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

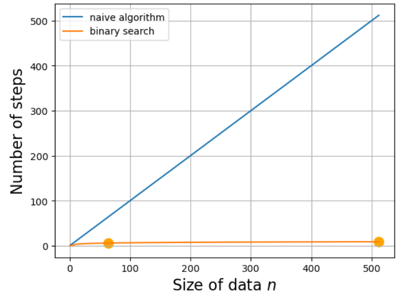
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Lecture 06
Algorithmic Thinking II

November 26, 2024

Motivation

- we considered two simple examples that illustrate how we can **use algorithms to solve problems**
- we reconsider the pencil-and-paper addition algorithm and implemented it in `python`
- we motivated the search problem and showed how to address it with the **binary search algorithm**
- we showed how we can compare the efficiency of algorithms in terms of their **computational complexity**
- today we continue our introduction to **algorithmic thinking** by addressing the following questions



computational complexity of **naive search algorithm** vs. **binary search algorithm** in list with size n

today's agenda
- how can we **efficiently sort data**?
- how can we **encrypt data** and ensure **secure communication**?
- can we find an **efficient algorithm for any problem**?

Notes:

- **Lecture L06: Algorithmic Thinking II** 26.11.2024
- **Educational objective:** We introduce two basic algorithms for the sorting problem and investigate the runtime of these algorithms. We further discuss basic encryption algorithms and introduce the concept of public-key cryptography.
  - The Sorting Problem
  - Recursion and MergeSort
  - Encryption Algorithms
  - Public-Key Cryptography
- **Exercise Sheet 4** due 03.12.2024

Notes:

Sorting problem

- **binary search algorithm** assumes list of objects is **sorted** in ascending (or descending) order
- to sort objects we must be able to **compare them**, i.e. for a pair a, b we must be able to determine $a \geq b$
- how can we compare pairs of
  - numbers,
  - words,
  - books,
  - emojis?

input:

7	2	47	23	5	11

desired output:

2	5	7	11	23	47

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**

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:

Notes:

Worst-case complexity of BubbleSort

worst-case example

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

best-case example

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

$$n = 6$$

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

We call the maximum runtime on any input the **worst-case time complexity** or **worst-case computational complexity** of an algorithm.

Linear vs. polynomial complexity

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

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

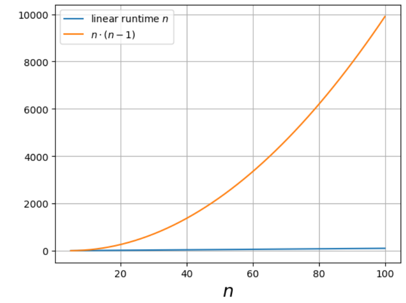
  i.e. number of required steps grows as **second power (i.e. square) of input size n**

- 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

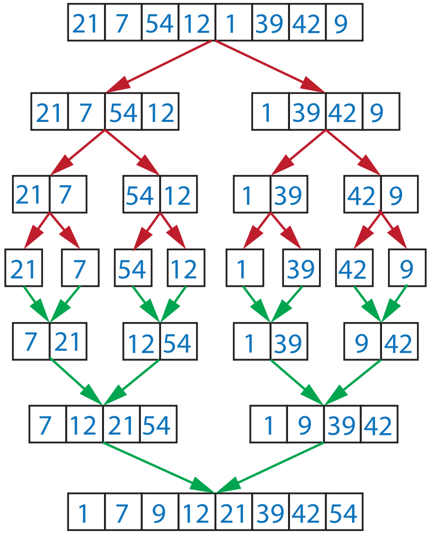
Notes:

- for runtime we typically use the term “time complexity” because there are other aspects of “computational complexity” such as the amount of memory (or disk space) required by an algorithm. This is often called “space complexity” and the term computational complexity then encompasses both time and space complexity.

Notes:

A divide-and-conquer algorithm

- can we **sort a list faster** than `BubbleSort`?
- assume that we have **two already sorted lists** l_1 **and** l_2 with n_1 and n_2 elements respectively
- 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



MergeSort algorithm

- how can we implement `MergeSort` in `python`
- assume we have a **function** `merge` that merges two sorted lists to a new sorted list
- **function** `merge_sort` that sorts a list l with n entries must perform the following steps ...
  1. split list in two equally large lists l1 and l2
  2. **call itself** on l1 and l2
  3. merge sorted lists to result
- concept of a function calling itself is called **recursion**
- recursion terminates when list only contains single (or zero) element (which we call **"base case"**)
- common way to apply **divide-and-conquer**, i.e. function calls itself on smaller problem instances

```python
def merge(l1: list, l2: list) -> list
  l = []
  ...
  return l

def merge_sort(l: list) -> list
  if len(l) < 2:
    return l
  mid = math.floor(len(l)/2)
  l1 = l[:mid]
  l2 = l[mid:]
  l1_s = merge_sort(l1)
  l2_s = merge_sort(l2)
  l_s = merge(l1_s, l2_s)
  return l_s
```

python implementation of recursive `MergeSort` algorithm

Notes:

Notes:

- the term recursion comes from the Latin word "recurro", which translates to "to come back"

Complexity of MergeSort?

- assume that `merge_sort` requires $T(n)$ steps for **list with n elements**
- in each step, we run `merge_sort` on **two lists with half the size** and then **merge results**
- each step thus requires

$$\underbrace{2 \cdot T(\frac{n}{2})}_{\text{recursive calls of merge\_sort}} + \underbrace{n}_{\text{merge lists}}$$

- how often do we need to split until we we are left with lists with single entry?
- like for binary search, we **need $\log_2(n)$ splits** for list with n entries

example: list with $n = 128$ entries

step	number of lists	length of each list
0	1	128
1	2	64
2	4	32
3	8	16
4	16	8
5	32	4
6	64	2
7	128	1

for $n = 128$ we need $7 = \log_2(128)$ steps

Complexity of MergeSort?

- we found **recursive formula** for runtime of `MergeSort`

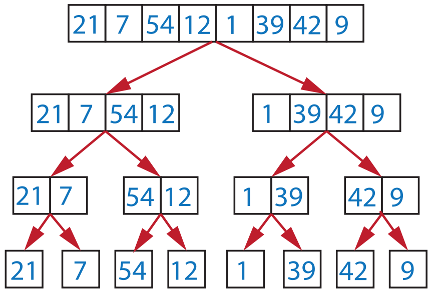
$$T(n) = 2 \cdot T(\frac{n}{2}) + n$$

- for $n = 1$ we do not split or merge, i.e. for **base case** we have

$$T(1) := 0$$

- we can now **recursively calculate** …
  - $T(2) = 2 \cdot T(1) + 2 = 2$
  - $T(4) = 2 \cdot T(2) + 4 = 2 \cdot 2 + 4 = 8$
  - $T(8) = 2 \cdot T(4) + 8 = 2 \cdot 8 + 8 = 24$
  - $T(16) = 2 \cdot T(8) + 16 = 2 \cdot 24 + 16 = 64$
  - $T(32) = 2 \cdot T(16) + 32 = 2 \cdot 64 + 32 = 160$
- one can **solve this recursive formula** as

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



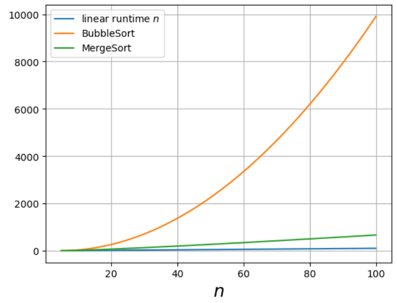
Notes:

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 that requires less than $n \log_2(n)$ steps on average**
- <u>but:</u> there are specialized algorithms to **sort n integer numbers within a known range** with linear runtime

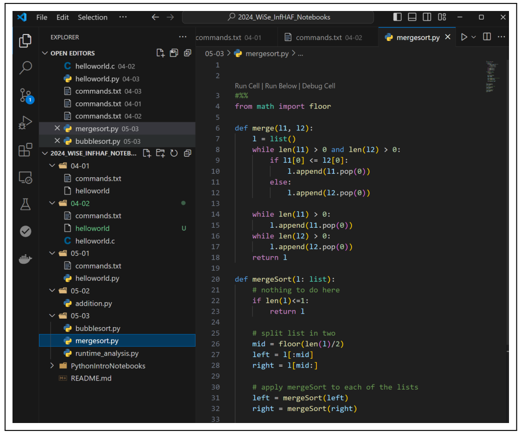
→ self-study questions



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

Practice Session

- we implement `BubbleSort` in `python`
- we implement the **recursivce** divide-and-conquer method `MergeSort`
- we investigate the **runtime of both algorithms** for increasingly large inputs



practice session

see directory 06-01 in `gitlab` repository at

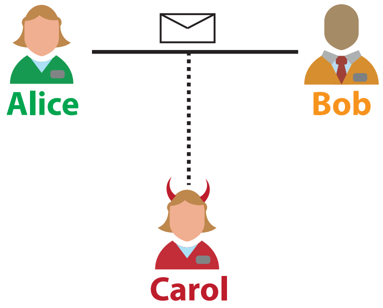
→ `https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks`

Notes:

Notes:

Cryptographic algorithms

- some of the **oldest algorithms in human history** are **cryptographic algorithms** that facilitate secure communication
- common scenario involves two persons **Alice (A)** and **Bob (B)** that wish to exchange messages via **public channel**
- **adversary Carol (C)** can intercept and/or manipulate messages
- how can Alice an Bob **securely communicate** despite adversary Carol?



Cryptography

Cryptography refers to the practice and study of **secure communication** in the presence of **adversarial** attacks. → Wikipedia

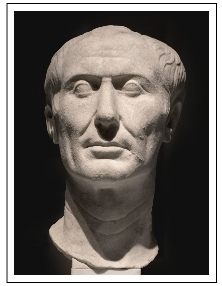
A simple encryption algorithm

- we assume that Alice and Bob prearranged a **shared secret key** (e.g. during a previous secret meeting)
- Alice uses key to **encrypt** message in **plaintext** and sends **ciphertext** to Bob
- Bob uses key to **decrypt ciphertext** and obtains **plaintext**
- any idea for a **simple encryption algorithm**?

Caesar cipher

- **secret key** is number k between 0 and 25 (alphabet with 26 letters)
- each letter in plaintext is replaced by letter found by shifting letter in alphabet k positions to right
- positions larger than 25 are wrapped around to zero

- simple **substitution cipher** allegedly used by Julius Caesar to send military commands (using $k = 3$)



Julius Caesar
100 BC – 44 BC

0	1	2	3	4	5	6	7	...
a	b	c	**d**	e	f	g	h	...
d	e	f	g	h	i	j	k	...

Caesar cipher with key $k = 3$

Notes:

Notes:

Caesar cipher in `python`

- assume that plaintext message is given as **list of characters** in python, e.g. `['h', 'e', 'l', 'l', 'o', ' ', 'c', 'a', 'e', 's', 'a', 'r']`
- simplest approach limited to 26 (lower or uppercase) letters in alphabet
- we can use Caesar cipher to shift **any character** based on its underlying binary encoding
- **use ASCII/Unicode table** to map characters to numbers and vice-versa, e.g. 'a' $\leftrightarrow$ 97
- in `python` we can use functions `ord` (map character to unicode index) and `chr` (map unicode index to character)

plaintext:

hello caesar

cipher text:

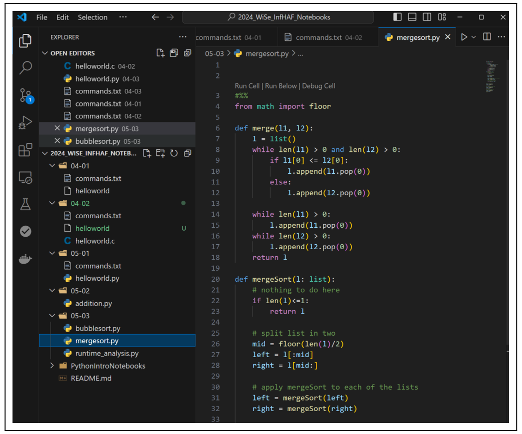
khoor fdhvdu

0	1	2	3	4	5	6	7	...
a	b	c	**d**	e	f	g	h	...
d	e	f	g	h	i	j	k	...

Caesar cipher with key $k = 3$

Practice Session

- we implement the Caesar cipher in `python`
- we use it to encrypt and decrypt a message
- we investigate the **security of the Caesar cipher** and perform a **brute-force attack on the key**



practice session

see directory 06-02 in `gitlab` repository at

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

Notes:

Notes:

A (slightly) better approach

- how many different keys do we have in the Caesar cipher?
  - for alphabet with length $l = 26$ we have 26 possible shifts
  - for 16-bit unicode we have $2^{16} = 65536$ possible shifts
  - takes less than 1s on common computer
- Caesar cipher is extremely easy to break by **brute-force attack** that simply tries each possible key
- <u>idea:</u> instead of limiting ourselves to shifts, we could apply **arbitrary substitution of characters**
- secret key = **substitution table** that gives substitution for each character in plaintext
- how efficient is a **brute force attack** now?

a	b	c	d	e	f	g	h	...
j	t	x	f	z	a	f	k	...

substitution table

Security of substitution ciphers

- how many different **permutations** do we have for `['a', 'b', 'c']`?
- for alphabet with n letters we have

$$n! := n \cdot (n-1) \cdot (n-2) \cdot \ldots 2 \cdot 1$$

  possible substitutions
- factorial $n!$ can be calculated in **recursive fashion**

$$n! = n \cdot (n-1)!$$

- for 26 letters and assuming we can test 1 billion keys per second, **brute-force attack may take 3.2 billion years**
- for $n = 65$, number $n!$ of possible keys is **larger than number of atoms in the universe**
- is this really secure?

n	$n!$
1	1
2	2
3	6
4	24
5	120
6	720
...	...
25	$1.55 \cdot 10^{25}$
26	$4.03 \cdot 10^{26}$

Notes:

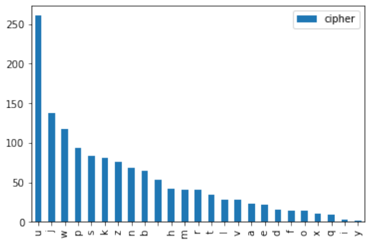
Notes:

- It is easy to see that each substitution is just a different sequence of the 26 letters of the alphabet.
- In order to understand in how many different ways we can substitute 26 letters of the alphabet, we thus have to consider so-called permutations, i.e. the possible ways in which we can arrange those letters.
- Let us consider this for an alphabet with three letters a, b, and c. For the first position, we have three choices (a, b, c). This choice can be combined with all choice for the second position, however since we already fixed the first one there are only two choices left. This gives a total of $3 \cdot 2$ for the first two positions. Fixing those two positions also fixes the last one (since we only have three letters), i.e. there is only one choice left and we thus have a total of $3 \cdot 2 \cdot 1 = 6$ possible ways in which we can arrange three letters.
- Using the same reasoning, for four letters we find $4 \cdot 3 \cdot 2 \cdot 1 = 24$ different permutations and for the general case of n letters we have $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$.
- This number is called the **factorial** $n!$ of n and it plays an important role in combinatorics. In the table above, you can see that the factorial grows extremely fast!
- Because of this fast growth, the number of possible keys is so large that a brute-force attack is not efficient. However, this still does not imply that this simple cipher is secure!

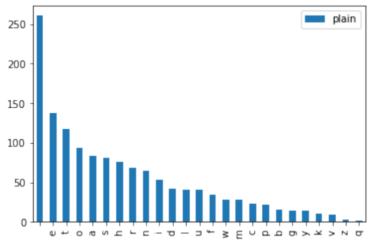
Crpytographic analysis

Carol receives the following cipher text encrypted using **substitution method with unknown key**

zxuyguxhuexzuzxuyguzvczukluzvgusjglzkxeupvgzvghuzkluexynghukeuzvguokeiuzxuljwwghuzvgu luceiuchhxpluxwuxjzhc gxjluwxhzjeguxhuzxuzcbgucholuc ckelzuculgcuxwuzhxjyngluceiuytuxd ugeiuzvgouzxuikguzxulnggduexuoxhguceiuytuculnggduzxulctupgug iuzvguvgchzucrvguceiuzvguzvx…



character frequencies in **cipher text**



character frequencies in **English texts**

apart from brute-force attacks, we can use **cryptanalysis** to reduce the space of possible keys

Vigenère cipher

- **monoalphabetic substitution** is easy to break using brute-force or frequency analysis
- idea: use **polyalphabetic substitution** that applies **multiple substitutions for same character**

example: key $k = 5237$

- plaintext message = "hello world"
- for first character apply Caesar shift with $k = 5$
- for second character apply Caesar shift with $k = 23$
- for third character apply Caesar shift with $k = 7$
- for fourth character apply Caesar shift with $k = 5 \dots$

- proposed by Blaise de Vigenère in 1585
- unbreakable if key is as long as the message (so-called **one-time pad**)

k	a	b	c	d	e	f	g	h	i	…
0	a	b	c	d	e	f	g	h	i	…
1	b	c	d	e	f	g	h	i	j	…
2	c	d	e	f	g	h	i	j	k	…
3	d	e	f	g	h	i	j	k	l	…
4	e	f	g	h	i	j	k	l	m	…
5	f	g	h	i	j	k	l	m	n	…
6	g	h	i	j	k	l	m	n	o	…
7	h	i	j	k	l	m	n	o	p	…
8	i	j	k	l	m	n	o	p	q	…
9	j	k	l	m	n	o	p	q	r	…

Vigenère table

Notes:

- By simply calculating the frequencies of letters in the cipher text and comparing them to the frequency of characters in the (known) language of the plaintext, it is often possible to guess a sufficient number of substitutions to actually decrypt the message. In any case, this reduces the space of possible keys so that we can break the encryption!
- In our example, it is immediately clear that the character t is the substitution of the most frequent character e.

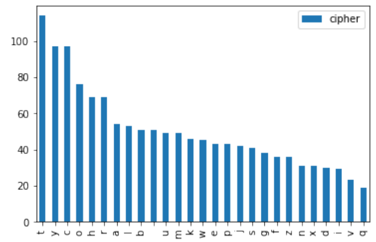
Notes:

- Their vulnerability to frequency analyses is a problem of all monoalphabetic substitution methods, i.e. methods that always replace one letter in the alphabet by another letter.
- We can solve this issue by polyalphabetic substitution methods that use a more sophisticated approach that replaces the same letter by different letters.
- The so-called Vigenere cipher is an example for such a method. It is based on a table, which lists all possible Caesar shifts. The idea is then to use a different shift for each character, depending on its position. Which shift is used for a given character is determined based on a key, which gives the sequence of shift values that are simply repeated if the message is longer than the key.
- You can see that the Caesar cipher is just a special case of the Vigenere cipher where the key contains only a single number (which is thus applied to all characters).
- In fact, whenever we repeat the key because the message is longer than the key, we use the same mapping again, which can again make the method vulnerable to frequency analysis if the message is much longer than the key. The same holds if we reuse the same key for multiple messages!
- However, if the key is as long as the message and we use the key only once, the Vigenere method is unbreakable (and it is thus still used today in the so-called one-time pad encryption).

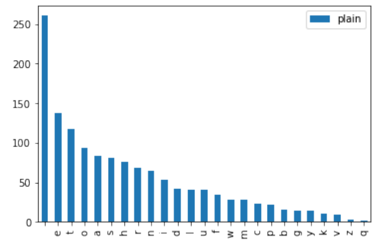
Frequency analyis?

Carol receives the following cipher text encrypted using **Vigenère encryption with unknown key**

whoveyyutbhtycrtqy rrdmobsyckyojucbwbcg urhmwyrycllogo vhkobnyckyofilncmctsspiyfttfocl bnebcubx mpchimrzqhhil dyumigeyyuthh rknyourkbcuvuilbwtptsckchuttpyxv ysykqxovyyysiclilqcybx …



character frequencies in **cipher text**

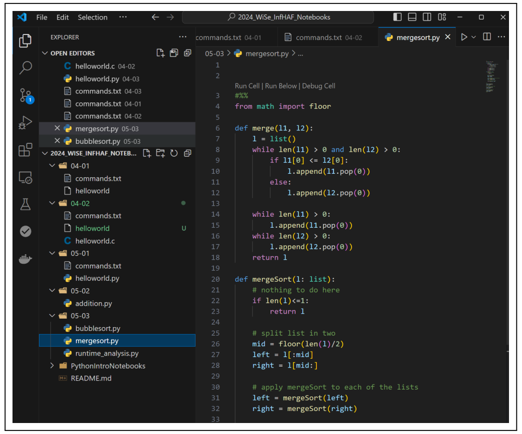


character frequencies in **English texts**

depending on key length, polyalphabetic substitution hinders application of cryptanalysis

Practice Session

- we implement the substition and Vigenere cipher in `python`
- we use it to encrypt and decrypt a message



practice session

see directory 06-03 in `gitlab` repository at

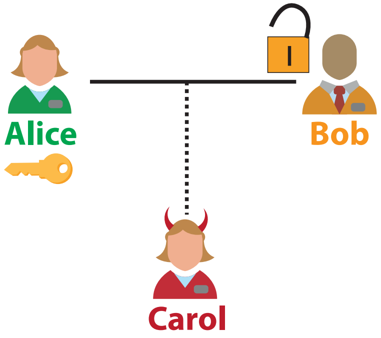
→ https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks

Notes:

Notes:

Public-key cryptography

- so far we assumed that Alice and Bob use **symmetric encryption methods** that require a **shared secret key**
- is this practical (e.g. on the Internet)?
- **public key cryptography** refers to **asymmetric methods** that do not require shared secret key
- consider analogy where Alice and Bob use **physical key and lock** to securely communicate



idea

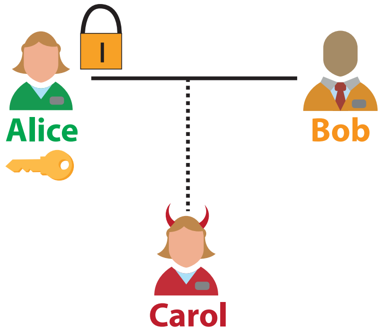
- Alice has access to locks and a corresponding **key**
- Alice sends **open locks** to any communication partner
- Bob **uses lock on a box**, and sends locked box with message to Alice
- Alice **uses her key to open locked box** and reads message

- lock = **public key** used to “encrypt message”
- key = **private key** used to “decrypt message”

A public-key encryption algorithm?

- public key uses **one-way function** that is easy to apply (i.e. close lock) but difficult to reverse (i.e. open lock)
- consider **prime factorization of a number**
- given prime numbers 79 and 37 it is **easy to compute product** $79 \cdot 37 = 2923$
- but: given number 2923 it is **difficult to find (unique) prime factors** 79 and 37
- basis for **RSA algorithm** that uses pair of prime numbers p, q as private key and $n = pq$ as public key

→ Rivest, Shamir and Adleman, 1978



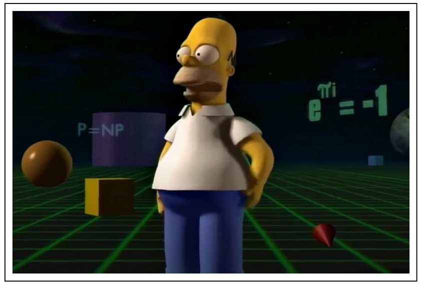
Notes:

- importantly, different from shared keys of symmetric encryption algorithms the public key can be made available publicly (i.e. posting it on a website) as it can only be used to encrypt messages, not to decrypt messages
- anyone who wants to securely communicate with Alice, just obtains the public key and uses it to encrypt a message to her. Only Alice can decrypt these messages using her private key.

Notes:

Hard problems in computer science

- RSA algorithm relies on the fact that it is **difficult** to find prime factors of a large number
- but who can guarantee that noone will find an efficient algorithm (thus breaking RSA) in the future?
- prime factorization belongs to so-called **NP complexity class** of problems
- there are **no known algorithms** that can solve an NP problem faster than trying all possibilities (brute force)
- unclear whether we have not discovered such algorithms yet or whether they do not exist
- **algorithms for quantum computers** can efficiently solve some problems in NP and thus pose a threat for cryptographic methods → post-quantum cryptography



Homer in 3D

image credit: Screenshot from The Simpsons, Matt Groening & 20th Television, fair use

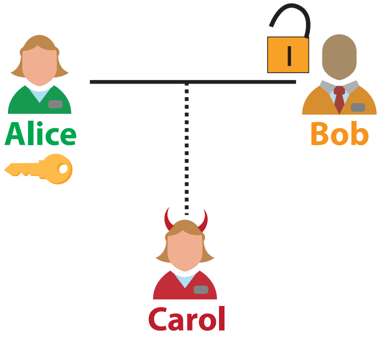
Digital signatures

- apart from encryption we often want to **verify the authenticity** of messages or data

use cases

- receiving an E-Mail from your friend
- visiting the website of your bank
- downloading a software from the Web

- how can Bob verify that received message is actually from Alice?
- idea: use private/public keys in inverse fashion
  1. Alice uses her **private key to encrypt message**
  2. Bob uses Alice's **public key to decrypt message**
  3. successful decryption proves that message is actually from Alice
- method/algorithm to verify authenticity of messages is called **digital signature**



Notes:

- We commonly refer to the class of problems that can be solved in polynomial time (e.g. quadratic, cubic, etc.) as P. In contrast, those problems for which the best we can do is to verify a known solution in polynomial time fall in the class of NP.
- Whether these two classes are actually identical (i.e. $P = NP$), that is whether we can solve problems in NP in polynomial time is the biggest open question in computer science. The consequences of $P = NP$ would be so drastic (i.e. we would suddenly be able to efficiently solve incredibly hard problems), that most computer scientists believe that the two classes are not equal.

Notes:

History of public-key cryptography

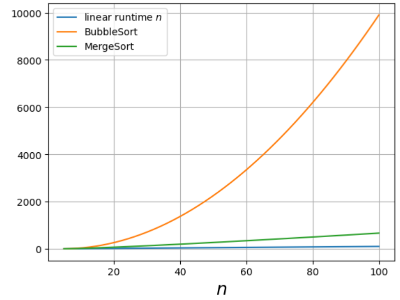
- until 1970s: government agencies (i.e. secret services) held monopoly on
- first publicly proposed public-key cryptographic algorithm was developed by **Diffie and Hellman** in 1976
- in 1991 Phil Zimmermann publicly released software **Pretty Good Privacy (PGP)**, which implemented RSA algorithm
- resulted in investigation by US Customs service for violation of **Arms Export Control Act**
- protocols using public-key cryptography have since become **key infrastructure for secure Internet communication** (e.g. SSL/TLS, S/MIME)

Phil Zimmermann
born 1954

image credit: User Beao, Wikimedia Commons, CC-SA

In summary

- we introduced two fundamental algorithms to address the **sorting problem**
- we showed how we can implement the divide-and-conquer algorithm **MergeSort** in a recursive fashion
- we introduced basic **symmetric encryption algorithms** and discussed their limitations
- we explained principles behind **asymmetric public-key cryptography** which can be used to **encrypt and authenticate** messages
- we highlighted that cryptographic methods often rely on **known hard problems in computer science**



Notes:

Notes:

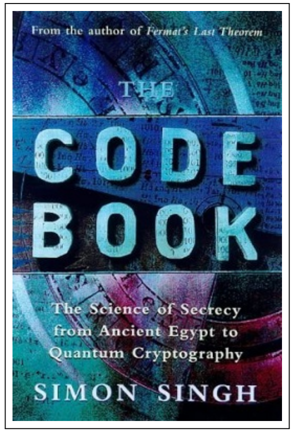
Self-study questions

1. Give a formulation of the BubbleSort algorithm in python and explain it in your own words.
2. Give an example for an input for which the BubbleSort algorithm performs the maximum/minimum number of comparisons.
3. Count the number of swaps in an input list with n elements, where BubbleSort performs the maximum number of comparisons.
4. Investigate the BucketSort algorithm for integers in a fixed range and explain why it takes less than $n \log_2 n$ steps on average.
5. Give a formulation of the MergeSort algorithm in python and explain it in your own words.
6. Explain why monoalphabetic substitution methods like the Caesar cipher are not secure.
7. Explain why we can - in general - not use frequency analyses to break the Vigenère cipher.
8. What is the difference between symmetric and asymmetric encryption algorithms?
9. Assuming you can test one billion keys per second, calculate how many years a brute-force attack can take to break a monoalphabetic substitution cipher with an alphabet of 26 characters.
10. Explain how we can use public-key cryptography to securely communicate without secretly exchanging a shared key.
11. Explain how public-key cryptography can be used to verify the authenticity of messages.

Further reading

References

- EH Friend: **Sorting on Electronic Computer Systems**, Journal of the ACM, Vol. 3, 1956
- W Diffie, ME Hellman: **New Directions in Cryptography**, IEEE Transactions on Information Theory, 1976
- R Rivest, A Shamir, L Adleman: **A Method for Obtaining Digital Signatures and Public-Key Cryptosystems**, 1978
- Simon Singh: **The Code Book**, Fourth Estate, 1999



Notes:

Notes: