MEIM MASTER IN ENTREPRENEURSHIP INNOVATION MANAGEMENT IN COLLABORATION WITH MIT SLOAN

IN COLLABORATION WITH MIT MANAGEMENT SLOAN SCHOOL

UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

MASTER MEIM 2022-2023

DIGITAL TECH

[High Performance] Computing

Lesson 1

Prof. Giulio Giunta

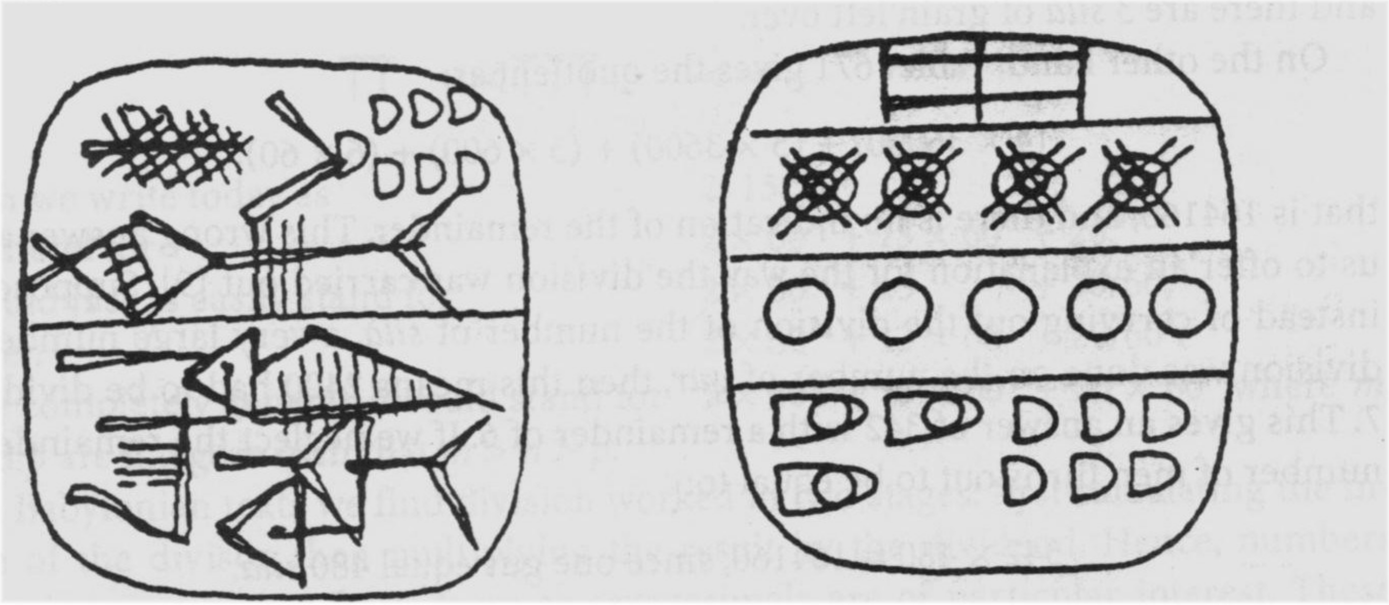
Prof. of Scientific Computing, Università degli Studi di Napoli Parthenope

Algorithms are old stuff

algorithm for dividing the amount of grain of a barn among several individuals:

input: capacity of granary C, predetermined part per person P

output: number N of people who can receive the portion of wheat: N = C/P



Sumerian tablet, Euphrates valley, 2500 B.C.

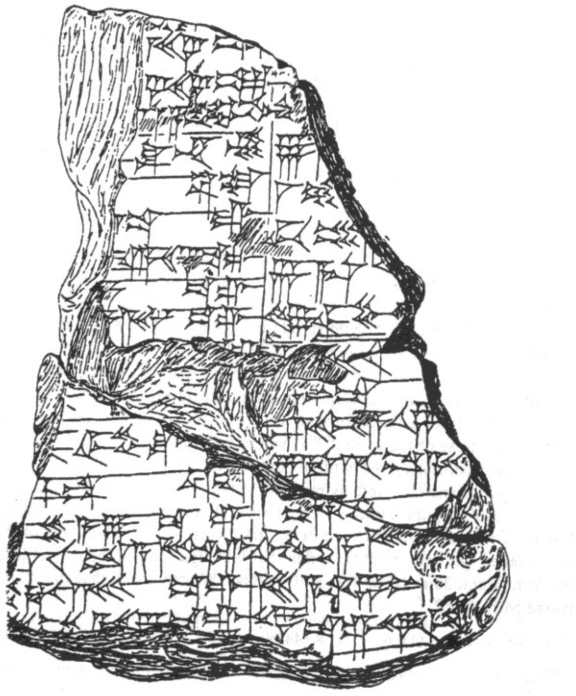
Algorithms are old stuff

algorithm for dividing the amount of grain of a barn among several individuals:

dividend / divisor

Step 1) calculation of the inverse (reciprocal) of the divisor

Step 2) multiplication by the dividend



Babylonian tablet, 1800 B.C.

Reflection on the speed of humanity's evolution

Indonesia, 40.000 BC

Chauvet Cave, 35.000 BC

Altamira Cave, 16.000 BC

Lascaux Cave, 14.000 BC

Sistine Chapel, 1.400 AD

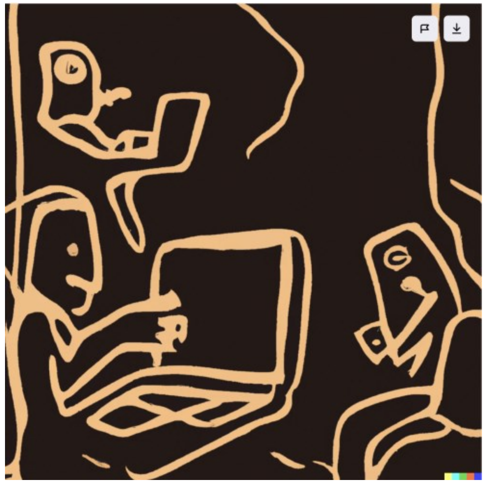
Reflection on the speed of humanity's evolution

Indonesia, 40.000 BC

Chauvet Cave, 35.000 BC

Altamira Cave, 16.000 BC

Lascaux Cave, 14.000 BC



DALL-E, today

Prompt: depict a computer and a programmer as a Lascaux cave painting

Algorithms are old stuff

A more complex algorithm: Euclid's algorithm (300 BC) for the greatest common divisor

```
def gcd(m,n):
 if m < n:
     (m,n) = (n,m)
 r = m
 while  r != 0:
     r = m % n
     (m, n) = (n, r)
 return n
```

Euclid, detail of the fresco The School of Athens, Raphael 1510

Algorithms are old stuff

A more complex algorithm: Euclid's algorithm (300 BC) for the greatest common divisor

```
def gcd(m,n):
    if m < n:
        (m,n) = (n,m)
    r = m % n
    if r == 0:
        return n
    else:
        return (gcd(n, r))
```

Euclid, detail of the fresco The School of Athens, Raphael 1510

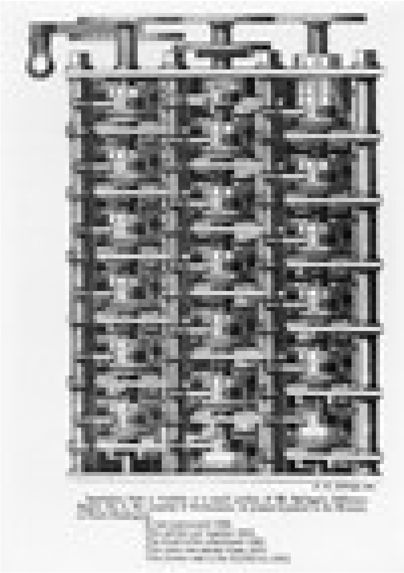
recursive version of Euclid's algorithm

Executing Algorithms

«It is not worthy of excellent men to waste hours as slaves in the manual activity of calculating, which could certainly be entrusted to a machine»

Gottfried von Leibniz
(Leipzig 1646 -1716)

Executing Algorithms



Difference Engine

Analytical Engine

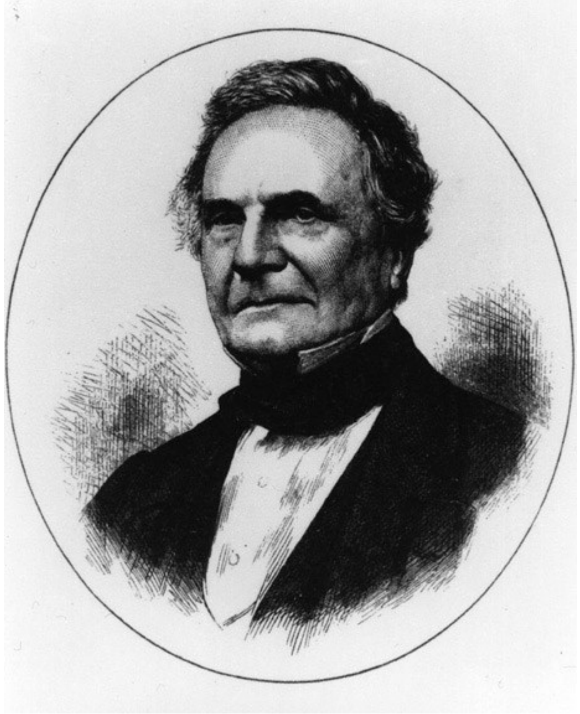
Charles Babbage
(London 1791-1871)

Executing Algorithms

a politician asks Babbage :

Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out

Babbage's answer :
I am not able to rightly apprehend the kind of confusion of ideas that could provoke such a question !



Charles Babbage
(London 1791-1871)

Charles Babbage Institute
Center for History of Information Technology
http://www.cbi.umn.edu/

Virginia Tech, USA
History of computing and virtual museum
http://ei.cs.vt.edu/~history

Executing Algorithms

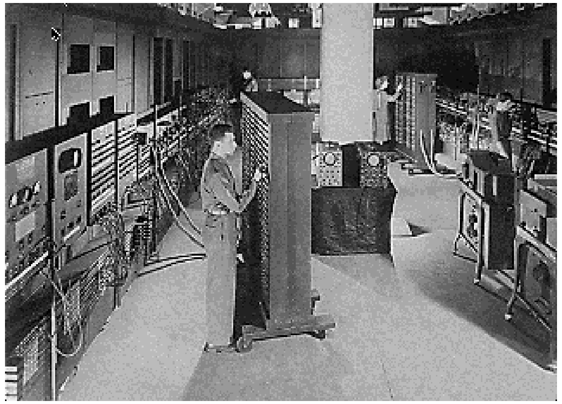
It is not necessary to have an infinite number of different machines available to perform different tasks. It is enough to have only one. The problems of producing various machines for different tasks turn into desk work, which consists of programming the universal machine to do those tasks. (A. Turing, 1940)

The machine must calculate, then it must contain an arithmetic central, which constitutes the first specific module.
The logical control of the machine, that is, the appropriate sequential frequency of its operations can be carried out by a central control module.
The machine must perform long sequences of operations, then it must have a considerable memory, which constitutes the third specific module.
The machine must also maintain an input - output contact with the outside.
The machine must have components to transfer information between the various modules.(J. Von Neumann, *I Draft of a Report on the EDVAC*, 1945)

John von Neumann
(Budapest 1903 - Washington 1957)

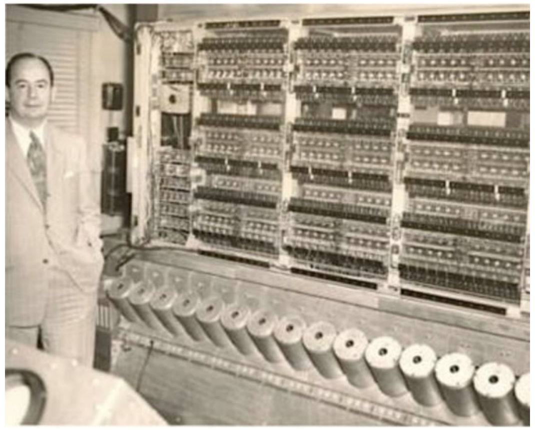
Alan Turing
(London 1912 –
Manchester 1954

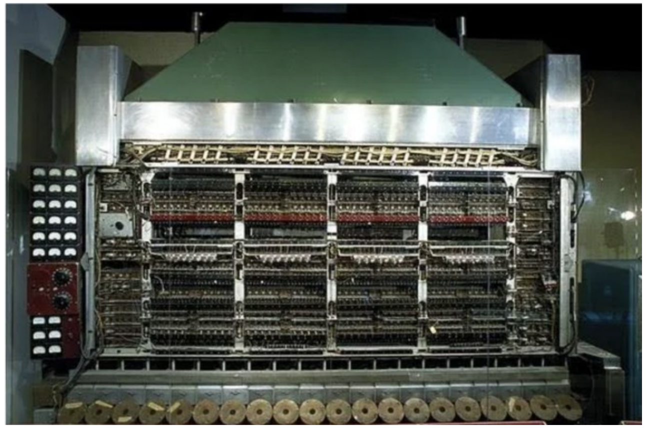
Executing Algorithms



ENIAC, the first electronic calculator 1943-45

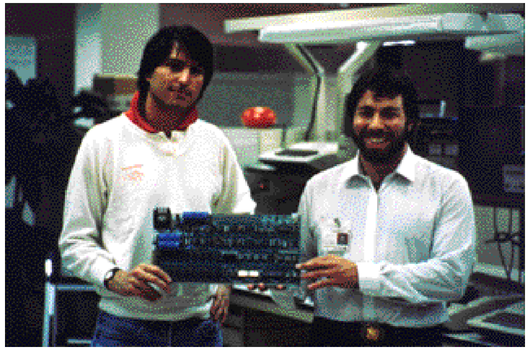
the term Computer meant the operators





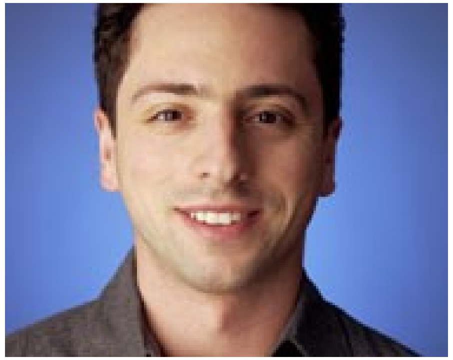
EDVAC, the first electronic computer 1946-49

Executing Algorithms and more



Steve Jobs and Steve Wozniack
1977 – the **pc** was born (Apple)

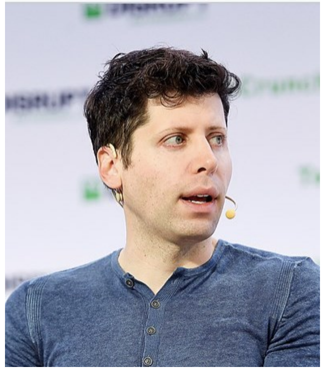
Tim Berners-Lee
1980 – the **web** was born



Larry Page and Sergey Brin
1998 – **Google** was born

Jeff Bezos
2003 – **Amazon** was born

Mark Zuckerberg
2004 – **Facebook** was born



Sam Altman
2022 – **ChatGPT** was born

A brief history of Computing

Inventing the computer

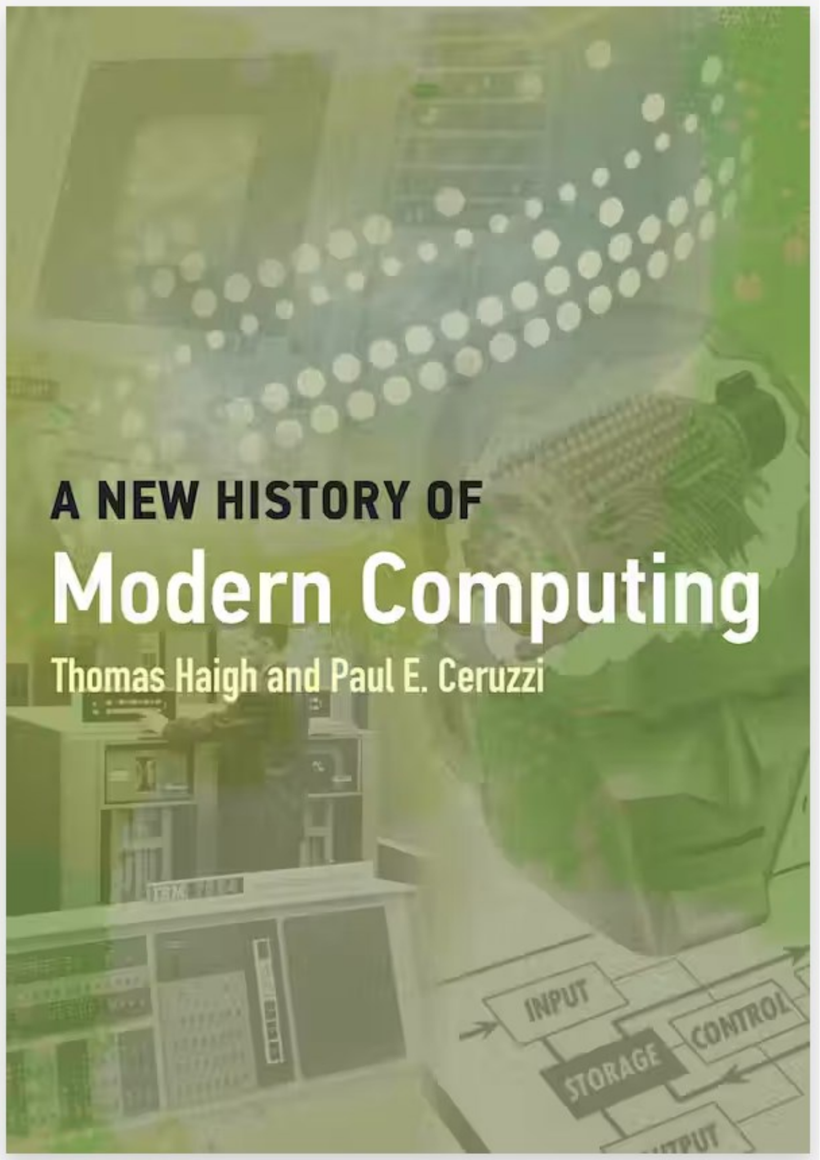
1946-1950 - New York Times: «Electronic computer flashes answers, may speed Engineering». ENIAC, EDVAC, programming tools, the Association for Computing Machinery, the commercialization of computing

The computer becomes a scientific supertool

1952-1971 - the first scientific computers, IBM 701, 704 and successors, floating point numbers, early compilers, Fortran, Algol, IBM 7070, CDC 6600, Cray 1

The computer becomes a data processing device

1951-1974 - the first administrative computers, IBM 700 big business computers, IBM's Big Hit: small computers, IBM 650, IBM 1401, sorting and report generation, Cobol, IBM 360, dreams of a management revolution, total integrated management system, database management systems, creation of the software industry



A brief history of Computing

The computer becomes a real-time control system

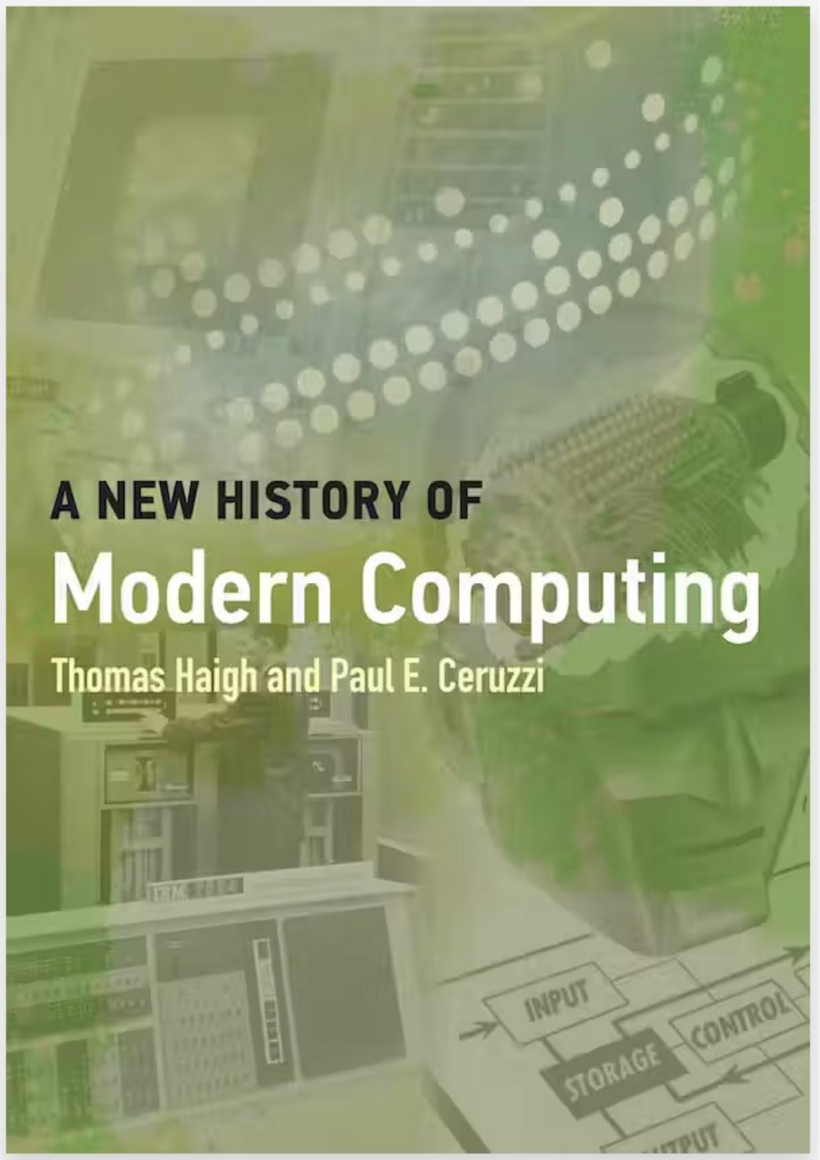
1950-1970 - the Cold War project at MIT, SAGE network, NASA mission control, miniaturization: missiles and minicomputers, Digital Equipment Corporation and PDP, integrated circuits, chips for Apollo, system and software reliability

The computer becomes an interactive tool

1961-1971 - Spacewar the first game on MIT's PDP-1, Timesharing operating systems, terminals, software engineering, the NATO conference on software engineering, Unix

The computer becomes a communication platform

1965-1981 - Communication and collaborating on timesharing systems, electronic mail at MIT, Plato and the beginning of computer aided instruction, ARPANET and packet switching, internetworking, TCP and IP, commercial networks and online services, Videotex and Minitel. Internet commercialization, Domain Name System



A brief history of Computing

The computer becomes a personal plaything

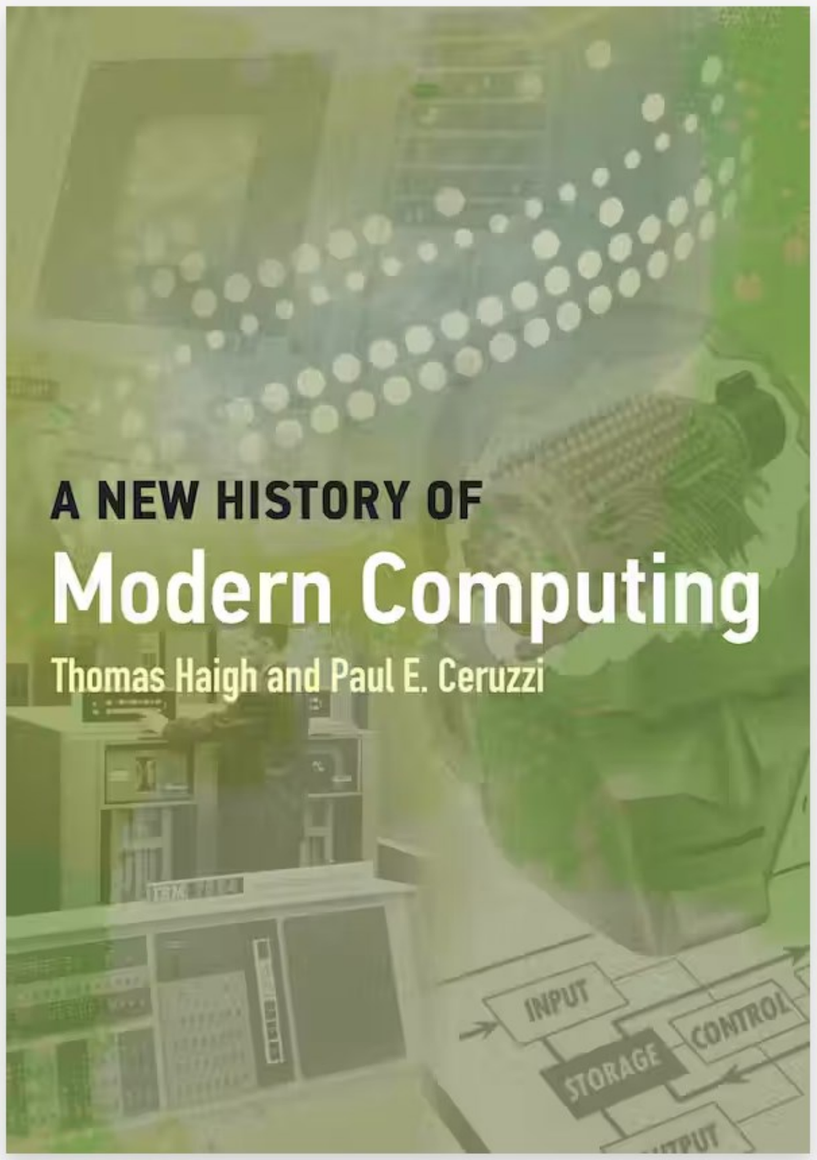
1975-1980 - electronic calculator, HP programmable pocket calculator, microprocessors and Gordon Moore, Altair 8800 the first personal (micro) computer, expandability and modularity, mass market personal computers, Commodore, Apple and Tandy, videogames, the Atari VCS console, Apple II, computers come home

The computer becomes office equipment

1981-1990 - IBM's pc, Apple II pc, personal computers for business, word processing, Visicalc and spreadsheet programs, PC-DOS, the IBM pc becomes an industry, MS-DOS, cloning the IBM pc, Ethernet for local area networks, portable pc

The computer becomes a graphical tool

1984-1995 - Apple MacIntosh, Graphical User Interface, Xerox invents graphical computing, Smalltalk and object-oriented programming, what-you-see-is-what-you-get, GUI on pc, Graphics workstations



A brief history of Computing

The computer becomes a minicomputer

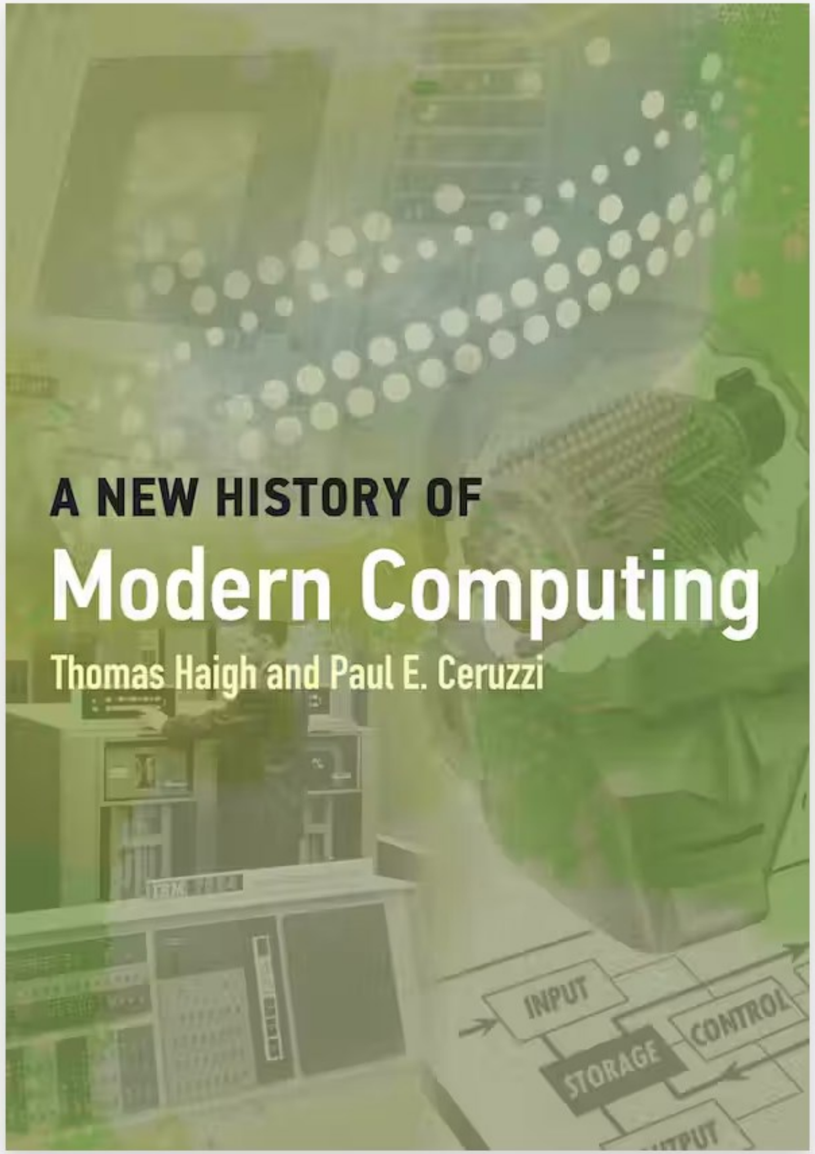
1990-2000 - moving beyond DOS, Windows 3, Windows begins to dominate, Microdsoft Office, relational databse management systems, Client Server applications, graphical laptops, RISC servers

The computer becomes a universal media device

1985-2010 - origins of digital media, multimedia, the Media Lab at MIT, computerizing music and speech, sampling and digital recordings, digitizing music, compact discs, CD-Roms, digitizing images, document transmission by fax, scanners, digital cameras, digital videos, digital television, downloading music, music players, graphics and games, virtual realitiy, 3D graphics accelerator and 3D games, 3D comes to game consoles

The computer becomes a publishing platform

1995-2004 - the World Wide Web, hypertext, graphical web browser, Internet server providers, web publishing, web payments, advertising, Google and Search advertising, encryption, the browser war, Linux, the triumph of open source



A brief history of Computing

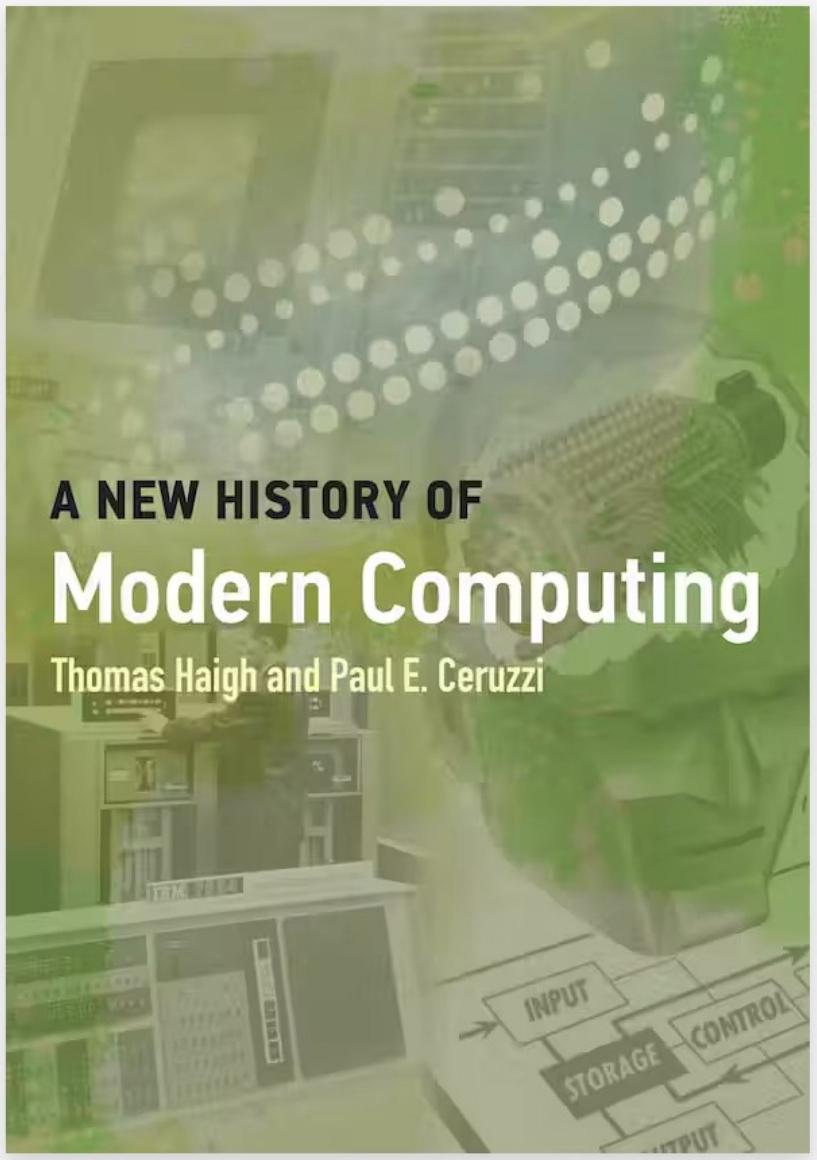
The computer becomes a network

2004-now - Data centers, the Cloud, multiple core processors, virtualization, streaming video, socila media, Facebook and advertising triumphant, Craiglist and Wikipedia, subscription and cloud storage

The computer becomes is everywhere and nowhere

2000-now - specialized mobile devices, handheld computers, pen computers, cellphones, pagers and beepers, GPS navigation systems, the Iphone, the Apple store, the mobile cloud, voice controlled assistants, tablets, evolution of smartphone, Android, smartphones and developing world, the Internet of Things in which everyday objects are equipped with internet connectivity, enabling them to send and receive data

this has led to new forms of technology interaction and paved the way for new services and business models



A brief history of Computing

Epilogue: a Tesla in the Valley

Silicon Valley and disruptive innovation, the convergence of cars and computers, the Tesla model S

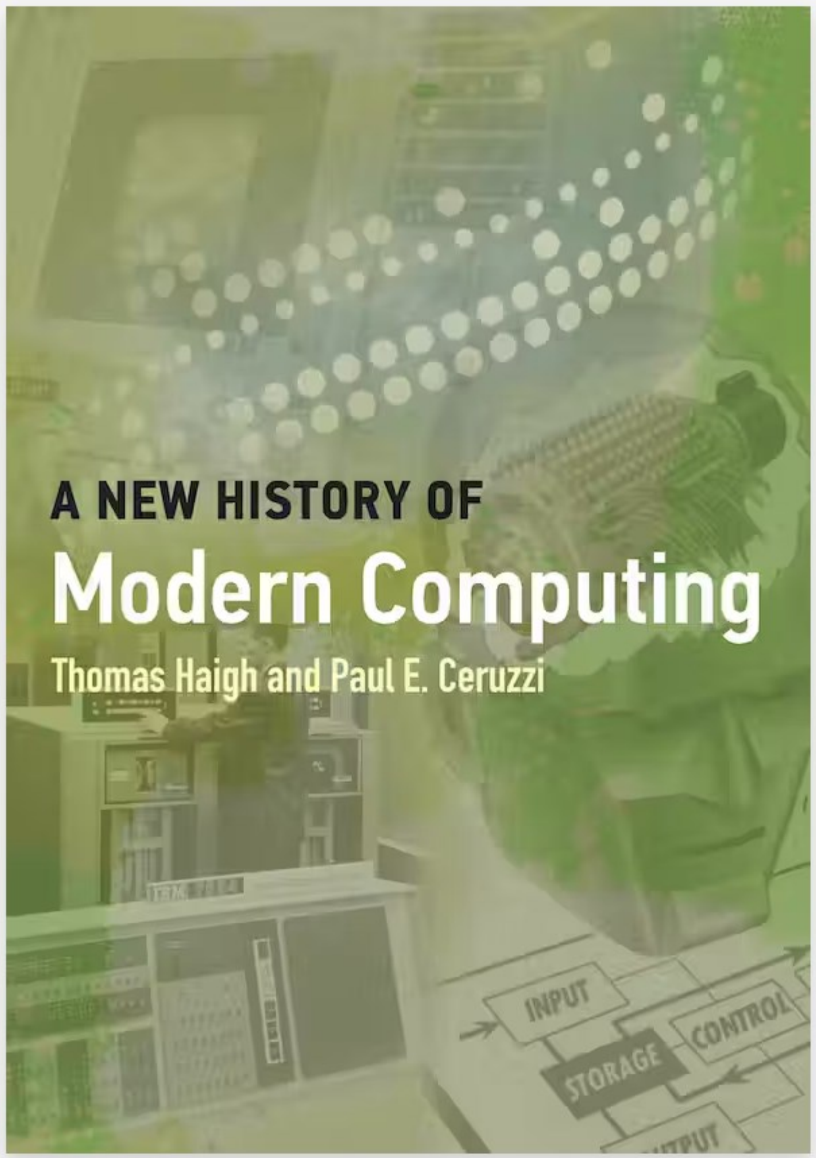
Tesla model S is equipped with nearly 60 on board computers

a powerful computer with a 3-core processor with a17-inch touch screen and for infotainment

a computer with AI specific for Autopilot and self-driving features

embedded computers for:

air suspension, parking brake, rear-view camera, power seats, wipers and rain, control each door, control radio signal from cellphone, 16 computers to control batteries, Controller Area Network, cellular data connection for system updates and traffic information, control of several cameras, radar and sonar systems for Autopilot

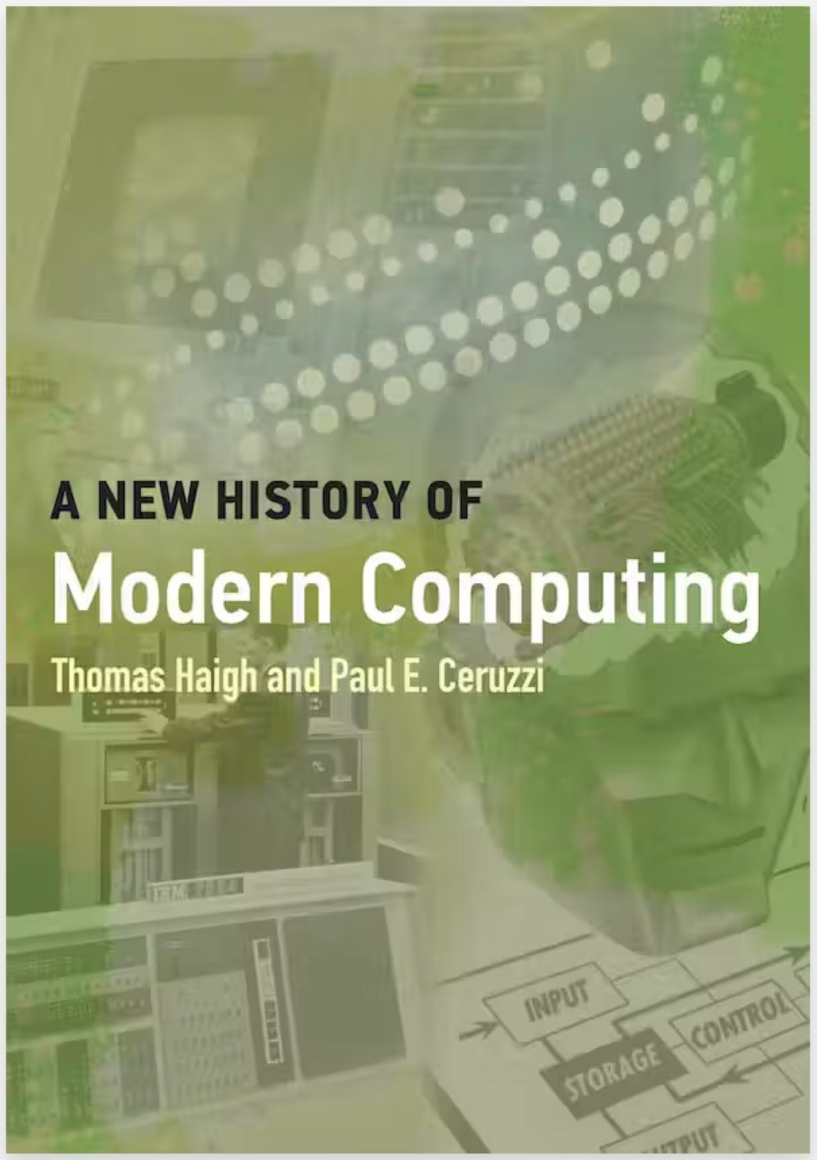


A brief history of Computing

Elon Mask: «Tesla is a Silicon Valley software company as much as it is a hardware company»

Technological evolution is not a linear or predetermined process.

Rather, it is the result of a complex interaction between technological innovation, social and cultural context, and the decisions of individuals, companies, and governments



lines of technological development

- miniaturization
- connectivity

↓

✓ more computing power
✓ greater storage capacity
✓ easy access to computing power and big data availability

Some basic terminology

G (iga) $=10^9$
T (era) $=10^{12}$
P (eta) $=10^{15}$
E (xa) $=10^{18}$
Z (etta) $= 10^{21}$
Y (otta) $=10^{24}$

Tbyte (Terabyte) $=10^{12}$ byte
Pflops (Petaflops) $=10^{15}$ floating-point operations per second
Eops (Exaops) $=10^{18}$ operations per second

How big are big numbers ?

how big is number 10^{20} ?

www.wolframalpha.com

number of people who have ever lived on Earth

1.085×10^{11} (people)

[as estimate for the year 2017 by the Population Reference Bureau, which assumes homo sapiens appeared about 50,000 B.C.]

How big are big numbers ?

how big is number 10^{20} ?

hypothesis: average life of **60 years** **convert 60 years to seconds**

1.892×10^9 (seconds)

hypotesis: about half of life spent talking, with a rapidity of 5 characters per second. Then, all humans who have lived on Earth have uttered altogether

about 5×10^{20} characters; that is 500 Exabytes allow you to memorize all the words spoken by mankind

How big are big numbers ?

Chat GPT prompt:
Estimate the number of bytes required to store every word spoken by humans since the origin of Homo sapiens

a common format for storing speech is MP3, which can be compressed down to about 128 kilobits per second (kbps) for decent quality. In bytes, this is 128 / 8 = 16 kilobytes (KB) per second.

Let's consider an average lifespan of 50 years for simplicity (though this varies significantly throughout human history and across different regions). If we assume that each person speaks for about an hour a day, then each person would generate about 1 hour * 60 minutes/hour * 60 seconds/minute = 3600 seconds of speech per day. Over a 50-year lifespan, this would amount to 50 years * 365 days/year * 3600 seconds/day = 65,700,000 seconds of speech

How big are big numbers ?

Chat GPT prompt:
Estimate the number of bytes required to store every word spoken by humans since the origin of Homo sapiens

if we consider that Homo sapiens have been around for roughly 300,000 years, and assume a constant global population of 1 billion people (which is a significant underestimate for recent centuries but an overestimate for most of human history), we get a total of about 300,000 years * 1 billion people * 65,700,000 seconds/person = 1.971 * 10^24 seconds of speech. Finally, multiplying this by the storage required for one second of speech gives us: 1.971 * 10^24 seconds * 16 KB/second = 3.1536 * 10^28 kilobytes. Converting this to more manageable units, we get approximately 3.1536 * 10^22 gigabytes, or 3.1536 * 10^13 exabytes

How big are big numbers ?

Chat GPT prompt:

I believe your previous estimate is indeed likely too high

Let's try another approach. Consider an average word to be 5 characters long. In ASCII, each character is 1 byte, so each word is approximately 5 bytes.

Assuming an average talking speed of about 125 words per minute, if we hypothesize that each person talks for about 2 hours each day, then the number of words spoken by a single person in a day would be: 125 words/minute * 60 minutes/hour * 2 hours/day = 15,000 words/day. Now, let's consider Homo sapiens have been around for roughly 50,000 years. If we assume an average lifespan of 30 years (which is more realistic when considering early Homo sapiens), the total number of words spoken by a person in their lifetime would be: 15,000 words/day * 365.25 days/year * 30 years = 164,287,500 words

How big are big numbers ?

Chat GPT prompt:
I believe your previous estimate is indeed likely too high

Assuming a constant global population of 100 million people (a rough estimate that might balance out the smaller populations in the past and the larger ones in the present), the total number of words spoken by all people throughout history would be:
164,287,500 words/person * 100,000,000 people * (50,000 years / 30 years) ≈ 2.74 * 10^20 words

The total storage required in bytes would then be:
2.74 * 10^20 words * 5 bytes/word = 1.37 * 10^21 bytes = 1.37 Zettabytes

How good is an algorithm ?

efficiency of an algorithm

time and **memory** of a computer are **computing resources**

the **performance** evaluation of a program is the quantification of the use of the computing resources necessary for its execution

How good is an algorithm ?

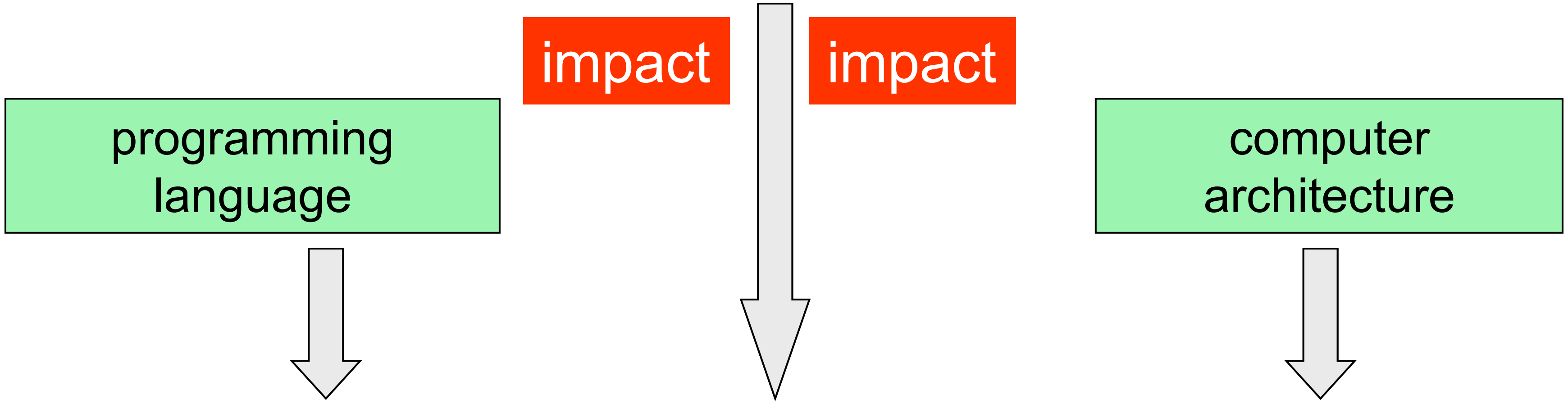
the total **number** of **operations** and **data** of an algorithm is **proportional** to the **time** and **memory** required for the execution of the program that implements the algorithm on a specific computer

computational complexity of an algorithm

determine the **amount of resources** for the execution of an algorithm

How good is an algorithm ?

computational complexity of an **algorithm**



performance of the **program (software)**
that implements the **algorithm**

How good is an algorithm ?

the **total number of operations** to be executed by an algorithm **depends on** the **number of input data**

number of input data

=

computational dimension (or **input size**) of the (instance of the) problem

when the computational dimension of a problem grows,
how do the number of operations (execution time) and the number of data (memory) of the algorithm change **?**

How good is an algorithm ?

Computational Complexity Theory investigates issues related to the amount of resources required for the execution of algorithms and the inherent difficulty of providing efficient algorithms to solve specific problems

- **time complexity** of an algorithm
- **space complexity** of an algorithm
- **inherent difficulty** of a **problem**

to classify
algorithms and **problems** in **complexity classes**

How good is an algorithm ?

time complexity of an algorithm

- identify the **computational dimension** of the problem
- identify the **dominant operation** (or dominant operations) of the algorithm

the **time complexity function** $T(n)$
of an algorithm is a function that expresses the number of dominant operations depending on the **computational dimension** n of the problem

How good is an algorithm ?

space complexity of an algorithm

- identify the **computational dimension** of the problem

the **space complexity function** $S(n)$ of an algorithm is the function that expresses the total size of the data structures used to store input, local and output data, depending on the **computational dimension** n of the problem

How good is an algorithm ?

time and **space complexity** of an algorithm

may also depend on
input data values

worst case
time complexity and **space complexity**

Time complexity of algorithms

```
for i in range (n):
        q dominant operations
```

time complexity $T(n) = qn$
that is
$T(n)$ is proportional to n

LINEAR
time complexity

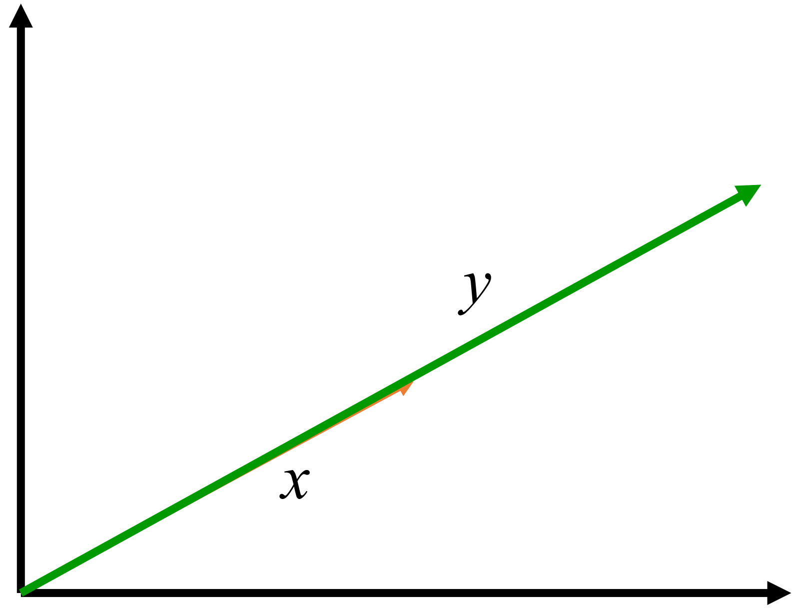
```
for i in range(n):
        q dominant operations
for j in range (n):
        q dominant operations
```

time complexity $T(n) = 2qn$
that is
$T(n)$ is proportional to 2n
(proportional to n)

Time complexity of algorithms

LINEAR
time complexity

scalar by vector (multiple of a vector)



$$x \in R^n \quad , \alpha \in R$$

$$y = \alpha x \qquad \in R^n$$

$$y_i = \alpha x_i \qquad i = 1 : n$$

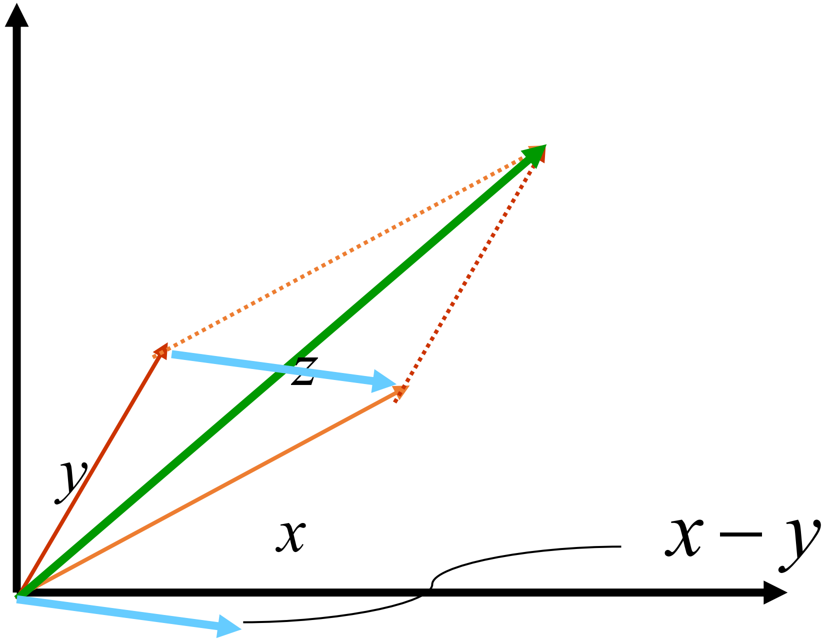
$$T(n) = n = O(n)$$

Time complexity of algorithms

LINEAR
time complexity

sum of vectors

$$x, y \in \Re^n$$



$$z = x + y \qquad \in R^n$$

$$z_i = x_i + y_i \qquad i = 1:n$$

$$T(n) = n = O(n)$$

why vectors are so important?

a vector is a mathematical concept that represents a quantity or object in a multi-dimensional space. It can be used to represent various types of data, such as documents, human behaviors, and images, allowing for quantitative evaluation of similarity and commonalities among them

Document representation: vectors can be used to create text models based on word frequencies or other attributes. For example, a vector can be constructed where each dimension represents a word in the vocabulary, and the value associated with that dimension is the frequency of that word in the document

Human behaviors: vectors can be used to represent actions, preferences, or characteristic traits of a person. For example, vectors can be used to represent a person's musical preferences, where each dimension represents a music genre, and the value associated with that dimension indicates the interest or preference for that genre

Images: vectors can be used to represent the visual features of an image or the content of each pixel of an image (raster). Image analysis can extract various attributes such as color, shape, texture, and object arrangements in the image. These attributes can be represented as a vector, where each dimension corresponds to an attribute, and the value associated with that dimension indicates the intensity or presence of that attribute in the image

why vectors are so important?

vectors can be used to create **buyer profiles**. In the field of marketing and consumer analysis, vectors can represent the characteristics, preferences, and behaviors of buyers. These vectors can be created using different dimensions, such as:

Demographics: vectors can include demographic attributes such as age, gender, income, geographic location, marital status, and more

Purchase preferences: vectors can represent buyers' purchase preferences, such as preferred product categories, favorite brands, preferred purchase channels (online or physical stores), price sensitivity level, and so on

Online behavior: vectors can be created using digital data like social media interactions, past online purchases, online searches, ad clicks, and other online activities. This data can be used to create a behavioral profile of buyers

Feedback and reviews: vectors can incorporate information derived from product reviews or buyer feedback. For example, ratings and opinions expressed by buyers can be represented as attributes within the vectors

why vectors are so important?

Once buyer profile vectors are created, they can be used for various purposes, such as:

Market segmentation: vectors can be used to group buyers into homogeneous segments based on their common characteristics and preferences. This can help develop targeted marketing strategies for each segment

Personalized recommendations: buyer profile vectors can be used to generate personalized recommendations of products or services. By relying on similar buyer profile vectors, products or offers that may interest a specific buyer can be suggested

Future behavior prediction: analysis of buyer profile vectors can contribute to predicting future buyer behavior, such as the likelihood of purchasing certain products, brand loyalty, or participation in loyalty programs

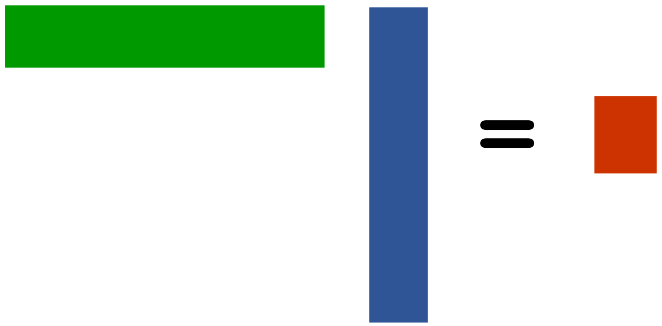
Time complexity of algorithms

LINEAR
time complexity

dot (scalar) product of vectors

$$x, y \in R^n$$

$$s = x^T y \in R$$



$$s = \sum_{i=1}^{n} x_i y_i$$

$$T(n) = 2n = O(n)$$

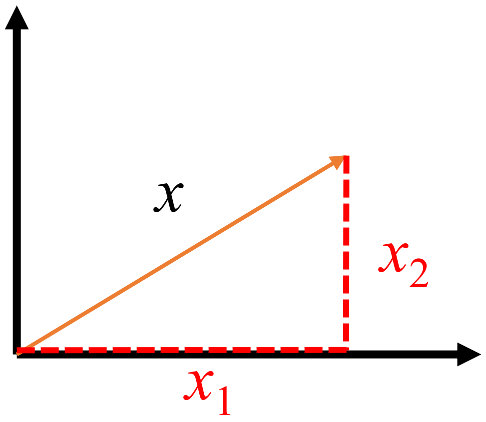
Time complexity of algorithms

LINEAR
time complexity

the Euclidean norm (norm 2) of a vector is the geometric length of a vector

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$$

$$\|x\|_2 = \sqrt{x^T x}$$



$$T(n) = 2n = O(n)$$

Time complexity of algorithms

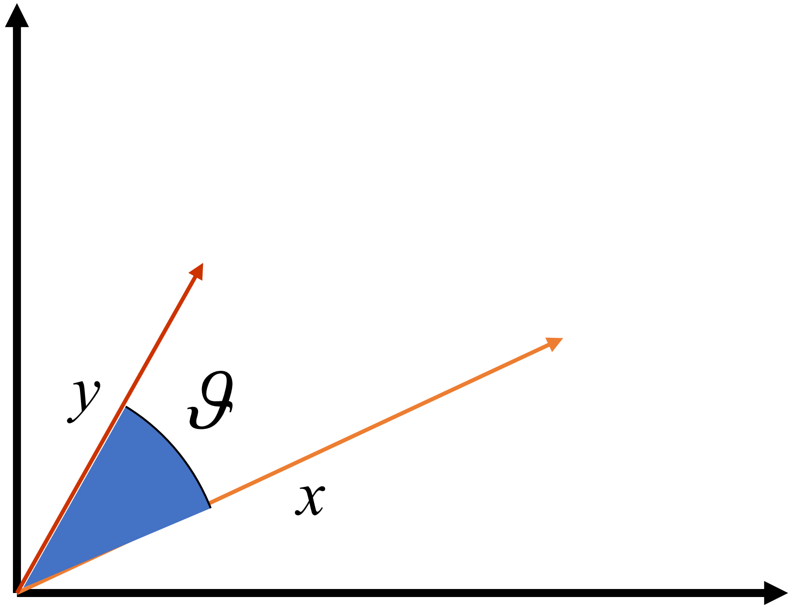
LINEAR time complexity

angle between two vectors

$$x^T y = \|x\|_2 \|y\|_2 \cos(\vartheta)$$

$$\cos(\vartheta) = \frac{x^T y}{\|x\|_2 \|y\|_2}$$

$$x^T y = 0 \Leftrightarrow x \perp y$$



the dot product of two vectors is a useful tool for understanding the relationship between vectors

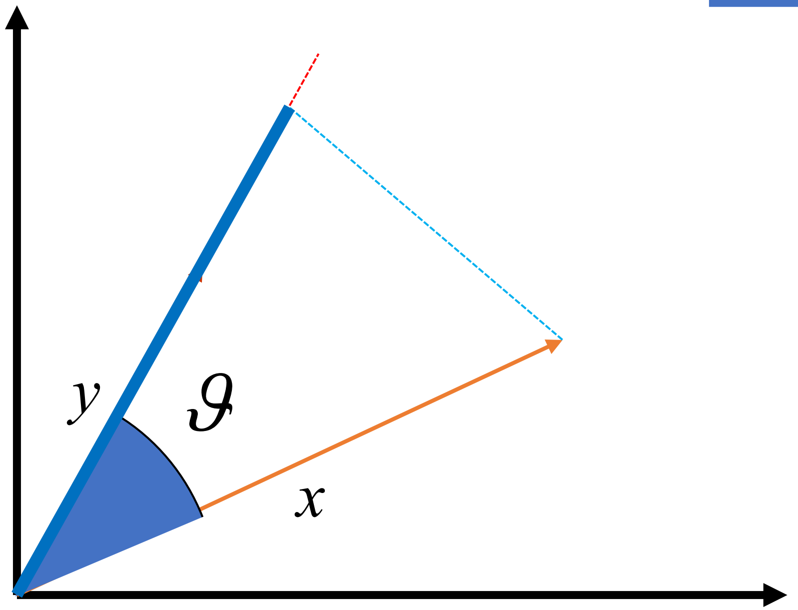
Time complexity of algorithms

LINEAR
time complexity

$$x^T y = \|x\|_2 \|y\|_2 \cos(\vartheta)$$

$$\cos(\vartheta) = \frac{x^T y}{\|x\|_2 \|y\|_2}$$

orthogonal projection



the dot product of two vectors is a useful tool for understanding the relationship between vectors

$\|x\|_2 \cos(\vartheta) =$ length of orth projection of x onto y

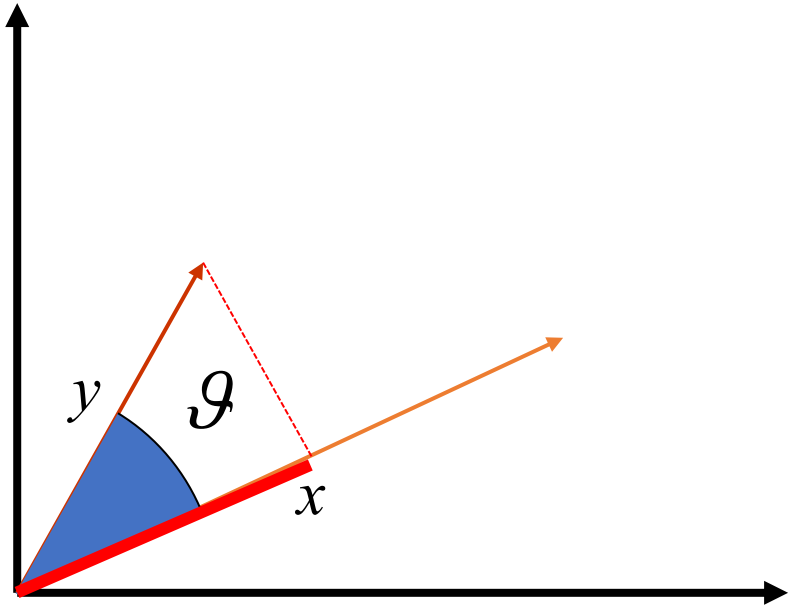
Time complexity of algorithms

LINEAR
time complexity

orthogonal projection

$$x^T y = \|x\|_2 \|y\|_2 \cos(\vartheta)$$

$$\cos(\vartheta) = \frac{x^T y}{\|x\|_2 \|y\|_2}$$

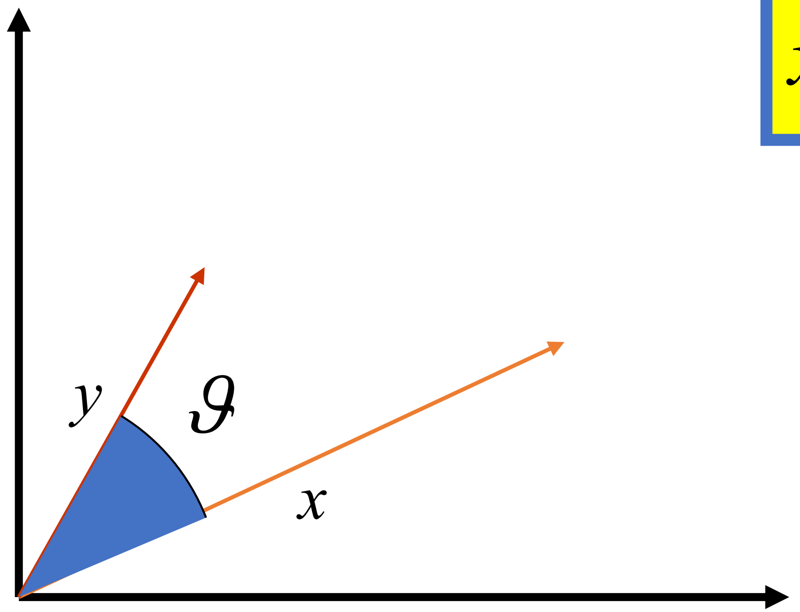


the dot product of two vectors is a useful tool for understanding the relationship between vectors

$\|y\|_2 \cos(\vartheta)$ = length of orth projection of y onto x

Time complexity of algorithms

LINEAR
time complexity



$$x^T y = \|x\|_2 \|y\|_2 \cos(\vartheta)$$

$$\cos(\vartheta) = \frac{x^T y}{\|x\|_2 \|y\|_2}$$

the dot product tells us how much of one vector is in the direction of the other vector

$\cos(\vartheta)$ is a **measure of similarity** between the two vectors

0 implies no similarity (perpendicularity)
1 implies maximum similarity (same direction, same verse)
-1 implies maximum opposition (same direction, opposite direction)

Time complexity of algorithms

```
for i in range (n):
    for j in range (n):
        q  dominant operations
```

time complexity

$$T(n) = q\,n^2$$

that is

T(n) is proportional to n^2

QUADRATIC
time complexity

```
for i in range (n):
    for j in range (i,n):
        q  dominant operations
```

time complexity

$$T(n) = q(1+2+3+\ldots+n)$$
$$= qn(n+1)/2$$

that is

T(n) is proportional to n^2

Time complexity of algorithms

QUADRATIC
time complexity

product of matrix by vector

$$A \in R^{m \times n} \;,\; v \in R^{n}$$

$$z = Av \in R^{m}$$

$$z_i = \sum_{j=1}^{n} a_{ij} v_j \qquad , i = 1 : m$$

$$T(m,n) = m(2n-1) = O(mn)$$

$$T(n) = 2n^2 - n = O(n^2)$$

Time complexity of algorithms

QUADRATIC time complexity

product of matrix by vector

the product of a matrix times a vector is a linear transformation of one vector into another vector

a linear transformation has the property of preserving the result of a sum of two vectors and a multiplication of a scalar by a vector (multiple of a vector)

$$v \to w \qquad r \to s$$

$$(v + \alpha r) \to w + \alpha s$$

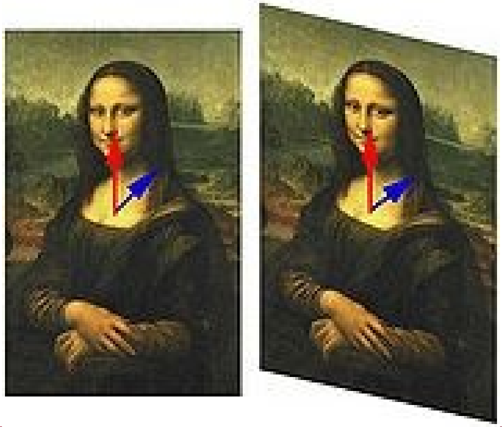
Time complexity of algorithms

QUADRATIC
time complexity

each pixel is a vector

rotation

scaling

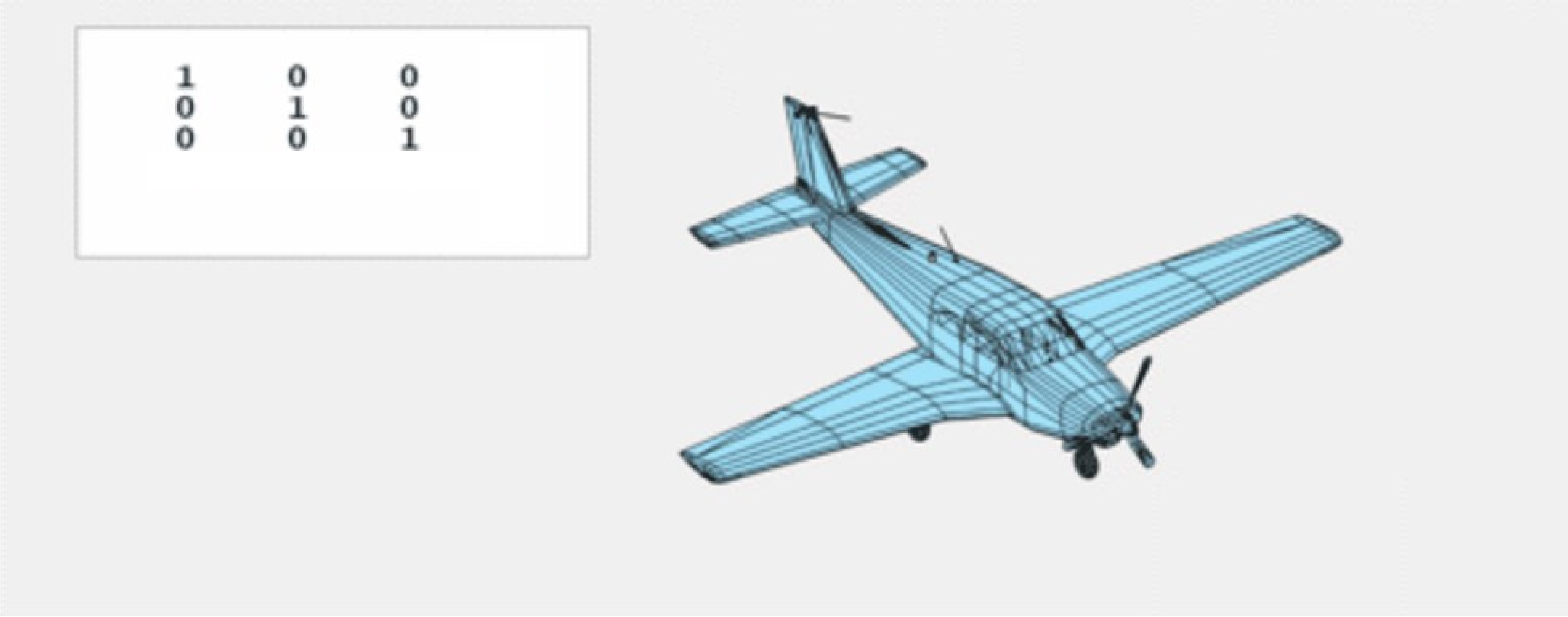


shearing
leaves all points on an axis unchanged; the other points are translated parallel to the axis in proportion to their distance from the axis (preserves areas)

Time complexity of algorithms

QUADRATIC
time complexity

pitch
3D rotation around the wing axis (x-axis)

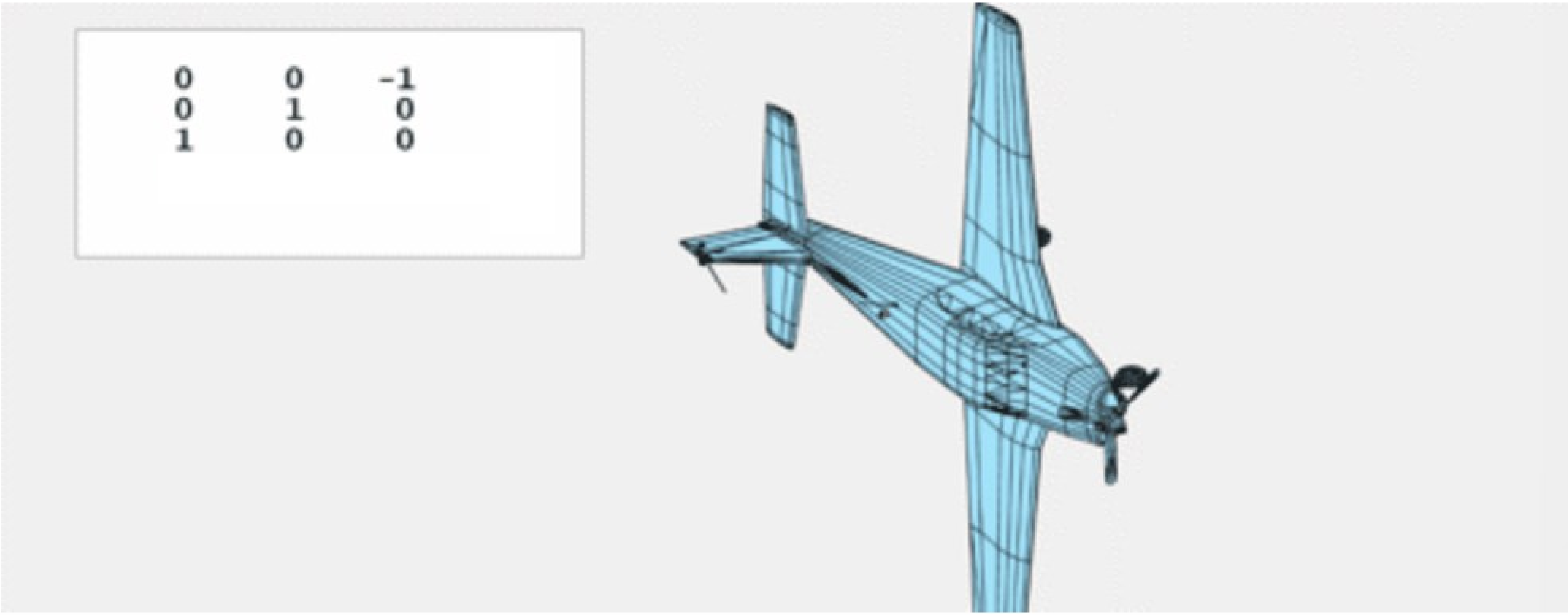


Time complexity of algorithms

QUADRATIC
time complexity

roll

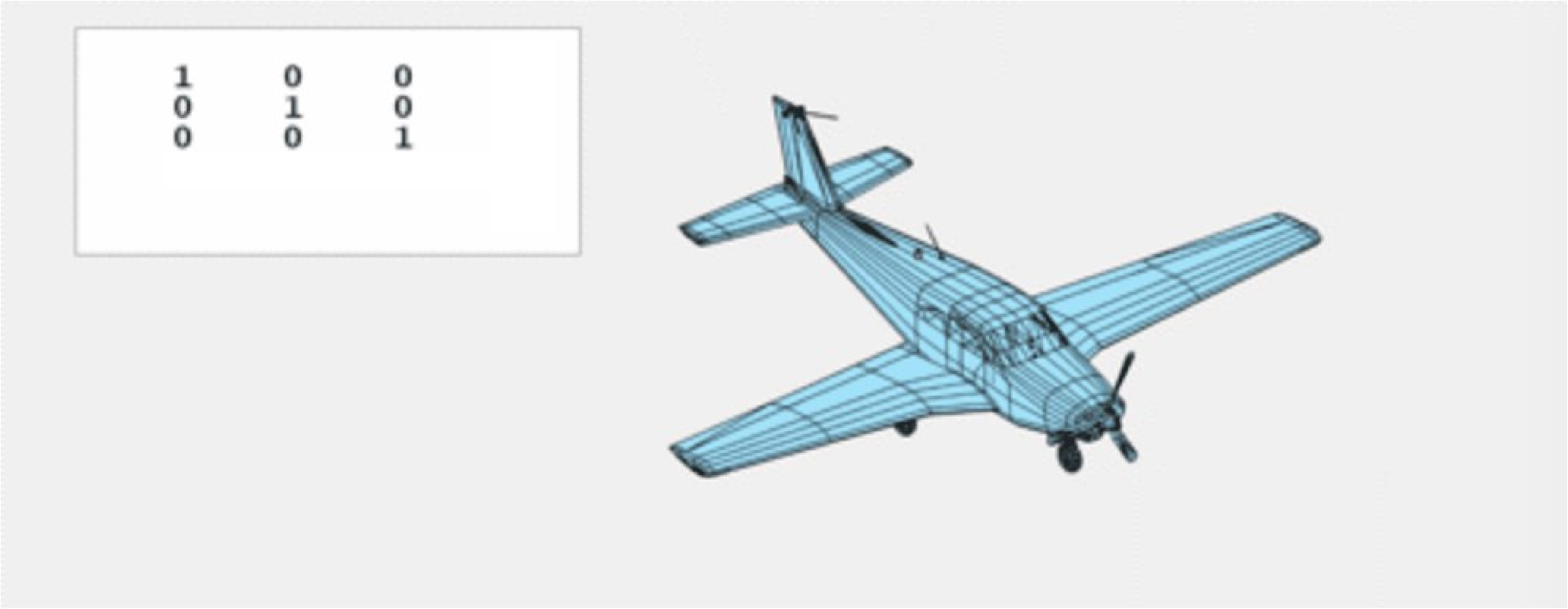
3D rotation around the longitudinal axis (y-axis)



Time complexity of algorithms

QUADRATIC
time complexity

yaw
3D rotation around the vertical axis (z-axis)



Time complexity of algorithms

```
for i in range (n):
    for j in range (n):
        for k in range (n):
            q dominant operations
```

time complexity $T(n) = q\ n^3$
that is
$T(n)$ is proportional to n^3

CUBIC
time complexity

Time complexity of algorithms

CUBIC time complexity

product of two matrices

the product of two matrices is the combination of two linear transformations

$$A \in R^{m \times n}, B \in R^{n \times p}$$

$$C = AB \in R^{m \times p}$$

$$T(m,n,p) = 2mnp = O(mnp)$$

$$T(n) = 2n^3 = O(n^3)$$

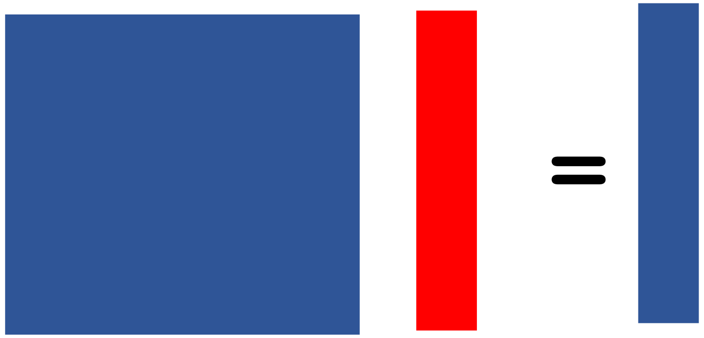
Time complexity of algorithms

CUBIC
time complexity

solution of a linear system of equations

$$A \in R^{n\times n}, b \in R^n$$

$$Ax = b, \qquad x \in R^n$$



$$T(n) = n^3/3 = O\left(n^3\right)$$

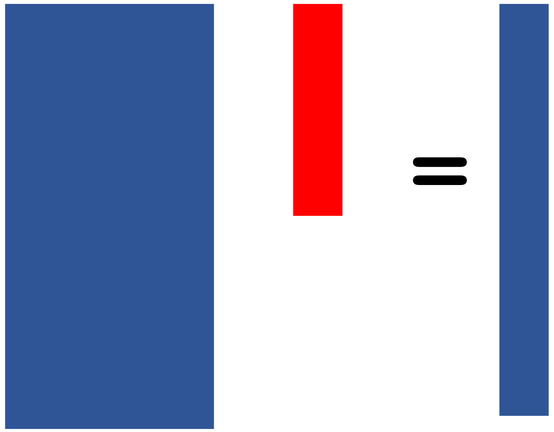
Time complexity of algorithms

CUBIC
time complexity

solution of an overdetermined linear system of equations (regression)

$A \in R^{m \times n}, b \in R^m$

$$Ax = b, \qquad x \in R^n$$



$$T(m,n) = O\left(mn^2/2\right)$$

Time complexity of algorithms

```
i = 0
while 2**i < n:
    i = i+1
        q dominant operations
```

```
i = n
while i >= 1:
    i = i/2
        q dominant operations
```

time complexity $T(n) = q\log_2 n$

that is

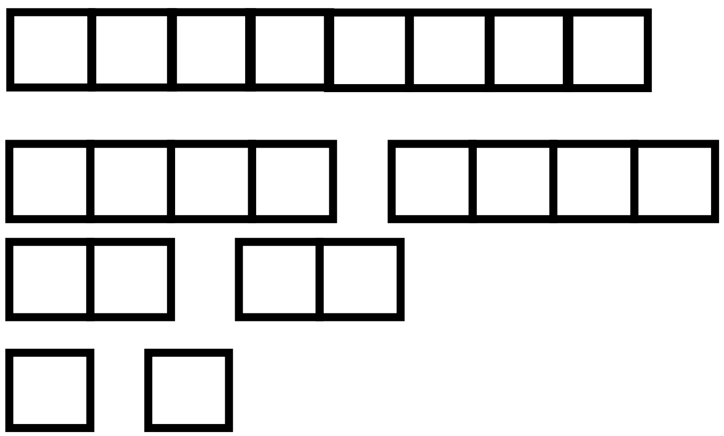
$T(n)$ is proportional to $\log_2 n$

LOGARITHMIC

time complexity

Time complexity of algorithms

how many times can you divide an array of size n in half?



n=8

LOGARITHMIC
time complexity

n power of 2: $\log_2(n)$ times

in general: smallest integer greater than $\log_2(n)$ times

Time complexity of algorithms

LOGARITHMIC
time complexity

Binary search is a search algorithm used to find a specific **target value** within a **sorted list**

1. Start by defining the target value you want to find in the sorted list
2. Identify the middle element of the sorted list
3. Compare the middle element with the target value:
   a) If the middle element is equal to the target value, the search is successful, and you can return the index of the middle element
   b) If the middle element is greater than the target value, the target value must be in the left half of the sorted list. Repeat the process starting from step 2 on the left half of the list
   c) If the middle element is less than the target value, the target value must be in the right half of the sorted list. Repeat the process starting from step 2 on the right half of the list
4. Repeat steps 2 and 3, dividing the search range in half each time, until the target value is found, or the search range becomes empty (indicating that the target value is not present in the list)

Time complexity of algorithms

LOGARITHMIC time complexity

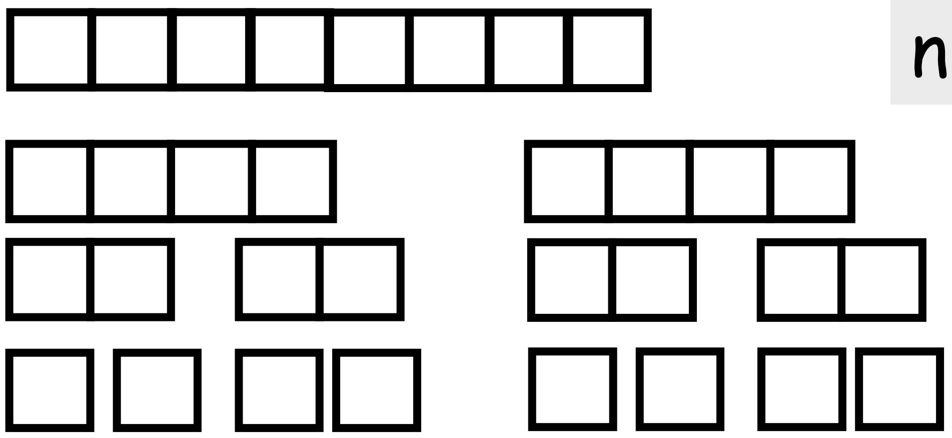
Suppose we have a **sorted list** of numbers: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

1. we want to find the **target value** 9 within this list
2. we start by identifying the **middle element** of the list, which is 11
3. since 9 is less than 11, we know that the target value must be in the left half of the list. So, we discard the right half ([13, 15, 17, 19]) and focus on the left half ([1, 3, 5, 7, 9])
4. we repeat the process. The middle element of the left half is 5. Since 9 is greater than 5, we discard the left half ([1, 3, 5]) and focus on the right half ([7, 9])
5. we repeat the process again. The middle element of the right half is 9, which is the target value we were searching for. The search is successful, and we return the index of the middle element, which is 4

in this example, the binary search algorithm found the **target value** 9 at index 4 of the sorted list

Time complexity of algorithms

algorithms that carry out $log_2(n)$ steps and at each step execute n dominant operations



n =8

time complexity $T(n) = n \log_2 n$

that is

$T(n)$ is proportional to $n \log_2 n$

LIN-LOG

time complexity

n power of 2: $n \log_2(n)$ operations

Time complexity of algorithms

LIN-LOG
time complexity

Merge sort is a popular sorting algorithm that follows the divide-and-conquer approach. Here's a quick overview of how Merge sort works:

1. Divide: the unsorted list is recursively divided into smaller sublists until each sublist contains only one element or is empty
2. Conquer: the sublists are then merged back together in a sorted order. This is done by repeatedly comparing and merging pairs of sublists until a single sorted list is obtained
3. Merge: during the merge step, two sublists are compared element by element, and the smaller element is placed into a new resulting list. This process continues until all elements from both sublists are merged into the final sorted list
4. The resulting sorted list is the output of the Merge sort algorithm

Time complexity of algorithms

LIN-LOG
time complexity

Suppose we have an unsorted list: [5, 3, 8, 2, 1, 6]

1. Divide: the list is divided into sublists: [5, 3, 8] and [2, 1, 6]
2. Conquer: each sublist is further divided until we have single-element sublists: [5], [3], [8], [2], [1], [6]
3. Merge: the single-element sublists are then merged back together in sorted order
   - first, we compare and merge [5] with [3] to get [3, 5].
   - then, we compare and merge [8] with [2] to get [2, 8]
   - next, we compare and merge [1] with [6] to get [1, 6]
   - now, we compare and merge [3, 5] with [2, 8] to get [2, 3, 5, 8]
   - finally, we compare and merge [1, 6] with [2, 3, 5, 8] to get the fully sorted list: [1, 2, 3, 5, 6, 8]
4. The sorted list [1, 2, 3, 5, 6, 8] is the output of the Merge sort algorithm

Time complexity of algorithms

information on the time complexity of an algorithm can be obtained experimentally, by executing a program that implements the algorithm

run the program to solve the problem of dimensione **n**, and
then **2n,4n,8n,16n**
and finally analyze the execution times

n	execution time (sec)	increasing factor of time execution (current / previous)

Time complexity of algorithms

n	execution time (sec)	increasing factor of time execution (current / previous)
1000	0.0201	
2000	0.0408	2.004
4000	0.0793	1.967
8000	0.1611	2.031
16000	0.3182	1.975

Time complexity of algorithms

increasing factor *almost* constant and approximately equal to **2**

if you **double** the dimension
the execution time **doubles**

↓

time complexity $T(n)$ of the algorithm is proportional to n

Time complexity of algorithms

if the time complexity T(n) of the algorithm is proportional to n $T(n) = \alpha \cdot n$

then:

$$T(n) = \alpha \cdot n$$
$$T(2n) = \alpha \cdot 2n$$

and it must hold that:

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

Time complexity of algorithms

if the time complexity T(n) of the algorithm is proportional to n^2

$$T(n) = \alpha \cdot n^2$$

then:

$$T(n) = \alpha \cdot n^2$$

$$T(2n) = \alpha \cdot (2n)^2 = \alpha \cdot 4n^2$$

and it must hold that:

$$\frac{T(2n)}{T(n)} = \frac{\alpha \cdot 4n^2}{\alpha \cdot n^2} = 4$$

Time complexity of algorithms

if the time complexity T(n) of the algorithm is proportional to n^3

$$T(n) = \alpha \cdot n^3$$

then:

$$T(n) = \alpha \cdot n^3$$

$$T(2n) = \alpha \cdot (2n)^3 = \alpha \cdot 8n^3$$

and it must hold that:

$$\frac{T(2n)}{T(n)} = \frac{\alpha \cdot 8n^3}{\alpha \cdot n^3} = 8$$

Time complexity of algorithms

increasing factor of time execution, doubling the computational dimension	presumable form of time complexity
2	proportional to n
4	proportional to n^2
8	proportional to n^3
16	proportional to n^4

time complexity **proportional to a POWER of** n

↓

POLYNOMIAL complexity

Asymptotic time complexity of algorithms

Let $T(n)$ and $g(n)$ be two **non negative** and **non decreasing** functions, then

$$T(n) = O(g(n))$$

T is of the order of g

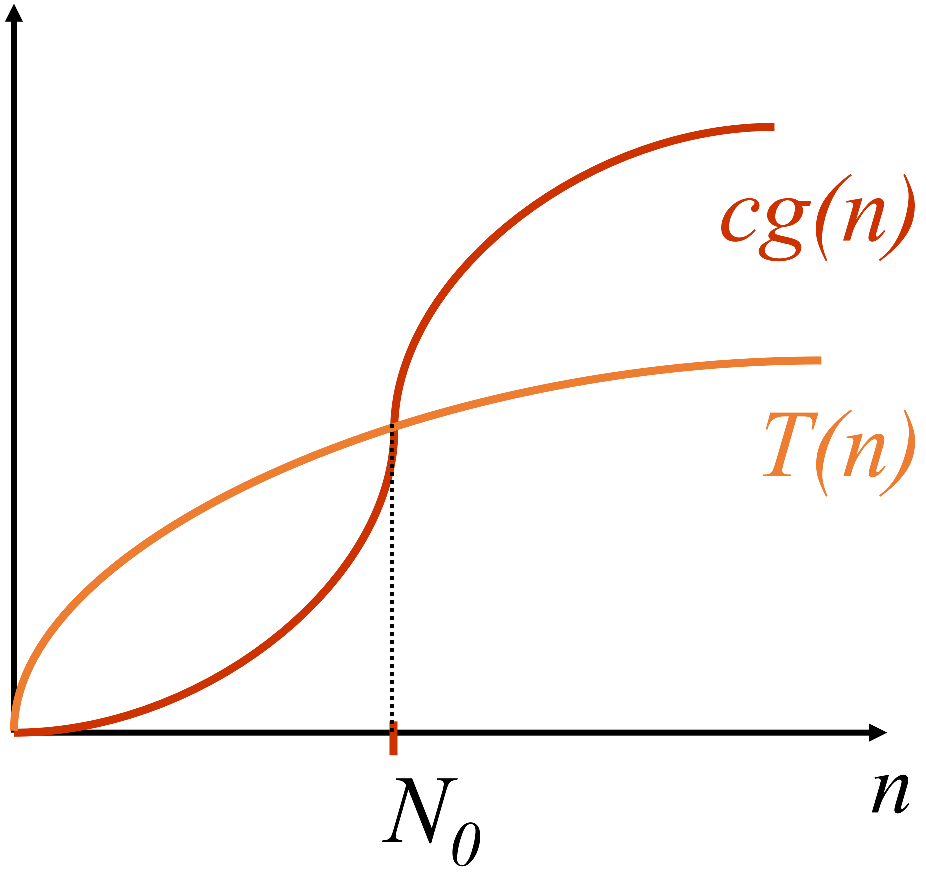
if there are two **positive constants** c and N_0 such that:

$$T(n) \leq cg(n) \qquad n \geq N_0$$

Asymptotic time complexity of algorithms

$$T(n) = O\big(g(n)\big)$$

the curve $T(n)$ stays **below** (or coincides) the curve $cg(n)$, starting from N_0



Asymptotic time complexity of algorithms

$$T(n) = O(g(n))$$

Exercise: if $T(n) = \sum_{i=0}^{p} a_i n^i$

that is, $T(n)$ is a polynomial of degree p then: $T(n) = O(n^p)$

Exercise: if $T(n) = 2n^2 + 3n + 5$ then: $T(n) = O(n^2)$

Asymptotic time complexity of algorithms

$$T(n) = O\big(g(n)\big)$$

Exercise: give a function $T(n)$ such that

$$T(n) = O(1)$$

Exercise: yes or no:

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

Complexity classes

T(n) = $O(1)$	constant
T(n) = $O(\log_2 n)$	logarithmic
T(n) = $O(n)$	linear
T(n) = $O(n \log_2 n)$	lin-log
T(n) = $O(n^2)$	quadratic
T(n) = $O(n^k)$	polynomial of degree k
T(n) = $O(2^n)$	exponential
T(n) = $O(n!)$	factorial

Can we always design an algorithm to solve any problem?

complexity	1 sec	1 minute	1 hour
$O(n)$	10^8	$6 \cdot 10^9$	$3.6 \cdot 10^{11}$
$O(n \log_2 n)$	$\approx 4 \cdot 10^6$	$\approx 2 \cdot 10^8$	$\approx 1 \cdot 10^{10}$
$O(n^2)$	10^4	77459	$6 \cdot 10^5$
$O(2^n)$	26	32	38
$O(n!)$	11	12	14

dimensions of problems that can be solved, with a 100 Mops/sec computer

in WolframAlpha: `solve 10^(-8)*x^2=60`

Can we always design an algorithm to solve any problem?

complexity	1 sec	1 minute	1 hour
$O(n)$	10^{12}	$6 \cdot 10^{13}$	$3.6 \cdot 10^{15}$
$O(n \log_2 n)$	$2.5 \cdot 10^{10}$	$\approx 1.4 \cdot 10^{12}$	$\approx 7.5 \cdot 10^{13}$
$O(n^2)$	10^6	$\approx 7.7 \cdot 10^6$	$6 \cdot 10^7$
$O(2^n)$	35	45	51
$O(n!)$	14	16	17

dimensions of problems that can be solved, with a 1000000 Mops/sec computer

in WolframAlpha: `solve 10^(-12)*2^x=60`

Can we always design an algorithm to solve any problem?

algorithms that can be used for effective problem solving

↓

polynomial time complexity algorithms

~~exponential/factorial time complexity algorithms **cannot be used** in any practical problem solving~~

Can we always design an algorithm to solve any problem?

exponential/factorial time complexity algorithms **cannot be used** in any practical problem solving

- ✓ algorithm of **factorial** time complexity,
- ✓ computer which executes 1 operazione in 10^{-12} sec,
- ✓ problem of computational dimensione 100

in WolframAlpha:
```
10^(-12)*factorial(100)/(86400*365)
```

total execution time : 10^{138} years

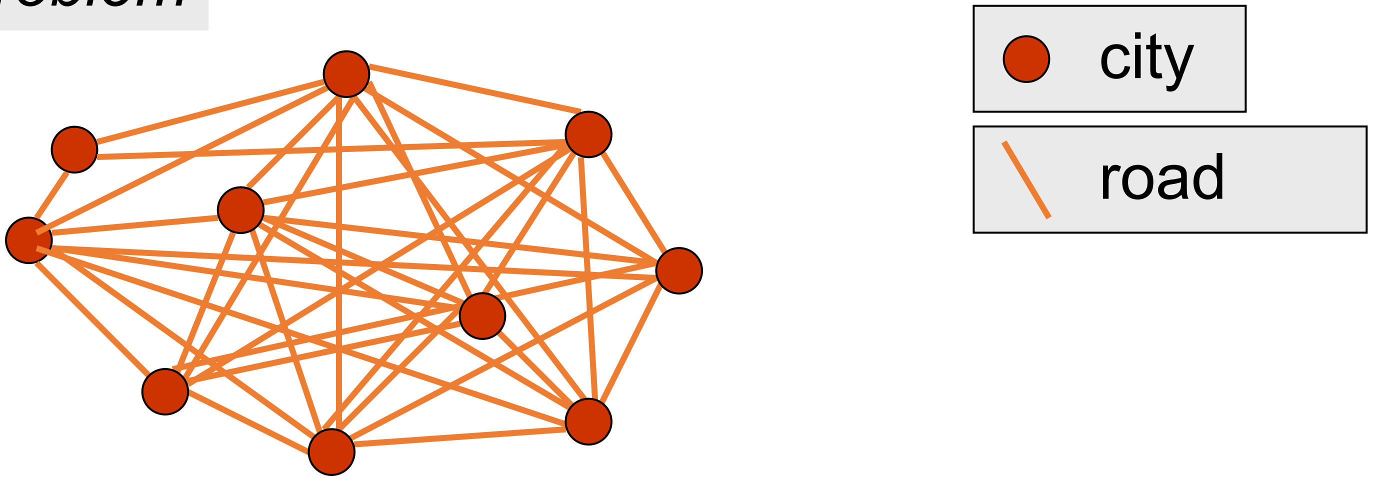
MEIM | MASTER IN ENTREPRENEURSHIP INNOVATION MANAGEMENT IN COLLABORATION WITH MIT SLOAN

UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

Are there problems solved only by exponential/factorial time complexity algorithms ?

TSP, *travelling salesman problem*

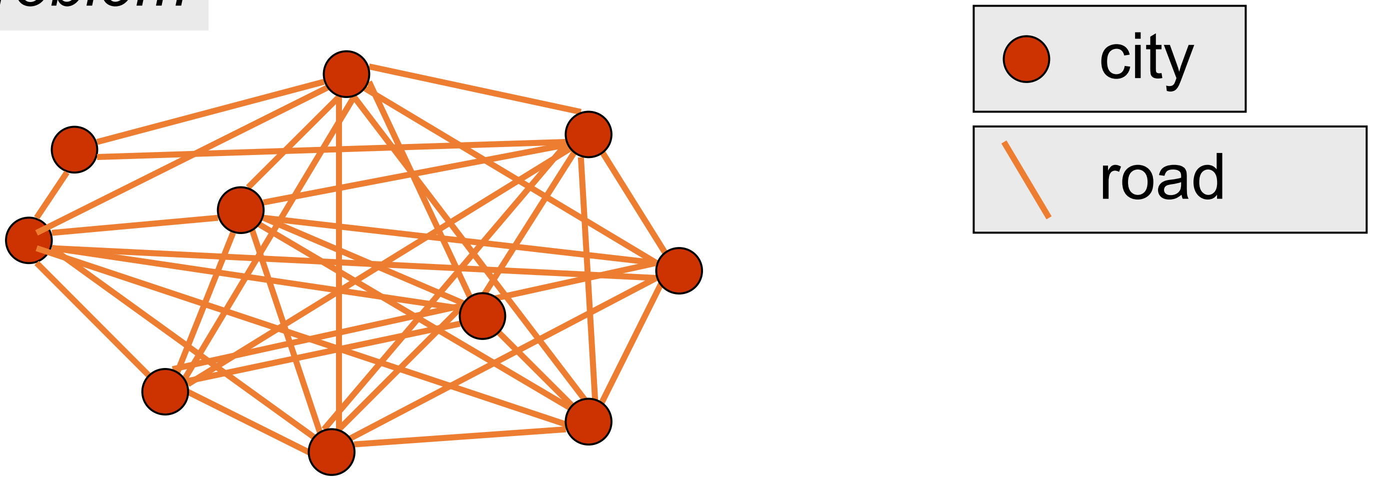
$$T(n) = O(n!)$$



given a set of cities and travel costs, from any city to any other city, determine the **cheapest** route that allows you to visit **each** city **exactly once** and then return to the city of departure

Are there problems solved only by exponential/factorial time complexity algorithms ?

TSP, *travelling salesman problem*



$$T(n) = O(n!)$$

the only known algorithm that solves this problem is a brute force algorithm that computes all possible routes and determines the cheapest one

the number of possible routes is equal to the **number of the permutations** of n cities

Algorithms ARE technology

problem of computational dimension $n=10^7$	algorithm of quadratic complexity	algorithm of lin-log complexity
computer power	10000 Mops/sec	100 Mops/sec
execution time	$\frac{(10^7)^2 \text{ operations}}{10^{10} \text{ operations/sec}}$	$\frac{10^7 \cdot \log_2 10^7}{10^8 \text{ operationis/sec}}$
	$=10^4$ sec	$=2$ sec

Scalability of algorithms

Scalability is a key concept in processing large-scale problems in various fields, including science, economics, and artificial intelligence

Scalability refers to an algorithm's ability **to efficiently handle an increase in the size of the problem**

in terms of high-performance computing, a "scalable" algorithm is one that can effectively leverage a growing number of processing units to solve larger problems or solve problems at the same speed or faster

Scalability of algorithms

1. **Science:** many scientific problems require large-scale computation. For example, simulating physical systems, like Earth's climate or the behavior of subatomic particles, requires massive amounts of computing power. The algorithms used in these simulations must be highly scalable to handle the complexity and size of the data. For instance, discretization methods like finite differences, finite elements, and finite volumes are often used to solve large-scale partial differential equations that model these physical systems

2. **Artificial Intelligence:** many machine learning algorithms need to handle large amounts of data. Deep learning algorithms, such as convolutional neural networks (CNN) for image recognition or recurrent neural networks (RNN) for natural language processing, must be highly scalable. Stochastic gradient descent (SGD) is an example of a scalable optimization algorithm used in deep learning. Similarly, tree-based methods can be scaled to handle large data sets

underpins

Scalability of algorithms

3. **Economics:** in economic and financial analysis, large-scale optimization algorithms are often needed. For instance, in portfolio risk management, one might want to minimize the risk of a portfolio of thousands or even millions of assets

linear and convex quadratic programming algorithms are commonly used in these cases

algorithms such as the simplex method, interior point method, and gradient methods are scalable and can handle problems of significant size

to achieve **scalability**, problems can be broken down into smaller sub-problems that can be solved in parallel

this approach underpins **parallel / distributed computing** and **GPU computing**, which are key techniques for scalability in HPC and AI

More on Scalability

Scalability is not just about **algorithms**, but also about **system architecture**

For example, a distributed database needs to be carefully designed to ensure that it can scale with increasing data size and query load

Scalability in system architecture refers to the ability of the system to handle increased load by adding resources, typically in the form of hardware like servers or storage

two main types of scalability exist: **vertical** (or scaling up) and **horizontal** (or scaling out)

More on Scalability

Vertical Scaling: this involves increasing the capacity of a single machine in the system, such as adding more powerful processors, more memory, or more storage. Vertical scaling can be a simple and effective way to increase system performance, but it has its limitations. There is usually a physical limit to how much you can scale up a single machine, and high-performance hardware can be expensive. Additionally, a system that relies on a single powerful machine can have a single point of failure, which can be a risk for system reliability

Horizontal Scaling: this involves adding more machines to the system to distribute the load. Each machine handles a part of the overall workload, and the load balancer is typically used to distribute requests among the machines. Horizontal scaling can provide a high degree of scalability and can be more cost-effective than vertical scaling, as it can take advantage of less expensive commodity hardware. However, not all applications can be easily distributed across multiple machines. Also, managing a large number of machines can add complexity to system management, including challenges with data consistency and coordination between servers

More on Scalability

a third approach, called **diagonal scaling**, combines elements of both vertical and horizontal scaling. In diagonal scaling, new machines are added to the system (like horizontal scaling), and the capacity of each machine is increased over time (like vertical scaling)

in addition to these, the rise of **cloud computing** has introduced the concept of **elastic scaling**, where resources are added or removed on demand based on the current workload. This can provide cost savings, as you only pay for the resources you use, and can ensure that the system can handle peaks in demand

another important aspect of scalable system architecture is the design of the software itself

software must be designed to take advantage of multiple processors (through multithreading or multiprocessing) and multiple machines (through distributed computing techniques)

this often involves considerations of data partitioning, replication, consistency, and fault tolerance

More on Scalability

- ✓ computing power is becoming a bottleneck for developing AI
- ✓ estimated training time of ChatGPT-3 is nearly 288 years on a single V100 Nvidia GPU
- ✓ the current cost (March 2023) of running ChatGPT is $100,000 per day
- ✓ Microsoft's Azure cloud is hosting ChatGPT so that OpenAI does not have to invest in a physical server room (partially true!)
- ✓ considering Microsoft's current rates, it is $3 an hour for a single A100 GPU, and each word generated on ChatGPT costs $0.0003
- ✓ at least eight GPUs are in use to operate on a single ChatGPT query
- ✓ so, when ChatGPT generates an average response of 30 words, it will cost nearly 1 cent for the company
- ✓ through such an estimation, OpenAI could be spending at least $100K per day or $3 million monthly on running costs

MASTER MEIM 2022-2023

High Performance Computing

……. to be continued