int q; /* Middle of the sequence */
  if (p < r) {  /* if p = r: only 1 element */
    q = p+((r-p)/2);
    merge_sort (A, p, q);  /* left subsequence */
    merge_sort (A, q+1, r);  /* right subsequence */
    merge (A, p, q, r);  /* merge subsequences */
  }
}
```

# Illustration of Merge Sort



# Merge Sort

It can be shown: worst-case running time of merge sort is $O(n \log n)$ (better than insertion sort)

In fact: any algorithm for sorting $n$ numbers that uses only comparisons and moves of numbers needs at least $\Omega(n \log n)$.

BTW: try the shell-command **sort**!

This is the end of our excursion.
We will now return to the introduction of learning methods.