Lecture 2: Programming & Algorithms

Seminar ‘Foundations of Data Science’

Prof. Dr. Karsten Donnay, Assistant: Marcel Blum

Course Outline

- Part 1: Foundations
  - *Day 1: Mon. 14.06.2021*: Information Coding & Data
  - ***Day 2: Tue. 15.06.2021*: Programming & Algorithms**
  - *Day 3: Wed. 16.06.2021*: Complexity & Efficiency
- Part 2: Applications
  - *Day 4: Thu. 17.06.2021*: Data Collection & Quality
  - *Day 5: Fri. 18.06.2021*: Research on Digital Media

Overview of this Session

- Programming
  - Programming Fundamentals
  - Programming Paradigms
  - Best Practices
- Algorithms
  - Concepts
  - Recursion
  - Divide and Conquer

University of Zurich[UZH]

Department of Political Science

Programming

Programming Language

– **Definition:**
  A programming language is a formal computer language used to communicate instructions to a machine. The instructions (commands) can either we written in machine code or as abstract, human readable source code that is automatically translated to machine code.

– There is a very long line of programming languages by now
  - Languages are usually developed with specific set of applications in mind
  - Design of language reflects specific choices “optimal” for a given setting
  - General purpose languages can be used across domains
    – C, C++, Java, JavaScript, Python, Fortran, Pascal, PHP, Perl, Julia etc.

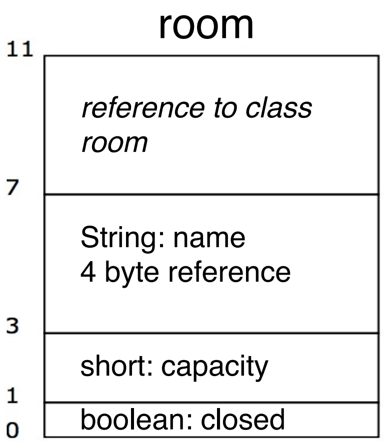
Programming Language

- Functions and Objectives
  - Machine readability
    - Efficient translation to machine language
    - Realized through context-free languages
  - Human readability
    - Abstraction from the computer architecture
    - Oriented on natural language
  - Abstraction
    - From the internal representation and storage of data
    - From the internal processes inside the processor

Programming Language

- Structured data abstraction
  - Data structure itself is the fundamental concept of most programming languages
    - Class
    - List
    - Dictionary
  - Programmer is usually not interested in the exact structure of an object in memory

```
public class room {
        private String name;
        private short capacity;
        private boolean closed;
}
```



Programming Language

- Control abstraction
  - Example: x = x + 3
    - Load value of variable x
    - Addition of 3
    - Save result in variable x
  - Methods (or functions) are not specifically marked in machine code
    - Execution through jump to respective memory address
    - Updating of the call stack (parameters, return address etc.)
    - Details of how this is done, are usually not relevant to the software programmer

Programming Language

- Syntax of a (programming) language
  - Structure:
    - Which characters are allowed?
    - How can these characters be combined to valid words?
    - How can words be composed to form valid “expressions”?
  - Does **not** specify the meaning (or correct semantics) of expressions
  - Example: natural language
    - Incorrect syntax: Carl soccer eats.
    - Correct syntax: Carl eats soccer.

      BUT not meaningful semantics

Programming Language

– Semantics
  - Specifies the meaning of a syntactically correct expression
    – e.g.: i = i + 1

      “The value of the memory block that i is pointing to is incremented by 1 after execution.”
  - Description of semantics usually in natural language
  - Semantic validation after validation of the syntax
    – Are the data types of operations compatible?
    – Is the identifier valid?
  - RStudio includes the option for “Code Diagnostics” to automatically check your code for inconsistencies…

Programming Language

– Simple example in R:

```
x = a b +
```

- Syntactically incorrect command
  – Structure of assignment is incorrect:
    variable on left hand side, then '=', then sum of a and b

```
x = a + b
```

– Syntactically correct command BUT semantic meaning is only clear if a and b are already previously defined…

Programming Language

- Lexical structure
  - Specifies structure of “words” or tokens
    - Reserved words
      `if, for, function, while …`
    - Constants

      numbers, e.g., `1, 42, -5, 1.2`

      strings, e.g., `“Hello World”` (including “”)
    - Special characters

      `:, +=, +, -,…`
    - Identifier

      `my_function, my_very_long_variable_name,…`

Programming Language

- Type Systems
  - Assignment of types to “objects” (variables, functions, objects)
    - Restrict the range of values of variables
    - Check correct usage to avoid errors at runtime
  - Primitive data types
    - `int, double` etc.
    - R flexibly assigns data types contrary to most other languages
  - Programs like Java or C have rules to check the correct type assignment etc.
  - R does all of this at runtime (and is slower because of that…)

Programming Language

- Classification
  - Strong vs. weak typing
    - Strict restrictions of operations that are allowed
    - x = “2” + 3: not allowed for strong typing (like R) but allowed for weak typing
    - Examples of languages with weak typing: Perl or PHP
  - Dynamic typing vs. static typing
    - Dynamic assignment of data type at runtime based on assigned values
    - R does dynamic typing
  - Explicit vs. implicit typing
    - Need to always specify data type (e.g. Java or C)
    - Automatic determination of type based on values (e.g. R)

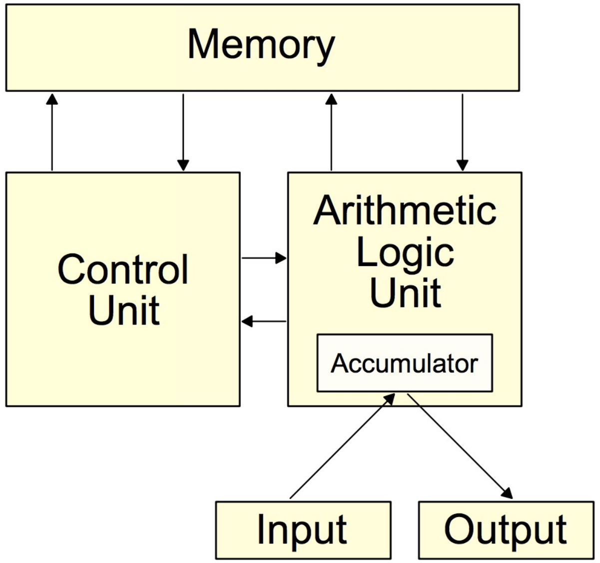
Programming Paradigms

Paradigm	Description	Main traits	Related paradigm(s)	Critique	Examples
Imperative	Programs as statements that *directly* change computed state (datafields)	Direct assignments, common data structures, global variables		Edsger W. Dijkstra, Michael A. Jackson	C, C++, Java, PHP, Python, Ruby
Structured	A style of imperative programming with more logical program structure	Structograms, indentation, no or limited use of goto statements	Imperative		C, C++, Java, Python
Procedural	Derived from structured programming, based on the concept of modular programming or the *procedure call*	Local variables, sequence, selection, iteration, and modularization	Structured, imperative		C, C++, Lisp, PHP, Python
Functional	Treats computation as the evaluation of mathematical functions avoiding state and mutable data	Lambda calculus, compositionality, formula, recursion, referential transparency, no side effects	Declarative		C++,[1] Clojure, Coffeescript,[2] Elixir, Erlang, F#, Haskell, Lisp, Python, Ruby, Scala, SequenceL, Standard ML, JavaScript
Event-driven including **time-driven**	Control flow is determined mainly by events, such as mouse clicks or interrupts including timer	Main loop, event handlers, asynchronous processes	Procedural, dataflow		JavaScript, ActionScript, Visual Basic, Elm
Object-oriented	Treats datafields as *objects* manipulated through predefined methods only	Objects, methods, message passing, information hiding, data abstraction, encapsulation, polymorphism, inheritance, serialization-marshalling	Procedural	Here and[3][4][5]	Common Lisp, C++, C#, Eiffel, Java, PHP, Python, Ruby, Scala
Declarative	Defines program logic, but not detailed control flow	Fourth-generation languages, spreadsheets, report program generators			SQL, regular expressions, CSS, Prolog, OWL, SPARQL
Automata-based programming	Treats programs as a model of a finite state machine or any other formal automata	State enumeration, control variable, state changes, isomorphism, state transition table	Imperative, event-driven		Abstract State Machine Language

Source: Wikipedia

Imperative Programming

– van-Neumann model (or architecture)
  - Central memory
  - Control unit with sequential execution of commands



Imperative Programming

- Language structure
  - **Instruction** or command is the central construct
    - Imperare, lat. = to order, to instruct
  - **Control structures** govern execution of instructions
    - e.g. `if` conditions, `for` loops, `if … else` jumps etc.
  - **Data structures** to organize data
    - Relevant information in array, list, dictionary etc. and update sequentially
  - **Functions and procedures** to structure the processing
    - Break up the code in smaller parts with well defined inputs and outputs

Imperative Programming

- Characteristics
  - Advantages
    - Step-by-step execution of instructions
    - Simple to understand
  - Disadvantages
    - Side effects:
      Methods can (unintentionally) change variable values and with that the state of the program
    - Sequential execution:
      Distributed execution on several processors is not easy to realize (van Neumann bottleneck)

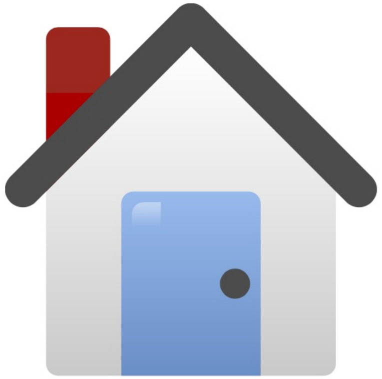
Object-oriented Programming

- Motivation
  - Why do we need object-oriented programs?
    - Since 70s programs were getting too large
      (UNIX, data bases, information systems etc.)
    - More and more variants of the same program
      (different versions, for different devices, different environments)
    - How do you manage that?
  - Similar problems for applications in simulation of complex structures
    - Simula: language for physical simulations
    - e.g., used in the construction of ships
    - Many objects, in different states

Objects

Objects

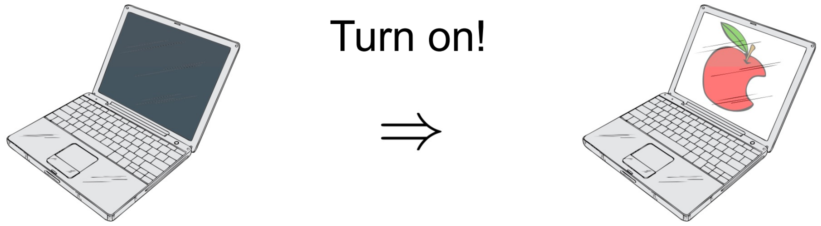
- Objects have properties or **states**
  - Changeable properties
    - Color, type of door
  - Unchangeable properties (almost)
    - no balcony, number of windows
- Objects can be **composed** of many smaller objects
  - House: doors, windows, balcony etc.
  - Car: wheels, engine, chassis, door etc.



- color = gray
- number of doors = 1
- number of windows = 0
- balcony = no

Objects

- Objects communicate **via messages**
  - Different responses as reaction to messages
    - Simple changes to properties (color, on/off)
    - Checking message, then changing of properties
  - Messages through call of a method
    - Parametrized (passing a value)
    - Non parametrized
    - Can give a return value



Object-oriented Programming

- **Classes** are types of objects
  - Share common properties but are different instances of that class
  - Class defines e.g. which variables exist but no their specific values
  - Variables for each object then then take on particular values
- Define **methods** that can change the state of an object
  - Methods are functions specifically defined for a given class
  - State of a given object can only be changed via method calls
  - Method call only changes the state of the object it is applied to
  - Note: certain properties of an object cannot/ are not meant to be changed

Object-oriented Programming

- Key advantages of using object-oriented programming
  - Clear specification of attributes and methods
    - All members of a class share the same properties
    - Properties can only changed via well-specified methods
    - Allows for very modular program structures
  - Inheritance:
    - Objects of a sub-class always also have all the properties and methods of the original class
    - Can further specify specific attributes or methods that only this sub-class of objects has
    - Allows for very efficient nested designs (also very code efficient)

Functional Programming

- Mathematical function as fundamental language element
  - In pure form, no assignments and no variables
  - No loops, hence recursion as central concept
  - Result of a function depends exclusively on parameters
    - No inner state variables
  - Functions can, like values, be used as parameters and return values

Functional Programming

- Characteristics
  - Advantages:
    - No side-effects
    - Simple to parallelize
    - Automatic memory management is easy
  - Disadvantages:
    - Unusual notation and usage
    - Very inefficient for iterative calculations

Functional Programming

- R follows a functional programming paradigm
  - Iterative calculations are very slow because changing one part of a data structure requires copying the whole structure
    - i.e., there are no inner states, variables
  - Applying same function to entire data structure is very fast because it applies to entire data at the same time
    - Function is the basic unit of execution
    - Key reason it is used for statistical applications…

Best Practices for Programming

- Many standards for “good” coding but common characteristics are
  - Extensive **documentation**
    - Others have to be able to understand and check your code
    - Enables taking existing code and expanding it
    - Simplifies finding errors and bugs
  - Good and consistent **structure**
    - Separation of specific functionality into separate methods/functions
    - One functionality should only exist once in the program
  - Systematic testing and **validation** of code
    - “Unit” testing of individual elements/methods
    - Benchmarking against test data and known results

Best Practices for Programming

- Role of platforms like GitHub, GitLab etc.
  - Quick **dissemination** of new code and functionalities
    - Simplifies code review
    - All changes/improvements visible to everyone
  - Full **transparency** and version tracking
    - It is clear who did what and when
    - Others can use the code but properly credit
      - Usually you provide licensing information (e.g. LGPL-3)
  - **Reputation** building
    - Best practices do not help much if nobody sees your work
    - Building software is collaborative and getting credit for your contribution builds your reputation

Algorithms

Algorithms

– **Definition:**
  An algorithm is a self-contained step-by-step set of operations to be performed.
  Algorithms perform calculation, data processing, and/or automated reasoning tasks.

– We will briefly look more closely into two key characteristics
  - Correctness
  - Complexity

Algorithms

– Correctness:
  - Partial correctness

    Every step in the algorithm has to be correct
  - Terminates

    For correct inputs, the algorithm always terminates

Algorithms

- Correctness: Examples
  - Ariane disaster on June 4, 1996
    - Ariane V88 rocket crashes during take-off
    - Overflow for conversion of float to int
    - Traveled through the entire system and shut down the control program
  - Millennium bug (or Y2K bug)
    - Most programs, including important control software, only saved the year using two digits
    - Uncertain what would happen switching from 99 to 0
    - Damage through hysteria much larger than actual consequences

Algorithms

- Correctness: Examples
  - Northeast blackout of 2003 (USA, Canada)
    - Wide-spread power outage on August 14, 2003
    - Software bug known as a “race condition” in control system stalled the control room alarm system for an hour
    - Lead to events piling up and shutting down the control, allowed small problems to to cascade out of control

Complexity

– What exactly do we mean by algorithmic complexity?
  - Typically, refers to time complexity of a given algorithm
  - Depends on a particular implementation
– Important distinctions
  - Different from memory complexity, i.e., how much memory does my algorithm need to execute
  - Different from problem complexity, i.e., how inherently “hard” is the problem I am solving (independent of my algorithmic solution)
– We will discuss complexity in much more detail in the next session…

Algorithmic Concepts

- We will briefly discuss two key algorithmic concepts
  - Recursion
  - Divide and Conquer
- They are important as implementation strategies for **efficient** solutions to complex problems

Recursion

- **Definition**:
  Recursion is a technique in mathematics, logic and computer science to specify a function through itself, i.e., using a recursive definition. The fundamental principle of recursion is to define the value of a function through previously calculated values of the same function.

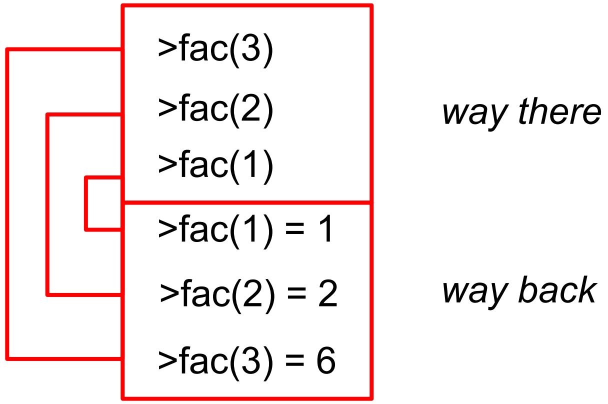
- Idea:
  - If the start value of function is known for sufficiently many arguments, it can be calculated
  - Function keeps calling itself until
    - Function converges on a specific target
    - Pre-defined abort condition is met
- Common examples
  - Factorial
  - Binomial coefficients

Linear Recursion

– How does recursion work?

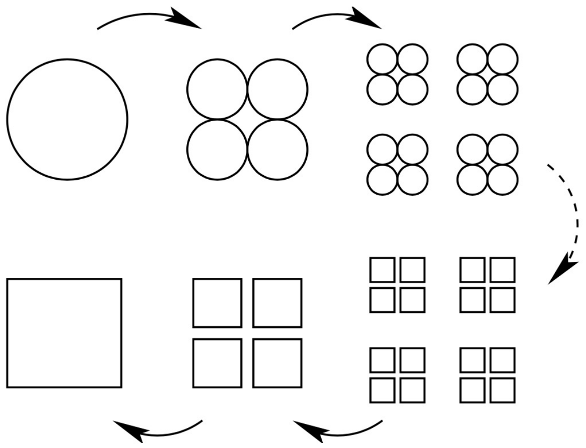
Algorithm in R

```
fac = function(n){
   if (n < 0){
      stop(“No values < 0 allowed”)
   }
   if (n == 0 | n == 1){
      return 1
   }
   result = n * fac(n-1)
   return result
}
```



Divide and Conquer

– Fundamental algorithmic principle
  - Separate a problem into separate sub-problems
  - Continue until those sub-problems are solvable
  - Recombine partial results to full solution



Divide and Conquer

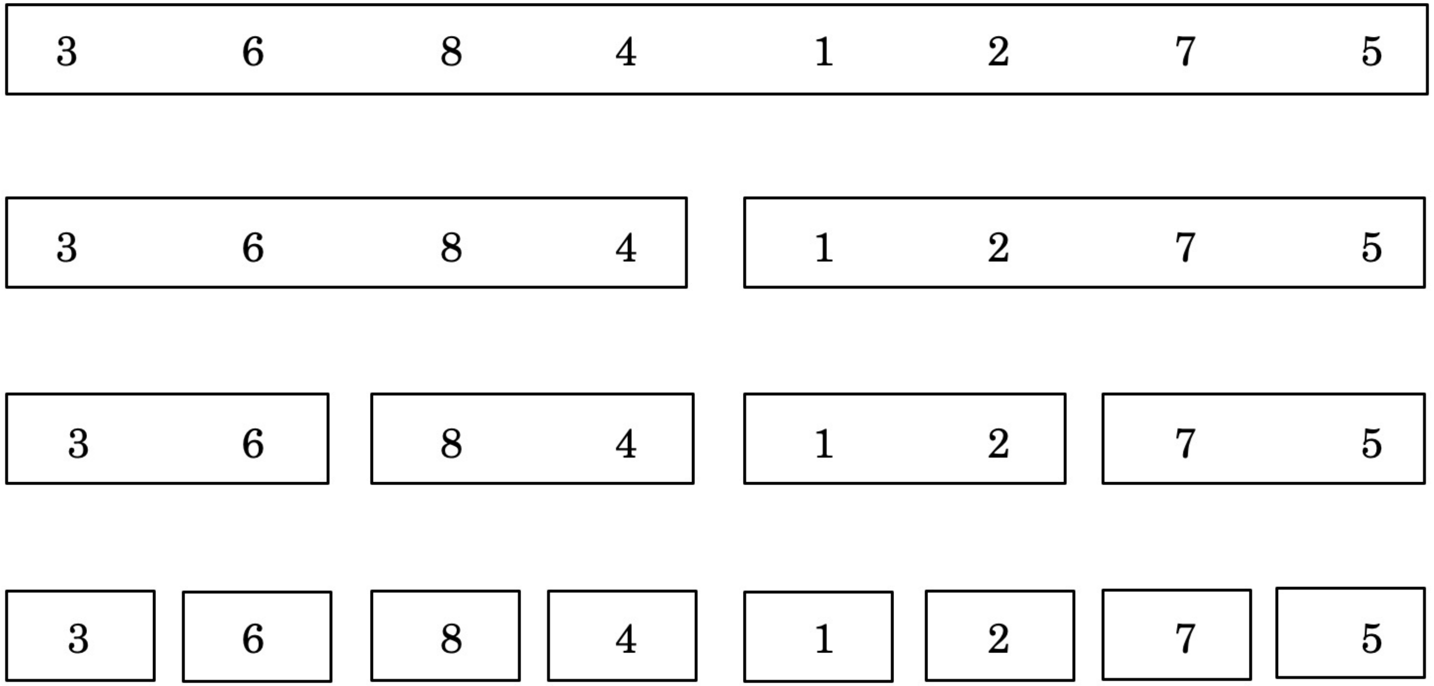
- Divide and conquer ≠ recursion
  - It is not the same as recursion, i.e., recursive calls of the same function
  - BUT recursion is a natural **implementation strategy** to solve divide and conquer problems
    - By definition, the sub-problems in divide and conquer are self-similar
    - Generate overall solution by combining solutions of sub-problems
  - You can also implement divide and conquer strategy iteratively though…

Divide and Conquer

- Example: **Mergesort** algorithm
  - Sorting by merging (sub-)lists
  - Principle:
    - List with one element is trivially sorted
    - Merging two sorted lists is simple
    - Compare first element of each sub-list
    - Append smaller of the two to merged list and remove from sub-list
    - Continue with remaining sub-lists
  - Algorithmic strategy
    - Repeated division of list until just one element remaining per list
    - Repeated merger or sorted sub-lists until only one sorted list remains

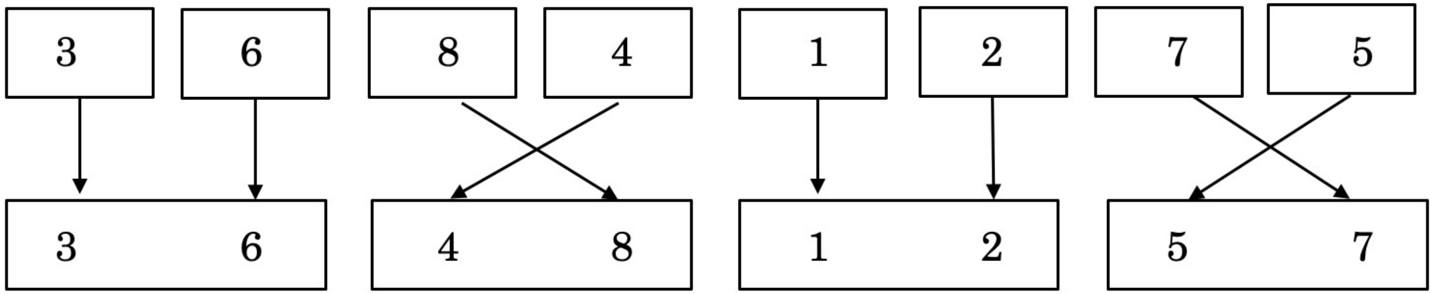
Divide and Conquer

– Example: **Mergesort** algorithm



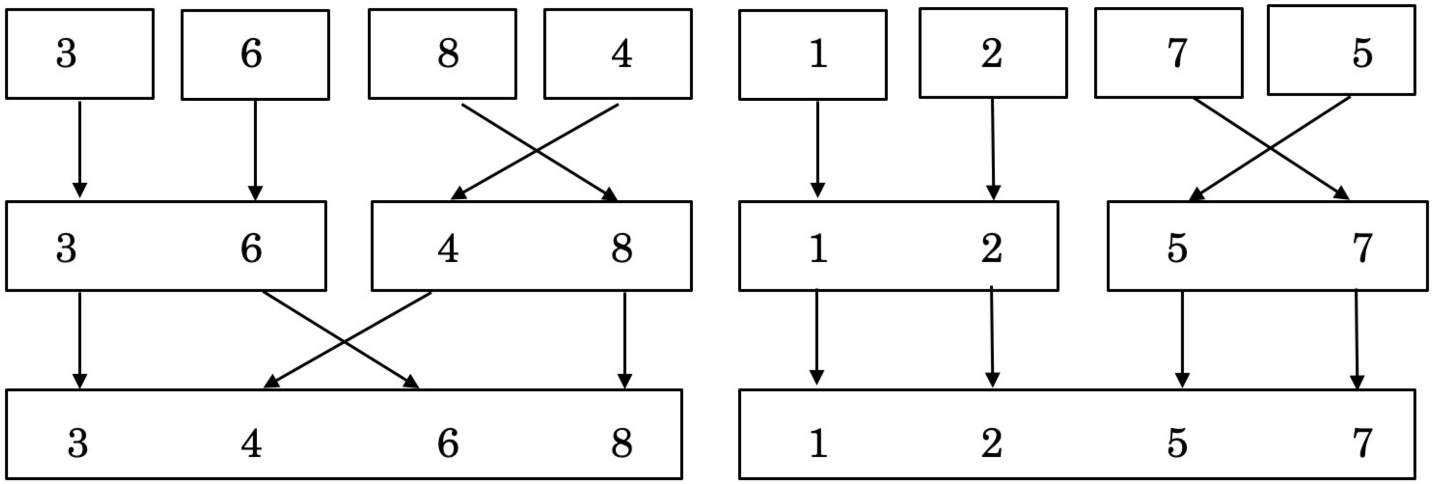
Divide and Conquer

– Example: **Mergesort** algorithm



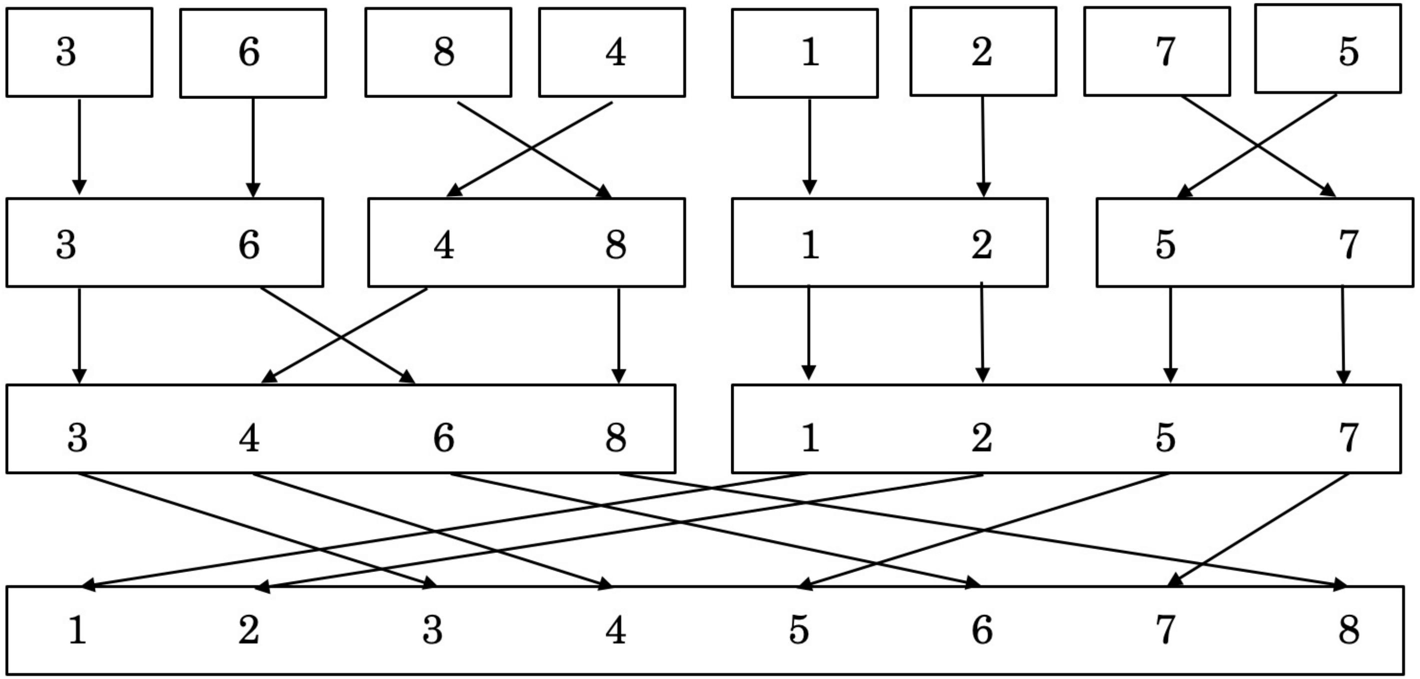
Divide and Conquer

– Example: **Mergesort** algorithm



Divide and Conquer

– Example: **Mergesort** algorithm



Divide and Conquer

- Example: **Mergesort** algorithm
  - Divide and conquer strategy
    - Splitting into two sub-lists (with half the size):
      - Recursive implementation of sub-list sorting
        - Call Mergesort in itself
        - Assume that the output is a sorted list in each recursion step
        - Requires well-defined base case (i.e., list with one element is already sorted)
    - Merging:
      - Systematic and efficient merger of partial lists to one large list

Up Next

- Exercise (this afternoon)
  - Data processing & cleaning
  - Manipulation of text data
  - Regular expressions
- Next lecture (tomorrow morning)
  - Complexity
  - Efficiency