

BST 234: Introduction to Data Structures and Algorithms

Georg, Anjali, Intekhab, and Christoph

Spring 2021

Who are we?

- ▶ Georg Hahn (Building 2, 436)

✉ ghahn@hsph.harvard.edu

research interests: algorithmic mathematics and statistics, efficient algorithms, methodology development, multiple testing, NP-complete problems

- ▶ Anjali Jha,

✉ anjalijha@hsph.harvard.edu

- ▶ Intekhab Hossain

✉ ihossain@g.harvard.edu

- ▶ Christoph Lange (Building 1, 419)

✉ langech2007@gmail.com

research interests: statistical genetics and bioinformatics, development of statistical methodology, software implementation of statistical methods, analysis of large-scale genetic studies

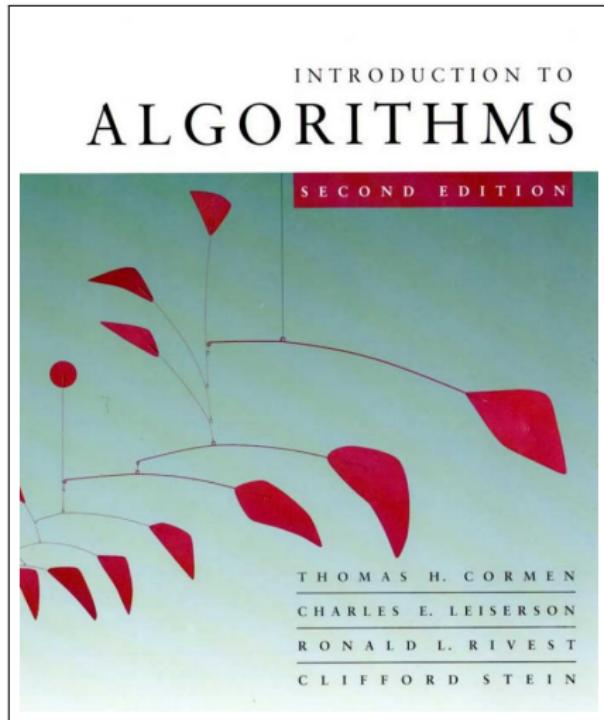
Office hours: tba

Who are you?

Goals of the course

- ▶ Understand complex data structures and their efficient implementation in software.
- ▶ Understand algorithms for data handling and data manipulation.
- ▶ Understand mathematical standards of numerical analysis and statistics, and their implementations.
- ▶ Knowledge of Python is vital (lab sessions will ensure this).

Text book for this course



"Introduction to Algorithms"
by Cormen, Leiserson, Rivest, Stein (MIT Press)

Course structure

1. Lectures cover the theory part: every Monday and Wednesday
2. Labs: programming and homeworks
3. Lab times: tba
4. Homework: 6 assignments
5. Grading: 40% homework assignments, 25% mid-term, 25% final project, 10% class participation

Contents of the course

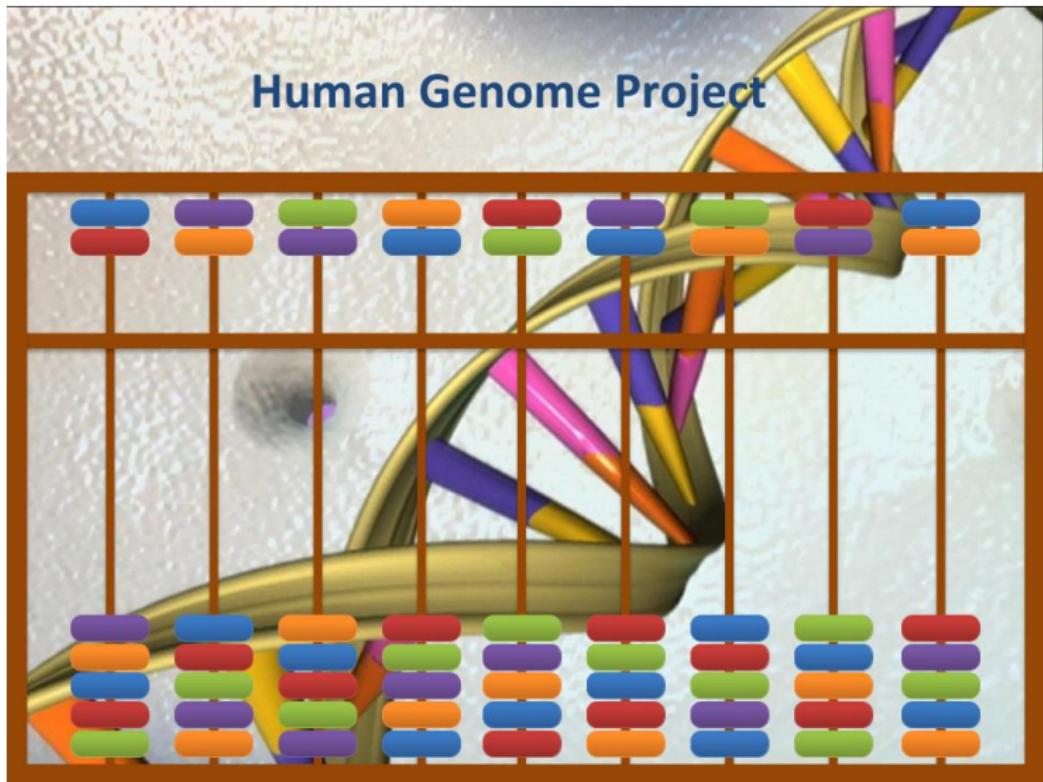
1. Introduction, random numbers
2. Concepts of algorithms, complexity and sorting algorithms
3. Data structures and heapsort
4. Greedy algorithms and dynamic programming
5. Greedy algorithms and dynamic programming (continued)
6. Introduction to parallel programming
7. Numerical aspects of computer algorithms, condition of a problem, numerical stability of an algorithm, numerical error, forward/backward-error
8. Efficient algorithms for linear algebra
9. Efficient algorithms for linear algebra (continued)
10. Least-Squares problem, eigenvalue decomposition
11. Numerical integration, Monte Carlo integration, importance sampling
12. Numerical optimization
13. Work on projects
14. Project presentation

midterm is during spring break

Quick overview/tour of the course

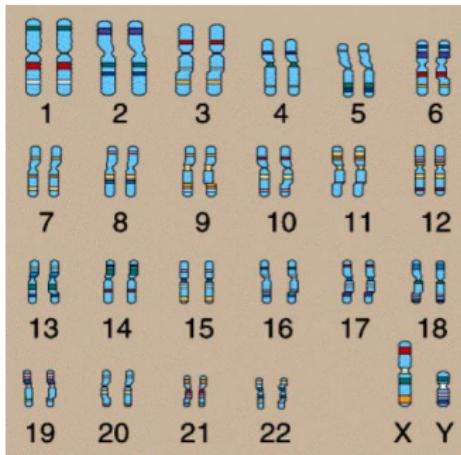
1. Algorithms
2. Data Structures
3. Numerical Aspects of Algorithm

Personal motivation for work in this field



The Human Genome Project

How to find disease genes in the human genome?



Difficult task:

- ▶ 22 chromosomes
- ▶ ~ 23,000 genes
- ▶ about 3 billion base-pairs

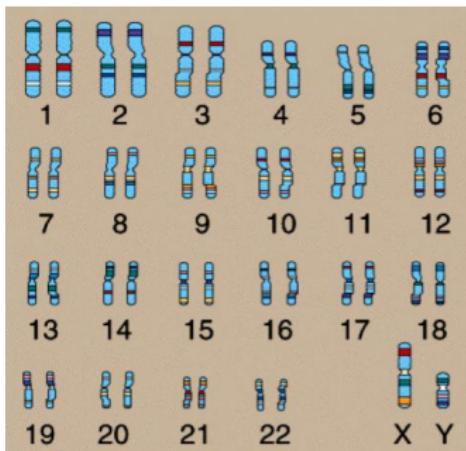
The Human Genome Project

Basic analysis:

- ▶ Native test for genetic association
- ▶ One locus with A and T alleles
- ▶ Case/control study:

	AA	AT	TT
Cases	500	250	52
Controls	511	190	38

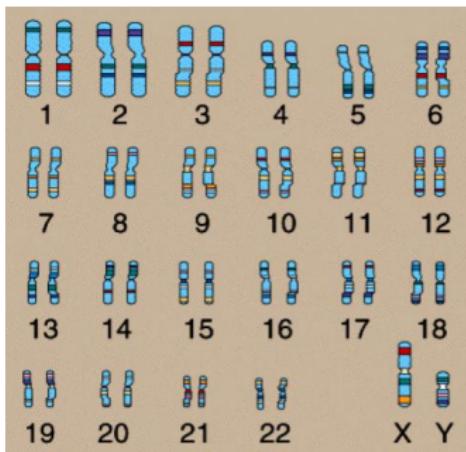
- ▶ Typical statistical analysis:
Amitrage-Trend test, χ^2 -test



The Human Genome Project

Technology development:

- ▶ Revolution in genotyping technology
- ▶ Before 2004: a couple of hundred markers for linkage studies
- ▶ 2004: 100,000 SNPs genome-wide
- ▶ 2006: 500,000 SNPs genome-wide
- ▶ 2008: 2,000,000 SNPs genome-wide
- ▶ 2010/11: high-throughput sequencing technology with up to 3 billion base-pairs



The Human Genome Project

Standard data storage format: LINKAGE format

- ▶ First line: list of all genetic loci
- ▶ For each study subject, one line
- ▶ Columns:

1 pedigree id

2 subject id

3 id of the mother (=0 if missing)

4 id of the father (=0 if missing)

5 gender (=1 male, =2 female)

6 affection status (=0 NA, =1 unaffected, =2 affected)

7 and 8 the two alleles for the first genetic locus ("A1")

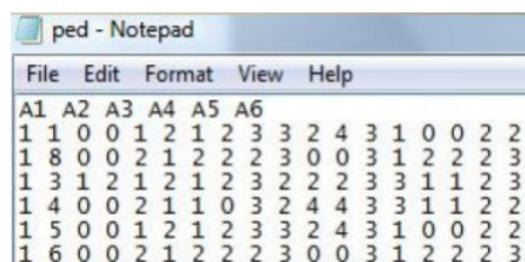
9 and 10 the two alleles for the second genetic locus ("A2")

A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
1	1	0	0	1	2	1	2	3	3
1	8	0	0	2	1	2	2	2	3
1	3	1	2	1	2	1	2	2	3
1	4	0	0	2	1	1	0	3	2
1	5	0	0	1	2	1	2	3	2
1	6	0	0	2	1	2	2	3	0

The Human Genome Project

Standard data storage format: LINKAGE format

- ▶ Storage requirements for each genetic locus:
"1 2" = 4 characters, 4 bytes
- ▶ Human genome: ~ 3 billion loci = $1.2 \cdot 10^{10}$ bytes or 11.17 GB
- ▶ Typical study size: 1000+



A1	A2	A3	A4	A5	A6
1	1	0	0	1	2
1	8	0	0	2	1
1	3	1	2	1	2
1	4	0	0	2	1
1	5	0	0	1	2
1	6	0	0	2	1
2	1	2	3	3	2
2	2	2	2	0	0
2	2	2	2	3	1
2	2	2	2	3	1
2	2	2	2	3	1
2	2	2	2	0	0
2	2	2	2	3	1
2	2	2	2	2	2

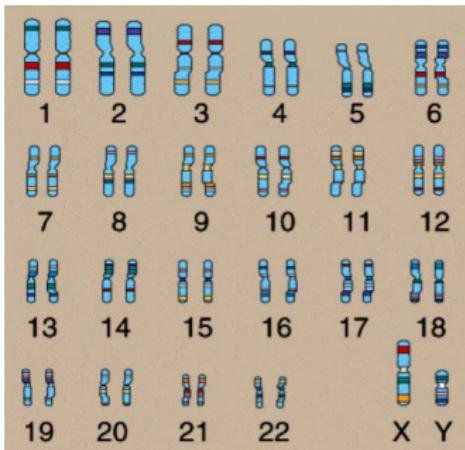
Bytes to Gigabytes:

- ▶ 1 Byte = 8 Bit
- ▶ 1 Kilobyte = 1024 Bytes
- ▶ 1 Megabyte = 1048576 Bytes
- ▶ 1 Gigabyte = 1073741824 Bytes

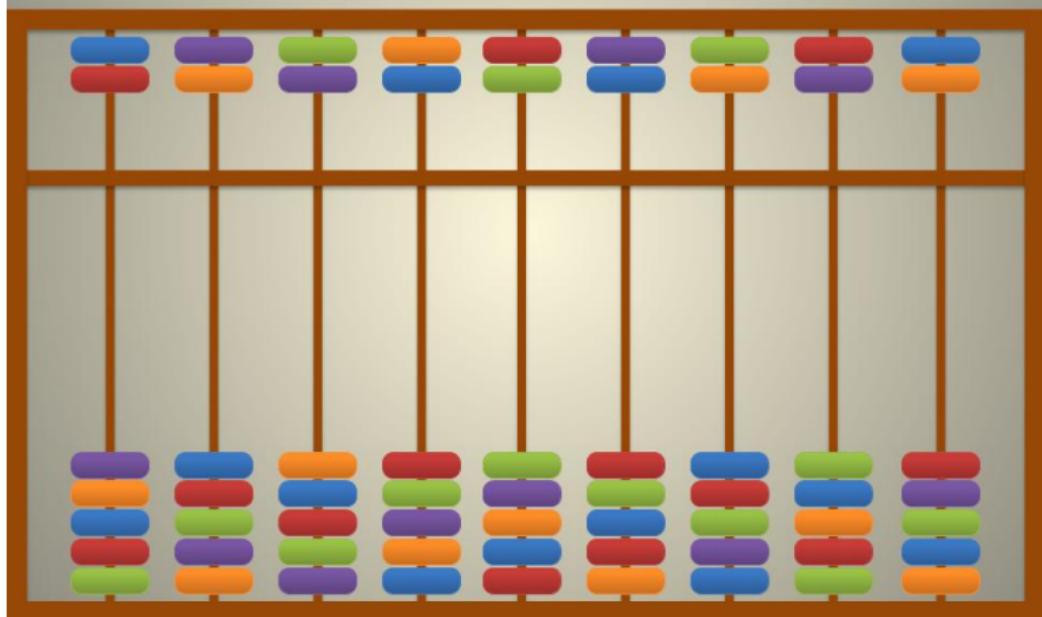
The Human Genome Project

Challenges:

- ▶ Data storage and manipulation in extremely large files
- ▶ Data analysis of many million/billion genetic loci:
typical size of genetic association studies is 1,000 to 100,000 study subjects



Numerical Aspects of Algorithm



Algorithms

- ▶ **Definition** (wikipedia):
"In mathematics and computer science, a **algorithm** is an effective method expressed as a finite list of well-defined instructions for calculating a function."
- ▶ Algorithm = "tool for solving a well-defined computational problem"
- ▶ Algorithm f : input → output

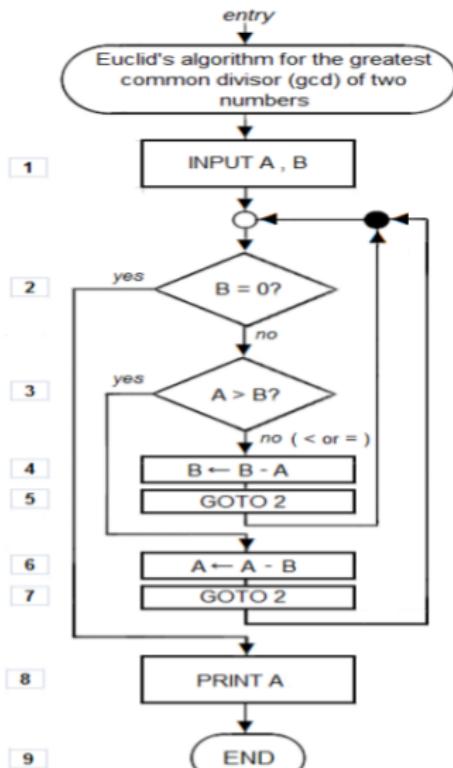
Example: Sorting numbers

- ▶ **Input:** set of numbers $\langle a_1, a_2, \dots, a_n \rangle$
- ▶ **Output:** Sorted sequence of input data $\langle a'_1, a'_2, \dots, a'_n \rangle$ with $a'_1 \leq a'_2 \leq \dots \leq a'_n$

A sorting algorithm f solves the problem defined by the input-output relationship.

Example: Euclid's algorithm

- ▶ Flow chart of an algorithm for calculating the greatest common divisor (g.c.d.) of two numbers a and b in locations named A and B .
- ▶ The algorithm proceeds by successive subtractions in two loops:
 - ▶ If the test $B \leq A$ yields "yes" (or true) – or more accurately, if the number b in location B is less than or equal to the number a in location A – then the algorithm specifies $B \leftarrow B - A$ (meaning the number $b - a$ replaces the old a).
 - ▶ Similarly if $A > B$ then $A \leftarrow A - B$.
 - ▶ The process terminates when (the contents of) B is 0, yielding the g.c.d. in A .



Exaples: Algorithms

- ▶ Sorting algorithms
- ▶ Optimization problems:
 - ▶ Traveling Salesman: chip design, airline schedule
- ▶ Mathematical problems:
 - ▶ Solution of linear equations/ matrix algebra
 - ▶ Integration
- ▶ Statistical problems:
 - ▶ Optimal designs/ power calculations
 - ▶ Computation of distributions
 - ▶ Computation of test statistics

Properties of algorithms

► **Correctness of an algorithm:**

- ▶ The algorithm terminates for valid instances after a **finite time**, providing the output values defined by the input-output relationship.
- ▶ If the algorithm is incorrect, the algorithm **does not terminate** or it provides **incorrect output values**.

► **Efficiency of an algorithm:**

- ▶ Defined by the requirements for **storage space and computing time**.
- ▶ Its complexity with respect to the number of input elements, i.e. how does computing time increase if the number of input elements is increased.

Example for efficiency: sorting algorithms

- ▶ Importance of correctness is intuitively clear
- ▶ Importance of efficiency: limited resources

Example:

- ▶ Sorting algorithm A requires n^2 steps for n elements
- ▶ Sorting algorithm B requires $n \log(n)$ steps for n elements
- ▶ Computer 1: 10^9 steps per second
- ▶ Computer 2: 10^7 steps per second
- ▶ Instance has 10^6 elements
- ▶ Required computing time:

	Algo A	Algo B
Computer 1	17 minutes	0.02 seconds
Computer 2	2 d 8 hours	2 second

Algorithms (cont.)

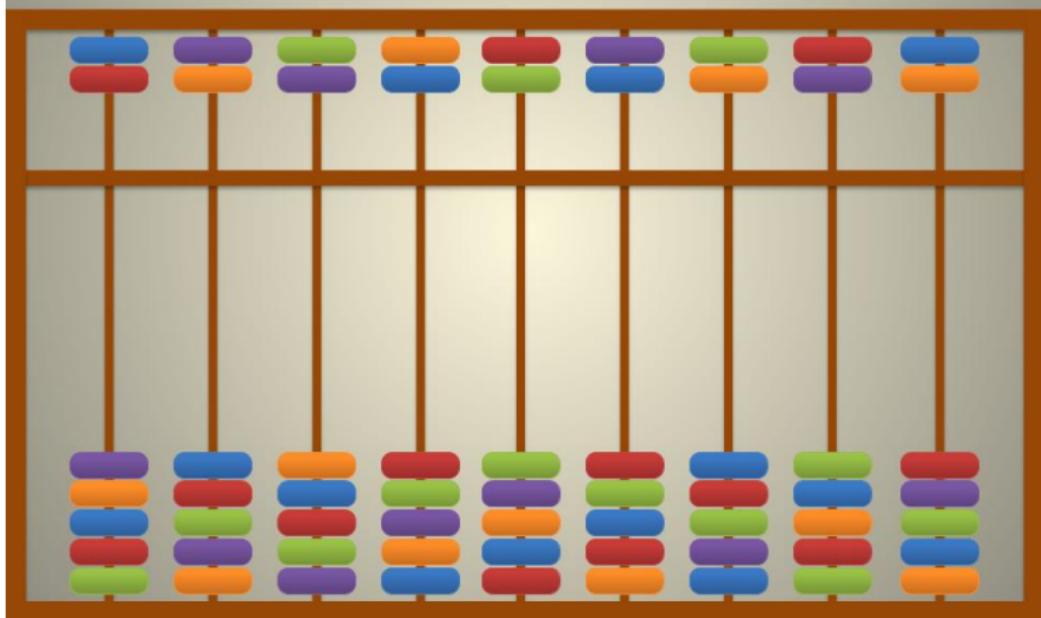
- ▶ **Feasibility:** Every step must be feasible.
- ▶ **Termination:** The algorithm terminates after a finite number of steps.
- ▶ **Deterministic:**
 - ▶ The output is well-defined for every instance
 - ▶ At any time, the next step is well-defined
- ▶ **Finite:**
 - ▶ The number of steps must be finite.
 - ▶ At any time point, the required memory must be finite.

Summary: Algorithms

- ▶ Precise, finite description of a procedure/ function (set of steps)
- ▶ Well-defined input-output relationship
- ▶ Correctness and efficiency are important features for the analysis of algorithms
- ▶ Complexity of algorithms is the major focus of research
- ▶ Runtime complexity is dominated by the increase in the size of the input, e.g. sorting algorithms

Data structures

Data Structures

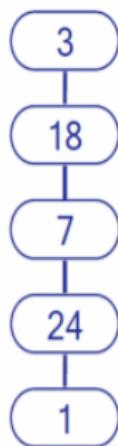


Data structures

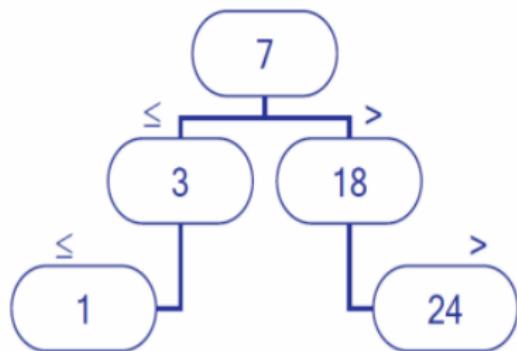
- ▶ **Algorithms** work on dynamic sets of elements (input → output):
 - ▶ Searching, inserting and deleting elements
 - ▶ Maximum, minimum element
- ▶ **Data structures** are used for the implementation of dynamic datasets. The efficiency of a data structure can depend on the desired manipulation operation (e.g., search, insert).
- ▶ The design of data structures should be driven by enabling an efficient implementation of the operations that are relevant in an algorithm.

Data structures: motivation

Alternative representation of a set of numbers:



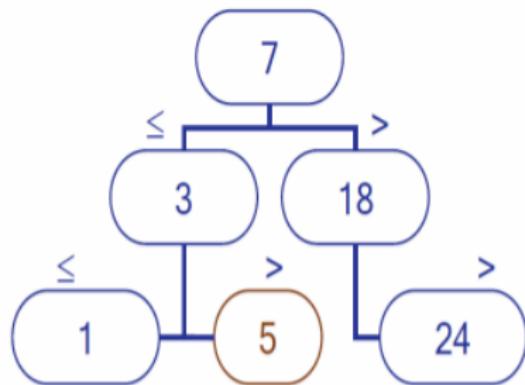
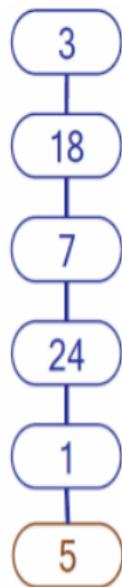
list



tree

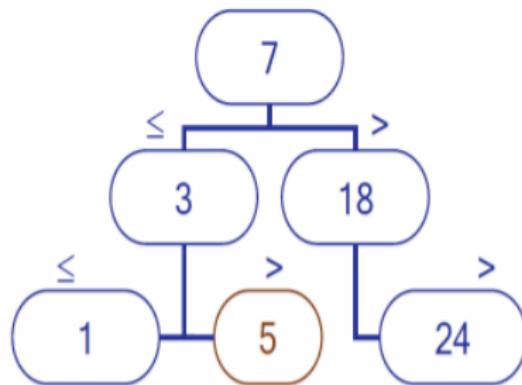
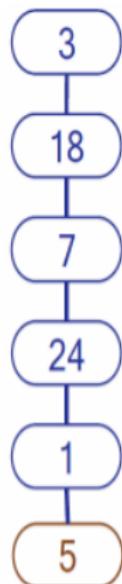
Motivation: insert number 5

More complex operation in a binary tree:



Motivation: search for number 5

More efficient operation in a binary tree:



Examples of data structures

- ▶ **Array:** access to elements based on an index
- ▶ **Linked list:** reference/ pointer to the next element
- ▶ **Stack:** dynamic set of elements, can only be read starting from the most recently added element (*LIFO: Last In First Out*)
- ▶ **Queue:** dynamic set of elements, can only be read starting from the most longest added element (*FIFO: First In First Out*)
- ▶ **Graphs or Trees:** elements have references/ pointers to a variable number of other elements

Example: More efficient data structure for SNP data

- Storage of genetic loci & how to modify it for use in assoc. study
- early 2000s, typical linkage format seen here
 - ▶ Idea: For any genetic locus A, there are three possible genotypes, e.g. AA, AT and TT, or the data can be missing
 - ▶ In total, the genetic coding at any genetic locus can be described by 1 out of 4 possible states \Rightarrow The information can be stored in 2 bits.

e.g. 1000 kids affected by asthma,
1000 unaffected, comparing specific loci

"poor file format"

OLD

labels for each genetic marker

ped - Notepad

A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11
1 1	0	0	1	2	1	2	3	3	2	4
1 8	0	0	2	1	2	2	2	3	0	0
1 3	3	1	2	1	2	3	2	2	2	3
1 4	0	0	2	1	1	0	3	2	4	4
1 5	5	0	0	1	2	1	2	3	3	1
1 6	6	0	0	2	1	2	2	2	3	0

↑ pedigree ID
↑ patient ID
other ID
parentInfo

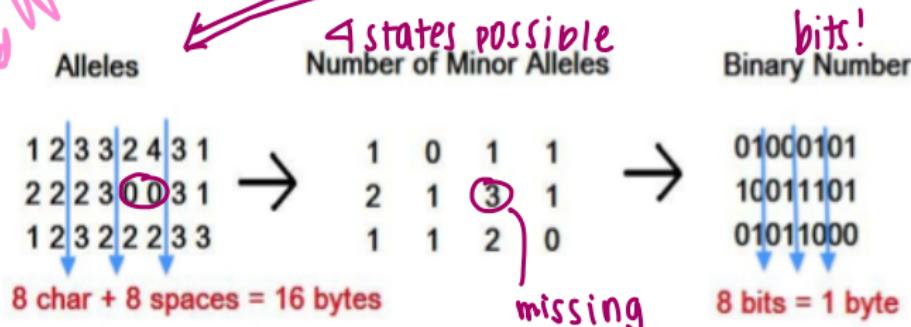
marker alleles

* see slide 13

genotype 1
for marker A5

Example: More efficient data structure for SNP data

New



- ▶ Storage requirements for each genetic locus: "1 2" = 2 bits = 1/4 byte
- ▶ Human genome: about 3 billion loci = $1/4 \cdot 3 \cdot 10^9$ byte = 0.69 GB instead of the previously required 11.7 GB
- ▶ Typical study size: 1000+

motivation: better data structures. Maybe specific to analysis required.

Context example: 1 genetic locus, disease status 0 or 1

ID	locus	y = disease	$x = \# B \text{ alleles} / \text{minor alleles}$
1	AA	0	0
2	AB	0	1
3	AA	1	0
4	AA	0	0
5	AB	1	1
6	BB	0	2

Goal: association test btwn loci & disease, min mem & comp

$y[i]$ is phenotypic information (disease status) for patient i

$$T \propto \sum_{i=1}^n f_i(y[i]) \cdot x[i] \quad \text{if } x[i] \text{ is genotype of } i^{\text{th}} \text{ patient}$$

f_i is some transformation for patient i

e.g. adjustment for covariates, typically age & sex

multiply to look at correlation. "crudely speaking" *

$$= \sum_{x=0}^2 \sum_{j: x[j]} = x f_j(y[j]) \cdot x[j]$$

$$= \sum_{x=0}^2 x \sum_{j: x[j]} = x f_j(y[j])$$

$$= \sum_{x=1}^2 x \sum_{j: x[j]} = x f_j(y[j])$$

* drops all patients w/ no minor allele, that's a lot of patients so can make a huge difference

this format allows us to store chunks of data together
mongo group aggregate combine?

Improve speed more by storing $\text{ind}_1 = [2 5]$ and $\text{ind}_2 = [6]$, indices corresponding to individuals with genotype 1 & 2

$$n_1 = \text{length}(\text{ind}_1(j)), n_2 = \text{length}(\text{ind}_2(j))$$

$$= \sum_{j=1}^{n_1} f_{ind_1[j]}(y[ind_1[j]]) + 2 \sum_{j=1}^{n_2} f_{ind_2[j]}(y[ind_2[j]])$$

Big improvement in storage & computational effort!

TO DO: experiment in code & see how much storage requirement changes

$$n = \text{sample size} \quad p = \text{maf} \in (0, \frac{1}{2})$$

$$X[i] \sim \text{Bin}(2, p)$$

$$Y[i] \sim \text{Bin}(1, 1/2)$$

$$T_1 = \sum X[i] Y[i]$$

$$T_2 = \sum_{j=1}^{n_1} f_{ind_1[j]}(y[ind_1[j]]) + 2 \sum_{j=1}^{n_2} f_{ind_2[j]}(y[ind_2[j]])$$

compare runtime as a function of n & p

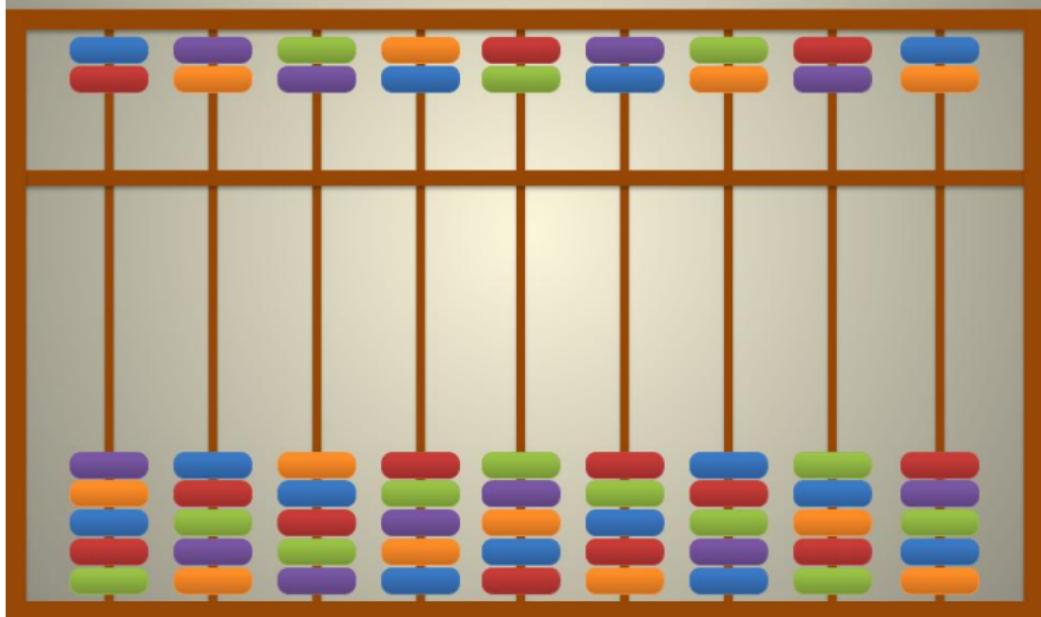
compare mem

compare overhead (?)

Just to think about the goal of the course, combined CS & biostat perspective.

Numerical Aspects of algorithms

Numerical Aspects of Algorithm



Numerical Aspects of Algorithms

In addition to complexity, requirements on CPU time and memory we also care about the following:

The **accuracy** of the results calculated by the algorithm and the **numerical stability** of the algorithm are important aspects of quantitative analysis.

Numerical Aspects of Algorithms

One can distinguish between different types of errors that affect accuracy:

- ▶ errors in the model
- ▶ errors in the input data
- ▶ round-off errors
- ▶ approximation errors/numerical method
- ▶ errors due to arithmetic, e.g. caused by operation such as + or *

errors can come from how we represent numbers - rounding errors
Floating point presentation of numbers: Definition

A **normalized floating point number** is a number $x \in \mathbb{R}$ that can be written as

$$x = \pm m * b^{\pm e}$$

with

- ▶ the **base** $b \in \mathbb{N}$ and $b > 1$
- ▶ the **mantissa** $m = m_1 b^{-1} + \dots + m_r b^{-r}$
- ▶ the **exponent** $e = e_{s-1} b^{s-1} + \dots + e_0 b^0$
- ▶ the **digits** $m_i, e_j \in \{0, \dots, b-1\}$
- ▶ the number of **significant digits** $s \in \mathbb{N}$ and $r \in \mathbb{N}$
- ▶ **normalized** $m_1 \neq 0$

Typical choices for base b

- ▶ $b = 10$ for humans
- ▶ $b = 2$ for computer

Floating point presentation of numbers: Example

Write the number $x = 12.85$ in floating-point format for the base $b = 2$:

Algorithm:

1. Find k so that: $2^k \leq x < 2^{k+1}$

Solution: $k = 3$

2. Then divide x by $8 = 2^3$. Compute remainder/modulo and set to x

Solution: 1 and $x = 12.85 - 8 = 4.85$

3. If $x > 0$ go to Step 1

Then x can be written as

$$12.85 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + \dots$$

After normalization (i.e., division by 2^4)

$$12.85 = (0.110011011\dots)_2 \cdot 2^4.$$

Example: The Patriot Missile Failure

On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people.

The time in tenths of second as measured by the system's internal clock was multiplied by $1/10$ to produce the time in seconds. This calculation was performed using a 24 bit fixed point register. In particular, the value $1/10$, which has a non-terminating binary expansion, was chopped at 24 bits after the radix point. The small chopping error, when multiplied by the large number giving the time in tenths of a second, led to a significant error.

Example: The Patriot Missile Failure (cont)

Indeed, the Patriot battery had been up around 100 hours, and an easy calculation shows that the resulting time error due to the magnified chopping error was about 0.34 seconds.

$$1/10 = 1/24 + 1/25 + 1/28 + 1/29 + 1/212 + 1/213 + \dots$$

In other words, the binary expansion of $1/10$ is

0.0001100110011001100110011001100....

Now the 24 bit register in the Patriot stored instead
0.00011001100110011001100 introducing an error of
0.0000000000000000000000000011001100... binary, or about
0.000000095 decimal. Multiplying by the number of tenths of a
second in 100 hours gives $0.000000095 \cdot 100 \cdot 60 \cdot 60 \cdot 10 = 0.34$.

A Scud travels at about 1,676 meters per second, and so travels
about 570 meter= about 1900 foot in this time.

instead of '1/10'

How to avoid - all in 64-bit, OR used $'1/8$ or $'1/10$ sec⁹ instead?

Example: The Explosion of the Ariane 5

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing 7 billion USD. The destroyed rocket and its cargo were valued at 500 million USD.

A board of inquiry investigated the causes of the explosion and in two weeks issued a report. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,767, the largest integer storable in a 16 bit signed integer, and thus the conversion failed.

Contents:

1. Pseudo-random number generators
2. Desired properties and drawbacks
3. Current gold standard

Today's topic: random numbers

1. Statistical simulation (Monte Carlo) is an important part of statistical method research.
2. The statistical theories/methods are all based on assumptions. So most theorems state something like if the data follow these models/assumptions, then ...
3. The theories can hardly be verified in real world data because (1) the real data never satisfy the assumption; and (2) the underlying truth is unknown (no gold standard).
4. In simulation, data are created in a well controlled environment (model assumptions) and all truth are known. So the claim in the theorem can be verified and its sensitivity to assumptions can be assessed.

One aspect: Random number generation

1. Random number generator is the basis of statistical simulation. It serves to generate random numbers from predefined statistical distributions which are required to conduct permutation testing or simulation studies.
2. In the "big data" area, often large scale simulation studies are required to study the behavior of models or to assess significance.
3. We need a fast and reliable mechanism/algorithm to generate large sets of random numbers ($> 10^6$)

Pseudo-random number generator (PRNG)

It is important to be able to efficiently generate independent random variables from the uniform distribution on $(0,1)$, since:

1. Random variables from all other distributions can be obtained by transforming uniform random variables;
2. Simulations/statistical modelling techniques require extremely large sets of random numbers that have to be generated computationally fast.

Pseudo-random number generator (PRNG)

Pseudo random numbers:

1. Typically integer number between 0 and m
2. Deterministic sequence of numbers, e.g. $z_{n+1} = f(z_n)$
3. Looks iid/random, i.e. they pass statistical test for randomness vs. real random numbers

NOT random, just looks random

A good random-number generator

A good random-number generator should satisfy the following properties:

1. **Uniformity**: The numbers generated appear to be distributed uniformly on $(0, 1)$
2. **Independence**: The numbers generated show no correlation with each other
3. **Diehard tests**: Passes all Diehard tests for randomness/qc-standard tests for PRNG
4. **Replication**: The numbers should be replicable (e.g., for debugging or comparison of different systems)
5. **Cycle length**: It should take long before numbers start to repeat
6. **Speed**: The generator should be fast
7. **Memory usage**: The generator should not require a lot of storage
8. **Parallel implementation**
9. **Cryptographically secure**

Pseudo-random number generator (PRNG)

Most random-number generators are of the form:

- ▶ Start with z_0 (called "seed")
- ▶ For $n = 1, 2, \dots$ generate

$$z_n = f(z_{n-1})$$

$$u_n = g(z_n)$$

where

- ▶ f is the pseudo-random number generator
- ▶ g is the output function

Then, u_0, u_1, \dots is the sequence of random numbers on the interval $(0, 1)$.

Mid-square method

$$z_n = \text{mid4}(\text{pad}(z_{n-1}^2))$$

$$u_n = z_n / 1000$$

- ▶ Start with a 4-digit number z_0 (seed)
- ▶ Square it to obtain 8-digits (if necessary, append zeros to the left)
- ▶ Take the middle 4 digits to obtain the next 4-digit number z_1 , then square z_1 and take the middle 4-digits again and so on.

We get uniform random number by placing the decimal point at the left of each z_i (i.e., divide by 10000).

Mid-square method

$$1234^2 = 1522756$$

padded = 01522756

$$\begin{array}{r} \underline{1234} \\ 1000 \\ \downarrow \end{array}$$
$$\begin{array}{r} \underline{5227} \\ 1000 \\ \downarrow \end{array}$$

Examples:

- ▶ For $z_0 = 1234$ we get: 0.1234, 0.5227, 0.3215, 0.3362, 0.3030, 0.1809, 0.2724, 0.4201, 0.6484, 0.0422, 0.1780, 0.1684, 0.8361, 0.8561, 0.2907, ...
- ▶ For $z_0 = 2345$ we get: 0.2345, 0.4990, 0.9001, 0.0180, 0.0324, 0.1049, 0.1004, 0.0080, 0.0064, 0.0040, ...

Two successive zeros behind the decimal will never disappear.

- ▶ For $z_0 = 2100$ we get: 0.2100, 0.4100, 0.8100, 0.6100, 0.2100, 0.4100, ...

Already after four numbers the sequence starts to repeat itself.

Clearly, random-number generators involve a lot more than doing something strange/funny to a number to obtain the next.

Linear congruential generator (LCG) commonly used

LCG is an algorithm that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation. They produce a sequence of integers between 0 and $m - 1$ according to

$$z_n = (a * z_{n-1} + c) \bmod m, \quad n = 1, 2, \dots$$

where a is the multiplier, c the increment and m the modulus. To obtain uniform random numbers on $(0, 1)$, we take

$$u_n = z_n / m.$$

Good choices for the parameter a, c and m are important.

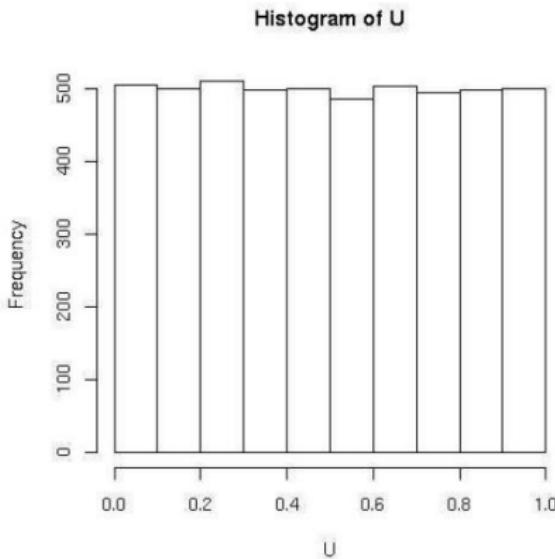
Properties of LCG results

The numbers z_i have the following properties:

1. $z_i \in \{0, 1, 2, \dots, m - 1\}$
2. z_i are periodic with period $p < M$. Indeed, since at least two values in $\{z_0, z_1, \dots, z_m\}$ must be identical, therefore $z_i = z_{i+p}$ for some $p \dots M$.

In view of property 2 the number m should be as large as possible, because a small set of numbers make the outcome easier to predict. But still p can be small even if m is large!

Example results of LCG



5000 uniform draws, $a = 1229$, $c = 1$, $m = 2048$, $z_0 = 1$

Independence of LCG results

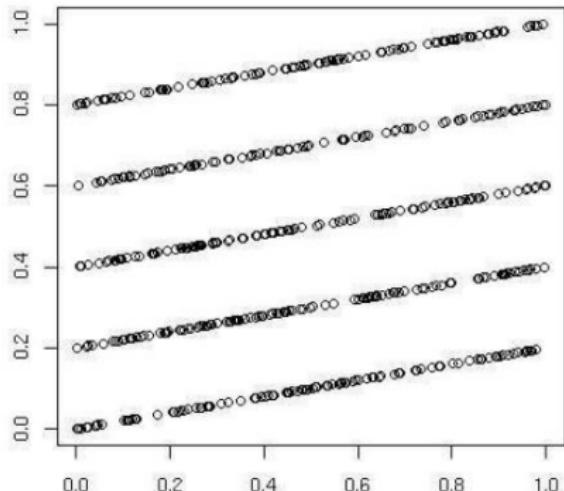
1. Are the numbers z_i obtained with the congruential linear generator uniformly distributed? It seems so, as the histogram is quite flat. ✓
2. Does this really mean that the numbers z_i are good uniform deviates? Are they independent? ✗
3. To study this, consider vectors

$$(z_i, z_{i+1}, \dots, z_{i+m-1}) \in [0, 1]^m$$

and analyze them with respect to their distribution. Good uniform random numbers should be uniformly distributed in $[0, 1]^m$.

Independence of LCG results

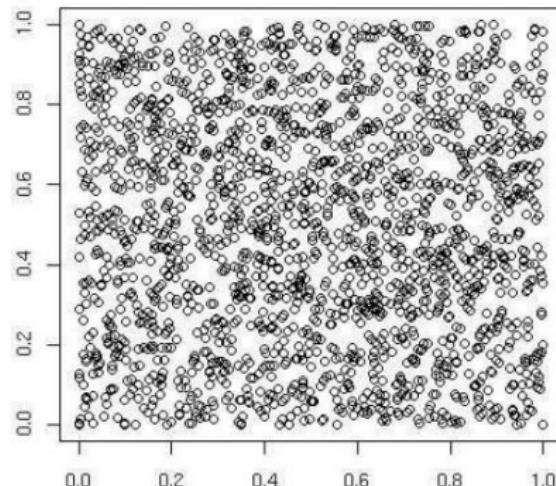
Plot pairs (z_i, z_{i+1}) for the linear congruential generator with $a = 1229, c = 1, m = 2048$:



bad!
Not indep!

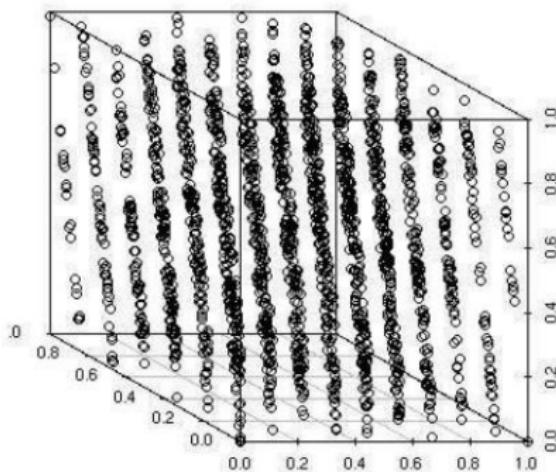
Independence of LGC results

Seemingly good congruential generator with $a = 2^{16} + 3$, $c = 0$, $m = 2^{31}$:



Independence

However, the generator with $a = 2^{16} + 3$, $c = 0$, $m = 2^{31}$ looks as follows in 3d:



In general, Marsaglia showed that the m -tuples (z_i, \dots, z_{i+m-1}) generated with linear congruential generators fall on a relatively low number of hyperplanes in \mathbb{R}^m . This is a major disadvantage of linear congruential generators.

Linear congruential generator (LCG)

The theory behind LCG is relatively easy to understand, and they are easily implemented and fast, especially on computer hardware which can provide modulo arithmetic by storage-bit truncation. A linear congruential generator has full period (=cycle length m) if and only if the following conditions hold:

1. The only positive integer number that exactly divides both m and c is 1 (the gcd); **gcd of $m \& c$ is 1**
2. If q is a prime number that divides m , then q divides $a - 1$;
3. If 4 divides m , then 4 divides $a - 1$.

Other approaches

Multiplicative congruential generators

These generators produce a sequence of integers between 0 and $m - 1$ according to

$$z_n = (az_{n-1}) \bmod m, \quad n = 1, 2, \dots$$

So they are linear congruential generators with $c = 0$. They cannot have full period, but it is possible to obtain period $m - 1$ (so each integer $1, \dots, m - 1$ is obtained exactly once in each cycle) if a and m are chosen carefully. For example, $a = 630360016$ and $m = 2^{31} - 1$ is a good choice.

Other approaches

Additive congruential generators

These generators produce integers according to

$$z_n = (z_{n-1} + z_{n-k}) \bmod m, \quad n = 1, 2, \dots$$

where $k > 2$. Uniform random numbers can again be obtained as $u_n = z_n/m$ as before. These generators can have a long period up to m^k .

Disadvantage: Consider the case $k = 2$ (the Fibonacci generator). If we take three consecutive numbers u_{n-2}, u_{n-1} and u_n , then it will never happen that $u_{n-2} < u_n < u_{n-1}$ or $u_{n-1} < u_n < u_{n-2}$, whereas for true uniform variables both of these orderings occur with probability 1/6.

Mersenne Twister

- ▶ Current gold standard
- ▶ <http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/emt.html>
- ▶ <https://www.mersenne.org/primes/>
- ▶ Very long period: $2^{19937} - 1 = 10^{19937 \cdot \log_{10}(2)} - 1 \approx 10^{6001}$
- ▶ Parallel implementations available *mersenne prime*
- ▶ Passes all Diehard tests
- ▶ Default PRNG in many software packages/programming languages

keep starting point

Concepts of algorithms

Contents:

1. Concepts of algorithms: What is an algorithm? What flavors of algorithms exist?
2. Complexity in time and space
3. Recursion and divide-and-conquer *← types*
4. Example: sorting algorithms

What is an algorithm?

- ▶ An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- ▶ An algorithm is thus a sequence of computational steps that transform the input into the output.
- ▶ many flavors: deterministic, randomized, sequential, recursive, polynomial, exponential runtime, etc.
- ▶ Example (formal definition of the sorting problem):
 - Input: A sequence of n numbers a_1, a_2, \dots, a_n .
 - Output: A permutation (reordering) $a_{(1)} \leq \dots \leq a_{(n)}$.
- ▶ Each specific input an algorithm is applied to is called an *instance of a problem*.

Instance of an algorithm

Definition: *An instance or implementation of an algorithm is a specific set of instructions and environment that:*

- 1. satisfies all constraints of the problem statement,*
- 2. contains all the information to apply the algorithm (correctly!).*

Example: Sorting problem

An instance is a specific list of numbers to be sorted, e.g.

{5, 3, 2, 6, 4, 8, 9, 1, 7}

Many examples of algorithms in real life

1. Displaying relevant results in internet search engines.
2. Public-key cryptography algorithms for electronic commerce.
3. Optimization algorithms to streamline train routes, flight times, etc.

More examples

- ▶ Sorting algorithms
- ▶ Optimization problems:
 - ▶ Inference
 - ▶ Classification
 - ▶ Route finding
 - ▶ Travelling salesman: chip design, transportation scheduling
- ▶ Mathematical problems:
 - ▶ Solution of linear equations/matrix algebra
 - ▶ Integration
- ▶ Statistical problems:
 - ▶ Optimal designs/power calculations
 - ▶ Computation of distributions
 - ▶ Computation of test statistics

Algorithms: Things to keep in mind

- ▶ **Feasibility:** Every step must be feasible.
- ▶ **Termination:** The algorithm terminates after a finite number of steps. (Technically this can be generalized to probabilistic algorithms: *Monte Carlo algorithms* give probabilistic answers, *Las Vegas algorithms* give deterministic answers but have a random runtime which may be infinite.) *means finite steps & finite mem & correct result*
- ▶ **Determinism:**
 - ▶ Output is well-defined for every instance.
 - ▶ At any time, the next step is well-defined (but it can depend on a coin flip).
- ▶ **Finiteness:**
 - ▶ Number of steps must be finite.
 - ▶ At any time point, the required memory must be finite.

Properties of an algorithm

1. Correctness of an algorithm: *Proofs!*

- ▶ The algorithm terminates for valid instances after a finite time, providing the output values defined by the input-output relationship.
- ▶ If the algorithm is incorrect, the algorithm does not terminate or provides incorrect output values.

2. Efficiency of an algorithm:

- ▶ Defined by the requirements for storage space and computing time.
- ▶ Its complexity with respect to the number of input elements, i.e., how does computing time increase if the number of input elements is increased.

Analysis of algorithms

Algorithms can be analyzed with a variety of aspects in mind. The most common ones are:

1. Correctness: Does the algorithm always produce the correct result?
2. Runtime: What is the best-case/ average-case/ worse-case runtime? Most commonly, the worse-case runtime is reported to account for all application cases, but the average runtime is also useful (especially for problems where many instances are easily solvable, but some require a very high number of operations).

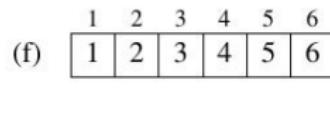
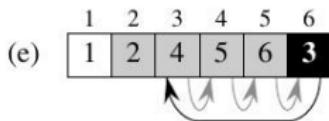
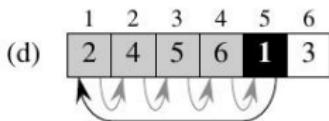
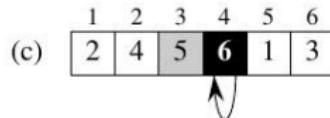
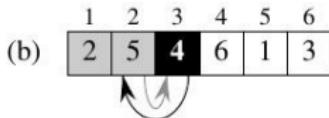
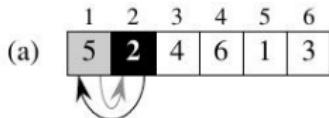
Sorting: insertion sort



$O(n^2)$

- ▶ Given sequence of numbers [5, 2, 4, 6, 1, 3]
- ▶ Idea: Take next unseen element and move it to the appropriate location in the list of seen and sorted elements

Sorting: insertion sort



Steps:

1. The first element only is automatically sorted.
2. (a) Consider the second element (number 2) and sort it into the current sorted list consisting of only the number 5.
3. (b) Consider the next unseen element (number 4) and likewise find its position in the current sorted list [2, 5]
4. (c) Repeat until all elements are considered.

Pseudocode

INSERTION-SORT(A)

```
1  for  $j \leftarrow 2$  to  $\text{length}[A]$ 
2      do  $\text{key} \leftarrow A[j]$ 
3           $\triangleright$  Insert  $A[j]$  into the sorted sequence  $A[1 \dots j - 1]$ .
4           $i \leftarrow j - 1$ 
5          while  $i > 0$  and  $A[i] > \text{key}$ 
6              do  $A[i + 1] \leftarrow A[i]$ 
7                   $i \leftarrow i - 1$ 
8           $A[i + 1] \leftarrow \text{key}$ 
```



Pseudocode of insertion sort (What is a pseudocode...?)

Correctness

One way of proving correctness of a sequential algorithm (such as insertion sort) consists in finding an *invariant* which is always true in each iteration independently of the progress of the algorithm.

Here: What is the invariant statement for insertion sort?

Correctness

One way of proving correctness of a sequential algorithm (such as insertion sort) consists in finding an *invariant* which is always true in each iteration independently of the progress of the algorithm.

Here: What is the invariant statement for insertion sort?

Answer: In iteration i of the insertion sort algorithm, the first i elements of the array are in sorted order. (Proof: By construction of the algorithm, every new element i is inserted among the $i - 1$ sorted elements at the correct position, thus establishing the correct order of the first i elements. Thus after step n , the entire array is sorted.)

1. The runtime or running time of an algorithm (on a particular input) is the number of "primitive" steps (or "operations") which is required to compute the result.
2. The definition of a "primitive step" or "operation" is context dependent: For example, in many cases, the runtime is given as a function of the input length or the number of required multiplications, whereas in the analysis of multiplication algorithms, the runtime is given as a function of the length of the input.
3. Usually, the runtime is given in some sensible measure of the input complexity (for sorting algorithms: the length of the list of numbers to be sorted).
4. Usually, polynomial runtimes are considered "good" (feasible) for practical applications, while exponential runtimes are not.

Runtime of insertion sort

We consider the number of steps in iteration $i \in \{1, \dots, n\}$:

1. In iteration $i \geq 2$, we pick the new unseen number and determine its position among the previously sorted $i - 1$ numbers. For this we have to go through those numbers, which requires at most $i - 1$ steps.
2. Once we found the position, we need to make space by shifting some numbers by one position. In the worst case the new number belongs to the start, in which case we have to move all $i - 1$ numbers.
3. Copying the new number to its position takes constant time.

Together, if T is the number of steps,

$$T = \sum_{i=2}^n 2 \cdot (i - 1) = 2 \sum_{i=1}^{n-1} i = 2 \cdot \frac{(n-1)n}{2} = n^2 - n \approx \underline{\underline{n^2}}$$

Growth of functions and O -notation

We have seen that the dominant term in the runtime/computational effort for insertion sort is n^2 . In general, we are only interested in the dependence of the runtime on the dominant term.

We define for functions $f(n)$ and $g(n)$:

1. Asymptotically tight bound:

$$\Theta(g(n)) = \{f(n) : \text{there exist constants } c_1, c_2 > 0 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}$$

2. "Little-o" (convergence to zero):

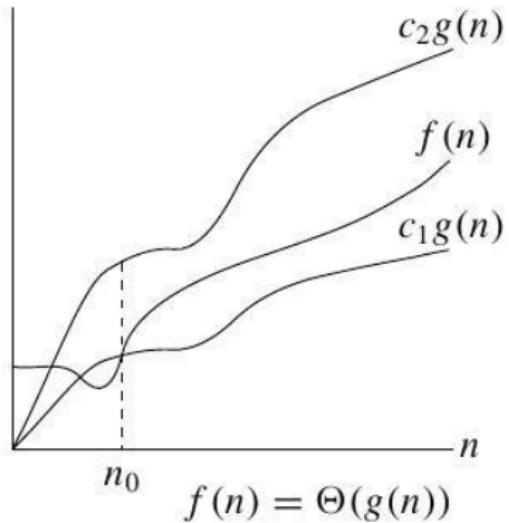
$$o(g(n)) = \left\{ f(n) : \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \right\}$$

goes to 0 in comp to $g(n)$
e.g. $O(n^2) \Rightarrow o(n^3)$

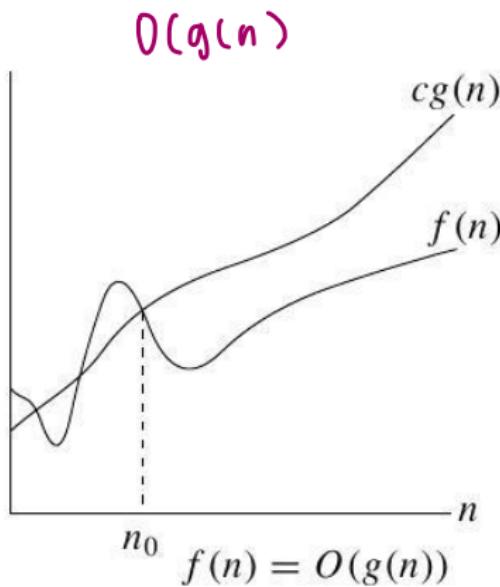
3. "Big-O" (asymptotic upper bound):

$$O(g(n)) = \{f(n) : \text{there exists constants } c > 0 \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \forall n \geq n_0\}$$

O-notation in a picture



(a)



(b)

For insertion sort, the exact runtime (the number of primitive operations) was $T(n) = n^2 - n \in O(n^2)$. We say that insertion sort has a quadratic runtime.

Recursion

- ▶ A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values.
- ▶ Example: Computation of $n!$ ("n factorial")

Sequential algorithm:

```
res = 1;  
for i = 1 to n do  
    | res = res · i;  
end  
return res;
```

How would you do that recursively?

Recursive algorithm to compute n!

```
function factorial(n):  
    if n==0 then return 1  
    else return n*factorial(n-1)
```

► Example:

$$\begin{aligned}\text{factorial}(3) &= 3 * \text{factorial}(2) = 3 * [2 * \text{factorial}(1)] = \\ &6 * [1 * \text{factorial}(0)] = 6\end{aligned}$$

Recursion can allow for very elegant statements of complicated algorithms.

have to store intermediate results somewhere

Divide and conquer paradigm

"Divide and conquer" is a computational approach which uses recursion to solve problems. It involves three steps:

1. **Divide** the problem into a number of subproblems
2. **Conquer** the subproblems by solving them recursively again. However, if a subproblem is small enough it might be faster to not divide it further and to solve it with a "conventional" approach.
3. **Combine** the solutions of all subproblems into the solution of the overall problem.

Mergesort

We use divide and conquer to sort numbers. The Mergesort algorithm is such a method which is based on the following observation:

- ▶ Two sorted lists can be merged into one sorted list in linear time.

8	10	12	44	63	66
Ptr1					

2	3	20	106	144
Ptr2				

2	3	8	10						
Ptr3									

Pseudocode of the Merge step: Notation

- ▶ A is an array
- ▶ p, q and r are indices for the array
 - ▶ $A[p..q]$ is the first sorted sub array
 - ▶ $A[(q + 1)..r]$ is the second sorted sub array

Pseudocode of the Merge step

```
MERGE( $A, p, q, r$ )
1    $n_1 \leftarrow q - p + 1$ 
2    $n_2 \leftarrow r - q$ 
3   create arrays  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$ 
4   for  $i \leftarrow 1$  to  $n_1$ 
5       do  $L[i] \leftarrow A[p + i - 1]$ 
6   for  $j \leftarrow 1$  to  $n_2$ 
7       do  $R[j] \leftarrow A[q + j]$ 
8    $L[n_1 + 1] \leftarrow \infty$ 
9    $R[n_2 + 1] \leftarrow \infty$ 
10   $i \leftarrow 1$ 
11   $j \leftarrow 1$ 
12  for  $k \leftarrow p$  to  $r$ 
13      do if  $L[i] \leq R[j]$ 
14          then  $A[k] \leftarrow L[i]$ 
15               $i \leftarrow i + 1$ 
16          else  $A[k] \leftarrow R[j]$ 
17               $j \leftarrow j + 1$ 
```

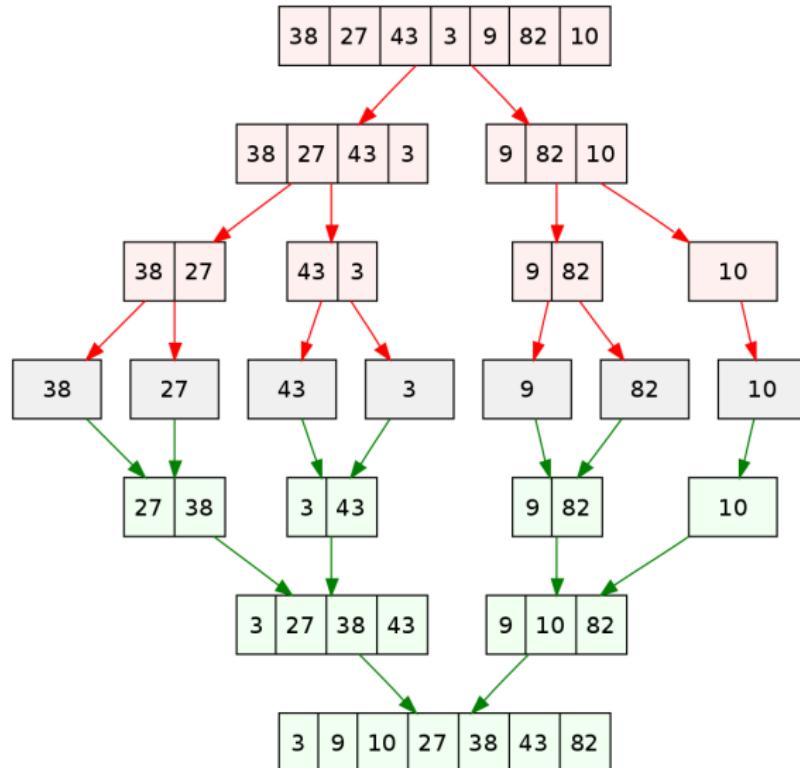
merge is $O(n)$

Idea of Mergesort

Recursive argument:

- ▶ We are given an input list of n numbers (wlog assume n even).
- ▶ We split this list into two lists. The first list contains the first $n/2$ numbers, the second list contains the second $n/2$ numbers.
- ▶ We call Mergesort on both lists of size $n/2$.
- ▶ By induction we can assume that the lists come back sorted.
- ▶ We can combine the two sorted sublists (of size $n/2$ each) into the sorted full list in linear time, i.e. in $O(n)$.

Mergesort in one picture



Algorithm and correctness

Letting L be a list of numbers, the Mergesort algorithm can thus be summarized as:

Mergesort(L):

1. If L has size 1 then **return** L
2. Split L into two sublists L_1, L_2 of equal size in any way
3. $S_1 = \text{Mergesort}(L_1)$
4. $S_2 = \text{Mergesort}(L_2)$
5. **return** $\text{Merge}(S_1, S_2)$

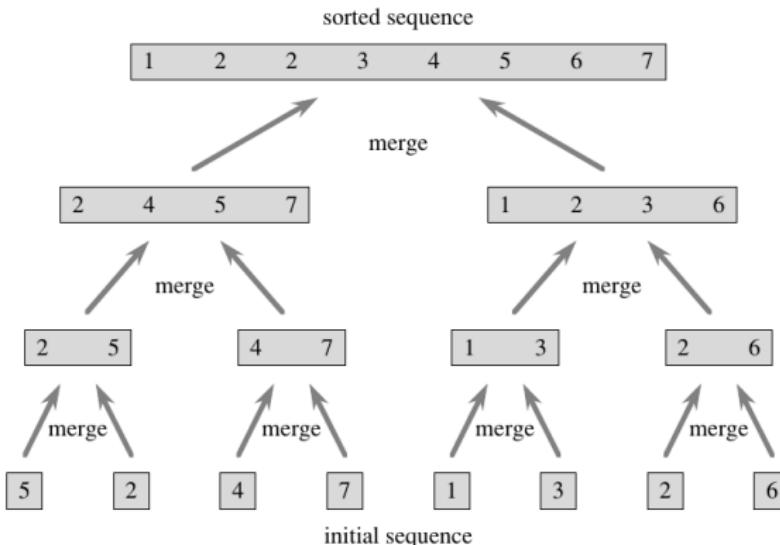


Merge sort is $O(n \log_2 n)$

Correctness by induction:

- ▶ Base case $n = 1$: Mergesort sorts lists of size 1 correctly.
- ▶ Induction hypothesis: Mergesort sorts lists of size $\leq n$ correctly.
- ▶ Inductive step $n \rightarrow n + 1$: After splitting, the two recursive calls of Mergesort on the splitted lists of size less than $n + 1$ work correctly by the induction hypothesis and return two sorted lists. The Merge step then correctly combines both into one sorted list.

Runtime of Mergesort



Three observations:

1. In each recursion layer, each number appears exactly once in some sublist.
2. In each layer, each number is only looked at twice (once when splitting the list and once when merging later), so the runtime per layer is linear.
3. The number of layers is determined by the number of times one can divide a list of n elements into two lists. This is $\log_2 n$.

Runtime of Mergesort

We have seen that Mergesort operates with a recursion of depth $\log_2 n$, and in each layer all n numbers are being looked at. Thus the runtime is $O(n \log_2 n)$.

Another way to see this is to write the runtime as:

$$T(n) = 2T(n/2) + n.$$

This means:

1. The runtime $T(n)$ to sort n numbers is equal to whatever it takes to sort 2x an array of size $n/2$, plus the effort for combining the two sorted sublists in a linear run (this is the $+n$ term).
2. We can solve this recursion by substitution (next slide).

Runtime of Mergesort

We continue to solve the recursion:

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + c \\&= 2\left[2T\left(\frac{n}{4}\right) + n/2\right] + n \\&= 2^2T\left(\frac{n}{2^2}\right) + 2n \\&= \dots \\&= 2^{\log_2 n}T(1) + \log_2(n) \cdot n\end{aligned}$$

since the subdivision of problems into sizes $\frac{n}{2^d}$ continues until the problems reach size 1, which occurs after $d = \log_2 n$ subdivisions. Since $T(1) = 1$ (sorting one number takes constant time), we have

$$T(n) = n + n \cdot \log_2(n) \in O(n \log n).$$

There is a general way to calculate the runtime of recursive algorithms: Please look up the "Master theorem".

More on the efficiency of algorithms

Efficiency of an algorithm is defined by

- ▶ Required number of instructions (processor time)
- ▶ Required memory space
- ▶ Dependence of the time and memory requirements upon the number of input elements

Efficiency of algorithms

- ▶ Execution time of most algorithms is dominated by the computation time for large number of input elements.
- ▶ Hardware and software differences usually play a minor role.
- ▶ Memory requirements can also be important in the analysis of efficiency and complexity.
- ▶ Here we focus (as in most cases) on the time complexity of algorithms.

Machine model for runtime analysis

Machine:

- ▶ With free and unlimited memory access
- ▶ One CPU (normally...)
- ▶ We do not distinguish between different types of memory (e.g. cache)

Important measure is number of primitive steps:

- ▶ Each primitive step of the algorithm requires **the same time**
- ▶ Each primitive step requires **constant** amount of time
- ▶ Run-time of any one primitive does not depend on the input
- ▶ Run-time difference of the different steps/operations are ignored: Adding, copying, if-statements, multiplications, starting a sub-routine
- ▶ Sorting is not a primitive! Sensible definition of step/basic operations/primitives

Runtime and efficiency

Run-time depends on the complexity of the input

- ▶ Usually the number of elements: alternative, the length/ size.
One way or another it all boils down to number of bits.
- ▶ Runtime can often be described by a function f of the number of input elements n , i.e. $f(n)$.

Example:

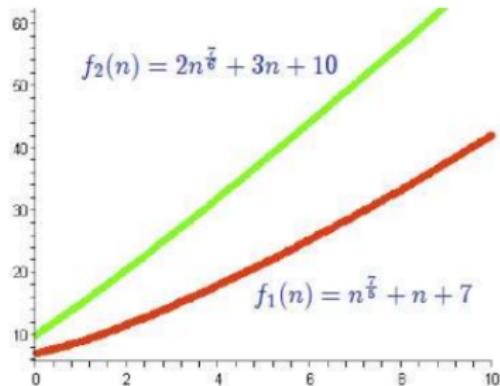
1. Runtime of algorithm 1: $f_1(n) = n^{\frac{7}{5}} + n + 7$
2. Runtime of algorithm 2: $f_2(n) = 2n^{\frac{7}{6}} + 3n + 10$

How can we compare the efficiency of the two algorithms?

Comparison of the runtimes

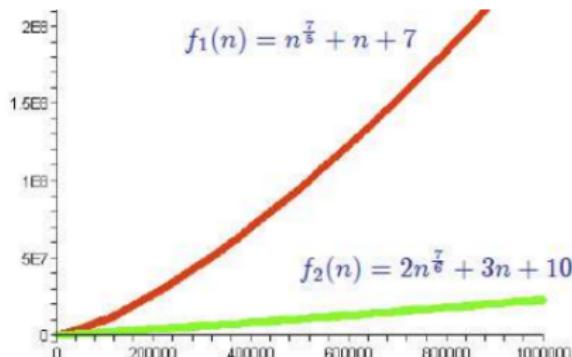
this is what we're interested in

$$0 \leq n \leq 10$$



Number of input elements n

$$0 \leq n \leq 1000,000$$



Number of input elements n

We are always interested in large- n behavior, meaning the asymptotic runtime.

Things to keep in mind

- ▶ Growth rate is usually defined by the highest order term
- ▶ Lower order expressions can be ignored
- ▶ Constant factors can be ignored
- ▶ Growth rate (for large n): Algorithm 2 is more efficient than Algorithm 1
- ▶ Asymptotic behavior is important
- ▶ In the previous example:

Runtime of algorithm 1: $f_1(n) = n^{\frac{7}{5}} + n + 7 \in O(n^{\frac{7}{5}})$

Runtime of algorithm 2: $f_2(n) = 2n^{\frac{7}{6}} + 3n + 10 \in O(n^{\frac{7}{6}})$

alg 1 is to a higher power!

prefer alg 2

Exercise on O-notation

- ▶ $2n^2 + 7n - 20 \in O(n^2)$
- ▶ $2n^2 + 7n \log n - 20 \in O(?)$ n^2
- ▶ $7n \log n - 20 \in O(?)$ $n \log n$
- ▶ $5 \in O(?)$ 1
- ▶ $2n^2 + 7n \log n + n^3 \in O(?)$ n^3

Sometimes the notation $f = O(g)$ is used. This is incorrect since $o()$, $\Theta()$, $O()$ are sets. Thus only $f \in O(g)$ is correct.

Useful rules to handle O-notation

- ▶ $f \in O(f)$
- ▶ $O(O(f)) = O(f)$
- ▶ $k \cdot O(f) = O(f)$
- ▶ $O(f + k) = O(f)$, where k is a constant
- ▶ $O(f) + O(g) = O(\max\{f, g\})$
- ▶ $O(f) \cdot O(g) = O(f \cdot g)$

Runtimes of basic components of a program

We will look at the runtime of...

- ▶ Basic operations/statements
- ▶ Sequential execution of basic operations/statements
- ▶ Conditional statements
- ▶ Loops

Basic operations

Basic operations:

```
i1 := 0;
```

$O(1)$

Basic operations

Sequential execution of basic operations/statements:

i1 := 0;	O(1)	
i2 := 0;	O(1)	
...	...	
i327 := 0;	O(1)	$327 \cdot O(1) = O(1)$

Basic operations: loops

for i := 1 to n do	O(n)	O(1) · O(n) = O(n)
begin		
a[i] := 0;	O(1)	
end;		

for i := 1 to n do	O(n)	O(n)	O(1) · O(n) = O(n)
begin			
a1[i] := 0;	O(1)		
...	...	37 · O(1)	
a37[i] := 0;	O(1)	= O(1)	
end;			

Basic operations: if statements

if x<100 then	O(1)	O(1)	O(max{1,n}) = O(n)	
y:=x;	O(1)			
else		O(n) · O(1) = O(n)		
for i:=1 to n	O(n)			
if a[i]>y then	O(1)			
y:=a[i];	O(1)			

Basic operations: Example

- ▶ Input: Array x with n numbers
- ▶ Output: Array a with n numbers. $a[i]$ is the average of numbers $x[0]$ to $x[i]$

```
for i:=0 to n-1 do      O(n)
    begin
        s:=0;
        for j:=0 to i do      Worst case O(n)
            begin
                s:=s+x[j];
            end;
        a[i]:=s/(i+1);      O(1)
    end;
return a;
```

→ $O(n^2)$

Basic operations: Example (cont)

```
for i:=0 to n-1 do
begin
    s:=0;
    for j:=0 to i do
        s:=s+x[j];
    a[i]:=s/(i+1);
end;
```

O(n)	O(n)	$O(n) \cdot O(i) =$ $O(n^2)$	
O(1)	O(i)		
O(i+1)			
O(3)	O(1)		
O(4)			

How often is the statement in the inner loop executed?

$$1 + 2 + \dots + n = \frac{n(n+1)}{2} = O(n^2).$$

Basic operations: Example (cont)

Is there a more efficient way?

$$a[i] = \frac{1}{i+1} (x[0] + x[1] + \dots + x[i])$$

$$a[i] \cdot (i + 1) = x[0] + x[1] + \dots + x[i]$$

$$a[i + 1] = \frac{1}{i+2} (x[0] + x[1] + \dots + x[i] + x[i + 1])$$

$$a[i + 1] = \frac{1}{i+2} (a[i] \cdot (i + 1) + x[i + 1])$$

What is the advantage of this approach?

Homework assignment: Please compute $O(\cdot)$ of the approach.
Please implement both approaches in Python and provide some actual runtime comparisons.

Homework (part 2): Please show

$$f = O(f)$$

$$O(O(f)) = O(f)$$

$$kO(f) = O(f)$$

$$O(f + k) = O(f)$$

$$O(f) + O(g) = O(\max\{f, g\})$$

$$O(f) \cdot O(g) = O(f \cdot g)$$

Data structures

Contents:

1. Types and benefits of popular data structures
2. Particulary: array/ vector, list, (doubly) linked list, stack, queue
3. Implementation examples
4. Trees: binary, n-ary, terminology, traversal
5. Binary search trees and operations on such trees
6. Heaps and heapsort

Algorithms work on data structures

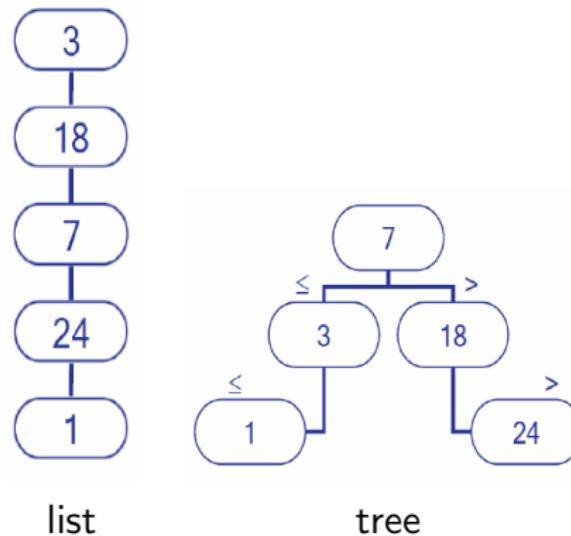
- ▶ Algorithms work on dynamic sets of elements (input and output): for example, to perform searching, insertion or deletion of elements, finding maximum or minimum elements.
- ▶ Data structures are used for the implementation of dynamic datasets.
- ▶ The efficiency of a data structure can depend on the methods available for manipulation (e.g. search, insert).
- ▶ The design of data structure should be driven by enabling an efficient implementation of the operations that are relevant in the algorithm. What is more, the efficiency of algorithms often depends on the availability of tailored data structures for the operations needed in an algorithm.

Data structures

- ▶ Definition: *Particular way of organizing and storing data in a computer so that it can be accessed and modified efficiently.* (wikipedia)
- ▶ Examples: collection of data values, relationship between them, algorithms applied to data
- ▶ Close relationship between efficiency (runtime and memory requirement) of an algorithm and the data structures used
- ▶ Small set of common data structures behind most algorithms (and more complex data structures)

Motivation

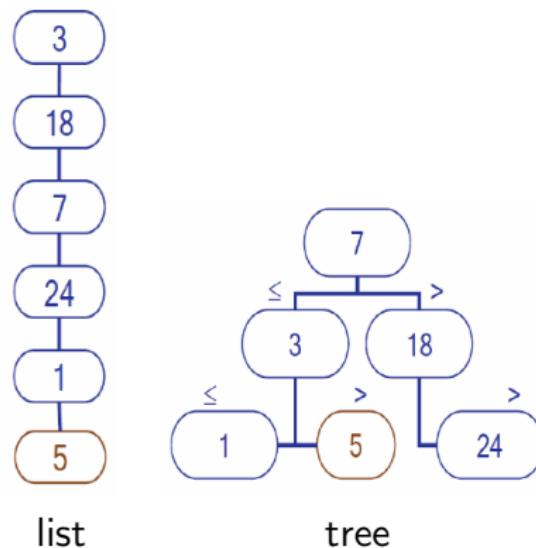
We want to insert the element 5 to a set:



How to do this best? How many operations are needed?

Motivation

It's more efficient in a tree:

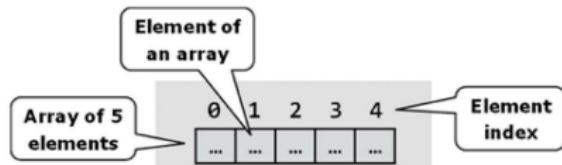


Overview of data structures

- ▶ Array: access to elements based on an index
- ▶ Linked List: reference/pointer to the next element
- ▶ Stack: dynamic set of elements, can only be read starting from the most recently added element, called *Last In First Out (LIFO)*
- ▶ Queue: dynamic set of elements, can only be read starting from the most longest added element, called *First In First Out (FIFO)*
- ▶ Graphs or Trees: elements have references/pointers to a variable number of other elements

Array or vector

Array or vector: fixed length list in which elements are accessed by index

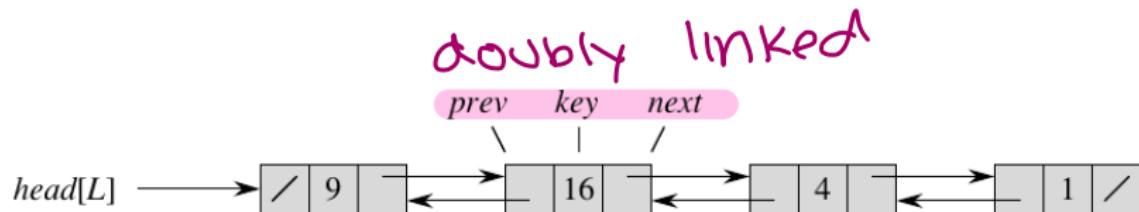


How would you...

1. Access one element? $O(1)$
2. Append one? $O(1)$ can only do it if there's space
3. Prepend one? $O(n) \rightarrow$ move $n-1$, then insert

What are the runtimes of the above operations?

Linked list



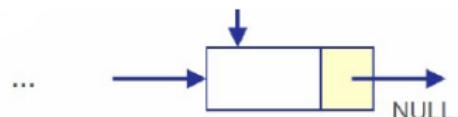
- ▶ A linked list is a **dynamic data structure**, i.e. the **number of elements** in it can be modified during runtime.
- ▶ Elements can be anything (basic or composite/complex data structures, for instance either numbers or lists again)
- ▶ Elements are **linked with pointers**.
- ▶ Single, doubly, multiple linked lists.

Array versus linked list

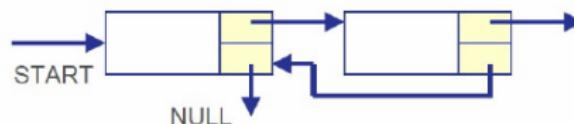
- ▶ Linked lists store one element per list item, with small additional memory requirement for the pointers.
- ▶ Insert and delete operations can be done without need for copying other elements. $O(1)$
- ▶ Number of elements is variable
- ▶ In turn, no direct element access, that is access is not $O(1)$

Variations of linked lists

singly linked list



Doubly-linked list

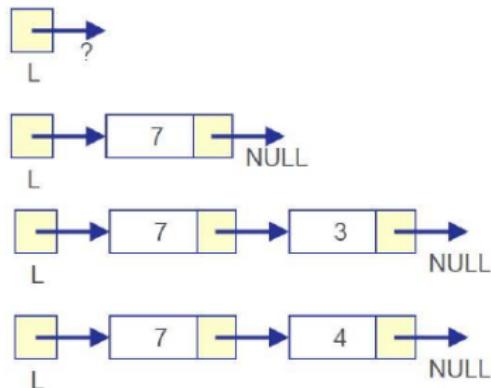


Implementation example for the Java language

```
public class Listnode {  
    private int data;  
    private Listnode next;  
  
    // two constructors  
    public Listnode(int d) {  
        data = d; next = null;  
    }  
    public Listnode(int d, Listnode n) {  
        data = d; next = n;  
    }  
  
    // get and set data  
    public int getData() { return data; }  
    public void setData(int d) { data = d; }  
  
    // get and set the neighbor  
    public Listnode getNext() { return next; }  
    public void setNext(Listnode n) { next = n; }  
}
```

How to use the Listnode class

- ▶ Listnode L;
- ▶ L = new Listnode(7);
- ▶ L.setNext(new Listnode(3));
- ▶ L.getNext().setData(4);

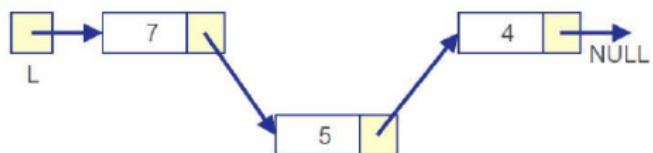


Inserting into a linked list

- ▶ original list



- ▶ after insertion:



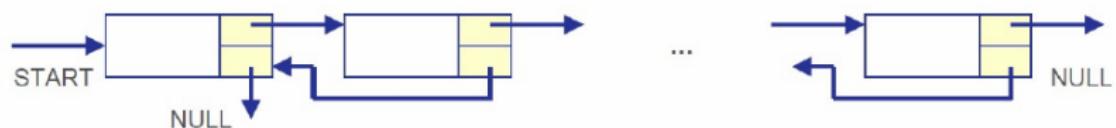
in Java: L.setNext(new Listnode(5,L.getNext()));

- ▶ idea: set "next" pointer of first element to a newly created element, whose next pointer is again set to the previous second element, thus keeping the list together

Single versus doubly linked lists



Doubly-linked list

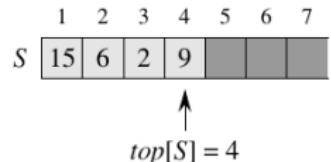


$O(n)$?

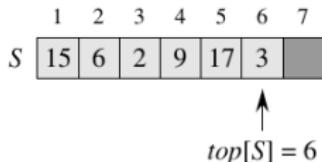
- ▶ What is the runtime of $append(list, element)$ and $prepend(list, element)$ for (a) single and (b) doubly linked lists?
- ▶ When should each list type be used?
- ▶ What are its pros and cons?

Stacks

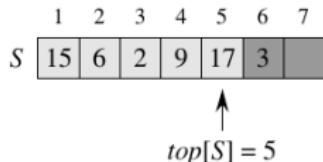
Example of a stack:



(a)



(b)



(c)

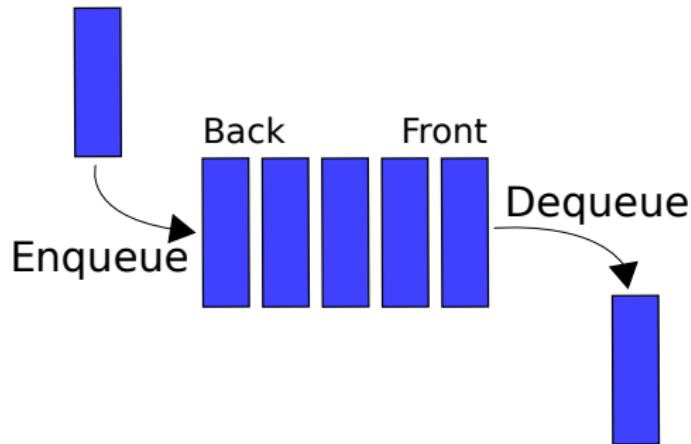
- ▶ Stack: Dynamic set of elements, can only read the most recently added element, **Last In First Out (LIFO)**
- ▶ Realized here as an array with pointer to the top element.
- ▶ Possible operations: check if empty, push element in $O(1)$, read the top element in $O(1)$ (called "pop").
- ▶ How about searching, deletion of an arbitrary element, or inserting an element?

search $O(n)$
del $O(n)$

inserting (in order) $O(n)$
pop up to n , re-push up to $n \rightarrow 2n \rightarrow O(n)$

Queue

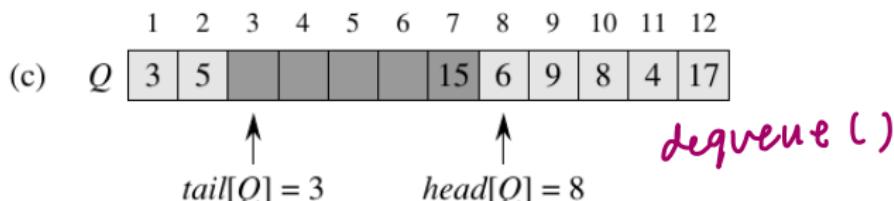
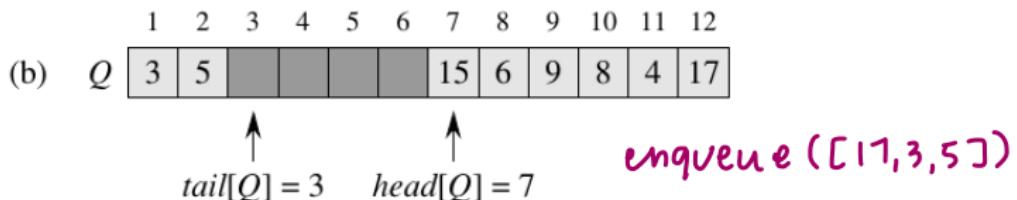
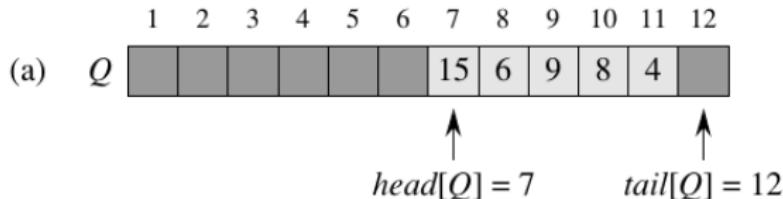
Queue: Data structure which allows to add items ("tasks") to a pile. Items are added at the end (the "Enqueue" operation) and stored items are requested at the front (the "Dequeue" operation).



- ▶ Available operations: Add and request from the queue.
- ▶ Time complexity?
- ▶ How about searching, deletion, arbitrary insertion?

Queue

Example of a queue: The queue can be realized as an array whose ends are identified (to make it effectively circular)



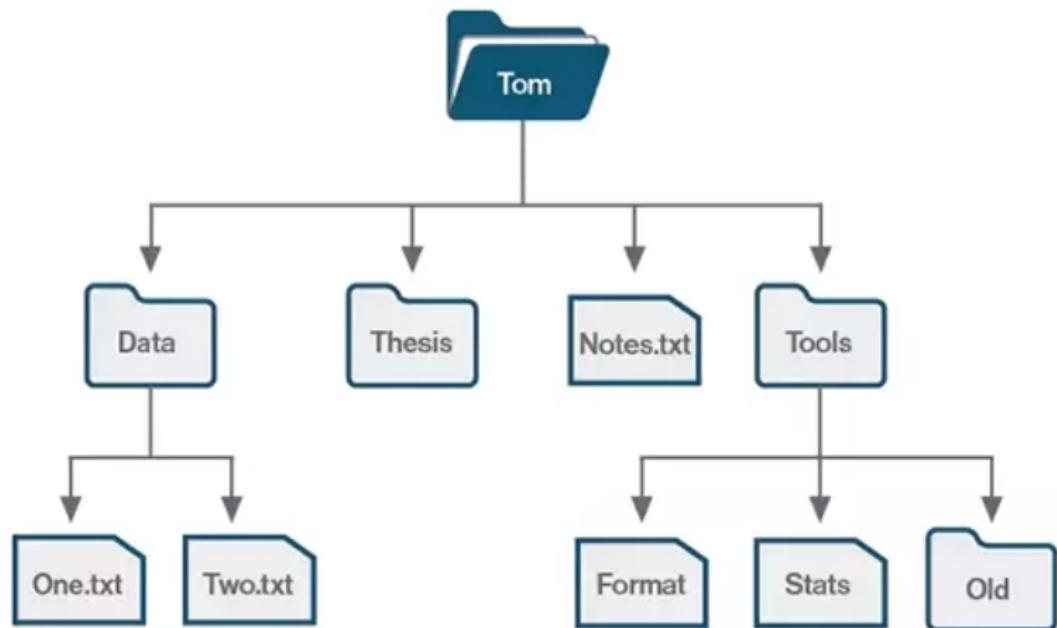
Queue

- ▶ Queue: Dynamic set of elements, can only be read starting from the element that was added the longest time ago, **First In First Out (FIFO)**
- ▶ We saw how to realize a queue with an array. How would you implement a queue using a linked list?

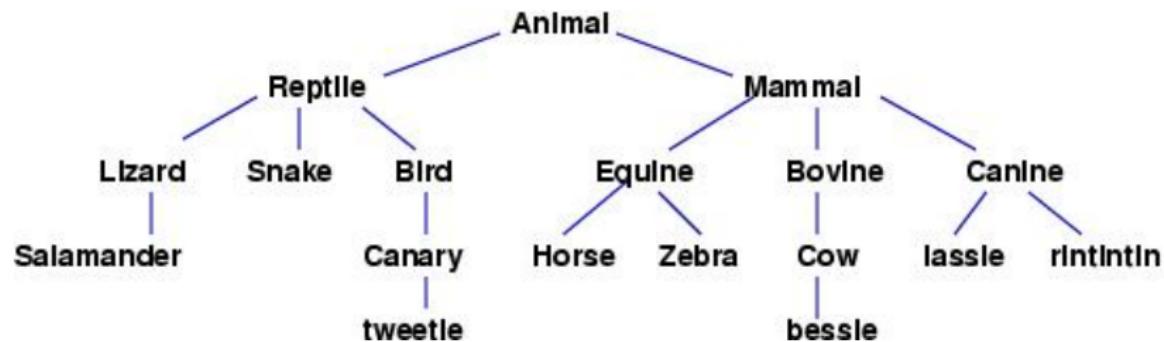
Difference to previous data structures:

- ▶ Arrays and lists organize data in linear order.
- ▶ Trees are hierarchical, not linear.
- ▶ Trees can capture much more complex relationships between data elements.

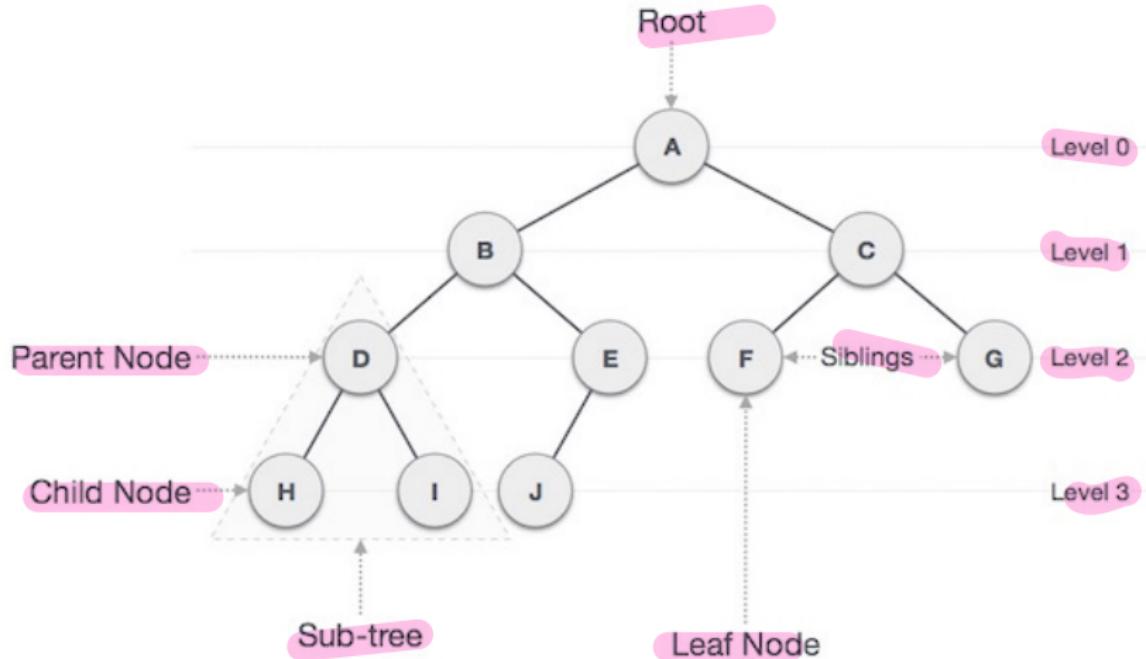
Example 1: File structure on a computer



Example 2: Hierarchical structures



Tree terminology



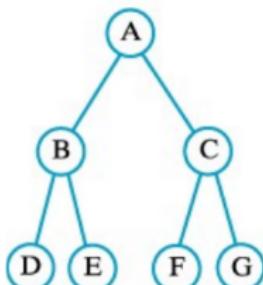
- ▶ A tree is a set of nodes connected by edges.
- ▶ Nodes can have: parents, children, descendants, siblings.
- ▶ Nodes are arranged in levels: top node is the "root", nodes without children are "leaves".

Tree terminology

- ▶ Any node in a tree is reached from the root by a path. The length of the path is the number of edges on it.
- ▶ The height of the tree is the number of levels of the tree. A tree can have a maximum, minimum, average height.
- ▶ A sub-tree of a node is the tree rooted at one of its children. (On the previous slide the sub-tree is marked of node B.)

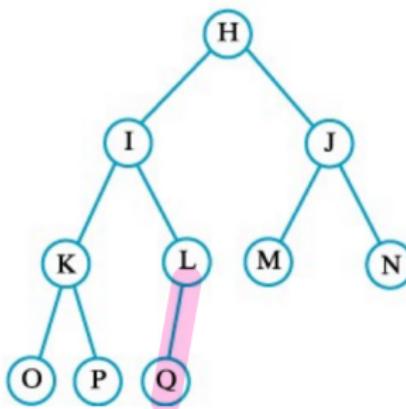
Binary and n-ary trees

(a) Full tree

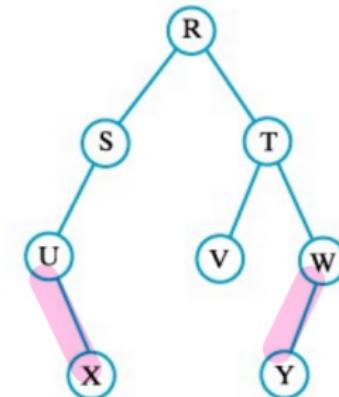


Left children: B, D, F
Right children: C, E, G

(b) Complete tree



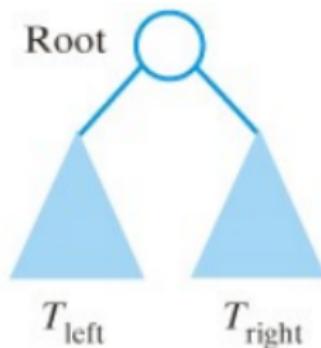
(c) Tree that is not full and not complete



- ▶ A tree can be binary (max. 2 children per node) or n-ary.
- ▶ In a full binary tree every node apart from the leaves has two children.
- ▶ A complete binary tree is a tree in which each level, except possibly the last, is completely filled and all nodes are as far left as possible.

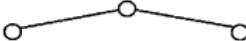
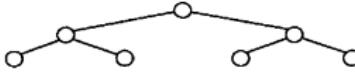
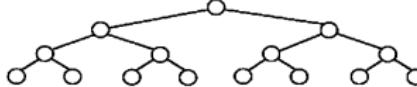
Binary tree

A binary tree has the following form (or is the empty tree):



where T_{left} and T_{right} are again binary trees.

Number of nodes in a full tree

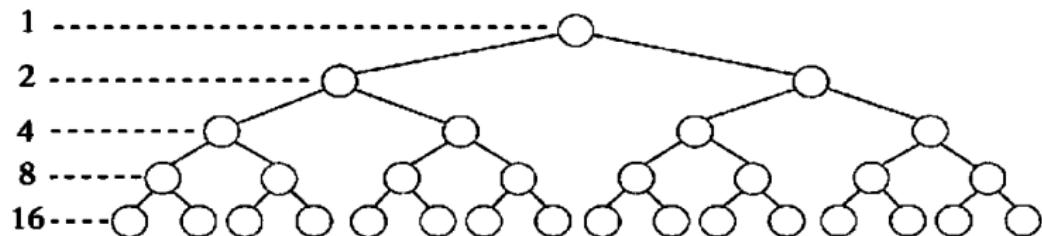
	height	number of nodes
	1	$2^1 - 1 = 1$
	2	$2^2 - 1 = 3$
	3	$2^3 - 1 = 7$
	4	$2^4 - 1 = 15$

Formula: A full tree of height h has $n = \sum_{i=0}^{h-1} 2^i = 2^h - 1$ nodes.

Number of nodes per level

The number of nodes on level $d \in \{0, 1, \dots\}$ is 2^d :

**Number of
nodes per level**



Remember: The root is at level zero...

Conversely, the height of a binary and complete/full tree with n nodes is $h = \log_2(n + 1)$.

Traversal of a tree

- ▶ Tree traversal is a common operation (in linear data structures such as lists this is trivial)
- ▶ During traversal, we must visit each element exactly once. The order of visiting is not unique.
- ▶ When visiting a node: process its data (e.g., print data, or check if new maximum is found). A traversal can pass through a node without visiting it.

Trees love recursion

Idea:

- ▶ visit the current node (start with root)
- ▶ visit all nodes in the left subtree (recursive call)
- ▶ visit all nodes in the right subtree (recursive call)

current
left
right

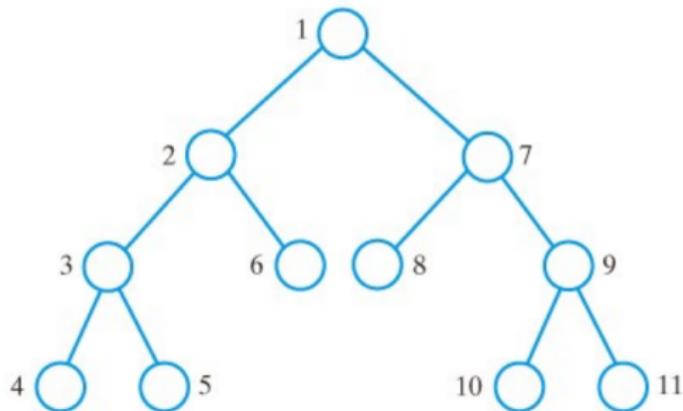
Many (implementation) examples to follow on the next slides...

Different orders of traversal possible: before, between, after visiting two subtrees. Generalizes directly to n-ary trees.

Preorder traversal

sLR

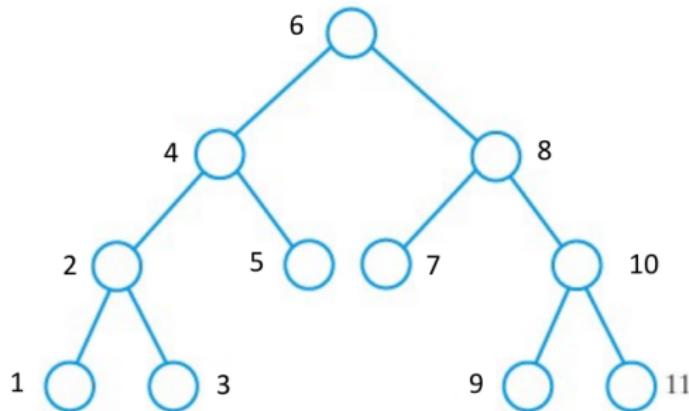
Preorder traversal: visit current node before visiting its subtrees



In-order traversal

LSR

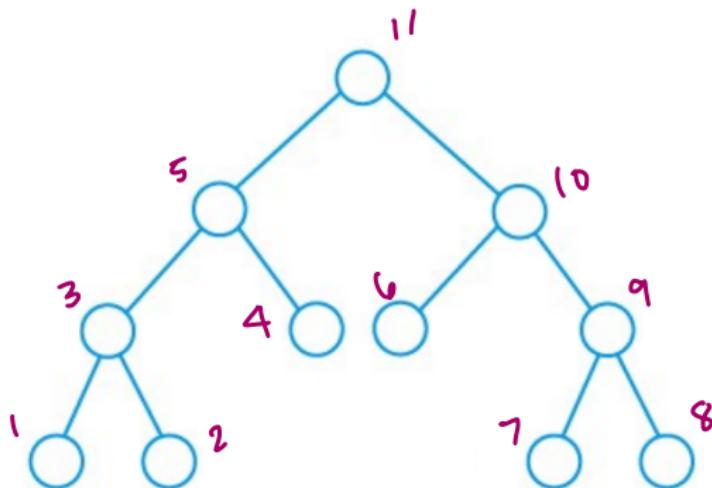
In-order traversal: visit left subtree of a node, visit node, visit right subtree of that node



Postorder traversal

LRS

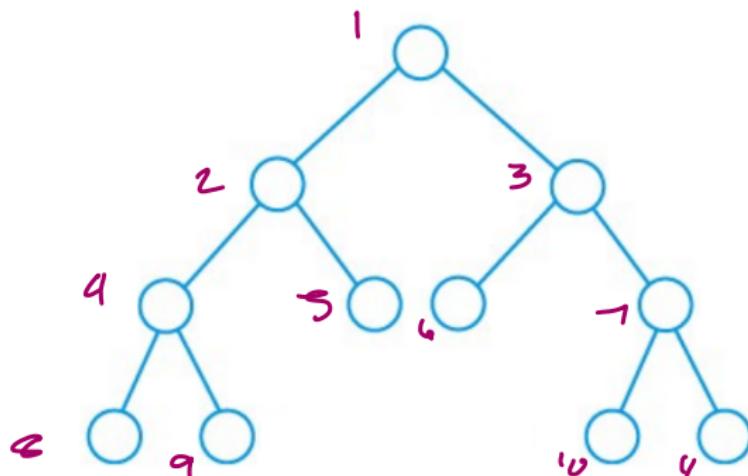
- Postorder traversal: visit current node after having visited its subtrees



What is the ordering in the above example tree?

Level-order traversal

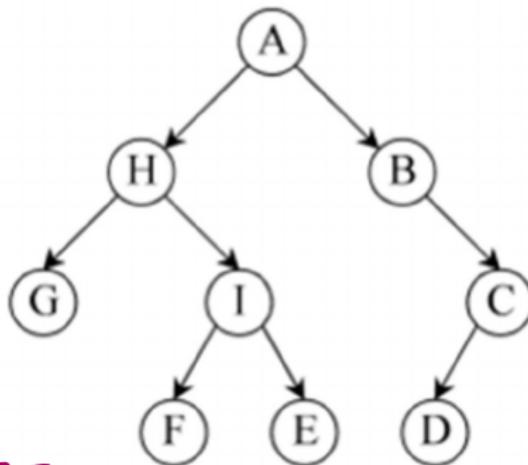
Level-order traversal: beginning with the root, visit nodes one level at a time



What is the ordering in the above example tree?

Example 1: orderings

In the example graph below, what are the orders of visiting all nodes for pre-, in-, post-, and level-order?



Pre-order: A H G I F E B C D

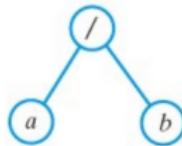
In-order: G H F I E A B C D

Post-order: G F E I H D C B A

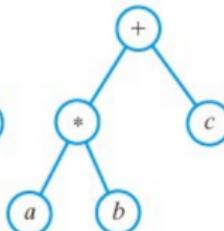
level-order: A H B G I L F E D

Example 2: algebraic expression trees

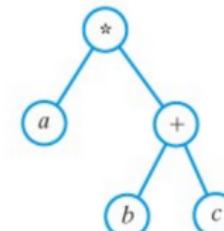
(a) a / b



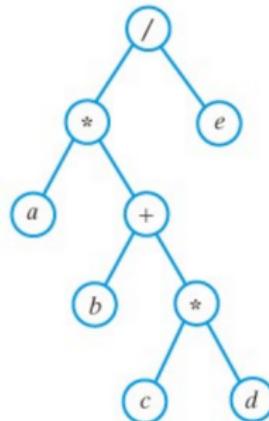
(b) $a * b + c$



(c) $a * (b + c)$



(d) $a * (b + c * d) / e$



using LSR
(in-order)

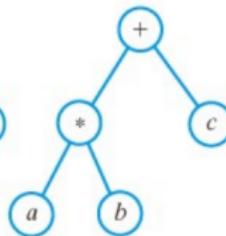
- ▶ One use of trees: encode controlled languages
- ▶ Above: root and nodes contain **binary operations**, leaves contain **operands**
- ▶ The order of the leaves matches the order of operands. Such a binary tree is called an **expression tree**. Encoding parentheses not necessary since encoded in subtree structure.

Example 2: algebraic expression trees

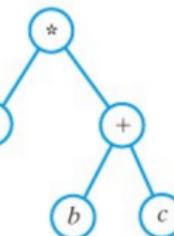
(a) a / b



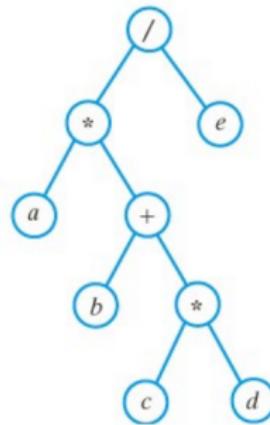
(b) $a * b + c$



(c) $a * (b + c)$



(d) $a * (b + c * d) / e$



Reading/ visiting the nodes results in different expressions:

- ▶ in-order traversal produces original expression
- ▶ preorder traversal produces prefix expression $+ * abc$
- ▶ postorder traversal produces postfix expression $ab * c +$

Pseudo-code for pre- and postorder traversal

```
function preorder(node):
    visit(node)
    if node has left child:
        preorder(node.left)
    if node has right child:
        preorder(node.right)

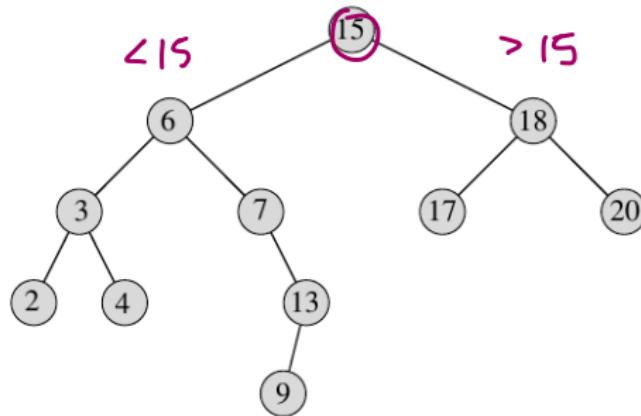
function postorder(node):
    if node has left child:
        postorder(node.left)
    if node has right child:
        postorder(node.right)
    visit(node)
```

Binary search trees

- ▶ If node data is not sorted, searching a tree is not more efficient than searching a list
- ▶ A search tree keeps the data organized at all times, including after operations such as insert and delete, to make operations (such as "search") more efficient

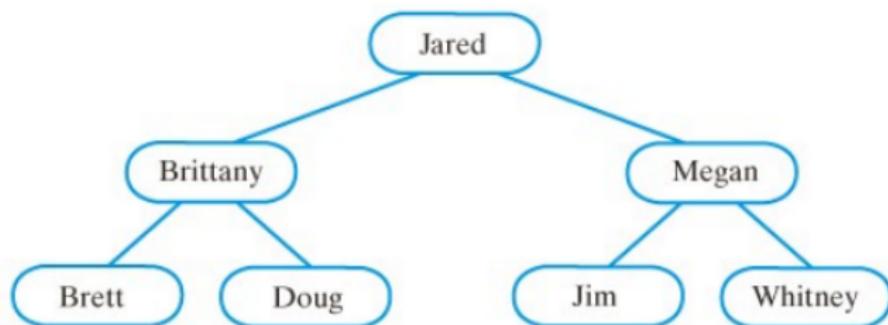
Binary search tree

Definition: (Binary search tree) A *binary search tree contains comparable objects, meaning that the operations $<$, $>$, $=$ are well-defined for all elements. For any node n , all elements in its left subtree are smaller than n , and all the elements in its right subtree are bigger than n .*



Common operations: search element, find minimum/ maximum, find successor/ predecessor, insert or delete elements

Elements in binary search trees do not have to be numbers

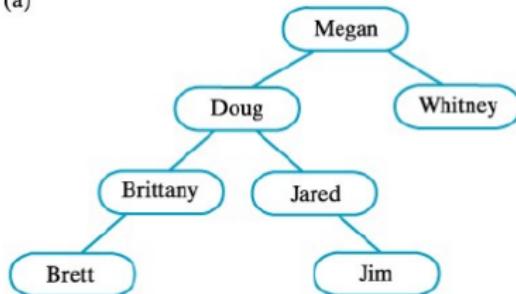


In this example, the comparison function between elements is the lexicographical ordering of words.

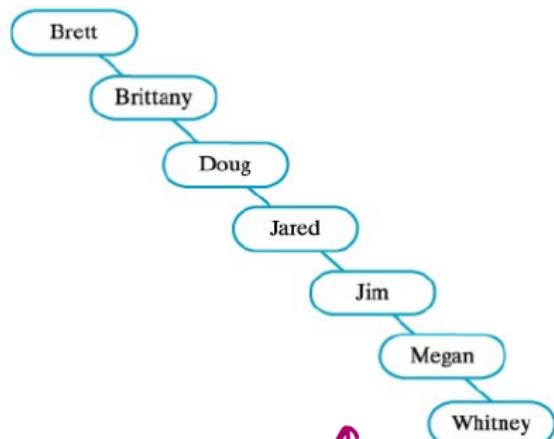
Balanced versus imbalanced trees

Two trees containing the same information:

(a)



(b)

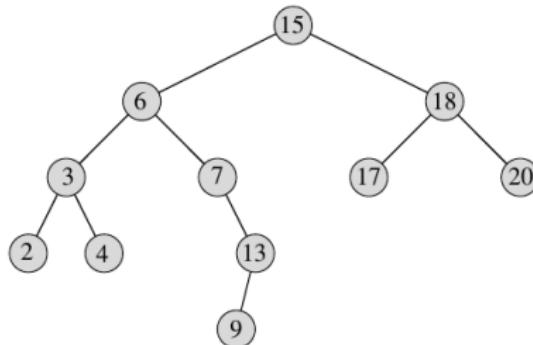


a bit
better than list

no better
than list

How is searching affected on either tree?

Search for an element



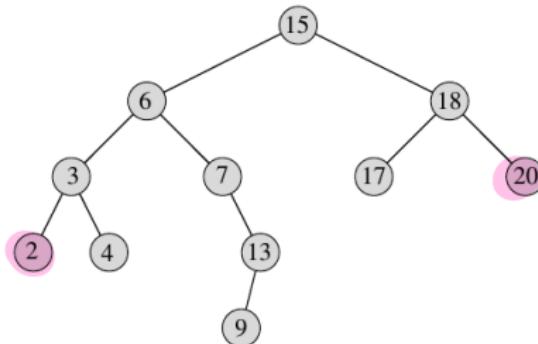
Recursion is helpful:

TREE-SEARCH(x, k)

- 1 **if** $x = \text{NIL}$ or $k = \text{key}[x]$
- 2 **then return** x
- 3 **if** $k < \text{key}[x]$
- 4 **then return** TREE-SEARCH($\text{left}[x], k$)
- 5 **else return** TREE-SEARCH($\text{right}[x], k$)

Searching a binary tree of height h is $O(h)$.

Find minimum/ maximum



Since the elements in any left subtree are smaller than the subtree root, we only need to follow the left subtrees (analogously for the maximum):

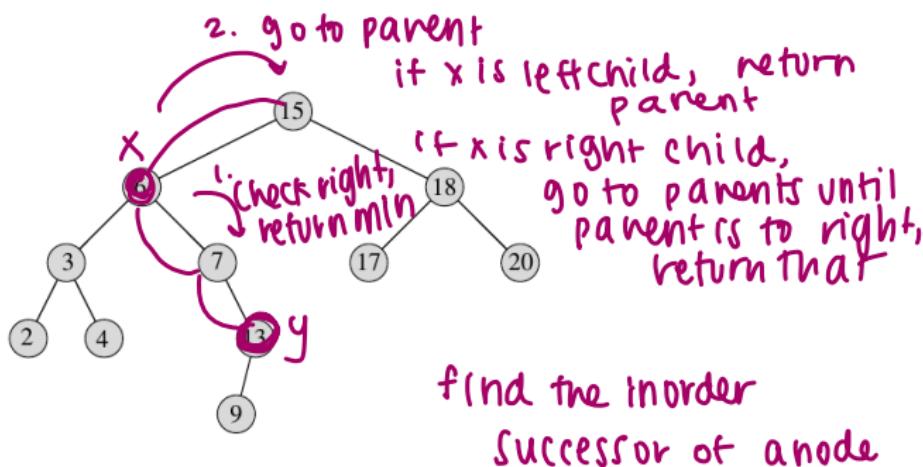
TREE-MINIMUM(x)

- 1 **while** $left[x] \neq \text{NIL}$
- 2 **do** $x \leftarrow left[x]$
- 3 **return** x

TREE-MAXIMUM(x)

- 1 **while** $right[x] \neq \text{NIL}$
- 2 **do** $x \leftarrow right[x]$
- 3 **return** x

Successor



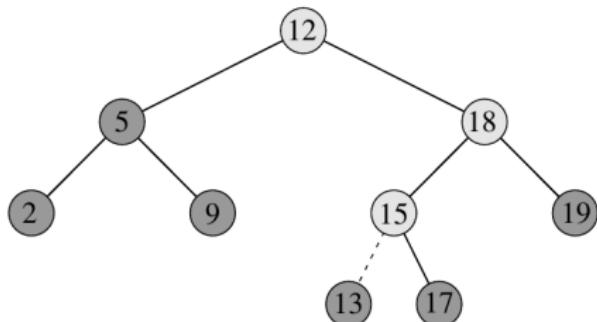
Recursion is helpful:

TREE-SUCCESSOR(x)

- 1 **if** $right[x] \neq \text{NIL}$
- 2 **then return** TREE-MINIMUM($right[x]$)
- 3 $y \leftarrow p[x]$ **parent of** x
- 4 **while** $y \neq \text{NIL}$ and $x = right[y]$
- 5 **do** $x \leftarrow y$
- 6 $y \leftarrow p[y]$
- 7 **return** y

What is the runtime?

Insert an element



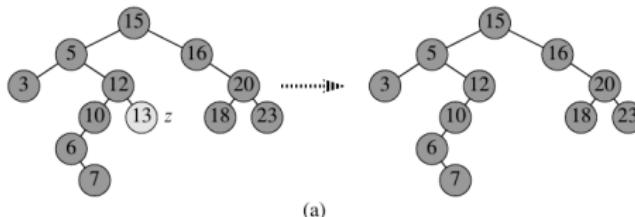
TREE-INSERT(T, z)

```
1   $y \leftarrow \text{NIL}$ 
2   $x \leftarrow \text{root}[T]$ 
3  while  $x \neq \text{NIL}$ 
4      do  $y \leftarrow x$ 
5          if  $\text{key}[z] < \text{key}[x]$ 
6              then  $x \leftarrow \text{left}[x]$ 
7              else  $x \leftarrow \text{right}[x]$ 
8       $p[z] \leftarrow y$ 
9      if  $y = \text{NIL}$ 
     $\triangleright$  Tree  $T$  was empty
10     then  $\text{root}[T] \leftarrow z$ 
11     else if  $\text{key}[z] < \text{key}[y]$ 
12         then  $\text{left}[y] \leftarrow z$ 
13         else  $\text{right}[y] \leftarrow z$ 
```

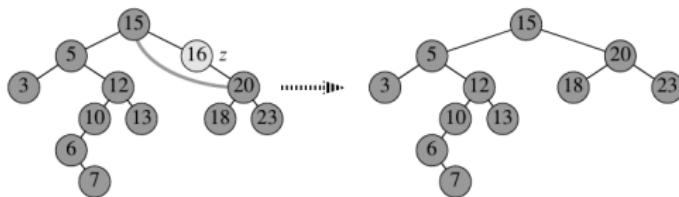
To insert an element (here, element 13), we first find the correct position by tree traversal (light shaded nodes), and then append the element as a new leaf (dashed line).

Delete an element

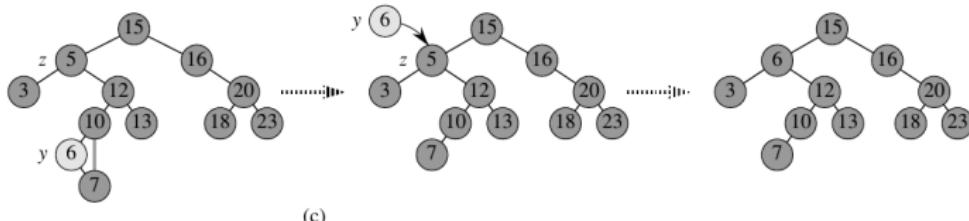
Deletion is more complicated depending on the position of the element to be deleted:



(a)

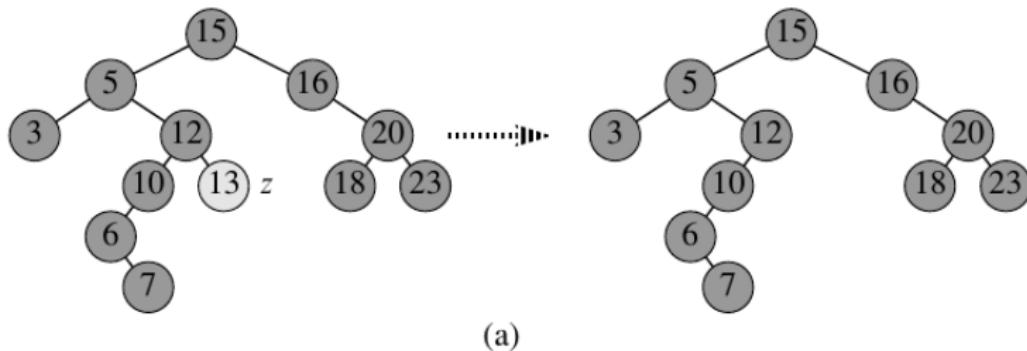


(b)



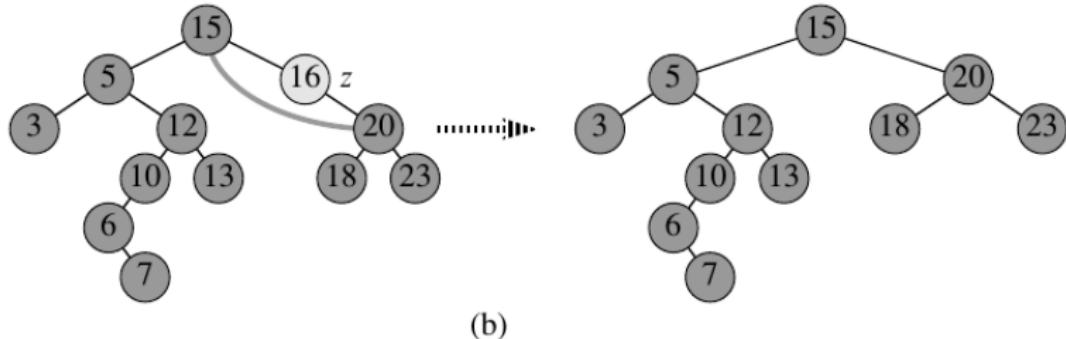
(c)

Delete an element: Three cases



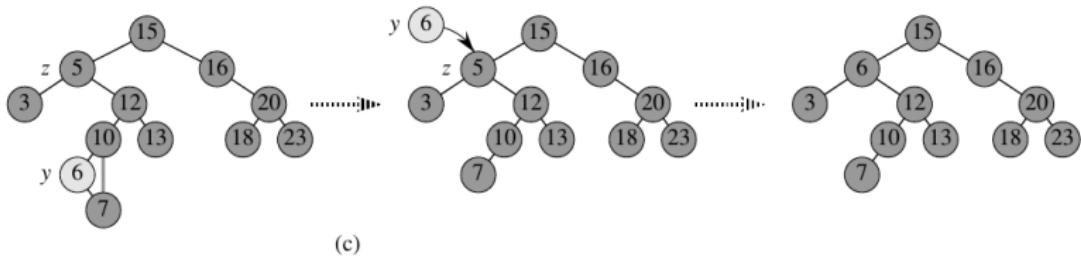
Case (a): Node z has to be deleted. z has no children, so we just remove it.

Delete an element: Three cases



Case (b): If node *z* has one child, we remove *z* and put the child in its place.

Delete an element: Three cases



Case (c): If node z has two children, we find a replacement y for z which is bigger than z 's left child, but smaller than z 's right child, and put it in z 's place. (Why is it always possible to find such a y ?)

Pseudo-code for deleting an element

```
TREE-DELETE( $T, z$ )
1  if  $left[z] = \text{NIL}$  or  $right[z] = \text{NIL}$ 
2    then  $y \leftarrow z$ 
3    else  $y \leftarrow \text{TREE-SUCCESSOR}(z)$ 
4    if  $left[y] \neq \text{NIL}$ 
5      then  $x \leftarrow left[y]$ 
6      else  $x \leftarrow right[y]$ 
7    if  $x \neq \text{NIL}$ 
8      then  $p[x] \leftarrow p[y]$ 
9    if  $p[y] = \text{NIL}$ 
10     then  $root[T] \leftarrow x$ 
11     else if  $y = left[p[y]]$ 
12       then  $left[p[y]] \leftarrow x$ 
13       else  $right[p[y]] \leftarrow x$ 
14   if  $y \neq z$ 
15     then  $key[z] \leftarrow key[y]$ 
16           copy  $y$ 's satellite data into  $z$ 
17   return  $y$ 
```

More on balanced trees

Many operations on a tree depend on the height of the tree. We want to keep trees balanced to (a) reduce their height and thus (b) keep operations fast – even as the tree keeps changing or growing.

Methodology has been developed for that in the literature:

- ▶ red-black trees
- ▶ AVL (Adelson-Velsky and Landis) self-balancing binary search trees

...for further reading...

Heaps

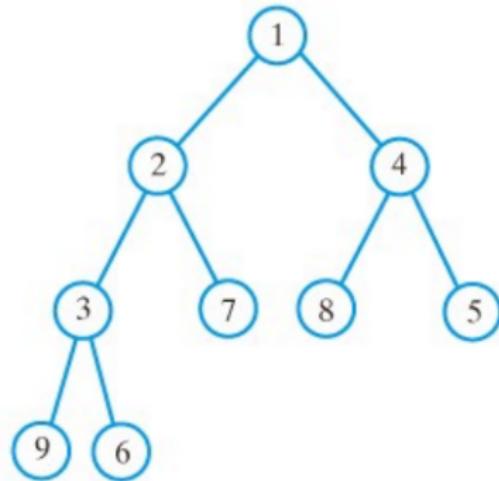
What are heaps?

- ▶ A heap is a complete binary tree. Nodes contain *comparable* objects.
- ▶ **Heap property:** Each node is not smaller (for max-heap) or bigger (for min-heap) than all its descendants.
- ▶ In max-heap, the root is \geq all its descendants, thus making it the largest value. In min-heap, the root is \leq all its descendants, thus making it the smallest value.

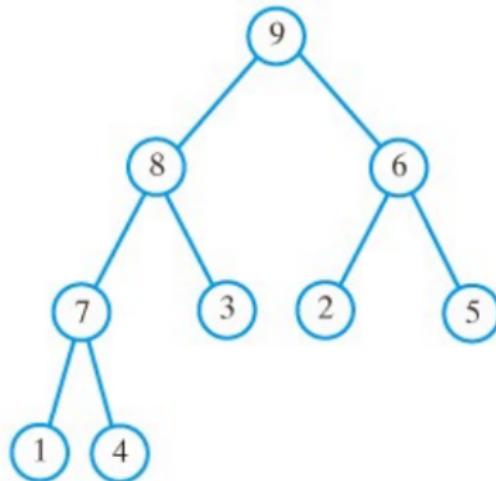
What is heapsort?

- ▶ Sorting algorithm built upon the heap data structure.
- ▶ Sorting will follow as easy consequence from the heap tree.
- ▶ Like insertion sort, heapsort is able to sort an array *in place*, which e.g. mergesort is not capable of. Yet heap sort achieves runtime $O(n \log n)$.

Example of min- and max-heap



min-heap



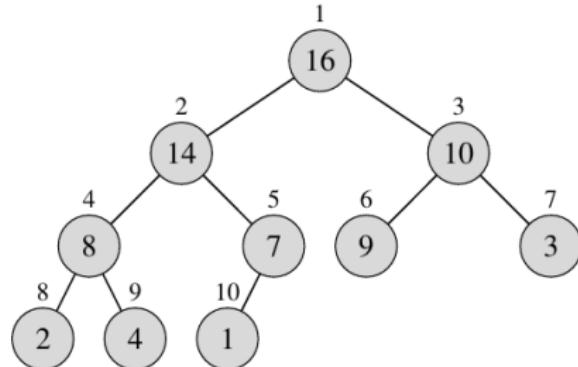
max-heap

Pseudo-code of the heap interface

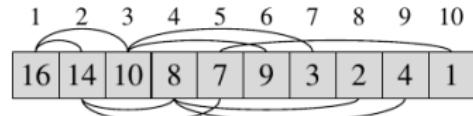
Pseudo-code of a *heap class*, i.e. what functions does a heap data structure need to provide:

```
public interface Heap {  
    public void add(Comparable newEntry);  
    public Comparable removeRoot();  
    public Comparable getRoot();  
    public boolean isEmpty();  
    public int getSize();  
    public void clear();  
}
```

Heap data structure



(a)



(b)

Heap can be stored in an array A :

- ▶ Element $A[1]$ contains root of heap.
- ▶ Parent of node i is $A[\lfloor i/2 \rfloor]$
- ▶ Left child of node i is $A[2i]$, and right child is $A[2i + 1]$

max-heap property: $A[\text{parent}(i)] \geq A[i]$ for all i

min-heap property: $A[\text{parent}(i)] \leq A[i]$ for all i

Heap data structure

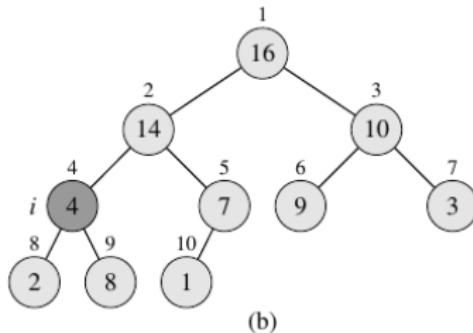
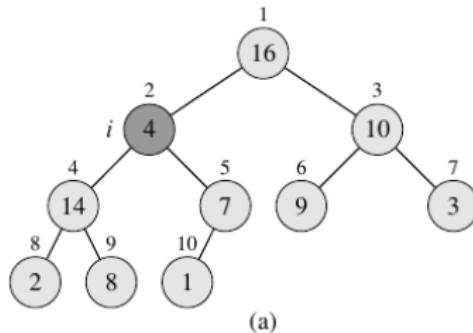
Advantages of the heap data structure:

- ▶ When binary tree is complete: can use level-order traversal to store data in consecutive locations
- ▶ Easy location of data in a node's parent or children: parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$, children are $A[2i]$ and $A[2i + 1]$.

In the next slides we always consider max-heap.

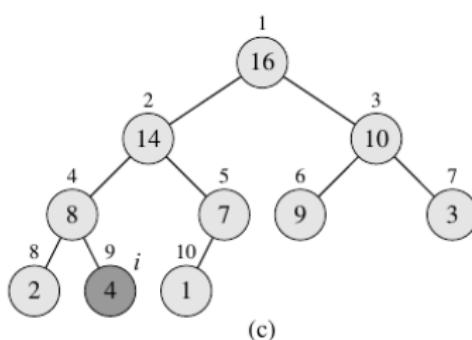
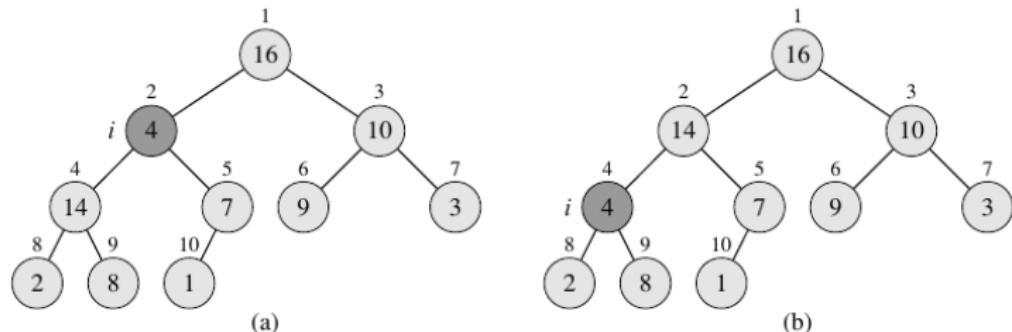
Building a heap

Fixing the heap property for one entry:



- ▶ Element 4 violates the heap property.
- ▶ We assume the two subtrees of element 4 are heaps.
- ▶ Idea: Swap entry with largest child if heap property is not satisfied. This fixes the current node issue but might cause one of the subtrees to not be a heap any more. Repeat recursively.

Building a heap: full picture



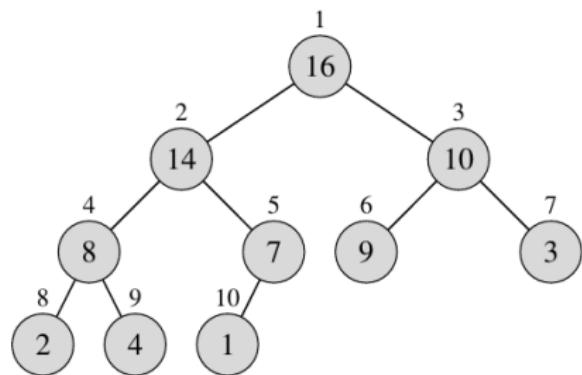
This "bubbles" down an element until its correct position in the heap is found. This procedure is called **max-heapify**.

Pseudo-code of max-heapify

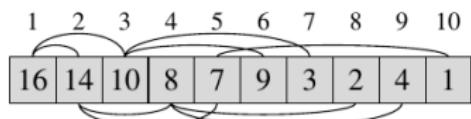
```
MAX-HEAPIFY( $A, i$ )
1    $l \leftarrow \text{LEFT}(i)$ 
2    $r \leftarrow \text{RIGHT}(i)$ 
3   if  $l \leq \text{heap-size}[A]$  and  $A[l] > A[i]$ 
4     then  $\text{largest} \leftarrow l$ 
5     else  $\text{largest} \leftarrow i$ 
6   if  $r \leq \text{heap-size}[A]$  and  $A[r] > A[\text{largest}]$ 
7     then  $\text{largest} \leftarrow r$ 
8   if  $\text{largest} \neq i$ 
9     then exchange  $A[i] \leftrightarrow A[\text{largest}]$ 
10    MAX-HEAPIFY( $A, \text{largest}$ )
```

Runtime: Roughly speaking, we bubble down the tree which has depth $O(\log n)$, thus the runtime of max-heapify is $O(\log n)$.

Runtime of max-heapify



(a)



(b)

More precisely, when initializing the heap we fill up levels with elements in unsorted order, thus the difference in depth of any two subtrees can only be 1. We now show that for any node, the imbalance of its two subtrees is at most $2/3$.

Runtime of max-heapify

Consider node n to be heapified. Assume its two subtrees have sizes h and $h - 1$. Then the heapify procedure will swap n with the root of either subtree and continue in a recursive fashion, thus cutting down the effort by the size of the unconsidered subtree.

Assume the subtree we recursive is the bigger one, size $2^h - 1$.
The total subtree containing n has size $1 + (2^h - 1) + (2^{h-1} - 1)$.

Thus by continuing in a recursive fashion on one subtree only, we reduce the problem to

$$\begin{aligned}\frac{2^h - 1}{1 + (2^h - 1) + (2^{h-1} - 1)} &= \frac{2^h - 1}{2^{h-1}(2 + 1) - 1} = \frac{2^h - 1}{3 \cdot 2^{h-1} - 1} \\ &\leq \frac{2 \cdot 2^{h-1}}{3 \cdot 2^{h-1} - 1} = \frac{2}{3 - 2^{-(h-1)}} \rightarrow \frac{2}{3}\end{aligned}$$

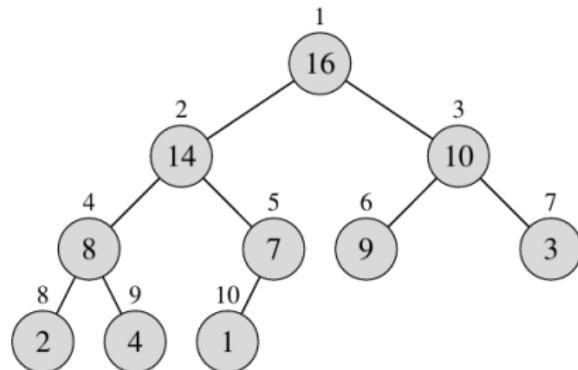
Runtime of max-heapify

The runtime of swapping a node with its child is $\Theta(1)$. Since in the worst case (asymptotically) the problem reduces to at least size $2/3$ (if it happens that we continue to heapify in the smaller subtree), we have

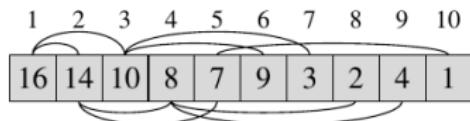
$$T(n) = T(2/3 \cdot n) + \Theta(1).$$

Solving this with the master theorem gives $O(\log n)$, i.e. the runtime is linear in the tree height.

Building a heap



(a)



(b)

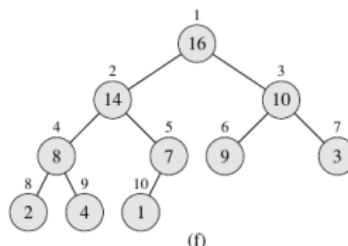
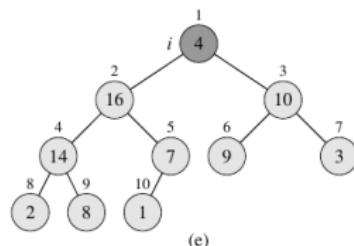
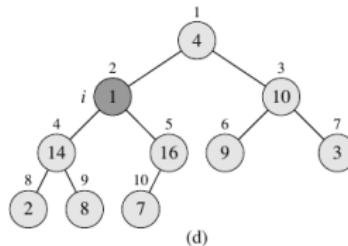
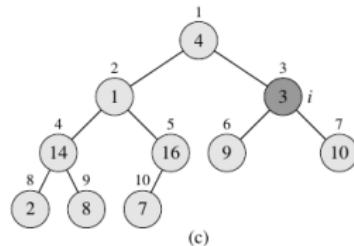
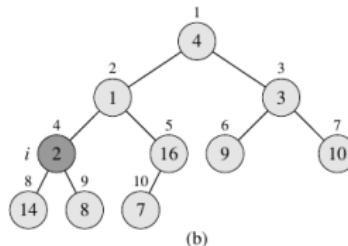
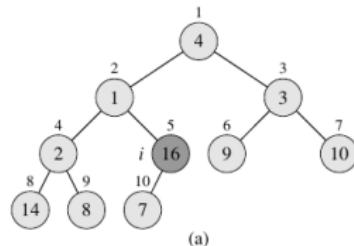
Building a heap from an array is easy now: All leaves satisfy the heap property (they do not have any subtrees). Then we go bottom up and heapify every node:

BUILD-MAX-HEAP(A)

```
1 heap-size[ $A$ ]  $\leftarrow$  length[ $A$ ]  
2 for  $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$  downto 1  
3     do MAX-HEAPIFY( $A, i$ )
```

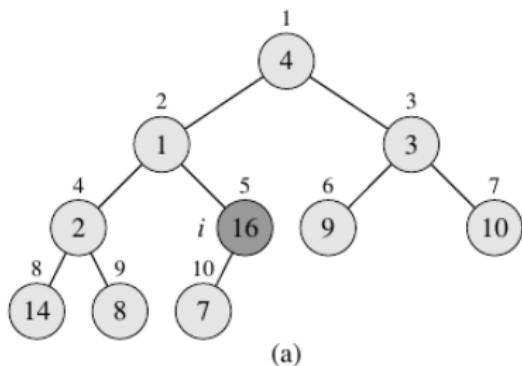
Building a heap: full picture (steps to follow)

A	4	1	3	2	16	9	10	14	8	7
---	---	---	---	---	----	---	----	----	---	---

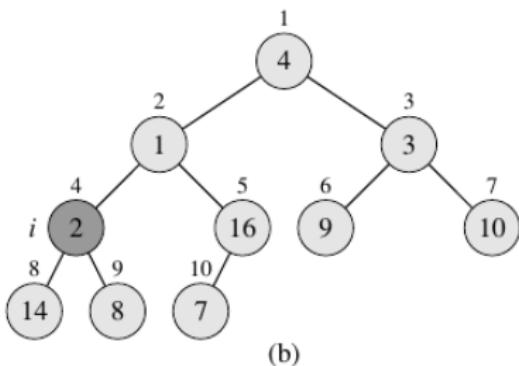


Building a heap

Input array: [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

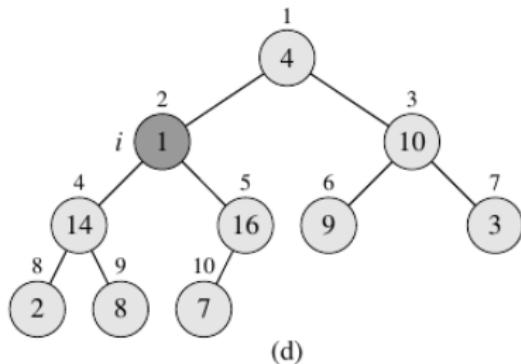
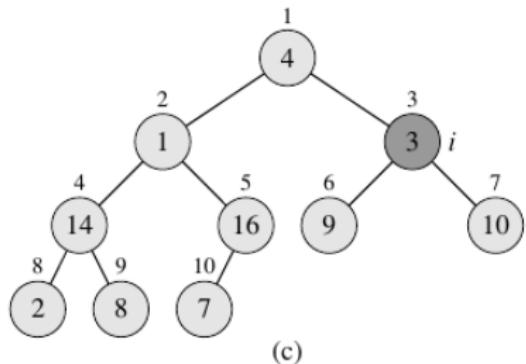


(a)

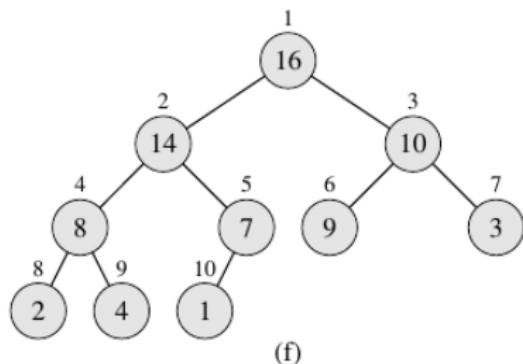
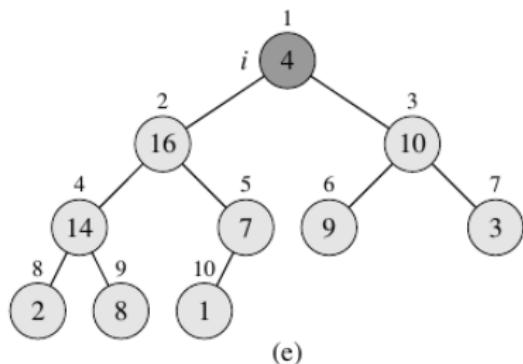


(b)

Building a heap



Building a heap



Heap is complete now for input array [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

Runtime of Build-Max-Heap

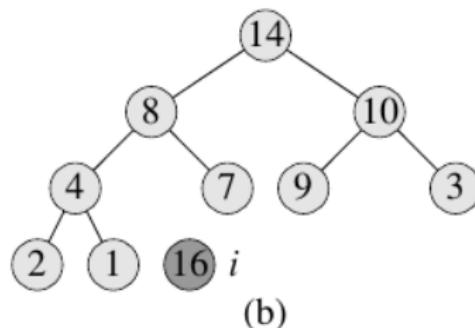
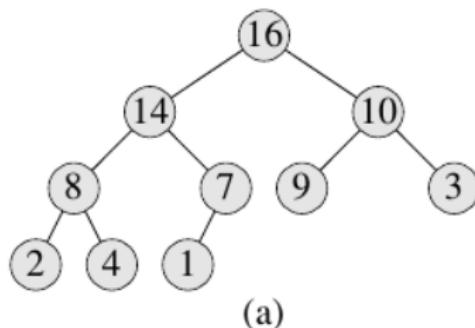
BUILD-MAX-HEAP(A)

- 1 $\text{heap-size}[A] \leftarrow \text{length}[A]$
- 2 **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1
- 3 **do** MAX-HEAPIFY(A, i)

We saw that the runtime of max-heapify is $O(\log n)$. Thus an immediate upper bound for the runtime of Build-Max-Heap is $O(n \log n)$. In fact, it can be shown that the runtime is $O(n)$ – using either bound does not change the asymptotic runtime of Heapsort.

Heapsort

How do we use all this for a sorting algorithm?



Above: Remove root 16, replace by leaf 1 and let 1 sink down.

Main idea:

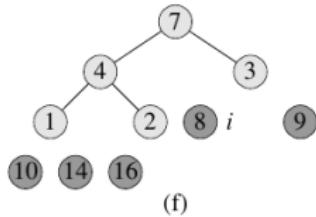
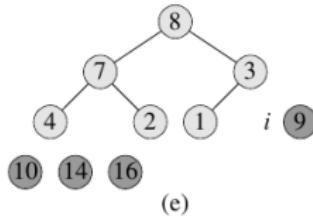
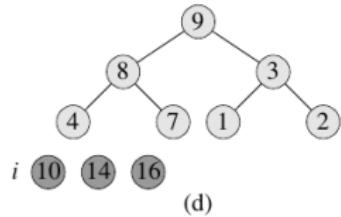
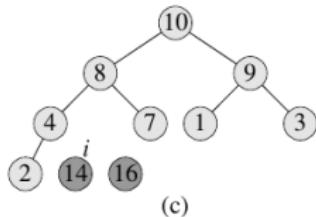
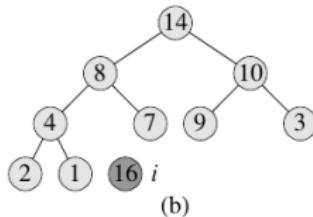
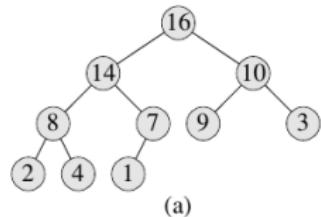
1. Reorder array elements into a heap in $O(n)$.
2. Remove root (the largest element), replace with last leaf (other elements unaffected and stay in order), place it where the root is and bubble it down to re-establish heap property in $O(\log n)$. Total $O(n \log n)$.

Heapsort pseudo-code

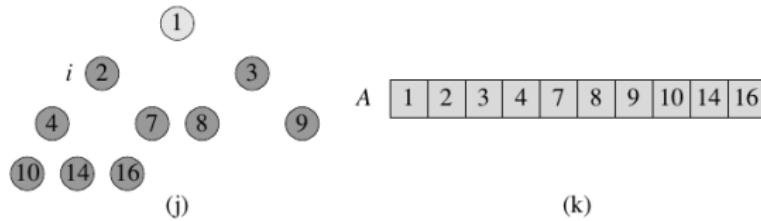
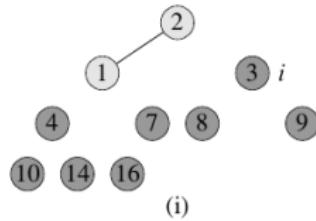
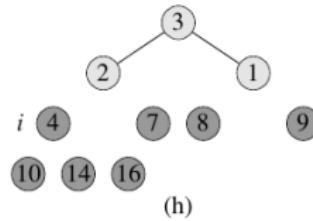
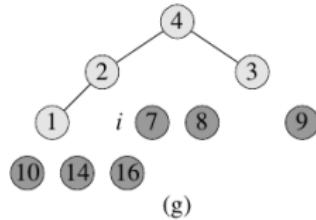
HEAPSORT(A)

- 1 BUILD-MAX-HEAP(A)
- 2 **for** $i \leftarrow \text{length}[A]$ **downto** 2
- 3 **do** exchange $A[1] \leftrightarrow A[i]$
- 4 $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
- 5 MAX-HEAPIFY($A, 1$)

Heapsort example 1



Heapsort example 1 continued



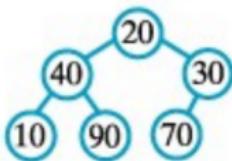
Heapsort example 2: array manipulations

Array view

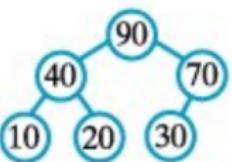
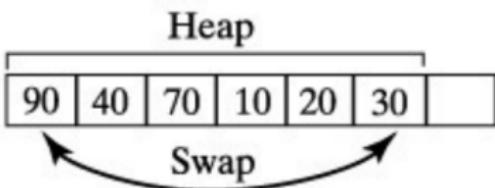
(a) The original array

20	40	30	10	90	70	
----	----	----	----	----	----	--

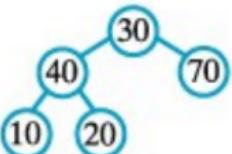
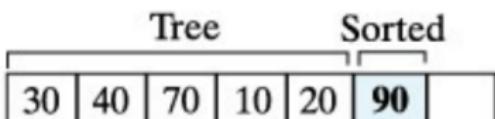
Tree view



(b) After reheap

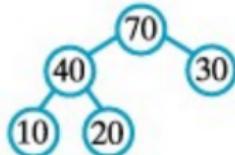
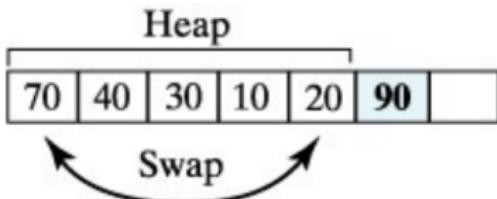


(c) After swapping

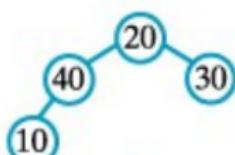
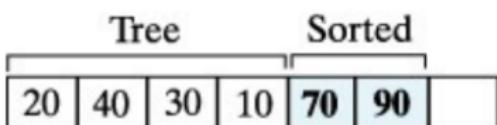


Heapsort example 2: array manipulations

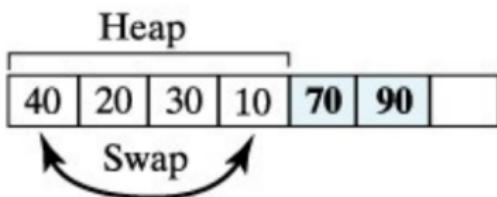
(d) After reheap



(e) After swapping



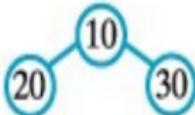
(f) After reheap



Heapsort example 2: array manipulations

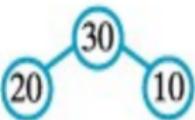
(g) After swapping

Tree	Sorted
10 20 30 40 70 90	



(h) After reheap

Heap
30 20 10 40 70 90



Swap ↗

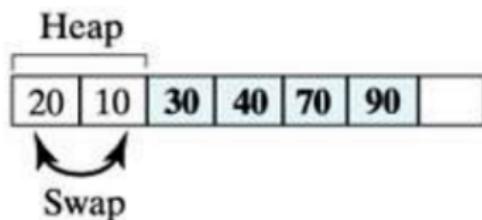
(i) After swapping

Tree	Sorted
10 20 30 40 70 90	

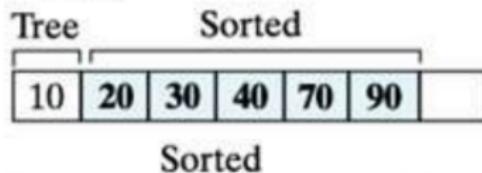


Heapsort example 2: array manipulations

(j) After reheap



(k) After swapping



(l) Array is sorted



Greedy algorithms and dynamic programming

Contents:

1. Divide-and-conquer
2. Greedy algorithms: the greedy choice property and optimal substructure principle
3. The Knapsack problem and NP-completeness
4. Dynamic programming

Paradigms

Paradigms of this section:

1. Divide and conquer: divide problem into several subproblems, solve recursively, combine
2. Greedy algorithms: always choose locally optimal next step
3. Dynamic programming: write down recursive (math) formula which reduces the optimal solution to a smaller problem, then solve all subproblems needed to express the overall solution, apply shortcuts if available

Divide and conquer

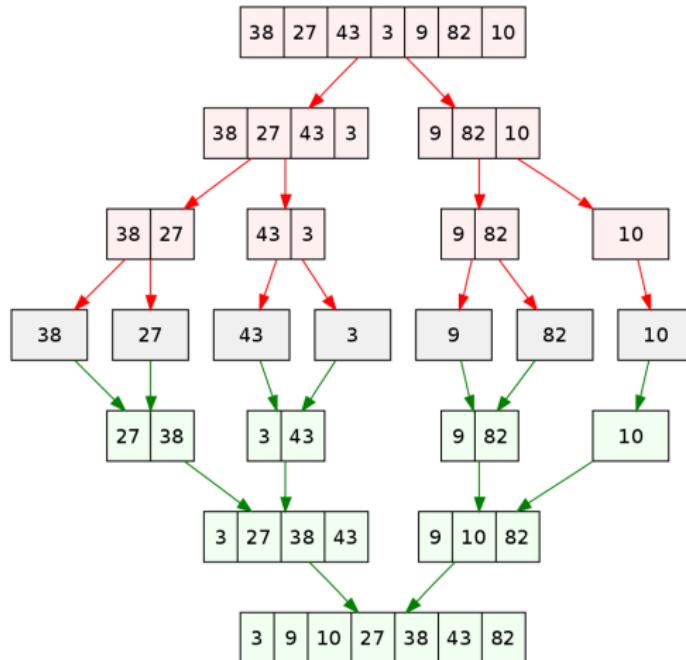


From Julius Caesar's *divide et impera*.

- ▶ Divide up problem into several subproblems
- ▶ Solve each subproblem recursively
- ▶ Combine solutions into an overall solution

Divide and conquer: MergeSort

Example: Sorting with 2 subproblems of size $n/2$, resulting in a runtime of $O(n \log n)$, e.g. MergeSort. Runtimes of divide-and-conquer methods can be computed with the Master Theorem.



Greedy algorithms

- ▶ When we have a choice to make, we make the one that looks best *right now*. We make a locally optimal choice in the hope that this will also lead to a globally optimal solution.
- ▶ Greedy algorithms often solve optimization problems.
- ▶ Greedy algorithms are often easy to state and code.
- ▶ Question: When are greedy solutions also optimal?
 - ▶ Problem must exhibit *optimal substructure*.
 - ▶ Problem must exhibit the *greedy-choice* property.

Name a problem which can be solved by a greedy approach... and one which cannot.

Greedy strategy

- ▶ Greedy choice: the one that looks best at any given moment during execution of the algorithm.
- ▶ **Greedy choice:** Prove that whenever there is a choice, one of then optimal choices is the greedy one. This justifies choosing the greedy option. Show that all but one of the subproblems resulting from the greedy choice are empty.
- ▶ **Optimal substructure:** This property is satisfied if an optimal solution can be constructed efficiently from optimal solutions of its subproblems.

Example: Counting money

- ▶ Task: return certain amount of money using fewest possible bills and coins
- ▶ Greedy solution: at each step, take the largest possible bill and coin, subtract from the current amount and repeat until entire amount is counted out
- ▶ Example: To return \$6.39, we choose...
 - ▶ a \$5 bill (total \$5)
 - ▶ a \$1 bill (total \$6)
 - ▶ a 25 cent coin (total \$6.25)
 - ▶ a 10 cent coin (total \$6.35)
 - ▶ four 1 cent coins (total \$6.39)
- ▶ For US bills and coins, the greedy algorithm always returns the optimal solution

Pseudo-code

CASHIERS-ALGORITHM (x, c_1, c_2, \dots, c_n)

SORT n coin denominations so that $c_1 < c_2 < \dots < c_n$

$S \leftarrow \emptyset$ ← set of coins selected

WHILE $x > 0$

$k \leftarrow$ largest coin denomination c_k such that $c_k \leq x$

IF no such k , RETURN "no solution"

ELSE

$x \leftarrow x - c_k$

$S \leftarrow S \cup \{k\}$

RETURN S

What is the invariant of the algorithm?

Correctness

Theorem: Cashier's algorithm is optimal for
 $\{c_1, \dots, c_n\} = \{1, 5, 10, 25, 100\}$

Proof by induction:

- ▶ Induction start: $x = 1$ is returned correctly as coin 1.
- ▶ Assume we are trying to return amount x . We assume that the induction hypothesis is true for any $x' < x$.
- ▶ Let $c_k \leq x < c_{k+1}$ the largest coin to fit into amount x . We claim we must return coin k . If not the table below shows that c_k cannot be substituted.
- ▶ Problem reduces to returning $x - c_k < x$ which is solved correctly by induction.

k	c_k	number	max value in opt. solution
1	1	4	4
2	5	1	$4+5=9$
3	10	2	$20+4=24$
4	25	3	$75+24=99$
5	100	no limit	

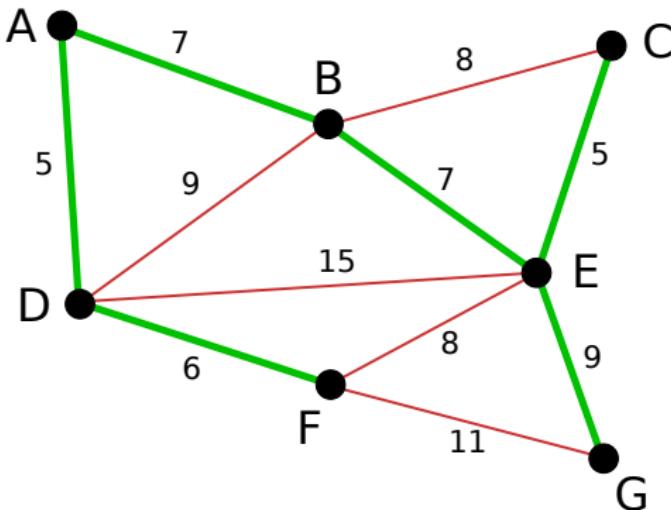
When is the Cashier algorithm not optimal?

- ▶ Consider US stamps: 1, 10, 21, 34, 70, 100, 350, 1225, 1500 cents
- ▶ Cashier's algorithm: $140 = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1$
- ▶ Optimal: $140 = 70 + 70$



Cashier's algorithm might not even lead to a feasible solution
(Example: set $\{7, 8, 9\}$ then $15 = 9 + ?$, although solution would be $15 = 8 + 7$)

Minimum spanning tree (MST)



Joseph Kruskal (1956)

Definition: A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Kruskal's algorithm

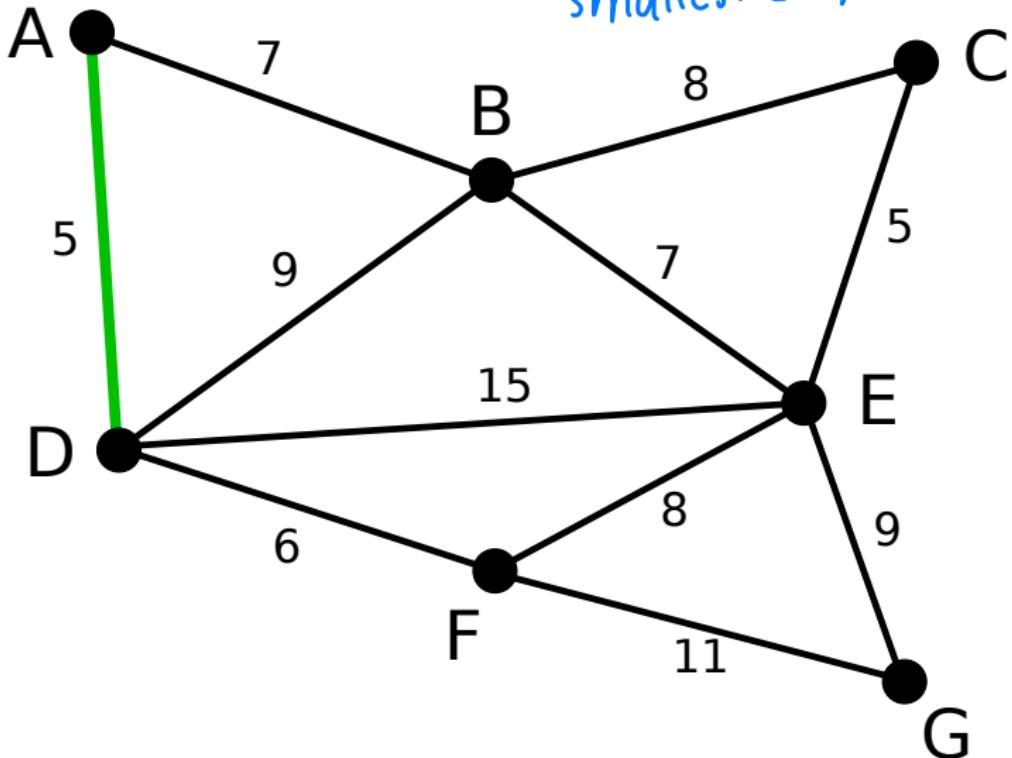
MST-KRUSKAL(G, w)

```
1    $A \leftarrow \emptyset$ 
2   for each vertex  $v \in V[G]$ 
3       do MAKE-SET( $v$ )
4   sort the edges of  $E$  into nondecreasing order by weight  $w$ 
5   for each edge  $(u, v) \in E$ , taken in nondecreasing order by weight
6       do if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7           then  $A \leftarrow A \cup \{(u, v)\}$ 
8               UNION( $u, v$ )
9   return  $A$ 
```

Greedy strategy: in each step, add the smallest unconsidered edge
which connects a new vertex to the MST (runtime?)

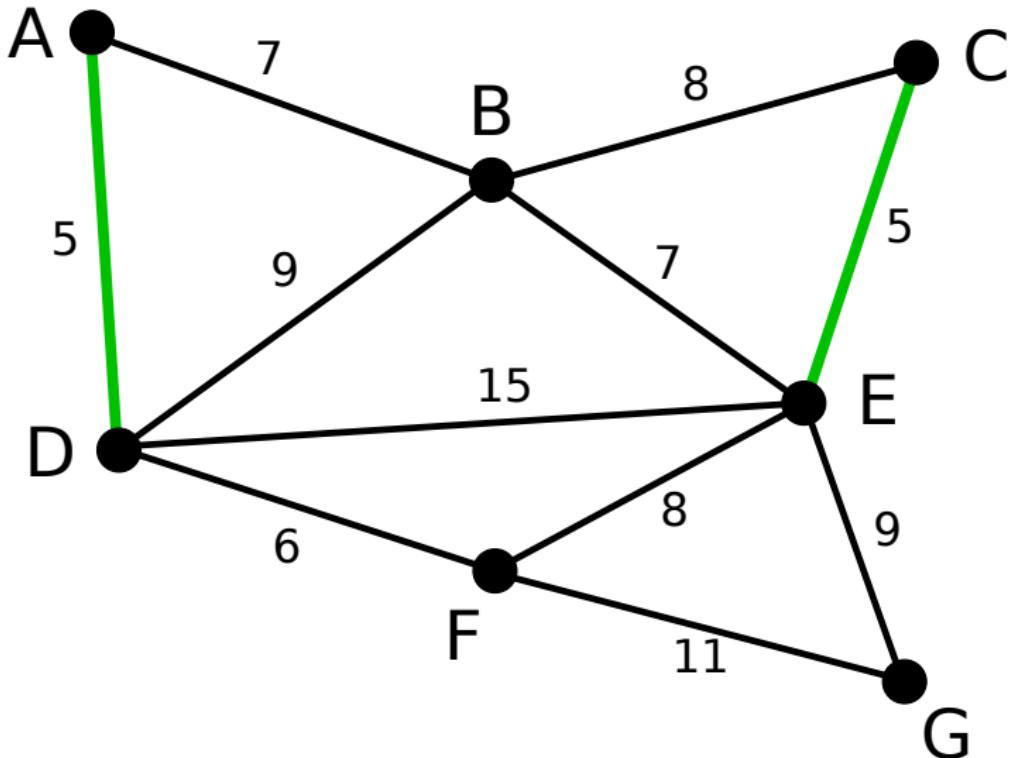
Kruskal 1/6

① first choose
smallest edge



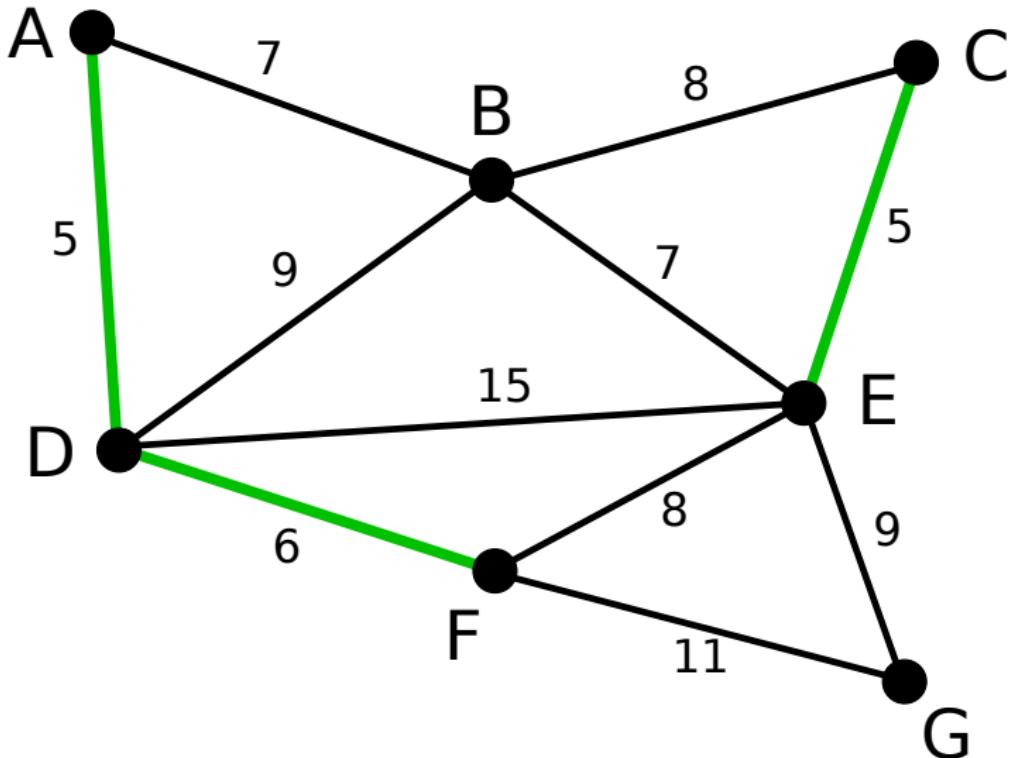
Kruskal 2/6

② next smallest

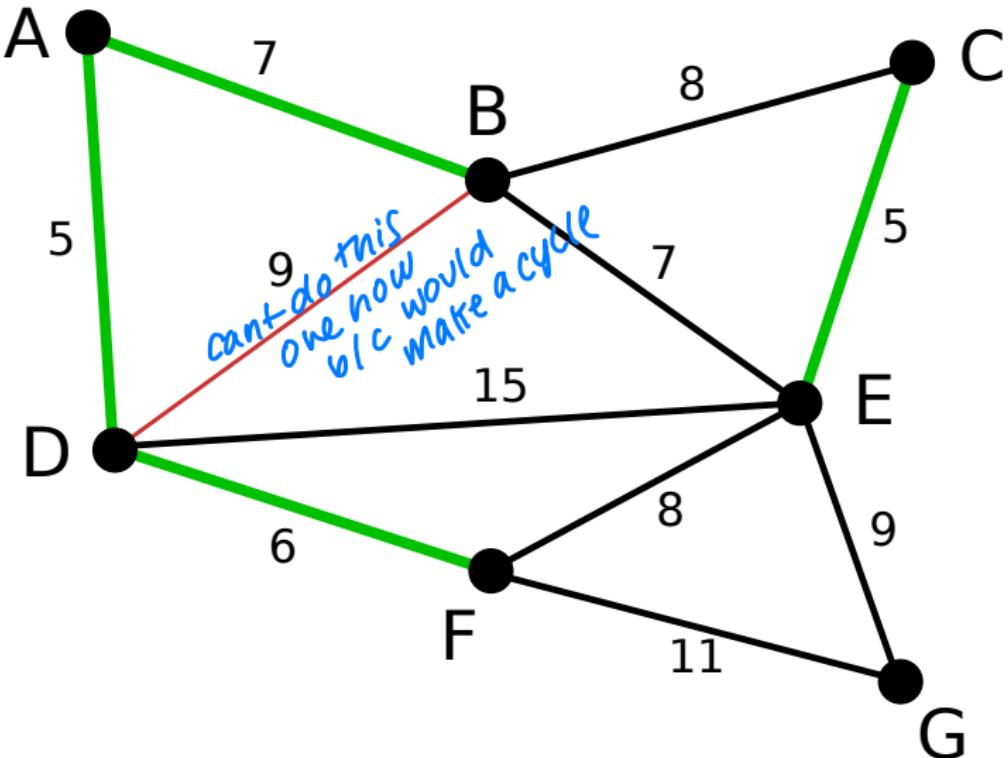


Kruskal 3/6

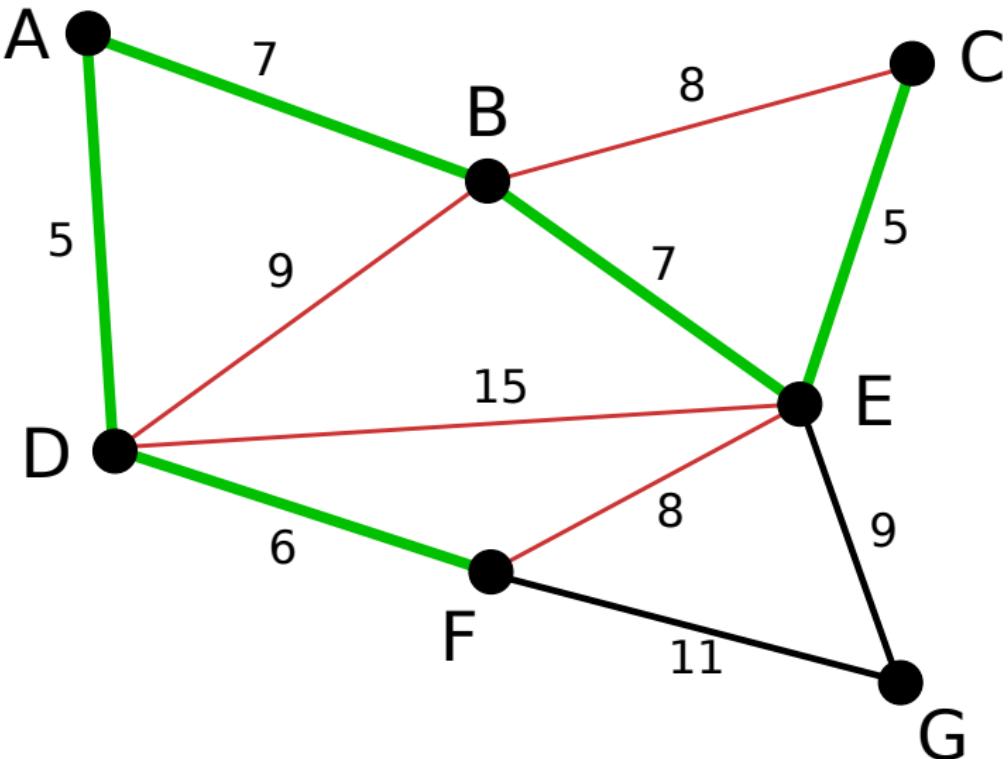
③ next smallest



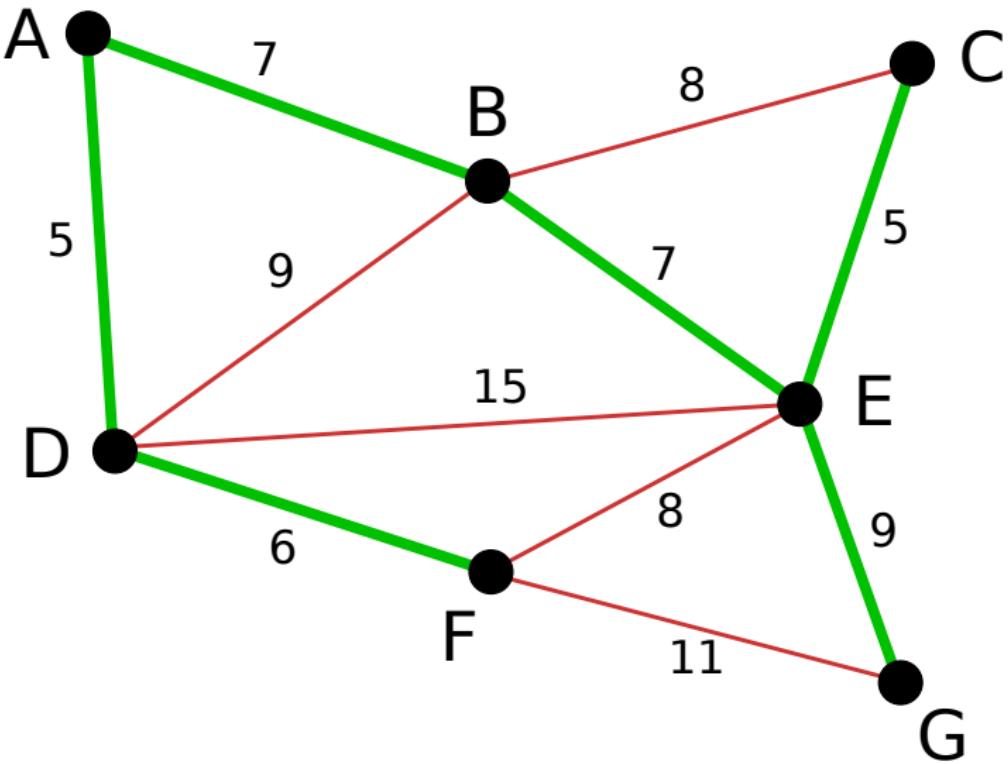
Kruskal 4/6



Kruskal 5/6



Kruskal 6/6



Correctness of Kruskal's algorithm

Let $G = (V, E)$ with $E = \{e_1, \dots\}$ be a graph. We prove that Kruskal's algorithm produces a spanning tree, and that this spanning tree is indeed an MST.

Kruskal's algorithm produces a spanning tree: Let G be a connected, weighted graph and let Y be the subgraph of G produced by the algorithm. Y cannot have a cycle as we only connect isolated nodes in every step. Y cannot be disconnected, since the first encountered edge that joins two components of Y would have been added by the algorithm. Thus, Y is a spanning tree of G .

Correctness of Kruskal's algorithm

Kruskal's algorithm produces an MST:

Induction:

- ▶ Show proposition (P): If F is the set of edges chosen at any stage of the algorithm, then there is some minimum spanning tree that contains F .
- ▶ Induction start: When F is empty, the statement is true for any minimum spanning tree.
- ▶ Hypothesis: Assume statement is true for some non-final edge set F and let T be a minimum spanning tree that contains F . Two cases:
 1. If the next chosen edge e is also in T , then P is true for $F + e$.
 2. If $e \notin T$ then $T + e$ has a cycle C . This cycle contains edges which do not belong to F , since e does not form a cycle when added to F but does in T . Let f be an edge which is in C but not in $F + e$. Note that f also belongs to T , and P has not been considered by the algorithm. Therefore f must have a weight at least as large as e . Then $T - f + e$ is a tree, and it has the same or less weight as T . So $T - f + e$ is an MST containing $F + e$ and again (P) holds.

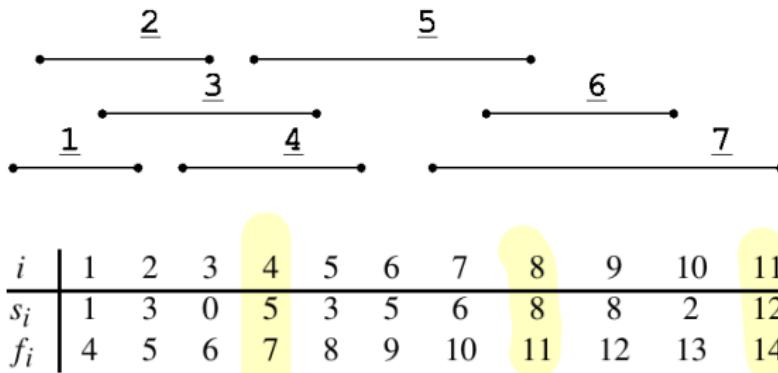
Therefore, by the principle of induction, (P) holds when F has become a spanning tree, and thus F is a minimum spanning tree itself.

Activity selection

optimal schedule to fit most activities

Also called *task scheduling*...

- ▶ Set $S = \{a_1, \dots, a_n\}$ of proposed activities
- ▶ Each activity a_i has a start time s_i and a finish time f_i , and activity a_i takes place during the half-open interval $[s_i, f_i)$
- ▶ Two activities are *compatible* if their intervals do not overlap.
- ▶ Activity selection: select maximal number of compatible activities.



What are compatible activities here?

Activity selection

Roadmap:

- ▶ Formulate dynamic programming solution for the activity selection problem.
- ▶ Consider several choices to determine which subproblems to look at for the optimal solution.
- ▶ Realize we only need to look at one choice, the greedy choice, and that selecting the greedy choice makes one subproblem empty so that effectively only one nonempty subproblem remains.
- ▶ Arrive at recursive greedy algorithm.

Optimal substructure

Define

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\},$$

the set of all activities that start after activity a_i finishes and before activity a_j starts. All activities in S_{ij} are compatible with a_i and a_j , and in fact with all others that finish no later than a_i and start no earlier than a_j . We also add two artificial activities a_0 and a_{n+1} with trivial start ($T = 0$) and end ($T = n + 1$) times.

Idea: Suppose a_k is part of the optimal solution. Then select activities from a_1, \dots, a_{k-1} which are compatible and finish before a_k starts. Likewise select activities from a_{k+1}, \dots, a_n which are compatible and start after a_k finishes. The two subproblems must be solved optimally.

Optimal substructure

Sort activities according to end times

$$f_0 \leq f_1 \leq \cdots \leq f_n < f_{n+1}.$$

Claim: $S_{ij} = \emptyset$ whenever $i \geq j$.

Proof by contradiction: assume such two activities exist in S_{ij} , then $f_i \leq s_k < f_k \leq s_j < f_j$, contradicting that activity i finishes later than f_j .

Thus the problem reduces to finding a compatible maximum-size solution from S_{ij} for $0 \leq i < j \leq n + 1$ since for other combinations of indices the set S_{ij} is empty.

Optimal substructure

Assume the optimal solution A_{ij} of S_{ij} contains activity a_k . Then the solutions A_{ik} of S_{ik} and A_{kj} of S_{kj} used within the overall solution must be optimal as well. This allows to split the problem, leading to the following dynamic programming formulation:

Let $c[i, j]$ be the size of the optimal set of compatible activities in S_{ij} . We have $c[i, j] = 0$ whenever $S_{ij} = \emptyset$, in particular when $i \geq j$. Then,

$$c[i, j] = c[i, k] + 1 + c[k, j],$$

before ↘ act k ↗ #after

under the assumption that we know that a_k is part of the optimal solution.

But we don't know k ...

Optimal substructure

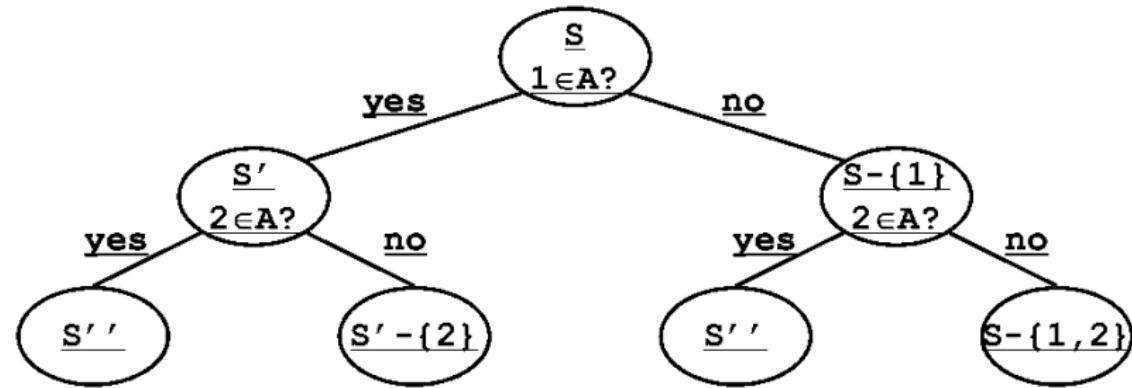
We saw that assuming we know k ,

$$c[i, j] = c[i, k] + 1 + c[k, j].$$

Instead, we use the recursive formulation:

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{i < k < j, a_k \in S_{ij}} c[i, k] + 1 + c[k, j] & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Repeated subproblems



Activity selection problem exhibits greedy choice property: locally optimal choice \rightarrow globally optimal choice

Greedy solution

If S is an activity selection problem sorted by finish times, then there exists an optimal solution $A \subseteq S$ such that $\{1\} \in A$.

- ▶ Here, $\{1\}$ is the first activity, no matter what it is.
- ▶ Proof: If there exists an optimal solution B which does not contain $\{1\}$, we can always replace the first activity in B with $\{1\}$ (why?).
- ▶ Same number of activities, thus still optimal...

Optimal substructure property

The problem also exhibits the *optimal substructure property*:

- ▶ There is an optimal solution to the subproblem S_{ij} that includes the activity with the smallest finish time in S_{ij} .
- ▶ Thus we can assume the optimal solution contains a_1 (sorted by increasing finish times).
- ▶ Therefore, no need to search the entire space with the dynamic programming formulation. Instead, use the greedy choice in every step.

Algorithm

```
RECURSIVE-ACTIVITY-SELECTOR( $s, f, i, n$ )
1    $m \leftarrow i + 1$ 
2   while  $m \leq n$  and  $s_m < f_i$        $\triangleright$  Find the first activity in  $S_{i,n+1}$ .
3       do  $m \leftarrow m + 1$ 
4   if  $m \leq n$ 
5       then return  $\{a_m\} \cup$  RECURSIVE-ACTIVITY-SELECTOR( $s, f, m, n$ )
6   else return  $\emptyset$ 
```

Runtime: $O(n)$.

Algorithm can be converted into an iterative method.

Recap of idea

- ▶ Sort the activities by finish time.
- ▶ Schedule the first activity in that ordering.
- ▶ Then schedule the next activity which starts after the previous activity finishes.
- ▶ Repeat until no more candidate activity exist.

Intuition: Always select the activity which finishes first in order to maximize the remaining time used to schedule additional activities.

Recipe for greedy methods

- ▶ Formulate problem such that we have to make a choice and are left with one subproblem to solve afterwards.
- ▶ Prove there is always an optimal solution that makes the greedy choice (thus justifying the greedy choice).
- ▶ Show that:
$$\begin{aligned} \text{greedy choice} + \text{optimal subproblem solution} \\ = \text{optimal overall solution} \end{aligned}$$
- ▶ Solve top-down with the greedy choice. Might have to process input data in order to apply the greedy choice (e.g., sort finish times in increasing order).

Recap: Greedy paradigm

Two ingredients:

- ▶ **greedy choice property**: a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- ▶ **optimal substructure**: an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.

Examples for both properties? (Shortest path, MST, ...)

Dynamic programming

Typically based on some recursion formula of the type

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{i < k < j, a_k \in S_{ij}} c[i, k] + 1 + c[k, j] & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Dynamic programming vs. greedy:

Greedy algorithms differ from dynamic programming: In dynamic programming, we make a choice at each step, but the choice usually depends on the solutions to subproblems. Consequently, we typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems. In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem arising after the choice is made.

Which problems can be solved with a greedy method?

How can one tell if a greedy algorithm will solve a particular optimization problem? There is no way in general, but the greedy-choice property and optimal substructure are the two key ingredients. If we can demonstrate that the problem has these properties, then a greedy algorithm is not far...

Because the optimal-substructure property is exploited by both the greedy and dynamic-programming strategies, one might be tempted to generate a dynamic programming solution to a problem when a greedy solution suffices, or one might mistakenly think that a greedy solution works when in fact a dynamic programming solution is required.

Now: illustrate subtleties between the two techniques...

Which problems can be solved with a greedy method?

- ▶ Minimum spanning tree
- ▶ Activity selection
- ▶ Shortest path and many other graph problems such as max-flow

Which ones cannot be solved by a greedy method? Many ... among them NP-complete problems.

Knapsack problem

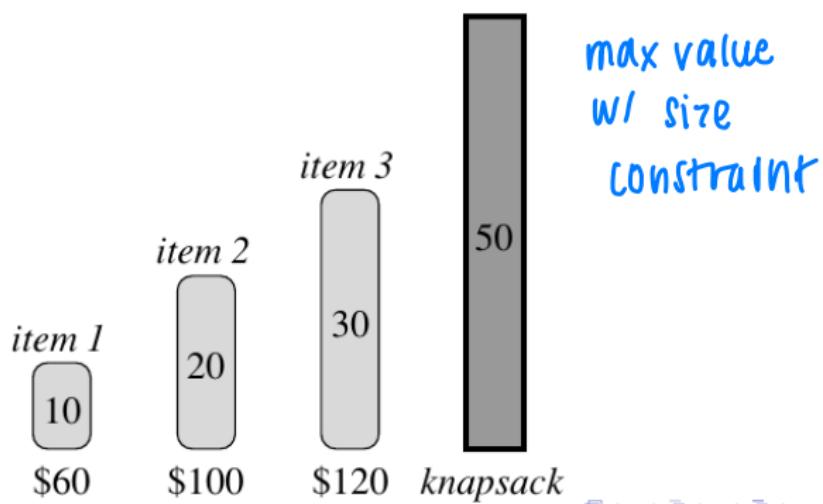
A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?



Knapsack problem

Mathematically:

- ▶ Knapsack of capacity W , set S consisting of n items.
- ▶ Each item $i \in S$ has weight w_i and value v_i .
- ▶ Task: Which items to choose such their value is maximized and they fit into the knapsack, i.e., $\max_{T \subseteq S} \sum_{i \in T} v_i$ subject to $\sum_{i \in T} w_i \leq W$. Each item can only be chosen once.



Two variations

- ▶ **0-1 Knapsack problem:** Each item must either be included in the Knapsack or left out. No greedy strategy but dynamic programming solution leading to a *pseudo-polynomial* time algorithm.
- ▶ **Fractional Knapsack problem:** We are allowed to take fractions of the items. Fractional Knapsack can be solved by a greedy strategy (in polynomial time)! Which one?

Think of items as either gold nuggets (0-1 Knapsack) or buckets of gold dust (Fractional Knapsack problem). Which problem seems easier? Is there a general trend?

Fractional Knapsack

Fractional Knapsack can be solved by a greedy strategy! Which one?

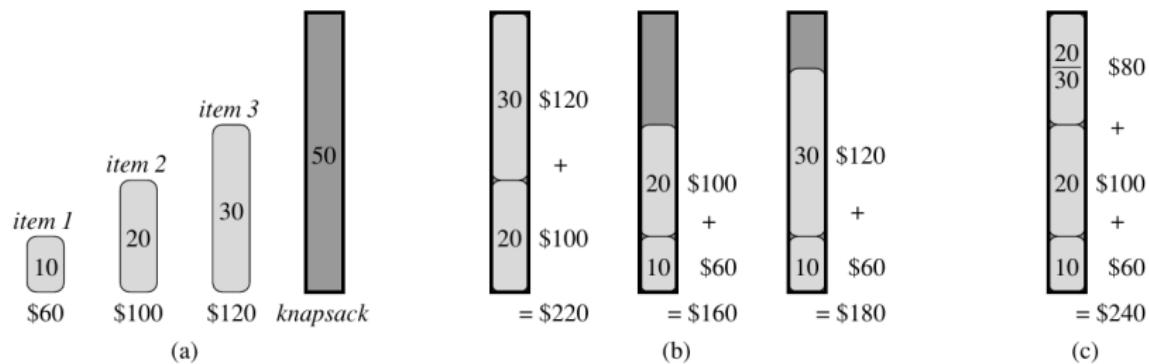
- ▶ Compute value per weight v_i/w_i for each item.
- ▶ Take as much as you can of the item having the highest ratio of value per weight.
- ▶ If Knapsack capacity W is not yet reached, repeat with next item in decreasing sorted order.

Runtime: $O(n \log n)$ (for sorting), where n is the number of items.

Correctness?

Fractional Knapsack

Greedy strategy does not work for the 0-1 Knapsack problem.



Greedy strategy selects item 1 since its ratio value/weight is best, but item 1 is not part of the optimal solution (b). The fractional solution is shown in (c).

Fractional Knapsack

Questions:

1. Prove that the fractional knapsack problem has the greedy-choice property. (Proof by contradiction...)
2. Prove that the fractional knapsack problem has the optimal substructure property. (Proof by induction...)

0-1 Knapsack

0-1 Knapsack is NP-complete. NP-completeness means the problem is at least as hard as any problem in NP. All NP-complete problems can be expressed as a binary optimization problem:

$$\max \sum_{i \in T} v_i \quad \text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

This can be formulated as: Find $z_i \in \{0, 1\}$ such that

$$\max_{z_i} \sum_{i=1}^n z_i v_i \quad \text{subject to} \quad \sum_{i=1}^n z_i w_i \leq W.$$

The problem is called *0-1 problem* because we try to find the optimal assignment of indicators $z_i \in \{0, 1\}$ which denote if item i is accepted or rejected in the optimal solution. Other examples?

Brute force

- ▶ For NP-complete problems, any known methods are asymptotically not faster than exponential.
- ▶ That means we have to try all 2^n possible combinations (think of binary vectors of length n which indicate which item to take and which item to leave out – there are exactly 2^n such vectors).
- ▶ We go through all combinations and find the one with the most total value and with total weight less or equal to W .
- ▶ Running time will be $O(2^n)$.

Dynamic programming solution of Knapsack

- as if a sorting algo depends on how big #'s are
- can drop if W is const
always ?? unclear answer from george

We want to give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is number of items and W is the maximum weight of items that the thief can put in his knapsack.

- b/c not same time for 2 probs w/
same n ?

Disclaimer: Such a method is valid but does not solve P=NP since its runtime depends on the values of the items. In the strict sense, the runtime is only allowed to depend on n .

Dynamic programming solution of Knapsack

We need to carefully identify subproblems.

If items are labeled $\{1, \dots, n\}$ then a subproblem would be to find an optimal solution for the subset

$$\underline{S_k = \{\text{items labeled } 1, 2, \dots, k\}}.$$

- ▶ Subproblem definition is valid.
- ▶ Question: Can we describe the final solution S_n in terms of the subproblems S_k . Answer: We cannot. Why?

Counterexample

Consider the following items with given weight and value for a knapsack of $W = 20$:

item	weight	value	S_4	S_5
1	2	3	x	x
2	3	4	x	x
3	4	5	x	x
4	5	8	x	x
5	9	10		x

Observations:

- ▶ Subproblem S_4 considers the items $1, \dots, 4$ only. Subproblem S_5 considers the items $1, \dots, 5$.
- ▶ S_4 : best solution is $\{1, 2, 3, 4\}$ with weight 14 and value 20.
- ▶ S_5 : best solution is $\{1, 3, 4, 5\}$ with weight 20 and value 26.
- ▶ The solution of S_4 is not part of the solution of S_5 , hence *optimal substructure* property not satisfied.

Dynamic programming solution of Knapsack

- ▶ Counterexample showed that classic substructure does not hold.
- ▶ New definition: We add another parameter w which represents the exact weight of each subset of items.
- ▶ *The subproblem will be to compute $V[k, w]$, the optimal value of subproblems containing items up to k with weight at most w .*

Recursive formula

Consider S_k
max weight w

Recursive formula for dynamic programming formulation:

$$V[k, w] = \begin{cases} V[k - 1, w] & \text{if } w_k > w, \\ \max\{V[k - 1, w], V[k - 1, w - w_k] + v_k\} & \text{otherwise.} \end{cases}$$

don't take k *do take k*

Intuition: The best subset of S_k with total weight w is either...

- ▶ the best subset of S_{k-1} which has total weight w , or
- ▶ the best subset of S_{k-1} which has total weight $w - w_k$ plus item k .

Recursive formula

Recursive formula for dynamic programming formulation:

$$V[k, w] = \begin{cases} V[k - 1, w] & \text{if } w_k > w, \\ \max\{V[k - 1, w], V[k - 1, w - w_k] + v_k\} & \text{otherwise.} \end{cases}$$

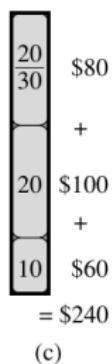
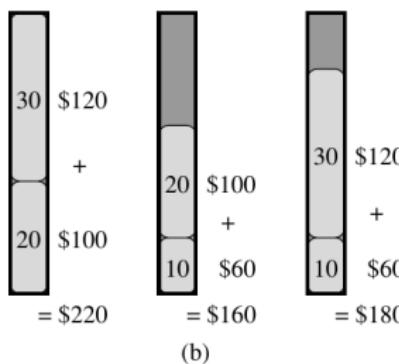
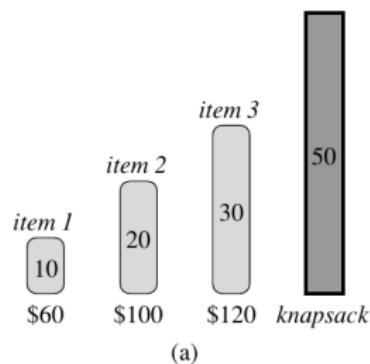
Look at the condition in the definition of $V[k, w]$:

- ▶ The best subset of S_k with total weight w either contains item k or not...
- ▶ If $w_k > w$ then surely item k cannot be part of the optimal solution.
- ▶ If $w_k \leq w$ then item k can be in the solution and we choose the option with higher value.

solution = $V[n, W]$

Knapsack optimal substructure

- ▶ The previous construction ensures optimal substructure for 0-1 Knapsack.
- ▶ Consider the most valuable assignment with weight $\leq W$. If we remove some item j , we know that the remainder must be the most valuable assignment weighting at most $W - w_j$.
- ▶ Still ... we won't have the greedy choice property.



Optimal substructure

We have seen the following recursive formula:

$$V[k, w] = \begin{cases} V[k - 1, w] & \text{if } w_k > w, \\ \max\{V[k - 1, w], V[k - 1, w - w_k] + v_k\} & \text{otherwise.} \end{cases}$$

Pseudo-code from wikipedia

```
// Input:  
// Values (stored in array v)  
// Weights (stored in array w)  
// Number of distinct items (n)  
// Knapsack capacity (W)  
// NOTE: Array "v" and "w" are assumed to start with index 1.  
  
for j from 0 to W do:  
    V[0, j] := 0  
  
for i from 1 to n do:  
    for j from 0 to W do:  
        if w[i] > j then:  
            V[i, j] := V[i-1, j]  
        else:  
            V[i, j] := max(V[i-1, j], V[i-1, j-w[i]] + v[i])
```

Runtime $O(nW)$ and space requirement $O(nW)$. This is seen immediately from the two nested loops). Again, the runtime depends on the problem instance, hence *pseudo-polynomial* time

Example run

We are given the Knapsack problem:

item	weight	value
1	23	505
2	26	352
3	20	458
4	18	220
5	32	354
6	27	414
7	29	498
8	26	545
9	30	473
10	27	543

All there is to do is to apply formula in correct order... see Python demo.

More on dynamic programming

- ▶ Dynamic Programming (DP) is an algorithm design technique for optimization problems: often minimizing or maximizing.
- ▶ Like divide and conquer, DP solves problems by combining solutions to subproblems.
- ▶ Unlike divide and conquer, subproblems are not independent, e.g., subproblems may share subsubproblems. However, it needs to hold true that solutions to one subproblem do not affect the solutions to other subproblems.
- ▶ DP reduces computation by
 - ▶ Solving subproblems in a bottom-up fashion.
 - ▶ Storing solution to a subproblem the first time it is solved.
 - ▶ Looking up the solution when subproblem is encountered again.

Overlapping subproblems

DP is a problem-solving strategy that applies to problems composed of overlapping sub-problems:

Example: Computing the n th Fibonacci number $F(n)$, defined by

$$F(N) = F(N - 1) + F(N - 2)$$

$$F(N - 1) = F(N - 2) + F(N - 3)$$

DP can be thought of as "recursion with memory". We will not solve the same sub-problem more than once. Instead, the answers to sub-problems are stored for later recall, a process called *memorizing*.

When sub-problems do not overlap, we use another strategy: Divide and conquer. Example: MergeSort.

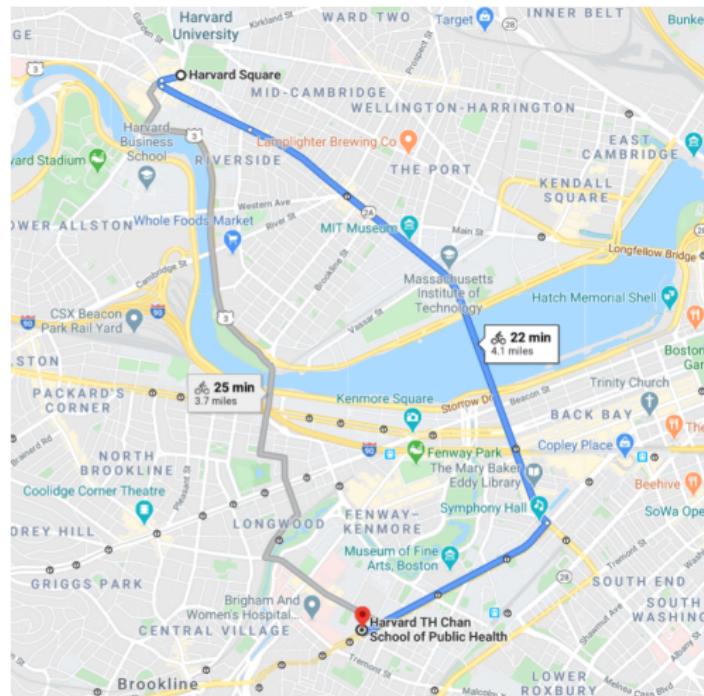
Dynamic programming and optimal substructure

DP is effective for problems with optimal substructure, meaning that the "grand problem" can be solved by efficiently combining the results of its sub-problems.

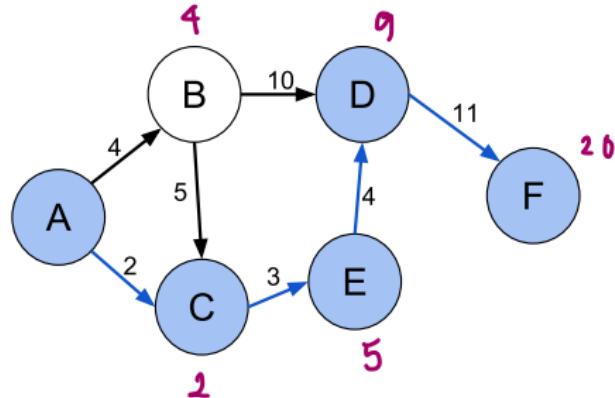
- ▶ Example of a problem with optimal substructure: If the shortest path from A to B goes through C, then it contains the shortest path from A to C.
- ▶ Example of a problem without optimal substructure: If the cheapest ticket from A to B has layovers at C then D, this does not imply that the cheapest ticket from A to D has a layover in C.

Dynamic programming and optimal substructure

Example of a problem with optimal substructure: If the shortest path from A to B goes through C, then it contains the shortest path from A to C.



Dijkstra's algorithm



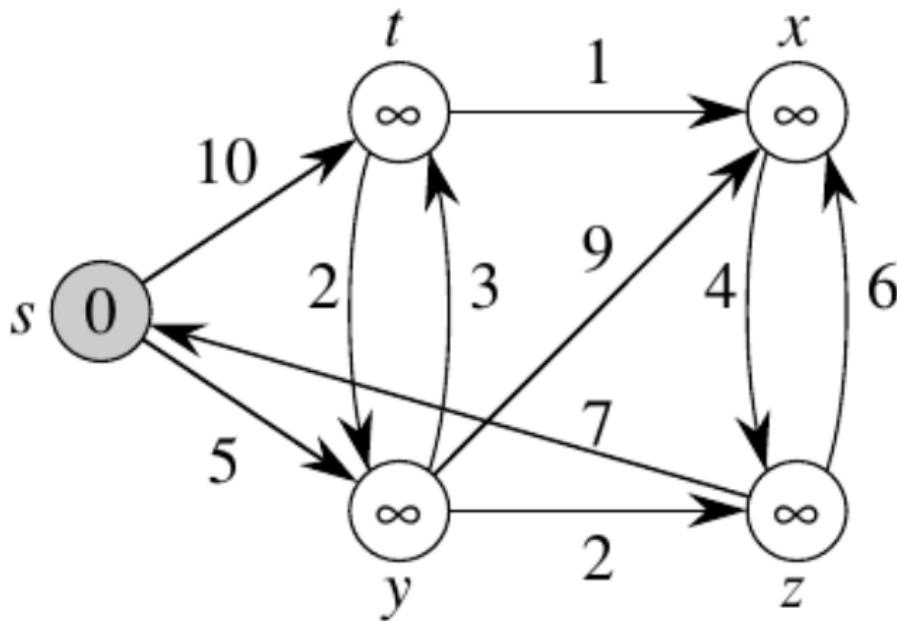
Observation: If Z is a vertex on the shortest path p from A to F then the optimal path from A to Z (and from Z to F) must be contained in p . Dynamic programming formulation:

$$d(x, y) = \min_{z \in V} \{d(x, z) + d(z, y)\},$$

where $d(\cdot, \cdot)$ is the shortest distance between two nodes.

Dijkstra's algorithm

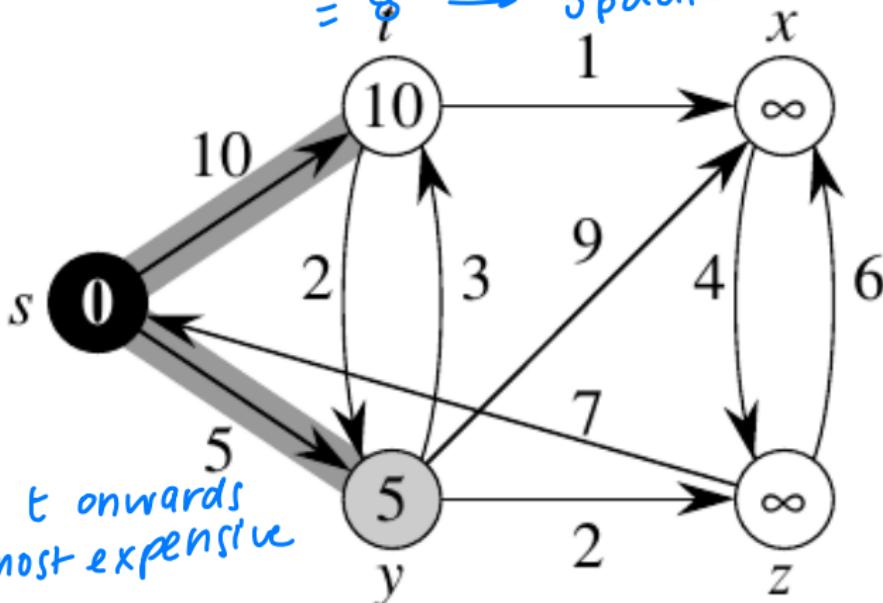
compute k update dist from start to every other node



(a)

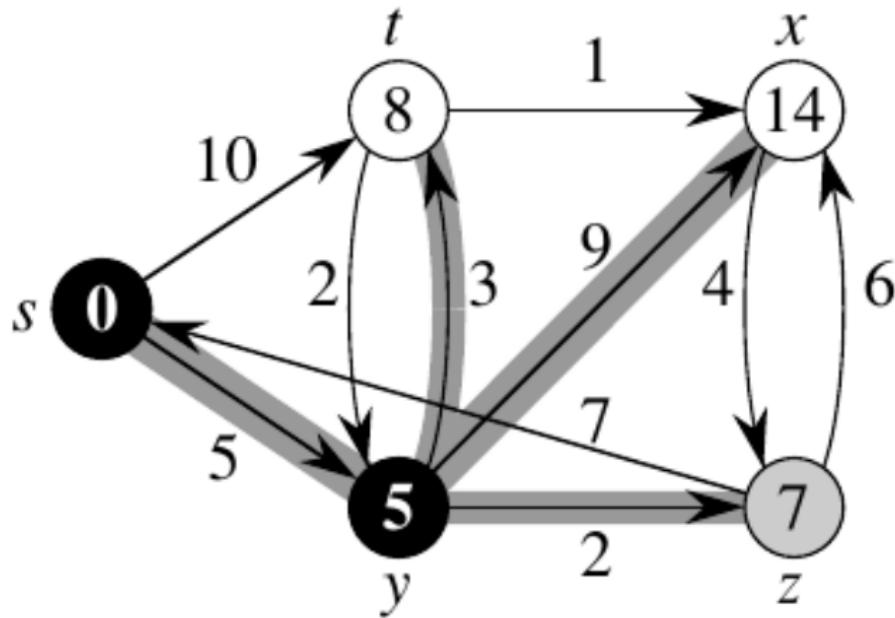
Dijkstra's algorithm

$$\begin{aligned} & \min \{d(s,t), d(s,y) + d(y,t)\} \\ &= \min \{10, 8\} \\ &= 8 \rightarrow \text{update} \end{aligned}$$



- dont look t onwards b/c its most expensive so far
- update nodes through s-y path (b)

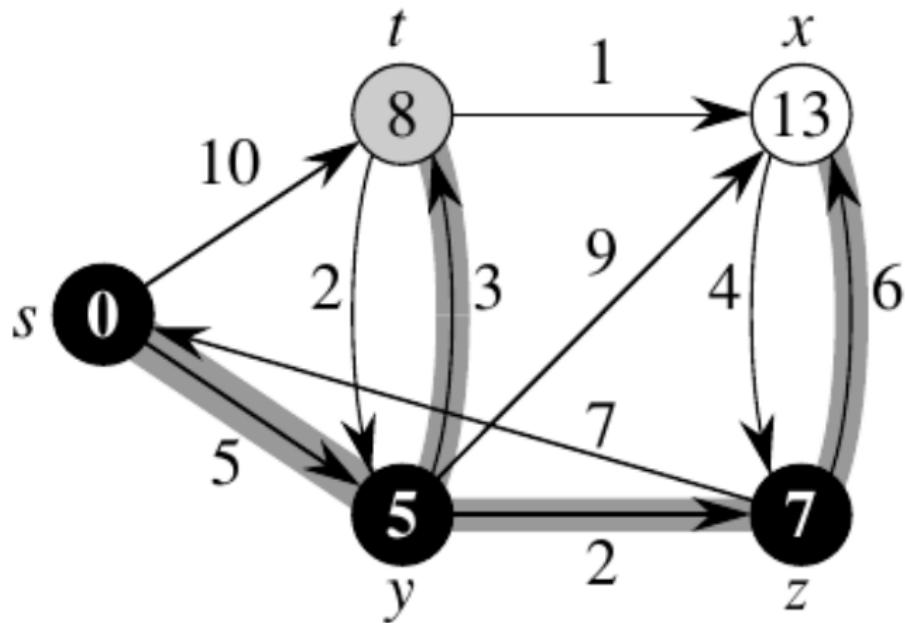
Dijkstra's algorithm



(c)

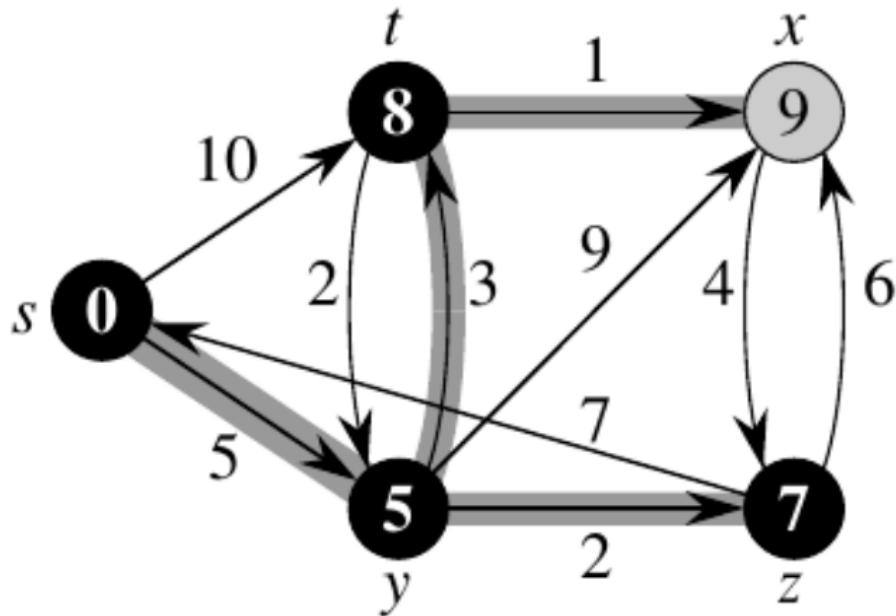
update through s-y-z path which is min

Dijkstra's algorithm



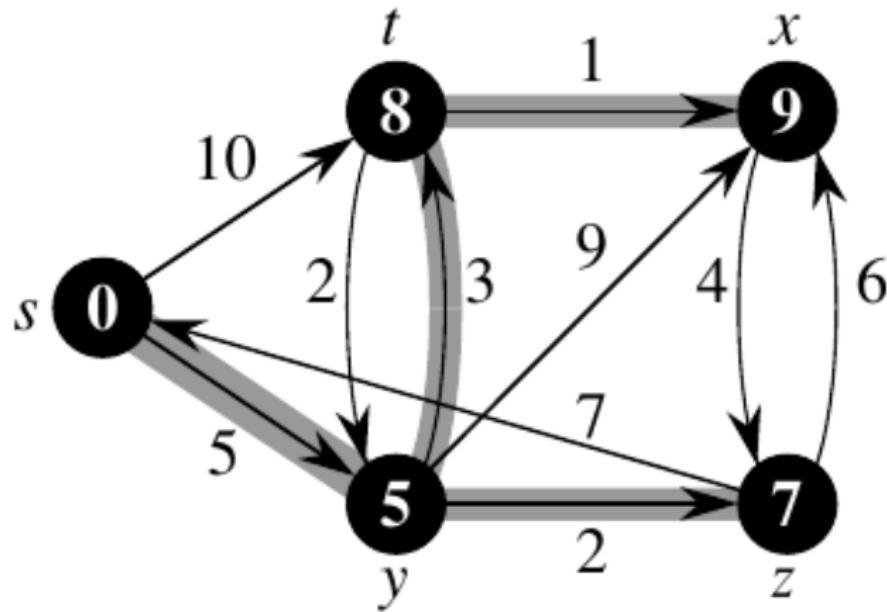
(d)

Dijkstra's algorithm



(e)

Dijkstra's algorithm



(f)

Recipe for dynamic programming

- ▶ Characterize structure of an optimal solution.
- ▶ Define value of optimal solution recursively.
- ▶ Compute optimal solution values either top-down with caching or bottom-up in a table.
- ▶ Construct an optimal solution from computed values.

Sequence alignment

Alignment of sequences. Example:

mathamatiks

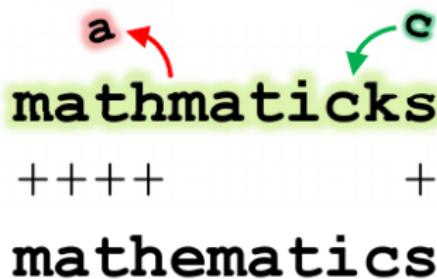
++++ +++++ +

mathematics

Score: 9

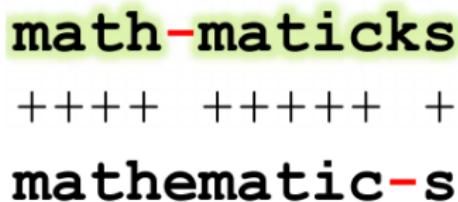
Sequence alignment

Only adding and deleting letters:


mathmaticks
++++ +
mathematics

Score: 5

Allowing gaps:


math-maticks
++++ ++++ +
mathemati-c-s

Score: 10

Dynamic programming solution

Smith-Waterman algorithm:

- ▶ Input sequences $A = a_1 \dots a_n$ and $B = b_1 \dots b_m$ to be aligned according to some score function.
- ▶ Determine the substitution matrix and the gap penalty scheme:
 1. $s(a, b)$ similarity score
 2. W_k the penalty for introducing a gap of length k
- ▶ Construct a scoring matrix H and initialize its first row and first column. The size of the scoring matrix is $(n + 1) \times (m + 1)$ with indices starting from zero:

$$H_{k0} = H_{0l} = 0 \quad 0 \leq k \leq n, 0 \leq l \leq m$$

Dynamic programming solution

Smith-Waterman algorithm: Dynamic programming recursion

$$H_{ij} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j) \\ \max_{k \geq 1} H_{i-k,j} - W_k \\ \max_{l \geq 1} H_{i,j-l} - W_l \\ 0 \end{cases}$$

where

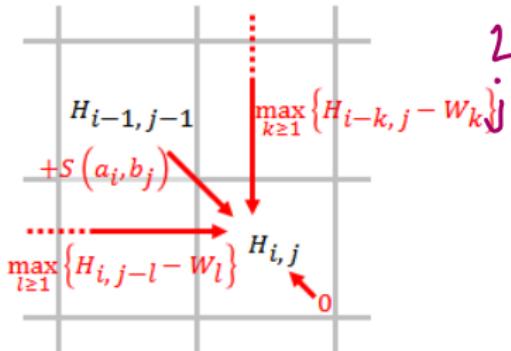
- ▶ $H_{i-1,j-1} + s(a_i, b_j)$ is the score of aligning a_i and b_j
- ▶ $H_{i-k,j} - W_k$ is the score if a_i is at the end of a gap of length k
- ▶ $H_{i,j-l} - W_l$ is the score if b_j is at the end of a gap of length l

Need to program and run...

Shortcut leading to a greedy method?

Smith-Waterman algorithm

G - T T - A C
G G T T G A C



Fill the scoring matrix

	T	G	T	T	A	C	G	G
G	0	0	0	0	0	0	0	0
G	0	0	3	→1	0	0	0	3
G	0	0	3	→1	0	0	0	6
T	0	3	→1	6	→4	→2	0	1
T	0	3	→1	4	9	→7	5	3
G	0	1	6	→4	7	6	4	8
A	0	0	4	3	5	10	8	6
C	0	0	2	1	3	8	13	11
T	0	3	→1	5	4	6	11	10
A	0	1	0	3	2	7	9	8

Parallel computing

Contents:

- ▶ Speedup
- ▶ Parallel architectures
- ▶ Message passing
- ▶ Flynn's Classifications
- ▶ Software tools
- ▶ Embarrassingly Parallel Computations
- ▶ Synchronization and locks
- ▶ MPI, Map/Reduce

Slides adapted from Curtis Huttenhower.

Good resource for further reading:

- ▶ <https://www.clear.rice.edu/comp422/lecture-notes/index.html>

Speedup and Efficiency of Parallel Algorithms

- ▶ Let $T^*(n)$ be the time complexity of a sequential algorithm to solve a problem P of input size n .
- ▶ Let $T_p(n)$ be the time complexity of a parallel algorithm to solves P on a parallel computer with p processors.

Speedup:

- ▶ $S_p(n) = T^*(n)/T_p(n)$
- ▶ $S_p(n) \leq p$. Why is that?
- ▶ Best possible: $S_p(n) = p$ when $T_p(n) = T^*(n)/p$

Efficiency

Speedup:

- ▶ $S_p(n) = T^*(n)/T_p(n)$
- ▶ $S_p(n) \leq p$
- ▶ Best possible: $S_p(n) = p$ when $T_p(n) = T^*(n)/p$

Efficiency:

- ▶ $E_p(n) = T_1(n)/(pT_p(n))$, where $T_1(n)$ is the time needed to run the parallel algorithm on a single processor
- ▶ Best possible: $E_p(n) = 1$

Parallel architectures

- ▶ Assumes a “PRAM” architecture
- ▶ *Parallel, random-access machine*
- ▶ One memory pool, all processors access in constant time

This is the theory – in practice:

- ▶ Single Instruction Stream, Multiple Data Stream (SIMD). One global control unit connected to each processor.
- ▶ Multiple Instruction Stream, Multiple Data Stream (MIMD). Each processor has a local control unit.

Parallel architectures

Shared-Address-Space:

- ▶ Each processor has access to main memory
- ▶ Processors may be given a small private memory for local variables

Message-Passing:

- ▶ Each processor is given its own block of memory
- ▶ Processors communicate by passing messages directly instead of modifying memory locations

Design of parallel algorithms

- ▶ Modify an existing sequential algorithm
 - ▶ Exploit parts of algorithm that are naturally parallelizable.
Example?
 - ▶ Sometimes embarrassingly parallel: units completely independent. Example?
- ▶ Design a completely new parallel algorithm: May have no natural sequential analog.
- ▶ Brute force!
 - ▶ Use an existing sequential algorithm, but give each processor differential initial conditions (e.g., MCMC).
 - ▶ Have a compiler optimize a sequential algorithm (e.g. loop unrolling)
 - ▶ Use an advanced CPU to optimize code (hyperthreading).

SPMD vs. MPMD

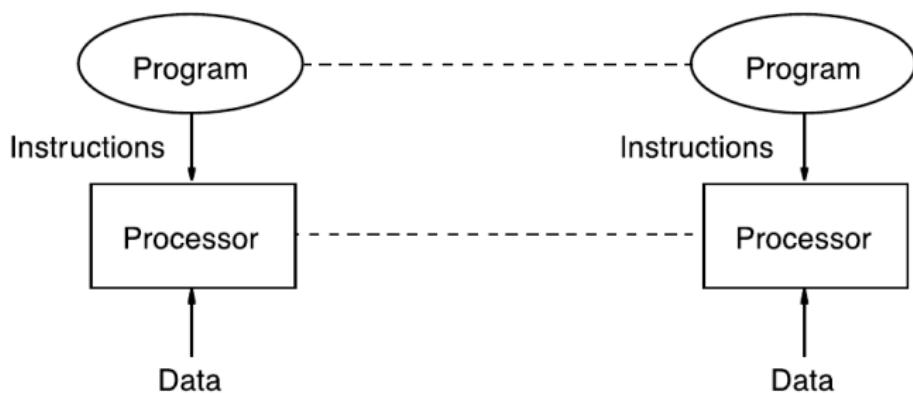
- ▶ **SPMD** (single program, multiple data): Write a single program that performs the same operation on multiple data (sub)sets (multiple chefs baking many lasagnas, rendering different frames of a movie).
- ▶ **MPMD**: Write different programs to perform different operations on multiple data (sub)sets (multiple chefs preparing a four-course dinner, rendering different parts of a movie frame).
- ▶ Can also write hybrid algorithms in which processes perform partially overlapping tasks.

Single Instruction Stream-Multiple Data Stream (SIMD) Computer

- ▶ A specially designed computer – a single instruction stream from a single program, but multiple data streams exist. Instructions from program broadcast to more than one processor.
- ▶ Each processor executes same instruction in synchronism, but using different data.
- ▶ Developed because a number of important applications that mostly operate upon arrays of data.

Multiple Program Multiple Data (MPMD) Structure

Within the MIMD classification, each processor will have its own program to execute:



Puzzle

- ▶ Solve 5,000 pieces puzzle.
- ▶ Time to complete puzzle for one person: n hours.
- ▶ How can we decrease wall time to completion?



Puzzle

- ▶ Add another person to the table: Effect on wall time?
Communication? Resource contention?
- ▶ Add p people to the table: Effect on wall time?
Communication? Resource contention?
- ▶ What about: p people, p tables, $5000/p$ pieces each?

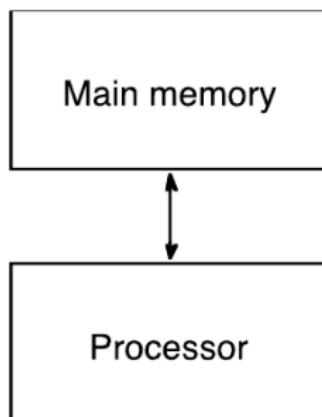
Parallel Algorithm Design: PCAM

- ▶ **Partition:** Decompose problem into fine-grained tasks to maximize potential parallelism.
- ▶ **Communication:** Fix a communication pattern among tasks.
- ▶ **Agglomeration:** (Re)combine into coarser-grained task results.
- ▶ **Mapping:** Assign tasks to processors, subject to tradeoff between communication cost and concurrency.

Conventional computer

Conventional computer: processor executing a program stored in (main) memory.

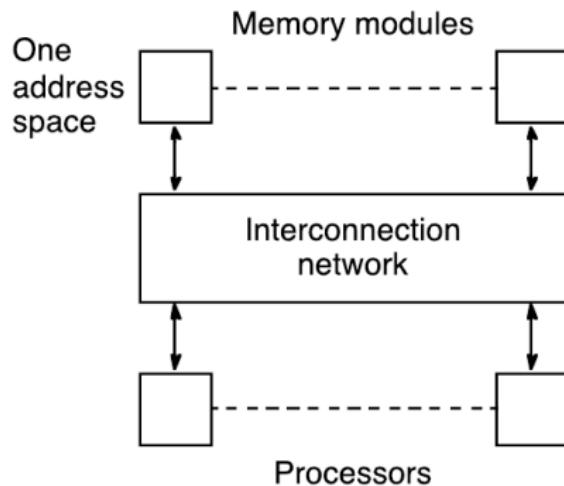
Each main memory location located by its address. Addresses start at 0 and extend to $2^b - 1$ (assuming there are b bits (binary digits) used per address).



Instructions (to processor), data (to/from processor)

Shared Memory Multiprocessor System

Natural way to extend single processor model. We have multiple processors connected to multiple memory modules, such that each processor can access any memory module:

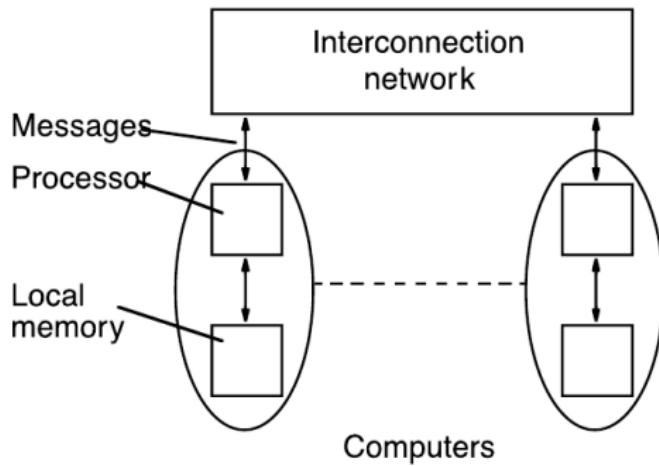


Programming Shared Memory Multiprocessors

- ▶ Threads: programmer decomposes program into individual parallel sequences, called threads, each being able to access variables declared outside threads (very common, e.g., Python threads)
- ▶ Sequential programming language with preprocessor compiler directives to declare shared variables and specify parallelism (e.g., OpenMP which is industry standard)
- ▶ Sequential programming language with added syntax to declare shared variables and specify parallelism (very common, e.g., C, Java but needs compiler/library support).
- ▶ Sequential programming language and ask parallelizing compiler to convert it into parallel executable code (SSE).

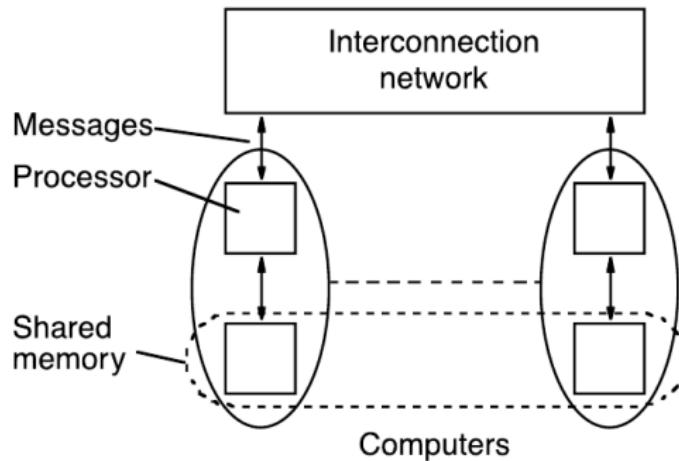
Message-Passing Multicomputer

Complete computers connected through an interconnection network:



Distributed Shared Memory

Making main memory of group of interconnected computers look as though a single memory with single address space. Then can use shared memory programming techniques.



Flynn's Classifications

Flynn (1966) taxonomy: classification for computer architectures, based upon instruction streams and data streams:

- ▶ Single instruction stream-single data stream (SISD) computer
- ▶ Single processor computer – single stream of instructions generated from program. Instructions operate upon a single stream of data items.

Multiple Instruction Stream-Multiple Data Stream (MIMD) Computer

- ▶ General-purpose multiprocessor system – each processor has a separate program and one instruction stream is generated from each program for each processor. Each instruction operates upon different data.
- ▶ Both the shared memory and the message-passing multiprocessors so far described are in the MIMD classification.

Networked Computers as a Computing Platform

- ▶ A network of computers became a very attractive alternative to expensive supercomputers and parallel computer systems for high-performance computing in the early 1990s.
- ▶ Several early projects.
- ▶ Notable: Hadoop + Map/Reduce. Cloud.

Networked Computers – Advantages

- ▶ Key advantages: Very high performance workstations and PCs readily available at low cost.
- ▶ The latest processors can easily be incorporated into the system as they become available: highly modular.
- ▶ Existing software can be used or modified: programming languages continue to better support and/or automate parallelization.

Software Tools for Clusters

- ▶ Message-Passing Interface (MPI) – standard defined in 1990s.
- ▶ Provides a set of user-level libraries for message passing. Use with regular programming languages (C etc.)
- ▶ Most modern languages have intrinsic support.
 - ▶ Python (mostly) thread pooling + multiprocessing.
 - ▶ Java synchronization.
 - ▶ Concurrency libraries everywhere: Go, Objective C, etc.

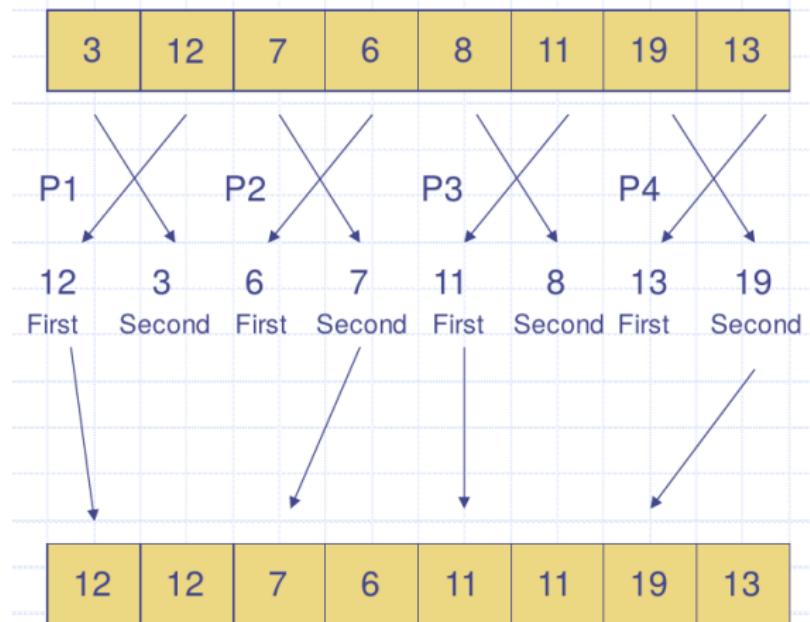
Example: Finding the Largest Key in an Array

- ▶ In order to find the largest key in an array of size n , at least $n - 1$ comparisons must be done.
- ▶ A parallel version of this algorithm will still perform the same amount of compares, but by doing them in parallel it will finish sooner.

Example: Finding the Largest Key in an Array

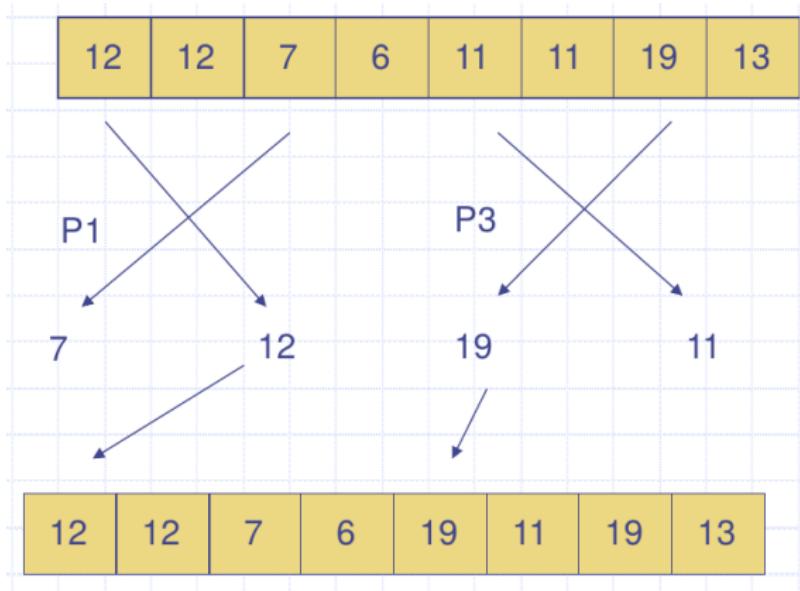
- ▶ Assume (for convenience) that n is a power of 2 and $n/2$ processors execute the algorithm in parallel.
- ▶ Each processor reads two array elements into local variables first and second.
- ▶ It then writes the larger value into the first of the array locations that it read.
- ▶ This yields $O(\log n)$ steps for the largest key to be placed in $S[0]$ (the first array location).

Example: Finding the Largest Key in an Array



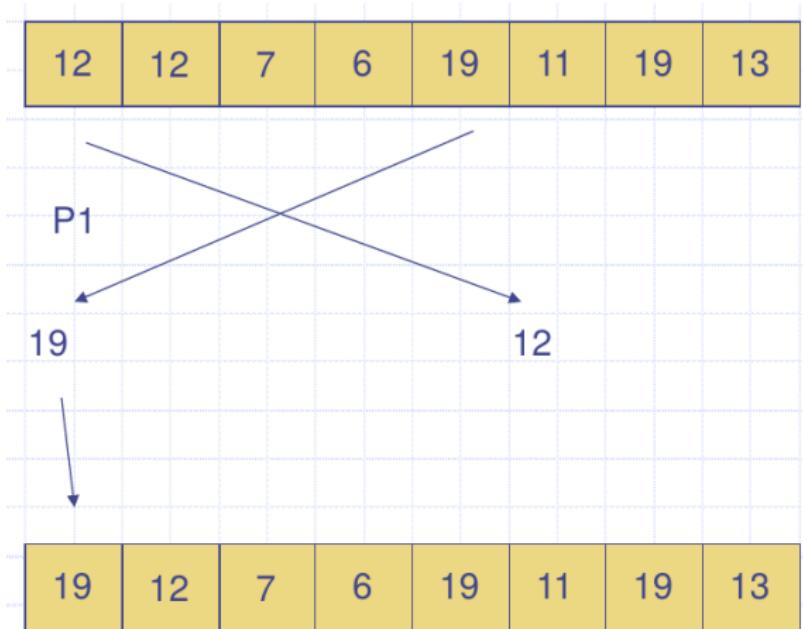
What is the difference here between a sequential and parallel implementation?

Example: Finding the Largest Key in an Array



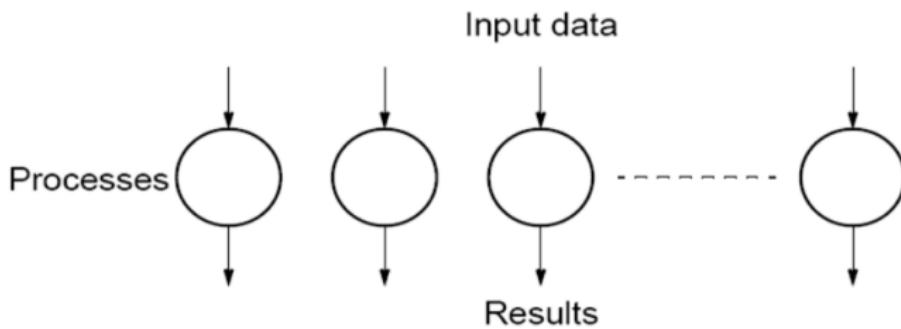
Sequential algorithm would do the same, but needs $n/2$ timesteps for it, while the parallel algorithm needs 1 timestep.

Example: Finding the Largest Key in an Array



Embarrassingly Parallel Computations

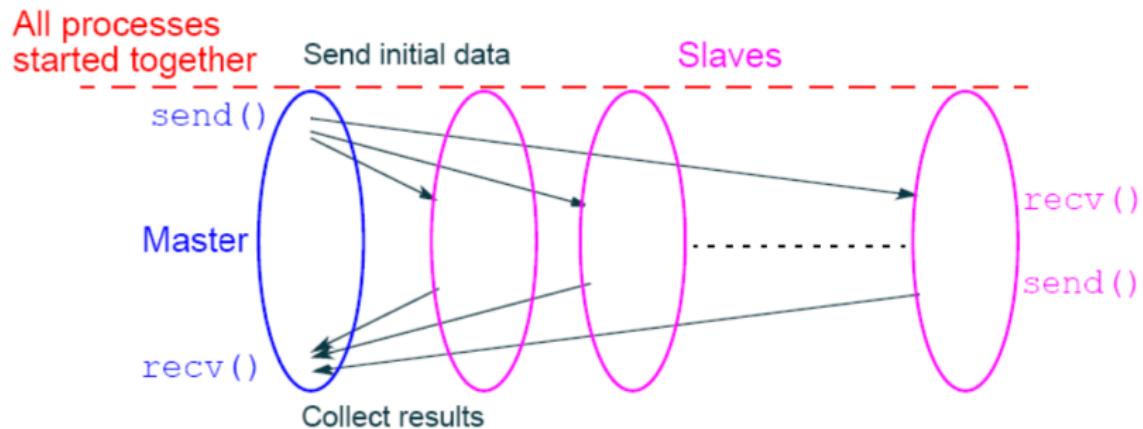
Definition: A computation that can obviously be divided into a number of completely independent parts, each of which can be executed by a separate process(or).



- ▶ No (or very little) communication between processes.
- ▶ Each process can do its tasks without any interaction with other processes.
- ▶ Examples?

Embarrassingly Parallel Computations

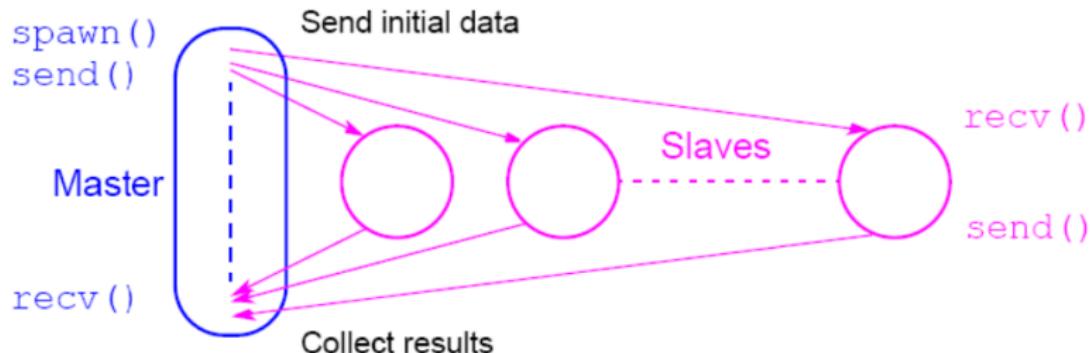
Master-slave approach with static process creation, e.g. usual MPI:



Embarrassingly Parallel Computations

Master-slave approach with dynamic process creation:

Start Master initially

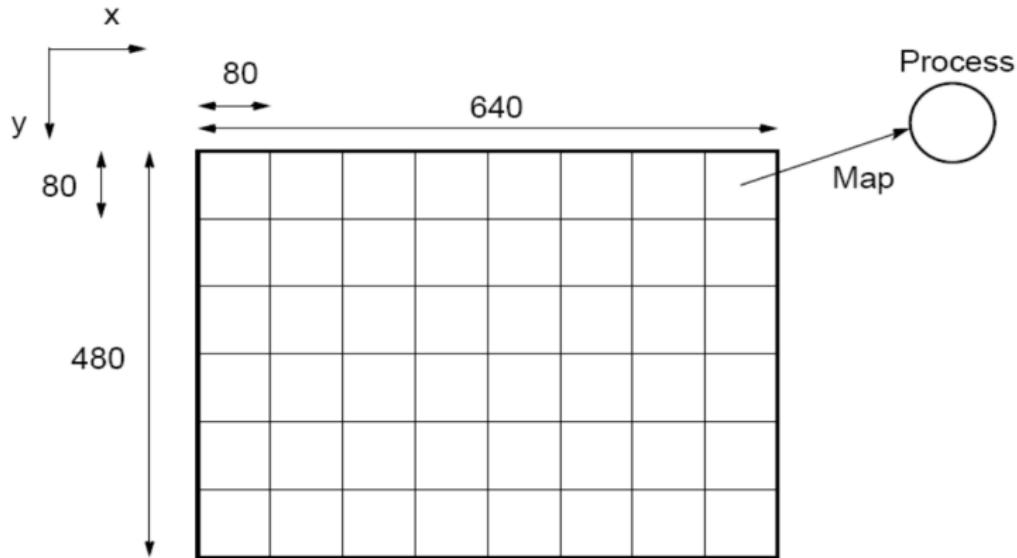


Embarrassingly Parallel Computations

Examples:

- ▶ Image processing
- ▶ Monte Carlo calculations, sampling
- ▶ Sequence search
- ▶ Divide-and-conquer: sorting, numerical integration, N-body problems in physics

Parallel image processing

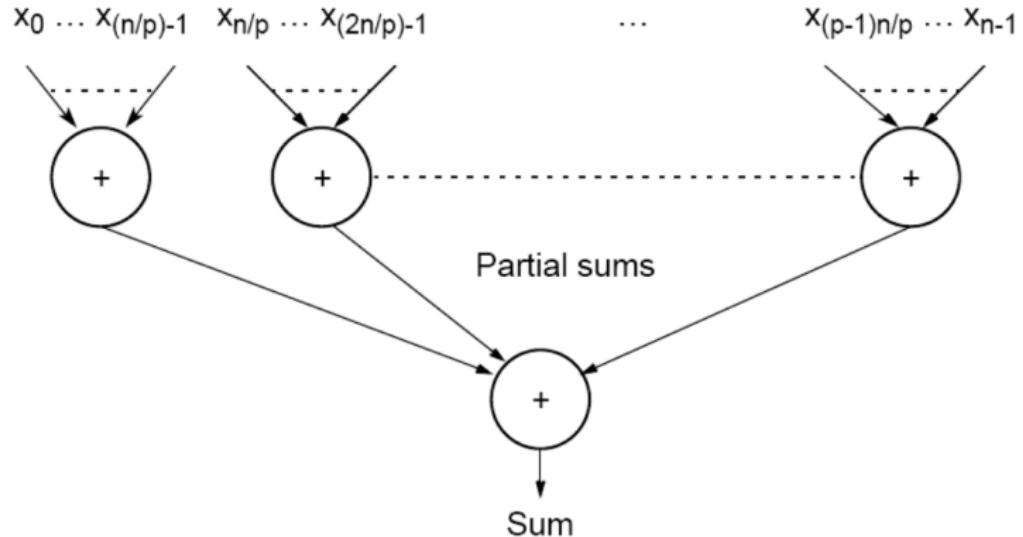


Divide-and-conquer

- ▶ Remember: Dynamic programming is characterized by overlapping subproblems (not good for parallelization)
- ▶ Divide-and-conquer: non-overlapping subproblems → suitable for parallelization

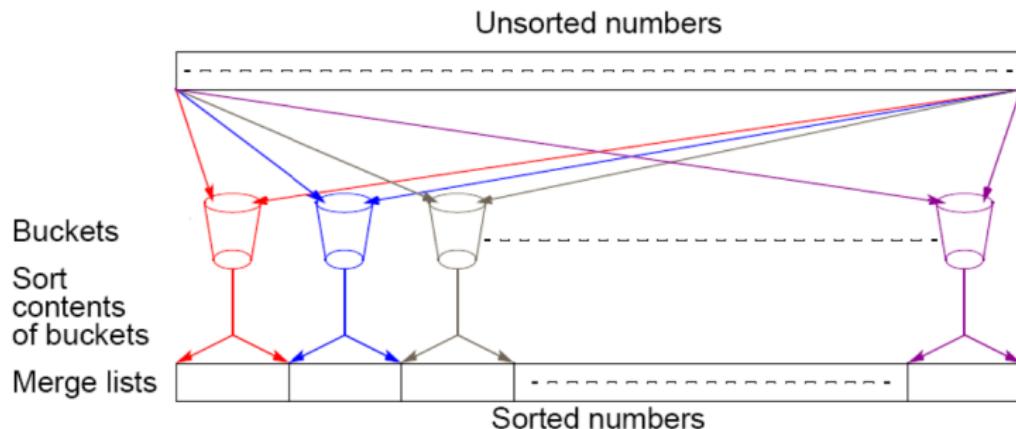
Now: computations of sums, sorting

Divide-and-conquer: sums



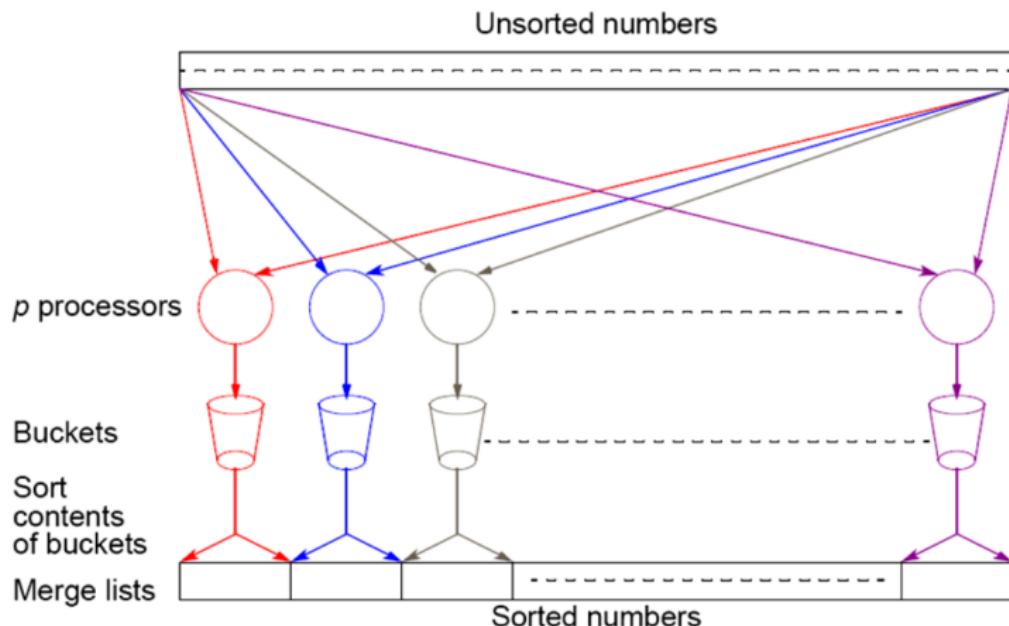
Divide-and-conquer: BucketSort (parallel MergeSort)

- ▶ Divide data into region, one region per bucket.
- ▶ Sort bucket data with any algorithm.
- ▶ Sequential time complexity for the above step:
 $O(n/m \log(n/m))$
- ▶ Merge together into fully sorted sequence in $O(n)$.
- ▶ Overall runtime: $O(n + n/m \log(n/m))$



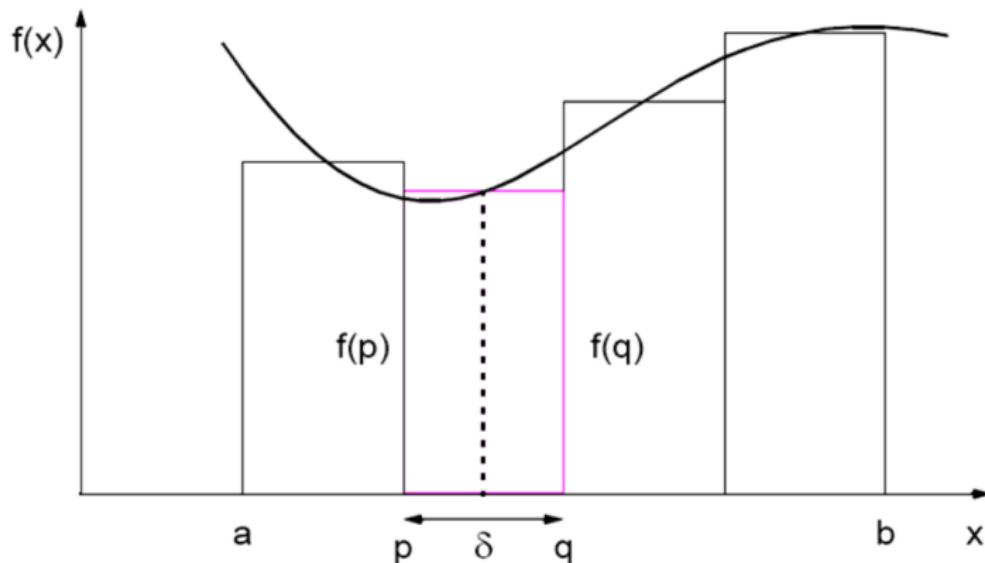
Divide-and-conquer: Parallel BucketSort

One processor per bucket:



Numerical integration using rectangles

Integral per region is approximated with the area of a rectangle.
One processor per rectangle:



Synchronization

Why is synchronization important?

- ▶ In theory, nondeterminism is good. In the real world, it's bad: we often need computations to complete with the same result each time. (Aside: Monte Carlo/ Las Vegas algorithms...)
- ▶ This means avoiding race conditions in which the outcome is determined by chance (whichever parallel process happens to finish first...).
- ▶ Synchronization provides protected operations and data that guarantee coordination among multiple parallel processes.

Example: free money at ATM

Thread 1	Thread 2
<pre> (\$10) function withdraw(\$amount) { (\$10,000) \$balance = getBalance(); if(\$amount <= \$balance) { (\$9,990) \$balance = \$balance - \$amount; echo "You have withdrawn: \$amount";</pre>	<pre> (\$10) function withdraw(\$amount) { (\$10,000) \$balance = getBalance(); if(\$amount <= \$balance) { (\$9,990) \$balance = \$balance - \$amount; echo "You have withdrawn: \$amount"; setBalance(\$balance); (\$9,990) } else { echo "Insufficient funds."; } } setBalance(\$balance); (\$9,990) } else { echo "Insufficient funds."; }</pre>

How to fix this?

Thread-Safe Routines:

- ▶ **Thread safe if they can be called from multiple threads simultaneously and always produce correct results.**
- ▶ For example, screen output is thread safe: prints messages without mashing together characters. However, it can interleave individual characters...
- ▶ System routines that return time may not be thread safe.
- ▶ Routines that access shared data may require special care to be made thread safe.

Accessing Shared Data

Accessing shared data needs careful control. Consider two processes each of which is to add one to a shared data item x . We want to read the location of x , compute $x + 1$, and then write the result back to the location:

Instruction	Process 1	Process 2
<code>x = x + 1;</code>	<code>read x</code>	<code>read x</code>
	<code>compute x + 1</code>	<code>compute x + 1</code>
	<code>write to x</code>	<code>write to x</code>

Synchronization

- ▶ **Synchronization:** A mechanism to ensure that only one process accesses a particular resource at a time is to establish sections of code involving the resource as so-called **critical sections**.
- ▶ Need to arrange that only one such critical section is executed at a time. This mechanism is known as **mutual exclusion**.
- ▶ This generally must be supported at the operating system level.

Locks

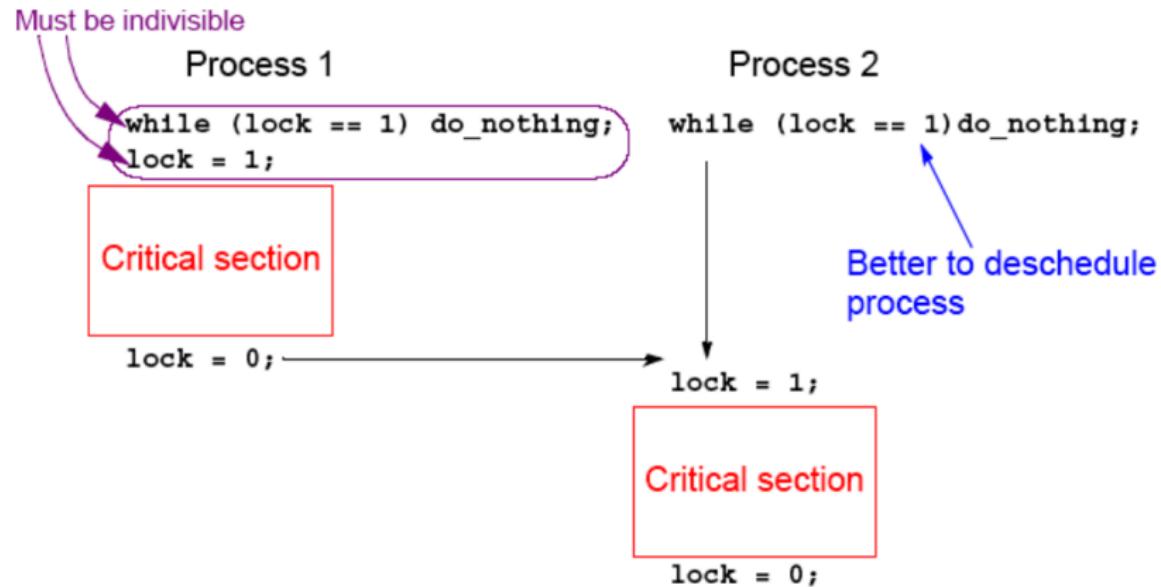
Locks:

- ▶ Simplest mechanism for ensuring mutual exclusion of critical sections.
- ▶ A lock is a 1-bit variable which indicates with a 1 that a process has entered the critical section and a 0 to indicate that no process is in the critical section.
- ▶ *Operates much like that of a door lock.*

Example: A process coming to the "door" of a critical section and finding it open may enter the critical section, locking the door behind it to prevent other processes from entering. Once the process has finished the critical section, it unlocks the door and leaves.

Locks

Setting the lock must be indivisible in the code:



Locks

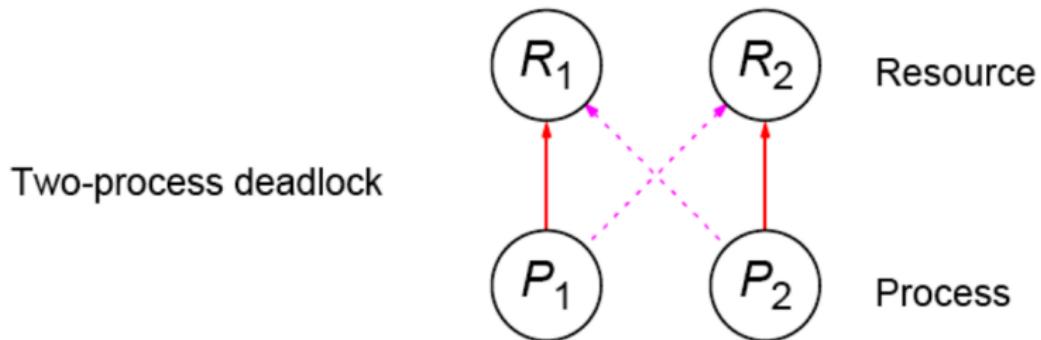
- ▶ One implementation of locks is mutually exclusive lock variables, or **mutex** variables:

```
mutex_lock(&mutex1);  
    critical section  
mutex_unlock(&mutex1);
```

- ▶ If a thread reaches a mutex lock and finds it locked, it will wait for the lock to open. If more than one thread is waiting for the lock to open when it opens, the system will select one thread to be allowed to proceed.

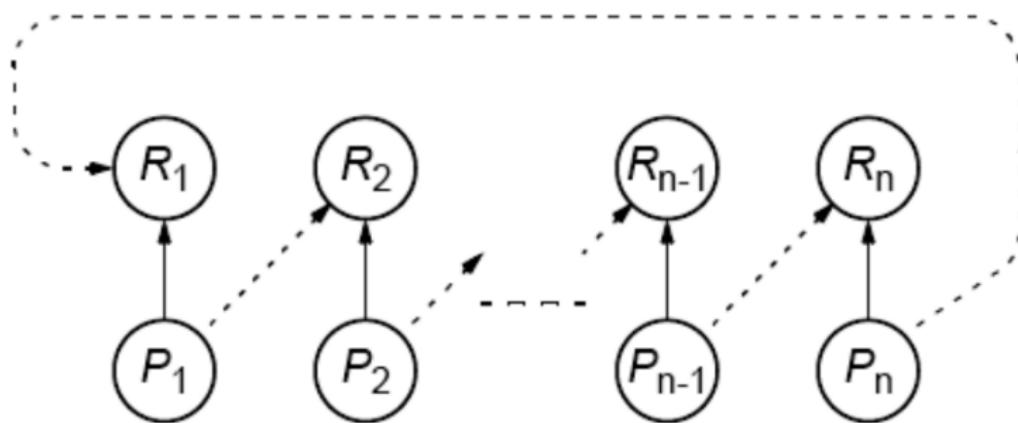
Deadlock

Can occur with two processes when one requires a resource held by the other, and this process requires a resource held by the first process:



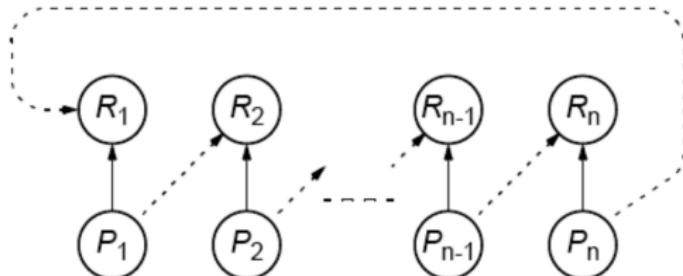
Deadlock

Deadlocks can occur in all sorts of interlocking, circular ways with several processes having a resource wanted by another:



Some implementations allow a routine to test whether a lock is actually closed without blocking the thread...

Counting semaphore



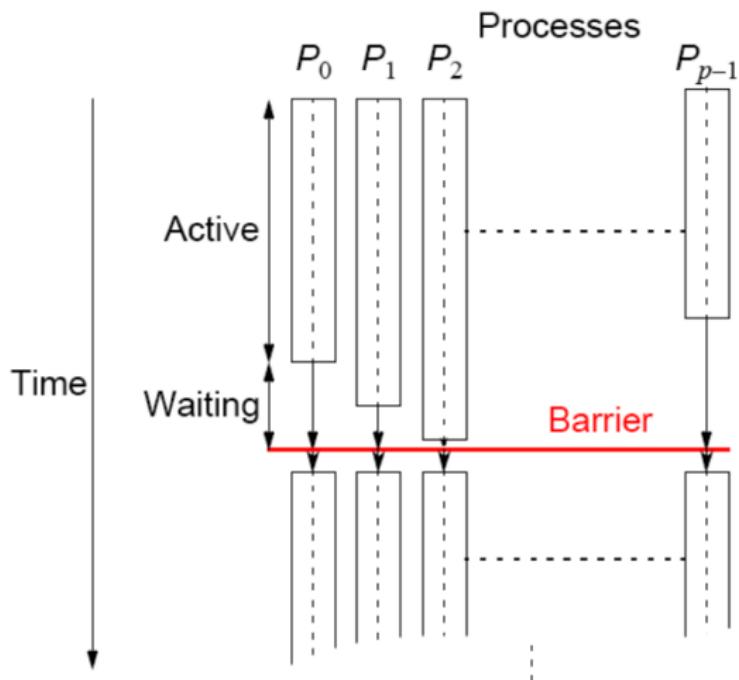
- ▶ Can take on positive values other than zero and one. It provides, for example, a means of recording the number of "resource units" available.
- ▶ Block: Wait until s is greater than zero, then decrements s by one and allows the process to continue.
- ▶ Release: Increments s by one and releases one of the waiting processes (if any). Indicates exit from a critical section.
- ▶ *PV semaphore*: Devised by Dijkstra in 1968. Letter P is from the Dutch word *passeren* ("to pass"), letter V is from *vrijgeven* ("to release").

Synchronous Computations

- ▶ In a (fully) synchronous application, all processes synchronize at regular points.
- ▶ **Barrier:** A basic mechanism for synchronizing processes – inserted at the point in each process where it must wait.
- ▶ All processes can continue from this point when all the processes have reached it – or alternatively, in some implementations, when a stated number of processes have reached this point.

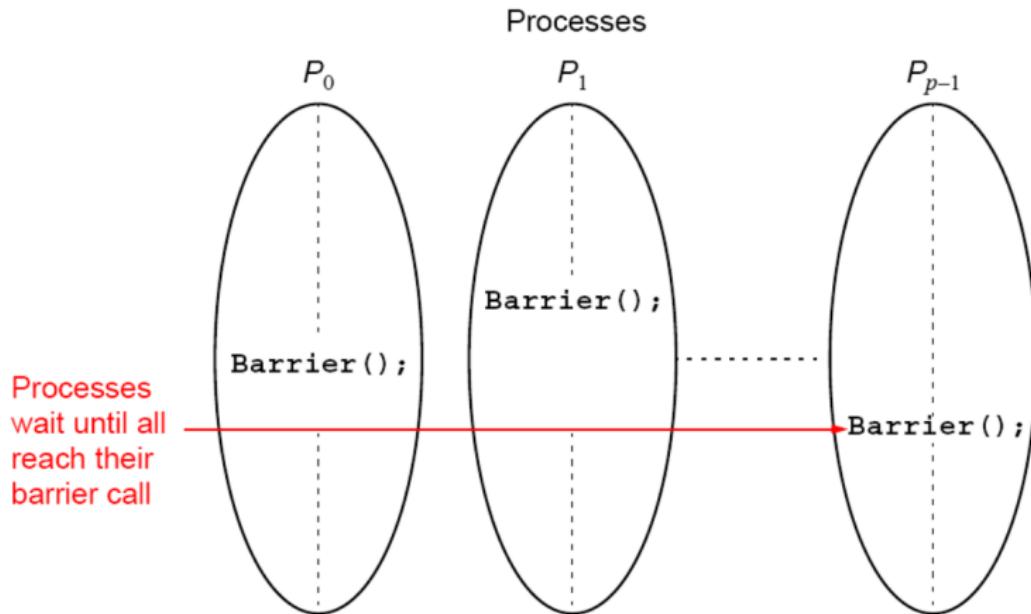
Example: barrier

Processes reaching barrier at different times:



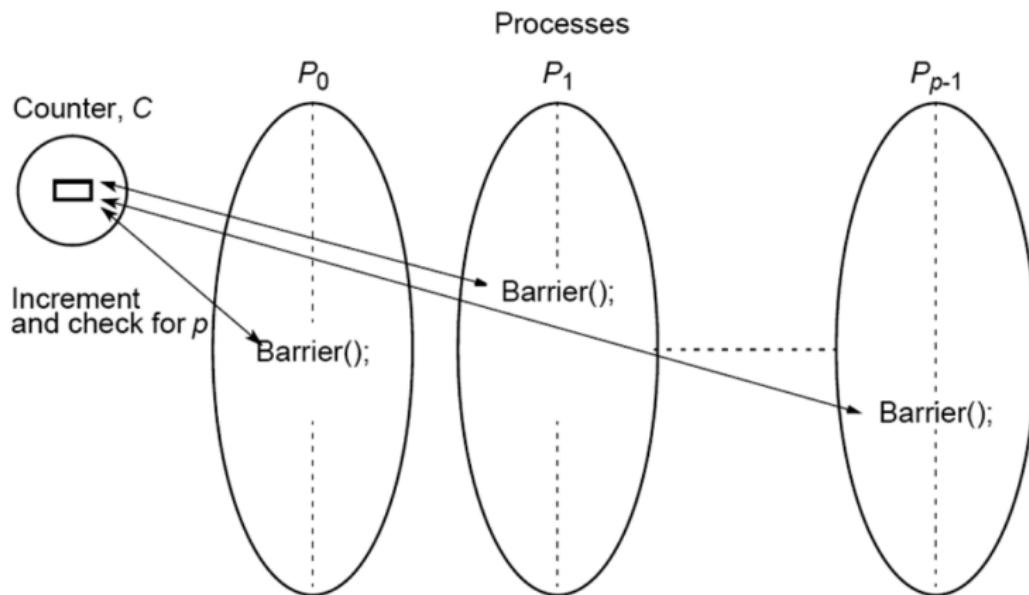
Barrier

In message-passing systems, barriers provided with library routines:



Barrier Implementation

Centralized counter implementation (a linear barrier):



Barrier might be used more than once... Processes might enter barrier a second time...

Synchronized Computations

Synchronized computations can be classified as:

- ▶ **Fully synchronous**
- ▶ **Locally synchronous**

In fully synchronous computations, all processes involved in the computation must be synchronized. In locally synchronous computations, processes only need to synchronize with a set of logically nearby processes (instead of all processes involved in the computation).

Example: fully synchronous

To add the same constant to each element of an array:

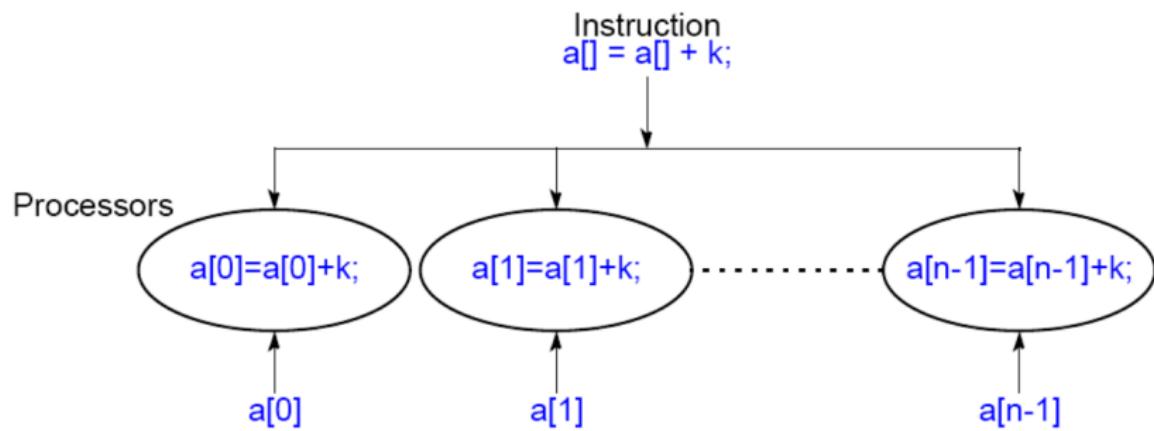
```
for (i = 0; i < n; i++) {  
    a[i] = a[i] + k;  
}
```

The statement

```
a[i] = a[i] + k;
```

could be executed simultaneously by multiple processors, each using a different index $i \in \{1, \dots, n\}$.

Example: fully synchronous/ data parallel



Locally Synchronous Computation

Heat Distribution Problem: An area has known temperatures along each of its edges. Find the temperature distribution within.

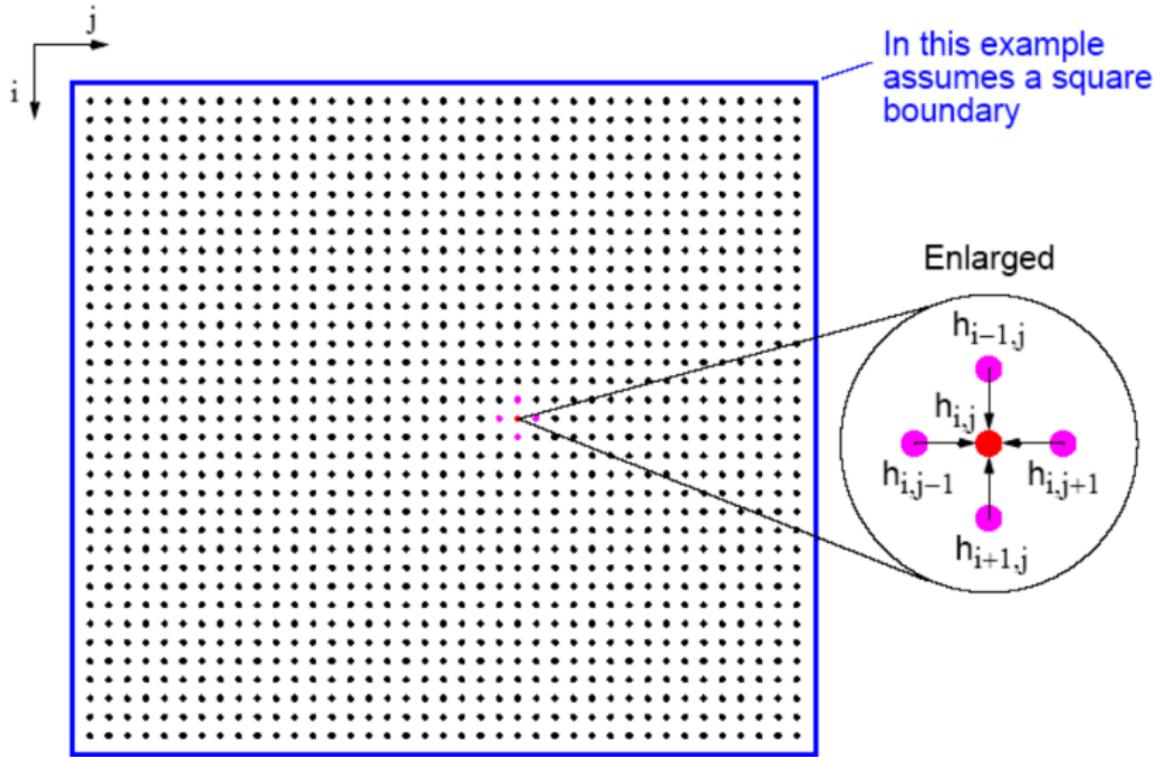
Approach:

- ▶ Divide area into fine mesh of points, $h_{i,j}$.
- ▶ Temperature at an inside point is taken to be the average of temperatures of four neighboring points. Convenient to describe edges by points.
- ▶ Obtain temperature of each point by iteratively computing averages:

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

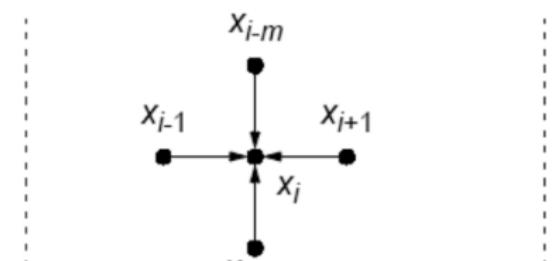
for $0 < i < n$, $0 < j < n$ for a fixed number of iterations or until convergence.

Heat Distribution Problem



Heat Distribution Problem

Natural ordering:



Heat Distribution Problem

Result will look something like this:



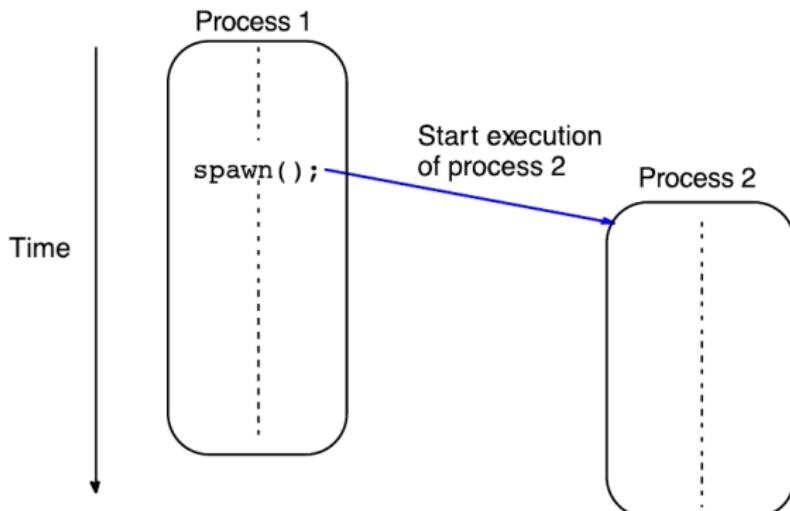
Message Passing

Two primary mechanisms needed in parallelization:

1. A method of creating separate processes for execution on different processors or computers.
2. A method of sending and receiving messages.

Multiple Program Multiple Data (MPMD) Model

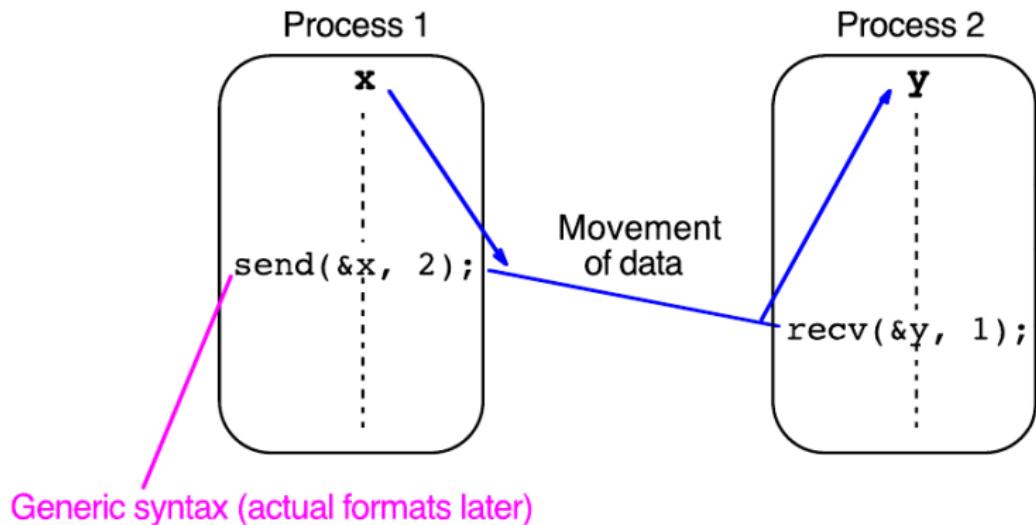
Separate programs for each processor. One processor executes master process. Other processes started from within master process: *dynamic process creation*.



Example: GIMPS

Basic point-to-point Send and Receive Routines

Passing a message between processes using *send()* and *recv()* library calls:



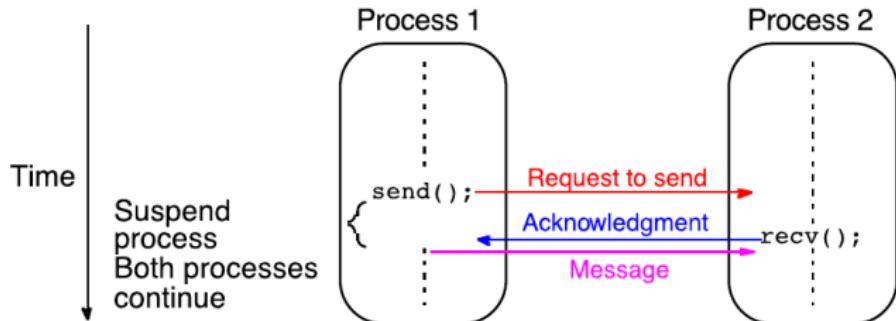
Has the message arrived on time? Was the receiver listening?

Synchronous Message Passing

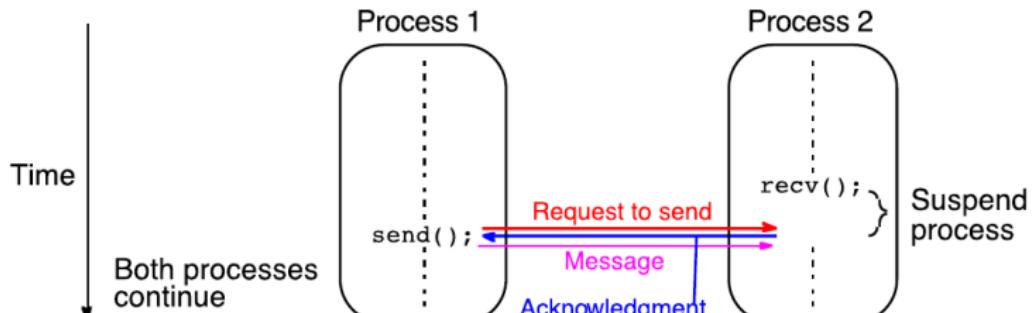
Routines that actually return when message transfer completed.

- ▶ **Synchronous send routine:** Waits until complete message can be accepted by the receiving process before sending the message.
- ▶ **Synchronous receive routine:** Waits until the message it is expecting arrives.
- ▶ Synchronous routines intrinsically perform two actions. They transfer data and they synchronize processes.

Synchronous Message Passing



(a) When `send()` occurs before `recv()`



(b) When `recv()` occurs before `send()`

Asynchronous Message Passing

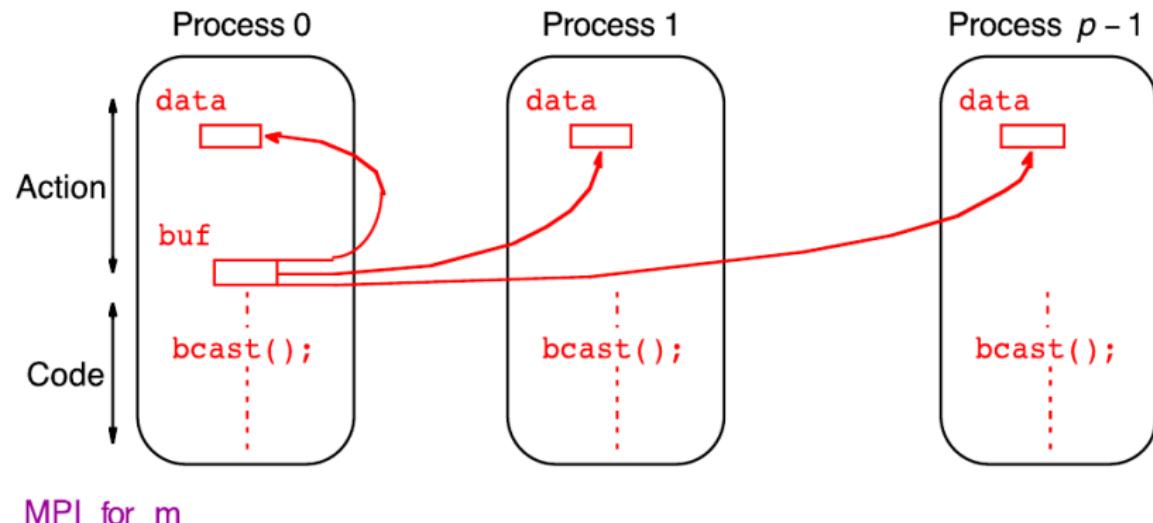
- ▶ Routines that do not wait for actions to complete before returning. They usually require local storage for messages.
(As well as additional mechanisms for enforcing synchronicity.)
- ▶ More than one version depending upon the actual semantics for returning.
- ▶ *In general, they do not synchronize processes but allow processes to move forward sooner. Must be used with care.*

To follow: basic operations for message sending and gathering...

Asynchronous Message Passing: Broadcast

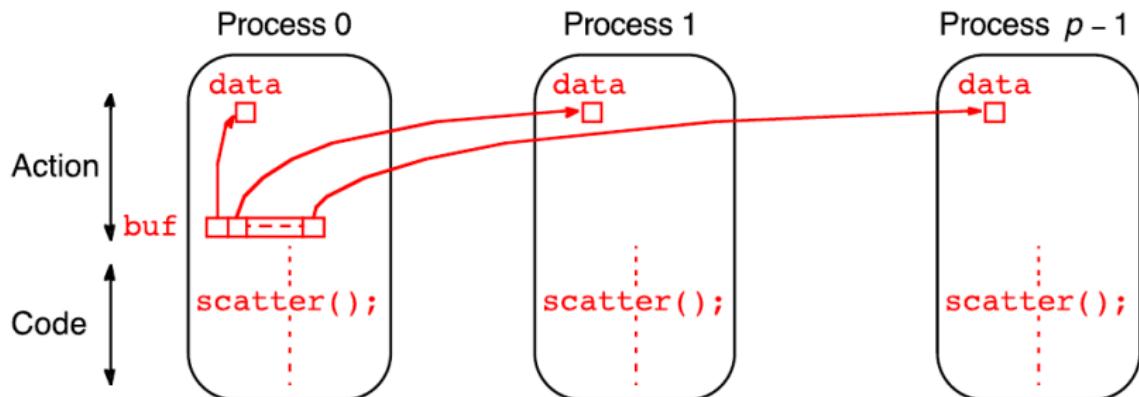
Send same message to all processes concerned with a problem.

Multicast: send same message to defined group of processes.



Scatter (Map)

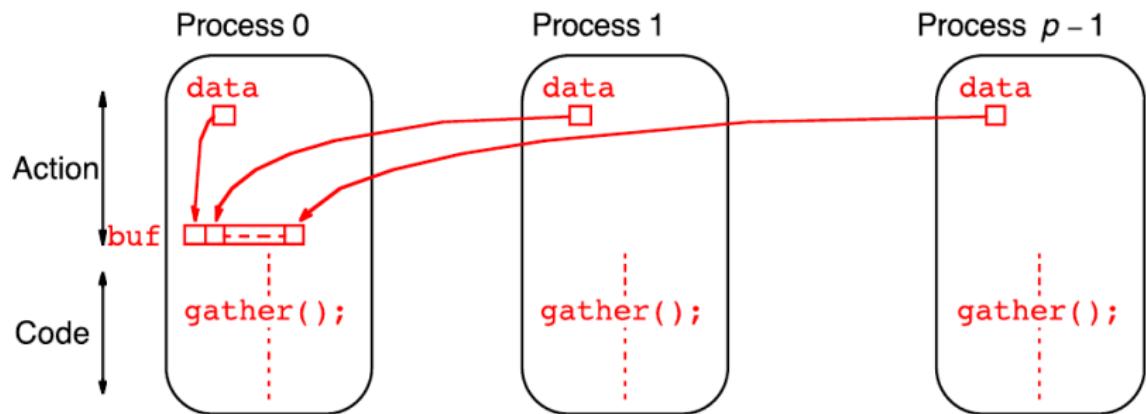
Send each element of an array in root process to a separate process. Contents of i th location of array sent to i th process.



MPI for m

Gather

Having one process collect individual values from set of processes.

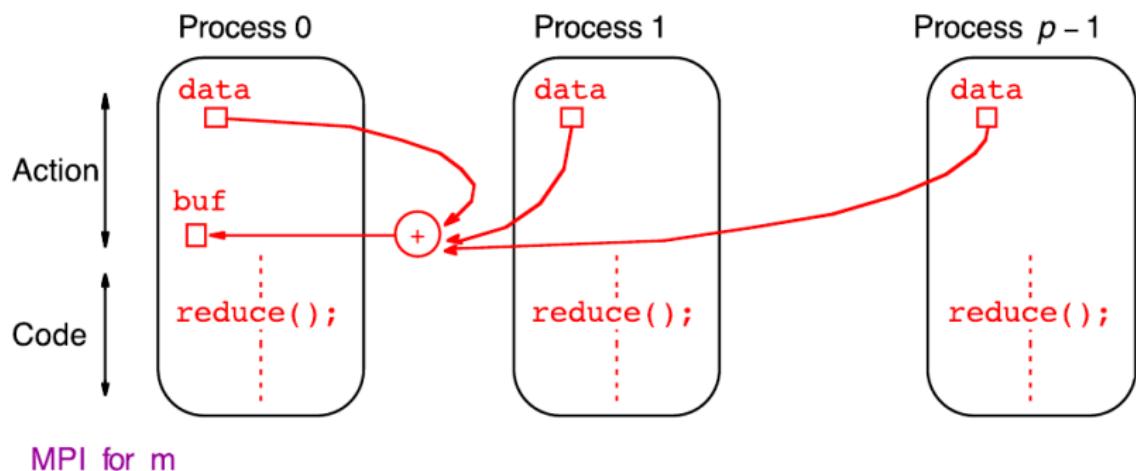


MPI for m

Reduce

Gather operation combined with specified arithmetic/logical operation.

Example: Values could be gathered and then added together by root/master:



Message Passing Interface (MPI)

- ▶ Message passing library standard developed by group of academics and industrial partners to foster more widespread use and portability.
- ▶ Defines routines, not implementation.
- ▶ Several free implementations exist.

Process Creation and Execution:

- ▶ Purposely not defined – Will depend upon implementation.
- ▶ Only static process creation supported in MPI v1. All processes must be defined prior to execution and started together.
- ▶ Originally SPMD model of computation. MPMD also possible with static creation – each program to be started together specified.

Communicators

- ▶ Defines scope of a communication operation.
- ▶ Processes have IDs associated with communicator.
- ▶ Initially, all processes enrolled in a "universe" called *MPI_COMM_WORLD*, and each of the p processes is given a unique rank ID, a number from 0 to $p - 1$.
- ▶ Other communicators can be established for groups of processes.

MPI: Using SPMD Computational Model

```
main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    .
    .
    .
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*find process rank */

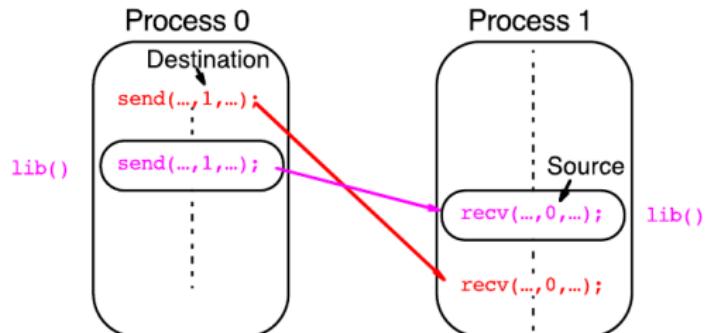
    if (myrank == 0)
        master();
    else
        slave();
    .
    .
    .
    MPI_Finalize();
}
```

where *master()* and *slave()* contain the code to be executed by master process and slave process, respectively.

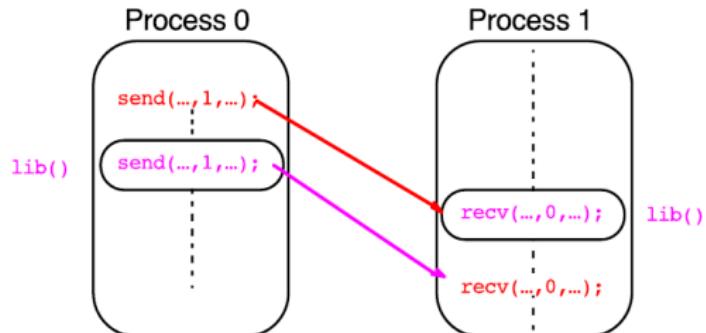
Unsafe message passing

MPI contains a framework to guard against the following:

(a) Intended behavior



(b) Possible behavior



MPI: communication domains

Realized with the help of "communicators":

- ▶ Defines a communication domain – a set of processes that are allowed to communicate between themselves.
- ▶ Communication domains of libraries can be separated from that of a user program.
- ▶ Used in all point-to-point and collective MPI message-passing communications.

Contents:

- ▶ Complexity classes
- ▶ Definition of NP-hard and NP-complete problems, and examples thereof
- ▶ SAT and other NP problems
- ▶ Reductions
- ▶ Quadratic unconstrained binary optimization
- ▶ Quantum Annealing

Complexity classes

- ▶ Big-O notation abstracts runtime and facilitates comparison.
- ▶ Complexity classes abstract it even more.

P and NP:

problems solvable in polynomial time

- ▶ P: class of all polynomial time algorithms, e.g. $O(n^k)$ for any fixed k (including $k = 0$).
- ▶ NP: class of nondeterministic polynomial time algorithms.

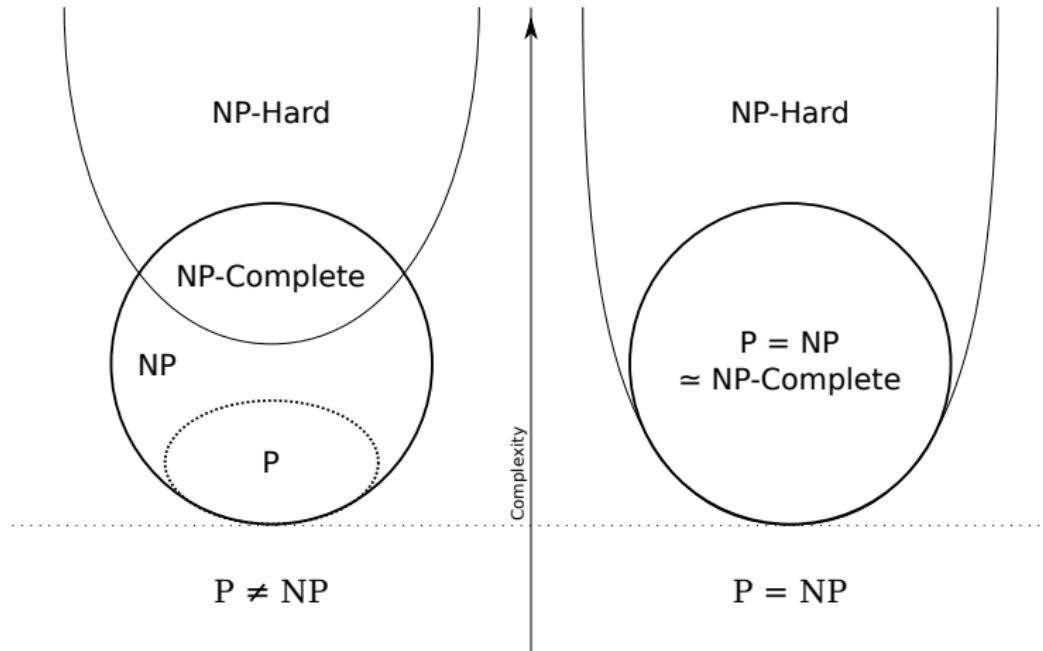
Solutions of those can be verified in polynomial time.

Equivalently, those problems have a polynomial time given an infinitely parallel computer. like OG knapsack

→ given a solution, verifies it's a valid solution (not necessarily best)

Clearly $P \subseteq NP$, but...

Complexity classes



NP-Completeness

Three classes of problems:

- ▶ **P**: problems solvable in polynomial time.
- ▶ **NP**: problems verifiable in poly time.
- ▶ **NPC**: problems in NP and "as hard as" any problem in NP.
Among NPC, we distinguish:
 1. **NP-Complete**: decision problem (Is $s \in L$?).
 2. **NP-Hard**: function problem (How many $x \in L$).

Polynomial problems

P: Poly time algorithm

- ▶ input size n (in some encoding)
- ▶ worst case running time $O(n^k)$ for some constant k

Examples:

- ▶ Pretty much all algorithms which are actually applied to real life problems...
- ▶ Shortest path (google maps), scheduling, sorting, spanning tree, max flow computations
- ▶ Heuristics for NP problems

NP-Completeness (NPC)

- ▶ A problem p is NP-Complete if $p \in \text{NP}$, and any other problem $p' \in \text{NP}$ can be reduced to (=translated into) p in polynomial time.
- ▶ So if p can be solved in polynomial time, then all problems in NP can be solved in polynomial time.

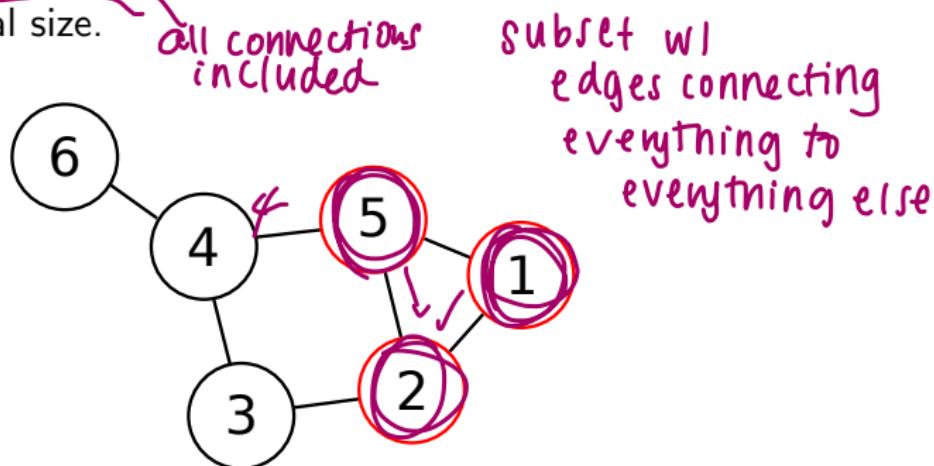
Examples for NP-complete problems:

- ▶ Circuit Satisfiability, Traveling Salesman, Maximum Clique...
- ▶ All current known NP-hard problems have been proved to be NPC.

Let's look at a few NP-complete problems...

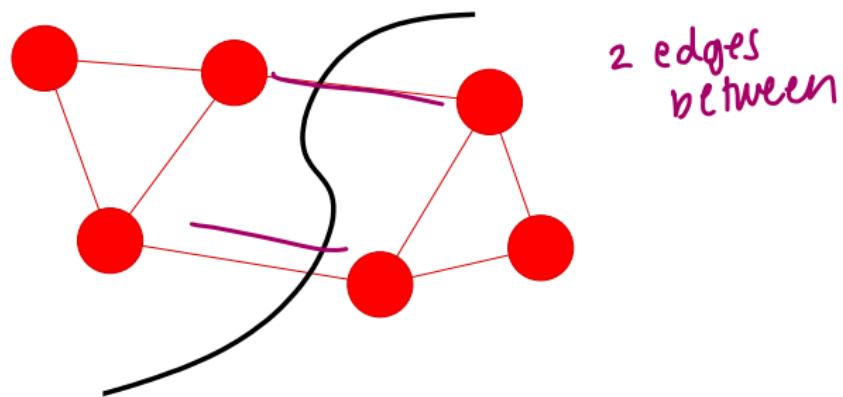
Maximum Clique

Let $G = (V, E)$ be a graph. A subset $C \subseteq V$ of vertices is a clique if $(v, w) \in E$ for all $v, w \in C$, $v \neq w$. A maximum clique is a clique of maximal size.



Graph partitioning

Let $G = (V, E)$ be a graph. Find $V_1, V_2 \subseteq V$ such that $V_1 \cap V_2 = \emptyset$, and $V_1 \cup V_2 = V$, and the number of edges between V_1 and V_2 is minimal.



Number partitioning

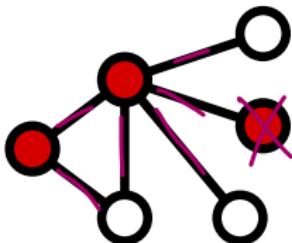
Number partitioning asks the following: Given a set of N positive numbers $S = \{n_1, \dots, n_N\}$, is there a partition of this set of numbers into two disjoint subsets R and $S \setminus R$, such that the sum of the elements in both sets is the same?

Find $R \subseteq \{1, \dots, N\}$ such that

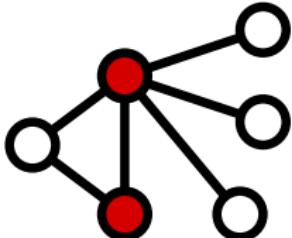
$$\sum_{i \in R} n_i = \sum_{i \notin R} n_i.$$

Minimum vertex cover

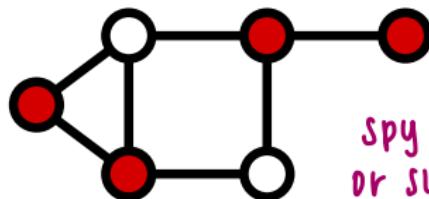
Let $G = (V, E)$ be a graph. A vertex cover $V' \subseteq V$ is a set of vertices such that for any $(v, w) \in E$ it follows that $v \in V'$ or $w \in V'$. A minimum vertex cover is a vertex cover of minimal size.



vertex cover



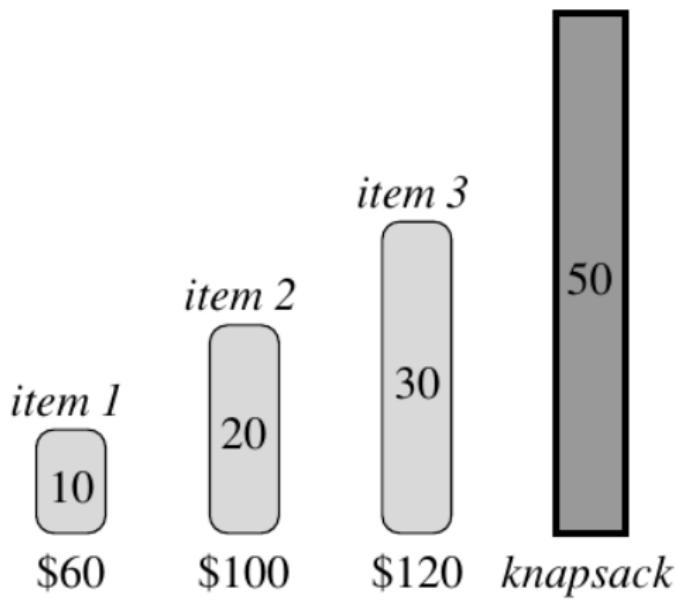
minimum vertex cover



spy software
or surveillance,
look @ traffic
at vertex cover
sites.
to get
all communicatio
ns

Knapsack

Given a capacity W and a set of n items $S = \{s_1, \dots, s_n\}$. Each $s \in S$ has a weight w and a value v . Find a set $T \subseteq \{1, \dots, n\}$ such that $\sum_{i \in T} w_i \leq W$ and the total value $\sum_{i \in T} v_i$ is maximal.

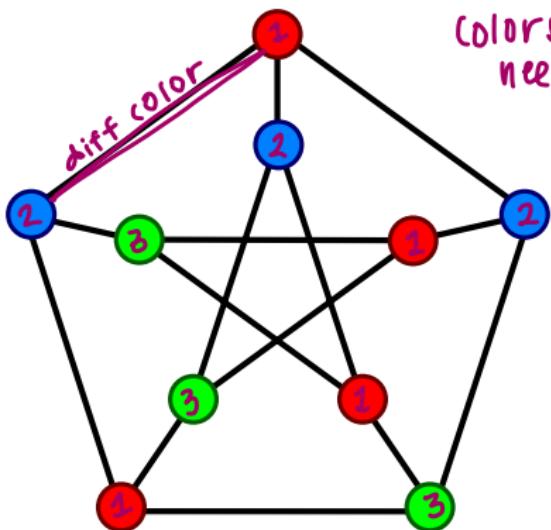


Graph coloring

Let $G = (V, E)$ be a graph. Assign one of n colors to each $v \in V$ such that the colors of $v, w \in V$ with $(v, w) \in E$ are not the same.

how many (min)

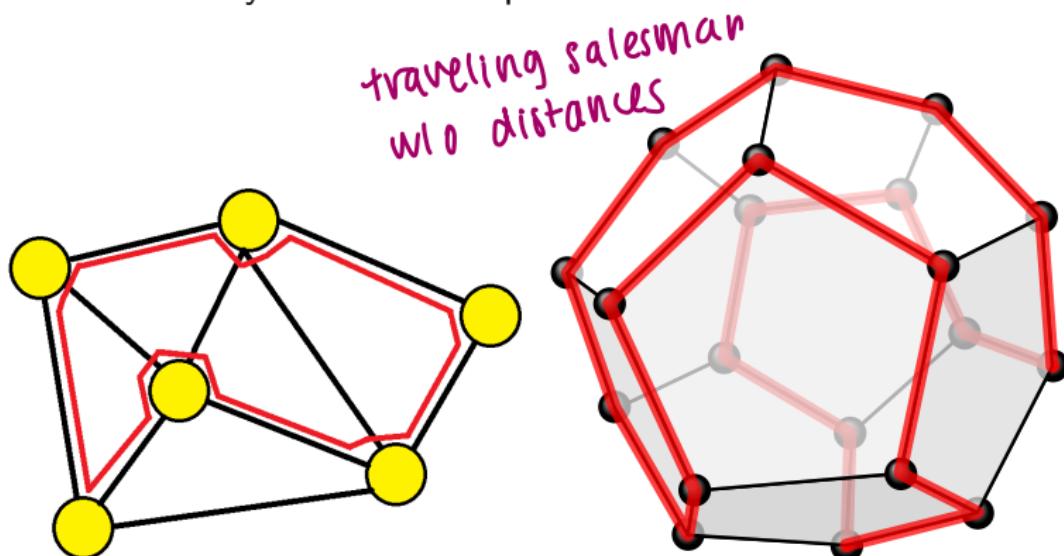
colors do we
need? \rightarrow chromatic
number



greedy: check
colors of neighbors
& assign a
different one

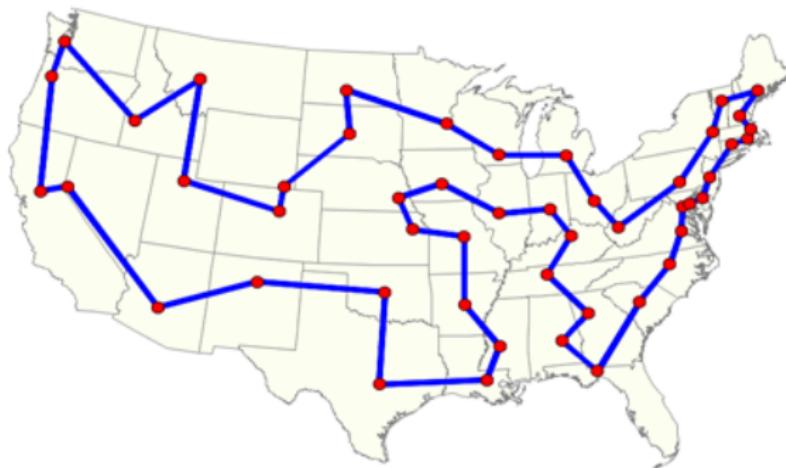
Hamiltonian paths and cycles

Let $G = (V, E)$ be a directed or undirected graph. A Hamiltonian path is a path through all vertices which visits each vertex exactly once. A Hamiltonian cycle is a Hamiltonian path which is a cycle, that is the first and last vertex are the same. Computing a Hamiltonian cycle is NP-complete.



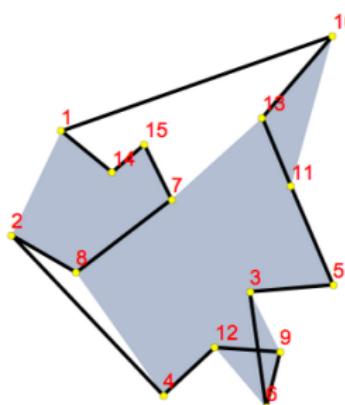
Travelling Salesman problem

Let $G = (V, E)$ be a weighted graph. TSP asks to find a cycle through all $v \in V$ such that all vertices are visited exactly once (apart from the last one), and the total weight of the route is minimal.

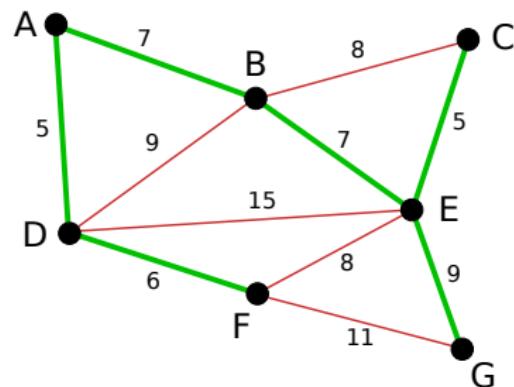


Heuristics for TSP

The nearest neighbor heuristic starts at an arbitrary node and goes to the nearest one which has not been visited yet. The MST heuristic builds a spanning tree and uses each edge twice.



nearest neighbor



MST

SAT: Boolean satisfiability problem

- ▶ SAT: Given a formula in boolean logic, e.g.

$$(a \vee b) \wedge (a \vee c) \wedge (b \vee c)$$

determine if there is an assignment of values to the variables that makes the formula true.

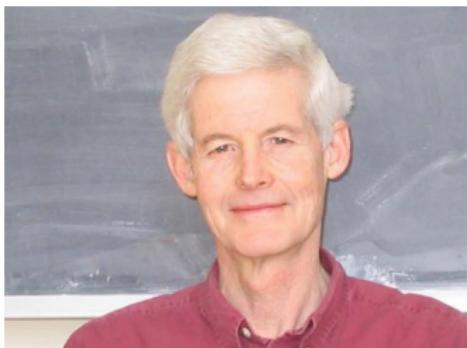
- ▶ Usually, the problem asks to satisfy many such clauses with n literals each (n -SAT), e.g.

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_7 \vee x_8) \wedge \dots$$

Why is it in NP?

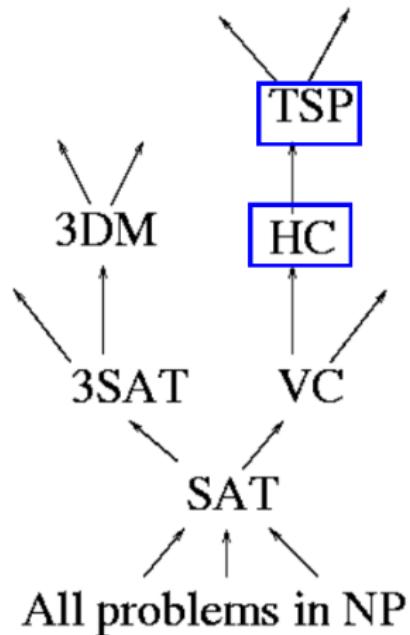
SAT is NP-Complete

- ▶ Steven Cook (1971) showed that SAT could be used to simulate any non-deterministic Turing machine!
- ▶ Idea: consider the tree of possible execution states of the Turing machine:
 1. A Boolean logic formula can represent this tree: the "state transition" function.
 2. Formula also asserts the final state is one where a solution has been found.
 3. "Guessed" variables determine which branch to take.



Graph of NP-Completeness

- ▶ Cook also first showed that satisfiability of Boolean formulas (SAT) is NP-complete.
- ▶ Hundreds of other problems (from scheduling and databases to optimization theory) have since been shown to be NPC.
- ▶ How? By deriving an algorithm that converts a known NPC problem to the problem under consideration in polynomial time. Then, your problem is also NPC!



How to solve NP problems?

Exact solution is often infeasible, but...

- ▶ **Approximation:** Instead of searching for an optimal solution, search for a solution that is at most a factor from an optimal one (existence of such algorithms depends on the problem).
e.g. greedy coloring
- ▶ **Randomization:** Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability.
- ▶ **Restriction:** By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible (sometimes also polynomial ones).
e.g. fractional knapsack
- ▶ **Parameterization:** Often there are fast algorithms if certain parameters of the input are fixed.
- ▶ **Heuristic:** An algorithm that works "reasonably well" in many cases, but for which there is no proof that it is both always fast and always produces a good result.

Reductions

- ▶ A polynomial-time reduction is a method for solving one problem using another.
- ▶ A polynomial-time reduction proves that the first problem is no more difficult than the second one, because whenever an efficient (i.e., polynomial) algorithm exists for the second problem, one exists for the first problem as well.
- ▶ If problem A can be reduced to B then we know that *A is not harder than B*, denoted

$$A \leq_P B,$$

where the notation \leq_P shows the polynomial reduction.

Reductions

Definition: Problem X reduces to problem Y if you can use an algorithm that solves Y to help solve X .

Implication:

$$\text{cost of solving } X = M^*(\text{cost of solving } Y) + \text{cost of reduction}$$

Applications:

- ▶ Designing algorithms: given algorithm for Y , can also solve X .
- ▶ Proving hardness: if X is hard, then so is Y .
- ▶ Classifying problems: establish relative difficulty among problems.

Polynomial reduction

Definition: Problem X polynomially reduces to problem Y if any arbitrary instance of problem X can be solved using:

1. Polynomial number of standard computational steps for reduction.
2. One call to subroutine for Y .

Notation: $X \leq_P Y$

Example: All algorithms for which we know polynomial time algorithms poly-time reduce to one another. Poly-time reduction of X to Y makes sense only when X or Y is not known to have a polynomial time algorithm.

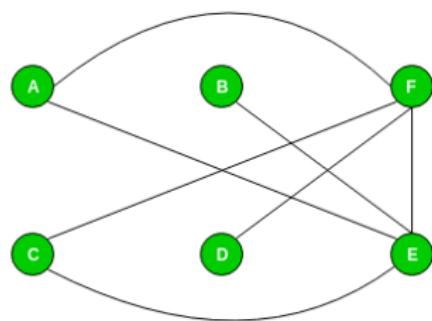
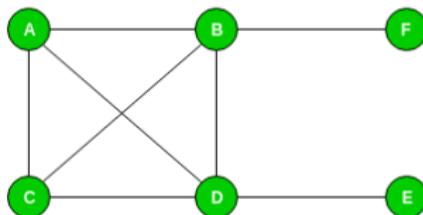
Tractability and intractability

Tractable problems can be solved in polynomial time. Intractable problems cannot.

1. Establish **tractability**: If $X \leq_P Y$ and Y is tractable then so is X . To be precise, solve Y in polynomial time and then use reduction to solve X .
2. Establish **intractability**: If $Y \leq_P X$ and Y is intractable, then so is X . To be precise, assuming X can be solved in polynomial time, then so could Y through the reduction. Contradiction. Therefore X is intractable.
3. **Transitivity**: If $X \leq_P Y$ and $Y \leq_P Z$ then $X \leq_P Z$.

Example: Reduction

Observation: Let $G = (V, E)$ be any graph. C is a clique in G if and only if $V \setminus C$ is a vertex cover in the complement of G .



Left: $\{A, B, C, D\}$ form the maximum clique. Right: $\{F, E\}$ are the vertex cover.

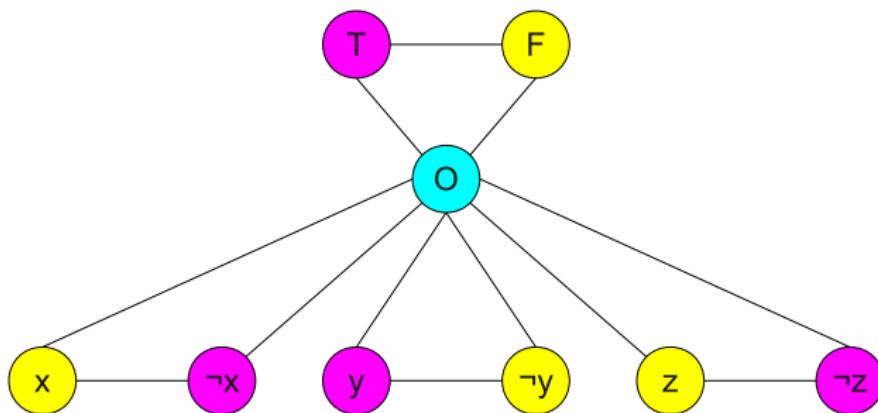
Question: How can we use this to reduce MAX CLIQUE to VERTEX COVER?

Example: Reduction

Claim: 3-SAT \leq_P 3-COLOR

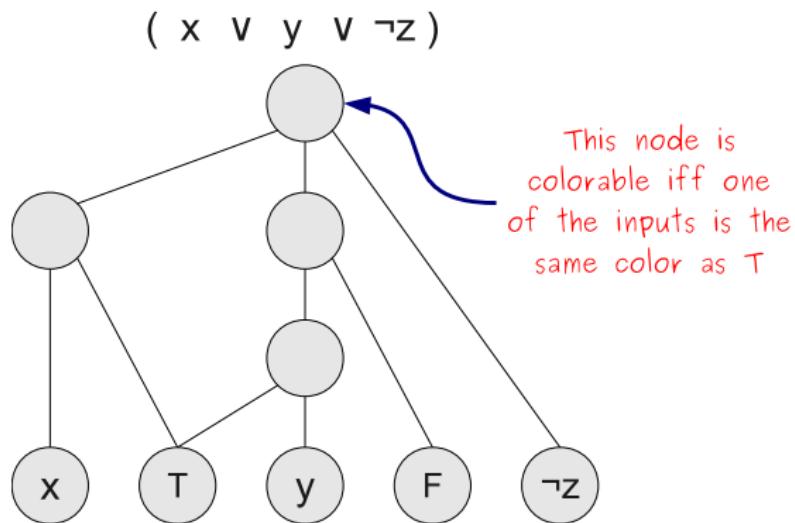
Idea: Construct graph of 3-cliques such that (a) the graph is only colorable with 3 colors if the 3-SAT clause is satisfiable, and (b) the size of the graph is polynomial in the size of the 3-SAT problem.

$$(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$$



Example: Reduction

Claim: 3-SAT \leq_P 3-COLOR



Quadratic unconstrained binary optimization

- The D-Wave quantum annealer is designed to minimize QUBO (quadratic unconstrained binary optimization) and Ising problems

$$H(x_1, \dots, x_n) = \sum_{i=1}^n a_i x_i + \sum_{i < j} a_{ij} x_i x_j, \quad (1)$$

via a process called *quantum annealing*.

- Linear weights $a_i \in \mathbb{R}$ and quadratic couplers $a_{ij} \in \mathbb{R}$ define the problem.
- QUBO: $x_i \in \{0, 1\}$; Ising: $x_i \in \{-1, +1\}$.
- Many NP-hard problems expressible as minimizations of (1), see Lucas (2014).

Customized computers
just to solve these
problems



Quadratic unconstrained binary optimization

Find $x_i \in \{0, 1\}$, $i \in \{1, \dots, n\}$, which minimizes

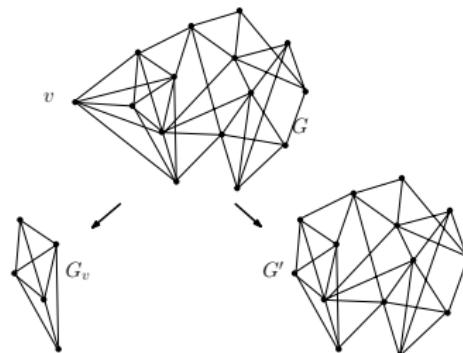
$$H(x_1, \dots, x_n) = \sum_{i=1}^n a_i x_i + \sum_{i < j} a_{ij} x_i x_j.$$

End of story? It is just the beginning...

- ▶ Maximum Cut
- ▶ Maximum Clique
- ▶ Vertex Cover
- ▶ Graph Coloring
- ▶ Travelling Salesman
- ▶ etc.

Quadratic unconstrained binary optimization

- Decomposition:



- Lower bound: $H(x_1, \dots, x_n) \geq L$ where L is computable in polynomial time.
- Compute partial solution: "I don't know the optimal solution, but I know that $x_2 = 1$ in any global optimum" (again computable in polynomial time):

[0 1 x x x 0 x x ...]

- Further techniques...

Numerical Aspects of Algorithms

Contents:

1. Binary representation of floating point numbers in a computer
2. Precision and accuracy
3. Traditional and fast multiplication via divide-and-conquer paradigm
4. Vector norms
5. Relative and absolute numerical errors
6. Well-posed and ill-posed numerical problems
7. Condition number and well/ill-conditioned problems
8. Sources of numerical errors
9. Stability of linear equation systems

Good properties of algorithms and sources of errors

Desired properties:

1. The accuracy of the results calculated by the algorithm; and
2. the numerical stability of the algorithm

are important aspects of quantitative analysis.

Type of errors:

1. errors in the model
2. errors in the input data
3. round-off errors
4. approximation errors/numerical method
5. errors due to arithmetic, e.g. caused by + or * operations

Floating point numbers

A normalized floating point number is a number $x \in \mathbb{R}$ that can be written as

$$x = \pm m \cdot b^{\pm e},$$

with

- ▶ the base $b \in \mathbb{N}$ and $b > 1$
- ▶ the mantissa $m = m_1 b^{-1} + \dots + m_r b^{-r}$
- ▶ the exponent $e = e_{s-1} b^{s-1} + \dots + e_0 b^0$
- ▶ the digits $m_i, e_j \in \{0, \dots, b - 1\}$
- ▶ the number of significant digits $s \in \mathbb{N}$ and $r \in \mathbb{N}$
- ▶ m is normalized, that is $m_1 \neq 0$

Typical choices for the base b include $b = 10$ for humans and $b = 2$ on a computer.

Conversion to binary

Algorithm to write $x \in \mathbb{R}$ in base b representation:

1. Find k so that $b^k < x < b^{k+1}$
2. Divide x by b^k : the floored quotient becomes the digit, afterwards redefine $x := x - b^k$. $\lfloor \frac{x}{b^k} \rfloor$
3. Set $k := k - 1$. If $x \neq 0$ or $k > 0$ then go to step 1.

Example: Write $x = 12.85$ in floating-point format for base $b = 2$.

- ▶ First, $k = 3$. $\lfloor \frac{12.85}{2^3} \rfloor = \lfloor \frac{12.85}{8} \rfloor = 1$
- ▶ Quotient of $x/2^k$ is 1. Then, $x := 12.85 - 8 = 4.85$.
- ▶ Repeating this yields $\lfloor \frac{4.85}{2^2} \rfloor = 1$

$$12.85 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^{-2} + 1 \cdot 2^{-4} + \dots$$

- ▶ After normalization (i.e., division with 2^4) we obtain

$$\underline{12.85} = (\underline{0.110011011\dots})_2 \cdot 2^4$$

need ∞ digits for proper representation

Machine number

On computers, there are only limited number of digits:

- ▶ r digits +1 for the mantissa
- ▶ s digits +1 for the exponent

The implementation/storage of a number:

$$x = \pm [m_1 b^{-1} + \dots + m_r b^{-r}] \cdot b^{\pm[e_{s-1} b^{s-1} + \dots + e_0 b^0]}$$

is done in the following form

$$(\pm)[m_1 \dots m_r](\pm)[e_{(s-1)} \dots e_0].$$

For a given (b, r, s) any number that can be written in the above form is called *machine number*. The smallest and largest machine numbers are $x_{\max / \min} = \pm(1 - b^{-r})b^{b^{s-1}}$.

IEEE format

"IEEE-Format" (standard format for machine numbers on Unix):
Base $b = 2$ and 64 Bits (=8 Bytes) for storage

$$x = \pm m \cdot 2^{c-1022}$$

1 ↗ 52 ↗ 11

with

- ▶ 1 Bit for the sign b
- ▶ 52 Bits for the mantissa: $m = 2^{-1} + m_2 2^{-2} + \dots + m_{52} 2^{-52}$ r
- ▶ 11 Bits for the exponent $c = c_0 2^0 + \dots + c_{10} 2^{10}$ s

This results in $x_{\max / \min} = \pm 2^{1024} \approx 1.8 \cdot 10^{308}$.

Machine numbers: rounding

The set of all possible machine numbers A will not cover all possible numbers in \mathbb{R} . Rounding or approximation is therefore required. We define the rounding function $rd(\cdot)$ by

$$|x - rd(x)| = \min_{y \in A} |x - y|.$$

Example: In the IEEE-format, the rounding function $rd(\cdot)$ can be implemented by

$$rd(x) = sign(x) \cdot \begin{cases} 0.m_1 \dots m_{52} \cdot 2^e & \text{for } m_{53} = 0 \\ (0.m_1 \dots m_{52} + 2^{-52}) \cdot 2^e & \text{for } m_{53} = 1 \end{cases}$$

Rounding error

The absolute rounding error caused by the rounding function $rd(\cdot)$ is given by

$$|x - rd(x)| \leq \frac{1}{2}b^{-r+e}.$$

The relative rounding error caused by the rounding function $rd(\cdot)$ is given by

$$\left| \frac{x - rd(x)}{x} \right| \leq \frac{1}{2}b^{-r+1}.$$

Machine precision

The **machine precision** is defined by

$$\text{eps} = \frac{1}{2}b^{-r+1}.$$

We can rewrite the rounding function $rd(\cdot)$ as

$$rd(x) = x \cdot (1 + \epsilon) \quad \text{with } |\epsilon| \leq \text{eps}.$$

Example: For the IEEE-format the maximal rounding error is

$$\text{eps}_{\text{IEEE}} \leq \frac{1}{2}2^{-51} \approx 10^{-16}.$$

Basic arithmetic operations

The basic arithmetic operations $* \in \{+, -, \cdot, /\}$ are replaced by the corresponding machine operations $\otimes \in \{\oplus, \ominus, \odot, \oslash\}$. In order to minimize the overall numerical error, the machine implementation of the basic arithmetic operations has to satisfy

$$x \otimes y = rd(x * y) = (x * y)(1 + \epsilon) \quad |\epsilon| \leq \text{eps}.$$

It is important to note that the commutative law and the associative law do not hold for the machine implementation of the basic arithmetic operations:

$$(x \oplus y) \oplus z \neq x \oplus (y \oplus z),$$

$$(x \oplus y) \odot z \neq (x \odot z) \oplus (y \odot z).$$

Especially, it holds

$$x \oplus y = x \quad \text{for } |y| \leq \frac{|x|}{b} \text{eps}.$$

Example

Consider the following numbers:

$$a = 0.23371258 \cdot 10^{-4}$$

$$b = 0.33678429 \cdot 10^2$$

$$c = -0.33677811 \cdot 10^{-2}$$

Let's compute the sum $a + b + c$:

$$\begin{aligned}(a \oplus b) \oplus c &= 0.33678452 \cdot 10^2 \oplus -0.33677811 \cdot 10^{-2} \\&= 0.64100000 \cdot 10^{-3}\end{aligned}$$

$$\begin{aligned}a \oplus (b \oplus c) &= 0.233712858 \cdot 10^{-4} \oplus 0.61800000 \cdot 10^{-3} \\&= 0.64137126 \cdot 10^{-3}\end{aligned}$$

$$a + b + c = 0.641371258 \cdot 10^{-3}$$

Algorithm for integer addition

We are given two n -digit numbers $X = \sum_{i=0}^{n-1} x_i b^i$ and $Y = \sum_{i=0}^{n-1} y_i b^i$ in base b . By padding the number with zeros we can assume without loss of generality that X and Y have equal length. We want to compute $Z = X + Y$:

```
d = 0
for i = 0 to n - 1 do
    s = xi + yi + d
    zi = s mod b
    d = s div b
end
```

Runtime: $O(n)$

Is this as fast as it can be? Answer: Yes. (Why?)

Naïve integer multiplication

Same setup with $X = \sum_{i=0}^{n-1} x_i b^i$ and $Y = \sum_{i=0}^{n-1} y_i b^i$ in base b .
We aim to compute $Z = X \cdot Y$:

```
Z = 0
for i = 0 to n - 1 do
    for j = 0 to n - 1 do
        | Z = Z + (xi · yj) · (bi · bj)
    end
end
```

Runtime: $O(n^2)$

Question: Can this be improved?

Divide and conquer multiplication

Idea: Split up the multiplication of the two big numbers into several multiplication of smaller numbers and proceed recursively.

We assume n is even (pad one digit), and base $b = 2$. We split the two numbers in the middle into a *left* and *right* part:

$$X = \underbrace{x_{n-1} \dots x_{n/2}}_{X_L} \underbrace{x_{n/2-1} \dots x_0}_{X_R}$$

$$Y = \underbrace{y_{n-1} \dots y_{n/2}}_{Y_L} \underbrace{y_{n/2-1} \dots y_0}_{Y_R}$$

Then, $X = X_L \cdot 2^{n/2} + X_R$ and $Y = Y_L \cdot 2^{n/2} + Y_R$ and thus

$$X \cdot Y = X_L Y_L \cdot 2^n + (X_L Y_R + X_R Y_L) \cdot 2^{n/2} + X_R Y_R.$$

Divide and conquer multiplication

Let's look at

$$X \cdot Y = X_L Y_L \cdot 2^n + (X_L Y_R + X_R Y_L) \cdot 2^{n/2} + X_R Y_R.$$

We have splitted up the multiplication problem of two n -digit numbers into 4 multiplications of $n/2$ -digit numbers, hence:

$$T(n) = 4T(n/2) + n,$$

since the cost for splitting the numbers and for the additions is linear. Solving the recursion yields:

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &= 4(4T(n/4) + n/2) + n \\ &= \dots \\ &= 4^{\log_2 n} T(1) + \dots \\ &\propto n^2 \quad \leftarrow \text{same runtime as naïve method} \end{aligned}$$

Karatsuba–Ofman Trick

Let's look again at

$$X \cdot Y = \underline{X_L Y_L \cdot 2^n} + (X_L Y_R + X_R Y_L) \cdot 2^{n/2} + X_R Y_R.$$

↑
2 mult
↓

improve

Before we calculated $X_L Y_R + X_R Y_L$ using two multiplications.
However,

$$X_L Y_R + X_R Y_L = \underbrace{(X_L + X_R) \cdot (Y_L + Y_R)}_{\text{↑ 2 mult}} - X_L Y_L - X_R Y_R,$$

where $(X_L + X_R) \cdot (Y_L + Y_R)$ is **one multiplication** of two numbers of $O(n/2)$ digits, and $X_L Y_L$ and $X_R Y_R$ are already computed. This is called the **Karatsuba–Ofman Trick**.

Karatsuba–Ofman Trick

Thus we compute:

- ▶ $X_L Y_L$
- ▶ $X_R Y_R$
- ▶ $(X_L + X_R) \cdot (Y_L + Y_R) =: \Delta$

from which $X_L Y_R + X_R Y_L$ can be inferred in linear time using two subtractions (i.e., additions), leading to

$$X \cdot Y = X_L Y_L \cdot 2^n + (\Delta - X_L Y_L - X_R Y_R) \cdot 2^{n/2} + X_R Y_R$$

with only three multiplications of $n/2$ -digit numbers. Therefore,

$$T(n) = 3T(n/2) + n.$$

Solving the recursion as before yields

$$T(n) \in O\left(n^{\log_2(3)}\right) \approx O\left(n^{1.58}\right).$$

Numerical errors and vector norms

Review of vector norms:

- ▶ Magnitude, modulus, or absolute value for scalars generalizes to norms for vectors.
- ▶ We consider p -norms in \mathbb{R}^n defined by

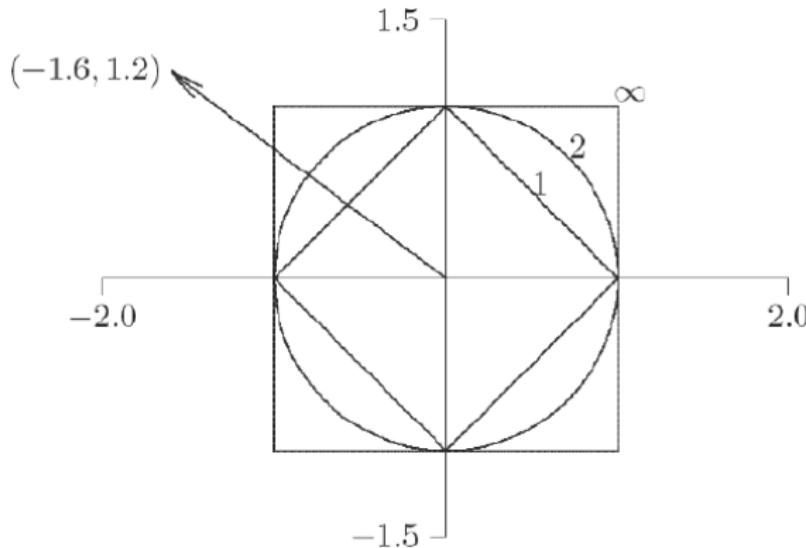
$$\|x\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{1/p}$$

for $p \in \mathbb{N}$ and $x \in \mathbb{R}^n$.

- ▶ Important special cases: 1-norm $\|x\|_1 = \sum_{i=1}^n |x_i|$,
 $\|x\|_2 = \sqrt{x_1^2 + \cdots + x_n^2}$ and ∞ -norm $\|x\|_\infty = \max_i |x_i|$.

Example of vector norms

Unit sphere in two dimensions for each norm:



- ▶ Displayed vector has the following norms: $\|x\|_1 = 2.8$, $\|x\|_2 = 2.0$, $\|x\|_\infty = 1.6$.
- ▶ In general, for any vector $x \in \mathbb{R}^n$: $\|x\|_1 \geq \|x\|_2 \geq \|x\|_\infty$.

Vector axioms

A norm must satisfy:

1. $\|x\| > 0$ for $x \neq 0$ (positive definite)
2. $\|\gamma x\| = |\gamma| \|x\|$ for any scalar γ (absolutely scalable)
3. $\|x + y\| \leq \|x\| + \|y\|$ (triangle inequality)

These properties are taken as the definition of a norm.

Useful variant of the triangle inequality: $\||x| - |y|\| \leq \|x - y\|$.

Numerical errors: propagated vs. computational error

We consider:

- ▶ A function f and its approximation \tilde{f}
- ▶ A value x and its approximation \tilde{x}

Then:

$$\|f(x) - \tilde{f}(\tilde{x})\| = \| \underbrace{f(x) - f(\tilde{x})}_{\text{propagated error}} + \underbrace{f(\tilde{x}) - \tilde{f}(\tilde{x})}_{\text{computational error}} \|$$

- ▶ propagated error: not affected by algorithm
- ▶ computational error: affected by algorithm

Total error = propagated error + computational error

Rounding vs. truncation error

- ▶ Rounding error: introduced by finite precision calculations in the computer arithmetic
- ▶ Truncation error: introduced by the algorithm through problem simplification, e.g. series truncation, iterative process truncation, etc.

Numerical errors through $+, -, *, /$

Numerical errors through basic arithmetic operations, that is $+$, $-$, $*$, $/$

In any mathematical algorithms, many arithmetic operations have to be executed.

Question: How does the initial rounding error propagate through the calculations of the algorithm? For this, let us examine the basic arithmetic operations.

Numerical errors through +,-,*,/

Assumptions:

- ▶ Inputs x and y are the exact values
- ▶ The corresponding machine numbers are \tilde{x} and \tilde{y}
- ▶ The relative numerical error is given by:

$$\delta_x = \frac{\tilde{x} - x}{x}, \quad \delta_y = \frac{\tilde{y} - y}{y}.$$

This implies that $\tilde{x} = x(1 + \delta_x)$ and $\tilde{y} = y(1 + \delta_y)$.

Let us now assume that $|\delta_x|, |\delta_y| \leq \epsilon < 1$ where ϵ denotes the machine precision *eps*.

Relative numerical error of multiplication

We want to multiply $x \cdot y$ but the computer calculates:

$$\begin{aligned}\tilde{x} \cdot \tilde{y} &= (x(1 + \delta_x)) \cdot (y(1 + \delta_y)) \\ &= (x \cdot y)(1 + \delta_x + \delta_y + \delta_x \delta_y) \\ &=: (x \cdot y)(1 + \delta)\end{aligned}$$

where the error δ satisfies

$$|\delta| = |\delta_x + \delta_y + \delta_x \delta_y| \leq |\delta_x| + |\delta_y| + |\delta_x \delta_y| \leq \epsilon(2 + \epsilon) \cong 2\epsilon$$

since $\epsilon^2 \ll \epsilon < 1$.

Then, the numerical error is given by

$$\left| \frac{\tilde{x} \cdot \tilde{y} - x \cdot y}{x \cdot y} \right| = |\delta| \leq 2\epsilon + \epsilon^2 \leq 3\epsilon.$$

Division

Let's look at the quotient of \tilde{x} and \tilde{y} now:

$$\frac{\tilde{x}}{\tilde{y}} = \frac{x(1 + \delta_x)}{y(1 + \delta_y)}.$$

If $|\delta_y| < 1$ then $\frac{1}{1 + \delta_y} = \sum_{j=0}^{\infty} (-1)^j \delta_y^j$. In this case, the fraction simplifies to

$$\begin{aligned}\frac{1 + \delta_x}{1 + \delta_y} &= (1 + \delta_x) \sum_{j=0}^{\infty} (-1)^j \delta_y^j \\ &= (1 + \delta_x) \left(1 + \sum_{j=1}^{\infty} (-1)^j \delta_y^j \right) \\ &= 1 + \delta_x + (1 + \delta_x) \sum_{j=1}^{\infty} (-1)^j \delta_y^j \\ &=: 1 + \delta.\end{aligned}$$

Division

As before, we bound $|\delta|$ as:

$$\begin{aligned} |\delta| &\leq \epsilon + (1 + \epsilon) \sum_{j=1}^{\infty} \epsilon^j \\ &= \epsilon + (1 + \epsilon)\epsilon \sum_{j=0}^{\infty} \epsilon^j \\ &= \epsilon \left(1 + \frac{1 + \epsilon}{1 - \epsilon}\right) \\ &= \epsilon \left(\frac{1 - \epsilon}{1 - \epsilon} + \frac{1 + \epsilon}{1 - \epsilon}\right) \\ &= 2\epsilon \frac{1}{1 - \epsilon}. \end{aligned}$$

If $\epsilon < \frac{1}{2}$ then the relative numerical error is $|\delta| \leq 4\epsilon$. **The relative error is of the same order as the one for the multiplication.**

Addition

Let's compute the sum of \tilde{x} and \tilde{y} :

$$\begin{aligned}\tilde{x} + \tilde{y} &= x(1 + \delta_x) + y(1 + \delta_y) \\&= x + y + x\delta_x + y\delta_y \\&= (x + y) \left(1 + \frac{x}{x+y}\delta_x + \frac{y}{y+x}\delta_y \right) \\&=: (x + y)(1 + \delta)\end{aligned}$$

where we bound $|\delta|$ as before:

$$\begin{aligned}|\delta| &\leq \left| \frac{x}{x+y} \right| |\delta_x| + \left| \frac{y}{x+y} \right| |\delta_y| \\&\leq (|\delta_x| + |\delta_y|) \max \left\{ \left| \frac{x}{x+y} \right|, \left| \frac{y}{x+y} \right| \right\} \\&\leq 2\epsilon \max \left\{ \left| \frac{x}{x+y} \right|, \left| \frac{y}{x+y} \right| \right\}.\end{aligned}$$

Addition

Problem: We saw that

$$|\delta| \leq 2\epsilon \max \left\{ \left| \frac{x}{x+y} \right|, \left| \frac{y}{x+y} \right| \right\}$$

but the right hand side cannot be solely bounded in ϵ !

Looking at the denominator we see that for $x \approx -y$, the numerical error becomes arbitrarily large!

Addition

Example: Given two numbers in base b which differ from digit $r + 1$ onwards,

$$x = 0.d_1 \dots d_r d_{r+1} \dots d_m \cdot b^e,$$

$$y = -0.d_1 \dots d_r \tilde{d}_{r+1} \dots d_m \cdot b^e,$$

with $d_j < \tilde{d}_j$ for $j \geq r + 1$. Due to numerical cancellations in the first r digits, we obtain

$$x + y = 0.0 \dots 0 \hat{d}_{r+1} \dots \hat{d}_m \cdot b^e.$$

In this case, the bound on the addition error becomes

$$\left| \frac{x}{x+y} \right| = \left| \frac{0.d_1 \dots d_m \cdot b^e}{0.\hat{d}_{r+1} \dots \hat{d}_m \cdot b^{e-r}} \right| \geq \frac{b^{-1} b^e}{b^{e-r}} = b^{e-r},$$

where it was used that $d_1 \neq 0$ implies $0.d_1 \dots d_m \geq b^{-1}$ and that trivially, $0.\hat{d}_{r+1} \dots \hat{d}_m \leq 1$.

Addition

On the last slide we saw that the bound on the addition error becomes

$$\left| \frac{x}{x+y} \right| = \left| \frac{0.d_1 \dots d_m \cdot b^e}{0.\hat{d}_{r+1} \dots \hat{d}_m \cdot b^{e-r}} \right| \geq \frac{b^{-1}b^e}{b^{e-r}} = b^{e-r},$$

where it was used that $d_1 \neq 0$ implies $0.d_1 \dots d_m \geq b^{-1}$ and that trivially, $0.\hat{d}_{r+1} \dots \hat{d}_m \leq 1$.

Thus r digits in the mantissa are lost.

Well-posed and ill-posed numerical problems

Definition: The problem is **well-posed** if

1. a solution exists
2. it is unique
3. it depends continuously on the problem/input data, meaning that small perturbations in the problem data results in a small perturbation in the solution.

Note: The problem can be well-posed but still sensitive to perturbations. An algorithm should attempt to simplify the problem, but not make the sensitivity worse than it already is.

Well-posed and ill-posed numerical problems

Problem simplification:

- ▶ infinite → finite
- ▶ nonlinear → linear
- ▶ high order → low order.

Sources of numerical errors

Before computation:

- ▶ modeling approximations
 - ▶ empirical measurements, human errors
 - ▶ previous computations
- *cannot be controlled*

During computation:

- ▶ truncation or discretization
 - ▶ rounding errors
- *can be controlled through error analysis*

Sources of numerical errors

We have seen: errors before and during computation...

- ▶ Accuracy depends on both, but we can only control the second part.
- ▶ Uncertainty in input may be amplified by problem.
- ▶ Perturbations during computations may be amplified by algorithm.

Two general types of error

1. *Data error* is the error in the data x . In reality, numerical analysis involves solving a problem with approximate data \hat{x} (measurements) since the exact data is often unavailable.
2. *Computational error* refers to the error that occurs when attempting to compute $f(\hat{x})$. Effectively, we must approximate $f(\hat{x})$ by the quantity $\hat{f}(\hat{x})$, where \hat{f} is a function that approximates f .

The condition number

The condition number is a measure of well/ill-posedness of a problem.

Example: Find the intersection of two (straight) lines given by

$$g_1 = \{(x_1, x_2) \in \mathbb{R}^2 : a_{11}x_1 + a_{12}x_2 = b_1\}$$

$$g_2 = \{(x_1, x_2) \in \mathbb{R}^2 : a_{21}x_1 + a_{22}x_2 = b_2\}$$

for given parameters $b_i \in \mathbb{R}$, $a_{ij} \in \mathbb{R}$, $1 \leq i, j \leq 2$.

Intersection of two lines

Expressed in matrix form, the intersection of two lines is the solution of $Ax = b$ with

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \in \mathbb{R}^{2 \times 2}$$

and $b = (b_1, b_2)^\top \in \mathbb{R}^2$.

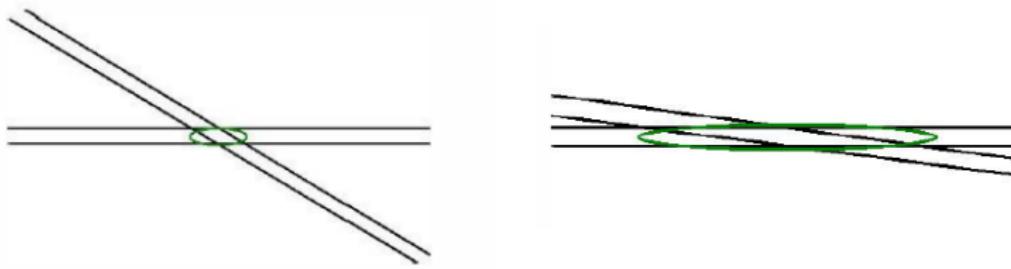
The solution (the intersection) is $x = A^{-1}b$ given the inverse of A exists. We define $f(A, b) := A^{-1}b$.

Question: How stable will the computation of A^{-1} be?

As a picture

We want to find the intersection of two straight lines.

Question: What is the impact of round-off errors in the matrix A and the vector b on the function f and the solution x ?



Left picture: well-defined intersection. Right picture: precise intersection point is less clear. We intuitively say that the left scenario corresponds to a well-posed problem and the right scenario is an ill-posed problem.

Well-posed and ill-posed problems

We formalize this notion now.

- ▶ We assume the problem is formally given by some function

$$f : X \rightarrow Y$$

between some input space X and some output space Y .

- ▶ In order to measure if the problem is well/ill-posed we need distance measures in both X and Y . That is we require norms $\|\cdot\|_X$ and $\|\cdot\|_Y$ on both spaces, e.g.

$$\|x\|_2 = (x^\top x)^{1/2} = \left(\sum_{i=1}^n |x_i|^2 \right)^{1/2}.$$

Relative input/output errors

Assuming all arithmetic operations are exact we obtain the following picture:

$$\begin{array}{ccc} \textbf{input} & \textbf{problem} & \textbf{output} \\ \tilde{x} \in X & \longrightarrow f : X \rightarrow Y & \longrightarrow \tilde{y} = f(\tilde{x}) \end{array}$$

error:

$$\Delta x = \tilde{x} - x$$

error:

$$\Delta y = f(\tilde{x}) - f(x)$$

The **relative input error** and **relative output error** are given by

$$\delta_x := \frac{\|\Delta x\|_X}{\|x\|_X} \quad \delta_y := \frac{\|\Delta y\|_Y}{\|y\|_Y},$$

where $y = f(x)$.

Relative and absolute condition numbers

The **relative input error** and **relative output error** are given by

$$\delta_x := \frac{\|\Delta x\|_X}{\|x\|_X} \quad \delta_y := \frac{\|\Delta y\|_Y}{\|y\|_Y}.$$

The quantity δ_x measures the relative error on x in the norm on the space X , and analogously for δ_y .

The **relative condition number** of the problem f is given by

$$\kappa_f := \frac{\delta_y}{\delta_x},$$

and the **absolute condition number** of the problem f is

$$\kappa_{f,\text{abs}} := \frac{\|\Delta y\|_Y}{\|\Delta x\|_X}.$$

Interpretations of both?

Well/ill-conditioned problems

Definition:

- ▶ Well-conditioned (insensitive) problem: small relative change in the inputs gives commensurate relative change in the solution
- ▶ Ill-conditioned (sensitive) problem: relative change in output is much larger than the one of the input data

We say that a problem is

- ▶ *well-conditioned* if for small input error $\delta_x \rightarrow 0_+$ there are small upper limits for κ_f , e.g. $\kappa_f \approx 1$.
- ▶ otherwise the problem is called *ill-conditioned*.

Example: Taylor expansion

With the help of the Taylor expansion we can derive a formula for the relative output error.

Assume we want to compute $f(\tilde{x})$, and Taylor expand it at some x :

$$f(\tilde{x}) = f(x) + (\tilde{x} - x)^\top f'(x) + \frac{1}{2}(\tilde{x} - x)^\top f''(x)(\tilde{x} - x) + O(\|\tilde{x} - x\|^3).$$

If $\tilde{x} \approx x$, it suffices to look at the first order:

$$f(\tilde{x}) = f(x) + (\tilde{x} - x)^\top f'(x),$$

or equivalently,

$$f(\tilde{x}) - f(x) = (\tilde{x} - x)^\top f'(x).$$

Example: Taylor expansion

We saw the first order approximation $f(\tilde{x}) - f(x) = (\tilde{x} - x)^\top f'(x)$.
Then,

$$\begin{aligned}\delta_y &= \left| \frac{f(\tilde{x}) - f(x)}{f(x)} \right| = \left| \frac{(\tilde{x} - x)^\top f'(x)}{f(x)} \right| \\ &= \left| \sum_{i=1}^n \underbrace{\frac{\partial f(x)}{\partial x_i} \frac{x_i}{f(x)}}_{\varphi_i} \cdot \underbrace{\frac{\tilde{x}_i - x_i}{x_i}}_{\delta_x} \right|.\end{aligned}$$

Take-away: The condition number can be expressed in terms of the relative input error δ_x and the *amplification factor(s)* φ_i . This makes sense: The condition number is an indicator of how severely input errors propagate (or get amplified) in a computation.

Summary of notions

- ▶ Absolute forward error:

$$f(x + \Delta x) - f(x) \approx f'(x)\Delta x.$$

- ▶ Relative forward error:

$$\frac{f(x + \Delta x) - f(x)}{f(x)} \approx \frac{f'(x)\Delta x}{f(x)}.$$

- ▶ Relative condition number can be rewritten using the derivative of f :

$$\kappa_f = \frac{\delta_y}{\delta_x} = \frac{f'(x)\Delta x/f(x)}{\Delta x/x} = \frac{xf'(x)}{f(x)}.$$

Example: Multiplication

We want to compute the function $f(x) = x_1 \cdot x_2$, where $x = (x_1, x_2)^\top \in \mathbb{R}^2$.

Derivatives: $\frac{\partial f(x)}{\partial x_i} = x_2$ and $\frac{\partial f(x)}{\partial x_2=x_1}$.

Then, the amplification factors are

$$\varphi_1 = \frac{\partial f(x)}{\partial x_1} \cdot \frac{x_1}{f(x)} = x_2 \cdot \frac{x_1}{x_1 x_2} = 1,$$

$$\varphi_2 = \frac{\partial f(x)}{\partial x_2} \cdot \frac{x_2}{f(x)} = x_1 \cdot \frac{x_2}{x_1 x_2} = 1.$$

Multiplication is a well-conditioned problem.

Stability of an algorithm

Definition: An algorithm to compute a function f is called *stable* if the numerical error Δy remains of the same magnitude as the condition number of f .

- ▶ **Stability is a property of the algorithm.**
- ▶ **The condition number is a feature of the problem.**

Example: Root finding

We aim to compute the roots of $f(x) = x^2 - 2a_1x + a_2$.

We will see now that the problem is well-conditioned. One root is given by:

$$x^* = a_1 - \sqrt{a_1^2 - a_2}.$$

Numeric example:

$$a_1 = 6.0002, \quad a_2 = 0.01.$$

("exact" solution: $8.3336 \cdot 10^{-4}$)

Example: Root finding

Numeric example:

$$a_1 = 6.0002, \quad a_2 = 0.01.$$

Algorithm 1 computes $x^* = a_1 - \sqrt{a_1^2 - a_2}$ "from the inside to the outside":

$$\begin{aligned}y_1 &:= a_1 \cdot a_1 &\longrightarrow & 36.002|40004 \\y_2 &:= y_1 - a_2 &\longrightarrow & 35.992|40004 \\y_3 &:= \sqrt{y_2} &\longrightarrow & 5.9993|66637 \\x^* &:= a_1 - y_3 &\longrightarrow & 0.0009|00000\end{aligned}$$

Algorithm 1 is not stable. What happened?

Example: Root finding

Algorithm 2 computes

$$x^* = a_1 - \sqrt{a_1^2 - a_2} = \frac{a_2}{a_1 + \sqrt{a_1^2 - a_2}}.$$

Computations:

$$\begin{aligned}y_1 &:= a_1 \cdot a_2 &\longrightarrow 36.002|40004 \\y_2 &:= y_1 - a_2 &\longrightarrow 35.992|40004 \\y_3 &:= \sqrt{y_2} &\longrightarrow 5.9993|66637 \\y_4 &:= a_1 + y_3 &\longrightarrow 11.9995|66637 \\x^* &:= a_2/y_4 &\longrightarrow 0.0008333|3\end{aligned}$$

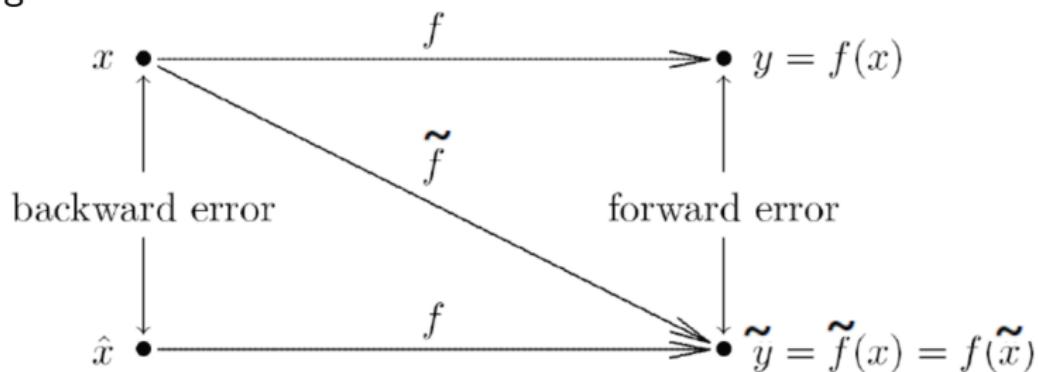
Algorithm 2 is stable. What happened?

Backward vs. forward error

Suppose we want to compute $y = f(x)$, where $f : \mathbb{R} \rightarrow \mathbb{R}$, but obtain an approximate value $\tilde{y} = f(\tilde{x})$. We define:

- ▶ **Forward error:** $\Delta y = \tilde{y} - y$
- ▶ **Backward error:** $\Delta x = \tilde{x} - x$

Diagram:



Formal definitions

Definition: (Forward error). Let x be a real number and let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a function. If \hat{y} is a real number that is an approximation to $y = f(x)$, then the **forward error** in \hat{y} is the difference $\Delta y = \hat{y} - y$. If $y \neq 0$, then the **relative forward error** in \hat{y} is defined as

$$\frac{\Delta y}{y} = \frac{\hat{y} - y}{y}.$$

An alternative approach is to instead view the computed value \hat{y} as the *exact solution* of a nearby problem with modified data, i.e. $\hat{y} = f(\hat{x})$ where \hat{x} is a perturbation of x . This leads to the **backward error**.

Formal definitions

Definition: (Backward error). Let x be a real number and let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a function. Suppose that \hat{y} is an approximation to $y = f(x)$, and that \hat{y} is in the image of f , meaning that $\hat{y} = f(\hat{x})$ for some real number \hat{x} . Then, the quantity $\Delta x = \hat{x} - x$ is the **backward error** in \hat{y} . If $x \neq 0$, then the **relative backward error** in \hat{y} is defined as

$$\frac{\Delta x}{x} = \frac{\hat{x} - x}{x}.$$

Formal definitions

Definition: (Condition number) Let x be a real number and let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a function. The **absolute condition number**, denoted by κ_{abs} , is the ratio of the magnitude of the forward to backward errors:

$$\kappa_{\text{abs}} = \frac{|f(\hat{x}) - f(x)|}{|\hat{x} - x|}.$$

The **relative condition number** of the problem $y = f(x)$ is

$$\kappa_{\text{rel}} = \frac{|(f(\hat{x}) - f(x))/f(x)|}{|(\hat{x} - x)/x|} = \frac{|\Delta y/y|}{|\Delta x/x|}.$$

The relative condition number is usually referred to as the *condition number*. Problems with a condition number much larger than 1 are called *ill-conditioned*.

Backward error analysis

Questions:

- ▶ How much does the original problem have to change to give the result which was actually obtained?
- ▶ How much data error in the input would explain all errors in the computed result?

Backward error analysis is often easier than forward error analysis. Oftentimes, an approximate solution is "good" if it is the exact solution to a nearby problem.

Backward vs. forward error

We want to compute $y = \sqrt{2}$ and instead we obtain the approximation $\hat{y} = 1.4$.

- ▶ The approximation $\hat{y} = 1.4$ has absolute forward error

$$\Delta y = |\hat{y} - y| = |1.4 - 1.41421\dots| \approx 0.0142$$

or a relative forward error of about 1%.

- ▶ Since $\hat{y} = 1.4 = \sqrt{1.96} =: \sqrt{\hat{x}}$ the absolute backward error is

$$\Delta x = |\hat{x} - x| = |1.96 - 2| = 0.04$$

or a relative backward error of 2%.

Rule of thumb

Some guidelines for developing stable algorithms:

- ▶ Avoid subtractions of numbers of similar magnitude.
- ▶ Subtraction that cannot be avoided should be placed at the beginning of the algorithm.
- ▶ Addition of numbers should be done when they are of similar magnitude.

You can check your existing programs for optimized numeric implementations...

Homework

Consider two ways to evaluate a polynomial for some $x \in \mathbb{R}$:

1. $p(x) = a_0 + a_1x + \dots + a_nx^n$ (traditional)
2. $p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \dots + x(a_{n-1} + xa_n) \dots)))$
(Horner's method)

Which way is more stable? Which way is more efficient?

Stability of linear equation systems

We want to compute the solution of

$$Ax = b,$$

where $A \in \mathbb{R}^{n \times n}$ is a given matrix, $b \in \mathbb{R}^n$ is a given vector, and $x \in \mathbb{R}^n$ is unknown.

The solution is given by

$$f(A, b) := A^{-1}b = x.$$

Question: Is this a well-conditioned or ill-conditioned problem.
What is the decisive quantity?

Stability of linear equation systems

Question: Is the problem of computing

$$f(A, b) := A^{-1}b = x.$$

a well- or ill-conditioned problem?

For this, let's look at the perturbed system

$$\tilde{A}\tilde{x} = \tilde{b}$$

with $\tilde{A} = A + \Delta A$ and $\tilde{b} = b + \Delta b$. The relative input errors are:

$$\delta_A = \frac{\|\tilde{A} - A\|}{\|A\|},$$

$$\delta_b = \frac{\|\Delta b\|}{\|b\|}.$$

Stability of linear equation systems

For $\tilde{A}\tilde{x} = \tilde{b}$, the relative input errors are

$$\delta_A = \frac{\|\tilde{A} - A\|}{\|A\|} \quad \delta_b = \frac{\|\Delta b\|}{\|b\|}.$$

Then the relative output error is

$$\delta_x = \frac{\|\Delta x\|}{\|x\|}.$$

Question: How does δ_x behave as a function of the relative input error?

Stability of linear equation systems

Question: How does δ_x behave as a function of the relative input error?

Theorem: For the system $Ax = b$ and relative input errors δ_A and δ_b on A and b , it holds true that

$$\delta_x \leq \frac{\kappa(A)}{1 - \delta_A \cdot \kappa(A)} \cdot (\delta_b + \delta_A),$$

where $\kappa(A)$ is the condition number of the matrix A defined by

$$\kappa(A) := \text{cond}(A) := \|A\| \cdot \|A^{-1}\|.$$

The condition number of the matrix A is the decisive factor.

Matrix norm

The norm of a matrix A is defined by

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}.$$

Observations:

- ▶ The norm of a matrix measures the maximal proportion/magnitude a vector is stretched when multiplied by A .
- ▶ The norm $\|\cdot\|$ on the left is the matrix norm, and it is reduced to norms on vectors $\|\cdot\|$ on the right hand side.

Examples of matrix norms

Let $A = (a_{ij})$ be a matrix.

- ▶ The matrix norm which corresponds to the 1-norm of a vector is the *absolute column sum*:

$$\|A\|_1 = \max_j \sum_{i=1}^n |a_{ij}|.$$

- ▶ The matrix norm which corresponds to the ∞ -norm is the *absolute row sum*:

$$\|A\|_\infty = \max_i \sum_{j=1}^n |a_{ij}|.$$

- ▶ A good way to memorize these relationships is the observation that both norms agree with their corresponding vector norms for an $n \times 1$ matrix.

Properties of matrix norms

Analogously to vector norms, a matrix norm needs to satisfy:

1. $\|A\| > 0$ for all $A \neq 0$.
2. $\|\gamma A\| = |\gamma| \cdot \|A\|$ for any scalar γ .
3. $\|A + B\| \leq \|A\| + \|B\|$.

The matrix norms $\|A\|_1$ and $\|A\|_\infty$ from the previous slide also satisfy:

1. $\|AB\| \leq \|A\| \cdot \|B\|$
2. $\|Ax\| \leq \|A\| \cdot \|x\|$ for any vector x

Condition number

We have seen the definition of the condition number

$$\text{cond}(A) := \kappa(A) := \|A\| \cdot \|A^{-1}\|$$

for any square non-singular matrix A . By convention,
 $\text{cond}(A) = \infty$ for singular matrices A .

Since

$$\|A\| \cdot \|A^{-1}\| = \left(\max_{x \neq 0} \frac{\|Ax\|}{\|x\|} \right) \cdot \left(\min_{x \neq 0} \frac{\|Ax\|}{\|x\|} \right)^{-1}$$

the condition number measures the ratio of maximum stretching to maximum shrinking that the matrix does to any nonzero vector.

Large $\text{cond}(A)$ means that A is nearly singular.

Properties of the condition number

- ▶ For any matrix A , we have $\text{cond}(A) \geq 1$.
- ▶ For the identity matrix, $\text{cond}(I) = 1$.
- ▶ For any matrix A and scalar γ , $\text{cond}(\gamma A) = \text{cond}(A)$.
- ▶ For any diagonal matrix $D = \text{diag}(d_1, \dots, d_n)$, we have

$$\text{cond}(D) = \frac{\max_i |d_i|}{\min_i |d_i|}.$$

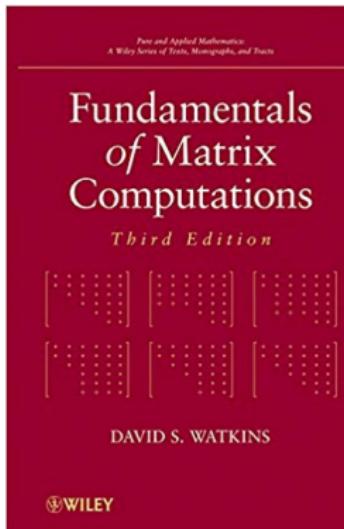
Homework

Compute the condition number of the following two matrices:

$$A_1 = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \quad A_2 = \begin{pmatrix} 168 & 113 \\ 113 & 76 \end{pmatrix}$$

Which one do you think will have the larger condition number?

Text book for the second part of the course



David S Watkins:
Fundamentals of Matrix Computations, Second/Third Edition

Systems of linear equations

Contents:

1. Singular and nonsingular matrices
2. Linear equation systems, triangular systems
3. Gauss elimination
4. LU and Cholesky decompositions

Matrix basics

- ▶ Question: Given matrix $A \in \mathbb{R}^{m \times n}$ and vector $b \in \mathbb{R}^m$, find $x \in \mathbb{R}^n$ such that $Ax = b$.
- ▶ In other words: "Can b be expressed as a linear combination of the columns of A ?"
- ▶ The solution vector x contains the coefficients for the linear combination of columns of A .
- ▶ Three possibilities: solutions may exist or not, and may or not be unique.

Nonsingular matrices

Consider the case $m = n$, that is $A \in \mathbb{R}^{n \times n}$.

A matrix A is called **nonsingular** if it has any of the following equivalent properties:

- ▶ the inverse matrix A^{-1} exists (and is unique)
- ▶ $\det(A) \neq 0$
- ▶ $\text{rank}(A) = n$
- ▶ $Az \neq 0$ for any vector $z \neq 0$

Nonsingular matrices

- ▶ The existence and uniqueness of the solution of $Ax = b$ depends on whether A is singular or nonsingular.
- ▶ If A is singular, existence of the solution x may also depend on b .
- ▶ We say that the system is *consistent* if $b \in \text{span}(A)$, that is if b belongs to the space spanned by the columns of A .

Summary:

A	b	# solutions
nonsingular	arbitrary	one (unique)
singular	$b \in \text{span}(A)$	infinitely many
singular	$b \notin \text{span}(A)$	none

Example

Consider the system of linear equations

$$2x_1 + 3x_2 = b_1$$

$$5x_1 + 4x_2 = b_2$$

which is expressed in matrix form as

$$Ax = b \quad \text{with } A = \begin{bmatrix} 2 & 3 \\ 5 & 4 \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.$$

This system is nonsingular (since A^{-1} exists) regardless of the value of b .

For example, for $b = [8, 13]^\top$, the solution is $x = [1, 2]^\top$.

Triangular form

What type of linear equation system is easy to solve?

- ▶ Consider one matrix row (=one equation) with only one nonzero entry. Then the solution for the corresponding x_i is computable with only one division.
- ▶ If another row exists having a nonzero entry for x_i and another x_j , then likewise x_j can be solved with one substitution and one division.
- ▶ Continuing in this fashion shows that if the matrix has a *triangular* form, then all components of x can be computed in succession.

Example of a triangular system

Consider the system

$$\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix}.$$

- ▶ Since the matrix is triangular, we can solve the system via back-substitution: $4x_3 = 8$ yields $x_3 = 2$.
- ▶ Afterwards, using $x_3 = 2$ the second row becomes $x_2 + x_3 = 4$, implying $x_2 = 2$.
- ▶ Finally, substituting $x_2 = 2$ and $x_3 = 2$ into the first row yields $2x_1 = -2$ and thus $x_1 = -1$.

Triangular form

Two triangular forms are of interest:

$$\begin{bmatrix} x_{11} & & 0 \\ \vdots & \ddots & \\ x_{1n} & \cdots & x_{nn} \end{bmatrix}$$

$$\begin{bmatrix} x_{11} & \cdots & x_{n1} \\ & \ddots & \vdots \\ 0 & & x_{nn} \end{bmatrix}$$

lower triangular

upper triangular

- ▶ lower triangular: $x_{ij} = 0$ for $i < j$, i.e. all entries *above* main diagonal are zero
- ▶ upper triangular: $x_{ij} = 0$ for $i > j$, i.e. all entries *below* main diagonal are zero
- ▶ successive substitution is especially easy to formulate for lower and upper triangular matrices
- ▶ any triangular matrix can be permuted into lower or upper triangular form using suitable row permutations

Gaussian elimination

Consider the lower triangular system

$$L = \begin{bmatrix} x_{11} & & 0 \\ \vdots & \ddots & \\ x_{1n} & \cdots & x_{nn} \end{bmatrix}.$$

Forward substitution in $L = (l_{ij})$ yields:

$$x_1 = b_1 / l_{11}$$

$$x_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j \right) / l_{ii} \quad \text{for all } i \in \{2, \dots, n\}$$

Pseudo-code for Gaussian elimination

Forward substitution in $L = (l_{ij})$ yields:

$$x_1 = b_1 / l_{11}$$

$$x_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j \right) / l_{ii} \quad \text{for all } i \in \{2, \dots, n\}$$

```
// loop over columns
for  $j = 1$  to  $n$  do
    if  $|l_{jj}| = 0$  then
        | stop // stop if matrix is singular
    end
     $x_j = b_j / l_{jj}$  // compute solution component
    for  $i = j + 1$  to  $n$  do
        |  $b_i = b_i - l_{ij} x_j$  // update right-hand side
    end
end
```

Pseudo-code for Gaussian elimination

Backward substitution in $U = (u_{ij})$ yields:

$$x_n = b_n / u_{nn}$$

$$x_i = \left(b_i - \sum_{j=i+1}^n u_{ij}x_j \right) / u_{ii} \quad \text{for all } i \in \{n-1, \dots, 1\}$$

```
// loop backwards over columns
for  $j = n$  to 1 do
    if  $u_{jj} = 0$  then
        | stop // stop if matrix is singular
    end
     $x_j = b_j / u_{jj}$  // compute solution component
    for  $i = 1$  to  $j - 1$  do
        |  $b_i = b_i - u_{ij}x_j$  // update right-hand side
    end
end
```

How to transform into triangular form?

- ▶ To transform a general linear equation system into triangular form, we need to replace selected nonzero matrix entries by zeros.
- ▶ This can be accomplished by taking linear combinations of rows.
- ▶ Consider the 2-column: $[a_1, a_2]^\top$.
- ▶ If $a_1 \neq 0$, multiplying the column with a selected matrix yields:

$$\begin{bmatrix} 1 & 0 \\ -a_2/a_1 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} a_1 \\ 0 \end{bmatrix}.$$

- ▶ This matrix is called an *elimination matrix* for the column $[a_1, a_2]^\top$.

General elimination matrices

More generally we can eliminate *all entries* below the k th position in an n -vector a with the transformation:

$$M_k a = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -m_{k+1} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -m_n & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where $m_i = a_i/a_k$ for all $i \in \{k + 1, \dots, n\}$. The divisor a_k must be nonzero and is called the **pivot**.

Elimination matrices

- ▶ The matrix M_k is called an **elimination matrix**. It adds multiples of row k to each subsequent row, where the multipliers m_i are chosen such that the resulting rows are zero.
- ▶ M_k is unit lower triangular and nonsingular.
- ▶ We can write

$$M_k = I - m_k e_k^\top$$

where

$$m_k = [0, \dots, 0, m_{k+1}, \dots, m_n]^\top$$

and e_k is the k th column of the identity matrix I (or equivalently, e_k is the k th unit vector).

- ▶ $M_k^{-1} = I + m_k e_k^\top$ which means that $M_k^{-1} =: L_k$ is the same as M_k but with reversed signs of all multipliers.

Example

Consider the column

$$a = \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix}.$$

Then,

$$M_1 a = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}$$

and

$$M_2 a = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 0 \end{bmatrix}.$$

Example

Note that

$$L_1 = M_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}, \quad L_2 = M_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1/2 & 1 \end{bmatrix}$$

and

$$M_1 M_2 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 1/2 & 1 \end{bmatrix}, \quad L_1 L_2 = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1/2 & 1 \end{bmatrix}.$$

What does the second result imply?

Gauss elimination

- ▶ Multiplication of elimination matrices requires only to copy matrix elements. No numerical operations needed.
- ▶ To reduce a general system of linear equations $Ax = b$ to upper triangular form, first choose M_1 with a_{11} as pivot and eliminate the first column of A below the first row.
- ▶ This transforms the system into $M_1Ax = M_1b$ but leaves the solution unchanged.
- ▶ Next, choose M_2 using a_{22} as pivot and eliminate the second column of M_1A below the second row.
- ▶ This transforms the system into $M_2M_1Ax = M_2M_1b$ but leaves the solution still unchanged.
- ▶ This process continues for each successive column until all subdiagonal entries have been zeroed.

Gauss elimination

Eliminating all rows and columns results in a sequence

$$\begin{aligned}M_{n-1} \cdots M_1 A x &= M_{n-1} \cdots M_1 b \\MAx &= Mb\end{aligned}$$

- ▶ This system can now be solved by back-substitution to obtain the solution for x of the original system $Ax = b$.
- ▶ This process is called **Gaussian elimination**.



Johann Carl Friedrich Gauss
(1777 – 1855)

Example: Gauss elimination

We are given the system $Ax = b$ with

$$A = \begin{bmatrix} 2 & -1 & -3 \\ 6 & 1 & -10 \\ -2 & -7 & 8 \end{bmatrix}, \quad b = \begin{bmatrix} 4 \\ -1 \\ 25 \end{bmatrix}.$$

Example: Gauss elimination

Recipe: Add $- <\text{first row}> * <\text{factor on the left}>$ to the second column:

$$\begin{matrix} 3 \\ -1 \end{matrix} \begin{bmatrix} 2 & -1 & -3 \\ 6 & 1 & -10 \\ -2 & -7 & 8 \end{bmatrix} \quad L = \begin{bmatrix} 1 \\ 3 \\ -1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & -1 & -3 \\ & & \\ & & \end{bmatrix}$$

$$\begin{matrix} -2 \end{matrix} \begin{bmatrix} 2 & -1 & -3 \\ 0 & 4 & -1 \\ 0 & -8 & 5 \end{bmatrix} \quad L = \begin{bmatrix} 1 \\ 3 \\ -1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & -1 & -3 \\ 4 & -1 \\ & & \end{bmatrix}$$

$$\begin{bmatrix} 2 & -1 & -3 \\ 0 & 4 & -1 \\ 0 & 0 & 3 \end{bmatrix} \quad L = \begin{bmatrix} 1 \\ 3 \\ -1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & -1 & -3 \\ 4 & -1 \\ 3 & & \end{bmatrix}$$

LU decomposition

After running the Gauss algorithm on the last slide, we obtained

$$L = \begin{bmatrix} 1 & & \\ 3 & 1 & \\ -1 & -2 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & -1 & -3 \\ & 4 & -1 \\ & & 3 \end{bmatrix}.$$

We observe that

$$LU = \begin{bmatrix} 2 & -1 & -3 \\ 6 & 1 & -10 \\ -2 & -7 & 8 \end{bmatrix} = A.$$

LU decomposition

In more generality, we have:

- ▶ The product $L_k L_j$ is unit lower triangular if $k < j$, hence

$$L = M^{-1} = M_1^{-1} \cdots M_{n-1}^{-1} = L_1 \cdots L_{n-1}$$

is unit lower triangular.

- ▶ By design, $U = MA$ is upper triangular.
- ▶ Combined, we obtain

$$A = LU,$$

where L is unit lower triangular and U is upper triangular.

- ▶ We obtain the **LU factorization** or **LU decomposition** into triangular factors with the Gauss elimination algorithm.

LU decomposition

- ▶ We assume we computed $A = LU$.
- ▶ Then the linear equation system $Ax = b$ becomes $LUX = b$. Denoting $Ux = y$, we first solve $Ly = b$ via forward substitution in a lower triangular system. After obtaining y , we solve $Ux = y$ for x via back substitution in an upper triangular system.
- ▶ We note that $y = Mb$ is equal to the transformed right hand side of the b vector after Gaussian elimination.
- ▶ Gaussian elimination and LU factorization are two equivalent ways to express the same solution process.

LU algorithm

For solving $Ax = b$ we can apply the LU algorithm:

- ▶ 1. step: Compute LU factorization of A , i.e. $A = LU$.
- ▶ 2. step: Use forward substitution to solve $Ly = b$.
- ▶ 3. step: Use backward substitution to solve $Ux = y$.

The tutorial of this course will look at an implementation of the LU algorithm.

Remember:

$$Ax = b \quad \Leftrightarrow \quad \begin{cases} Ly = b \\ Ux = y \end{cases}$$

LU algorithm with row permutations

We are given the system $Ax = b$ with

$$A = \begin{bmatrix} 0 & 3 & -2 \\ 4 & -2 & 1 \\ 2 & -1 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} -1 \\ -1 \\ 8 \end{bmatrix}.$$

Observe that $a_{11} = 0$, thus the pivot is zero! To compute the LU decomposition, we need to apply a permutation matrix which swaps the first two rows!

Permutation matrices

- ▶ A **permutation matrix** is a matrix with a single 1 per row and column, and zeros everywhere else. That is, a permutation matrix is an identity matrix with permuted rows and columns.
- ▶ Note that this implies $P^{-1} = P^T$.
- ▶ Pre-multiplying both sides of $Ax = b$ with a permutation matrix P , that is $PAx = Pb$, reorders rows but leaves the solution x unchanged.
- ▶ Post-multiplying A with a permutation matrix, that is $APx = b$, reorders the columns of A which permutes the entries of the original solution in x as follows:

$$x = (PA)^{-1}b = P^{-1}A^{-1}b = P^T(A^{-1}b).$$

Example

We first swap the first and third rows:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & 3 & -2 \\ 0 & 1 & 0 & 4 & -2 & 1 \\ 0 & 0 & 1 & 2 & -1 & 1 \end{array} \right] \Rightarrow \left[\begin{array}{ccc|ccc} 0 & 0 & 1 & 2 & -1 & 1 \\ 0 & 1 & 0 & 4 & -2 & 1 \\ 1 & 0 & 0 & 0 & 3 & -2 \end{array} \right].$$

To obtain the corresponding permutation matrix, we simply start with the identity and perform the same swaps on the identity matrix.

We now perform an elimination step from the first row to the rows below, leading to:

$$\left[\begin{array}{ccc|ccc} 0 & 0 & 1 & 2 & -1 & 1 \\ 0 & 1 & 0 & 4 & -2 & 1 \\ 1 & 0 & 0 & 0 & 3 & -2 \end{array} \right] \Rightarrow \left[\begin{array}{ccc|ccc} 0 & 0 & 1 & 2 & -1 & 1 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 3 & -2 \end{array} \right]$$

Example

After the elimination step, we were left with

$$\left[\begin{array}{ccc|ccc} 0 & 0 & 1 & 2 & -1 & 1 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 3 & -2 \end{array} \right].$$

We need to do one more elimination step from row 2. We observe that we again have a pivot zero in a_{22} . We thus swap rows two and three:

$$\left[\begin{array}{ccc|ccc} 0 & 0 & 1 & 2 & -1 & 1 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 3 & -2 \end{array} \right] \Rightarrow \left[\begin{array}{ccc|ccc} 0 & 0 & 1 & 2 & -1 & 1 \\ 1 & 0 & 0 & 0 & 3 & -2 \\ 0 & 1 & 0 & 0 & 0 & -1 \end{array} \right].$$

Since A is now in upper triangular form, we obtained U .

Example

To summarize, we obtained

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -2 \\ 0 & 0 & -1 \end{bmatrix}, \quad P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Using those matrices, we see that

$$LU = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -2 \\ 4 & -2 & 1 \end{bmatrix}, \quad PA = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -2 \\ 4 & -2 & 1 \end{bmatrix}.$$

LU algorithm with row permutations

For solving $Ax = b$ with pivoting after row permutations we can apply the following LU algorithm:

1. step: Compute LU factorization of A such that $PA = LU$.
2. step: Use forward substitution to solve $Ly = Pb$.
3. step: Use backward substitution to solve $Ux = y$.

Homework: Please modify the implementation of the LU algorithm of the tutorial in such a way that it uses row pivoting.

Remember:

$$Ax = b \quad \Leftrightarrow \quad PAx = Pb \quad \Leftrightarrow \quad \begin{cases} Ly = Pb \\ Ux = y \end{cases}$$

LU/ Gauss algorithm with row pivoting

Important: The Gauss/LU algorithm is only a stable algorithm for solving linear equation systems if used with the appropriate pivot strategy!

Algorithmic complexity:

- ▶ LU factorization of an $n \times n$ matrix requires about $n^3/3$ floating point multiplications and a similar number of additions, thus $O(n^3)$ runtime.
- ▶ Forward and backward substitution for a single right hand vector b require $O(n^2)$ multiplications and a similar number of additions.

Solving a system with same matrix but different vectors b

Assume we have computed a decomposition $A = LU$.

- ▶ If the right hand side b of the linear equation system $Ax = b$ changes to b' , but A does not, then we do not need to repeat the LU factorization.
- ▶ Instead, we can immediately repeat the forward and backward substitution for the new b' .
- ▶ This is substantial savings in computational effort, since solving for the new b' costs only $O(n^2)$, whereas computing a new factorization (or applying the classical Gauss algorithm) costs $O(n^3)$.

Stability of LU/Gauss-algorithm

- ▶ Only with the appropriate pivot-strategy the Gauss/ LU algorithm is stable for the solution of linear equation systems.
- ▶ Otherwise we incur a possible **loss of significance!**

Example: Let's consider the system

$$0.035x_1 + 3.62x_2 = 9.12$$

$$1.17x_1 + 1.42x_2 = 5.89$$

with solution $x_1 = 2$ and $x_2 = 2.5$. We assume the machine precision is $(10; 3, 0)$, so 3 digits for multiplications.

Strategy A: no pivoting

We compute the LU decomposition:

$$\begin{bmatrix} \mathbf{0.035} & 3.62 \\ 1.17 & 1.42 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 0.035 & 3.62 \\ 33.4 & -\mathbf{120} \end{bmatrix}.$$

We obtain the decomposition into

$$\hat{L} = \begin{bmatrix} 1 & 0 \\ 33.4 & 1 \end{bmatrix} \quad \hat{U} = \begin{bmatrix} 0.035 & 3.62 \\ 0 & -120 \end{bmatrix}.$$

The entry -120 results as a numerical artifact of the 3 digits we can represent. In particular, letting ρ denote the function mapping onto machine precision,

$$\begin{aligned} \rho(1.42 - \rho(33.4 \cdot 3.62)) &= \rho(1.42 - \rho(120.908)) = \rho(1.42 - 121) \\ &= -120. \end{aligned}$$

Strategy A: no pivoting

On the last slide we obtain the decomposition into

$$\hat{L} = \begin{bmatrix} 1 & 0 \\ 33.4 & 1 \end{bmatrix} \quad \hat{U} = \begin{bmatrix} 0.035 & 3.62 \\ 0 & -120 \end{bmatrix}.$$

Then,

$$\hat{L}\hat{U} = \begin{bmatrix} 0.035 & 3.62 \\ 1.169 & 0.908 \end{bmatrix} \neq A = \begin{bmatrix} 0.035 & 3.62 \\ 1.17 & 1.42 \end{bmatrix}.$$

Strategy A: no pivoting

Let's use forward substitution to solve $Lc = b$, where $b = [9.12, 5.89]^\top$. Using \hat{L} we obtain

$$\hat{c}_1 = 9.12$$

$$\hat{c}_2 = \rho(5.89 - \rho(33.4 \cdot 9.12)) = \rho(5.89 - 305) = -299$$

Backward substitution to solve $Rx = b$. Using \hat{R} we obtain

$$\hat{x}_2 = \rho \left(\frac{-299}{-120} \right) = 2.49$$

$$\hat{x}_1 = \rho \left(\frac{\rho(9.12 - \rho(3.62 \cdot 2.49)))}{0.035} \right) = \rho \underbrace{\left(\frac{9.12 - 9.01}{0.035} \right)}_{\text{loss of significance}} = 3.14.$$

Actual solution is $x_1 = 2$, $x_2 = 2.5$.

Strategy B: Maximum column element as pivot

We again compute the LU decomposition:

$$1.17x_1 + 1.42x_2 = 5.89$$

$$0.035x_1 + 3.62x_2 = 9.12$$

We obtain the decomposition into

$$\hat{L} = \begin{bmatrix} 1 & 0 \\ 0.0299 & 1 \end{bmatrix} \quad \hat{U} = \begin{bmatrix} 1.17 & 1.42 \\ 0 & 3.58 \end{bmatrix}.$$

Then,

$$\hat{L}\hat{U} = \begin{bmatrix} 1.17 & 1.42 \\ 0.035 & 3.62 \end{bmatrix} = A.$$

Strategy B: Maximum column element as pivot

Let's use forward substitution to solve $Lc = b$, where
 $b = [5.89, 9.12]^\top$. Using \hat{L} we obtain

$$\hat{c}_1 = 5.89$$

$$\hat{c}_2 = \rho(9.12 - \rho(0.0299 \cdot 5.89)) = \rho(9.12 - 0.176) = 8.94$$

Backward substitution to solve $Rx = b$. Using \hat{R} we obtain

$$\hat{x}_2 = \rho \left(\frac{8.94}{3.58} \right) = 2.50$$

$$\hat{x}_1 = \rho \left(\frac{\rho(5.89 - \rho(2.50 \cdot 1.42)))}{1.17} \right) = \rho \underbrace{\left(\frac{5.89 - 3.55}{1.17} \right)}_{\text{no loss of significance}} = 2.$$

Actual solution is $x_1 = 2$, $x_2 = 2.5$.

Modified linear equation system

Let's scale equation one by a factor of 100:

$$3.5x_1 + 362x_2 = 912$$

$$1.17x_1 + 1.42x_2 = 5.89$$

Observation: The solution is unchanged, i.e. $x_1 = 2$, $x_2 = 2.5$.

With the previous strategy (no pivoting) we obtain:

$$\hat{L} = \begin{bmatrix} 1 & 0 \\ 0.334 & 1 \end{bmatrix} \quad \hat{U} = \begin{bmatrix} 3.5 & 362 \\ 0 & -120 \end{bmatrix},$$

leading to

$$\hat{c} = \begin{bmatrix} 912 \\ -299 \end{bmatrix} \quad \hat{x} = \begin{bmatrix} 3.14 \\ 2.49 \end{bmatrix}.$$

Strategy C: Scale by relative maximum

We scale each equation with its relative maximum.

Strategy: Compute ratios of coefficients and select the pivot from the equation having the maximal ratio.

$$1.17x_1 + 1.42x_2 = 5.89 \quad q_1 = \frac{1.17}{1.42} = 0.824$$

$$3.5x_1 + 362x_2 = 912 \quad q_2 = \frac{3.5}{362} = 0.00967$$

Since $0.824 > 0.00967$ we select the pivot 1.17.

Strategy C: Scale by relative maximum

Using the pivot according to strategy C we obtain

$$\hat{L} = \begin{bmatrix} 1 & 0 \\ 2.99 & 1 \end{bmatrix} \quad \hat{U} = \begin{bmatrix} 1.17 & 1.42 \\ 0 & 358 \end{bmatrix},$$

leading to

$$\hat{c} = \begin{bmatrix} 5.89 \\ 894 \end{bmatrix} \quad \hat{x} = \begin{bmatrix} 2 \\ 2.5 \end{bmatrix}.$$

Remarks:

- ▶ **The pivot strategy using the relative maximum provides a stable LU algorithm.**
- ▶ The scaling of the equations is actually not necessary. The ratios q_i are only computed to select the pivot.

SPD matrices (symmetric positive definite)

If A is symmetric and positive definite, then the LU factorization can be arranged such that $U = L^\top$, leading to the **Cholesky decomposition**

$$A = LL^\top,$$

where L is a lower diagonal matrix with positive diagonal entries.



André Cholesky
(1875–1918)

Cholesky decomposition

- ▶ An algorithm to compute the Cholesky factorization can be derived by equating the matrix entries of A and LL^\top .
- ▶ For example, in the case of $A \in \mathbb{R}^{2 \times 2}$, we have

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} \\ 0 & l_{22} \end{bmatrix},$$

leading to

$$l_{11} = \sqrt{a_{11}}, \quad l_{21} = a_{21}/l_{11}, \quad l_{22} = \sqrt{a_{22} - l_{21}^2}.$$

- ▶ General formulas for $n \times n$ matrices exist.

Cholesky decomposition

Pseudo-code of the general Cholesky decomposition:

```
for k = 1 to n do
    
$$l_{kk} := \left( a_{kk} - \sum_{j=1}^{k-1} l_{kj}^2 \right)^{1/2}$$

    for i = k + 1 to n do
        
$$l_{ik} := \frac{1}{l_{kk}} \left( a_{ik} - \sum_{j=1}^{k-1} l_{ij} \cdot l_{kj} \right)$$

    end
end
```

Cholesky decomposition

Features of the Cholesky decomposition algorithm for symmetric positive definite matrices:

- ▶ All n square roots are of positive numbers, so the algorithm is well defined.
- ▶ No pivoting is required to maintain numerical stability.
- ▶ Only lower triangle of A is accessed, and hence upper triangular portion need not be stored.
- ▶ Only $n^3/6$ multiplications and similar number of additions are required.

Cholesky decomposition

Advantages of the Cholesky decomposition which follow from the remarks on the previous slide:

- ▶ Cholesky factorization requires only about half the work and half the storage in comparison with the LU factorization of a general matrix by Gaussian elimination, and also avoids the need for pivoting.
- ▶ Homework: Implementation of solution for linear system based on the Cholesky decomposition.

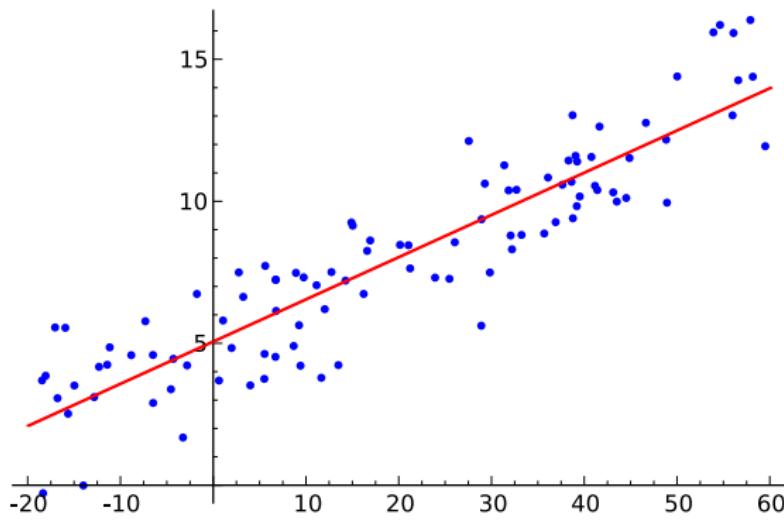
Least squares

Contents:

1. Review of the least squares method
2. QR decomposition (which we have seen)
3. Givens rotations
4. Householder transformations

Linear model

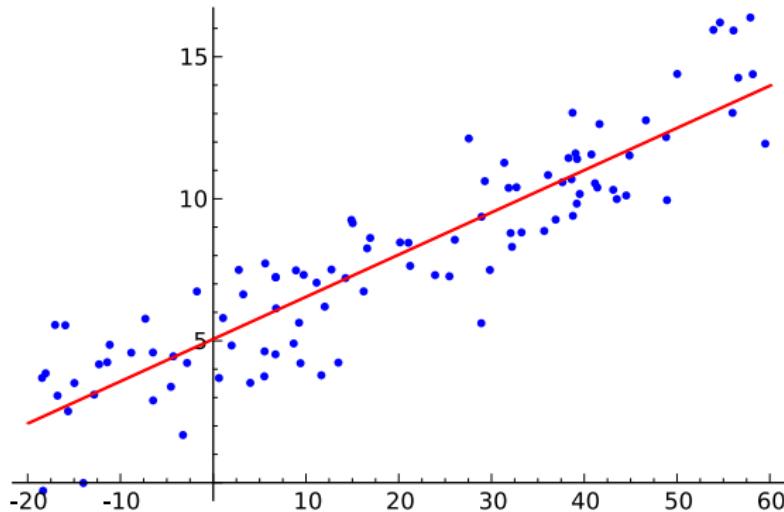
Task: Fit a linear line to given data points.



(source: Wikipedia page "least squares")

Linear model: $y_i = a_1 x_i + a_0$ for given data points (x_i, y_i) ,
 $i \in \{1, \dots, m\}$.

Linear model



The method of **least squares** minimizes the quadratic distance to the fitted line:

$$\min_{a_0, a_1} \sum_{i=1}^m [y_i - (a_1 x_i + a_0)]^2$$

Question: Why not use the distance measured in absolute value?

Data fitting

Least squares can be used more generally to fit any functional form $y_i = f_a(x_i)$ to data points (x_i, y_i) , $i \in \{1, \dots, m\}$, where a is a set of parameters.

Examples:

- ▶ Polynomial fitting

$$f_a(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

is a linear problem since the data points x_i are given, and only $a = [a_0, a_1, \dots]$ are unknown.

- ▶ Fitting the following sum of exponentials

$$f_a(t) = a_1e^{a_2t} + \cdots + a_{n-1}e^{a_nt}$$

is a nonlinear problem in $a = [a_0, a_1, \dots]$.

- ▶ For now we will only consider linear least squares problems.

Example of a polynomial fit

We want to fit a polynomial of degree 2 to data points with the least squares method. Assume $f(x) = a_0 + a_1x + a_2x^2$, then

$$Ax = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \cong \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = b,$$

where $\{(x_1, y_1), \dots, (x_5, y_5)\}$ are given.

The columns (or rows) of the matrix A are successive powers of independent variables. Such a matrix is called **Vandermonde matrix**.

Example of a polynomial fit

Assume we are given the data points

x	-1.0	-0.5	0.0	0.5	1.0
y	1.0	0.5	0.0	0.5	2.0

Substituting into the matrix from the previous slide leads to the *overdetermined* 5×3 system

$$Ax = \begin{bmatrix} 1 & -1.0 & 1.0 \\ 1 & -0.5 & 0.25 \\ 1 & 0.0 & 0.0 \\ 1 & 0.5 & 0.25 \\ 1 & 1.0 & 1.0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \cong \begin{bmatrix} 1.0 \\ 0.5 \\ 0.0 \\ 0.5 \\ 2.0 \end{bmatrix} = b$$

with solution (which we will see later how to compute),

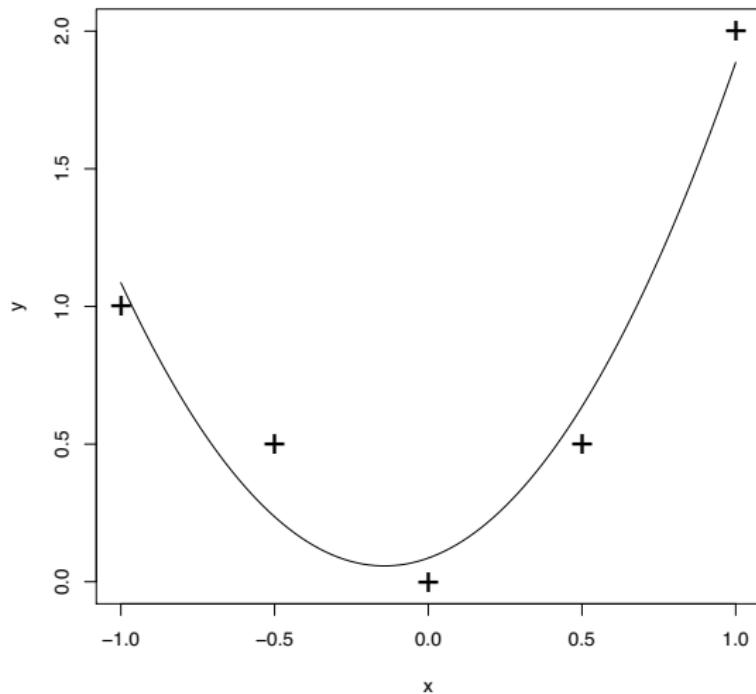
$$a = [0.086, 0.4, 1.4]^\top$$

leading to the fitted polynomial

$$f(x) = 0.086 + 0.4x + 1.4x^2.$$

Example of a polynomial fit

The data points and fitted polynomial are:



Least squares

- ▶ Assume we have m observations and n covariates.
- ▶ Since we fit a linear function to the data, we can express the problem as finding x such that

$$Ax = b,$$

where with a change of notation,

1. $A \in \mathbb{R}^{m \times n}$ is a matrix containing the data,
2. $b \in \mathbb{R}^m$ is the response (formerly the y -values),
3. x contains the n unknowns.

- ▶ Typically, $m > n$ thus the system is *overdetermined*.
- ▶ We write $Ax \cong b$ since typically the system is not solvable exactly.

Existence and uniqueness of solutions

- ▶ Since the system $Ax \cong b$ is overdetermined we solve it in the Euclidean distance,

$$\min_x \| \underbrace{Ax - b}_ {=: r(x)} \|_2 = \min_x \|r(x)\|_2.$$

- ▶ The least squares problem $Ax \cong b$ always has a solution.
- ▶ The solution is unique if and only if the columns of A are *linearly independent*, i.e. $\text{rank}(A) = n$ where n is the number of unknowns (thus $A \in \mathbb{R}^{m \times n}$).
- ▶ If $\text{rank}(A) < n$, then A is *rank deficient* and the solution of the linear least squares problem is not unique.
- ▶ We assume A has full column rank n .

Least squares has a closed-form solution

- ▶ The solution to $Ax \cong b$ in the metric

$$\min_x \|Ax - b\|_2 = \min_x \|r(x)\|_2$$

has a closed-form solution which will be derived now.

- ▶ For this, consider the Euclidean norm of the residual vector:

$$\begin{aligned}\|r\|_2^2 &= r^\top r \\ &= (b - Ax)^\top (b - Ax) \\ &= b^\top b - 2x^\top A^\top b + x^\top A^\top A x.\end{aligned}$$

Least squares has a closed-form solution

- ▶ We have seen that

$$\|r\|_2^2 = b^\top b - 2x^\top A^\top b + x^\top A^\top Ax.$$

- ▶ We want to minimize the residual. Taking the derivative with respect to the unknowns x and setting it to zero yields

$$2A^\top Ax - 2A^\top b = 0$$

which reduces to an $n \times n$ **system of normal equations**

$$A^\top Ax = A^\top b.$$

Least squares has a closed-form solution

We have seen that the least squares problem reduces to an $n \times n$ system of normal equations

$$A^T A x = A^T b,$$

for which, in fact, a closed-form solution exists.

In particular, if A has linearly independent columns, then $A^T A$ is invertible. The matrix

$$A^+ = (A^T A)^{-1} A^T$$

is called the **Moore-Penrose** inverse for a real-valued matrix A . We obtain the **closed-form solution**

$$x = (A^T A)^{-1} A^T b = A^+ b.$$

Example of closed form solution

Let's reconsider the polynomial fit from earlier with

$$A = \begin{bmatrix} 1 & -1.0 & 1.0 \\ 1 & -0.5 & 0.25 \\ 1 & 0.0 & 0.0 \\ 1 & 0.5 & 0.25 \\ 1 & 1.0 & 1.0 \end{bmatrix}, \quad b = \begin{bmatrix} 1.0 \\ 0.5 \\ 0.0 \\ 0.5 \\ 2.0 \end{bmatrix}.$$

We compute

$$A^T A = \begin{bmatrix} 5.0 & 0.0 & 2.5 \\ 0.0 & 2.5 & 0.0 \\ 2.5 & 0.0 & 2.125 \end{bmatrix}, \quad A^T b = \begin{bmatrix} 4.0 \\ 1.0 \\ 3.25 \end{bmatrix}.$$

Since $A^T A$ is invertible, we obtain

$$x = (A^T A)^{-1} A^T b = [0.086, 0.4, 1.4]^\top.$$

Problem: Computing the inverse is expensive!

Solution via Cholesky factorization

The system $Ax \cong b$ can also be solved in the following way.

- ▶ If the $m \times n$ matrix A has full rank n , then $A^\top A$ is positive definite and thus we can compute its Cholesky factorization

$$A^\top A = LL^\top.$$

- ▶ Looking at the normal equations

$$A^\top Ax = A^\top b$$

we see that the right hand side is known, thus we can use the Cholesky factorization to efficiently compute the solution of

$$LL^\top x = (A^\top b)$$

which will have the same solution as the linear least squares problem $Ax \cong b$.

Solution via Cholesky factorization

- ▶ We started with the rectangular (overdetermined) system $Ax = b$, where $A \in \mathbb{R}^{m \times n}$.
- ▶ Using the normal equations we transformed to the system $A^T A x = A^T b$ having the same solution space, where $A^T A \in \mathbb{R}^{n \times n}$.
- ▶ Using the Cholesky factorization, we can reduce the problem to solving two triangular systems.

Picture:

rectangular → square → triangular

Example of the Cholesky approach

Let's reconsider the polynomial fit from earlier with

$$A = \begin{bmatrix} 1 & -1.0 & 1.0 \\ 1 & -0.5 & 0.25 \\ 1 & 0.0 & 0.0 \\ 1 & 0.5 & 0.25 \\ 1 & 1.0 & 1.0 \end{bmatrix}, \quad b = \begin{bmatrix} 1.0 \\ 0.5 \\ 0.0 \\ 0.5 \\ 2.0 \end{bmatrix}.$$

We have already computed

$$A^T A = \begin{bmatrix} 5.0 & 0.0 & 2.5 \\ 0.0 & 2.5 & 0.0 \\ 2.5 & 0.0 & 2.125 \end{bmatrix}, \quad A^T b = \begin{bmatrix} 4.0 \\ 1.0 \\ 3.25 \end{bmatrix}.$$

Example of the Cholesky approach

Since $A^T A$ is symmetric and positive definite, the Cholesky factorization of it can be computed:

$$A^T A = LL^T = \begin{bmatrix} 2.236 & 0 & 0 \\ 0 & 1.581 & 0 \\ 1.118 & 0 & 0.935 \end{bmatrix} \begin{bmatrix} 2.236 & 0 & 1.118 \\ 0 & 1.581 & 0 \\ 0 & 0 & 0.935 \end{bmatrix}.$$

- ▶ Solving the lower triangular system $Lz = A^T b$ by forward substitution yields $z = [1.789, 0.632, 1.336]^\top$.
- ▶ Solving the upper triangular system $L^T x = z$ by backward substitution yields $x = [0.086, 0.400, 1.429]^\top$.

Numerical issues

The transformation step which yields the normal equations has numerical disadvantages:

- ▶ Information can be lost when computing $A^\top A$.
- ▶ For example, consider

$$A = \begin{bmatrix} 1 & 1 \\ \epsilon & 0 \\ 0 & \epsilon \end{bmatrix},$$

where ϵ is a positive number smaller than $\sqrt{\text{eps}}$ (square root of the machine precision).

- ▶ Then, in floating point arithmetic we obtain the singular matrix

$$A^\top A = \begin{bmatrix} 1 + \epsilon^2 & 1 \\ 1 & 1 + \epsilon^2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

Numerical issues and computational complexity

- ▶ Computing $A^\top A$ takes $O(m \cdot n^2)$.
- ▶ The condition number has the property that

$$\text{cond}(A^\top A) = \text{cond}(A)^2,$$

thus the sensitivity of the normal equations is much worse than for the original system. Since $\text{cond}(A) > 1$ this is a major issue.

- ▶ Recall the definition of the *relative condition number*

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|.$$

Numerical issues and computational complexity

- ▶ One can show that $\kappa(A) = \|A\| \cdot \|A^{-1}\|$ can be written as

$$\kappa(A) = \frac{\max_{\|x\|=1} \|Ax\|}{\min_{\|x\|=1} \|Ax\|},$$

and thus

$$\kappa(A)^2 = \frac{\max_{\|x\|_2=1} \langle Ax, Ax \rangle}{\min_{\|x\|_2=1} \langle Ax, Ax \rangle} = \frac{\lambda_{\max}(A^\top A)}{\lambda_{\min}(A^\top A)} = \kappa(A^\top A).$$

- ▶ Therefore, if $\kappa(A) > 1$, the relative condition number of $A^\top A$ in the normal equations is a lot worse!
- ▶ Moreover, potential loss of significance when calculating $(A^\top A)_{ij} = \sum_{k=1}^m a_{ki} a_{kj}$ due to cancellation of significant digits.

Example: LR decomposition and condition number

We consider the following matrix and its inverse

$$A = \begin{bmatrix} 0.001 & 0.001 \\ 1 & 2 \end{bmatrix}, \quad A^{-1} = \begin{bmatrix} 2000 & -1 \\ -1000 & 1 \end{bmatrix}.$$

We compute the LR decomposition as

$$L = \begin{bmatrix} 1 & 0 \\ 1000 & 1 \end{bmatrix}, \quad R = \begin{bmatrix} 0.001 & 0.001 \\ 0 & 1 \end{bmatrix}.$$

The inverse matrices are

$$L^{-1} = \begin{bmatrix} 1 & 0 \\ -1000 & 1 \end{bmatrix}, \quad R^{-1} = \begin{bmatrix} 1000 & -1 \\ 0 & 1 \end{bmatrix}.$$

The condition numbers are

$$\text{cond}(A) \approx 6000, \quad \text{cond}(L) \approx 10^6,$$

so the condition number of L is about 166 times larger than the one of the original matrix A .

Alternative: QR decomposition

- ▶ We have seen that the normal equations may cause numerical issues.
- ▶ Therefore, we seek an alternative way avoiding those difficulties.
- ▶ Idea: Robust transformation which results in an easier problem while preserving the solution.

Alternative: QR decomposition

What type of transformation leaves the least squares solution unchanged?

- ▶ A square matrix is orthogonal if $Q^\top Q = I$.
- ▶ Multiplying a vector with an orthogonal matrix preserves its Euclidean norm as

$$\|Qv\|_2^2 = (Qv)^\top Qv = v^\top Q^\top Qv = v^\top v = \|v\|_2^2.$$

- ▶ Therefore, multiplying both sides of $Ax = b$ does not change its solution.

Details on the QR decomposition

- ▶ Let's fill out the details of this approach.
- ▶ Reminder: We are given $A \in \mathbb{R}^{m \times n}$ with $m > n$ for the problem $Ax = b$.
- ▶ We seek an orthogonal matrix $Q \in \mathbb{R}^{m \times m}$ such that

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix},$$

where $R \in \mathbb{R}^{n \times n}$ is upper triangular.

Why?

Details on the QR decomposition

- ▶ Assume we have found an orthogonal matrix $Q \in \mathbb{R}^{m \times m}$ and an upper triangular $R \in \mathbb{R}^{n \times n}$ such that

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}.$$

- ▶ In this case, $Ax \cong b$ is transformed into a *triangular least squares problem*

$$Q^\top Ax = \begin{bmatrix} R \\ 0 \end{bmatrix} x \cong \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = Q^\top b.$$

- ▶ $Q^\top Ax \cong Q^\top b$ has the same solution as $Ax \cong b$ since

$$\|r\|_2^2 = \|b - Ax\|_2^2 = \left\| b - Q \begin{bmatrix} R \\ 0 \end{bmatrix} x \right\|_2^2 = \left\| Q^\top b - \begin{bmatrix} R \\ 0 \end{bmatrix} x \right\|_2^2.$$

QR decomposition and least squares

Let's go back to the least squares minimization problem $Ax = b$.
For this, write

$$Q^\top b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad \begin{array}{l} \} \quad n \text{ entries,} \\ \} \quad m - n \text{ entries.} \end{array}$$

Then, resuming with the result from the last slide, we obtain

$$\begin{aligned} \|Ax - b\|_2^2 &= \left\| \begin{bmatrix} R \\ 0 \end{bmatrix} x - Q^\top b \right\|_2^2 = \left\| \begin{bmatrix} R \\ 0 \end{bmatrix} x - \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \right\|_2^2 \\ &= \|Rx - b_1\|_2^2 + \|b_2\|_2^2. \end{aligned}$$

QR decomposition and least squares

We have seen that

$$\|Ax - b\|_2^2 = \|Rx - b_1\|_2^2 + \|b_2\|_2^2.$$

We observe that $\|b_2\|_2^2$ does not depend on x . Moreover, the minimum of $\|Rx - b_1\|_2^2$ is zero, and it is achieved for $Rx = b_1$, thus

$$\min_x \|Ax - b_1\|_2 \quad \Leftrightarrow \quad \text{solution of } Rx = b_1.$$

Algorithm:

1. Compute QR decomposition $A = QR$.
2. Use backward substitution to solve $Rx = b_1$ with $Q^\top b = [b_1, b_2]^\top$ as defined on the previous slide.

Numerical considerations for QR method

Recall that $A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$. Then,

$$\|Ax\|_2 = \left\| Q \begin{bmatrix} R \\ 0 \end{bmatrix} x \right\|_2 = \|Rx\|_2,$$

using the fact that multiplication with the orthogonal Q does not change the Euclidean norm. Therefore, we have

$$\kappa_2(A) = \kappa_2(R).$$

Remarks:

- ▶ The advantage of the algorithm on the previous slide is that the condition number of the problem remains unchanged.
- ▶ An implementation of the LU algorithm in C++ will be shown in the tutorials.

Computing the QR decomposition

- ▶ To compute the QR decomposition of $A \in \mathbb{R}^{m \times n}$ with $m > n$, we eliminate subdiagonal entries of successive columns of A , eventually reaching upper triangular form.
- ▶ This can be done similarly to the Gauss algorithm, but using orthogonal transformations instead of elementary elimination matrices.
- ▶ Possible methods include:
 1. **Gives rotations**
 2. **Householder transformations**

We will now look at both methods...

Givens rotations

Consider the following matrix

$$A_1 = \begin{bmatrix} 6 & 5 & 0 \\ 5 & 1 & 4 \\ 0 & 4 & 3 \end{bmatrix}.$$

Idea of Givens rotations: Can we use rotations of the coordinate system to introduce zeros below the diagonal, eventually reaching upper triangular form?

Answer: Yes, with a technique called Givens rotations (James Givens, 1910–1993)

Theory of Givens rotation

A Givens rotation is represented by a matrix of the form

$$G(i,j,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix},$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$.

In summary, the nonzero elements of the Givens rotation matrix $G(i,j,\theta)$ are

$$g_{kk} = 1 \quad \text{for } k \neq i,j$$

$$g_{ii} = g_{jj} = c$$

$$g_{ji} = -g_{ij} = -s.$$

Theory of Givens rotation

Why *Givens rotation*? → The product $G(i, j, \theta)x$ represents a counterclockwise rotation of the vector x in the (i, j) plane of θ radians, hence the name Givens rotation.

We looked at the matrix

$$A_1 = \begin{bmatrix} 6 & 5 & 0 \\ 5 & 1 & 4 \\ 0 & 4 & 3 \end{bmatrix}$$

and we want to clear first the element "5". Since we do not mind what happens with the other elements in the first two rows it suffices to look at the 2-vector [6, 5]. *The following example is taken from Wikipedia, page "Givens rotation".*

Givens rotation for a 2-vector

We have seen it suffices to consider the rotation of a 2-vector $[a, b]^\top$:

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix},$$

where $r = \sqrt{a^2 + b^2}$ is the *length* of $[a, b]^\top$.

Defining

$$c = \frac{a}{r}$$

$$s = -\frac{b}{r}$$

we see immediately that $[a, b]^\top$ can be rotated to introduce a zero.

If the matrix under consideration is bigger than 2×2 , we still consider two rows to transform, but fill out the rest of the Givens rotation matrix using the general form quoted in the previous slides. Let's look at an example...

Example of Givens rotation

Looking again at

$$A_1 = \begin{bmatrix} 6 & 5 & 0 \\ 5 & 1 & 4 \\ 0 & 4 & 3 \end{bmatrix}$$

we see that we want to introduce a zero where "5" is.

We compute the Givens rotation $\begin{bmatrix} c & -s \\ s & c \end{bmatrix}$ from the previous slides and apply it to the vector $\begin{bmatrix} 6 \\ 5 \end{bmatrix}$.

Using the general form of the Givens rotation, we obtain the rotation matrix

$$G_1 = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

with $r = \sqrt{6^2 + 5^2} \approx 7.81$, $c = 6/r \approx 0.77$, $s = -5/r \approx -0.64$.

Example of Givens rotation

Applying

$$G_1 = \begin{bmatrix} 0.77 & 0.64 & 0 \\ -0.64 & 0.77 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

to the matrix

$$A_1 = \begin{bmatrix} 6 & 5 & 0 \\ 5 & 1 & 4 \\ 0 & 4 & 3 \end{bmatrix}$$

results in

$$A_2 = G_1 A_1 = \begin{bmatrix} 7.81 & 4.48 & 2.56 \\ 0 & -2.43 & 3.07 \\ 0 & 4 & 3 \end{bmatrix}.$$

Example of Givens rotation

Since the matrix

$$A_2 = G_1 A_1 = \begin{bmatrix} 7.81 & 4.48 & 2.56 \\ 0 & -2.43 & 3.07 \\ 0 & 4 & 3 \end{bmatrix}$$

is not yet in upper triangular form we apply another Givens rotation to the second and third rows, that is to the submatrix starting with the column $[-2.43, 4]^\top$, leading to

$$G_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix},$$

where $r = \sqrt{(-2.43)^2 + 4^2} = 4.68$, $c = -2.43/r \approx -0.52$,
 $s = -4/r \approx -0.85$.

Example of Givens rotation

Using A_2 and G_2 from the previous slide, we compute

$$A_3 = G_2 A_2 = \begin{bmatrix} 7.81 & 4.48 & 2.56 \\ 0 & 4.68 & 0.97 \\ 0 & 0 & -4.18 \end{bmatrix}.$$

The new matrix A_3 is the upper triangular matrix needed to perform an iteration of the QR decomposition.

Example of Givens rotation

As done with elimination matrices, rotation matrices can be multiplied to yield one rotation matrix performing all steps combined. That is,

$$G = G_2 G_1$$

will have the property that

$$A_3 = GA_1 = \begin{bmatrix} 7.81 & 4.48 & 2.56 \\ 0 & 4.68 & 0.97 \\ 0 & 0 & -4.18 \end{bmatrix}.$$

Givens rotation: final remarks

Final remarks on implementation and computational complexity:

- ▶ The number (volume) of operations is $\propto \frac{1}{2}mn^2 - \frac{1}{6}n^3$ for an $(m \times n)$ -matrix.
- ▶ Givens rotations need 6 flops for computing the Givens matrix entries (in contrast to the Householder reflections on the next slides), thus leading to an effort $\propto 3mn^2 - n^3$. This is 50% more expensive than Householder transformations.
- ▶ A straightforward implementation of Givens rotations requires about 50% more work than Householder method, and also requires more storage, since each rotation requires two numbers (c and s) to define it.
- ▶ These disadvantages can be overcome, but that requires more complicated implementations.
- ▶ Givens rotations can be advantageous for computing the QR factorization when many entries of matrix are already zero, since those eliminations can then be skipped.

Householder transformations

Consider the matrix

$$A = \begin{bmatrix} \mathbf{2} & -2 & 18 \\ \mathbf{2} & 1 & 0 \\ \mathbf{1} & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix}$$

Idea: Let's consider the first column of A (bold) which we call a . The idea of the Householder transformation method is to construct a matrix H such that $Ha = \alpha e_1$ for some α . Here, e_1 is the first unit vector. That is, we eliminate all but the first entry in one go!

Theory of Householder transformations

Consider the matrix

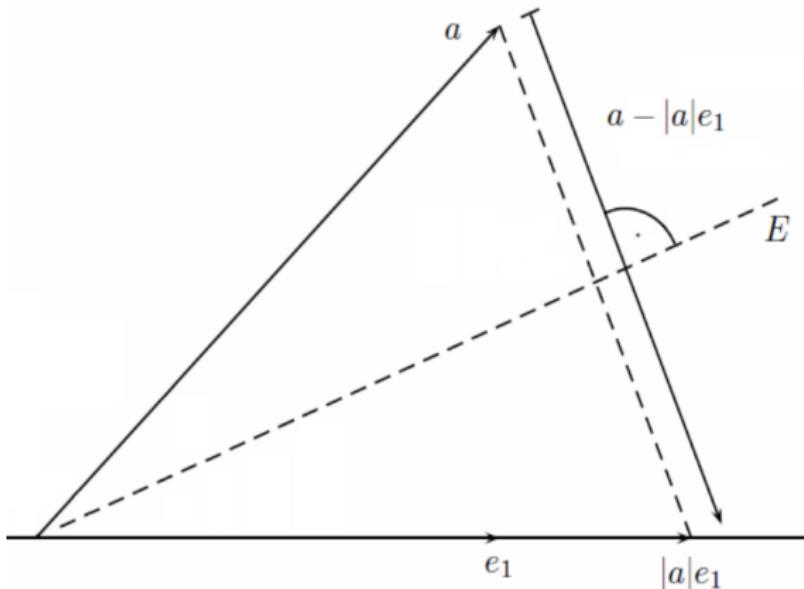
$$A = \begin{bmatrix} 2 & -2 & 18 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix}.$$

In the first step we seek an orthogonal matrix H such that $Ha = \alpha e_1$, where a is the first column of A (bold).

Since H is orthogonal, it leaves the Euclidean norm unchanged and thus $\|Ha\| = \|a\|$. Moreover, $\|\alpha e_1\| = |\alpha| \|e_1\| = |\alpha|$. From this we see that $|\alpha| = \|a\|$ and thus $\alpha = \pm \|a\|$.

Question: What is the form of a matrix H such that $Ha = \alpha e_1$?

Theory of Householder transformation



We need to construct H such that $Ha = \alpha e_1$. The picture shows a general vector a which is projected to $\|a\|e_1$. Let's define $u := a - \|a\|e_1$ and $v := u/\|u\|$.

Theory of Householder transformation

Let H be an orthogonal matrix. We have seen that $Ha = \alpha e_1$ implies $\alpha = \pm \|a\|$. Without loss of generality assume $\alpha = +\|a\|$.

Using $u := a - \|a\|e_1$ and $v := u/\|u\|$, we obtain

$$\begin{aligned} Ha &= \|a\|e_1 \\ &= a - (a - \|a\|e_1) \\ &= a - 2 \frac{(a + \|a\|e_1)(a^\top a + \|a\|a^\top e_1)}{\|a\|^2 + 2a^\top e_1\|a\| + \|a\|^2\|e_1\|^2} \\ &= (I - 2vv^\top)a. \end{aligned}$$

Therefore,

$$H = I - 2vv^\top.$$

Remark: The sign of α is chosen so that α has the opposite sign of a_1 (first entry of vector a) in order to avoid cancellation.

Properties of H

H is a symmetric and orthogonal projection matrix, that is $H^2 = I$.

Proof: Using the fact that v is normed, we obtain

$$H^2 = (I - 2vv^\top) \cdot (I - 2vv^\top) = I - 4vv^\top + 4v(v^\top v)v^\top = I.$$

Example: Householder transformation

We again consider the matrix

$$A = \begin{bmatrix} 2 & -2 & 18 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix}.$$

Define $a = [2, 2, 1]^\top$, we find the reflection v_1 as

$$u_1 = a + \|a\|e_1 = \begin{bmatrix} 5 \\ 2 \\ 1 \end{bmatrix},$$
$$v_1 = u_1 / \|u_1\|.$$

Remark: The vector $u_1 = a - \|a\|e_1$ equally works since in both cases there won't be a cancellation in the first entry.

Example: Householder transformation

On the last slide we had

$$u_1 = a + \|a\|e_1 = \begin{bmatrix} 5 \\ 2 \\ 1 \end{bmatrix}, \quad v_1 = u_1 / \|u_1\|,$$

and thus

$$\begin{aligned} H_1 &= I - 2v_1 v_1^\top = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - 2 \frac{1}{\|u_1\|^2} \begin{bmatrix} 5 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 5 & 2 & 1 \end{bmatrix} \\ &= \begin{bmatrix} -2/3 & -2/3 & -1/3 \\ -2/3 & 0.7333 & -0.1333 \\ -1/3 & -0.1333 & 0.9333 \end{bmatrix}. \end{aligned}$$

Indeed,

$$H_1 A = \begin{bmatrix} -3 & 0 & -12 \\ 0 & 1.8 & 12 \\ 0 & 2.4 & -6 \end{bmatrix}.$$

Example: Householder transformation

We have seen

$$H_1 A = \begin{bmatrix} -3 & 0 & -12 \\ 0 & \mathbf{1.8} & \mathbf{12} \\ 0 & 2.4 & -6 \end{bmatrix}.$$

We apply another Householder transformation to the submatrix

$$A_2 = \begin{bmatrix} \mathbf{1.8} & 12 \\ 2.4 & -6 \end{bmatrix}. \text{ Now we have}$$

$$u_2 = \begin{bmatrix} 1.8 \\ 2.4 \end{bmatrix} + \sqrt{1.8^2 + 2.4^2} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 4.8 \\ 2.4 \end{bmatrix},$$

$$v_2 = u_2 / \|u_2\|.$$

Example: Householder transformation

We have calculated $u_2 = [4.8, 2.4]^\top$ and $v_2 = u_2 / \|u_2\|$. As before,

$$I - 2v_2 v_2^\top = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - 2 \frac{1}{\|u_2\|^2} \begin{bmatrix} 4.8 \\ 2.4 \end{bmatrix} \begin{bmatrix} 4.8 & 2.4 \end{bmatrix} = \begin{bmatrix} -0.6 & -0.8 \\ -0.8 & 0.6 \end{bmatrix}.$$

This leads to the second Householder transformation

$$H_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.6 & -0.8 \\ 0 & -0.8 & 0.6 \end{bmatrix}.$$

Indeed,

$$H_2 H_1 A = H_2 \cdot \begin{bmatrix} -3 & 0 & -12 \\ 0 & 1.8 & 12 \\ 0 & 2.4 & -6 \end{bmatrix} = \begin{bmatrix} -3 & 0 & -12 \\ 0 & -3 & 12 \\ 0 & 0 & 6 \end{bmatrix}.$$

Householder and QR decomposition

How do we get the QR decomposition from the Householder matrices?

We have seen that

$$H_2 H_1 A = H_2 \cdot \begin{bmatrix} -3 & 0 & -12 \\ 0 & 1.8 & 12 \\ 0 & 2.4 & -6 \end{bmatrix} = \begin{bmatrix} -3 & 0 & -12 \\ 0 & -3 & 12 \\ 0 & 0 & 6 \end{bmatrix} =: \mathbf{R}.$$

At the same time, since all H_i are orthonormal matrices, they all satisfy $H_i^{-1} = H_i^\top$, and thus

$$H_2 H_1 \mathbf{A} = \mathbf{R} \quad \Rightarrow \quad \mathbf{A} = H_1^{-1} H_2^{-1} \mathbf{R} = \underbrace{H_1^\top H_2^\top}_{=: \mathbf{Q}} \mathbf{R} = \mathbf{Q} \mathbf{R}.$$

Remarks on the Householder transformation

- ▶ To compute the QR factorization of A we use the Householder transformations to eliminate the subdiagonal entries of each successive column.
- ▶ Each Householder transformation is applied to the entire matrix, but it does not affect prior columns, thus preserving existing zeros.
- ▶ When applying the Householder transformation to an arbitrary vector a we actually do not need to compute a matrix-vector multiplication:

$$Ha = (I - 2vv^\top)a = a - 2(vv^\top)a.$$

Thus the Householder transformation is much cheaper than a general matrix–vector multiplication and it requires only the computation of v , not the full matrix H .

Remarks on the Householder transformation

- As we have seen in the example, the process just described produces a factorization

$$H_n \cdots H_1 A = \begin{bmatrix} R \\ 0 \end{bmatrix},$$

where R is an upper triangular $(n \times n)$ -matrix.

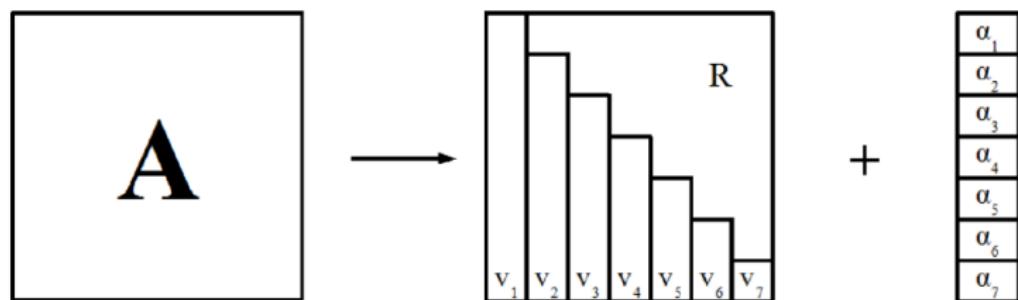
- If $Q = H_1^\top \cdots H_n^\top$ then $A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$.
- When applying this to the least squares problem $Ax = b$, the sequence of Householder transformations is also applied to the right hand side b .
- We then solve the triangular least squares problem

$$\begin{bmatrix} R \\ 0 \end{bmatrix} x \cong Qb.$$

Remarks on the Householder transformation

- ▶ For solving the least squares problem, the sequence of Householder transformations need not be computed explicitly.
- ▶ R can be stored in upper triangular array which initially contains A .
- ▶ The Householder vectors v_i can be stored in the lower triangular portion of the matrix A .

Picture:



Example of fitting a polynomial

Back to the initial example of fitting a polynomial. We had

$$A = \begin{bmatrix} 1 & -1.0 & 1.0 \\ 1 & -0.5 & 0.25 \\ 1 & 0.0 & 0.0 \\ 1 & 0.5 & 0.25 \\ 1 & 1.0 & 1.0 \end{bmatrix}, \quad b = \begin{bmatrix} 1.0 \\ 0.5 \\ 0.0 \\ 0.5 \\ 2.0 \end{bmatrix}.$$

The first Householder vector is

$$v_1 = \begin{bmatrix} 3.236 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Example of fitting a polynomial

The first Householder vector

$$v_1 = \begin{bmatrix} 3.236 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

leads to the first Householder transformation

$$H_1 A = \begin{bmatrix} -2.236 & 0 & -1.118 \\ 0 & -0.191 & -0.405 \\ 0 & 0.309 & -0.655 \\ 0 & 0.809 & -0.405 \\ 0 & 1.309 & 0.345 \end{bmatrix}, \quad H_1 b = \begin{bmatrix} -1.789 \\ -0.362 \\ -0.862 \\ -0.362 \\ 1.138 \end{bmatrix}.$$

Example of fitting a polynomial

The second Householder vector is then

$$v_2 = \begin{bmatrix} 0 \\ -1.772 \\ 0.309 \\ 0.809 \\ 1.309 \end{bmatrix},$$

leading to the second Householder transformation

$$H_2 H_1 A = \begin{bmatrix} -2.236 & 0 & -1.118 \\ 0 & 1.581 & 0 \\ 0 & 0 & -0.725 \\ 0 & 0 & -0.589 \\ 0 & 0 & 0.047 \end{bmatrix}, \quad H_2 H_1 b = \begin{bmatrix} -1.789 \\ 0.632 \\ -1.035 \\ -0.816 \\ 0.404 \end{bmatrix}.$$

Example of fitting a polynomial

The third Householder vector is then

$$v_3 = \begin{bmatrix} 0 \\ 0 \\ -1.660 \\ -0.589 \\ 0.047 \end{bmatrix},$$

leading to the third Householder transformation

$$H_3 H_2 H_1 A = \begin{bmatrix} -2.236 & 0 & -1.118 \\ 0 & 1.581 & 0 \\ 0 & 0 & 0.935 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad H_3 H_2 H_1 b = \begin{bmatrix} -1.789 \\ 0.632 \\ 1.336 \\ 0.026 \\ 0.337 \end{bmatrix}.$$

Example of fitting a polynomial

Using

$$H_3 H_2 H_1 A = \begin{bmatrix} -2.236 & 0 & -1.118 \\ 0 & 1.581 & 0 \\ 0 & 0 & 0.935 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = R$$

and

$$H_3 H_2 H_1 b = [-1.789, 0.632, 1.336, 0.026, 0.337]^\top = c_1$$

we can now solve the upper triangular system $Rx = c$ by back substitution to obtain the solution (the coefficients of the fitted polynomial)

$$x = [0.086, 0.400, 1.429]^\top.$$

Comparison of methods

- ▶ Just like Givens rotations, the number (volume) of operations is $\propto \frac{1}{2}mn^2 - \frac{1}{6}n^3$ for an $(m \times n)$ -matrix. However, Householder reflections require 4 flops per matrix entry, leading to a total effort $\propto 2mn^2 - \frac{2}{3}n^3$ or 50% less than Givens rotations.
- ▶ If $m \gg n$, that is if there are more data points m than unknowns n , then the Householder QR decomposition requires about twice as much effort as normal equations.

Advantages of the Householder QR decomposition

- ▶ Householder QR is more accurate and more broadly applicable than normal equations.
- ▶ However, these advantages may not be worth the additional cost if the problem is sufficiently well conditioned that the normal equations are sufficient for solving it.

Eigenvalues

Contents:

- ▶ Eigenvalues

The Eigenvalue problem is encountered in many applications/problems in biostats, bioinfo, big-data, etc., such as:

- ▶ Principal component analysis
- ▶ Multivariate analysis
- ▶ Genetics
- ▶ Imagine processing/recognition
- ▶ PageRank algorithm by Google
- ▶ Language processing algorithms
- ▶ Stability control in physics
- ▶ etc.

Example: Population substructure in genetics

Different populations usually have different allele frequency distributions.

Unknown population substructure can be the cause of confounding/ false positive/ false negative findings.

Consider the following example: There are two populations with different allele frequency distributions and different disease prevalence for the disease of interest in your study.

→ A genetic association analysis is then more likely to find the genetic loci for which the allele frequency distributions considerably differs from the disease loci, if we do not adjust for it.

Example: Population substructure in genetics (cont.)

Idea: If we have a set of 10000+ loci that are distributed across the genome and are uncorrelated, we can compute the genetic variance/covariance matrix for the study subjects. We can then compute the eigenvalues and eigenvectors and run a principal component analysis.

If we have no population substructure in our sample, we would expect one homogeneous cloud of points in the principal component plots.

If we have different sub-populations in our sample/study, we would hope to be able to identify the sub-populations in the principal component plots.

The Eigenvalue Problem

Given the matrix $A \in \mathbb{C}^{n \times n}$, $n \in \mathbb{N}$, we want to find a solution of

$$Av = \lambda v$$

where $v \in \mathbb{C}^n$ and $\lambda \in \mathbb{C}$. The scalar λ is called the **eigenvalue** of A associated with the **eigenvector** v . The set of all eigenvalues of A is called the **spectrum** of A .

Frequently we will work with eigenvectors with unit norm. This is always possible, since for any given eigenvector v of A , the vector

$$\hat{v} = v / \|v\|$$

is also an eigenvector of A associated with the same eigenvalue. This normalization process can be done for any vector norm.

The Eigenvalue Problem

- ▶ Here, we will mainly focus on the special case where $A \in \mathbb{R}^{n \times n}$ and A is symmetric ($A = A^\top$).
- ▶ All eigenvalues of a symmetric matrix A are real, that is $\lambda_1, \dots, \lambda_n \in \mathbb{R}$. Furthermore, there is an orthogonal matrix Q , i.e. $Q^\top Q = I_{n,n} = \text{diag}(1, \dots, 1) \in \mathbb{R}^{n \times n}$, such that

$$Q^\top A Q = \text{diag}(\lambda_1, \dots, \lambda_n).$$

The Eigenvalue Problem

- ▶ How can we obtain the eigenvalues of a matrix $A \in \mathbb{R}^{n \times n}$?
- ▶ One can show that

$$Av = \lambda v \iff \chi_A(\lambda) = \det(\lambda I_{n,n} - A) = 0.$$

- ▶ We see that

$$Av = \lambda v \iff (A - \lambda I_{n,n})v = 0,$$

thus the eigenvectors lay in the null space of $A - \lambda I_{n,n}$.

- ▶ Instead of solving the eigen-equation directly, we could use the characteristic polynomial of χ_A and find its roots.

The Eigenvalue Problem: Example

Assume that $A = \text{diag}(1, 2, 3, \dots, 20) \in \mathbb{R}^{n \times n}$. Then

$$\begin{aligned}\chi_A(\lambda) &= \det(\lambda I_{n,n} - A) \\ &= (\lambda - 1) * (\lambda - 2) * \dots * (\lambda - 19) * (\lambda - 20) \\ &= \lambda^{20} - 210\lambda^{19} + \dots\end{aligned}$$

How do the roots of this polynomial change if we introduce a small perturbation?

$$\tilde{\chi}(\lambda) = \chi(\lambda) - \epsilon \lambda^{19} \quad \text{where} \quad \epsilon = 2^{-23} \approx 10^{-7}.$$

The Eigenvalue Problem: Example

The $\tilde{\chi}(\lambda)$ has the followings roots:

$\lambda_1 = 1.000000000$	$\lambda_{10/11} = 10.095266145$	$\pm 0.643500904i$
$\lambda_2 = 2.000000000$	$\lambda_{12/13} = 11.793633881$	$\pm 1.652329728i$
$\lambda_3 = 3.000000000$	$\lambda_{14/15} = 13.992358137$	$\pm 2.518830070i$
$\lambda_4 = 4.000000000$	$\lambda_{16/17} = 16.730737466$	$\pm 2.812624894i$
$\lambda_5 = 4.999999928$	$\lambda_{18/19} = 19.502439400$	$\pm 1.940330347i$
$\lambda_6 = 6.000006944$	$\lambda_{20} = 20.846908101$	
$\lambda_7 = 6.999697234$		
$\lambda_8 = 8.007267603$		
$\lambda_9 = 8.917250249$		

What do you think?

The Eigenvalue Problem

Is finding the roots of the characteristic polynomial a suitable numerical approach to find the eigenvalues of a matrix?

- ▶ No general solution to find roots of polynomials of degree $n \geq 4$ (Abel–Ruffini theorem).
- ▶ Based on the example, the root finding problem seems to be ill-conditioned.

The example on the previous slide was "discovered" by James H. Wilkinson (1919–1986): "*the most traumatic experience in my career as a numerical analyst*"

Iterative approaches to the Eigenvalue problem

Algorithm: *The power method or von Mises iteration*

Idea: We assume that $A \in \mathbb{R}^{n \times n}$ and we are given some vector $x_0 \in \mathbb{R}^n$, $x_0 \neq 0$. We define the following iteration:

$$x_{i+1} = A \cdot x_i.$$

We hope that after several iterations, $x_i \approx v$ where $v \in \mathbb{R}^n$ is the eigenvector corresponding to the largest absolute eigenvalue of A .

Intuition: fixed point iteration...

Theorem for von Mises iteration

Let $A \in \mathbb{R}^{n \times n}$ be symmetric, i.e. $A = A^\top$, and let A have one single (non-repeated) eigenvalue $\lambda_1 = \lambda_1(A)$ with

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \dots \geq |\lambda_n|$$

where $\lambda_2, \dots, \lambda_n$ are the other eigenvalues of A . Let the vector $x_0 \in \mathbb{R}^n$ satisfy

$$x_0^\top \cdot v_1 \neq 0,$$

where v_1 the eigenvector that corresponds to λ_1 . Then, the series $y_i := x_i / \|x_i\|$ with x_i defined by the power method, converges to

$$\lim_{i \rightarrow \infty} y_i = v_1 / \|v_1\|.$$

Furthermore, the Rayleigh quotient $\rho_i = y_i^\top A y_i$ converges to the first eigenvalue:

$$\lim_{i \rightarrow \infty} \rho_i = \lambda_1.$$

Remark: It is true in general that $\lambda_i = v_i^\top A v_i$.

Idea: Why does the power method work?

We denote the n eigenvectors of A by v_1, \dots, v_n and let $\alpha_i = x_0^\top v_i$. Then, starting from

$$x_0 = \sum_{j=1}^n \alpha_j v_j$$

we obtain

$$\begin{aligned} x_i &= A^i x_0 \\ &= \sum_{j=1}^n \alpha_j \lambda_j^i v_j \\ &= \alpha_1 \lambda_1^i \left(v_1 + \underbrace{\sum_{j=2}^n \frac{\alpha_j}{\alpha_1} \left(\underbrace{\frac{\lambda_j}{\lambda_1}}_{<1} \right)^i v_j}_{<1} \right). \end{aligned}$$

Thus x_i will be proportional to v_1 as $i \rightarrow \infty$.

Direct iteration/power iteration

Summary:

- ▶ We can only identify λ_1 and v_1 , but not the others.
- ▶ The convergence speed of the power method is geometric with ratio $\left(\frac{\lambda_2}{\lambda_1}\right)$. If $|\lambda_1| \approx |\lambda_2|$ (that is, if the gap between the first and second eigenvalue is small), convergence will be slow.

Alternative idea to get another eigenvalue (smallest eigenvalue λ_n):

$$Av_n = \lambda_n v_n \iff A^{-1}v_n = \lambda_n^{-1}v_n.$$

Here, convergence speed is given by $\left(\frac{\lambda_n}{\lambda_{n-1}}\right)^i$. Is there a way to make the eigenvalues artificially small?

Convergence speed

We start with eigenvalue λ_j and the corresponding eigenvector v_j :

$$\begin{aligned} & Av_j = \lambda_j v_j \\ \iff & (A - \tilde{\lambda} I_{n,n})v_j = (\lambda_j - \tilde{\lambda})v_j \quad \text{for some } \tilde{\lambda} \in \mathbb{R} \\ \iff & (A - \tilde{\lambda} I_{n,n})^{-1}v_j = \frac{1}{(\lambda_j - \tilde{\lambda})}v_j \end{aligned}$$

If we apply now the power iteration, then we get the convergence speed:

$$\max_{\substack{k=1, \dots, n \\ k \neq j}} \left| \frac{\lambda_j - \tilde{\lambda}}{\lambda_k - \tilde{\lambda}} \right|$$

What does this imply?

Inverse iteration

The inverse iteration is given by

$$x_{i+1} = (A - \tilde{\lambda} I_{n \times n})^{-1} x_i.$$

If $\tilde{\lambda}$ is a good guess for λ_j , meaning $\max_{\substack{k=1, \dots, n \\ k \neq j}} \left| \frac{\lambda_j - \tilde{\lambda}}{\lambda_k - \tilde{\lambda}} \right| \ll 1$, then y_i

and ρ_i will converge to the eigenvector v_j and eigenvalue λ_j .

- ▶ The convergence speed depends on the guess $\tilde{\lambda}$ for λ_j .
- ▶ Instead of computing the inverse of the matrix $(A - \tilde{\lambda} I_{n \times n})^{-1}$, we will solve the linear equation system

$$(A - \tilde{\lambda} I_{n \times n}) x_{i+1} = x_i.$$

How do we implement this and what are the concerns?

Example

Let's consider the example matrix

$$A = \begin{pmatrix} -1 & 3 \\ -2 & 4 \end{pmatrix}$$

It is easy to see that the eigenvalues of A are $\lambda_1 = 2$ and $\lambda_2 = 1$.
Our guess is $\tilde{\lambda} = 1 - \epsilon$, thus

$$(A - \tilde{\lambda}I_{n \times n}) = \begin{pmatrix} -2 + \epsilon & 3 \\ -2 & 3 + \epsilon \end{pmatrix}$$

$$(A - \tilde{\lambda}I_{n \times n})^{-1} = \frac{1}{\epsilon(\epsilon + 1)} \underbrace{\begin{pmatrix} 3 + \epsilon & -3 \\ 2 & -2 + \epsilon \end{pmatrix}}_{=:B}$$

Good initial guesses for eigenvalues

- ▶ How can we obtain the initial/good guesses for $\lambda_1, \dots, \lambda_n$?
- ▶ If A is symmetric, we know that there is an orthogonal matrix Q such that $Q^T Q = I$ with $A = Q^T \text{diag}(\lambda_1, \dots, \lambda_n) Q$.
- ▶ But there is no direct way to get the orthogonal matrix T .
- ▶ However, this was the motivation for the QR-algorithm.

An iterative way to find good guesses for λ_i : The QR-algorithm

Data: matrix $A \in \mathbb{R}^{n \times n}$ and integer $m \in \mathbb{N}$

Result: matrix A_m and orthogonal matrix Q_m

$A_0 := A$

for $i=1, \dots, m$ **do**

(Q_i, R_m) := QR-decomposition of A_i ;

($A_i := R_i * Q_i$)

end

See R example...

Features of the QR-Algorithm

- ▶ Under certain regularity conditions and

$$|\lambda_1| > |\lambda_2| > \dots > |\lambda_{n-1}| > |\lambda_n|$$

the matrix A_m converges to an upper triangular matrix in which the diagonal elements are the eigenvalues of A .

- ▶ If the matrix A is also symmetric, then the matrix A_m converges to a diagonal matrix whose elements are the eigenvalues of A .
- ▶ There are several generalizations of these convergence criteria.
- ▶ The proof of the above statements is very technical and thus omitted here.

Reference: Martin H. Gutknecht and Beresford N. Parlett (2011). "From QD to LR, or, how were the QD and LR algorithms discovered?". IMA Journal of Numerical Analysis, 31(3), pp. 741–754.

QR-algorithm

- ▶ The QR-decomposition in each step is, especially for large matrices, very computationally intensive and many decompositions have to be computed to stabilize the diagonal elements of A_m .
- ▶ Can we transform the matrix A so that the QR-decomposition becomes cheaper/less computational intensive?

Transformation of a symmetric matrix A

There is no direct way to compute the orthogonal matrix Q such that

$$Q^\top A Q = \text{diag}(\lambda_1, \dots, \lambda_n)$$

We could use a Householder matrix H satisfying

$$HA = \begin{pmatrix} \tilde{a}_{11} & \tilde{a}_{12} & \dots & \dots & \dots & \tilde{a}_{1n} \\ 0 & \tilde{a}_{22} & \dots & \dots & \dots & \tilde{a}_{2n} \\ 0 & \tilde{a}_{32} & \dots & \dots & \dots & \tilde{a}_{3n} \\ \vdots & \vdots & \dots & \dots & \dots & \vdots \\ 0 & \tilde{a}_{n2} & \dots & \dots & \dots & \tilde{a}_{nn} \end{pmatrix}$$

However, HAH^\top would destroy the format, i.e. it would create non-zero elements in the first column.

Transformation of a symmetric matrix A

What happens if we construct a Householder matrix H for A which introduces 0 below a_{21} instead of a_{11} ?

$$H_1 A H_1^\top = \begin{pmatrix} \tilde{a}_{11} & \tilde{a}_{12} & 0 & \dots & \dots & 0 \\ \tilde{a}_{21} & \tilde{a}_{22} & \tilde{a}_{23} & \dots & \dots & \tilde{a}_{2n} \\ 0 & \tilde{a}_{32} & \dots & \dots & \dots & \tilde{a}_{3n} \\ \vdots & \vdots & \dots & \dots & \dots & \vdots \\ 0 & \tilde{a}_{n2} & \dots & \dots & \dots & \tilde{a}_{nn} \end{pmatrix}$$

Proof: optional homework

Transformation of a symmetric matrix A

We can therefore construct a sequence of $n - 1$ Householder H_j matrices such that

$$H_{n-1} \dots H_1 A H_1^\top \dots H_{n-1}^\top = \begin{pmatrix} \tilde{a}_{11} & \tilde{a}_{21} & 0 & 0 & \dots & \dots & \dots \\ \tilde{a}_{21} & \tilde{a}_{22} & \tilde{a}_{32} & 0 & \dots & \dots & \dots \\ 0 & \tilde{a}_{32} & \tilde{a}_{33} & \tilde{a}_{43} & 0 & \dots & \dots \\ 0 & 0 & \tilde{a}_{43} & \tilde{a}_{44} & \ddots & 0 & \dots \\ \vdots & \vdots & 0 & \ddots & \ddots & \ddots & \ddots \\ \vdots & \vdots & \vdots & 0 & \ddots & \ddots & \ddots \end{pmatrix} \in \mathbb{R}^{n \times n}$$

The transformed matrix is a symmetric tri-diagonal matrix. The numerical burden/complexity for the transformation to tri-diagonal form is $O(n^3)$.

Transformation to a tri-diagonal matrix

What are the advantages of this matrix structure in the QR-algorithm?

- ▶ The numerical burden of the QR -decomposition is greatly reduced. Householder reflections or Givens rotations have to be applied only to the off-diagonal elements below the diagonal which requires only $O(n^2)$ operations. In contrast, the effort is $O(n^3)$ for the QR -decomposition of a regular $(n \times n)$ -matrix.

Sparse matrices

Contents:

- ▶ Sparse matrices
- ▶ Storage formats
- ▶ Advantages and disadvantages

Sparse matrices: Examples

In numerical analysis and computer science, a sparse matrix or sparse array is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered dense.

Example:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 6 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Big data applications:

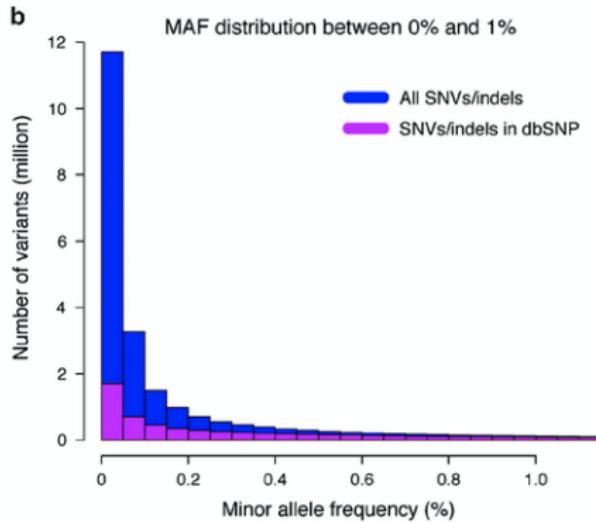
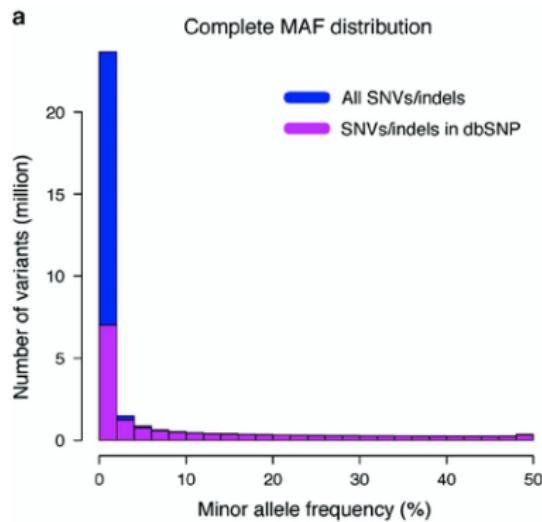
- ▶ Connection matrix: Air traffic/direct connection, linked web-pages, etc.
- ▶ "User-Activity" matrix, e.g. for twitter, online shopping, etc.

Sparse Matrices: Genetic studies

- ▶ In genetic association studies, marker scores (number of minor alleles at a specific locus) are often used as covariates in statistical models.
- ▶ The marker score matrix $X = (x_{i,j}) \in \mathbb{R}^{n \times m}$ stores the number of minor alleles $x_{i,j} \in \{0, 1, 2\}$ in the i th study subject/patient and at the j th marker locus.
- ▶ Realistic values: $n > 100,000$ and $m > 1,000,000$.

Sparse Matrices: Genetic studies

Example: Marker allele frequency (MAF) distribution



Sparse Matrices: Genetic studies

Probabilities of obtaining the three possible matrix entries:

- ▶ $P(X = 0) = (1 - MAF)^2$
- ▶ $P(X = 1) = 2 * MAF * (1 - MAF)$
- ▶ $P(X = 2) = MAF^2$

Therefore, most $x_{ij} = 0$, some $x_{ij} = 1$, and very few $x_{ij} = 2$.

The matrix X will be very sparse.

Sparse Matrices: Why do we care?

In order to scale any application, we need to consider:

- ▶ savings in storage space
- ▶ savings in computing time

When storing and manipulating sparse matrices on a computer, it is beneficial and often necessary to use specialized algorithms and data structures that take advantage of the sparse matrix structure.

Algorithms performing operations on standard dense-matrix structures are slow and inefficient when applied to large sparse matrices. This is because processing and memory are wasted on zero entries. Sparse data is by nature more easily compressed and thus requires significantly less storage. Algorithms and operations have to be modified to work efficiently on sparse data structures.

Sparse matrices: Sparsity coefficient

The number of zero-valued elements divided by the total number of elements (e.g., $m \cdot n$ for an $m \times n$ matrix) is called the sparsity p of the matrix:

$$p = \frac{\text{"number of 0-value elements"}}{m \cdot n}.$$

Sparse Matrices: Storage formats

Three ways to store a matrix $A = (a_{ij})$:

- ▶ **Naïve** (or "traditional", classic) format: A matrix is stored as one vector of length $n \cdot m$ and a dimension attribute.
- ▶ **Triplet** format: three vectors of length z , encoding triplets (i, j, a_{ij}) for the position of each non-zero matrix element and its value, and a dimension attribute, where z is the number of non-zero elements.
- ▶ **Compressed sparse row** or CSR format: eliminate redundant row indices.

Example: Storage format

Example: We want to store the matrix

$$A = \begin{pmatrix} 1 & 0.1 & 0 & 0.2 & 0.3 \\ 0.4 & 2 & 0 & 0.5 & 0 \\ 0 & 0 & 3 & 0 & 0.6 \\ 0.7 & 0.8 & 0 & 4 & 0 \\ 0.9 & 0 & 0.0 & 0 & 5 \end{pmatrix} \quad z = 15$$

Naïve format encoding all columns:

1, .4, 0, .7, .9, .1, 2, 0, .8, 0, 0, 0, 3, 0, .0, .2, .5, 0, 4, 0, .3, 0, .6, 0, 5

Storage requirement?

Example: Storage format

Example: We want to store the matrix

$$A = \begin{pmatrix} 1 & 0.1 & 0 & 0.2 & 0.3 \\ 0.4 & 2 & 0 & 0.5 & 0 \\ 0 & 0 & 3 & 0 & 0.6 \\ 0.7 & 0.8 & 0 & 4 & 0 \\ 0.9 & 0 & 0.0 & 0 & 5 \end{pmatrix} \quad z = 15$$

Triplet format:

i	1	1	1	1	2	2	2	3	3	4	4	4	5	5	5
j	1	2	4	5	1	2	4	2	3	1	2	4	1	3	5
a_{ij}	1	.1	.2	.3	.4	2	.5	3	.6	.7	.8	4	.9	.0	5

Storage requirement?

Example: Storage format

Example: We want to store the matrix

$$A = \begin{pmatrix} 1 & 0.1 & 0 & 0.2 & 0.3 \\ 0.4 & 2 & 0 & 0.5 & 0 \\ 0 & 0 & 3 & 0 & 0.6 \\ 0.7 & 0.8 & 0 & 4 & 0 \\ 0.9 & 0 & 0.0 & 0 & 5 \end{pmatrix} \quad z = 15$$

Compressed sparse row (CSR) format:

i	1	1	1	1	2	2	2	3	3	4	4	4	5	5	5
j	1	2	4	5	1	2	4	2	3	1	2	4	1	3	5
a_{ij}	1	.1	.2	.3	.4	2	.5	3	.6	.7	.8	4	.9	.0	5

Example: Storage format

Example: We want to store the matrix

$$A = \begin{pmatrix} 1 & 0.1 & 0 & 0.2 & 0.3 \\ 0.4 & 2 & 0 & 0.5 & 0 \\ 0 & 0 & 3 & 0 & 0.6 \\ 0.7 & 0.8 & 0 & 4 & 0 \\ 0.9 & 0 & 0.0 & 0 & 5 \end{pmatrix} \quad z = 15$$

Compressed sparse row (CSR) format:

ptr	0		4		7		9		12			
j	1	2	4	5	1	2	4	2	3	1	2	4
a_{ij}	1	.1	.2	.3	.4	2	.5	3	.6	.7	.8	.9

Example: Storage format

Compressed sparse row (CSR) format:

ptr	0		4		7		9		12			
j	1	2	4	5	1	2	4	2	3	1	2	4
a_{ij}	1	.1	.2	.3	.4	2	.5	3	.6	.7	.8	.9

5 .0 5

Specification of the CSR format:

- ▶ $ptr[0] = 0$
- ▶ $ptr[1] = ptr[0] + \text{"number of non-zero elements in row 1"}$
- ▶ $ptr[2] = ptr[1] + \text{"number of non-zero elements in row 2"}$

Memory requirement?

Advantages/disadvantages of the previous storage formats

- ▶ Naïve format: No savings in storage and computation (for sparse matrices). *Status quo* format.
- ▶ Triplet format: Savings in storage and computation for sparse matrices but loss in storage and computation for full matrices. The format is intuitive and easy to construct.
- ▶ Compressed sparse row (CSR) format: Apart from intuition, same as triplet format. Faster element access and widely used in available algorithms. Arbitrary choice for "row" vs. "column" format (CSC), therefore well suited for matrix-vector multiplications.

Homework Part 1: Storage format for genotype matrices

- ▶ Matrix X with entries $X_{i,j} \in \{0, 1, 2\}$, the number of copies of the minor allele.
- ▶ As the allele frequency of the minor allele is by definition smaller than 0.5, the matrix X is sparse.
- ▶ Question: How sparse are such matrices?
- ▶ Please write matrix X in triplet format and CSR format.

Homework Part 2

- ▶ Given two sparse $n \times n$ -times matrices, A and B . What is the expected sparsity of $A + B$ and $A * B$?
- ▶ For genotype matrices, define a new CSR format that takes the special data structure of the genotype data into account. For the evaluation use values for n and m that make the simulation studies feasible on your laptop/computer.
 - ▶ Using simulation studies (by drawing from the provided MAF data), how much storage space do you save on average?
 - ▶ At least how much storage space do you save in 95% of the cases/simulation studies?

Systems of nonlinear equations

Contents:

- ▶ Basics of nonlinear equation systems
- ▶ Sensitivity and conditioning
- ▶ Convergence rate
- ▶ Methods: bisection method, fixed point iteration, Newton method, secant method, Broyden's method
- ▶ Extensions to higher dimensions

Overview of nonlinear equations

Generally speaking, we are looking for the zero of an arbitrary function $f : X \rightarrow \mathbb{R}$, that is $x \in X$ such that

$$f(x) = 0.$$

Such a value x is called **root** or **zero** of f . The problem of finding x is called **root finding** or **zero finding**.

Overview of nonlinear equations

Root finding is well defined in more than one dimension:

- ▶ Consider a single nonlinear equation in one unknown,

$$f : \mathbb{R} \rightarrow \mathbb{R}.$$

We seek $x \in \mathbb{R}$ such that $f(x) = 0$.

- ▶ Consider a system of n *coupled* nonlinear equations in n unknowns,

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^n.$$

Solutions are vectors $x \in \mathbb{R}^n$ for which $f(x)$ is the zero vector.

Example

- ▶ Example of a nonlinear equation in one dimension:

$$x^2 - 4 \sin(x) = 0$$

for which $x = 1.9$ is one approximate solution.

- ▶ Example of nonlinear equations in two dimensions:

$$x_1^2 - x_2 + 0.25 = 0$$

$$-x_1 + x_2^2 + 0.25 = 0$$

with solution $x = [0.5, 0.5]^\top$.

Existence and uniqueness

- ▶ In general, establishing existence and uniqueness are more complicated for nonlinear equations than for linear ones.
- ▶ **Definition:** A bracket for a function $f : \mathbb{R} \rightarrow \mathbb{R}$ is an interval $[a, b]$ such that $\text{sign}(f(a)) \neq \text{sign}(f(b))$.
- ▶ If f is continuous, and $[a, b]$ is a bracket for f , then the *Intermediate Value Theorem* implies the existence of $x^* \in [a, b]$ such that $f(x^*) = 0$.
- ▶ There is no simple analog for n dimensions.

Nonlinear equations can have any number of solutions

Examples:

- ▶ $\exp(x) + 1 = 0$ has no solution.
- ▶ $\exp(-x) - x = 0$ has one solution.
- ▶ $x^2 - 4 \sin(x) = 0$ has two solutions.
- ▶ $x^3 + 6x^2 + 11x - 6 = 0$ has three solutions.
- ▶ $\sin(x) = 0$ has infinitely many solutions.

Sensitivity and conditioning

- ▶ For root finding, the *absolute condition number* for a function $f : \mathbb{R} \rightarrow \mathbb{R}$ and its root x^* is $1/|f'(x^*)|$.
- ▶ Thus the root finding problem is ill-conditioned if the tangent line at x^* is nearly horizontal.
- ▶ In particular, the root finding problem is ill-conditioned for roots with multiplicity ≥ 2 .
- ▶ Observe that conditioning of the root finding problem is opposite to that for evaluating the function.

Sensitivity and conditioning

In higher dimensions:

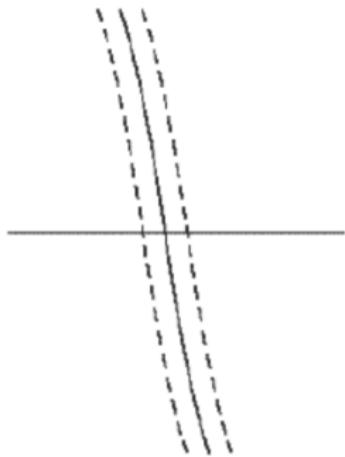
- ▶ The *absolute condition number* for finding the root $x^* \in \mathbb{R}^n$ in n dimensions is

$$\|J_f^{-1}(x^*)\|,$$

where J_f is the *Jacobian* matrix of $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$, defined as $(J_f(x))_{ij} = \partial f_i(x) / \partial x_j$.

- ▶ The root finding problem is ill-conditioned if the Jacobian matrix is nearly singular.

Well-conditioned and ill-conditioned root finding



well-conditioned



ill-conditioned

Accuracy of approximate solution

Question: What is a good approximation \hat{x} of the zero x^* of a nonlinear equation?

We could think of

$$\|f(\hat{x})\| \approx 0 \quad \text{or} \quad \|\hat{x} - x^*\| \approx 0.$$

- ▶ First statement corresponds to "small residual", second one measures the distance to the (unknown) true zero.
- ▶ Both solution criteria are not necessarily close simultaneously.
- ▶ Small residual (=left statement) implies accurate solution (=right statement) only if the problem is well-conditioned.

Convergence rate

- ▶ Many solvers are iterative methods, e.g. Newton iteration, bisection, etc.
- ▶ For iterative methods we define the error in iteration k as

$$e_k = \|x_k - x^*\|,$$

where x_k is the approximation in iteration k and x^* is the true solution.

- ▶ Some methods like bisection maintain an interval $[a_k, b_k]$ known to contain the solution in iteration k . For those, we can take the error to be the length of the interval $e_k := b_k - a_k$.

Convergence rate

Definition: The sequence e_k converges with rate r if

$$\lim_{k \rightarrow \infty} \frac{\|e_{k+1}\|}{\|e_k\|^r} = C,$$

where C is a finite nonzero constant.

Examples:

	convergence type	digits gained per iteration
$r = 1$	linear convergence ($C < 1$)	constant
$r > 1$	superlinear convergence	increasing
$r = 2$	quadratic convergence	double

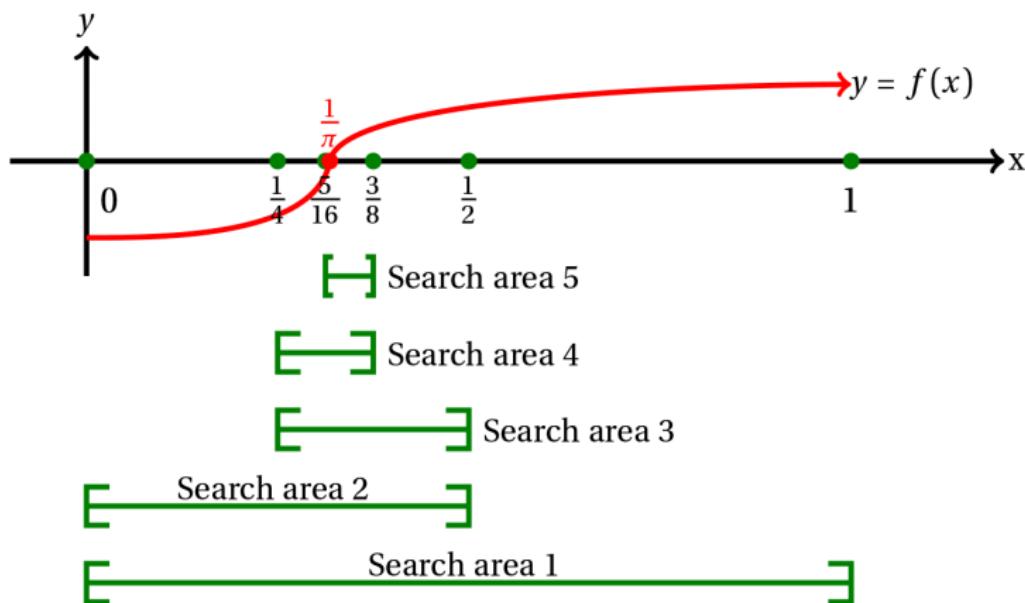
Methods

In the following, let's look at a couple of methods to compute approximate solutions of x^* :

- ▶ Bisection method
- ▶ fixed point iteration
- ▶ Newton method
- ▶ Secant method
- ▶ Broyden's method

Bisection method

Idea: We begin with an initial bracket $[a, b]$ for f which we know contains x^* , and repeatedly halve its length until x^* has been isolated with desired accuracy.



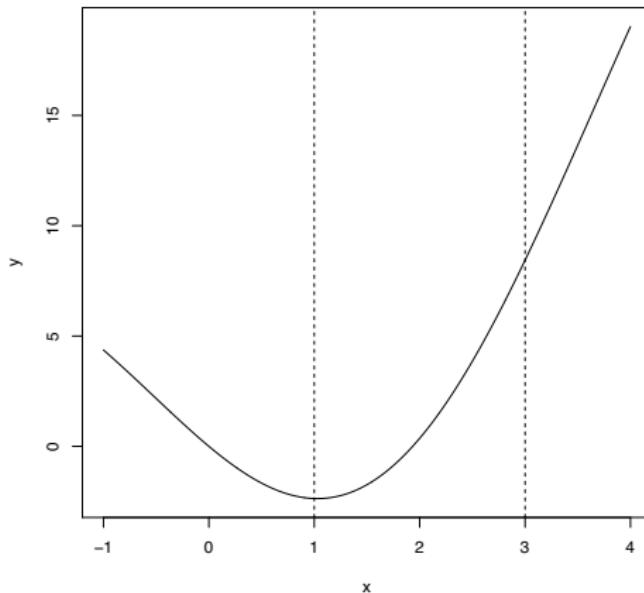
Bisection method: pseudo-code

Pseudo-code for the bisection method:

```
while ( $b - a$ ) > tol do
     $m \leftarrow a + (b - a)/2$ 
    if sign( $f(a)$ ) = sign( $f(m)$ ) then
        |  $a \leftarrow m$ 
    end
    else
        |  $b \leftarrow m$ 
    end
end
```

Bisection method: numerical example

We apply the bisection method to $f(x) = x^2 - 4 \sin(x)$ with initial bracket $[1, 3]$.



Bisection method: numerical example

We apply the bisection method to $f(x) = x^2 - 4 \sin(x)$ with initial bracket $[1, 3]$.

a	$f(a)$	b	$f(b)$
1.000000	-2.365884	3.000000	8.435520
1.000000	-2.365884	2.000000	0.362810
1.500000	-1.739980	2.000000	0.362810
1.750000	-0.873444	2.000000	0.362810
1.875000	-0.300718	2.000000	0.362810
1.875000	-0.300718	1.937500	0.019849
1.906250	-0.143255	1.937500	0.019849
1.921875	-0.062406	1.937500	0.019849
1.929688	-0.021454	1.937500	0.019849
1.933594	-0.000846	1.937500	0.019849
1.933594	-0.000846	1.935547	0.009491
1.933594	-0.000846	1.934570	0.004320
1.933594	-0.000846	1.934082	0.001736

Bisection method

Remarks:

- ▶ The bisection method only makes use of the sign of the function values, not their magnitude.
- ▶ Bisection is certain to converge, but does so slowly.
- ▶ In each iteration, the length of the interval containing the solution is halved. The **convergence is linear** with $r = 1$ and $C = 0.5$.
- ▶ That means one bit of accuracy is gained in the approximate solution with each new iteration.

Bisection method

Remarks:

- Given the starting interval $[a, b]$ (the initial bracket), after k iterations, the length of the interval containing the solution is

$$\frac{b - a}{2^k}.$$

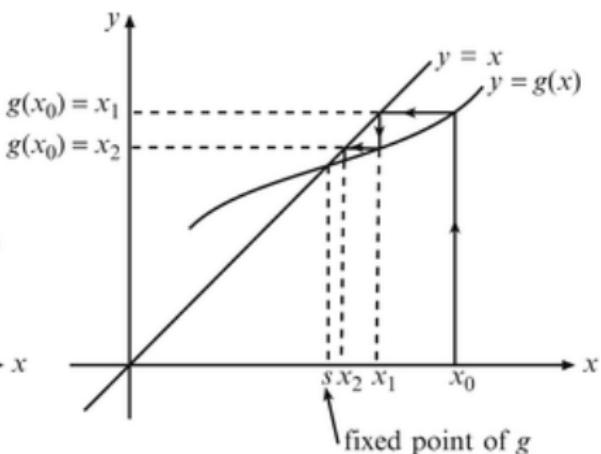
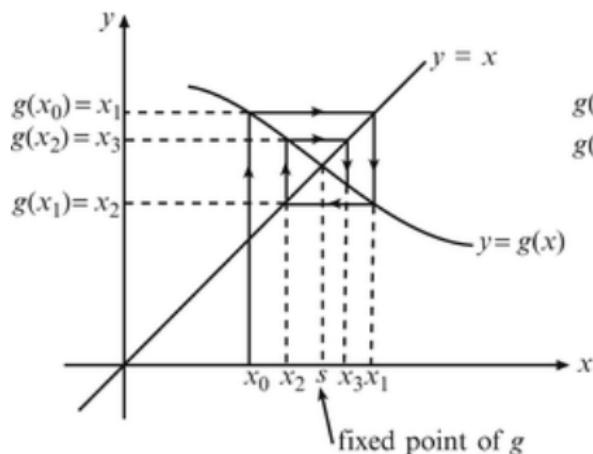
Thus achieving an error tolerance of tol requires

$$\left\lceil \log_2 \left(\frac{b - a}{tol} \right) \right\rceil$$

iterations, regardless of the function f involved.

Fixed point iteration

Idea: Starting from some x_0 , we repeatedly apply a function g to its last function value.



Fixed point iteration

- ▶ A **fixed point** of a function $g : \mathbb{R} \rightarrow \mathbb{R}$ is a value x such that

$$x = g(x).$$

- ▶ Many iterative methods for solving nonlinear equations use **fixed point iteration schemes** of the form

$$x_{k+1} = g(x_k),$$

where fixed points of g are solutions of $f(x) = 0$ for some function f . Example: Newton method!

- ▶ Fixed point iteration is also called *functional iteration* since the function g is repeatedly applied to a starting value x_0 .
- ▶ For a given problem $f(x) = 0$, there might be many equivalent fixed point problems $x = g(x)$ with different choices for g .

Examples

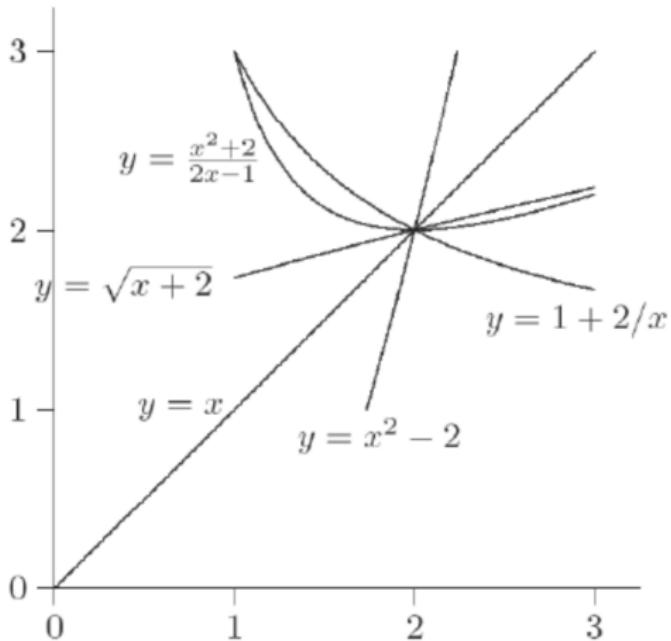
Let $f(x) = x^2 - x - 2$. Then the fixed points of

- ▶ $g(x) = x^2 - 2$
- ▶ $g(x) = \sqrt{x + 2}$
- ▶ $g(x) = 1 + 2/x$
- ▶ $g(x) = \frac{x^2+2}{2x-1}$

are all solutions to the equation $f(x) = 0$.

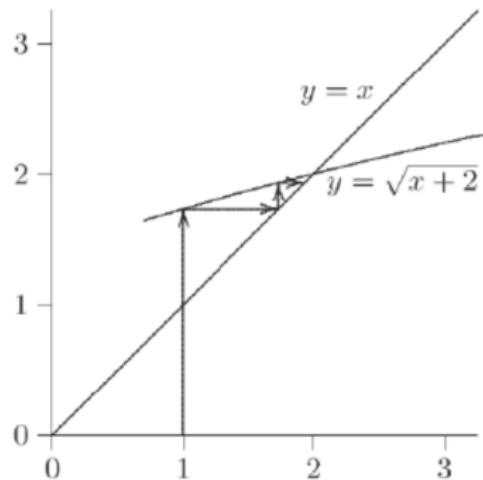
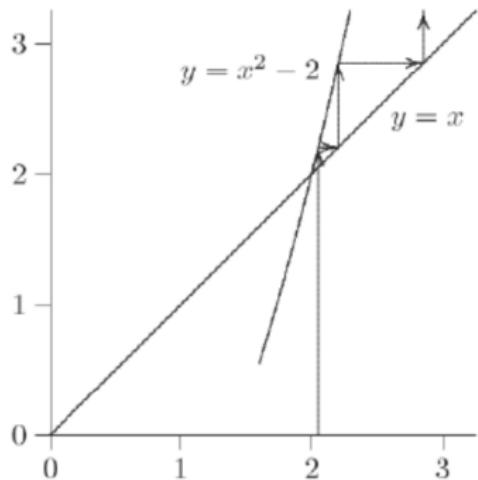
Examples

Fixed points of the following functions are zeros of
 $f(x) = x^2 - x - 2$:



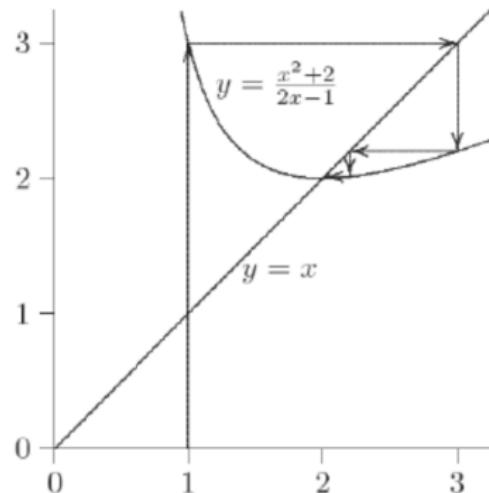
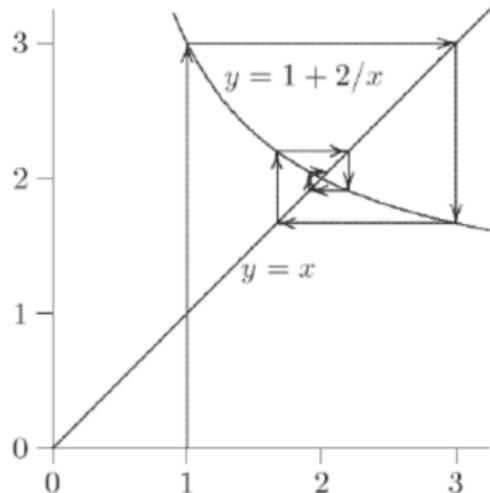
Examples

Functions $g(x) = x^2 - 2$ and $g(x) = \sqrt{x} + 2$:



Examples

Functions $g(x) = 1 + 2/x$ and $g(x) = \frac{x^2+2}{2x-1}$:



Convergence of the fixed point iteration

- ▶ If $x^* = g(x^*)$ and $|g'(x^*)| < 1$ then there is an interval I containing x^* such that the iteration

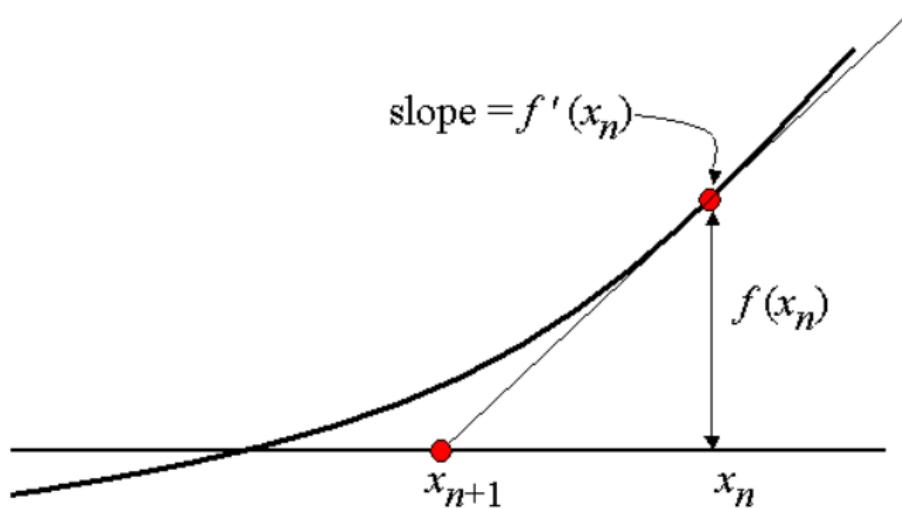
$$x_{k+1} = g(x_k)$$

converges to x^* if the initial value $x_0 \in I$ (the fixed point is an "attractor").

- ▶ If $|g'(x^*)| > 1$ then the iterative scheme diverges (the fixed point is repelling).
- ▶ **The asymptotic convergence rate of the fixed point iteration is usually linear** with constant $C = |g'(x^*)|$.
- ▶ If $g'(x^*) = 0$ then the convergence rate is at least quadratic!

Newton method

Idea: Newton's method approximates a nonlinear function f at the current approximation x_n of the zero x^* with the tangent line through $(x_n, f(x_n))$, and uses the zero of that tangent as the next estimate x_{n+1} .



Details of Newton's method

In particular, the tangent through $(x_n, f(x_n))$ has slope $f'(x_n)$ and is given by

$$y = f'(x_n) \cdot x + f(x_n) - f'(x_n) \cdot x_n,$$

where x is the unknown variable and x_n is fixed.

The zero of the tangent is

$$0 = f'(x_n) \cdot x + f(x_n) - f'(x_n) \cdot x_n \quad \Rightarrow \quad x = x_n - \frac{f(x_n)}{f'(x_n)}.$$

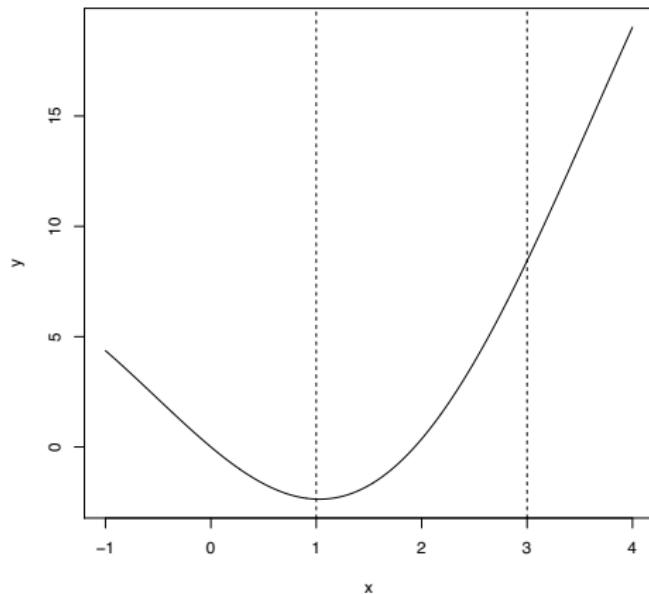
Thus the next approximation is given by the **Newton iteration**

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Under conditions, the sequence of x_n converges to the zero x^* of f .

Example of Newton iteration

We again try to find the zero of $f(x) = x^2 - 4 \sin(x)$ with starting value $x_0 = 3$. This example was used for the bisection method.



Example of Newton's method

We first compute the derivative of $f(x) = x^2 - 4 \sin(x)$ as

$$f'(x) = 2x - 4 \cos(x).$$

Thus the Newton iteration scheme becomes

$$x_{n+1} = x_n - \frac{x_n^2 - 4 \sin(x_n)}{2x_n - 4 \cos(x_n)}.$$

Starting with $x_0 = 3$ we obtain

x	f(x)	f'(x)
3.000000	8.435520	9.959970
2.153058	1.294772	6.505771
1.954039	0.108438	5.403795
1.933972	0.001152	5.288919
1.933754	0.000000	5.287670

Convergence of Newton's method

- ▶ Let's look again at the Newton iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

- ▶ Defining $g(x) := x - f(x)/f'(x)$ we see that Newton's method essentially **transforms the nonlinear equation $f(x) = 0$ into the fixed point problem $x = g(x)$.**
- ▶ Let's compute

$$g'(x) = 1 - \frac{[f'(x)]^2 - f(x)f''(x)}{[f'(x)]^2} = \frac{f(x)f''(x)}{[f'(x)]^2}.$$

- ▶ If x^* is a simple root (that is, $f(x^*) = 0$ and $f'(x^*) \neq 0$) then $g'(x^*) = 0$ and thus the **convergence rate of Newton's method is quadratic ($r = 2$)**.
- ▶ Starting value x_0 must be close to x^* for convergence.

Newton's method and multiple zeros

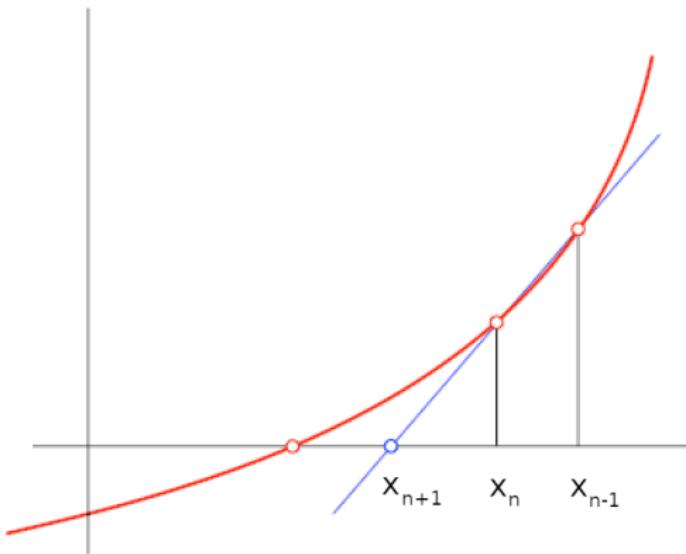
For roots/zeros with multiplicity $m > 1$, the **convergence rate of Newton's method is only linear**, with constant $C = 1 - 1/m$.

Example: Both functions $f_1(x) = x^2 - 1$ and $f_2(x) = x^2 - 2x + 1$ have zero $x^* = 1$, but $f'_1(x^*) \neq 0$ and $f'_2(x^*) = 0$.

k	$f_1(x) = x^2 - 1$	$f_2(x) = x^2 - 2x + 1$
0	2.0	2.0
1	1.25	1.5
2	1.025	1.25
3	1.0003	1.125
4	1.00000005	1.0625
5	1.0	1.03125

Secant method

Idea: Evaluating the gradient in Newton's method is computationally expensive. Instead, the secant method approximates the tangent to a nonlinear function f by the line through the previous two iterates.



Secant method

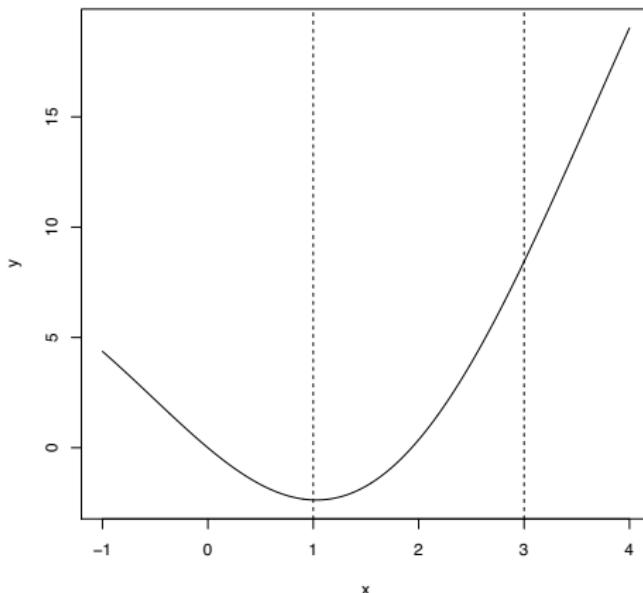
- ▶ In each iteration, the Newton method requires the evaluation of both the function and its derivative. However, computing the derivative might be inconvenient or expensive.
- ▶ The secant method uses the finite difference of the previous two successive iterates to approximate the gradient. Since the $f(x_{n-1})$ at the previous iterate x_{n-1} has already been computed, we only need to store its value for the next step.
The iteration becomes

$$x_{n+1} = x_n - f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

- ▶ The convergence rate of the secant method is normally **superlinear** with $r \approx 1.618$.

Example of the secant method

We again try to find the zero of $f(x) = x^2 - 4 \sin(x)$ with starting value $x_0 = 3$. This example was used for the bisection and Newton method.



Example of the secant method

We again try to find the zero of $f(x) = x^2 - 4 \sin(x)$ with starting value $x_0 = 3$. This example was used for the bisection and Newton method.

Starting with $x_0 = 1$ and $x_1 = 3$, we obtain

x	f(x)
1.000000	-2.365884
3.000000	8.435520
1.438070	-1.896774
1.724805	-0.977706
2.029833	0.534305
1.922044	-0.061523
1.933174	-0.003064
1.933757	0.000019
1.933754	0.000000

Higher-degree interpolation

- ▶ The secant method uses a linear interpolation to approximate the function whose zero is sought.
- ▶ We can obtain a higher convergence rate by using higher-degree polynomial interpolation.
- ▶ For example, quadratic interpolation (called "Muller's method") has a superlinear convergence with $r \approx 1.839$.
- ▶ Unfortunately, using higher degree polynomials also has disadvantages:
 - ▶ the interpolating polynomial might not have real roots
 - ▶ the roots might not be easy to compute themselves
 - ▶ it might not be obvious which root to use as the next iterate

Systems of nonlinear equations

We now look at the more general case of solving systems of nonlinear equations in higher dimensions.

Solving systems of nonlinear equations is much more difficult because:

- ▶ A wider variety of behavior is possible, thus determining existence and number of solutions or finding a good starting guess is much more complex.
- ▶ There is no simple way, in general, to guarantee convergence to the desired solution or to bracket a solution yielding a safe method.
- ▶ The computational overhead rapidly increases with the dimension of the problem.

Fixed point iteration in higher dimensions

- ▶ The *fixed point* problem for $g : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is to find a vector x such that $x = g(x)$.
- ▶ The corresponding **fixed point iteration** is

$$x_{k+1} = g(x_k).$$

- ▶ If $\rho(G(x^*)) < 1$, where ρ is the spectral radius (the largest eigenvalue in absolute value) and $G(x)$ is the Jacobian matrix of g evaluated at $x \in \mathbb{R}^n$, then the fixed point iteration converges if started close enough to x^* .
- ▶ The convergence is usually linear with constant $C = \rho(G(x^*))$.
- ▶ If $G(x^*) = 0$ (the zero matrix) then the convergence is at least quadratic.

Newton's method in higher dimensions

- ▶ In n dimensions, Newton's method for $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ generalizes to

$$x_{k+1} = x_k - J(x_k)^{-1}f(x_k),$$

where $J(x)$ is the Jacobian matrix of f given by
 $(J(x))_{ij} = \frac{\partial f_i}{\partial x_j}(x)$.

- ▶ In practice, we do not explicitly invert the matrix $J(x_k)$, but instead we solve the linear system

$$J(x_k)s_k = -f(x_k)$$

for the **Newton step** s_k and then compute the next iterate as

$$x_{k+1} = x_k + s_k.$$

Example of Newton's method

- We use Newton's method to solve the nonlinear system

$$f(x) = \begin{bmatrix} x_1 + 2x_2 - 2 \\ x_1^2 + 4x_2^2 - 4 \end{bmatrix} = 0.$$

- The Jacobian matrix is

$$J_f(x) = \begin{bmatrix} 1 & 2 \\ 2x_1 & 8x_2 \end{bmatrix}.$$

- If we start with $x_0 = [1, 2]^\top$ then

$$f(x_0) = \begin{bmatrix} 3 \\ 13 \end{bmatrix}, \quad J_f(x_0) = \begin{bmatrix} 1 & 2 \\ 2 & 16 \end{bmatrix}.$$

- Solving the linear system $J_f(x_0)s_0 = -f(x_0)$ yields

$$s_0 = \begin{bmatrix} -1.83 \\ -0.58 \end{bmatrix} \quad \Rightarrow \quad x_1 = x_0 + s_0 = \begin{bmatrix} -0.83 \\ 1.42 \end{bmatrix}.$$

Example of Newton's method

- ▶ Evaluating the function and its derivative at $x_1 = [-0.83, 1.42]^\top$ gives

$$f(x_1) = \begin{bmatrix} 0 \\ 4.72 \end{bmatrix}, \quad J_f(x_1) = \begin{bmatrix} 1 & 2 \\ -1.67 & 11.3 \end{bmatrix}.$$

- ▶ Solving the linear system $J_f(x_1)s_1 = -f(x_1)$ yields

$$s_1 = \begin{bmatrix} 0.64 \\ -0.32 \end{bmatrix} \quad \Rightarrow \quad x_2 = x_1 + s_1 = \begin{bmatrix} -0.19 \\ 1.10 \end{bmatrix}.$$

- ▶ Evaluating f and J_f at the new point x_2 yields

$$f(x_2) = \begin{bmatrix} 0 \\ 0.83 \end{bmatrix}, \quad J_f(x_2) = \begin{bmatrix} 1 & 2 \\ -0.38 & 8.76 \end{bmatrix}.$$

- ▶ Iterations eventually converge to the solution $x^* = [0, 1]^\top$.

Convergence of Newton's method in higher dimensions

- We look at the fixed point operator for Newton's method:

$$g(x) = x - J(x)^{-1}f(x).$$

- Differentiating g and evaluating at the solution x^* yields

$$G(x^*) = I - \left[J(x^*)^{-1}J(x^*) + \sum_{i=1}^n f_i(x^*)H_i(x^*) \right] = 0,$$

since $f_i(x^*) = 0$ by definition, where $H_i(x^*)$ is the component matrix of the derivative of $J(x)^{-1}$.

Convergence of Newton's method in higher dimensions

From the last slide we conclude:

- ▶ The **convergence rate of Newton's method for nonlinear systems is usually quadratic**, provided the Jacobian matrix $J(x^*)$ is nonsingular.
- ▶ As usual, the initial value x_0 must be close enough to the solution x^* for convergence.

Computational cost of Newton's method

The computational cost per iteration of Newton's method for a dense problem in n dimensions is:

- ▶ Computing the Jacobian matrix is $O(n^2)$ scalar function evaluations.
- ▶ Solving the linear system is $O(n^3)$ operations.

Secant updating and pseudo-Newton methods

- ▶ **Secant updating** methods reduce the computational cost by
 - ▶ Using function values at successive iterates to build an approximate Jacobian matrix, thus avoiding the explicit evaluation of derivatives.
 - ▶ Updating the factorization of the approximate Jacobian rather than refactoring in each iteration.
- ▶ Most secant updating methods have superlinear but not quadratic convergence rates.
- ▶ Secant updating methods often have a lower overall computational cost than Newton's method because they have a lower cost per iteration.

Broyden's method

- ▶ **Broyden's method** is a typical secant updating method.
- ▶ Beginning with an initial guess x_0 for the solution and an initial approximate Jacobian B_0 , the following steps are iterated until convergence:

input: x_0 (*initial guess*), B_0 (*initial Jacobian approximation*)

for $k = 0, 1, \dots$ **do**

Solve $B_k s_k = -f(x_k)$ for s_k

$x_{k+1} \leftarrow x_k + s_k$

$y_k \leftarrow f(x_{k+1}) - f(x_k)$

$B_{k+1} = B_k + \frac{[(y_k - B_k s_k) s_k^\top]}{s_k^\top s_k}$

end

Robust Newton-type methods

- ▶ Newton's method and its variants may fail to converge if started too far away from the solution x^* .
- ▶ Certain *safeguards* can enlarge the convergence region of Newton-type methods.
- ▶ The simplest precaution is the **damped Newton method**, in which the new iterate is defined as

$$x_{k+1} = x_k + \alpha_k s_k,$$

where s_k is the Newton step and α_k is a scalar parameter chosen to ensure progress towards the solution.

- ▶ The parameter α_k reduces the Newton step when it is too large (given $\alpha_k < 1$). However, $\alpha_k = 1$ suffices near the solution and still yields a fast asymptotic convergence rate.

Permutations and Monte Carlo

Contents:

1. null hypothesis and alternative hypothesis
2. permutation testing and Monte Carlo sampling
3. p-values
4. How many permutations are needed?
5. advantages and disadvantages of permutation tests

Statistical simulations: permutation testing, simulation studies, etc.

- ▶ Statistical simulation (Monte Carlo) is an important part of statistical method research.
- ▶ The statistical theories/methods are all based on assumptions. So most theorems state something like "if the data follow these models/assumptions, then ...".
- ▶ The theories can hardly be verified in real world data because (1) the real data never satisfy the assumption, and (2) the underlying truth is unknown (no "gold standard").
- ▶ In a simulation, data are "created" in a well controlled environment (model assumptions), having the advantage that the ground truth is known. Thus assumptions in theorems can be verified and their sensitivity to the assumptions can be assessed.

First step for Monte Carlo simulations

We need a good random number generator!

- ▶ Random number generation is the basis of statistical simulation. It serves to generate random numbers from predefined statistical distributions which are required to conduct permutation testing or simulation studies.
- ▶ In the "big data" era, often large scale simulation studies are required to study the behavior of models or to assess significance.
- ▶ We need a fast and reliable mechanism/algorithm to generate large sets of random numbers ($> 10^6$).

Pseudo-random number generators (PRNG)

A *good* random-number generator should satisfy the following properties:

- ▶ Uniformity: The numbers should be distributed uniformly in $(0, 1)$.
- ▶ Independence: The generated numbers show no dependence on each other, in particular they are uncorrelated.
- ▶ "Diehard" tests: Passes all "Diehard" tests for randomness/qc-standard tests for PRNG.
- ▶ Replication: The numbers should be replicable (e.g., for debugging or comparison of different systems).
- ▶ Cycle length: It should take long before numbers start to repeat.
- ▶ Speed: The generator should be fast.
- ▶ Memory usage: The generator should not require a lot of storage.
- ▶ Optional: Parallel implementation should be possible.
- ▶ Optional: The generator should be cryptographically secure.

We have seen the gold standard: *Mersenne Twister*.

Permutation testing

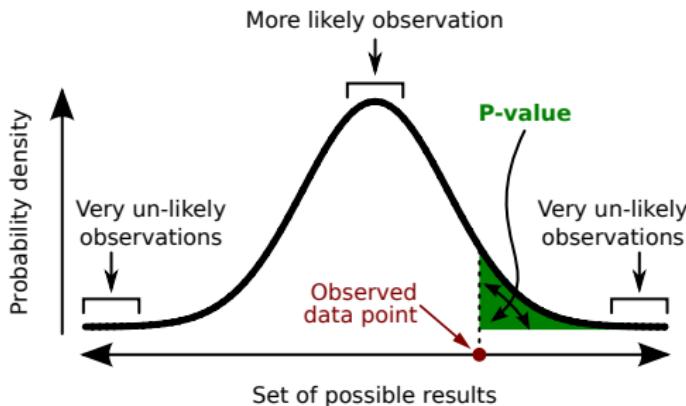
- ▶ In statistical inference, it is important to know the distribution of some statistics under the null hypothesis H_0 , so that quantities like p-values can be computed.
- ▶ The null distribution is available theoretically in some cases and requires assumptions about the data that may not be verifiable in applications.

Example: Assume $X_i \sim N(\mu, \sigma^2)$ for $i \in \{1, \dots, n\}$. Under $H_0 : \mu = 0$, we have $\sum_{i=1}^n X_i / n \sim N(0, \sigma^2/n)$. In this case, H_0 can be tested by comparing $\sum_{i=1}^n X_i / n$ with $N(0, \sigma^2/n)$.

When the null distribution cannot be obtained, it is useful to use a permutation test to "create" a null distribution from the data.

Testing one hypothesis

- ▶ Test null hypothesis H_0 against alternative H_1 using statistic T to summarise data.
- ▶ Compute a p-value, *the probability of observing an outcome at least as extreme as the one that was actually observed, assuming that the null hypothesis is true.*
- ▶ P-value quantifies the idea of statistical significance of evidence: $p = \mathbb{P}(\text{Data}|H_0)$.
- ▶ Reject the null hypothesis if $p \leq \alpha$ for some type I error α .



(source: wikipedia “P-value”)

Basic idea of permutation testing

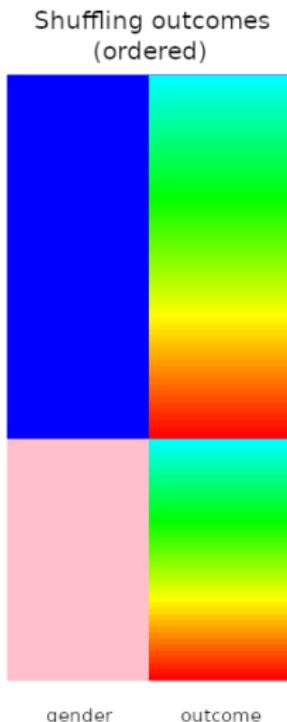
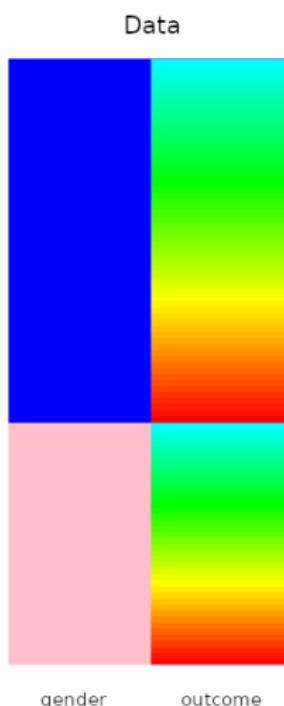
The basic idea of a permutation test for H_0 is:

- ▶ Permute the data under H_0 for a number of times. Each time recompute the test statistics. The distribution of test statistics obtained from the permuted data forms the null distribution.
- ▶ Compare the observed test statistics (for the unpermuted data you started with) with the null distribution to assess statistical significance of the unpermuted data, i.e. count the number of times the test statistics computed based on the permuted data is more extreme than the observed test statistic. This allows us to compute a p-value for our observations.

If the null hypothesis is true then the permuted/shuffled datasets should look like the real data, otherwise they should look different from the real data.

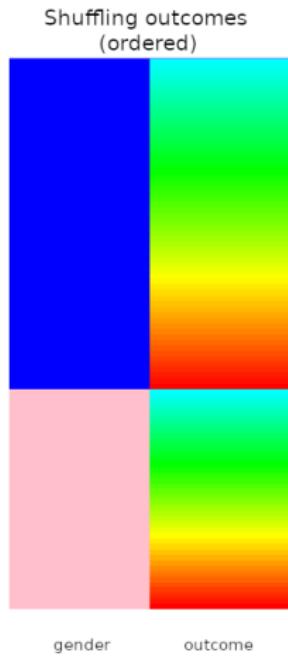
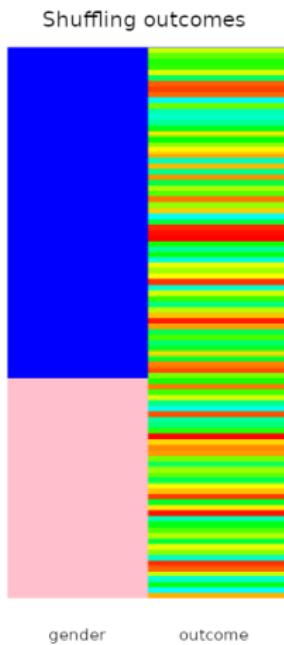
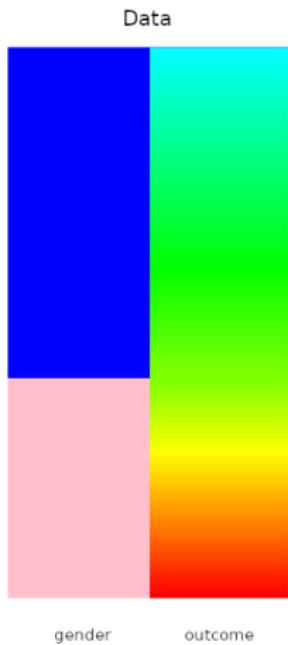
Permutation testing

Null hypothesis:



Permutation testing

Alternative hypothesis:



Example of a permutation test

Setting:

- ▶ We consider a linear regression setting without intercept:

$$Y = \beta X + \varepsilon.$$

- ▶ We want to test the coefficient:

$$H_0 : \beta = 0.$$

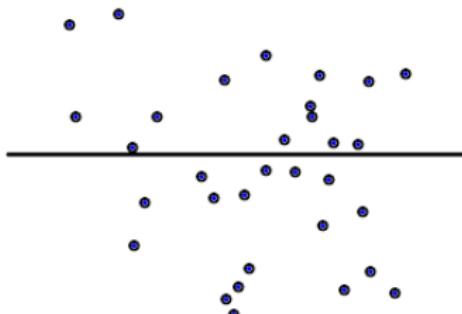
- ▶ The observed data are pairs (x_i, y_i) for $i \in \{1, \dots, n\}$, where x_i are vectors and y_i are scalars.
- ▶ We use the ordinary least square estimator for β , given by

$$\hat{\beta}(X, y) = \arg \min_{\beta} \|X\beta - y\|_2^2,$$

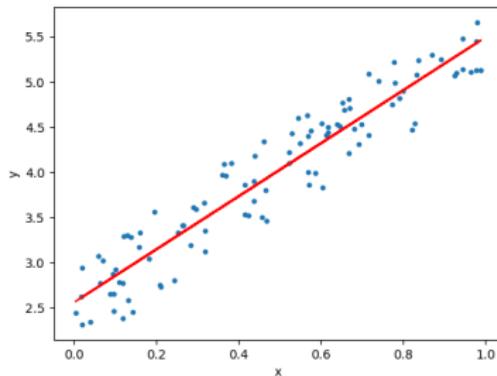
where X is a matrix containing the vectors x_i as columns, and $y = (y_1, \dots, y_n)$.

Example of a permutation test

If $\beta = 0$:



If $\beta \neq 0$:



Example of a permutation test

Setting: Linear regression $Y = \beta X + \varepsilon$ with data (x_i, y_i) for $i \in \{1, \dots, n\}$.

The permutation test steps are:

1. Keeping y_i unchanged, permute (change the orders of) each x_i to obtain a new vector, denoted as x_i^* .
2. Obtain estimate under the permuted data: $\hat{\beta}^*(X^*, y)$.
3. Repeat steps 1 and 2 for a sufficient number of steps so that $\hat{\beta}^*$ form the null distribution of $\hat{\beta}$.
4. Compute the p-value $\mathbb{P}(|\hat{\beta}^*| > |\hat{\beta}|)$.

Note: The random shuffling of x_i is based on the H_0 , i.e. assuming that there is no association between x and y .

Permutation tests in R

```
> x = rnorm(100);
> y = 0.2 * x + rnorm(100)
> summary(lm(y ~ x-1))
```

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
x	0.1502	0.1050	1.431	0.156

```
> nsims = 5000
> beta.obs = coef(lm(y ~ x-1))
> beta.perm = rep(0, nsims)
> for(i in 1:nsims) {
+   xstar = sample(x)
+   beta.perm[i] = coef(lm(y ~ xstar-1))
+ }
> mean(abs(beta.obs) < abs(beta.perm))
[1] 0.157
```

How many permutations do we need?

- ▶ The p-value is estimated as the proportion

$$\hat{p} = \frac{\#\text{exceedances over the observed test statistic}}{\#\text{simulations}}$$

or mathematically

$$\hat{p} = \frac{\sum_{i=1}^m \mathbb{I}(|T_i^*| > |T|)}{m},$$

where m is the number of simulations, T is the observed test statistics for the given data, T_i^* are the test statistics for the permuted data, and $\mathbb{I}(\cdot)$ is the indicator function.

- ▶ A confidence interval for the p-value estimate is often helpful to decide on the required number of replicates or simulations.
- ▶ This confidence interval is often based on the confidence interval of a binomial proportion (since the p-value can be viewed as draws).

How many permutations do we need?

The approximate 95%-confidence interval for a proportion \hat{p} based on m replicates/simulations is given by

$$\hat{p} \pm 2\sqrt{\frac{\hat{p}(1 - \hat{p})}{m}} \approx \hat{p} \pm 2\sqrt{\frac{\hat{p}}{m}}$$

for small p-values. Thus, to estimate a p-value of order 10^{-k} with $k \in \mathbb{N}$, we must have

$$10^{-k} > 4\sqrt{\frac{10^{-k}}{m}}.$$

This implies $m > 16 * 10^k \approx 10^{k+1}$ simulations are required (assuming the approximations hold).

Permutation testing: advantages and disadvantages

Advantages:

- ▶ Distribution free. Does not rely on asymptotic theory or assumptions.
- ▶ Does not require fully developed approaches in terms of the test statistic, i.e. any reasonable choice for the test statistic can be used.

Disadvantages:

- ▶ The theory of permutation tests is often not fully developed.
- ▶ Computationally more challenging, especially for a small significance level and large data set. Generating permutations can be very costly.

Summary

- ▶ **Monte Carlo methods** (or Monte Carlo experiments) are a broad class of algorithms that rely on repeated random sampling to obtain numerical results. They typically require the generation of large sets of pseudo-random numbers.
- ▶ Their essential idea is using randomness to solve problems that might be deterministic in principle. They are often used in computation and mathematical problems and are most useful when it is difficult or impossible or more time consuming to use other approaches.
- ▶ Monte Carlo methods are mainly used in 3 problem classes:
 1. validation of statistical/mathematical models, assessment of model features, e.g. model assumptions/robustness/power
 2. numerical integration
 3. evaluation of algorithms/data structures: debugging of code, performance evaluation

Numerical integration and MCMC

Contents:

- ▶ Numerical integration
- ▶ Monte Carlo integration
- ▶ Importance sampling
- ▶ MCMC, Metropolis–Hastings algorithm, and Gibbs sampling

The figures on the following slides are taken from the wikipedia pages for "Riemann sum", "Trapezoidal rule", "Simpson's rule", "Newton–Cotes formulas".

Numerical integration

- We consider a one dimensional integral of the form:

$$I(f) = \int_a^b f(x)dx.$$

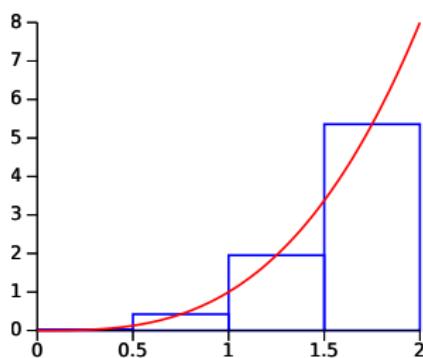
- In some cases, this integral can be evaluated numerically. In other cases, numerical approximations of $I(f)$ are needed.

Numerical integration

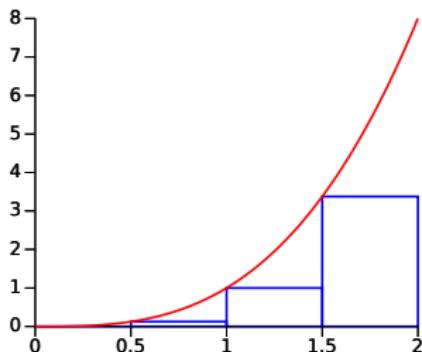
- ▶ We want to compute $I(f) = \int_a^b f(x)dx$.
- ▶ To approximate the integral, we can divide $[a, b]$ into n (equal) segments of length $h = (b - a)/n$,

$$(a, a + h, a + 2h, \dots, a + (n - 1)h, b)$$

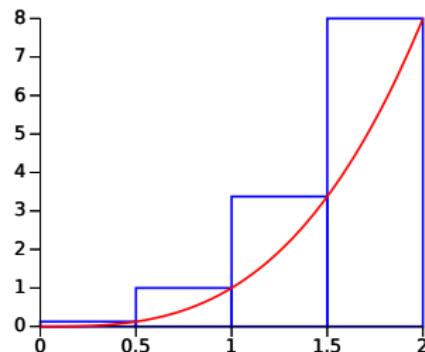
and evaluate f at each of the segment points:



Riemann sums



left Riemann sum



right Riemann sum

Assuming that f is *Riemann integrable* (in particular, continuous functions are integrable), the two *Riemann sums*

$$R_n^-(f) = \sum_{i=0}^{n-1} h \cdot f(a + ih) \quad R_n^+(f) = \sum_{i=1}^n h \cdot f(a + ih)$$

both exist as $n \rightarrow \infty$ and converge to the integral $I(f)$, where $h = (b - a)/n$.

Riemann sums

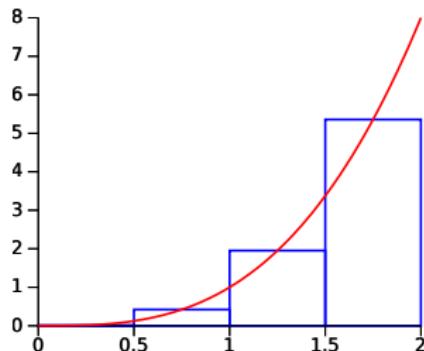
- We have seen that for an integrable function, the two Riemann sums

$$R_n^-(f) = \sum_{i=0}^{n-1} h \cdot f(a + ih) \quad R_n^+(f) = \sum_{i=1}^n h \cdot f(a + ih)$$

both exist as $n \rightarrow \infty$ and converge to the integral $I(f)$, where $h = (b - a)/n$.

- **The integral $I(f)$ can be defined as the limit of either the left or right Riemann sums.**

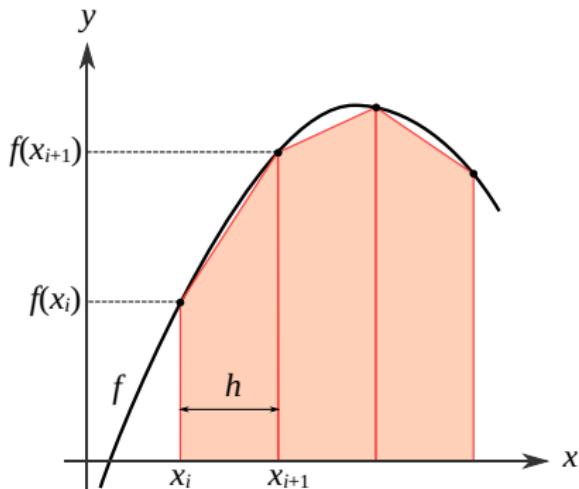
Other integration rules: Midpoint rule



- ▶ The midpoint rule approximates the integral with rectangles having a height determined by the function value at the midpoint of each grid cell.
- ▶ The area of the i th rectangle is $h \cdot f\left(a + \left(i - \frac{1}{2}\right) h\right)$.
- ▶ Thus the integral approximation is

$$I(f) = \int_a^b f(x)dx \approx \sum_{i=1}^n h \cdot f\left(a + \left(i - \frac{1}{2}\right) h\right).$$

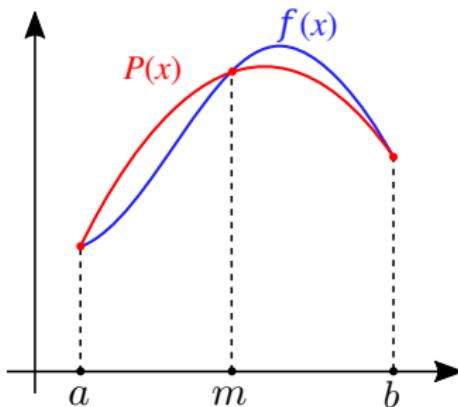
Other integration rules: Trapezoidal rule



- ▶ Each strip is approximated by a trapezoid.
- ▶ The area of the i th strip is $\frac{h}{2} \cdot [f(a + (i - 1)h) + f(a + ih)]$.
- ▶ This is equivalent to averaging left and right Riemann sums.
- ▶ Thus the integral approximation is

$$I(f) = \int_a^b f(x)dx \approx \sum_{i=1}^n \frac{h}{2} \cdot [f(a + (i - 1)h) + f(a + ih)].$$

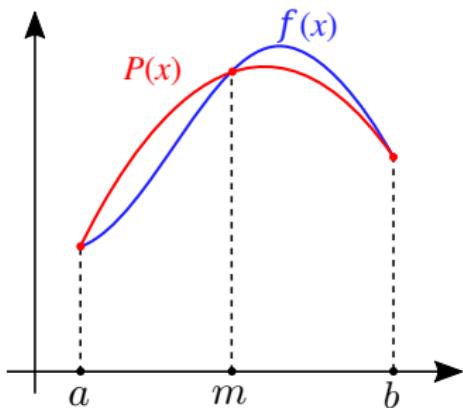
Other integration rules: Simpson's rule



- ▶ We can use arbitrary higher order polynomials (easy to integrate!) for each segment.
- ▶ Simpson's rule approximates f in $[a, b]$ with the quadratic polynomial (parabola) at a , b and $m = (a + b)/2$, leading to the approximation

$$P(x) = f(a) \frac{(x-m)(x-b)}{(a-m)(a-b)} + f(m) \frac{(x-a)(x-b)}{(m-a)(m-b)} + f(b) \frac{(x-a)(x-m)}{(b-a)(b-m)}.$$

Other integration rules: Simpson's rule



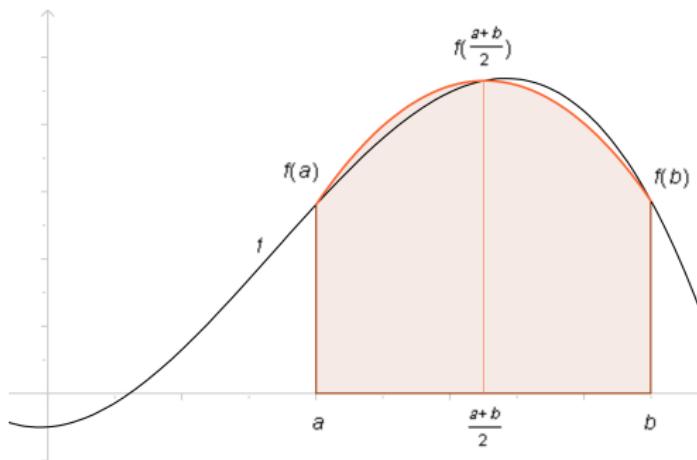
- ▶ Simpson's rule approximates f in $[a, b]$ with the parabola

$$P(x) = f(a) \frac{(x-m)(x-b)}{(a-m)(a-b)} + f(m) \frac{(x-a)(x-b)}{(m-a)(m-b)} + f(b) \frac{(x-a)(x-m)}{(b-a)(b-m)}.$$

- ▶ Using $h = (b - a)/2$ we thus obtain:

$$\int_a^b f(x) dx \approx \int_a^b P(x) dx = \frac{h}{3} [f(a) + 4f((a+b)/2) + f(b)].$$

Newton–Cotes rule



- ▶ The trapezoidal (linear) and Simpson (quadratic) rules can be generalized.
- ▶ **They are special cases of the Newton–Cotes rules for higher order polynomials.**

Newton–Cotes rule

- In particular, let x_0, \dots, x_n be given points which are symmetric around $\frac{a+b}{2}$ with distance $h = (b-a)/n$, then

$$\int_a^b f(x) dx = \sum_{i=0}^n f(x_i) \underbrace{\int_a^b l_i(x) dx}_{=w_i} = \sum_{i=0}^n f(x_i) w_i.$$

where

$$l_j(x) = \prod_{\substack{0 \leq i \leq n \\ i \neq j}} \frac{x - x_i}{x_j - x_i}$$

is the j th Lagrange basis polynomial.

- If n even then the integral is exact if f is a polynomial with degree $\leq n + 1$.
- Rules for numerical integration are referred to as **quadratures**.

Newton–Cotes rule

Depending on n , the Newton–Coates rule yields known quadratures as special cases:

degree n	step size h	formula	name
1	$b - a$	$\frac{h}{2}(f(x_0) + f(x_1))$	Trapezoid rule
2	$\frac{b-a}{2}$	$\frac{h}{3}(f(x_0) + 4f(x_1) + f(x_2))$	Simpson's rule
3	$\frac{b-a}{3}$	$\frac{3h}{8}(f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3))$	Simpson's 3/8 rule

Gaussian quadrature

- ▶ The Newton–Cotes quadrature uses equally spaced points. Thus the integral is an approximation of weighted function evaluations on an equidistant grid.
- ▶ Does using grids without equidistant grid points make sense? For example, if the integral has regions where the integrand is large it makes sense to place more grid points there.
- ▶ This approach is called *Gaussian quadrature*.

Gaussian quadrature

We consider the numerical integration of $I(f) = \int_a^b f(x)dx$.

Assuming $f(x)$ can be represented as $f(x) = f^*(x)w(x)$ for some function $w(x)$ we obtain

$$I(f) = \int_a^b f(x)dx = \int_a^b f^*(x)w(x)dx.$$

The weighted representation always exists. Given the weight function $w(x)$ on $[a, b]$, we determine the locations x_i and coefficients $w_i = w(x_i)$ such that

$$\int_a^b f(x)dx = \sum_{i=0}^n f(x_i)w_i$$

is exact for polynomials $f(x_i)$ of degree $\leq 2n - 1$. The x_i fulfilling this condition are the roots of the so-called *Legendre polynomials*.

Gaussian quadrature

Some more details on the Gaussian quadrature:

1. Transform the integral to $[-1, 1]$. This is always possible:

$$\int_a^b f(x)dx = \frac{b-a}{2} \int_{-1}^1 f\left(\frac{b-a}{2}\zeta + \frac{a+b}{2}\right) d\zeta.$$

2. Choose x_i as roots of the Legendre polynomial P_n .
3. Write $\int_{-1}^1 f(x)dx = w_0f(x_0) + \dots + w_nf(x_n)$ and solve this system for the *polynomial basis* $f(x) = 1, x, x^2, \dots, x^n$. We obtain $n+1$ linear equations for the $n+1$ unknowns w_0, \dots, w_n which we can then solve.
4. Alternatively, we can compute w_i according to the explicit formula $w_i = \frac{2}{(1-x_i^2)[P'_n(x_i)]^2}$.

Gaussian quadrature: Example

Let's consider $n = 2$, that is we will be able to integrate polynomials exactly up to degree $2n + 1 = 5$.

Calculating the roots of the Legendre polynomial P_2 yields $x_0 = -\sqrt{3/5}$, $x_1 = 0$ and $x_2 = \sqrt{3/5}$.

Gaussian quadrature: Example

Then,

$$f = 1 \quad \Rightarrow \quad \int_{-1}^1 1 dx = 2 = c_0 + c_1 + c_2$$

$$f = x \quad \Rightarrow \quad \int_{-1}^1 x dx = 0 = c_0 x_0 + c_1 x_1 + c_2 x_2$$

$$f = x^2 \quad \Rightarrow \quad \int_{-1}^1 x^2 dx = \frac{2}{3} = c_0 x_0^2 + c_1 x_1^2 + c_2 x_2^2$$

$$f = x^3 \quad \Rightarrow \quad \int_{-1}^1 x^3 dx = 0 = c_0 x_0^3 + c_1 x_1^3 + c_2 x_2^3$$

$$f = x^4 \quad \Rightarrow \quad \int_{-1}^1 x^4 dx = \frac{2}{5} = c_0 x_0^4 + c_1 x_1^4 + c_2 x_2^4$$

$$f = x^5 \quad \Rightarrow \quad \int_{-1}^1 x^5 dx = 0 = c_0 x_0^5 + c_1 x_1^5 + c_2 x_2^5$$

Gaussian quadrature: Example

Solving the previous linear equation system for c_i yields

$$c_0 = \frac{5}{9}, \quad c_1 = \frac{8}{9}, \quad c_2 = \frac{5}{9}.$$

Thus the exact integration formula for polynomials up to degree $2n + 1 = 5$ is

$$I(f) = \int_{-1}^1 f(x) dx = \frac{5}{9}f\left(-\sqrt{3/5}\right) + \frac{8}{9}f(0) + \frac{5}{9}f\left(\sqrt{3/5}\right).$$

Frequently encountered problems

There can be issues related to the integration range:

- ▶ Intervals over infinite ranges can be handled in multiple ways.
They can be transformed to finite ranges as seen before.
Common transformations are $1/x$, $\frac{e^x}{1+e^x}$, e^{-x} , and $\frac{x}{1+x}$.
- ▶ The transformation to remove an infinite range must be chosen carefully since it can introduce singularities.
- ▶ The integration must be truncated with a controlled error to avoid singularities.

Any cumulative distribution function can be used as the basis for such a transformation:

- ▶ The exponential cdf transforms the positive real line to $[0, 1]$.
- ▶ The cdf of a real valued distribution transforms doubly infinite ranges $(-\infty, \infty)$ to $[0, 1]$.

Splitting integration ranges

Splitting the integration range into many intervals is not always advantageous or viable.

- ▶ This can be numerically inefficient with many variables (higher dimensions) since we need a large number of intervals for a good approximation.
- ▶ For many parameters (higher dimensions) we need grids to evaluate the function on.
- ▶ Where a function has a peak, a finer grid is needed, and non-uniform grids will be better than uniform ones.
- ▶ Methods can become quite complicated in practice.

One alternative: Monte Carlo integration

Monte Carlo integration

We consider the previous problem of evaluating the definite integral

$$I(f) = \int_a^b f(x)dx.$$

Idea: The main idea of Monte Carlo integration is the observation that expected values of continuous random variables are integrals. Those expectations can be approximated as empirical means using draws from a random variable.

Monte Carlo integration

The idea of Monte Carlo integration is to write

$$I(f) = \int_a^b f(x)dx$$

as the expectation of a random variable.

In particular, in the simplest case, for $X \sim \text{Uniform}([a, b])$, we obtain

$$\begin{aligned} I(f) &= \int_a^b f(x)dx = (b-a) \int_a^b \frac{1}{b-a} f(x)dx \\ &= (b-a) \cdot \mathbb{E}(f(X)) \approx (b-a) \cdot \frac{1}{n} \sum_{i=1}^n f(X_i), \end{aligned}$$

where X_i for $i \in \{1, \dots, n\}$ are independent draws from X .

Monte Carlo integration

Note that the naïve Monte Carlo estimate

$$\hat{\theta} = (b - a) \cdot \frac{1}{n} \sum_{i=1}^n f(X_i)$$

of $I(f)$ with $X_i \sim \text{Uniform}([a, b])$, $i \in \{1, \dots, n\}$, is unbiased since

$$\begin{aligned}\mathbb{E}(\hat{\theta}) &= (b - a) \cdot \frac{1}{n} \sum_{i=1}^n \mathbb{E}(f(X_i)) \\ &= (b - a)\mathbb{E}(f(X)) \\ &= \int_a^b f(x)dx = I(f),\end{aligned}$$

since X, X_1, \dots, X_n are iid.

Variance of the naïve Monte Carlo estimate

The variance of $\hat{\theta}$ is

$$\begin{aligned}Var(\hat{\theta}) &= (b-a)^2 \frac{1}{m^2} \sum_{i=1}^n Var(f(X_i)) \\&= \frac{(b-a)^2}{m} Var(f(X)) \\&= \frac{b-a}{m} \int_a^b \left[f(x) - \int_a^b f(t) dt \right]^2 dx\end{aligned}$$

The variance is a measure of the *roughness* of the function f .

Monte Carlo integration

Let's compute the integral

$$\int_2^4 x^2 dx = \left[\frac{1}{3}x^3 \right]_2^4 = \frac{56}{3} \approx 18.66667.$$

In R we obtain

```
> x <- runif(10**6,min=2,max=4)
> (4-2) * mean(x**2)
[1] 18.66688
```

Monte Carlo integration

Some remarks:

- ▶ When using quadratures for numerical integration discussed earlier, such as the Newton–Coates formulas, approximations of the integrand are made for specified sub-domains of the integration range (e.g., sub-intervals). With the quadrature approach, error bounds can be stated for the integral evaluation. These bounds are typically based on derivatives of the integrand.
- ▶ There are no such approximations for a Monte Carlo quadrature, since a Monte Carlo quadrature relies on random sampling.
- ▶ In place of the (traditional) error bound, the variance of the Monte Carlo estimate (the roughness measure) is used instead to quantify the uncertainty of the estimate.

Monte Carlo integration

Some more remarks:

- ▶ For most one dimensional integrals, Monte Carlo integration is not needed but can still be applied.
- ▶ The square root of $\text{Var}(\hat{\theta})$ can be used to define confidence intervals for θ , that is, for $\mathbb{E}(\hat{\theta})$.
- ▶ **The order of the error of Monte Carlo integration is $O(1/\sqrt{n})$ where n is the number of Monte Carlo samples. This is independent of the dimension of the integral!**
- ▶ For this reason, Monte Carlo quadratures are typically the best option for multivariate integrals.

Importance sampling



Nicholas Metropolis

Importance sampling was introduced by Nicholas Constantine Metropolis in 1953. Nicholas Metropolis worked at Los Alamos on the Manhattan project, that is on the first nuclear fission processes together with Enrico Fermi (developed quantum physics) and Edward Teller (father of the hydrogen bomb).

Importance sampling

- ▶ Importance sampling, also called *biased sampling*, generates random numbers more economically.
- ▶ Instead of choosing points from a uniform distribution to evaluate a function of interest, we use a distribution which concentrates at the points where the function is large, thus focusing on the most important regions. This auxiliary distribution is called the *importance distribution*.
- ▶ Importance sampling allows to reduce the variance of the (integral) estimate. We thus speak of a *sampling technique for variance reduction*.

Importance sampling: details

We have seen the same idea previously for the special case of the uniform distribution:

$$I = \int_a^b f(x)dx = \int_a^b \frac{f(x)}{g(x)}g(x)dx = \mathbb{E}_X \left(\frac{f(X)}{g(X)} \right)$$

for $X \sim g$.

- ▶ We can interpret $\int_a^b \frac{f(x)}{g(x)}g(x)dx$ as the expected value of $f(X)/g(X)$ for some random variable X drawn from the importance distribution g .
- ▶ So we essentially exchange the distribution used to compute the integral from f to g . The distribution g should be chosen to closely resemble f in order to sample f most accurately.

Importance sampling: details

We rewrite the integral $\int_a^b f(x)dx$ as

$$\int_a^b f(x)dx = \int_a^b \frac{f(x)}{g(x)}g(x)dx = \mathbb{E}_X \left(\frac{f(X)}{g(X)} \right) \approx \frac{1}{n} \sum_{i=1}^n \frac{f(x_i)}{g(x_i)}$$

which is then approximated using independent samples x_1, \dots, x_n from g .

Remark: If g closely resembles f then f/g is close to one and thus the variance of the integrand is a lot smaller than the one obtained if we sampled f with draws from a uniform distribution. To summarize, sampling f with draws from a non-uniform g is more efficient than naïve MC approximation.

Algorithm: importance sampling

We want to obtain random draws from a distribution f .

1. Choose the proxy g to sample from which should resemble f .
2. Generate samples x_1, \dots, x_n from g .
3. Compute the Monte Carlo estimate

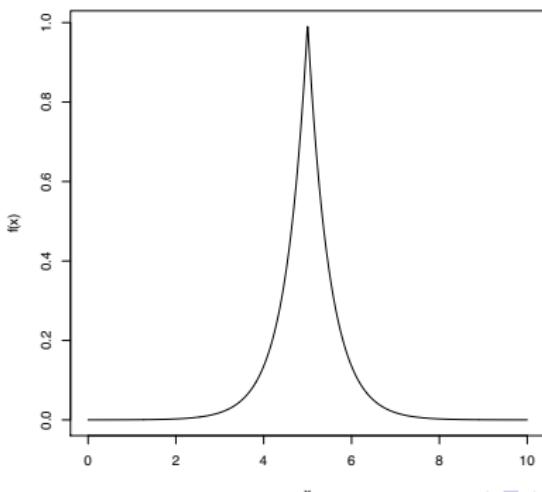
$$\int_a^b f(x)dx \approx \frac{1}{n} \sum_{i=1}^n \frac{f(x_i)}{g(x_i)}.$$

Importance sampling: examples

Example 1: Suppose we want to evaluate

$$I = \int_0^{10} e^{-2|x-5|} dx,$$

that is $f(x) = e^{-2|x-5|}$. We do not know the analytic expression for this integral.

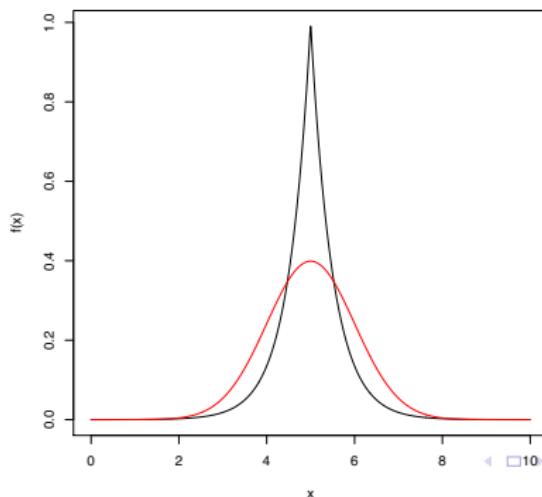


Importance sampling: examples

As a proxy we use a normal distribution with mean $\mu = 5$ and standard deviation $\sigma = 2$, thus the importance distribution is

$$g(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right).$$

The Monte Carlo estimate will be $\frac{1}{n} \sum_{i=1}^n \frac{f(x_i)}{g(x_i)}$, where x_1, \dots, x_n are samples from $N(\mu = 5, \sigma = 2)$.



Importance sampling: examples

In R we obtain

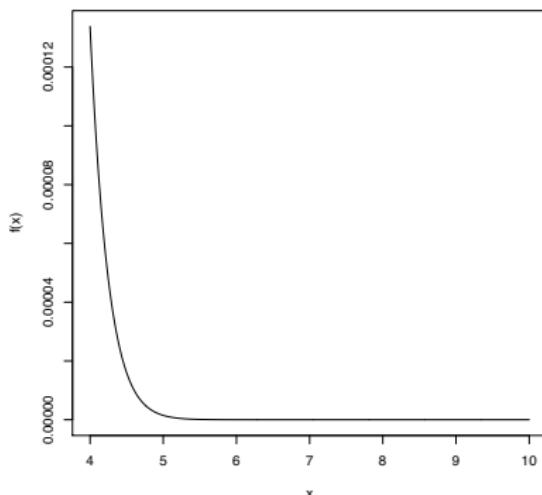
```
> f <- function(x) exp(-2*abs(x-5))
> n <- 10**6
> x <- rnorm(n, mean=5, sd=2)
> mean(f(x)/dnorm(x,mean=5,sd=2))
[1] 1.00014
```

Importance sampling: examples

Example 2: Suppose we want to evaluate

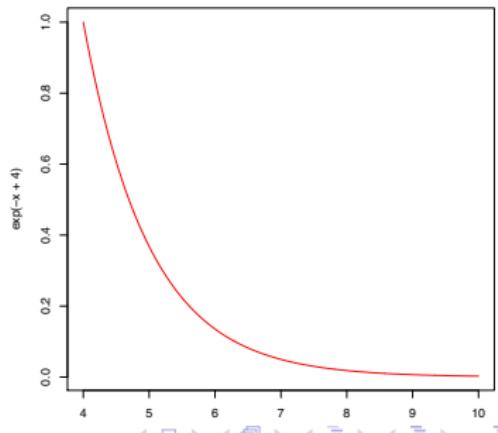
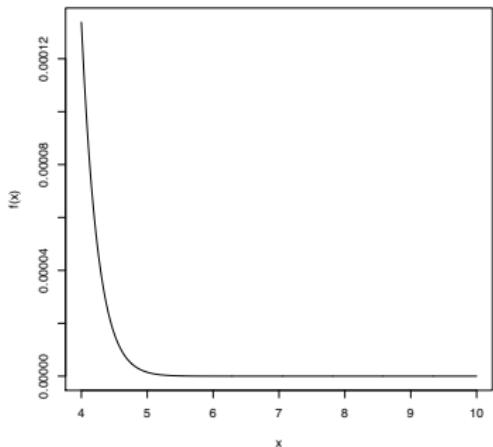
$$I = \int_4^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx,$$

that is a truncated normal distribution on the interval $[4, \infty)$. With R we obtain $I = 1 - \text{pnorm}(4) = 3.167124 \cdot 10^{-5}$.



Importance sampling: examples

Looking at the normal distribution in $[4, \infty)$ we see that its density is very small (left plot). Thus if we use another normal as a proxy to sample from, and discard values outside $[4, \infty)$, we will hardly get any valid samples. A better idea is to use a shifted exponential distribution to $[4, \infty)$ with shape parameter $\lambda = 1$, having the density e^{-x+4} : sampling from it will produce most samples around 4, and the exponential is likewise defined on $[4, \infty)$ (right plot).



Importance sampling: examples

In R we obtain

```
> 1-pnorm(4)
[1] 3.167124e-05
> n <- 10**6
> x <- 4 + rexp(n, rate=1)
> mean(dnorm(x)/exp(-x+4))
[1] 3.164051e-05
```

Variance reduction

We look at the variance reduction we obtain in the second example by sampling from the exponential.

First, we could have naïvely sampled from a normal distribution and counted the proportion in $[4, \infty)$. Using $n = 10^6$ we obtain the proportion $\theta_1 = 3.167124 \cdot 10^{-5}$ having the variance

$$\text{Var}(\theta_1) = \frac{\theta_1(1 - \theta_1)}{n} = 3.1669 \cdot 10^{-11}.$$

Second, the importance sampling estimator is

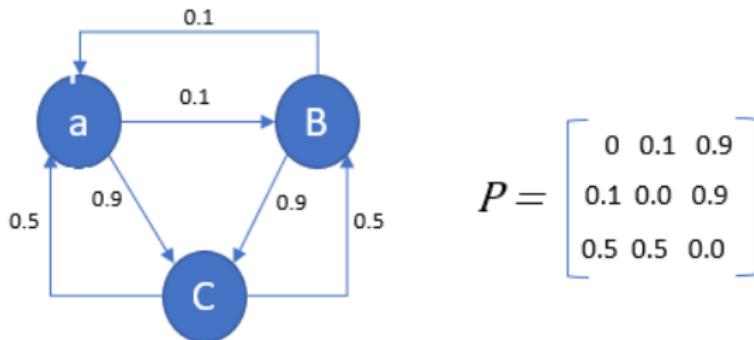
$$\theta_2 = \frac{\frac{1}{\sqrt{2\pi}} e^{-x^2/2}}{e^{-x+4}}.$$

We do not have a closed form expression for the variance of θ_2 . However, simulation gives $\text{Var}(\theta_2) = 1.488178 \cdot 10^{-15}$. This is a reduction in variance of around a factor 20000.

Importance sampling: final remarks

- ▶ Importance sampling provides a method for integration that can result in estimators with a substantial reduction in variance compared to the naïve Monte Carlo sampling estimator.
- ▶ Importance sampling is not a panacea: a poor choice of the importance distribution can result in very poor convergence properties of the method and a high variance of the resulting estimator.
- ▶ It is advised to look at the variance of the resulting estimator when assessing the appropriateness of a particular importance function.
- ▶ When the ratio of f/g is unbounded, too much weight (=importance) is given to a few values of x , leading to an increase in variance.

Markov Chain Monte Carlo (MCMC)



- ▶ MC integration from the previous slides is preferable when the number of dimensions is high. However, we want to find a general way to sample from an arbitrary distribution π .
- ▶ MCMC allows us to sample from a distribution π by generating a Markov chain whose states are visited with the desired distribution π .

What is a Markov chain?

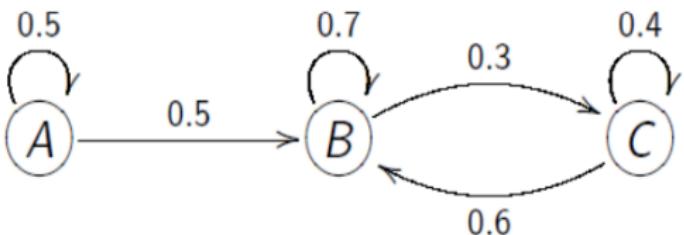
Definition: A Markov chain is a stochastic process in which future states are independent of past states given the present state.

Mathematically, the *Markov property* holds: The Markov property refers to the memoryless property of a stochastic process, that is

$$\mathbb{P}(X_{n+1} = x_{n+1} | X_n = x_n, \dots, X_0 = x_0) = \mathbb{P}(X_{n+1} = x_{n+1} | X_n = x_n).$$

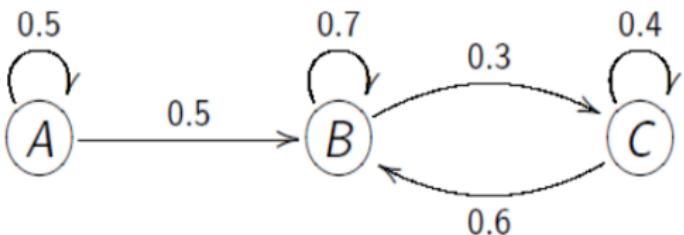
That is, the probability that the next state X_{n+1} takes some value x_{n+1} , given the entire history of the chain, is the same as if we only knew the current state $X_n = x_n$.

Some properties of Markov chains



A Markov chain is **irreducible** if it is possible to go from any state to any other state in n steps for some $n \in \mathbb{N}$. The chain above is *reducible* but not irreducible, since state A cannot be reached again once the chain leaves A .

Some properties of Markov chains



A state i of a Markov chain is **periodic** if any return to state i must occur in multiples of k time steps. Formally, the period of a state i is defined as

$$k = \gcd\{n > 0 : \mathbb{P}(X_n = i | X_0 = i) > 0\}.$$

If $k = 1$ then the state is called **aperiodic**. A **Markov chain is aperiodic** if all its states are aperiodic.

Some properties of Markov chains

A Markov chain is **recurrent** if for any state i , given the chain starts in i , it will eventually return to i with probability 1.

A Markov chain is **positive recurrent** if the expected return time to state i is finite. Otherwise, it is called **null recurrent**.

A distribution π is a **stationary distribution** of a Markov chain with transition probabilities p_{ij} (from state i to j) if $\pi_j = \sum_i \pi_i p_{ij}$. In words, π describes the limiting distribution with which all states are visited as $n \rightarrow \infty$.

Some properties of Markov chains

Denote the *transition kernel* determining the distribution on the states of Markov chain as $\mathbb{P}(x \rightarrow x')$. Then the **detailed balance** condition holds if

$$\pi(x)\mathbb{P}(x \rightarrow x') = \pi(x')\mathbb{P}(x' \rightarrow x).$$

Detailed balance says that each transition $x \rightarrow x'$ is reversible. For every pair of states x, x' the probability of being in state x and transitioning to state x' must be equal to the probability of being in state x' and transitioning to state x .

Construct Markov chains for a given target distribution

Goal: We want to construct a Markov chain such that the stationary distribution of the process (in the limit $n \rightarrow \infty$) is a given target distribution π that we specify.

Solution: Metropolis–Hastings algorithm

N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller (1953). Equations of State Calculations by Fast Computing Machines, *Journal of Chemical Physics*, 21:1087–1092.

The method was later generalized by Hastings (1970). This marked the beginning of *Markov Chain Monte Carlo* (MCMC).

Construct Markov chains for a given target distribution

The trick lays in the detailed balance equation:

- ▶ The detailed balance condition is a sufficient (but not necessary) condition for the existence of a stationary distribution π .
- ▶ If the chain is *ergodic* (it is aperiodic and positive recurrent), then π is unique.
- ▶ If detailed balance $\pi(x)\mathbb{P}(x \rightarrow x') = \pi(x')\mathbb{P}(x' \rightarrow x)$ is satisfied for a chain for some π , where $\mathbb{P}(x \rightarrow x')$ is the transition probability of the chain, then π is its stationary distribution.

Metropolis–Hastings: idea

The idea of Metropolis–Hastings is to enforce detailed balance!

The algorithm works like this:

We want to draw samples from a distribution $P(x)$. We only need to know P up to its normalizing constant. Since we cannot draw from P , we propose draws from a *proposal distribution* $g(x'|x)$ which we can sample from. We compute some *acceptance probability* $\alpha(x'|x)$ and accept the proposed sample x' from g with probability $\alpha(x'|x)$ or else discard it.

- ▶ **The accepted samples from g will behave like a sample drawn from P !**
- ▶ *The acceptance probability acts like the transition kernel for this chain.*

Metropolis–Hastings: algorithm

Detailed algorithm:

1. At $t = 0$, initialize a random state x_0 .
2. For $t = 1, 2, \dots$ do:
 - 2.1 Generate a sample x' from the proposal distribution $g(x'|x_{t-1})$.
 - 2.2 Compute the acceptance probability

$$\alpha(x'|x_{t-1}) = \min \left\{ 1, \frac{P(x')g(x_{t-1}|x')}{P(x_{t-1})g(x'|x_{t-1})} \right\}.$$

- 2.3 Accept x' with probability $\alpha(x'|x_{t-1})$, that is set $x_t := x'$.
Otherwise discard x' and set $x_t := x_{t-1}$.

Remark: The acceptance is realized using a draw $U \sim \text{Uniform}(0, 1)$: If $U < \alpha(x'|x_{t-1})$ then x' is accepted. Otherwise it is discarded.

Why does Metropolis–Hastings work?

- ▶ For simplicity, we assume that the proposal distribution is symmetric. Such a proposal is called a *Metropolis proposal*.
- ▶ Among others, the popular choice $g(x'|x) = N(x', \mu = x, \sigma^2)$, that is a Normal distribution centered at the current x for some variance σ^2 is symmetric.
- ▶ In this case, we have

$$\alpha(x'|x_{t-1}) = \min \left\{ 1, \frac{P(x')}{P(x_{t-1})} \right\},$$

where $x' \sim g(x'|x_{t-1})$.

Why does Metropolis–Hastings work?

- ▶ Multiplying $\alpha(x'|x_{t-1})$ with $P(x_{t-1})$ yields

$$\alpha(x'|x_{t-1})P(x_{t-1}) = \min \{ P(x_{t-1}), P(x') \}.$$

- ▶ If we switch x' and x_{t-1} we get

$$\alpha(x_{t-1}|x')P(x') = \min \{ P(x'), P(x_{t-1}) \}.$$

- ▶ Naturally, the two right hand sides are the same, thus

$$\alpha(x'|x_{t-1})P(x_{t-1}) = \alpha(x_{t-1}|x')P(x').$$

Why does Metropolis–Hastings work?

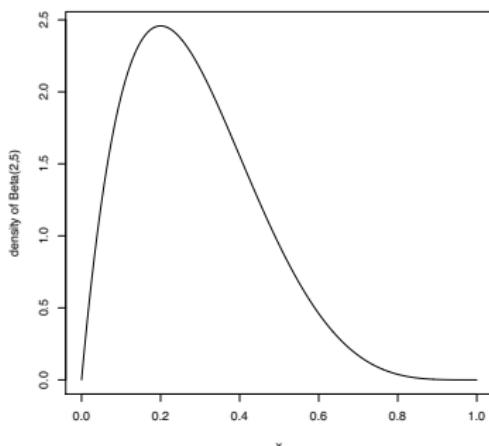
- ▶ We have seen that

$$\alpha(x'|x_{t-1})P(x_{t-1}) = \alpha(x_{t-1}|x')P(x').$$

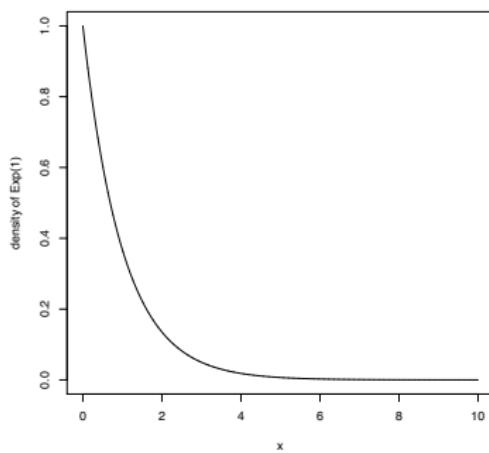
- ▶ **Consequence:** The above equation is the detailed balance condition for the chain that moves from x_{t-1} to x' with transition kernel $\alpha(x'|x_{t-1})$, which is exactly what the algorithm does. Since detailed balance holds, $P(\cdot)$ will be the stationary distribution of the chain, which is the distribution we specified to sample from!

Example of Metropolis–Hastings

Suppose we want to sample from a $\text{Beta}(2, 5)$ distribution which we cannot handle with the inversion method. We do know though how to sample from an exponential distribution $\text{Exp}(1)$, for instance via the inversion method. The exponential distribution is defined on $[0, \infty)$ but this will not be a problem.



Beta(2, 5) distribution defined on $[0, 1]$



Exp(1) distribution on $[0, \infty)$

Example of Metropolis–Hastings

```
nmax <- 20000 #first 10000 samples are discarded as burn-in
x <- rep(0,nmax)
x[1] <- runif(1)
counter <- 1

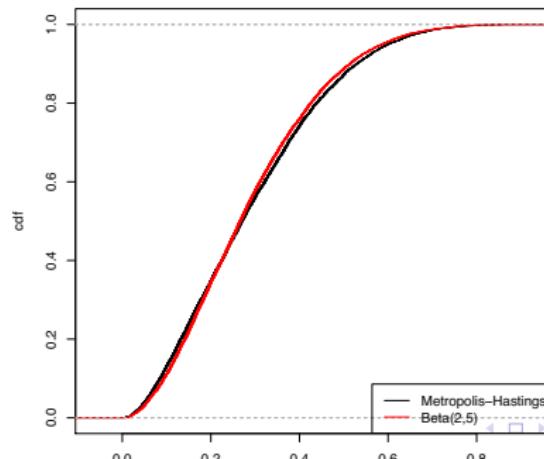
while(counter<nmax) {
  # proposal
  xnew <- rexp(1,rate=1)
  # acceptance probability
  pxnew <- dbeta(xnew,shape1=2,shape2=5)
  px <- dbeta(x[counter],shape1=2,shape2=5)
  gx <- dexp(x[counter],rate=1)
  gxnew <- dexp(xnew,rate=1)
  a <- min(1, pxnew/px * gx/gxnew )
  if(runif(1)<a) {
    counter <- counter+1
    x[counter] <- xnew
  }
}

plot(ecdf(x[10001:20000]), xlab="x", ylab="cdf", main="")
lines(ecdf(rbeta(10000,shape1=2,shape2=5)),col="red")
legend("bottomright",col=c("black","red"),lty=1,
       legend=c("Metropolis-Hastings","Beta(2,5)"))
```

Example of Metropolis–Hastings

When running the code from the last slide, around 52000 proposals from the exponential distribution had to be generated to produce $2 \cdot 10^4$ samples, that is the acceptance rate was around 28%.

We plot (a) the empirical CDF of the vector $x[10001 : 20000]$, that is the last 10^4 accepted samples of Metropolis–Hastings after the burn-in (black) and (b) the empirical CDF of 10^4 actual Beta(2, 5) samples (red) from `rbeta(10000, shape1=2, shape2=5)`:



One special case: Gibbs sampling

Task: Suppose that $x = (x_1, \dots, x_n)$ is n -dimensional with $n \geq 2$. We want to draw samples from some joint distribution

$$\pi(x_1, \dots, x_n).$$

Prerequisite: We assume we can sample from the conditional distribution of each x_i , that is from

$$\pi(x_i | x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n).$$

One special case: Gibbs sampling

- ▶ In particular, Gibbs sampling updates in turn all variables in each new time step $t + 1$.
- ▶ Each variable is drawn from its conditional distribution, where all other variables are fixed at their most recent values.
- ▶ Starting with some initial values $(x_1^{(0)}, \dots, x_n^{(0)})$ we update:

$$x_1^{(t+1)} \sim \pi(x_1 | x_2^{(t)}, x_3^{(t)}, \dots, x_n^{(t)})$$

$$x_2^{(t+1)} \sim \pi(x_2 | x_1^{(t+1)}, x_3^{(t)}, \dots, x_n^{(t)})$$

$$x_3^{(t+1)} \sim \pi(x_3 | x_1^{(t+1)}, x_2^{(t+1)}, x_4^{(t)}, \dots, x_n^{(t)})$$

...

$$x_n^{(t+1)} \sim \pi(x_n | x_1^{(t+1)}, \dots, x_{n-1}^{(t+1)})$$

Example of Gibbs sampling

We want to draw from the joint distribution

$$p(x, y) \propto e^{-xy}.$$

Then $p(x|y)$ is $\text{Exp}(y)$ and $p(y|x)$ is $\text{Exp}(x)$, that is both conditional distributions are exponential distributions with the other variable as parameter.

Starting with $(x^{(0)}, y^{(0)})$ we iterate:

- ▶ Draw $x^{(1)}$ from $\text{Exp}(y^{(0)})$ and $y^{(1)}$ from $\text{Exp}(x^{(1)})$.
- ▶ Draw $x^{(2)}$ from $\text{Exp}(y^{(1)})$ and $y^{(2)}$ from $\text{Exp}(x^{(2)})$.
- ▶ ...

After a burn-in period, the resulting pairs $(x^{(n)}, y^{(n)})_{n \in \mathbb{N}}$ will behave like a sample from $p(x, y)$.

Optimization

Contents:

- ▶ Different types of optimization methods
- ▶ Unconstrained and constrained optimization
- ▶ Karush-Kuhn-Tucker conditions
- ▶ Line search and steepest descent
- ▶ Saddle points, valleys, collinearity
- ▶ Ridge regression
- ▶ Newton-Raphson in higher dimensions
- ▶ Quasi-Newton methods
- ▶ Conjugate gradient method
- ▶ Probabilistic optimization, especially simulated annealing

Good resource for further reading:

- ▶ <https://www.deeplearningbook.org/>

The optimization problem

- ▶ Most statistical models, machine learning approaches or deep learning algorithms require the optimization of an **objective function** f over some space M .
- ▶ Without loss of generality we assume we want to minimize f , thus we consider

$$\min_{x \in M} f(x).$$

- ▶ The function f is also referred to as the **cost function**, **loss function**, **error function** or **criterion function**.
- ▶ Minimization and maximization are equivalent problems. The maximization of f is equivalent with the minimization of $-f$.

The optimization problem

- We denote the value that minimizes f by x^* :

$$x^* = \arg \min_{x \in M} f(x).$$

- The set M will usually be a subset of \mathbb{R}^N , but it can also be a discrete set.
- If M is a set of continuous values then we are dealing with a problem in **continuous optimization**.
- If M contains discrete values we speak of **combinatorial optimization**. One example is the estimation of the parameters of a hypergeometric distribution. This topic, although important, is not covered here.

Examples

Many fundamental problems in statistics, physics, etc. can be expressed as the problem of *finding $x \in M$ such that some error, loss, bias etc. is minimized*. For instance,

- ▶ estimation of parameters in statistical models
- ▶ maximum-likelihood estimation
- ▶ least-squares approaches
- ▶ machine-learning and deep-learning approaches
- ▶ physics
- ▶ all fields of engineering
- ▶ operation research
- ▶ economics
- ▶ material science
- ▶ ...

Central role of optimization

Optimization plays a central role in statistics, data science, machine learning and many more areas.

Many machine learning problems are formulated as the minimization of some loss function on a training set of examples. The loss functions expresses the discrepancy between the predictions of the model being trained and the actual problem instances. Optimization algorithms can then minimize this loss.
(Wikipedia)

Example: We train a classifier to distinguish between cats and dogs on a training set of photos which have been labeled by humans first.

Types of optimization methods

We can consider three types of optimization types:

- ▶ **Exhaustive search:** If the total number of distinct permissible parameter combinations, then it might be feasible to simply iterate through all of them and evaluate the objective function on each combination to determine the optimum.
- ▶ **Branch and bound:** In some cases, it is feasible to subdivide the search space. If additionally one can compute bounds on the value the objective function will take on each partition, one might be able to exclude parts of the search space without searching them.
- ▶ **Probabilistic methods:** Probabilistic methods such as simulated annealing can be used if deterministic ones do not work, but typically require a lot longer.

Types of direct optimization methods

Newton-type methods require the derivative of the function to be optimized. Probabilistic methods are gradient free (and Hessian free), and so are direct search methods:

- ▶ **Grid search:** We sample the search space on a regular or irregular grid and evaluate the objective function at every grid point. We take the minimal value we find as the minimum, and possibly refine the search space around it. Grid search is not particularly efficient, but it is a good starting point to get a general idea of the objective function, for instance to find good starting points for other methods. Plotting a function is technically a form of grid search.
- ▶ **Shotgun search:** Similar idea to grid search, but uses random points.

Some direct search algorithms (when applied in higher dimensions) use one dimensional optimization as a subroutine for finding a promising direction or step size.

Role of smooth functions

- ▶ Numerical optimization cannot test a function at all possible values. If we probe a function at selected points, we (implicitly) assume that the function does not change too much in a neighborhood of those points, thus the function needs to be *smooth*.
- ▶ **Definition:** A smooth function is a function with continuous derivative in some domain. The degree of smoothness of a function is the number of continuous derivatives it has in some domain. We write $f \in C^k$ if the derivatives $f^{(1)}, \dots, f^{(k)}$ exist and are continuous.

Multivariate optimization

- ▶ Many (statistical) problems involve multiple parameters, for instance a likelihood function for a regression in two or more parameters. The derivative of the log-likelihood is not one dimensional, but leads to a multivariate gradient (called a vector valued score function).
- ▶ At the optimum, the vector-valued gradient is the zero vector.
- ▶ Many methods are available to solve these systems.
These are mostly generalizations of one dimensional methods.
Bracketing methods for single equations do not have multivariate extensions.

Unconstrained and constrained optimization problems

We now look at arbitrary optimization problems defined through a function f . There are two cases:

1. The unconstrained optimization problem

$$\min_{x \in \mathbb{R}^N} f(x).$$

2. The constrained optimization problem

$$\min_{x \in S} f(x) \quad \text{with} \quad S \subsetneq \mathbb{R}^N.$$

Unconstrained optimization

- ▶ Given a smooth function $f : \mathbb{R}^N \rightarrow \mathbb{R}$ we seek $x^* \in \mathbb{R}^N$ such that

$$f(x^*) = \min_{x \in \mathbb{R}^N} f(x).$$

- ▶ Possible problems:
 - ▶ x^* is not guaranteed to exist or be unique.
 - ▶ There may be one or more local minima in addition to the global minimum.
 - ▶ The extremal points (where $\nabla f = 0$) may be saddle points or maxima rather than minima.

Constrained optimization

- ▶ Constrained optimization problems are often encountered in applications in biostatistics, big data, etc.
- ▶ The **Karush-Kuhn-Tucker (KKT)** formalism provides a general solution/ framework to solve constrained optimization problems.
- ▶ The idea is to transform the constrained optimization problem into an unconstrained problem using the **generalized Lagrangian function**, that is we append the constraints to the objective function using *Lagrange multipliers*.

Examples of constrained optimization

- ▶ Least-Square regression

$$\min_{x \in \mathbb{R}^N} \|Ax - b\|^2,$$

where $A \in \mathbb{R}^{M \times N}$, $b \in \mathbb{R}^M$, $M \geq N$.

- ▶ Ridge regression:

$$\min_{x \in \mathbb{R}^N} \|Ax - b\|^2 \quad \text{subject to} \quad \sum_{i=1}^N x_i^2 \leq t,$$

where $A \in \mathbb{R}^{M \times N}$, $b \in \mathbb{R}^M$.

- ▶ Lasso regression:

$$\min_{x \in \mathbb{R}^N} \|Ax - b\|^2 \quad \text{subject to} \quad \sum_{i=1}^N |x_i| \leq t,$$

where $A \in \mathbb{R}^{M \times N}$, $b \in \mathbb{R}^M$.

The Karush-Kuhn-Tucker conditions

The KKT formalism allows us to add constraints to the objective function which are given by either equalities or inequalities.

Let's assume we aim to optimize

$$\min_{x \in S} f(x)$$

subject to a region S which can be parameterized by the following conditions

- ▶ $g^{(i)}(x) = 0$ for all $i \in \{1, \dots, p\}$
- ▶ $h^{(j)}(x) \leq 0$ for all $j \in \{1, \dots, q\}$.

That is, we need to be able to express S as

$$S = \left\{ x : g^{(i)}(x) = 0 \ \forall i \text{ and } h^{(j)}(x) \leq 0 \ \forall j \right\}.$$

The Karush-Kuhn-Tucker conditions

We assume we want to optimize

$$\min_{x \in S} f(x)$$

subject to

$$S = \left\{ x : g^{(i)}(x) = 0 \ \forall i \text{ and } h^{(j)}(x) \leq 0 \ \forall j \right\}.$$

Then, under certain regularity conditions, the solution is given by the optimization of the unconstrained generalized Lagrangian:

$$\begin{aligned} & \min_x \max_{\lambda_1, \lambda_2, \dots} \max_{\alpha_1, \alpha_2, \dots > 0} L(x, \lambda_1, \lambda_2, \dots, \alpha_1, \alpha_2, \dots) \\ &= \min_x \max_{\lambda_1, \lambda_2, \dots} \max_{\alpha_1, \alpha_2, \dots > 0} \left(f(x) + \sum_{i=1}^p \lambda_i g^{(i)}(x) + \sum_{j=1}^q \alpha_j h^{(j)}(x) \right). \end{aligned}$$

Example: Lasso regression

We have seen that the Lasso regression problem is given by

$$\min_{x \in \mathbb{R}^N} \|Ax - b\|^2 \quad \text{subject to} \quad \sum_{i=1}^N |x_i| \leq t,$$

where $A \in \mathbb{R}^{M \times N}$, $b \in \mathbb{R}^M$.

Using a Lagrange multiplier λ for the single constraint $g(x) = \sum_{i=1}^N |x_i|$, we obtain

$$\min_{x \in \mathbb{R}^N} \left(\|Ax - b\|^2 + \lambda \sum_{i=1}^N |x_i| \right).$$

What is a solution?

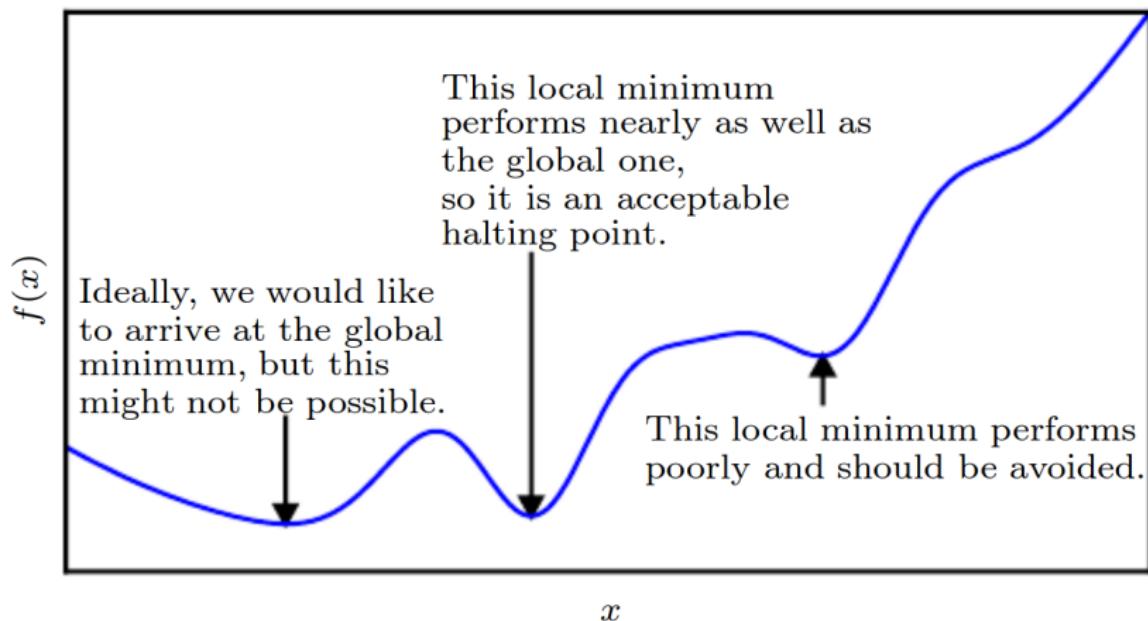
- ▶ A point x^* is a global minimizer (or global minimum) if

$$f(x^*) \leq f(x) \quad \forall x \in \mathbb{R}^N.$$

- ▶ A point \tilde{x} is a local minimizer (or local minimum) if there is a neighborhood $U \subsetneq \mathbb{R}^N$ of \tilde{x} such that

$$f(\tilde{x}) \leq f(x) \quad \forall x \in U.$$

Examples: Local minimum versus global minimum



Plot from "Deep learning" (chapter 4) by Goodfellow et al. (MIT).

A function may have multiple local minima in addition to the global minimum, and also multiple (local and global) maxima.

Necessary conditions for local extrema

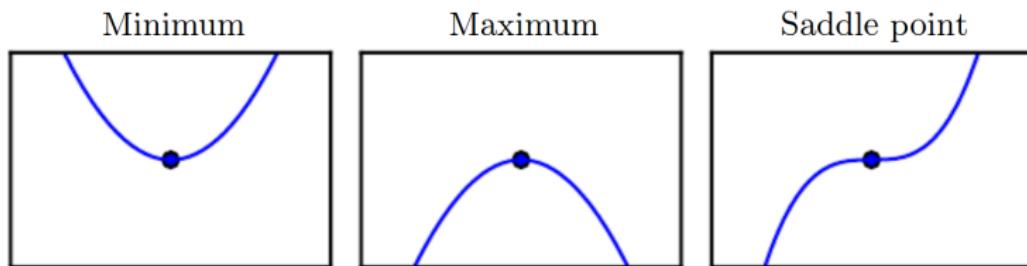
- ▶ If \tilde{x} is a local minimizer of f and $\nabla^2 f$ exists and is continuously differentiable in an open neighborhood of \tilde{x} , then

$$\nabla f(\tilde{x}) = 0 \quad \text{and} \quad \nabla^2(\tilde{x}) \quad \text{is positive definite.}$$

- ▶ \tilde{x} is called a stationary point if $\nabla f(\tilde{x}) = 0$.

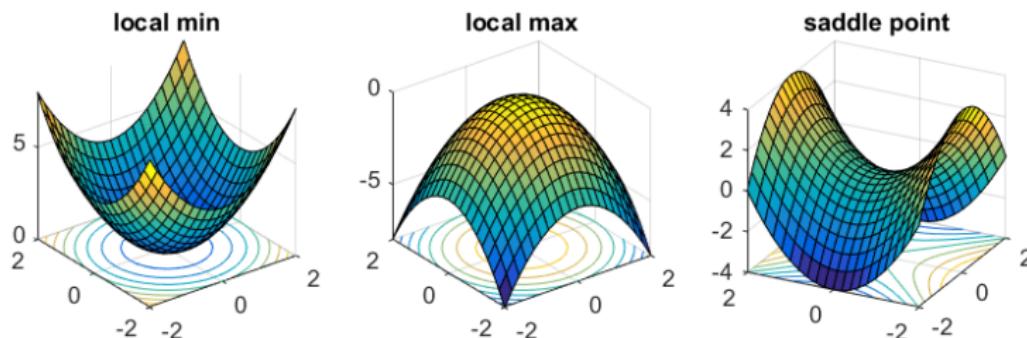
Example of stationary points

In one dimension:



Maximum, minimum and saddle point. Plot from "Deep learning" (chapter 4)
by Goodfellow et al. (MIT).

In two dimensions:

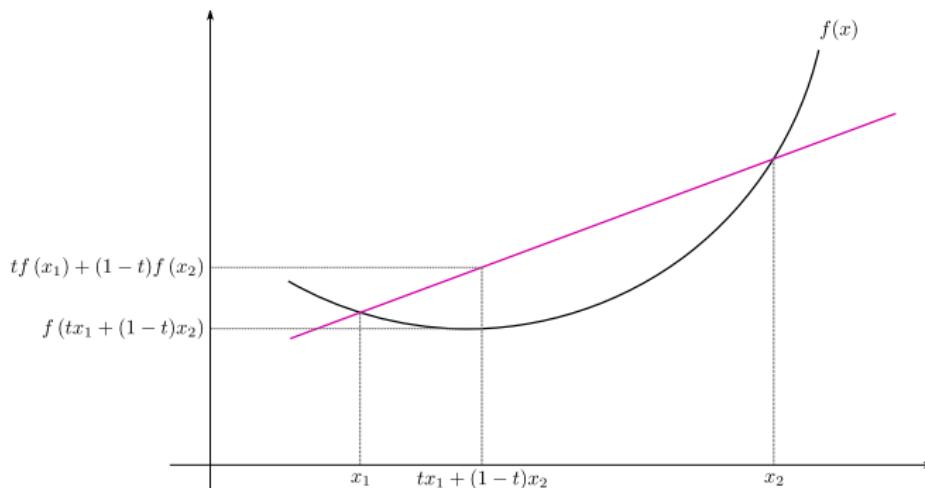


Convex function f

Definition: A function $f : \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in \mathbb{R}^N$ it holds true that

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2) \quad \forall t \in [0, 1].$$

In words: *The function f stays below the line connecting any two points on it.*



Plot taken from wikipedia, article "Convex function"

Features of convex functions

Convex functions have the following two important properties:

- ▶ When f is convex, any local minimizer (minimum) x^* is a global minimizer (minimum) of f .
- ▶ In addition, if f is differentiable, then any stationary point x^* is a global minimizer (minimum) of f .

Algorithm for numerical optimization

Examples for important convex loss functions f :

- ▶ least squares objective function
- ▶ Ridge regression
- ▶ Lasso regression

It is important to note that the popularity of e.g. the Lasso regression is also due to the fact that this objective function can be "easily" optimized, and that a global minimum can easily be found.

However, in general, we will not be that lucky. Many machine learning tasks are non-convex, e.g. maximum likelihood estimation of latent variable models or training multi-layer neural networks.

How to find solutions to an optimization problem?

Setting: Suppose we are given the objective function of some optimization problem.

→ If the first order derivative of f exists, we know that we can find a local minimum/maximum and saddle points by solving

$$\nabla_x f(x) = 0.$$

If the second order derivative of f exists, i.e. $\nabla^2 f(x)$, the extremal point can be characterized as local minimum/ maximum or saddle point.

What are the issues with this approach?

- ▶ Typically no analytical solution, need numeric method.
- ▶ A solution of $\nabla_x f(x) = 0$ does not guarantee that we found a global minimum (or maximum).

Possible strategies for optimization algorithms

How can we find the minimum of an arbitrary function f ?

- ▶ Line search methods
- ▶ Trust region methods

We have a look at both strategies...

Possible strategies for optimization algorithms

Line-search methods:

- ▶ Given a point $x_k \in \mathbb{R}^N$, find a descent direction $p_k \in \mathbb{R}^N$.
- ▶ Find the step length $\alpha_k \in \mathbb{R}$ which minimizes

$$\min_{\alpha_k} f(x_k + \alpha_k \cdot p_k).$$

- ▶ The next point then becomes $x_{k+1} = x_k + \alpha_k \cdot p_k$.
- ▶ Need stopping rule/termination criterion.

Possible strategies for optimization algorithms

Trust-region methods:

- ▶ For a function f , construct a model function $m_k : \mathbb{R}^N \rightarrow \mathbb{R}$.
- ▶ Define a trust region $R(x_k)$ inside which $f \approx m_k$.
- ▶ Solve the minimization problem:

$$\min_{p \in \mathbb{R}^N : x_k + p \in R(x_k)} m_k(x_k + p).$$

- ▶ Set next estimate to $x_{k+1} = x_k + p_k$.
- ▶ Need stopping rule/termination criterion.

Methods to find descent directions for line search

- ▶ Method 1: Steepest descent method
 - ▶ First order approximation
 - ▶ Linear convergence (global convergence)
- ▶ Method 2: Newton's method
 - ▶ Second order approximation
 - ▶ Quadratic convergence (local convergence)
- ▶ Method 3: Conjugate Gradient (CG) method

Again we will have a look at all three options...

The steepest descent method

The steepest descent method (first order approximation) chooses the direction p such that $f(x_k + p)$ is minimized.

Question: How to choose p ?

For this let's look at Taylor's expansion of f at x_k ,

$$f(x_k + p) \approx f(x_k) + p^\top \nabla f(x_k).$$

We need to minimize $p^\top \nabla f(x_k)$ under the condition $\|p\| = 1$:

$$\begin{aligned} p^\top \nabla f(x_k) &= \|p\| \cdot \|\nabla f(x_k)\| \cdot \underbrace{\cos(\theta(p, \nabla f(x_k)))}_{\leq -1} \\ \Rightarrow p_k &= \frac{-1}{\|\nabla f(x_k)\|} \nabla f(x_k) \end{aligned}$$

Now that we know the direction p_k , we select its multiplier $a_k = \arg \min_{\alpha_k} f(x_k + \alpha_k \cdot p_k)$. However, this approach will only work as long as $p^\top \nabla f(x_k) < 0$. Why is this important?

Termination criteria for line search

- ▶ Suppose f is a smooth objective function, $N = 1$ and x^* is a local minimum. Since $\nabla f(x^*) = 0$, the Taylor expansion yields

$$f(x^* + \delta x) \approx f(x^*) + \frac{1}{2}(\delta x)^2 \nabla^2 f(x^*).$$

- ▶ The perturbation in f caused by a perturbation δx in x^* is

$$\delta f = f(x^* + \delta x) - f(x^*) \approx \frac{1}{2}(\delta x)^2 \nabla^2 f(x^*).$$

- ▶ Therefore, if f is within δf of its minimum, we can expect x to be within

$$\delta x \approx \sqrt{\frac{2\delta f}{\nabla^2 f}}$$

of the optimal value x^* .

Termination criteria for line search

- ▶ **Conclusion:** If we can compute $f(x)$ up to a relative precision of ε_{eps} , the corresponding precision of x is of order $\sqrt{\varepsilon_{\text{eps}}}$.
- ▶ That is, we cannot expect to compute the location of the optimum to anywhere near the machine precision, but only its square root. Thus, numerical optimization algorithms usually include the location termination criterion

$$\|x_{k+1} - x_k\| < \text{xtol},$$

where xtol is set to a few tens or hundreds times $\sqrt{\varepsilon_{\text{eps}}}$.

Additional termination criteria for line search

- ▶ **Closeness of the function value at the optimum:**

$$\|f(x_{k+1}) - f(x_k)\| < \text{ftol},$$

where ftol is a few tens or hundreds times of ε_{eps} .

- ▶ **Gradient:** If the gradient is easy to compute, then it is possible to add a condition on the gradient such as

$$\|\nabla f(x_{k+1})\| < \text{gtol},$$

where gtol is a few tens or hundreds times ε_{eps} .

- ▶ As a safety precaution, it is customary to set an **upper limit on the number of iterations** to be performed.

Choice of the step size for line search

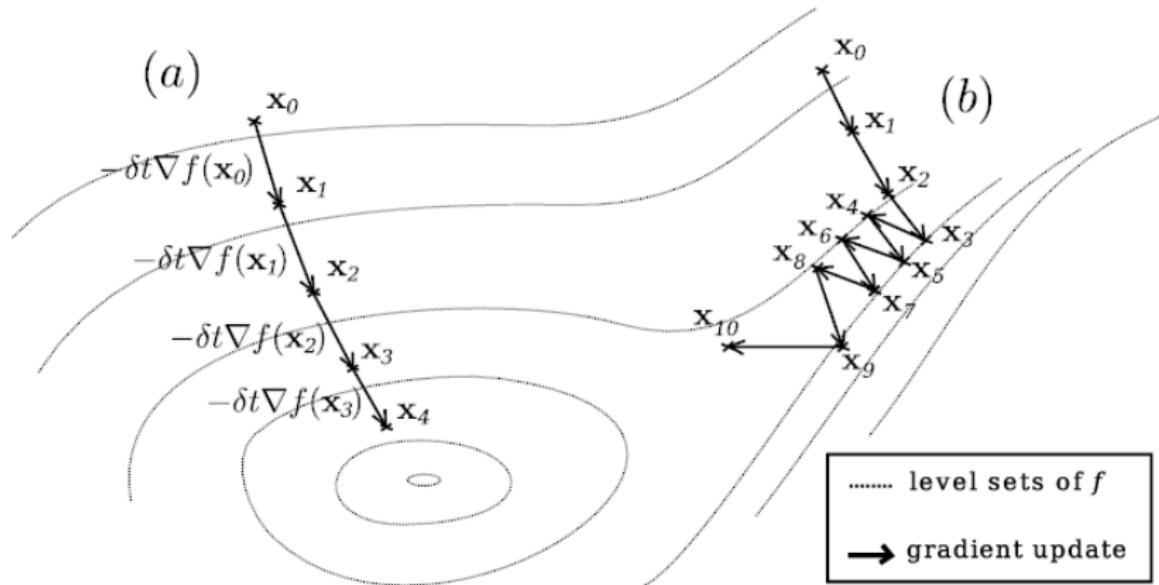
How to find solution α_k for $a_k = \arg \min_{\alpha_k} f(x_k + \alpha_k \cdot p_k)$:

- ▶ Non-trivial problem, especially if the evaluation of f is expensive.
- ▶ The step-size a_k should not be too small and, at the same time, not too large.
- ▶ There are guidelines: For example:
The Powell–Wolfe rule is based on $\nabla f(x)$ and asks us to select a_k such that the following 2 conditions are satisfied:
 - ▶ Armijo rule (upper bound on step-size)
 - ▶ curvature condition (lower bound on step-size)

More details:

https://en.wikipedia.org/wiki/Wolfe_conditions

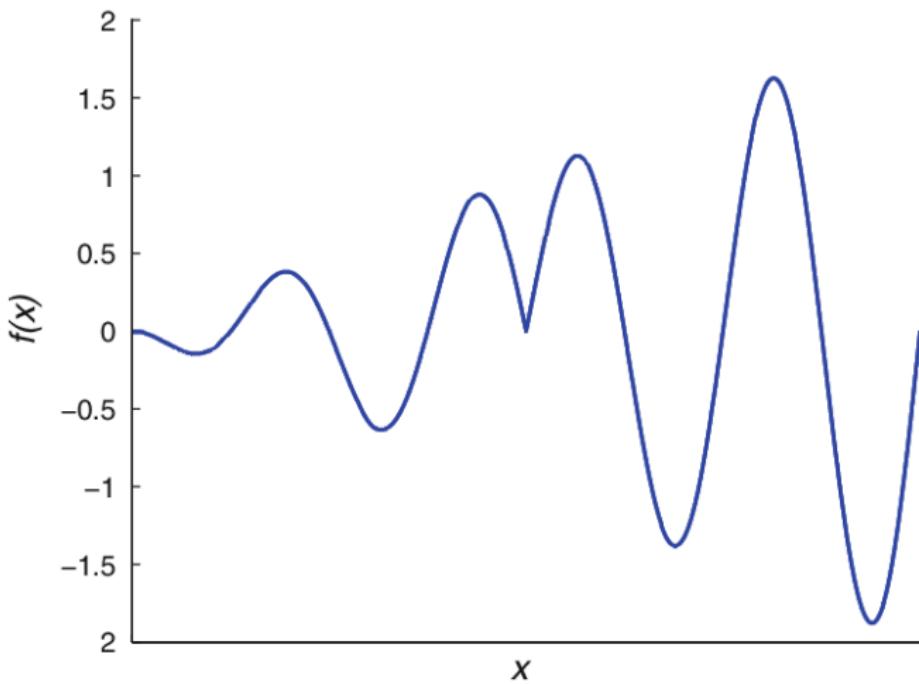
Examples: steepest descent method



Gradient descent

Multiple local minima

Multiple local minima are a potential problem for numerical optimization algorithms:

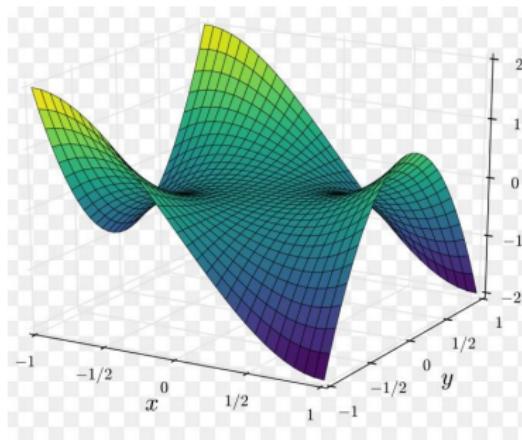
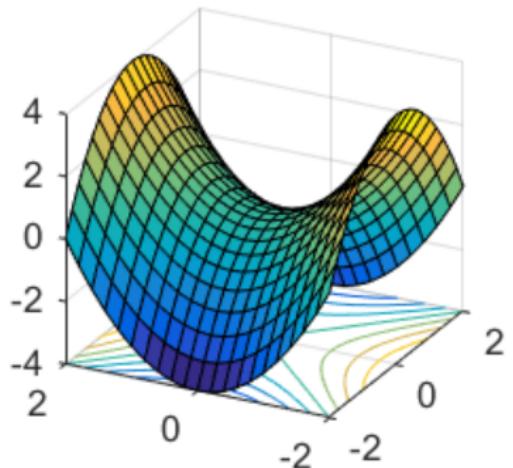


Multiple local minima

- ▶ An objective function may have multiple local minima in addition to a global minimum.
- ▶ It is also possible to have multiple global minima, all of which achieve the same minimum value at different locations.
- ▶ There is no good general method for finding the global minimum in multiple dimensions.
 - ▶ One strategy is to use multiple starting points in the hope that one of them leads to the global minimum.
 - ▶ Another one is statistical optimization techniques such as simulated annealing.

Saddle points

Potential problems for numerical optimization: saddle points



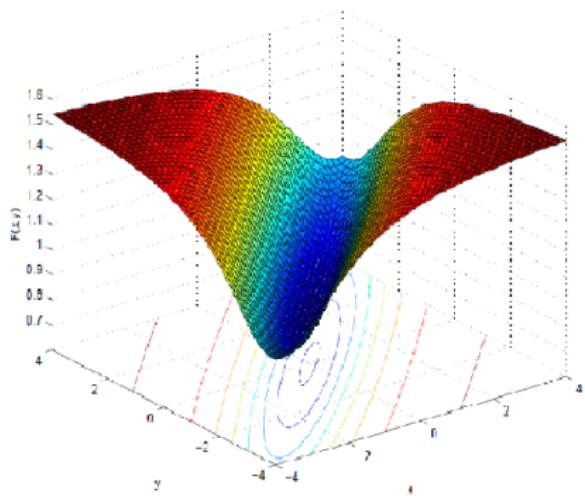
What is the problem here?

Saddle points

- ▶ At a saddle point, there is a minimum in one direction and a maximum in the other direction. A saddle point tends to cause problems for optimization algorithms, or at least their users.
- ▶ An iterative algorithm can, when it reaches a saddle point, go downhill into any of two (or more) different minima; the one actually chosen may not be the one that the user wants.

Long narrow valleys

Another potential problems for numerical optimization: long narrow valleys



What is the problem here?

Long, narrow valleys

- ▶ Long, narrow valleys (known as ridges to those who maximize) tend to cause problems for optimization algorithms.
- ▶ On the floor of the valley, the gradient is near zero and points along the axis of the valley; but it rotates by ninety degrees and increases rapidly as you move up the sides of the valley.
- ▶ Many optimization algorithms tend to zig-zag a long the floor to reach the minimum and use many more cycles than would seem necessary.

Comments on Collinearity

- ▶ The presence of multiple minima is not usually the problem in statistics, but ridges are.
- ▶ ∇f along the axis of a ridge can be nearly zero.

Where do ridges come from? Answer: **Collinearity**.

- ▶ In linear regression $X\beta = y$, collinearity can result in ridges in the likelihood. Near-collinearity causes the $X^T X$ matrix to be almost singular (difficult to invert) and thus ill-conditioned.
- ▶ Near-collinearity of covariates in regression results in large standard errors for parameter estimates and large variances for model predictions, thus making them not reliable.
- ▶ Two approaches to fixing collinearity are **principal components** and **ridge regression**.

Some more comments on collinearity: principal components

- ▶ Principal components analysis is applied to reduce the number of predictors prior to fitting the linear regression. For this one includes those variables which are collinear.
- ▶ The principal component variables are then included in the regression model.
- ▶ We cannot interpret those individual variables which were included via the principal components.

Some more comments on collinearity

Ridge regression is similar to least squares regression, but introduces an extra parameter called a "ridge parameter", which essentially measures the deviation of ridge regression from least squares regression. The ridge parameter is set by the user, and it trades off the bias of parameters with amount of collinearity present.

To be precise, for the problem $X\beta = y$ we have:

- ▶ **Least squares:**

$$\hat{\beta} = (A^T A)^{-1} A^T y$$

- ▶ **Ridge regression:**

$$\hat{\beta} = (A^T A + k \mathcal{I}_{N \times N})^{-1} A^T y,$$

where k is the ridge parameter and $\mathcal{I}_{N \times N}$ is the identity matrix. The ridge estimate is a penalized least squares estimate.

Ridge regression

We saw on the previous slide:

- ▶ Least squares:

$$\hat{\beta} = (A^\top A)^{-1} A^\top y$$

- ▶ Ridge regression for parameter k :

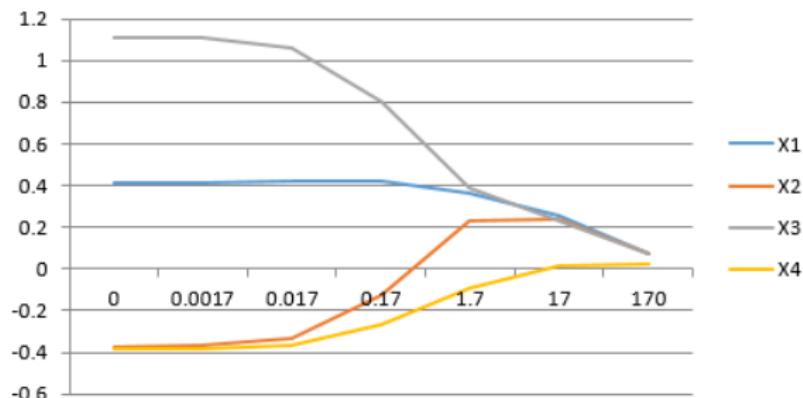
$$\hat{\beta} = (A^\top A + k\mathcal{I}_{N \times N})^{-1} A^\top y.$$

Some comments:

- ▶ Small positive values of k improve the conditioning of the problem, and reduce the variance of the estimates.
- ▶ As $k \rightarrow 0$, we obtain the least squares estimates. As $k \rightarrow \infty$, the estimates go to zero and we obtain an intercept only model.

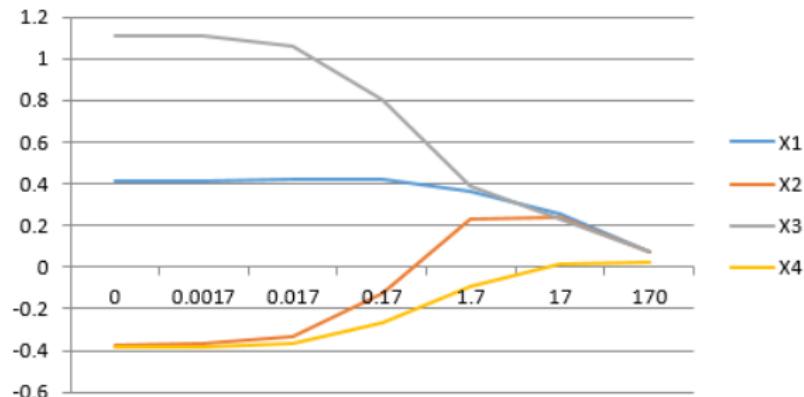
Pros and cons of the ridge estimator

- ▶ The ridge estimates are biased, but the reduced variance of ridge estimates often results in a reduced *mean squared error* (MSE) compared to LS estimates. Thus, ridge regression reduces the variance of the parameters at the cost of biased parameters. This is called **bias-variance trade-off**.
- ▶ Consider the range of ridge parameters in order to assess the impact on parameter estimates. We plot the ridge parameter k on the x-axis, and the standardized coefficients on the y-axis. This is called *Ridge Trace*.



Ridge trace

Ridge trace:



- ▶ We choose k for which the estimated coefficients stabilize, are not rapidly changing, and the signs make sense.
- ▶ However, there is no real justification for this or any other selection approach, e.g. cross-validation.

Optimization by Root Finding

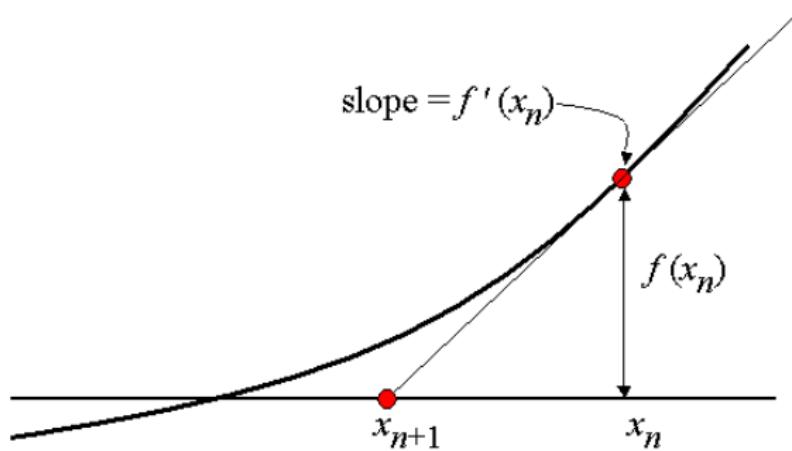
Idea: We know how to find the zero (root) of a function. If the gradient ∇f of f exists, and can be easily computed, we can solve $\nabla f(x) = 0$ with a root finding algorithm and thus find the optimum of f .

- ▶ Optimization problem is converted into a root finding problem.
- ▶ We solve $0 = f'(x) = \nabla f(x)$.
- ▶ Resulting set of equations can be solved analytically or numerically.
- ▶ Gradient of the objective function must be easily computable. However, it is not always possible, or economical, to obtain the derivatives.
- ▶ Approach can be sensitive to numerical precision issues.

Newton-Raphson method

Starting from some initial value x_0 , Newton's method iteratively approximates the zero of a function f by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$



Newton-Raphson method to find extremal points

- ▶ We can use the Newton-Raphson algorithm to find the zero of ∇f , and thus an extremal point of f .
- ▶ For this we need that both the first and second derivatives of f can be computed.
- ▶ Drawbacks:
 - ▶ The multivariate Newton-Raphson method is quite sensitive to the starting value.
 - ▶ The evaluation of the first and second derivatives of f is more complicated in the multivariate scenario.
 - ▶ A general drawback about finding minima via $\nabla f = 0$ is that minima, maxima and saddle points will be found indiscriminately. One must test the solution afterwards to see what type it is.

Newton-Raphson method

Newton's method is a special case of general gradient descent in some direction p_k .

Consider the second order Taylor approximation of f at x_k :

$$f(x_k + p) \approx f(x_k) + p^\top \nabla(x_k) + \frac{1}{2} p^\top \nabla^2 f(x_k) p$$

For a given estimate x_k , the minimization of $f(x_k + p)$ is at

$$\nabla f(x_k + p) = 0,$$

where p is the variable and $f(x_k)$, $\nabla f(x_k)$, $\nabla^2 f(x_k)$ are constants.

Newton-Raphson method

Newton's method is a special case of general gradient descent in some direction p_k .

After Taylor expanding ∇f , solving for p yields

$$\begin{aligned}\nabla f(x_k + p) &\approx \nabla f(x_k) + \nabla^2 f(x_k)p \stackrel{!}{=} 0 \\ \Rightarrow p_k &= -(\nabla^2 f(x_k))^{-1} \nabla f(x_k),\end{aligned}$$

called **Newton's direction**.

Overall, we obtain the multidimensional Newton-Raphson method

$$x_{k+1} = x_k - (\nabla^2 f(x_k))^{-1} \nabla f(x_k).$$

Newton-Raphson method

Claim: Choosing the Newton direction always decreases the objective function.

Proof: We consider the Taylor expansion

$$f(x_k + p_k) \approx f(x_k) + p_k^\top \nabla f(x) + \frac{1}{2} p_k^\top \nabla^2 f(x_k) p_k$$

and substitute $p_k = -(\nabla^2 f(x_k))^{-1} \nabla f(x_k)$, leading to

$$\begin{aligned} f(x_k + p_k) &\approx f(x_k) + p_k^\top \nabla f(x) + \frac{1}{2} p_k^\top \nabla^2 f(x_k) p_k \\ &= f(x_k) - \nabla f(x_k)^\top (\nabla^2 f(x_k))^{-1} \nabla f(x_k) \\ &\quad + \frac{1}{2} \nabla f(x_k)^\top (\nabla^2 f(x_k))^{-1} \nabla f(x_k) \\ &= f(x_k) - \underbrace{\frac{1}{2} \nabla f(x_k)^\top (\nabla^2 f(x_k))^{-1} \nabla f(x_k)}_{\geq 0} \leq f(x_k) \end{aligned}$$

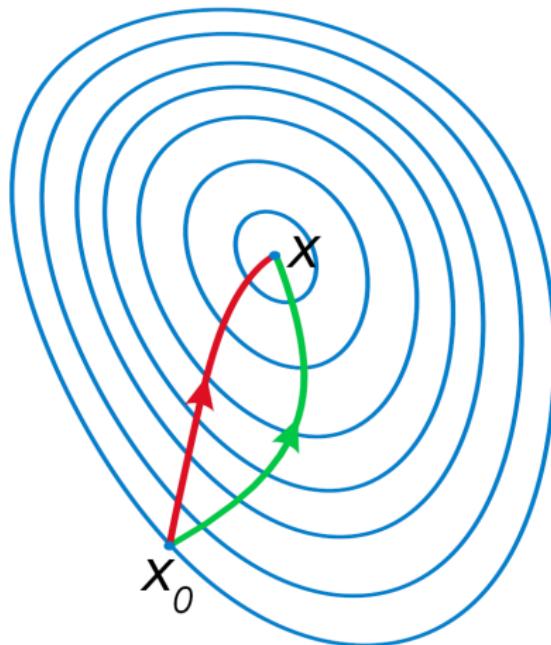
What does this imply?

Newton-Raphson method: Illustration

Depending on whether we apply Newton-Raphson to f or ∇f , we either find the root of f or a local extremum (characterized by $\nabla f(x) = 0$).

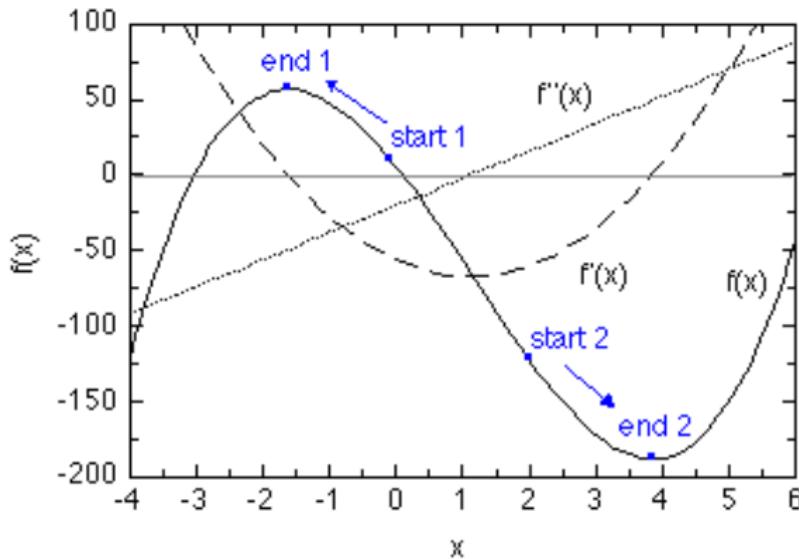
- ▶ In one dimension, the Newton-Raphson root finding method computes the tangent to the function at the current estimate, finds the root of the tangent and uses it as the next estimate.
- ▶ In one dimension, the Newton-Raphson minimization method amounts to using the first and second derivatives at the current iterate to define a parabola, finding the point at which that parabola attains its minimum, and using that point as the next iterate.
- ▶ This method can fail if the parabola points up rather than down, or if the minimum is too far away.

Newton-Raphson: Illustration



A comparison of gradient descent (green) and Newton's method (red) for minimizing a function (with small step sizes). Newton's method uses curvature information (i.e., second derivatives) to take a more direct route. (Wikipedia)

Potential issues with Newton-Raphson

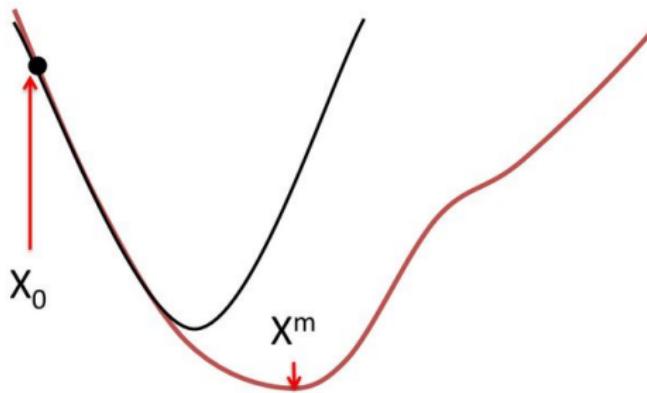


Root finding of $f(x) = 3x^3 - 10x^2 - 56x + 5$.

What are potential ways to address this issue?

Newton-Raphson for minimization in a picture

- ▶ In one dimension, Newton-Raphson for minimization (i.e., applied to ∇f) amounts to define a parabola and using its minimum as the next iterate.
- ▶ This method can go wild if the parabola points up as opposed to down, or if the minimum is too far away from the true minimum.



Variation of Newton's method

Issues:

- ▶ When computing zeros via $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, it can happen that the derivative is not defined or too close to zero, making Newton's method unstable.
- ▶ When computing extrema via $x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$ it often happens that the Hessian becomes indefinite or ill-conditioned. Moreover, even if possible, computing $f'(x_n)/f''(x_n)$ requires effort $O(n^3)$ which is an effort we might want to avoid.

Let us now quickly state a few important variants of the Newton method...

Variation of Newton's method

Variations:

- ▶ In case the Hessian becomes unavailable, we can use *modified Newton methods* such as **quasi-Newton methods** to approximate the Hessian. This can happen if the Hessian becomes indefinite, ill-conditioned, or even singular, or simply too expensive to compute. Quasi-Newton methods determine the search direction by other matrices or by updating it from the computed gradients.
- ▶ If the inverse of Hessian is too expensive to compute we can use *inexact Newton methods*.

Quasi-Newton methods

Idea: Quasi-Newton methods do not need the Hessian for minimization/ maximization. Instead, they take rescaled steps in the direction of the gradient.

Important method: **BFGS** (Broyden-Fletcher-Goldfarb-Shanno)

- ▶ BFGS is one of the most popular quasi-Newton methods.
- ▶ Developed for problems where $\nabla f(x)$ is the gradient of a scalar objective function $f(x)$ (e.g., maximum likelihood). In this case the Hessian $\nabla^2 f$ is symmetric and positive definite.
- ▶ BFGS ensures that the updated Hessian remains symmetric and positive definite at every iteration, leading to a more stable algorithm.
- ▶ Convergence order is somewhere between 1.5 and 2.

Alternatives to classical optimization algorithms

- ▶ Momentum-based approaches are motivated by physics with the aim to minimize oscillations, avoid local minima and accelerate convergence. Methods add some fraction of the previous descent direction to the current one, i.e.

$$\tilde{p}_{k+1} = p_k + \eta p_{k-1} \quad \text{with } \eta \in (0, 1).$$

- ▶ Adaptive approaches: the learning rate or the parameters change in the current update step depending on the parameter values in the previous step, e.g. each element of $x \in \mathbb{R}^N$ is updated at a different rate.

Alternatives to classical optimization algorithms: Discrete Newton methods

- ▶ Another approach to avoiding direct calculations involving the derivatives is a *discrete Newton method*.
- ▶ These methods replace the elements of the Jacobian matrix with difference approximations, such as
$$J_{ij}(x) = \frac{f_i(x + h_{ij}e_j) - f_i(x)}{h_{ij}}$$
with "crude" updates applied to the elements of J after each iteration. Typically, $h_{ij} = h_j$ is used.
- ▶ Choosing the right value for the step size h_{ij} can be tricky. If chosen too small, the rounding error makes the derivative too inaccurate. If chosen too large, the derivative might also be inaccurate because the higher order terms in the series expansion are neglected.
- ▶ The convergence rate of discrete Newton methods is about order 1.5. If all h_i 's are set to a constant h , the convergence rate is linear.

Gauss-Seidel iteration

- ▶ One such approach is the *Gauss-Seidel iteration*. The function $f(x)$, with $x = (x_1, \dots, x_n)$, is considered to be a function of x_i at a time with all other entries held constant. The algorithm cycles through all x_i in each iteration and then repeats.
- ▶ As one x_i is considered at a time, optimizing $f(\dots, x_i, \dots)$ becomes a one dimensional problem which can be solved by simple line search methods.
- ▶ When Newton-Raphson is used for the single parameter optimization, this is called *Gauss-Seidel-Newton iteration*.
- ▶ The Gauss-Seidel-Newton iteration is simple to implement but can be slow.

Conjugate gradient methods for linear equation systems

We will now give a detailed overview of an important method to solve linear equation systems called the *conjugate gradient method*.

- ▶ It is only applicable to particular systems of linear equations, namely those whose matrix is symmetric and positive-definite.
- ▶ It has three main advantages:
 - ▶ The gradient is always nonzero and linearly independent of all previous direction vectors.
 - ▶ It is based on an especially simple formula to determine the new direction vector. This simplicity makes the method only slightly more complicated than steepest descent.
 - ▶ Since the new directions are based on the gradients, the process makes good uniform progress towards the solution at every step.

Conjugate gradient methods for linear equation systems

Consider the quadratic optimization problem

$$\min_x f(x) = \min_x \left(\frac{1}{2} x^\top A x - b^\top x \right)$$

- ▶ $A \in \mathbb{R}^{N \times N}$ large, sparse, symmetric matrix and $x, b \in \mathbb{R}^N$.
- ▶ Optimal solution is attained if $\nabla f(x) = 0$, thus

$$\nabla \left(\frac{1}{2} x^\top A x - b^\top x \right) = Ax - b \stackrel{!}{=} 0$$
$$r(x) := Ax - b \quad \text{the "residual".}$$

- ▶ Steepest descent along the gradient

$$-\nabla f(x) = -r(x) = b - Ax.$$

The solution of $Ax = b$ can be iteratively obtained without inverting A . Why is this good?

Steepest descent + line search

We aim to iteratively compute

$$\min_x \left(\frac{1}{2} x^\top A x - b^\top x \right).$$

Algorithm:

1. Start with an initial guess x_0 .
2. In iteration k , set search direction to $p_k = -\nabla f(x) = b - Ax_k$.
3. The optimal step length is given by

$$a_k = \arg \min_{\alpha_k} f(x_k + \alpha_k \cdot p_k)$$

having the optimal solution

$$a_k = \frac{r_k^\top r_k}{p_k^\top A p_k}, \quad r_k = Ax_k - b.$$

4. Update $x_{k+1} = x_k + a_k p_k$ and go to step 2.

What are potential issues with this approach?

Potential issues

Potential issues of solving $Ax = b$ by transforming it into an optimization problem?

- ▶ The steepest descent method has a slow convergence speed.
- ▶ How about using Newton's method instead (having quadratic convergence)?

Matrix inner product, norm, and conjugate

Let A be a symmetric positive definite matrix.

- ▶ A induces an A *inner product* defined as

$$(x, y)_A := x^\top A y.$$

- ▶ The inner product induces an A *norm* defined as

$$\|x\|_A = \sqrt{x^\top A x}.$$

- ▶ We say that two vectors x and y are A *conjugate* for a symmetric positive definite matrix A if

$$x^\top A y = 0,$$

meaning that x and y are orthogonal under the inner product induced by A .

Conjugate direction and conjugate gradient

- The **conjugate directions** are a set of search directions $\{p_0, p_1, p_2, \dots\}$ such that

$$p_i^\top A p_j = 0 \quad i \neq j.$$

- We now consider again the residuals $r_k = Ax_k - b$. Using conjugate directions we compute the new search direction p_k as a linear combination involving the previous direction p_{k-1} :

$$p_{k+1} = -r_k + \beta_k p_k.$$

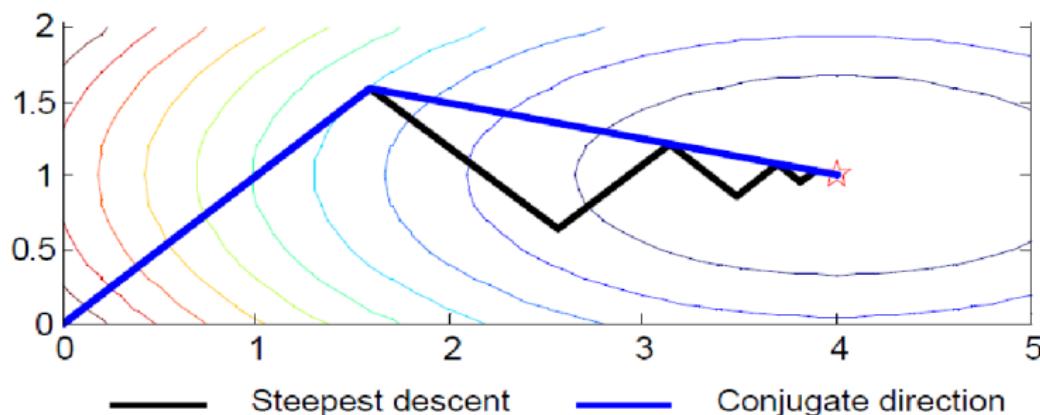
- We define p_{k+1} so that it is A -conjugate to p_k , meaning that $p_{k+1}^\top A p_k = 0$:

$$0 \stackrel{!}{=} p_{k+1}^\top A p_k = -r_k^\top A p_k + \beta_k p_k^\top A p_k.$$

- Thus, $\beta_k = \frac{p_k^\top A r_k}{p_k^\top A p_k} = \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}.$

Example: Steepest descent vs. conjugate direction

Example: $Ax = b$ with $A = \begin{bmatrix} 0.25 & 0 \\ 0 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, $x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$.



The linear CG algorithm

With some linear algebra, we can simplify the Conjugate Gradient algorithm:

1. Given A and b , initial value x_0 , and error tolerance ϵ .
2. Set $r_0 := Ax_0 - b$ and direction $p_0 := -r_0$.
3. For $k = 0, 1, 2, \dots$ until $|r_k| \leq \epsilon$:

$$\alpha_k = \frac{r_k^\top r_k}{p_k^\top A p_k}$$

$$x_{k+1} = x_k + \alpha_k p_k$$

$$r_{k+1} = r_k + \alpha_k A p_k$$

$$\beta_{k+1} = \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$$

$$p_{k+1} = -r_{k+1} + \beta_{k+1} p_k$$

Properties of linear CG

Some remarks on the CG algorithm:

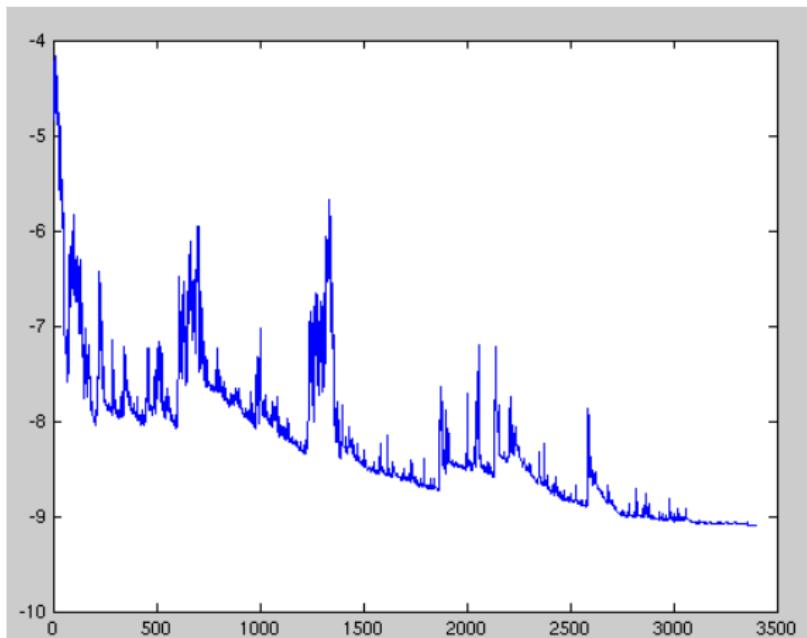
- ▶ One matrix-vector multiplication per iteration.
- ▶ Only four vectors are required: x_k , r_k , p_k , and Ap_k .
- ▶ The matrix A typically in compressed format.
- ▶ The CG algorithm guarantees convergence in r iterations, where r is the number of distinct eigenvalues of A .
- ▶ The convergence is typically ensured by termination criterion (=error criterion).
- ▶ For non-symmetric matrices, pre-conditioning approaches can be applied. Numerical stability can become an issue in this case.

Alternatives to classical optimization algorithms

- ▶ **Mini-batch algorithm:** compute the gradient for multiple subsets of the data in each step leading to an increased convergence speed. Often the "preferred" optimization algorithm in deep learning.
- ▶ **Stochastic gradient descent:** in each step, we select a random subset of the data, e.g. a random subset of the rows of the matrix X , and compute the gradient for this subset. Reduced numerical cost, but slower convergence speed. Some approaches also do not compute the full gradient but only take a step in the direction of one (or more) partial derivatives.

We will now look at both...

Convergence of mini-batch algorithm



Fluctuations in the total objective function as gradient steps with respect to mini-batches are taken. (Example from the wikipedia article "Stochastic gradient descent")

Alternatives to classical optimization algorithms

There exist numerous extensions of moment-based optimization algorithms and adaptive optimization algorithms:

- ▶ Another important class is gradient-free optimizers which only require function evaluations.
- ▶ Those are AI-type of optimization algorithms which are typically based on some social-behavioural patterns/intelligence, or physical processes:
 - ▶ Genetic algorithms
 - ▶ Simulated Annealing
 - ▶ Particle Swarm optimization
 - ▶ Artificial Bee Colony algorithm
 - ▶ Fish School search
 - ▶ Ant Colony optimization (mostly combinatorial optimization)

Simulated annealing

- ▶ One alternative to classical optimization algorithms is *Simulated Annealing* (Kirkpatrick et al., 1983).
- ▶ Simulated annealing (SA) only requires function evaluations of f , no gradients.
- ▶ SA is a statistical optimization technique.

Simulated annealing: basic idea

Idea of SA

Starting from some initial value $x := x_0$, we propose a new point x' to explore which is drawn from some distribution. If $f(x') < f(x)$ we accept the move and set $x := x'$. Otherwise we do not discard x' right away, but might still accept it with a probability proportional to

$$\exp \left[(f(x) - f(x'))/t(n) \right],$$

where $t(n) \rightarrow 0$ as the iteration number $n \rightarrow \infty$.

Simulated annealing parameters I

What is the reasoning behind the acceptance move?

- ▶ The acceptance move allows us to jump out of local minima!
- ▶ Moves x' that make the objective function worse again allow us to leave local minima and (potentially) find a global one.

Simulated annealing parameters II

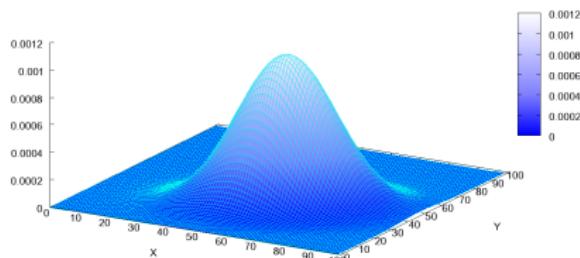
What is $\exp[(f(x) - f(x'))/t(n)]$ and how should $t(n)$ be chosen?

- ▶ If $f(x') < f(x)$ we accept the move x' (in this case $(f(x) - f(x'))/t(n)$ is positive and thus the $\exp(\cdot)$ term is larger than 1).
- ▶ If $f(x') \geq f(x)$ we only accept with the above probability. This probability is antiproportional to (a) the (undesired) increase $f(x) - f(x') < 0$, and in fact exponentially downweighted, and (b) the *temperature* $t(n)$.
- ▶ Choosing $t(n) = \log(n)^{-1}$ ensures that, under conditions, the global optimum is found as $n \rightarrow \infty$ (Henderson et al., 2003)!
- ▶ As $t(n)$ makes the acceptance probability go to zero, SA turns into a steepest descent algorithm over time.
- ▶ At the beginning, SA is in an *exploration phase* in which it is common to leave (local) minima again, whereas over time this is made increasingly hard and SA will settle on a minimum.

Simulated annealing parameters III

How should x' be proposed?

- ▶ In each iteration we propose a new move x' . There are several options to do this.
- ▶ The *independent proposal* samples x' uniformly from the search space, causing the algorithm to *jump around*.
- ▶ More common is to choose x' in a neighborhood of the current best solution x . A popular choice is to sample x' from a multivariate normal distribution $N_d(x, \Sigma)$ centered at x with some covariance matrix Σ , where $x \in \mathbb{R}^d$ is the dimension.



Source: Wikipedia article "Multivariate normal distribution"

Simulated annealing

The full algorithm to minimize any function f :

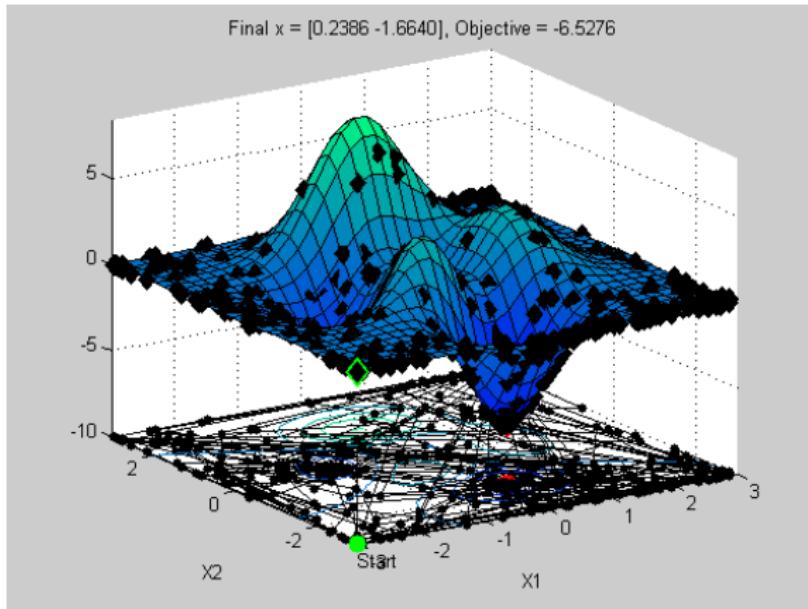
1. Start with initial value x_0 . Select proposal distribution $p(x|x_0)$ and temperature function $t(n)$.
2. For K steps or *until convergence* repeat:
 - 2.1 Draw proposal $x' \sim p(x|x_0)$.
 - 2.2 Compute $\alpha := \exp[(f(x) - f(x'))/t(n)]$. Accept x' with probability $\min(1, \alpha)$, i.e. set $x := x'$.

Simulated annealing

Good choices are $p(x|x_0) = N_d(x_0, \Sigma)$ and $t(n) = \log(n)^{-1}$, where $x_0 \in \mathbb{R}^d$. Convergence can be defined in many ways, e.g. new accepted moves are only ϵ away from old ones (that is, $\|x - x'\| < \epsilon$), or no move has been accepted for the past N_0 steps.

Other statistical optimization techniques (e.g., particle swarm optimization) work in a similar way.

Simulated annealing: example of surface exploration



Picture taken from

<http://www.math.uwaterloo.ca/~hwolkowi/henry/reports/talks.d/t09talks.d/09waterloomatlab.d/optimTipsWebinar/html/optimTipsTricksWalkthrough.html>

When to use statistical optimization methods such as SA?

Advantages:

- ▶ Simple algorithms.
- ▶ Require function evaluations only and no gradients.
- ▶ Can be used in high dimensions.
- ▶ Designed to find the global optimum.
- ▶ Can be used to solve combinatorial optimization problems as well, e.g. the Traveling Salesman problem.

Disadvantages:

- ▶ Typically very slow convergence speed.