Introduction to Informatics for Students from all Faculties

Prof. Dr. Ingo Scholtes

Chair of Machine Learning for Complex Networks
Center for Artificial Intelligence and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
Würzburg, Germany

ingo.scholtes@uni-wuerzburg.de

Lecture 05
Algorithmic Thinking

November 19, 2024

Notes:

- **Lecture L05: Algorithmic Thinking** 19.11.2024
- **Educational objective:** We introduce algorithms for basic problems like binary search and sorting. We discuss the runtime of algorithms and cover basic data structures.
  - Python Data Structures
  - A Simple Algorithm
  - Binary Search Algorithm
  - Basic Sorting Algorithms
- **Exercise Sheet 4** due 26.11.2024

Motivation

- we introduced **high-level programming languages** that are translated to machine code
- we wrote a first “Hello World” program in C and inspected the machine code generated by the **compiler**
- we distinguished between **compiled languages** like `C/C++` and **interpreted languages** like `python`
- we introduced basics of the **python syntax**

open issues

- how to write programs that **solve actual problems**?
- what are **algorithms** and how we can we implement them in high-level languages like python?
- need to develop **algorithmic thinking**, which is key to understand how a computer (scientist) works

Notes:

Recap: `python` Syntax

- python programs are stored in text files (typically with extension `.py`)
- **one line in text file = one instruction**

key `python` statements

- assignment (=) used to **assign value to a variable**
- `def` used to define a **function**
- `import` statement used to import functions from modules
- `if` and `else` used to **conditionally execute instructions**
- `for` and `while` used to **repeatedly execute instructions in a loop**

```
import time

def main():
  for i in range(5):
    text = "Hello World!"
    print(text)
    text = 42
    print(text)
  sleep(5)
```

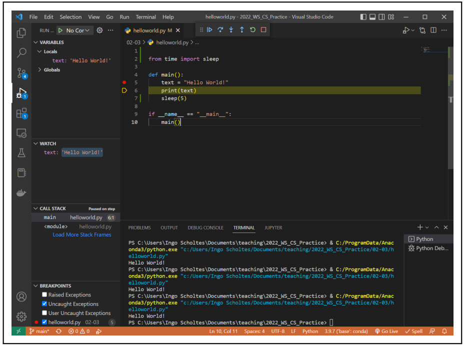
- "blocks" of instructions grouped by **indentation level**
- `python` is **whitespace-sensitive**, i.e. placement of newline, space or tab characters changes semantics
- `python` enforces meaningful formatting of code, making programs easy to read for humans

Notes:

- Note that the “grammatical structure” of a (programming) language is called “syntax”, which contains the Greek words “syn” (together) and “taxis” (ordering/composition). The syntax of a language defined keywords and determines the ordering of characters that constitutes a valid sentence or (program) in a (programming) language.

Integrated Development Environment (IDE)

- all we need to write `python` program is **text editor** and **python interpreter** (i.e. executable `python.exe`)
- sufficient for small single-file programs
- what about **complex software with hundreds of files and millions of lines in code?**
- integrated development environments (IDEs) are specialized tools to **support and simplify development of complex software**
- IDEs provide advanced functions to edit and format code, semantically highlight/color keywords, compile and/or execute program, and find errors



Open Source IDE Visual Studio Code

definition

An **Integrated Development Environment (IDE)** is a software that simplifies the programming of computers. It minimally provides functions to **edit source code files**, compile and/or execute programs, and **find errors at compile- and run-time**.

Notes:

Python Data Structures

- all programming languages support **basic data types**
  - integer numbers, i.e. $42, -55, 0$
  - floating point numbers, i.e. $4.52, 1.567e2, 2.0e-2$
  - character types, i.e. `"c"`, `"t"`
  - string types, i.e. `"Lecture"`
- `python` is a **dynamically-typed language**, i.e. we can assign any type to a variable
- what if we need more **complex structures to store data**?
  - list of numbers
  - all sentences of a book
  - mapping from numbers to text
  - queue of jobs to be executed in sequence
- `python` standard library provides **complex data types** that can hold list, sequences, dictionaries of values

```
import time

def main():
  for i in range(5):
    text = "Hello World!"
    print(text)
    text = 42
    print(text)
  sleep(5)
```

Notes:

Python Lists

- `python` **lists** can hold **ordered sequence of elements** of any type

adding / removing eleemnts

- append allows to **append additional values** at end of list
- pop can be used to **remove and return element** at a given index (or at the end)
- `remove` **deletes first occurrence of a value**

- we can use zero-based **integer indexing** to read/write elements at specific position
- **slicing operator** `[start:end:step]` can be used to return new list with selected elements
- using append and pop(0) **we can use list as queue**, where elements are returned in fist-in-first-out (FIFO) order

```
# create list
l = [2, 42, 120, 18, 420]

# adding/removing elements
l.append('hello')
l.remove(120)
print(l.pop())
print(l.pop(0))

# index-based access
l[1] = 43

# elements up to index 2
# (excluding 2)
print(l[:2])

# elements starting from index 1
# (including 1)
print(l[1:])

# initialize list with 42 zero entries
l = [0]*42
```

Notes:

Python Tuples

- by appending, assigning or removing elements, `python` **lists can dynamically change their size** and **elements can change** during lifetime of list
- requires complex implementation that makes some operations relatively slow
- for **fixed-size ordered sequences that cannot change**, we can use `python` tuples
- **indexing and slicing** works the same as for lists
- elements cannot change and size of tuple cannot grow or shrink
- we can use + operator to **concatenate two tuples**, returning a new tuple

```python
# create tuple
t = (2, 42, 120, 189, 420)

# index-based access
print(t[1])

# slicing
print(t[:2])
print(t[1:])

# NOT VALID
t[1] = 43

# returns new tuple with additional
# elements
t2 = t + (4,5,6)
print(t2)
```

Notes:

Python Sets

- `lists` and `tuples` are **ordered sequences**
- checking whether an **element is in a list/tuple** requires to **test all elements** → naive linear search
- for **unordered collection of objects without duplicates** we can use `python set`
- useful to **eliminate duplicate elements** and **quickly test for membership**
- **python sets** are unordered and thus **do not support indexing or slicing**

```python
# create set
s = {2, 'hello', 120, 42, 189, 420}
print(s)

# check membership
print('hello' in s)

# add element
s.add(32)

# remove element
s.remove('hello')

# NOT VALID
s[1]
s[:4]
```

Notes:

Python Dictionaries

- we often need an **associative mapping** that maps **unique keys** to values
  - string/number as unique identifier (key)
  - arbitrary data of record (value)
- useful to **quickly find data** that are stored under a given key
- we can read/write entries using **index syntax** similar to lists
- but: index does not need to be an integer
- **reverse lookup** (i.e. find key(s) for a given value) not supported

```python
# create dictionary
d = { 'hello': 'Hallo',
      'world': 'Welt',
      'teacher': ['Ingo', 'Scholtes']
    }

# check membership of key
print('hello' in d)

# access value of given key
print(d['hello'], d['teacher'])

# assign value to (new) key
d['audience'] = 'Studierende'
```

Notes:

Practice Session

- we show how to install the Open Source python **distribution Anaconda**
- we use the **integrated development environment** (IDE) Visual Studio Code to write and execute a simple python program
- we use VS Code to **rename variables and refactor code**
- we use the **debugger of Visual Studio Code** for a step-wise execution of python statements
- we demonstrate lists, tuples, sets, and dictionaries in python

```
import time

def main():
    text = "Hello World!"
    print(text)
    text = 42
    print(text)
    sleep(5)
```

practice session

see directory 05-01 in gitlab repository at

→ https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks

Notes:

What is an algorithm?

$$\begin{array}{cccc} 1 & 2 & 5 & 7 \\ 2_1 & 9 & 3 & 2 \\ \hline 4 & 1 & 8 & 9 \end{array}$$

definition → L01 - Motivation

An **algorithm** is a sequence of precisely defined (mathematical) instructions that must be executed to solve a given problem.

- algorithm takes a (possibly empty) **input** and produces – after a **finite number of steps** – a desired **output**
- expressing an algorithm in terms of a **programming language** allows us to **implement** it on a computer

Example: pecil-and-paper algorithm to add two numbers

step 1 start at right-most position
step 2 add digits at current position
step 3 write last digit of sum below current position
step 4 for sums ≥ 10 additionally **carry over** 1 to position on the left
step 5 move one position to left and go to step 2

Notes:

Group Exercise 05-01

- Assume that we want to implement the **pen-and-pencil algorithm to add two decimal numbers with an arbitrary number of digits**. Specify a reasonable input and output of this algorithm.

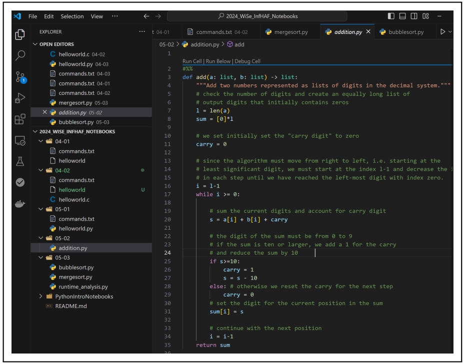
- Develop a `python` function `add` that implements the pen-and-pencil algorithm, using control structures like `while`, `for`, `if` as well as a `python` `list`

Notes:

- In lecture L02 we have seen how we can use digital logics to implement the addition of two binary numbers in terms of hardware. Thanks to the fact that this operation is implemented in the ALU of the CPU, we can directly add two 32 or 64 bit numbers by a single machine instruction (e.g. `ADD`). This implies that we can use the ADD operator + in high-level languages like python, which is directly mapped to this machine instruction.
- As an exercise, we pretend that there was no such operation that allows to add numbers with more than one digit. Let us implement the **pen-and-pencil algorithm to add decimal numbers** in python.
- Note that such an algorithm can still be useful if we want to add two numbers that cannot be represented by 64 bits or less, which may not be supported by the ALU of a common CPUs.

Practice Session

- we use lists to **implement the algorithm** developed in the previous group exercise in `python`
- we **test our algorithm** with different inputs



practice session

see directory 05-02 in `gitlab` repository at

→ `https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks`

Notes:

Search problems

- we are frequently confronted with **standard problems** that can be solved by well-understood **standard algorithms**
- exemplary standard problem: search for an object

search problems

Search problems are a class of problems that seek to quickly **find a given object** within a certain **data structure**.

- examples
  - problem 1: search name "Turing" in arbitrary list of 10,000 names
  - problem 2: search name "Turing" in a phonebook
- optimal solution to the search problem depends on **prior knowledge on the data structure**

image credit: DALL-E generated image, prompt "needle in a haystack"

Notes:

Searching in sorted data

- we can **sort a list of objects** whenever for pair of objects a and b we can determine whether $a \geq b$
- we assume that we search object x in a **list sorted in ascending order**, i.e. list l where for index i we have

$$l[i+1] \geq l[i]$$

- **naive algorithm** checks $x == l[i]$ for index $i = 0, 1, \dots$

binary search algorithm

- test if x is larger/smaller/equal than middle element c
- if $x == c$ return object
- if $x > c$ repeat search in elements **right of** c
- if $x < c$ repeat search in elements **left of** c

- binary search is example for **divide-and-conquer algorithm**
- both algorithms give correct result, but **which one is "better"?**

searching for $x = 17$

sorted list

2	3	5	11	17	23	47

step one

2	3	5	**11**	17	23	47
-	-	-	-	17	23	47

step two

-	-	-	-	17	**23**	47
-	-	-	-	17	-	-

step three

-	-	-	-	**17**	-	-

found $x = 17$!

Notes:

Complexity of binary search

we can evaluate algorithms in terms of **computational complexity**, i.e. we count how many steps they **maximally require** to produce the correct output for a given input?

example 1: how many steps do we need in list l with 32 **objects**

naive (linear) search algorithm

step	tested element
1	0
2	1
3	2
...	...
32	31

requires 32 **steps for** 32 **objects**

binary search algorithm

step	tested element
1	16
2	8 or 24
3	4, 12, 20, or 28
4	2, 6, 10, 14, 18, 22, 26, or 30
5	0, 1, 3, 5, 7, 8, 9, 11, 13, 15, 16, 17, ... or 31

requires 5 **steps for** 32 **objects**

Notes:

Complexity of binary search

we can evaluate algorithms in terms of **computational complexity**, i.e. we count how many steps they **maximally require** to produce the correct output for a given input?

example 2: how many steps do we need in list l with 64 **objects**

naive (linear) search algorithm

step	tested element
1	0
2	1
3	2
4	3
...	...
63	64

binary search algorithm

step	tested element
1	32
2	16 or 48
3	8, 24, 40, or 56
4	4, 12, 20, 28, ... or 60
5	2, 6, 10, 14, 18, 22, ... or 62
6	0, 1, 3, 5, 7, 9, 11, 13, 15 ... or 63

requires 64 **steps for** 64 **objects**

requires 6 **steps for** 64 **objects**

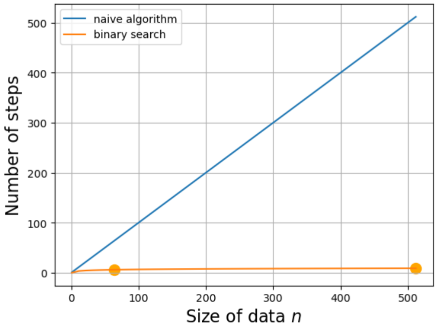
Notes:

Linear vs. logarithmic complexity

- naive search algorithm requires one step for each each entry in the input list, i.e. runtime is **proportional to the input size**
- we say an algorithm has **linear complexity** if for input with size n it requires at most

$$c + x \cdot n$$

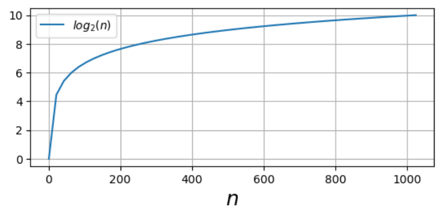
  steps for some numbers c and x
- thanks to sorted input, **binary search algorithm** requires **less than linear** number of steps
- how does number of steps grow as we increase input size n?



Notes:

Logarithms

Logarithm



For a number n, the logarithm $log_b(n)$ with base b of n is the number x such that $b^x = n$.

logarithms base $b = 2$

n	binary number	$log_2(n)$
$2^0 = 1$	1	0
$2^1 = 2$	10	1
$2^2 = 4$	100	2
$2^3 = 8$	1000	3
$2^4 = 16$	10000	4

logarithms base $b = 10$

n	$log_{10}(n)$
$10^0 = 1$	0
$10^1 = 10$	1
$10^2 = 100$	2
$10^3 = 1000$	3
$10^4 = 10000$	4

Notes:

- It is easy to see that $log_b(n)$ increases proportionally with the number of digits of n in a b-nary numeral system

Complexity of search algorithms

we can evaluate algorithms in terms of **computational complexity**, i.e. we count how many steps they **maximally require** to produce the correct output for a given **input of size** n?

how many steps do we need in list l with $n = 64$ **objects**

naive (linear) search algorithm

step	tested element
1	0
2	1
3	2
4	3
...	...
64	63

binary search algorithm

step	tested element
1	32
2	16 or 48
3	8, 24, 40, or 56
4	4, 12, 20, 28, ... or 60
5	2, 6, 10, 14, 18, 22, ... or 62
6	0, 1, 3, 5, 7, 9, 11, 13, 15 ... or 63

requires n **steps for** n **objects**

requires $log_2 n$ **steps for** n **objects**

Notes:

Sorting problem

- for binary search, we assumed that the list of objects is **sorted**
- to sort objects we must be able to **compare them**, i.e. for each pair a, b we must be able to determine $a \geq b$
- how can we compare pairs of
  - numbers,
  - words,
  - books,
  - emojis?

sorting problem

The **sorting problem** refers to the problem of sorting a list of **pairwise comparable objects** in ascending or descending order.

- in the following, we consider the sorting problem for a list of **integer numbers**

input:

7	2	47	23	5	11

desired output:

2	5	7	11	23	47

Notes:

BubbleSort algorithm

- simple idea: repeatedly **compare pairs** of numbers and **swap them** if they are in the wrong order
- with each swap ...
  - larger numbers progressively move to right
  - smaller numbers progressively move to left
- in each pass of the algorithm, we must compare **all subsequent pairs** of numbers in the list
- if we have zero swaps during a pass, we know that the list is sorted!
- in the example, we needed
  - $4 \cdot 5 = 20$ comparisons
  - $4 + 2 + 1 = 7$ swaps
- how many comparisons do we need in **best/worst case**?

third pass

2	**7**	5	11	23	47
2	7	5	11	23	47

2	**7**	**5**	11	23	47
2	5	7	11	23	47

2	5	**7**	**11**	23	47
2	5	7	11	23	47

2	5	7	**11**	**23**	47
2	5	7	11	23	47

2	5	7	11	**23**	**47**
2	5	7	11	23	47

5 comparisons, 1 swap

Notes:

Worst-case complexity of BubbleSort

worst-case runtime

For an input **list sorted in reverse order** `BubbleSort` algorithm requires n passes with $n-1$ comparisons each.

best-case runtime

For an input **list that is already sorted** `BubbleSort` algorithm requires a single pass with $n-1$ comparisons.

47	23	11	7	5	2

$$n = 6$$

$$n \cdot (n-1) = 6 \cdot 5 = 30 \text{ comparisons}$$

2	7	5	11	23	47

$$n = 6$$

$$n - 1 = 5 \text{ comparisons}$$

Notes:

Linear vs. polynomial complexity

- for input list with n elements, BubbleSort has **worst-case runtime** of

$$n \cdot (n-1) = n^2 - n$$

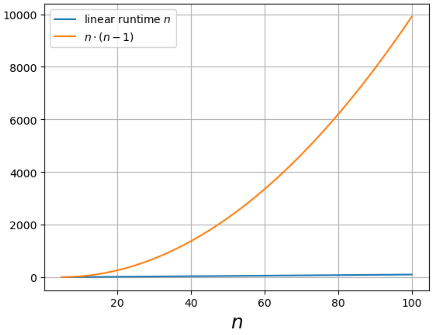
  i.e. number of required steps grows as **second power (i.e. square) of input size n**

- we call expressions of the form

$$a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots a_0 \cdot n^0$$

  polynomial

- for polynomials with power larger than one, runtime **grows over-proportionally** with input size

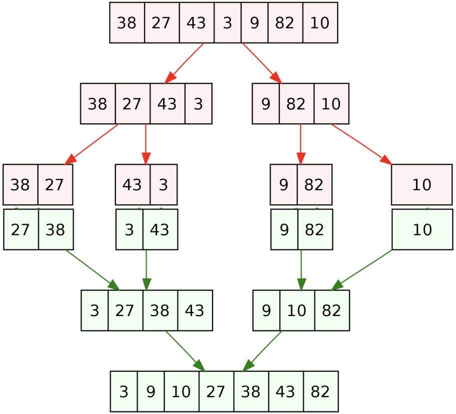


linear vs. polynomial growth of complexity

Notes:

MergeSort

- can we **sort a list faster** than `BubbleSort`?
- assume that we have **two already sorted lists** l_1 **and** l_2
- in $n = n_1 + n_2$ steps we can **merge** l_1 **and** l_2 into a new sorted list l
- we can apply **divide-and-conquer idea** behind binary search to sorting
- phase 1: repeatedly **split** input until we are left with lists with one element (which are already sorted)
- phase 2: repeatedly **merge** increasingly large (sorted) lists until full list is sorted



Notes:

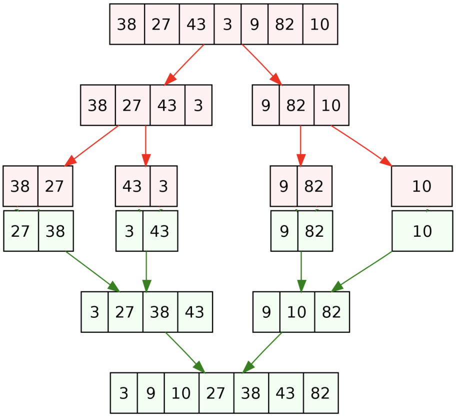
Complexity of MergeSort

- assume that for a **list with n elements** `MergeSort` takes $T(n)$ steps
- each split/merge then requires

$$2 \cdot T(\frac{n}{2}) + n$$

- starting with $T(1) = 0$ we have
  - $T(2) = 2 \cdot T(1) + 2 = 2$
  - $T(4) = 2 \cdot T(2) + 4 = 2 \cdot 2 + 4 = 8$
  - $T(8) = 2 \cdot T(4) + 8 = 2 \cdot 8 + 8 = 24$
  - $T(16) = 2 \cdot T(8) + 16 = 2 \cdot 24 + 16 = 64$
  - $\dots$
- we can calculate runtime of `MergeSort` as

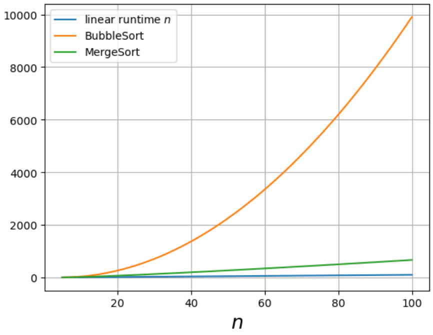
$$T(n) \approx n \cdot \log_2(n)$$



Notes:

Complexity of sorting?

- with `BubbleSort` we can sort n numbers in $n - 1$ steps in best case and $n \cdot (n - 1)$ in worst case
- `MergeSort` improves worst-case complexity of `BubbleSort` from n^2 to $n \log_2(n)$
- **on average** `MergeSort` requires $n \log_2(n)$ steps
- to sort n objects based on **pairwise comparisons**, there is **no algorithm exist that requires less than $n \log_2(n)$ steps on average**
- but: there are specialized algorithms to **sort n integer numbers in a fixed range** with linear runtime → self-study questions

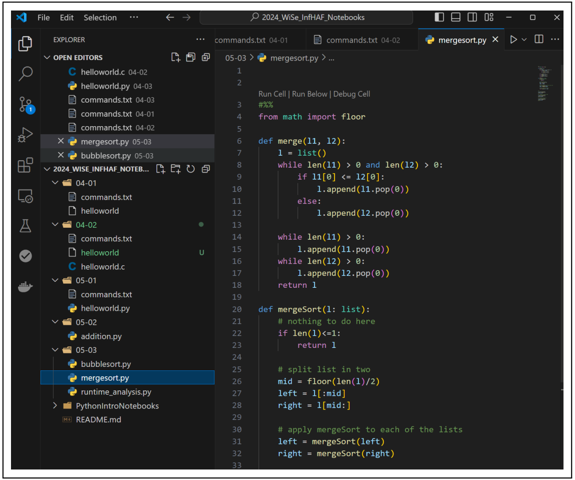


worst-case complexity of `MergeSort` vs. `BubbleSort`

Notes:

Practice Session

- we implement `BubbleSort` in `python`
- we implement the divide-and-conquer method `MergeSort`
- we study the **runtime of both algorithms** in increasingly large input lists



practice session

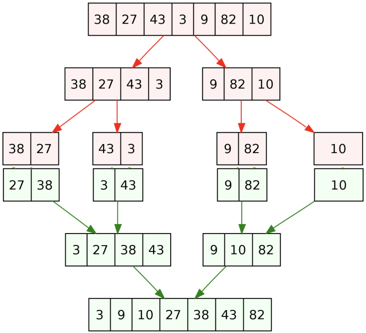
see directory 05-03 in `gitlab` repository at

→ `https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks`

Notes:

In summary

- we covered basic **python data structures** like sets, tuples, lists, and dictionaries
- we introduced **basic algorithms for standard computational problems** like searching and sorting
- we evaluated the **computational complexity** of sort and search algorithms
- we highlighted the difference between **logarithmic, linear, and polynomial runtime**



Notes:

Self-study questions

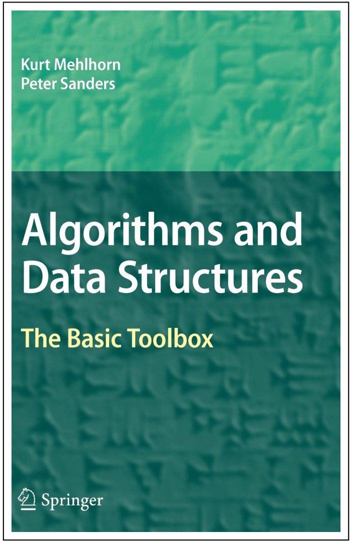
1. Explain the differences between a `set`, a `tuple`, a `dictionary` and a `list` in python.
2. Give a formulation of the Pen-And-Pencil algorithm to add two numbers in python and explain it in your own words.
3. Extend the Pen-And-Pencil algorithm from the group exercise such that it can add numbers given as sequences of digits in an arbitrary k-nary numeral system.
4. Give a formulation of the Binary Search algorithm in python and explain it in your own words.
5. Could we generalize the Binary Search algorithm such that in each step we split the list into three equally large parts, which would lead to a runtime $log_3(n)$?
6. Give a formulation of the BubbleSort algorithm in python and explain it in your own words.
7. Give an example for an input for which the BubbleSort algorithm performs the maximum/minimum number of comparisons.
8. Count the number of swaps in an input list with n elements, where BubbleSort performs the maximum number of comparisons.
9. Give a formulation of the MergeSort algorithm in python and explain it in your own words.
10. Investigate the BucketSort algorithm for integers in a fixed range and explain why it takes less than $n \log_2 n$ steps on average.

Notes:

Literature

References

- K Mehlhorn, P Sanders: **Algorithms and Data Structures - The Basic Toolbox**, Springer, 2008
- TH Cormen, CE Leiserson, RL Rivest, C Stein: **Introduction to Algorithms**, MIT Press, 2001
- F Kaefer, P Kaefer: **Introduction to Python Programming for Business and Social Science Applications**, SAGE Publications, 2020
- DE Knuth: **The Art of Computer Programming. Vol. 3: Sorting and Searching**, Addison-Wesley, 1998
- EH Friend: **Sorting on Electronic Computer Systems**, Journal of the ACM, Vol. 3, 1956



Notes: