Big Data Analytics

Christian Borgelt

Dept. Artificial Intelligence and Human Interfaces
Paris-Lodron-University of Salzburg
Jakob-Haringer-Straße 2, 5020 Salzburg, Austria

christian.borgelt@plus.ac.at
christian@borgelt.net
http://www.borgelt.net/

https://meet.google.com/vnq-ddnd-ssc

(several slides courtesy of Barbara Pampel, University of Konstanz, Germany)

Organization

Lesson

- Lessons take place once a week, Thursday, 14:30–17:00 hours, in the GI-Lab.
  Usually: 1:00 to 1:15 hours instruction/exercises, 15 minutes break,
  1:00 to 1:15 hours instruction/exercises (total: 2:15 hours + break).

Exercises

- There will be a total of 12 exercise sheets ($\Rightarrow$ Blackboard-System).
- Exercises have to be solved in groups of **two** people.

  Team up and send exactly **one** email per group, with subject "`[BDA 2023] group`", to

  `christian.borgelt@plus.ac.at`,

  with your team partner in `Cc` and the names of the team members in the body.

  Example:

```
From: alice.ace@stud.sbg.ac.at
To: christian.borgelt@plus.ac.at
Cc: bob.best@stud.sbg.ac.at
Subject: [BDA 2023] group

Alice Ace
Bob Best
```

Organization

Exercises

- Solutions to exercises have to be submitted by email.
  - Python scripts: Either as pure Python (extension "`.py`")
    or as Jupyter Notebook (extension "`.ipynb`").

    The mail server of the university sometimes blocks access
    to email attachments with the extension "`.py`".
    To circumvent this problem, please change the extension to "`._py`"
    or pack the file into a zip archive (extension "`.zip`").
  - Theoretical exercises: Either as a PDF file (portable document format)
    or as MarkDown in a Jupyter Notebook.

- Send exactly **one** email per group, with subject "`[BDA 2023] solution` x", to

  `christian.borgelt@plus.ac.at`,

  with your team partner in `Cc` and the names of the team members in the body.

Example:

```
From: alice.ace@stud.sbg.ac.at
To: christian.borgelt@plus.ac.at
Cc: bob.best@stud.sbg.ac.at
Subject: [BDA 2023] solution 1

Alice Ace
Bob Best
```

Organization

Exercises

- (Attempts at) Solutions are submitted and presented in the lesson.
  In order to pass the course, you have to
  - Score **sufficiently many points** for your submitted solutions.
    (see next slide for details and (tentative) grading)
  - Declare yourself willing to present your (attempt at a) solution
    ("vote" for presenting a solution) for **at least half** of the exercises.
    ($\Rightarrow$ to be expressed via a poll before a lesson, URL will be sent by email).
  - Actually present your (attempt at a) solution **at least twice**.

    Votes and presentations are counted **per group**, *not* per person.

- It may happen that adaptations of the rules are necessary.
  - There may be not enough exercises so that each group can present twice.
  - Grading (next slide) may be adapted to avoid too many failing grades.

Organization

Grading

- There will be a total of 12 exercise sheets ($\Rightarrow$ Blackboard-System).
- For each exercise sheet a maximum of 8 points can be scored.
- The total number of points that can be scored is $12 \times 8 = 96$ points.
- There will be (at least) 2 supplementary exercises (4 points each).
  These are meant for catching up on missed exercises
  and to compensate for some badly solved exercises.
- (Tentative) Grading: Suppose your group scored a total of p points.

$0 \leq p < 48$:		failed	(5)
$48 \leq p < 60$:		sufficient	(4)
$60 \leq p < 72$:		satisfactory	(3)
$72 \leq p < 84$:		good	(2)
$84 \leq p$	:	very good	(1)

Contents

General

- Introduction to "Big Data" and corresponding techniques; problems of handling large data sets and approaches to solutions.

Algorithmics

- Streaming Algorithms
- External Memory Algorithms
- Parallel Computation

Tools and Techniques

- Python / Jupyter Notebook
- Online computation, recursion and indirection/references

What This Course Does Not Cover

Data Mining

- We will look at some simple Data Mining algorithms, . . .

  . . . , but this is **not** a Data Mining course.

- If you want to know more about Data Mining:

  Lecture (VU) "**(Basic) Data Mining**" (given in winter semester)

  - Data & Knowledge, Knowledge Discovery Process
  - Principles of Modeling, Model Evaluation/Validation
  - Some Basic Methods: k-Nearest Neighors, Bayes Classifiers, Decision and Regression Trees, Clustering

Recommender Systems

- Recommender systems will be mentioned, . . .

  . . . , but are **not** a primary topic of this lecture.

Example 1: Amazon

Simple Example: Amazon

- Basically a selling platform (Amazon Marketplace)
- Amazon Marketplace provides:
  - Connection of suppliers to customers
  - A common market place (one interface for all shops)
  - Additional services (storage, shipment, payment, search)
  - **Recommendations**

What is the Difference to Competitors?

- Amazon knows customers, products, sales, and views (of products).
- Same is true for its competitors.
- But: In comparison, Amazon has many more customers.

Amazon's Advantages

- More customers, more transactions, more views (of products)

  $\Rightarrow$ Larger data collection

  $\Rightarrow$ Better recommendations

- Estimate: 1/3 of Amazon's sales are generated by recommendations.
- More data = better recommendations/sales predictions?
  - Simple answer: Essentially yes
  - Actual answer: It's a bit more complicated

- What are we trying to find out?
  - Learning/data mining and artificial intelligence are not that new.
  - Somehow huge amounts of data *can* make a difference.
  - **Question:** How and why?

Example 2: Target and the Pregnant Teen

Target

- Target is a large discounter chain
  (similar to Walmart/Aldi/Lidl/Hofer).
- Target uses Data Analytics for Marketing.

The Pregnant Teen Story

Charles Duhigg: "How Companies Learn Your Secrets"
The New York Times Magazine, Feb. 16, 2012
`https://www.nytimes.com/2012/02/19/magazine/shopping-habits.html`

- Shopping habits are
  very ingrained, difficult to change.
- A certain times in the life of a person, change is easier to bring about.
- Target advertises baby products to a teenager.
- Father complains, finds out that his daughter is actually pregnant.
- Proves that algorithm predicts pregnancy better than family members.

How did Target Predict the Pregnancy?

In a Nutshell

- Collect data about customers (who buys what and when).
- Predict what they are interested in.
- Adjust advertisement to a specific person (“personalization”).

Data Collection

- Each customer gets a unique ID (credit card number, email, . . .).
- Collected data is connected to the customer used for interest prediction.

Examples of Data to Collect

- Items purchased together
- Time/place of purchase
- Device? Weather? — Whatever can be collected.

How did Target Predict the Pregnancy?

Search for Patterns

- Simple: people buy what they always bought
- Recommendation: "Customers who bought this, usually also buy …"

Concrete Targeting (example: young parents)

- Influence/change shopping habits
  (very ingrained, changeable only at certain times)
- A new child is a perfect opportunity:
  - Parents have to buy a lot of stuff
    (without having too much money).
  - At this stage they are more likely to get bound to new brands
    (formation of new shopping habits).

  $\Rightarrow$ Prediction of pregnancy is crucial for advertisement.

How did Target Predict the Pregnancy?

Data Gathering

- Customers are described by their purchases
  (items, time, payment method, ...).
- Products are described by purchases
  (e.g. family products are mostly bought on weekends).

Given Enough Records, Patterns Emerge

- Typical purchase histories (as in the pregnancy example).
- Typical customers (as in "I always buy beer and chips").
- New products that become popular vs. products that are ignored.
- These patterns can be very complicated.
- More data leads to more opportunities (e.g. more complex patterns).

How did Target Predict the Pregnancy?

Patterns in the Example

- Quoting a Target analyst:
  - They identified about 25 products.
  - Together these allow to compute a "pregnancy prediction" score.
- Examples: Pregnant women buy food supplements like calcium, magnesium and zinc "sometime in the first 20 weeks". They also tend to buy more unscented lotion.

Business Impact

- Start of program: 2002 (Data Analyst Andrew Pole gets hired)
- Revenue growth: \$44 Billion (2002) $\rightarrow$ \$67 Billion (2010)
- It is assumed that data mining was crucial for this growth.

Source: `https://www.nytimes.com/2012/02/19/magazine/shopping-habits.html`

Example 3: Machine Translation

The Task

- Automatic translation of text
- Given: Text T in language A
- Desired result: Text T' in language B
- Example: **Google's Translator** `http://translate.google.at/`

Word Mappings

- Maintain a dictionary $W : A \rightarrow B$.
- Replace each $w \in T$ by $W(w)$.

Problems

- Words don't match exactly between languages (ambiguities) neither in meaning, nor in number.
- **Grammar**

Machine Translation: A Naïve Approach

Grammar is hard

- Language is noisy, fluid, grammar is often not exact.
- Even exact grammar is hard (semantics, context)
  (Chomsky hierarchy, theory of formal languages)

New Approach: Translation by Example

- Model with very few assumptions and simple rules.
- Rules expressed by probabilities.
- "Examples" are taken from a corpus of manually translated documents.

Basic Idea: (attention: the following is very simplified)

- Training: Learn probability P that some text T' is translation of text T.
- Translation: Find T' with maximal P.

Based on: `http://michaelnielsen.org/blog/introduction-to-statistical-machine-translation/`

Machine Translation: A Naïve Approach

Example: Translate French text F to English text E.

- $P(E \mid F)$ — Probability that E is a correct translation of F.
- Let $F = f_1 f_2 \ldots$ (f_i French sentence).
  Let $E = e_1 e_2 \ldots$ (e_i English sentence).

First Splitting

- Assumption: French sentence f_i corresponds to English sentence e_i.
- E is correct, if each e_i translates f_i **independent** of other sentences.

  $\Rightarrow P(E \mid F) = \prod_i P(e_i \mid f_i)$
- Try to maximize $P(e_i \mid f_i)$ for all i independently.

  $\Rightarrow$ For each sentence, find most probable translation sentence.
  (Contatenated sentences yield full translated text.)

Machine Translation: A Naïve Approach

Consider a Concrete Pair of Sentences:

- Je ne vous connais pas. $\Leftrightarrow$ I don't know you.

Je	$\rightarrow$	I	connais	$\rightarrow$	know
vous	$\rightarrow$	you	ne . . . pas	$\rightarrow$	don't

- **Observations**:
  - Words are translated (Je $\rightarrow$ I).
  - Some words change place (vous $\rightarrow$ you).
  - Some words change "number" (ne . . . pas $\rightarrow$ don't)
- **Formalize Observations into Concrete Probabilities:**

translation (Je $\rightarrow$ I)	$P(e \mid f)$	f is translation of e
distortion (you $\rightarrow$ nous)	$P(s \mid t, l)$	word at position t is replaced by word at position s in sentence of length l
fertility (ne pas $\rightarrow$ don't)	$P(n \mid f)$	f is replaced by n English words

Machine Translation: A Naïve Approach

How Does this Help for $P(E \mid F)$?

- Recall the assumption $P(E \mid F) = \prod_i P(e_i \mid f_i)$
- For $P(E \mid F)$ to be high, *every* $P(e_i \mid f_i)$ must be high.
- Same principle can be applied on the sentence level.

Breaking up Sentences

- $P(e_i \mid f_i)$ has many parts.
  - translation, distortion, fertility for every word
  - some more properties, full set unknown
  - combination by product (assuming independence)
- $P(e_i \mid f_i) \rightarrow 1$, if all the parts $\rightarrow 1$.

  $\Rightarrow$ use translation, distortion, fertility as indicators

Machine Translation: A Naïve Approach

Machine Translation: Summary

- Assumption: translation can be broke down to simple probabilities.
- Learning: estimate individual probabilities.
- Translation: find most probable sentences.

Why is this a Big Data Application?

- Individual probabilities are learned from real, manually translated texts.
- This is an individual problem for each pair of languages.
- Many individual probabilities have to be determined.
- Can be estimated from example texts (more texts $\rightarrow$ better estimates).
- Quality grows with additional knowledge (texts) fed into the system.

Machine Translation: A Naïve Approach

- **Data Sources**
  - Many books have been translated into various languages.
  - Approximate probabilities are derived by counting in corpus.
- **Open Questions**
  - Matching of sentences and words (we assumed independence).
  - Finding the actual translation (we only have partial probabilities)
  - Further information: `http://www.mt-archive.info/`
- **Why Does This Work?**
  - It works only partially. Test it yourself: `https://translate.google.at`
  - Still, the quality of the results is surprising.
- **Does It Scale?**
  - Why is it not always correct?
  - What would be the impact of adding more data?

The Notion "Big Data"

- No agreement on a definition.
- Usual understanding: Methods that
  - involve statistics / data mining / machine learning
  - necessarily involve massive amounts of data
  - (hopefully) improve with additional input
- In this course: **Algorithmics**
  - streaming algorithms
  - external memory algorithms
  - parallel computation
- In this Course: **Tools and Techniques**
  - Python / Jupyter Notebook
  - Online computation, recursion and indirection/references

Streaming Algorithms

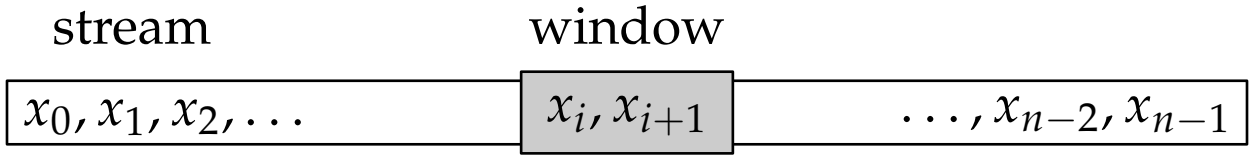
Streaming Algorithms

The Problem

- Large data set on a slow device (external memory).
- Small (in comparison) working memory.

Streaming Approach

- Read the data sequentially ($\Rightarrow$ data stream).
- Process the data in small chunks (slide a "window" over the data).



- Generally: Approximate answers to queries in sublinear space.

  The working memory needed to answer some query should not grow in proportion to the size of the data processed, but considerably less.

Notation

- X is the stream (as a set) of all data points (from universe $\mathcal{U}$).
- x_i is a data point, usually the data point currently read.
- $n = |X|$ is the number of data points.
- $m = |\mathcal{U}|$ is the number of attributes (dimensionality of the data).
- $X_{i:j} = \{x_k \in X \mid i \leq k < j\}$ is the subset of the $(j - i)$ data points i to $j - 1$.
  (like slicing in Python, $X_{0:n}$ is the whole stream, $X_{i:i}$ is empty)
- p is the number of passes over the stream.
  (One pass may not always suffice, although that is the ideal.)
- M is the size of the working memory used.
- $\mathcal{A}(X)$ is the output of the algorithm $\mathcal{A}$ applied to data stream X.

Streaming Algorithms: Objectives

- **Memory**
  - We want M, the size of the memory used, to be *sublinear*:

    $$M = o(\min\{m, n\})$$

    Here o is a Landau-Symbol (or O-Notation).
    Meaning: M should grow less quickly than the argument of $o(\cdot)$.
  - The *holy grail* (metaphor for ultimately desired performance) is:

    $$M = O(\log m + \log n)$$

    Here O is another Landau-Symbol (or O-Notation).
    Meaning: M should not grow significantly faster than the argument of $O(\cdot)$.
- **Accuracy**
  - We want to compute an (ϵ, δ)-approximation of the true value $\phi(X)$:

    $$P\Big(|\mathcal{A}(X) - \phi(X)| > \epsilon\Big) \leq \delta$$

Offline vs. Online vs. Streaming Algorithms

- **Offline** (or batch)
  - Random access to any part of the data at any time.
- **Online**
  - Only sequential access to the data is possible.
  - No knowledge about future data points is available.
  - One pass (one cannot go back to earlier data points).
  - Answers are ready all the time (e.g. sliding window).

  (related: anytime algorithms)
- **Streaming**
  - Several passes are possible, but usually avoided
    (if and as far as possible; depends on the problem).
  - Answers are usually only ready at the end of the execution.

Streaming Algorithms

Basic Problems for Streaming Algorithms

- Moments (e.g. mean, variance, . . .)
- Frequency of elements
- Number of unique elements
- Histograms and frequency moment estimation
- Sampling from streams
- Sorting
- [Clustering] (not enough time for this)

Descriptive Statistics

- Many of the above problems fall into the realm of descriptive statistics.
- Descriptive statistics are often used for exploratory data analysis.

Descriptive Statistics

Statistics: Characterizations

Statistics is the art to collect, to display, to analyze, and to interpret data in order to gain new knowledge.

"Applied Statistics" [Lothar Sachs 1999]

[...] statistics, that is, the mathematical treatment of reality, [...]

Hannah Arendt [1906–1975] in "The Human Condition", 1958

There are three kinds of lies: lies, damned lies, and statistics.

Benjamin Disraeli [1804–1881] (attributed by Mark Twain, but disputed)

Statistics, *n*. Exactly 76.4% of all statistics (including this one) are invented on the spot. However, in 83% of cases it is inappropriate to admit it.

The Devil's IT Dictionary

Basic Notions

- **Object, Case**
  Data describe objects, cases, persons etc.

- **(Random) Sample**
  The objects or cases described by a data set is called a *sample*,
  their number is the *sample size*.

- **Attribute**
  Objects and cases are described by *attributes*,
  patients in a hospital, for example, by age, sex, blood pressure etc.

- **(Attribute) Value**
  Attributes have different possible *values*.
  The age of a patient, for example, is a non-negative number.

- **Sample Value**
  The value an attribute has for an object in the sample is called *sample value*.

Scale Types / Attribute Types

Scale Type	Possible Operations	Examples
nominal (categorical, qualitative)	test for equality	sex/gender blood group
ordinal (rank scale, comparative)	test for equality greater/less than	exam grade wind strength
metric (interval scale, quantitative)	test for equality greater/less than difference maybe ratio	length weight time temperature

- Nominal scales are sometimes divided into *dichotomic* (binary, two values) and *polytomic* (more than two values).
- Metric scales may or may not allow us to form a **ratio** of values:
  weight and length do, temperature (in $^\circ C$) does not (it does in $^\circ K$).
  time as duration does, time as calender time does not.
- **Counts** may be considered as a special type (e.g. number of children).

Descriptive Statistics: Characteristic Measures

Idea: Describe a given sample by few characteristic measures
and thus **summarize the data**.

- **Localization Measures**

  Localization measures describe, often by a single number,
  where the data points of a sample are located in the domain of an attribute.

- **Dispersion Measures**

  Dispersion measures describe how much the data points
  vary around a localization parameter and thus indicate
  how well this parameter captures the localization of the data.

- **Shape Measures**

  Shape measures describe the shape of the distribution of the data points
  relative to a reference distribution.
  The most common reference distribution is the normal distribution (Gaussian).

Localization Measures: Mode and Median

- **Mode** x^*
  The mode is the attribute value that is most frequent in the sample.
  It need not be unique, because several values can have the same frequency.
  It is the most general measure, because it is applicable for all scale types.

- **Median** $\tilde{x}$
  The median minimizes the sum of absolute differences:

$$\sum_{i=0}^{n-1} |x_i - \tilde{x}| = \min. \qquad \text{and thus it is} \qquad \sum_{i=0}^{n-1} \operatorname{sgn}(x_i - \tilde{x}) = 0$$

  If $x = (x_{(0)}, \ldots, x_{(n-1)})$ is a sorted data set, the median is defined as

$$\tilde{x} = \begin{cases} x_{(\frac{n-1}{2})}, & \text{if } n \text{ is odd,} \\ \frac{1}{2}\left(x_{(\frac{n}{2}-1)} + x_{(\frac{n}{2})}\right), & \text{if } n \text{ is even.} \end{cases}$$

  The median is applicable to ordinal and metric attributes.
  (For non-metric attributes either $x_{(\frac{n}{2}-1)}$ or $x_{(\frac{n}{2})}$ needs to be chosen for even n.)

Localization Measures: Arithmetic Mean

- **Arithmetic Mean** $\bar{x} = \hat{\mu}(X)$
  The arithmetic mean minimizes the sum of squared differences:

$$\sum_{i=0}^{n-1}(x_i - \bar{x})^2 = \min. \qquad \text{and thus it is} \qquad \sum_{i=0}^{n-1}(x_i - \bar{x}) = \sum_{i=0}^{n-1} x_i - n\bar{x} = 0$$

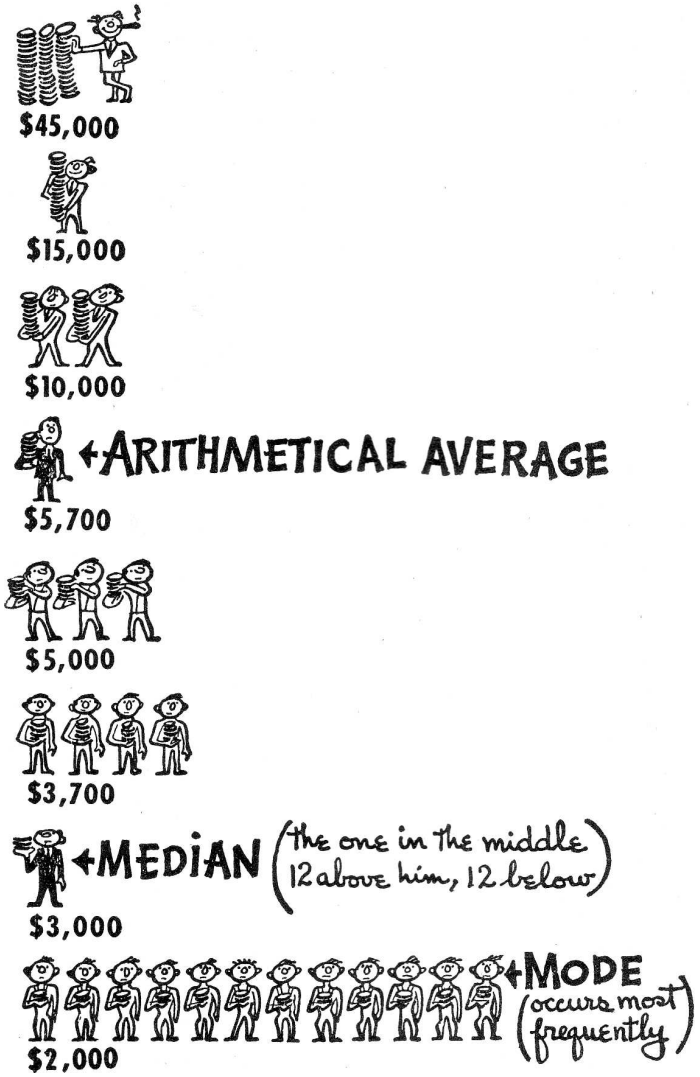
  The arithmetic mean is defined as

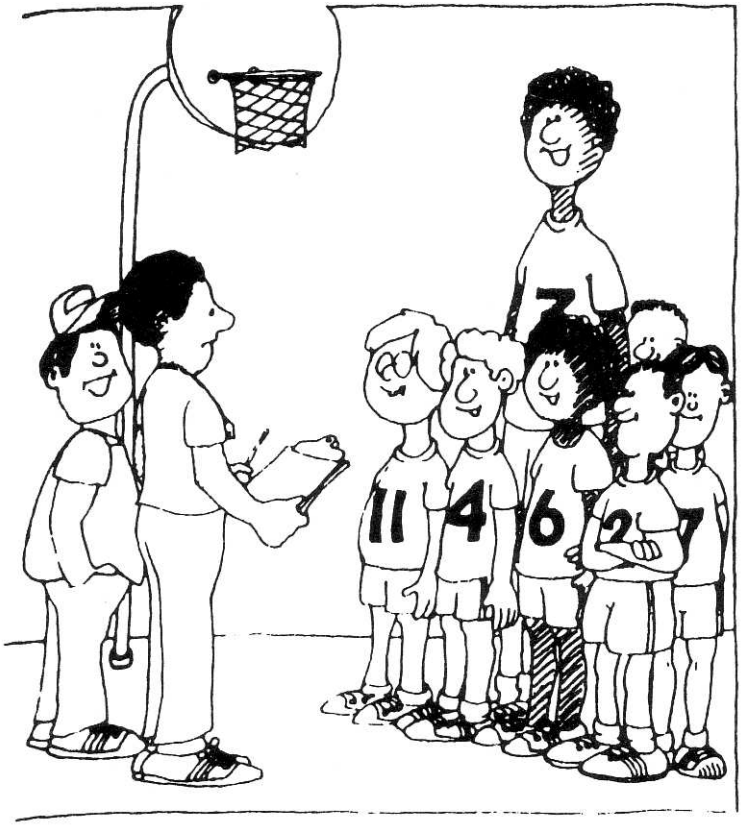
$$\bar{x} = \hat{\mu}(X) = \frac{1}{n}\sum_{i=0}^{n-1} x_i.$$

  The arithmetic mean is only applicable to metric attributes.

- Even though the arithmetic mean is the most common localization measure, the **median** is preferable if
  - there are few sample cases,
  - the distribution is asymmetric, and/or
  - one expects that outliers are present.

How to Lie with Statistics





»Sollen wir das arithmetische Mittel als durchschnittliche Körpergröße nehmen und den Gegner erschrecken, oder wollen wir ihn einlullen und nehmen den Median?«

Dispersion Measures: Range and Interquantile Range

A man with his head in the freezer and feet in the oven
is *on average* quite comfortable.

old statistics joke

- **Range** R
  The range of a data set is the difference
  between the maximum and the minimum value.

$$R = x_{\max} - x_{\min} = \max_{i=0}^{n-1} x_i - \min_{i=0}^{n-1} x_i$$

- **Interquantile Range**
  The p-quantile of a data set is a value such that a fraction of p
  of all sample values are smaller than this value.
  (The median is the $\frac{1}{2}$-quantile.)

  The p-interquantile range, $0 < p < \frac{1}{2}$, is the difference between
  the $(1-p)$-quantile and the p-quantile.

  The most common is the *interquartile range* ($p = \frac{1}{4}$).

Dispersion Measures: Average Absolute Deviation

- **Average Absolute Deviation**
  The average absolute deviation is the average of the absolute deviations of the sample values from the median or the arithmetic mean.

- Average Absolute Deviation from the **Median**

$$d_{\tilde{x}} = \frac{1}{n} \sum_{i=0}^{n-1} |x_i - \tilde{x}|$$

- Average Absolute Deviation from the **Arithmetic Mean**

$$d_{\bar{x}} = \frac{1}{n} \sum_{i=0}^{n-1} |x_i - \bar{x}|$$

- It is always $d_{\tilde{x}} \leq d_{\bar{x}}$, since the median minimizes the sum of absolute deviations (see the definition of the median).

Dispersion Measures: Variance and Standard Deviation

- **(Empirical) Variance** $s^2 = \hat{\sigma}^2(X)$
  It would be natural to define the variance as the average squared deviation:

$$v^2 = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})^2.$$

  However, inductive statistics suggests that it is better defined as (Bessel's correction, after Friedrich Wilhelm Bessel, 1784–1846):

$$s^2 = \frac{1}{n-1} \sum_{i=0}^{n-1} (x_i - \bar{x})^2.$$

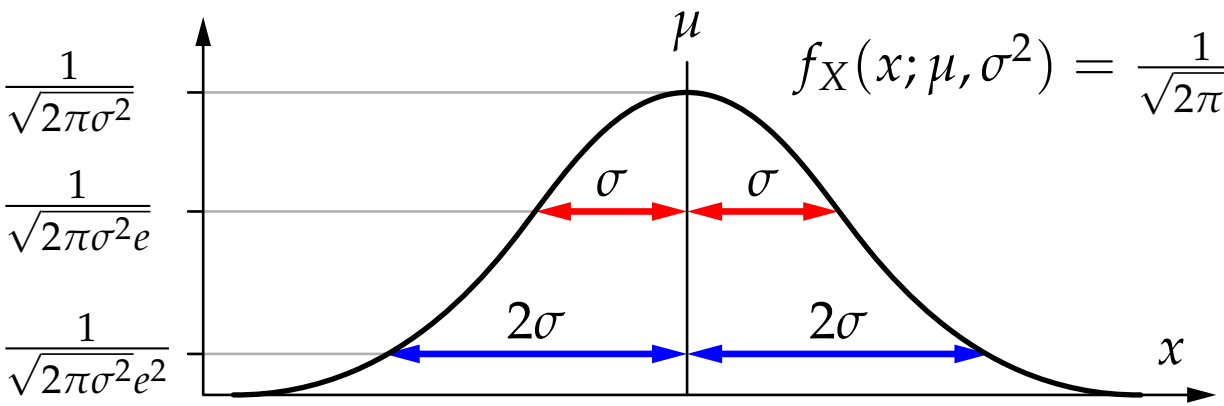
- **(Empirical) Standard Deviation** $s = \hat{\sigma}(X)$
  The standard deviation is the square root of the variance, i.e.,

$$s = \sqrt{s^2} = \sqrt{\frac{1}{n-1} \sum_{i=0}^{n-1} (x_i - \bar{x})^2}.$$

Dispersion Measures: Variance and Standard Deviation

- **Special Case: Normal/Gaussian Distribution**

  The variance/standard deviation provides information about the height of the mode and the width of the curve.



- μ: expected value, estimated by mean value $\bar{x} = \hat{\mu}$

  σ^2: variance, estimated by (empirical) variance $s^2 = \hat{\sigma}^2$

  σ: standard deviation, estimated by (empirical) standard deviation $s = \hat{\sigma}$

  (Details about parameter estimation $\Rightarrow$ statistics courses.)

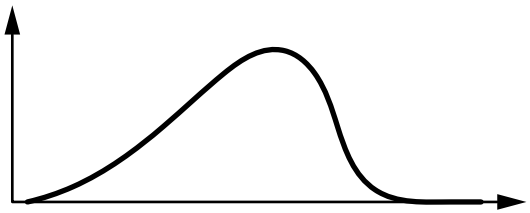
Shape Measures: Skewness

- The **skewness** α_3 (or **skew** for short) measures whether, and if, how much, a distribution differs from a symmetric distribution.

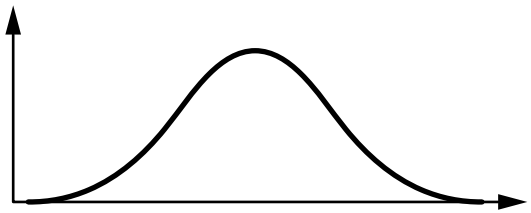
- It is computed from the 3rd moment about the mean, which explains the index 3.

$$\alpha_3 = \frac{1}{n \cdot v^3} \sum_{i=0}^{n-1} (x_i - \bar{x})^3 = \frac{1}{n} \sum_{i=0}^{n-1} z_i^3$$

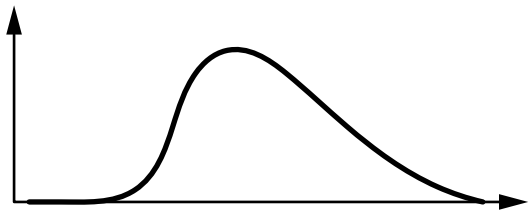
$$\text{where} \quad z_i = \frac{x_i - \bar{x}}{v} \quad \text{and} \quad v^2 = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})^2.$$



$\alpha_3 < 0$: right steep



$\alpha_3 = 0$: symmetric



$\alpha_3 > 0$: left steep

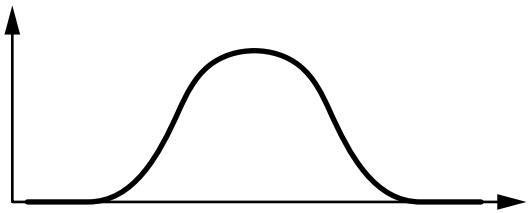
Shape Measures: Kurtosis

- The **kurtosis** or **excess** α_4 measures how much a distribution is arched, usually compared to a Gaussian distribution.

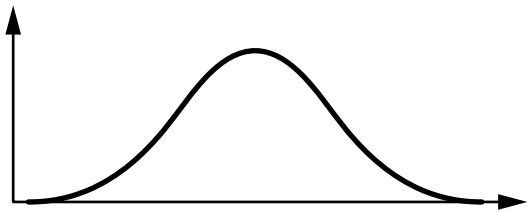
- It is computed from the 4th moment about the mean, which explains the index 4.

$$\alpha_4 = \frac{1}{n \cdot v^4} \sum_{i=0}^{n-1} (x_i - \bar{x})^4 = \frac{1}{n} \sum_{i=0}^{n-1} z_i^4$$

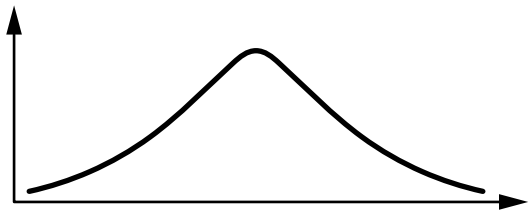
$$\text{where} \quad z_i = \frac{x_i - \bar{x}}{v} \quad \text{and} \quad v^2 = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})^2.$$



$\alpha_4 < 3$: platikurtic



$\alpha_4 = 3$: Gaussian



$\alpha_4 > 3$: leptokurtic

Moments of Data Sets

- The k-th (raw) **moment** of a data set is defined as

$$M_k = \frac{1}{n} \sum_{i=0}^{n-1} x_i^k.$$

The first (raw) moment is the **mean** $M_1 = \bar{x}$ of the data set.
Using the moments of a data set the **variance** s^2 can also be written as

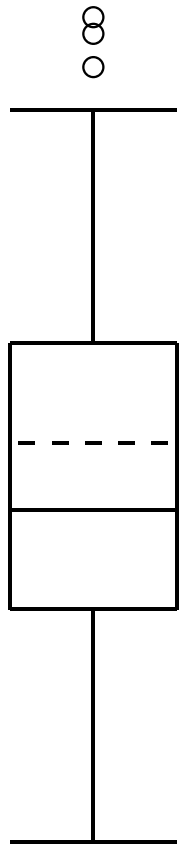
$$s^2 = \frac{1}{n-1} \left(M_2 - \frac{1}{n} M_1^2 \right) \qquad \text{and also} \qquad v^2 = \frac{1}{n} M_2 - \frac{1}{n^2} M_1^2.$$

- The k-th **moment about the mean** is defined as

$$\bar{M}_k = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})^k.$$

It is $\bar{M}_1 = 0$ and $\bar{M}_2 = v^2$ (i.e., the **average squared deviation**).
The **skewness** is $\alpha_3 = \frac{\bar{M}_3}{\bar{M}_2^{3/2}}$ and the **kurtosis** is $\alpha_4 = \frac{\bar{M}_4}{\bar{M}_2^2}$.

Visualizing Characteristic Measures: Box Plots



outliers

$x_{\max}$ maximum
(or $\max\{x \mid x \leq Q_3 + 1.5(Q_3 - Q_1)\}$
or 97.5% quantile)

Q_3 3. quartile

$\bar{x}$ arithmetic mean

$\tilde{x} = Q_2$ median/2. quartile

Q_1 1. quartile

(or $\min\{x \mid x \geq Q_1 - 1.5(Q_3 - Q_1)\}$
or 2.5% quantile)

$x_{\min}$ minimum

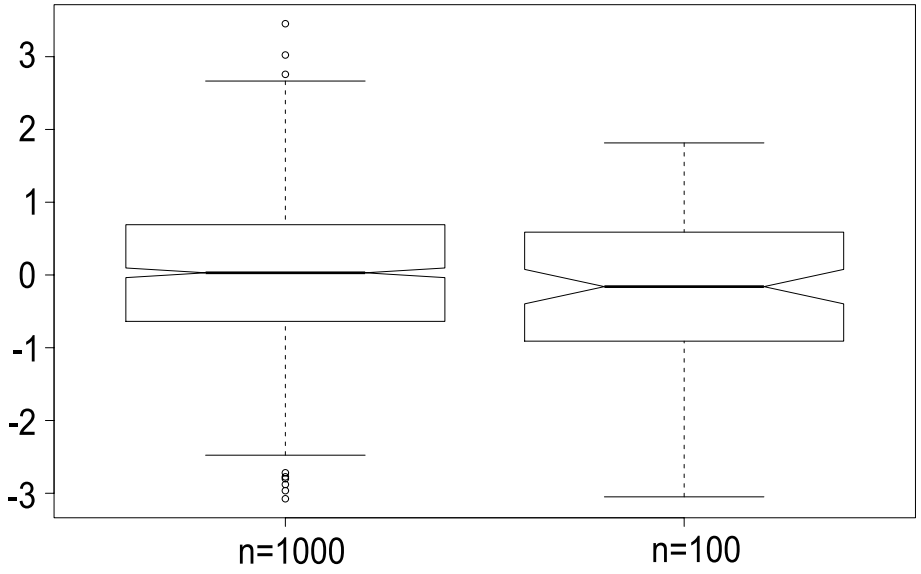
A box plot is a common way to combine some important characteristic measures into a single graphical representation.

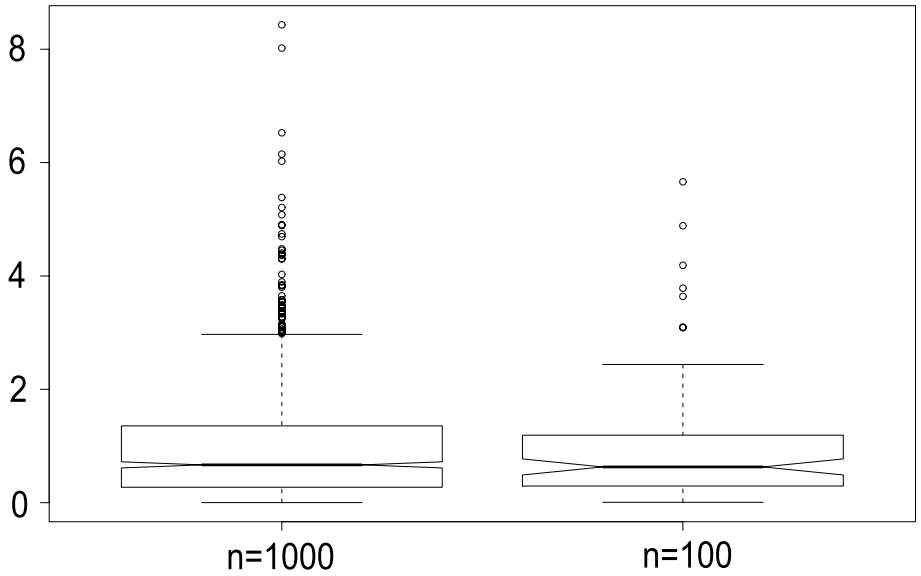
Often the central box is drawn constricted $(\rangle\langle)$ w.r.t. the meadian (or the arithmetic mean) in order to emphasize its location.

The "whiskers" are often limited in length to $1.5(Q_3 - Q_1)$. Data points beyond these limits are suspected to be outliers.

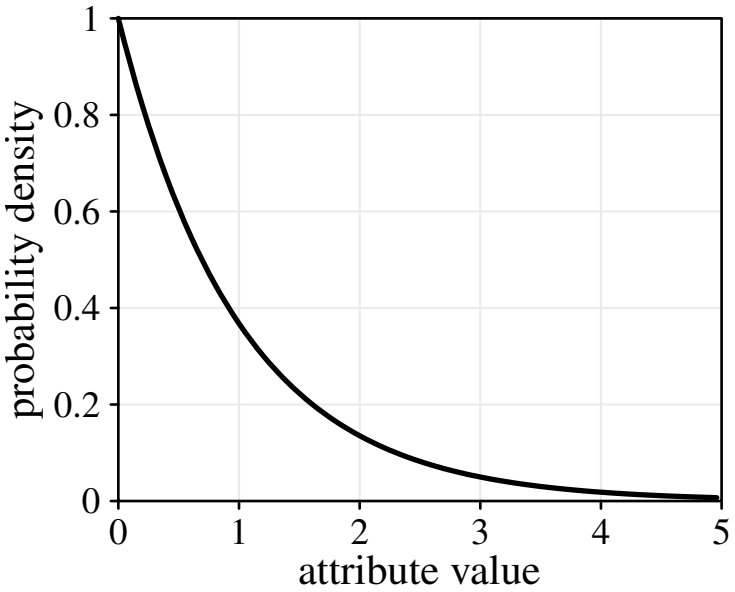
Box plots are often used to get a quick impression of the distribution of the data by showing them side by side for several attributes or data subsets.

Box Plots: Examples





- left top: two samples from a standard normal distribution.
- left bottom: two samples from an exponential distribution.
- right bottom:
  probability density function of the exponential distribution with $\lambda = 1$.



Computing Characteristic Measures (especially for Big Data)

Computing Characteristic Measures: Arithmetic Mean

- **Trivial Approach:** Implement the definition of the mean.

```
def mean (X):
    n = 0; z = 0
    for x in X:
        z += x
        n += 1
    return z / n
```

Reminder:

$$\bar{x} = \hat{\mu}(X) = \frac{1}{n}\sum_{i=0}^{n-1} x_i$$

- Even simpler: exploit basic Python functions. (not an online approach!)

```
def mean (X):
    return sum(X) / len(X)
```

- Problem: **For large data sets, the sum can become very large.**
- Why is a large sum a problem?
  - Computers store real-valued numbers only with finite precision.
  - As a consequence, summing a very large and a very small number can lead to a loss of precision (result is not exactly correct).

Excursion: Number Representation in a Computer

- In contrast to e.g. Roman numerals, which is an **additive counting system** with a supplementary rule for the subtractive writing of certain numbers, we usually use a place-value system.

  Roman numerals:

symbol	I	V	X	L	C	D	M
value	1	5	10	50	100	500	1000

  Examples: `MMXVII` $\widehat{=}$ 2017, `MCMLXVI` $\widehat{=}$ 1966, `MDCCXCIV` $\widehat{=}$ 1794.

- A **place-value**, **denominational** or **polyadic number system** is a number system, in which the (additive) value of a symbol depends on its location (place, position). Examples (Base 10):

$$742 = 7 \cdot 100 + 4 \cdot 10 + 2 \cdot 1 = 7 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0$$

$$0.358 = 3 \cdot \tfrac{1}{10} + 5 \cdot \tfrac{1}{100} + 8 \cdot \tfrac{1}{1000} = 3 \cdot 10^{-1} + 5 \cdot 10^{-2} + 8 \cdot 10^{-3}$$

- A decisive element for the development of place-value systems was the introduction of the zero “0” as a gap indicator (for unoccupied locations). Only this allowed a further development of mathematics / arithmetics.

Excursion: Number Representation in a Computer

- In principle, any number can be chosen as the base of a number system.
- In the **duodecimal** (base 12) and the **sexagesimal system** (base 60) (simple) fractions are often easier to express in a place-value system.

fraction	$^1/_2$	$^1/_3$	$^1/_4$	$^1/_5$	$^1/_6$	$^1/_8$
decimal system	0.5	0.333 . . .	0.25	0.2	0.1666 . . .	0.125
duodecimal system	0.6	0.4	0.3	0.2497 . . .	0.2	0.15
sexagesimal system	$0.d_{30}$	$0.d_{20}$	$0.d_{15}$	$0.d_{12}$	$0.d_{10}$	$0.7d_{30}$

- The **binary system** (base 2) used in computers has the advantage of having the smallest possible number of digits, namely only two: 0 und 1.
- Since we may need to work with different number systems (different bases), we need a notation how a number is to be interpreted.

  We will indicate the base as a subscript to the right of the number (if this subscript is missing: base 10):

  $342_{12} = 482_{10} = 482 = 742_8 = 2122_6 = 13202_4 = 122212_3 = 111100010_2.$

Excursion: Number Representation in a Computer

- The number 742 may also be interpreted in a number system with a base other than 10, e.g. in the **octal system** (base 8):

$$742_8 = 7 \cdot 8^2 + 4 \cdot 8 + 2 \cdot 1 = 448 + 32 + 2 = 482_{10}.$$

  Hence it is important to explicitly state the base, unless the context already makes it clear which base is used.

- If a number is to be interpreted in a number system with base b, then the valid **digit symbols** (or **digits** for short) are $0, \ldots, b-1$. In contrast, b is not a valid digit in a number system with base b. Rather:

$$8_8 = 8 \cdot 8^0 = 1 \cdot 8^1 + 0 \cdot 8^0 = 10_8.$$

- In computing the **hexadecimal system** (base 16) is also very common. The hexadecimal digits are

$$\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}.$$

  That is, $\mathtt{A}_{16} = 10_{10}$, $\mathtt{B}_{16} = 11_{10}$, $\mathtt{C}_{16} = 12_{10}$, $\mathtt{D}_{16} = 13_{10}$, $\mathtt{E}_{16} = 14_{10}$, $\mathtt{F}_{16} = 15_{10}$.

Excursion: Number Representation in a Computer

- General interpretation of a number in a number system with base b and digits $d_k, \ldots, d_1, d_0$, with $k \geq 0$ and $\forall i, 0 \leq i \leq k : \ 0 \leq d_i < b$:

$$[d_k \ldots d_1 d_0]_b \ = \ d_k \cdot b^k \ + \ \ldots \ + \ d_1 \cdot b^1 \ + \ d_0 \cdot b^0.$$

  Reminder: $b^1 = b$, $b^0 = 1$.
  Remark: The symbol "$\forall$" means "for all".
  The expression "$\forall i, 0 \leq i \leq k : \ 0 \leq d_i < b$" therefore means
  "For all i that lie between 0 (included) and k (included),
  the digit d_i lies between 0 (included) and b (excluded)."

- Some programming languages support stating numbers (as constants) in certain other number systems (except the standard base 10), usually in the **binary** (base 2), **octal** (base 8) and **hexadecimal system** (base 16).

  As an example, we consider Python (stating the constant n):

base b	Python	example
2	`0b`n	`0b101010`
8	`0o`n	`0o742`

base b	Python	example
10	n	`12345`
16	`0x`n	`0xCAFE`

Excursion: Number Representation in a Computer

- Numbers in a computer come in two fundamentally different forms: **integer numbers** (integers, ints) and **floating-point numbers** (floats).

  (Remark: **fixed-point numbers** are essentially integers, with a certain fixed number of digits interpreted as being after the decimal point.)

- In many programming languages, **integer numbers** have a fixed size.

	size bits	value range unsigned		signed	
`char/byte`	8	0...	255	−128...	127
`short`	16	0...	65535	−32768...	32767
`int`	32	0...	4294967295	−2147483648...	2147483647
`long`	64	0...	18446744073709551615	−9223372036854775808...	9223372036854775807

- This can lead to problems with **over-** or **underflow**:
  Example in C/C++: $2\,000\,000\,000 + 3\,000\,000\,000 = -1\,994\,967\,296$

- However, in Python **arbitrarily large integer numbers** can be represented.
  $\Rightarrow$ Large sums of integers are not a problem (in Python).

Floating-point Numbers

- Floating-point numbers are represented according to the IEEE 754 norm.
  (IEEE Standard for Binary Floating-Point Arithmetic for Microprocessor Systems)

- A floating-point number x is generally represented as

$$x = s \cdot m \cdot b^e.$$

  It consists of the following elements:

  - the sign s (1 bit),
  - the mantissa m (p bits),
  - the base b (for IEEE 754 it is $b = 2$),
  - the exponent e (r bits).

- Different specific formats of floating-point numbers differ in the number of bits that are used for the mantissa (p bits) and the exponent (r bits).

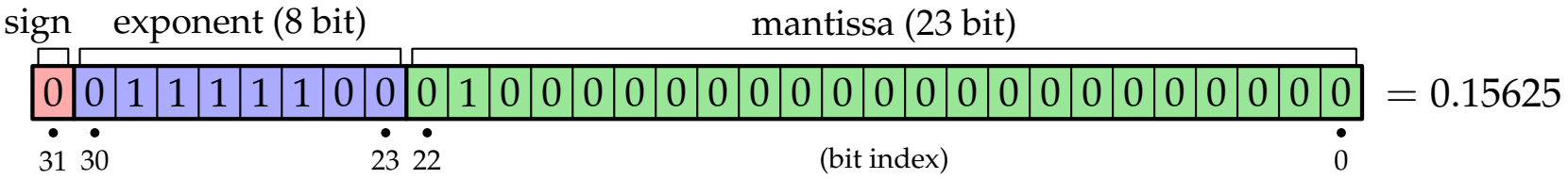
Floating-point Numbers

- The sign $s = (-1)^S$ is stored in a single bit S,
  such that $S = 0$ for positive and $S = 1$ for negative numbers.
- The exponent e is stored as a non-negative binary number E
  by adding a fixed bias B: $E = e + B$.
- The mantissa $1 \leq m < 2$ is a value that is computed from the p mantissa bits with value M as $m = 1 + M/2^p$. One may also say that the mantissa bit pattern is extended by a 1 (and a decimal point) to the left: $m = 1.M$.
- In order to ensure the mantissa encoding, a *normalization* is necessary.
- Some typical floating-point formats used in computers are:

type	size $(1 + r + p)$	exponent (r)	mantissa (p)	values e	bias B
single	32 bit	8 bit	23 bit	$-126 \leq e \leq 127$	127
double	64 bit	11 bit	52 bit	$-1022 \leq e \leq 1023$	1023
quadruple	128 bit	15 bit	112 bit	$-16382 \leq e \leq 16383$	16383

Floating-point Numbers

- Example of a single precision floating-point number:



- Floating-point formats of the IEEE 754 type have the following restrictions:
  - Numbers with the smallest absolute values are not normalized. (Denormalization is indicated by the exponent.)
  - The relative distance of two floating-point numbers is greater than ϵ and less than or equal to 2ϵ. Specifically the distance between 1 and the next larger floating-point number is equal to 2ϵ.

type	ϵ	decimals	smallest absolute number (normalized)	(denormalized)	largest number
single	$2^{-(23+1)} \approx 6.0 \cdot 10^{-8}$	7...8	$2^{-126} \approx 1.2 \cdot 10^{-38}$	$2^{-23} \cdot 2^{-126} \approx 1.4 \cdot 10^{-45}$	$(2 - 2^{-23}) \cdot 2^{127} \approx 3.4 \cdot 10^{38}$
double	$2^{-(52+1)} \approx 1.1 \cdot 10^{-16}$	15...16	$2^{-1022} \approx 2.2 \cdot 10^{-308}$	$2^{-52} \cdot 2^{-1022} \approx 4.9 \cdot 10^{-324}$	$(2 - 2^{-52}) \cdot 2^{1023} \approx 1.8 \cdot 10^{308}$

Floating-point Numbers

- Standard Python uses double precision floating-point numbers.
  (The library `numpy` allows you to specify the precision via a "`dtype`" argument.)

  $\Rightarrow$ In Python floating-point numbers can **not** be **arbitrarily large**.

  $\Rightarrow$ In Python the **precision** of floating-point numbers **is limited**.

- Artificial example (non-IEEE 754) with $p = 4$ (i.e. mantissa has 4 bits).

  Compute $48 + 3$ in such floating-point numbers:

number	encoded	shifted exponent
48	$1.1000 \cdot 2^5$	$1.1000 \cdot 2^5$
$+\,3$	$+\,1.1000 \cdot 2^1$	$+\,0.0001 \cdot 2^5$
$= 50$	$= 1.1001 \cdot 2^5$	$= 1.1001 \cdot 2^5$

  $\Rightarrow$ If exponents are very different, low-value digits may disappear.

  The errors can even accumulate over multiple additions.

Computing Characteristic Measures: Arithmetic Mean

- **Offline Approach:** $\hat{\mu}(X) = \frac{1}{n} \sum_{i=0}^{n-1} x_i$ (see implementation from before).

  (Remark: The possibility of random access yields no advantage here.)

- **Naïve Online Approach:**

  Maintain counter i and (partial) sum $z_i = \sum_{k=0}^{i-1} x_k$.

  Initialize $i = 0$ and $z_0 = 0$.
  For each data point x_i, update $z_{i+1} \leftarrow z_i + x_i$ and $i \leftarrow i + 1$.

  At any time $\mu_i = \hat{\mu}(X_{0:i}) = \frac{z_i}{i}$ can be computed (provided $i > 0$).

- **Problems of the Naïve Online Approach:**
  - Unless there are positive and negative numbers that partially cancel, the absolute value of the sum tends to grow with the number of elements.

  $\Rightarrow$ The floating-point number representing the sum may overflow.

  $\Rightarrow$ Adding a small number to a large sum may lose precision.

Computing Characteristic Measures: Arithmetic Mean

- **Improved Online Approach:**

  Maintain counter i and (partial) mean $\mu_i = \hat{\mu}(X_{0:i})$.

  Initialize $i = 0$ and $\mu_0 = 0$.
  For each data point x_i, update μ_{i+1} (see below) and $i \leftarrow i + 1$.

$$\mu_{i+1} = \frac{1}{i+1}\sum_{k=0}^{i} x_k = \frac{1}{i+1}\left(x_i + \sum_{k=0}^{i-1} x_k\right) = \frac{x_i + i \cdot \mu_i}{i+1}$$

$$= \frac{x_i + (i+1)\cdot\mu_i - \mu_i}{i+1} = \mu_i + \frac{x_i - \mu_i}{i+1}$$

- **Advantage of the Improved Online Approach:**

  The mean of the preceding elements is always directly available.
  Numbers are kept smaller in absolute value (mean instead of sum).
  $\Rightarrow$ over- and underflow are impossible. (since $\forall i : \mu_i \in [\min(X), \max(X)]$)

- **Problem of the Improved Online Approach:**

  If the numbers are very different, precision may still be lost.

Kahan Summation Algorithm

- **Simple (Offline) Solution:** Sort numbers by absolute value before summation.
  - Sorting is costly (generally $O(n \log n)$ time).
  - Not well suited for online / stream processing.
- **Kahan Summation Algorithm** (not(!) mean computation — yet)
  - Each addition might result in a small loss of precision.
  - Determine and accumulate the losses for correction.

function kahan (X): [William Kahan 1965]

```
z ← 0                    # actual sum
c ← 0                    # correction variable
for x in X:              # traverse the elements / data points
    y ← x − c            # incorporate correction into value
    t ← z + y            # compute the next sum
    c ← (t − z) − y      # (t − z) is the actual increase
    z ← t                # c is the difference between
return z                 #     the correct and the actual increase
```

Kahan Summation Algorithm: Example

variable	binary			decimal
$z =$	$0.0000_2 \cdot 2^5$			0_{10}
$c =$	$0.0000_2 \cdot 2^0$			0_{10}
$x =$	$1.1000_2 \cdot 2^5$			48_{10}
$y =$	$1.1000_2 \cdot 2^5$	$-0.0000_2 \cdot 2^0$		
$=$	$1.1000_2 \cdot 2^5$			48_{10}
$t =$	$0.0000_2 \cdot 2^0$	$+1.1000_2 \cdot 2^5$		
$=$	$1.1000_2 \cdot 2^5$			48_{10}
$c =$	$(\ 1.1000_2 \cdot 2^5$	$-0.0000_2 \cdot 2^0\ )$	$-1.1000_2 \cdot 2^5$	
$=$	$0.0000_2 \cdot 2^0$			0_{10}
$z =$	$1.1000_2 \cdot 2^5$			48_{10}
$x =$	$1.1000_2 \cdot 2^1$			3_{10}
$y =$	$1.1000_2 \cdot 2^1$	$-0.0000_2 \cdot 2^0$		
$=$	$1.1000_2 \cdot 2^1$			3_{10}
$t =$	$1.1000_2 \cdot 2^5$	$+1.1000_2 \cdot 2^1$		
$=$	$1.1001_2 \cdot 2^5$			50_{10}
$c =$	$(\ 1.1001_2 \cdot 2^5$	$-1.1000_2 \cdot 2^5\ )$	$-1.1000_2 \cdot 2^1$	
$=$	$-1.0000_2 \cdot 2^0$			-1_{10}
$z =$	$1.1001_2 \cdot 2^5$			50_{10}

The correction variable c indicates by how much the sum in the variable z differs from the actual sum.

Here: $c = -1$, that is, z is too small by 1 (actual sum: 51).

Note, however: The correction is not always fully exact.

Computing Characteristic Measures: Variance

- **Trivial Approach:** Implement the definition of the variance.

```
def var (X):
    n = 0; z = y = 0
    for x in X:
        z += x; n += 1
    mean = z / n
    for x in X:
        y += (x - mean)**2
    return y / (n-1)
```

Reminder:

$$s^2 = \hat{\sigma}^2(X) = \frac{1}{n-1} \sum_{i=0}^{n-1} (x_i - \bar{x})^2$$

- Even simpler: exploit basic Python functions and list comprehension.

(not an online approach!)

```
def var (X):
    mean = sum(X) / len(X)
    return sum([(x - mean)**2 for x in X]) / (len(X)-1)
```

- **Problems of the Trivial Approach:**
  - Data needs to be traversed **twice** $\Rightarrow$ not applicable in online setting.
  - The sum of squared deviations may be even larger than the sum of values.

Computing Characteristic Measures: Variance

One Pass Solution: With some fairly simple transformations, we can rewrite the computation of the variance as:

$$
\begin{aligned}
s^2 &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2 = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_i^2-2x_i\bar{x}+\bar{x}^2\right) \\
&= \frac{1}{n-1}\left(\sum_{i=1}^{n}x_i^2-2\bar{x}\sum_{i=1}^{n}x_i+\sum_{i=1}^{n}\bar{x}^2\right) \\
&= \frac{1}{n-1}\left(\sum_{i=1}^{n}x_i^2-2n\bar{x}^2+n\bar{x}^2\right) = \frac{1}{n-1}\left(\sum_{i=1}^{n}x_i^2-n\bar{x}^2\right) \\
&= \frac{1}{n-1}\left(\sum_{i=1}^{n}x_i^2-\frac{1}{n}\left(\sum_{i=1}^{n}x_i\right)^2\right) = \frac{1}{n-1}\left(M_2-\frac{1}{n}M_1^2\right)
\end{aligned}
$$

- Advantage: The sums $\sum_{i=1}^{n} x_i$ and $\sum_{i=1}^{n} x_i^2$ can both be computed in the same traversal of the data and from them both mean and variance can be computed.

Computing Characteristic Measures: Variance

- **Naïve Online Approach:**

  Maintain counter i and (partial) sums $z_i = \sum_{k=0}^{i-1} x_i$ and $q_i = \sum_{k=0}^{i-1} x_i^2$.

  Initialize $i = 0$, $z_0 = 0$ and $q_0 = 0$.

  For each data point x_i, update $z_{i+1} \leftarrow z_i + x_i$, $q_{i+1} \leftarrow q_i + x_i^2$ and $i \leftarrow i + 1$.

  At any time $\sigma_i^2 = \hat{\sigma}^2(X_{0:i}) = \frac{1}{i-1}\left(q_i - \frac{1}{i} z_i^2\right)$ can be computed (provided $i > 1$).

- **Problems of the Naïve Online Approach:**
  - Unless there are positive and negative numbers that partially cancel, the absolute value of the sum tends to grow with the number of elements.
  - The sum of squared values tends to grow even larger than the sum of values.

  $\Rightarrow$ The floating-point number representing the sums may overflow.

  $\Rightarrow$ Adding a small number to a large sum may lose precision. (However, this can be solved with the Kahan summation algorithm.)

Computing Characteristic Measures: Variance

Simple Way to Improve the Naive Online Approach:

- **Mean Shift**
  - The variance is *invariant* to a *shift* of the input:

$$\hat{\sigma}^2(X) = \hat{\sigma}^2(X') \qquad \text{with } \forall i \in \{0, \ldots, n-1\} : x'_i = x_i - \Delta$$

  - Objective: Choose Δ so that the mean is minimized.
  - This also minimizes the squared values used in the naive approach.

    $\Rightarrow$ Smaller sums, numerical errors are reduced.

- **Implementation**
  - Estimate some proper value for the shift Δ (this is not trivial).
  - One pass computation of variance with shift:

$$\hat{\sigma}^2(X) = \frac{1}{n-1}\left(\sum_{i=0}^{n-1} x_i'^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} x'_i\right)^2\right) = \frac{1}{n-1}\left(\sum_{i=0}^{n-1} (x_i - \Delta)^2 - \frac{1}{n}\left(\sum_{i=0}^{n-1} (x_i - \Delta)\right)^2\right)$$

Computing Characteristic Measures: Variance

- **Improved Online Approach:** We consider $\delta_j^2 = \sum_{k=0}^{j-1}(x_k - \mu_j)^2$.

$$
\begin{aligned}
\delta_{i+1}^2 - \delta_i^2 &= \sum_{k=0}^{i}(x_k - \mu_{i+1})^2 - \sum_{k=0}^{i-1}(x_k - \mu_i)^2 \\
&= (x_i - \mu_{i+1})^2 + \sum_{k=0}^{i-1}(x_k - \mu_{i+1})^2 - \sum_{k=0}^{i-1}(x_k - \mu_i)^2 \\
&= (x_i - \mu_{i+1})^2 + \sum_{k=0}^{i-1}\left(x_k^2 - 2x_k\mu_{i+1} + \mu_{i+1}^2 - x_k^2 + 2x_k\mu_i - \mu_i^2\right) \\
&= (x_i - \mu_{i+1})^2 + \sum_{k=0}^{i-1}\left((\mu_{i+1}^2 - \mu_i^2) - 2x_k(\mu_{i+1} - \mu_i)\right) \\
&= (x_i - \mu_{i+1})^2 + (\mu_{i+1} - \mu_i)\sum_{k=0}^{i-1}\left((\mu_{i+1} + \mu_i) - 2x_k\right) \\
&= (x_i - \mu_{i+1})^2 + (\mu_{i+1} - \mu_i)\left(\sum_{k=0}^{i-1}(\mu_{i+1} - x_k) + \underbrace{\sum_{k=0}^{i-1}(\mu_i - x_k)}_{=0}\right)
\end{aligned}
$$

Computing Characteristic Measures: Variance

- **Improved Online Approach:** We consider $\delta_j^2 = \sum_{k=0}^{j-1}(x_k - \mu_j)^2$.

$$
\begin{aligned}
\delta_{i+1}^2 - \delta_i^2 &= \ldots \qquad \text{(continued from preceding slide)} \\
&= (x_i - \mu_{i+1})^2 + (\mu_{i+1} - \mu_i)\left(\sum_{k=0}^{i-1}(\mu_{i+1} - x_k) + \underbrace{\sum_{k=0}^{i-1}(\mu_i - x_k)}_{=0}\right) \\
&= (x_i - \mu_{i+1})^2 + (\mu_{i+1} - \mu_i)\left(\underbrace{\sum_{k=0}^{i}(\mu_{i+1} - x_k)}_{=0} - (\mu_{i+1} - x_i)\right) \\
&= (x_i - \mu_{i+1})^2 + (\mu_{i+1} - \mu_i)(x_i - \mu_{i+1}) \\
&= (x_i - \mu_{i+1}) \cdot ((x_i - \mu_{i+1}) + (\mu_{i+1} - \mu_i)) \\
&= (x_i - \mu_{i+1}) \cdot (x_i - \mu_i) \qquad \underbrace{= \frac{i}{i+1}(x_i - \mu_i)^2}_{\Rightarrow \text{ exercise}}
\end{aligned}
$$

$\Rightarrow$ Smaller squares, therefore smaller sum to compute.

Computing Characteristic Measures: Variance

- **Improved Online Approach:** [B.P. Welford 1962]

  Maintain counter i, (partial) mean $\mu_i = \mu(X_{0:i})$ and $\delta_i^2 = \delta^2(X_{0:i})$.

  Initialize $i = 0$ and $\mu_0 = 0$ and $\delta_0^2 = 0$.

  For each data point x_i, update μ_{i+1} and δ_{i+1}^2 (see below) and $i \leftarrow i + 1$.

$$\begin{aligned} \mu_{i+1} &\leftarrow \mu_i + \tfrac{1}{i+1}(x_i - \mu_i) \\ \delta_{i+1}^2 &\leftarrow \delta_i^2 + (x_i - \mu_{i+1})(x_i - \mu_i) \end{aligned}$$

- **Advantages of the Improved Online Approach:**
  - Mean is readily available and variance can be computed as $\sigma_i^2 = \frac{1}{i-1}\,\delta_i^2$.
  - Numbers are kept smaller $\Rightarrow$ over- or underflow is (very) unlikely.
- **Problems of the Improved Online Approach:**
  - If the numbers are very different, precision may still be lost.

    $\Rightarrow$ Can be solved with the Kahan summation algorithm.

Computing Characteristic Measures: Mean & Variance

Another Way to Improve the Naive Online Approach:

- **Recursive Computation**
  - Core problem of the naive approach: Loss of precision, because (relatively) small numbers are added to large sums.
  - Solution idea: Compute total sums by adding partial sums. This approach can be implemented recursively.

- **Mean** For $n \geq 2$, choose r, $0 < r < n$, ideally $r = \lceil \frac{n}{2} \rceil$ or $r = \lfloor \frac{n}{2} \rfloor$.

$$\begin{aligned}
\mu_{0:n} = \hat{\mu}(X_{0:n}) &= \frac{1}{n} \sum_{i=0}^{n+1} x_i = \frac{1}{n} \left(\sum_{i=0}^{r-1} x_i + \sum_{i=r}^{n-1} x_i \right) \\
&= \frac{1}{n} \left(r \left(\frac{1}{r} \sum_{i=0}^{r-1} x_i \right) + (n-r) \left(\frac{1}{n-r} \sum_{i=r}^{n-1} x_i \right) \right) \\
&= \frac{1}{n} \Big(r \mu_{0:r} + (n-r) \mu_{r:n} \Big) = \frac{r}{n} \mu_{0:r} + \frac{n-r}{n} \mu_{r:n}
\end{aligned}$$

Computing Characteristic Measures: Mean & Variance

Variance For $n \geq 2$, choose r, $0 < r < n$, ideally $r = \lceil \frac{n}{2} \rceil$ or $r = \lfloor \frac{n}{2} \rfloor$.

$$\begin{aligned}\delta^2_{0:n} &= \sum_{i=0}^{n-1}(x_i - \mu_{0:n})^2 = \sum_{i=0}^{r-1}(x_i - \mu_{0:n})^2 + \sum_{i=r}^{n-1}(x_i - \mu_{0:n})^2 \\ &= \sum_{i=0}^{r-1}(x_i - (\tfrac{r}{n}\mu_{0:r} + \tfrac{n-r}{n}\mu_{r:n}))^2 + \sum_{i=r}^{n-1}(x_i - (\tfrac{r}{n}\mu_{0:r} + \tfrac{n-r}{n}\mu_{r:n}))^2\end{aligned}$$

Consider only first sum (second sum is considered on next slide):

$$\begin{aligned}&\sum_{i=0}^{r-1}(x_i - (\tfrac{r}{n}\mu_{0:r} + \tfrac{n-r}{n}\mu_{r:n}))^2 \\ &= \sum_{i=0}^{r-1}(x_i - \tfrac{r}{n}\mu_{0:r} - \tfrac{n-r}{n}\mu_{r:n})^2 = \sum_{i=0}^{r-1}((x_i - \mu_{0:r}) + (\tfrac{n-r}{n}\mu_{0:r} - \tfrac{n-r}{n}\mu_{r:n}))^2 \\ &= \sum_{i=0}^{r-1}((x_i - \mu_{0:r})^2 + 2\tfrac{n-r}{n}(x_i - \mu_{0:r})(\mu_{0:r} - \mu_{r:n}) + (\tfrac{n-r}{n}(\mu_{0:r} - \mu_{r:n}))^2) \\ &= \underbrace{\sum_{i=0}^{r-1}(x_i - \mu_{0:r})^2}_{=\delta^2_{0:r}} + 2\tfrac{n-r}{n}(\mu_{0:r} - \mu_{r:n}) \underbrace{\sum_{i=0}^{r-1}(x_i - \mu_{0:r})}_{=0} + (\tfrac{n-r}{n})^2 r\,(\mu_{0:r} - \mu_{r:n})^2\end{aligned}$$

Computing Characteristic Measures: Mean & Variance

Consider only second sum (first sum was considered on preceding slide):

$$\sum_{i=r}^{n-1} (x_i - (\tfrac{r}{n}\,\mu_{0:r} + \tfrac{n-r}{n}\,\mu_{r:n}))^2$$

$$= \sum_{i=r}^{n-1} (x_i - \tfrac{r}{n}\,\mu_{0:r} - \tfrac{n-r}{n}\,\mu_{r:n})^2 \quad = \quad \sum_{i=r}^{n-1} ((x_i - \mu_{r:n}) + (\tfrac{r}{n}\,\mu_{r:n} - \tfrac{r}{n}\,\mu_{0:r}))^2$$

$$= \sum_{i=r}^{n-1} ((x_i - \mu_{r:n})^2 + 2\tfrac{r}{n}(x_i - \mu_{r:n})(\mu_{r:n} - \mu_{0:r}) + (\tfrac{r}{n}(\mu_{r:n} - \mu_{0:r}))^2)$$

$$= \underbrace{\sum_{i=r}^{n-1}(x_i - \mu_{r:n})^2}_{=\delta^2_{r:n}} + 2\tfrac{n-r}{n}(\mu_{0:r} - \mu_{r:n}) \underbrace{\sum_{i=r}^{n-1}(x_i - \mu_{r:n})}_{=0} + (\tfrac{r}{n})^2\,(n-r)(\mu_{r:n} - \mu_{0:r})^2$$

Result:

$$\delta^2_{0:n} = \delta^2_{0:r} + \delta^2_{r:n} + \left((\tfrac{n-r}{n})^2\,r + (\tfrac{r}{n})^2\,(n-r)\right)(\mu_{0:r} - \mu_{r:n})^2$$

$$= \delta^2_{0:r} + \delta^2_{r:n} + \tfrac{r(n-r)}{n}(\mu_{0:r} - \mu_{r:n})^2$$

Special case $r = n - r = \frac{n}{2}$: $\quad \delta^2_{0:n} = \delta^2_{0:r} + \delta^2_{r:n} + \frac{n}{4}(\mu_{0:r} - \mu_{r:n})^2$

Computing Characteristic Measures: Mean & Variance

Even more special case $n = 2, r = 1$:

$$\mu_{0:2} = \frac{1}{2}\mu_{0:1} + \frac{1}{2}\mu_{1:2} = \frac{1}{2}\left(\frac{1}{1}\sum_{i=0}^{0} x_i\right) + \frac{1}{2}\left(\frac{1}{1}\sum_{i=1}^{1} x_i\right) = \frac{1}{2}(x_0 + x_1)$$

$$\begin{aligned}
\delta^2_{0:2} &= \delta^2_{0:1} + \delta^2_{1:2} + \frac{1}{2}(\mu_{0:1} - \mu_{1:2})^2 \\
&= \sum_{i=0}^{0}(x_i - \mu_{0:1})^2 + \sum_{i=1}^{1}(x_i - \mu_{1:2})^2 + \frac{1}{2}(\mu_{0:1} - \mu_{1:2})^2 \\
&= \sum_{i=0}^{0}(x_i - x_0)^2 + \sum_{i=1}^{1}(x_i - x_1)^2 + \frac{1}{2}(x_0 - x_1)^2 \\
&= (x_0 - x_0)^2 + (x_1 - x_1)^2 + \frac{1}{2}(x_0 - x_1)^2 = \frac{1}{2}(x_0 - x_1)^2
\end{aligned}$$

Sanity check:

$$\begin{aligned}
\delta^2_{0:2} &= \sum_{i=0}^{1}(x_i - \mu_{0:2})^2 = \sum_{i=0}^{1}(x_i - \tfrac{1}{2}(x_0 + x_1))^2 \\
&= (x_0 - \tfrac{1}{2}x_0 - \tfrac{1}{2}x_1)^2 + (x_1 - \tfrac{1}{2}x_0 - \tfrac{1}{2}x_1)^2 = (\tfrac{1}{2}x_0 - \tfrac{1}{2}x_1)^2 + (\tfrac{1}{2}x_1 - \tfrac{1}{2}x_0)^2 \\
&= \tfrac{1}{4}(x_0 - x_1)^2 + \tfrac{1}{4}(x_1 - x_0)^2 = \tfrac{1}{2}(x_0 - x_1)^2
\end{aligned}$$

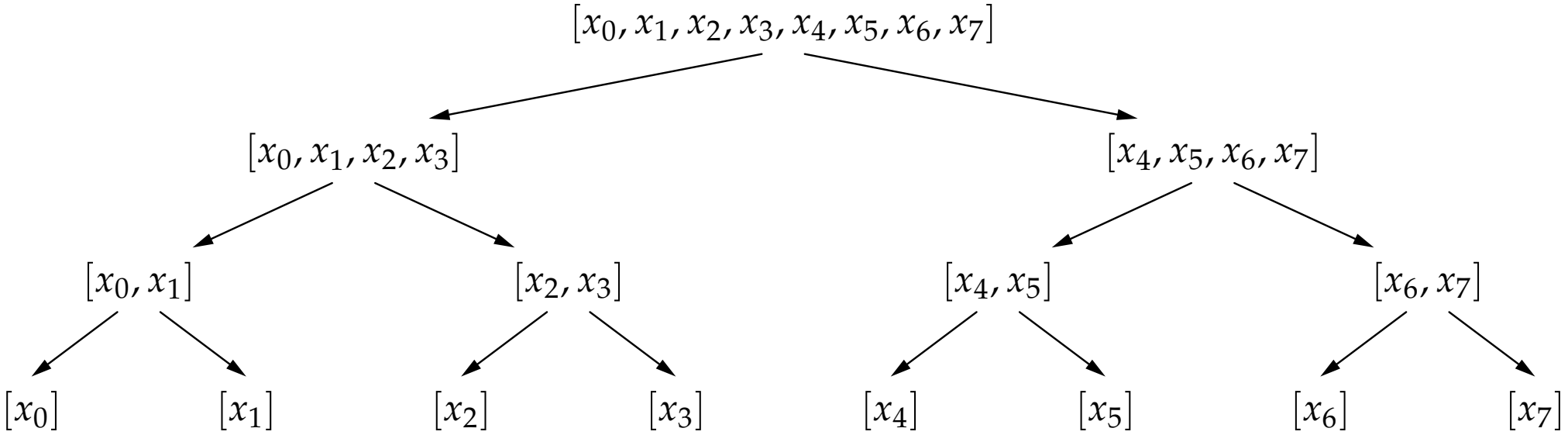
Computing Characteristic Measures: Mean & Variance

- We may also define generally: $\mu_{i:i+1} = x_i$ and $\delta^2_{i:i+1} = 0$.
- **Recursive Processing:** (divide-and-conquer)
  - A divide-and-conquer scheme can be described as a set of subproblems.
  - Here: each subproblem is a section $X_{i:k}$ of the data (specified as $i:k$), for which the (partial) mean $\mu_{i:k}$ and the value $\delta^2_{i:k}$ are to be computed. Hence the solution of a subproblem is a pair $(\mu_{i:k}, \delta^2_{i:k})$.
  - Unless it is $k = i + 1$ (trivial case, see above), a subproblem is split by computing $r = \lfloor \frac{i+k}{2} \rfloor$ to form $X_{i:r}$ and $X_{r:k}$.
  - The two subproblems are processed recursively. This yields their solutions $(\mu_{i:r}, \delta^2_{i:r})$ and $(\mu_{r:k}, \delta^2_{r:k})$.
  - The solutions are combined to produce $(\mu_{i:k}, \delta^2_{i:k})$:

$$\begin{aligned} \mu_{i:k} &= \tfrac{r-i}{k-i}\,\mu_{i:r} + \tfrac{k-r}{k-i}\,\mu_{r:k} \\ \delta^2_{i:k} &= \delta^2_{i:r} + \delta^2_{r:k} + \tfrac{(r-i)(k-r)}{k-i}\,(\mu_{i:r} - \mu_{r:k})^2 \end{aligned}$$

Computing Characteristic Measures: Mean & Variance

Example Stream: ($n = 8$)



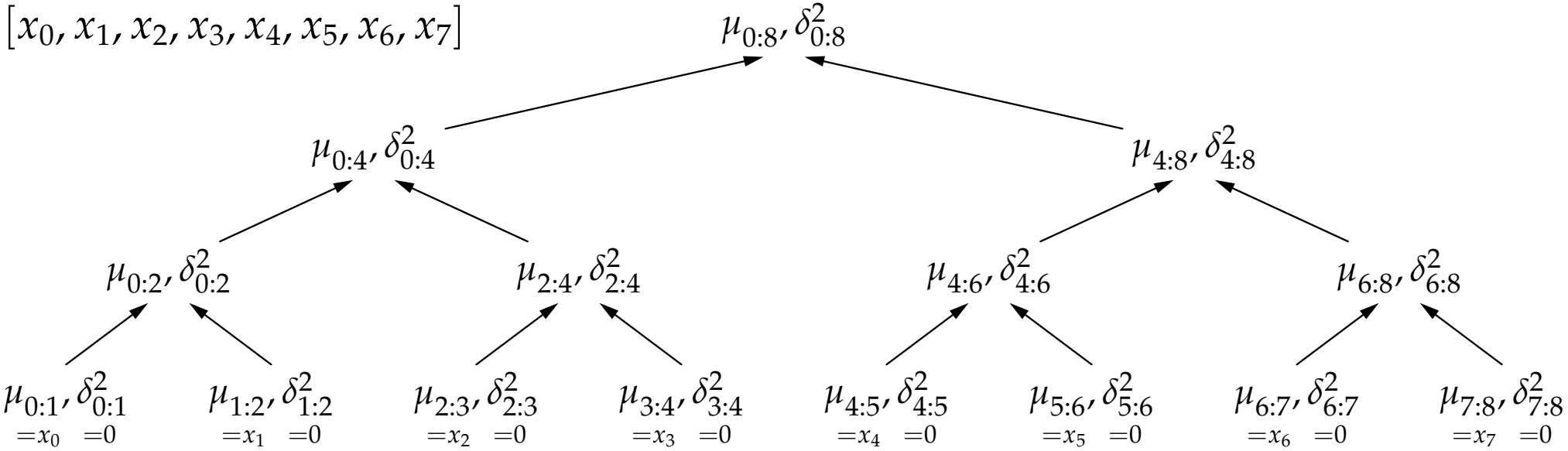
- The stream is recursively split into smaller and smaller substreams.
- Finally single element streams are reached, which are trivial to process.
- The single element streams yield $\mu_{i:i+1} = x_i$ and $\delta^2_{i:i+1} = 0$.
- Solution combination:

$$\begin{aligned} \mu_{i:k} &= \tfrac{1}{2}\left(\mu_{i:r} + \mu_{r:k}\right) \\ \delta^2_{i:k} &= \delta^2_{i:r} + \delta^2_{r:k} + \tfrac{k-i}{4}\left(\mu_{i:r} - \mu_{r:k}\right)^2 \end{aligned}$$

Computing Characteristic Measures: Mean & Variance

Example Stream: $(n = 8)$

$[x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7]$



- Each stream data point is accessed exactly once (level $\ell = 0$).
- The levels above the data points each halven the number of subproblems. For $n = 2^\alpha$, $\alpha \in \mathbb{N}_0$, the number ν of nodes on level ℓ is $\nu(\ell) = n2^{-\ell} = 2^{\alpha-\ell}$.
- The height of the computation tree is $1 + \log_2 n$ (including level 0).
- The total number of nodes (including those on level 0) is $2n - 1$.

Computing Characteristic Measures: Mean & Variance

- If $n \neq 2^\alpha$, $\alpha \in \mathbb{N}_0$, the recursive divide-and-conquer scheme still works, but the more general solution combination formulae are needed. $\Rightarrow$ exercise
- However, a recursive divide-and-conquer scheme requires that (the size n of) the stream is known beforehand.
- As a consequence, it is not (directly) usable for online processing.
- Yet, the scheme resulting for $n = 2^\alpha$, $\alpha \in \mathbb{N}_0$, can be adapted:
  - It is (implicitly) assumed that the final size is some $n = 2^\alpha$, $\alpha \in \mathbb{N}_0$.
  - The stream elements are processed consecutively, one by one.
  - The recursion tree is built from left to right, maintaining in memory solutions to "latest" subproblems per level that have already been solved.
  - Even though this leads (generally) to multiple solutions in memory, these allow to compute the current mean and variance quickly.
  - Implementation: Maintain the "latest" subproblem solutions in a buffer. Propagate new solutions upward in this buffer.

Computing Characteristic Measures: Mean & Variance

init.

ℓ	ϱ
2	—
1	—
0	—

- Start with an empty partial solution buffer.
  (This buffer need not even have full size right from the start.)
- Process stream elements one by one.
- Create partial solution for the next processed stream element.
- Propagate the partial solution up in the buffer,
  merging it with already existing partial solutions,
  until an empty slot in the buffer has been reached.
- Store the created partial solution in the buffer.

Computing Characteristic Measures: Mean & Variance

init.

ℓ	ϱ
2	—
1	—
0	—

x_0

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

- For the 1st stream element, create the partial solution $[0:1, \mu_{0:1} = x_0, \delta^2_{0:1} = 0]$.
- Since the partial solution buffer is empty at level 0, this partial solution can simply be stored.

Computing Characteristic Measures: Mean & Variance

init.

ℓ	ϱ
2	—
1	—
0	—

x_0

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

x_1

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

$1:2, \mu_{1:2}, \delta^2_{1:2}$

- For the 2nd stream element, create the partial solution $[1:2, \mu_{1:2} = x_1, \delta^2_{1:2} = 0]$.
- Since the partial solution buffer is **not** empty at level 0, this partial solution has to be combined with the buffer entry:

$$\begin{aligned} \mu_{0:2} &= \tfrac{1}{2}\mu_{0:1} + \tfrac{1}{2}\mu_{1:2} \\ \delta^2_{0:2} &= \delta^2_{0:1} + \delta^2_{1:2} + \tfrac{1}{2}(\mu_{0:1} - \mu_{1:2})^2 \end{aligned}$$

Computing Characteristic Measures: Mean & Variance

init.

ℓ	ϱ
2	—
1	—
0	—

x_0

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

x_1

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

$1:2, \mu_{1:2}, \delta^2_{1:2}$

x_1

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	—

- For the 2nd stream element, create the partial solution $[1:2, \mu_{1:2} = x_1, \delta^2_{1:2} = 0]$.
- Since the partial solution buffer is **not** empty at level 0, this partial solution has to be combined with the buffer entry:

$$\begin{aligned} \mu_{0:2} &= \tfrac{1}{2}\mu_{0:1} + \tfrac{1}{2}\mu_{1:2} \\ \delta^2_{0:2} &= \delta^2_{0:1} + \delta^2_{1:2} + \tfrac{1}{2}(\mu_{0:1} - \mu_{1:2})^2 \end{aligned}$$

- The buffer is cleared at level 0 (because the entry was combined) and the combined solution is moved up to level 1.
- Since the buffer is empty at level 1, the combined solution is stored there.

Computing Characteristic Measures: Mean & Variance

init.

ℓ	ϱ
2	—
1	—
0	—

x_0

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

x_1

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

$1:2, \mu_{1:2}, \delta^2_{1:2}$

x_1

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	—

x_2

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{1:2}, \delta^2_{1:2}$

- For the 3rd stream element, create the partial solution $[2:3, \mu_{2:3} = x_2, \delta^2_{2:3} = 0]$.
- Since the partial solution buffer is empty at level 0, this partial solution can simply be stored.

Computing Characteristic Measures: Mean & Variance

init.

ℓ	ϱ
2	—
1	—
0	—

x_0

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

x_1

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

$1:2, \mu_{1:2}, \delta^2_{1:2}$

x_1

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	—

x_2

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{1:2}, \delta^2_{1:2}$

x_3

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{2:3}, \delta^2_{2:3}$

$2:4, \mu_{2:4}, \delta^2_{2:4}$

$3:4, \mu_{3:4}, \delta^2_{3:4}$

- For the 4th stream element, create the partial solution $[3:4, \mu_{3:4} = x_3, \delta^2_{3:4} = 0]$.
- Since the partial solution buffer is **not** empty at level 0, this partial solution has to be combined with the buffer entry.

  This yields the partial solution $[2:4, \mu_{2:4}, \delta^2_{2:4}]$, which is moved up.

Computing Characteristic Measures: Mean & Variance

init.

ℓ	ϱ
2	—
1	—
0	—

x_0

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

x_1

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

$1:2, \mu_{1:2}, \delta^2_{1:2}$

x_1

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	—

x_2

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{1:2}, \delta^2_{1:2}$

x_3

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{2:3}, \delta^2_{2:3}$

$2:4, \mu_{2:4}, \delta^2_{2:4}$

$3:4, \mu_{3:4}, \delta^2_{3:4}$

x_3

ℓ	ϱ
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	—

- Since the partial solution buffer is **not** empty at level 1, this partial solution has to be combined with the buffer entry.

  This yields the partial solution $[0:4, \mu_{0:4}, \delta^2_{0:4}]$, which is moved up.

- Since the buffer is empty at level 2, this partial solution can be stored.

Computing Characteristic Measures: Mean & Variance

init.

ℓ	ϱ
2	—
1	—
0	—

x_0

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

x_1

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

$1:2, \mu_{1:2}, \delta^2_{1:2}$

x_1

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	—

x_2

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{1:2}, \delta^2_{1:2}$

x_3

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{2:3}, \delta^2_{2:3}$

$2:4, \mu_{2:4}, \delta^2_{2:4}$

$3:4, \mu_{3:4}, \delta^2_{3:4}$

x_3

ℓ	ϱ
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	—

x_4

ℓ	ϱ
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	$4:5, \mu_{4:5}, \delta^2_{4:5}$

- For the 5th stream element, create the partial solution $[4:5, \mu_{4:5} = x_4, \delta^2_{4:5} = 0]$.
- Since the buffer is empty at level 0, this partial solution can be stored.

Computing Characteristic Measures: Mean & Variance

init.

ℓ	ϱ
2	—
1	—
0	—

x_0

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

x_1

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

$1:2, \mu_{1:2}, \delta^2_{1:2}$

x_1

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	—

x_2

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{1:2}, \delta^2_{1:2}$

x_3

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{2:3}, \delta^2_{2:3}$

$2:4, \mu_{2:4}, \delta^2_{2:4}$

$3:4, \mu_{3:4}, \delta^2_{3:4}$

x_3

ℓ	ϱ
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	—

x_4

ℓ	ϱ
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	$4:5, \mu_{4:5}, \delta^2_{4:5}$

x_5

ℓ	ϱ
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	$4:5, \mu_{4:5}, \delta^2_{4:5}$

$5:6, \mu_{5:6}, \delta^2_{5:6}$

For the 6th stream element, processing is similar to the 2nd element.

Computing Characteristic Measures: Mean & Variance

init.

ℓ	ϱ
2	—
1	—
0	—

x_0

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

x_1

ℓ	ϱ
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

$1:2, \mu_{1:2}, \delta^2_{1:2}$

x_1

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	—

x_2

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{1:2}, \delta^2_{1:2}$

x_3

ℓ	ϱ
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{2:3}, \delta^2_{2:3}$

$2:4, \mu_{2:4}, \delta^2_{2:4}$
$3:4, \mu_{3:4}, \delta^2_{3:4}$

x_3

ℓ	ϱ
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	—

x_4

ℓ	ϱ
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	$4:5, \mu_{4:5}, \delta^2_{4:5}$

x_5

ℓ	ϱ
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	$4:5, \mu_{4:5}, \delta^2_{4:5}$

$5:6, \mu_{5:6}, \delta^2_{5:6}$

x_5

ℓ	ϱ
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	$4:6, \mu_{4:6}, \delta^2_{4:6}$
0	—

Computing Characteristic Measures: Mean & Variance

init.

ℓ	ϱ
3	—
2	—
1	—
0	—

x_0

ℓ	ϱ
3	—
2	—
1	—
0	$0{:}1, \mu_{0:1}, \delta^2_{0:1}$

x_1

ℓ	ϱ
3	—
2	—
1	$0{:}2, \mu_{0:2}, \delta^2_{0:2}$
0	—

x_2

ℓ	ϱ
3	—
2	—
1	$0{:}2, \mu_{0:2}, \delta^2_{0:2}$
0	$2{:}3, \mu_{1:2}, \delta^2_{1:2}$

x_3

ℓ	ϱ
3	—
2	$0{:}4, \mu_{0:4}, \delta^2_{0:1}$
1	—
0	—

x_4

ℓ	ϱ
3	—
2	$0{:}4, \mu_{0:4}, \delta^2_{0:1}$
1	—
0	$4{:}5, \mu_{4:5}, \delta^2_{4:5}$

x_5

ℓ	ϱ
3	—
2	$0{:}4, \mu_{0:4}, \delta^2_{0:1}$
1	$4{:}6, \mu_{4:6}, \delta^2_{4:6}$
0	—

x_6

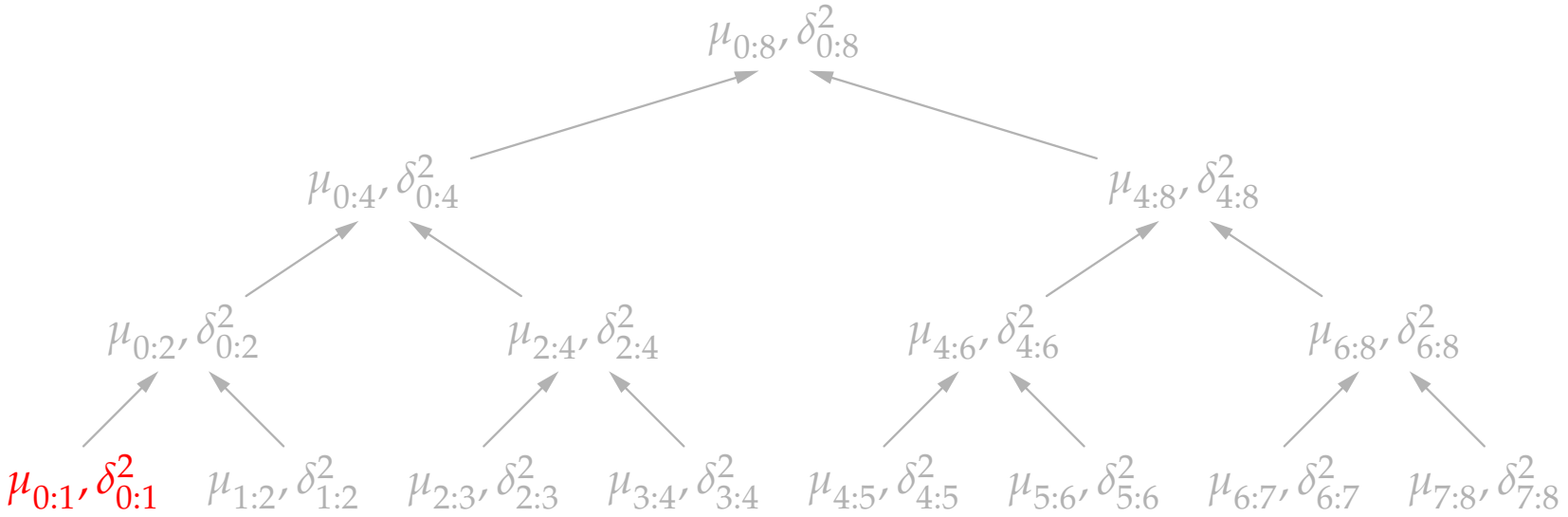
ℓ	ϱ
3	—
2	$0{:}4, \mu_{0:4}, \delta^2_{0:1}$
1	$4{:}6, \mu_{4:6}, \delta^2_{4:6}$
0	$6{:}7, \mu_{6:7}, \delta^2_{6:7}$

x_7

ℓ	ϱ
3	$0{:}8, \mu_{0:8}, \delta^2_{0:8}$
2	—
1	—
0	—

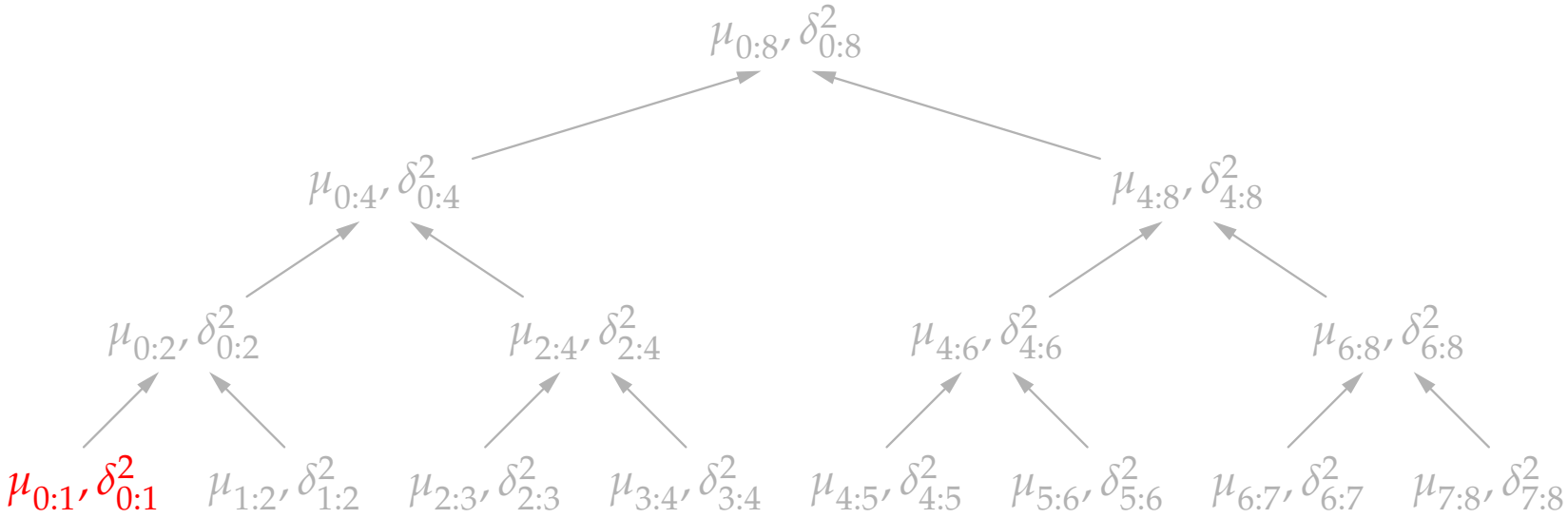
- Subproblem solutions are propagated up until an empty slot is reached.
- Occurring section ranges $i{:}k$ are pairwise disjoint.
- All occurring section ranges $i{:}k$ together cover the processed part of the stream.

Computing Characteristic Measures: Mean & Variance

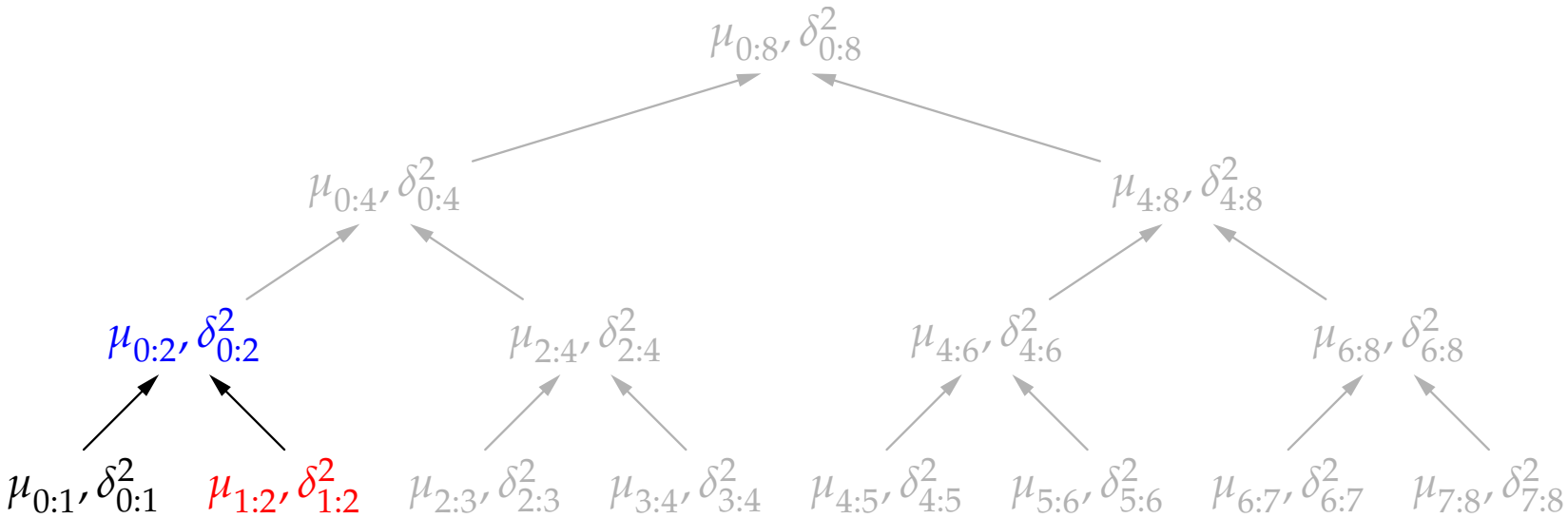


ℓ	ϱ
3	—
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$

Computing Characteristic Measures: Mean & Variance

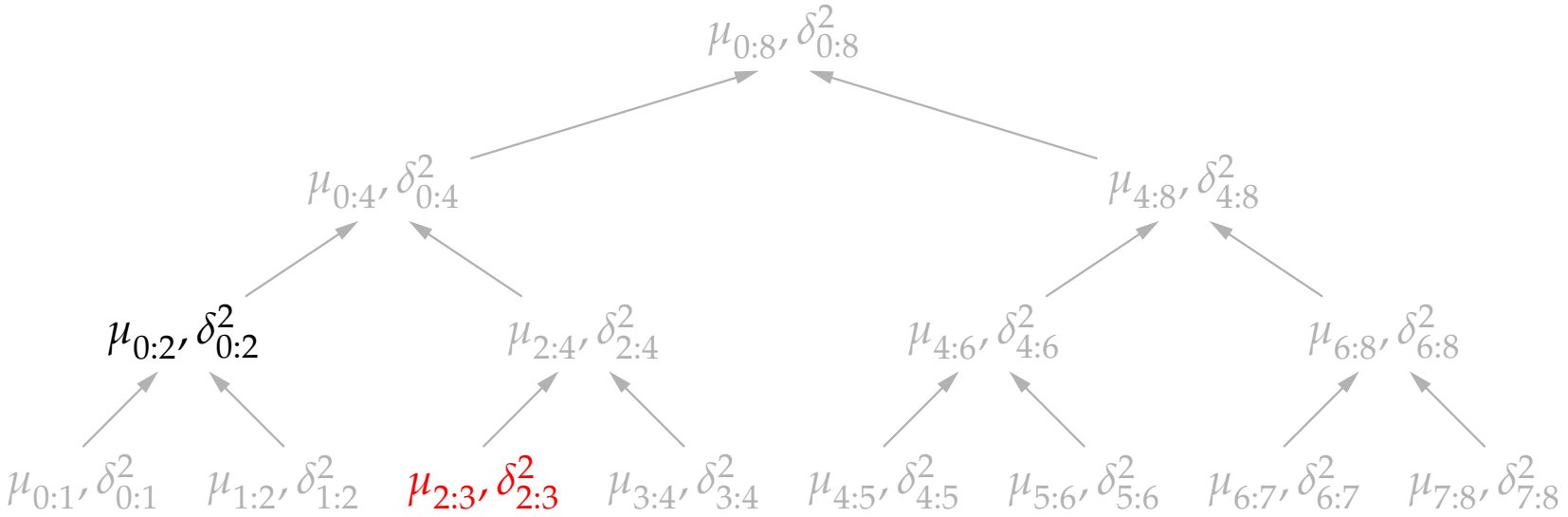


ℓ	ϱ
3	—
2	—
1	—
0	$0:1, \mu_{0:1}, \delta^2_{0:1}$



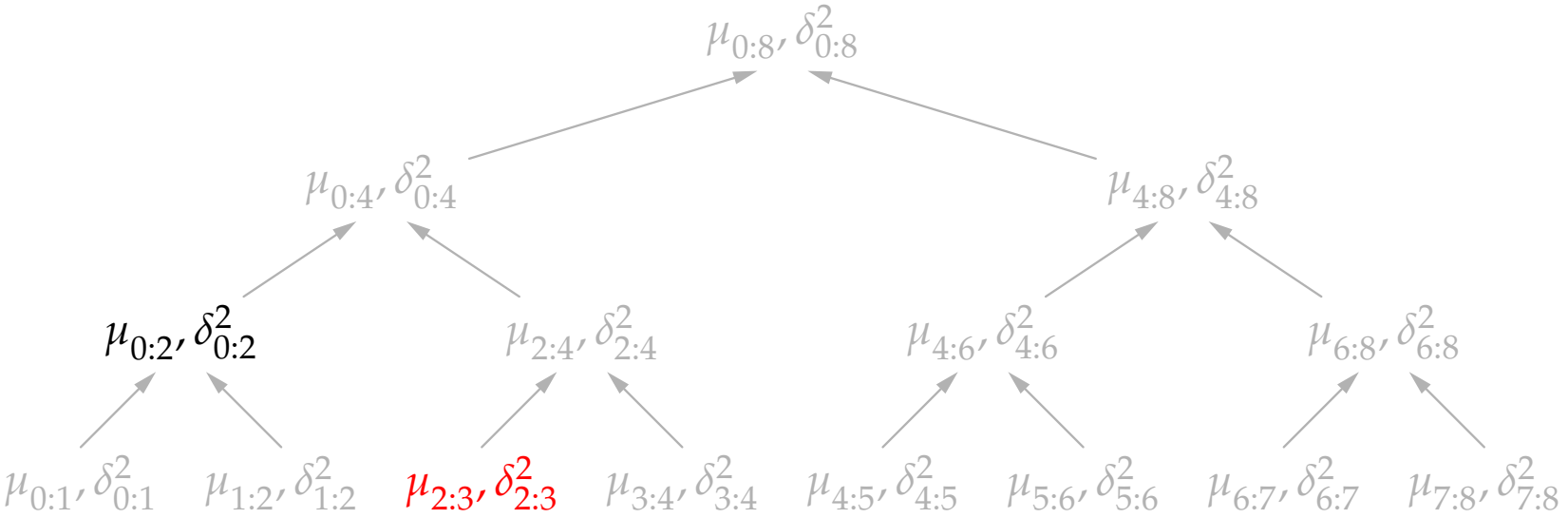
ℓ	ϱ
3	—
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	—

Computing Characteristic Measures: Mean & Variance

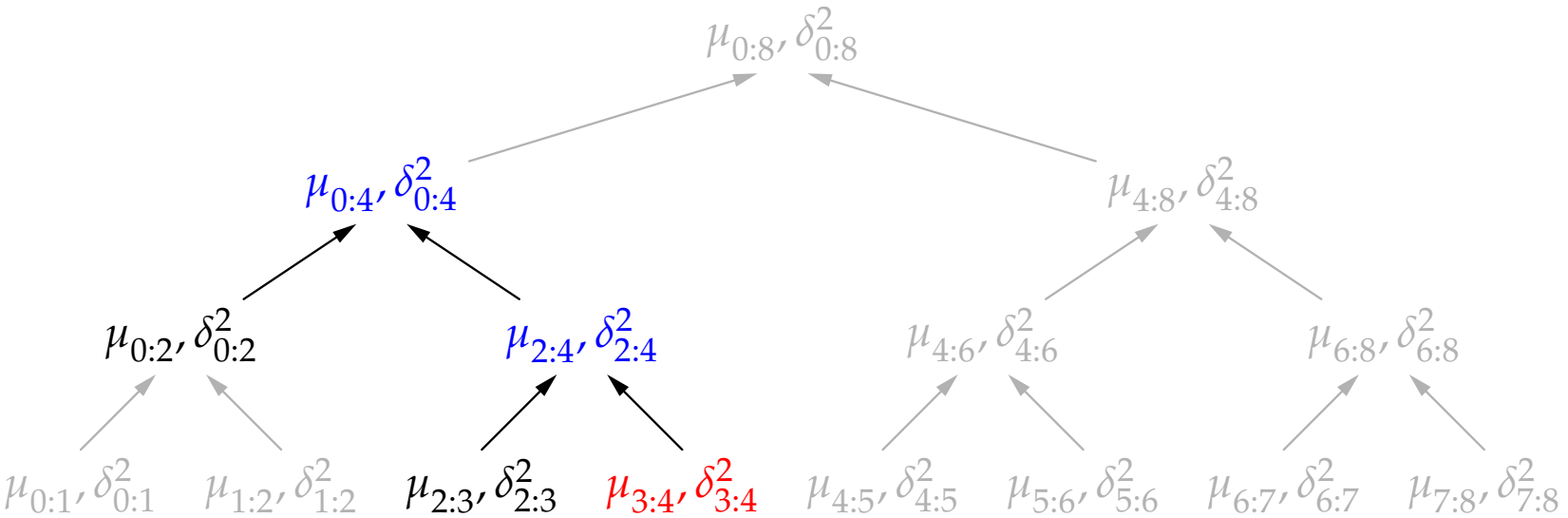


ℓ	ϱ
3	—
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{2:3}, \delta^2_{2:3}$

Computing Characteristic Measures: Mean & Variance

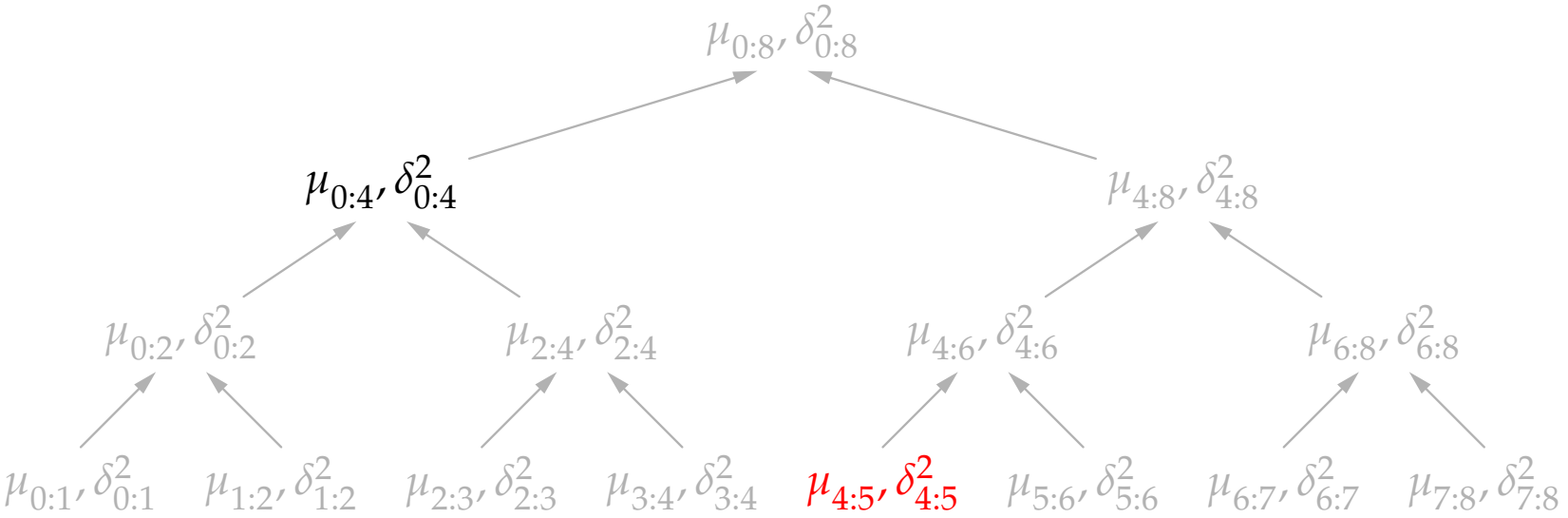


ℓ	ϱ
3	—
2	—
1	$0:2, \mu_{0:2}, \delta^2_{0:2}$
0	$2:3, \mu_{2:3}, \delta^2_{2:3}$



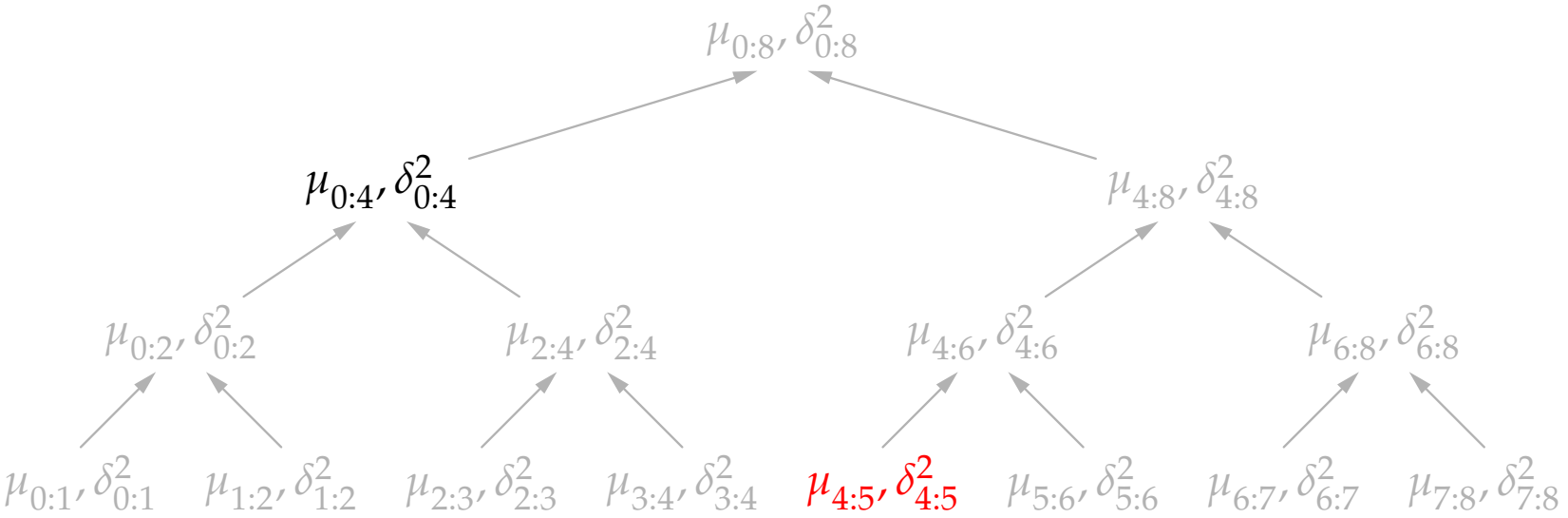
ℓ	ϱ
3	—
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	—

Computing Characteristic Measures: Mean & Variance

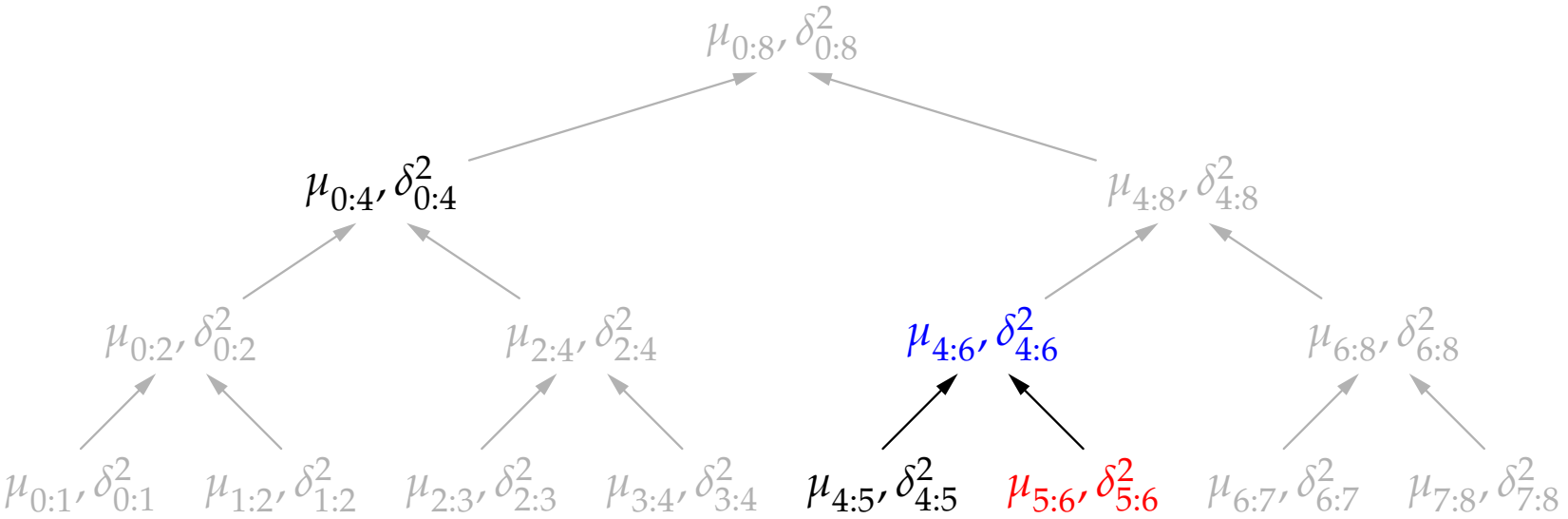


ℓ	ϱ
3	—
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	$4:5, \mu_{4:5}, \delta^2_{4:5}$

Computing Characteristic Measures: Mean & Variance

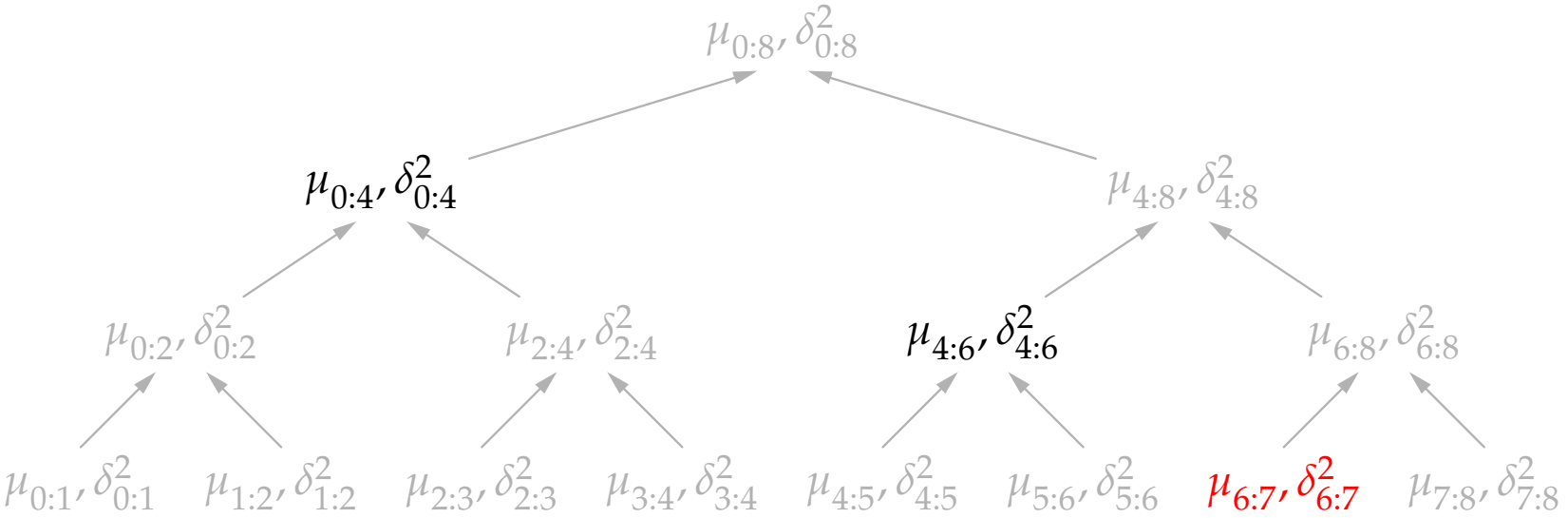


ℓ	ϱ
3	—
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	—
0	$4:5, \mu_{4:5}, \delta^2_{4:5}$



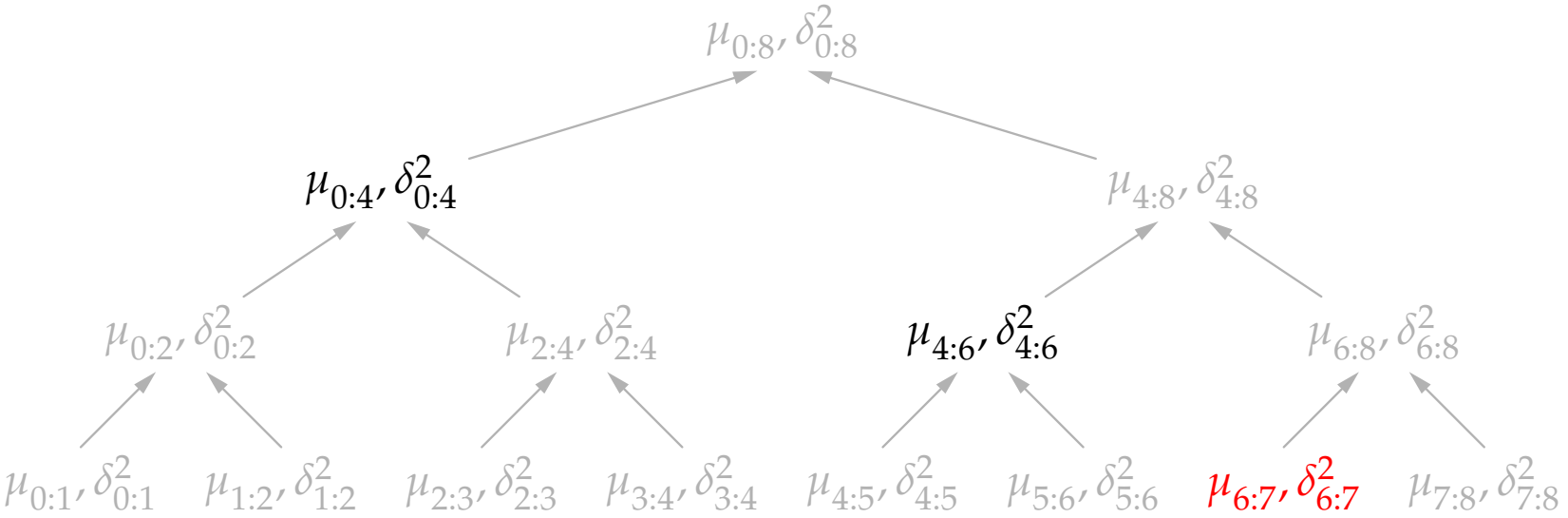
ℓ	ϱ
3	—
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	$4:6, \mu_{4:6}, \delta^2_{4:6}$
0	—

Computing Characteristic Measures: Mean & Variance

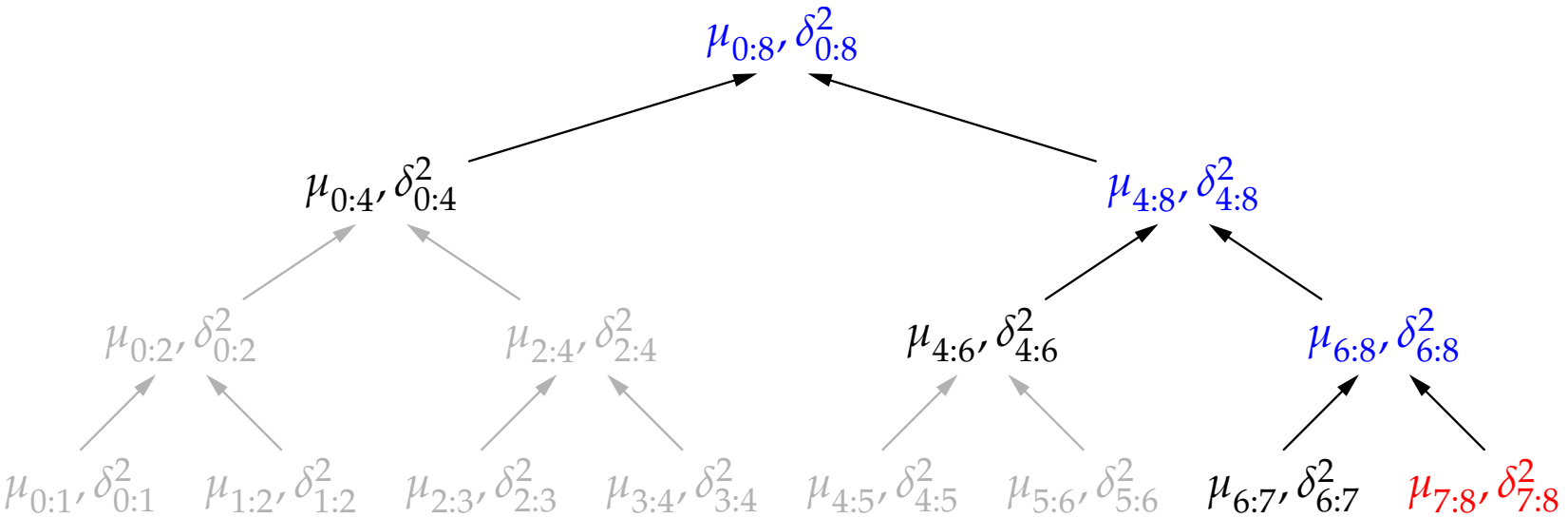


ℓ	ϱ
3	—
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	$4:6, \mu_{4:6}, \delta^2_{4:6}$
0	$6:7, \mu_{6:7}, \delta^2_{6:7}$

Computing Characteristic Measures: Mean & Variance



ℓ	ϱ
3	—
2	$0:4, \mu_{0:4}, \delta^2_{0:4}$
1	$4:6, \mu_{4:6}, \delta^2_{4:6}$
0	$6:7, \mu_{6:7}, \delta^2_{6:7}$



ℓ	ϱ
3	$0:8, \mu_{0:8}, \delta^2_{0:8}$
2	—
1	—
0	—

Computing Characteristic Measures: Mean & Variance

- The number of slots (empty and occupied) needed for a stream of size n is $\log_2 n$.

  $\Rightarrow \quad M = \lceil \log_2 n \rceil$ (**logarithmic space/memory** as desired).

- From the (partial) solutions in the occupied slots,
  mean and variance can be computed with a number of operations
  that is linear in the number of occupied slots.

  $\Rightarrow$ Query time is logarithmic in the size of the (already processed) stream.

- Total time for filling the slots for a stream is linear in the size of the stream,
  even though for some elements more than one operation is needed.

  $\Rightarrow$ So-called **"amortized" linear time**.

  The number of operations per step is given by the so-called **ruler function** a:
  $2^{a(n)}$ divides $2n$ (analogous to the tics on a ruler, hence the name).

  If you are counting in binary and the least significant bit is numbered 1, the next bit is 2 etc., $a(n)$ is the bit that is incremented when increasing from $n-1$ to n.

  The sum of the first n elements of the ruler function grows linearly in n.

Computing Characteristic Measures: Mean & Variance

- **Main Problems:**
  - Large sums, potential overflow of floating point numbers.
  - Potential loss of precision when adding large and small numbers.
- **Solution Possibilities:**
  - Try to keep the sums as small as possible:
    Improved online approaches to computing mean and variance.
  - Correct for loss of accuracy by clever summation:
    Kahan Summation Algorithm.
  - Recursive computation of mean and variance:
    Tends to sum numbers of roughly equals size.
- **Alternative: Two Passes** (rather to be avoided)
  - Minimizes sums for variance computation, but costly.

Frequency Estimation and Finding Frequent Items

Frequency Estimation

Problem Formalization

- Given: stream X of n elements x_i from some universe $\mathcal{U}$
  (e.g. internet traffic, web site accesses etc.).
- Elements can appear arbitrarily often, but some of them may not appear.
- Let $f(x)$ denote the frequency of an element $x \in \mathcal{U}$.
- Estimate $\hat{f}(x)$ with $f(x) - \epsilon \cdot n \leq \hat{f}(x) \leq f(x) + \epsilon \cdot n$ for each $x \in \mathcal{U}$.

Special Case: Majority Problem

- Given: stream X of n elements x_i from some universe $\mathcal{U}$
  (e.g. internet traffic, web site accesses etc.).
- Elements can appear arbitrarily often, but some of them may not appear.
- Let $f(x)$ denote the frequency of an element $x \in \mathcal{U}$.
- Question: Is there an item $x \in \mathcal{U}$ with $f(x) > \frac{n}{2}$ (majority item)?

Frequency Estimation: Majority Problem

Boyer–Moore Algorithm [Robert S. Boyer & J. Strother Moore 1981]

- Keep track of only one candidate c by storing its occurrence count o.
  Start with the first element on the stream.
- If the current element on the stream is the candidate, increment the counter o.
- If the current element on the stream is not this candidate,
  then decrement the occurrence counter for the candidate.

  Delete the candidate if its occurrence counter reaches zero.
- Make the next element on the stream the new candidate.
  (Note: current element is simply ignored / skipped.)

Example

element x	9	3	9	4	9	6	9	9	7
candidate c	9	–	9	–	9	–	9	9	9
counter o	1	0	1	0	1	0	1	2	1

Result
candidate: $c = 9$
counter: $o = 1$

Frequency Estimation: Majority Problem

Boyer–Moore Algorithm [Robert S. Boyer & J. Strother Moore 1981]

```
c ← ∅; o ← 0                    # initialize candidate and occurrence counter
for x in X:                     # traverse the stream elementwise
    if x = c then               # if the current element is the candidate,
        o ← o + 1               # count the occurrence of the candidate
    else if c = ∅ then          # if currently there is no candidate,
        c ← x                   # make the current element the candidate
        o ← 1                   # and initialize the occurrence counter
    else                        # if current element is not the candidate,
        o ← o − 1               # decrement the occurence counter
        if o = 0 then c ← ∅     # if counter reaches 0, delete the candidate
```

Example

element x	5	3	5	4	5	6	9	9	7
candidate c	5	–	5	–	5	–	9	9	9
counter o	1	0	1	0	1	0	1	2	1

Result
candidate: $c = 9$
counter: $o = 1$

Frequency Estimation: Majority Problem

Boyer–Moore Algorithm [Robert S. Boyer & J. Strother Moore 1981]

- If there is a majority element, it is the candidate, because for such an element the counter is incremented more often than it is decremented.

  Attention: If an element occurs as exactly half of the stream elements ($f(x) = \frac{n}{2}$), it may not be the candidate (example: 2 1 3 1 4 1 5 1 6 1).

- However, the candidate need not be a majority element (see second example).

- A second pass is needed to get the actual occurrence count.

```
o ← 0                          #  initialize the counter
for x in X:                    #  traverse the stream elementwise
    if x = c then              #  if the current element is the candidate,
        o ← o + 1              #  count the occurrence of the candidate
```

- Side Remark: The following is a valid Python statement.

```
o += x == c
```

What does this do? Why does it work?

General Frequency Estimation Problem

Problem Formalization

- Given: stream X of n elements x_i from some universe $\mathcal{U}$
  (e.g. internet traffic, web site accesses etc.).
- Elements can appear arbitrarily often, but some of them may not appear.
- Let $f(x)$ denote the frequency of an element $x \in \mathcal{U}$.
- Estimate $\hat{f}(x)$ with $f(x) - \epsilon \cdot n \leq \hat{f}(x) \leq f(x) + \epsilon \cdot n$ for each $x \in U$.

Approach with Boyer–Moore Algorithm for the Majority Problem

- Find a majority element c (if one exists) with frequency o.
- If a majority element c exists, set $\hat{f}(c) = o$ and $\hat{f}(x) = \frac{n-o}{2}$ for all $x \neq c$.

  If no majority element exists, set $\hat{f}(x) = \frac{n}{4}$ for all x.
- This "solves" the frequency estimation problem with $\epsilon = \frac{n-o}{2n}$ or $\epsilon = \frac{1}{4}$, resp.
  (However, this is not really useful due to the large value of ϵ.)

Finding Frequent Items

Problem Formalization

- Given: stream X of n elements x_i from some universe $\mathcal{U}$
  (e.g. internet traffic, web site accesses etc.).
- Elements can appear arbitrarily often, but some of them may not appear.
- Let $f(x)$ denote the frequency of an element $x \in \mathcal{U}$.
- Question: Are there items $x \in \mathcal{U}$ with $f(x) > {}^{n}/_{k}$? ($k = 2$: majority problem)

Misra–Gries Algorithm [Jayadev Misra & David Gries 1982]

- Generalize the solution to the majority problem to k elements.
- Keep track of the elements with the highest frequency
  in a dictionary with size $k - 1$.
- If the stream proceeds without an element being counted / added,
  decrement the counters of the stored elements.

Finding Frequent Items

Misra–Gries Algorithm [Jayadev Misra & David Gries 1982]

```
c ← ∅                                  # initialize a dictionary of candidates
for x in X:                            # traverse the stream (1st pass)
    if x ∈ c then                      # if the current element is in the dictionary,
        c[x] ← c[x] + 1                # count the occurrence of the candidate
    else if |c| < k − 1 then           # if less than k − 1 candidates are counted,
        c[x] ← 1                       # initialize an occurrence counter
    else                               # if current element is not in dictionary,
        for y in c:                    # traverse candidates in dictionary
            c[y] ← c[y] − 1            # decrement each occurence counter
            if c[y] = 0 then del c[y]  # if counter reaches 0, delete the candidate

for y in c:                            # traverse the collected candidates and
    c[y] ← 0                           # (re)initialize their counters
for x in X:                            # traverse the stream again (2nd pass)
    if x ∈ c:                          # if an element is among the candidates,
        c[x] ← c[x] + 1                # increment its counter
```

Finding Frequent Items

Example ($n = 12, k = 4 \Rightarrow \frac{n}{k} = 3$)

stream	9	3	9	4	9	6	9	9	7	9	4	9
candidates	9:1	9:1	9:2	9:2	9:3	9:2	9:3	9:4	9:4	9:5	9:5	9:6
	–	3:1	3:1	3:1	3:1	–	–	–	7:1	7:1	7:1	7:1
	–	–	–	4:1	4:1	–	–	–	–	–	4:1	4:1

Final set of frequent items: $\{9\}$ (7 and 4 occur less than $\frac{n}{k} + 1 = 4$ times)

Example ($n = 12, k = 4 \Rightarrow \frac{n}{k} = 3$)

stream	9	3	2	4	9	6	8	3	7	9	4	2
candidates	9:1	9:1	9:1	–	9:1	9:1	9:1	–	7:1	7:1	7:1	–
	–	3:1	3:1	–	–	6:1	6:1	–	–	9:1	9:1	–
	–	–	2:1	–	–	–	8:1	–	–	–	4:1	–

Final set of frequent items: $\emptyset$ (no candidates at the end)

Finding Frequent Items

Correctness of the Misra–Gries Algorithm

- If an element occurred more often (in the processed part of the stream) than all of those not being counted together, its counter is greater than 0.
- If an item occurs more than $\frac{n}{k}$ times, its counter must be positive in the end.
- A second pass is needed to count the actual occurrences of the candidates.

Analysis of the Misra–Gries Algorithm

- Increment the counter of an element if it is in the dictionary (e.g. $O(\log k)$).
- Decrement all counters if an element is not counted currently ($O(k)$).
- 2 passes (determine candidates, determine final counts).
- Total runtime depends on dictionary implementation (e.g. $O(n \log k)$).
- Requires $O(k)$ space for the candidate dictionary.

General Frequency Estimation Problem

Approach with Misra–Gries Algorithm

- After the second pass, the exact frequencies of the candidates are known.
- Set $\hat{f}(x) = c[x] = f(x)$ for $x \in c$ and $\hat{f}(x) = \frac{n}{2k}$ for $x \notin c$.
  ($f(x)$: true frequency, $\hat{f}(x)$: estimated frequency)
- This solves the frequency estimation problem with $\epsilon = \frac{1}{2k}$.
  (Estimation is exact for $x \in c$, while for $x \notin c$ we know $0 \leq f(x) \leq \frac{n}{k}$.)
- Better than Boyer–Moore algorithm (provided $k > 2$), but still not great...

Approach with Misra–Gries Algorithm without 2^{nd} pass

- After the first pass, we have $f(x) - \frac{n}{k} \leq c[x] \leq f(x)$ for $x \in c$.
- Set $\hat{f}(x) = c[x] + \frac{n}{2k}$ for $x \in c$ and $\hat{f}(x) = \frac{n}{2k}$ for $x \notin c$.
- This solves the frequency estimation problem with $\epsilon = \frac{1}{2k}$.
- Worse for $x \in c$ (no longer exact), but single pass frequency estimation.

Finding Frequent Items

Space Saving Algorithm [A. Metwally, D. Agrawal & A. El Abbadi 2005]

$c \leftarrow \emptyset$ # initialize a dictionary of candidates
for x **in** X: # traverse the stream (1^{st} pass)
    if $x \in d$ **then** # if the current element is in the dictionary,
        $c[x] \leftarrow c[x] + 1$ # count the occurrence of the candidate
    else if $|c| < k$ **then** # if less than k candidates are counted,
        $c[x] \leftarrow 1$ # initialize an occurrence counter
    else # if current element is not in dictionary,
        $y_{\min} \leftarrow \text{argmin}_{y \in c}\, c[y]$ # find element/item with minimal count
        $v \leftarrow c[y_{\min}]$ # and retrieve this count
        del $c[y_{\min}]$ # delete element with minimal count
        $c[x] \leftarrow v + 1$ # start counting the current element

Purpose:

- Idea: Get better final counts after the first pass.
- Items more frequent than $\frac{n}{k}$ are undercounted less (actually not at all).

Finding Frequent Items

Example ($n = 12, k = 4 \Rightarrow \frac{n}{k} = 3$)

stream	9	3	9	4	9	6	9	9	7	9	4	9
candidates	9:1	9:1	9:2	9:2	9:3	9:3	9:4	9:5	9:5	9:6	9:6	9:7
	–	3:1	3:1	3:1	3:1	3:1	3:1	3:1	7:2	7:2	7:2	7:2
	–	–	–	4:1	4:1	4:1	4:1	4:1	3:1	3:1	4:2	4:2
	–	–	–	–	–	6:1	6:1	6:1	4:1	4:1	3:1	3:1

Final set of frequent items: $\{9\}$ (3, 4, and 7 occur less than $\frac{n}{k} + 1 = 4$ times)

Example ($n = 12, k = 4 \Rightarrow \frac{n}{k} = 3$)

stream	9	3	2	4	9	6	8	3	7	9	4	2
candidates	9:1	9:1	9:1	9:1	9:2	9:2	9:2	9:2	7:3	7:3	7:3	7:3
	–	3:1	3:1	3:1	3:1	6:2	6:2	6:2	9:2	9:3	9:3	9:3
	–	–	2:1	2:1	2:1	3:1	8:2	8:2	6:2	6:2	4:3	4:3
	–	–	–	4:1	4:1	2:1	3:1	3:2	8:2	8:2	6:2	2:3

Final frequency estimates: $\emptyset$ (all candidates occur less than $\frac{n}{k} + 1 = 4$ times)

General Frequency Estimation Problem

Correctness of the Space Saving Algorithm

- It is always $\sum_{x \in c} c[x] = n$, since each element increments one counter.
- For a full dictionary it is $v = \min_x c[x] \leq \lfloor \frac{n}{k} \rfloor$, since the k counters sum up to n. (If the dictionary is not full at the end of the stream, all counts are exact.)
- For $x \in c$ it is $f(x) \leq c[x] \leq f(x) + v - 1$, so set $\hat{f}(x) = c[x] - \frac{v}{2}$.
- For $x \notin c$ it is $0 \leq f(x) \leq v$, so set $\hat{f}(x) = \frac{v}{2}$.
- This solves the frequency estimation problem with $\epsilon = \frac{v}{2n} \leq \frac{1}{2n}\frac{n}{k} = \frac{1}{2k}$.

Analysis of the Space Saving Algorithm

- Also needs a second pass for the Frequent Items Problem.
- Total runtime depends on dictionary implementation (e.g. $O(n \log k)$).
- Requires $O(k)$ space for the candidate dictionary.

General Frequency Estimation Problem

Example ($n = 12, k = 4 \Rightarrow \frac{n}{k} = 3$)

stream	9	3	9	4	9	6	9	9	7	9	4	9
candidates	9:1	9:1	9:2	9:2	9:3	9:3	9:4	9:5	9:5	9:6	9:6	9:7
	–	3:1	3:1	3:1	3:1	3:1	3:1	3:1	7:2	7:2	7:2	7:2
	–	–	–	4:1	4:1	4:1	4:1	4:1	3:1	3:1	4:2	4:2
	–	–	–	–	–	6:1	6:1	6:1	4:1	4:1	3:1	3:1

Final frequency estimates: 9 : 6.5, 7 : 1.5, 4 : 2.5, 3 : 0.5, rest: 0.5.

Example ($n = 12, k = 4 \Rightarrow \frac{n}{k} = 3$)

stream	9	3	2	4	9	6	8	3	7	9	4	2
candidates	9:1	9:1	9:1	9:1	9:2	9:2	9:2	9:2	7:3	7:3	7:3	7:3
	–	3:1	3:1	3:1	3:1	6:2	6:2	6:2	9:2	9:3	9:3	9:3
	–	–	2:1	2:1	2:1	3:1	8:2	8:2	6:2	6:2	4:3	4:3
	–	–	–	4:1	4:1	2:1	3:1	3:2	8:2	8:2	6:2	2:3

Final frequency estimates: 7 : 1.5, 9 : 1.5, 4 : 1.5, 2 : 1.5, rest: 1.5.

Frequency Estimation and Finding Frequent Items

- **Majority Problem: Boyer–Moore Algorithm**
  - Maintain one candidate and its counter.
  - Second pass needed for actual count.
- **Finding Frequent Items: Misra–Gries Algorithm**
  - Maintain $k - 1$ candidates and their counters.
  - Second pass needed for actual counts.
  - One pass suffices for (somewhat less good) frequency estimates.
- **Finding Frequent Items: Space Saving Algorithm**
  - Maintain k candidates and their counters.
  - Second pass needed for actual counts.
  - One pass suffices for (often fairly good) frequency estimates.
  - Yields better estimates (in one pass) especially for skewed data.

Counting Distinct Elements

Counting Distinct Elements

Problem Formalization

- Given: stream X of elements x from some universe $\mathcal{U}$
  (e.g. identifiers from a set of permissible identifiers)
- Elements can appear arbitrarily often, but some of them may not appear.
- Question: How many **different** elements did appear?

Application Example

- Count the number of visitors (IP-addresses) of a web site.

  (IP: internet protocol, specifies rules for internet data transmission;
  e.g. IPv4: `141.201.80.5` for `www.plus.ac.at`
  or IPv6: `2a00:1450:4001:811::2003` for `www.google.at`;
  visitors can be identified by the IP-address of their device)

- Storing all occurring IP-addresses is not an option.

  $\Rightarrow$ **Estimate number using hashing and probabilities.**

Hash Functions

Map Data of Arbitrary Size to Fixed-size Values

- Input: Object from a (large) universe $\mathcal{U}$.
- Output: Hash value of fixed length from a (small) set of target values $\mathcal{H}$.
- Values of $\mathcal{H}$ are often called *buckets* or *bins*.
- Two objects being mapped to the same bucket $\Rightarrow$ **collision**.

Examples

- $\mathcal{U} = \mathbb{N}, \mathcal{H} = \{0, \ldots, k-1\}$, function: $h(u) = u \bmod k$.
- $\mathcal{U} = \mathbb{N}, \mathcal{H} = \{0, \ldots, k-1\}$, function: $h(x) = (ax + b) \bmod k$.
  a and b can be varied to construct different functions.
- Hash function for strings used in Java:

```
h = 0
for c in s:  h = h * 31 +c
h = h % k
```

Counting Distinct Elements

Flajolet–Martin Algorithm [Philippe Flajolet & G. Nigel Martin 1984]

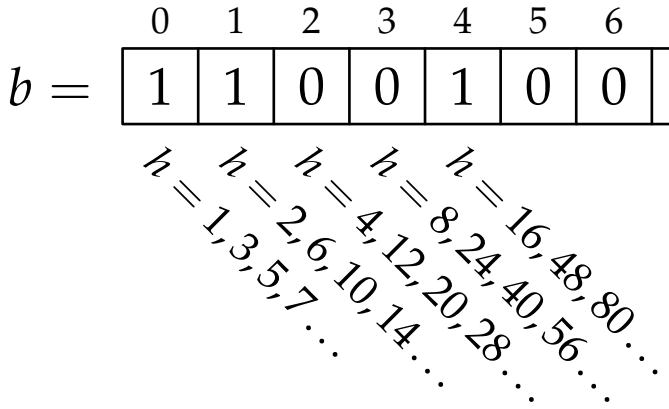
- Binary function $f : X \to \{0,1\}^K$ (map to bit sequences).
- The more **different** values you apply f to,
  the higher the probability to hit very **special** results.
- Turned around:
  By checking whether or not **special** results are hit in the output,
  one can estimate how many **different** input values there were.
- Use a hash function $h : X \to \{0,1\}^K$,
  e.g. a binary representation of hash values with K bits.
- Assume that the bits of $h(x)$ are uniformly and independently distributed.
  (As if each bit was determined by an independent coin flip.)
- Idea: Count number of t(r)ailing zeros in the bit representation.
  $t(h) = \max\{j \mid h \bmod 2^j = 0\}$ (tail length $0 \leq t(h) \leq K$)

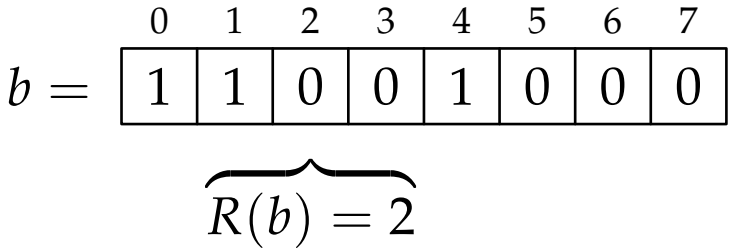
Counting Distinct Elements: Bit Masks & Tail Length

- Let r be the number of different elements (seen so far).
- Introduce a new random variable $b \in \{0,1\}^{K+1}$.
  K should be sufficiently larger than the expected $\log_2 r$,
  but may also be increased while the counting is in progress.
- Initialize $b(j) = 0$ for $j = 0, \ldots, K$ (zero indexed).
  For every x_i set $b(t(h(x_i)))$ to 1. (record observed tail length)
- The variable b keeps track of tail lengths seen so far
  and is used for approximate counting with probabilities.
- $b(0)$ is expected to be set to 1 by $1/2$ of the x_i,
  $b(1)$ is expected to be set to 1 by $1/4$ of the x_i,
  ...
  $b(k)$ is expected to be set to 1 by $1/2^{k+1}$ of the x_i,
- $b(k)$ is almost certainly 0 if $k \gg \log_2 r$ and
  $b(k)$ is almost certainly 1 if $k \ll \log_2 r$.

Counting Distinct Elements: Bit Masks & Tail Length

- Probability of $b(k) = 1$ depends on r $\quad\Rightarrow\quad$ use this for approximate counting.
- For the approximate counting, derive a new random variable R:
  $R(b) = \min\{i \mid b(i) = 0\}$ (smallest non-occurring tail length)
- It can be shown that $\mathbb{E}(R) \approx \log_2(\varphi r)$ where $\varphi = 0.77351\ldots$
- The number of distinct elements r can be approximated as $\hat{r} \approx \dfrac{2^R}{\varphi}$.





- $b[0] \leftarrow 1$ if $h(x) = 1(2j+1), j \in \mathbb{N}_0$,
  $b[1] \leftarrow 1$ if $h(x) = 2(2j+1), j \in \mathbb{N}_0$,
  $b[2] \leftarrow 1$ if $h(x) = 4(2j+1), j \in \mathbb{N}_0$,
  $b[3] \leftarrow 1$ if $h(x) = 8(2j+1), j \in \mathbb{N}_0$.

Counting Distinct Elements

Example ($n = 6, r = 5$): $X = [5, 7, 2, 5, 3, 6]$, $h(x_i) = x_i \bmod 8$ $\Rightarrow K = 3$.
($n = 6$ elements in stream, $r = 5$ distinct elements — element $x = 5$ occurs twice)

x_i	$h(x_i)$	tail length	bitmask b before	and after
5	101	0	0 0 0 0	1 0 0 0
7	111	0	1 0 0 0	1 0 0 0
2	010	1	1 0 0 0	1 1 0 0
5	101	0	1 1 0 0	1 1 0 0
3	011	0	1 1 0 0	1 1 0 0
6	110	1	1 1 0 0	1 1 0 0

Result: $R(b) = 2 \quad \Rightarrow \quad \hat{r} \approx 5.17$

Problems / Refinements

- R is not very precise. (It has a large standard deviation $\sigma(R) \approx 1.12$.)
- Precision could be increased by averaging multiple functions.

Flajolet–Martin Algorithm: Accuracy

- Improve accuracy with k different hash function:
  - Use one bitmask per hash function, that is, bitmasks $b_0, \ldots, b_{k-1}$).
  - Compute $R^* = \frac{1}{k} \sum_{j=1}^{k} R(b_j)$.
  - R^* has standard deviation $\sigma(R^*) \approx \frac{1.12}{\sqrt{k}}$.
  - However: Computing many $h_j(x_i)$ is costly.
- Improve accuracy with only one hash function:
  - Use k different bitmaps $b_0, \ldots, b_{k-1}$.
  - For each x_i update only bitmask: $b_{h(x_i) \bmod k}[t(h(x_i) \operatorname{div} k)] \leftarrow 1$.
  - Compute $R^* = \frac{1}{k} \sum_{j=1}^{k} R(b_j)$.
  - R^* has standard deviation $\sigma(R^*) \approx \frac{1.12}{\sqrt{k}}$.

Counting Distinct Elements

Flajolet–Martin Algorithm [Philippe Flajolet & G. Nigel Martin 1984]

$b \leftarrow [[0,\ldots,0],\ldots,[0,\ldots,0]]$ # initialize bit masks (k bit masks of length $K+1$)
for x **in** X # traverse the stream elements
    $z \leftarrow h(x)$ # compute hash value of current element
    $b[z \bmod k][t(z \operatorname{div} k)] \leftarrow 1$ # select a bit mask and set tail length bit
$R^* = 0$ # initialize final (averaged) estimate
for j **in** $[0,\ldots,k-1]$: # traverse the k bit masks
    $R^* \leftarrow R^* + R(b[j])$ # sum estimates from the different bit masks
$R^* \leftarrow \frac{1}{k} R^*$ # compute average result from all bitmasks
$\hat{r} \leftarrow 2^{R^*} / \ 0.77351$ # estimate number of distinct elements

Analysis of the Flajolet–Martin Algorithm

- Runtime $O(n + k \log r)$ (traverse stream + evaluate bit masks)
- Space / memory $O(k \log r)$ (k bit masks of size $K = O(\log r)$)

Flajolet–Martin Algorithm: Implementation Aspects

- As shown on the preceding slide, one may work with a single hash function, but multiple bit masks, only one of which is updated for each element.
- However, unless the stream is sufficiently long, the estimate may be worse than with a single bit mask (see exercise).
- Another alternative to using k hash functions:
  - Generate k random integers $r_0, \ldots, r_{k-1}$.
  - Compute k hash values as $z_j = h(x) + r_j$, $j = 0, \ldots, k-1$.
  - Update k bit masks, each with one of the k hash values.

  However, k bit masks increase the time complexity of the updates to $O(nk)$.
- Bit masks may be implemented as simple integer numbers. In Python, the q-th bit of a bit mask b (as an integer) can be set and tested by `b |= 1 << q` and `b & (1 << q)`, respectively. Why does this work?

Counting Distinct Elements

Flajolet–Martin Algorithm: Summary

- Apply a hash function to each element of a stream.
- The hash function should yield values with a binary representation, in which bits are set uniformly and independently.
- Record whether hash values with certain special properties occur.

  Here: Hash values with a certain number of trailing zeros.
- The more stream elements are processed, the higher the chance of seeing large numbers of trailing zeros.
- Use the smallest number of trailing zeros that did *not* (yet) occur to estimate the number of elements processed so far.
- The Flajolet–Martin Algorithm was improved in several ways, until an algorithm was found that uses nearly optimal space $O(\epsilon^{-2} + \log(m))$ and optimal time $O(n)$. [Daniel M. Kane, Jelani Nelson & David P. Woodruff 2010]

Sampling from a Stream

Sampling from a Stream

What is a Sample?

- Representative subset of the original data.
- Sample distributions should approximate true distributions of full stream.

Problems to be Solved:

- Selection of samples: How can it be done (effectively/efficiently)?
- Sampling may introduce a bias into the estimation.
- Sampling over time (maintain sample for extending stream).

Why is Sampling Potentially Problematic?

- Incoming data: E.g. database system serving clients (user, query, time).
- Queries appear multiple times
- Objective: Estimate number of queries that appear 1, 2, 3, . . . times.

Why is Sampling Potentially Problematic?

Sampling: Naive Approach that does Not Work.

- Randomly sample a certain percentage (e.g. 10%) of the data points.
- Estimate repetitions / multiplicities from resulting sample.

Special Case: Unique and Double Database Queries

- Unique query A and double query B, i.e., B appears as B_1 and B_2.
- A random sample $\mathcal{S}$ of 10% of the database query stream is drawn.
- $P(X \in \mathcal{S}) = 0.1$ for $X \in \{A, B_1, B_2\}$
- $P(B_1 \in \mathcal{S} \wedge B_2 \in \mathcal{S}) = 0.1 \cdot 0.1 = 0.01$ (assuming independence)
  $P(B_1 \in \mathcal{S} \wedge B_2 \notin \mathcal{S}) = 0.1 \cdot (1 - 0.1) = 0.09$
  $P(B_1 \notin \mathcal{S} \wedge B_2 \in \mathcal{S}) = (1 - 0.1) \cdot 0.1 = 0.09$
  $P(B_1 \notin \mathcal{S} \wedge B_2 \notin \mathcal{S}) = (1 - 0.1) \cdot (1 - 0.1) = 0.81$
- B appears to be a unique query with odds 18:1.

Sampling Using Hash Functions

Solution: Frequency-Independent Sampling

- Sample items independent of their frequency
- If an item is sampled, retain all of its copies.
- Use a hash function to check whether an item has already been sampled.

  Important assumption:
  Items are approximately equally distributed to hash values / buckets.

Algorithm: Sample $(j/k) \cdot n$ Items.

- Hash function $h : X \to \{0, 1, \ldots, k-1\}$ (k hash values/buckets)
- if $h(x_i) < j$ add x_i to sample. (j/k: fraction of hash buckets sampled from)
- Works if n is known in advance and j/k is small enough.
- Can be adapted to arbitrary n and fixed sample size.

Sampling: Concrete Example

Task: Extract a 2GB Sample from a Stream of Incoming IDs

- Input objects with `id` as string $s = c_0 c_1 \dots c_m$ (i.e. word s with characters c_i).
  - $m \in \{3, \dots, 20\}$ (lengths of identifiers)
  - $c_i \in \{\text{a}, \dots \text{z}, \text{A}, \dots \text{Z}\} = \Omega$ (Ω: alphabet, $|\Omega| = 52$)
- Convert `id` to a number:
  - assign a value to each character: $v(\text{a}) = 0, v(\text{b}) = 1, \dots, v(\text{Z}) = 51$.
    (often simpler alternative: ASCII/UTF-8 values)
  - Compute $h^*(s) = \sum_{i=0}^{m} v(c_i) \cdot 52^i$.
  - Use Horner's method for the implementation:
    $v_m = v(c_m), v_{m-1} = v_m \cdot 52 + v(c_{m-1}), \dots, h^*(c_0 c_1 \dots c_m) = v_0 = v_1 \cdot 52 + v(c_0)$.
- Apply hash function $h(s) = h^*(s) \bmod 1000$.
- Sample objects with $h(s) < j$ starting with $j = 1000$.
- Whenever the memory usage exceeds 2GB, reduce j by 10.

Sampling Using Hash Functions

Problems and Potential Solutions

- Problem: Characters are often not equally likely (e.g. in language)
  - We ignore this problem here or assume that hashing takes care of it.
- Problem: What if $j = 1$ and sample size still exceeds 2GB?
  - Rerun and use more buckets (e.g. 10000).
    (Only possible in streaming, but not in online scenario.)
  - Use a second hashing stage (also applicable in online scenario):
    Apply another hash function to the selected objects
    (i.e., those that pass the first hash-based filter).
- If two passes are possible (streaming scenario):
  - Use Flajolet–Martin algorithm to estimate the number of unique objects.

    $\Rightarrow$ yields expected number u of unique items per hash bucket.
  - Choose an appropriate j (and k) based on u and s/u.

Maintaining Sets

Motivating Scenario

- A sample $S \subset X$ can be used to learn properties of the overall distribution
  - Example: queries and their frequencies (see above)
- In the following, we derive further statistics about the samples.
  - Example: frequency distributions
    (how many queries have certain frequencies)
- For each x_i we need to decide whether $x_i \in S$ (S: sample set to maintain).
- We could exploit the filters resulting from the sampling process.
- Problem: new (unseen) objects could end up in "sample buckets".

  (But we only want to maintain the sample and count occurrences;
  we do not want to extend the sample, but maintain the sample set.)
- Objective: decide set membership without further information.

Maintaining Sets

A First Approach

- Use fine-grained hashing to split samples into small chunks (of objects ending up in the same hash bucket).
- Remember the buckets that contain an $x_i \in S$.
- Discard all elements that are not assigned to these buckets.

Result: A "Dirty" Filter

- If $x_i \in S$, then x_i is certainly retained / counted.
- If $x_i \notin S$, then x_i **might** be discarded.

  However, depending on $h(x_i)$, it may also be retained/counted, even though it is not part of the set to be maintained.
- In the following, we improve this general idea using hash functions.

  $\Rightarrow$ **Bloom Filters** [Burton Howard Bloom 1970]

Bloom Filters: Idea and Introduction

- Input: A large set S of objects (set/sample to maintain).
- Objective: Decide whether $x_i \in S$ without storing S.

Formalization of the Idea (not a Bloom filter yet)

- Hash function $h : X \to \{0, 1, \ldots, b-1\}$ with b sufficiently large.
- Bit array B of length b with $B[i] = \begin{cases} 1 & \text{if } \exists s \in S : h(s) = i, \\ 0 & \text{otherwise.} \end{cases}$
- For each $x_i \in S$ we have $B[h(x_i)] = 1$ (true positive).

  Note: There cannot be any false negatives.
- For each $x_i \notin S$, we have
  $B[h(y_i)] = 1$ (false positive, number to be minimized) or
  $B[h(x_i)] = 0$ (true negative, reject x_i as desired).
- Elements of S certainly pass the filter; others might pass but hopefully only few.

Bloom Filters: Definition and Usage

Bloom Filter [Burton Howard Bloom 1970]

- Set S to be maintained and bit array B as before.
- A **collection** of hash functions $h_1, \ldots, h_k$ mapping to H.
- Set $B[i] = 1$ if for **any** $s \in S$ and **any** h_j it is $h_j(s) = i$. Or formally:

$$B[i] = \begin{cases} 1 & \text{if } \exists s \in S : \exists j \in \{1, \ldots, k\} : h_j(s) = i, \\ 0 & \text{otherwise.} \end{cases}$$

- Object o passes the filter if $\forall j \in \{1, \ldots, k\} : B[h_j(o)] = 1$.

- Spreading to more hash functions (larger k) avoids collisions.
  - It suffices if one of the h_j does not produce a collision with any $s \in S$.
- Allows derivation of probabilistic properties.
  - E.g., how many h_i and size of B are needed for a given S.

Bloom Filters: Properties

- Bit array size $|B| = b$ (memory usage in bits), e.g. $b = 10^6$.
- Number k of hash functions (time consumption of test), e.g. $k = 5$.
- $|S| = n$ elements stored ("yield" in a sense), e.g. $n = 10^5$.
- **Probability of a false positive** approximately $(1 - e^{-nk/b})^k$. (see next slides)
  - Example: $(1 - e^{-5 \cdot 10^5 / 10^6}) \approx 0.0094$ (0.94%)
  - Memory usage: 10^6 bits $\approx$ 122 kB (10 bits/id).
  - If an ID has 10 bytes storing all IDs requires 10^7 bytes $\approx$ 9.5MB +overhead.
- **Dependencies on/between parameters**
  - More memory and fewer elements n improve the false positive probability.
  - More hash functions not necessarily! (because more bits get set)
    E.g. $k = 4 : p_{\text{fp}} \approx 1.2\%$, $k = 7 : p_{\text{fp}} \approx 0.82\%$, $k = 10 : p_{\text{fp}} \approx 1\%$
  - Each b, n pair has an optimal $k = \frac{b}{n} \ln(2)$. (see next slides).

Probability of False Positives in Bloom Filters

- The probability of a certain bit in the Bloom filter's array B still being 0 after inserting one item with a single hash function is $1 - \frac{1}{b}$.
- Therefore, after inserting n items with k hash values each, the probability of a certain bit still being 0 is $p_0 = (1 - \frac{1}{b})^{nk}$ (assuming independence!).
- Hence, the probability of all bits being 1 for the k hash values when checking a new element (false positive probability) is

$$p_{\text{fp}} = \left(1 - \left(1 - \tfrac{1}{b}\right)^{nk}\right)^k.$$

- We exploit the well-known relationship (choosing $z = nk$ and $x = -\frac{nk}{b}$)

$$\lim_{z \to \infty} \left(1 + \tfrac{x}{z}\right)^z = e^x \qquad \Rightarrow \qquad \lim_{nk \to \infty} \left(1 - \tfrac{1}{b}\right)^{nk} = e^{-\frac{nk}{b}}$$

to obtain (because nk, though not ∞, is supposed to be very large)

$$p_{\text{fp}} = \left(1 - \left(1 - \tfrac{1}{b}\right)^{nk}\right)^k \approx \left(1 - e^{-\frac{nk}{b}}\right)^k.$$

Probability of False Positives in Bloom Filters

- To find a k that minimizes the false positive probability p_{fp} for given b and n, we exploit that a vanishing derivative is a necessary condition for a minimum and applying a monotone function does not change the location of a minimum.

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}k} \ln(p_{\text{fp}}) &= \frac{\mathrm{d}}{\mathrm{d}k} \ln\left(\left(1 - e^{-\frac{nk}{b}}\right)^k\right) = \frac{\mathrm{d}}{\mathrm{d}k} k \cdot \ln\left(1 - e^{-\frac{nk}{b}}\right) \\
&= \left(\frac{\mathrm{d}}{\mathrm{d}k} k\right) \cdot \ln\left(1 - e^{-\frac{nk}{b}}\right) + k \cdot \frac{\mathrm{d}}{\mathrm{d}k} \left(\ln\left(1 - e^{-\frac{nk}{b}}\right)\right) \\
&= \ln\left(1 - e^{-\frac{nk}{b}}\right) + k \cdot \frac{1}{1 - e^{-\frac{nk}{b}}} \cdot \frac{\mathrm{d}}{\mathrm{d}k} \left(1 - e^{-\frac{nk}{b}}\right) \\
&= \ln\left(1 - e^{-\frac{nk}{b}}\right) + k \cdot \frac{1}{1 - e^{-\frac{nk}{b}}} \cdot \left(-e^{-\frac{nk}{b}}\right) \frac{\mathrm{d}}{\mathrm{d}k} \left(-\frac{nk}{b}\right) \\
&= \ln\left(1 - e^{-\frac{nk}{b}}\right) + k \cdot \frac{1}{1 - e^{-\frac{nk}{b}}} \cdot \left(-e^{-\frac{nk}{b}}\right) \cdot \left(-\frac{n}{b}\right) \\
&= \ln\left(1 - e^{-\frac{nk}{b}}\right) + \frac{nk}{b} \cdot \frac{e^{-\frac{nk}{b}}}{1 - e^{-\frac{nk}{b}}},
\end{aligned}
$$

Probability of False Positives in Bloom Filters

- The derivative of the logarithm of the false positive probability p_{fp} w.r.t. k is

$$\frac{\mathrm{d}}{\mathrm{d}k}\ln(p_{\text{fp}}) \;=\; \frac{\mathrm{d}}{\mathrm{d}k}\ln\left(\left(1-e^{-\frac{nk}{b}}\right)^k\right) \;=\; \ln\left(1-e^{-\frac{nk}{b}}\right)+\frac{nk}{b}\cdot\frac{e^{-\frac{nk}{b}}}{1-e^{-\frac{nk}{b}}}.$$

- This derivative is 0 for $k=\frac{b}{n}\ln(2)$, because then we get

$$e^{-\frac{nk}{b}} \;=\; e^{-\frac{\frac{b}{n}\cdot\ln(2)\cdot n}{b}} \;=\; e^{-\ln(2)} \;=\; \frac{1}{e^{\ln(2)}} \;=\; \frac{1}{2}$$

  and therefore, if this is inserted (side remark: $k=\frac{b}{n}\ln(2) \Leftrightarrow \frac{nk}{b}=\ln(2)$)

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}k}\ln(p_{\text{fp}}) \;&=\; \ln\left(1-e^{-\frac{nk}{b}}\right)+\frac{nk}{b}\cdot\frac{e^{-\frac{nk}{b}}}{1-e^{-\frac{nk}{b}}}\\ &=\; \ln\left(1-\tfrac{1}{2}\right)+\ln(2)\cdot\frac{\frac{1}{2}}{1-\frac{1}{2}} \;=\; \ln\left(\tfrac{1}{2}\right)+\ln(2) \;=\; \ln(1) \;=\; 0.\end{aligned}$$

- Note that in this whole derivation it is assumed that the events (that is, bits being set) are independent, which is not entirely true. Nevertheless this derivation gives a useful approximation in practice.

Bloom Filters: Counting Distinct Elements

- A Bloom filter can be used to **count distinct elements**.

  (This is an alternative to the Flajolet–Martin Algorithm, but it needs more memory.)

- **Trivial Approach**
  - Add all stream elements to a (sufficiently large) Bloom filter.
  - Count how many elements set a new bit in the filter (flip a bit from 0 to 1), as this indicates a new, yet unseen element.
  - This approach necessarily underestimates the number of distinct elements.

- **Better Approach** [S. Joshua Swamidass & Pierre Baldi 2007]
  - The number of set bits clearly depends on the number of added items.
  - The number of distinct items in a Bloom filter can be estimated as

$$n^* \approx -\frac{b}{k} \ln(1 - \frac{b_1}{b}),$$

  where b_1 is the number of set bits in the bit array B. (see next slides)

Bloom Filter: Counting Distinct Elements

Estimating the Number of Elements in a Bloom Filter

- Approach: Derive an expression for the expected number of set bits in a Bloom filter after n items have been inserted with k hash values for each.
- The probability of a certain bit in the Bloom Filter's array B still being 0 after inserting one item with a single hash function is $1 - \frac{1}{b}$ (see above).
- Therefore, after inserting n items with k hash values each, the probability of a certain bit still being 0 is $p_0 = (1 - \frac{1}{b})^{nk}$ (assuming independence!).
- That is, assuming that bits are set uniformly distributed over the bit array, we can model their setting to 1 as coin flips for each bit array element, which set a bit in the array to 1 with probability $p_1 = (1 - (1 - \frac{1}{b})^{nk})$.
- Then the number of set bits is binomially distributed with parameters p_1 and b:

$$P_{\text{binom}}(A = a; p_1, b) \quad = \quad \binom{b}{a} \cdot (p_1)^a \cdot (1 - p_1)^{b-a}.$$

(Here A is a random variable describing the number of set bits in a Bloom filter.)

Bloom Filter: Number of Contained Elements

Estimating the Number of Elements in a Bloom Filter

- The expected value of a binomial distribution is $E_{\text{binom}}(X; p, n) = np$, so here

$$\mathbb{E}_{\text{binom}}(A; p_1, b) \;=\; b \cdot (1 - (1 - p_1)) \;=\; b \cdot (1 - (1 - \tfrac{1}{b})^{nk}).$$

- Assuming that the observed number b_1 of bits set to 1 is close to the expected value, we get (using the same approximation as before)

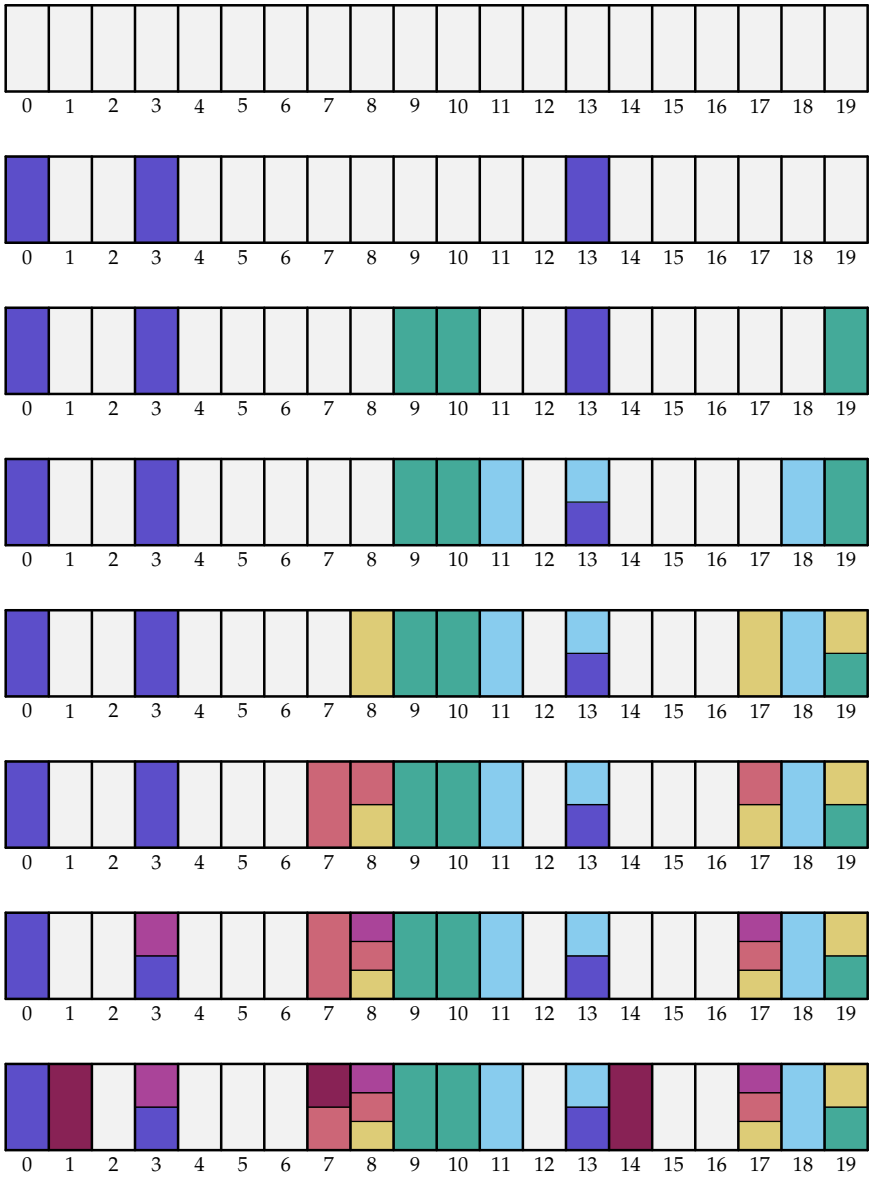
$$b_1 \;\approx\; b \cdot (1 - (1 - \tfrac{1}{b})^{nk}) \;\approx\; b \cdot (1 - e^{-\frac{nk}{b}}).$$

- Solving this equation for n, the number of added elements, we get

$$\begin{aligned} b_1 \approx b \cdot (1 - e^{-\frac{nk}{b}}) \quad &\Leftrightarrow \quad \tfrac{b_1}{b} \approx 1 - e^{-\frac{nk}{b}} \\ &\Leftrightarrow \quad e^{-\frac{nk}{b}} \approx 1 - \tfrac{b_1}{b} \\ &\Leftrightarrow \quad -\tfrac{nk}{b} \approx \ln(1 - \tfrac{b_1}{b}) \\ &\Leftrightarrow \quad n \approx -\tfrac{b}{k} \ln(1 - \tfrac{b_1}{b}). \end{aligned}$$

Bloom Filter: Counting Distinct Elements

word x_i	$h_1(x_i)$	$h_2(x_i)$	$h_3(x_i)$	$\hat{n}$
x_1: beauty	0	3	13	1.08
x_2: peace	10	19	9	2.38
x_3: love	18	11	13	3.41
x_4: trust	8	19	17	4.62
x_5: hope	8	17	7	5.32
x_6: talent false positive!	8	17	3	5.32
x_7: faith	14	1	7	7.00



Sampling from and Filtering a Stream: Summary

- Straightforward (random) sampling often does not work properly.
  - E.g. repetitions / multiplicities may be estimated wrongly.
- A correct sample may be obtained by fixing the selection.
  - E.g. limit the sample to certain items (users, ids, queries etc.)
  - Include all copies of an item in a sample.
- Decide with (a) hash function(s) whether an element is in a sample or not.

  $\Rightarrow$ Filter w.r.t. a large set of objects with a **Bloom Filter**.
  - Does this solve our "count duplicates" problem?

    $\Rightarrow$ Additional exercise explores this.
- A standard Bloom Filter does not allow to "delete" added elements.

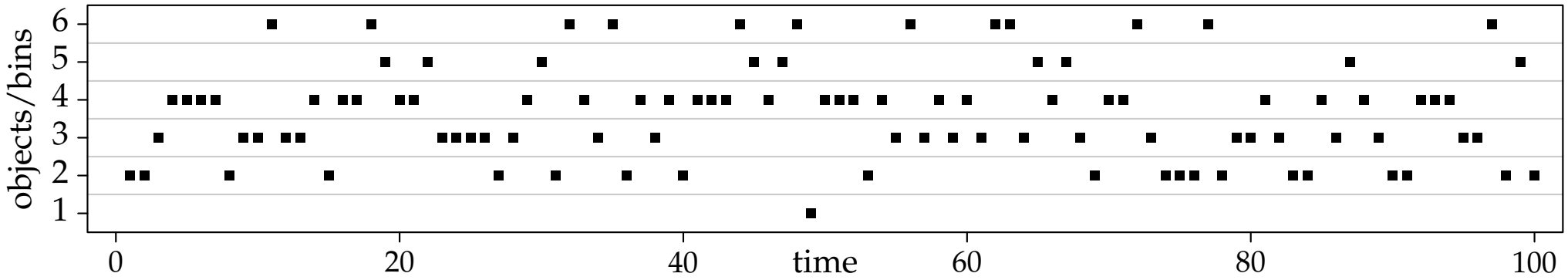
  So-called "counting" Bloom Filters and their variants allow for this possibility.

  (Essentially: Instead of mere bits use counters, which can then be decremented for deletion.)

Histograms and Frequency Distributions

Stream and Histogram

Consider the following input:

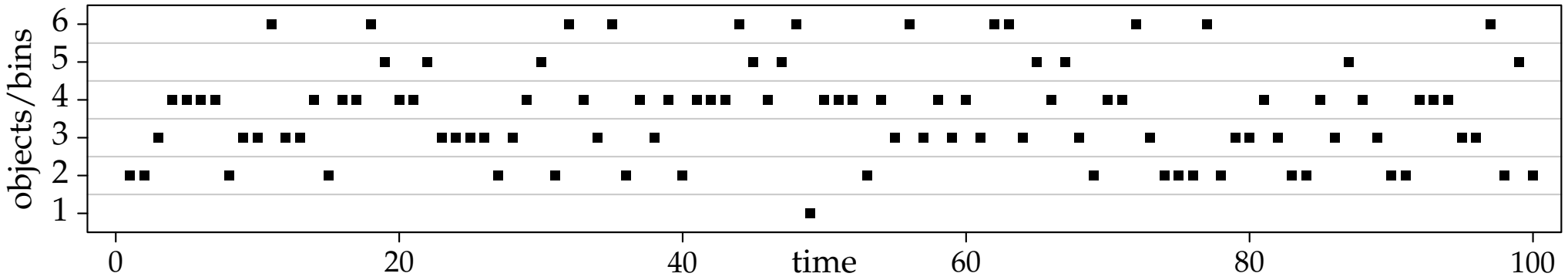


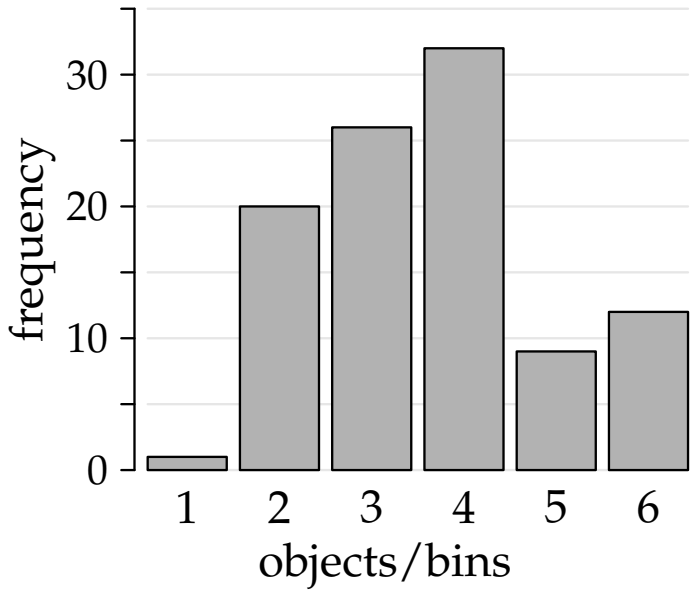
- As time (or the stream) progresses, data points come in.
  - E.g. users issue requests.
- Items are distinguished by some id or bucket/bin (from hashing).
- Some items are seen more often (e.g. 4), some less often (e.g. 1)
  - E.g. user 4 sends requests with high frequency, while user 1 sends only one request.

$\Rightarrow$ This is highly valuable information for an analysis.

Stream and Histogram

Consider the following input:

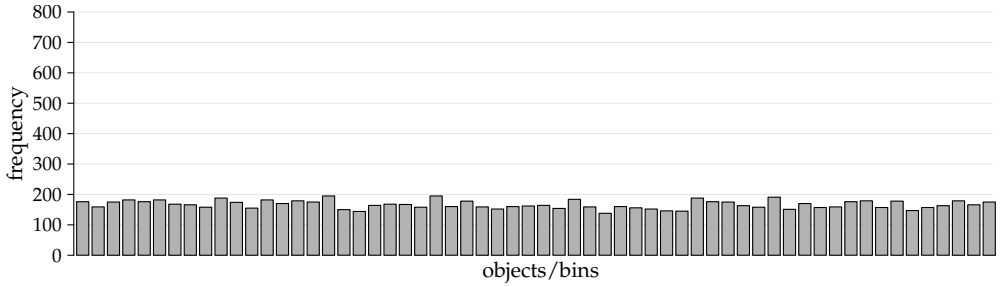


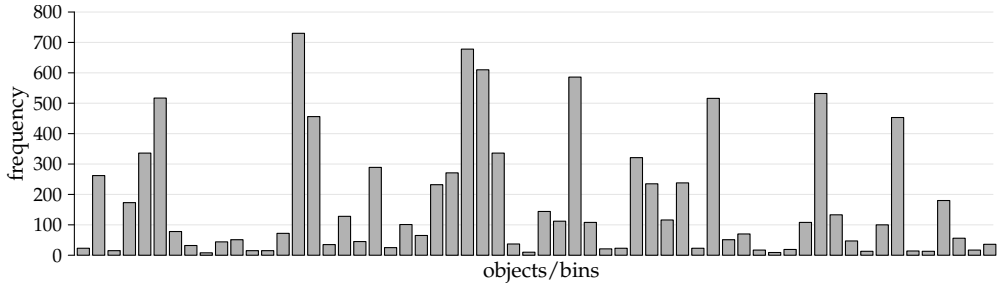


- To analyze such frequency distributions, histograms are helpful.
- A **histogram** is a graphical representation of the frequency distribution of a nominal (or categorial) variable. (Metric variables require binning/discretization.)
- The bar heights represent the frequencies of the values (or intervals if metric).

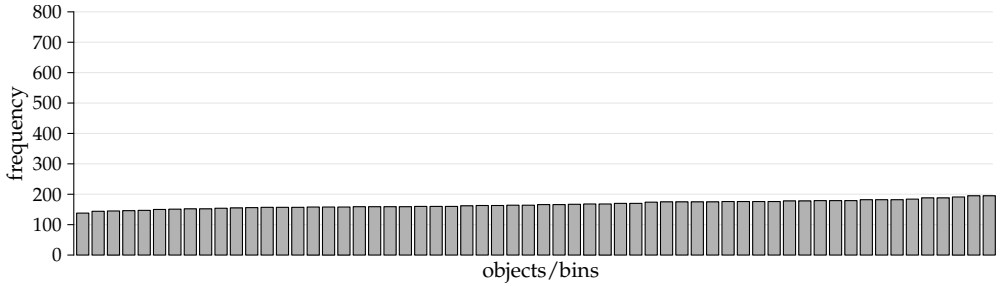
Comparing Histograms: Different Distributions

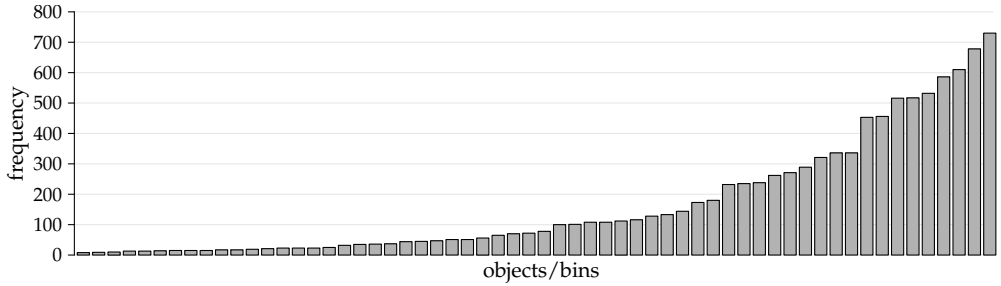
- Example of two different streams of observations:





- Both histograms depict the same number of objects (60) and the same number of data points (10 000).
- However, objects have different probabilities of being observed.
- Sorting objects by their frequency makes the difference very clear:



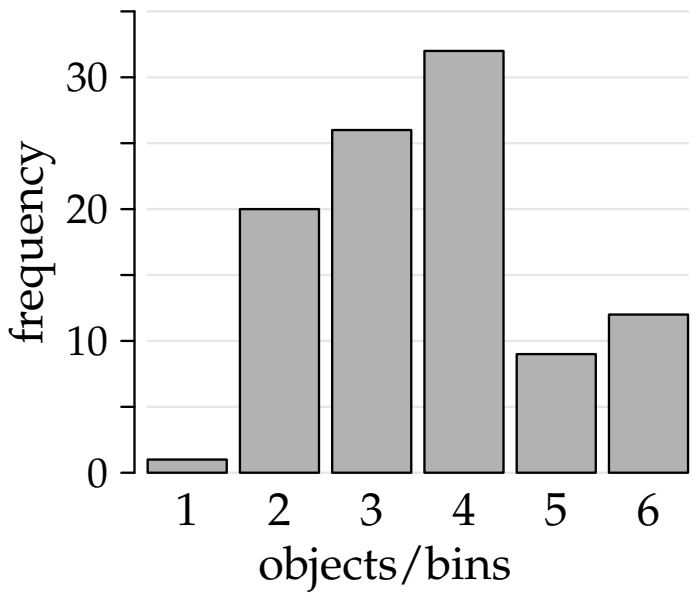


Frequency Distributions

- Information about the distribution of observations
  is crucial for many applications.
  - E.g. traffic monitoring and routing in computer networks:
    frequency of certain hosts connecting (sessions).
- Knowing the complete, exact histogram ...
  - would be ideal, certainly helpful, but
  - is often not possible, due to the large number of distinct objects.
- Workaround:
  - Characterize the histogram without knowing the complete picture.
  - Characteristic properties of histograms are easier to determine.
  - Analogous to descriptions of distributions on $\mathbb{R}$.

    $\Rightarrow$ descriptive statistics as studied earlier, especially
    characteristic measures, and dispersion measures in particular

Characterizing Frequency Distributions



- The variation of the frequencies of objects is sometimes referred to as **frequency skew**.
- This is somewhat misleading/confusing, since it is **not** the **skewness** of the frequency distribution that is desired (in the sense studied in the descriptive statistics part).
- Rather a **dispersion measure** is needed.

- Let O be the set of all objects and let f_o be the frequency of object $o \in O$.
- $F_0 = \sum_{o \in O} (f_o)^0 = \sum_{o \in O} 1$ is the number of distinct objects seen so far.
- $F_1 = \sum_{o \in O} (f_o)^1 = \sum_{o \in O} f_o$ is the total number of objects seen so far.
- Generalization: $F_k = \sum_{o \in O} (f_o)^k$

  F_k is the **k-th moment** of the frequency distribution (k-th frequency moment).

  (Analogous to raw moments of real-valued distributions, see reminder below.)

Frequency Moment Estimation

- F_0: **Number of Distinct Objects**
  - Flajolet–Martin algorithm
  - Bloom filter with Swamidass–Baldi estimation
- F_1: **Total Number of Objects**
  - Simply count the stream elements.
- F_2: **Dispersion of Object Frequencies**
  - This is what we want to estimate next.
- Motivation:
  - A smaller F_2 indicates a less varied distribution.
  - Related to the Gini coefficient (or Gini index), [Corrado Gini 1936]
    a measure of statistical dispersion popular in economics and social science.
  - F_2 can be used to limit approximation errors
    or for query optimization in databases.

Frequency Moment Estimation

Reminder: Moments of Real-Valued Distributions

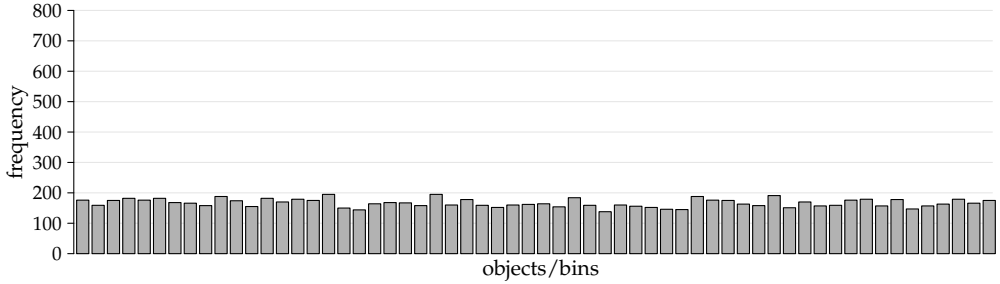
- The k-th (raw) **moment** of a data set is defined as

$$M_k = \frac{1}{n} \sum_{i=0}^{n-1} x_i^k.$$

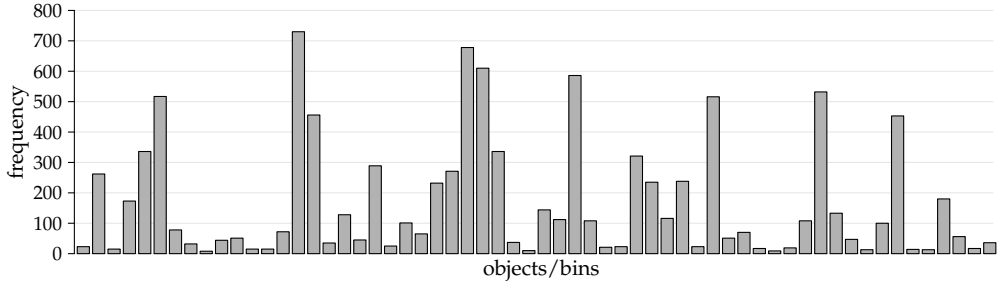
- The k-th **moment about the mean** is defined as

$$\bar{M}_k = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})^k.$$

- The **frequency moment** $F_2 = n \cdot M_2$, where M_2 is the second (raw) moment of the frequency distribution (which is an integer distribution).



$F_2 = 1\,677\,120$



$F_2 = 3\,889\,454$

Frequency Moment Estimation

- Storing and counting distinct objects is usually impossible.
- Hence the second frequency moment needs to be approximated.

Alon–Matias–Szegedy Algorithm [N. Alon, Y. Matias & M. Szegedy 1999]

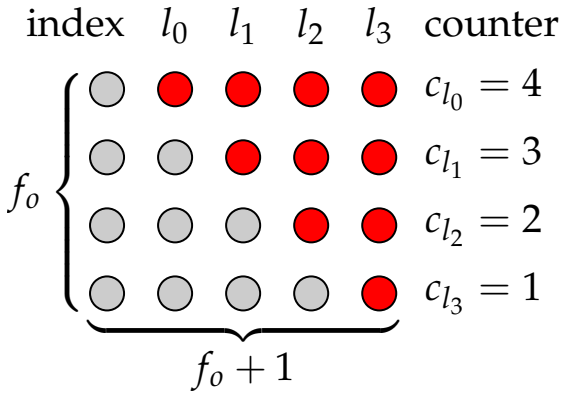
- As usual, we assume that we are given a stream with n elements $x_0, \ldots, x_{n-1}$.
- Sample, without replacement, k locations from the n locations in a stream, that is, choose k random indices $i_j \in \{0, \ldots, n-1\}$, $0 \leq j < k$.
- Traverse the stream elements and on reaching location i_j:
  - Store the object $o_j = x_{i_j}$ encountered at index i_j.
  - Start counting occurrences of this object in a counter c_j.
- Estimate the second frequency monent F_2 as

$$\hat{F}_2 = \frac{n}{k} \sum_{j=0}^{k-1} (2c_j - 1)$$

Alon–Matias–Szegedy Algorithm: Why does it work?

- Suppose $k = n$, that is, a counter is started with every stream element.
- Consider an object o that occurs f_o times in the stream (o has frequency f_o).
- Let $l_0, \ldots, l_{f_o-1} \in \{0, \ldots, n-1\}$ be the locations/indices at which o occurs, that is, $x_r = o$ for $r \in \{l_0, \ldots, l_{f_o-1}\}$ and $x_r \neq o$ for $r \notin \{l_0, \ldots, l_{f_o-1}\}$.
- At the end of the algorithm it is $c_{l_0} = f_o$, $c_{l_1} = f_o - 1, \ldots, c_{l_{f_o-1}} = 1$, because the first counter c_{l_0} counts all occurrences of object o, the second counter c_{l_1} counts all occurrences except the first, …, and the last counter $c_{l_{f_o-1}}$ only counts the very last occurrence.
- Therefore we have

$$\sum_{j=0}^{f_o-1} c_{l_j} = \sum_{j=0}^{f_o-1} (f_o - j) = \sum_{j=1}^{f_o} j = \frac{f_o(f_o+1)}{2}.$$



Alon–Matias–Szegedy Algorithm: Why does it work?

- As a consequence, we can write:

$$\sum_{j=0}^{f_o-1} c_{l_j} = \frac{f_o(f_o+1)}{2} \quad \Leftrightarrow \quad 2\sum_{j=0}^{f_o-1} c_{l_j} = f_o^2 + f_o \quad \Leftrightarrow \quad \left(\sum_{j=0}^{f_o-1} 2c_{l_j}\right) - f_o = f_o^2$$

$$\Leftrightarrow \quad \sum_{j=0}^{f_o-1} 2c_{l_j} - \sum_{j=0}^{f_o-1} 1 = f_o^2 \quad \Leftrightarrow \quad \sum_{j=0}^{f_o-1} (2c_{l_j} - 1) = f_o^2.$$

- Let O be the set of distinct objects occurring in the stream. Then it is

$$F_2 \quad = \quad \sum_{o \in O} f_o^2 \quad = \quad \sum_{o \in O} \sum_{j=0}^{f_o-1} (2c_{l_j^{(o)}} - 1) \quad = \quad \sum_{i=0}^{n-1} (2c_i - 1),$$

  since by summing for all objects over all locations at which these objects occur, we sum over all locations in the stream.

- Hence the formula works if counters are started at all stream locations. What effect does sampling k locations/indices have?

Alon–Matias–Szegedy Algorithm: Why does it work?

- Suppose that the sampled locations are "representative" for the stream.
- In that case, each of the f_o locations/indices at which object o occurs has the same probability of $p = \frac{k}{n}$ of being sampled.
- We may also say that of the f_o possible counters per object on average $\frac{k}{n} f_o$ are actually used if k locations/indices are sampled.
- As a consequence, the expected sum of the k counters is

$$\mathbb{E}\left(\sum_{j=0}^{k-1}(2c_{i_j}-1)\right) \quad = \quad \frac{k}{n}\cdot\sum_{i=0}^{n-1}(2c_i-1) \quad = \quad \frac{k}{n}\cdot F_2.$$

- Assuming that the actual sum is close to the expected value, we get

$$\hat{F}_2 \quad = \quad \frac{n}{k}\cdot\sum_{j=0}^{k-1}(2c_{i_j}-1).$$

Frequency Moment Estimation: Example

Alon–Matias–Szegedy Algorithm: Example

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
x_i	c	e	c	f	a	e	g	f	f	b	b	c	g	b	a	a	f	d	a	e

- Number of distinct objects $F_0 = 7$, number of objects $F_1 = n = 20$.
- Choose (random) locations $i_0 = 4$, $i_1 = 6$, $i_2 = 13$, $i_3 = 18$.
- Location $i_0 = 4$: encounter a, counting results in $c_{i_0} = 4$.
  Location $i_1 = 6$: encounter g, counting results in $c_{i_1} = 2$.
  Location $i_2 = 13$: encounter b, counting results in $c_{i_2} = 1$.
  Location $i_3 = 18$: encounter a, counting results in $c_{i_3} = 1$.
- Estimate: $\hat{F}_2 = \frac{20}{4}\big((2{\cdot}4-1)+(2{\cdot}2-1)+(2{\cdot}1-1)+(2{\cdot}1-1)\big) = \frac{20}{4}{\cdot}12 = 60$.
- Actual value: $F_2 = 4^2+3^2+3^2+1^2+3^2+4^2+2^2 = 64$.

Frequency Moment Estimation: Summary

- F_0: **Number of Distinct Objects**
  - Flajolet–Martin algorithm
  - Bloom filter with Swamidass–Baldi estimation
  - Space complexity: size of hash table
- F_1: **Total Number of Objects**
  - Simply count the stream elements.
  - Space complexity: single counter
- F_2: **Dispersion of Object Frequencies**
  - Alon–Matias–Szegedy algorithm
  - Space complexity: k object stores and counters
- All of these algorithms are simple to implement and yield fairly good approximations.
  (Counting for F_1, of course, is trivial and always exact.)

Sorting: Mergesort

Sorting Large Data Sets

The Importance of Sorting

- About 20-25% of computing resources in large systems are used for sorting or sorting related tasks. [Donald E. Knuth 1998]

Approaches to Efficient Sorting

- Generally: Divide-and-Conquer schemes
- Basically two contrary ideas:
  - Merging (main work done after recursion)
  - Distributing (main work done before recursion)
- Both basic ideas have many different variants.
- Can be adapted in connection with different internal/external memory models, including the model of multiple disks.
- All try to minimize input/output operations and, of course, runtime.

Mergesort: Divide-and-Conquer

Basic Principle of Mergesort

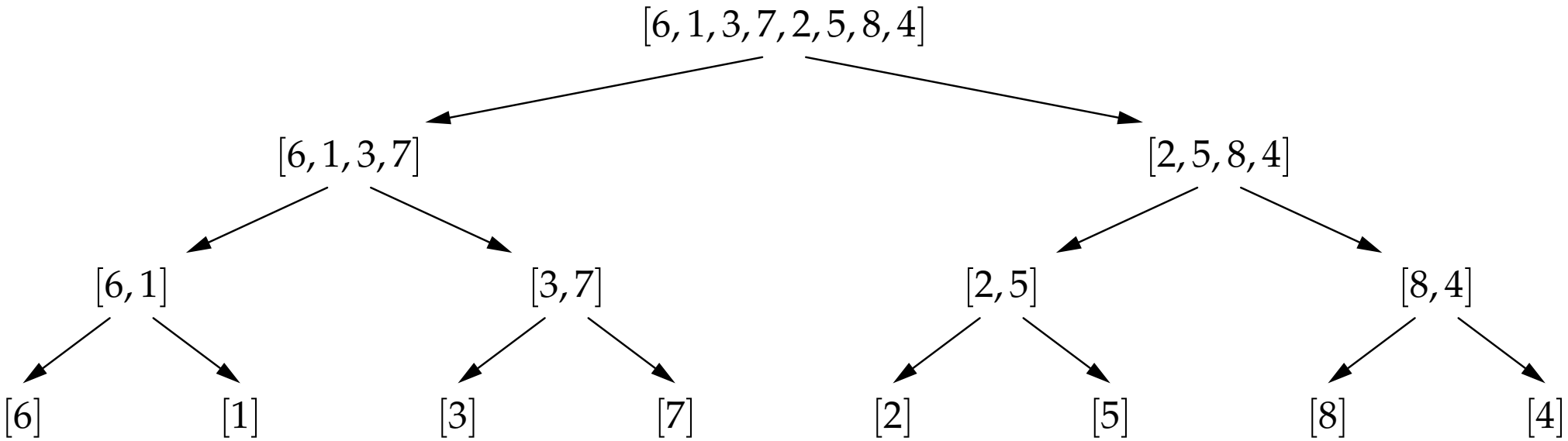
- Given: Array X with n elements $x_0, \ldots, x_{n-1}$.
- Desired: Sorted array X, that is, its elements permuted such that
  $x_{\pi(0)} \leq x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n-1)}$. ($\pi$: permutation)
- Approach: Sort by splitting and merging.

```
function mergesort (X)                  # Attention: only principle!
    k ← ⌊n/2⌋                           # get index of split point
    L ← mergesort(X_{0:k})              # split off and sort the left part
    R ← mergesort(X_{k:n})              # split off and sort the right part
    return merge(L, R)                  # merge the sorted parts
```

- Main work is done after the recursion:
  - Splitting involves fairly little cost.
  - Main work consists in merging the sorted parts.

Mergesort: Divide-and-Conquer

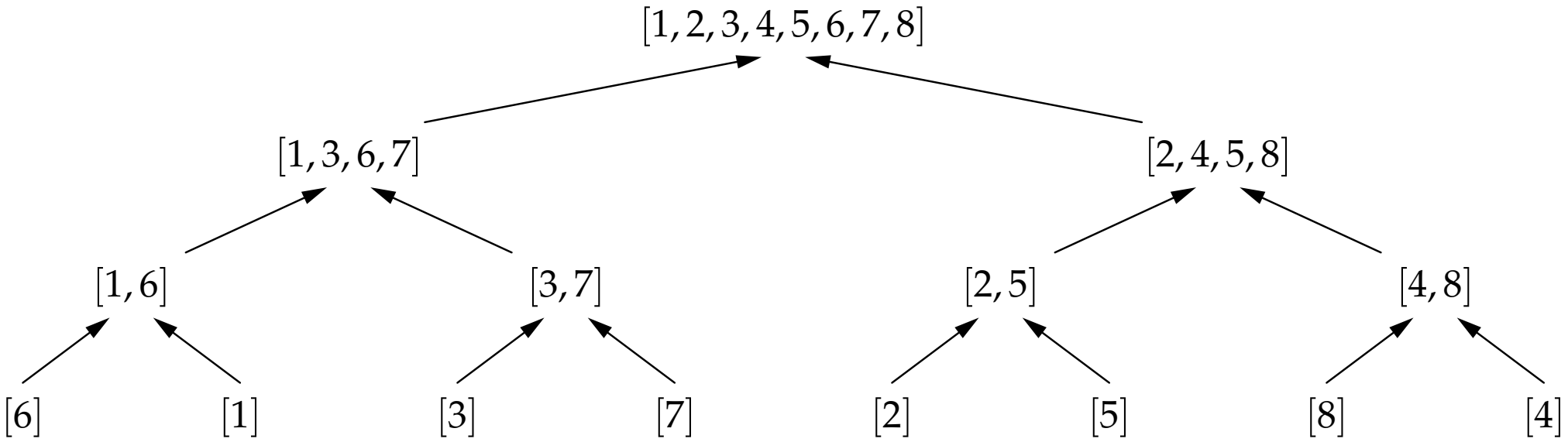
Example Data: ($n = 8$)



- The given data set is recursively split into smaller and smaller data sets.
- Finally single data elements are reached, which are trivial to process.
- The single element data sets are obviously sorted.
- The actual work is done on the way back up: merge the parts.

Mergesort: Divide-and-Conquer

Example Data: ($n = 8$)

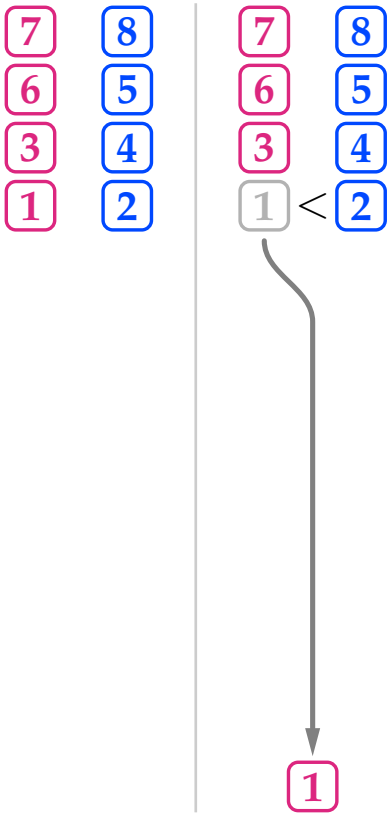


- The actual work is done on the way back up: merge the parts.
- In each node two sorted arrays are merged into a single result array.
- The merging traverses the input lists elementwise and always transfers the smaller element to the output list.
- Merging transfers each element once, tree is $\lceil \log_2 n \rceil$ high. $\Rightarrow O(n \log n)$

Mergesort: Merging Sorted Lists

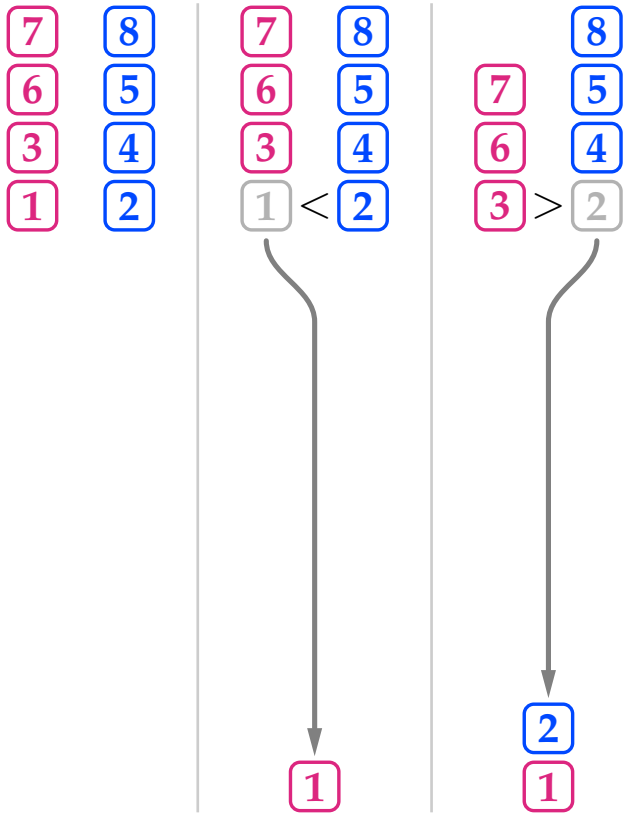
- The core of Mergesort is an operation that merges two sorted lists into one.
- It always transfers the smaller head element from an input list to an output list.

Mergesort: Merging Sorted Lists



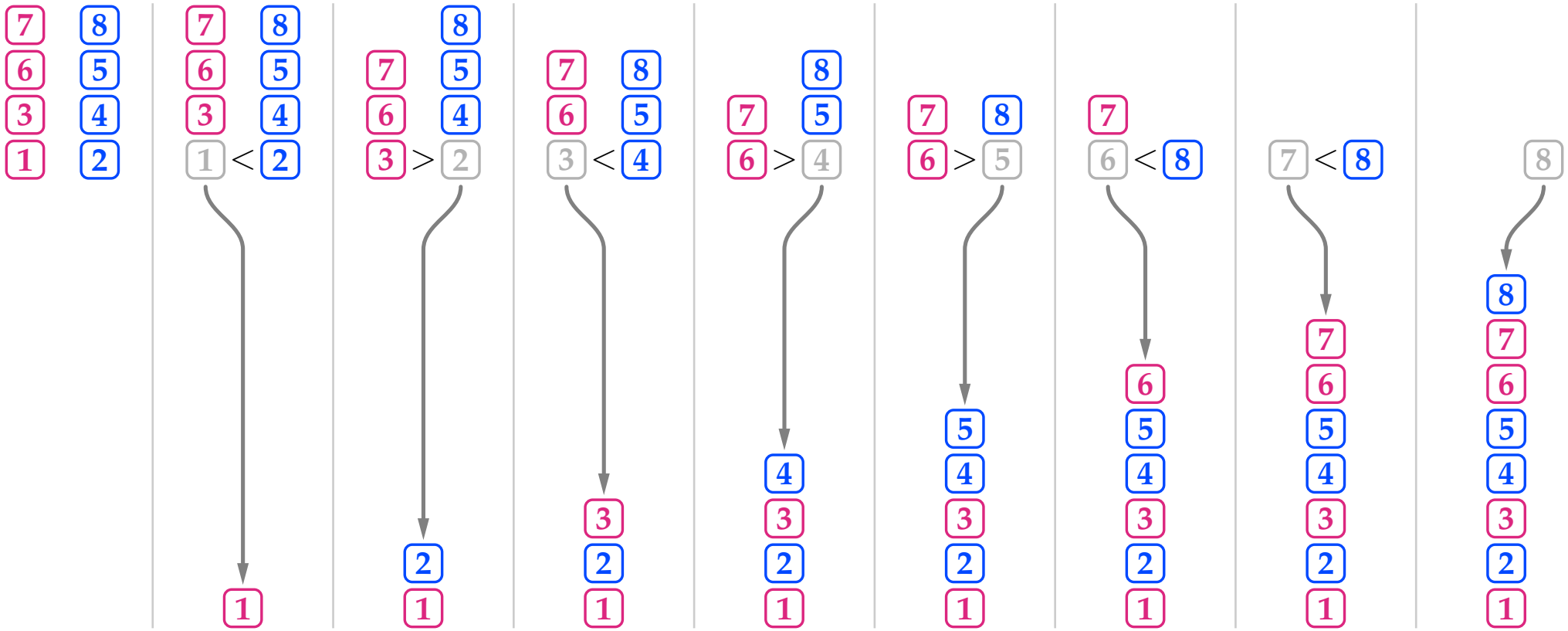
- Since the head element of the left list is smaller, it is passed to the output list.
- The left list thus becomes shorter by one element.

Mergesort: Merging Sorted Lists



- After the removal of the head element from the left list, the head element of the right list is smaller.
- Hence the head element of the right list is moved to the output list.

Mergesort: Merging Sorted Lists



- The process continues until one input list becomes empty.
- Afterward the remaining elements of the non-empty list are simply moved to the output list.

Mergesort: Observations

- Data can be partitioned without reading it (split is arbitrary).
- Individual blocks can be sorted in memory:
  - Load elements from one or more blocks.
  - Sort block (or blocks) in memory
  - Write sorted block (or blocks) back to disk.
- Number of partitions is arbitrary.

  $\Rightarrow$ Multiple parts can be merged in one operation.

Multiway Merging

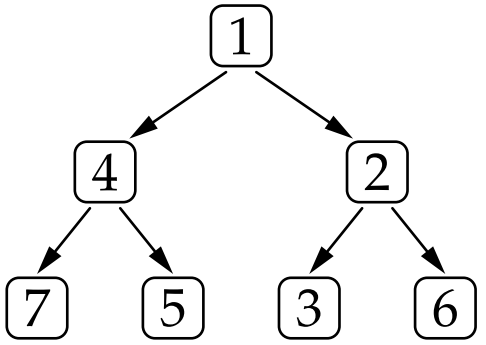
- In principle like two-way merging: always transfer smallest head element.
- Main problem: How to find the list/block holding the smallest element?
- Linear search is not advisable if the number of lists/blocks is (very) large.

Mergesort: Multiway Merging

- The smallest head element can be found efficiently with a **priority queue**.
- A simple data structure for a priority queue is a so-called **heap**.
  - A heap is essentially a **(binary) tree**, each node of which holds a value.
    (In a binary tree each node has at most two children.)
  - Such a binary tree has minimum (maximum) **heap structure**,
    if the value of a parent is no greater (no less) than values of its children.
- Example minimum heap:

  Note that all values of parents
  are smaller than the values of their children.

  Note also that there is no particular order
  of siblings (of the same parent/across parents).

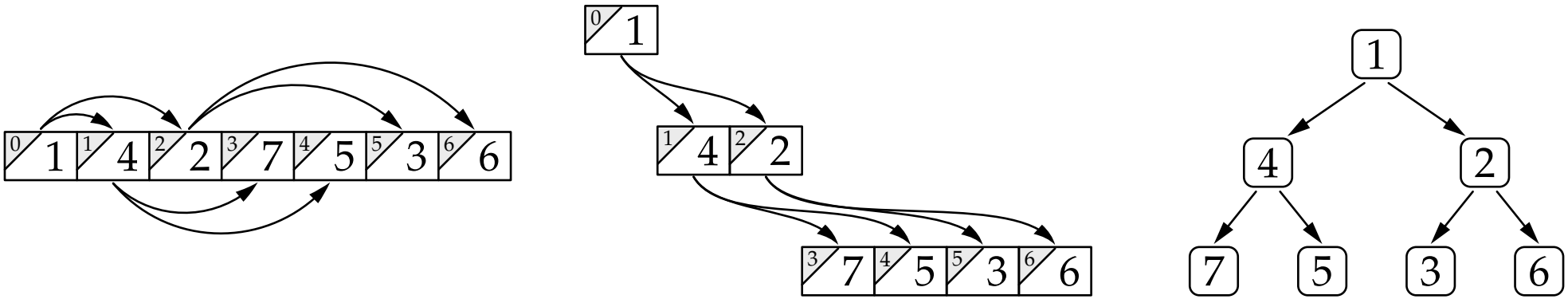


- A heap is **balanced**, that is, the height of sibling subtrees differs by at most 1.
- The height of the whole tree is $O(\log k)$ where k is the number of its nodes.

Mergesort: Multiway Merging

Representing a Heap as an Array

- Each array element represents a tree node and holds its value.
- The children of the node with index i are at indices $2i + 1$ and $2i + 2$.
  (This lists the nodes in a layer-wise traversal.)



- Multiway merging with a heap works as follows:
  - First a (minimum) heap is built for the initial head elements.
  - In each step the root element is passed to the output list.
  - Then the root element is replaced by the next element in its list, which sifts down in the heap to its proper position.

Mergesort: Multiway Merging

Letting an Element Sift Down in a Heap (Attention: minimum heap)

- Let a value v sift down in the heap section $H[\ell:r]$. (H is an array/a list)

```
function sift (H, ℓ, r, v):          # Attention: only principle for simple values!
    i ← 2ℓ + 1;                                  # compute index of first child
    while i < r:                                 # sift loop (while within heap)
        if i + 1 < r and H[i + 1] < H[i]:        # if second child is smaller,
            i ← i + 1;                           # go to second child
        if v ≤ H[i]:                             # if the sift element is smaller
            break;                               # than the selected child, abort
        H[ℓ] ← H[i];                             # let the child ascend in heap
        ℓ ← i;                                   # go to child that ascended
        i ← 2i + 1;                              # compute index of its first child
    H[ℓ] ← v;                                    # store the sift element
```

- This function can be used for both building and updating a heap.
  - Updating: sift$(H, 0, k, v)$. (replaces root element and lets it sift down)

Mergesort: Multiway Merging

Building a Heap (works for both minimum and maximum heaps)

- If a heap is represented by a simple array with k elements, the second half of the array (i.e. $i > \lfloor \frac{k}{2} \rfloor$) always has heap structure.
  (Simply, because there are no parent-child relationships between these elements.)
- Hence only the elements in the first half ($i \leq \lfloor \frac{k}{2} \rfloor$) have to be put into place.
- This can be done by traversing them in the order of decreasing index and letting them sift down in the (partial) heap already built.

  for i **in** $[\lfloor \frac{k}{2} \rfloor - 1, \lfloor \frac{k}{2} \rfloor - 2, \ldots, 1, 0]$: # traverse elements in first half
  sift$(H, i, k, H[i])$; # let elements sift down to build heap
- After a heap has been built, its root node contains the smallest/greatest value.
- Letting an element sift down in a heap of size k has time complexity $O(\log k)$, because the element is passed down a path in the tree, which has height $\lceil \log_2 k \rceil$.
- Therefore building a heap has time complexity $O(k \log k)$.

Mergesort: Multiway Merging

Using a Heap for Multiway Merging

- The heap contains the lists to merge as elements (say: k lists to be merged).
- The heap structure refers to the order relationships of their head elements.
- The root element of the heap refers to the list with the smallest head element. That is, a heap makes it easy to find the next element to transfer.
- The head element of the list at the root is transferred to the output list.
- The new head element of this list may violate the heap structure, so we have to let the list sift down in the heap to restore heap structure.
- Afterward the root of the heap refers again to the list with the smallest head.
- Restoring heap structure takes one sift operation and hence $O(\log k)$.
- A linear search for the smallest head element would have time complexity $O(k)$, which for large k is far worse.

Multiway Mergesort for External Memory

- Partition data to sort into chunks of size M (M: main memory size).
- Sort each chunk in memory (in place/*in situ*), write sorted chunks back to disk.
- Result: $k = \lceil \frac{N}{M} \rceil$ sorted (sub-)streams (N: total size of data).
- Internal sorting takes $O(M \log M)$ time per chunk,
  for all chunks together $O(\frac{N}{M} \cdot M \log M) = O(N \log M)$.
- Chunk sorting requires $O(\frac{N}{b})$ input/output operations (b: block size),
  because each block has to be read once and written once.
- Use k-way merging to create completely sorted stream.
  - Reduce (at most) $\lfloor \frac{M}{b} \rfloor - 1$ sorted streams to one.
    (Need one block per input chunk plus one for the output chunk, $\lfloor \frac{M}{b} \rfloor$ fit into memory.)
  - Repeat until all streams are combined into one.
    (If $\lfloor \frac{M}{b} \rfloor \leq k$, multiple merging steps are necessary.)
  - Total number of input/output operations: $O\left(\frac{N}{b} \log_{\frac{M}{b}} \frac{N}{b}\right)$.

Excursion: Heapsort

- A very simple sorting algorithm is **straight selection**,
  which repeatedly finds the largest element in a section and swaps it to the end.

```
function selectionsort (X):                 # sorting by straight selection
    n ← len(X);                             # get the number of elements
    for i in [n−1, n−2, ..., 2, 1]:         # traverse list prefixes backwards
        m ← 0;                              # initialize index of largest element
        for k in [1, 2, ..., i]:            # traverse remaining prefix elements
            if X[k] > X[m]: m ← k;          # if larger element found, note index
        exchange X[m] and X[i];             # swap largest element to end of prefix
```

- Straight selection sort is very inefficient: $O(n^2)$.
- The main problem is the linear search for the largest element.
- However, this is the same problem that one should avoid in multiway merging:
  a linear search for the smallest head element of an input list is inefficient.
- Solution: use a **heap** to find the largest (or smallest) element.

Excursion: Heapsort

- In principle, **Heapsort** works like straight selection, that is,
  it repeatedly finds the largest element in a section and swaps it to the end.
- The difference is that it finds the largest element (attention: maximum heap!)
  by organizing the array (prefix) in which to find it as a (maximum!) heap.

```
function heapsort (X):                          # sorting by selection with a heap
    n ← len(X);                                 # get the number of elements
    for i in [⌊n/2⌋−1, ⌊n/2⌋−2, ..., 1, 0]:     # traverse elements in first half
        sift(H, i, k, H[i]);                    # let elements sift down to build heap
    for i in [n−1, n−2, ..., 2, 1] :            # traverse list prefixes backwards
        exchange X[0] and X[i];                 # swap largest element to end of prefix
        sift(H, 0, i−1, H[0]);                  # let new top element sift down in heap
```

- Using a (maximum) heap to find the largest element is much more efficient.
  - The sift operation takes at most $O(\log n)$ time (tree height/path length).
  - Building the heap needs $\frac{n}{2}$ sift operations;
    processing the $n-1$ prefixes needs one sift operation each. $\Rightarrow O(n \log n)$.

Sorting: Quicksort and Hyper-Quicksort

Parallel Computing

- Up to now: Streaming, online processing, data on external memory.
  Now: Speeding up processing by parallelization, data in internal memory.
- Modern computers possess **multiple processors**, permitting parallel execution.
- **Parallelism** can be modelled in different ways.
  - There are p (independent) processing units.
  - Each processing unit has access to some form of memory.
  - There is some form of communication between processors.
- For **single machine parallel computing** (our topic here):
  - p is the number of CPUs (Central Processing Units) in the machine.
  - All CPUs access the same memory.
  - Communication can be realized via memory access.
- This is also known as the **PRAM Model** (Parallel Random Access Machine).

Realizability of Parallelization

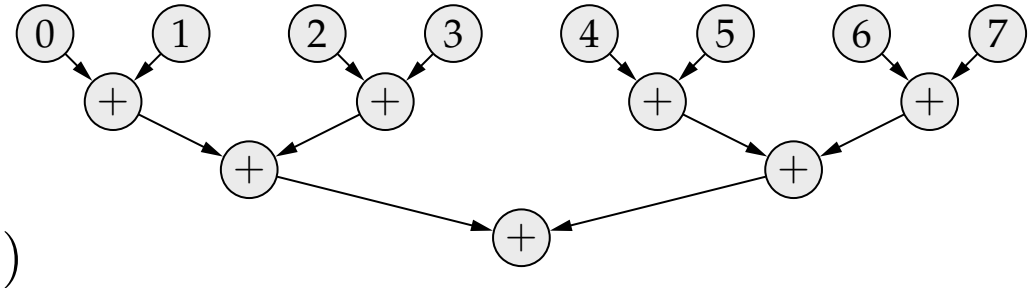
- Some problems have a natural parallel solution, for example:
  - Apply one operation to all elements in a collection.
  - Divide-and-Conquer algorithms.
- Other problems can benefit from parallelism, but not in a trivial way.
  - Common problems: communication between processors, load balancing.
- Yet other problems do not benefit from parallelism.
- Problems are categorized by the benefit resulting from using additional CPUs.
- Formalized by a complexity class called **Nick's Class** (NC).
  [after Nick Pippenger, coined by Stephen Cook 1985]

  Nick's Class is the set of decision problems decidable in polylogarithmic time on a parallel computer with a polynomial number of processors.

Parallel Computing: Recursive Reduction Scheme

- Functions of the form

  function $f(X)$:
  $X_1, \ldots, X_k = \text{split}(X)$
  return $\text{combine}(f(X_1), \ldots, f(X_k))$



  can be evaluated using a recursive reduction scheme.

- Implementations of the function f are often named **reduce**.
  (Loosely related to **MapReduce** — later . . . if there is time.)

- Differences are determined by the split and combine functions.
  - Arbitrary split as in Mergesort, recursive mean & variance (see above).
  - Non-arbitrary split as in Quicksort (see below).
  - Combination functions that are
    – commutative ($\text{combine}(A, B) = \text{combine}(B, A)$) and
    – associative ($\text{combine}(\text{combine}(A, B), C) = \text{combine}(A, \text{combine}(B, C))$)
    allow for an arbitrary processing order.

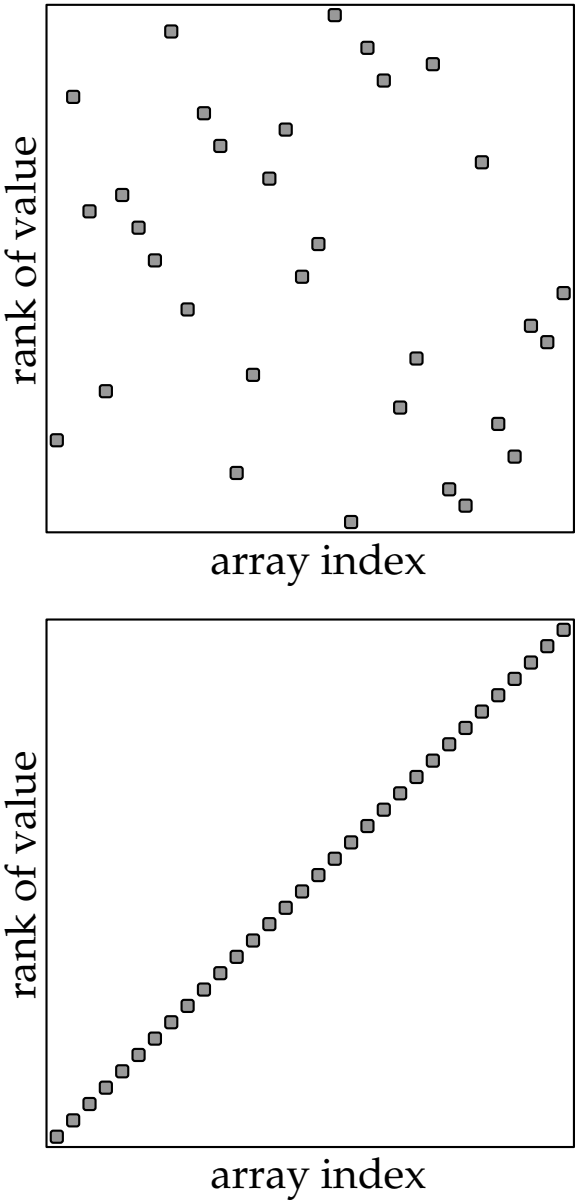
Parallel Computing: Sorting

- Here we take a look at how **sorting** can benefit from parallelization.
- As a Divide-and-Conquer scheme, Mergesort can fairly easily be parallelized.
- However: Mergesort cannot work "in place" (use only the array to be sorted).
  The merge operation needs a separate chunk of memory for the output list.

  $\Rightarrow$ Required memory is twice as large as the data to sort.
- A Divide-and-Conquer sorting scheme that can work "in place" is **Quicksort** (no additional memory needed). [C. Antony R. Hoare 1960]
- Fundamental idea of Quicksort:
  - Choose a pivot element (from the objects to be sorted).
  - Split the data w.r.t. this pivot element, moving
    all elements less than (or equal to) the pivot to the start of the array and
    all elements greater than the pivot to the end of the array,
  - Recursively process the two resulting sections.

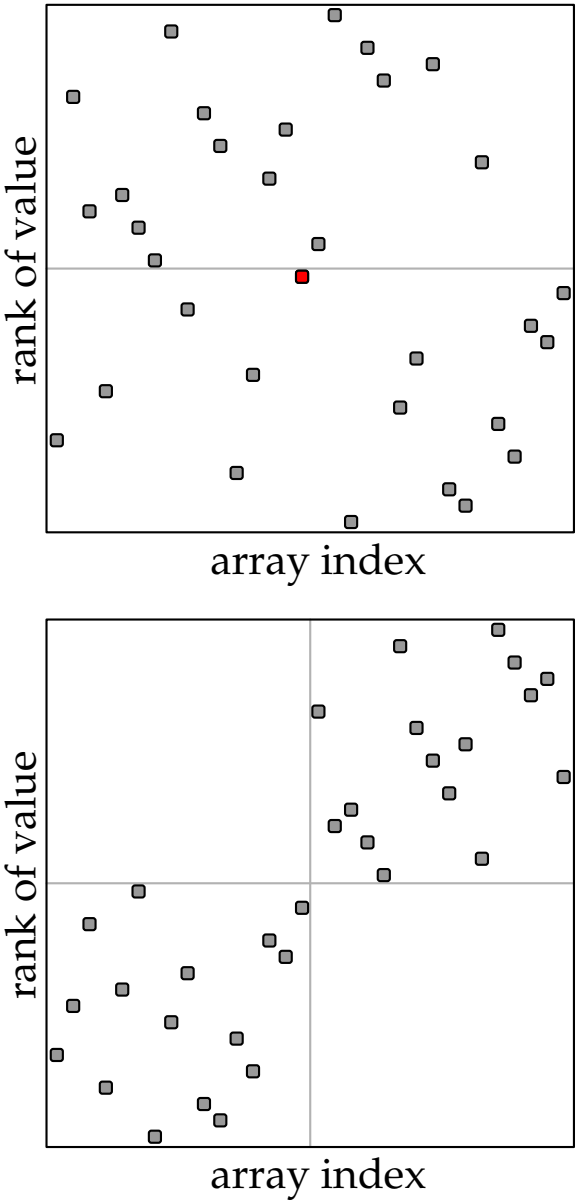
Visualization of Sorting

- For understanding Quicksort,
  a simple visualization is beneficial.
- The state of the array to be sorted
  is depicted with two dimensions.
  - Horizontal axis: array index
  - Vertical axis: rank of value at array index
- The top diagram on the right
  shows an unsorted array:
  Array indices and value ranks differ.
- The objective of sorting is to make
  an array index and the rank of the value
  stored at that index coincide (for all indices).
- The bottom diagram on the right
  shows a sorted array.



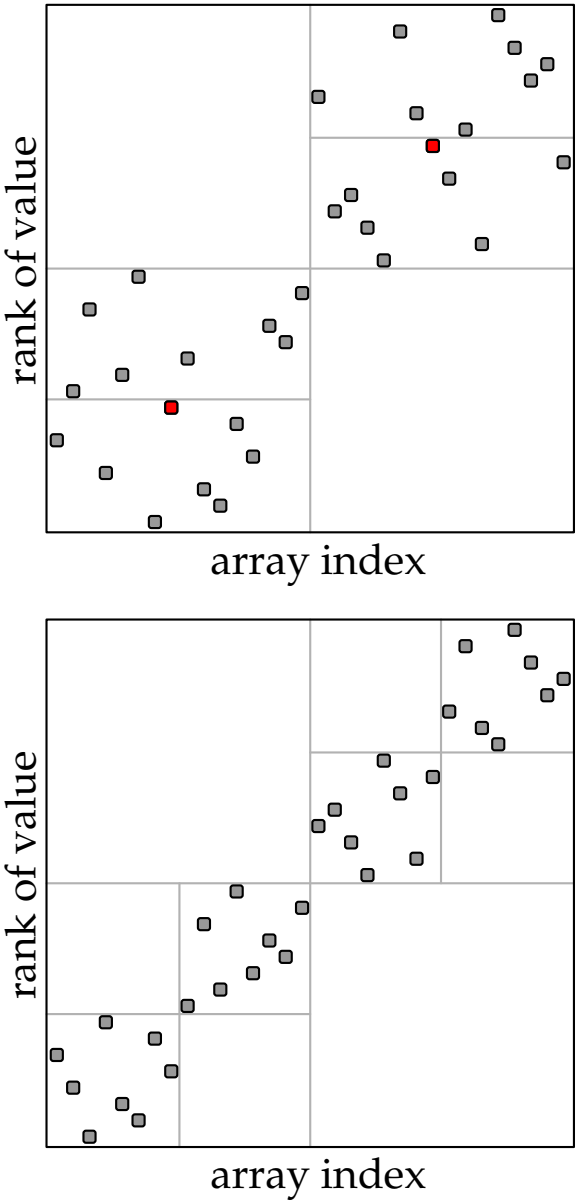
Quicksort

- Quicksort first chooses a pivot element, commonly (one of) the middle elements(s) (red box in the top diagram on the right).
- All elements less than this pivot are moved to the front/left of the array.
- All elements greater than this pivot are moved to the end/right of the array.
- The resulting situation is shown in the bottom diagram on the right.
- Note that all array elements to the left of the vertical line are smaller than those to the right of the vertical line.
- Clearly, we are closer to the desired end: all array elements on the diagonal.



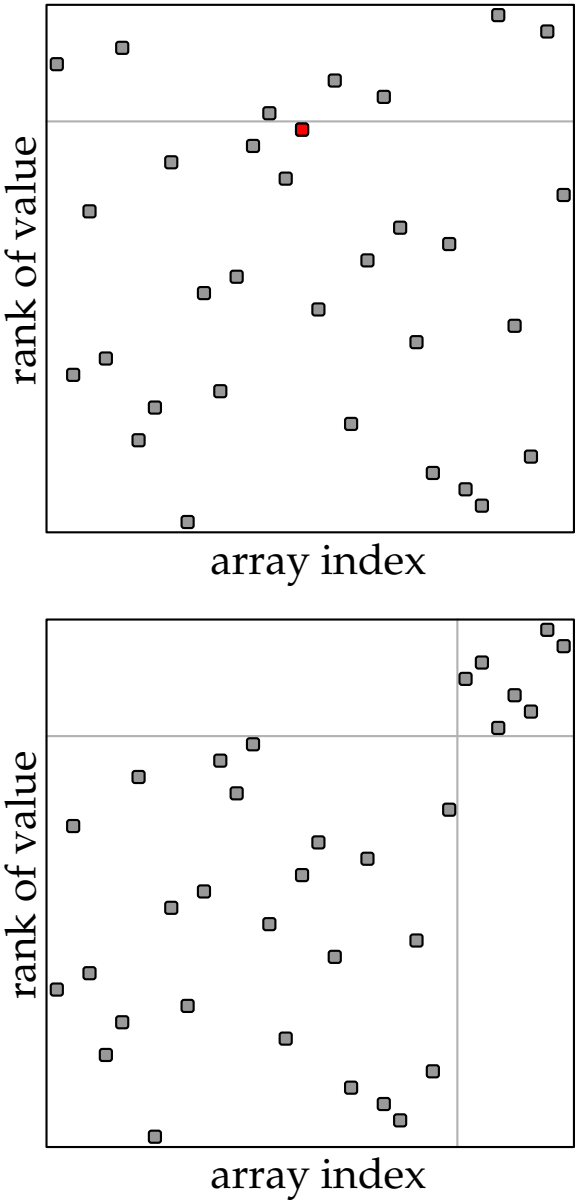
Quicksort

- In the recursion, the process repeats:
  For each section a pivot element is chosen
  (red boxes in the top diagram on the right).
- All elements less than this pivot
  are moved to the front/left of the section.
- All elements greater than this pivot
  are moved to the end/right of the section.
- The resulting situation is shown
  in the bottom diagram on the right.
- Note that all array elements to the left
  of a vertical line are smaller than
  those to the right of that vertical line.
- Clearly, we are closer to the desired end:
  all array elements on the diagonal.



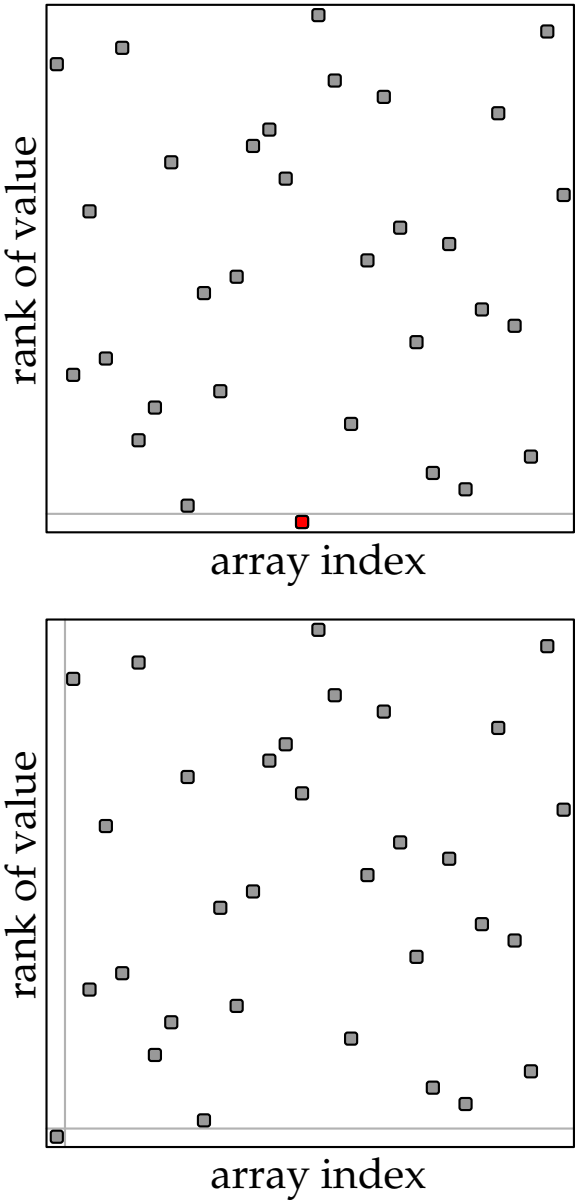
Quicksort

- The diagrams on the preceding slides show an ideal situation wr.t. the pivots: all pivots are the median elements.
- In practice, this will not always be the case.
- Rather it is to be expected that the split into sections is uneven.
- This is illustrated in the diagrams on the right: The chosen pivot is not the median element.
- Moving elements less than (or equal to) the pivot to the front/left of the array and elements greater than the pivot to the end/right, still gets us closer to the desired end.
- However, the recusion will be uneven.
  $\Rightarrow$ Choosing a good pivot is important.



Quicksort

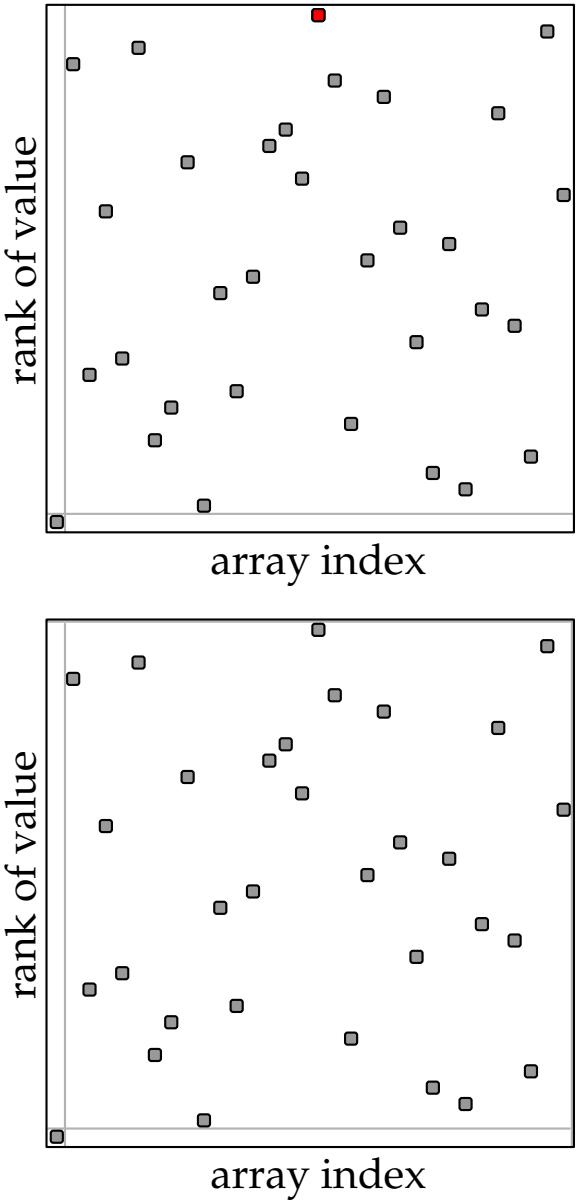
- To emphasize the choice of a good pivot, the diagrams on the right show a very unfortunate choice: the pivot is the smallest element.
- In this case only the pivot ends up on the left, all other elements are moved to the right.
- Even though we get closer to the desired end, the progress is very small.
- The recursion is maximally uneven: no work needs to be done on the left section.
- The right section comprises almost all elements, so the work is hardly diminished.
- If this repeats, the time complexity deteriorates to that of straight selection: $O(n^2)$.
  (Find smallest/greatest element and swap to start/end.)



Quicksort

- The next step is even worse:
  The chosen pivot is the largest element
  (red box in top diagram on the right).
- Moving elements less than (or equal to) the pivot
  to the front/left of the array and elements
  greater than the pivot to the end/right,
  leaves the right section empty.
- Hence, in the recursion nothing has changed
  and no progress is made (endless recursion).
- This indicates that one has to be careful
  when implementing Quicksort, especially,
  how to treat the pivot element.
- How this problem can be overcome
  is left for the exercises.

  (Hint: Maybe deal with the pivot separately.)



Quicksort: Divide-and-Conquer

- Given: Array X with n elements $x_0, \ldots, x_{n-1}$.
- Desired: Sorted array X, that is, its elements permuted such that
  $x_{\pi(0)} \leq x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n-1)}$. ($\pi$: permutation)
- Approach: Sort by splitting with a pivot and concatenating.

```
function quicksort (X)                          # Attention: only principle!
    p ← X[⌊n/2⌋]                                # get pivot element from middle
    L ← quicksort([x for x in X if x < p])      # collect elements less than pivot
    R ← quicksort([x for x in X if x > p])      # collect elements greater than pivot
    return concatenate(L, [p], R)               # concatenate the sorted parts
```

- Attention: This scheme assumes that all elements to be sorted are different!

  If there are equal elements, the above scheme needs adaptations
  (apart from adding a termination criterion for the recursion).

- Difference to Mergesort: Main work is done in the splitting process.
  Merging partial solutions is trivial (concatenation).

Quicksort: Influence of the Pivot

- If the chosen pivot element is the median element (or close to it), the resulting sections are (roughly) equal.
- In this **best case** the sections sizes halven in each recursion step.

  As a consequence, the time complexity is $O(n \log n)$.
- If the chosen element is the smallest (largest) element (or close to it), the resulting sections are maximally unequal.
- In this **worst case** one of the sections is hardly smaller than the original, while the other section is (almost) empty.

  As a consequence, the time complexity is $O(n^2)$.
- In the **average case** (considering all possible array contents), Quicksort tends to have a time complexity of $O(n \log n)$, so usually no effort is expended on finding a good pivot.
- However, the possibility of the worst case $O(n^2)$ is annoying.

Finding a Good Pivot: Median of Medians

- The optimal pivot is the median of the elements in the array,
  because this leads to equal section sizes (differing by at most one element).
- Finding the true median is costly, but an approximation is feasible.
- Idea: Approximation by recursively computing a **median of medians**.
  - Split array into k (roughly) equal sections.
  - Process each section recursively to get an approximate median for each.
  - Return the median of the obtained k approximate medians.
  - Most common choice: $k = 3$ (makes median computation easy).
- This approach guarantees a fairly good pivot, close to the median.
- However, it is fairly costly, involving a recursive scheme,
  in each recursion step of the Quicksort recursion.
- Therefore, in practice, merely the element at the middle index is chosen,
  or the median of the first, last and middle element of the array.

Excursion: Combining Quicksort with Insertion Sort

- A very simple sorting algorithm is **straight insertion**,
  which repeatedly inserts the next element into an already sorted section.

```
function insertionsort (X):                  # sorting by straight insertion
    for i in [1,2,...,len(X)−1]:             # traverse elements to insert
        t ← X[i];                            # note next element to insert
        k ← i;                               # initialize index of insertion point
        while k > 0 and X[k−1] > t:          # while preceding element is greater,
            X[k] ← X[k−1];                   # shift this element to the right
            k ← k−1;                         # move insertion point to the left
        X[k] ← t;                            # store element at insertion point
```

- Straight insertion sort is very inefficient: $O(n^2)$.
  The main problem is the linear shifting of the list elements.
- However, for short lists, insertion sort is more efficient than Quicksort.
- Idea: Use Quicksort only down to a certain section size (e.g. 8 elements),
  and apply, as a final step, insertion sort to the whole array.

Parallelizing Quicksort: Idea

- Modern computers possess multiple processors (CPUs with multiple cores).
  Let $p = 2^k$, $k \in \mathbb{N}$, be the number of available processors.
- Each processor gets a chunk of (roughly) n/p of the objects to be sorted.
- The objective of the parallelization is to make the chunks independent.
  Here in particular: objects held by processor P_i should be smaller than objects held by processor P_j whenever $i < j$.
- Problem: We cannot let every processor choose its own pivot.
  Such an approach will—in all likelihood—lead to p different pivots, which makes it impossible to compare the resulting partitions.
- Solution: Let one processor choose a pivot and broadcast it to the others.
- Each processor splits its objects with the received pivot (done in parallel).
- Result: Each processor has a chunk of objects that are greater than the pivot and a chunk of objects that are less than or equal to the pivot.

Parallelizing Quicksort: Idea

- Pairs of processors swap parts less/greater than pivot.

  $\forall i; 0 \leq i < \frac{p}{2}$: Processor P_i gives part with greater objects to processor $P_{i+\frac{p}{2}}$.

  $\forall i; \frac{p}{2} \leq i < p$: Processor P_i gives part with smaller objects to processor $P_{i-\frac{p}{2}}$.

- Result: Processors $P_0, \ldots, P_{\frac{p}{2}-1}$ hold objects less than or equal to the pivot.
  Processors $P_{\frac{p}{2}}, \ldots, P_{p-1}$ hold objects greater than the pivot.

- As long as there are at least two processors in each part,
  recurse into the two parts and process each with their processors.

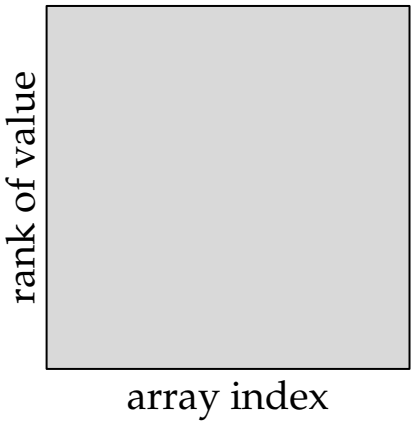
- After the processing returns from the recursion, we have:
  objects held by processor P_i are smaller than
  objects held by processor P_J whenever $i < j$.

- Each processor sorts its chunk of objects (e.g. with standard Quicksort).

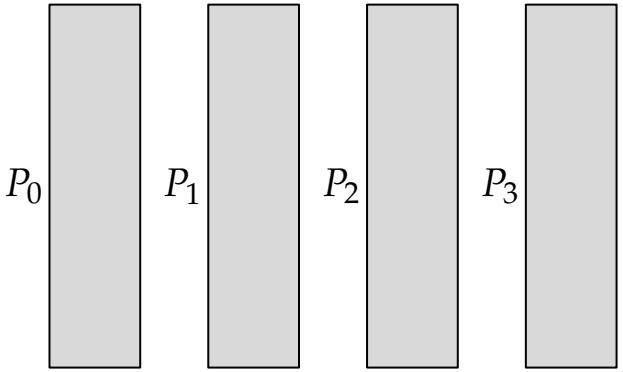
- By concatenating the chunks from the processors (in the order of their indices),
  we obtain a fully sorted list of objects.

Parallelizing Quicksort: Idea

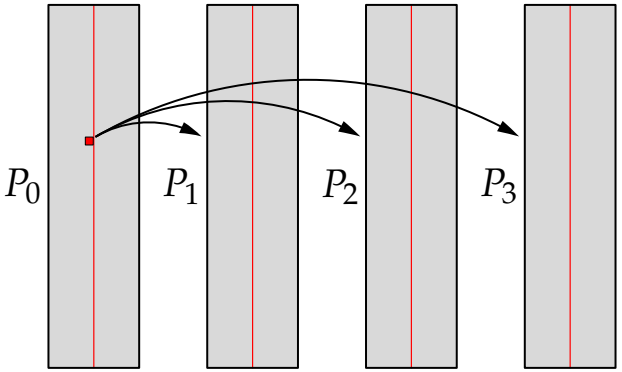
1. Initial situation: unsorted array.



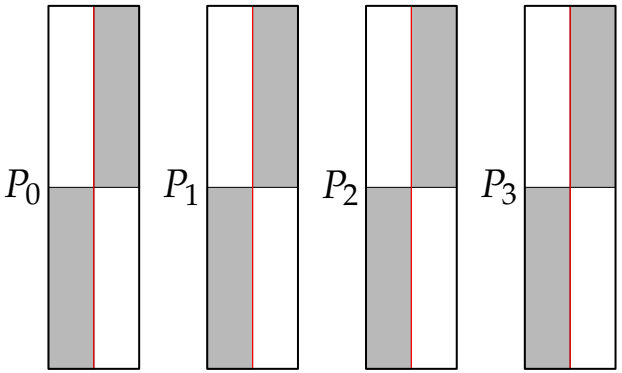
2. Split into chunks and assign to processors.



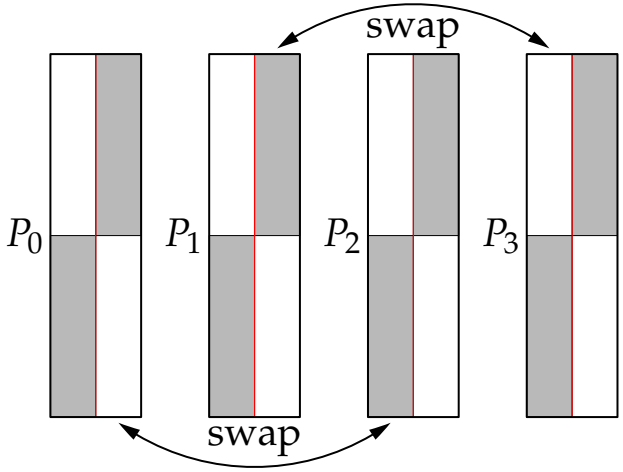
3. Choose pivot and broadcast.

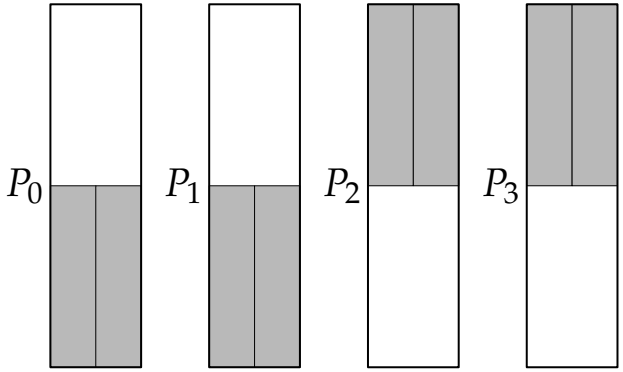


4. Processors split chunks with pivot.



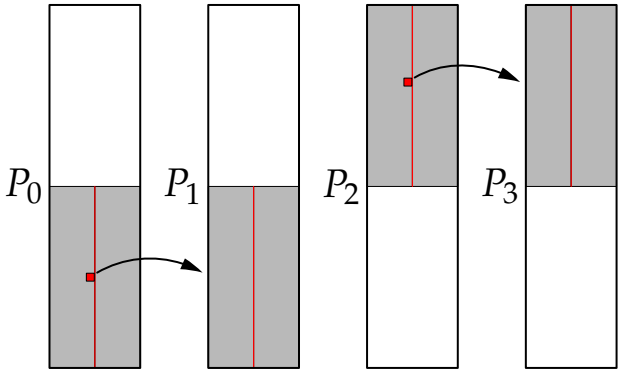
5. Swap parts with other processors.



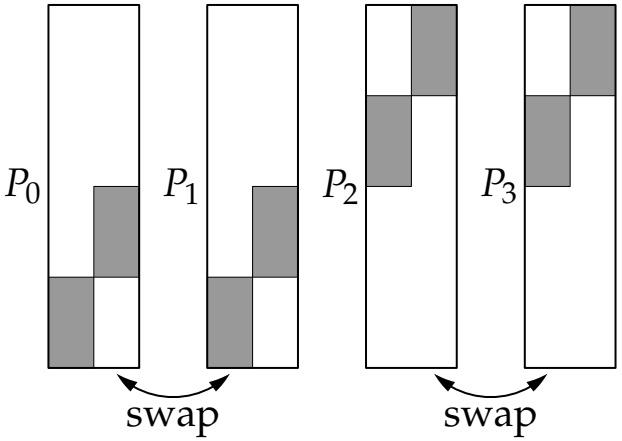


Parallelizing Quicksort: Idea

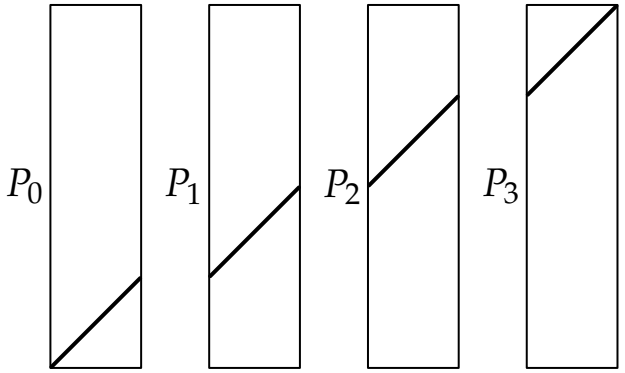
6. Recursion: Choose pivot and broadcast.



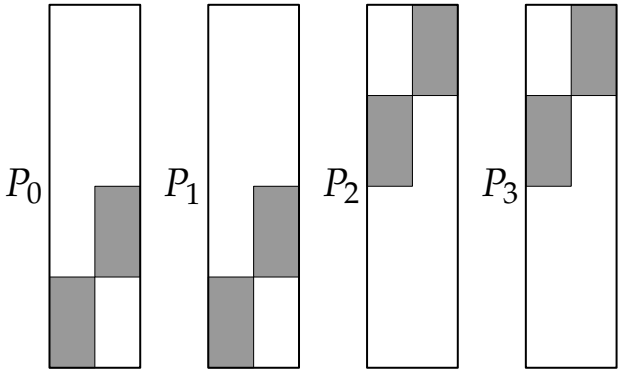
8. Swap parts with other processors.



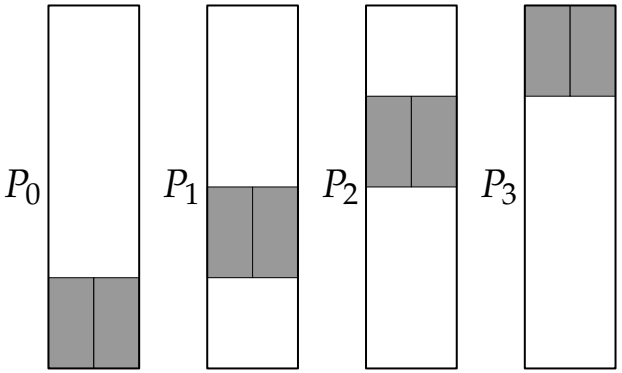
9. Sort chunks of each processor.



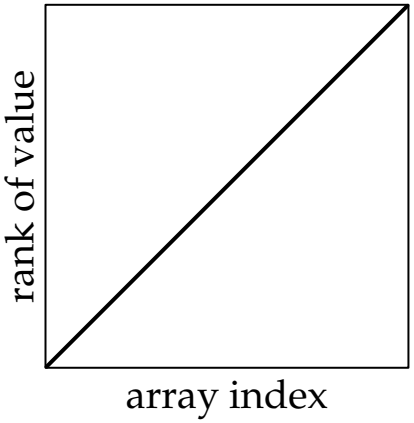
7. Processors split chunks with pivot.



After swap, desired situation is reached.



10. Concatenate sorted chunks of processors.



Parallelizing Quicksort: Analysis

- The partioning recursion has depth $k = \log_2 p$ (since we assumed $p = 2^k$).
- In the partitioning process, the work on a subset S_i assigned to processor P_i is linear in the subset size s_i (number of objects in S_i).

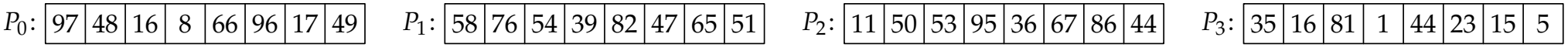
  (Splitting a chunk of objects w.r.t. the pivot visits each object in the chunk once.)
- The time complexity of sequential sorting (for each processor) is $O(s_i \log s_i)$.
- In order to get a actual speed-up from the parallelization, the partition sizes should be as equal as possible.

  (If all but one processor get very few objects, possibly even just one, and all remaining objects are assigned to the one remaining processor, processing is rather slower than with a single processor due to the communication overhead/swapping parts.)
- How equal the partition sizes are depends on the pivot.

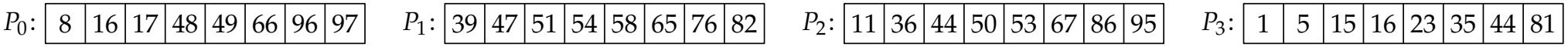
  Therefore: Try to find a pivot that is as good as possible.
- Idea: Instead of sorting at the very end of the process, sort at the beginning as well, to obtain a median object.

Parallelizing Quicksort: Example

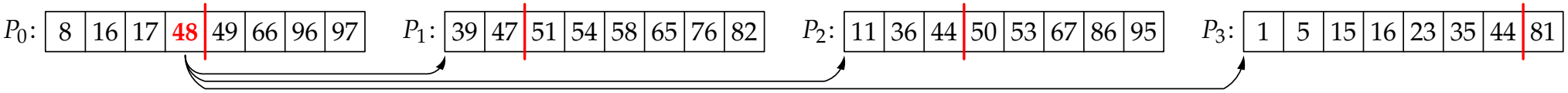
1. Objects to be sorted are distributed (roughly) equally to the processors.



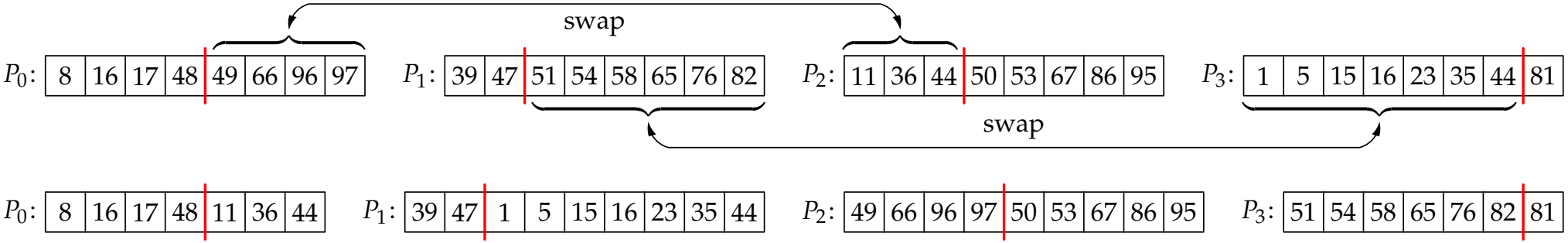
2. Each (or at least 1st) processor sorts the objects assigned to it.



3. 1st processor chooses a pivot (median) and broadcasts it to the other processors.

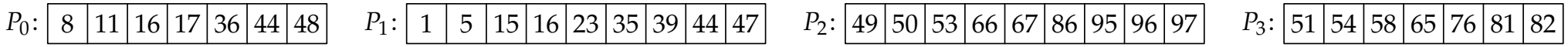


4. Pairs of processors swap objects less/greater than the pivot (joining objects less/greater).

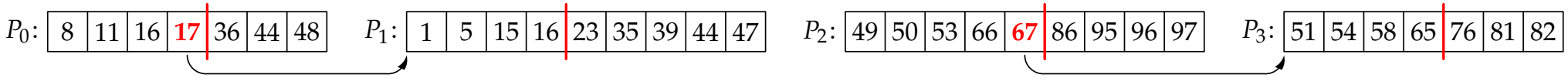


Parallelizing Quicksort: Example

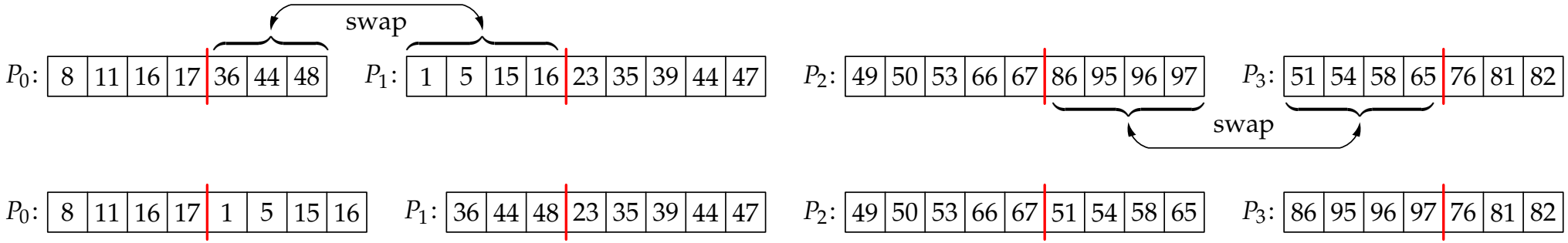
5. Each (or at least 1st and 3rd) processor sorts the objects assigned to it.



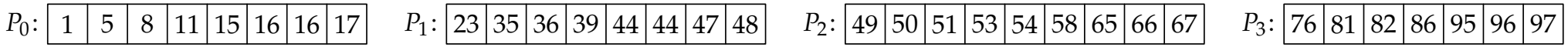
6. Two processors choose a pivot (median); each distributes it to another processors.



7. Pairs of processors swap objects less/greater than the pivot (joining objects less/greater).



8. Each processor sorts the objects assigned to it.



Parallelizing Quicksort: Analysis and Improvement

- The partioning recursion has again depth $k = \log_2 p$.
- The partition sizes tend to be more equal (though not perfectly).
- However, we still have $O(s_i \log s_i)$ for the sorting in **each** recursion step.

Hyper-Quicksort

- If *all* processors sort their chunks in the first step (top level of recursion), we have two sorted lists in each chunk after swapping.

  (The scheme above only required processors that choose pivots to sort their chunks. However, actually only the median needs to be found: $O(s_i)$ on average, but beyond scope.)
- Already sorted lists/parts can be merged (as in Mergesort), reducing the time complexity of (re-)sorting a chunk to $O(s_i)$.

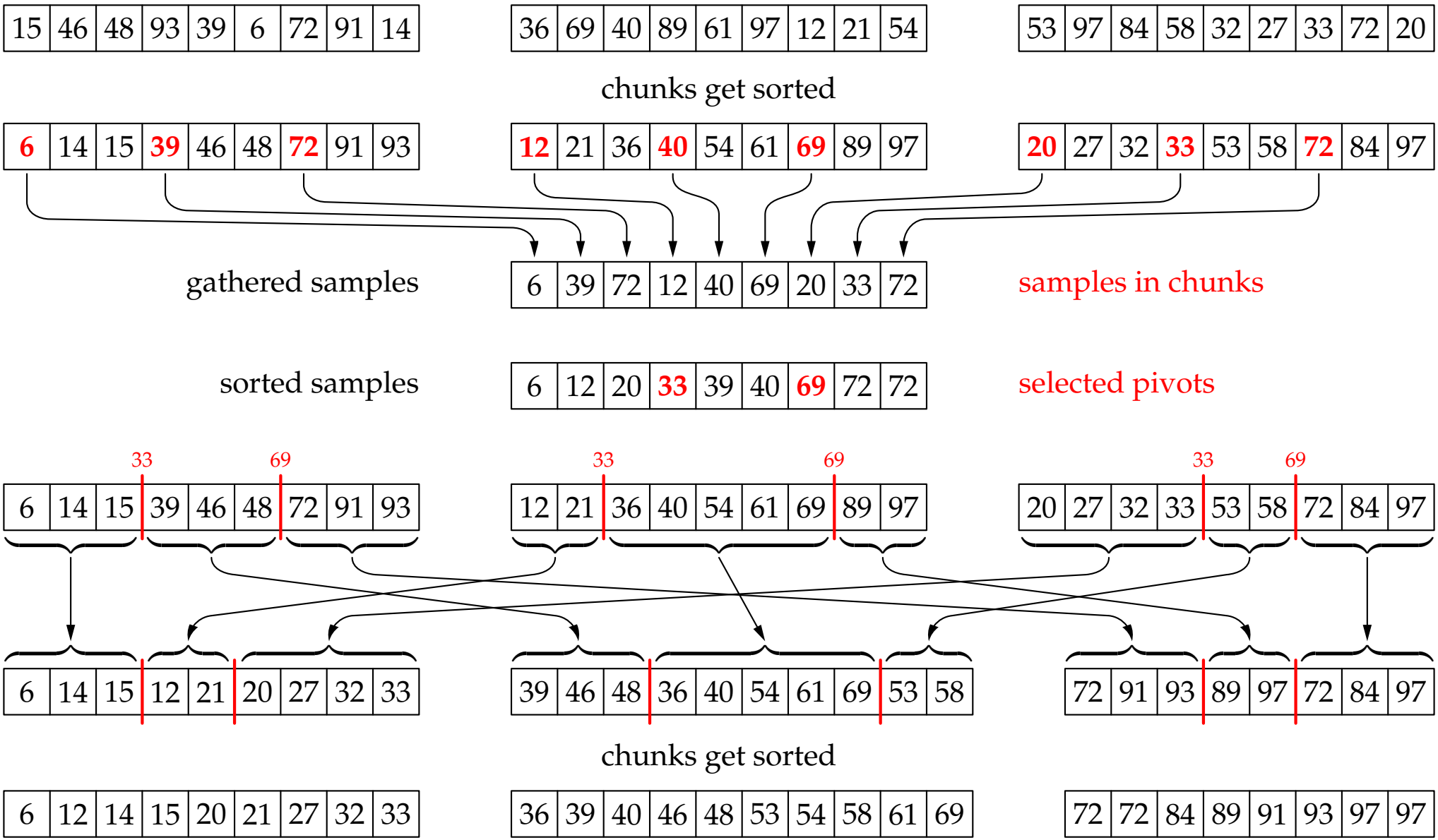
  (Because each object has to be transferred to the output list.)
- Partitioning only needs to find the split point in a sorted list: $O(\log s_i)$

  (But merging the parts dominates, so still time complexity $O(s_i)$.)

Parallel Sorting Using Regular Sampling

- Up to now: number p of processors is a power of 2, that is, $p = 2^k$.
  Now: arbitrary number p of processors.
- Each processor sorts its chunk of objects and selects the objects at local indices
$$0, \quad 1\frac{n}{p^2}, \quad 2\frac{n}{p^2}, \quad \ldots, \quad (p-1)\frac{n}{p^2}.$$
  as a regular sample from its chunk of objects.
- One processor gathers and merges the samples, sorts the result (size p^2), selects $p-1$ pivot values and broadcasts them to all other processors.
- Each processor partitions its sorted chunk w.r.t. the pivots into p disjoint parts.
- Each processor P_i keeps the i-th part of the partition and transmits each j-th part, $j \neq i$, to processor j.
- Each processor merges its p parts into a single chunk (multiway merging).
- The chunks of the processors are concatenated in the order of their indices.

Parallel Sorting Using Regular Sampling: Example



Parallel Sorting Using Regular Sampling: Analysis

- Again sorting is on chunks of (roughly) equal size $s = {}^{n}/_{p}$,
  therefore sorting all chunks has time complexity $O(p \cdot s \log s) = O(n \log s)$.
- Finding the pivots via regular sampling involves:
  - choosing p^2 sample objects: time complexity $O(p^2)$.
  - sorting the p^2 sample objects: time complexity $O(p^2 \log p^2)$.
  - selecting the $p-1$ pivots: time complexity $O(p)$.
  - However, p is fixed, so the time complexity is actually constant $O(1)$.
- Partitioning the chunks (finding split points): time complexity $O(p \log s)$.
  (The split positions can be found with $p-1$ binary searches, one per pivot).
- Exchanging $p(p-1)$ parts between processors: time complexity $O(p^2)$.
  (Only indicators of ranges in memory need to be transferred, not the actual objects.)
- Merging parts of all chunks: time complexity $O(p \cdot s) = O(n)$.
  (Using multiway merge, the time complexity is actually $O(p \cdot s \log p)$, but p is fixed.)

Summary

What we covered in this course (less than anticipated):

Streaming Algorithms (Data does not fit into Main Memory)

- Moments/Characteristic Measures (e.g. mean, variance, ...)
- Number of Unique Elements
- Frequency Estimation and Finding Frequent Items
- Sampling from a Stream
- Histograms and Frequency Distributions
- Sorting (Mergesort)

Parallel Algorithms (Data is processed faster with Parallelization)

- Sorting (Quicksort and Hyper-Quicksort)